CSCI-1200 Data Structures — Spring 2019
Lecture 22 – Priority Queues

Review from Lectures 20 & 21

- Hash Tables, Hash Functions, and Collision Resolution
- Performance of: Hash Tables vs. Binary Search Trees
- STL's `unordered_set` (and `unordered_map`)
- Using a hash table to implement a set/map
  - Hash functions as functors/function objects
  - Iterators, find, insert, and erase
- Using STL's `for_each`
- STL Queue and STL Stack
- What's a Priority Queue?

Today's Lecture

- A Priority Queue as a Heap
- A Heap as a Vector
- Building a Heap
- Heap Sort

22.1 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 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;
  }
}
```

22.2 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;
}
```

22.3 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
```

22.4 Heap Operations 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?

22.5 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 first node with less than two children is at location $\lfloor (n-1)/2 \rfloor$.
- The standard library (STL) `priority_queue` is implemented as a binary heap.

22.6 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,
```

22.7 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?

22.8 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.

22.9 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.

22.10 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).

22.11 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. The right child of each node has the lower null path length.
- **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.

22.12 Leftist Heap Operations

- The push/insert and pop/delete_min operations will depend on the merge operation.
- Here is the fundamental idea behind the merge operation. Given two leftist heaps, with pointers to their root nodes h1 and h2 respectively, 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.

22.13 Leftist Heap Implementation

- Our Node class:

```
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 merge_helper does most of the work. These functions call each other recursively.

```
template <class T>
LeftNode<T>* merge(LeftNode<T> *h1,LeftNode<T> *h2) {
  if (!h1)
    return h2;
  else if (!h2)
    return h1;
  else if (h2->value > h1->value)
    return merge_helper(h1, h2);
  else
    return merge_helper(h2, h1);
}

template <class T>
LeftNode<T>* merge_helper(LeftNode<T> *h1, LeftNode<T> *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;
}
```

22.14 Leftist Heap 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
```