At the end of last lecture, I showed how to represent a heap using an array. Let’s revisit the `add` and `removeMin` methods and express them using this representation.

**add(element)**

Suppose we have a heap with \( k \) elements which is represented using an array, and now we want to add a \( k+1 \)-th element. Last lecture we sketched an algorithm doing so. Here I’ll re-write that algorithm using the simple array indexing scheme. Let `size` be the number of elements in the heap. These elements 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.

```java
add(element)
{
    size = size + 1 // number of elements in heap
    heap[ size ] = element // assuming array has room for another element
    i = size

    // the following is sometimes called "upHeap"
    while (i > 1 and heap[i] < heap[ i/2 ]){
        swapElements( i, i/2 )
        i = i/2
    }
}
```

Here is an example. Let’s add `c` to a heap with 8 elements.

```
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)
```

**Building a heap**

We can use the `add` method to build a heap as follows. Suppose we have a list of `size` elements and we want to build a heap.

```java
buildHeap(list){
    create new heap array // size == 0, length > list.size
    for (k = 0; k < list.size; k++)
        add( list[k] ) // add to heap[ ]
}
```
Time complexity

How long does it take to build a heap in the best and worst case? Before answering this, let’s introduce some notation. We have seen the “floor” operation a few times. Recall that 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:

- \([x]\) is the largest integer that is less than or equal to \(x\). \([\ ]\) is called the floor operator.
- \([x]\) is the smallest integer that is greater than or equal to \(x\). \([\ ]\) is called the ceiling operator.

For any positive integer \(i \geq 1\), there is a unique integer – we call it level – such that

\[2^{\text{level}} \leq i < 2^{\text{level}+1}\]

or

\[\text{level} \leq \log_2 i < \text{level} + 1,\]

and so

\[\text{level} = \lfloor \log_2 i \rfloor.\]

In particular, if \(i\) is the index in the array representation of elements/nodes in a heap, then \(i\) is found at level \(\text{level}\) in the corresponding binary tree representation.

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. In this case, building a heap takes time proportional to the number of nodes \(n\). So, best case is \(\Omega(n)\).

What about the worst case? Since \(\text{level} = \lfloor \log_2 i \rfloor\), when we add element \(i\) to the heap, in the worst case we need to do \(\lfloor \log_2 i \rfloor\) swaps up the tree to bring element \(i\) to a position where it is less than its parent, namely we may need to swap it all the way up to the root. 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 \(O(n \log_2 n)\). Indeed this is a upper tight bound and so we could say that this worst case \(t(n)\) is \(\Theta(n \log n)\). (It is a bit confusing to hear a “worst case” stated as \(\Theta(n \log n)\) so in the lecture slides I only illustrated the \(O(n \log n)\) bound, i.e. upper asymptotic bound.)
removeMin

Next, recall the removeMin() algorithm from last lecture. We can write this algorithm using the array representation of heaps as follows.

```java
removeMin()
{
    heap[1] = heap[size]
    heap[size] = null
    size = size - 1
    downHeap(1, size)  // see next page
    return element
}
```

This algorithm saves the root element to be returned later, and then moves the element 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 element from a starting position in the array down to some maximum position in the heap. I will use this helper method in a few ways in this lecture.
downHeap(start, maxIndex){ // move element from starting position // down to at most position maxIndex
    i = start
    while (2*i <= maxIndex){ // if there is a left child
        child = 2*i
        if child < size { // if there is a right sibling
            if (heap[child + 1] < heap[child]) // if rightchild < leftchild ?
                child = child + 1
        }
        if (heap[child] < heap[i]){ // Do we need to swap with child?
            swapElements(i, child)
            i = child
        }
    }
}

This is essentially the same algorithm we saw last lecture. What is new here is that (1) I am expressing it in terms of the array indices, and (2) there are parameters that allow the downHeap to start and stop at particular indices.

**Heapsort**

A heap can be used to sort a set of elements. First, we take the elements and build a heap. At this point we are sure that the minimum element is the root. We then remove the minimum element from the heap, replace it by the last element and then downheap that element from the root. The size of the heap is reduced by 1 each time, and we use the freed slot in the array to store the removed element.

heapsort(list){
    buildheap(list)
    for i = 1 to size{
        swapElements( heap[1], heap[size + 1 - i])
        downHeap( 1, size - i)
    }
    return reverse(heap)
}

Note that after i times through the loop, the remaining heap has size - i elements, and the last i elements in the array hold the smallest i elements in the original list. So, we only downheap to index size - i.

The end result is an array that is sorted from largest to smallest. If we want to order from smallest to largest, we simply reverse the order of the elements. This can be done in $\Theta(n)$ time, by swapping $i$ and $n + 1 - i$ for $i = 1$ to $\frac{n}{2}$.
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 elements (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, ...) |
| w | u | l | k | f | e | d | b | a | (removed w, and done) |

Note that the last pass through the loop doesn’t do anything since the heap has only one element left (w in this example), which is the largest element. We could have made the loop go from $i = 1$ to $size - 1$. 
Faster algorithm for building a heap

We have seen a $O(n \log_2 n)$ algorithm for building a heap. I will next present algorithm that runs in time $\Theta(n)$.

Having a fast algorithm for building a heap would be useful if $n$ were large and if we wanted to change how keys/nodes were compared from time to time. We would rebuild the heap, based on the new “compare to” definition. For example, in an investment banking application, the comparison might be based on a prices of things that fluctuate over time.

The faster method for building a heap is based on the `downHeap()` method. The input is an array of size $n$ (with elements in any order). The output is a heap.

\[
\text{buildHeapFast(list)} \{
\text{create new heap array (size == 0, length > list.size)}
\text{for (k = size/2; k >= 1; k--)}
\text{downHeap( k, size )}
\}
\]

Here we are using a slightly more general `downHeap()` method than we used last lecture. Here we can `downHeap`, not just from the root node, but from any node. The only line that changed is the first one.

The `buildHeapFast` algorithm begins at node $k = n/2$ and works its way down to $k = 1$. Notice that since each of the nodes $\text{size}/2+1$ to $\text{size}$ is a leaf, each of these nodes is itself a heap, namely a heap with one element.

For each $k$, the number of swaps done by `downHeap()` is the height of the node $k$ in the tree. Thus the total number of swaps that we need to do is the total of the heights of the nodes in the tree. Next lecture I will show that this total is $\Theta(n)$.

Example

An initial arrangement of $n = 6$ keys is shown on the left. I show the state of the tree before the $k$th node is downHeaped, and the final state.

<table>
<thead>
<tr>
<th>$k=3$ (c)</th>
<th>$k=2$ (x)</th>
<th>$k=1$ (w)</th>
<th>final</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>w</td>
<td>w</td>
<td>b</td>
</tr>
<tr>
<td>/ \</td>
<td>/ \</td>
<td>/ \</td>
<td>/ \</td>
</tr>
<tr>
<td>x c --&gt;&gt;  x b --&gt;&gt;  p b --&gt;&gt;  p c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/ \</td>
<td>/ \</td>
<td>/ \</td>
<td>/ \</td>
</tr>
<tr>
<td>p r b</td>
<td>p r c</td>
<td>x r c</td>
<td>x r w</td>
</tr>
</tbody>
</table>

Some properties of complete binary trees

I begin today by considering some interesting properties of complete binary trees. At the end of the lecture, I will use these properties to show you a remarkably fast algorithm for building a heap. *The slides have many pictures which should help you see what’s going on in this lecture.*
Sum of the depths of all nodes

One quantity that often comes up in tree algorithms is the average depth of nodes. The average depth is just the sum of the depths of all nodes, divided by the number of nodes. So we will ask the more basic question, what is the sum of the depths of all nodes? We specifically address the case of complete binary trees. To simplify the math slightly, let’s assume that all levels of the tree are full, i.e. including level \( l = h \) which is the height of the tree. Then,

\[
n = 2^{h+1} - 1
\]

and so

\[
h = \log(n + 1) - 1.
\]

Let \( t(n) \) be the sum of the depths of all nodes, then (see the slides),

\[
t(n) = \sum_{i=1}^{n} \lfloor \log i \rfloor.
\]

Since we are assuming all levels are full, we can rewrite \( t(n) \) exactly as a function of \( h \) and solve:

\[
t(h) = \sum_{l=0}^{h} l 2^l
\]

\[
= \sum_{l=0}^{h} l x^l, \quad x = 2
\]

\[
= x \sum_{l=0}^{h} l x^{l-1}
\]

\[
= x \sum_{l=0}^{h} \frac{dx^l}{dx}
\]

\[
= x \frac{d}{dx} \sum_{l=0}^{h} x^l
\]

\[
= x \frac{d}{dx} \frac{x^{h+1} - 1}{x - 1}
\]

\[
= x \frac{(h+1)x^h(x-1) - (x^{h+1} - 1)}{(x-1)^2}
\]

\[
= 2 \left( (h+1)2^h - 2^{h+1} + 1 \right), \quad \text{since } x = 2
\]

\[
= 2 \left( (h+1)2^h - 2 \cdot 2^h + 1 \right)
\]

\[
= 2 \left( (h-1)2^h + 1 \right)
\]

\[
= (h-1)2^{h+1} + 2
\]

Substituting for \( h \), we get

\[
t(n) = (\log(n + 1) - 2)(n + 1) + 2.
\]

Thus, \( t(n) \) is \( \Theta(n \log n) \). The intuition here is that most of the nodes in the tree are near the leaves and since the height of the tree is \( \lfloor \log n \rfloor \), most of the leaves have depth which is either \( \lfloor \log n \rfloor \) or very close to it.
Sum of the heights of all nodes

Recall that the height of a node in a tree is the maximum path length from the node to a leaf. For a complete binary tree that has all the levels including \( h \) full, the height of every node at level \( l \) will be \( h - l \). I emphasize that \( h \) here refers to the height of the tree i.e. the height of the root node.

<table>
<thead>
<tr>
<th>height</th>
<th>number of nodes of that height</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h )</td>
<td>1 (namely the root of the tree)</td>
</tr>
<tr>
<td>( h-1 )</td>
<td>2 (namely the children of the root)</td>
</tr>
<tr>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td>( i )</td>
<td>( 2^{{h-i}} )</td>
</tr>
<tr>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td>0</td>
<td>( 2^h ) (or less, if last level is not full)</td>
</tr>
</tbody>
</table>

Define \( t(n) \) now to be sum of heights of all nodes. We write it first in terms of \( h \):

\[
t(h) = \sum_{l=0}^{h} (h - l) \ 2^l
\]

\[
= h \sum_{l=0}^{h} 2^l - \sum_{l=0}^{h} l \ 2^l
\]

\[
= h(2^{h+1} - 1) - (h - 1)2^{h+1} - 2, \quad \text{from above}
\]

\[
= 2^{h+1} - h - 2
\]

In terms of \( n \), we have

\[
t(n) = n - \log(n + 1)
\]

which is \( \Theta(n) \).

Notice that if there are \( n \) nodes in the heap, then the parent of the last node is \( \frac{n}{2} \). In particular, nodes \( \frac{n}{2} + 1, \ldots, n \) are leaves in the tree. We will use this fact in the following.

**ASIDE: Java’s PriorityQueue class**

The Java API defines a class `PriorityQueue<T>` where type `T` implements the `Comparable` interface. I will discuss what this means later, when we get to the object oriented design part of the course.

The `PriorityQueue<T>` class has several methods, including an `add` and `remove` method. You can check out the Java API for the details.

[https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html](https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html)

From the Java API: "This implementation provides \( O(\log(n)) \) time for the enqueuing and dequeuing methods (offer, poll, remove() and add); linear time for the remove(Object) and contains(Object) methods; and constant time for the retrieval methods (peek, element, and size)."