CS211, Lecture 20 Priority queues and Heaps

Readings: Weiss,
sec. 6.9,
secs. 21.1--21.5.

When they've got two queues going, there's never any queue! P.J. Heaps.

(The only quote I could find that had both queues and heaps in it.)

Priority queue

In various contexts, one needs a list of items, each with a priority.

Operations:

1. Add an item (with some priority).
2. Find an item with maximum priority.
3. Remove a maximum-priority item.

That is a priority queue.

Example: files waiting to be printed, print in order of size (smaller the size, the higher the priority).

Example: Job scheduler. Many processes waiting to be executed. Those with higher priority numbers are most important and should be give time before others.

Priority-queue interface

```
/** An instance is a priority queue */
public interface PriorityQueue {
   void insert(Comparable x);   /** Insert x into the priority queue */
   void makeEmpty();            /** Make the queue empty */
   boolean isEmpty();           /** = "queue is empty" */
   int size();                  /** = the size of the queue */

   /** = largest item in this queue --throw exception if queue is empty*/
   Comparable findMax();

   /** = delete and return largest item --throw exception if queue is empty*/
   Comparable removeMax();
}
```

x.compareTo(y) is used to see which has higher priority, x or y. Objects x and y could have many fields.

Weiss also allows the possibility of changing the priority of an item in the queue. We don't discuss that feature.

Priority-queue implementations

Possible implementations. Assume queue has n items. We look at average-case times. O(1) means constant time.

1.Unordered array segment b[0..n-1].
Insert: O(1), findMax: O(n), removeMax: O(n).

2. Ordered array segment b[0..n-1].
Insert: O(n), findMax:O(1), removeMax: O(n)

3. Ordered array segments b[0..n-1], from largest to smallest.
Insert: O(n), findMax:O(1), removeMax: O(1)

4.binary search tree (if depth of tree is a minimum).
Insert: O(log n), findMax:O(log n), removeMax: O(log n)

But how do we keep the tree nicely balanced?

Priority-queue implementations

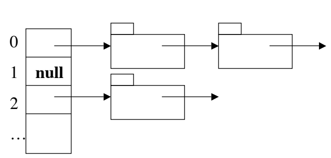
Possible implementations. Assume queue has n items. We look at average-case times. O(1) means constant time.

5. Special case. Suppose the possible priorities are 0..n-1.

Keep an array priority[0..n-1] in which priority[p] is a linked list of items with priority p.

Variable highestp: the highest p for which priority[p] is not empty (-1 if none)

Insert, worst case: O(1), findMax:O(1), removeMax: O(n)



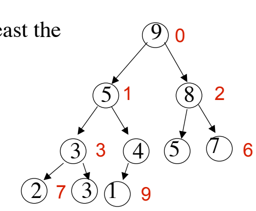
Definition of a heap

First, number nodes in breadth-first order.

A **complete binary tree** is one in which: if node number n is present, so are nodes 0..n-1.

A **heap** is a complete binary tree in which:

the value in any node is at least the values in its children.



Caution: Weiss numbers nodes 1..n instead of 0..n-1.

Because a heap is a complete binary tree, it goes nicely in an array b

Place node k in b[k]!

The parent of node k is in b[(k-1)/2].

The parent of node 9 is in b[(9-1)/2], which is b[4]

The parent of node 8 is in b[(8-1)/2], which is b[3]

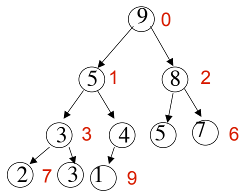
The children of node k are in

b[k*2 + 1]

b[k*2 + 2]

Children of 3 are in

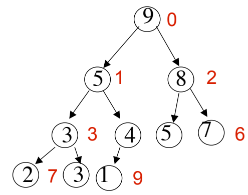
b[7] and b[8]



Where is the maximum value of a heap?

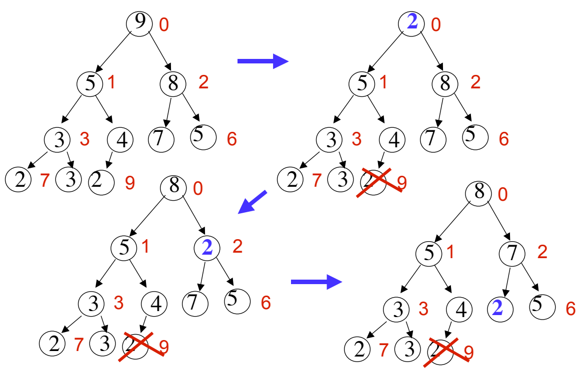
Max value of a heap is in node 0.
Node 0 is in b[0] in our array implementation.

Therefore, retrieving the max takes time O(1) (constant time).



Removing the max from an n-node tree takes time O(log n)

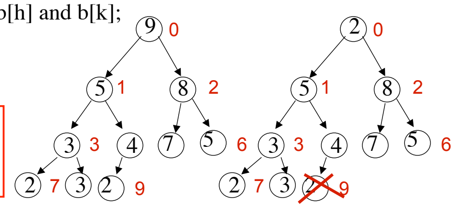
/** Precondition: n > 0. Remove max value from heap. */



Removing the max from an n-node tree takes time O(log n)

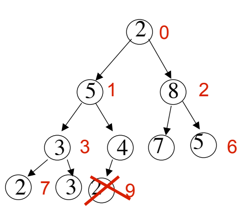
```
/** Remove max value from heap. Precondition: n > 0. */
n= n – 1; b[0]= b[n];
// Bubble b[0] down to its proper place in b[0..n-1]
int k= 0;
// inv: the only possible offender of the heap property is b[k]
while (k has a child that is larger) {
        Let h be the larger of k's childs;
        Swap b[h] and b[k];
        k= h;
}
```

Whenever some computation is messy, write a function for it



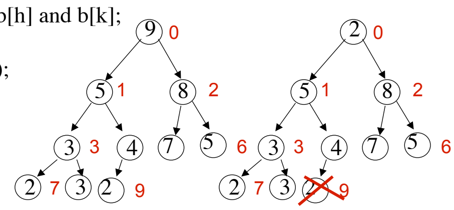

```
/** k is a node in b[0..n-1]. If k has a larger child, return the
    larger of its children; otherwise, return k */
public static int f(int[] b, int n, int k) {
   int h= 2*k+1;    // first child; will be the larger child
   if (h >= n) return k;  // k has no children
   // Set h to index of the larger of k's childs.
   if (h+1 < n || b[h+1].compareTo(b[h]) > 0)
        h= h+1;
   if (b[k].compareTo(b[h] <= 0)
        return h;
   return k;
}
```

The children of node k are in b[k*2 + 1], b[k*2 + 2]



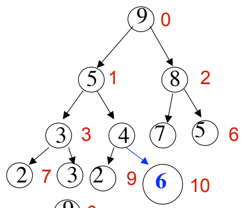
Removing the max from an n-node tree takes time O(log n)

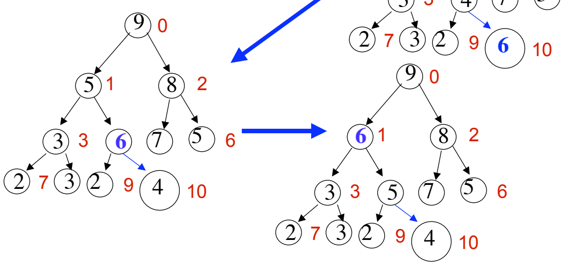
```
/** Remove max value from heap. Precondition: n > 0. */
n= n – 1; b[0]= b[n];
// Bubble b[0] down to its proper place in b[0..n-1]
int k= 0; h= f(k);
// inv: the only possible offender of the heap property is b[k]
      if k offends, h is its larger child; otherwise, h = k
while (h != k) {
        Swap b[h] and b[k];
        k= h;
        h= f(k);
}
```



Inserting a value into a heap takes time O(log n)

```
/** Insert ob into heap b[0..n-1]. */
b[n]= ob; n= n+1;
Bubble b[n-1] up
```





Heap sort

```
/** Sort b[0..n-1] */
  // Make b[0..n-1] into a heap
     invariant: b[0..k-1] is a heap (P1 below)
     for (int k= 0; k != n; k= k+1)
        { Bubble b[k] up }

  // Sort the heap b[0..n-1]
     invariant: Picture P2 below
     int h= n;
     while (h != 0) {
        h= h-1;
        Swap b[0] and b[h];
        Bubble b[0] down in b[0..h-1]
     }
```

Each part time O(n log n)

Total time is O(n log n)

P1: b [0 … this is a heap … k … n]

P2: b [0 … this is a heap … h … sorted, >= b[0..h-1] … n]