Computer Science II — CSci 1200
Lecture 21
Priority Queues and Heaps

Review from Lecture 20

- Collision resolution
- Using a hash table to implement a `set`.
  - Function objects
  - Overall design
  - Iterators
  - Fundamental operations: find, insert and erase.
- We will complete our discussion today:
  - Erase
  - Resize
  - Iterators
  - Limitations

Today's Class

- Idea of a priority queue
- A priority queue as a heap — a complete binary tree
- Percolate up and percolate down operations
- A heap as a vector
- Making a heap
- Heap sort

Priority Queue — Fundamental Operations

- Priority queues are used in prioritizing operations. Examples include 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 interface to a priority queue.
- The main operations are `insert` or `push`, and `pop` (or `delete_min`).

Data Structure Options

- Vector or list, either sorted or unsorted
  - Here 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 minimimum priority value were stored at the root.
  - We will achieve this using a "heap", giving up the complete ordering imposed in the binary search tree.

Binary Heaps

- **Definition:** 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.

Pop / 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.
  - 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;
    }
}
```

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;
}
```

Analysis

- Both `percolate_down` and `percolate_up` are $O(\log n)$ in the worst-case.
- But, `percolate_up` (and as a result `push`) is $O(1)$ in the average case.
- This analysis will be discussed briefl in class.

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
```

Vector Implementation

- In the vector implementation, the tree is never explicitly constructed. Instead the heap is stored as a vector, with child and parent "pointers" implicit.
- To do this, number the nodes in the tree top to bottom and left to right, starting with 0. 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.
- We will explore this implementation further in Lab 12.

Exercise

1. 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,
```

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`.
- It can be shown that this requires at most $O(n)$ operations.
- If instead, we ran `percolate_up` from each index starting at n-1 down to 0, we would incur a $O(n \log n)$ cost.

Heap Sort

- Here 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.
- This can also be done "in place" so that a separate vector is not needed.

Summary

- 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.