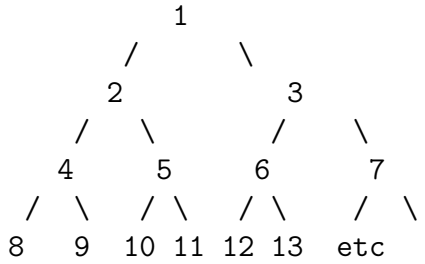
Implementing a heap using an array

Suppose we number the nodes of a heap by a level order traversal and start with index 1, rather than 0. This makes indexing slightly simpler (see below).



These numbers are NOT the elements stored at the node, rather we are just numbering the nodes so we can index them.

The above picture suggests a relationship between a node's index and its children's index. If the node index is i, then its children have indices $2i$ and $2i+1$. If this relationship is indeed true, then we can derive a similar relationship the index of a node and its parent: given a node index $j > 1$, then j can be written either $2i$ (if j is even) or $2i+1$ (if j is odd) where $i > 0$. Thus, the parent of any $j > 1$ is $\lfloor j/2 \rfloor$.

Verifying the parent/child index relationship

Let's make sure the above relationships hold by deriving them more formally (than just looking at a picture). The index i of any node at level l can be written

$$i = 2^l - 1 + m \tag{1}$$

for some m such that $1 \leq m \leq 2^l$. To see this, note there are

$$\sum_{i=0}^{l-1} 2^i = 2^l - 1$$

nodes in total in levels $0, \ldots, l-1$.

The index of the right child of i is the number of nodes up to level l, which is $2^{l+1} - 1$, plus the children of the m nodes at level l, which are from index 2^l to i. There are $2m$ such children. Thus, the right child of node i has index

$$i + (2^l - m) + 2m = (2^l - 1 + m) + 2^l - m + 2m = 2^{l+1} + 2m - 1$$

which is just $2i + 1$, as you can verify from Eq. (1), i.e.

$$2i + 1 = 2(2^l - 1 + m) + 1 = 2^{l+1} - 2 + 2m + 1 = 2^{l+1} + 2m - 1$$

Notes:

- In class, I only sketched out the above details.
- Do you need to memorize this for the Quiz/Final? No, you don't. But you should be able to understand the elements of the derivation, e.g. you should be able to tell me how many nodes there can be at a particular level in a binary tree, or how many nodes there can be up to l levels in a binary tree.

Building a heap

These indices suggest that we can use an array to store (references to) nodes in a heap. To keep the notation simple, we keep the array element 0 empty and start at index 1, as described above.

Last class we looked at how to build a heap. The idea was simple: you begin with an empty heap. Then you add all the elements of your collection into the heap, one-by-one. We did this by inserting a new last node in the heap and then swapping it with its parent if the parent's value was greater, and proceeding up the tree.

Here we consider how to implement the algorithm using an array. Suppose that elements $1, \ldots, j-1$ in the array define a heap, where $j > 1$. We wish to add element j to the heap. We do so by comparing the element at j to its parent's element. If its parent's element is greater, then we need to swap. We continue moving our element "up the heap" until it is less than or equal to its parent. At this point, the first j elements define a value heap. The `makeHeap` algorithm iterates this `upHeap` algorithm from $j = 2$ to the size of the array. *Here is the pseudocode. This was not given in class.*

```
ALGORITHM  upHeap(a, j) is
INPUT:  an array a whose elements 1,.. j-1 define a heap.
OUTPUT: an array whose elements 1,.. j define a heap (namely
        the above heap with element j inserted into it)

  child = j;
  while (child > 1) {
      parent := floor(child / 2)
      if (a[parent] > a[child])
         swap(a[parent], a[child])
         child = parent
      else
         return
  }
```

And here is an algorithm for making a heap which is based on `upHeap`.

```
ALGORITHM: makeHeap(a) is
INPUT:   an array of unsorted elements
OUTPUT:  a heap
  j = 2;
  while j <= a.size{
     upHeap(a, j);
     j++
  }
```

Example

We begin with an unordered sequence of elements in an array, and then we add them to the heap. The boundary between the heap and non-heap nodes are marked with `|`.

```
  1   2   3   4   5   6   7   8   9
-----------------------------------
| f   b   u   e   l   a   k   d   w
  f | b   u   e   l   a   k   d   w    (added f)
  b   f | u   e   l   a   k   d   w    (added b)
  b   f   u | e   l   a   k   d   w    (added u)
  b   e   u   f | l   a   k   d   w    (added e)
  b   e   u   f   l | a   k   d   w    (added l)
  a   e   b   f   l   u | k   d   w    (added a)
  a   e   b   f   l   u   k | d   w    (added k)
  a   d   b   e   l   u   k   f | w    (added d)
  a   d   b   e   l   u   k   f   w |  (added w)
```

Worst case is $\Omega(n \lg n)$

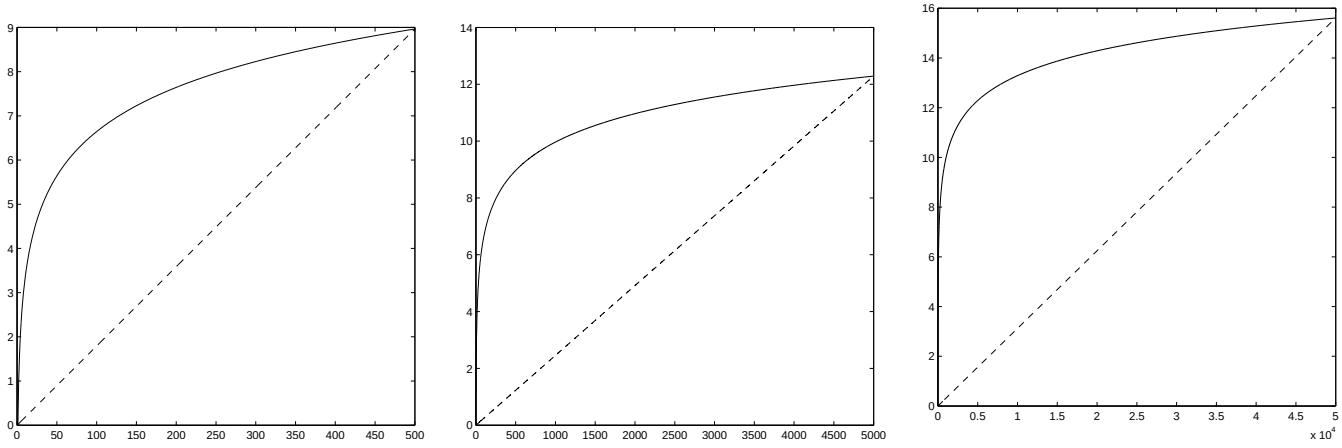
The above algorithm for building a heap runs in $\Omega(n \log n)$ in the worst case. Let's see why. Node i is at level l and by inspection we see that

$$2^{l-1} \leq i < 2^l - 1$$

and $\lfloor \lg i \rfloor = l$. Thus, when we "insert" a new element at node i, in the worst case we need to do $\lfloor \lg i \rfloor$ swaps which will bring the new element i to the root. (This worst case arises when the new node is less than all nodes at indices less than i.) Since we are adding n nodes in total, and since $\lfloor \lg i \rfloor \geq \lg i - 1$, we have (in the worst case) that the number of swaps is:

$$t(n) \geq \sum_{i=1}^{n} (\lg i - 1) = \sum_{i=1}^{n} \lg i - n.$$

The expression $\sum_{i=1}^{n} \lg i$ is the area under the solid curve ($\lg i$) in the figures below ($n = 500, 5000, 50000$).



This area under the solid curve is clearly greater than the area under the dashed line (a triangle). The area under the dashed line is $\frac{n \lg n}{2}$ since its is half the rectangle of width n and height $\lg n$.

Thus,

$$t(n) \geq \frac{n \lg n}{2} - n.$$

We can show $t(n)$ is $\Omega(n \lg n)$ by showing that there exists an n_0 and c such that

$$\frac{n \lg n}{2} - n > cn \lg n$$

for all $n \geq n_0$.

Proof not shown in class, but provided here for completeness..

$$\begin{aligned} & \frac{n \lg n}{2} - n > cn \lg n \\ \Leftrightarrow \quad & n(\frac{\lg n}{2} - 1) > cn \lg n \\ \Leftrightarrow \quad & \frac{\lg n}{2} - 1 > c \lg n \\ \Leftrightarrow \quad & \lg n \cdot (\frac{1}{2} - c) > 1 \end{aligned}$$

Choosing $0 < c < \frac{1}{2}$ we get $\frac{1}{2} - c = \epsilon > 0$, and so we just need to choose n_0 such that $\lg n_0 > \frac{1}{\epsilon}$. And we are done...

One final note

It is easy to see that $t(n)$ is $O(n \lg n)$ since

$$t(n) \leq \sum_{i=0}^{n} \lg i \leq n \lg n.$$

Thus, in the worst case, $t(n)$ is both $\Omega(n \lg n)$ and $O(n \lg n)$, it means that in the worst case $t(n)$ is $\Theta(n \lg n)$.