CSCI-1200 Data Structures — Fall 2021
Lecture 22 – Hash Tables, part 1

Announcements: Test 3 Information

- Test 3 will be held Thursday, November 18th from 6-7:50pm.
- Your exam room & zone assignment will be posted on Submitty.
  Note: We will re-shuffle the room & zone assignments from Exams 1 & 2.
- Coverage: Lectures 1-22, Labs 1-12, HW 1-8.
- Studying
  - Practice problems from previous tests are available on the course website.
  - Sample solutions to the practice problems will be posted on Wednesday morning.
  - The best way to prepare is to completely work through and write out your solution to each problem, *before* looking at the answers.
  - You should practice timing yourself as well. The test will be 110 minutes and there will be 100 points. If a problem is worth 25 points, budgeting 25 minutes for yourself to solve the problem is a good time management technique.
  - OPTIONAL: Prepare a 2 page, black & white, 8.5x11", portrait orientation .pdf of notes you would like to have during the test. This may be digitally prepared or handwritten and scanned or photographed. The file may be no bigger than 2MB. You will upload this file to Submitty gradeable "Test 3 Notes Page (Optional)" before Wednesday November 17th @11:59pm. We will print this and attach it to your test. No other notes may be used during the test.
  - Going in to the test, you should know what big topics will be covered on the test. As you skim through the problems, see if you can match up those big topics to each question. Even if you are stumped about how to solve the whole problem, or some of the details of the problem, make sure you demonstrate your understanding of the big topic that is covered in that question.
  - Re-read the problem statement carefully. Make sure you didn't miss anything.
- Bring to the test room:
  - Your mask. Masks must be worn 100% of the time during the exam.
    Please properly wear a well fitted surgical mask or N95 or KN95 mask to the exam room.
    We will have surgical masks at the front of the room if you do not already have one.
  - NOTE: Please use the restroom before entering the exam room.
    Students must remain in their seats until they are ready to turn in their exam.
  - Your Rensselaer photo ID card. We will be checking IDs when you turn in your exam.
  - Pencil(s) & eraser (pens are ok, but not recommended). The test *will* involve handwriting code on paper, short answer problem solving, and may require you to draw a memory diagram. Neat legible handwriting is appreciated. We will be somewhat forgiving to minor syntax errors – it will be graded by humans not computers :)
  - Do not bring your own scratch paper. The exam packet will include sufficient scratch paper.
  - Computers, cell-phones, smart watches, calculators, music players, etc. are not permitted. Please do not bring your laptop, books, backpack, etc. to the test room – leave everything in your dorm room. *Unless you are coming directly from another class or sports/club meeting.*

Review from Lecture 21

- Some variants on the classic data structures...
  - unrolled linked list
  - skip list
  - bounding volume hierarchy
  - trie (a.k.a. prefix tree)

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
- Using a hash table to implement a set/map
  - Iterators, find, insert, and erase

22.1 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*.

22.2 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.

22.3 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.

22.4 Caller ID with an STL Vector

```
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?

22.5 Caller ID with an STL Map

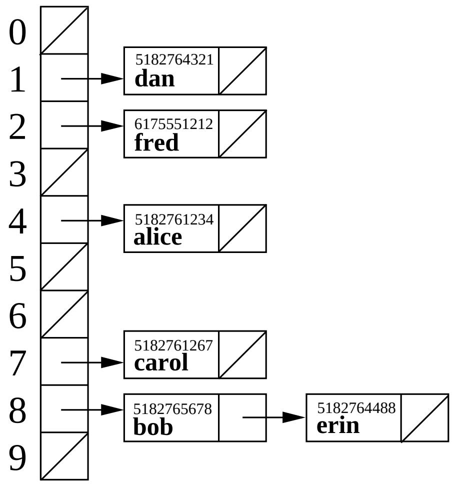
```
std::map<int,std::string> phonebook;

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?

22.6 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;
}

// 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) {




}
```

22.7 Exercise: Choosing a Hash Function

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

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

22.8 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?

22.9 What makes a Good Hash Function?

- Goals: **fast O(1) computation** and a **random-like (but deterministic), 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:

```
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:

```
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.

22.10 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.

- *NOTE: We'll see another method for collision resolution in Lecture 23.*

22.11 Building our own Hash Table: A Hash Set

- The class is templated over both the key type and the hash functor type.
  NOTE: We'll talk about functions vs. functors in Lecture 23. Just pretend it's a function for now!

```
template  < class KeyType, class HashFunc >
class ds_hashset {    ...   };
```

- We use separate chaining for collision resolution. Hence the main data structure inside the class is:

```
std::vector< std::list<KeyType> > m_table;
```

22.12 Hash Set Iterators

- Iterators move through the hash table in the order of the storage locations rather than the chronological order of insertion or a sorted ordering imposed by `operator<`.
- Thus, the visiting/printing order appears random-like, and 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 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.

22.13 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.

22.14 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.

22.15 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.

22.16 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.

22.17 Resize

- Cannot simply call resize on the current vector. Must make a new vector of the correct size, and re-insert each key into the resized vector. Why? **Exercise:** Write `resize()`
- NOTE: 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.

```cpp
#ifndef ds_hashset_h_
#define ds_hashset_h_
// The set class as a hash table instead of a binary search tree.  The
// primary external difference between ds_set and ds_hashset is that
// the iterators do not step through the hashset in any meaningful
// order.  It is just the order imposed by the hash function.
#include <iostream>
#include <list>
#include <string>
#include <vector>
#include <algorithm>

// 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:
  typedef 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 {
  public:
    friend class ds_hashset;   // allows access to private variables
  private:

    // ITERATOR REPRESENTATION
    ds_hashset* m_hs;
    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) : m_hs(hs), m_index(-1) {}
    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=(const iterator& 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 KeyType& 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); }
    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); }
```

```cpp
    // increment and decrement
    iterator& operator++() {
      this->next();
      return *this;
    }
    iterator operator++(int) {
      iterator temp(*this);
      this->next();
      return temp;
    }
    iterator & operator--() {
      this->prev();
      return *this;
    }
    iterator operator--(int) {
      iterator temp(*this);
      this->prev();
      return temp;
    }

  private:
    // Find the next entry in the table
    void next() {
      ++ m_list_itr;  // next item in the list

      // 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 -- ;

      else {
        // 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; ++p;
        for (; p != m_hs->m_table[m_index].end(); ++p, ++m_list_itr) {}
      }
    }
  };
  // end of ITERATOR CLASS
  // =================================================================
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) {}

  // 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) {}

  ˜ds_hashset() {}

  ds_hashset& operator=(const ds_hashset<KeyType,HashFunc>& old) {
    if (&old != this) {
      this->m_table = old.m_table;
      this->m_size = old.m_size;
      this->m_hash = old.m_hash;
    }
    return *this;
  }

  unsigned int size() const { return m_size; }


  // Insert the key if it is not already there.
  std::pair< iterator, bool > insert(KeyType const& key) {
    const float LOAD_FRACTION_FOR_RESIZE = 1.25;
    if (m_size >= LOAD_FRACTION_FOR_RESIZE * m_table.size())
      this->resize_table(2*m_table.size()+1);
    // implemented in Lecture 22 & Lab 12












  }

  // Find the key, using hash function, indexing and list find
  iterator find(const KeyType& key) {
    unsigned int hash_value = m_hash(key);
    unsigned int index = hash_value % m_table.size();
    hash_list_itr p = std::find(m_table[index].begin(),
                                m_table[index].end(), key);
    if (p == m_table[index].end())
      return this->end();
    else
      return iterator(this, index, p);
  }

  // Erase the key
  int erase(const KeyType& key) {
    // Find the key and use the erase iterator function.
    iterator p = find(key);
    if (p == end())
      return 0;
    else {
      erase(p);
      return 1;
    }
  }

  // Erase at the iterator
  void erase(iterator p) {
    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 Lab 12








  }

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

  // A public print utility.
  void print(std::ostream & ostr) {
    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;
    }
  }

private:
  // resize the table with the same values but a
  void resize_table(unsigned int new_size) {
    // implemented in Lab 12











  }
};
#endif
```