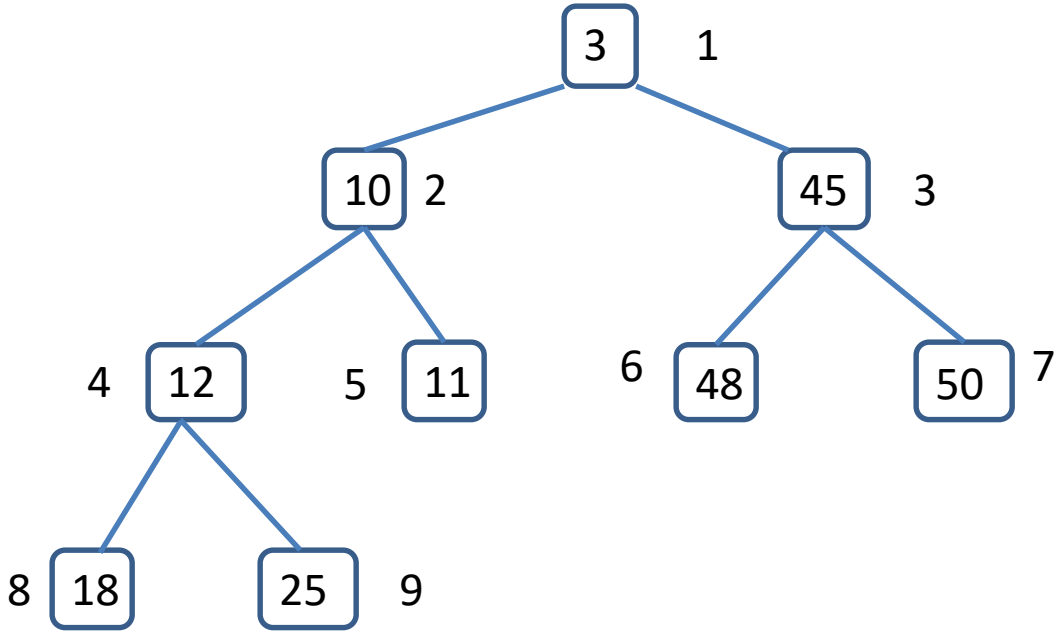
Priority Queues -- Implementamtion

See Chapter 21 of the text, pages 807-839.

Binary heaps are often implemented in arrays, using a convenient indexing system for trees. We put the root at index 1. The two children of the node at index n are at indices 2*n and 2*n+1. Alternatively, the parent of the node at index i is at index i/2. If the tree is *complete*, meaning that every level except the bottom is completely filled and the bottom level has entries filled from left to right, then there are no gaps in the array.

Here is a picture of a heap and its corresponding array. The index of each node is also indicated in the tree:

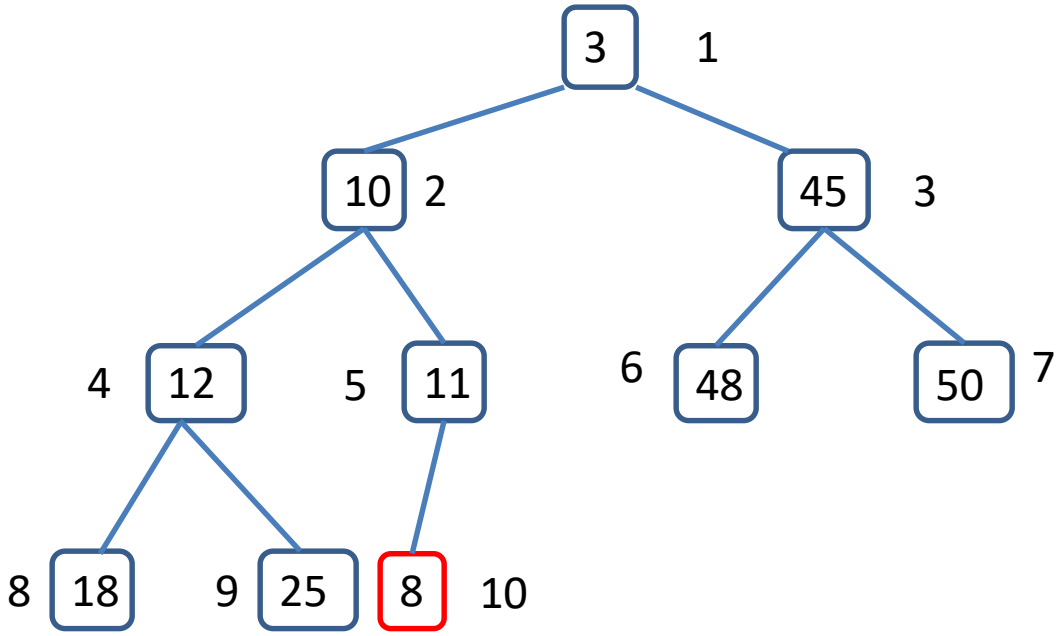


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	3	**10**	**45**	**12**	**11**	**48**	**50**	**18**	**25**	

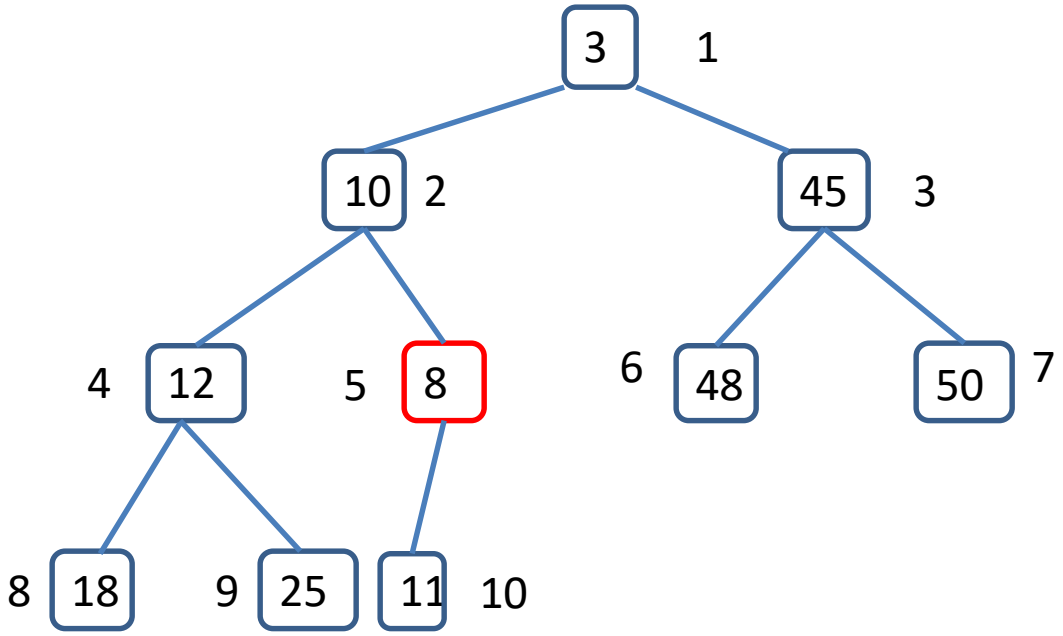
Note that a complete heap with N nodes uses entries 1 to N of an array of size N+1.

To insert an element into the heap we start by placing it at the next available spot. If the heap has n elements indexed from 1 to n, we put the new element at index n+1. If it has value greater than its parent node (index (n+1)/2), we are done. If not, we interchange it with its parent node and try again. The new value "percolates" up the tree until the heap property is satisfied.

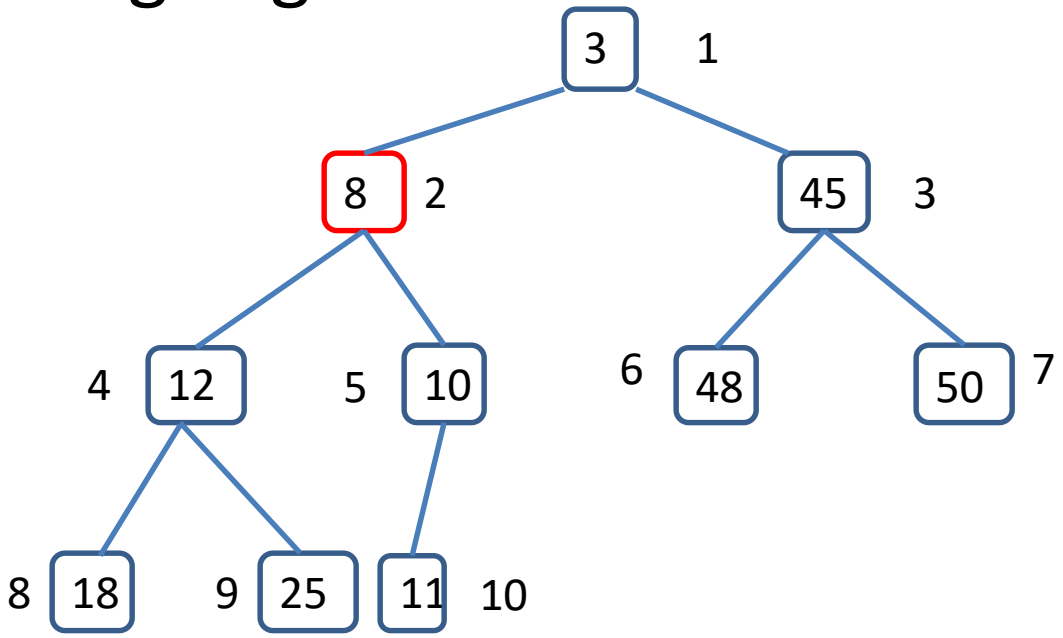
Here is an example. We add value 8 to the heap we just displayed. First we insert it as the next leaf:



The value 8 is less than that of its parent node so we interchange it with its parent:

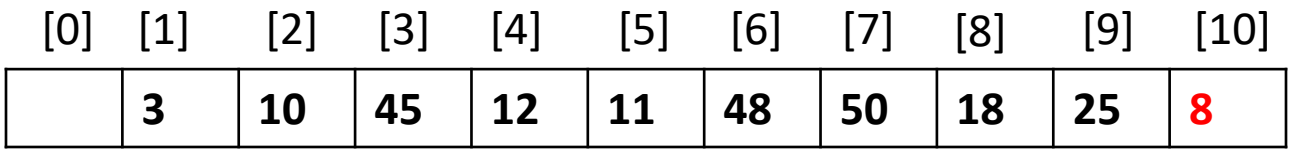


Its value is still less than its parent's so we interchange again:

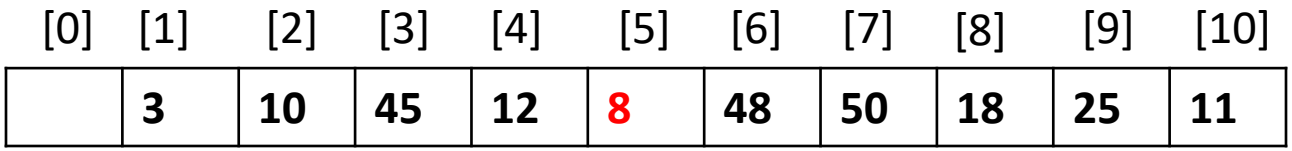


Our node now satisfies the heap property, so we stop interchanging and the entire tree is again a heap.

Here is this process in terms of the underlying array. We start by adding value 8 to the end:



The value we just inserted into index 10 is less than that of its parent at index 5, so we switch these values:



Our value at index 5 is still less than its parent at index 2 so we switch those:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	3	**8**	**45**	**12**	**10**	**48**	**50**	**18**	**25**	**11**

This now satisfies the heap property.

Here is code for adding a value x to a heap that is stored in array *nodes*:

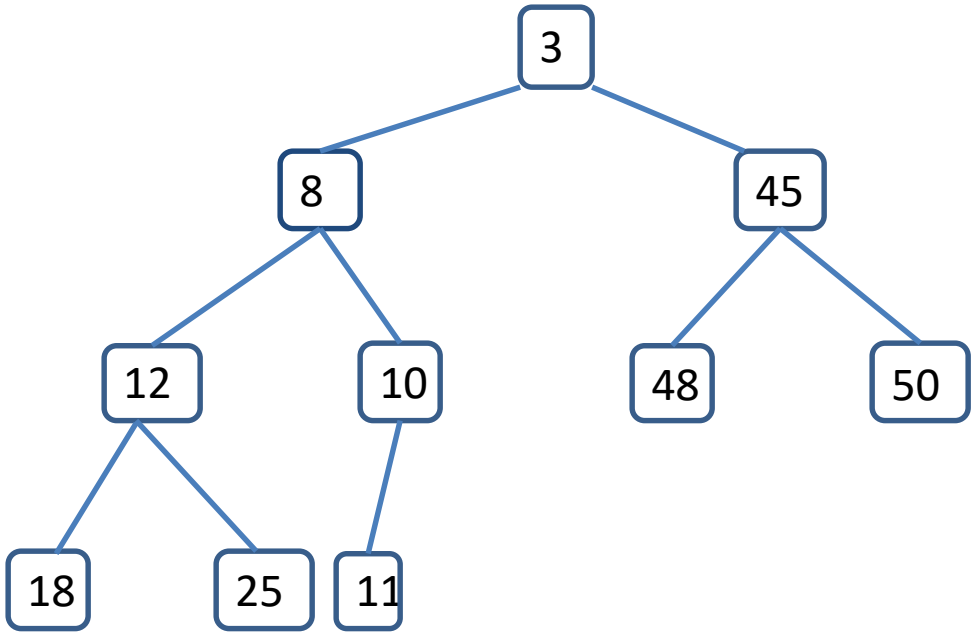
```
boolean add(E x) {
    if (size == CAPACITY)
            return false;

    int hole = size+1; // index where we might put x
    size += 1;
    nodes[hole] = x;
    percolateUp(hole);
    return true;
}
```

Here is code for percolateUp( ):

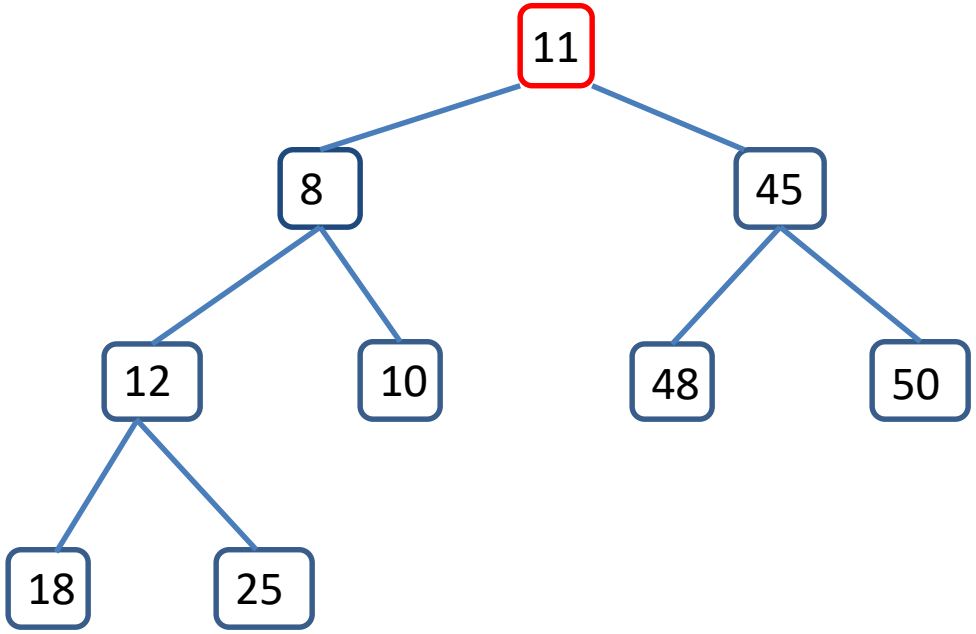
```
void percolateUp( int hole ) {
    nodes[0] = nodes[hole]; // guarantees we will stop when hole=1
    while (compare(nodes[hole],  nodes[hole/2])< 0 ) {
            swap(nodes, hole, hole/2);
            hole = hole/2;
    }
}

void swap( E [ ] nodes, int i, int j) {
    E temp = nodes[i];
    nodes[i] = nodes[1j];
    nodes[j] = t emp;
}
```

It is easy to find the smallest element; this is the first element of the array, or the root of the tree. We need an operation that removes the smallest element. In order to maintain a complete tree we must ultimately remove the last leaf, or last element of the array. We make a hole at the root and pass it down through the tree until we find where we can insert the value of the last leaf. We call this process "percolating down" the tree.

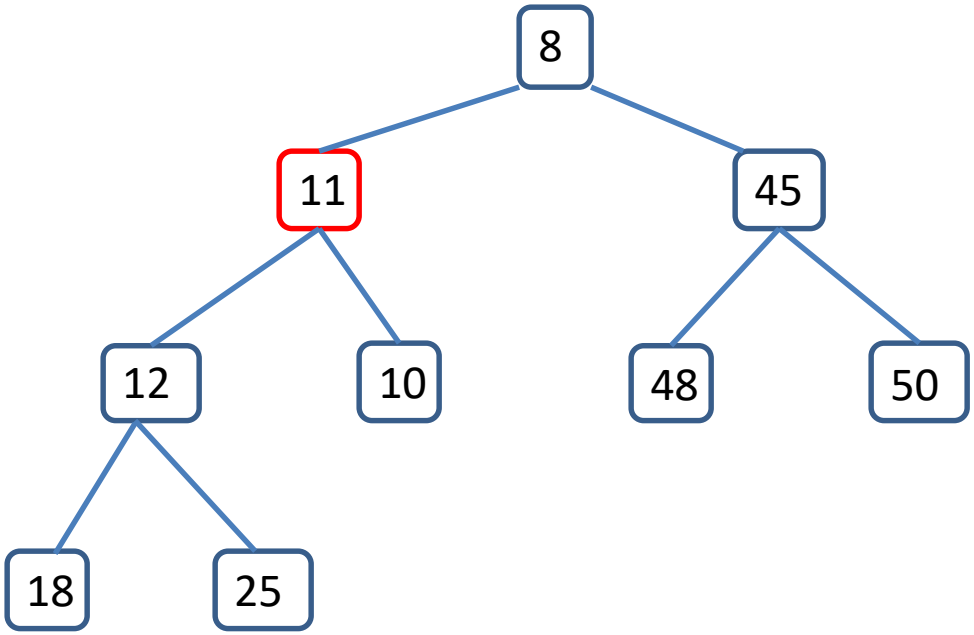
Graphically it looks like this. We start with a heap and want to remove the root.



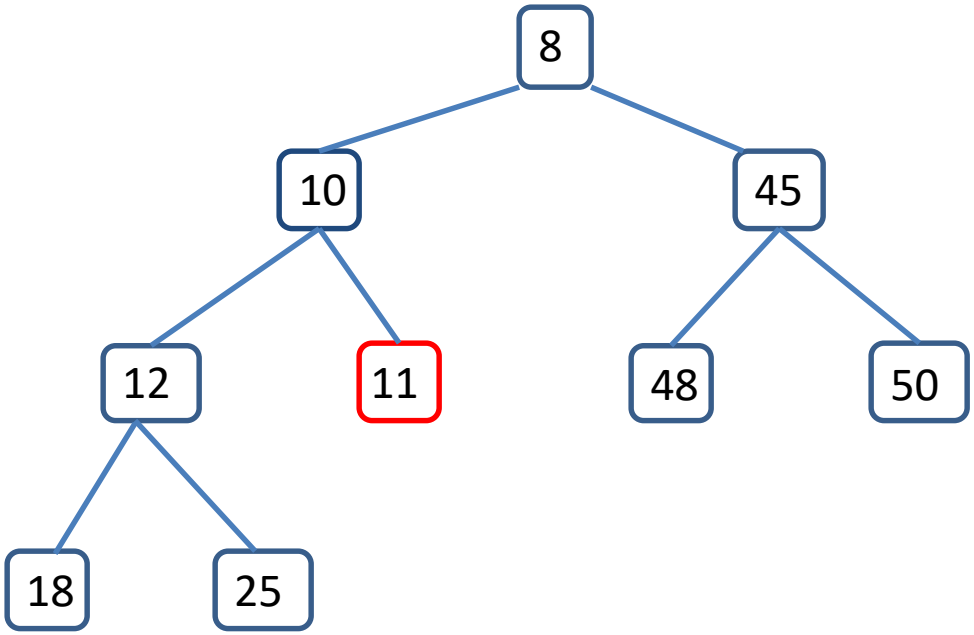
We make a hole at the root and put the value of the last leaf there, deleting the last leaf node.



The smaller of the two children of the hole has value 8; this is smaller than the value in our hole so we switch these items, moving the hole down:



Again the smaller of the two children of the hole is smaller than our leaf, so we move this smaller child into the hole:



This time the hole has no child so we are done.

Here is code for removing and returning the smallest value in the heap:

```
E removeMin( ):
        E min = nodes[1];
        nodes[1] = nodes[size];
        size -= 1;
        percolateDown(1);
        return min;
}
```

```
void percolateDown( int hole) {
    E value = nodes[hole]; // value being passed down
    while (2*hole <= size) {
        int smallChild; // index of smaller child
        if (2*hole == size)
            smallChild = size;
        else {
            if (compare( nodes[2*hole], nodes[2*hole+1]) < 0)
                smallChild = 2*hole;
            else
                smallChild = 2*hole+1;
        }
        if (compare(value, nodes[smallChild]) <=  0)
            break;
        else {
            swap(nodes, hold, smallChild);
            hole = smallChild;
        }
    }
}
```

Here is the really cool part. We can turn an array into a heap in linear time! We start at the leaves and work our way up. Of course, there is nothing to do at the leaves, they are already heaps. When we get to a node we will have already turned its leaves into heaps, so all that we need to do is to percolate the value at the node downward:

```
void buildHeap( ) {
        for (int i = size/2; i > 0; i--)
                percolateDown(i);
}
```