CSCI-1200 Data Structures — Spring 2016
Lecture 21 – Priority Queues I

Today's Lecture

- STL Queues and Stacks
- Whats a Priority Queue?
- A Priority Queue as a Heap
- `percolate_up` and `percolate_down`

21.1 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 $O(1)$

21.2 Tree Traversal using a Stack

Preorder traversal of a binary tree is relatively simple. Begin by pushing the root onto the stack. Until the stack is empty, pop a node from the stack, visit the node, and the push its children onto the stack.

```
#include <stack>

void Preorder(Node* root) {
    stack<Node*> s;

    if (root != NULL) {
        s.push(root);
    }

    while (!s.empty()) {
        Node* p = s.top();
        s.pop();

        Visit(p);   // process the node

        if (p->right != NULL) s.push(p->right);
        if (p->left != NULL) s.push(p->left);
    }
}
```

Inorder traversal is more complicated. The leftmost node must be processed first. In processing a node, we move down the left branch until we reach the left most node. When popping a node, we visit the node because we know that its left children have been visited. After visiting the node, we push its right child and repeat the process.

```
void Inorder(Node* root) {
    stack<Node*> s;
    Node* p = root;

    while (p != NULL) {
        s.push(p);
        p = p->left;
    }

    while (!s.empty()) {
        p = s.top();
        s.pop();

        Visit(p);   // process the node

        p = p->right;
        while (p != NULL) {
            s.push(p);
            p = p->left;
        }
    }
}
```

Postorder traversal is similar to inorder traversal but more difficult. Instead of popping the node, it leaves the node on the stack and visits the right node. After processing the right subtree, it pops the node and visits it. It visits each node twice. To avoid processing the node twice, a second stack of bools is used to keep track of whether the node has been processed.

21.3 Breadth First Traversal using a Queue

A queue is a convenient structure to assist in breadth first tree traversal.

```
void BreadthFirst(Node* root) {
    queue<Node*> q;
    q.push(root);

    while(! q.empty) {
        Node* n = q.front();
        Visit(n);   // process the node

        if (n->left != NULL)
           q.push(n->left);
        if (n->right != NULL)
           q.push(n->right);

       q.pop();
    }
}
```

21.4 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 `insert` or `push`, and `pop` (or `delete_min`).

21.5 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*.

21.6 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.

21.7 Exercise: Drawing Binary Heaps

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

Binary Heap Visualization https://www.cs.usfca.edu/~galles/visualization/Heap.html

21.8 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;
  }
}
```

21.9 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;
}
```

21.10 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?

21.11 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
```

21.12 Vector Implementation

- 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.

21.13 Exercise

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.

21.14 Exercise

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,
```

21.15 Building A Heap

- In order to build a heap from a vector of values, for each index from $\lfloor (n-1)/2 \rfloor$ down to 0, run `percolate_down`. Show that this fully organizes the data as a heap and requires at most $O(n)$ operations.
- If instead, we ran `percolate_up` from each index starting at index 0 through index n-1, we would get properly organized heap data, but incur a $O(n \log n)$ cost. Why?

21.16 Heap Sort

- Heap Sort is a simple algorithm to sort a vector of values: build a heap and then run n consecutive `pop` operations, storing each "popped" value in a new vector.
- It is straightforward to show that this requires $O(n \log n)$ time.
- **Exercise:** Implement an *in-place* heap sort. An in-place algorithm uses only the memory holding the input data – a separate large temporary vector is not needed.

21.17 Summary Notes about Vector-Based Priority Queues

- Priority queues are conceptually similar to queues, but the order in which values / entries are removed ("popped") depends on a priority.
- Heaps, which are conceptually a binary tree but are implemented in a vector, are the data structure of choice for a priority queue.
- In some applications, the priority of an entry may change while the entry is in the priority queue. This requires that there be "hooks" (usually in the form of indices) into the internal structure of the priority queue. This is an implementation detail we have not discussed.