Chapter 6- Heaps

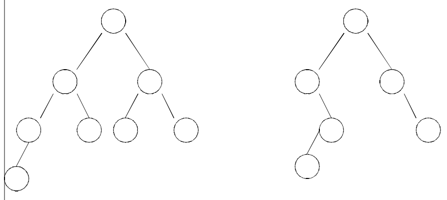
6.1 Heaps

Heaps are binary trees such that it has two properties

1) Structural requirement: It is a complete binary tree, with perhaps some right most leaves removed

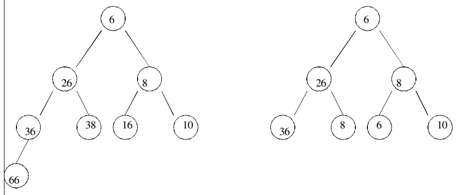
2) Data requirement: Data at each node is smaller than the data at its left and right children. These are called min-heaps. The max-heaps store larger data in the parent node.

Heaps



Which of these satisfy heap structure requirement and why?

Heaps



Which of these satisfy heap structure and data requirement? why or why nor?

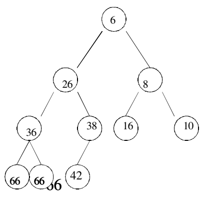
Heaps

The operations we would like to provide for the heap data structure are:
addElement(T obj)
removeMin()
findMin() // simply returns the element at root.

It would also inherit the following operations from Binary Tree
isEmpty()
count()
search(T elem)
iteratorInorder()
iteratorPreorder()
iteratorPostOrder()
iteratorLevelOrder()

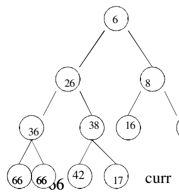
addElement

Suppose we wanted to insert 17 into the heap. Since heap must obey heap structure property, there is one location where a new node may be inserted, that is as right child of 38. But if the data is not right with respect to heap ordering, we need to swap with some data in the tree.



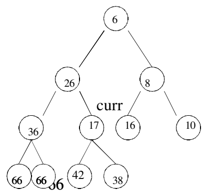
addElement

Let us start with curr as the new node and see if the data there is correct relative to its parent. If not swap the data at child and data at the parent.



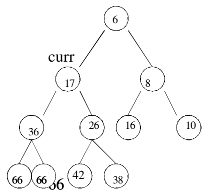
addElement

Resulting in the following tree. curr then moves up to the parent. But data at curr 17, violates data with its parent 26, so swap the data again and move the curr up the tree.



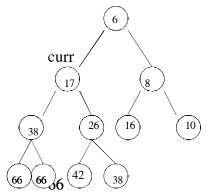
addElement

Resulting in the following tree. Since 17 is bigger than 6, we are done. How do insure that in this process we are not violating the relation between curr and its other child(in this case 17 and 36?) .



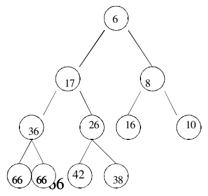
addElement

When we introduced the new node, how did we know whose child it should be?



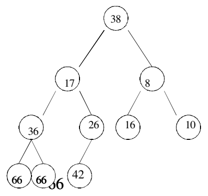
removeMin

To remove the minimum element(we will not provide remove operation for other elements), we simply return the element at root, however, we will remove the lastNode and reinsert the data at the last node into the root.



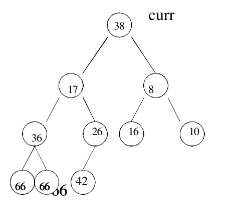
removeMin

Resulting in a tree like this. However the data at root is incorrect. But we will swap it with one its children thus propagating the problem down the tree.



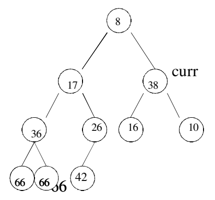
removeMin

Which one the children should we swap the data at curr with? We will have to choose the smallest of the two so that heap data ordering property is preserved.



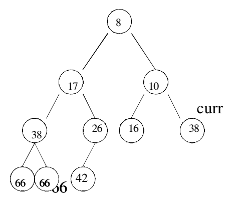
removeMin

Then we move the curr down to the node that we chose to swap. And repeat the process until data ordering is fixed or curr points to a leaf node.



removeMin

Why would this process work correctly?



Heap ADT

The heap node will be same as the Binary Tree node and also Heap ADT will be subclass of Binary Tree ADT.

```
public class Heap<T> extends BinaryTree<T>{
     BTNode<T> lastNode;

    public Heap(){
        super();
        lastNode = null;
    }
    public Heap(T rootElem){
         super(rootElem);
         lastNode = root;
    }
```

Heap ADT

```
  public void addElement(T elem){
     BTNode toInsert = new BTNode(elem);
     if (lastNode.lefchild != null){
              lastNode.parent.rightChild = toInsert;
              toInsert.parent = lastNode;
              lastNode = toInsert;
     }
     else{
              BTNode nextParent = getNextParent();
              toInsert.parent = nextParent;
              nextParent.leftchild = toInsert;
              lastNode = toInsert;
     }
     BTNode curr = toInsert;
      while (curr.parent!= null){
               if (curr.elem.compareTo(curr.parent.elem) < 0)
                          curr = curr.parent;
      }
} // complete this code by writing getNextParent()
```

Heap ADT

```
    public T  removeMin(){
      // write on your own
    }

} // end class HeapADT
```

Heap ADT- Time Complexity

addElement(T obj)	--- O(h)
removeMin()	--- -O(h)
findMin()	---- O(1)
isEmpty()	same as BT
size()	same as BT
search(T item)	same as BT
iteratorInorder()	same as BT
iteratorPreorder()	same as BT
iteratorPostOrder()	same as BT
iteratorLevelOrder()	same asBT
isEmpty()	same as BT

Priority Queues

We would like to have and ADT for adding items that have priority associated with each item. When we add to the queue it should be added according to priority(?) and when we ask for next item to remove we should get the item with the highest priority.

```
public class PriorityQueueNode<T>{
            private int priority;
            private T elem;
           public PriorityQueueNode( int priority, T elem){
                this.priority = priority;
                this.elem = elem;
           }
 //setters and getters for priority and element
         public int compareTo(PriorityQueueNode n){
                if(this.priority > n.getPriority())
                       return 1;
               else if ( this.priority = n.getPriority())
                       return 0;
               else return -1;
}
```

Priority Queues

```
 public class PriorityQueue<T>   extends Heap<T>{

          public PriorityQueue(){
              super();
          }

          pubilc void addElement( T object, int priority){
                    PriorityQueueNode pn = new PriorityQueueNode(object, prority);
                    super.addElement(node);
          }

          public T removeNext(){
                     PriorityQueueNode<T> temp = super.removeMin();
                      return temp.getElement();
          }

} // end PriorityQueue Class.
```

Priority Queues- Time Complexity

public PriorityQueue() --- O(1)

pubilc void addElement(T object, int priority) --- O(h)

public T removeNext() -- O(h)

Is this better than implementing Priority Queues with linked lists or arrays?

Exercises

1. What are the differences between heaps and binary search trees?

2. What happens if the heap is a complete binary tree and then addElement is called. Demonstrate the answer with an example of heap of depth 3.

3. Draw a heap that results from adding the following elements in the given order.
   34 45 3 87 65 32 1 12 17

4. Implement the Stack data structure using heaps

5. Implement Queue data structure using heaps.