CSCI-1200 Data Structures — Fall 2019
Lecture 20 – Priority Queues, part 1

Announcements: Test 3 Information

- Test 3 will be held Thursday, November 14th from 6-7:50pm. Your exam room & zone assignment will be posted on Submitty. *Note: We will re-shuffle the room & zone assignments from Exams 1 & 2.*
- Coverage: Lectures 1-20, Labs 1-11, HW 1-8.
- OPTIONAL: Prepare a 2 page, black & white, 8.5x11", portrait orientation `.pdf` of notes you would like to have during the exam. This may be digitally prepared or handwritten and scanned or photographed. The file may be no bigger than 2MB. You will upload this file to Submitty before Wednesday night @11:59pm. We will print this and attach it to your exam. *You MAY NOT bring a hardcopy of your notes to the exam.*
- Practice problems from previous exams are available on the course website. Solutions to the problems will be posted on Wednesday morning.
- All students must bring their Rensselaer photo ID card. Please leave your phones, computers, backpacks etc. in your dorm room (unless you are coming directly from another class).

Review from Lecture 19

- Limitations of our `ds_set` implementation, brief intro to red-black trees
- Operators as non-member functions, as member functions, and as friend functions.

Today's Lecture

- STL Queue and STL Stack
- What's a Priority Queue?
- Definition of a Binary Heap
- A Priority Queue as a Heap
- Implementing Pop (Delete Min) and Push (Insert)
- A Heap as a Vector

20.1 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 $O(1)$

20.2 Suggested Exercises: Tree Traversal using a Stack and Queue

Given a pointer to the root node in a binary tree:

- Use an STL `stack` to print the elements with a pre-order traversal ordering. *This is straightforward.*
- Use an STL `stack` to print the elements with an in-order traversal ordering. *This is more complicated.*
- Use an STL `queue` to print the elements with a breadth-first traversal ordering.

20.3 What's a Priority Queue?

- Priority queues are used in prioritizing operations. Examples include a personal "to do" list, what order to do homework assignments, 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 `push` (a.k.a. `insert`), and `pop` (a.k.a. `delete_min`).

20.4 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*.

20.5 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.

20.6 Exercise: Drawing Binary Heaps

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

Draw several other trees with these values that *not* binary heaps.

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

- The value at the top (root) of the tree is replaced by the value stored in the last leaf node.
  This has echoes of the erase function in binary search trees.

- The last leaf node is removed.
  QUESTION: But how do we find the last leaf? Ignore this for now...

- The value now at the root likely breaks the heap property. We use the `percolate_down` function 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 re-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;
  }
}
```

20.8 Implementing Push (a.k.a. Insert)

- To add a value to the heap, a new last leaf node in the tree is created to store that value.

- Then the `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;
}
```

20.9 Push (Insert) and Pop (Delete-Min) Usage 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
```

20.10 Heap Operations Time Complexity Analysis

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

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

20.11 Implementing a Heap with a Vector (instead of Nodes & Pointers)

- In the vector implementation, the tree is never explicitly constructed. Instead the heap is stored as a vector, and the child and parent "pointers" can be implicitly calculated.
- To do this, number the nodes in the tree starting with 0 first by level (top to bottom) and then scanning across each row (left to right). These are the vector indices. Place the values in a vector in this order.
- As a result, for each subscript, i,
  - The parent, if it exists, is at location $\lfloor (i-1)/2 \rfloor$.
  - The left child, if it exists, is at location $2i+1$.
  - The right child, if it exists, is at location $2i+2$.
- For a binary heap containing n values, the last leaf is at location $n-1$ in the vector and the last internal (non-leaf) node is at location $\lfloor (n-1)/2 \rfloor$.
- The standard library (STL) `priority_queue` is implemented as a binary heap.

20.12 Heap as a Vector Exercises

- Draw a binary heap with values: 52 13 48 7 32 40 18 25 4, first as a tree of nodes & pointers, then in vector representation.

- Starting with an initially empty heap, show the vector contents for the binary heap after each `delete_min` operation.

```
push 8, push 12, push 7, push 5, push 17, push 1,
pop,
push 6, push 22, push 14, push 9,
pop,
pop,
```