Class Notes CS 3137

1 Priority Queues

- *Priority Queue* - *PQ* is a queue that is ordered by priority - a value stored in the node that allows us to determine which node precedes another node, which establishes a priority ordering.
- Priority Queues have the following functions associated with them:
  - Initialize PQ to be empty
  - Test whether PQ is empty
  - Test whether PQ is full
  - Insert a new item into the PQ,
  - Delete item of highest priority
- There are many ways to implement PQ's. If we use an array, we can just insert any item into the PQ at the end. Then a delete will need to search the entire array to find the highest priority item. Cost: Insert: $O(1)$, Delete: $O(N)$.
- If we do inserts into the array to preserve priority ordering, we will need to:
  - Find the spot in the array where the new insertion occurs
  - Shuffle all the array elements to adjust the array for the new element
- Cost of an insert is now $O(N)$ - we may have to shuffle all the elements if insertion is at beginning of array. Delete is now $O(1)$ since the first element in the array is the highest priority item
- The same analysis applies if we use a linked list, sorted by priority. Insertion will be $O(N)$ and deletion $O(1)$.

2 Heaps

- Can we improve on this time complexity for a PQ?
- We can by using a *Heap*. A Heap is a complete binary tree with values stored in its nodes such that no child has a value smaller then the value of its parent. Put another way, every node has a smaller value than its children. This is also called a *Min-Heap* since the smallest value must by definition be stored at the root (can you prove this must be so?).

- A *Max-Heap* just reverses the idea: no child has a value larger than its parent, and the root contains the largest value.

3 Array Implementation of Heaps

- Although we are talking about using a binary tree to create a heap, there is a particularly easy and efficient way to store complete binary trees in arrays.
- In this representation, we use array positions beginning with 1 (not zero). The root will always be stored in array position 1. The two children of the root will be in array positions 2 and 3.
- Following this idea, every tree node at array position i will have its children at array positions $2 * i$ and $2 * i + 1$.
- Because we are using arrays, we need to allocate the maximum amount of space at compile time. This implementation will reserve space for each node having 2 children, whether they do or not.
- To store a *complete* binary tree in an array we need to keep track of the largest array position that is being used (N). We can summarize how to access tree members with this table:

To Find:	Use:	Provided:
Left Child of A[i]	A[2*i]	2*i $\leq$ N
Right Child of A[i]	A[2*i+1]	2*i+1 $\leq$ N
Parent of A[i]	A[i/2]	i>1
Root	A[1]	A is nonempty
Whether A[i] is a leaf	True	2i > N

- Given the array implementation, we can now write access functions to perform insertions and deletions into the heap, preserving the PQ ordering. We will use a min-heap for our example.
- INSERT: the trick here is to add the new item to the end of the array, and "bubble" or "percolate" it up towards the root if it is of smaller key value than its parent node.
- DELETEMIN: Remove the root of the tree (highest priority item). We now need to readjust the tree to keep it in heap order. To do this, place the last item in the array at the root, and "bubble" it down towards the leaf nodes until it reaches the spot it belongs in according to min-heap ordering.

- The cost of an insert operation is $O(LogN)$ since we may possibly have to bubble the new item up the height of tree, which is $Log(N)$.
- The cost of a delete operation is $O(1)$ to remove the root (highest priority item) and $O(LogN)$ to readjust the heap after we remove the root. Remember that in a DELETEMIN, we replace the root with the last item in the heap and bubble it down (possibly) the height of tree, which is $Log(N)$.
- A simple sorting technique that takes advantage of the heap idea is *Heapsort*. Essentially, to sort N items, $A_1 \ldots A_n$, you do the following:

```
for(i=1; i<=N; i++)
   insert item i into Heap

for(i=1; i<=N; i++)
   Deletemin from Heap and readjust Heap
```

- Intuitvely, the cost of this is $O(N\,Log(N))$, since we have to do N insertions N deletions at cost $Log(N)$ each.
- There is an interesting observation about turning an array of elements into a heap - a *Heapify* operation. In an array of N elements, about half the elements are leaves. Element $N/2$ (integer division) is the last element that has a child - why?
- We can exploit this to heapify an array by only bubbling down the first $N/2$ elements of the array, which speeds the process up.

4 JAVA Implementation of Heaps

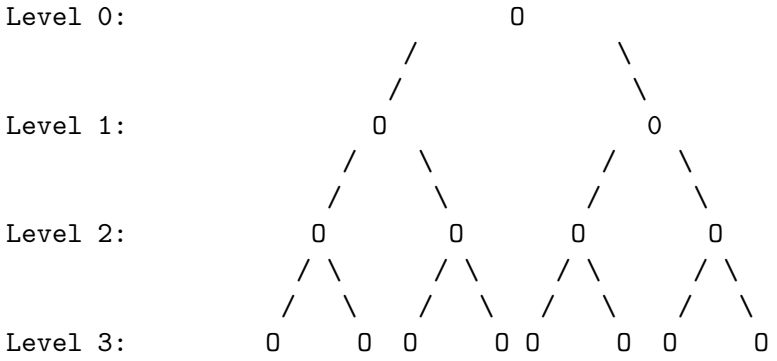
Below is code for Heap operations. In the code below I use the java.lang implementation of the interface Comparable which is compatible with Java 1.2.

```
//This program  generates a set of random integers and inserts
//them into a min-heap and then returns the numbers in order

import java.util.Random;
public class MyBinaryHeap
{
public MyBinaryHeap( int capacity )
{
     currentSize = 0;
     array = new Comparable[ capacity + 1 ];
}

public void insert( Comparable x )
{
     // percolate up
    int hole = ++currentSize;
    for( ; hole > 1 && x.compareTo( array[ hole / 2 ] ) < 0; hole /= 2 )
         array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;
}

public Comparable findMin( )
{
     if( isEmpty( ) )
         return null;
     return array[ 1 ];
}

public Comparable deleteMin( )
{
     if( isEmpty( ) )
        return null;
     Comparable minItem = findMin( );
     array[ 1 ] = array[ currentSize-- ];
     percolateDown( 1 );
     return minItem;
}

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

```
}

public boolean isEmpty( )

{    return currentSize == 0;  }

private void percolateDown( int hole )
{
   int child;
   Comparable tmp = array[ hole ];

   for( ; hole * 2 <= currentSize; hole = child ) {
      child = hole * 2;
      if( child != currentSize &&
             array[ child + 1 ].compareTo( array[ child ] ) < 0 )
         child++;
      if( array[ child ].compareTo( tmp ) < 0 )
         array[ hole ] = array[ child ];
      else      break;
    }
    array[ hole ] = tmp;
}

private int currentSize;      // Number of elements in heap
private Comparable [ ] array; // The heap array

public static void main( String [ ] args )
{
   int NUMINSERTS = 12;
   int MAXINTEGER = 3456;
   int i;
   MyBinaryHeap h = new MyBinaryHeap( NUMINSERTS );
   Random generator= new Random();
   for( i = 1 ; i <= NUMINSERTS; i ++) {
      int x= generator.nextInt(MAXINTEGER);
      System.out.println("Random number inserted is " + x);
      h.insert( new Integer( x ) );
   }
   for( i = 1; i <=NUMINSERTS; i++ )
      System.out.println( "min is " + h.deleteMin() );
}
}
```

5 Cost of Creating a Heap

- We can analyze the cost of putting an array into heap order. Remember, we can always represent a full or complete binary tree with an array implementation.
- Start with a full binary tree with $K = 4$ levels and $N = 2^K - 1$ nodes, and level $K - 1$ is the maximum level.



- To make this into a heap, we will use a method that starts at the bottom of the tree and will "bubble down" every node that isn't in heap order.
- Let index i = 1,2,3,4. The number of nodes at each level $K - i$ is 2^{K-i}.. In the picture above, all 2^{4-1} nodes at level $4 - 1$ are leaves. This means that these nodes are already in heap order - they are bigger than their children (they don't have any children!) and we don't have to bubble them down.
- However, nodes at the other levels as we move up the tree from the deepest level can bubble down to satisfy the heap priority. But they only have to move a few levels, not necessarily the maximum depth of the tree.
- So, the 2^{K-2} nodes at depth $K - 2$ only have to bubble down 1 level. In the picture above, all 2^2 nodes at level 2 only have to bubble down 1 level. Similarly, all 2^1 nodes at level 1 only have to bubble down 2 levels in the tree. Finally the single 2^0 node at level 0 needs to bubble down 3 levels.

index	nodes	heap level	bubble down levels
i	2^{K-i}	K-i	i-1
1	8	3	0
2	4	2	1
3	2	1	2
4	1	0	3

- We can summarize this as follows:

  Let $i = 1, \ldots, K$. The 2^{K-i} nodes at depth $K-i$ only have to bubble down $i-1$ levels

- So, to create a heap, we can count the maximum number of bubble downs possible, which is:

$$Max. \# of\ Moves\, to\, make\, a\, Heap\ =\ \sum_{i=1}^{K} 2^{K-i}(i-1)$$

$$\sum_{i=1}^{K} 2^{K-i}(i-1)\ =\ 2^{K-1}\cdot 0 + 2^{K-2}\cdot 1 + 2^{K-3}\cdot 2 + \ldots + 2^{0}\cdot(K-1)\ =\ \sum_{i=0}^{K-1} 2^{i}(K-(i+1))$$

  This becomes:

$$K\ \sum_{i=0}^{K-1} 2^{i}\ -\ \sum_{i=0}^{K-1} 2^{i}(i+1)$$

  The first terms sum is just $K \cdot (2^K - 1)$ and the second term's sum is $(K-1)\,2^K + 1$ (you can prove this by induction). The two sums add up to $2^K - K + 1$ Since the number of nodes $N = 2^K - 1$, we can say that this is an $O(N)$ operation

- So here is a heapify function....

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

- So, to analyze the Heapsort algorithm, we create a heap from an unordered array at cost N, and once we create the heap, we need to perform N deletions at cost $Log(N)$ for each deletion, yielding a total cost of $N\ +\ N\,Log(N)$ which is $O(N Log(N))$.