Chapter 6 – Heaps and Priority Queues

The Priority Queue ADT – is not a tree, but may be implemented as a type of tree, called a heap

An ADT that stores a collection of prioritized elements that support:

- Arbitrary element **insertion** – **insert** in any order
- **Removal** of elements in order of priority – always extract the one with the highest priority (which could be the lowest value, depending on how priorities are numbered). We'll talk about **deleteMin** in this text, to be consistent with author.
- Each element is identified by a key, which serves to name its priority – keys are not unique, several objects might have the same priority (and thus same key)
    - Must be a clear ordering among keys; given multiple keys, which has priority
    - Comparator - a class of objects external to the key that can provide comparison rules. (Java has a Comparator interface which require implementation of a **compare** method.) Defined later in these notes.
    - keys may be composite

Most basic operations of the Priority Queue ADT:

insert( )
deleteMin( )

Ancillary methods:

size( )
isEmpty( )
min( )

The main operations are **deleteMin**( ) and **insert**( ), so the physicial data structure chosen for implementation should optimize efficiency based on those operations.

Based on the fact that **removeMin**( ) needs to be an efficient operation, one can think of sorting data by inserting it into a Priority Queue and then subsequently removing all items from the queue, in order by key.

Three possible physical implementation schemes for a Priority Queue:

1. An unordered list – insert is cheap O(1), but removeMin is expensive O(n); to sort: n inserts is O(n), followed by n removeMins $O(n^2)$ – selection sort algorithm
2. An ordered list – insert is expensive O(n), but removeMin is cheap O(1); to sort n inserts is $O(n^2)$, and n removeMins is O(n) – insertion sort algorithm
3. Binary search tree – insert of O(log n), removeMin is O(log n)

The (Binary) **Heap** data structure: supports both insertions and removeMins in log n time (worst case): READ "Thanks Heaps!" article posted on piazza.

- Two properties
  1. structure property
     - Complete binary tree – has structural property: all leaf nodes on two levels, last level filled from left to right. Tree as short as possible.
       - Height of tree minimized – fewer nodes from root to leaves.
       - This means that we can store the heap, efficiently, in an array
  2. Heap-Order property – a relational property among a parent and its children
     - the key at the parent is always smaller than or equal (in terms of relationship) to the key of either child; on a path from root to leaf nodes, keys are non-decreasing
     - root's key is always "smallest"
- An array is a very efficient implementation for a heap
  - Store root at array[1], so
  - The children of any node at index *i*, are at array[2*i] and array [2*1+1]
  - Quick to find children, given index of parent; quick to find parent, given index of child
  - Array provides direct access

Priorities (keys) must be entities that are **comparable** (a total order). While most examples looked at have keys that are integers (and we all understand how integers are comparable), keys might be of other types (and objects might need to be compared in different ways different times).

Implementing a priority queue with a heap:

- Heap – as defined previously, each node contains a key (which represents its priority)

See author's implementation, skeleton page 228.

Homework – bring ready to turn in at next class.

Page 263:

6.2 For part b, see linear algorithm on page 235, in buildHeap section)

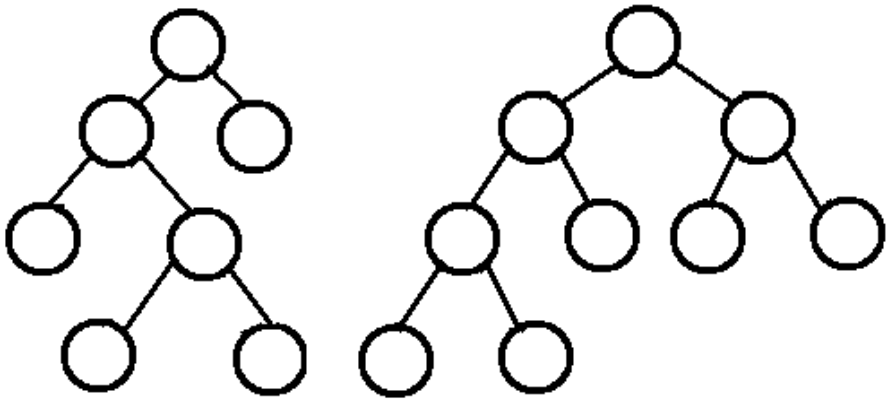
6.3

Defintions:

Full binary tree - is a tree in which every node other than the leaves has two children. Or, perhaps more clearly, every node in a binary tree has exactly 0 or 2 children.

Complete binary tree - is a binary tree in which every level, *except possibly the last*, is completely filled, and all nodes are as far left as possible.

Image from: http://www.cs.cmu.edu/~adamchik/15-121/lectures/Trees/trees.html



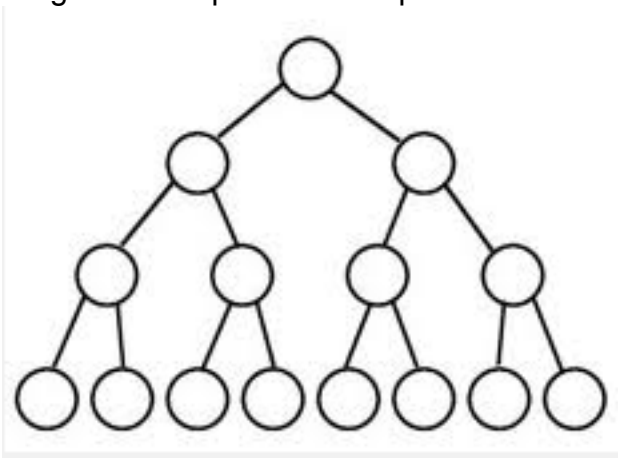
full tree

complete tree

This tree also **full**.

Perfect binary tree - is a ***full*** *binary tree* in which all *leaves* are at the same *depth* or same *level*, and in which every parent has two children. A perfect binary tree is also **complete**.

Image from: http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/FullvsComplete.html



Chapter 6 notes –

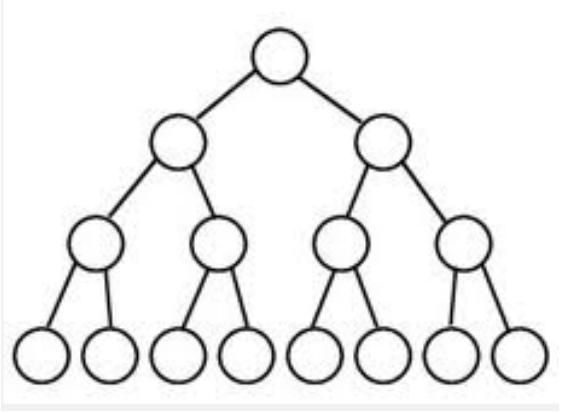
insert –

- on average O(1), experiments show that 2.607 comparisons required so on average the insert moves elements up 1.607 levels
    - so to insert n elements, average time is O(n)
- worst-case O(log n) – why?
    - so to insert n elements, worst case time is O(n log n)

buildHeap – the worst case here is O(n), we'll see why

- how it works: dump all values into an array, indexing starting at 1; conceptualize as binary tree, then

```
for i ← currentSize/2 downTo 1
    percolateDown(i)
```



Want to find worst case – upper bound – for comparisons required using buildHeap.

- Worst case – every node is originally positioned as far away from its original spot as possible – every node will move all the way down to a leaf (that is will move down the entire height of the subtree it roots).
- The maximum number of comparisons is 2 times the sum of all heights in the tree
-

Sum of height of nodes in tree above is 3 + 4 + 4 + 0 = 11

- Root has height 3
- Nodes at level 2 have height 2, sum of heights is 2x2 =4
- Nodes at level 3 have height 1, sum of heights is 4 x 1 = 4
- Leaf nodes have height 0, sum of heights is 8 x 0 = 0

The maximum number of comparisons that will be done in buildHeap is the sum of the heights of all nodes * 2.

Sum of heights, where h represents height, is 2^{h+1} -1 – (h + 1). Apply to sample tree above with height h = 3:

2^{3+1} – 1 – (3 + 1) = 2^4 -1 – (4) = 16 – 5 = 11.
Sum of heights is 11, make at most 11 * 2 = 22 comparisons for tree with 2^{3+1} – 1 = 15 nodes.