STUDENT OUTLINE

Lesson 41 – Priority Queues

INTRODUCTION: In this lesson we consider priority queues. A priority queue is essentially a list of items, each associated with a priority. In general, different items may have different priorities and we speak of one item having a higher priority than another. Given such a list we can determine which is the highest (or the lowest) priority item in the list. Items are inserted into a priority queue in any arbitrary order. However, items are withdrawn from a priority queue in order of their priorities starting with the highest priority item first.

The key topics for this lesson are:

A. Priority Queues
B. Heaps
C. Heap Deletion and Insertion
D. Storage of Complete Trees
E. The `PriorityQueue` Interface

VOCABULARY:

COMLETE TREE	HEAP PROPERTY
HEAP	PRIORITY QUEUE
HEAPSORT	

DISCUSSION: A. Priority Queues

1. Often the items added to a queue have a priority associated with them: this priority determines the order in which they exit the queue - highest priority items are removed first. In this curriculum guide, we will follow the convention that the smallest value has the highest priority.

2. For example, consider the software that manages a printer. In general, it is possible for users to submit documents for printing much more quickly than it is possible to print them. A simple solution is to place the documents in a FIFO queue. In a sense this is fair, because the documents are printed on a first-come, first-served basis.

   However, a user who has submitted a short document for printing will experience a long delay when much longer documents are already in the queue. An alternative solution is to use a priority queue in which the shorter a document, the higher its priority. By printing the shortest documents first, we reduce the level of frustration experienced by the users. In fact, it can be shown that printing documents in order of their length minimizes the average time a user waits for her document.

© ICT 2003, www.ict.org, All Rights Reserved
Use permitted only by licensees in accordance
with license terms (http://www.ict.org/javalicense.pdf)

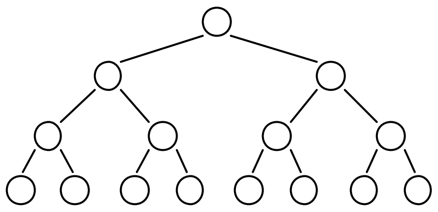
3. As we have seen, we could use a tree structure - which generally provides O(log n) performance for both insertion and deletion. Unfortunately, if the tree becomes unbalanced, performance will degrade to O(n) in worst cases. This will probably not be acceptable when dealing with dangerous industrial processes, nuclear reactors, flight control systems and other life-critical systems.

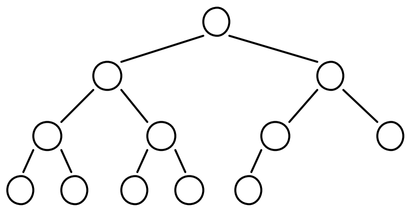
4. There is a structure that will provide guaranteed O(log n) performance for both insertion and deletion: it's called a *heap*.

B. Heaps

1. Heaps are based on the notion of a *complete tree*. A binary tree is called *completely full* if all its levels are filled with nodes. A binary tree is completely full if it is of height *h*, and has 2^h-1 nodes. Each level contains twice as many nodes as the preceding level.

2. A binary tree is called *complete* if it has no gaps on any level. The last level may have some leaves missing on the right as shown below:


Full tree


Complete tree

3. A heap is a binary tree that satisfies two conditions:

   a. It is a complete tree
   b. The value in each node does not exceed any value in that node's left and right subtrees.

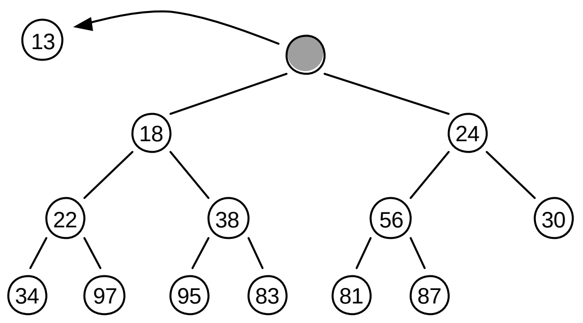
   Heaps are allowed to have more than one data item with the same value, and values in the left subtree do not have to be ranked lower than values in the right subtree.

4. A heap can be used as a priority queue: the highest priority item is at the root and is trivially extracted. But if the root is deleted, we are left with two sub-trees and we must efficiently re-create a single tree with the heap property. The value of the heap structure is that we can both extract the highest priority item and insert a new item in O(log n) time.

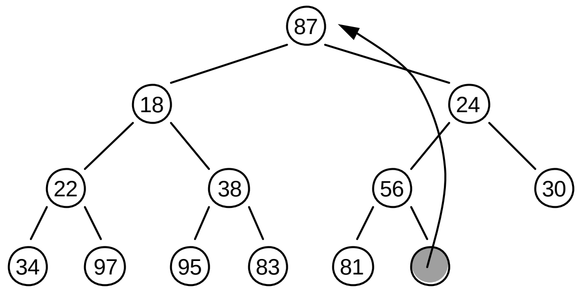
© ICT 2003, www.ict.org, All Rights Reserved
Use permitted only by licensees in accordance
with license terms (http://www.ict.org/javalicense.pdf)

C. Heap Deletion and Insertion

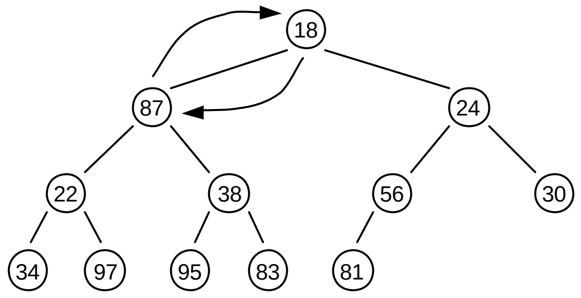
1. Removing an item from a priority queue is straightforward if the queue is represented by a binary heap. The next item to leave the queue will always be the item at the top (root) of the heap.



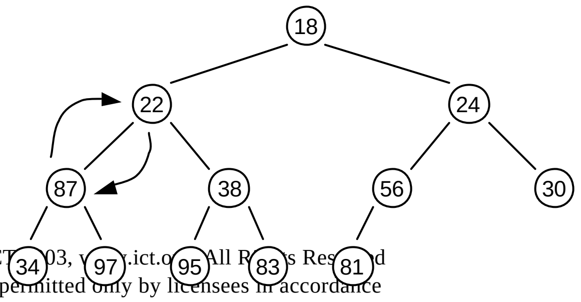
2. The shape of the heap is restored by removing the last leaf and placing it into the root. For the heap shown below, the position that must become empty is the one occupied by the 87. This is placed in the vacant root position.



3. This has violated the condition that the root must be greater than each of its children. To repair the order, we apply the "heapify" procedure in which the value from the root moves down the heap until it falls into place.



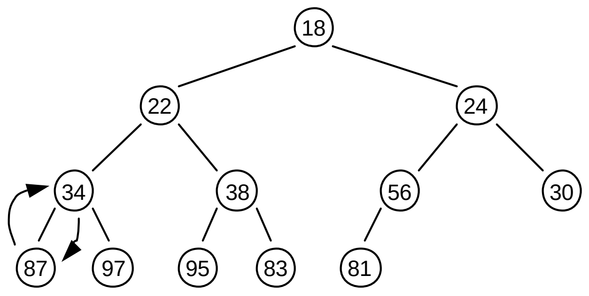
4. At each step down the value 87 is swapped with its smaller child.



© ICT 2003, www.ict.org, All Rights Reserved
Use permitted only by licensees in accordance
with license terms (http://www.ict.org/javalicense.pdf)

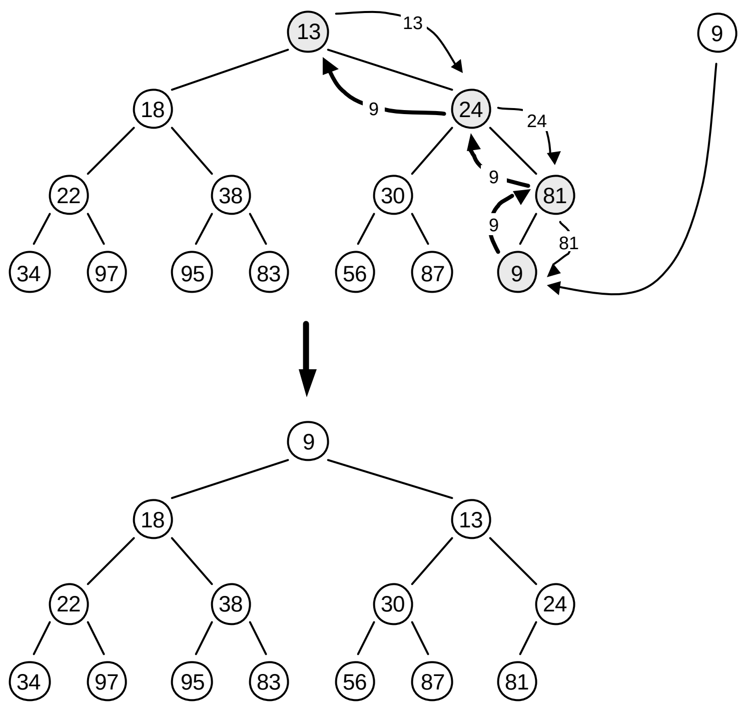
© ICT 2003, www.ict.org, All Rights Reserved
Use permitted only by licensees in accordance
with license terms (http://www.ict.org/javalicense.pdf)

5. The heap property still has not been restored in the left subtree. So again interchange the 87 with the smaller of its children.



6. We need to make at most *h* interchanges of a root of a subtree with one of its children to fully restore the heap property. Thus deletion from a heap is O(log n).

7. To add an item to a heap, we follow the reverse procedure. First we add the new node as the last le af, and then apply a “reheap up” procedure to restore the ordering property of the heap. “Reheap up” moves the new node up the tree, swapping places with its parent until the order is restored. For example, adding the value 9 to the original heap would result in the following sequence of steps:



8. Again, we require O(log n) exchanges.

© ICT 2003, www.ict.org, All Rights Reserved
Use permitted only by licensees in accordance
with license terms (http://www.ict.org/javalicense.pdf)

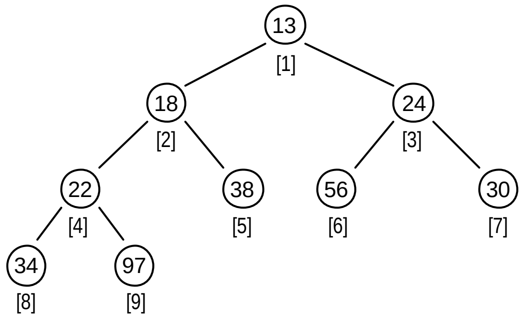
D. Storage of Complete Trees

1. The properties of a complete tree lead to a very efficient storage mechanism using `n` sequential locations in an array.

2. An important benefit of the array representation is that we can move around the tree, from parent to children or from child to parent, by simple arithmetic. In general, if we number the nodes from 1 at the root then

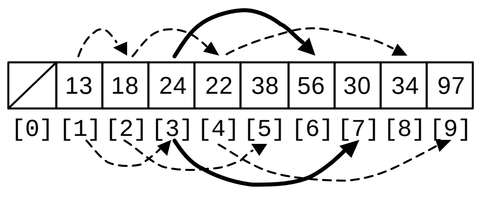
   a. The left and right children of node `i`, if they are present, are at `2i` and `2i+1`
   b. The parent of node `i` is at `i/2` (truncated to an integer).

3. If `items` is the array, the root corresponds to `Items[1]`; subsequent slots in the array store the nodes in each consecutive level from left to right.



Items[0]	<null>
Items[1]	13
Items[2]	18
Items[3]	24
Items[4]	22
Items[5]	38
Items[6]	56
Items[7]	30
Items[8]	34
Items[9]	97

4. In a Java implementation, it is convenient to leave `Items[0]` unused. With this numbering of nodes, the children of the node `Items[i]` can be found in `Items[2*i]` and `Items[2*i+1]`, and the parent of `Items[i]` is in `Items[i/2]`.



E. The `PriorityQueue` Interface

1. A priority queue contains items ranked according to some relation of order and provides methods to add an item and to remove and return the smallest item. The items in a priority queue do not have to all be different; if several items have the smallest rank, the removal method can remove and return any one of them. In a Java implementation, we assume that the items are `Comparable` objects.

© ICT 2003, www.ict.org, All Rights Reserved
Use permitted only by licensees in accordance
with license terms (http://www.ict.org/javalicense.pdf)

2. The Java library packages do not supply an interface specifically for priority queues. The `PriorityQueue` interface[*] shown below defines four methods: `isEmpty`, `add`, `removeMin`, and `peekMin`. The methods in this interface are analogous to the ones in the `Stack` interface and the `Queue` interface.

```
public interface PriorityQueue
{
  // Returns true if the number of elements in the
  // priority queue is 0; otherwise, returns false
  boolean isEmpty();

  // obj has been added to the priority queue;
  // number of elements in the priority queue is increased by 1
  void add(Object obj);

  // The smallest item in the priority queue is removed and
  // returned; the number of elements in the priority queue
  // is decreased by 1. Throws an unchecked exception if the
  // priority queue is empty
  Object removeMin();

  // The smallest item in the priority queue is returned; the
  // priority queue is unchanged. Throws an unchecked exception
  // if the priority queue is empty
  Object peekMin();
}
```

See ***ArrayPriorityQueue.java***.

3. The Java Library does not supply a class that implements a priority queue. A simplistic class that implements a priority queue can be put together very quickly based on an `ArrayList` or `LinkedList` (for the full `ArrayList` implementation of a priority queue see *ArrayPriorityQueue.java*). However, a more efficient implementation can be developed based on *heaps*.

SUMMARY/ REVIEW:

In this lesson, we developed a formal representation of a priority queue as a Java interface. We discussed the concept of a heap and the implementation of an efficient priority queue based on a heap. In the lab exercise, we will develop a heap based priority queue and use it to sort a file using the *Heapsort* algorithm.

This now concludes our coverage of different methods of data storage in the curriculum guide. As you continue in computer science, you will no doubt learn about other data structures and algorithms. Keep reading and learning!

ASSIGNMENT: Lab Exercise L.A.41.1, *Heapsort*

[*] Adapted from the College Board's *AP Computer Science AB: Implementation Classes and Interfaces*.

© ICT 2003, www.ict.org, All Rights Reserved
Use permitted only by licensees in accordance
with license terms (http://www.ict.org/javalicense.pdf)

LAB EXERCISE

Heapsort

Background:

A priority queue, implemented as a heap, can be used for sorting. To do that we must add all the items to a priority queue in any order, and then remove them one by one. The items will be returned in ascending order. This efficient algorithm is called *Heapsort*.

To realize the heapsort algorithm, it is necessary to develop a class that implements the `PriorityQueue` interface using a binary heap.

This lab will use a text file, "*file20.txt*", which is similar to the one used in the binary search lab in Lesson 28. The file has been saved in random order by `Id` number. Your program must build a binary heap based on the `Id` field. The priority queue should be implemented as a `HeapPriorityQueue` of type `Item`.

Assignment:

1. Here are some of the specifications for the methods to be added to the `HeapPriorityQueue` class:

    a. You are to write a method `isEmpty` that returns true if the number of element in the priority queue is 0; otherwise it returns false.

    b. An `add` method will add a new item to the heap, rearranging the heap as necessary to preserve the heap structure.

    c. The `removeMin` method will return and remove the highest priority item from the priority queue. If the queue is empty, a `NoSuchElementException` should be thrown.

    c. The `peekMin` method will return the highest priority item from the priority queue. If the queue is empty, a `NoSuchElementException` should be thrown.

    e. It is recommended that a `heapify` helper method be created as described in the lesson to reorganize the heap to preserve the heap structure after the removal of the root item.

    f. The heap structure should be contained in an `ArrayList`. To aid in coding, the root of the binary heap should start at index 1.

    g. Reading the data file is a similar process to that used in *Store.java* in Lesson 27.

    h. Printing the list involves the same code as used in the previous lessons.

© ICT 2003, www.ict.org, All Rights Reserved
Use permitted only by licensees in accordance
with license terms (http://www.ict.org/javalicense.pdf)

2. If your instructor chooses, you will be provided with a program shell consisting of a `main` menu method, testing methods, and stubbed methods for routines you must develop. Here are some of the specifications of this program shell.

    a. A `HeapSort` test method is provided. A method to read the data file is provided. However, the `sort` method is stubbed out as a print statement.

    b. The `Item` class is provided.

    c. The `remove` method returns a **`null`** value.

    d. A shell for the `HeapPriorityQueue` class is provided. The `add`, `removeMin`, `peekMin`, and `isEmpty`, methods are stubbed out.

    g. Methods to read the data file and print the list are provided.

<u>Instructions:</u>

1. Modify and write code as necessary to satisfy the above specifications.

2. Print out the entire source code.

3. Include a printed run output of the file in original and sorted order.

© ICT 2003, www.ict.org, All Rights Reserved
Use permitted only by licensees in accordance
with license terms (http://www.ict.org/javalicense.pdf)