

CSCI-1200 Data Structures — Spring 2013

Lecture 21 – Priority Queues and Leftist Heaps

Review from Lecture 20

- Hash Table Implementation & HW9 Overview
- Operators as non-member functions, as member functions, and as friend functions.

Today's Class

- STL Queues and Stacks
- What's a Priority Queue?
- A Priority Queue as a Heap, `percolate_up` and `percolate_down`
- A Heap as a Vector, Building a Heap, Heap Sort
- Merging heaps are the motivation for *leftist heaps*

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. 2 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 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`).

21.3 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 using a binary *heap*, giving up the complete ordering imposed in the binary *search tree*.

21.4 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.

21.5 Exercise: Drawing Binary Heaps

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

21.6 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.
 - 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;
    }
}

```

21.7 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;
}

```

21.8 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? (The full answer is beyond the scope of this course.)

21.9 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

```

21.10 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.

21.11 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.

21.12 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,
```

21.13 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. Why?

21.14 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.

21.15 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.

21.16 Leftist Heaps — Overview

- Our goal is to be able to merge two heaps in $O(\log n)$ time, where n is the number of values stored in the larger of the two heaps.
 - Merging two binary heaps (where every row but possibly the last is full) requires $O(n)$ time
- Leftist heaps are binary trees where we deliberately attempt to eliminate any balance.
 - Why? Well, consider the most *unbalanced* tree structure possible. If the data also maintains the heap property, we essentially have a sorted linked list.
- Leftists heaps are implemented explicitly as trees (rather than vectors).

21.17 Leftist Heaps — Mathematical Background

- **Definition:** The *null path length* (NPL) of a tree node is the length of the shortest path to a node with 0 children or 1 child. The NPL of a leaf is 0. The NPL of a NULL pointer is -1.
- **Definition:** A *leftist tree* is a binary tree where at each node the null path length of the left child is greater than or equal to the null path length of the right child.
- **Definition:** The *right path* of a node (e.g. the root) is obtained by following right children until a NULL child is reached.
 - In a leftist tree, the right path of a node is at least as short as any other path to a NULL child.
- **Theorem:** A leftist tree with $r > 0$ nodes on its right path has at least $2^r - 1$ nodes.
 - This can be proven by induction on r .
- **Corollary:** A leftist tree with n nodes has a right path length of at most $\lfloor \log(n + 1) \rfloor = O(\log n)$ nodes.
- **Definition:** A *leftist heap* is a leftist tree where the value stored at any node is less than or equal to the value stored at either of its children.

21.18 Leftist Heap Operations

- The `insert` and `delete_min` operations will depend on the `merge` operation.
- Here is the fundamental idea behind the merge operation. Given two leftist heaps, with `h1` and `h2` pointers to their root nodes, and suppose `h1->value <= h2->value`. Recursively merge `h1->right` with `h2`, making the resulting heap `h1->right`.
- When the leftist property is violated at a tree node involved in the merge, the left and right children of this node are swapped. This is enough to guarantee the leftist property of the resulting tree.
- `Merge` requires $O(\log n + \log m)$ time, where m and n are the numbers of nodes stored in the two heaps, because it works on the right path at all times.

21.19 Merge Code

```
template <class T>
class LeftNode {
public:
    LeftNode() : npl(0), left(0), right(0) {}
    LeftNode(const T& init) : value(init), npl(0), left(0), right(0) {}
    T value;
    int npl;          // the null-path length
    LeftNode* left;
    LeftNode* right;
};
```

Here are the two functions used to implement leftist heap merge operations. Function `merge` is the driver. Function `merge1` does most of the work. These functions call each other recursively.

```
template <class Etype>
LeftNode<Etype>* merge(LeftNode<Etype> *H1, LeftNode<Etype> *H2) {
    if (!h1)
        return h2;
    else if (!h2)
        return h1;
    else if (h2->value > h1->value)
        return merge1(h1, h2);
    else
        return merge1(h2, h1);
}
```

```
template <class Etype>
LeftNode<Etype>* merge1(LeftNode<Etype> *h1, LeftNode<Etype> *h2) {
    if (h1->left == NULL)
        h1->left = h2;
    else {
        h1->right = merge(h1->right, h2);
        if(h1->left->npl < h1->right->npl)
            swap(h1->left, h1->right);
        h1->npl = h1->right->npl + 1;
    }
    return h1;
}
```

21.20 Exercises

1. Explain how `merge` can be used to implement `insert` and `delete_min`, and then write code to do so.

2. Show the state of a leftist heap at the end of:

```
insert 1, 2, 3, 4, 5, 6
delete_min
insert 7, 8
delete_min
delete_min
```