Review from Lecture 22 & 23

- STL Queues & Stacks
- Definition of a Priority Queue / Binary Heap
- `percolate_up` and `percolate_down`
- A Heap as a Vector, Building a Heap, Heap Sort
- Something weird & cool in C++... Function Objects, a.k.a. *Functors*

Today’s Lecture

- “the single most important data structure known to mankind”
- Hash Tables, Hash Functions, and Collision Resolution
- Performance of: Hash Tables vs. Binary Search Trees
- Collision resolution: separate chaining vs open addressing
- 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

24.1 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*.

24.2 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.

24.3 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.

24.4 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?

24.5 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?

24.6 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;
}

Node* 5182764321
    dan

Node* 617551212
    fred

Node* 5182761234
    alice

Node* 5182765678
    bob

Node* 5182764488
    erin
```c
// 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) {

}
```

### 24.7 Exercise: Choosing a Hash Function

- What’s a good hash function for this application?

- What’s a bad hash function for this application?

### 24.8 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?

### 24.9 What makes a Good Hash Function?

- Goals: fast O(1) computation and a random, uniform distribution of keys throughout the table, despite the actual distribution of keys that are to be stored.

- For example, using: $f(k) = abs(k) \% N$ as our hash function satisfies the first requirement, but may not satisfy the second.
Another example of a dangerous hash function on string keys is to add or multiply the ascii values of each char:

```cpp
unsigned int hash(string const& k, unsigned int N) {
  unsigned int value = 0;
  for (unsigned int i=0; i<k.size(); ++i)
    value += k[i]; // conversion to int is automatic
  return k % N;
}
```

The problem is that different permutations of the same string result in the same hash table location.

This can be improved through multiplications that involve the position and value of the key:

```cpp
unsigned int hash(string const& k, unsigned int N) {
  unsigned int value = 0;
  for (unsigned int i=0; i<k.size(); ++i)
    value = value*8 + k[i]; // conversion to int is automatic
  return k % N;
}
```

The 2nd method is better, but can be improved further. The theory of good hash functions is quite involved and beyond the scope of this course.

### 24.10 How do we Resolve Collisions? METHOD 1: Separate Chaining

- Each table location stores a linked list of keys (and values) hashed to that location (as shown above in the phonebook hashtable). Thus, the hashing function really just selects which list to search or modify.

- This works well when the number of items stored in each list is small, e.g., an average of 1. Other data structures, such as binary search trees, may be used in place of the list, but these have even greater overhead considering the (hopefully, very small) number of items stored per bin.

### 24.11 How do we Resolve Collisions? METHOD 2: Open Addressing

- In open addressing, when the chosen table location already stores a key (or key-value pair), a different table location is sought in order to store the new value (or pair).

- Here are three different open addressing variations to handle a collision during an `insert` operation:
  - **Linear probing:** If \( i \) is the chosen hash location then the following sequence of table locations is tested (“probed”) until an empty location is found:
    \[
    (i+1)\%N, (i+2)\%N, (i+3)\%N, \ldots
    \]
  - **Quadratic probing:** If \( i \) is the hash location then the following sequence of table locations is tested:
    \[
    (i+1)\%N, (i+2^2)\%N, (i+3^2)\%N, \ldots
    \]
    More generally, the \( j^\text{th} \) “probe” of the table is \( (i + c_1 j + c_2 j^2) \mod N \), where \( c_1 \) and \( c_2 \) are constants.
  - **Secondary hashing:** when a collision occurs a second hash function is applied to compute a new table location. This is repeated until an empty location is found.

- For each of these approaches, the `find` operation follows the same sequence of locations as the `insert` operation. The key value is determined to be absent from the table only when an empty location is found.

- When using open addressing to resolve collisions, the `erase` function must mark a location as “formerly occupied”. If a location is instead marked empty, `find` may fail to return elements in the table. Formerly-occupied locations may (and should) be reused, but only after the `find` operation has been run to completion.

- Problems with open addressing:
  - Slows dramatically when the table is nearly full (e.g. about 80% or higher). This is particularly problematic for linear probing.
  - Fails completely when the table is full.
  - Cost of computing new hash values.
24.12 Hash Table in STL?

- The Standard Template Library standard and implementation of hash table have been slowly evolving over many years. Unfortunately, the names “hashset” and “hashmap” were spoiled by developers anticipating the STL standard, so to avoid breaking or having name clashes with code using these early implementations...
- STL's agreed-upon standard for hash tables: unordered_set and unordered_map
- Depending on your OS/compiler, you may need to add the -std=c++11 flag to the compile line (or other configuration tweaks) to access these more recent pieces of STL. (And this will certainly continue to evolve in future years!) Also, for many types STL has a good default hash function, so you may not always need to specify both template parameters!

24.13 Our Copycat Version: A Set As a Hash Table

- The class is templated over both the key type and the hash function type.
  ```cpp
  template < class KeyType, class HashFunc >
  class ds_hashset { ... };
  ```
- We use separate chaining for collision resolution. Hence the main data structure inside the class is:
  ```cpp
  std::vector< std::list<KeyType> > m_table;
  ```
- We will use automatic resizing when our table is too full. Resize is expensive of course, so similar to the automatic reallocation that occurs inside the vector push_back function, we at least double the size of underlying structure to ensure it is rarely needed.

24.14 Our Hash Function Object

- Remember from last lecture that “function objects” or “functors” are just a class wrapper around a function, and that function is implemented as the overloaded function call operator for the class.
- Often the programmer/designer for the program using a hash function has the best understanding of the distribution of data to be stored in the hash function. Thus, they are in the best position to define a custom hash function (if needed) for the data & application.
- Here’s an example of a (generically) good hash function for STL strings, wrapped up inside of a class:
  ```cpp
  class hash_string_obj {
  public:
    unsigned int operator() (std::string const& key) const {
      // This implementation comes from
      // http://www.partow.net/programming/hashfunctions/
      unsigned int hash = 1315423911;
      for(unsigned int i = 0; i < key.length(); i++)
        hash ^= ((hash << 5) + key[i] + (hash >> 2));
      return hash;
    }
  }
  ```
- Once our new type containing the hash function is defined, we can create instances of our hash set object containing std::string by specifying the type hash_string_obj as the second template parameter to the declaration of a ds_hashset. E.g.,
  ```cpp
  ds_hashset<std::string, hash_string_obj> my_hashset;
  ```
- Alternatively, we could use function pointers as a non-type template argument.
  (We don’t show that syntax here!).

24.15 Hash Set Iterators

- Iterators move through the hash table in the order of the storage locations rather than the ordering imposed by (say) an operator<. Thus, the visiting/printing order depends on the hash function and the table size.
  - Hence the increment operators must move to the next entry in the current linked list or, if the end of the current list is reached, to the first entry in the next non-empty list.
- The declaration is nested inside the ds_hashset declaration in order to avoid explicitly templating the iterator over the hash function type.
• The iterator must store:
  – A pointer to the hash table it is associated with. This reflects a subtle point about types: even though
    the iterator class is declared inside the ds_hashset, this does not mean an iterator automatically knows
    about any particular ds_hashset.
  – The index of the current list in the hash table.
  – An iterator referencing the current location in the current list.

• Because of the way the classes are nested, the iterator class object must declare the ds_hashset class as a
  friend, but the reverse is unnecessary.

24.16 Implementing begin() and end()

• begin(): Skips over empty lists to find the first key in the table. It must tie the iterator being created to
  the particular ds_hashset object it is applied to. This is done by passing the this pointer to the iterator
  constructor.

• end(): Also associates the iterator with the specific table, assigns an index of -1 (indicating it is not a normal
  valid index), and thus does not assign the particular list iterator.

• Exercise: Implement the begin() function.

24.17 Iterator Increment, Decrement, & Comparison Operators

• The increment operators must find the next key, either in the current list, or in the next non-empty list.

• The decrement operator must check if the iterator in the list is at the beginning and if so it must proceed to
  find the previous non-empty list and then find the last entry in that list. This might sound expensive, but
  remember that the lists should be very short.

• The comparison operators must accommodate the fact that when (at least) one of the iterators is the end, the
  internal list iterator will not have a useful value.

24.18 Insert & Find

• Computes the hash function value and then the index location.

• If the key is already in the list that is at the index location, then no changes are made to the set, but an iterator
  is created referencing the location of the key, a pair is returned with this iterator and false.

• If the key is not in the list at the index location, then the key should be inserted in the list (at the front is
  fine), and an iterator is created referencing the location of the newly-inserted key a pair is returned with this
  iterator and true.

• Exercise: Implement the insert() function, ignoring for now the resize operation.

• Find is similar to insert, computing the hash function and index, followed by a std::find operation.

24.19 Erase

• Two versions are implemented, one based on a key value and one based on an iterator. These are based on
  finding the appropriate iterator location in the appropriate list, and applying the list erase function.

24.20 Resize

• Must copy the contents of the current vector into a scratch vector, resize the current vector, and then re-insert
  each key into the resized vector. Exercise: Write resize()

24.21 Hash Table Iterator Invalidation

• Any insert operation invalidates all ds_hashset iterators because the insert operation could cause a resize of
  the table. The erase function only invalidates an iterator that references the current object.
#ifndef ds_hashset_h_
define ds_hashset_h_

#include <iostream>
#include <list>
#include <string>
#include <vector>

// The ds_hashset is templated over both the type of key and the type // of the hash function, a function object.
template <class KeyType, class HashFunc>
class ds_hashset {
private:

    template<typename std::list<KeyType>::iterator hash_list_itr;

public:

    // =================================================================================
    // THE ITERATOR CLASS
    // Defined as a nested class and thus is not separately templated.
    class iterator {
        friend class ds_hashset;

        // allows access to private variables
        private:

            // ITERATOR REPRESENTATION
            int m_index; // current index in the hash table
            hash_list_itr m_list_itr; // current iterator at the current index

        private:

            // private constructors for use by the ds_hashset only
            iterator(ds_hashset* hs, int index, hash_list_itr loc) :
                m_hs(hs), m_index(index), m_list_itr(loc) {}  

        public:

            // Ordinary constructors & assignment operator
            iterator() : m_hs(0), m_index(-1) {}
            iterator(iterator const& itr) :
                m_hs(itr.m_hs), m_index(itr.m_index), m_list_itr(itr.m_list_itr) {}
            iterator& operator=(iterator const& old) {
                m_hs = old.m_hs;
                m_index = old.m_index;
                m_list_itr = old.m_list_itr;
                return *this;
            }

            // The dereference operator need only worry about the current
            // list iterator, and does not need to check the current index.
            const KeyTypes operator*() const { return *m_list_itr; }

            // The comparison operators must account for the list iterators
            // being unassigned at the end.
            friend bool operator== (const iterator& lft, const iterator& rgt) {
                return lft.m_hs == rgt.m_hs && lft.m_index == rgt.m_index &&
                (lft.m_index == -1 || lft.m_list_itr == rgt.m_list_itr);
            }

        private:

            // Find the next entry in the table
            void next() {
                // If we are at the end of this list
                if (m_list_itr == m_hs->m_table[m_index].end()) {
                    // Find the next non-empty list in the table
                    for (++m_index;
                        m_index < int(m_hs->m_table.size()) && m_hs->m_table[m_index].empty();
                        ++m_index) {}
                    // If one is found, assign the m_list_itr to the start
                    if (m_index != int(m_hs->m_table.size()))
                        m_list_itr = m_hs->m_table[m_index].begin();
                    // Otherwise, we are at the end
                    else
                        m_index = -1;
                }
            }

            // Find the previous entry in the table
            void prev() {
                // If we aren’t at the start of the current list, just decrement
                // the list iterator
                if (m_list_itr != m_hs->m_table[m_index].begin())
                    m_list_itr --;
                // Otherwise, back down the table until the previous
                // non-empty list in the table is found
                for (--m_index;
                    m_index >= 0 && m_hs->m_table[m_index].empty(); --m_index) {
                    // Go to the last entry in the list.
                    m_list_itr = m_hs->m_table[m_index].begin();
                    // hash_list_itr p = m_list_itr;
                    for (; p != m_hs->m_table[m_index].end(); ++p, ++m_list_itr) {}
                    m_list_itr --;
                }
            }
        }
    };

    // end of ITERATOR CLASS
    // =================================================================================

};
#endif // ds_hashset_h_
private:
    // HASH SET REPRESENTATION
    std::vector<std::list<KeyType>> m_table; // actual table
    HashFunc m_hash; // hash function
    unsigned int m_size; // number of keys

public:
    // HASH SET IMPLEMENTATION
    // Constructor for the table accepts the size of the table. Default
    // constructor for the hash function object is implicitly used.
    ds_hashset(unsigned int init_size = 10) : m_table(init_size), m_size(0) {}  // implemented in lecture or lab

    // Copy constructor just uses the member function copy constructors.
    ds_hashset(const ds_hashset<KeyType, HashFunc>& old)
        : m_table(old.m_table), m_size(old.m_size) {}  // implemented in lecture or lab

    // Insert the key if it is not already there.
    std::pair<iterator, bool> insert(KeyType const& key) {
        // implemented in lecture or lab
        return {this, index, p};
    }

    // Erase the key
    int erase(const KeyType& key) {
        return 0;
    }

    // Erase at the iterator
    void erase(iterator p) {
        // implemented in lecture or lab
        m_table[p.m_index].erase(p.m_list_itr);
    }

    // Find the first entry in the table and create an associated iterator
    iterator begin() {
        // implemented in lecture or lab
        return iterator(this, index, p);
    }

    // Create an end iterator.
    iterator end() {
        iterator p(this);
        p.m_index = -1;
        return p;
    }

    // A public print utility.
    void print(std::ostream & ostr) {
        // implemented in lecture or lab
        for (unsigned int i=0; i<m_table.size(); ++i) {
            ostr << i << " : ";
            for (hash_list_itr p = m_table[i].begin(); p != m_table[i].end(); ++p)
                ostr << *p;
            ostr << std::endl;
        }
    }

    // Find the key, using hash function, indexing and list find
    iterator find(const KeyType& key) {
        // implemented in lecture or lab
        return iterator(this, index, p);
    }