HASHING using STL

1. 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*.

2.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.

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 scaledup 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;
```

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?

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?

6. Now let’s implement Caller ID with a Hash Table

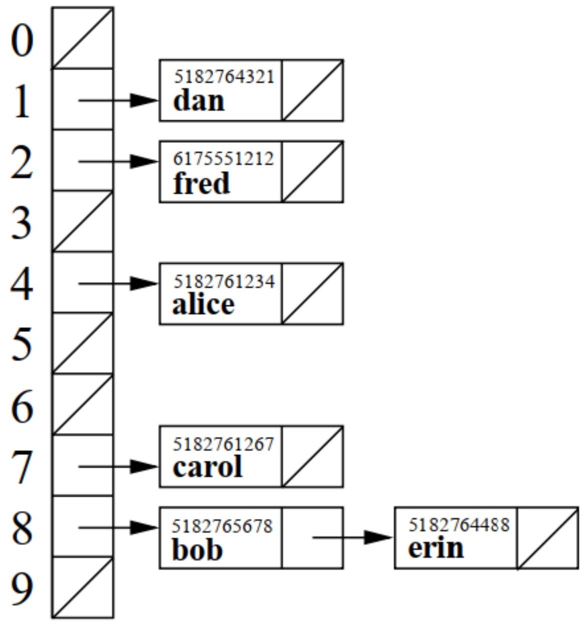
```
#define PHONEBOOK_SIZE 10
class Node {
public:
  int number;
  string name;
  Node* next;
};

// 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) {
}
```



7. 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
return k % N;      automatic
}
```

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; for (unsigned int i=0; i<k.size(); ++i)
}
```

- 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.

8. 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 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.

9. Building our own Hash Table: A Hash Set

- The class is templated over both the key type and the hash functor type.
  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;

10. 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.

11. 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.

12. 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.

13. 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.

14. 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.

15. 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.

16. Using STL's for each

- First, here's a tiny helper function:

```
void float_print (float f) {
std::cout << f << std::endl;
}
```

- Let's make an STL vector of floats:

```
std::vector<float> my_data;
my_data.push_back(3.14);
my_data.push_back(1.41);
my_data.push_back(6.02);
my_data.push_back(2.71);
```

- Now we can write a loop to print out all the data in our vector:

```
std::vector<float>::iterator itr;
for (itr = my_data.begin(); itr != my_data.end(); itr++) {
float_print(*itr); }
```

- Alternatively we can use it with STL's for_each function to visit and print each element:

```
std::for_each(my_data.begin(), my_data.end(), float_print);
```

Wow! That's alot less to type. Can I stop using regular for and while loops altogether?

- We can actually also do the same thing without creating & explicitly naming the float_print function. We create an *anonymous function* using *lambda*:

```
std::for_each(my_data.begin(), my_data.end(), [](float f){ std::cout << f << std::end; });
```

Lambda is new to the C++ language (part of C++11). But lambda is a core piece of many classic, older programming languages including Lisp and Scheme. Python lambdas and Perl anonymous subroutines are similar. (In fact lambda dates back to the 1930's, before the first computers were built!) You'll learn more about lambda more in later courses like CSCI 4430 Programming Languages!

17. Function Objects, a.k.a. *Functors*

- In addition to the basic mathematical operators + - * / < > , another operator we can overload for our C++ classes is the *function call operator*. Why do we want to do this? This allows instances or objects of our class, to be used like functions. It's weird but powerful.
- Here's the basic syntax. Any specific number of arguments can be used.

```
class my_class_name {
public:
// ... normal class stuff ...
my_return_type operator() ( /* my list of args */ );
};
```

18. Why are Functors Useful?

- One example is the default 3rd argument for std::sort. We know that by default STL's sort routines will use the less than comparison function for the type stored inside the container. How exactly do they do that?
- First let's define another tiny helper function:

```
bool float_less(float x, float y) {
return x < y;
}
```

- Remember how we can sort the my_data vector defined above using our own homemade comparison function for sorting:

```
std::sort(my_data.begin(),my_data.end(),float_less);
```

If we don't specify a 3rd argument:

```
std::sort(my_data.begin(),my_data.end());
```

This is what STL does by default:

```
std::sort(my_data.begin(),my_data.end(),std::less<float>());
```

- What is std::less? It's a templated class. Above we have called the default constructor to make an instance of that class. Then, that instance/object can be used like it's a function. Weird!
- How does it do that? std::less is a teeny tiny class that just contains the overloaded function call operator.

```
template <class T> class less { public:
bool operator() (const T& x, const T& y) const { return x < y; }
};
```

You can use this instance/object/functor as a function that expects exactly two arguments of type T (in this example float) that returns a bool. That's exactly what we need for std::sort! This ultimately does the same thing as our tiny helper homemade compare function (but for any type T)!

19. Another more Complicated Functor Example

- Constructors of function objects can be used to specify *internal data* for the functor that can then be used during computation of the function call operator! For example:

```
class between_values {
private:
float low, high;
public:
between_values(float l, float h) : low(l), high(h) {}
bool operator() (float val) { return low <= val && val <= high; }
};
```

- The range between low & high is specified when a functor/an instance of this class is created. We might have multiple different instances of the between_values functor, each with their own range. Later, when the functor is used, the query value will be passed in as an argument. The function call operator accepts that single argument val and compares against the internal data low & high.

- STL has a find function that can be used for vectors to find a specific value (it simply loops over the structure using an iterator in *O*(*n*) time).

```
std::vector<int>::iterator itr;
itr = std::find(my_data.begin(), my_data.end(), 3);
if (itr != my_data.end()) {
std::cout << "Yes, the value 3 is in this container." << endl;
}
```

- STL also has a find_if construct that we can use with our between_values functor. For example: between_values two_and_four(2,4);

```
if (std::find_if(my_data.begin(), my_data.end(), two_and_four) != my_data.end()) {
std::cout << "Found a value greater than 2 & less than 4!" << std::endl;
}
```

- Alternatively, we could create the functor without giving it a variable name. And in the use below we also capture the return value to print out the first item in the vector inside this range. Note that it does not print all values in the range.

```
std::vector<float>::iterator itr;
itr = std::find_if(my_data.begin(), my_data.end(), between_values(2,4)); if (itr != my_data.end())
{
std::cout << "my_data contains " << *itr
<< ", a value greater than 2 & less than 4!" << std::endl;
}
```

\Weird Things we can do in C++" Finished { Now back to Hash Tables!

20. 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!)
- For many types STL has a good default hash function, so in those cases you do not need to provide your own hash function. But sometimes we do want to write our own.

21. Writing our own Hash Functions or Hash Functors

- 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: *Note: This implementation comes from* http://www.partow.net/programming/hashfunctions/

```
unsigned int MyHashFunction(std::string const& key) {
  unsigned int hash = 1315423911;
  for(unsigned int i = 0; i < key.length(); i++)
      hash ^= ((hash << 5) + key[i] + (hash >> 2));
  return hash;
}
```

- Alternately, this same string hash code can be written as a functor { which is just a class wrapper around a function, and the function is implemented as the overloaded function call operator for the class.

```
class MyHashFunctor {
public:
  unsigned int operator() (std::string const& key) const {
    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 MyHashFunctor as the second template parameter to the declaration of a ds_hashset. E.g.,

```
ds_hashset<std::string, MyHashFunctor> my_hashset;
```

22. Using STL's Associative Hash Table (Unordered Map)

- Using the default std::string hash function.
  - With no specified initial table size.
    ```
    std::unordered_map<std::string,Foo> m;
    ```
  - Optionally specifying initial (minimum) table size.
    ```
    std::unordered_map<std::string,Foo> m(1000);
    ```
- Using a home-made std::string hash function. Note: We are required to specify the initial table size.
  - Manually specifying the hash function type.
    ```
    std::unordered_map<std::string,Foo,std::function<unsigned int(std::string)>
                                  m(1000, MyHashFunction);
    ```
  - Using the decltype specifier to get the \declared type of an entity".
    ```
    std::unordered_map<std::string,Foo,decltype(&MyHashFunction)>
                                 m(1000, MyHashFunction);
    ```
- Using a home-made std::string hash functor or function object.
  - With no specified initial table size.
    ```
    std::unordered_map<std::string,Foo,MyHashFunctor> m;
    ```
  - Optionally specifying initial (minimum) table size.
    ```
    std::unordered_map<std::string,Foo,MyHashFunctor> m(1000);
    ```
- Note: In the above examples we're creating an association between two types (STL strings and custom Foo object). If you'd like to just create a set (no associated 2nd type),

simply switch from unordered_map to unordered_set and remove the Foo from the template type in the examples above.

23. How do we Resolve Collisions? METHOD 1: Separate Chaining (Also discussed earlier)

- Each table location stores a linked list of keys (and values) hashed to that location. 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. How do we Resolve Collisions? METHOD 2: Open Addressing

- Let's eliminate the individual memory allocations and pointer indirection / dereferencing that are necessary for separate chaining. This will improve memory / data access performance.
- We will directly store the data (key/key-value pair) in the the top level vector, and store at most one item per index/location.
- When the chosen table index/location already stores a key (or key-value pair), we will seek a different table location 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, ...
  - *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*3)%N, (i+4*4)%N, ...

More generally, the *j*th \probe" of the table is (i + c1j + c2j2) mod N where c1 and c2 are constants.

  - *Secondary hashing*: When a collision occurs a second hash function is applied to compute a new table location. If that location is also full, we go to a third hash function, etc. This is repeated until an empty location is found.
    We can generate a sequence/family of hash functions by swapping in a fixed random-like sequence of constant values (e.g., big primes) into the same general function structure.
- 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 (\never occupied"), *find* may fail to return elements that are actually in the table. Formerly-occupied locations may (and should) be reused, but only after the *find* search operation has been run to

completion (either finding the element or encountering a \never occupied" location) to determine the item is definitely not in the table.

- When using open addressing it is critical to monitor *how full* the table is - specifically the counts of “currentlyoccupied", “formerly occupied", and \never occupied" locations.
  - Hash table performance degrades when the sum of counts of currently and formerly occupied cells is high (e.g., greater than 80%).
  - These operations will fail completely if the table is full (no \never occupied" locations remain).
  - For performance critical applications, it is helpful to run benchmark tests with real-world data (number of insert/find/erase operations, typical values, specific hash function, actual hardware, etc.) to determine the optimal table size and capacity threshhold to balance memory usage and running time. Determine when to *resize* the table { increase (or decrease) the table size to better fit the number of values currently held in the table. Or only *groom* the data { recreate the table at the current size, and re-insert all values so all *formerly occupied* labels can be cleared. A maximum allowed hash table load factor as low as 50% or 60% might be appropriate for some applications.

25. Collision Resolution: Separate Chaining vs. Open Addressing - Discussion

- Advantages of open addressing over separate chaining:
  - No linked lists! No pointers! It’s faster! (Indirect memory accesses are slow!)
- Problems with open addressing:
  - Memory cache performance can be poor when we are jumping around unpredictably in the top level array.
  - Cost of computing new hash values (linear < quadratic < secondary hashing).
  - Capacity and performance must be closely monitored. Expensive re-sizing and grooming may be necessary.
  - Careful testing and parameter tuning is necessary to achieve optimal memory/speed performance.

REFERENCES

1. **https://www.geeksforgeeks.org/hashing-data-structure/**
2. **https://www.geeksforgeeks.org/stdhash-class-in-c-stl/**
3. **https://www.javatpoint.com/hashing-in-data-structure**
4. **https://en.cppreference.com/w/cpp/utility/hash**
5. **https://www.geeksforgeeks.org/hash-table-vs-stl-map/**
6. **https://www.codechef.com/problems/AUHASH**
7. **https://www.interviewbit.com/problems/valid-sudoku/**
8. **https://www.cs.rpi.edu/**

```
#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); }

    // 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
```