At the end of last lecture, I showed how to represent a heap using an array. The idea is that an 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 $i/2$.

ASIDE: General binary trees can be represented as arrays

You can always use an array to represent a binary tree if you want, by using the above indexing scheme. However, there are costs to doing so when the binary tree is NOT a complete binary tree, namely there maybe large gaps in the array. We would just have null references at these gap points. For example, below is a binary tree (left) and its array representation (right) where `-` in the array indicates null. Note that there is a `null` in the 0-th element. It is only there to make the parent-child indexing formula simpler to think about. If you were really implementing this array and the various methods, you could use the 0-th slot and adjust the parent-child index relationship appropriately.

```
        c
      /   \
     f     d
      \     \               -cfd-a-m------zp
       a     m
            / \
           z   p
```

Let's next revisit the `add` and `removeMin` methods and rewrite them in terms of this array.

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.

```
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"  -- see below

   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.

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

We can write this in a slightly different way.

```
buildHeap(list){
   create new heap array      //  size == 0,  length > list.size
   for (k = 1; k <= list.size; k++){
       heap[k] = list[k-1]       //  say list index is 0, .. ,size-1
       upHeap(k)
   }
}
```

where `upHeap(k)` is defined as follows.

```
upHeap(k){    //  assumes we have an underlying array structure
    i = k
    while (i > 1  and heap[i] < heap[ i/2 ]){
       swapElements( i, i/2 )
       i = i/2
    }
}
```

Are we sure that the buildHeap() method indeed builds a heap? Yes, and the argument is basic mathematical induction. Adding the first element gives a heap with one element. If adding the first k elements results in a heap, then adding the $k+1$-th element also results in a heap since the upHeap method ensures this is so.

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. 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:

- $\lfloor x \rfloor$ 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 elements/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

$$2^{level} \leq i < 2^{level+1}$$

or

$$level \leq \log_2 i < level + 1,$$

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

What about the worst case? Since $level = \lfloor \log_2 i \rfloor$, 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:

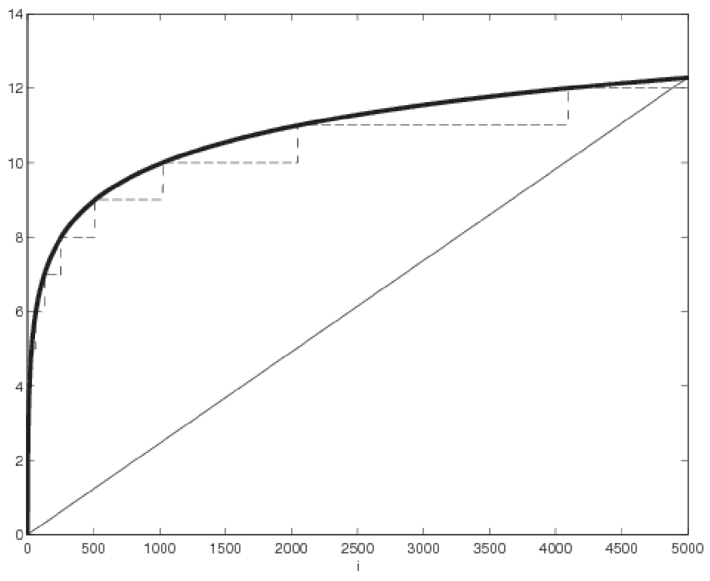
$$t(n) = \sum_{i=1}^{n} \lfloor \log_2 i \rfloor$$

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 $\Theta(n \log_2 n)$. I



removeMin

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

```
removeMin(){
    element = heap[1]           //   heap[0] not used.
    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 < maxIndex) {           // if there is a right sibling
         if (heap[child + 1] < heap[child])
                                        //  if rightchild < leftchild ?
         child = child + 1
      }
      if (heap[child] < heap[ i ]){   // swap with child?
         swapElements(i , child)
         i = child
      }
      else  break    // exit while loop
   }
}
```

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.