CSCI-1200 Data Structures — Spring 2015
Lecture 22 – Priority Queues I

Review from Lecture 21

- Finishing binary search trees & the `ds_set` class
- Operators as non-member functions, as member functions, and as friend functions.

Today's Lecture

- Finish last lecture – Operators!
- STL Queues and Stacks
- Whats a Priority Queue?
- A Priority Queue as a Heap
- `percolate_up` and `percolate_down`

22.1 Additional STL Container Classes: Stacks and Queues

- We've studied STL vectors, lists, maps, and sets. These data structures provide a wide range of flexibility in terms of operations. One way to obtain computational efficiency is to consider a simplified set of operations or functionality.
- For example, with a hash table we give up the notion of a sorted table and gain in find, insert, & erase efficiency.
- 2 additional examples are:
  - **Stacks** allow access, insertion and deletion from only one end called the *top*
    - There is no access to values in the middle of a stack.
    - Stacks may be implemented efficiently in terms of vectors and lists, although vectors are preferable.
    - All stack operations are $O(1)$
  - **Queues** allow insertion at one end, called the *back* and removal from the other end, called the *front*
    - There is no access to values in the middle of a queue.
    - Queues may be implemented efficiently in terms of a list. Using vectors for queues is also possible, but requires more work to get right.
    - All queue operations are $O(1)$

22.2 What's a Priority Queue?

- Priority queues are used in prioritizing operations. Examples include a personal "to do" list, jobs on a shop floor, packet routing in a network, scheduling in an operating system, or events in a simulation.
- Among the data structures we have studied, their interface is most similar to a queue, including the idea of a `front` or `top` and a `tail` or a `back`.
- Each item is stored in a priority queue using an associated "priority" and therefore, the `top` item is the one with the lowest value of the priority score. The `tail` or `back` is never accessed through the public interface to a priority queue.
- The main operations are `insert` or `push`, and `pop` (or `delete_min`).

22.3 Some Data Structure Options for Implementing a Priority Queue

- Vector or list, either sorted or unsorted
  - At least one of the operations, `push` or `pop`, will cost linear time, at least if we think of the container as a linear structure.
- Binary search trees
  - If we use the priority as a `key`, then we can use a combination of finding the minimum key and erase to implement `pop`. An ordinary binary-search-tree insert may be used to implement `push`.
  - This costs logarithmic time in the average case (and in the worst case as well if balancing is used).
- The latter is the better solution, but we would like to improve upon it — for example, it might be more natural if the minimum priority value were stored at the root.
  - We will achieve this with binary *heap*, giving up the complete ordering imposed in the binary *search tree*.

22.4 Definition: Binary Heaps

- A binary heap is a complete binary tree such that at each internal node, p, the value stored is less than the value stored at either of p's children.
  - A complete binary tree is one that is completely filled, except perhaps at the lowest level, and at the lowest level all leaf nodes are as far to the left as possible.
- Binary heaps will be drawn as binary trees, but implemented **using vectors**!
- Alternatively, the heap could be organized such that the value stored at each internal node is greater than the values at its children.

22.5 Exercise: Drawing Binary Heaps

Draw two different binary heaps with these values: 52 13 48 7 32 40 18 25 4

22.6 Implementing Pop (a.k.a. Delete Min)

- The top (root) of the tree is removed.
- It is replaced by the value stored in the last leaf node. This has echoes of the erase function in binary search trees. *NOTE: We have not yet discussed how to find the last leaf.*
- The last leaf node is removed.
- The (following) `percolate_down` function is then run to restore the heap property. This function is written here in terms of tree nodes with child pointers (and the priority stored as a `value`), but later it will be written in terms of vector subscripts.

```
percolate_down(TreeNode<T> * p) {
  while (p->left) {
    TreeNode<T>* child;
    //  Choose the child to compare against
    if (p->right && p->right->value < p->left->value)
      child = p->right;
    else
      child = p->left;
    if (child->value < p->value) {
      swap(child, p); // value and other non-pointer member vars
      p = child;
    }
    else
      break;
  }
}
```

22.7 Push / Insert

- To add a value to the heap, a new last leaf node in the tree is created and then the following `percolate_up` function is run. It assumes each node has a pointer to its parent.

```
percolate_up(TreeNode<T> * p) {
  while (p->parent)
    if (p->value < p->parent->value) {
      swap(p, parent);  // value and other non-pointer member vars
      p = p->parent;
    }
    else
      break;
}
```

22.8 Analysis

- Both `percolate_down` and `percolate_up` are $O(\log n)$ in the worst-case. Why?

- But, `percolate_up` (and as a result `push`) can be $O(1)$ in the average case. Why?

22.9 Exercise

Suppose the following operations are applied to an initially empty binary heap of integers. Show the resulting heap after each `delete_min` operation. (Remember, the tree must be **complete**!)

```
push 5, push 3, push 8, push 10, push 1, push 6,
pop,
push 14, push 2, push 4, push 7,
pop,
pop,
pop
```