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 indexing scheme gives a simple relationship between a node’s index and its children’s index. If the node index is \( i \), then its children have indices \( 2i \) and \( 2i + 1 \). Similarly, if a non-root node has index \( i \) then its parent has index \( i/2 \).

**Implementing a heap using an array**

It is very common to use an array to represent a heap, rather than a binary tree, and to use the parent/child indexing scheme described above rather than using nodes and parent, leftchild, and right child references. Note that the heap is still a complete binary tree. We will still be talking about “parent”, “left child”, and “right child”. However, we will be using an array to represent the binary tree.

Also note that you can always use an array to represent a binary tree if you want, and use the above indexing scheme. However, the benefits of doing so are questionable when the binary tree is NOT a complete binary tree. For example, below is a binary tree (left) and its array representation (right) where - indicates null.

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

**Adding an element to a heap (array representation)**

Suppose we have a heap with \( k \) keys which is represented using an array and we want to add a new key. Last class we sketched an algorithm doing so. Now we re-write that algorithm using the simple array indexing scheme. Let \( a.size \) be the number of keys in the array. These keys are stored in array slots 1 to \( a.size \). (Recall that index/slot 0 is unused.)
add( a, key ){
    a.size = a.size + 1
    a[ a.size ] = key  // assuming array has room for another key
    upheap( a, a.size )
}

where the upHeap() helper method is:

upHeap(a, j){  // assumes
    i = j
    while (i > 1 and a[i] < a[ i/2 ]){
        swapKeys( i, i/2 )
        i = i/2
    }
}

Building a heap

We can use the above upHeap operation to make a heap as follows. Begin with an array a whose keys are in positions 1 to a.size. The keys are in no particular order. If we just take the key a[1], then this single key defines a heap. Then, given a heap with k keys, we upHeap the key k+1, giving us a heap out of the first k+1 keys. By induction, we end up with a heap of size n = a.size.

buildHeap(a){

    // INPUT:  an array a of unsorted keys, indexed from 1 to size (size > 1)
    // OUTPUT: the keys of the array rearranged so that the array is a heap

    for (k = 2; k <= a.size; k++)
        upheap( a, k );
}

Example

Let’s suppose the heap is initially empty and then we add items k,f,e,a,c,b in that order. Here is how the heap evolves.

```
   k     f   e   a   a   a
   /   / \  / \  / \  / \  
  k  k  f  e  f  c  f  c  b
   /   / \  / \  / \  / \  
 k  k  e  k  e  f
```
Notation: floor and ceiling (rounding)

Recall the following notation from lecture 21 slides.

- \( \lfloor x \rfloor \) 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.

Note that for any positive integer \( i \geq 1 \), there is a unique integer \( l \) such that

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

or

\[
l \leq \log i < l + 1,
\]

and \( l = \lfloor \log i \rfloor \). In particular, if \( i \) is the index in the array representation of keys/nodes in a heap, then \( l \) is the level where you find \( i \) in the corresponding binary tree representation.

Let’s have a look at the worse and base cases. If node \( i \) is at level \( l \) (the root is at level \( l = 0 \)), then \( l = \lfloor \log i \rfloor \). Thus, when we add node \( i \) to the heap, in the worst case we need to do \( \lfloor \log i \rfloor \) swaps up the tree to bring the new 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. Since we are adding \( n \) nodes in total, the worst case number of swaps is:

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

To visualize this sum, consider the plot below which show the \( \log i \) (thick) and \( \lfloor \log i \rfloor \) (dashed) curves up to \( i = 500 \) (left) and \( i = 5000 \) (right). The area under the dashed curve is the above summation. It should be visually obvious from the figures that

\[
\frac{1}{2} n \log n < t(n) < n \log n
\]

where the left side of the inequality is the area under the diagonal line from \((0,0)\) to \((n, \log n)\) and the right side \((n \log n)\) is the area under the rectangle of height \( \log n \). From the above inequalities, we conclude that \( t(n) \) is both \( O(n \log n) \) and \( \Omega(n \log n) \). Note the reasoning here. I used a figure to convince you that the the above inequalities are true for large \( n \), i.e. the upper and lower bound of \( t(n) \). I then used the inequalities to make the big O and \( \Omega \) statements about \( t(n) \).

The best case for building a heap is that the initial array is already a heap! In this case, the \( t(n) \) for the algorithm is proportional to \( n \). The \texttt{buildheap(a)} algorithm still goes through the “for loop”. For the \( k \)-th key, it checks if the parent of that key is smaller. The answer is “no” in each case, since we are considering the best case here. Thus, \( t(n) = n \) which is both \( O(n) \) and \( \Omega(n) \). I emphasize that the \( t(n) \) for the best and worst cases are different! You could think of them as \( t_{\text{worst}}(n) \) and \( t_{\text{best}}(n) \), if that helps.
removeMin and downHeap

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

```java
removeMin()
{
    n = a.size
    key = a[1]       // to be returned
    a[1] = a[n]
    downHeap(a, n-1)
    a.size = a.size - 1
    return key
}
```

This algorithm takes a heap as input, saves the root to be returned later, and then moves the key at position `a.size` to the root. The situation now is that node 2 and its descendants define a heap, and node 3 and its descendants define a heap. But the tree itself typically won’t satisfy the heap property.

The `downHeap` method is then applied to move the root key down.

```java
downHeap(a, n){
    i = 1
    while (2*i <= n){       // There is a left child
        child = 2*i
        if child < n {        // There is also a right child
            if (a[child + 1] < a[child]) // Is rightchild < leftchild ?
                child++
        }
        if (a[child] < a[i]){    // Do we need to swap with child?
            swapKeys(i, child)
            i = child
```
We already went over this algorithm 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. First, we build a heap. Then, we are sure that the minimum key is the root. The idea of heapsort is to remove the minimum key from the heap, replace it by the last key and then downheap that key from the root. (Note the size of the heap is reduced by 1.) We use the freed slot in the array (the last key) to store the removed (smallest) key.

**INPUT:** heap \( a[1 \ldots n] \) i.e. minimum is \( a[1] \)

**OUTPUT:** a sorted array \( a[1 \ldots n] \) (sorted from maximum to minimum)

```plaintext
for i = 1 to n{
    swapKeys( a[1], a[n+1 - i])
    downHeap( a, n-i)
}
```

Note that after \( i \) times through the loop, the remaining heap is of size \( n - i \) only, and the last \( i \) keys in the array stored the smallest \( i \) keys in the set.

**Note that the final array will be sorted from largest to smallest.** Is this a problem? Not at all, since we can reverse the keys in an array in \( O(n) \) time, i.e. 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 keys (on the right).

```
1 2 3 4 5 6 7 8 9
-------------------------------------
 a d b e l u k f w | (removed a, put w at root, ...)
 b d k e l u w f | a (removed b, put f at root, ...)
 d e k f l u w | b a (removed e, put u 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)
```