

In the last two lectures, we looked at how numbers are represented in binary. These numbers representations are used in two ways. The first is to represent *data* in your programs. The second is to represent the *addresses* in memory. These could be addresses of data, or they could be addresses of the program instructions. (The program also sits in memory.) Today we will discuss these two different uses of binary number representations.

You probably learned in COMP 202 that data organized in groups of 8 bits, called *bytes*. You also learned about primitive types vs. reference types in Java. You were told that different primitive types in Java use a different number of bits, and hence a different number of bytes, namely `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double` use 1,1,2,2,4,8,4,8, bytes respectively. Think of a variable that is a *primitive type* as naming a particular byte in memory which is the starting location (address) of a data item. The *type* of the variable says how many bytes this data item occupies in memory.

Variables that are *reference types* are different. A reference variable also names a location in memory, but this location is where one finds an *address* of an object, that is, the starting address of the object which is located somewhere else in memory. Thus, you should think of both primitive and reference variables as naming (standing for) a number which is a location in memory which is the starting location where something is stored: either a data item itself (primitive type), or the address of some other data item e.g. an object (reference type).

Sometimes I write *starting address* rather than just address. Why? In Java only `boolean` and `byte` types use one byte. Everything else uses multiple bytes. When I say “starting address”, I am just reminding you that the item being stored really takes more than one byte, and the address only refers to one of the bytes. (Yes, you can assume that any item is stored in sequential bytes, so to know which bytes an item occupies in memory you just need to know the starting address and the number of bytes occupied by that item.)

## Memory addresses as 32 bit numbers

Most computers that you will work with have either a 32 bit or 64 bit “address space”.<sup>1</sup> An address is just a number, so we are talking about numbers from 0 to  $2^{32} - 1$  (in the case of 32 bits) or 0 to  $2^{64} - 1$  (in the case of 64 bit). To keep things simple, I will typically be talking about a 32 bit address space, but the things I say apply just as well to 64 bit address spaces.

Note that since each 32 bit address indexes one byte (8 bits) of memory, such a byte can hold only a part of a piece of data.<sup>2</sup> That is, every data element is ultimately coded in bits which are partitioned into bytes, and each of these bytes must have a location (address) somewhere in memory. With 32 bits to represent a (byte) address, one has  $2^{32}$  possible byte addresses.

[ASIDE: I did not mention this in class but in case you did not see it in COMP 202 (or equivalent), you should know that:

- $2^{10}$  bytes = 1 kilobyte (1 KB)  $\approx 10^3$  bytes (one thousand)
- $2^{20}$  bytes = 1 megabyte (1 MB)  $\approx 10^6$  bytes (one million)
- $2^{30}$  bytes = 1 gigabyte (1 GB)  $\approx 10^9$  bytes (one billion)

---

<sup>1</sup>Indeed this is what is meant when you are installing some new software and you are asked to choose between 32 and 64 bits and other version info about your operating system.

<sup>2</sup>or a part of an instruction... instructions are also stored in memory and so instructions also have addresses.

Thus,  $2^{32}$  bytes is 4 gigabytes (4 GB). ]

Interestingly, when we are talking as above about addresses of things in a program, we are *not* referring to physical locations in the computer’s memory *hardware* (RAM or hard disk). Rather, when a program is running (on a computer), the program addresses I was referring to above must be *mapped* (translated) into the corresponding physical locations of the bytes in the physical storage of the computer. You will learn about how this mapping is done in COMP 273 (or ESCE 221), once you know something about computer architecture.

To intuitively understand the need for this mapping from 32-bit program addresses to physical addresses, consider an analogy to addresses that you use in your daily life. Homes and businesses and other workplaces have addresses, but they are not physical addresses on Earth, in the sense of latitude and longitude and altitude. Rather, the addresses that we use have “street names and numbers” and maybe “room numbers” etc. We rely on maps to relate these postal addresses to physical locations.

The same holds for the telephone. You have a person’s telephone number, and the telephone company is responsible for mapping that number to a physical location. You, as the user, typically are not concerned with a phone’s location (at least, that is not what the phone is for). Rather you just want to speak to the person who owns the phone. (This is of course a key advantage of cell phones over home phones.)

So it is with computer programs. When we say that a computer program uses a 32 bit address space, we don’t mean that the physical computer has  $2^{32}$  bytes of physical memory (4 GB). Rather, we just mean that the program (software) running on that computer can refer to and use  $2^{32}$  distinct byte locations. (Such memory is also called *virtual memory*. )

As a Java or C or C++ or whatever programmer, you never see this mapping from program addresses to physical addresses. Rather, the “map” is organized and maintained by the operating system of the computer – namely Windows or Linux or whatever. Indeed, when you compile a program and you have an “executable” file (sometimes called a binary), this file still uses the 32 bit program addressing scheme that I have been talking about. When this files then is executed (run), the operating system and hardware map at runtime these virtual program addresses to physical addresses on the computer.

Note: *I don’t expect you to fully understand this stuff*. Rather, I just want to expose you to it and give you some intuition. I believe this intuition will help you alot in the weeks to come, when we discuss various data structures (linear and non-linear). Let’s now turn to one such data structure that you are familiar with: arrays.

## Arrays

You have all seen arrays in COMP 202 (or equivalent). In Java, an array consists of elements (primitives or objects) that are all of the same type. Array elements are indexed using positive integers, starting from 0. Each index corresponds to one element in the array.

Suppose we have a Java class `Student`, and we have an instruction

```
Student[] studentArray = new Student[13];
```

This is in fact two declarations

```
Student[] studentArray;
```

```
studentArray = new Student[13];
```

The first defines a reference variable (4 bytes) which holds the address of an object, namely a `Student` array. The second constructs/instantiates a `Student` array. The array can reference up to 13 students and these references are initialized to `null`. (`null` is the special address 0 in our 32 bit address space.)

Later we instantiate objects of the class *Student* and use the array to reference them, as in

```
studentArray[0] = new Student("Fred");
```

```
studentArray[2] = new Student("Mustafa");
```

In this case, we would have a `Student` object with name “Fred”. These object would have starting addresses (32 bit numbers) that are somewhere in memory. These starting addresses are stored in the `Student` array object.

[ASIDE: although the `Student` array object holds 13 addresses, this object is bigger than  $13 \times 4$  bytes. For example, the object also contains an integer that specifies the number of slots (13, in the example above), and it will also needs to contain information that specifies it is a `Student` array, rather than some other type of array. I will say more on this at the end of the course.]

## Algorithm for sorting an array (“insertion sort”)

Let’s use arrays to solve a problem that comes up often in programming, namely *sorting*. Suppose we have an array of objects that is in no particular order and the objects are such that it is meaningful to talk about an ordering e.g. the objects might be numbers, or they might be strings which can be sorted alphabetically (e.g. to make a phonebook). Given the under-ordered items in an array, we would like to rearrange the items in the array such that they are sorted.

There are possible algorithms for doing so. “Insertion sort” is one of the simplest to describe. The basic idea of the algorithm is to assume that the  $k$  elements of the array (indices  $0, \dots, k - 1$ ) are already in the correct order, and then insert element at index  $k$  into its correct position with respect to the first  $k$  elements. We start with  $k = 0$ . The first element is clearly in its correct position if we only have one element, so there is nothing to do.

Now suppose that the first  $k$  elements are correctly ordered relative to each other. How do we put the element at index  $k$  into its correct position? The idea is to look backwards from index  $k$  until we find the right place for it. Element  $a[k - 1]$  is the largest of all  $a[0], \dots, a[k - 1]$  by assumption, since the first  $k$  elements are in their correct order. When stepping backwards, if the element in the next position is greater than  $a[k]$ , then we move that element forward in the array to make room for the  $a[k]$ . If, on the other hand, if we find an element that less than (or equal to)  $a[k]$  then we go no further.

The algorithm is listed on the next page. You should step through it and make sure you follow it. I suggest you also have a look at an applet that allows you to visualize how the algorithm works, such as

<http://tech-algorithm.com/articles/insertion-sort>

---

**ALGORITHM: INSERTION SORT**INPUT: an array  $a[ ]$  with  $N$  elements that can be compared ( $<$ ,  $=$ ,  $>$ )OUTPUT: the array  $a[ ]$  containing the same elements, in increasing order

```

for  $k = 1$  to  $N - 1$  do
   $tmp \leftarrow a[k]$ 
   $i \leftarrow k$ 
  while  $(i > 0) \ \& \ (tmp < a[i - 1])$  do
     $a[i] \leftarrow a[i - 1]$ 
     $i \leftarrow i - 1$ 
  end while
   $a[i] = tmp$ 
end for

```

---

The following material was not discussed in the lecture, but you should read it and understand it.

**Analysis of insertion sort**

Suppose you are given an array  $a[ ]$  which is of size  $N$ , and you step through this algorithm line by line. How many steps will you take? Interestingly, the answer depends on the data.

Suppose the operations inside the **for** loop but outside the **while** loop take  $c_1$  time steps, and the operations inside the **while** loop take  $c_2$  times steps. Then, in the worst case, you need to take about  $c_1N + c_2(1 + 2 + \dots + N - 1)$ , where the latter expression occurs in the case that, for each  $k$ , the **while** loop decrements  $i$  all the way from  $k$  back to 0. But you may recall from high school math that

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

so you can see that in the worst case the algorithm takes time that depends on  $N^2$ . This worst case scenario occurs in the case that the array is already sorted, but it is sorted in the wrong direction, namely *from largest to smallest*.

In the “best” case, the array is already correctly sorted from smallest to largest. Then the condition tested in the **while** loop will be false every time (since  $a[i - 1] < tmp$ ), and so each time we hit the **while** statement, it will take a *constant* amount of time. This is the *best case* scenario, in the sense that the algorithm executes the fewest operations in this case. Since there are  $N$  passes through the **for** loop, the time taken is proportional to  $N$ .