CS 61B: Lecture 25
Monday, October 25, 2010

Today's reading: Goodrich & Tamassia, Sections 8.1-8.3.

# PRIORITY QUEUES

A priority queue, like a dictionary, contains _entries_ that each consist of a key and an associated value. However, whereas a dictionary is used when we want to be able to look up arbitrary keys, a priority queue is used to prioritize entries. Define a total order on the keys (e.g. alphabetical order). You may identify or remove the entry whose key is the lowest (but no other entry). This limitation helps to make priority queues fast. However, an entry with any key may be inserted at any time.

For concreteness, let's use Integer objects as our keys. The main operations:
- insert() adds an entry to the priority queue;
- min() returns the entry with the minimum key; and
- removeMin() both removes and returns the entry with the minimum key.

```
                  5
---------         |      ---------               ---------
|4: womp|         v      |4: womp|               |       |
|7: gong|-insert(k, v)->|7: gong|-removeMin()->|7: gong|-min()
|       |            ^   |5: hoot|     |         |5: hoot|   |
---------            |   ---------     v         ---------   v
                   hoot            (4, womp)              (5, hoot)
```

Priority queues are most commonly used as "event queues" in simulations. Each value on the queue is an event that is expected to take place, and each key is the time the event takes place. A simulation operates by removing successive events from the queue and simulating them. This is why most priority queues return the minimum, rather than maximum, key: we want to simulate the events that occur first first.

```
public interface PriorityQueue {
  public int size();
  public boolean isEmpty();
  Entry insert(Object k, Object v);
  Entry min();
  Entry removeMin();
}
```

See page 340 of Goodrich & Tamassia for how they implement an "Entry".

## Binary Heaps: An Implementation of Priority Queues

A _complete_binary_tree_ is a binary tree in which every row is full, except possibly the bottom row, which is filled from left to right as in the illustration below. Just the keys are shown; the associated values are omitted.

```
         2       index:  0   1   2   3   4   5   6    7   8    9   10
        / \
       /   \           ------------------------------------------------
      5     3          |   | 2 | 5 | 3 | 9 | 6 | 11 | 4 | 17 | 10 | 8 |
     / \   / \         ------------------------------------------------
    9   6 11  4          ^
   / \ /                 |
 17 10 8                 \--- array index 0 intentionally left empty.
```

A _binary_heap_ is a complete binary tree whose entries satisfy the _heap-order_property_: no child has a key less than its parent's key. Observe that every subtree of a binary heap is also a binary heap, because every subtree is complete and satisfies the heap-order property.

Because they are complete, binary heaps are often stored as arrays of entries, ordered by a level-order traversal of the tree, with the root at index 1. This mapping of tree nodes to array indices is called _level_numbering_.

Observe that if a node's index is i, its children's indices are 2i and 2i+1, and its parent's index is floor(i/2). Hence, no node needs to store explicit references to its parent or children. (Array index 0 is left empty to make the indexing work out this nicely. If we instead put the root at index 0, node i's children are at 2i+1 and 2i+2, and its parent is at floor([i-1]/2).)

We can use either an array-based or a node-and-reference-based tree data structure, but the array representation tends to be faster (by a significant constant factor) because there is no need to read and write node references, cache performance is better, and finding the last node in the level order is easier.

Just like in hash tables, either each tree node has two references (one for the key, and one for the value), or each node references an "Entry" object (from page 340 of Goodrich and Tamassia).

Let's look at how we implement priority queue operations with a binary heap.

[1] Entry min();

The heap-order property ensures that the entry with the minimum key is always at the top of the heap. Hence, we simply return the entry at the root node. If the heap is empty, return null or throw an exception.

[2] Entry insert(Object k, Object v);

Let x be the new entry (k, v), whose key is k and whose value is v. We place the new entry x in the bottom level of the tree, at the first free spot from the left. (If the bottom level is full, start a new level with x at the far left.) In an array-based implementation, we place x in the first free location in the array (excepting index 0).

Of course, the new entry's key may violate the heap-order property. We correct this by having the entry bubble up the tree until the heap-order property is satisfied. More precisely, we compare x's key with its parent's key; if x's key is less, we exchange x with its parent, then repeat the procedure with x's new parent. For instance, if we insert an entry whose key is 2:

```
         2                  2                  2                  2
        / \                / \                / \                / \
       /   \              /   \              /   \              /   \
      5     3            5     3            5     3            2     3
     / \   / \    =>    / \   / \    =>    / \   / \    =>    / \   / \
    9   6 11  4        9   6 11  4        9   2 11  4        9   5 11  4
   / \ /              / \ / \            / \ / \            / \ / \
 17 10 8            17 10 8  2         17 10 8  6         17 10 8  6
```

As this example illustrates, a heap can contain multiple entries with the same key. (After all, in a typical simulation, we can't very well outlaw multiple events happening at the same time.)

When we finish, is the heap-order property satisfied? Yes, if the heap-order property was satisfied before the insertion. Let's look at a typical exchange of x with a parent p (right) during the insertion operation. Since the heap-order property was satisfied before the insertion, we know that p <= s (where s is x's sibling), p <= l, and p <= r (where l and r are x's children). We only swap if x < p, which implies that x < s; after the swap, x is the parent of s. After the swap, p is the parent of l and r. All other relationships in the subtree rooted at x are maintained, so after the swap, the tree rooted at x has the heap-order property.

```
     p          x
    / \        / \
   s   x  =>  s   p
  /\   /\    /\   /\
      l  r       l  r
```

For maximum speed, don't put x at the bottom of the tree and bubble it up. Instead, bubble a hole up the tree, then fill in x. This modification saves the time that would be spent setting a sequence of references to x that are going to change anyway.

Goodrich & Tamassia have insert() return an Entry object representing (k, v). I don't know why.

[3]  Entry removeMin();

If the heap is empty, return null or throw an exception. Otherwise, begin by removing the entry at the root node and saving it for the return value. This leaves a gaping hole at the root. We fill the hole with the last entry in the tree (which we call "x"), so that the tree is still complete.

It is unlikely that x has the minimum key. Fortunately, both subtrees rooted at the root's children are heaps, and thus the new mimimum key is one of these two children. We bubble x down the heap as follows: if x has a child whose key is smaller, swap x with the child having the minimum key. Next, compare x with its new children; if x still violates the heap-order property, again swap x with the child with the minimum key. Continue until x is less than or equal to its children, or reaches a leaf.

Consider running removeMin() on our original tree.

```
         2                  8                  3                  3
        / \                / \                / \                / \
       /   \              /   \              /   \              /   \
      5     3            5     3            5     8            5     4
     / \   / \    =>    / \   / \    =>    / \   / \    =>    / \   / \
    9   6 11  4        9   6 11  4        9   6 11  4        9   6 11  8
   / \ /              / \                / \                / \
  17 10 8            17 10              17 10              17 10
```

Above, the entry bubbled all the way to a leaf. This is not always the case, as the example at right shows.

```
      1                  4                  2
     / \                / \                / \
    /   \              /   \              /   \
   2     3    =>      2     3    =>      4     3
  / \   / \          / \   /            / \   /
 9   6 11  4        9   6 11           9   6 11
```

For maximum speed, don't put x at the root and bubble it down. Instead, bubble a hole down the tree, then fill in x.

Running Times
-------------
There are other, less efficient ways we could implement a priority queue than using a heap. For instance, we could use a list or array, sorted or unsorted. The following table shows running times for all, with n entries in the queue.

| | Binary Heap | Sorted List/Array | Unsorted List/Array |
|---|---|---|---|
| min() | Theta(1) | Theta(1) | Theta(n) |
| insert() | | | |
| worst-case | Theta(log n) * | Theta(n) | Theta(1) * |
| best-case | Theta(1) * | Theta(1) * | Theta(1) * |
| removeMin() | | | |
| worst-case | Theta(log n) | Theta(1) ** | Theta(n) |
| best-case | Theta(1) | Theta(1) ** | Theta(n) |

*   If you're using an array-based data structure, these running times assume that you don't run out of room. If you do, it will take Omega(n) time to allocate a larger array and copy the entries into it. However, if you double the array size each time, the _average_ running time will still be as indicated.

**  Removing the minimum from a sorted array in constant time is most easily done by keeping the array always sorted from largest to smallest.

In a binary heap, min's running time is clearly in Theta(1).

insert() puts an entry x at the bottom of the tree and bubbles it up. At each level of the tree, it takes O(1) time to compare x with its parent and swap if indicated. An n-node complete binary tree has ceiling(log2 n) levels. In the worst case, x will bubble all the way to the top, taking Theta(log n) time.

Similarly, removeMin may cause an entry to bubble all the way down the heap, taking Theta(log n) worst-case time.

Bottom-Up Heap Construction
---------------------------
Suppose we are given a bunch of randomly ordered entries, and want to make a heap out of them. We could insert them one by one in O(n log n) time, but there's a faster way. We define one more heap operation.

[4]  void bottomUpHeap();

First, we make a complete tree out of the entries, in any order. (If we're using an array representation, we just throw all the entries into an array.) Then we work backward from the last internal node (non-leaf node) to the root node, in reverse order in the array or the level-order traversal. When we visit a node this way, we bubble its entry down the heap as in removeMin().

Before we bubble an entry down, we know (inductively) that its two child subtrees are heaps. Hence, by bubbling the entry down, we create a larger heap rooted at the node where that entry started.

```
                                                             +-+
       9                  9                  9               |2|
      / \                / \                / \              /-\
     /   \              /   \-+          +-/   \            /   \
    4     7     =>     4    |2|    =>    |2|    2     =>   4     2
   / \   / \          / \   /-\          /-\   / \        / \   / \
  2   8 2   6        2   8 7   6        4   8 7   6      9   8 7   6
```

The running time of bottomUpHeap is tricky to derive. If each internal node bubbles all the way down, then the running time is proportional to the sum of the heights of all the nodes in the tree. Page 371 of Goodrich and Tamassia has a simple and elegant argument showing that this sum is less than n, where n is the number of entries being coalesced into a heap. Hence, the running time is in Theta(n), which beats inserting n entries into a heap individually.

Postscript:  Other Types of Heaps (not examinable)
---------------------------------

Binary heaps are not the only heaps in town.  Several important variants are called "mergeable heaps", because it is relatively fast to combine two mergeable heaps together into a single mergeable heap.  We will not describe these complicated heaps in CS 61B, but it’s worthwhile for you to know they exist in case you ever need one.

The best-known mergeable heaps are called "binomial heaps," "Fibonacci heaps," "skew heaps," and "pairing heaps."  Fibonacci heaps have another remarkable property:  if you have a reference to an arbitrary node in a Fibonacci heap, you can decrease its key in constant time.  (Pairing heaps are suspected of having the same property, but nobody knows for sure.)  This operation is used frequently by an important algorithm for finding the shortest path in a graph. The following running times are all worst-case.

| | Binary | Binomial | Skew | Pairing | Fibonacci |
|---|---|---|---|---|---|
| insert() | O(log n) | O(log n) | O(1) | O(log n) * | O(1) |
| removeMin() | O(log n) | O(log n) | O(log n) | O(log n) | O(log n) |
| merge() | O(n) | O(log n) | O(1) | O(log n) * | O(1) |
| decreaseKey() | O(log n) | O(log n) | O(log n) | O(log n) * | O(1) |

 *   Conjectured to be O(1), but nobody has proven or disproven it.

The time bounds given here for skew heaps, pairing heaps, and Fibonacci heaps are "amortized" bounds, not worst case bounds.  This means that, if you start from an empty heap, any sequence of operations will take no more than the given time bound on average, although individual operations may occasionally take longer.  We’ll discuss amortized analysis late this semester.