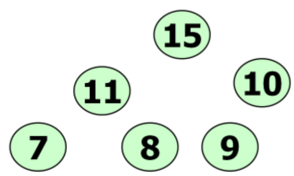
# THE HEAP

# heap

The heap is essentially an array-based binary tree with either the biggest or smallest element at the root.



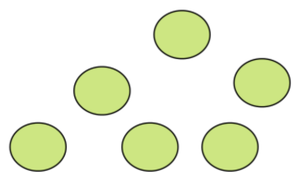
# heap

**Every parent in a Heap will always be smaller or larger than both of its children.**

**This rule will hold true for every level of the heap.**

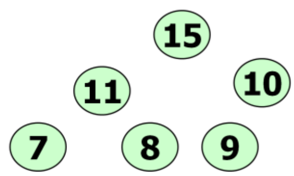
# complete tree

**In a complete tree, every level that can be filled is filled. Any levels that are not full have all nodes shifted as far left as possible.**

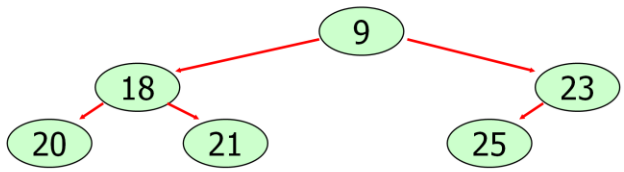


# heap

A Heap is a complete tree.

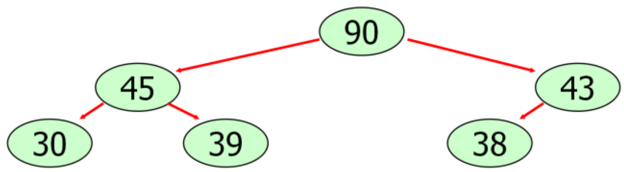


# min heap



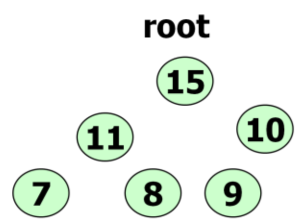

A min heap is a binary tree that has a root smaller than all of its children.

# max heap




A max heap is a binary tree that has a root larger than all of its children.

# heap




root

| | | | | | |
|---|---|---|---|---|---|
| 15 | 11 | 10 | 7 | 8 | 9 |

# array-based tree

**Because a heap will always be a <u>complete tree</u>, it makes sense to use an array to store the values.**

**root**

| | | | | | |
|---|---|---|---|---|---|
| 15 | 11 | 10 | 7 | 8 | 9 |

# array-based tree

root

| | | | | | |
|---|---|---|---|---|---|
| 15 | 11 | 10 | 7 | 8 | 9 |

left = i*2+1 right=i*2+2

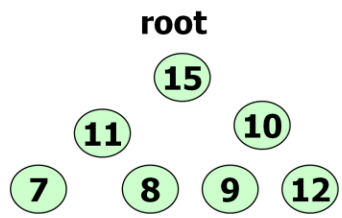
left child of root = [0 *2 + 1]
right child of root = [0 *2 + 2]

# add

**The easiest way to add a new item to a heap implemented with an array is to add the new value at the end of the array and then move the new item up the tree as far as it needs to go.**

**add will use swapUp to restructure the tree so that it remains a heap.**

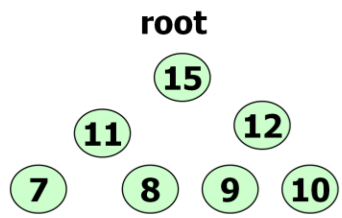
# swapUp




root

| | | | | | | |
|---|---|---|---|---|---|---|
| 15 | 11 | 10 | 7 | 8 | 9 | 12 |

# swapUp




root

| | | | | | | |
|---|---|---|---|---|---|---|
| 15 | 11 | 12 | 7 | 8 | 9 | 10 |

# swapUp

```
int bot = length-1
while( bot>0 )
{
   int parent = (bot-1)/2
   if list[parent] < list[bot]
      swap list[parent] and list[bot]
      bot = parent
  else
      stop
}
```

# swapUp

swapUp starts access at the bottom of tree

swapUp checks to see that the bottom index has not gone past the root of tree. The root is at index position 0.

Next, locate bottom's parent = (bot–1) / 2.

Check if bottom is larger than its parent
  If it is → swap bottom and parent

Finally, set bottom to parent and start the process over.

This method can be written iteratively or recursively.

# remove

**When you remove from a Heap, you are taking off the largest value or value with the highest priority.**

**You just take the top value off and save it.**

**Next, you move the last item in the tree to the root and move the new root down the tree as far as it can go.**

**remove will use swapDown to restructure the tree so it remains a heap.**

# swapDown

```
int root=0;
while(root<list.size())

  define max and left and right indexes

  if left child exists
     if right child exists
         find max
     else
         max is left
   else   stop

   if max > root
         swap
         root = max
   else  stop
```

# swapDown

swapDown starts access at the root

swapDown first generates the index values of the root's children.        root * 2 + 1        root * 2 + 2

Make sure root is less than bottom
Find the largest child
Determine if largest child is larger than root
    If it is → swap largest child and root

Root is set to the index of the largest child and the process starts over again.

This method can be written iteratively or recursively.

# Start work on the labs

# PQ Review

A PriorityQueue is a queue structure that organizes the data inside by the natural ordering or by some specified criteria.

The Java PriorityQueue is a min heap as it removes the smallest items first.

The Java PriorityQueue stores Comparables.

# PQ Methods

# PriorityQueue
## frequently used methods

| Name | Use |
| --- | --- |
| add(x) | adds item x to the pQueue |
| remove() | removes and returns min priority item |
| peek() | returns the min item with no remove |
| size() | returns the # of items in the pQueue |
| isEmpty() | checks to see if the pQueue is empty |

# add

```
PriorityQueue<Integer> pQueue;
pQueue = new PriorityQueue<Integer>();

pQueue.add(11);
pQueue.add(10);
pQueue.add(7);
out.println(pQueue);
```

**OUTPUT**

[7, 11, 10]

# remove

```
PriorityQueue<Integer> pQueue;
pQueue = new PriorityQueue<Integer>();

pQueue.add(11);
pQueue.add(10);
pQueue.add(7);
out.println(pQueue);
out.println(pQueue.remove());
out.println(pQueue);
```

**OUTPUT**

```
[7, 11, 10]
7
[10, 11]
```

Open
pqadd.java
pqremove.java

# isEmpty

```
PriorityQueue<Integer> pQueue;
pQueue = new PriorityQueue<Integer>();

pQueue.add(11);
pQueue.add(10);
pQueue.add(7);

while(!pQueue.isEmpty())
{
  out.println(pQueue.remove());
}
```

**OUTPUT**

```
7
10
11
```

# Open pqueueisempty.java

# Continue work on the labs