The following are informal lecture notes for COMP 250 Winter 2022 by Prof. Michael Langer in the School of Computer Science. See this web page for accompanying slides and exercises.

## Addition

Let's try to remember your first experience with numbers, way back when you were a child in grade school. In grade 1, you learned how to count and you learned how to add single digit numbers. (4 + 7 = 11, 3 + 6 = 9, etc). Soon after, you learned a method for adding *multiple digit* numbers, which was based on the single digit additions that you had memorized. For example, you learned to compute things like:

$$\begin{array}{rr} & 2343 \\ + & \underline{4519} \\ & ? \end{array}$$

The method that you learned was a sequence of computational steps: an *algorithm*. What was the algorithm ? Let's call the two numbers $a$ and $b$ and let's say they have $N$ digits each. Then the two numbers can be represented as an array of single digit numbers $a[\,]$ and $b[\,]$. We can define a variable *carry* and compute the result in an array $r[\,]$.

```
          a[N-1]  .. a[0]
        + b[N-1]  .. b[0]
         ---------------
     r[N] r[N-1] ..  r[0]
```

The algorithm goes column by column, adding the pair of single digit numbers in that column and adding the carry (0 or 1) from the previous column. We can write the algorithm in pseudocode.[1]

**Algorithm 1** Addition (base 10): Add two $N$ digit numbers $a$ and $b$ which are each represented as arrays of digits

```
carry = 0
for $i = 0$ to $N - 1$ do
    $sum \leftarrow a[i] + b[i] + carry$
    $r[i] \leftarrow sum \% 10$
    $carry \leftarrow sum / 10$
end for
$r[N] \leftarrow carry$
```

The operator / is integer division, and ignores the remainder, i.e. it rounds down (often called the "floor"). The operator % denotes the "remainder" operator.

Note that the above algorithm requires that you can compute (or look up in a table or memorized) the sum of two single digit numbers with '+' operator, and also add a carry of 1 to that result. You learned those single digit sums when you were very little, and you did so by counting on your fingers.

[1]By "pseudocode", I mean something like a computer program, but less formal. Pseudocode is not written in a real programming language, but good enough for communicating between humans i.e. me and you.

## Subtraction

Soon after you learned how to perform addition, you learned how to perform subtraction. Subtraction was more difficult to learn than addition since you needed to learn the trick of borrowing, which is the opposite of carrying. In the example below, you needed to write down the result of 2-5, but this is a negative number and so instead you change the 9 to an 8 and you change the 2 to a 12, then compute $12 - 5 = 7$ and write down the 7.

```
     924
  -  352
  -------
     572
```

The borrowing trick is similar to the carry trick in the addition algorithm. The borrowing trick allows you to perform subtraction on $N$ digit numbers, regardless on how big $N$ is.

The next example is more subtle. To perform the subtraction, you need to borrow twice. Effectively, you are writing $900 = 800 + 90 + 12$.

```
     900
  -  352
  -------
     548
```

## Multiplication

Later on in grade school, you learned how to multiply two numbers. For two positive integers $a$ and $b$,

$$a \times b = a + a + a + \cdots + a$$

where there are $b$ copies on the right side. In this case, we say that $a$ is the *multiplicand* and $b$ is the *multiplier*.

ASIDE: The above summation can also be thought of geometrically, namely consider a rectangular grid of tiles with $a$ rows and $b$ columns. You understood that the number of tiles is that same if you were to write it as $b$ rows of $a$ tiles. This gives you the intuition that $a \times b = b \times a$, a fact which is not at all obvious if you take only the summation above.

The summation definition above suggests an algorithm for computing $a \times b$: I claim this algorithm is very slow. To see why, think of how long it would take you $a$ and $b$ each had several digits. e.g. if $a = 1234$ and $b = 6789$, you would have to perform 6789 summations!

**Algorithm 2** Slow multiplication (by repeated addition).

```
$product = 0$
for $i = 1$ to $b$ do
    $product \leftarrow product + a$
end for
```

To perform multiplication more efficiently, one uses a more sophisticated algorithm, which you learned in grade school. An example of this sophisticated algorithm is shown here.

```
     352
  *  964
 -------
    1408
   21120
  316800
 -------
  339328
```

Notice that there are two stages to the algorithm. The first is to compute a 2D array whose rows contain the first number $a$ multiplied by the single digits of the second number $b$ (times the corresponding power of 10). This requires that you can compute single digit multiplication, e.g. 6 $\times$ 7 = 42. As a child, you learned a "lookup table" for this, usually called a "multiplication table". The second stage of the algorithm required adding up the rows of this 2D array.

**Algorithm 3** Grade school multiplication (using powers of 10)

```
for $j = 0$ to $N - 1$ do
   $carry \leftarrow 0$
   for $i = 0$ to $N - 1$ do
      $prod \leftarrow (a[i] * b[j] + carry)$
      $table[j][i + j] \leftarrow prod\%10$          // assume $table[N][2N]$ was array initialized to 0.
      $carry \leftarrow prod/10$
   end for
   $table[j][N + j] \leftarrow carry$
end for
$carry \leftarrow 0$
for $i = 0$ to $2 * N - 1$ do
   $sum \leftarrow carry$
   for $j = 0$ to $N - 1$ do
      $sum \leftarrow sum + table[j][i]$          // could be more efficient since many $table[N][2N]$ are 0.
   end for
   $r[i] \leftarrow sum\%10$
   $carry \leftarrow sum/10$
end for
```

Of course, when you were a child, your teacher did not write out this algorithm for you. Rather, you saw examples, and you learned the pattern of what to do. Your teacher explained why this algorithm did what it was supposed to, by telling you about sums of powers of 10. You didn't need to think about *why* it worked while you mechanically solved problems. You just needed to know the algorithm.

## Division

The fourth basic arithmetic operation you learned in grade school was division. Given two positive integers $a, b$ where $a > b$, the integer division $a/b$ can be thought of as the number of times (call it

$q$) that $b$ can be subtracted from $a$ until the remainder is a positive number less than $b$. This gives

$$a = q\ b + r$$

where $0 \leq r < b$, where $q$ is called the *quotient* and $r$ is called the *remainder*. Note that if $a < b$ then quotient is 0 and the remainder is $a$. Also recall that $b$ is called the divisor and $a$ is called the *dividend* so

$$dividend = quotient * divisor + remainder.$$

The definition of $a/b$ as a repeated subtraction suggests the following algorithm:

**Algorithm 4** Slow division (by repeated subtraction):

```
$q = 0$
$r = a$
while $r \geq b$ do
  $r \leftarrow r - b$
  $q \leftarrow q + 1$
end while
```

This repeated subtraction method is very slow if the quotient is large. There is a faster algorithm which uses powers of 10, similar in flavour to what you learned for multiplication. This faster algorithm is of course called "long division".

## Long Division (fast division): an example

The long division method that you learned in grade school is more challenging to formulate as an algorithm. Let's review an example of how it is performed.

Suppose $a$ = `41672542996` and $b$ = `723`. The goal is now to efficiently compute a quotient $q$ and remainder $r$ such that $a = qb + r$.

The methods starts off like this: (please excuse the dashes used to approximate horizontal lines). You first write down:

```
      --------------
723 |   41672542996
```

You then asks yourself, does 723 divide into 416? The answer is No, since 723 is greater than 416. Then you ask if 723 divides into 4167, and the answer is yes since 723 is less than 4167. Then you need to figure out how many times 723 divides into 4167. You figure out somehow that the answer is 5. So you write down:

```
           5

      --------------
723 |   41672542996
```

You then multiply 5 by 723 which is 3615 and subtract this from 4167 to get 552, which you write as follows:

```
          5
     --------------
723 |  41672542996
       3615
       ----
        552
```

To continue, you "bring down the 2" (that is, the underlined 2 in the dividend 4167$\underline{2}$542996) and then you figure out how many times 723 goes into 5522, where the second 2 in 5522 is the one you brought down.

What's going on here? As a hint, let's write the above example a bit differently.

```
          50000000
    -------------
723 |  41672542996
       36150000000
       -----------
        5522542996
```

What I've done here is add in some zeros to indicate that the lonely 5 at the top of the previous example was in a particular place where it represented the number 50000000. We were then multiplying 723 by this number, which gave 36150000000, i.e.

$$41672542996 = 50000000 * 723 + 5522542996.$$

Note that the remainder is what we get on the bottom row when we copy down, not just the 2, but all the numbers of the original quotient.

Long division would then continue by dividing the remainder (5522542996) again by the divisor (723), and so on. Eventually, we would end up with

$$41672542996 = quotient * 723 + remainder$$

where $0 \leq remainder < 723$.

## Computational Complexity of Grade School Algorithms

For the multiplication and division operations, I presented two versions each and I argued that one was fast and one was slow. We would like to be more specific about this, specifically, how do we quantify how long it takes an algorithm to run. You might think that we want an answer to be in *seconds*. However, for such a real measure we would need to specify details about the programming language and the computer that we are using. We would like to avoid these real details, because languages and computers change over the years, and we would like our theory not to change with it.

The notion of speed of an algorithm in computer science is not measured in real time units such as seconds. Rather, it is measured as a mathematical function that depends on the *size of the problem*. We talk about the *computational complexity* of an algorithm as a function $t(N)$ of the size

$N$ of the problem we are solving. In the case of arithmetic operations on two integers $a, b$ which have $N$ digits each, we say that the size of the problem is $N$.

Let's briefly compare the addition and multiplication algorithms in terms of the number of operations required, in terms of the number of digits $N$. The addition algorithm has some instructions which are only executed once, and it has a `for` loop which is run $N$ times. For each pass through the loop, there is a fixed number of simple operations which include $\%, /, +$ and assignments $\leftarrow$ of values to variables. Let's say that the instructions that are executed just once take some constant time $c_1$ and let's say that the instructions that are executed within each pass of the `for` loop take some other constant $c_2$. We could try to convert these constants $c_1$ and $c_2$ to units of seconds (or nanoseconds!) if had a particular implemenetation of the algorithm in some programming language and we were running it on some particular computer. But we won't do that because this conversion is beside the main point. The main point rather is that these constants have *some* value which is independent of the variable $N$. To summarize, we would say that the addition algorithm requires $c_1 + c_2 N$ operations, i.e. a constant $c_1$ plus a term that is proportional (with factor $c_2$) to the number $N$ of digits. A key concept which we will elaborate in a few weeks is that, for large values of $N$, the dominating term will be the last term. We will say that the algorithm runs in time "order" of $N$, or $O(N)$.

What about the multiplication algorithm? We saw that the multiplication algorithm involves two main steps, each having a pair of `for` loops, one nested inside the other. This "nesting" leads to $N^2$ passes through the inner loop. There are some instructions that executed in the outer `for` loops but not in the inner `for` loops, and these instructions are executed $N$ times. There are also some instructions that are executed outside of all the `for` loops and these are executed a constant number of times. Let $c_3$ be the constant amount of time (say in nanoseconds) that it takes to execute all the instructions that are executed just once, and let $c_4$ be the constant amount of time that it takes to execute all the instructions that are executed $N$ times, and let $c_5$ be the constant amount of time that it takes to execute all the instructions that are executed $N^2$ times, we have that the total amount of time taken by the multiplication algorithm is

$$c_3 + c_4 N + c_5 N^2.$$

Unlike with the addition algorithm, for large values of $N$, now the dominating term will be the $N^2$ term. We will say that the algorithm runs in time $O(N^2)$. We will see a formal definition at the end of the course, once we have seen many more examples of different algorithms and spent more time discussing their complexity.

## Integer division and the modulo operator in mathematics

As discussed last lecture, given two positive integers $a, b$ where $a > b$, we can understand $a/b$ by thinking of the slow division algorithm: repeatedly subtract $b$ from $a$ until the result is greater than or equal to 0 and strictly less than $b$. The number of subtractions is the *quotient* $q$. We can write

$$a = q\ b + r$$

where $0 \le r < b$, where $r$ is the *remainder*.

What if we allow $a$ to be either positive or negative? In the case that $a$ is negative, the slow division algorithm doesn't work. So what do we mean by $a/b$ in that case ? The "quotient remainder theorem" from mathematics says that, given integers $a$ (either positive or negative) and $b > 0$, the $q$ are $r$ in the above equation are uniquely defined.

In the case ($b > 0$), the operator that maps $a$ and $b$ to the remainder $r$ is called *modulus*:

$$a \bmod b \ = \ r.$$

Here are some examples:

$$11 \bmod 3 = 2$$

$$10 \bmod 5 = 0$$

$$-3 \bmod 5 = 2$$

The last one is especially interesting since is a case of $a < 0$. The quotient remainder theorem still holds when $a < 0$, as it only requires that $b > 0$. For this example,

$$-3 = -1 * (5) + 2$$

so $q = -1$ and $r = 2$. See other examples on the slides.

## Integer divison (/) and remainder (%) in Java

In Java, the division operator `a / b` and remainder operator `a % b` are defined for negative and positive values of either `a` or `b`. Here we briefly summarize. For details, see here.

The integer division operator `a/b` yields an integer. The sign of `a/b` is defined as the sign of `a*b`, namely positive if and only if `a` and `b` have the same sign. The magnitude of `a/b` is $rounddown(|\frac{a}{b}|)$ where $\frac{a}{b}$ is a fraction which may be a non-integer.

Java does not have a modulo operator. Instead it has a remainder operator `%`. The integer division and remainder operators in Java are related by:

$$(a/b) * b \ + \ (a \ \% \ b) = a$$

and this definition holds for both positive and negative `a` and `b`. For example, in Java,

$$-\texttt{3} \ \% \ \texttt{5} = -3$$

whereas in mathematics, -3 = 1(-5) + 2, and so

$$-3 \ \texttt{mod} \ 5 = 2.$$

I do not expect you to learn what the remainder operator does when $b < 0$. But the case of `b > 0` does come up, namely when indexing array. If you are thinking that $a \% b$ means "a mod b" and then you will be assuming that it returns a value from 0 to $b - 1$ when $b > 0$. However, this is not so. You will get an out-of-bounds error in your code when you write `a[i % a.length]` in the case that the index `i` is not be in the range 0 to `a.length-1`.

## Base expansion

In our everyday lives, we typically represent numbers using decimal (the ten digits from 0,1, ... 9) or "base 10". The reason we do so is that we have ten fingers. There is nothing special otherwise about the number ten.

Computers typically do not represent numbers using decimal. Instead, internally at least, computers represent numbers using binary or "base 2". What does it mean to represent a number in some "base" ? (ASIDE: in computer science, the term *radix* is sometimes used instead of *base.*)

In decimal, we write numbers using *digits* $\{0, 1, \ldots, 9\}$, in particular, as sums of powers of ten, for example,

$$(238)_{10} \;=\; 2 * 10^2 + 3 * 10^1 + 8 * 10^0$$

In base 2, in binary, we represent numbers using *bits* $\{0, 1\}$, as a sum of powers of 2

$$(11010)_2 \;=\; 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0.$$

In base 3, we represent a number as a sum of powers of 3

$$(22102)_3 \;=\; 2 * 3^4 + 2 * 3^3 + 1 * 3^2 + 0 * 3^1 + 2 * 3^0$$

Note on the left side that I have put brackets and little subscripts 10, 2, 3 to indicate that we are using a particular representation (decimal or binary or base 3). We don't need to always put the brackets and subscripts in, of course, but sometimes it helps to put them in to be clear what the base is. For example $(22102)_3$ represents a different number than $(22102)_{10}$.

In general, let the *base* be an integer greater than 1. Then we can express any positive integer $m$ *uniquely* as a sum of powers of *base*, as follows:

$$m = d_k \; base^k + \cdots + d_2 \; base^2 + d_1 \; base^1 + + d_0 \; base = \sum_{i=0}^{k} d_i \; base^i$$

where $0 \leq d_k < base$ and $d_k \neq 0$. This representation is called the *base expansion of m*, and it can be written

$$(d_k, d_{k-1}, \ldots, d_1)_{(base)} \; .$$

For example, we could talk abou the base 10 expansion, or base 3 or base 2 expansion. Note that, when representing a number in some *base*, we can only use the digits from 0 up to $base - 1$.

## Converting from binary to decimal

It is trivial to convert a number from a binary representation to a decimal representation. You just need to know the decimal representation of the various powers of 2.

$2^0 = 1,\ 2^1 = 2,\ 2^2 = 4,\ 2^3 = 8,\ 2^4 = 16,\ 2^5 = 32,\ 2^6 = 64,\ 2^7 = 128,\ 2^8 = 256,\ 2^9 = 512,\ 2^{10} = 1024,\ \ldots$

Then, for any binary number, you write each of its '1' bits as a power of 2 using the decimal representation you are familiar with. Then you add up these decimal numbers, e.g.

$$11010_2 \;=\; 16 + 8 + 2 = 26.$$

You know how to count in decimal, so let's consider how to count in binary. You should verify below that the binary representation is a sum of powers of 2 that indeed corresponds to the decimal representation in the leftmost column. In the left two columns below, I only used as as many digits or bits as I needed to represent the number. In the right column, I used a fixed number of bits, namely 8. 8 bits is called a *byte*.

| decimal | binary | binary (8 bits) |
|---|---|---|
| 0 | 0 | 00000000 |
| 1 | 1 | 00000001 |
| 2 | 10 | 00000010 |
| 3 | 11 | 00000011 |
| 4 | 100 | 00000100 |
| 5 | 101 | 00000101 |
| 6 | 110 | 00000110 |
| 7 | 111 | 00000111 |
| 8 | 1000 | 00001000 |
| 9 | 1001 | 00001001 |
| 10 | 1010 | 00001010 |
| 11 | 1011 | 00001011 |
| ... | ... | ... |

## Hexadecimal

What if we want to use a base that is greater than 10? To do so, we need to have more symbols than the familiar ten digits $0, 1, \ldots, 9$. One commonly used base is 16, which is known as *hexadecimal*. To represent numbers in base 16, one needs six more symbols, namely "digits" that represent the numbers 10, 11, 12, 13, 14, 15. One typically uses `a,b,c,d,e,f` or `A,B,C,D,E,F`. For example,

$$(\texttt{5AF8})_{16} = 5 * 16^3 + 10 * 16^2 + 15 * 16 + 8 = (23288)_{10}$$

which we usually write as 23,288. In general, we write $m$ in hexadecimal like this:

$$m = \sum_{i=0}^{k} h_i \; 16^i.$$

where $h_i$ are hexadecimal digits. We'll have more to say about this below.

## Converting from base 2 to hexadecimal

The main reason for using hexadecimal is that it provides an easier way for us (as humans) to work with binary numbers. When we write down binary numbers with lots of bits, we can quickly get lost. No one wants to look at a string of say 32 or 64 bits. A common and easy solution is to use *hexadecimal* instead. We group bits of the binary number into 4-tuples ($2^4 = 16$).

To understand why it makes sense to group into 4-tuples, think of

$$m = b_k 2^k + \cdots + b_8 2^8 + b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0$$

We can rewrite the right side (assuming $k > 11$) by forming 4-tuples:

$$b_k 2^k + \cdots + (b_{11} 2^3 + b_{10} 2^2 + b_9 2^1 + b_8) 2^8 + (b_7 2^3 + b_6 2^2 + b_5 2^1 + b_4) 2^4 + (b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0)$$

where we now are expressing the number as sums of powers of $2^4 = 16$, and each of the terms in brackets is a number from 0 to 15, depending on the $b_i$ binary coefficients (bits). We can write this sum again as:

$$b_k 2^k + \cdots + (b_{11} 2^3 + b_{10} 2^2 + b_9 2^1 + b_8)\ 16^2 + (b_7 2^3 + b_6 2^2 + b_5 2^1 + b_4)\ 16^1 + (b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0)\ 16^0$$

which is just

$$\sum_{i=0} h_i 16^i$$

as desired.

Put another way, each 4-bit group can encode 16 combinations of bit values. We encode these 16 digits with symbols `0,1,...,9,a,b,c,d,e,f`. In particular, the symbol `a` represents the binary number 1010 which is 10 in decimal, `b` represents the binary number 1011 which is 11 in decimal, `c` represents 1100 which is 12 in decimal, ..., `f` represents 1111 which is 15 in decimal. Note in writing "binary number" here I have left out the $(\ )_2$ notation.

| Binary | Decimal | Hexadecimal |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |
| 1000 | 8 | 8 |
| 1001 | 9 | 9 |
| 1010 | 10 | a (or A) |
| 1011 | 11 | b (or B) |
| 1100 | 12 | c (or C) |
| 1101 | 13 | d (or D) |
| 1110 | 14 | e (or E) |
| 1111 | 15 | f (or F) |

We commonly (but not always) write hexadecimal numbers as `0x`______ where the underline is filled with characters from `0,...,9,a,b,c,d,e,f`. For example,

$$\texttt{0x2fa3} = 0010\ 1111\ 1010\ 0011.$$

When we write hexadecimal numbers with capital letters, we use a large `X`, for example `0X2FA3`.

What if the number of bits is not a multiple of 4? Should you group the bits starting at the left or right? You *group starting at the rightmost bit* (the least significant bit). For example, if we have six bits string 101011 , then we represent it as `0x2b`. The reason for doing so is that we can pad 0's to the left of a number without changing its value, i.e. $\underline{00}101011$ is the same as 101011. (But if we were to pad with zeros to the right, we would change the value!)

## Converting from decimal to binary (or to any other base)

How do you convert a decimal number to binary or to any other base $b$ ? I will give a simple algorithm for doing so soon which is based on the quotient remainder theorem from last lecture.

I first explain the idea in base 10 where we have a better intuition. Let $m$ be a positive integer which is written in decimal. Then,

$$m = 10 * (m/10) + (m\%10).$$

Note that $m/10$ chops off the rightmost digit and multiplying by 10 appends a 0. So dividing by 10 and then multiplying by 10 might not get us back to the original number. What is missing is the remainder part, which we dropped in the divison.

In binary, the same idea holds. If we represent a number $m$ in binary, then to divide by 2, we chop off the rightmost bit (which becomes the remainder) and we shift right each bit by one position. To multiply by 2, we shift the bits to the left by one position and put a 0 in the rightmost position. So, for example, if $m = 27$,

$$m = (11011)_2 \;=\; 1 * 2^4 + 1 * 2^3 + 1 * 2^1 + 1 * 2^0$$

then dividing by 2 gives 13, or in binary

$$(11011)_2/2 \;=\; (1101)_2$$

Multiplying $(1101)_2$ by 2 gives

$$(11010)_2 \;=\; 1 * 2^4 + 1 * 2^3 + 1 * 2^1$$

that is, 13*2 =26. More generally, for any $m$,

$$m = 2 * (m/2) + (m \;\%\; 2).$$

So, let's assume you are given a number $m$ in base 10. How do we write $m$ in binary. The method below is so simple you could have learned it in grade school. The algorithm repeatedly divides by 2 and the "remainder" bits $b[i]$ are the bits of the binary representation. After I present the algorithm, I will explain *why* it works.
Note this algorithm doesn't say anything about the base in which how $m$ is represented. If *you* are the one executing the algorithm, then you would represent $m$ in base 10. But the algorithm itself does not require this. It only requires that you can carry out division by 2 and mod 2.

Here is an example of how *you* would execute the algorithm (by hand). Note that the bits are computed from the lowest to highest powers of 2. Also note that the value of $m$ shown in the table in row $i$ is the value of $m$ at the *start* of the while loop for that value of $i$, rather than the value of $m$ at the end of the while loop.

**Algorithm 5** Convert decimal to binary

```
INPUT: a number $m$
OUTPUT: $m$ expressed in base 2 using a bit array $b[\,]$
  $i \leftarrow 0$
  while $m > 0$ do
     $b[i] \leftarrow m \% 2$
     $m \leftarrow m / 2$
     $i \leftarrow i + 1$
  end while
  return $b[\,]$
```

**Example: Convert 241 from decimal to binary**

| i | m | $b$[i] |
|---|---|---|
| 0 | $\boxed{241}$ | 1 |
| 1 | 120 | 0 |
| 2 | 60 | 0 |
| 3 | 30 | 0 |
| 4 | 15 | 1 |
| 5 | 7 | 1 |
| 6 | 3 | 1 |
| 7 | 1 | 1 |
| 8 | 0 | |

Thus, $(241)_{10} = (11110001)_2$. Note that I have put the case $i = 8$ in the table, but the algorithm fails the test condition of the while loop when $i = 8$ since $m = 0$ and so it doesn't execute the loop in that case. This is why there is no value $b[8]$.

## Converting from base 10 to any base

Recalling the quotient remainder theorem, we can write any $m$ in terms of any *base* as:

$$m = (m/base) * base + (m \ \% \ base).$$

which is of the form,

$$dividend = quotient * divisor + remainder.$$

or

$$a = q * base + r$$

where $q = m/base$ and $r = m\%base$. If you understand this, and if you understand the algorithm that converts from decimal to binary, then you should be able to understanding that algorithm can be used to convert from base 10 to any *base*. You just need to divide by *base* and use the quotient and remainder, instead of dividing by 2 to using the quotient and remainder.

**Algorithm 6** Convert from base 10 to some other *base*

**INPUT: a number $m$**

**OUTPUT: $m$ expressed in some $base$ using an array $d[\,]$ (digits)**

```
  $i \leftarrow 0$
  while $m > 0$ do
    $d[i] \leftarrow m \ \% \ base$
    $m \leftarrow m \ / \ base$
    $i \leftarrow i + 1$
  end while
  return $d[\,]$
```

## Exponential and logarithms (should be review)

When you signed up for COMP 250, you might have been expecting to learn a lot about *algorithms*. You were probably not expecting to learn a lot about *logarithms*. The words algorithm and logarithm have the same letters, but they mean quite different things!

You learned about logarithms in your Calculus class, in particular, you learned about the number $e \approx 2.718$ and the properties of the function $y = e^x$ and its inverse function $x = \ln_e y$, which is called the "natural logarithm" of $y$ or the logarithm base $e$ of $y$.

In computer science, we usually use logarithms base 2. Logarithms are very important in this course. Since you may be rusty on them and maybe you didn't fully understand in your Calculus class, here we will review some basic and important properties that you will need to be familiar with in COMP 250.

Let's begin with the definition. The logarithm is the inverse function of the exponential, for some given base. If we consider take the exponential function for some base $b > 0$, namely

$$y = b^x, \qquad (*)$$

then the inverse function is called the *logarithm* of $x$ (base $b$). The inverse function as a mapping from $x$ to $y$ is written

$$y = \log_b x.$$

Note that, by saying it is the *inverse* function, we mean that it "undoes" the mapping $y = b^x$ from $x$ to $y$, that is,

$$\log_b y \equiv x. \qquad (**)$$

Note that if we substitute (*) into (**) or vice-versa, we get:

$$\log_b(b^x) = x$$

and

$$b^{\log_b y} = y.$$

This property should be obviously true if you understand the definition.

Note that the above *definition* assumes that $b^x$ is meaningful. I think you all agree that $b^x$ is meaningful when $x$ is an integer, namely you are multiplying $x$ copies of $b$. But it is not so clear what it means when $x$ is not an integer. To define $b^x$ properly when $x$ is not an integer, you would need to learn some more sophisticated math.[2]

[2]See MATH 242 Real Analysis. For those not taking that course, no worries. We won't need it.

Here are a few quick examples to warm up:

$$\log_2 8 = 3, \qquad i.e.\ \ 2^3 = 8$$
$$\log_2 16 = 4 \qquad i.e.\ \ 2^4 = 16$$
$$\log_2 1 = 0 \qquad i.e.\ \ 2^0 = 1$$
$$\log_8 2 = \frac{1}{3} \qquad i.e.\ \ 8^{\frac{1}{3}} = 2$$
$$\log_2 \frac{1}{8} = -3 \qquad i.e.\ \ 2^{-3} = \frac{1}{8}$$
$$\log_8 \frac{1}{2} = -\frac{1}{3} \qquad i.e.\ \ 8^{-\frac{1}{3}} = \frac{1}{8^{\frac{1}{3}}} = \frac{1}{2}$$

We next review a few basic properties of logs which hold for any $b, c > 0$. You should know these properties and be comfortable with them. (The properties don't require that $n, m$ are integers. They can be real. But the properties are easier to visualize if they are integers.)

- $\boxed{b^{n+m} = b^n b^m}$
- $\boxed{(b^n)^m = b^{nm}}$
- $\boxed{\log_b(xy) = \log_b x + \log_b y}$

  To see why, let $u = \log_b x$, so $x = b^u$ and let $v = \log_b y$ and so $y = b^v$. Then,

  $$\log_b(xy) = \log_b(b^u b^v) = \log_b b^{u+v} = u + v = \log_b x + \log_b y$$

- $\boxed{\log_b(x^n) = n \log_b x}$

  To see why, let $u = \log_b x$, so that $x = b^u$. Then,

  $$\log_b(x^n) = \log_b((b^u)^n) = \log_b(b^{un}) = un = n \log_b x.$$

- $\boxed{\log_b x = (\log_b c)(\log_c x)}$

  To see why:

  $$\log_b x = \log_b(c^{\log_c x}) = (\log_c x)(\log_b c)$$

  from the previous property.

There are other properties we could review, but those are the main ones we'll use in the course.

## Grade school arithmetic revisited

Recall from a few lectures ago when we discussed grade school arithmetic. We presented algorithms for addition and multiplication in base 10. But these algorithms in fact work for any base. For example, here is the addition algorithm:

**Algorithm 7** Addition

```
INPUT: Two $N$ digit numbers $a$ and $b$ which are each represented in the same $base$
OUTPUT: their sum $r$ represented in that $base$
  $carry \leftarrow 0$
  for $i = 0$ to $N - 1$ do
    $sum \leftarrow a[i] + b[i] + carry$
    $r[i] \leftarrow sum \% base$
    $carry \leftarrow sum / base$
  end for
  $r[N] \leftarrow carry$
  return $r[\,]$
```

For example, let's add two numbers which are written in binary. I've written the binary representation on the left and the decimal representation on the right.

```
11010    <-carries
11010               26
+ 1011              +11
------             ----
100101               37
```

Make sure you see how this is done, namely how the "carries" work. For example, in column 0, there is no carry and we just have $0 + 1$. The result is 1 and the carry is 0. In column 1, we have $1 + 1$ plus no carry from column 0 (in fact, column 1 represents $1 * 2^1 + 1 * 2^1$). Note that the sum here is $2 * 2^1 = 2^2$ and so we carry a 1 over column 2 which represents the $2^2$ terms. Etc.

Here is another example.

```
 11111110            <-carries

  1111111                  127
+ 0000001                +   1
   ------                 ----
 10000000                  128
```

This example is reminiscent of a car odometer rolling over (see video). It is also a useful example for the following math problem.

## How many bits $N$ do we need to represent $m$ ?

Before we discuss how numbers are represented in Java, it is helpful to know a bit more about binary representations of numbers. We ask a fundamental question: how many bits $N$ do we need to represent a *positive* integer $m$? To answer this question, we can write

$$m = b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \dots b_1 2 + b_0 \tag{1}$$

where $b_{N-1} = 1$, that is, *we only use as many bits as we need.* Enforcing $b_{N-1} = 1$ is similar to the idea that in decimal we don't usually write, for example, 0000364 but rather we write 364. We don't write 0000364 because the 0's on the left don't contribute anything. Similarly we typically wouldn't write 5 in binary as 00000000101 or 000101, etc, but instead we would write it as 101.

Recall the last example on the previous page. From this example and from the arguments of last lecture, the following claims should be clear. The smallest number $m$ with $N$ bits is $(10000...000)_2$ which is $2^{N-1}$. The largest number $m$ with $N$ bits is $(11111...111)_2$ which is $2^N - 1$. Therefore, any number $m$ with $N$ bits must satisfy:

$$2^{N-1} \leq m \leq 2^N - 1$$

We can replace the second inequality as follows:

$$2^{N-1} \leq m < 2^N.$$

Taking the log (base 2) of all terms gives that

$$N - 1 \leq \log_2 m < N.$$

since log is monotonically increasing (so taking log doesn't change the inequalities). Thus, $\log_2 m$ belongs to the interval $[N-1, N)$ which is closed on the left and open on the right. Therefore $floor(\log_2 m) = N - 1$ and so

$$N = floor(\log_2 m) + 1.$$

This formula will come up several times throughout the course.

## Java Primitive Types

All programming languages allow you to represent numbers using a standard number of bits. Typically the number of bits is a multiple of 8: a group of eight bits called a *byte*. In Java, there are several *primitive types* which use the following number of bytes: `boolean` (1), `byte` (1), `char` (2), `short` (2), `int` (4), `long` (8), `float` (4), and `double` (8).

A `boolean` variable uses a byte (8 bits), even though it needs just 1 of these 8 bits. The reason it uses 8 bits is that memory locations in the computer are indexed by bytes, not bits. So there is some waste of bits when representing a `boolean` variable. Oh well...

### integers

Variables of type `byte`, `short`, `int`, `long` can be used to represent integers. They allow both negative and positive integers to be represented. They use $N = 8, 16, 32, 64$ bits, respectively, that is, 1,2,4, 8 bytes. Specifically, these $N$ bits codes represent the integers in the range

$$\{-\,2^{N-1}, \dots,\ 0,\ 1,\ 2, \dots,\ 2^{N-1} - 1\,\}.$$

Think of arithmetic as if we are walking around a circle. When you declare a variable be of type `int` in Java, you are declaring it to be a number in the range $\{-\ 2^{31}, \ldots,\ 0,\ 1,\ 2, \ldots,\ 2^{31} - 1\ \}$, where $2^{31} = 2,147,483,648$.

This leads to some peculiar behaviors known as *overflow* and *underflow*. If you write

```
int x = 2147483647;   // maximum positive int   2^31 - 1
System.out.println(x+1);
```

then it prints out -2147483648, which is the case of overflow. If you write

```
int y = -2147483648;   //  minimum negative int  -2^32
System.out.println(y-1);
```

then it prints out 2147483647, which is the case of underflow.

This 'wraparound' behavior is conceptually similar to what happens with the modulus operator which we discussed last lecture. When we apply *i mod m*, the result is a numbers from 0 to $m - 1$. When we do arithmetic with the integer primitive types, we just use a different range of the $2^N$ numbers which includes both positive and negative numbers. (See slides for figures.)
If you want to learn more about integer representations, in particular how negative numbers are represented in binary, then see my COMP 273 lecture notes.

## non-integer (fractional) numbers

The binary representation of fractional numbers (`float` and `double`) is more complicated and so I'll only sketch out the basics. If you are keen to learn more, see my COMP 273 lecture notes.

[ASIDE: Fractional numbers are represented using scientific notation, but in binary. You are probably familiar with scientific notation in base 10, e.g.

$$6.352122 * 10^{-87}.$$

You can do something similar but in binary. Again, you will learn how this works in COMP 273 or ECSE 222. ]

Let's consider a few interesting examples of what you might experience (to your horror) as a programmer. First, consider these instructions:

```
int x = 3.0;       //   gives a compiler error
int x = 3;
float  y = 3.0;    //   gives a compiler error
float  y = 3.0f;   //   note the 'f'
double z = 3.0;
```

Java does not consider the "literals" 3.0 and 3 to be the same thing. The literal 3.0 is encoded by default as a `double` and the literal 3 is encoded by default as an `int`. If you want to encode 3.0 as a float rather than double, you need to write it as 3.0f as above.
Here are a few more examples:

```
double x = 1;        // legal, but bad style!
int x = 1.0/2;      // compiler error!
double y = 1/4;      // no compiler error, but it stores 0 (not .25)
double y = 1.0/4;    // gives the correct answer
```

We will see why some of these are allowed when we discuss casting a bit later.

Another interesting case to consider is:

```
double y = 1.0/3;
```

You know the answer: 0.33333 and on to infinity. But the computer cannot represent this number exactly and so it has to approximate somehow using the 8 byte code that `double` offers.
What about this one?

```
double y = 0.9;
```

Ha! In fact, the same problem comes up. With the `double` representation, you cannot represent this number exactly, even though in decimal you only need two digits. Is this a problem? Sometimes yes!

```
System.out.println( 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1  );
```

The output is 0.9999999999999999 rather than 1.0. I remember the first time something like this happened to me when I was first learning to program, and I thought my computer was broken. That was until I asked someone who knew more than me for help.

## Java `char` primitive type: ASCII and unicode

Variables of type `char` are used to represent single characters. You assign values to these variables as follows:

```
   char c = 'a' ;
```

where we can put letters and numbers and other keyboard symbols in single quotes. These are called *character literals*. We also have special characters that we can indicate with a combination of two keyboard characters, typically a backslash followed by another character. For example,

```
    c = '\n' ;            \n  represents the {\it new line} character
    c = '\t' ;            \t  represents the {\it tab} character
    c = '\'' ;            \'  represents a single quotation mark character
    c = '\"' ;            \"  represents a double quotation mark character
    c = '\\' ;            \\  represents a backslash character
```

These pairs of characters are called *escape sequences*.

A `char` variable is represented by two bytes or $2^{16}$ values. The first 128 unicode characters are encoded as `000000000*******` where seven `*` places hold either 0 or 1, i.e. 128 combinations. These 128 characters are the ASCII characters. See `http://www.asciitable.com/`. It is more common to write out ASCII codes using hexadecimal which we discussed back in lecture 2. ASCII only has 128 characters, so we usually think of them as a one byte only; the leftmost bit of this byte is 0.

With two bytes, Unicode allows us to represent $2^{16}$ symbols. This covers the characters of many foreign languages, as well as various emoji's that people like to communicate. [ASIDE: The unicode standard (UTF-16, specifically) even allows one to go beyond the 16 bit representation, in an analogous manner to escape sequences that were mentioned above. There are certain unicode symbols that mean the symbol doesn't stand on its own, but rather it is followed by another unicode symbol. ]

We can compare char values using operators ==, <, >, >=, <= which essentially compares their code values. Here are some examples:

```
char letter0 = 'g';
char letter1 = ‘k';
System.out.println( letter0 < letter1 );      //  prints true
System.out.println( ‘g' < ‘G' );              //  prints false
System.out.println( ‘%' >= ‘&' );             //  prints false
```

## Casting primitive types

In the above examples, we saw many cases where we "cast" from one type to another. What are the rules for casting? When converting between primitive types, sometimes we need to cast and sometimes we don't. In Java, one speaks about *wider* and *narrower* types, and you can think of this for now as a rule for defining when one needs to cast or not. Here are the Java primitive types again, but now in order from wider to narrower. The number of bytes is shown in brackets. Note that the ordering does not correspond exactly to the ordering on byte size.

```
double  (8)        widest
float   (4)
long    (8)
int     (4)
short   (2)
char    (2)
byte    (1)       narrowest
```

Widening conversions occur automatically. Narrowing conversions require an explicit *cast* in the code; otherwise you will get a compiler error. Here are some examples.

```
int    i = 3;
double d = 4.2;

d = i;                  // widening (in assignment)
d = 5.3 * i;            // widening in a binary expression (by "promotion")
i = (int) d;            // narrowing (by casting)
float f = (float) d;    //  "

char   c = 'g';
int    index = c;      // widening
c = (char) index;      // narrowing
```

Here is another example. The letter `a` has ASCII code 97, and so to list the letters from `a` to `z` we can just count from 97 to 97+25.

```
  char c = 'a';
  for (int i=0; i < 26; i++)
      System.out.print( (char) (c + i));
```

The expression `c + i` combines a character and an int, and it casts the char `c` to its corresponding int value to perform the sum operation. If we don't cast the result back to char before printing, then we will print out the ASCII codes as numbers 97 to 122 (97 + 25), rather than the characters `a` to `z`.

And here is one last example. Recall overflow and underflow for integers. Let's take the case of the `byte` type ($N = 8$ bits). The largest value that is represented is 127. So if we do the following:

```
byte k = 127;
System.out.println(k+1);    //  prints 128 (promotion to int)
System.out.println( (byte) (k+1));   // prints -128
```

By the way, the reason for writing `(byte)` before printing is that the operation `k+1` "upcasts" the value to an `int` by promotion. To print out the value as a `(byte)`, then we need to "downcast" this `int` back to a `byte`. Similarly if we do the following:

```
  byte j = -128;
  System.out.println( (byte) (k-1));
```

then we don't get an error. Instead, it prints out 127.

Here are some examples with `char` and `short`. This pair of types both have 16 bits, and Java requires an explicit cast in either direction.

```
char c = 'q';
short s = 2;    //    allowed  (special case of a literal)

s =  i;         // compile time error
s = (short) i;

s = c;          // compile time error
s = (short) c;

c =  s;         //   compile time error
c =  (char)     s;
```

And that's all for today, folks!

(JVM, JRE, JDE, IDE,...)
No lecture notes for today – please see slides

## Arrays

Often we have many data items that are all of some given type. These could be numbers, strings, etc. We don't want to define a separate variable for each data item.

```
 int  int1,  int2, int3, ....,  int500;
```

Rather we want a single data structure where we can access these data items using a number index.

An array is a data structure that holds a set of elements that are of the same type. Each element in the array can be accessed or indexed by a unique number which is its position in the array. An array has a capacity or `length`, which is the maximum number of elements can be stored in the array.

In Java, we can have arrays of primitive types or reference type. For today, we'll only discuss primitive types. For example,

```
double[]   arr = new double[15];
```

creates an array of type `double` which has 15 slots, and a variable `arr` that references this array. Note that we use the keyword `new` here.

The variable `arr` is different from the underlying array: the variable arr *references* the array. It says where to find the array in memory. Think of the value of this variable as the address of the array in memory.

For number arrays, each slot is initialized to value 0. We can then assign a value to any slot of an array e.g.

```
arr[12] = -3.2;
```

We can also declare an array variable without initializing it. In this case we do not specify the size of the array that it will reference. There are two ways to do it:

```
double[]  arr1  ;
double    arr2[];
```

We might do this if we know that we will need an array of doubles, but not know in advance how big the array should be.

We can create an array as follows.

```
double[]  arr = new double[9];
```

The `new` keyword is required. The values are initialized to 0.0. Similarly,

```
long[]     longArr = new long[12];
char[]     charArr = new char[15];
boolean[]  boolArr = new boolean[6];
```

These arrays are given initial values in each slot of 0, \u0000, and `false`, respectively. We can then assign values to particular slots as follows:

```
      longArr[7]  = 4000000000000000001;
      charArr[3]  = '%';
```

Note, for the case of `long`, note that we append a an `l` to the end of the literal on the right side. This is similar to how you need to append a `f` to indicate that a number is to be stored as a float. If you don't append an `l` and the number is larger than can fit into an `int` then you get a compiler error.

We can initialize arrays by hardcoding them as the following examples show:

```
int[]   arr = {3, 7, -5, 2, 19};
char[]  ch = { ‘L’, ‘U’, ‘N’, ‘C’, ‘H’};
```

## Arrays and for loops

It is quite common to iterate through the elements of an array. For example,

```
for (int i=0; i < arr.length; i++){
   arr[i] = 2.0*i;
}
```

This dot notation (`arr.length`) will come up often. Another way to iterate through an array is to use a Java *enhanced for loop*.

```
for (double d : arr){
   System.out.println(d);
}
```

Note that this *implicitly* uses an index to step through the array, but we do not have access to the index.

### Shifting elements in an array

It is quite common to shift elements in an array, so its worth looking at several ways in which to do this. For example:

```
for (int j=1; j < arr.length; j++){
    kArr[j-1] = arr[j];
}
```

This shifts the elements backwards. Note that the value in slot 0 is erased. To shift the elements forward, we need to do it slightly differently. If we try:

```
for (int j=1; j < arr.length; j++){
   arr[j] = arr[j-1];
}
```

then we will have a problem. The element in slot 0 will be copied to all the other slots. Not good! Instead, we can do this:

```
for (int j = arr.length-1; j > 0; j--){
    arr[j] = arr[j-1];
}
```

That's better, but it still doesn't quite solve our problem. We still erase an element, namely the last element in the array. If we want to avoid erasing any elements, then we can do a *circular shift*:

```
int tmp = arr[ arr.length-1 ];
for (int j = arr.length-1; j > 0; j--){
   arr[j] = arr[j-1];
}
arr[0] = tmp;
```

This moves the last element to the first slot, and shifts forward all the other elements. Note that we need to use a temporary variable to do so.

### Duplicating an array

What if we want to duplicate an array? One naive way you might try is:

```
int[]  jArray = {3, 5, 2, -7, 6};
int[]  kArray = new int[ jArray.length ];
kArray = jArray;
```

Unfortunately this does not work. Rather, `kArray` will just reference (point to) the same array as `jArray` references. (See the slides for illustration.) Instead, if you want to duplicate an array, then you need to copy the values from `jArray` to `kArray`.

```
int[]  jArray = {3, 5, 2, -7, 6};
int[]  kArray = new int[5];
for  (int m=0;   m < jArray.length;  m++){
    kArray[m] = jArray[m];
}
```

### Java methods: pass-by-value

In Java, when a method has a parameter, it is the *value* of the parameter that is passed to the method. In the case of a variable that is of type array (of int/char/double/etc), the value of that variable is a reference to an array that is stored in memory, namely the value is the location of the starting point of that array in memory.

For example, take this demo method. (I will explain `static` a few lectures from now.)

```
static double  demoPassArray ( double[]  doubleArray   ){
    doubleArray[0] = 23.45;
}
```

We are passing the value of the variable `doubleArray` and that value is a reference to an array i.e. the location of the array. Suppose you call this method in the code below:

```
double[]  arr = {3.0, 5.0, 2.2, -7.1, 6.35};
demoPassArray( arr )
System.out.print( arr[0] );
```

The method will assign the value 23.45 to the slot 0 in the actual array. Notice that even though the method doesn't return anything, the array has been modified. The value 23.45 will be printed out, not the original value 3.0 that was in that slot.

In the case of a primitive type variable, the value of the variable is the number/character/boolean that is stored in memory rather than the location in memory of that number/character/boolean.

This is different behavior from the following, where we pass in a primitive type as an argument to the method:

```
static void  demoPassDouble ( double x  ){
    x  =  175.0;
}
```

When we call this method as follows, now we are passing in the value 2.0.

```
double   y = 2.0;
demoPassDouble( y )
System.out.print( y );
```

The method parameter `x` is initialized to this value, but then gets reassigned the value 175.0. However, `x` behaves as a local variable in the method `demoPassDouble`, with a memory location that is distinct from the variable `y`. So when the method exits, `y` still has its original value and that's what gets printed out.

Finally, we can also return an array from a method. In this example, the method makes a copy of an array and returns the copy. Specifically, it returns a reference to the new array.

```
int[]  copyArray( int[] oldArray ){
   int[]  newArray = new  int[ oldArray.length ];
   for (int i=0; i < oldArray.length; i++){
     newArray[i] =  oldArray[i];
   }
   return newArray;
}
```

## Multidimensional arrays

The arrays we have considered up to now are one dimensional. It is also possible to have multidimensional arrays. You can think of these as arrays of arrays (2D) or arrays of arrays of arrays (3), etc.

For example, here is two dimensional array of `int`'s:

```
int[][]  matrix1 =  new int[4][5];
matrix1[2][4] = 345;
```

and here we have assigned a value to a particular slot in this 2D array. We can think of this array as a one dimensional array of length 4, such that each slot in this array is a one dimensional array of length 5. It is easier to imagine this interpretation with an example:

```
int[][]  matrix2 =  { {5, 7, 23, 3, 65},
                      {23, -45, 56, 0, 16},
                      {234, 3, -564, 3, 345},
                      {6, 30, 46, 23, 23} };
```

One can also define a 2D array such that the second dimension has a different length in each slot. Such arrays are called `ragged` or `jagged` arrays.

```
int[][]  ragged =  { {5, 7, 23},
                     {23, -45, 56, 0, 16},
                     {234},
                     {6, 30} };
```

In this example, the lengths are 3, 5, 1, 2 and `ragged[1][4]` would have the value 16. See the slides for an illustration.

Arrays are used for images and video. For example, a video is a sequence of image frames. The following would be a four dimensional array: two dimensions for the pixel row and column, one dimension of length 3 for color (RGB), and one dimension of length Nframes for time.

```
  int[Nrows][Ncols][3][Nframes]  video;
```

## Arrays have constant time access

A fundamental property of arrays is that the time it takes to access an element does not depend on the number of slots (`length`) in the array. This constant access time property holds, whether we are writing to a slot in an array,

```
a[i] = x;
```

or reading from a slot in an array

```
x  =   a[i];
```

You will understand this property better once you have taken COMP 206 and especially COMP 273, but the basic idea can be explained now. We consider just 1D arrays for simplicity.

The array is located somewhere in the computer memory and the starting location can specified by a number called an *address*. Think of this as like an apartment number in a building, or a number address of a house on a street. Each array slot then sits at some location relative to that starting address and this slot location can be easily computed. To find out where `a[k]` is, you just add the address of `a[0]` to `k` times some constant which is the amount of memory used by each slot. The number of bytes of each memory slot will be different for arrays of `int` versus arrays of `double` versus arrays of `Shape`, but the same principle holds: namely that one can access any slot in constant time since each slot takes the same amount of memory and so we just need to calcululate where to jump to.

Last lecture we discussed arrays. The array type is our first example of a *reference type*. We saw that if a variable is of type array then the value of that variable is a reference to an array. So, think of the variable as storing a number which is an address in memory, namely the address of the array. We next consider two more reference types: `String` and various "wrapper classes" namely `Integer, Double`, etc. We will then discuss how to define our own classes.

## `String` class

Strings are conceptually similar to arrays. A string is a sequence of characters. The sequence has a length, and we can talk about the character in any particular position.

```
  String  s  =  ‘‘Hello’’;
  int  m = s.length();
```

Then `m` would have the value 5.

```
  char  c = s.charAt(1);
```

Then `c` would have the value 'e';

```
  c = s.charAt(8);
```

The following would produce a runtime error (StringIndexOutOfBoundsException)

```
  int  m = s.indexOf(‘o’);
```

The `m` would have the value 4, whereas

```
  m = s.indexOf(‘p’);
```

would return value -1 since that character is not in the string "Hello".

We can use the `+` operator to concatenate strings: Suppose we have

```
String  s0  =  ‘‘Hello’’  ;
String  s1  =  ‘‘ there’’ ;
```

The following expressions (and more) each produce the string `Hello there`.

```
‘‘Hello’’ + ‘‘ there’’
s0 + s1
s0.concat(s1)
‘‘Hello’’.concat(‘‘ there’’)
```

We often wish to ask whether two strings are equal.

```
String  s0 = ‘‘Hello’’ ;
String  s1 = ‘‘Hello’’;
boolean  b  =  s.equals( s1 );  //  true
```

The `String equals()` method goes through each character of the two strings and verifies that they are the same.

It is tempting to use the "==" operator to check if two strings are equal. However, the "==" sometimes produces surprising results.

**ASIDE: why to use `equals()` to compare strings**

The reason for the surprising results are over our heads at this point, but I can still tell you something about it. It has to do with how Java implements strings. At runtime the JVM only maintains one copy of each literal string. So if you write

```
  String s1 = "surprise";
  String s2 = "surprise";
```

then there will be just one string object created there, and `s1` and `s2` will reference that one object. This implies – as we will see a few lectures from now – that `s1 == s2` will have the value `true`.

Next consider what happens when you concatenate two literal strings e.g. `"sur" + "prise"` . In this case, the Java compiler translates this immediately into one string `"surprise"` rather than letting this concatenation happen at runtime. So if you write

```
  String s1 = "surprise";
  String s2 = "sur" + "prise";
```

then you get the same result as above, namely `s1 == s2` will have the value `true`.

Now consider what happens when we have a String variable and we concatenate:

```
System.out.println( "sur" + "prise" == "surprise" );   // true (see above)
String s = "sur";
System.out.println( s1 + "prise"  == "surprise" );     // false
```

The reason that the result is false is that the concatenation in the last line occurs at runtime (not compile time), and the JVM creates a second String object. The == operator checks whether objects are the same, and so returns false in this case.

It is safer to just always use `equals()` for string comparison.

```
  System.out.println( (s1 + "prise").equals("surprise") );    // true
```

**END OF ASIDE: why to use `equals()` to compare strings**

A more details to note about `String`. First,

```
String  name  =  ‘‘Suzanne Fortier’’ ;
name = name.toUpperCase();
```

The second line assigns to `name` the string `‘‘SUZANNE FORTIER’’`. A common mistake is to write just

```
name.toUpperCase()
```

and assume that this changes the string that name references. It doesn't. Rather, a new (upper case) string is created and returned. You need to write that returned string somewhere.

Also, `String` objects cannot be changed. (One says they are "immutable".)

```
String s = "cats";
s.charAt(0) = ’r’;      // compile-time error!
```

There is no String method that allows you to set the value of a character. Rather, one would have to make a completely new string. See the Java API from String for how to do this.

## Wrapper classes

We have seen primitive types such as boolean, char, byte, short, int, long, float, double. There are also classes associated with this type, namely Boolean, Character, Byte, Short, Integer, Long, Float, Double. These classes associate certain constants and methods with the type.

For example `Byte.MAX_VALUE` has value $2^7-1$. `Short.MAX_VALUE` has value $2^{15}-1$. `Integer.MAX_VALUE` has value $2^{31}-1$. `Long.MAX_VALUE` has value $2^{63}-1$. `Float.MAX_VALUE` and `Double.MAX_VALUE` have the largest (finite) values that you can represent with a float or double, respectively. Use `MIN_VALUE` instead of `MAX_VALUE` to get the smallest negative values.

Sometimes we have numerical data represented as strings and we wish to convert them to number types e.g. to convert from a `String` to an `int`, use:

```
int i = Integer.parseInt(‘‘54");
```

wherease to convert from a `String` to an `Integer`, use:

```
Integer j = Integer.valueOf(‘‘54");
```

To convert from a `String` to a `double`, use:

```
double z = Double.parseDouble(‘‘2.7");
```

whereas to convert from a `String` to a `Double`, use:

```
Double y = Double.valueOf(‘‘2.7");
```

The wrapper classes have many other methods. See the Java API for each wrapper class e.g. Integer

Wrapper classes used to have constructors but they they are “deprecated” as of Java 8. Instead, of using the `new` keyword, we use the valueOf method.

```
Boolean  b  =  Boolean.valueOf(false);
Integer  i  =  Integer.valueOf(-45);
Double   x  =  Double.valueOf(3.75);
```

Alternatively, one can just write:

```
Boolean  b  =  false;
Integer  i  =  -45;
Double   x  =  3.75;
```

This casting from a primitive to reference type is called *auto-boxing.*

## `Math` class

Another very useful class is `Math`. For example, `Math.PI` is the value $\pi$. The `Math` class has many useful methods. Suppose we declare a `double` variable `x`.

- `Math.sqrt(x)` returns the value $\sqrt{x}$.
- `Math.random( )` returns a random number in (0,1).

- `Math.log(x)` returns the value $\log_e x = lnx$.
- `Math.log10(x)` returns the value $\log_{10} x$ (There is no method for taking log to a general base b.)
- `Math.sin(x)` returns the value $\sin(x)$

and so forth.

## Defining your own classes

We can also define our own classes. A class definition has the form:

```
class ClassName {

    // field declarations

    //   constructors

    ClassName( ... ){      // no return type
         ...
    }

   // other method declarations
}
```

Note the Java naming convention: class names begin with an upper case letter (String, Integer, Math,...). Constants should be all upper case, e.g. Math.PI. Variables, methods, package names (and some other things) should begin with a lower case character.

Here is an example. We will use this example next lecture too.

```
class Point2D {

    int   x;
    int   y;

    //   constructor

    //   methods for operating on a point
    //   e.g  moving it
}
```

## Default and no-argument constructors

If we don't declare a constructor, then the Java compiler automatically creates a *default constructor* e.g. `Point2D()` which has no arguments and no method body. We would call it with:

```
  Point2D  p = new Point2D();
```

The fields `x` and `y` would be given the value initial 0.

If we explicitly define a constructor with no arguments then this is called a *no argument* constructor. e.g.

```
Point2D(){ }
```

In that last example, the no argument constructor has no method body. But that is not a requirement. A no-argument constructor could also have a method body, for example,

```
Point2D(){
   System.out.println("I am a useless print statement.");
}
```

The method body could also initialize values in the class to be something other than the standard Java default values (`int` are initialized to 0, etc).

It is also common to have constructors with parameters. Typically, the parameters provide initial values for the fields, e.g. :

```
   Point2D(int x0, int y0){
      x = x0;
      y = y0;
   }
```

If we define only this constructor, then the default constructor would not exist. So if we want both this constructor and the no-argument constructor to exist, then we would also need to define the no-argument constructor explicitly.

```
  Point2D(){  }
```

Having more than one version of a constructor is called *overloading* the constructor. We will see lots of examples of overloading as we go along.

Classes can have many methods. For example, for the `Point2D` class, we might want to define two more methods:

```
void  moveTo(int  x0,  int  y0){
   x  =  x0;
   y  =  y0;
}

void  moveBy(int  deltaX,  int deltaY){
   x  +=  deltaX;
   y  +=  deltaY;
}
```

Note that the `x` and `y` variables were defined in the class.

## Java keyword `this`

It can be confusing to keep track of which variables and method belong to which class (or object). The keyword `this` is helpful here. The following example shows a few ways in which '`this`' is used, namely `this.x` and `this.y` refer to the class fields.

```
class Point2D {
   int   x;
   int   y;

   Point2D(int x, int y}{
      this.x = x;
      this.y = y;
   }

   void  moveTo(int  x,  int  y){
      this.x  =  x;
      this.y  =  y;
   }

   void  moveBy(int  deltaX,  int deltaY){
      this.x  +=  deltaX;
      this.y  +=  deltaY;
   }
}
```

Note that we now don't need variable names `x0` and `y0` as in the previous example.

As an example of how this class might be used, consider yet another class:

```
public class  AnotherClass {

   public static void main ( String[] args ) {

      Point2D   p1 = new Point2D(3, 4};
      p1.moveTo( 7, 7 );

      Point2D   p2 = new Point2D(8, 2};
      p2.moveBy( 2, 0 );
   }
}
```

When running this class (main method), two Point2D objects would be created and then would be moved to different positions, namely (7,7) and (10,2) respectively.

Last lecture we introduced how you can define your own classes in Java. A key idea is that a class is a template for objects. Objects of that class are created using the constructor methods of the class. We say that objects of a class are *instances* of that class. We also discussed variables that are called *reference* variables. These variables reference objects of a particular type. The value of a reference variable is a reference, namely it is some coded information that tells you where to find that object. We sometimes call this an address, although in Java this is not quite technically correct so we'll try to be careful and use the Java term 'reference' rather than 'address'.)

## `null` keyword and variable initialization

When a reference variable does not reference any object, we say the value of this variable is `null`. For example, consider this code as part of some method:

```
Point2D   p;           //  value not initialized
p  =  new  Point2D();
p  = null;       //   we can assign a reference to null
```

This example a local variable p of type Point2D is declared in some method.[3] The method then constructs a new object of type Point2D and p references that object. Then, p is assigned null. Now, nothing is referencing that object anymore. In Java, the object is eventually 'garbage collected' which means that the space it was using in memory can be used for something else.

Note that in the above, I wrote in the comment that when p is declared, it is *not* initialized, whereas in the slide in the lecture recording I had written that the value was initialized to null which was a mistake. (The slide has been corrected.) The reason that p is not initialized in the above code is that p is a local variable in a method, and *local variables are not given default values.*

I have added a slide to elaborate that point – see example below. Note that the Test() *constructor does not need to explicitly initialize the fields.* The fields are given default values, depending on their type. For reference variables, the default value is null. Note that this is different from the case above, where local variables in a method are *not* initialized.

```
class  Test {
    int   i1;     //  initialized by default to 0
    String   s1;     //  initialized by default to null
    double[]  dArr1;   //  initialized by default to null

    myMethod(){
        int  i2;             //  not initialized !
        String  s2;          //  not initialized !
        double[]  dArry2;     //  not initialized !
        // bla bla
    };
    Test() {}
}
```

[3]In the lecture recording slide, I had written that this variable is initialized to null. This was a typo, and I have replaced this slide.

## Null Pointer Exception

If we try to use a reference variable that has the value null in a case where the program expects an object, we will get a runtime error called a NullPointerException. Consider the code below which is part of some method. Note we need to initialize the variables to null, since otherwise we would get a compiler error.

```
int[]  intArray = null;
String   s = null;
Double   x = null;

intArray[0] = 3;
char     c = s.charAt(0);
double   y =  x.doubleValue();
```

For all three of the latter instructions, we would get a run-time error, namely NullPointerException.

## Aliasing

In general, the term "aliasing" means that we have different names for the same object.[4]

Recall the example from lecture 5:

```
int[]  arr1 = {3, 5, 2, -7, 6};
int[]  arr2 = new int[ arr1.length ];
arr2 = arr1;
```

In the last line, the variable `arr2` will reference the same array as the variable `arr1` references. This is an example of aliasing. Another example is:

```
Point2D  p1 = new  Point2D( 23, 85 );
Point2D  p2 = new  Point2D( 5,  6 ;
p2 = p1;
```

Similarly, both p1 and p2 would reference the same object, namely the first one (x,y) = (23, 85). The other Point2D object would be garbage collected since no variable would reference it. In particular, the expression `p1 == p2` would be false after the second instruction but it would true after the third instruction. A slightly different example is:

```
p1 = new  Point2D( 5, 6 );
p2 = new  Point2D( 5, 6 );
```

In this case, we would still have two different objects, and so `p1 == p2` would be false, even though the fields in the objects are identical. This is because `p1 == p2` is checking if the variables are pointing the same object, but not actually looking inside the object(s).

What if we write the following?

[4]This term is used not just in programming, but also in daily life. People go under different aliases, for example, The Artist Formerly Known as Prince.

```
p2 = p1;
p2.x  = 400;
```

Since p2 and p1 reference the same object, the value of p1.x will also be changed to 400. This is not evident when you are just looking at those two lines of code. Similarly, if we next have

```
p2 = null;
```

then p1 will still reference that same object, even though p2 will not. For this reason, the object will not be garbage collected.

**There lecture notes are incomplete. Please see slides.**

## Packages

[Note: I discussed packages in lecture 4 when I gave an overview of Java programming. But I did not have lecture notes for that lecture. So I'll include some of that discussion of packages here along with more material that I cover in today's actual lecture.]

A package is a set of classes. Please go to the Java 8 API. [5] On the upper left corner of that web page is a listing of dozens of packages. A few familiar ones are :

- `java.lang - String, Math, ...` which has many useful methods like `sqrt`
- `java.util` - has many of the data structure classes that we will use in this course.

Each class belongs to a package. The full name of any class is the name of the package following by the name of the class, for example `java.lang.Math`. You can have packages within packages e.g. `java.lang` is a package within the `java` package. The package and class structures correspond to how the class files are organized in the file system. The packages are directories (or folders) and subdirectories (or subfolders) and the classes are the .java files in these directories. A package may contain multiple classes. These classes are called *package members*.

When you define a class, you can specify which package it belongs to by starting the class definition with the package name. The first line of your class file will be

```
package packageName;
```

If that package name does not correspond to the directory in which that .java file is found, then your IDE will complain. For example, in the file `Dog.java` we define the `Dog` class and at the top of the file we specify that it is in an `animals` directory, i.e. `animals/Dog.java` .

```
package animals;
class Dog {  }
```

When the `.java` file is compiled by the Java compiler, the compiler produces a `.class` file. The `package` definition specifies the directory where this `.class` file goes. Usually the `.java` file goes in a `src/` directory and the `.class` file goes in a `bin/` directory.

Suppose a class `A` needs to use a class `B` that is a member of some other package. Then class `A` needs to tell the compiler where to find class `B`. There are three ways for class `A` to do this. The first way is to prepend the package name e.g.

```
animals.Dog   myDog = new animals.Dog();
```

This is called *fully specifying* the class name (relative to the class path). The second way would be to import the class:

```
import animals.Dog;
```

The import statement comes after the package statement. It tells the compiler that the class `Dog` is found in the package `animals`. Then it is enough to just refer to Dog.

[5] Java 9 introduced *modules* which group together packages. The API from Java 9 onwards makes it more difficult to see the set of packages.

```
    Dog    myDog = new Dog();
```

The advantage of using an import statement is that it avoids typing the full class name. The disadvantage is that when we (human) read code locally and we see a class name – for example, in a variable type declaration – then we won't necessarily know what package that class belongs to.

The third way would be to import all package members in that package

```
    import animals.*;
```

Earlier I mentioned the Java package `Java.lang` which contains lots of useful classes, e.g. `String`. You do not need to import this package, because the compiler imports it automatically.

Note that if you using imports then you have to be careful not to create class conflicts. You cannot define your own `String` class, for example. Or if you are importing `Dog.java` from your `animals` package, then you cannot have a `Dog.java` file already in the package that you are importing into.

## Visibility (access) modifiers

The above discussion suggests that if you create a class A, you can access any other class B just by specifying its full path or by importing it. But that's not the whole story. Each class (B) also needs to define where each of its fields and methods is visible. This is done using a *visibility modifiers.*

Here are the visibility modifiers in order of decreasing visibility:

- `public` - visible from every package
- default (package) - visible from any class within the same package. This is the default modifier – if you don't write a modifier than the Java compiler assumes you mean this one.
- `private` - can be seen only from within that class

There is also a modifier called `protected` but you need to know about class inheritance to understand what it is. (We will cover inheritance later in the course.)

See the slides for some examples of different visibilities.

### Encapsulation – getters and setters

If you look at various classes the Java API, you hardly ever see class fields listed – even though you know from your experience making your own classes that it is almost impossible to do anything interesting in a class without having fields. The reason that fields are not listed in the Java API for most classes is that the fields are typically private and so they are not part of the interface.

To access the information in class fields, one generally uses getter and setter methods. The reason for this is best seen for the setter methods. Often there are restrictions on values that the fields can take. A field may have type `String` but perhaps we don't want to allow *any* String. For example, someone's `firstName` should not be allowed to be the String "`9*@!+`", and for a McGill Student ID (say it is `String`), a string such as `150912664` should be allowed, but the string "`1234troll`" should not.

Getter methods e.g. `getFirstName()` are examples of "accessors", and setter methods like `setFirstName(String s)` are examples of "mutators". Accessors typically retrieve information and do not change the values of any fields. Mutators change the values of fields, but typically don't return anything. It is possible for a method to be both an accessor and mutator, for example, a method might write a new value into a variable and return the previous value that had been there.

**Why don't method variables have a visibility modifier ?**

Methods often have local variables, called *method variables*. These are not given a visibility modifier. Why not? Methods themselves have a visibility, and so do the classes in which they are defined. However, the variables that are defined within a method are (by definition) not defined outside that method, so other methods (and other classes!) cannot mention them. So it would make no sense to give local variables a visibility modifier.

## UML (Unified Modelling Language) diagrams

When you are working with many classes, you would like to keep track of the names of the fields and methods and also their properties. The usual way to do this is to draw a ("UML") diagram with a bunch of boxes, and within each box you list these various names. (There are relationships between classes as well, which are indicated by different types of arrows between the boxes, but we will not deal these arrows in COMP 250. You will learn about these in depth if you take COMP 303.)

Here is an example of UML diagram for a `Dog` class. My apologies for the two dashed lines lines and four dashed horizontal lines. They should be solid, defining a box partitioned into three parts.

```
 -----------------------------------------------
|            Dog                                |
|-----------------------------------------------|
|       - name : String                         |
|       - owner : Person                        |
|-----------------------------------------------|
|       + Dog(name: String)                     |
|       + Dog(name: String, owner: Person)      |
|       + getName () : String                   |
|       + getOwner () : Person                  |
|       + setName (String)                      |
|       + setOwner (Person)                     |
|       + eat()                                 |
|       + bark(numOfTimes : int)                |
|       + hunt(): Rabbit                        |
 -----------------------------------------------
```

The access modifiers `private` and `public` are indicated on the left by `-` and `+` respectively. Note that the parameters of methods are written *variablename* : *type* whereas in the Java code they are written in the opposite order. Also, when a methods return a value, the type is written after the method. Finally, note how : are used a separators.

There is much more to UML diagrams then this. (For example, you can indicate that a method or field is static by underlining it.) But for now this gives you the basic idea.

## Arrays of reference type variables

When we discussed arrays back in lecture 5, all of our examples were primitive types. We also can have arrays of reference type variables such as strings, for example;

```
String[]  arrString1 = {"several", "different", "strings:}
```

or

```
String[]  arrString2 = new String[8];
arrString2[0] = "this";
arrString2[3] = "little";
arrString2[5] = "doggie";
```

We can also define an array of objects of any class. Suppose we have a class `Shape`. We can define array of `Shape`'s and put objects into this array, for example:

```
Shape[]  shapes = new Shape[428];
shape[239] =  new Shape("triangle", "blue");
```

This array can reference up to 428 `Shape` objects. At the time we construct the array, there will be no `Shape` objects, and each slot of the array will hold a reference value of `null` which simply means there is nothing at that slot.

So if each of these arrays was empty before the above instructions (i.e. after construction of the array), then after the instruction each array would have one element in it.

# Lists

In the next few lectures, we look at data structures for representing and manipulating *lists*. We are all familiar with the concept of a list. You have a TODO list, a grocery list, etc. A list is different from a "set". The term "list" implies that there is a positional ordering. It is meaningful to talk about the first element, the second element, the i-th element of the list, for example. For a set, we don't necessarily have an order.

What operations are performed on lists? Examples are getting or setting the $i$-th element of the list, or adding or removing elements from the list.

```
get(i)          //  Returns the i-th element (but doesn't remove it)
set(i,e)        //  Replaces the i-th element with e
add(i,e)        //  Inserts element e into the i-th position
add(e)          //  Inserts element e (e.g. at the end of the list)
remove(i)       //  Removes the i-th element from list
remove(e)       //  Removes element e from the list (if it is there)
clear()         //  Empties the list.
isEmpty()       //  Returns true if empty, false if not empty.
size()          //  Returns number of elements in the list
:
```

There are many ways to implement lists. Today we'll look at a way that uses an array. The data structure we will describe is called an *array list.*

## Using an array to represent a list

Suppose we have an array `a[ ]`. For now I won't specify the type because it is not the main point here. We want to use this array to represent a list. Suppose the list has `size` elements. We will keep the elements at positions 0 to `size-1` in the array, so element $i$ in the list is at position $i$ in the array. This is hugely important so I'll say it again. With an array list, the elements are squeezed into the lowest indices possible so that there are no holes. This property requires extra work when we add an remove elements, but the benefit is that we always know where the $i$th element of the list is, namely it is at the $i$th slot in the array.

Let's sketch out algorithms for the operations above. Here we will not worry about syntax so much, and instead just write it the algorithm as pseudocode.

We first look at how to access an element in an array list by a read (get) or write (set).

### `get(i)`

To get the i-th element in a list, we can do the following:

```
if (i >= 0) &  (i < size)
   return a[i]
```

Note that we need to test that the index `i` makes sense in terms of the list definition. If the condition fails and we didn't do the test, we would get in index out of bounds exception.

We set the value at position i in the list to a new value as follows:

### `set(i,e)`

```
if (i >= 0) &  (i < size)
   a[i] = e
```

This code replaces the existing value at that position in the list. If there were a previous value in that slot, it would be lost. An alternative is to return this previous element, for example,

```
if (i >= 0) & (i < size){
    tmp = a[i]
    a[i] = e
    return tmp
}
```

Indeed, the Java ArrayList method `set` does return the element that was previously at that position in the list. See the set method in the Java API.

Next we "add" an element `e` to $i$-th position in the list. But rather than replacing the element at that position which we did with a `set` operation, the `add` method *inserts* the element. To make room for the element, it displaces (shifts) the elements that are currently at index $i, i+1, ..., size-1$. Here we can assume that $i \leq size$. If we want to add to the end of the list then we would add at position $size$. Moreover, we also assume for the moment that the size of the list is strictly less than the number of slots of the underlying array, which is typically called the length (`a.length`) of the array. The length of the array is also called the *capacity* of the array. The capacity or length of the array is always greater or equal to the size of the array list that uses this array.

## add(i,e)

```
   if (i > 0) & (i <= size){
      for (k = size; k > i; k--)      //  (do nothing, if i == size)
          a[k] = a[k-1]               //  shift to bigger index
      a[i] = e                        //  insert into now empty slot
      size = size + 1
   }
```

The above algorithm doesn't deal with the case that the the array is full i.e. `size == length`. We need to augment the above code to handle that case, namely we need to make a new and bigger array. The following code would go at the beginning and would ensure that the condition `size < length` is met when the above code runs.

```
//   insert this pseudocode at start of above add(i,e) pseudocode
if (size == length){
   b = new array with 2 * length slots
   for (int k=0; k < length; k++)
       b[k] = a[k]                    //  copy elements to bigger array
   a = b
}
```

Note that the above pseudocode allows us to add at the end of the list. That is, calling `add(size, e)` does *not* generate an index out of bounds exception. This is equivalent to calling a method `add(e)` which by default adds at the end of the list, without specifying the size of the list as a parameter. Also, note that `add(e)` would be different from the `set` method, which does not allow us to call `set(size,e)`.

Also note that in the Java `ArrayList` class, the `add` is overloaded: there is an `add(int i, E e)` and an `add(E e)` method where `e` is of type `E`. Lists in Java also have overloaded `remove` methods, where `remove(int i)` removes the element at index `i` and an `remove(E e)` removes the first occurance of an element `e`. Let's turn to `remove` next.

## remove(i)

Removing an element from an arraylist is very similar to adding an element, but the steps go backwards. We again shift all elements by one position, but now we shift back by 1 rather than forward by 1. The `for` loops goes forward from slot $i$ to $size - 2$, rather than backward from $size$ to $i + 1$

```
if ((i >= 0) & (i < size)){
   tmp = a[i]                       // save it for later
   for (k = i; k < size-1; k++){
      a[k] = a[k+1]                 //  copy back by one position
   }
   size = size - 1
   a[size] = null      //  optional,  but perhaps cleaner
   return tmp
}
```

One final, general, and important point which concerns array lists of $N$ elements where $N$ is large: Adding to or removing from near the back end of the list is fast (ignoring the case where you want to add to a full array), since few shift operations are necessary. However, adding or removing from the front of the list will be slow, since most of the $N$ elements need to be shifted. We will return to this point in the next few lectures when we compare array lists to linked lists.

## Java's `ArrayList<E>` class and generic types

Java has an `ArrayList` class that implements the various methods such as we discussed and uses an array as its underlying data structure. You should check out what these methods are for the `ArrayList` class in the Java API

Whenever you construct an `ArrayList` object, you need to specify the type of the elements that will be stored in it. You can think of this as a parameter that you pass to the constructor. In Java, the syntax for specifying the type uses <> brackets. For example, to declare an ArrayList of objects that are of type `Shape`, use:

```
ArrayList<Shape>  shapes = new ArrayList<Shape>( );
```

If you look at the Java API, you'll see that the class is defined `ArrayList<E>` where `E` is called a *generic type*. We will see many examples of generic types later.

Note that the generic type needs to be a reference type; it cannot be a primitive type. So if you want an arraylist of integers, then you need to use the wrapper class:

```
ArrayList<Integer>  shapes = new ArrayList<Integer>( );
```

This is another reason why wrapper classes are needed in Java.

Just a few points to mention before we move on... First, although the `ArrayList` class implements a list using an underlying array, the user of this class does not index the elements of the array using the familiar array syntax `a[ ]`. Why not? For one thing, the user (the client) doesn't know what is the name of the underlying array, since it is `private`. Instead the user accesses an array list element using a `get` or `set` or other methods.

Second, because the `ArrayList` class uses an array as its underlying data structure, if one uses `add` or `remove` for an element at the *front* of your list, this operation will take time proportional to `size` (the number of elements in the list) since all the other elements needs to be shifted by one position. This shifting can be slow. Thus, although arrays allow you to get and set values in very little (and constant) time, they can be slow for adding and removing from near the front of the list.

## Singly Linked lists

**[See the slides for the figures to accompany these lecture notes. ]**

We next look at another list data structure - called a linked list - that partly avoids the problem we discussed that array lists have when adding or removing from the front of the list. (Linked lists are not a panacea. They have their own problems, as we'll see).

With array lists, each element was referenced by a slot in an array. With linked lists, each element in the list is referenced by a list *node*. A linked list node is an object that contains:

- a reference to an element of a list
- a reference to the `next` node in the linked list.

In Java, we can define a linked list node class as follows:

```
class  SNode<T>{
   T            element;
   SNode<T>     next;
}
```

where `T` is the generic type of the object in the list, e.g. `Shape` or `Student` or some predefined Java class like `Integer` or `String`. We use the `SNode` class to define a `SLinkedList` class.

Any non-empty list has a first element and a last element. If the list has just one element, then the first and last elements are the same. A linked list thus has a first node and a last node. A linked list has variables `head` and `tail` which reference the first and last node, respectively. If there is only one node in the list, then `head` and `tail` point to the same node.

Here is a basic skeleton of an `SLinkedList` class.

```
class  SLinkedList<T>{
   SNode<T>       head;
   SNode<T>       tail;
   int            size;

   private class  SNode<T>{
      T            element;
      SNode<T>     next;
   }
}
```

We make the `SNode` class a private inner class[6] since the client of the linked list class will not ever directly manipulate the nodes. Rather the client only accesses the elements that are referenced by the nodes.

[6]For info on inner classes (and nested classes in general), see
`https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html`

As I discuss in the slides, suppose we have a linked list with `size` = 4. How many objects do we have in total? We have the four `SNode` objects. We have the four elements (objects of type `Shape`, say) that are referenced by these nodes. We also have the `SLinkedList` object which has the head and tail references. So this is 9 objects in total. ( For an array list with four elements, we would have 6 objects: the four elements in the list, the arraylist object, and the underlying array object.)

One important difference between linked lists and array lists is that linked list nodes can be anywhere in memory, whereas the underlying array in array list is a block of consecutive locations in memory. Having linked list nodes anywhere is memory makes them flexible - there is no notion of running out of room, which is what happens when the underlying array of an arraylist becomes full. The disadvantage of linked lists, as we'll see, is that we don't have constant time access to elements.

Let's focus for the moment on one advantages of a linked list. It allows you to add an element or remove an element at the front of the list in a constant amount of time.

```
addFirst( e ){                    //  add element e to front of list
   construct newNode
   newNode.element = e
   newNode.next = head      (*)
   head = newNode           (**)
   size++
   if (size == 1)
      tail = head
}
```

The order of the instructions (*) and (**) matters. If we had used the opposite order, then the `head = newNode` instruction would indeed point to the new first node. However, we would not remember where the old first node was. The `newNode.next = head` instruction would cause `newNode.next` to reference itself.

Also notice that we have considered the case that initial the list was empty. This special case ("edge case") will arise sometimes. Whenever you write methods, ask yourself what are the edge cases and make sure you test for them. I may omit the edges cases, sometimes intentionally (to keep it simple), sometimes unintentionally. Don't hesitate to ask if you notice one is missing.

Let's now look at an algorithm for removing the element at the front of the list. The idea is to advance the `head` variable. But there are a few other things to do too:

```
removeFirst(){
//  test for empty list omitted  (throw an exception)
   tmp = head             // remember first element, so we can return it
   head = head.next
   tmp.next = null        // not necessary but conceptually cleaner
   size = size - 1
   if (size == 0)
       tail = null        // edge case:  one element in list
   return tmp.element
}
```

Notice how we have used `tmp` here. If we had just started with (`head = head.next`), then the old first node in the list would still be pointing to the new first node in the list, even though the old first node isn't part of the list. (This is not really a problem since nothing is referencing that node, so there's no way to reach it, and hence no way it can cause a problem. ) Also, in the code here, the method returns the element. Note how this is achieved by the tmp variable.

What about adding or removing an element at the back of a linked list? Adding at the back just requires manipulating the `tail` reference.

```
addLast( e ){
   construct newNode
   newNode.element = e
   tail.next = newNode
   tail = tail.next
   size = size + 1
}
```

Removing an element from the back of a liked list is more complicated, however. The reason is that you need to modify the `next` reference of the node that comes *before* the tail node which you want to remove. But you have no way to directly access the node that comes before `tail`, and so you have to find this node by searching from the front of the list. Yikes!

The algorithm begins by checking if the list has just one element. If it does, then the last node is the first node and this element is removed. Otherwise, it scans the list for the element that comes before the last element.

```
removeLast(){
   e = tail.element
   if (head == tail){
      head = null
      tail = null
   }
   else{
      tmp = head
      while (tmp.next != tail){
         tmp = tmp.next
      }
      tmp.next = null
      tail = tmp
   }
   size = size-1
   return e
}
```

This method requires about `size` steps. This is much more expensive than what we had with an array implementation, where we had a constant cost in removing the last element from a list.

I finished the lecture by discussing computational complexity. I'll put that discussion in the next lecture, when we'll look at doubly linked lists.

## Doubly linked lists

The "S" in the `SLinkedList` class from last lecture stood for "singly", namely there was only one link from a node to another node. Today we look at "doubly linked" lists. Each node of a doubly linked list has two links rather than one, namely references to the previous node in the list and to the next node in the list. These reference variables are typically called `prev` and `next`. As with the singly linked list class, the node class is usually declared to be a private inner class. Here we define it within a `DLinkedList` class.

```
   class  DLinkedList<E>{
        DNode<E>        head;
        DNode<E>        tail;
        int             size;
          :

        private class  DNode<E>{
           E            element;
           DNode<E>        next;
           DNode<E>        prev;
             :
        }
   }
```

The key advantage of doubly linked lists over a singly linked list is that the doubly linked lists allows us to quickly access elements near the back of the list. For example, to remove the last element of a doubly linked list, one simply does the following:

```
removeLast(){
   e  =  tail.element
   tail      =  tail.prev
   tail.next = null
   size      = size-1
   return e
}
```

## Dummy nodes

When writing methods, one often has to consider *edge cases*. For doubly linked lists, the edge cases are the first and last elements. These cases require special attention since `head.prev` and `tail.next` will be `null` which can cause errors in your methods if you are not careful.

[**ADDED: Feb. 10**] For example, consider the `removeLast()` method above. What if the list only had one element? We would be removing that one element and the list would then be empty. This itself is not a problem. But consider what the above code does. The `tail = tail.prev` instruction would assign `tail` to null, since the list has just one node. But if `tail` is null then `tail.next` would not makes sense, so you would get a null pointer exception. Even if you corrected the code to avoid this, there would still be another problem: the code does not modify the `head` reference. But if we remove the one node in a list, then the `head` reference should become null. Such edge cases arise for several methods, and can lead to errors if one is not careful.

To avoid such errors, it is common to define doubly linked lists by using a "dummy" head node and a "dummy" tail node, instead of head and tail reference variables.[7] The dummy nodes are objects of type `DNode` just like the other nodes in the list. However, these nodes have a `null` element. Dummy nodes do not contribute to the `size` count, since the purpose of `size` is to indicate the number of elements in the list. See figures in slides.

```
class  DLinkedList<E>{
   DNode<E>        dummyHead;
   DNode<E>        dummyTail;
   int             size;
    :

   // constructor

   DLinkedList<E>(){
      dummyHead = new DNode<E>();
      dummyTail = new DNode<E>();
      dummyHead.next   =   dummyTail;
      dummyTail.prev   =   dummyHead;
      size      = 0;
   }

   //  ... List methods and more
}
```

Let's now look at some `DLinkedList` methods. We'll start with a basic getter which gets the i-th element in the list:

```
   get( i ){
     node = getNode(i)
     return  node.element
   }
```

This method uses a helper method `getNode(i)` that I'll discuss below. It is worth having a helper method because we can re-use it for several other methods. For example, in the Exercises PDF for this lecture, I ask you for code to remove the i-th node. Here is the pseudocode solution:

```
  remove( i ){
     node = getNode(i)
     node.prev.next = node.next
     node.next.prev = node.prev
     size--
  }
```

[7] Dummy nodes can be defined for singly linked lists too.

This code modifies the `next` reference of the node that comes before the i-th node, that is `node.prev`, and it modifies the `prev` reference of the node that comes after the i-th node, that is `node.next`. Because we are using dummy nodes, this mechanism works even if i = 0 or i = size-1. Without dummy nodes, `node.prev` is `null` when i = 0, and `node.next` is null when `i = size - 1`, so the above code would have an error if we didn't use dummy nodes. (Note that I am not bothering to set the `next` and `prev` references in the removed node to `null`. But you could do that if you want.)

Here is an implementation of the `getNode(i)` method. This method would be *private* to the `DLinkedList` class.

```
getNode(i){
   node = dummyHead.next
   for (k = 0;  k < i; k++)
      node = node.next
   return node
 }
```

One can be more efficient than that, however. When index i is greater than size/2, then it would be faster to start at the tail and work backwards to the front, so one would need to traverse size/2 nodes in the worst case, rather than size nodes as above.

```
getNode(i){
   if (i < size/2){
       node = dummyHead.next
       for (k = 0;  k < i; k++)
           node = node.next
   }
   else {
       node = dummyTail.prev
       for (k = size-1;  k > i; k--)
          node = node.prev
       }
   return node
}
```

The `remove(i)` method still takes `size/2` operations in the worst case. Although this worst case is a factor of 2 smaller for doubly linked list than for singly linked lists, it still grows linearly with the size of the list. Thus we say that the `remove(i)` method for doubly linked lists still is $O(N)$ when $N$ is the size of the size. This is the same time complexity for this method as we saw for array lists and for singly linked lists. Saving a factor of 2 by using the trick of starting from the tail of the list half the time is useful and does indeed speed things up, but only by a proportionality factor. It doesn't change the fact that in the worst case the time is takes grows linearly with the size of the list.

## Java `LinkedList`

Java has a `LinkedList` class which is implemented as a doubly linked list. Like the `ArrayList` class, it uses a generic type which is write as `T` below.

```
LinkedList<T>  list  =  new  LinkedList<T>();
```

The `LinkedList` class has more methods than the `ArrayList` class. In particular, the `LinkedList` class has `addFirst()` and `removeFirst()` methods. Recall removing elements from near the front was expensive for an array list. So if you are doing this a lot in your algorithm, then you probably don't want to be using an `ArrayList` for your list. So it makes sense that that the `ArrayList` class wouldn't have such methods. But adding and removing the first elements from a list is cheap for linked lists, so that's why it makes sense to have these methods in the Java `LinkedList` class. Of course, you could just use `remove(0)` or `add(0,e)`, but a specialized implementation `addFirst()` and `removeFirst()` might be a bit faster and the code would be easier to read – both of which are worth it if the commands are used often. In addition, the `LinkedList` class has an `addLast()` and `removeLast()` method, whereas the `ArrayList` class does not have these methods.

## Time Complexity

The table below compares the time complexity for adding/removing an element from the head/tail of an array or singly or doubly linked list that has size $N$. The problem cases are the ones that are O(N).

```
                   array list    singly linked list      doubly linked list
                   ----------    ------------------      ----------
addFirst( e )         O(N)          O(1)                    O(1)
removeFirst()         O(N)          O(1)                    O(1)

addLast( e )          O(1)          O(1)                    O(1)
removeLast()          O(1)          O(N)                    O(1)

get( i )              O(1)          O(N) <- worst case -> O(N)
                                    O(1) <- best case  -> O(1)
```

A few notes

- The Java `ArrayList` class doesn't actually have an `addFirst` or `removeFirst` method, so just consider the equivalent `add(0,e)` and `remove(0)` as mentioned above.
- `addLast` for an array list is $O(N)$ in the special case that the array is full. But this case doesn't happen much if our policy is to double the size of the array. (See Exercises for what the complexity is to add $N$ elements consecutively.)
- `get(i)` can be very fast for a singly or doubly linked list e.g. if `i=0` in which case it is $O(1)$ i.e. it doesn't depend on the size of the list. But in the worst case it takes time proportional to the length of the list, so it would be $O(N)$.

## How not to iterate through a linked list

Suppose we add $N$ students to the front (or back) of a linked list.

```
for (k = 0; k < list.size();   k ++)            //  size == N
   System.out.println( list.addFirst(  new E()  ));
```

Adding $N$ students to an empty linked list takes time proportional to $N$ since adding *each* element to the front (or back) of a linked list takes a constant amount of time, i.e. independent of the size of the list

What if we wanted to print out the elements in a list? Without thinking about it much, the following would seem to work fine.

```
for (k = 0; k < list.size();   k ++)            //  size == N
     System.out.println( list.get(  k  ));
```

However, this turns out to be very inefficient, and it matters for large lists. (See Exercises.) For simplicity, suppose that `get` were implemented by starting at the head and then stepping through the list, following the next reference until we get to the i-the element. Then, with a linked list as the underlying implementation, the above for loop would require

$$1 + 2 + 3 + ... + N = \frac{N(N+1)}{2}$$

steps. This is $O(N^2)$ which gets large when $N$ is large. It is obviously inefficient since we really only need to walk through the list and visit each element once. The problem here is that each time through the loop the `get(k)` call starts again at the beginning of the linked list and walks to the k-th element.

## Java enhanced for loop

What alternatives to we have to using repeated `get`'s ? In Java, we can use something called an *enhanced for loop*. The syntax is:

```
  for  (E    e :   list){
         //  do something with element e
  }
```

where `list` is our variable of type `LinkedList<E>` and `e` is a variable that will reference the elements of type `E` that are in the list, from position, 0, 1, 2, ..., size-1.

## Space Complexity ?

We've considered how long it takes for certain list operations, using different data structures (array list, singly linked and doubly linked lists). What about the space required? Array lists use the least amount of total space since they require just one reference for each element of the list. (These 'slots' must be adjacent in memory, so that the address of the k-th slot can be computed quickly.)

Singly linked lists take twice as much space as array lists, since for each element in a singly linked lists, we require a reference to the element and a reference to the next node in the list. Doubly linked lists require three times as much space as array lists, because they require a reference to each element and also references to the previous and next node.

All three data structures require space proportional to the number of elements $N$ in the list, however. So we would say that all require $O(N)$ space.

## Cloning a linked list: shallow versus deep copy

Finally, we discussed what it means to make a copy of a linked list of some type, say `Shape`. Suppose we have a linked list that is referenced by a variable *list1*. What we *don't* mean is that we do:

```
LinkedList<Shape> list2 = list1;
```

That would just be aliasing. Rather, I am asking about making a copy of the list.

There are two ways to make a copy. One is that we copy the nodes of the original list and link them together, but we don't copy the `Shape` objects of the list. This is called a *shallow copy* of the list. The nodes of the new list point to the same sequence of Shape objects, but there is just one version of each of the Shape objects.

The second way is to copy the nodes of the original list and link them together, and also to make copies of the objects themselves, and have the new list nodes reference the copiess. This is called a *deep copy* of the list. Here we would have two versions of each of the Shape objects.

The Java `LinkedList` class has a `clone` method (see API), and this method makes a shallow copy rather than a deep copy.

We will discuss shallow versus deep copy a few lectures from now, when we revisit what we mean by the `equals()` method. The reason this method is relevant is that we will want to decide what it means to say that two `LinkedList` objects are equal!

Why would one would want to use a shallow versus deep copy. Here are some intuitive examples. (Note, there is nothing specific about linked lists in these examples. The same concepts of shallow versus deep copy applies for array lists.)

For shallow copy, suppose you have a list of student exams and you have several different graders who are each marking a different question on the exam. Then each grader could have a shallow copy and would iterate through the list of exams, and each grader would assign a grade to a different question. A shallow copy would fine here, since in the end we want each question of each exam to be graded, and we want to end up with one graded copy of each exam.

For the above example, you might ask why we even need to make a copy. Why not just have two variables that reference the same list (aliasing)? Great question! To fully justify the shallow copy, I need to add a bit more to the example. Suppose that one of the graders wants to sort the exams in the list, e.g. according to the student last name. In this case, this grader would need to rearrange the nodes of their list. This grader should have their own copy of the nodes (the shallow copy); otherwise, rearranging the links in the original list would mess up the other graders that are iterating through the list. Note that a shallow copy (not a deep copy) is really what we want here. There should be just one copy of each exam.

For deep copy, suppose you have a list of applications for jobs, and several different potential employers would get access to that list. Each potential employer would go through the list, and perhaps mark up the application. In this case, a shallow copy is not good enough because each employer is marking up the application, and so each employer should have their own copy of each application.

## Quadratic[8] sorting

*[See the slides for figures to complement the discussion in these notes.]*

One common problem that comes up in computing that you want to sorting a list $N$ elements. Let's say you are given a list and you want the modify the order of elements so that they are *increasing*. Here we are assuming that elements are comparable, in that for any two elements A and B that we are considering, either $A < B$ or $A > B$. (It could also be that $A$ equals $B$, but we won't concern ourselves with this case because either order of two such elements would be fine.) For example, the elements in the list might be numbers, or they might be strings which can be ordered alphabetically. Today we will look a few simple algorithms. Later in the course, we'll look at more complicated algorithms which are much faster when the list is large.

We will discuss three algorithms today. These will be presented with pseudocode only and without committing to a particular data structure e.g. array list or doubly linked list. The algorithms could be implemented in principle with either of these data structures, but the going into the data structure details would be distracting.

## Bubblesort

The first algorithm is called *Bubblesort*, and it is perhaps the simplest to describe. You traverse through the list repeatedly, and whenever you find two neighboring elements that are out of order, you swap them. Elements gradually make their way to their correct position in the list. The algorithm is called bubblesort because it invokes the concept of bubbles rising in a fluid, which some people apparently find helpful to think about. Here it is:

```
for i  = 0 to list.size - 2 {   //  pass i through list
  for k = 0 to list.size - 2 {
    if ( list[k] > list[k+1] ) {
       list.swap(k,  k+1)
    }
  }
}
```

What can we say after one pass through the list? We can say that the largest element in the list will be at the end of the list. The reason is that the inner `for` loop will eventually hit this element and will then drag it to the end of the list via successive swaps.

What can we say about the position of the smallest element in the list after one pass? Not much, except that it won't be at the end of the list (unless all elements are equal). For example, if the smallest element of the list starts out at the end of the list, i.e. in position $N-1$, then in the first pass through the inner loop, the element will be moved only to position $N-2$. More generally, the smallest element will be moved one position earlier in the list, unless it was already at the beginning of the list. The reason it will move one earlier is that first time we reach it, it will be some position `k+1` (assuming it is not in the front of the list) and then it will be moved to position `k`, but then the inner loop will continue on and it will stay where it is – at least `i` is incremented.

[8]Quadratic means that the time complexity grows like $N^2$, i.e. $O(N^2)$.

How many passes through the list will we need to put all elements in order? We will need at most $N-1$ passes. Take the case that the smallest element starts off at the end of the list at index position $N-1$. In the first pass it moves to position $N-2 = N-1-1$. In the second pass, it moves to position $N-3 = N-1-2$. etc. After the $k'th$ pass, it has moved to position $N-1-k$. Thus, after $k = N-1$ passes, the smallest element will have moved from position $N-1$ to position 0.

Do you always need $N-1$ passes to put all the elements in order? No. For example, if elements are already sorted, then the outer loop only needs to run once. Let's modify the pseudocode so that we can detect when the elements are in order, and stop.

```
for i  = 0 to list.size - 2 {
   swapped = false              //  will set to true if we swap
   for k = 0 to list.size - 2 - i  {
      if ( list[k] > list[k+1] ) {
         list.swap(  k, k+1 )      // flag that assumption was wrong
         swapped = true            //  i.e. at least one swap
      }
    }
    if !(swapped)                  // check if we did a swap
       break
}
```

## Selection sort

The second algorithm is *Selection Sort.* The idea here is to partition the list into two parts. The first part contains the smallest elements, in order. The second part contains the remaining elements.

The algorithm uses an index counter $i$ and starts with $i = 0$. For each $i$, the sorted part is the list up to and *not* including position $i$, and the "rest" part is the list from positions $i$ to $list.size-1$. For each $i$, the algorithm finds the minimum element in the rest part. If this minimum element is at a position (called `index` in the pseudocode below) that is different than $i$, then it swaps this minimum element at `index` with the element at position $i$. The element at position $i$ then becomes the last element in the sorted part.

```
for  i = 0  to list.size - 2   {
   index = i
   minValue = list[ i ]
   for k = i+1  to  list.size - 1  {
      if ( list[k] < minValue ){
         index = k
         minValue = list[k]
      }
   }
   if ( index != i )
      list.swap(  i,  index )
}
```

How many times does the inner loop get executed? Let `N = list.size`. When `i` is 0, the inner loop is executed `N-1` times. When `i` is 1, it is executed `N-2` times. When `i = N-2`, the inner loop is executed once. Therefore the total is

$$N - 1 + N - 2 + N - 3 + ... + 3 + 2 + 1.$$

which is $\frac{N(N-1)}{2}$ which is roughly $N^2/2$.

Note a few differences with bubblesort. One difference is that in the best case bubble sort only takes one pass through the outer loop, whereas with selection sort the outer loop always runs $N-1$ times. Thus bubble sort is faster in the best case. However, one advantage of selection sort over bubble sort is that selection sort does fewer swaps in the typical case. This can be a concern in practice since swaps take a bit more time (since you have to write to a temporary variable, so its three steps to do one swap). Indeed, selection sort does at most one swap in each pass in the inner loop, namely it only swaps at the last step of the inner loop. So it does at most $N-1$ swaps in total. It still does $O(N^2)$ instruction steps though; so in the best case, selection is slower than bubblesort; but in the worst case, selection sort is faster than bubble sort even though they are both $O(N^2)$.

## Insertion Sort

*Insertion sort*, is similar to first two, in that it uses two nested loops. In particular, it is similar to selection sort in that it maintains a partition into two sublists: a list of sorted elements at the front, and the rest. The size of the sorted part is increased by one by the end of each pass through the inner loop. However, whereas selection sort considers all the remaining unsorted elements (in the "rest" of the list) and find the smallest one, insertion sort considers only the first element in the rest of the list; insertion sort finds where this element belongs relative to the sorted front part of the list, and inserts this next element into the proper position.

The algorithm goes through an outer loop $N-1$ times. In the $i$th pass through the loop (starting at $i = 1$), the algorithm inserts element at index $i$ into its correct position with respect to the elements up to and including position $i-1$, which are already in their correct order.

How does the algorithm put the element at index $i$ into its correct position with respect to elements at indices 0 to $i-1$? The idea is first get the element at index $i$ and remember it. Then search backwards from index $i-1$ until you find the right place for this element to be inserted (the correct position). As you search back, shift forward by 1 position any element that is bigger than the element to be inserted, creating a hole that moves backwards through the list as the elements are shifted forward. Once a list element is found that less than or equal to the element to be inserted, insert the save element (formerly at position `list[i]`) into the hole (at position `k`).

```
for i = 1 to list.size - 1   {  //  index of element to maybe move
   e =  list[i]
   k = i
   while (k > 0) and ( e < list[ k - 1] ){   // Check if list[k-1]
      list[k] =  list[k - 1]                 // needs to move forward.
      k  = k -1
   }
   list[k] = e
}
```

What is the time complexity of insertion sort, specifically *how many times is the body of the inner loop executed* ? The outer goes from i = 1 to N-1, and the inner goes from k = i down to 1 (in the worst case). You can verify that again the answer is:

$$1 + 2 + \ldots N - 1 = \frac{N(N-1)}{2}$$

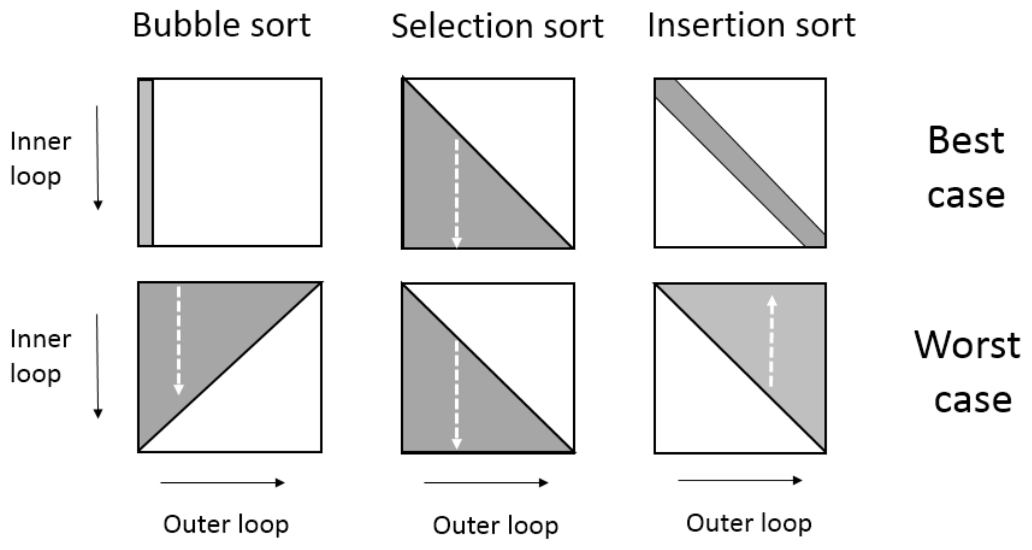
which again is $O(N^2)$.

Note that the inner loop only continues as long as the while condition is met. For example, if `e` were greater than `list[i-1]` already at the start of the `while` loop, then the inner loop would end immediately because element $i$ would already be in its correct position.

Note that insertion sort doesn't use swaps. As we search back through the list and shift elements forwards, we are not swapping. Rather we are just shifting. It is like we do a circular shift (recall lecture 5 slide 21) but only only on part of the list.

## Comparison of the algorithms

In the lecture, I discussed the best and worst cases of the three algorithms using the figure below. Think of the two dimensions of the square as the indices of inner and outer loops, so at any time within the inner loop of the algorithm(s) you are at one of the positions in the square. The algorithm goes column by column (each value of the outer loop). The grey regions show the values that the algorithm actually reaches.



## Inheritance

In our daily lives, we classify the many things around us. The world has objects like "dogs" and "cars" and "food" and we are familiar with talking about these objects as classes: "Dogs are animals that have four legs and people have them as pets and they bark, etc". We also talk about specific objects (instances): "When I was growing up I had a beagle named Buddy. Like all beagles, he loved to hunt rabbits."

We also talk about classes of objects at different levels. For example, take animals, dogs, and beagles. Beagles are dogs, and dogs are animals, and these "is-a" relationships between classes are very important in how we talk about them. Buddy the beagle was a dog, and so he was also an animal. But certain things I might say about Buddy make more sense in thinking of him as an animal than in thinking about him as a dog or as a beagle. For example, when I say that Buddy *was born* in 1966, this statement is tied to him being animal rather than him being a dog or a beagle. (Being born is something animals do in general, not something specific to dogs or beagles.) So being born is something that is part of the "definition" of a being an animal. Dogs automatically "inherit" the being-born property since dogs are animals. Similarly, beagles automatically inherit it since they are dogs, and dogs are animals.

A similar classification of objects is used in object oriented programming. In Java, for example, we can define new (sub)classes from existing classes. When we define a class in Java, we specify certain fields and methods. When we define a subclass, we say that the subclass "extends" the superclass. The superclass is often called the *base class* or *parent class.* The subclass is often called *derived class* or *extended class.*

We say that a subclass *inherits* the fields and methods of the superclass, namely by default the subclass automatically has the same field and methods as the superclass. We also may introduce entirely new fields and methods into the subclass. Alternatively, some of the fields or methods of the subclass may be given the same names as those of a superclass class. We will examine these choices over the next few lectures.

## Example

Suppose we have a class `Dog`.

```
class Dog  {
   String        dogName
   String        ownerName
   int           serialNumber
   void    Dog(){ ..  }
     :
   void    bark(){
        System.out.println("woof");
   }
}
```

Below are some examples of subclasses of `Dog`. When we declare the `Beagle`, `Doberman`, `Poodle` classes, we don't need to re-declare all the fields of the `Dog` class. These fields are automatically inherited, because of the keyword `extends`. We also don't have to re-declare the `bark` method, since this method is inherited. We can define other methods that are specialized to specific breeds of dogs. For example:

```
class Beagle extends Dog{
   Rabbit  hunt(){ ...
   }
}
class Doberman extends Dog{
   void  fight(){  ...
   }
}
class Poodle extends Dog{
   void  show(){  ...
   }
}
```

Here I will leave the implementations to your imagination.

## Constructor chaining

Constructor methods are not inherited. The reason is that an object belongs to exactly one class, and the object is constructed by the constructor of that class. So it would make no sense to inherit the constructor of superclass. That said, since fields and methods of the superclass are inherited, so a constructor does make use of the superclass'es constructor, namely it uses the superclass'es constructor to define its instance fields.

When an object of a subclass is instantiated using one of this subclass's constructors, the fields of the object are created including the fields of the superclass and the fields of the superclass'es superclass, etc. This is called *constructor chaining*. How is it achieved ?

The first line of any constructor is

```
super(...);   //  possibly with parameters
```

If you leave this line out, then the Java compiler puts in the following with no parameters:

```
super();
```

This causes the superclass'es no-argument constructor code to be executed, and fields created and possibly initialized. Note that the superclass may have its own `super(..)` constructor with or without parameters, which causes the fields of *all* the ancestor classes automatically to be inherited and initialized.

The following example illustrates some of the details of constructor chaining. The superclass `Animal` has two constructors. The subclass `Dog` constructor chooses among them by including parameters of the `Dog` constructor's `super()` calls to match the signature (number and types of arguments) of the superclass constructor. Specifically, the class `Dog` has a `String` field that specifies the owner. The `Dog(String birthplace, String owner)` constructor could in principle have used either the `Animal()` or `Animal(String birthplace)` constructor. It does the latter by calling `super(birthplace)`. If we comment out that constructor call then it would instead defaulted to a `super()` call whihc would assign the birth date but not the birth place.

```
import java.util.Date;

class Animal {
   Date   birthdate;
   String  birthplace;

   Animal() {
      this.birthdate  =  new Date();
   }

   Animal(String birthplace ) {
      this();
      this.birthplace = birthplace;
   }
}

class  Dog  extends  Animal {
   String name;

   Dog() {  }    //   automatically calls super().

   Dog(String birthplace,  String  name) {
      super(birthplace);
      this.name  = name;
   }

   public static void main(String[] args) {
      Dog d1 = new Dog();
      Dog d2 = new Dog( "New York City", "Fluffy");
   }
}
```

If the superclass does not have a no-argument constructor, then the subclass'es `super()` call will not work, and thus the subclass cannot have a no-argument constructor (and a compiler error would occur. Similarly if you call `super(...)` with parameters that don't match a constructor from superclass, then the compiler will give an error.

One more detail: Java does not allow you to write `super.super`. There is no way for a sub-class to explicitly invoke a method from the superclass'es superclass.

ASIDE: A subclass can declare a field with the same name as the one in the superclass. This "hides" the inherited field, and makes makes code difficult to understand, so it is not recommended.

If you want to learn more about how subclasses work, see online tutorials

## Overloading versus Overriding, and the method signature

When a subclass method and superclass method have the same method name and the same number, types, and order of parameters, then we say that the subtype method *overrides* the supertype method. When the method name is the same but the type, number, or order of parameters changes, then we say the method is *overloaded*.

We have seen examples of overloading in previous lectures (such as the `add` and `remove` methods of the ArrayList and LinkedList classes). We have also seen examples today, with constructor methods e.g `Dog`.

Overriding a method is different from overloading it. Overloading a method means having two versions of the method with different parameters. Overriding can only occur from a child class (subclass) to parent class (superclass), whereas overloading can occur either within classes (as in `Dog`) or between a child and parent class. For an example of the latter, suppose class `Dog` has a method:

```
bark() {
   System.out.print(``woof'');
}
```

We might define a subclass `AnnoyingDog` with the following `bark` method:

```
bark(int n){
   for (int i;  i < n; i++)  {
       System.out.print(``woof'');
}
```

This would be overloading between classes. Note that the `AnnoyingDog` class would have both `bark` methods.

### [**ASIDE (Feb. 8): when can you change modifiers and return type?**]

As you know, a method is defined by an visibility (access) modifier, a return type, a method name, and method parameters (order and type). The term *method signature* refers just to the method name and the parameters (types and order). The return type and access modifiers are not part of the method signature.

When overloading a method, we keep the method name and we change the parameter types and/or order. (If we had the same parameters, then it would be overriding, not overloading.) With overloading, we can also change the access modifier and the return type. The reason we can do so is that we think of overloaded methods as completely different methods, and the differences can be detected by the compiler (at compile time) and the JVM (at runtime), namely by examining the parameters of the method.

When one overrides a method, however, one is more constrained. Here, one cannot change the access modifier and return type willy-nilly. If the superclass's method returns a primitive type, then the subclass method's must return exactly the same primitive type. If the superclass's method returns a reference type, then the subclass method must either return that same reference type or a narrower reference type. For example, suppose you have a (trivial and useless) method `me()` in the super class A:

```
  A  me(){ return this;}
```

In the subclass B, you might want to override the method as follows:

```
  B  me(){ return this;}
```

This would be fine, and indeed it might make the code more readable. Java allows this change in return type because it really shouldn't create a problem. (If you want to read more about this, you can google "covariant return type Java".)

As for changing the access modifier when overriding, Java requires that the access be the same or greater. The spirit here is that an instance of a subclass should be able to do anything (including being seen by other classes) that an instance of the superclass object can do.

[**end of ASIDE**]

Finally, here are some examples of subclasses of `Dog` where we have overrided (or overridden) the method `bark`:

```
class Beagle extends Dog{
   void  bark(){
      System.out.println("aaaaawwwwooooooo");
   }
}

class Doberman extends Dog{
   void  bark(){
      System.out.println("GRRRR!  WO WO WO!");
   }
}
```

Consider the three examples below.

```
Dog    myDog1 = new Dog();
myDog1.bark();                  //  prints "woof"  -- see previous page

Beagle myDog2 = new Beagle();   //  prints "aaaaawwwwooooooo"
myDog2.bark();

Dog    myDog3 = new Beagle();   //  prints  ????
myDog3.bark();
```

The third example is allowed, even though the variable type is different than the constructor. We will discuss this important case more in an upcoming lecture.

## Java `final` modifier

The `final` modifier provides some flexibility on when we can extend classes or now.

```
final class Dog  {
    :
}

class Beagle extends Dog  {
   :
}
```

In this case, we would get compiler error in the `Beagle` definition because `Dog` cannot be extended.

Java library classes such as the wrapper classes `Integer, Double,` etc are all `final`, as are `Math` and `String`.

Methods in a class can also be final. This means that a subclass cannot override them. So for example if class `Dog` is not final and it has a method `bark()`, then if we make this a `final` method, then subclasses of `Dog` will not be able to override it. You would get a compiler error if you tried to redefine `bark()` as below. (`Beagle` would still inherit the `Dog.bark()` method though!)

```
class Dog  {
    final void bark(){ ...  }
}

class Beagle extends Dog  {
    void bark(){ ...  }       // compiler error

}
```

[ASIDE: What about `final` fields in a class? One cannot override fields, but that's not something I want you to think about. (Its called hiding a field. I've mentioned it previously.) ]

A common usage of the `final` modifier for variables is has nothing to do with inheritance. Rather, if we declare a variable as `final` then it means we cannot change the value of the variable once it has been initialized. This is usually done for constants in your program. For example,

```
final int x = 3;
x = 10;
```

would cause a compiler error, since x was assigned a value and then we want to change it. This hold for reference types too. For example, the second line below would cause a compiler error:

```
final Dog myDog = new Dog(``Willie'');
myDog = new Dog(``Max'');
```

Although you cannot change the reference, you *can* change the object that is being referenced. For example, the following is fine:

```
final Dog yourDog = new Dog(``Snoopy'');
yourDog.setName(``Max'');
```

The same would hold with a `final` variable that is referencing an array. You cannot change

```
final int[] arr = {1, 2, 3};
arr[0] = 7;
```

would be fine, but

```
arr  =  new int[]{2, 7};
```

would cause a compiler error.

## Java `Object` class

In Java, every class directly extends exactly one other class,[9] with one exception to be discussed below. The definition of a class is of one of the two forms:

```
    class MyClass

    class MySubclass extends MySuperclass
```

where `extends` is a Java keyword, as mentioned above. If you don't use the keyword word `extends` in the class definition then Java automatically makes `MyClass` extend a class called `Object`. So, the first definition above is equivalent to

```
class MyClass extends Object
```

The `Object` class contains a set of methods that are useful no matter what class you are working with. We will discuss some of these methods shortly.

An instantiation of any class is always some object, and so this object either belongs to class `Object` or some subclass of `Object`, or some subclass of subclass of `Object`, etc. As stated under the `Object` entry in the Java API: the class "Object is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class."

### Java `equals( Object )` method

In natural languages such as English, when we talk about particular instances of classes e.g. particular dogs, it always makes sense to ask "is this object the same as that object?" We can ask whether two rooms or dogs or hockey sticks or computers or lightbulbs are the same. Of course, the definition of "same" needs to be given. When we say that two hockey sticks are the same, do we just mean that they are the same brand and model, or do we mean that the lengths and blade curve are equal, or do mean that the instances are identical as in, "is that the same stick you were using yesterday, because I thought that one had a crack in it?"

In Java, the Object class has an `equals( Object )` method, which checks if one object is the same instance as the other, namely if `o1` and `o2` are declared to be of type `Object`, then `o1.equals(o2)` returns true if and only if `o1` and `o2` reference the same object. For the `Object`

[9]C++ allows for *multiple inheritance*, that is, a class can extend more than one superclass. This leads to complications, for example, if two superclasses have a method with a common name, which one gets inherited?

class, the `equals(Object)` method does the same thing as the "==" operator, namely it checks if two referenced *objects* are the same.

For many other classes, we may want to override the `equals(Object)` method, namely use a less restrictive version of the `equals` method. It is also possible in principle to overload it, for example, by defining an `equals(Dog)` method in the `Dog` class. However, I will not discuss overloading of the equals() method, and my understanding is that it is generally better not to use it to avoid confusion and surprises.

We have an intuitive notion of what we mean by 'equals'. But since the `equals()` method is so fundamental in Java, the designers of the Java language specified quite formally how the method should behave. It is very similar to the mathematic definition of *equivalence classes* which you learn about in MATH 240. When writing your own classes and overriding this method, you should be aware of this. See details of the equals(Object) method here. For example,

- `x.equals(x)` should always be true
- `x.equals(y)` should have the same true or false value as `y.equals(x)`
- if `x.equals(y)` and `y.equals(z)` are both true, then `x.equals(z)` should be true.

The `String` class overrides the `equals(Object)` method. (Recall the discussion of String.equals() in lecture 6.)

What about other classes? Consider the fictitious `Shape` class which I keep talking about in the slides. When should we say that two `Shape` objects are equal? We might use the default definition of equals inherited from the Object class and say that two Shape objects are equal if they are literally the same object. Alternatively, we might say that all triangles are equal or all circles are equal (regardless of their size), whereas a triangle and circle are not equal. Or we might say two Shape objects are equal if they have the same color. Or we might say that two Shapes are equal if they are the same type (triangle vs. circle) *and* same color *and* size, etc. Its arbitrary. Note here I am just repeating the ideas discussed above (hockey stick) but now I am more concretely discussing Java equals.

Another example mentioned is the Java `LinkedList` class which also overrides the `equals(Object)` method. If a `LinkedList` object invokes this method on some other object, the result will be true only if that other object is also a LinkedList object and each of the elements in the two lists are "equal", in particular, the elements of the two linked list objects are "equal" according to the `equals` method of these elements.

## Java `Object clone()` method

Another commonly used method in class `Object` is `clone()`. Cloned objects are supposed to obey the following:

- The expression `x == x.clone()` should return false.
- The expression `x.equals( x.clone())` should returns true (suggested, but not required).

These two conditions make intuitive sense. The point of cloning is to create a different object instance (first condition), but the clone is suppose to be the same as the original in whatever sense we define "same" to mean for that object's class. Note that the second condition doesn't hold for `Object` objects. But that's ok: in fact, one is not allowed to clone an `Object` object. (See Exercises.)

[ASIDE: I will not discuss cloning much more, since there are some subtleties to how it is allowed to be used, and I don't want to spend too much time on these details.]

## Inheritance and Visibility Modifiers

Last lecture we discussed the basics of inheritance and how we can organize classes into a hierarchy. This might bring to mind the notion of packages which we discussed in lecture 8, and which specify how class files are stored in a file system and which also determine which class are visible to which (via access/visibility modifiers). The inheritance hierarchy is not the same as the package hierarchy. That said, there are some relationships between the two.

First let's recall the basics about visibilty. If a class is declared to be public, then this class is visible from any other class – although it will need to be imported into the other class if it is in a different package.

Suppose class B extends class A – and thus class A must be visible from class B. Which class A members are visible from class B? By *members* of a class A, we mean the fields, methods, inner classes of A. By *visible* here, I mean that a class B method can name/refer to the class A member

As a first example, let's suppose that the following two classes are in the same package.

```
class Dog{
    String name;
    :
}

class Beagle extends Dog {

   Beagle(String name){
      this.name = name;
   }
}
```

In this case, since Beagle extends Dog, Beagle inherits the `name` field. Since Beagle and Dog classes are in the same package, no modifier is needed and Beagle methods can use `this.name` to access the `name` field. For example, the above Beagle constructor does this.

If Beagle and Dog were in different packages, then in the above code we would need to add a public modifier to both the class Dog as well as the field `name`. We would also need to add an import statement to Beagle,

```
  import package1.Dog;
```

so that the compiler knows where to find the Dog class. See the full example in the slides.

What if we were to put a `private` modifier on the `name` field of Dog?

```
class Dog{
   private String name;
      :
}
```

In this case, we would get a compiler error in Beagle, since `name` field would not visible in Beagle. Heads up, however! A Beagle object would still have the field `name`. The problem is that the `name` is not accessible, i.e. Beagle's methods cannot state `this.name` because this gives a compiler error as just mention. Officially, one says that the `Beagle` subclass does not inherit the private field `name` from the superclass. According to Oracle, which maintains Java (!): "The subclass does not inherit the private members of its parent class." . (See link.) However, what this actually means is that the field is not *visible/accessible* to the subclass, in the sense that methods cannot mention the field name. *But* a `Beagle` object will still have the field. You can verify this yourself in your IDE. So how can it assign a value to this field?

Rather than accessing the field by its name, a `Beagle` object has to access it using getters and setters. For example, suppose `Dog` class has a setter method `setName`():

```
package package1;
class Dog{
  private String name;
  public void setName(String name){
      this.name = name;
  }   :
}
```

Rather than accessing the field by its name, a `Beagle` constructor can use this inherited setter method to set its `name` field:

```
package package2;
import package1.Dog
class Beagle extends Dog {

    Beagle(String name){
        this.setName(name);   <------  see here
    }
}
```

Admittedly, it is a bit confusing to say that the subclass doesn't *inherit* the private field, when plainly the subclass object does have the field. But we have to live with it, since we can't really argue with Oracle at this point. :)

## ASIDE: the `protected` modifier

Java provides an access modifier `protected` that comes between "package-private" and `public`. This modifier handles the situation above, in which a subclass is in different package. The `protected` modifier allows access to a field or method in the parent class, regardless of whether the subclass is in the same package or in a different package. We will not be using this modifier and I won't examine you on it, but it is worth mentioning – especially if you do any reading on your own, since you might run into it.

An example of the a protected method is the `Object.clone()` method. I am not going to explain why this method is protected. It is well beyond the scope of the course, so let's just move on.

## Object.hashCode)()

The Object classes's hashCode() method returns a positive integer. You can think of it as the address of the object, although this is not strictly required. Many classes override the hashCode() method. We will see some examples later in the course.

## Object.toString())()

The Object class also has a toString() method. The Object class'es toString() method returns a string as follows:

```
className +  @ + Integer.toHexString( hashCode() )
```

So you can see it uses the hashCode, written in hexadecimal. For example, if we write

```
  Object obj = new Object();
  System.out.println( obj );
```

then it might return something like `java.lang.Object@5305068a`.

The `String` class overrides the `toString()` method in an obvious way: if a string object invokes the toString() method, then it just returns itself. Why would it do something that seems neither interesting nor useful? In fact, it is quite useful. Suppose you have a string variable `s` and you want to print its value. Then you would write:

```
 System.out.println( s );
```

If `String` *didn't* override the Object classes toString() method, then the above print statement would produce something like `java.lang.String@523043` which is obviously not what we want to print. Rather we want to print out the string itself!

For other classes, a common use of `toString()` is is in debugging, where you want to print information about an object to the console. Suppose you had a class Dog. If you override toString() in this class, then you might write the method so that it returns a string with the dog's name and birthdate and owner. *As another example, check out the Course.toString() method in Assignment 1.*

## Type Conversion (or Casting)

We have seen how variables of one type can be cast to another. So far we've only considered primitive types and autoboxing/unboxing. We saw how some primitive types can be narrower or wider than others. Similar ideas can be used for reference types. With reference types, a subclass is narrower than its superclass, and a superclass is wider than its subclass. e.g. If class `Beagle` extends class `Dog`, then class `Beagle` is narrower than `Dog`, or equivalently, `Dog` is wider than `Beagle`.

Heads up! Although a subclass is by definition narrower than the superclass, the subclass will typically have more fields and methods than the superclass (since the subclass inherits the fields and methods from superclass). So when we talk about narrowing down, we're typically talking about a bigger object, i.e. with more fields.

Conversions can also occur between reference types. However, *reference type conversions do not change the referenced object.* Rather, the conversion only tells the compiler that you (the programmer) expect or allow the object to be a certain type at runtime.

Widening conversions from a subclass to superclass occur automatically. Here we say that we are casting *upwards* (upcasting). Upcasting is sometimes called *implicit casting.* We cast *downwards* ("downcasting") when we are casting from a superclass to a subclass. Like with primitive types, when we downcast reference types we need to be explicit about it.

In fact, we have seen upcasting before, e.g. last lecture we saw:

```
Dog  myDog = new Beagle();
```

This is analogous to:

```
double  myDouble = 3;   //    from int to double.
```

We have not seen downcasting before for reference types, however. So let's see how this works with some examples.

Consider the two instructions:

```
  Dog    myDog  =  new Poodle();     //  upcast,  widening
  myDog.show();
```

The second line gives a compiler error because show() is not a method in the Dog class. Although the first line says myDog will reference a Poodle when the program actually runs, the compiler ignores this fact. The compiler only cares that `myDog` is declared to be of type `Dog` and the class `Dog` doesn't have a method `show()`.

Let's now take the same first line as above and try something else.

```
Dog  myDog = new Poodle();  // Upcasting.
Poodle  myPoodle = myDog;
```

The second line also generates a compiler error, since the *implicit* downcast Dog to Poodle not allowed. So how about the following?

```
Dog  myDog = new Poodle();           // Upcast
Poodle  myPoodle = (Poodle) myDog;   // Allowed (explicit downcast)
        myPoodle.show()
       ((Poodle) myDog).show();
```

The compiler is fine with all four lines above. Moreover, there will not be a runtime error, since `myDog` and `myPoodle` will reference a `Poodle` at runtime.

## `instanceof` operator

Sometimes we wish to check if an object is an instance of a particular class. We can use the `instanceof` operator for this. The `instanceof` operator takes two arguments: the first is a referencetype variable `var`; the second is a class `C`, ie.

```
var instanceof C
```

The operator returns true if and only if the object referenced by `var` is an instance of the class `C` or any class that extends `C`. Again, examples are the best way to understand this.

The first example just revisits the same idea that Dog, Beagle, and Poodle objects can all be considered instances of Dog.[10]

```
Dog  d = new Dog();
System.out.println( d instanceof Dog);      // true

Beagle b = new Beagle();
System.out.println( b instanceof Dog);     // true

d = new Poodle();
System.out.println( d instanceof Dog);     // true

System.out.println( d instanceof String);     // false
```

Another example is how we can use instanceof to make sure that downcasting will not cause a run time error:

```
class  Test {
   static void dogMethod(Dog dog) {
      if  (dog instanceof Beagle) {
          Beagle b = (Beagle) dog;
          b.hunt();              //  or just ((Beagle) dog).hunt()
      }
   }
}
```

In this case, if the method's argument is not a `Beagle` object, then the method won't do anything, which may be better than a runtime error!

Another example is how we sometimes use `instanceof` when overriding equals() :

```
public class Shape {
   public boolean equals( Object obj ) {
      if(obj instanceof Shape) {
          return this.getArea == ((Shape) obj).getArea();
      else return false;
   }
}
```

Here we'll first check that argument `obj` is indeed of the right type, in this case `Shape`. Once that's been verified, we can make our comparison of `this` object with the object that we wish to compare it to. This comparison only makes sense if the object's are both `Shape`'s, and in particular, they

[10]Yes, I realize this is confusing when you first read it. You might think that a Poodle object is an instance of the Poodle class, and that a Poodle object is *not* an instance of the Dog class! But in fact, the way we will talk about it is the former, not the latter. And I promise you: it will make sense as you work with it more an more, just like in real life you would be very comfortable saying "That poodle over there is a dog!"

have the `Shape.getArea()` method. If the object passed in the argument is the wrong type, then the method would return false.

ASIDE: note that the compiler does require that you have the modifier `public` and return type `boolean` when overriding the `Object.equals()` method. You will get a compiler error if you use different modifiers.

Finally, a common mistake that programmers make when first using inheritance in Java is they make excessive use of the `instanceof` method. They write things like:

```
Dog  dog;
:
if  (dog instanceof Doberman)
   ((Doberman) dog).threaten();
else if (dog instanceof Beagle)
   ((Beagle) dog).howl ();
```

where they check what type of object a variable references and then call methods appropriate to that object, while casting to make sure the compiler doesn't complain. This is considered bad style, and it can also lead to code that is difficult to maintain since there are too many cases to test. Instead, one uses another method (called *polymorphism*). We'll return to this next lecture.

By now you should be familiar with the *Java API*. The API (*application program interface*) gives a user many predefined classes with implemented methods. What makes the API an "interface" is that the implementation is typically hidden. You are only given the name of a class, the signatures of the methods (along with their return type and modifiers), comments on what these methods do, and possibly some public fields of the class.

The word "interface" within "Java API" should not be confused with related but different usage of the word, namely the Java reserved word `interface`, which is what we'll discuss next.

## Java `interface`

In Java, if we specify *only* the modifiers, return types, and signatures of the methods of a class but we don't specify the implementation, then technically we don't have a class. What we have instead is an `interface`. So, an `interface` is a Java program component that is like a class, but it doesn't contain the bodies of the methods.

We say that a class `implements` an interface if the class implements each method that is defined in the interface. In particular, the method signatures must be the same as in the interface. So, if we say a class `C` implements an interface `I`, then `C` must implement all the methods from interface `I`, which means that `C` specifies the body of these methods. In addition, the class `C` can have other methods.

## e.g. `List` interface

Consider the two classes `ArrayList<T>` and `LinkedList<T>` which are used to implement lists. These two classes share many method signatures. Of course, the underlying implementations of the methods are very different, but the result of the methods are the same, in the sense of maintaining a list. For example, if you have a list and then you remove the 3rd item, you get a well defined result. This new list should not depend on whether the original list was implemented with a linked list or with an array.

The `List<T>` interface includes familiar method signatures such as:

```
void       add(T o)
void       add(int index, T element)
boolean  isEmpty()
T          get(int index)
T          remove(int index)
int        size()
```

Both `ArrayList<T>` and `LinkedList<T>` implement this interface, namely they implement all the methods in this interface.

The `ArrayList` class also has other methods that are not part of the `List` interface. For example, the `ensureCapacity(int)` method will expand the underlying array to the number of slots of the input argument if the current array has fewer than that many slots. The `trimToSize( )` method does the opposite. It will shrink the length of the underlying array so that the number of slots is equal to the current number of elements in the list. Note that these two methods make no sense for an `LinkedList`.

There are also methods for the `LinkedList` class that would not be suitable for `ArrayList` class, namely `addFirst` and `removeFirst`. There is nothing special about these operations for array lists. They would be implemented exactly the same as on any other index and they would be expensive because of the shifts necessary. If these operations are commonly needed, then one would tend to use a linked list instead since these operations are inexpensive for linked lists.

Why is the `List` interface useful? Sometimes you may wish to write a program that uses either an `ArrayList<T>` or a `LinkedList<T>` but you may not care which. You want to be flexible, so that the code will work regardless of which type is used. In this case, you can use the generic Java interface `List<T>`. For example,

```
void myMethod( List<String> list ){
       :
    list.add("hello");
       :
}
```

Java allows you to do this. The compiler will see the `List` type in the parameter, and it will infer that the argument passed to this method will be an object of some class that implements the `List` interface. As long as `list` only invokes methods from the `List` interface, the compiler will not complain. You can also do thinks such as the following, namely have the same variable `list` reference different types of lists at different times in the program.

```
   List<String>        list;

   list  =  new  ArrayList<String>();
   list.add("hello");
   :
   list  =  new  LinkedList<String>();
   list.add( new String(\hi") );
```

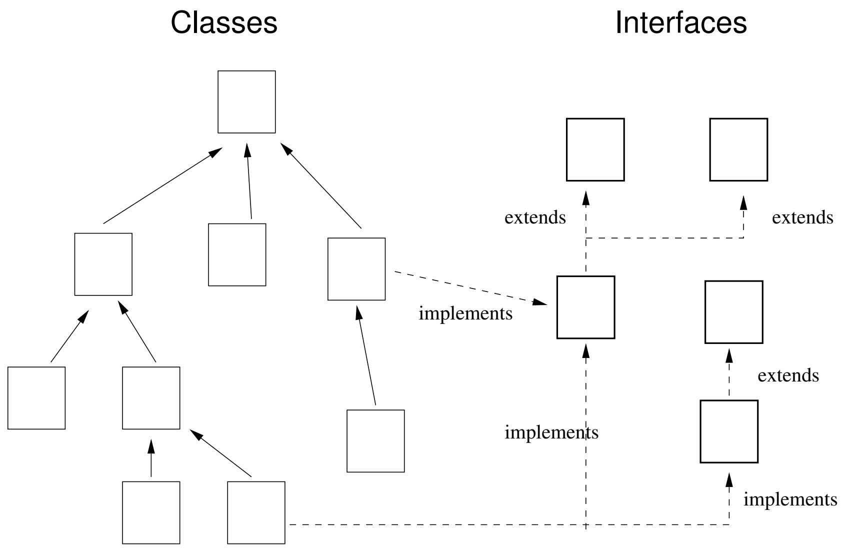
Or, more usefully, we could have some method that takes in a list and doesn't care if it is Arraylist or LinkedList.

```
void  someFlexibleListMethod(    List<String> list, String s ){
  :
   list.add(s);
     :
   list.remove(0);
}
```

Note that the method would only be able to use methods from the List interface. But this might be enough.

## Interfaces and the Java class hierarchy

Earlier when we discussed Java classes and their inheritance relationships, we considered a hierarchy where each class (except `Object`) extends some other unique class. See below left.



How do Java interfaces fit into the class hierarchy? As shown above right, an interface is another "node" in the inheritance diagram. We used dashed arrows to indicate inheritance relationships that involve interfaces.

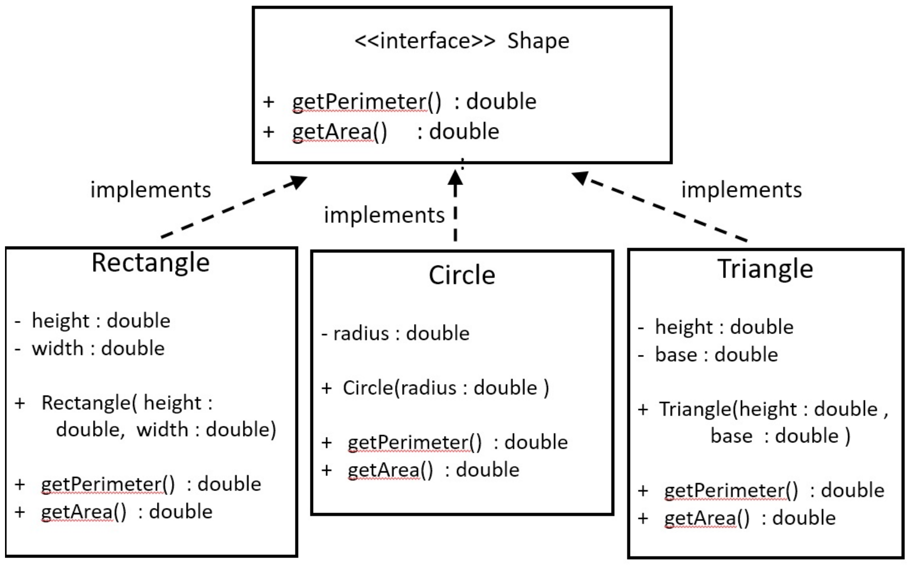
- If a class `C` implements an interface `I` then we put a dashed arrow from `C` to `I`. Recall that a "class implements an interface" means that the class provides the method body for each method signature defined in the interface.
- One interface (say `I2`) can extend another interface (say `I1`). This means that `I2` inherits all the method signatures from `I1`. We don't need to write the method signatures out again in the definition of `I2`. In the class diagram, we would put a dashed line from `I2` to `I1`.
- Each class (other than `Object`) directly extends exactly one other class. (Why? Suppose a class `C` were allowed to extend multiple classes(say **A** and `B`). Then it could happen that there might be a method conflict – superclasses **A** and `B` could contain a method with the same signature but with different bodies. Which of these methods would an object of class `C` inherit? )
- A class `C` can implement multiple interfaces. The parent interfaces can even contain the same methods. This is no problem since the interfaces do not contain the method bodies so there could be no issue with ambiguity in case of conflict. We would say:
  `class C2 extends C1 implements I1, I2, I3`

As another example of how interfaces can be useful, consider the following classes:

```
class  Shape {
  double  getPerimeter() { ...};
  double  getArea() { ... } ;
}
```

```
class   Rectangle  extends  Shape{
  double  getPerimeter() { ...};
  double  getArea() { ... } ;
}

class   Circle  extends  Shape{
  double  getPerimeter() { ...};
  double  getArea() { ... } ;
}

class   Triangle  extends  Shape{
  double  getPerimeter() { ...};
  double  getArea() { ... } ;
}
```

It is not obvious how to implement these methods for a general Shape class, so you probably wouldn't invoke these methods unless you had a specific kind of shape such as a Rectangle, Circle, or Triangle. For this reason we would use an interface instead:



```
  interface  Shape {
   double  getPerimeter();
   double  getArea();
  }

  class   Rectangle  implements  Shape{
    double  height;
   double  width;
```

```
    Rectangle(  double h,  double w ){ height = h; weight = w;}
    double   getArea(){   return height * width;   }
    double   getPerimeter(){   return 2*(height + width);  }
  }

  class   Circle implements  Shape{
   double  radius;
   Circle(  double r ){ radius = r; }
   double getArea(){   return   MATH.PI * radius * radius; }
   double getPerimeter(){ return 2*MATH.PI * radius }
   }

  etc...   Triangle
```

We these classes, we can then write things like:

```
  Shape  s  =   new  Rectangle( 30, 40 );
  s  =   new   Circle(  2.5  );
  s  =   new   Triangle(  4.5,  6.3  );
```

### A Motivating Example for Java Abstract Classes: Circular

Here is an example to illustrate one of the limitations of interfaces, and motivates the use of abstract classes which come next.

Many geometrical shapes have a `radius`, for example, of a circle, sphere, and cylinder. Suppose we wanted to define classes `Circle`, `Sphere`, `Cylinder` of shapes that have a radius. In each case, we might have a private field `radius` and public methods `getRadius()` and `setRadius()`. We might also want a `getArea()` method.

We could define an interface `Circular` as follows:

```
public interface Circular{
   public double getRadius();
   public void   setRadius(double radius);
   public double getArea();
}
```

and define each of these classes to implement this interface. The problem with such a design is that we would need to define each class to have a local variable `radius` and (identical) methods `getRadius()` and `setRadius()`. Only the `getArea()` methods would differ between classes. We could do this, but there is a better way to deal with these class relationships.

## Abstract classes

The better way is to use a hybrid of a class and an interface in which some methods are implemented but other methods are specified only by their signature. This hybrid is called an `abstract class`.

One adds the modifier `abstract` to the definition of the class and to each method that is missing its body. For example:

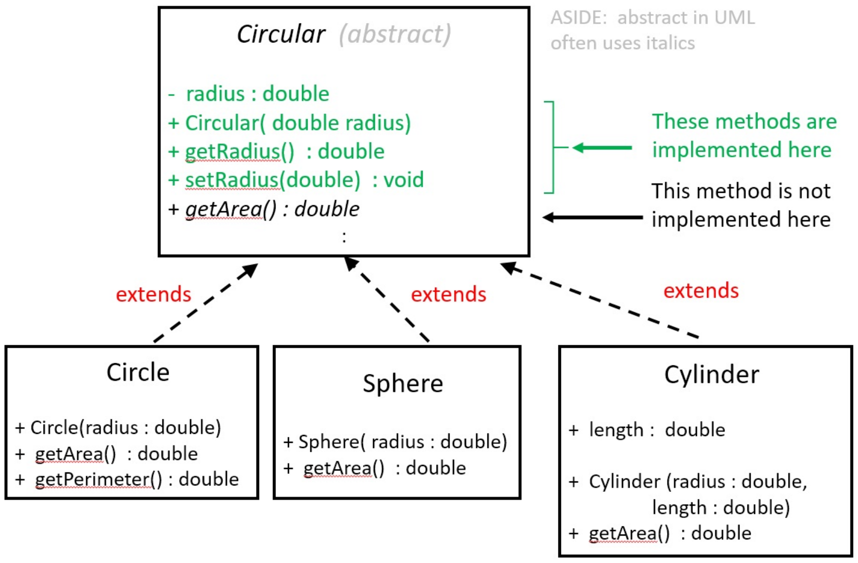
```
public abstract class Circular{

    private double radius;
    Circular(){};
    Circular(double radius){ this.radius = radius;   };

    public double getRadius(){ return radius;   }

    public void   setRadius(double radius){ this.radius = radius;    }

    public abstract double getArea();
}
```

This abstract class has just one abstract method `getArea()` that would need to be implemented by the subclass `Circle`, `Cylinder`, or `Sphere`. For example:

```
public class Circle extends Circular{

   Circle(double radius){ super(radius);   }

   double getArea(){
     double  r = this.getRadius();
     return  Math.PI * r*r;
   }

   double getPerimeter(){ return 2*MATH.PI * this.getRadius();  }

}

public class Cylinder extends Circular{
   double height;

   Cylinder(double radius, double h){
       super(radius);
       this.height = h;
   }

   double getArea(){  return 2* Math.PI * r * height; }
}
```

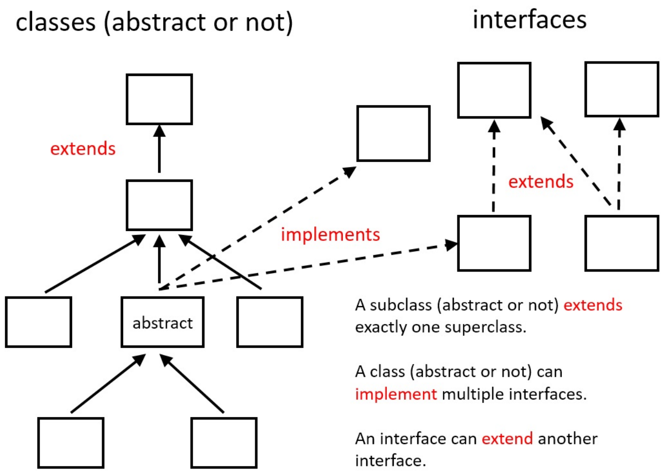
Note that the subclass `Circle`'s a method `getPerimeter()` would make no sense for a `Sphere` or `Cylinder` since perimeter is defined for 2D shapes, not 3D shapes. Similarly, a method like `getVolume()` would make sense for a `Sphere` and `Cylinder`, but not for a `Circle`.

Here is a UML diagram for the above classes. (ASIDE: abstract class names and abstract methods are shown in italics.)


*Circular* (abstract)
- radius : double
+ Circular( double radius)
+ getRadius() : double
+ setRadius(double) : void
+ *getArea() : double*
:
ASIDE: abstract in UML often uses italics
These methods are implemented here
This method is not implemented here
extends
extends
extends
Circle
+ Circle(radius : double)
+ getArea() : double
+ getPerimeter() : double
Sphere
+ Sphere( radius : double)
+ getArea() : double
Cylinder
+ length : double
+ Cylinder (radius : double, length : double)
+ getArea() : double

An abstract class *cannot* be instantiated. However, abstract classes do have constructor methods. The reason is that abstract classes can have fields and some implemented methods. So when the abstract class is extended by a concrete subclass, the subclass inherits these fields and methods. The subclass also provides the missing method bodies. Then, when the subclass is instantiated, the values of the inherited subclass fields are set by the superclass constructor (either via an explicit `super()` call, or by default). Thus, even if the superclass is abstract, it still needs a constructor.

Abstract classes also appear in class hierarchies/diagrams, along with interfaces:


classes (abstract or not)
interfaces
extends
implements
extends
abstract
A subclass (abstract or not) extends exactly one superclass.
A class (abstract or not) can implement multiple interfaces.
An interface can extend another interface.

Note that while a class can implement more than one interface, a class cannot extend two abstract classes. The reason for this policy is the same for why a class cannot extend two classes – namely if the two superclasses were to contain two different versions of an implemented method then it wouldn't be clear which of these two methods gets inherited by the subclass.

## Polymorphism

One can declare variables to have a type that is an abstract class, just as one can declare a variable to be of type class or of type interface.

```
C      c ;      //    C is a class
A      a ;      //    A  is an abstract class
I      i ;      //   I   is an interface
```

As we have seen, the declared type of a reference variable does not entirely determine the class of object that the variable can reference at runtime. At runtime, a variable can reference an object of its declared type, but it can also reference an object that is a "subtype" (subclass) of the variable's declared type. In the example above, `c` can reference any object of class `C` or subclass of `C` – for example, consider a Cat class and subclass SiameseCat. A variable `a` can reference any object whose class extends `A`, or a subclass of a class that extends `A`, etc. `i` can reference any object whose class implements `I`, or any object that extends a class that implements `I`, etc.

This property, that the object type can be narrower than the declared type, is called *polymorphism*. The name comes from Greek: poly means "many" and "morph" means forms.

When we discussed type conversion above, we concentrated on the type checking that is done by the compiler. When we discuss polymorphism, we assume a program has compiled fine, and we are concerned with which method is invoked at runtime. The method is determined by the class that the object belongs to. Consider, for example:

```
boolean  b;
Object  obj;
:                     //  some code not specified here
if (b)
   obj = new float[23];   // an array of floats
else
   obj = new Dog();
System.out.print(obj);   //  invokes the toString() method (*)
```

The compiler cannot say for sure which `toString()` method will be invoked since the compiler doesn't know for sure what the value of `b` will be when the `if (b)` condition is evaluated. Rather, the `toString()` method must be determined at runtime, when (*) is executed and the variable `obj` references either a `float[]` or a `Dog`. In each case, there will be a `toString()` method used which is appropriate for the object. (Recall that every class has a `toString()` method.)

In the slides I went over a few more examples: the example of `Dog.bark()` which is overridden by subclasses `Beagle` and `Doberman`, and example with the `Shape` interface, and an example with the `Circular` abstract class.

No lecture notes for today – please see slides

## Abstract Data Types (ADT)

We began our discussion of lists a few lectures ago by defining a list abstractly as a set things of a certain type and a set of operations that are applied to these things, such as getting, setting, adding and removing. We can describe the behavior of a list using these operations, without necessarily giving details on how the behavior is implemented. Indeed, we saw for lists that very different implementations can be used for achieving the same behavior (ignoring issues of time complexity, which sometimes do depend on the implementation).

A *list* is an example of an *abstract data type* (ADT). An ADT defines a data type by the values that data can have and operations on the data. An ADT is defined from the point of view of the user. What values and operations on these values are available to the use? An ADT is more abstract than a data structure. We will see two more ADT's in the next two lectures, namely the stack and the queue. We will see more examples later in the course.

Note that ADT's are not defined by a particular language. (Don't confuse the ADT list with the Java interface `List`.) They are related conceptually, but technically they are not the same thing. ADT's are an old idea in computer science and were around long before Java!

## Stack ADT

You are familiar with stacks in your everyday life. You can have a stack of books on a table. You can have a stack of plates on a shelf. In computer science, a *stack* is an abstract data type (ADT) with two operations: `push` and `pop`. You either push something onto the top of the stack or you pop the element that is on the top of the stack. A more elaborate ADT for the stack might allow you to check if the stack has any items in it (`isEmpty`) or to examine the top element without popping it (`top`, also known as `peek`).

Note that a stack is a kind of list, in the sense that it is a finite set of ordered elements. However, it has fewer operations you can apply on it. Unlike a list, a stack generally does not allow you to directly access the i-th element.

## Data structure for a stack

What is a good data structure for a stack? A stack is a list, so its natural to use one of the list data structures.

If you use an array list, then you should push and pop at the end of the list with `addLast()` or `removeLast`. [11] The reason is that if you add or remove from the front of an array list, you need to shift all the other elements each time which is inefficient. If you use a singly linked list to implement a stack, then you should push and pop at the front of the list, not at the back. The reason (recall) is that removing i.e. popping from the back of a singly linked list would be inefficient i.e. you need to walk through the entire list to find the node that points to the last element which you are popping. For a doubly linked list, it doesn't matter whether you push/pop at the front or at the back. Either work, but you have to be consistent: either do both at the front, or do both at the back.

[11] Here I give the Java-sounding names for these operations, although note that Java `ArrayList` class actually doesn't have these particular methods. But let's not worry about it, since the method name is not the point here.

**Example 1**

Here we make a stack of numbers. This example is mostly to illustrate the notation we'll use. The timeline goes left to right. We assume the stack is empty initially, and then we have a sequence of pushes and pops.

```
push(3), push(6), push(4), push(1), pop(), push(5), pop(), pop()
```

The elements that are popped will be `1`, `5`, `4` in that order, and afterwards the stack will have two elements in it, with `6` at the top and `3` below it. Here is how the stack evolves over time:

```
                   1               5
           4       4       4       4       4
      6    6       6       6       6       6      6
3     3    3       3       3       3       3      3
------------------------------------------------------
```

**Example 2: Balancing parentheses**

It often occurs that you have a string of symbols which include left and right parentheses that must be properly nested or balanced. (In this discussion, I will use the term "nested" and "balanced" interchangeably.) One checks for proper nesting using a stack.

Suppose there are multiple types of left and right parentheses, for example, `(`, `)`, `{`, `}`, `[`, `]`. Consider the string:

```
   ( ( [ ] ) ) [ ] { [ ] }
```

You can check for balanced parentheses using a stack. You scan the string left to right. When you read a left parenthesis you push it onto the stack. When you read a right parenthesis, you pop the stack (which contains only left parentheses) and check if the popped left parenthesis matches the right parenthesis. For the above example, the sequence of stack states would be as follows.

```
          [
      (   (   (                      [
   (  (   (   (   (       [       {  {   {
------------------------------------------------------
```

and the algorithm terminates with an empty stack. So the parentheses are properly balanced.

Here is an example where each type of parenthesis on its own is balanced, but overall the parentheses are not balanced.

```
    ( ( [ ) ] ) [ [ ] ] { [ } ]
```

```
             [
         (   (
     (   (   (
---------------- X  since next symbol is ")" which doesn't match top
```

The basic algorithm for matching parentheses is shown below. We assume the input has been already partitioned ("parsed") into disjoint *tokens*. For this example, a token can be one of the following:

- a left parenthesis (there may be various kinds)
- a right parenthesis (there may be various kinds)
- a string not containing a left or right parenthesis (operators, variables, numbers, etc)

```
ALGORITHM:  CHECK FOR BALANCED LEFT AND RIGHT PARENTHESES
INPUT:  SEQUENCE OF TOKENS
OUTPUT: TRUE OR FALSE (I.E. BALANCED OR NOT)

 while (there are more tokens) {
   token  =  get next token
   if token is a left parenthesis
      push(token)
   else {                                     //  token is a right parenthesis
      if stack is empty
         return false
      else {
         pop left parenthesis from stack
         if  popped left parenthesis doesn't match the right parenthesis
            return false
      }
   }
 }
```

**Example 3: HTML tags**

The above problem of balancing different types of parentheses might seem a bit contrived. But in fact, this arises in many real situations. An example is HTML *tags*. If you have never looked at HTML markup before, then open a web browser right NOW and look at "view → page source" and check out the tags. They are the things with the angular brackets.

Tags are of the form `<tag>` and `</tag>`. They correspond to left and right parentheses, respectively. For example, `<b>` and `</b>` are "begin boldface" and "end boldface". HTML tags are *supposed to be* properly nested. For example, consider

```
<b> I am boldface, <i> I am boldface and italic, </i> </b>
<i> I am just italic </i>.
```

The tag sequence is `<b><i></i></b><i></i>` and the "parenthesis" are indeed balanced, i.e. properly nested. Compare that too

```
<b> I am boldface, <i> I am boldface and italic </b>  I am just italic </i>
```

whose tags sequence is `<b><i></b></i>` which is not properly balanced. The latter is the kind of thing that novice HTML programmers write. It does make some sense, if you think of the tags as turning on or off some state (bold, italic). But the HTML language is not supposed to allow this. And writing HTML markup this way can get you into trouble since errors such as a forgotten or extra parenthesis can be very hard to find.

Many HTML authors write improply nested HTML markup. Because of this, web browsers typically will allow improper nesting. The reason is that web browser programmers (e.g. google employees who work on Chrome) want people to use their browser and if the browser displayed junk when trying to interpret improper HTML markup, then users of the browser would give up and find another browser.

See `http://www.w3schools.com` for basic HTML tutorials, if you are interested.

**Example 4: stacks in graphics**

The next example is a simple version of how stacks are used in computer graphics. Consider a drawing program which can draw unit line segments (say 1 cm). Suppose the pen tip has a *state* $(x, y, \theta)$ that specifies its $(x, y)$ position on the page and an angular direction $\theta$. This is the direction in which it will draw the next line segment (see below). The pen state is initialized to be (0,0,0), where $\theta = 0$ is in the direction of the $x$ axis.

Let's say there are five commands:

- `D` - draws a unit line segment from the current position and in the direction of $\theta$, that it, it draws it from $(x_0, y_0)$ to $(x_0 + \cos\theta, y_0 + \sin\theta)$. It moves the state position to the end of the line just drawn. Recall $\cos(0) = 1, \cos(90) = 0, \cos(180) = -1, \sin(0) = 0, \sin(90) = 1, \sin(180) = 0, ...$
- `L` - turns left (counter-clockwise) by 90 degrees
- `R` - turns right (clockwise) by 90 degrees
- `[` - pushes the current state onto the stack
- `]` - pops the stack, and current state $\leftarrow$ popped state

See the slides for a few examples.

Note that this simple language doesn't allow one to move the pen without drawing, except by returning to a position that the pen had been in previously. This means that the language can only be used to draw connected figures. To draw disconnected figures, you would need another instruction e.g. `M` could move the pen forward by a distance one without drawing.

**Example 5: the "call stack"**

We have been discussing stacks of things. One can also have a stacks of tasks. Imagine you are sitting at your desk getting some work done (main task). Someone knocks on your door and you let them in and chat. While chatting, the phone rings and you answer it. You finish the phone conversation and go back to the person in your office. Then maybe there is another interruption which you take care of, return to work, etc. In each case, when you are done with a task, you ask yourself "what was I doing just before I began this task?"

A similar stack of tasks occurs when a computer program runs. The program starts with a `main` method. The main method typically has instructions that cause other methods to be called. The program "jumps" to these methods, executes them and returns to the main method. Sometimes these methods themselves call other methods, and so the program jumps to these other methods, executes them, returns to the calling method, which finishes, and then returns to main.

A natural way to keep track of methods and to return to the 'caller' is to use a stack. Suppose `main` calls method `mA` which calls method `mB`, and then when `mB` returns, `mA` calls `mC`, which eventually returns to `mA`, which eventually returns to `main` which then finishes.

```
  Class    Demo {
     void   mA( ) {
        mB( );
        mC( );
     }
     void  mB( ) { ... }
     void  mC( ) { ... }
     void  main( ){
         mA(  );
     }
  }
```

The stack evolves as follows:

```
                         B             C
                  A      A      A      A      A
         main    main   main   main   main   main  main
---------------------------------------------------------
```

Also see the slides for an example using the `SLinkedList` code from the linked list exercises. I briefly showed how the `TestSLinkedList` calls the `addLast()` method of the LinkedList class, and I show the Eclipse call stack. When you use Eclipse in debugger mode, and you set breakpoints in the middle of methods, there is a panel that shows you the call stack.

## Stack overflow versus underflow

You are probably familiar with the term "stack overflow". What does this mean? Sometimes a stack is limited to a certain size. When that size has been reached and one tries to push another item onto the stack, we say that a stack overflow has occurred.

The term "stack underflow" is the opposite problem. Here we have a stack that is empty and we try to pop from it.

# Queue

Last lecture we looked at an abstract data type (ADT) called a "stack." I introduced the idea of a stack by saying that it was a kind of list, but with a restricted set of operations, `push` and `pop`. Today we will consider another kind of ADT called a "queue". A queue can also be thought of a list. However, a queue is again a restricted type of list since it has a limited set of operations.

You are familiar with queues in daily life. You know that when you have a single resource such as a cashier in the cafeteria, you need to "join the end of the line" and the person at the front of the line is the one that gets served next. There are many examples of queues in a computer system. When you type on your keyboard, the key values enter a queue (or 'buffer'). Normally you don't notice the queue because each keystroke value gets read and removed from the queue before the next one is entered. But sometimes the computer is busy doing something else, and you do notice the queue. There is a pause where nothing you type gets echoed to your screen (e.g. to your text editor), and then suddenly some sequence of characters you typed gets processed very quickly. Other examples of queues are printer jobs, CPU processes, client requests to a web server, etc.

The fundamental property of a queue is that, among those things currently in the queue, the one that is removed next is the one that first entered the queue, i.e. the one that was least recently added. This is different from a stack, where the one that is removed next is the newest one, or the most recently added. We say that queues implement "first come, first served" policy (also called FIFO, first in, first out), whereas stacks implement a LIFO policy, namely last in, first out.

The queue abstract data type (ADT) has two basic operations associated with it: `enqueue(e)` which adds an element to the queue, and `dequeue()` which removes an element from the queue. We could also have operations `isEmpty()` which checks if there is an element in the queue, and `peek()` which returns the first element in the queue (but does not remove it), and `size()` which returns the number of items in the queue. But these are not necessary for a core queue.

**Example**

Suppose we add (and remove) items `a,b,c,d,e,f,g` in the following order, shown on left. On the right is show the corresponding state of the queue *after* the operation.

```
OPERATION                 STATE AFTER OPERATION
                          (initially empty)
enqueue(a)                a
enqueue(b)                ab
dequeue()                 b
enqueue(c)                bc
enqueue(d)                bcd
enqueue(e)                bcde
dequeue()                 cde
enqueue(f)                cdef
enqueue(g)                cdefg
```

## Data structures for implementing a queue

### Singly linked list

One way to implement a queue is with a singly linked list. Just as you join a line at the back, when you add an element to a singly linked list queue, you manipulate the `tail` reference. Similarly, just as you serve the person at the front the queue, when you remove an item from a singly linked list queue, you manipulate the `head` reference. The `enqueue(E)` and `dequeue()` operations are implemented with `addLast(E)` and `removeFirst()` operations, respectively, when a singly linked list is used. Of course, if a singly linked list works, then a doubly linked list would work too, since it is a generalization of a singly linked list.

### Array list

What if we implement a queue using an array list? In this case, `enqueue(element)` would be `addLast(element)` and `dequeue()` would be `removeFirst()`. The `enqueue()` with array list is fine; the only issue that comes up is that we would need to copy to a larger underlying array in the case that the array is full. The main problem is the `dequeue()` method: `removeFirst()` would be inefficient for array lists since we would have to shift all the remaining elements.

### Circular array

An alternative to an array list – but which still uses an array – is treat the array as *circular*. The relationship between indices becomes

$$\texttt{tail} = (\texttt{head} + \texttt{size} - 1) \ \texttt{mod} \ \texttt{length}.$$

In the initial state, we have `size == 0` and `head == 0`. The formula gives `tail` the value `length - 1`. See example below, and suppose that the array has `length = 4`.

| | 0123 | head | tail | size |
|---|---|---|---|---|
| | ---- | 0 | 3 | 0 |
| enqueue(a) | a--- | 0 | 0 | 1 |
| enqueue(b) | ab-- | 0 | 1 | 2 |
| remove() | -b-- | 1 | 1 | 1 |
| enqueue(c) | -bc- | 1 | 2 | 2 |
| enqueue(d) | -bcd | 1 | 3 | 3 |
| enqueue(e) | ebcd | 1 | 0 | 4 |
| dequeue() | e-cd | 2 | 0 | 3 |
| enqueue(f) | efcd | 2 | 1 | 4 |

The next instruction is `enqueue(g)`. If we were simply to make a new larger array and copy the four elements to the front of this new array,

```
efcd----
```

then this would not work. We wouldn't be able to add the `g` since it is supposed to go after `f`.

Instead, we copy the elements to the larger array in such a way that the head is at position 0 in the array.

```
cdef----
```

that is, `head == 0, tail == 3`.

The algorithm for enqueueing would go like this:

```
enqueue( element ){     //  using a circular array
    if ( size == length )  {   //  increase length of array
       create a bigger array tmp[ ]    //  e.g. 2*length
       for i = 0  to  length  - 1
           tmp[i] = queue[ (head + i)  mod  length ]
       head = 0
       tail  =  length - 1
       queue = tmp
    }
    tail = (tail + 1)  mod length
    queue[ tail ] = element
    size = size + 1
}
```

Note that the `length` variable appears a few times in the above pseudocode, and that its value changes. Within the `for` loop of the `if` block, its value is the original length. But if the queue array is enlarged, then the `length` value is doubled and so its value would be different when it is used again outside the `if` block.

We see that we are copying the head to position 0 in the new array. That is, when `i==0`, we are copying `queue[head]` to `tmp[0]`. As I noted in the slides, this is not the only way to copy the elements to the bigger array. One could copy the head to the same position in the new array, and then copy all remaining elements after it, wrapping around if necessary.

Finally, note that we don't need to keep track of a `tail` variable here. Recalling the formula from earlier, `tail = (head + size - 1) mod length`. If we know the three variables on the right side, then we know `tail`. But the pseudocode is a bit easier to read if we represent tail explicitly.

Next consider dequeueing, which is simpler.

```
dequeue(){         //  check that size > 0  (omitted)
   element = queue[head]
   head = (head + 1) mod length
   size = size - 1
   return element
}
```

Note that it "advances" the `head` but does not change `tail`.

One subtlety here is that if there is just one element in the queue, then `head == tail` before we remove that element. So if remove this single element, then we advance the `head` index which will leave `head` with a *bigger* value than `tail` (unless `head` and `tail` were `length-1` in which case `head` becomes 0).

At the end of the lecture, I went over one of the exercises in the stack and queue exercises PDF.

The next core topic in the course is *recursion.* We will look at a number of recursive algorithms and we will analyze how long they take to run. Recursion can be a bit confusing when one first learns about it. One way to understand recursion is to relate it to a proof technique in mathematics called *mathematical induction*, which is what I'll cover today.

Before I introduce induction, let me give an example of a statement that you have seen before, along with a proof. The statement is:

$$\text{For all } n \geq 1, \qquad 1+2+3+\cdots+(n-1)+n = \frac{n(n+1)}{2}.$$

The following proof is slightly different from the one I gave in the slides. Here add up two copies of the left hand side, one forward and one backward:

$$1+2+3+\cdots+(n-1)+n$$

$$n+(n-1)+....+3+2+1.$$

Then pair up terms and sum:

$$1+n+2+(n-1)+3+(n-2)+...(n-1)+2+n+1$$

Note that each pair sums to $n+1$ and there are $n$ pairs, which gives $(n+1)*n$. We then divide by 2 because we added up two copies. This proves the statement above.

## Mathematical induction

The above proof requires a trick, and many proofs in mathematics are like that – they use a specific trick that seems to work only in a few cases. Mathematical induction is different in that it is a general type of proof technique. To understand a proof by mathematic induction, you need to understand the logic of the proof technique *in general.*

Mathematical induction allows one to prove statements about positive integers. We have some proposition $P(n)$ which is either true or false and the truth value may depend on $n$. We want to prove the statement: "for all $n \geq n_0$, $P(n)$ is true", where $n_0$ is some constant that we state explicitly. In the above example, this constant is 1, but sometimes we have some other constant that is greater than 1.

A proof by *mathematical induction* has two parts, and one needs to prove both parts.

1. a *base case*: the statement $P(n)$ is true for $n = n_0$.
2. *induction step*: for any $k \geq n_0$, *if* $P(k)$ is true, *then* $P(k+1)$ must also be true.

   When we talk about $P(k)$ in step 2, we refer to it as the "induction hypothesis". Note that $P(k)$ is just $P(n)$ with $n = k$. We are using parameter $k$ instead of $n$ to emphasize that we're in the context of proving the induction step.

The logic of a proof by mathematical induction goes like this. Let's say we can prove both the base case and induction step. Then, $P(n)$ is true for the base case $n = n_0$, and the induction step implies that $P(n)$ is true for $n = n_0 + 1$, and applying the induction step again implies that the statement is true for $n = n_0 + 2$, and so on forever for all $n \geq n_0$.

**Example 1**

**Statement**: for all $n \geq 1$,

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

**Proof:** The base case, $n_0 = 1$, is true, since

$$1 = \frac{1 \cdot (1+1)}{2}.$$

We next prove the induction step. For any $k \geq 1$, we *hypothesize* that $P(k)$ is true and then show it follows from the hypothesis that $P(k+1)$ would be true (if $P(k)$ were true).

$$\begin{aligned}
\sum_{i=1}^{k+1} i &= (\sum_{i=1}^{k} i) + (k+1) \\
&= \frac{k(k+1)}{2} + (k+1), \text{ by induction hypothesis that P(k) is true} \\
&= (k+1)(\frac{k}{2} + 1) \\
&= \frac{1}{2}(k+1)(k+2)
\end{aligned}$$

and so $P(k+1)$ is also true. This proves the induction step. The proof is complete, since we have proven the base case and the induction step.

**Example 2**

**Statement:** for all $n \geq 3$,

$$2n + 1 < 2^n.$$

**Proof:** The base case $n_0 = 3$ is easy to prove, i.e. $7 < 8$. (Note that setting the base case to be $n_0 = 2$ would not work, nor would $n_0 = 1$, since if we plug in then we get an expression $P(n)$ that is false.) That's why we chose $n_0 = 3$.)

To prove the induction step, let $k$ be any integer such that $k \geq 3$. We hypothesize that $P(k)$ is true and show that it would follow that $P(k+1)$ is also true. Note that $P(k)$ is the inequality

$$2k + 1 < 2^k$$

and $P(k+1)$ is the inequality

$$2(k+1) + 1 < 2^{k+1}.$$

To prove $P(k+1)$ we work with the expression on the left side of the latter inequality.

$$\begin{aligned}
2(k+1) + 1 &= 2k + 3 \\
&= (2k+1) + 2 \\
&< 2^k + 2, \quad \text{by induction hypothesis that } P(k) \text{ is true} \\
&< 2^k + 2^k, \quad \text{since } 2 < 2^k, \text{when } k \geq 3 \\
&= 2^{k+1} \ .
\end{aligned}$$

Thus, if $P(k)$ is true ($k \geq 3$), then $P(k+1)$ would be true.

[ASIDE: You might be asking yourself, how did I know to use the inequality $2 < 2^k$ ? The answer is that I knew what inequality I eventually wanted to have, namely $P(k+1)$. Experience told me how to get there.]

**Example 3**

**Statement:** For all $n \geq 5$, $n^2 < 2^n$.

**Proof:** The base case $n_0 = 5$ is easy to prove, i.e. $25 < 32$.
Next we prove the induction step. The induction hypothesis $P(k)$ is that inequality $k^2 < 2^k$ holds, where $k \geq 5$. We show that if $P(k)$ is true, then $P(k+1)$ must also be true, namely $(k+1)^2 < 2^{k+1}$. We start with the left side of $P(k+1)$.

$$\begin{aligned} (k+1)^2 &= k^2 + 2k + 1 \\ &< 2^k + 2k + 1, \quad \text{by induction hypothesis, for } \; k \geq 5 \\ &< 2^k + 2^k, \quad \text{from Example 2} \\ &= 2^{k+1} \end{aligned}$$

which proves the induction step.

Note that the base case choice is crucial here. The statement $P(n)$ is not true for $n = 0, 1, 2, 3, 4$. Also, note that the induction step happens to be valid for a larger range of $k$, namely $k \geq 3$ rather than $k \geq 5$. But we only needed it for $k \geq 5$.

**Example 4: upper bound on Fibonacci numbers**

Consider the Fibonnacci[12] sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ... where

$$F(0) = 0, \qquad F(1) = 1,$$

and, for all $n > 2$, we define $F(n)$ by

$$F(n) \equiv F(n-1) + F(n-2).$$

**Statement:**

$$\text{for all } n > 0, \quad F(n) < 2^n.$$

**Proof:**

The statement $P(k)$ is "$F(k) < 2^k$". We take the base case(s) to be two values $n_0 = 0, 1$. By definition, $F(0) = 0, F(1) = 1$. Since $F(0) = 0 < 2^0$ and $F(1) = 1 < 2^1$, $P(n)$ is true for both base cases.

The induction hypothesis $P(k)$ is that $F(k) < 2^k$ where $k \geq 2$. For the induction step, we hypothesize that $P(k)$ and $P(k-1)$ are true for some $k$ and $k-1$ and we show this would imply

[12] `http://en.wikipedia.org/wiki/Fibonacci_number`

$P(k+1)$ must also be true, that is, $F(k+1) < 2^{k+1}$. Again, we start with the left side of this inequality:

$$\begin{aligned} F(k+1) &\equiv F(k) + F(k-1) \\ &< 2^k + 2^{k-1} \quad \text{by induction hypothesis} \\ &< 2^k + 2^k \\ &= 2^{k+1} \end{aligned}$$

and so $P(k)$ is true indeed implies that $P(k+1)$ is true, and so we have proven the induction step. So we are done.

Next lecture, we will look at the technique of recursion and show how it is related to the idea of mathematical induction.

# Recursion

Recursion is a technique for solving problems in which the solution to the problem of size $n$ is based on solutions to versions of the problem of size smaller than $n$. Many problems can be solved either by a recursive technique or non-recursive (e.g. iterative) technique, and often these techniques are closely related. In the next few lectures, we'll look at several examples. We'll also see how recursion is closely related to mathematical induction, which I covered in the previous lecture.

## Example 1: Factorial

The factorial function is defined as follows:

$$n! = 1 \cdot 2 \cdot 3 \ldots (n-1) \cdot n$$

where 0! is defined to be 1 by convention. Unlike the sum of numbers to $n$ which we discussed last class, there is no formula that gives us the answer of taking the product of numbers from 1 to $n$, and we need to compute it. Here is an algorithm (written in Java) for computing it. Ignore the fact that `int` only can represent a finite range of values since that's not the point here.

```
  int factorial(int n){  //  assume  n >= 1
     int result = 1;
     for (int i = 1;  i <= n;  i++)
          result *= i;
     return result;
  }
```

Here is another way to define and compute $n!$ which is more subtle, namely if $n > 1$, then

$$n! = n \cdot (n-1)!$$

Here is the corresponding algorithm coded in Java which is *recursive.* Note that the method `factorial` calls itself.

```
static int factorial(int n){   //   algorithm assumes argument:  n >= 1
   if  (n == 0)
     return 1;      //  base condition
   else
     return  n * factorial( n - 1);
}
```

Recursive algorithms can't keep called themselves *ad infinitum.* Rather, they need to have a condition which says when to stop. This is called a *base condition.* For the `factorial` function, the base condition is that the argument is 0. Anytime you write a recursive algorithm, make sure you have a base condition and make sure you reach it. Typically this is ensured by having the parameter of the recursive call be smaller, *e.g.* `n-1` rather than `n` in the case of `factorial`.

Let's use mathematical induction to convince ourselves that the factorial algorithm is correct.

**Claim:** The recursive `factorial` algorithm indeed computes $n!$ for any input value $n \geq 0$.

**Proof:** First, the base case: If the parameter `n` is 1, then the algorithm returns 1. (Easy to verify.)

Second, the induction step: The induction hypothesis is that `factorial(k)` indeed returns $k!$. We want to show it follows that `factorial(k+1)` returns $(k+1)$ !. But this is easy to see by inspection, since the induction hypothesis implies that the algorithm returns $(k+1) * k!$, which is just $(k+1)!$.

## Example 2: Fibonacci numbers

Consider the Fibonnacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$F(n) = F(n-1) + F(n-2),$$

where $F(0) = 0, F(1) = 1$. The function $F(n)$ is defined in terms of itself, and so it is recursive.

Here is an iterative algorithm for computing the n-th Fibonacci number. We start at $n = 0, 1$ and move forward (assuming $n > 0$).

```
fibonacci(n){
   if ((n == 0) | (n == 1))
      return n
   else{
      f0 = 0
      f1 = 1
      for i = 2 to n{
         f2 = f1 + f0
         f0 = f1            //  set  F(n) for next round
         f1 = f2            //  set  F(n+1) for next round
      }
      return f2
   }
}
```

The method requires $n$ passes through the "for loop". Each pass takes a small (fixed) number of operations. So we would expect the number of steps to be about $cn$ for some constant $c$.

A *recursive algorithm* for computing the $n$th Fibonacci number is simpler to express:

```
fibonacci(n){    //   assume n > 0
   if ((n == 0) || (n == 1))
      return n
   else
      return fibonacci(n-1) + fibonacci(n-2)
}
```

Here is a proof that this recursive algorithm for computing the nth Fibonaccie number is correct. First, the base case: If the parameter `n` is 0 or 1, then the algorithm returns 0 or 1, respectively, which is correct.

Second, the induction step: the induction hypothesis is that `fibonacci(k)` and `fibonacci(k-1)` returns the $k$th and $(k-1)$-th Fibonacci number, for any $k \geq 1$. We want to show that this implies `fibonacci(k+1)` returns the $(k+1)$th Fibonacci number. But again this is easy to see by inspection, since the algorithm returns the sum of `k-1`th and `k`th Fibonacci numbers which is the `k+1`th Fibonacci number.

The recursive version turns out to be very slow, however, since it ends up calling `fibonacci` on the same parameter $n$ *many* times, which is unnecessary. For example, suppose you are asked to compute the 247-th Fibonacci number. `fibonacci(247)` calls `fibonacci(246)` and `fibonacci(245)`, and `fibonacci(246)` calls `fibonacci(245)` and `fibonacci(244)`. But now notice that `fibonacci(245)` is called twice. The problem here is that every time you want to compute `fibonacci(k)` where `k > 1`, you need to do *two* recursive calls. This leads to a combinatorial explosion in the number of calls. (I haven't provided a formal calculation here in exactly how many calls would be made, but hopefully you get the idea that many repetitions of the same calls occur. If not, see the picture in the slides.) In COMP 251, you'll see many problems of this nature and you'll learn a nice and simple technique for dealing with it - called *dynamic programming*. The idea for the fibonacci case is easy to explain. Just keep track of which fibonacci numbers you have already computed and only call recursively `fibonacci(k)` when you haven't computed it yet.

## Example 3: reversing a list

Let's next revisit a few algorithms for lists, and examine recursive versions. The first example is to reverse a list (see linked list exercises for an iterative version). The idea can be conveyed with the following picture. To reverse the list,

```
(a b c d e f g)
```

we can remove the first element `a` and reverse the remaining elements,

```
a  (b c d e f g)  ---->  a  (g f e d c b)
```

and then add the removed element at the end of the (reversed) list.

```
(g f e d c b a)
```

Here is the pseudocode. I have written it so that `list` is an argument to the various methods that are called, which is different notation from what we used in the list lectures.

```
reverse(list){     //   assume n > 0
   if list.size == 1     // base case
      return  list
   else{
      firstElement = removeFirst(list)
      list = reverse(list)        // list has only n-1 elements
      return addLast(list, firstElement)
   }
}
```

And here is some Java code for a class that implements the List methods:

```
static void reverse(List list) {
     if(list.size()==1) {
         return;
     }
     firstElement = list.remove(0);
     reverse(list);  // this list has n-1 elements
     list.add(firstElement);   // appends at the end of the list
}
```

## Example 4: sorting a list

Recall the selection sort algorithm. The basic idea was to maintain two lists: a sorted list, and a 'rest' list which is unsorted. The algorithm loops repeatedly through the rest list, removes the minimum element each time, and adds it to the end of the sorted list. (Since the sorted list consists of elements that are all smaller than or equal to elements in the rest list, all the elements of the rest list will eventually be added after elements in the sorted list.)

The recursive algorithm below is a similar idea. Remove the minimum element, sort the rest list (recursively), and add the minimum element to the front of the sorted rest list (that is, the sorted rest list will be after any minimum elements that have been removed). That may sound complicated, but look how simple the pseudocode is:

```
  sort(list){  //  assumes list.size >= 1
     if list.size == 1
        return list         // base case
     else{
        minElement = removeMin(list)
        list = sort(list)
        return addFirst(list, minElement)
     }
  }
```

## Example 5: Tower of Hanoi

Let's now turn to an example of a problem in which a recursive solution is very easy to express, and a non-recursive solution is very difficult to express (and I won't even both with the latter). The problem is called *Tower of Hanoi.* There are three stacks (towers) and a number $n$ of disks of different radii. (See `http://en.wikipedia.org/wiki/Tower_of_Hanoi`). We start with the disks all on one stack, say stack 1, such that the size of disks on each stack increases from top to bottom. The objective is to move the disks from the starting stack (1) to one of the other two stacks, say 2, while obeying the following rules:

1. A larger disk cannot be on top of a smaller disk.
2. Each move consists of popping a disk from one stack and pushing it onto another stack, or more intuitively, taking the disk at the top of one stack and putting it on another stack.

The recursive algorithm for solving the problem goes as follows. The three stacks are labelled $s1, s2, s3$. One of the stacks is where the disks "start". Another stack is where the disks should all be at the "finish". The third stack is the only remaining one.
[ASIDE: In the lecture and slides, I did not use the term "stack" to define the towers and moves, since I didn't want to distract you by thinking about the stack ADT. For this problem, I think it is conceptually simpler just to think of "moving" a disk and obeying rules 1 and 2, rather than to re-express the problem and rules in terms of stacks and push and pop operations.]

```
tower(n, start, finish, other)     // only call with n¿0
  if n==1 then
    move from start to finish          // i.e.  finish.push(  start.pop() )
  else
    tower(n-1, start, other, finish)
    move from start to finish          // i.e.  finish.push(  start.pop() )
    tower(n-1, other, finish, start)
  end if
```

Here I will label the stacks `A, B, C`. For example, `tower(1,A,B,C)` would result in:

```
move from A to B
```

What about `tower(2,A,B,C)` ? This would produce the following sequence of instructions:

```
tower(1,A,C,B)
move from A to B
tower(1,C,B,A)
```

and the two calls `tower(1,*,*,*)` would each move one disk, similar to the previous example (but with different parameters). So, in total there would be 3 moves:

```
move from A to C
move from A to B
move from C to B
```

Here are the states of the tower for `tower(3,A,B,C)` and the corresponding print instructions. Notice that we need to do the following:

```
tower(2,A,C,B)
move from A to B
tower(2,C,B,A)
```

The initial state is:

```
    *
    **
    ***
    ---       ---      ---    (initial)
```

So first we do `tower(2,A,C,B)`, which takes 3 moves:

```
      **
      ***      *
      ---      ---       ---      (after moving disk from A to B)



      ***      *         **       (after moving disk from A to C)
      ---      ---       ---


                         *
      ***                **       (after moving from B to C)
      ---      ---       ---

  Next we do "move from A to B":

                         *
               ***       **
      ---      ---       ---      (after moving from A to B)

Then we call  tower(2, C, B, A) which does the following 3 moves:

      *        ***       **
      ---      ---       ---      (after moving from C to A)


               **
      *        ***
      ---      ---       ---      (after moving from C to B)


                *
                **
                ***
      ---       ---      ---      (after moving from A to B)

   and we are done!
```

**Claim: For any $n \geq 1$ , towers of Hanoi algorithm is correct for $n$ disks**

For the algorithm to be "correct", we need to ensure that a larger disk is never place on top of a smaller disk, and that we move one disk at a time, and that the $n$ disks are eventually moved from the `start` to `finish`. The proof is by mathematical induction.

Base case: The rule is obviously obeyed if $n = 1$ and the algorithm simply moves the one disk from `start` to `finish`.
Induction step: Suppose the algorithm is correct if there are $n = k$ disks on some initial tower and the other towers contain only larger disks. This is the induction hypothesis. We need to show that the algorithm is therefore correct if there are $n = k + 1$ disks on some initial tower and the other towers contain only larger disks. For $n = k + 1$, the algorithm has three steps, namely,

- `tower(k,start,other,finish))`
- `move from start to finish`
- `tower(k,other,finish,start)`

The first recursive call to `tower` moves $k$ disks from *start* to *other*, while obeying the rules for these $k$ disks. (This is the induction hypothesis). The second step moves the biggest disk $(k + 1)$ from `start` to `finish`. This also obeys the rule, since `finish` does not contain any of the $k$ smaller disks (because these smaller disks were all moved to the `other` tower). Finally, the second recursive call to `tower` move $k$ disks from `other` to `finish`, while obeying the rules (again, by the induction hypothesis). This completes the proof.

How many moves does `tower(n, ...)` take? `tower(1, ...)` takes 1 move. `tower(2, ...)` takes 3 moves, namely two recursive calls to `tower(1, ...)` which take 1 move each, plus one move. `tower(3, ...)` makes two recursive calls to `tower(2, ...)` which we just said takes 3 moves each, plus one move, for a total of 3*2 + 1 = 7. Similarly, `tower(4, ...)` makes two recursive calls to `tower(3, ...)` which we just said takes 7 moves each, plus one move, for a total of 7*2 + 1 = 15. And so on... `tower(5, ...)` takes 2*15 + 1 = 31 moves, and `tower(6, ...)` takes 2*31 + 1 = 63 moves. In general, `tower(n, ...)` makes two recursive calls to `tower(n-1, ...)` plus one move. So one can prove by induction that `tower(n, ...)` takes $2^n - 1$ moves.

## Recursion and the Call Stack

In the stack lecture (16), I mentioned the ”call stack”. Each time a Java method calls another method (or a method calls itself, in the case of recursion), the computer needs to do some administration to keep track of the “state” of method at the time of the call. This information is called a “stack frame”. You will learn more about this in COMP 273, but it is worth mentioning now to take some of the mystery out of how recursion is implemented in the computer.

As an example, suppose the program calls `factorial(6)`. This leads to a sequence of recursive calls and subsequent returns from these calls. For example, right before *returning* from the `factorial(3)` call, we have made the following sequence of calls and returns:

```
factorial(6), factorial(5), factorial(4), factorial(3), factorial(2),
factorial(1), return from factorial(1), return from factorial(2)
```

the call stack looks like this,

```
frame for factorial(3):  [factn = 6, n=3] <---- top of stack
frame for factorial(4):  [factn = 0, n=4]
frame for factorial(5):  [factn = 0, n=5]
frame for factorial(6):  [factn = 0, n=6] <---- bottom of stack
```

Using the Eclipse debugger and setting breakpoint within the `factorial()` method, you can see how the stack evolves. I strongly recommend that you do this and verify how this works.

`http://www.cim.mcgill.ca/~langer/250/TestFactorial.java`

I would do the same for the Tower of Hanoi.

`http://www.cim.mcgill.ca/~langer/250/TestTowerOfHanoi.java`

See slides for screen shots.

I should emphasize that the call stack is not some abstract idea, but rather it is a real data structure used by the program that runs your Java program (called the "Java Virtual Machine"). The stack consists of *stack frames*, one for each method that is called. In the case of recursion, there is one stack frame for each time the method is called.

The stack frame contains all the information that is needed for that method. This includes local variables declared and used by that method, parameters that are passed to the method, and information about where the method returns when it is done, that is, who called the method.

You will learn about the call stack and stack frames work in much more detail in COMP 273. I mention it here because I want you to get familiar with the idea, and because I want you to be aware that the call stack and stack frame really exist. Indeed most decent IDEs will allow you to examine the call stack (see slides) and at least the current stack frame, i.e. the frame on top of the call stack. See the examples in the lecture slides.

In the next few lectures, we'll give a few more fundamental examples of recursion. The first is binary search, but before we go there let's revisit a problem we discussed at the beginning of the course.

## Converting a number to its binary representation

Recall how to convert a decimal number $n \geq 1$ to binary. Here I'll write the algorithm such that we print out the bits from low to high.

```
toBinary(n){  //  iterative
    i = 0
    while n > 0 {
      print  n % 2     //  bit i
      n = n/2
      i = i+1
    }
}
```

Next we write the algorithm recursively.

```
   toBinary(n){            //  algorithm assumes input n >= 1
        if  n >= 1{               //  otherwise base case, and do nothing
           print  n % 2
           toBinary( n/2 )
        }
   }
```

Note that this prints the bits from the lowest order to highest order, i.e. same as the iterative algorithm. If we were to swap the `print` and the `toBinary` calls, then we would print from highest order to lowest order. If you don't understand why, then see the "countdown" and "countup" examples in the exercises.

Finally, recall from lecture 2 that the iterative version of the algorithm loops about $\log_2 n$ times, where $n$ is the original number. For the same reason, the recursive version has about $\log_2 n$ recursive calls, one for each bit of the binary representation of the number. We next turn to another important example of an algorithm that has this $O(\log_2 n)$ time complexity.

## Binary search in a sorted (array) list

Suppose we have an array list of $n$ elements which are *already* sorted from smallest to largest. These could be numbers or strings sorted alphabetically. Consider the problem of searching for a particular element in the list. Say we return the index in $0, \ldots, n-1$ of that element if it is present in the list. If the element is not present in the list, then we return -1.

One way to do this would be to scan the values in the array, using say a `while` loop. In the worst case that the value that we are searching for is the last one in the array, we would need to scan the entire array to find it. This would take $n$ steps. Such a *linear search* is wasteful since it doesn't take advantage of the fact that the array is already sorted.

A much faster method, called *binary search*, takes advantage of the fact that the array is sorted. You are familiar with this idea. Think of when you look up a word in an index in the back of a book. Since the index is sorted alphabetically, you don't start from the beginning and scan. Instead, you jump to somewhere in the middle.[13] If the word you are looking for comes before those on the page, then you jump to some index roughly in the middle of those elements that come before the one you jumped to, and otherwise you jump to a position in the middle of those that come after the one you jumped to. The *binary search* algorithm does essentially what I just described. Let's first present an iterative (non-recursive) version of binary search algorithm.

```
binarySearch(list, value){          // iterative
    low = 0
    high = list.size - 1
    while  low <= high  {
       mid = (low + high)/ 2    //  so mid == low, if high - low == 1
       if list[mid] == value
          return mid
       else{ if value < list[mid]
               high = mid - 1       //  high can become less than low
            else
               low  = mid + 1  }
    }
    return  -1     // value not found
}
```

For each pass through the `while` loop, the number of elements in the array that still need to be examined is cut by at least half. Specifically, if `[low, high]` has an odd number of elements (2k+1), then the new `[low, high]` will have k elements, or less than half. If `[low, high]` has an even number of elements (2k), then the new `[low, high]` has either k or k-1 elements, whih is at most half. Note that the new `[low, high]` does not contain the `mid` element.

It follows that for an input array with $n$ elements, there are about $\log_2 n$ passes through the loop. This is the same idea as converting a number $n$ to binary, which takes about $\log_2 n$ steps to do, i.e. the number of times we can divide $n$ by 2 until we get 0.

The details of the above algorithm are a bit tricky and it is common to make errors when coding it up. There are two inequality tests: one is $\leq$ and one is $<$, and the way the `low` and `high` variables are updated is also rather subtle. For example, note that if the item is not found then the while loop exits when `high < low`. There is no explicit check for the case that `high == low`.

[13]Of course, if you are looking for a word that starts with "b", then you don't just into the middle, but rather you start near the beginning. But let's ignore that little detail here.

Here is a recursive version. Notice how the iterative and recursive versions of the algorithm are nearly identical. One difference is that the recursive version passes in the `low` and `high` variables.

```
binarySearch( list, value, low, high ){   // recursive
    if  low > high {
       return -1
    else{
       mid = (low + high) / 2
       if value == list[mid]
          return mid
       else if value < list[mid]
          return binarySearch(list, value, low, mid - 1 )
       else
          return binarySearch(list, value,  mid+1, high)
    }
}
```

**The lecture slides contain more material, including a discussion of time complexity and an introduction to mergesort. See the lecture 22 notes for the full mergesort notes.**

**In the lecture slides, I introduced mergesort in lecture 21. Here I will put all the mergesort material into these lecture (22) notes.**

## Mergesort

In lecture 11, we saw three algorithms for sorting a list of $n$ items. We saw that, in the worst case, all of these algorithm required $O(n^2)$ operations. Such algorithms will be unacceptably slow if $n$ is large.

To make this claim more concrete, consider that if $n = 2^{20} \approx 10^6$ i.e one million, then $n^2 \approx 10^{12}$. How long would it take a program to run that many instructions? Typical processors run at about $10^9$ basic operations per second (i.e. GHz). So a problem that takes in the order of $10^{12}$ operations would require thousands of seconds of processing time. ( Having a multicore machine with say 4 processors only can speed things up by a factor of 4 which doesn't change the argument here. )

We next consider an alternative sorting algorithm that is much faster that these earlier $O(n^2)$ algorithms. This algorithm is called *mergesort.* Here is the idea. If the list has just one element ($n = 1$), then do nothing. Otherwise, partition the list of $n$ elements into two lists of size about $n/2$ elements each, sort the two individual lists (recursively, using mergesort), and then merge the two sorted lists.

For example, suppose we have a list

$$< 8, 10, 3, 11, 6, 1, 9, 7, 13, 2, 5, 4, 12 > .$$

We partition it into two lists

$$< 8, 10, 3, 11, 6, 1 > \quad < 9, 7, 13, 2, 5, 4, 12 > .$$

and sort these (by applying mergesort recursively):

$$< 1, 3, 6, 8, 10, 11 > \quad < 2, 4, 5, 7, 9, 12, 13 > .$$

Then, we merge these two lists to get

$$< 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 > .$$

Here is pseudocode for the algorithm. Note that it uses a helper method `merge` which does most of the work.

```
mergesort(list){
   if  list.length == 1
      return list
   else{
      mid = (list.size - 1) / 2
      list1 =  list.getElements(0,mid)
      list2 =  list.getElements(mid+1, list.size-1)
      list1 = mergesort(list1)
      list2 = mergesort(list2)
      return  merge( list1, list2 )
   }
}
```

Below is the merge algorithm. Note that it has two phases. The first phase initializes a new list (empty), steps through the two lists, (`list1`) and (`list2`), compares the front element of each list and removes the smaller of the two, and this removed element to to the back of the merged list. *See the detailed example in the slides of lecture 21 for an illustration.*

The second phase of the algorithm starts after one of `list1` or `list2` becomes empty. In this case, the remaining elements from the non-empty list are moved to `list`. This second phase uses two `while` loops in the above pseudocode, and note that only one of these two loops will be used since we only reach phase two when one of `list1` or `list2` is already empty.

```
merge( list1,  list2){
   new empty list
   while  (list1 is not empty) & (list2 is not empty){
       if  (list1.first < list2.first)
          list.addlast( list1.removeFirst() )
       else
          list.addlast( list2.removeFirst() )
   }
   while list1 is not empty
       list.addlast( list1.removeFirst() )
   while list2 is not empty
       list.addlast( list2.removeFirst() )
   return  list
}
```

I have written the `mergesort` and `merge` algorithms using abstract list operations only, rather than specifying how exactly it is implemented (array list versus linked list). Staying at an abstract level has the advantage of getting us quickly to the main ideas of the algorithm: what is being computed and in which sequence? Heads up though: there are disadvantages of hiding the implementation details, i.e. the data structures. Sometimes the choice of data structure can be important for performance (as we saw when we compared arraylists and linked lists).

## mergesort is $O(n \log n)$

There are $\log n$ levels of the recursion, namely the number of levels is the number of times that you can divide the list size $n$ by 2 until you reach 1 element per list. The number of instructions that must be executed at each level of the recursion is proportional to the number $n$ of items in the list. This is most easily visualized using the example given in the slides, when I went through how all of the partitions and merges worked. There were $2 \log_2 n$ columns, half corresponding to the partitioning in to smaller lists and half corresponding to merges. The total number of elements in each columns is $n$. Making the columns in the merge steps require time roughly proportional to $n \log_2 n$ since there are $\log_2 n$ columns to be built. Admittedly, the above argument is hand waving! I will discuss the time complexity of mergesort more formally at the end of the course using a technique called *recurrences*.

To appreciate the difference between the number of operations for the earlier $O(n^2)$ sorting algorithms versus $O(n \log n)$ for mergesort, consider the following table.

| $n$ | $\log n$ | $n \log n$ | $n^2$ |
|---|---|---|---|
| $10^3 \approx 2^{10}$ | 10 | $10^4$ | $10^6$ |
| $10^6 \approx 2^{20}$ | 20 | $20 \times 10^6$ | $10^{12}$ |
| $10^9 \approx 2^{30}$ | 30 | $30 \times 10^9$ | $10^{18}$ |
| ... | ... | ... | ... |

When $n$ becomes large, the time it takes to run mergesort is significantly less than the time it takes to run bubble/selection/insertion sort. Roughly speaking, on a computer that runs $10^9$ operations per second running mergesort on a list of size $n = 10^9$ would take in the order of minutes, whereas running insertion sort would take centuries.

## Quicksort

Another well-known recursive algorithm for sorting a list is *quicksort*. This algorithm is similar to mergesort, in the sense that it partitions the list. But there are important differences.

At each call of quicksort, one sorts a list as follows. An element known as pivot chosen. For example, the pivot might be the first element or the last element, of some element chosen by a preliminary examination of the list. Let `pivot` be the value of the pivot. The remaining elements in the list are then partitioned into two lists: `list1` which contains those smaller than the pivot, and `list2` which contains those elements that are greater than or equal to the pivot. The two lists `list1` and `list2` are recursively sorted. Then the two sorted lists and the pivot are concatenated into a sorted list which contains all the original elements (specifically, `list1` followed by `pivot` followed by `list2`). Here is high level idea of how it works.

```
quicksort(list){
   if  list.length <= 1
      return list
   else{
      pivot = list.removeFirst()   // or some other element
      list1 = list.getElementsLessThan(pivot)
      list2 = list.getElementsNotLessThan(pivot) // i.e. the rest
      list1 = quicksort(list1)
      list2 = quicksort(list2)
      return  concatenate( list1, pivot, list2 )
   }
}
```

Unlike mergesort, most of the work in quicksort is done *prior* to the recursive calls. Given a pivot element, the algorithm goes through the rest of the list and compares each element to `e`. This takes time proportional to the size of `list` in that recursive call, since one needs to examine each of the elements in the list and decide whether to put it into either `list1` or `list2`. The concatenation that is done after sorting `list1` or `list2` might also involve some work, depending on the data structure used.

## Quicksort in-place implementation

One common implementation of quicksort uses a single array. No extra space required for copying the elements (unlike in mergesort, where it is very natural to use an extra array for the `merge` step). Sorting without using any extra space is called *in place* sorting.[14]

Here is the quicksort algorithm. The `partition` method is given below.

```
quicksort(list, low, high ){       //  \void"
   if  low < high {
       wall  =  partition(list, low, high)
       quicksort(list, low, wall - 1)
       quicksort(list, wall + 1,  high)
   }
}
```

Most of the work is done by the `partition` algorithm which works with an interval `[low,high]` within the list. See the slides for a detailed example of how `partition` works.

```
partition(list, low , high )
  pivot =  list[high]
  wall = low - 1
  for (i = low ; i <= high; i++)
     if ( list[i]  <=  pivot ){
        wall++
        if (wall != i)
            list.swap(wall, i)
     }
  return wall
}
```

The `partition` method begins by choosing the last element of the `[low,high]` interval, namely `list[high]`, as the `pivot`. The `partition` method contains a `for` loop that iterates a variable `i` through the list from position `low` to position `high`. It also updates the `wall` position. The idea is that whenever you find a `list[i]` that is less than or equal to the pivot, you increment `wall` in order to make room for the element `list[i]` you have just found. At the end of the for loop, the element will be the pivot itself.

When the `partition` algorithm terminates, all elements that have value less than `pivot` are positioned to the left of the pivot and all elements with value greater than `pivot` are positioned to the right of the pivot.

See the slides for an example. You should practice this algorithm to make sure you indeed understand it. Don't just practice the `partition` step; also practice the recursion.

We will discuss the time complexity of quicksort later in the course. I will try to add some questions to the exercises for now so you get an intuition of best case and worst case.

[14]Bubble sort, selection sort, and insertion sort were also "in place" sorting algorithms.

## (Rooted) Trees

Thus far we have been working mostly with "linear" collections, namely *lists*. For each element in a list, it makes sense to talk about the previous element (if it exists) and the next element (if it exists). It is often useful to organizes a collection of elements in a "non-linear" way. In the next several lectures, we will look at some examples. Today we begin with *(rooted) trees*.

Like a list, a rooted tree is composed of nodes that reference one another. With a tree, each node can have one "parent" and multiple "children". You can think of the parent as the "prev" node and the children as "next" nodes. The key difference here is that a node can have multiple children, whereas in a linked list a node has (at most) one "next" node.

You are familiar with the concept of rooted trees already. Here are a few examples.

- Many organizations have a hierarchical structures that are trees. For example, see the McGill organizational chart. `http://www.mcgill.ca/orgchart/` which is (almost) a tree. Note that the "lowest" level in this chart contains the Deans but of course there are thousands of employees at McGill "below" the Deans, which are not shown. For example, as a McGill professor in the School of Computer Science, I report to my department Chair, who reports to the Dean of Science, who reports to the McGill Provost, who reports to the Principal. A professor in the Department of Electrical and Computer Engineering reports to the Chair of ECE, who reports to the Dean of Engineering, who like the Dean of Science reports to the Provost, who reports to the Principal. (The "B reports to A" relationship in an organization hierarchy defines a "B is a child of A" relation in a tree – see below).

- Family trees. There are two kinds of family trees you might consider. The first defines the parent/child relation literally, so that the tree children of a node correspond to the actual children (sons and daughters) of the person represented by the node. The second tree is less conventional. It defines each person's mother and father as its "children". In such a tree, each person has two children (the person's real parents), and they each of two children (the person's four grandparents), etc.

- A file directory on a MS Windows or UNIX/LINUX operating system. For example, this lecture notes file is stored at

  ```
  C:\Users\Michael\Dropbox\TEACHING\250\LECTURENOTES\22-trees.pdf
  ```

[**ASIDE: The lecture slides used many figures to illustrate the ideas that I will be presenting in today's notes. I am not including the figures in these notes. You should consult those slides when reading these notes!**]

## Terminology for rooted trees

A (rooted) tree consists of a set of *nodes* or *vertices*, and *edges* which are ordered pairs of nodes or vertices. When we write an edge $(v_1, v_2)$, we mean that the edge goes from node $v_1$ to $v_2$. The "node" vs. "vertex" terminology is perhaps a bit confusing at first. When one is discussing data structures, it is more common to use the term "node" and when one is talking about trees as abstract data types one typically uses the term "vertex."

In COMP 250 we will only deal with a special type of tree called a *rooted* tree. A rooted tree has special node called the *root node*. Rooted trees have the following properties:

- For any node $v$ in the tree (except the root node), there is a unique node $p$ such that $(p, v)$ is an edge in the tree. The node $p$ is called the *parent* of $v$, . Naturally, $v$ is called a *child* of $p$. A parent can have multiple children.

  Notice that a tree with $n$ nodes has $n - 1$ edges. Why? Because for each node $v$ except the root node ($n - 1$ of these), there is a unique edge $(p, v)$ and these $n - 1$ edges are exactly the set of edges in the tree.

- *sibling relation*: Two nodes are siblings if they have the same parent.

- *leaf*: A node with no children is called a leaf node. A more complicated way of saying a "node with no children" is "a node $v$ such that there does *not* exist an edge $(v, w)$ where $w$ is a node in the tree". Leaves are also called *external* nodes.

- *internal* node: a node that has a child (i.e. a node that is not a leaf node).

  For example, in the linux or windows file system, files and empty directories are leaf nodes, and non-empty directories are internal nodes. A directory is a file that contains a list of references to its children nodes, which may be files (leaves) or subdirectories (internal nodes).

- *path*: a sequence of nodes $v_1, v_2, \ldots, v_k$ where $v_i$ is the parent of $v_{i+1}$ for all $i$.

- *length of a path*: the number of edges in the path. If a path has $k$ nodes, then it has length $k - 1$.

  You can define a path of length 0, namely a path that consists of just one node $v$. (This is useful sometimes, if you want to make certain mathematical statements bulletproof.)

- *depth* of a node in a tree (also called *level* of the node): the length of the (unique) path from the root to the node. Note that the root node is at depth 0.

- *height* of a node $v$ in a tree: the maximum length of a path from $v$ to a leaf. Note: a leaf has height 0.

- *height* of a tree: the height of the root node

- *ancestor*: $v$ is an ancestor of $w$ if there is a path from $v$ to $w$

- *descendent*: $w$ is a descendent of $v$ if there is a path from $v$ to $w$. Note that "$v$ is an ancestor of $w$" is equivalent to "$w$ is a descendent of $v$".

- A subtree is a subset of nodes and edges in a tree which itself is a tree. In particular, a subtree has its own root which may or may not be the same as the root of the original tree. Every node in a tree defines a subtree, namely the tree defined by this node and all its children. In particular, any tree is a subtree of itself. Every node in a tree also defines a subtree in which that node is the only node!

- *recursive definition of a rooted tree:* A rooted tree $T$ either has no nodes (empty tree), or it consists of a root node $r$ together with a set of zero or more non-root nodes, and these non-root nodes are partitioned into non-empty subtrees $T_1, \dots, T_k$ whose roots are the children of $r$. Note the word 'partition' here, which means that the subtrees are disjoint (they don't share any nodes).

We can also define operations on trees recursively. Here are a few common examples. The first is to compute the depth of a node.

```
depth(v){
  if (v is a root node)    //  that is,  v.parent == null
    return 0
  else
    return 1 + depth(v.parent)
}
```

*Notice that this method requires that we can access the parent of a node.* We have defined edges to be of the form (parent, child). If one implements an edge by putting a child reference in a parent node then, given a node v, we can only reference the child of this node, not the parent, and so we would not be able to compute the depth. Therefore, to use the above method for computing depth, we would need a node to have a reference to its parent. (Whether we need references to children also depends on what we will use the tree for. )

Another example operation is to compute the height of a node. Just like the depth, the height can be computed recursively.

```
height(v){
  if (v is a leaf)    //  that is,  v.child == null for any child
    return 0
  else{
    h = 0
    for each child w of v
       h = max(h, height(w))
    return 1 + h
  }
}
```

## Tree implementation in Java

The main decision one needs to make in implementing trees is how to represent the set of children of a node. If node can have at most two children (as is the case of a binary tree, which we will describe in lecture 24) then each node can be given exactly two reference variables for the children. If a node can have many children, then a more flexible approach is needed.

One common approach is to define the children by an array list or by a linked list.

```
class  TreeNode<T>{
    T   element;
    ArrayList<TreeNode<T>>   children;

    TreeNode<T>   parent;   // optional
     :
     :                     //  methods
}
```

This approach is perhaps overkill though, since one typicallly doesn't need to index the children by their number. Using a `LinkedList` (doubly linked list in Java) is also perhaps overkill if one doesn't need to be able to go both directions in the list of children. A singly linked list is often enough, and that is the approach below.

The *"first child, next sibling"* implementation of a tree uses the following node definition.

```
  class  TreeNode<T>{
      T   element;
      TreeNode<T>   firstChild;
      TreeNode<T>   nextSibling;
        :
        :                     //  methods
  }
```

It defines a singly linked list for the siblings, where the head is the `firstChild` and the next is the `nextSibling`. It is simpler than the arraylist implementation since it uses just two fixed references at each node (or 3, if one also has a parent link)

Finally, to define the rooted tree, we could use:

```
  class  Tree<T>{
     TreeNode<T>   root;
         :
         :                     //  methods
  }
```

The `root` here serves the same role as the `head` field in our implementation of `SLinkedList`. It gives you access to the nodes of the tree. The `TreeNode` class would then be defined as an inner class inside this `Tree` class.

## Exercise: Representing trees using lists

A tree can be represented using lists, as follows:

```
tree             =   root | ( root listOfSubTrees )
listOfSubTrees   =   tree |   tree listOfSubTrees
```

For example, let's draw the tree that corresponds to the following list, where the `root` elements are single digits. (Apologies for the ASCII art below. See the slides for prettier pictures. )

```
            ( 6 ( 2 1 7 ) 3 ( 4 5 ) ( 9 8 0 ) )
```

The first uses a separate edge for each parent/child pair.

```
      6
  /  |  \  \
 2  3   4   9
/ \     |  / \
1  7    5  8  0
```

The second uses the "first child, next sibling" representation.

```
      6
    /
    2 - 3 - 4 - 9
   /       /     \
   1-7     5       8-0
```

## Tree traversal

Often we wish to iterate through or "traverse" the nodes of the tree. We generally use the term *tree traversal* for this. There are two aspects to traversing a tree. One is that we need to follow references from parent to child, or child to its sibling. The second is that we may need to do something at each node. I will use the term "visit" for the latter. Visiting a node means doing some computation at that node. **We will see lots of examples in the slides! These notes cover the algorithms only.**

### Depth first traversal using recursion: pre- and post-order

The first two traversals that we consider are called "depth first". There are two ways to do depth-first-traversal of a tree, depending on whether you visit a node before its descendents or after its descendents. In a *pre-order* traversal, you visit a node, and then visit all its children. In a *post-order* traversal, you visit the children of the node (and their children, recursively) before visiting the node.

```
depthfirst_Preorder(root){
    visit root
    for each child of root
      depthfirst_Preorder(child)
}
```

See the lectures slides for an example of a preorder traversal and a number of the ordering of nodes visited.

A second example is shown below. a file system. The directories and files define a tree: the internal nodes are directories and the leaves are either empty directories or files. A common way of printing out the directories and files is shown below, where the indentation denotes one more level of the tree. We first print out the root directory, then list the subdirectories and files in the root directory. For each subdirectory, we also print its subdirectory and files, and so on. This is all done using a pre-order traversal. The visit would be the print statement. Here is a example of what the output of the print might look like. (This is similar to what you get on Windows when browsing files in the Folders panel.)

```
  My Documents                (directory)
     My Music                 (directory)
        Raffi                 (directory)
           Shake My Sillies Out  (file)
           Baby Beluga           (file)
        Eminem                (directory)
           Lose Yourself         (file)
     My Videos                (directory)
        :                     (file)
     Work                     (directory)
        COMP250               (directory)
            :
```

For a postorder traversal, one visits a node after having visited all the children of the node. Here is the pseudocode.

```
depthfirst_Postorder(root){
   for each child of root
      depthfirst_Postorder(child){
   visit root
}
```

Let's look at an example. Suppose we want to calculate how many bytes are stored in all the files within some directory including all its sub-directories. This is post-order because in order to know this total number of bytes we first need to know the total number of bytes in all the subdirectories. Hence, we need to visit the subdirectories first. Here is an algorithm for computing the number of bytes. It traverses the tree in postorder in the sense that it computes the sum of bytes in each subdirectory by summing the bytes at each child node of that directory.

```
numBytes(root){
  if root is a leaf
     return number of bytes at root   (0 if root is directory)
  else{
     sum = 0    // local variable
     for each child of root{
        sum += numBytes(child)}
     return sum
  }
}
```

## Depth first traversal without recursion

As we have discussed in earlier lectures (on stacks), recursive algorithms are implemented using a call stack which keep track of information needed in each call. You can sometimes avoid recursion by using an explicit stack instead. Here is an algorithm for doing a depth first traversal which uses a stack rather than recursion. As you can see by running an example (see lecture slides), this algorithm visits the list of children of a node in the opposite order to that defined by the `for` loop.

```
  treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty{
      cur = s.pop()
      visit cur
      for each child of cur
         s.push(child)
      //  'visit cur' could be put here instead
    }
  }
```

## Breadth first traversal

What happens if we use a queue instead of a stack in the previous algorithm?

```
treeTraversalUsingQueue(root){
   q = empty queue
   q.enqueue(root)
   while !q.isEmpty() {
     cur = q.dequeue()
     visit cur
     for each child of cur
        q.enqueue(child)
}
```

As shown in the example in the lecture, this algorithm visits all the nodes at each depth, before proceeding to the next depth. This is called *breadth first* traversal. The queue-based algorithm effectively does the following:

```
for i = 0 to height
    visit all nodes at level i
```

You should work through the example in the slides to make sure you understand why using queue here is diffrent from using a stack.

## A note about implementation

Recall first-child/next-sibling data structure for representing a tree, which we saw last lecture. Using this implementation, you can replace the line

```
    for each child of cur
       ...
```

with the following. Here we are iterating through the siblings i.e. a singly linked list of the children of a node.

```
     child = child.firstChild
     while (child != null){
        ...                   //  do something at that child
       child = child.nextSibling
     }
```

## Binary Trees

The *order* of a (rooted) tree is the maximum number of children of any node. A tree of order $n$ is called an $n$-ary tree. It is very common to use trees of order 2. These are called *binary trees*.

Each node of a binary tree can have two children, called the *left child* and *right child*. The terms "left" and "right" refer to their relative position when you draw the tree.

How many nodes can a binary tree have at each level? The root has one node. Level 1 can have two nodes (the two children of the root). Level 2 can have four nodes, namely each child at level 1 can have two children. You can easily see that the maximum number of nodes doubles at each level, and so level $l$ can have $2^l$ nodes. For a binary tree of height $h$, the maximum number of nodes is thus:

$$n_{max} = \sum_{l=0}^{h} 2^l = \frac{2^{h+1}-1}{2-1} = 2^{h+1}-1.$$

You have seen this geometric series before in the course, and you will see it again.

The minimum number of nodes in a binary tree of height $h$ is of $h+1$, namely if each node has at most one child and there is one leaf, which by definition has no children. It follows from the above two observations that

$$h+1 \leq n \leq 2^{h+1}-1$$

ASIDE: We can rearrange this equation to get a lower and upper bound of the height $h$ in terms of the number of nodes.

$$\log(n+1)-1 \leq h \leq n-1.$$

### Binary tree nodes: Java

We have seen the first-child/next-sibling data structure for general trees. For binary trees, one typically uses the following data structure for the node instead:

```
class  BTNode<T>{
   T             e;
   BTNode<T>   left;
   BTNode<T>   right;
}
```

One can have a `parent` reference too, if necessary, but we don't use it now. Recall that a `parent` reference is analogous to a `prev` reference in a doubly linked list.

### Binary tree traversal

A binary tree is a special case of a tree, so the algorithms we have discussed for general trees apply to binary trees as well. We saw two simple depth-first search algorithms for general trees, namely pre- and post-order. For binary trees these algorithms can be written as follows. Note that we test that the root is not null here, which would be the base case. This is slightly different from last lecture, where we assumed that the root was not null, and then only did the recursive call to children when there were children.

```
preorderBT(root){
    if (root is not null){             // base case
       visit root
       preorderBT(root.left)
       preorderBT(root.right)
    }
}

postorderBT(root){
    if (root is not null){             // base case
       postorderBT(root.left)
       postorderBT(root.right)
       visit root
    }
}
```

For binary trees, there is one further traversal algorithm to be considered, which is called *in-order traversal*.

```
inorderBT(root){
    if (root is not null){             // base case
       inorderBT(root.left)
       visit root
       inorderBT(root.right)
    }
}
```

You *could* define an inorder traversal for general trees. For example, you could visit the first child, then visit the root, then visit any remaining children. But such inorder traversals are typically not done for general trees.

**Example (different from one given in slides)**

```
          a
        /    \
       b      c
      / \    / \
     d   e  f   g
```

```
level order:  a b c d e f g   (breadth first)
pre-order:    a b d e c f g   (depth first)
post-order:   d e b f g c a       "
in-order:     d b e a f c g       "
```

## Expressions

You are familiar with forming expressions using *binary operators* such as `+,-,*, /, %, ^`. (The operator `^` is the power operator i.e. `x ^ n` is `power(x,n)`.) Each of the operators takes two arguments, called the left and right *operands*. Let's define a set of simple expressions recursively as follows, where the symbol `|` means *or*.

```
baseExpression  =  variable | integer
operator        =  + | - | * | / | ^
expression      =  baseExpression |  expression operator expression
```

So, an `expression` can consist of either a base expression, or it can consist of one expression followed by an operator followed by an expression. Notice that expressions are defined recursively, and that we have a base case.

## Expression trees

**[In the slides, I used slightly different examples from what I use below.]**

You can represent these expressions using trees, called *expression trees*. For example, `x + 4 * y` could be defined using either of these two trees. But as we'll see next, the meaning of the two is different.

```
     +                          *
   /   \                      /   \
  x     *                    +      y
      /   \                /   \
     4     y              x     4
```

When we have an expression with multiple operators, there is a particular order in which the operators are supposed to be applied. You learned these precedence orderings in grade school. For example "`x + 4 * y`" is to be interpreted as "`x + (4 * y)`" shown on the left, rather than "`(x + 4) * y`" shown on the right. As I mentioned in the lecture, my MS Windows calculator ignores the precedence ordering of * over +. There is also a convention that `6 ^ z ^ 8` means `6 ^ (z ^ 8)` rather than `(6 ^ z) ^ 8`. See the example that I gave in the slides.

The above precedence order implies that an expression such as

$$\mathtt{x + 4 * y - 6 \mathbin{\hat{}} z \mathbin{\hat{}} 8}$$

can be uniquely interpreted as if there were a nesting of brackets:

$$\mathtt{(x + (4 * y)) - (6 \mathbin{\hat{}} (z \mathbin{\hat{}} 8))}$$

and the expression can be represented as a tree:

```
             -
          /     \
         +       ^
```

```
 /  \    /  \
x    *  6    ^
    / \     / \
   4   y   z   8
```

Looking at the original expression with no brackets, you may be tempted to think that the + operator should be in the root rather than the - operator being in the root because the + operator is to the left. In fact, it works the other way: higher precedence means it is evaluated first, which means it is deeper in the tree.

Expression trees can be evaluated recursively as follows.

```
evaluateET(root){
   if (root is a leaf)   // root has no children
      return value
   else{ //  the root is an operator
      firstOperand = evaluateET(left child of root)
      secondOperand = evaluateET(right child of root)
      return evaluate(firstOperand, root, secondOperand)
   }
```

We may think of this algorithm as performing a *postorder* traversal of the tree in the sense that, to evaluate the expression defined by a tree, you *first* need to evaluate the left and right child of the root, and *then* you can apply the operator at the root.

## In-fix, pre-fix, post-fix expressions

You are used to writing expressions as two operands separated by an operator. This representation is called *infix*, because the operator is "in" between the two operands. For infix expressions, the order of evaluation is determined by precedence rules.

An alternative way to write an expression is to use *prefix* notation. Here the operator comes *before* the two operands. For example,

```
- + x * 4 y ^ 6 ^ z 8
```

which is interpreted as

```
(- (+ x (* 4 y))(^ 6 (^ z 8))) .
```

Notice that a prefix expression gives the ordering of elements visited in a preorder traversal of the expression tree.

An second alternative is a *postfix* expression, where the operators comes *after* the two operands, so

```
x 4 y * + 6 z 8 ^ ^ -
```

is interpreted as

```
((x (4 y *) + ) (6 (z 8 ^) ^ ) - ) .
```

The ordering of elements is the visit order in a post-order traversal of the expression tree.

One can formally define *in, pre, and post*fix expressions recursively as follows:

```
baseExpression     =  digit | letter
operator           =  + | - | * | / |  ^
infixExpression    =  baseExpression |  infixExpression operator infixExpression
prefixExpression   =  baseExpression |  operator prefixExpression prefixExpression
postfixExpression =  baseExpression |  postfixExpression  postfixExpression operator
```

[ASIDE: Prefix notation is sometimes called "Polish" notation – it was invented by a Polish logician, Jan Lukasiewicz (about 100 years ago). Postfix notation is sometimes called "reverse Polish notation" or RPN. Many calculators, in particular, Hewlett-Packard calculators require that users enter expressions using RPN. There are youtube videos like this showing how to do this, if you are interested.]

For computer science, the advantage of postfix (and prefix) expressions over infix expressions is that with postfix expressions you do not need a precedence rule to define the order of operations. Instead, as shown below, one can use a simple stack-based algorithm for evaluating a postfix expression. The algorithm does not need to know about precedence orderings, since these are "built in" to the postfix expression. See the slides for an example of how the stack evolves over time for a particular expression.

```
s = empty stack
cur = head;
while (cur != null){
   if (cur.element is a variable or number)
      s.push(cur.element)
   else{ //  cur is an operator
     operand2 = s.pop()    // opposite order to push
     operand1 = s.pop()    //   "
     operator = cur.element
     s.push( evaluate( operand1 operator operand2 ) )
   }
   cur = cur.next
}
```

## Binary Search Trees

Today we consider a specific type of binary tree in which there happens to be an ordering defined on the set of elements the nodes. If the elements are numbers then there is obviously an ordering. If the elements are strings, then there is also a natural ordering, namely the dictionary ordering, also known as "lexicographic ordering".

### Definition: binary search tree

A *binary search tree* is a binary tree such that

- each node contains an element, called a *key*, such that the keys are comparable, namely there is a strict ordering relation `<` between keys of different nodes
- any two nodes have different keys (i.e. no repeats/duplicates)
- for any node,
  - all keys in the left subtree are less than the node's key
  - all keys in the right subtree are greater than the node's key.

  This last condition is stronger than just saying that the left child's key is less than the node's key which is less than the right child's key. For example, the following is not a binary search tree.

```
    5
   /
  2
 / \
1   6
```

One important property of binary search trees is that *an inorder traversal of a binary search tree gives the elements in their correct order.* Here is an example with nodes containing keys `abcdefghijklm`. Verify that an inorder traversal gives the elements in their correct order.

```
            g
         /     \
       d          k
     /  \       /   \
   b     f     i      l
 /  \   /     / \      \
a    c  e    h   j       m
```

Here is another example binary search tree with the same set of keys:

```
                k
             /     \
           e         l
         /  \         \
       b      g         m
     /  \    / \
    a    c  f   i
          \    / \
           d  h   j
```

## BST operations

One performs several common operations on binary search trees:

- `find(key)`: given a key, find the node containing that key (or null if key is not in tree) and return a reference to that node
- `findMin()` or `findMax()`: find the node containing the smallest or largest key in the tree and return a reference to the node containing that key
- `add(key)`: insert a new node into the tree such that the node contains the key and the node is in its correct position (if the key is already in the tree, then do nothing)
- `remove(key)`: remove from the tree the node containing the key (if it is present), adjust the tree if necessary so that it is a binary search tree

Here are algorithms for implementing these operations.

```
find(root,key){                  //    returns a node
  if (root == null)
     return null
  else if (root.key == key))
    return root
  else if (key < root.key)
     return  find(root.left, key)
  else
     return  find(root.right, key)
}
```

```
findMin(root){            //  returns a node
   if (root == null)      // only necessary for the first call
     return null
   else if (root.left == null)
      return root
   else
      return findMin(root.left)
}
```

For example, here the minimum key is `a`.

```
         f
      /     \
    c        m
  /   \     /
 a    e    g
```

Notice however that the minimum key is not necessarily a leaf i.e. it can occur if the key has a right child but no left child. Here the minimum key is `c`:

```
         f
      /     \
    c        m
     \      /
      e    g
     /
    d
```

The reasoning and method is similar for `findMax`.

```
findMax(root){                  // returns a node
   if (root == null)
      return null
   else if (root.right == null))
      return root
   else
      return findMax(root.right)
}
```

Let's next consider adding (inserting) a key to a binary search tree. If the key is already there, then do nothing. Otherwise, make a new node containing that key, and insert that node into its *unique* correct position in the tree.

```
add(root,key){      // returns root
   if (root == null)              //  base case:
      root =  new BSTnode(key)    //  makes a new node and returns it
   else if (key < root.key){
      root.left = add(root.left,key)
   else if (key > root.key){
      root.right = add(root.right,key)
   return root
}
```

The base case is `root == null` and in that case a leaf is added. For the non-base case, the situation is subtle. The code says that a new node is added to either the left or right subtree and then the reference to that left or right subtree is re-assigned. It is reassigned to the root of the left or right subtree, which has the new node added it. Why is it necessary to reassign the reference like that?

Suppose we add the new node to the left subtree. If the new node is a descendent of the root node of the left subtree , then the assignment `root.left = add(root.left,key)` doesn't change the reference `root.left` in the `root` node; it just assigns it to the same node it was referencing beforehand. *However,* if `root.left` was null and then the new node was added to the left subtree of `root`, then the call `add(root.left,key)` will create and return the new node, namely it is the root of a subtree with one node. If we don't assign that node to `root.left` as the code says, then the new node that is created would not be added to the tree. That's why we do need to assign the reference.

Next, consider the problem of removing a node from a binary search tree.

```
remove(root, key){                  // returns root
    if( root  == null )
         return null
    else if ( key < root.key )                                    (*)
         root.left = remove( root.left, key )
    else  if ( key > root.key )                                    (*)
         root.right = remove( root.right, key)
    else  if root.left == null                                     (**)
         root  =  root.right         // or just "return root.right"
    else  if root.right == null                                    (**)
         root  = root.left           // or just "return root.left"
    else{                                                          (***)
         root.key = findMin( root.right).key
         root.right = remove( root.right, root.key )
    }
    return root
}
```

The (*) conditions handle the case that the key is not at the root, and in this case, we just recursively remove the key from the left or right subtree. Note that we replace the left or right subtree with a subtree that doesn't contain the key. To do this, we use recursion, namely we remove the key from the left or right subtree.

The more challenging case is that the key that we want to remove is at the root (perhaps after a sequence of recursive calls). In this case we need to consider four possibilities:

- the root has no left child
- the root has no right child
- the root has no children at all
- the root has both a left child and a right child

In the first two cases – see (**) in the algorithm – we just replace the root node with the subtree in the non-empty child. Note that the third case is accounted for here as well; since both children are null, the root will become null.

In the fourth case – (***) – we take the following approach. We replace the root with the minimum node in the right subtree. We do so in two steps. We copy the key from the smallest node in the right subtree into the root node, and then we remove the smallest node node from the the right subtree.

Finally, one common question is, why do we need a `return` statement? Why can't we just remove the node? To understand why, consider line (*) and suppose this condition is met and the left child is a leaf. After we call remove on the left child, the left child should become null. So we really do need that line `root.left = ...` , namely it will be assigned the value `null`. If we just wrote `remove( root.left, key)` without the assignment, then `root.left` would continue to point to that same node, rather than removing it.

**Example (remove)**

Take the following example with nodes `abcdefghijklm`:

```
            g
         /     \
       d          k
     /  \       /   \
   b     f     i      l
 /  \   /     / \      \
a    c  e    h   j      m
```

and then remove elements `g,f,k` in that order.

```
           h
        /     \
     d           k
   /   \       /   \
  b     f     i      l
 /  \   /       \      \
a    c  e        j      m
```

```
           h
        /     \
      d          k
     /  \       /   \
   b     e     i      l
 /  \           \      \
a    c           j      m
```

```
            h
         /     \
       d          l
     /  \       /   \
   b     e     i      m
 /  \           \
a    c           j
```

## Computational Complexity

Let's consider the best and worst case performance. The best case performance for this data structure tends to occur when the keys are arranged such that the tree is balanced, so that all leaves are at the same depth (or depths of two leaves differ by at most one). However, if one adds keys into the binary tree and doesn't rearrange the tree to keep it balanced, then there is no guarentee that the tree will be anywhere close to balanced, and in the worst case a BST with $n$ nodes could have height $n-1$. This implies the following best and worst cases:

| Operations/Algorithms for Lists | Best case, $t_{best}(n)$ | Worst case, $t_{worst}(n)$ |
|---|---|---|
| findMax(), findMin() | $O(1)$ | $O(n)$ |
| find(key) | $O(1)$ | $O(n)$ |
| add(key) | $O(1)$ | $O(n)$ |
| remove(key) | $O(1)$ | $O(n)$ |

In COMP 251, you will learn about balanced binary search trees, e.g. AVL trees or red-black trees. If a tree is balanced, then the operations are all $O(\log n)$. This is an interesting topic, but you'll need to wait for COMP 251.

## Priority Queue

Recall the definition of a queue. It is a collection where we remove the element that has been in the collection for the longest time. Alternatively stated, we remove the element that first entered the collection. A natural way to implement such a queue was using a linear data structure, such as a linked list or a (circular) array.

A *priority queue* is a different kind of queue, in which the next element to be removed is defined by a *priority*, which is a more general criterion. For example, in a hospital emergency room, patients are treated not in a first-come first-serve basis, but rather by the urgency of the case. To define the next element to be removed, it is necessary to have some way of comparing any two objects and deciding which has greater priority. *The next element to be removed is the one with greatest priority.* Heads up: with priority queues, one typically assigns small numerical values to high priorities. Think "my number one priority", "my number 2 priority", etc. (One could assign larger numerical values to higher priorities, but that's not what's typical.)

One way to implement a priority queue of elements (often called *keys*) is to maintain a sorted list. This could be done with a linked list or array list. Each time a key is added, it would need to be inserted where it belongs into the sorted list. If the number of keys were huge, however, then this would be an inefficient representation. For example, if we use a singly linked list, then we can remove from the front of the list in $O(1)$ time which is good. However, adding would still be $O(n)$ in the worst case since we would need to find where the key belongs in the list and this might require iterating through the whole list. Similarly, if we used an arraylist then the time for adding would again still be $O(n)$; although we could use binary search to find where the new key goes in $O(\log n)$, we would still need to need to shift the keys to make room which is $O(n)$. (If we sort the arraylist by decreasing value, then we can remove in O(1) time. So its the add that's the problem.)

## Heaps

The usual way to implement a priority is to use a data structure called a *heap.* As we will see, this will allow us to do both add and remove operations in $O(\log n)$ time.

To define a heap, we first need to define a *complete binary tree.* We say a binary tree of height $h$ is *complete* if every level $l$ less than $h$ has the maximum number ($2^l$) of nodes, and in level $h$ all nodes are as far to the left as possible. A *heap* is a complete binary tree with a key at each node, such that they keys are comparable and satisfy the property that *each node's key is less than the keys of its children.* This is the default definition of a heap, and is sometimes called a *min heap.*

A *max heap* is defined similarly, except that the key stored at each node is greater than the keys stored at the children of that node. Unless otherwise specified, we will assume a min heap. Note that it follows from the definition that the smallest key in a (min) heap is stored at the root.

As with stacks and queues, the two main operations we perform on heaps are `add` and `remove`.

### add

To add a key to a heap, we create a new node and insert it in the next available position of the complete tree. If level $h$ is not full, then we insert it next to the rightmost leaf. If level $h$ is full, then we start a new level at height $h + 1$.

Once we have inserted the new node, we need to ensure that the heap property is satisfied. The problem is that the key of the parent of the new node is greater than the key of this node. This problem is easy to solve. We can just swap the keys of the node and its parent. We then need to repeat the same test on the new parent node, etc, until we reach either the root, or until the key of the parent node is less than the key of the current node. This process of moving a node up the heap, is often called "upheap".

```
add(key){
   cur = new node at next leaf position
   cur.key = key
   while (cur != root) && (cur.key < cur.parent.key){
       swapkey(cur, parent)
       cur = cur.parent
   }
}
```

You might ask whether swapping the key at a node with its parent's key can cause a problem with the node's sibling (if it exists). No, it cannot. Before the swap, the parent is less than the sibling.[15] So if the current node is less than its parent, then the current node must be less than the sibling too (current < parent < sibling). So, swapping the node's key with its parent's key preserves the heap property with respect to the node's current sibling.

For example, suppose we have a heap with two keys `e` and `g` as shown below and so `e < g`. Then we add a key to the `*` position below and we find that `* < e`. So we swap them. Since `* < e` and `e < g`, we can be sure that `* < g`.

```
  e
/   \
g    *
```

Here is a bigger example. Suppose we add key `c` to the following heap.

```
          a
        /   \
       e      b
     / \    / \
    f   l   u  k
   /
  m
```

We add a node which is a sibling to `m` and assign `c` as the key of the new node.

[15]Here I say that one node is less than another, but what I really mean is that the key at one node is less than the key at the other node.

```
         a
       /   \
      e      b
     / \    / \
    f   l   u  k
   / \
  m   c
```

Then we observe that `c` is less than `f`, the key of its parent, so we swap `c,f` to get:

```
         a
       /   \
      e      b
     / \    / \
    c   l   u  k
   / \
  m   f
```

Now we continue up the tree. We compare `c` with its new parent's key `e`, see that the keys need to be swapped, and swap them to get:

```
         a
       /   \
      c      b
     / \    / \
    e   l   u  k
   / \
  m   f
```

Again we compare `c` to its parent. Since `c` is greater than `a`, we stop and we're done.

**removeMin**

Next, let's look at how we remove keys from a heap. Since the heap is used to represent a priority queue, we remove the minimum key, which is the root.

How do we fill the hole that is left by the key we removed ? We first copy the last key in the heap (the rightmost key in level $h$) into the root, and delete the node containing this last key. We then need to manipulate the keys in the tree to preserve the heap property that each parent is less than its children.

We start at the root, which contains a key that was previously the rightmost leaf in level $h$. We compare the root to its two children. If the root is greater than at least one of the children, we swap the root with the smaller child. Moving the smaller child to the root does not create a problem with the larger child, since the smaller child is smaller than the larger child.

```
removeMin(){                 // returns smallest key
   tmp = root.key
   remove last leaf node and put its key into the root
```

```
      cur = root
      while ((cur has a left child) and
             ((cur.key > cur.left.key) or
              (cur has a right child and cur.key > cur.right.key)))
          minChild = child with the smaller key
          swapkey(cur, minChild)
          cur = minChild
      }
      return tmp
   }
```

The condition in the while loop is rather complicated, and you may have just skipped it. Don't. There are several possible events that can happen and you need to consider each of them. One is that the current node has no children. In that case, there is nothing to do. The second is that the current node has one child, in which case it is the left child: we then need to decide if this child is smaller than the current node. The third is that the current node might have two children. In that case, we need to check if one of these two children is smaller than the current node, and swap with the smaller one.

Here is an example:

```
     a                               d              b
    / \             / \            /  \           /   \
   b   c   -->     b   c     -->  b    c   -->   d     c
  / \ /           / \ /          / \            / \
 k  e d          k  e d         k   e          k   e
```

If we apply `removeMin()` again and again until all the keys are gone, we get the following sequence of heaps with keys removed in the following order: `b, c, d, e, k`.

```
     c             d          e            k
    / \           / \        /
   d   e         k   e      k
  /
 k
```

## Implementing a heap using an array

A heap is defined to be a complete binary tree. If we number the nodes of a heap by a level order traversal and start with index 1, rather than 0, then we get an indexing scheme as shown below.

```
             1
         /       \
      2             3
   /   \        /    \
  4     5     6       7
/  \   / \   / \    /  \
8   9  10 11 12 13  etc
```

These numbers are NOT the keys stored at the node. Rather we are just numbering the nodes so we can index them.

This array representation defines a simple relationship between a tree node's index and its children's index. If the node index is $i$, then its children have indices $2i$ and $2i + 1$. Similarly, if a non-root node has index $i$ then its parent has index $i/2$.

## add(key)

Earlier I presented an algorithm for adding a key to a heap which was based on a binary tree structure. Here I'll re-write that algorithm using an array and the indexing scheme above. Let `size` be the number of keys in the heap. These keys are stored in array slots 1 to `size`, i.e. recall that slot 0 is unused so that we can use the simple relationship between a child and parent index.

```
add(key ){
  size = size + 1          // number of keys in heap
  heap[ size ] = key   // assuming array has room for another key
  i = size

//  the following is called "upHeap"

  while (i > 1  and heap[i] < heap[ i/2 ]){
    swapkeys( i, i/2 )
    i = i/2
  }
}
```

## Example

Suppose we have a heap with eight characters and we add one more, a `c`.

```
1   2   3   4   5   6   7   8   9
-------------------------------------
a   e   b   f   l   u   k   m   c
a   e   b   c   l   u   k   m   f   <----  c swapped with f (slots 9 & 4)
a   c   b   e   l   u   k   m   f   <----  c swapped with e (slots 4 & 2)
```

Next lecture we will continue with heaps, examining the time complexity of building a heap. We will also look at the heapsort algorithm.

At the end of last lecture we showed how a heap could be represented using an array, and we rewrote the `add` method using the array representation instead of the binary tree representation. Today we will examine how to build a heap by repeatedly calling the `add` method on a list of keys, and we will analyze the time complexity of building a heap. We will then review the `removeMin` method and rewrite it in terms of the array representation. Finally we will put this all together and show how to sort a list of keys using a heap – called *heapsort*.

## Building a heap

We can use the `add` method to build a heap as follows. Suppose we have a list of `size` keys and we want to build a heap.

```
buildHeap(list){
   create an empty heap
   for (k = 0; k < list.size; k++)
        heap.add( list[k] )
   return heap
}
```

We can write this in a slightly different way.

```
buildHeap(list){
   create an array arr with list.size+1 slots
   for (k = 1; k <= list.size; k++){
       arr[k] = list[k-1]       //  say list indices are 0, .. ,size-1
       upHeap(arr, k)
   }
}
```

where `upHeap(arr, k)` is defined as follows.

```
upHeap(arr, k){
    i = k
    while (i > 1  and  arr[i] < arr[ i/2 ]){
       swapkeys( i, i/2 )
       i = i/2
    }
}
```

## Time complexity

How long does it take to build a heap in the best and worst case? Before answering this, let's recall some notation. We have seen the "floor" operation a few times: it rounds down to the nearest integer. If the argument is already an integer then it does nothing. We also can define the ceiling, which rounds up. It is common to use the following notation:

- $\lfloor x \rfloor$ is the largest integer that is less than or equal to $x$. $\lfloor\ \rfloor$ is called the *floor* operator.

- $\lceil x \rceil$ is the smallest integer that is greater than or equal to $x$. $\lceil\ \rceil$ is called the *ceiling* operator.

Let $i$ be the index in the array representation of keys/nodes in a heap, then $i$ is found at level *level* in the corresponding binary tree representation. The level of the corresponding node $i$ in the tree is such that

$$2^{level} \le i < 2^{level+1}$$

or

$$level \le \log_2 i < level + 1,$$

and so

$$level = \lfloor \log_2 i \rfloor.$$

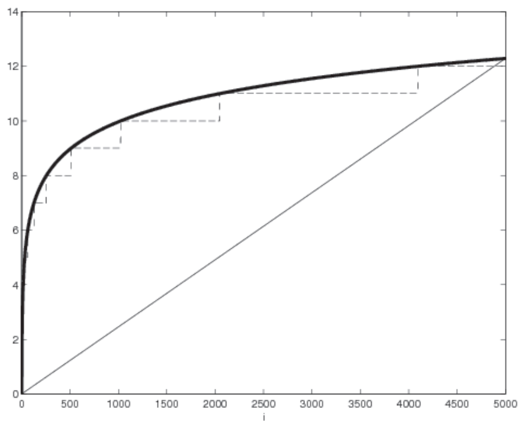
We can use this to examine the best and worst cases for building a heap.

In the best case, the node $i$ that we add to the heap satisfies the heap property immediately, and no swapping with parents is necessary. This can happen if the keys are added in their natural order. In this case, building a heap takes time proportional to the number of nodes $n$. So, best case is $O(n)$.

What about the worst case? Since $level = \lfloor \log_2 i \rfloor$, when we add key $i$ to the heap, in the *worst case* we need to do $\lfloor \log_2 i \rfloor$ swaps up the tree to bring key $i$ to a position where it is less than its parent, namely we may need to swap it all the way up to the root. (This can happen if the keys are added in their reverse order. ) If we are adding $n$ nodes in total, the worst case number of swaps is:

$$t(n) = \sum_{i=1}^{n} \lfloor \log_2 i \rfloor$$

To visualize this sum, consider the plot below which show the functions $\log_2 i$ (thick) and $\lfloor \log_2 i \rfloor$ (dashed) curves up to $i = 5000$. In this figure, $n = 5000$.



The area under the dashed curve is the above summation. It should be visually obvious from the figures that

$$\frac{1}{2} n \log_2 n < t(n) < n \log_2 n$$

where the left side of the inequality is the area under the diagonal line from (0,0) to $(n, \log_2 n)$ and the right side $(n \log_2 n)$ is the area under the rectangle of height $\log_2 n$. From the above inequalities, we conclude that in the worst of building a heap is $O(n \log_2 n)$.

## removeMin

Next, recall the `removeMin()` algorithm from last lecture. We can write this algorithm using the array representation of heaps as follows.

```
removeMin( arr ){
    key = arr[1]            //   heap[0] not used.
    arr[1] = arr[size]
    arr[size] = null
    size = size - 1
    downHeap(arr, size)       //   see below
    return key
}
```

This algorithm saves the root key to be returned later, and then moves the key at position `size` to the root. The situation now is that the two children of the root (node 2 and node 3) and their respective descendents each define a heap. But the tree itself typically won't satisfy the heap property: the new root will be greater than one of its children. In this typical case, the root needs to move down in the heap.

The `downHeap` helper method moves an key from a starting position in the array down to some maximum position in the heap. We will see later (in heapsort) why we need this maximum position as a parameter.

```
downHeap(arr, size){  //  move key from starting position
                      //  down to at most position size
   i = 1
   while (2*i <= size){        // check if there is a left child
      child = 2*i
      if (child < size) {      // check if there is a right sibling
         if (arr[child + 1] < arr[child])  //  is rightchild < leftchild ?
              child = child + 1    //  if so, then smaller child is right
      }
      if ( arr[child] < arr[ i ]){   // swap with child?
         swapkeys(child , i)
         i = child
      }
      else  return    // exit
   }
}
```

This is essentially the same algorithm we saw last lecture. What is new here is that I am expressing it in terms of the array indices.

## Heapsort

A heap can be used to sort a set of keys. The idea is simple. Just repeatedly remove the minimum key by calling `removeMin()`. This naturally gives the keys in their proper order.

Here I give an algorithm for sorting "in place". We repeatedly remove the minimum key the heap, reducing the size of the heap by one each time. This frees up a slot in the array, and so we insert the removed key into that freed up slot. This yields the keys sorted in the reverse order. That's easy to fix afterwards: just reverse the order of the keys.

The pseudocode below does exactly what I just described, although it doesn't say "`removeMin()`. Instead, it says to swap the root key i.e. `heap[1]` with the last key in the heap i.e. `heap[size+1-i]`. After `i` times through the loop, the remaining heap has `size - i` keys, and the last `i` keys in the array hold the smallest `i` keys in the original list. So, we only downheap to index `size - i` .

```
heapsort(list){
   arr = buildheap(list)
   n = list.size
   for i = 1 to n-1{
      swapkeys( arr, 1,  n + 1 - i)
      downHeap( arr,  n - i)
   }
   return reverse(arr)    // reverse keys in slots 1 to n
}
```

The end result is an array that is sorted from largest to smallest. Since we want to order from smallest to largest, we must `reverse` the order of the keys which are in slots 1 to $n$.

## Example

The example below shows the state of the array after each pass through the for loop. The vertical line marks the boundary between the remaining heap (on the left) and the sorted keys (on the right).

```
1   2   3   4   5   6   7   8   9
-------------------------------------
a   d   b   e   l   u   k   f   w |
b   d   k   e   l   u   w   f | a   (removed a, put w at root, ...)
d   e   k   f   l   u   w | b   a   (removed b, put f at root, ...)
e   f   k   w   l   u | d   b   a   (removed d, put w at root, ...)
f   l   k   w   u | e   d   b   a   (removed e, put u at root, ...)
k   l   u   w | f   e   d   b   a   (removed f, put u at root, ...)
l   w   u | k   f   e   d   b   a   (removed k, put w at root, ...)
u   w | l   k   f   e   d   b   a   (removed l, put u at root, ...)
w | u   l   k   f   e   d   b   a   (removed u, put w at root, ...)
```

Note that there is no need for an $n^{th}$ pass through the loop since the heap after $n-1$ passes has only one key left (`w` in this example), which is the largest key.

## Time complexity for heapsort

What is the time complexity of heapsort? If we have $n$ keys, then we have to go through the for loop $n$ times. Each time through, we need to swap down at most the height of the tree (and for many keys, we don't have to go that far). More precisely, the worst case number of swaps is

$$t(n) = \sum_{i=1}^{n} \lfloor \log_2(n-i) \rfloor$$

which is essentially the same for as the worst case for buildHeap earlier in the lecture. Thus, the worst case time complexity for the $n$ removals is $O(n \log_2 n)$.

The final step, where we reverse the keys in the array only needs to be done once. It is done in $O(n)$ time, by swapping $i$ and $n+1-i$ for $i = 1$ to $\frac{n}{2}$. So heapsort in the worst case is $O(n \log_2 n)$.

You might think the best case for heapsort is that the keys in the list are already in order, since in that case the buildheap step is $O(n)$ rather than $O(n \log_2 n)$, i.e. there are no swaps in the build heap step since each key is compared to its parent and found to be greater than its parent. However, the $n-1$ removes must be considered also. If the keys are already in order, then the largest $n/2$ keys are in the bottom level of the heap. So when we do the first $n/2$ removes, these largest keys will be brought into slot 1 of the array and will need to be downHeap'ed. They will tend to downheap quite far since they are the largest $n/2$ keys.

Even if the keys are not in order, once we build a heap, the last $n/2$ keys in the array will tend to be amoung the largest keys since they are at the bottom level of the heap. So again, when we do the first $n/2$ removes, these keys will be brought to the top and then have to be downheaped. Since they tend to be larger keys, they will tend to downheap to deeper levels.

What if the keys are in the opposite order to begin with? In this case, the buildheap step will take time $O(n \log_2 n)$ since each key that is added needs to be swapped all the way to the root.

So from the above, you can see that heapsort requires close to $\log n$ swaps for each of the keys. As it turns out, this makes heapsort run slower in practice than quicksort.

## ASIDE: Why is quicksort quicker than heapsort?

Real quicksort implementations do not choose the last element of the given list as the pivot. The reason is that choosing the last element would lead to $O(n^2)$ performance if the list is already sorted or nearly so, and this case does come up in practice sometimes. Instead, quicksort can choose a different element and swap this element with the last element of the input list, and then run the algorithm as presented (where pivot is the last element).

For example, a better pivot choice is to take the first, middle, and last elements of the list, and choose the median value of these three. If the list is sorted forward or sorted backwards to begin with, then choosing the "median of three" as it is called (instead of simply choosing the last element) will indeed split in the middle (yipeee!). And if the list has random order, then the median of three will tend to be closer to the actual median than if one just choses the last element. Of course, determining which of the three chosen ones *is* the median requires a bit of extra work for every split, and so it increases the constant factor at each step of the recursion. But in practice this extra work tends to pay off overall.

This is related to a question that was asked in class: why doesn't heapsort or quicksort just check at the start if the list is already sorted, which would take time only $O(n)$. Yes, you could do that. And it might make sense to do that if this case did arise in practice often enough. But if this case arises only rarely, then this would be extra work that you need to do every time, but that would benefit you only rarely.

## Maps

The last few lectures, we have looked at ways of organizing a set of elements that are comparable, namely binary search trees and heaps. When elements are comparable, we have the potential to access elements quickly, rather than having to search through all elements to find the one we want. Binary search trees are designed for this purpose, whereas priority queues are more specific as they are designed to access just the element with lowest priority value.

Today we will begin looking at ways of organizing things which doesn't require that they are comparable. We will specifically consider how to represent *maps*. You are familiar with maps already. In high school math and in Calculus and linear algebra, you have seen functions that go from (say) $\Re^n$ to $\Re^m$.
[ASIDE: To be more mathematically precise, a map is a set of ordered pairs $\{(x, f(x))\}$ where $x$ belongs to some set called the *domain* of the map, and $f(x)$ belongs to a set called the *co-domain*. The word *range* is used specifically for the set $\{f(x) : (x, f(x)) \text{ is in the map}\}$. That is, some values in the co-domain might not be reached by the map.]

### Maps as (key,value) pairs

You are also familiar with the idea of maps in your daily life. You might have an address book which you use to look up home or business addresses, telephone numbers, or email addresses. You index this information with a name. So the mapping is from name to address/phone/email. A related example is "Caller ID" on your phone. Someone calls from a phone number (the index) and the phone tells you the name of the person. Many other traditional examples are social security number or health care number or student number for the index, which maps to a person's employment record, health file, or student record, respectively. In our definitions below, we will use the more general term 'key' rather than 'index.

We will use the following definitions and notation for maps. Suppose we have two sets: a set of keys $K$, and a set of values $V$. A *map* is a set of ordered pairs

$$M = \{(k, v) : k \in K, v \in V\} \subseteq K \times V.$$

The pairs are called *entries* in the map.

A map cannot just be any set of (key, value) pairs, however. Rather, for any key $k \in K$, there is *at most* one value $v$ such that $(k, v)$ is in the map. We allow two different keys to map to the same value, but we do not allow one key to map to more than one value. Also note that not all elements of $K$ need to be keys in the map.

For example, let $K$ be the set of integers and let $V$ be the set of strings. Then the following is a map:

$$\{(3, \texttt{cat}), (18, \texttt{dog}), (35446, \texttt{meatball}), (5, \texttt{dog}), \}$$

whereas the following is not a map,

$$\{(3, \texttt{cat}), (18, \texttt{dog}), (35446, \texttt{meatball}), (5, \texttt{dog}), (3, \texttt{fish})\}$$

because the key 3 has two values associated with it.

## Map ADT

The basic operations that we perform on a map are:

- put(key, value) – this adds a new entry to the map

  What if the map previously contained a mapping for the key? As we will see, for a Java `Map` interface, the policy is that the old value is replaced by the new value, and the old value is returned. (If the map didn't contain that key, then `put` returns null.)

- get(key) – this returns the value associated with the entry (key, value).

  What if the given key did not have an entry in the map? The Java `Map` policy is that `get` would return null.

- remove(key) – this removes the entry (key, value)

  The Java `Map` policy is that it return the value if the key was present, and it returns null if the key wasn't present.

There are other `Map` methods such as `contains(key)` or `contains(value)` as well but the above are the main ones.

## Map data structures

How can we represent a map using data structures that we have seen in the course? We might have a key type `K` and a value type `V` and we would like our map data structure to hold a set of object pairs $\{\ (k, v)\ \}$, where $k$ is an object of type `K` and $v$ is an object of type `V`.

We could use an arraylist or a linked list of entries, for example. See the Exercises for a few questions on the time complexity of put, get, and remove in this case

What if we made a stronger assumption that the keys of map are comparable? In this case, we could organize the entries of the map using a sorted arraylist or a binary search tree. Again, see the Exercises. By the way, what about using a heap? A heap would not be an appropriate data structure for a map because it only allows you to efficiently get or remove the minimum key. For a general `get` operation, a heap would be $O(n)$ in the worst case, since one would have to traverse the entire heap. Note that the heap is represented using an array, so this would just be a loop through the elements of the array.

Another special case to consider is that the keys `K` are small positive integers. In this case, we could just use an array and have the key be an index into the array and the value be stored at that slot in the array. Note that this typically will *not* be an arraylist, since there may be gaps in the array between entries. Using an array would give us access to map entries in $O(1)$ time. However, this would only work well if the integer values are within a small range. For example, if the keys were social insurance numbers (in Canada), which have 9 digits, then it would not work well because we would need to define an array of size $10^9$ which is infeasible because it is too big.

Despite this being infeasible, let's make sure we understand the main idea here. Suppose we are a company and we want to keep track of employee records. We use social insurance number as a key for accessing a record. Then we could use a (unfeasibly large) array whose entries would be of type `Employee`. That is, you would use someone's social insurance number to index directly into the array, and that indexed array slot would hold a reference to an `Employee` object associated with that social insurance number. This would only retrieve a record if there were an `Employee` with that social insurance number; otherwise the reference would be null and the `get` call would return null.

For the rest of today, we consider the case that the keys map to a set of integers, without requiring these integers to be a small range. These integers are called *hash codes.* Next lecture we will return to the problem of mapping to a small range of integers, and we will treat these integers as an index into an array. This is the method of hash maps or hash tables.

## Java `hashCode()`

### Object.hashCode()

Recall way back when we discussed the Java Object class and its methods. We mentioned the hashCode() method but didn't say much about it, other than that we can think of it as a unique id for an object.[16] The expression "`obj1 == obj2`" means the same thing as "`obj1.hashCode() == obj2.hashCode()`". The statement can be either true or false, depending on whether the variables `obj1` and `obj2` reference the same object or not.

### String.hashCode()

Some classes override this default `hashCode()` method. To understand how `hashCode()` is defined for the `String` class, let's first as a warmup consider a simple map from `String` to positive integers. Suppose $s$ is a string which consists of characters $s[0]s[1]...s[s.length - 1]$. Consider the function

$$h(s) = \sum_{i=0}^{s.length-1} s[i]$$

which is just the sum of the codes of the individual characters. Notice that two strings that consist of the same letters but in different order would have the same $h()$ value. For example, "eat", "ate", "tea" all would have the same code. Since the codes of `a, e, t` are 97, 101, and 116, the code of each of these strings would be 97+101+116.

In Java, the `String.hashCode()` method is defined[17] :

$$h(s) = \sum_{i=0}^{s.length-1} s[i] \; x^{s.length-i-1}$$

where $x = 31$. So, for example,

$$h("eat") = 101 * 31^2 + 97 * 31 + 116$$

[16]When a Java programming is running, every object is located somewhere in the memory of the Java Virtual Machine (JVM). This location is called an *address.* In particular, each Java object has a unique 24 bit number associated with it which by default is the number returned when the object calls `hashCode()` method. (What exactly this 24 bit number means may depend on the implementation of the JVM. We will just assume that the number is the object's (starting) address in the JVM.) Since different objects must be non-overlapping in memory, it follows that they have different addresses.

[17]You might wonder why Java uses the value $x = 31$. Why not some other value? There are explanations given on the website stackoverflow, but I am not going to repeat them here. Other values would work fine too.

and

$$h("ate") = 97 * 31^2 + 116 * 31 + 101$$

Thus, here we have an example of how strings that have the same letters do not have the same hashCode.

What about if two strings have the same hashcode? Can we infer that the two strings are equal? No, we cannot. Two different strings might have the same hashcode. See the Exercises.

## ASIDE: Horner's rule

Suppose you wish to evaluate a polynomial

$$h(s) = \sum_{k=0}^{N} a_k \ x^k$$

It should be obvious that you don't want to separately calculate $x^2, x^3, x^4, ..., x^N$ since there would be redundancies in doing so. We only should use $O(N)$ multiplications, not $O(N^2)$. Horner's Rule describes how to do so.

The following example gives the idea of Horner's rule for the case of hashcodes for a `String` object with four characters:

$$s[0] * 31^3 + s[1] * 31^2 + s[2] * 31 + s[3] = ((s[0] * 31^1 + s[1]) * 31 + s[2]) * 31 + s[3]$$

For a `String` object of arbitary length, Horner's rule is the following algorithm :

```
h = 0
for  (i = 0;  i < s.length;  i++)
    h = h*31 +  s[i]
```

## Hashing

Recall the scenario from last lecture: suppose we have a map, that is, a set of ordered pairs $\{(k, v)\}$. We want a data structure such that, given a key $k$, we can quickly access the associated value $v$. If the keys were integers in a small range, say $0, 1, \ldots, m-1$, then we could just use an array of size $m$ and the keys could indices into the array. The locations $k$ in the array would correspond to the keys in the map and the slots in the array would hold references to the corresponding values $v$.

In most situations, the keys are not integers in some small range, but rather they are in a large range, or the keys are not integers at all – they may be strings, or something else. In the more general case, we need to define a function – called a *hash function* – that maps the keys to the range $0, 1, \ldots, m-1$. Then, we will put the two maps together: the hash function maps from keys to a small range of integer values, and from this small range of integer values to the corresponding values $v$. Let's now say more about how we make hash functions, and then we'll talk about a data structures that uses them called a "hash map" or "hash table".

## Hash function: hash code followed by compression

Given a space of keys $K$, a *hash function* is a mapping:

$$h : K \rightarrow \{0, 1, 2, m-1\}$$

where $m$ is some positive integer. That is, for each key $k \in K$, the hash function specifies some integer $h(k)$ between 0 and $m-1$. The $h(k)$ values in $0, \ldots, m-1$ are called *hash values.*

It is very common to design hash functions by writing them as a composition of two maps. The first map takes keys $K$ to a large set of integers. The second map takes the large set of integers to a small set of integers $\{0, 1, \ldots, m-1\}$. The first map is called a *hash code*, and the integer chosen for a key $k$ is called the *hash code* for that key. The second mapping is called *compression.* Compression maps the hash codes to *hash values*, namely in $\{0, 1, \ldots, m-1\}$.

The Java `hashCode()` method returns an `int`, and `int` values can be either positive or negative, specifically, they are in $\{-2^{31}, \ldots -1, 0, 1, \ldots 2^{31}-1\}$. The compression function that is used in Java for hashmaps is

$$\mid \texttt{hashCode}() \mid \bmod m,$$

i.e. gives a number in $\{0, 1, \ldots, m-1\}$, as desired. I will discuss the java `HashMap` class a bit later. A hash function is two functions:

$$\text{hash function } \; h \; = \; \text{compression} \; \circ \; \text{hash code}$$

where $\circ$ denotes the composition of two functions, and

$$\text{hash code} \quad : \quad \text{keys K} \rightarrow \text{integers}$$

$$\text{compression} \quad : \quad \text{integers} \rightarrow \{0, .., \mathrm{m}-1\}$$

and so

$$h : \text{keys } K \rightarrow \{0, 1, \ldots, m-1\},$$

i.e. the set of hash values is $\{0, 1, \ldots, m-1\}$.

Note that a hash function is itself a map. So there are a several different maps being used here, and in particular the word ”value” is being used in two different ways. The values $v \in \texttt{V}$ of the map we are ultimately trying to represent are not the same thing as the “hash values” $h(k)$ which are integers in $0, 1, \ldots, m-1$. Values $v \in \texttt{V}$ might be `Employee` records, or entries in an telephone book, for example, whereas hash values are indices in $\{0, 1, \ldots, m-1\}$.

## Collisions

To represent the $(k, v)$ pairs in our map, we use an array. The number of slots $m$ in the array is typically bigger than the number of $(k, v)$ pairs in the map. However, it can happen that two keys $k_1$ and $k_2$ map to the same hash value. This is called a *collision*. There are two ways a collision can happen: two keys might have the same hash code, or two keys might have different hash codes,but these two different hash codes map (compress) to the same hash value.

To allow for collisions, we use a linked list of pairs $(k, v)$ at each slot of our array. These linked lists are called *buckets*.[18] Note that we need to store a linked list of pairs $(k, v)$, not just values $v$. The reason is that when use a key $k$ to try to access a value $v$, we need to know which of the $v$'s stored in a bucket corresponds to which $k$. We use the hash function to map the key $k$ to a location (bucket) in the array; we then try to find the corresponding value $v$ in the list. We examine each entry $(k, v)$ in the linked list and check if the search key equals the key in that entry.

A good hash function will distribute the key/value pairs over the buckets such that, if possible, there is at most one pair per bucket. This is only possible if the number of entries in the hash map is less than or equal to the number of buckets. We define the *load factor* of a hash map to be the number of entries $(k, v)$ in the map divided by the number of slots in the array $(m)$. A load factor that is slightly less than one is recommended for good performance.

Having a load factor less than one does not guarentee good performance, however. In the worst case, all the keys in the collection hash to the same bucket, and then we would have one linked list only and access is $O(n)$ where $n$ is the number of entries. This is undesirable obviously. To avoid having such long lists, we want to choose a hash function so that there are few, if any, collisions.

The word *hash* means to “chop/mix up”.[19] We are free to choose whatever hash function we want and we are free to choose the size $m$ of the array. So, it isn't difficult to choose a hash function that performs well in practice, that is, a hash function that keeps the list lengths short. In this sense, one typically regards hash maps as giving $O(1)$ access. (To prove that the performance of hash map really is this good, one needs to do some math that is beyond COMP 250.)

As an example of a good versus bad hash function, consider McGill student ID's which are 9 digits. Many start with the digits 260. If we were to use an array of size say $m = 100,000$, then it would be bad to use the first five digits as the hash function since most students IDs would map to one of those two buckets. Instead, using the last five digits of the ID would be better.

### Java `HashMap`

In Java, the `HashMap<K,V>` class implements the hash map that we have been describing. The `hashCode()` method for the key class `K` is composed with the “mod $m$” (and absolute value) com-

[18]Storing a linked list of $(k, v)$ pairs in each hash bucket is called *chaining*.

[19]It should not be confused with the # symbol which is often the “hash” symbol i.e. hash tag on Twitter, or with formerly illicit drugs.

pression function where $m$ is the capacity of an underlying array, and it is an array of linked lists. The linked lists hold `(K,V)` pairs. Have a look at the Java API to see some of the methods and their signatures: `put` , `get`, `remove`, `size`, `containsKey`, `containsValue`, and think of how these might be implemented.

In Java, the default maximum load factor for the hash map is than 0.75 and there is a default array capacity as well. The `HashMap` constructor allows you specify the initial capacity of the array, and you can also specify the maximum load factor. You use `put()` to add a new entry to a hash map. When this makes the load factor go above 0.75 (or the value you specify), then a new array is generated, namely there is larger number $m$ of slots and the (key,value) pairs are remapped to the new underlying hash map. This happens "under the hood", similar to what happens with `ArrayList` when the underlying array fills up and so the elements needs to be remapped to a larger underlying array.

## `hashCode()` and `equals()`

For any class, the `hashcode()` and `equals()` method should be related as follows: if `k1.equals(k2)` is true, then `k1.hashCode() == k2.hashCode()` should be true. The reason is that when we call put, get, or remove, or other methods such as contains, we need to check if the key is already in the map. But this should be done by checking if there is a key in the map that is equal to the key we are searching for. If our key class overrides the Object class'es equals method, then we should use the key's equals method. (Otherwise, what would be the point of the key having its own equals method?)

What about the converse of the above rule? If `k1.hashCode() == k2.hashCode()` is true, then should we *require* that `k1.equals(k2)` returns true ? No, this would be too strong. For example, we often use strings for the keys of a map. Two different string keys in a map might have the same hash code. This would create a collision. We would prefer not to have collisions, but we do allow for them.

Bottom line: if you write a class which uses the `hashCode()` method and you want to override either the `Object` class'es `hashCode()` or `equals()` methods, then you should ensure this: if `k1.equals(k2)` returns true or false, then `k1.hashCode() == k2.hashCode()` should return true of false, respectively. This may require that you override *both* of them.

## Java `HashSet<K>` class

Java also has a `HashSet<T>` class. This is similar to a HashMap except that it only holds objects of type `<T>`, not key/value pairs. It uses the `hashCode()` method of the type `T`. Why would this class be useful? Sometimes you want to keep track of a set of elements and you just want to ask questions such as, "does some element e belong to my set, or not?" You can add elements to a set, remove elements from a set, compute intersections of sets or unions of sets." What is nice about the `HashSet` class is that it give you quick access to elements. Unlike a list, which requires $O(n)$ operations to check if an element is present in the worst case, a hash set allows you to check in time $O(1)$ – assuming a good hash function.

`HashSet<T>` uses an underlying array, just like hashmap. For the hash function it uses the `hashCode()` method for type `T`, together with the absolute value and "mod $m$" compression function where again $m$ is the capacity of the underlying array. Each bucket in the array holds a list of objects of types `T`. Think of a `HashSet<T>` as a `HashMap<T,V>` without the values `V`!

`HashSet` does not implement the `List` interface, but rather it implements the `Set` interface. (Check it out.) For example, suppose you simply want to keep track of a *set* of `Strings`. Suppose you don't need the strings to be ordered, i.e. you won't be asking for the $i$-th string in a lexicographic order (as you might for a sorted list), and you won't be asking for the string that was added last (as you would for a stack), and you won't be asking for the string that was added first (as you would for a queue). Suppose you simply want to keep track of a set of strings, and be able to ask questions like "is string $x$ a member of this set?" or perform operations like adding a string to the set or deleting a string from the set. In this case, using a hash set would work great since it would allow you to answer these questions in time $O(1)$. The cost of doing so, of course, is that you are limited in the operations you can do.

## Cryptographic hashing

Hash functions come up in many problems in computer science, not just in hash table data structures. Another common use is in password authentication. For example, when you log in to a website, you typically provide a username and password. On a banking site, you might provide your ATM bank card number or your credit card number, again along with a password. What happens when you do so?

You might think it works like this. The web server has a file that lists all the user names and corresponding passwords (or perhaps a hash map of key-value pairs, namely usernames and passwords, respectively). When you log in, the web server takes your user name, goes to the file, retrieves the password stored there for that username, and compares it with the password that you entered. This is *not* how it works however! The reason is that this method would not be secure. If someone (an employee, or an external hacker) were to break in an steal the file then that person would have access to all the passwords and would be able to access everyone's data.

Instead, the way it typically works is that the webserver stores the user names and *hashed* passwords. Then, when a user logs in, the web server takes the password that the user enters, hashes it (that is, applies the hash function), throws away the password entered by the user, and compares the hashed password to the hashed password that is stored in the file.

Why is then any better? The answer is that if a hacker could access the user names and hashed passwords then this itself would not be good enough to log in, since the process of logging in requires entering a password, not a hashed password.

Now you might wonder if it is possible to compute an *inverse hash map*: given a hashed password, can we compute the original password or perhaps some other password that is hashmapped to the given hashed password? If such a computation were feasible, then this inverse hashmap could be used again to hack into a user's account.

The answer is basically no. "Cryptographic" hashing functions are designed so that such that it *not* feasible to compute the inverse hash map. These hashing functions have a mixing property that two strings that differ only slightly will map to completely different hash values, and given a hash value, one can say almost nothing about a keyword that could produce that hash function.

A few final observations. First, a cryptographic hashing function does not need to be secret. Indeed there are standard cryptographic hashing functions. One example is MD5 `http://www.`

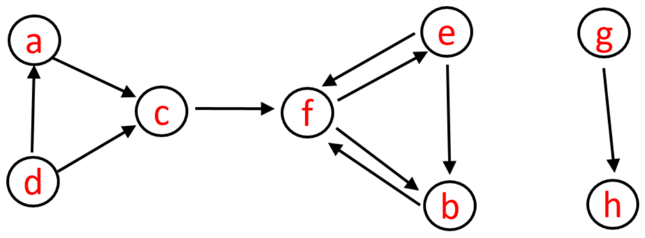
`miraclesalad.com/webtools/md5.php`. This maps any string to a sequence of 128 bits.[20]

Second, cryptographic hashing is not the same as encryption. With the latter, one tries to encode a string so that it is not possible for someone who is not allowed to know the string to decode the coded string and have the original one again. However, it *should* be possible for someone who *is* allowed to know the original string to be able to decode the encrypted string to get the original message back. So, the main difference is that with encryption it *is* possible to invert the "hash" function. (It isn't called hashing, when one is doing encryption, but the idea is similar.) You will learn a bit about encryption/decryption if you take MATH 240 Discrete Structures, and you will learn a lot about it if you take COMP 547 Cryptography and Data Security.

[20]If you check out that link, you'll see that the hash values are represented not as 128 bits there, but rather as 32 *hexadecimal* digits.

## Graphs

You are familiar with data structures such as arrays and linked lists (linear), and trees and hashtables (non-linear). We next consider a more general non-linear data structure, known as a graph. Here is an example.



Like in many previous data structures, a graph contains a set nodes. Each node has a reference to other nodes. For graphs, a reference from one node to another is called an *edge*.

In a linked list, there are "edges" from one node to the "next" and/or "previous" node. In a rooted tree, there are edges from children nodes to parent nodes or vice-versa. In a graph, there is no notion of a "next" or "prev" as in a list, or a child or parent as in a tree. Every node in a graph can potentially have an edge to every other node. We will discuss data structures for graphs below.

Graphs have been studied and used for many years, and some of the basic results go back a few hundred years, to mathematicians like Euler. Mathematically, a graph consists of a set $V$ called "vertices" and a set of edges $E \subseteq V \times V$. The edges $E$ in a graph is a set of *ordered* pairs of vertices.

When the ordering of the vertices in an edge is important, we say that we have a *directed graph*. One can also define graphs in which the edges do not have "arrows", that is, each edge is a pair of vertices but we don't care about the order. This is called an *undirected* graph, and in this case we would draw the edges as line segments with no arrows. We will deal with directed graphs only. (But undirected graphs are also important, and you'll see lots of examples in COMP 251.)

There are many examples of graphs in the world. In a flight network, $V$ might be a set of airports and $E$ might be direct flights between airports. In the world wide web, $V$ might be a set of html documents and $E$ would be the URLs (links) between documents. In a running Java program, $V$ might be a set of objects and $E$ would be a set of references, namely when an object has a field (reference variable) that references another object.

## Terminology

Here is some basic graph terminology that you need to know. Please see the slides for examples.

- *outgoing edges from $v$* - the set of edges of the form $(v, w) \in E$ where $w \in V$
- *incoming edges to $v$* - the set of edges of the form $(w, v) \in E$ where $w \in V$
- *in-degree of $v$* - the number incoming edges to $v$
- *out-degree of $v$* - the number outgoing edges from $v$

- *path* - a sequence of vertices $(v_1, \ldots, v_m)$ where $(v_i, v_{i+1}) \in E$ for each $i$. The length of a path in a graph is the number of edges in the path (not the number of vertices). The definition of path is essentially the same for graphs as it was for trees.
- *cycle* - a path such that the first vertex is the same as the last vertex

  If there were an edge $(v, v)$, then this would be considered as a cycle. Such edges are called loops.
- *directed acyclic graph* (DAG) – a directed graph that has no cycles. Such graphs are used to capture dependencies between objects or events. For example, the graph of prerequisite relationships in McGill courses is directed and acyclic.
- *weighted graph* – a graph that has a number (weight) associated with each edge; for example, in a flight network where the vertices are airports, the weight might be the time it takes to fly between two airports

In the slides, I highlight a few important graph problems that you will see in COMP 251.

## ASIDE: web graph and Google's pagerank

An important example of a graph is the world wide web. As mentioned earlier, the vertices are web pages and the edges are the hyperlinks from one web page to another.

Let's consider a few basic ideas of how Google search works. The idea goes back to the late 1990's when the co-founders Sergey Brin and Larry Page were graduate students. They had some very clever new ideas for how to improve the search for web pages on the world wide web. The idea is described in a paper draft. Here I will present one of the very basic ideas.

When you search for some set of terms on google, you want to find the most important web pages for those terms. One of the core early ideas was to define whether a web page $v$ is (relatively) important or not. Here are two factors:

- Which set of web pages $\{w\}$ point to (have a hyperlink to) this web page $v$, and how important are each of *those* web pages $w$ ?
- For each such web page $w$ that points to $v$, how many web pages does $w$ point to. From $v$'s perspective, $w$'s link to $v$ should contribute less if $w$ points to lots of other web pages.

Let $N_{out}(w)$ be the out-degree of vertex $w$. Then the *page rank* (or importance) of a web page $v$ is defined roughly as follows[21]

$$R(v) = \sum_{w:(w,v) \textit{ is an incoming edge to } v} \frac{R(w)}{N_{out}(w)}.$$

The idea for computing $R(v)$ is to initialize it to the value 1 for all $v$. Then, iteratively update $R(v)$ by plugging the current values of $R()$ into the right hand side.

[21] Even the basic formula is a bit more complicated than this, but hopefully you get the basic idea.

## Data structures for graphs

### Adjacency List

A graph is a generalization of a tree. Each node in a tree has a list of children. Similarly, each graph vertex $v$ has a list of other vertices $w$ that are adjacent to it. We call this an *adjacency list*, namely for each vertex $v \in V$, we represent a list of vertices $w$ such that $(v, w) \in E$. For example, here is the adjacency lists for the graph on page 1.

```
a  -  c
b  -  f
c  -  f
d  -  a,c
e  -  b,f
f  -  b,e
g  -  h
h  -
```

I have represented vertices in alphabetic order, and will do so in most examples.

Java does not have a `Graph` class since there are too many different types of graphs and no one standard way works best. So one needs to implement one's own `Graph` class. A very basic `Graph` class might be as simple as this:

```
class Graph<T>{

   class Vertex<T> {
      LinkedList<Vertex<T>>  adjList;
      T                      element;
   }

   :    //  various methods
}
```

However, it is common to have other vertex attributes and also to have attributes for the edges, in particular, edge weights. So a more common graph would be like this:

```
class Graph<T>{

   class Vertex<T> {
       LinkedList<Edge<T>>  adjList;
       T                    element;
   }

   class Edge<T> {
      Vertex<T>            endVertex;
      double               weight;
      :
}
```

Now the adjacency list for a vertex is a list of `Edge` objects and each edge is represented only by the end vertex of the edge. The start vertex of each edge does not need to be represented explicitly because it is the vertex that has the edge in its adjacency list `adjList`.

An important difference between rooted trees and graphs is that rooted trees have a special node (the root) where most methods begin. However, for graphs, we may wish to access any node. For this, we need a map.

We will have a label (key) for each of the vertices, and we will use the key to access the vertices by using a hash map. The key might be a string. For example, in a graph network of airports, YUL might be the key for Trudeau airpot, LAX for the main Los Angeles airport, etc.

```
class  Graph<T>{
     HashMap< String, Vertex<T> >    vertexMap;
       :
        //  Vertex and Edge inner classes as above
}
```

## Adjacency Matrix

A different data structure for representing the edges in a graph is an *adjacency matrix* which is a $|V| \times |V|$ array of booleans, where $|V|$ is the number of elements in set $V$ i.e. the number of vertices. The value 1 at entry $(v1, v2)$ in the array indicates that $(v1, v2)$ is an edge, that is, $(v1, v2) \in E$, and the value 0 indicates that $(v1, v2) \notin E$.

The adjacency matrix for the graph from earlier is shown below.

```
     abcdefgh
  a  00100000
  b  00000100
  c  00000100
  d  10100000
  e  01000100
  f  01001000
  g  00000001
  h  00000000
```

Note that in this example, the diagonals are all 0's, meaning that there are no edges of the form $(v, v)$. But graphs can have such edges (called *loops*). An example is given in the slides.

For a weighted graph, the entries would be numbers rather than boolean (0,1) values. Again, a 0 would indicate that there is no edge, as opposed to an edge with weight 0.

See the Exercises for some examples of when you would you an adjacency list versus adjacency matrix.

## Graph traversal

One problem we often need to solve when working with graphs is to decide if there is a sequence of edges (a path) from one vertex to another. If there are multiple paths then the problem is to find the "best" one. There are various versions of this problem. One familiar one to you is Google Maps which finds the shortest path from one location to another. You'll cover a shortest path algorithms in COMP 251.

Today we will consider the problem of finding the set of all vertices that can be reached from a given vertex $v$, or equivalently, the set of all vertices $w$ for which there is a path from $v$ to $w$.

### Depth First Traversal

Recall the depth first traversal algorithm for trees.

```
depthfirst_Tree(root){
    visit root                      // preorder
    for each child of root
        depthfirst_Tree(child)
}
```

This algorithm generalizes to graphs as follows.

```
depthfirst_Graph(v){
   v.visited = true
   for each w such that (v,w) is in E
      if !w.visited                          //  avoids cycles
         depthfirst_Graph(w)
}
```

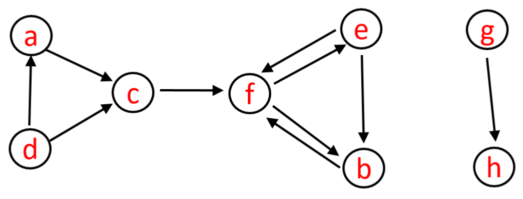
I call this algorithm a "traversal" but in fact it only reaches the nodes in the graph for which there is a path from the input vertex. This algorithm is sometimes called "depth first search" since we are searching for all vertices that can be reached from the input vertex.

Note that before running this algorithm, we need to set the `visited` field to false for all vertices in the graph. To do so, we need to access *all* vertices in the graph. This is a different kind of traversal, which is independent of the edges in the graph. For example, if you were to use a linked list to represent all the vertices in the graph, then you would traverse this linked list and set the `visited` field to false, before you called the above traversal algorithm. If you were using a hash map to represent the vertices in the graph, you would need to go through all buckets of the hash map by iterating through the hash map array entries and following the linked list stored at each entry. You would set the `visited` field to false on each vertex (value) in each bucket.

### Example

Let's run the above preorder depth first traversal algorithm on the graph shown below. Also shown is the *call stack* and how it evolves over time, and the *call tree*. (A node in the call tree represents one "call" of the `depthfirst_Graph` method. Two nodes in the call tree have a parent-child relationship if the parent node calls the child node.)

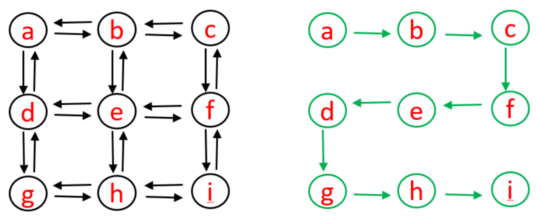
Note that the call stack is actually constructed when you run a program that implements this recursive algorithm, whereas the call tree is not constructed. The call tree is just a way of thinking about the recursion.



```
      CALL STACK                CALL TREE
        b   e                        a
      f f f f f                      |
    c c c c c c c                    c
  a a a a a a a a a                  |
                                     f
-------------------> time           / \
                                   b   e
```

Notice that nodes `d`, `g`, `h` are not visited.

Here is another example. The graph is on the left and the call tree is on the right. Cover up the tree on the right with your hand, and then do the depth first search for the tree, starting from (a). Assume that within each adjacency list, the elements are lexicographically (alphabetically) ordered.



## Non-recursive depth first traversal

We do not need recursion to do a depth first traversal. We can do depth first traversal using a stack. Our algorithm here generalizes the non-recursive tree traversal algorithm that used a stack.

The tree traversal algorithm using a stack went like this:

```
treeTraversalUsingStack(root){
  s.push(root)
  while !s.isEmpty(){
    cur = s.pop()
    visit cur
    for each child of cur
       s.push(child)
    }
}
```

Here we visited each node after popping it from the stack. Let's rewrite this slightly so that we visit each node *before* pushing it onto the stack. (It is basically at the same time as pushing on the stack, since the order of the two instructions within the for loop below doesn't matter.)

```
treeTraversalUsingStack(root){
   visit root
   s.push(root)
   while !s.isEmpty(){
      cur = s.pop()
      for each child of cur{
         visit child
         s.push(child)
      }
   }
}
```

Now let's generalize this to graphs. In the tree case, we pushed the children of a node onto the stack. In the graph case, we will push the adjacent vertices (nodes) onto the stack. *We only do so, however, if the adjacent vertex has not yet been visited.* The reason is that the graph may contain a cycle and we want to ensure that a vertex does not get pushed onto the stack more than once. See Exercises 12 (graphs) Question 6.

```
graphTraversalUsingStack(v){
    initialize empty stack s
    v.visited = true
    s.push(v)
    while (s is not empty) {
      u =  s.pop()
      for each w in u.adjList{
         if (!w.visited){
            w.visited = true
            s.push(w)
         }
 }
```

Note that this still is a preorder traversal. We visit a vertex before we visit any of the children vertices.

## Breadth first traversal

Like depth first traversal, breadth first traversal finds all vertices that can be reached from a given vertex $v$. However, breadth first traversal visits all vertices that are one edge away, before it visits any vertices are two vertices away, etc. We have already seen breadth first traversal in trees, also known as level order traversal. Breadth traversal in graphs is more general: the levels correspond to vertices that are a certain distance away (in terms of number of edges i.e. path length) from the starting vertex.

Since we are working with a graph rather than a tree, we visit the nodes before enqueueing them. When we reach a node, we only visit and enqueue it if it hadn't yet been visited.

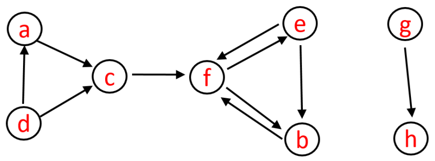
```
breathFirst(u){
  initialize empty queue q
  u.visited = true
  q.enqueue(u)
  while ( !q.empty) {
    v = dequeue()
    for each w in adjList(v)
      if !w.visited{
        w.visited = true
        enqueue(w)
      }
}
```

Since enqueue a vertex only if we have not yet visited it, we cannot enqueue a vertex more than once. Moreover, all vertices that are enqueued are eventually dequeued, since the algorithm doesn't terminate until the queue is empty.

Take the same example graphs as before, again starting with vertex a. Below we show the queue `q` as it evolves over time, namely we show the queue at the end of each pass through the `while` loop.

Since this is not a recursive function, we don't have a "call tree". But we can still define a tree. Each time we visit a vertex – *i.e.* `w in adjList(v)` and we set `w.visited` to `true` – we get a edge `(v, w)` in the tree. We can think of the `w` vertex as a child of the `v` vertex, which is why we are calling this a tree. Indeed we have a rooted tree since we are starting the tree at some initial vertex that we are searching from.

Note how the queue evolves over time, and the order in which the nodes are visited. The tree in this case happens to be the same as the tree defined by the depth first traversal.



```
   QUEUE q    TREE
(snapshots)
   a           a
   c           |
   f           c                 order visited = acfbe
   be          |
   e           f
             /   \
            b     e
```

**Another Example**

Here is another example, which better illustrates the difference between `depthFirst` and `breadFirst`. To simplify the drawing, I suppose here that the graph is undirected. In the slides, I did a directed version of this.

```
  GRAPH      ADJACENCY LIST    depth first       depth first         breadth first
                               (recursive)         (stack)             (queue)

a - b - c    a - (b,d)         a - b - c         a - b   c            a - b - c
|   |   |    b - (a,c,e)               |         |       |            |   |   |
d - e - f    c - (b,f)         d - e - f         d - e   f            d   e   f
|   |   |    d - (a,e,g)       |                 |       |            |   |   |
g - h - i    e - (b,d,f,h)     g - h - i         g - h - i            g   h   i
             f - (c,e,i)
             g - (d,h)
             h - (e,g,i)
             i - (f,h)

            order visited:     abcfedghi         abdeghifc            abdcegfhi
```

I claim that this algorithm first visits all nodes reachable from the start node via a path length 1, then visits all nodes reachable from the start node via a path length 2, etc, then visits all nodes reachable from the start node via a path length $k$, etc. You can prove such a claim by induction. Can you do the proof?

## Recurrences

We have seen many algorithms thus far. For each one we have tried to express how many basic operations are required as a function of some parameter $n$ which is typically the size of the input e.g. the number of elements in a list.

For algorithms that involve `for` loops, we can often write the number of operations of the loop component as a power of $n$, which corresponds to the number of nested loops. For example, if we have two nested `for` loops, each of which run $n$ times, then these loops take time proportional to $n^2$. The quadratic sorting algorithms and the grade school multiplication algorithm are a few examples.

For recursive algorithms, it is less obvious how to express the number of operations as a function of the size of the input. We have given some examples of recursive algorithms in the lectures, and argued what there $O(\ )$ behavior is. But this was informal only. For these examples and others, we would like to express in a more general and formal way how the time (or number of steps) it takes to solve the problem depends on $n$. In each case, we express a function $t(n)$ in terms of $t(...)$ where the argument depends on $n$ but it is a value smaller than $n$. Such a recursive definition of $t(n)$ is called a *recurrence relation* or just a *recurrence*.

### Example 1: reversing a linked list

Let $t(n)$ be the time it takes to reverse a list with $n$ elements. Think of how this is done in the case of a linked list. You remove the first element of the list. Then, you take the remaining $n-1$ element list and recursively reverse them. Then you add the element that you removed at the end of the reversed list.

Each recursive call reduces the problem from size $n$ to size $n-1$. This suggests a relationship:

$$t(n) = c + t(n-1)$$

where the constant $c$ is the time it takes in total to remove the first element from a list plus the time it takes to add that same element to the end of a list. We are not saying what $c$ is. All that matters is that it is constant: it doesn't depend on $n$. (Note that I am making an assumption here that the first element can be removed in constant time. If we are using an array list, then this assumption would be incorrect.)

To obtain an expression for $t(n)$ that is not recursive, we repeatedly substitute on the right side, as follows:

$$\begin{aligned}
t(n) &= c + t(n-1) \\
&= c + c + t(n-2) \\
&= c + c + c + t(n-3) \\
&= \dots \\
&= c(n-1) + t(1).
\end{aligned}$$

This method is called *backwards substitution.* Note $t(1)$ is the base case of the recursion and done in constant time.

Informally, we say that $t(n)$ is $O(n)$ since the time it takes is roughly proportional to $n$. A few lectures from now, we'll say more formally what we mean by $O(\ )$.

One often writes such a recurrence in a slightly simpler way ($c = 1$):

$$t(n) = 1 + t(n-1) \ .$$

The idea is that since the constant $c$ has no "units" anyhow, its meaning is unspecified except for the fact that it is constant, so we just treat it as a unit (1) number of instructions.

**Example 2: sorting a list by finding the minimum element**

Consider a recursive algorithm for sorting which finds the smallest element in a list and removes it, then sorts the remaining $n-1$ elements, and finally adds the removed element to the front of the list. We express the time taken, using the recurrence:

$$t(n) = c\ n + t(n-1) \ .$$

We could write the recurrence more precisely as

$$t(n) = c_1 + c_2\ n + t(n-1)$$

since in each recursive call there is some constant $c_1$ amount of work, plus some amount of work $c_1 n$ that depends linearly on the size $n$ of the list in that call. But let's keep it simple and consider just the first recurrence. Using back substitution:

$$\begin{aligned} t(n) &= c\ n + t(n-1) \\ &= c\ n + c \cdot (n-1) + t(n-2) \\ &= \dots \\ &= c\ \{\ n + (n-1) + (n-2)\ + \dots + (n-k)\} + t(n-k-1) \\ &= c\ \{\ n + (n-1) + (n-2)\ + \dots + 2 + 1\} + t(0) \\ &= \frac{cn(n+1)}{2} + t(0) \end{aligned}$$

Informally we will say that is $O(n^2)$ since the largest term here that depends on $n$ is $n^2$. Next week we will give a more formal definition of big O.

**Example 3: Tower of Hanoi**

Recall the Tower of Hanoi problem. Let $t(n)$ be the number of disk moves. The recurrence relation is:

$$t(n) = 1 + 2\ t(n-1).$$

There is the single disk move that is done in each call. One also needs to solve the problem twice for $n-1$ disks.

Proceeding by back substitution, we get

$$\begin{aligned}
t(n) &= 1 + 2\ t(n-1) \\
&= 1 + 2(1 + 2\ t(n-2)) \\
&= (1+2) + 4\ t(n-2) \\
&= (1+2) + 4\ (1 + 2\ t(n-3)) \\
&= (1+2+4)\ +\ 8\ t(n-3) \\
&= ... \\
&= (1+2+4+8+\cdots+2^{k-1}) + 2^k\ \ t(n-k) \\
&= (1+2+4+8+\cdots+2^{n-2}) + 2^{n-1}\ t(1) \\
&= (2^{n-1}-1) + 2^{n-1}\ t(1) \\
&= 2^n - 1,\ \text{since}\ \ t(1) = 1
\end{aligned}$$

where we have used the familiar geometric series

$$\sum_{i=0}^{m-1} x^i = \frac{x^m - 1}{x - 1}$$

for the case that $x = 2$.

Verify the above expression for yourself by considering $n = 1, 2, 3, ...$

**Example 4: binary search**

Recall the recursive binary search algorithm. We assume we have an ordered list of elements, and we would like to find a particular element `e` in the list. The algorithm computes the mid index and compares the element `e` to the element at that mid index. The algorithm then recursively calls itself, searching for `e` either in the lower or upper half of the list. Since the recursive call is on a list that is only half the size, we can express the time using the recurrence:

$$t(n) = c + t(\frac{n}{2})\ .$$

For the purposes of solving the recurrence, we suppose that $n$ is a power of 2. Then,

$$\begin{aligned}
t(n) &= c + t(\frac{n}{2}) \\
&= c + c + t(\frac{n}{4}) \\
&= c + c + \cdots + t(\frac{n}{2^k}) \\
&= c + c + \cdots + c + t(\frac{n}{n})\ ,\text{where } 2^k = n\ \ i.e.\ k = \log_2 n,\ \text{ and we have }\ \log_2\ c's \\
&= c\log_2 n + t(1)
\end{aligned}$$

So we say that binary search is $O(\log_2 n)$ since the largest term that depends on $n$ is $\log_2 n$.

Today we examine the recurrences for mergesort and quicksort.

## Mergesort

Recall the mergesort algorithm: we divide the list of things to be sorted into two approximately equal size sublists, sort each of them, and then merge the result. Merging two sorted lists of size $\frac{n}{2}$ takes time proportional to $n$, since the merging requires iterating through the elements of each list. If $n$ is even, then there are $\frac{n}{2} + \frac{n}{2} = n$ elements in the two lists. If $n$ is odd then one of the lists has size one greater than the other, but there are still $n$ steps to the merge. Let's assume that $n$ is a power of 2. This keeps the math simpler since we don't have to deal with the case that the two sublists are not exactly the same length. In this case, the recurrence relation for mergesort is:

$$t(n) = cn + 2t(\frac{n}{2}).$$

[ASIDE: If we were to consider a general $n$, then the correct way to write the recurrence would be:

$$t(n) = t(\ \lfloor\frac{n}{2}\rfloor\ ) + t(\ \lceil\frac{n}{2}\rceil\ ) + cn$$

where the $\lfloor\frac{n}{2}\rfloor$ means $floor(n/2)$ and $\lceil\frac{n}{2}\rceil$ means $ceiling(n/2)$, or "round down" and "round up", respectively. That is, rather than treating $n/2$ as an integer division and ignoring the remainder (rounding down always), we would be treating it as $n/2.0$ and either rounding up or down. In the lecture, I gave the example of $n = 13$ so $\lfloor\frac{n}{2}\rfloor = 6$ and $\lceil\frac{n}{2}\rceil = 7$. In COMP 250, we don't concern ourselves with this level of detail since nothing particualrly interesting happens in the general case, and we would just be getting bogged down with notation. The interesting aspect of mergesort is most simply expressed by considering the case that $n$ is a power of 2. ]

Let's solve the mergesort recurrent using backwards substitution:

$$\begin{aligned}
t(n) &= c\,n + 2\ t(\frac{n}{2}) \\
&= c\,n + 2\ (\ c\frac{n}{2} +\ 2t(\frac{n}{4})) \\
&= c\,n + c\,n + 4\ t(\frac{n}{4}) \\
&= c\,n + c\,n + 4\ (c\frac{n}{4}\ +\ 2\ t(\ \frac{n}{8}\ )) \\
&= c\,n +\ c\,n +\ c\,n + 8\ t(\frac{n}{8}) \\
&= c\,n\,k\ +\ 2^k\ t(\frac{n}{2^k}) \\
&= c\,n\log_2 n +\ n\ t(1), \quad \text{when } n = 2^k
\end{aligned}$$

which is $O(n\ \log_2 n)$ since the dominant term is $n\ \log_2 n$.

What is the intuition for why mergesort is $O(n \log_2 n)$ ? Think of the merge phases. The list of size $n$ is ultimately partitioned down into $n$ lists of size 1. If $n$ is a power of 2, then these $n$ lists are merged into $\frac{n}{2}$ lists of size 2, which are merged into $\frac{n}{4}$ lists of size 4, etc. So there are $\log_2 n$ "levels" of merging, and each require moving (as part of merging) all $n$ elements which requires $O(n)$ work.

See the lecture slides for a Java implementation of the `mergesort` and `merge` methods. The `mergesort` method takes an input array and copies the elements into two smaller arrays. The `merge` method then takes these smaller arrays and merges their elements back into the takes an input array and copies the elements into the original big array.

### How many recursive calls to `mergesort`?

Suppose the $n$ is a power of 2. When we run mergesort on a list of size $n$, how many calls $f(n)$ do we make to mergesort?

If $n \geq 2$, then we have the recurrence

$$f(n) = 1 + 2f(\frac{n}{2})$$

namely to mergesort a list of size $n$, we call mergesort on this list which then requires we call mergesort twice on a list of size $n/2$. The base case is $n = 1$, where we have just one call so $f(1) = 1$.

Let's solve the recurrence:

$$\begin{aligned}
f(n) &= 1 + 2\ f(\frac{n}{2}) \\
&= 1 + 2\ (1 +\ 2f(\frac{n}{4})) \\
&= 1 + 2 + 4f(\frac{n}{4}) \\
&= 1 + 2 + 4(1 + 2f(\frac{n}{8})) \\
&= 1 + 2 + 4 + 8f(\frac{n}{8}) \\
&= 1 + 2 + 4 + \cdots + 2^{k-1} + 2^k f(\frac{n}{2^k}) \\
&= 1 + 2 + 4 + \cdots + \frac{n}{2} + nf(1) \quad \text{when } n = 2^k \\
&= 1 + 2 + 4 + \cdots + \frac{n}{2} + n, \text{ since base case is } f(1) = 1 \\
&= \sum_{i=0}^{\log_2 n} 2^i \quad \text{geometric series, i.e.} n = 2^{\log_2 n} \\
&= 2n - 1
\end{aligned}$$

### How many recursive calls to `merge`?

Again, suppose the $n$ is a power of 2. When we run mergesort on a list of size $n$, each call to mergesort makes one call to merge. So you might think that the same recurrence applies.

However, that's not quite correct. In the base case $n = 1$, there are no calls to merge, so $f(1) = 0$ in this case. Using the same derivation as above, we would now get a different solution namely $n - 1$ instead of $2n - 1$.

**On the base case of mergesort**

With mergesort, we have a base case of $n = 1$. What if we had stopped the recursion at a larger base case? For example, suppose that when the list size has been reduced to 4 or less, we switch to running bubble sort instead of mergesort. Since bubble sort is $O(n^2)$, one might ask whether this would cause the mergesort algorithm to increase from $O(n\ \log_2 n)$ to $O(n^2)$.

Let's solve the recurrence for mergesort by assuming $t(n) = 2t(\frac{n}{2}) + c_1 n$ when $n > 4$ but that some other $t(n)$ holds for $n \leq 4$. Assume $n$ is a power of 2 (to simplify the argument). We want to stop the backsubstitution at $t(4)$ on the right side. So we let $k$ be such that $\frac{n}{2^k} = 4$, that is, $2^k = \frac{n}{4}$.

$$\begin{aligned} t(n) &= c\ n + 2\ t(\frac{n}{2}) \\ &= \ldots \\ &= c\ n\ k\ +\ 2^k\ t(\frac{n}{2^k}), \quad \text{and letting } 2^k = \frac{n}{4} \text{ gives...} \\ &= c\ n(\log_2 n - 2) + \ \frac{n}{4}\ t(4) \\ &= c\ n \log_2 n\ -\ 2cn + \ \frac{n}{4}\ t(4) \end{aligned}$$

Note that this is still $O(n\ \log_2 n)$ since the dominant term is $n\ \log_2 n$. Also note that by switching to bubble sort when $n = 4$, we really should write the solution as:

$$t(n) = c\ n \log_2 n\ -\ 2cn + \ \frac{n}{4}\ t_{bubble}(4).$$

Would this method would be faster than the original mergesort? It depends on which is faster: the recursive mergesort on a list of size 4 (or less) or bubblesort on a list of size 4 (or less). I say "or less" here because the size of the original list might not be a power of 2 and so the base case might be size 3 or 2 (e.g. a list of size 5 would be split into lists of size 3 and 2). Why would bubblesort be faster on small lists? Well, bubblesort doesn't have to do the extra work of making new arrays and copying; rather bubblesort (like insertion and selection sort) can be done in place.

## Quicksort

Let's now turn to the recurrence for quicksort. Recall the main idea of quicksort. We choose some element called the pivot, and then partition the remaining elements based on whether they are smaller than or greater than the pivot, recursively quicksort these two lists, and then concatentate the two, putting the pivot in between.

In the best case, the partition produces two roughly equal sized lists. This is the best case because then one only needs about $\log n$ levels of the recursion and approximately the same recurrence as mergesort can be written and solved.

What about the worst case performance? As we have discussed in earlier lectures, if the element chosen as the pivot happens to be smaller then all the elements in the list, or larger then all the elements in the list, then the two lists are of size 0 and $n - 1$. If this poor splitting happens for each list at each level of the recursion, then performance degenerates to that of the $O(n^2)$ sorting algorithms we saw earlier, namely the recurrence becomes

$$t(n) = cn + t(n-1)\ .$$

Solving this by backstitution (see last lecture) gives

$$t(n) = c\frac{n(n+1)}{2} + t(0)n$$

which is $O(n^2)$.

Why is quicksort called "quick" when its worst case is $O(n^2)$ ? In particular, it would seem that mergesort would be quicker since mergesort is $O(n \log n)$, regardless of best or worst case.

There are two basic reasons why quicksort is "quick". One reason is that the first case is easy to avoid in practice. This was discussed at the end of the heapsort 2 lecture, but I'll mention it again here as reminder. If one is a bit more clever about choosing the pivot, then one can make the worst case situation happen with very low probability One idea for choosing a good pivot is to examine three particular elements in the list: the first element, the middle element, and the last element (`list[0], list[mid], list[size-1]`. For the pivot, one sorts these three elements and takes the middle value (the median) as the pivot. The idea is that it is very unlikely for *all three* of these elements to be among the three smallest (or three largest). In particlar, if the list happens to be close to sorted (or sorted in the wrong direction) then the "median of three" will tend to be close the median of the entire list. *Note that the best split occurs if we take the pivot to be the median of the whole list.* In practice, such a simple idea works very well, and partitions have close to even size.

The second reason that quicksort is quick is that, if one uses an array list to represent the list, then it is possible to do the partition *in place*, that is, without using extra space. The "in place" property of quicksort is a big advantage, since it reduces the number of copies one needs to do. By contrast, the straightforward implementation of mergesort requires that we use additional arrays. Creating such arrays and using extra space ends up slowing you down in practice when $n$ is very large.

We have seen several algorithms in the course, and we have loosely characterized their runtimes $t(n)$ in terms of the size $n$ of the input. We say that the algorithm takes time $O(n)$ or $O(\log_2 n)$ or $O(n \log_2 n)$ or $O(n^2)$, etc, by considering how the runtime grows with $n$, ignoring constants and only considering the dominant term in $t(n)$. This level of understanding of $O(\ )$ is usually good enough for characterizing algorithms. However, we would like to be more formal about what this means. What does it mean to ignore constants? What do we mean by the dominant term?

## ASIDE: An analogy from Calculus: limits

A good analogy for informal versus formal definitions is one of the most fundamental ideas of Calculus: the limit. In your Calculus 1 course, you learned about various types of limits and you learned methods and rules for calculating limits e.g. squeeze rule, ratio test, l'Hopital's rule, etc.

You were also given the formal definition of a limit. This formal definition didn't play much role in your Calculus class. You were more concerned with using rules about limits than in fully understanding where these rules come from and why they work. If you go further ahead in your study of mathematics then you will find this formal definition comes up again[22].

Here is the formal definition of the limit of a sequence. *A sequence $t(n)$ of real numbers converges to (or has a limit of) a real number $t_\infty$ if, for any $\epsilon > 0$, there exists an $n_0$ such that for all $n \geq n_0$, $\mid t(n) - t_\infty \mid < \epsilon$.*

This definition is subtle. There are two "for all " logical quantifiers and there is one "there exists" quantifier, and the three quantifiers have to be ordered in just the right way to express the idea, which is this: if you take any finite interval centered at the $t_\infty$, namely $(t_\infty - \epsilon, t_\infty + \epsilon)$, then the values $t(n)$ of the sequence will all fall in that interval once $n$ exceeds some finite value $n_0$. That finite value $n_0$ may depend on $\epsilon$.

This is relevant to big O for two reasons. First, the formal definition of big O has a similar flavour to the formal definition of the limit of a sequence in Calculus. Second, we will see two lectures from now that there are rules that allow us to say that some function $t(n)$ is big O of some other function e.g. $t(n)$ is $O(\log_2 n)$, and these rules combine the formal definition of big O with the formal definition of a limit. Let's put limits aside for now, and return to them once we understand more about big O.

## Big O

Let $t(n)$ be a well-defined sequence of integers. In the last two lectures, such a sequence typically represented the time or number of steps it takes an algorithm to run as a function of $n$ which is the size of the input. However, today we put this interpretation aside and we just consider $t(n)$ to be a sequence of numbers, without any meaning. We will look at the behavior of this sequence $t(n)$ as $n$ becomes large.

[22]in particular, starting in Real Analysis (e.g. MATH 242 at McGill)

**Definition (preliminary)**

Let $t(n)$ and $g(n)$ be two sequences of integers[23], where $n \geq 0$. We say that $t(n)$ *is asymptotically bounded above by* $g(n)$ if there exists a positive number $n_0$ such that,

$$\text{for all } \ n \geq n_0, \quad t(n) \ \leq g(n).$$

That is, $t(n)$ becomes less than or equal to $g(n)$ once $n$ becomes sufficiently large.

**Example**

Consider the function $t(n) = 5n + 70$. It is never less than $n$, so for sure $t(n)$ is not asymptotically bounded above by $n$. It is also never less than $5n$, so it is not asymptotically bounded above by $5n$ either. But $t(n) = 5n + 70$ is less than $6n$ for sufficiently large $n$, namely $n \geq 12$, so $t(n)$ is asymptotically bounded above by $6n$. The constant 6 in $6n$ is one of infinitely many that work here. Any constant greater than 5 would do. For example, $t(n)$ is also asymptotically bounded above by $g(n) = 5.00001n$, although $n$ needs to be quite large before $5n + 70 \leq 5.00001n$.

The formal definition of big O below is slightly different. It allows us to define an asymptotic upper bound on $t(n)$ in terms of a *simpler* function $g(n)$, e.g. :

$$1, \log n, \ n, \ n \log n, \ n^2, \ n^3, \ 2^n, \ldots$$

without having a constant coefficient. To do so, one puts the constant coefficient into the definition.

## Definition (big O):

Let $t(n)$ and $g(n)$ be well-defined sequences of integers. We say $t(n)$ is $O(g(n))$ if there exist two positive numbers $n_0$ and $c$ such that, for all $n \geq n_0$,

$$t(n) \ \leq \ c \, g(n).$$

We say "$t(n)$ is big O of $g(n)$". I emphasize: this definition allows us to keep the $g(n)$ simple by having a constant factor ($c$) that is separate from the simple $g(n)$.

A few notes about this definition: First, the definition still is valid if $g(n)$ is a complicated function, with lots of terms and constants. But the whole point of the definition is that we keep $g(n)$ simple. So that is what we will do for the rest of the course. Second, the condition $n \geq n_0$ is also important. It allows us to ignore how $t(n)$ compares with $g(n)$ when $n$ is small. This is why we are talking about an *asymptotic* upper bound.

[23]Usually we are thinking of positive integers, but the definition is general. The $t(n)$ could be real numbers, positive or negative

**Example 1**

The function $t(n) = 5n + 70$ is $O(n)$. Here are a few different proofs:

First,

$$\begin{aligned} t(n) &= 5n + 70 \\ &\leq 5n + 70n, \text{ when } n \geq 1 \\ &= 75n \end{aligned}$$

and so $n_0 = 1$ and $c = 75$ satisfies the definition.

Here is a second proof:

$$\begin{aligned} t(n) &= 5n + 70 \\ &\leq 5n + 6n, \text{ for } n \geq 12 \\ &= 11n \end{aligned}$$

and so $n_0 = 12$ and $c = 11$ also satisfies the definition.

Here is a third proof:

$$\begin{aligned} t(n) &= 5n + 70 \\ &\leq 5n + n, \text{ for } n \geq 70 \\ &= 6n \end{aligned}$$

and so $n_0 = 70$ and $c = 6$ also satisfies the definition.

A few points to note:

- If you can show $t(n)$ is $O(g(n))$ using constants $c, n_0$, then you can always increase $c$ or $n_0$ or both, and these constants with satisfy the definition also. So, don't think of the $c$ and $n_0$ as being uniquely defined.
- There is no "best" choice of $c$ and $n_0$. The examples above show that if you make $c$ bigger (less strict) then you can make $n_0$ smaller (more strict, since the inequality needs to hold for more values of $n$). The big O definition says nothing about "best" choice of $c$ and $n_0$. It just says that there has to exist one such pair.
- There are inequalities in the definition, e.g. $n \geq n_0$ and $t(n) \leq cg(n)$. Does it matter if the inequalities are strict or not? No. If we were to change the definitions to be strict inequalities, then we just might have to increase the $c$ or $n$ slightly to make the definition work.

**An example of an incorrect big O proof**

Many of you are learning how to do proofs for the first time. It is important to distinguish a formal proof from a "back of the envelope" calculation. For a formal proof, you need to be clear on what you are trying to prove, what your assumptions are, and what are the logical steps that take you from your assumptions to your conclusions. Sometimes a proof requires a calculation, but there is more to the proof than calculating the "right answer".

For example, here is a typical example of an incorrect "proof" of the above: (in the lecture, I presented this a bit later)

$$\begin{array}{rcl} 5n+70 & \leq & cn \\ 5n+70n & \leq & cn, \quad n \geq 1 \\ 75n & \leq & cn \\ \text{Thus,} \quad c > 75, \quad n_0 = 1 & & \text{works.} \end{array}$$

This proof contains all the calculation elements of the first proof above. But the proof here is wrong, since it isn't clear which statement implies which. The first inequality may be true or false, possibly depending on $n$ and $c$. The second inequality is different than the first. It also may be true or false, depending on $c$. And which implies which? The reader will assume (by default) that the second inequality *follows from* the first. But does it? Or does the second inequality imply the first? Who knows? Such proofs tend to get grades of 0.[24] This is not the big O that you want. Let's turn to another example.

## ASIDE: another incorrect proof

When one is first learning to write proofs, it is common to leave out certain important information. Let's look at a few examples of how this happens.

### Claim:

For all $n \geq 1$, $2n^2 \leq (n+1)^2$.

If you are like me, you probably can't just look at that claim and evaluate whether it is true or false. You need to carefully reason about it. Here is the sort of incorrect "proof" you might be tempted to write, given the sort of manipulations I've been doing in the course:

$$\begin{array}{rcl} 2n^2 & \leq & (n+1)^2 \\ & \leq & (n+n)^2, \text{ where } n \geq 1 \\ & \leq & 4n^2 \end{array}$$

which is true, i.e. $2n^2 \leq 4n^2$. Therefore, you might conclude that the claim you started with is true.

Unfortunately, the claim is false. Take $n = 3$ and note the inequality fails since $2 \cdot 3^2 > 4^2$. The proof is therefore wrong. What went wrong is that the first line of the proof *assumes what we are trying to prove.* This is a remarkably common mistake.

Let's now get back to big O and consider another example.

[24]As mentioned in class, with 600 students, we just don't have the resources to ask you to write out these proofs on exams.

**Example 2**

Claim: The function $t(n) = 8n^2 - 17n + 46$ is $O(n^2)$.

Proof: We need to show there exists positive $c$ and $n_0$ such that, for all $n \geq n_0$,

$$8n^2 - 17n + 46 \leq cn^2 \ .$$

$$\begin{aligned} t(n) &= 8n^2 - 17n + 46 \\ &\leq 8n^2 + 46n^2, \qquad \text{for } n \geq 1 \\ &= 54n^2 \end{aligned}$$

and so $n_0 = 1$ and $c = 54$ does the job.

Here is a second proof:

$$\begin{aligned} t(n) &= 8n^2 - 17n + 46 \\ &\leq 8n^2, \quad n \geq 3 \end{aligned}$$

and so $c = 8$ and $n_0 = 3$ does the job.

**Miscellaneous but important notes**

Here are a few points to be aware of:

- We sometimes say that a function $t(n)$ is $O(1)$. What does this mean? Applying the definition, it means that there exists constants $c$ and $n_0$ such that, for all $n \geq n_0$, $t(n) \leq c$. That is, $t(n)$ is bounded above by a constant, namely $max\{t(0), t(1), \ldots, t(n_0), c\}$.
- You will not write that a function or algorithm is $O(3n)$ or $O(5log_2 n)$, etc. Instead, you should write $O(n)$ or $O(log_2 n)$, etc. Why? Because the point of big O notation is to avoid dealing with these constant factors. So, while it is still technically correct to write the above, in that it doesn't break the formal definition of big O, we just never do it.
- We generally want our $g(n)$ to be simple, and we also generally want it to be small. But the definition doesn't require this. For example, in the above example, $t(n) = 5n + 70$ is $O(n)$ but it is also $O(n \log n)$ and $O(n^2)$ and $O(n^3)$, etc. For this example, we say that $O(n)$ is a "tight bounds". We generally express $O(\ )$ using tight bounds. I will return to this point next lecture.

## Big O and sets of functions

When we characterize a function $t(n)$ as being $O(g(n))$ we usually use simple functions for $g(n)$ such as in the list of inequalities:

$$1 < \log_2 n < n < n \log_2 n < n^2 < n^3 < ... < 2^n < n! < n^n$$

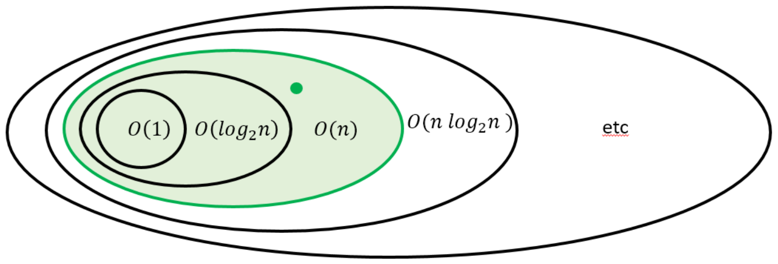
Note that these inequalities don't hold for all $n$. Each of the inequalities holds only for sufficiently large $n$: For example, $2^n < n!$ holds only for $n \geq 4$, and $n^3 < 2^n$ only holds for $n \geq 10$. But for each of the inequalities, there is some $n_0$ such that the the inequality holds for all $n \geq n_0$. This is good enough, since big O deals with asymptotic behavior anyhow.

When we talk about $O()$ of some function $t(n)$, we usually also use the "tightest" (smallest) upper bound we can. For example, if we observe that a function $t(n)$ is $O(\log_2 n)$, then generally we would not say that that $t(n)$ is $O(n)$, even though technically $t(n)$ *would* be $O(n)$, and it would also be $O(n^2)$, etc.

A related observation is that, for a given simple $g(n)$ such as listed in the sequence of inequalities above, there are infinitely many functions $t(n)$ that are $O(g(n))$. So let's think about the set of functions that are $O(g(n))$. Up to now, we have been saying that some function $t(n)$ *is* $O(g(n))$. But sometimes we say that $t(n)$ *is a member of the set of functions that are* $O(g(n))$, or more simply $t(n)$ "belongs to" $O(g(n))$. In set notation, one writes "$t(n) \in O(g(n))$" where $\in$ is notation for set membership. With this notation in mind, and thinking of various $O(g(n))$ as sets of functions, the discussion in the paragraphs above implies that we have strict containment relations on sets of $t(n)$ functions:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \ \cdots \subset O(2^n) \subset O(n!) \ldots$$

For example, any function $t(n)$ that is $O(1)$ must also be $O(\log_2 n)$, and any function $t(n)$ that is $O(\log_2 n)$ must also be $O(n)$,etc. It is common to use this set notation and say "$t(n) \in O(g(n))$" instead of "$t(n)$ is $O(g(n))$". The set containment relationships can be visualized as shown below. The green ellipse represents the set of functions that are $O(n)$.



## Some Big O Rules

Last lecture, we showed some simple examples of how to show that some function $t(n)$ is $O(g(n))$ for some other function $g(n)$. In these examples, we manipulated an inequality and found a $c$ and $n_0$. We don't want to have to do this every time we make a statement about big O, and so we would like to have some rules that allow us to avoid having to state a $c$ and $n_0$. Thankfully and not surprisingly, there are such rules.

I emphasize: the point of the rules that I am about to explain is that they allow us to make immediate statements about the $O(\ )$ behavior of some rather complicated functions. For example, we can just look at the following function

$$t(n) = 5 + 8 \log_2 n + 16n + \frac{n(n-1)}{25}$$

and observe it is $O(n^2)$ by noting that the largest term is $n^2$. The following rules *justify* this informal observation.

The "scaling rule" says that if we multiply a function by a positive constant, then we don't change the $O()$ class of that function. The "sum rule" says that if we have a function that is a sum of two terms, then the $O()$ class of the function is the larger of the $O()$ classes of the two terms. The "product rule" says that if we have a function that is the product of two functions, then the $O()$ class of the product function is given by the product of the (simple) functions that represent the $O()$ classes of the original two functions. Here are the statements of the rules, and the proofs.

## Scaling Rule

If $f(n)$ is $O(g(n))$ and $a > 0$ is a positive constant, then $af(n)$ is $O(g(n))$.

**Proof:** There exists a $c$ such $f(n) \le cg(n)$ for all $n \ge n_0$, and so $af(n) \le acg(n)$ for all $n \ge n_0$. Thus we use $ac$ and $n_0$ as constants to show $a\ f(n)$ is $O(g(n))$.

## Sum Rule

If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(max(g_1(n), g_2(n))$.

**Proof:**

From the three big O relationships given in the premise (i.e. the"if" part of the "if-then"), there exists constants $c_1, c_2, c_3, n_1, n_2, n_3$ such

$$f_1(n) \le c_1 g_1(n) \ \text{ for all n} \ge \text{n}_1$$

$$f_2(n) \le c_2 g_2(n) \ \text{ for all n} \ge \text{n}_2$$

Thus,

$$f_1(n) + f_2(n) \le c_1 g_1(n) + c_2 g_2(n) \text{ for all n } \ge \ max(n_1, n_2).$$

From this we can conclude:

$$f_1(n) + f_2(n) \le (c_1 + c_2)(max(g_1(n), g_2(n)) \text{ for all n } \ge \ \text{max}(\text{n}_1, \text{n}_2)$$

So we can let $c = c_1 + c_2$, $n_0 = max(n_1, n_2)$ which proves the claim.

## Product Rule

If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n)$, then $f_1(n)f_2(n)$ is $O(g_1(n)\ g_2(n))$.

**Proof:** Using similar constants as in the sum rule, we have: $f_1(n) * f_2(n) \le c_1 g_1(n) * c_2 g_2(n)$ for all $n \ge max(n_1, n_2)$. So we can take $c_1 * c_2$ and $max(n_1, n_2)$ as our two constants.

## Big Omega (asymptotic lower bound)

With big O, we defined an asymptotic *upper bound.* We said that one function grows at most as fast as another function. There is a similar definition for an asymptotic *lower* bound. Here we say that one function grows *at least* as fast as another function. The lower bound is called "big Omega".

The reason we care about lower bounds is that we want to say that something takes at least a certain amount of time to run. For example, if you want to find the largest element in a list of size $n$, then you have to examine all the elements and so it takes at least $n$ steps (and in fact it always takes you exactly $n$ steps).

An interesting example you will likely see in COMP 251 is comparison based sorting: these are sorting algorithms that require comparing pairs of elements. (All of the sorting elements we've seen are of this type, but there are other interesting types of sorting algorithms/problems that one can come up with.) You will see in COMP 251 that any comparison based sorting algorithm has to do at least $cn\log_2 n$ comparisons in the *average case*, that is, averaged over all possible permutations of the $n$ inputs. By saying "at least", we mean a lower bound.

**Definition (big Omega):** We say that $t(n)$ is $\Omega(g(n))$ if there exists positive constants $n_0$ and $c$ such that, for all $n \geq n_0$,

$$t(n) \;\geq\; c\,g(n).$$

The idea is that $t(n)$ grows at least as fast as $g(n)$ times some constant, for sufficiently large $n$. Note that the only difference between the definition of $O()$ and $\Omega()$ is the $\leq$ vs. $\geq$ inequality.

### Example

Claim: Let $t(n) = \frac{n(n-1)}{2}$. Then $t(n)$ is $\Omega(n^2)$.

To prove this claim, first note that $t(n)$ is less than $\frac{n^2}{2}$ for all $n$, so since we want a *lower* bound we need to choose a smaller $c$ than $\frac{1}{2}$. Let's try something smaller, say $c = \frac{1}{4}$.

$$\begin{array}{rrcl} & \frac{n(n-1)}{2} & \geq & \frac{n^2}{4} \\ \iff & 2n(n-1) & \geq & n^2 \\ \iff & n^2 & \geq & 2n \\ \iff & n & \geq & 2 \end{array}$$

Note that the "if and only if" symbols $\iff$ are crucial here. For any $n$, the first inequality is either true or false. We don't know which, until we check. But putting the $\iff$ in there, we are say that the inequalities in the different lines have the same truth value.

The last lines says $n \geq 2$, this means that the first inequality is true if and only if $n \geq 2$. Thus, we can use $c = \frac{1}{4}$ and $n_0 = 2$.

Are these the only constants we can use? No. Let's try $c = \frac{1}{3}$.

$$\begin{array}{rrcl} & \frac{n(n-1)}{2} & \geq & \frac{n^2}{3} \\ \iff & \frac{3}{2}n(n-1) & \geq & n^2 \\ \iff & \frac{1}{2}n^2 & \geq & \frac{3}{2}n \\ \iff & n & \geq & 3 \end{array}$$

So, we can use $c = \frac{1}{3}$ and $n_0 = 3$.

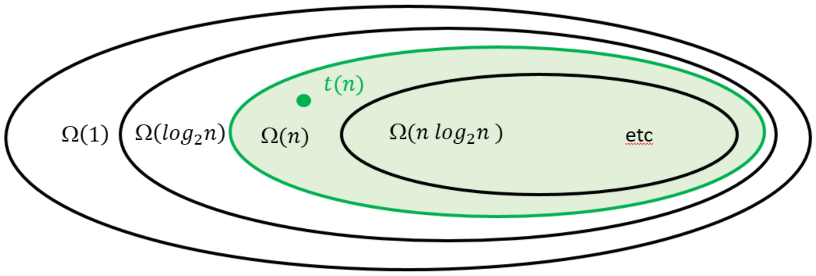
**Sets of $\Omega(\ )$ functions**

Recall earlier in the lecture when we discussed sets of functions that all have the same $O()$ behavior. We can do the same with $\Omega$. When we say that "$t(n) \in \Omega(g(n))$" for some $g(n)$, we mean that $t(n)$ is a member of the set of functions that are $\Omega(g(n))$. The set relationship is different from what we saw with $O()$, however, since we are now discussing lower bounds rather than upper bounds. For $\Omega$, the set membership symbols are in the opposite direction:

$$\Omega(1) \supset \Omega(\log n) \supset \Omega(n) \supset \Omega(n \log n) \supset \Omega(n^2) \ \cdots \supset \Omega(2^n) \supset \Omega(n!) \ldots$$

namely we have supersets instead of subsets.

For example, the largest set is $\Omega(1)$. The reason is that is relatively easy for a function to be in this set: the function just needs to be bounded below by a constant for $n$ sufficiently large, i.e. this is the definition of $\Omega(1)$. Indeed the only way any function would *not* be in $\Omega(1)$ is if it had an infinite subsequence of $n$ values for which $t(n)$ converged to 0. For example, $t(n) = \frac{1}{n}$ would be such a function. Such functions certainly exist, of course, but they are irrelevant for the discussion of time complexity, i.e. where $t(n)$ is the time (or number of steps) for an algorithm to run as a function of $n$.



We can also talk about *tight bounds* on $\Omega()$ of any function $t(n)$, namely again it is the smallest of the above $\Omega()$ sets that contains $t(n)$. In the above illustration, the function $t(n)$ is in $\Omega(n)$. It is therefore also in $\Omega(\log_2 n)$ and $\Omega(1)$. But the tight (lower) bound is $\Omega(n)$.

## Big Theta

In the last two lectures, we discussed $O(\ )$ and $\Omega(\ )$ bounds. As it turns out, for all time complexity functions $t(n)$ that we will care about in this course, there is be a simple function $g(n)$ such that $t(n)$ is both $O(g(n))$ and $\Omega(g(n))$. For example, $t(n) = \frac{n(n+1)}{2}$ is both $O(n^2)$ and $\Omega(n^2)$. In this case, we say that $t(n)$ is "big theta" of $n$, or $\Theta(g(n))$.

**Definition (big theta):** We say that $t(n)$ is $\Theta(g(n))$ if $t(n)$ is both $O(g(n))$ and $\Omega(g(n))$ for some $g(n)$. An equivalent definition is that there exists three positive constants $n_0$ and $c_1$ and $c_2$ such that, for all $n \geq n_0$,

$$c_1\ g(n)\ \leq\ t(n)\ \leq\ c_2\ g(n).$$

Obviously, $c_1 \leq c_2$. Note that here we have such one constant $n_0$. This is just the max of the $n_1, n_2$ constants of the $O(\ )$ and $\Omega(\ )$ definitions.

For example, a function such as $t(n) = 4 + 17\log_2(n+3) + 9n\log_2 n + \frac{1}{2}n(n-1)$ is $\Theta(n^2)$. Don't be thrown off by all the other terms. What matters for both $O()$ and $\Omega$ as $n$ gets big is only the term that depends on $n^2$.

### Sets of Big Theta functions

Since a function $t(n)$ is $\Theta(g(n))$ if it is both $O(g(n))$ and $\Omega(g(n))$, it means that the set of $\Theta(g(n))$ functions is the intersection of the set of $O(g(n))$ functions and the set of $\Omega(g(n))$ functions. Thus, unlike different $O()$ and $\Omega()$ classes which form nested sets, as discussed earlier, the big $\Theta$ classes form disjoint sets. If $t(n)$ belongs to one big $\Theta$ class, then it doesn't belong to any other $\Theta$ class.

Note that there exist functions $t(n)$ that do *not* belong to any $\Theta(g(n))$ class. For example, consider a function that has a constant value, say 5, when $n$ is even and has value $n$ when $n$ is odd. Such a function $t(n)$ is $\Omega(1)$ and it is $O(n)$ but it is neither $\Theta(1)$ nor $\Theta(n)$. Obviously one can contrive many such functions. But these functions rarely come up in real computer science problems. For every $t(n)$ function that we have discussed in this course, it belongs in some $\Theta()$ class.

If the time complexity functions that we care about in computer science are always characterized by some $\Theta(\ )$, then why do we bother talking about $O(\ )$ and $\Omega(\ )$ ? The answer is that often we are using these asympototic bounds because we want to express that something takes at most or at least a certain amount of time. When we want to say "at most", we are talking about an upper bound and so saying $O(\ )$ emphasizes this. When we say "at least", we are talking about a lower bound and so saying $\Omega(\ )$ that. I'll have more to say about that next.

## Best and worst case

The time it takes to run an algorithm depends on the size $n$ of the input, but it also depends on the values of the input. For example, to remove an element from an arraylist takes constant time (fast) if one is removing the last element in the list, but it takes time proportion to the size of the list if one is removing the first element. Similarly, quicksort is fastest if one chooses pivots that split the lists into two roughly equal sublists but it is slowest if one chooses the worst possible pivot at each step. So one cannot always say that a given algorithm always has a certain $t(n)$ behavior: there are different behaviors, depending on the inputs. (Note for *ArrayList.remove(i)*, the variable `i` is a parameter, and here I am considering this to be part of the input to the algorithm. The other part of the input would be the list of size $n$.)

Whe we talk about *best case* and *worst case* for an algorithm, we are restricting the discussion to a particular set of inputs in which the algorithm takes a minimum number of steps or a maximum number of steps, respectively. With this restriction, one writes the best or worst case time as two functions $t_{best}(n)$ or $t_{worst}(n)$ respectively.

The $t_{best}(n)$ or $t_{worst}(n)$ functions usually *each* can be characterized by some $\Theta(\ g(n)\ )$. However, the $g(n)$ may be different for $t_{best}(n)$ and $t_{worst}(n)$ as in the examples just mentioned. Here are some other examples that you should be very familiar with by now.

| List Algorithms | $t_{best}(n)$ | $t_{worst}(n)$ |
|---|---|---|
| add, remove element (array list) | $\Theta(1)$ | $\Theta(n)$ |
| add, remove an element (doubly linked list) | $\Theta(1)$ | $\Theta(n)$ |
| insertion sort | $\Theta(n)$ | $\Theta(n^2)$ |
| selection sort | $\Theta(n^2)$ | $\Theta(n^2)$ |
| binary search (sorted array) | $\Theta(1)$ | $\Theta(\log n)$ |
| mergesort | $\Theta(n \log n)$ | $\Theta(n \log n)$ |
| quick sort | $\Theta(n \log n)$ | $\Theta(n^2)$ |

Note that selection sort and mergesort have the same best and worst case complexities, but for the other algorithms the best and worst cases differ.

In the table above, I wrote the best and worst cases using $\Theta(\ )$. However, as I mentioned on the previoius page, for a given algorithm it is common to characterize $t_{best}(n)$ using a $\Omega(\ )$ bound and to characterize $t_{worst}(n)$ using a $O(\ )$ bound. One does so when want wishes to emphasize that one is talking about the best case one often wants to express how good (lower bound) the best case can be. Similarly when discussing the worst case one often wants to express how bad this worst case can be. I believe this is why `http://bigocheatsheet.com/` lists some best and worst case bounds using $\Omega$ and $\Theta$ respectively.

Note that it could still be mathematically correct to characterize $t_{best}(n)$ using a $O(\ )$ bound. For example, if we were to say that the best case of removing an element from an arraylist takes time $O(1)$, we would be emphasizing that it is *not worse (bigger) than* constant time. Similarly, if we were to say that the worst case of quicksort is $\Omega(n^2)$, we would be emphasizing that the worst case takes *at least* that amount of time. So again, it really depends what you are trying to say. [ASIDE: From what I've seen on various web posting and in textbooks, authors often are not clear about what they are trying to say when they use these terms, and in particular, people use $O()$ rather than $\Theta$ probably more than they should. Oh well... nothing to lose sleep over.]

## Using "limits" for asymptotic complexity

In Calculus 1, you are given a formal definition of a limit of a sequence. Typically you are not responsible for knowing this formal definition there. Rather, you are responsible for knowing methods to decide if a limit exists and what the limit is, but you were probably not asked for formal proofs. Instead you are given certain rules that you follow for determing the limits e.g. l'Hopital's rule, the ratio test, etc.

Similarly, we have rules for limits that show if one function is $O(\ )$, $\Omega(\ )$, and $\Theta(\ )$ of another function. To be complete here, I would need to use the formal definition of a limit from mathematics which I mentioned back in lecture 35. However, at this point in the semester, I don't think most of you would be in the mood to struggle with that (unless you've taken MATH 242 Real Analysis). So I'll keep it informal. But I do require you to know the rules that I'll present.

For the following rules, we suppose that $t(n)$ and $g(n)$ are two sequences of non-negative numbers.

**Limit rule: case 1a**

If $\lim_{n\to\infty} \frac{t(n)}{g(n)} = 0$, then $t(n)$ is $O(g(n))$.

[ASIDE i.e. I am not holding you responsible for the proof: The mathematical proof uses the formal definition of a limit. Since $\frac{t(n)}{g(n)} \to 0$ as $n \to \infty$, we have (by the definition of a limit) that for *any* $c > 0$, there would be an $n_0$ such that $|\frac{t(n)}{g(n)}| < c$ for all $n \geq n_0$. To show $t(n)$ is $O(g(n))$, we only need one such $c$ ("there exists $c > 0$..." is a much weaker condition than "for all $c > 0$..." so the limit rule easily holds.) ]

Note that the rule does not work in the opposite direction (the converse), that is, we cannot say that if $t(n)$ is $O(g(n))$ then $\lim_{n\to\infty} \frac{t(n)}{g(n)} = 0$, An example is the case that $t(n) = g(n)$ since $t(n)$ would be $O(g(n))$ but $\frac{t(n)}{g(n)} = 1$ for all $n$ and so the limit would not be 0.

**Limit rule: case 1b**

If $\lim_{n\to\infty} \frac{t(n)}{g(n)} = 0$, then $t(n)$ is not $\Omega(g(n))$.

To prove this rule, we first note the definition for big $\Omega$ (Omega) can be written equivalently as follows: there exist two positive numbers $n_0$ and $c > 0$ such that, for all $n \geq n_0$,

$$\frac{t(n)}{g(n)} \ \geq \ c \ .$$

Note that we do require $c > 0$. If we were to allow $c = 0$, then the definition would always hold for *any* strictly positive sequences $t(n)$ and $g(n)$, which would not express anything interesting since the sequences $t(n)$ and $g(n)$ were assumed to be positive.

The proof of the case 1b rule should now be obvious. The above inequality cannot hold when the limit $\frac{t(n)}{g(n)}$ is 0, since the limit of 0 means that the sequence would become less than any fixed $c > 0$ when $n$ is sufficiently large, whereas the above inequality says that the sequence would be greater than $c > 0$ when $n$ is sufficiently lager. In particular, notice that $t(n)$ cannot be $\Theta(g(n))$.

**Limit rule: case 2a**

If $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \infty$, then $t(n)$ is $\Omega(g(n))$.

The idea here is to flip the expression and realize that $\lim_{n\to\infty} \frac{g(n)}{t(n)} = 0$. Thus from case 1a, $g(n)$ is $O(t(n))$ and so it follows that $t(n)$ is $\Omega(g(n))$.

**Limit rule: case 2b**

If $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \infty$, then it cannot be that $t(n)$ is $O(g(n))$.

The reason that " $t(n)$ is $O(g(n))$" would require there is a $c > 0$ such that $\frac{t(n)}{g(n)} < c$ for all $n$ beyond some $n_0$. But this would contradict the fact that $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \infty$. In particular, $t(n)$ is not in $\Theta(g(n))$.

**Limit rule: case 3**

If $\lim_{n\to\infty} \frac{t(n)}{g(n)} = a$ where $0 < a < \infty$, then $t(n)$ is $\Theta(g(n))$.

The intuitive idea here is that if the limit exists, then $\frac{t(n)}{g(n)}$ will be bounded above by a constant once $n$ becomes sufficiently large. In particular, if we consider a number slightly larger than the limit $a$, say $a + \epsilon$ where $\epsilon > 0$, then we can be sure that there is an $n_0$ such that $\frac{t(n)}{g(n)} < a + \epsilon$ once $n \geq n_0$. But this tells us immediately that $t(n)$ is $O(g(n))$.

Similarly, if we consider a number slightly smaller than the limit $a$, say $a - \epsilon$ where $\epsilon > 0$, then we can be sure that there is an $n_0$ (possibly different from the one in the previous paragraph) such that $\frac{t(n)}{g(n)} > a - \epsilon$ once $n \geq n_0$. But this tells us immediately that $t(n)$ is $\Omega(g(n))$. So since $t(n)$ is both $O(g(n))$ and $\Omega(g(n))$, it means (by definition) that $t(n)$ is both $\Theta(g(n))$

And if you are still reading this and you understood what I wrote above, then you are in great shape: well done!!