CSCI-1200 Data Structures — Fall 2017
Lecture 22 – Priority Queues II & Hash Tables I

Review from Lecture 21

- STL Queue and STL Stack
- Definition of a Binary Heap
- What's a Priority Queue?
- A Priority Queue as a Heap

Today's Lecture (Some notes in Lecture 21 Handout)

- A Heap as a Vector
- Building a Heap
- Heap Sort
- Merging heaps are the motivation for *leftist heaps*
- "the single most important data structure known to mankind"
- Hash Tables, Hash Functions, and Collision Resolution
- Performance of: Hash Tables vs. Binary Search Trees

22.1 Review: Vector vs. List vs. BST vs Heap: Time Complexity Analysis

	find smallest value	remove smallest value	insert value
unsorted vector/array			
sorted vector/array			
unsorted linked list			
sorted linked list			
binary search tree			
binary heap			

22.2 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.

22.3 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,
```

22.4 Building A Heap ... starting with all of your data in an unorganized vector

- In order to build a heap from a unorganized 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?

22.5 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. Side goal: Keep the number of element copy/swap operations to a minimum!

22.6 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.

22.7 Definition: What's a Hash Table?

- A table implementation with *constant time access*.
  - Like a set, we can store elements in a collection. Or like a map, we can store key-value pair associations in the hash table. But it's even faster to do find, insert, and erase with a hash table! However, hash tables *do not* store the data in sorted order.
- A hash table is implemented with an array at the top level.
- Each element or key is mapped to a slot in the array by a *hash function*.

22.8 Definition: What's a Hash Function?

- A simple function of one argument (the key) which returns an integer index (a bucket or slot in the array).
- Ideally the function will "uniformly" distribute the keys throughout the range of legal index values (0 → k-1).
- **What's a collision?**
  When the hash function maps multiple (different) keys to the same index.
- **How do we deal with collisions?**
  One way to resolve this is by storing a linked list of values at each slot in the array.

22.9 Example: Caller ID

- We are given a phonebook with 50,000 name/number pairings. Each number is a 10 digit number. We need to create a data structure to lookup the name matching a particular phone number. Ideally, name lookup should be O(1) time expected, and the caller ID system should use O(n) memory (n = 50,000).
- Note: In the toy implementations that follow we use small datasets, but we should evaluate the system scaled up to handle the large dataset.
- The basic interface:

```
// add several names to the phonebook
add(phonebook, 1111, "fred");
add(phonebook, 2222, "sally");
add(phonebook, 3333, "george");
// test the phonebook
std::cout << identify(phonebook, 2222) << " is calling!" << std::endl;
std::cout << identify(phonebook, 4444) << " is calling!" << std::endl;
```

- We'll review how we solved this problem in Lab 9 with an STL `vector` then an STL `map`. Finally, we'll implement the system with a hash table.

22.10 Caller ID with an STL Vector

```
// create an empty phonebook
std::vector<std::string> phonebook(10000, "UNKNOWN CALLER");

void add(std::vector<std::string> &phonebook, int number, std::string name) {
  phonebook[number] = name; }

std::string identify(const std::vector<std::string> &phonebook, int number) {
  return phonebook[number]; }
```

Exercise: What's the memory usage for the vector-based Caller ID system?
What's the expected running time for find, insert, and erase?

22.11 Caller ID with an STL Map

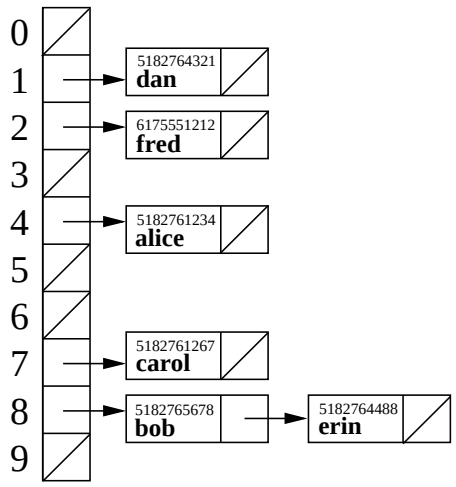
```
// create an empty phonebook
std::map<int,std::string> phonebook;

void add(std::map<int,std::string> &phonebook, int number, std::string name) {
  phonebook[number] = name; }
```

```
std::string identify(const std::map<int,std::string> &phonebook, int number) {
  map<int,std::string>::const_iterator tmp = phonebook.find(number);
  if (tmp == phonebook.end()) return "UNKNOWN CALLER"; else return tmp->second;
}
```

Exercise: What's the memory usage for the map-based Caller ID system? What's the expected running time for find, insert, and erase?

22.12 Now let's implement Caller ID with a Hash Table

```
#define PHONEBOOK_SIZE 10

class Node {
public:
  int number;
  string name;
  Node* next;
};

// create the phonebook, initially all numbers are unassigned
Node* phonebook[PHONEBOOK_SIZE];
for (int i = 0; i < PHONEBOOK_SIZE; i++) {
  phonebook[i] = NULL;
}

// corresponds a phone number to a slot in the array
int hash_function(int number) {



}

// add a number, name pair to the phonebook
void add(Node* phonebook[PHONEBOOK_SIZE], int number, string name) {



}

// given a phone number, determine who is calling
void identify(Node* phonebook[PHONEBOOK_SIZE], int number) {




}
```



22.13 Exercise: Choosing a Hash Function

- What's a good hash function for this application?
- What's a bad hash function for this application?

22.14 Exercise: Hash Table Performance

- What's the memory usage for the hash-table-based Caller ID system?
- What's the expected running time for find, insert, and erase?