

CSCI-1200 Computer Science II — Fall 2008

Lecture 20 – Hash Tables, Part II

Review from Lecture 19

- Operators as non-member functions, as member functions, and as friend functions.
- A hash table is a table implementation with *constant time access*. 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 *don't* store the data in sorted order.
- A hash table is implemented with an array at the top level. Each key is mapped to a slot in the array by a *hash function*, a simple function of one argument (the key) which returns an index (a bucket or slot in the array).
- CallerID Performance: Vectors vs. Binary Search Trees vs. Hash Tables

Today's Lecture

- Collision resolution: separate chaining vs open addressing
- Using a hash table to implement a *set*.
 - Function objects
 - Iterators, find, insert, and erase

Reading: Ford&Topp, Chapter 12 through Section 12.5.

20.1 What makes a Good Hash Function?

- Goals: (1) fast computation, and (2) a random, uniform distribution of keys throughout the table, *despite the actual distribution of keys that are to be stored*.
- For example, using: $f(k) = \text{abs}(k)\%N$ 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.

20.2 How do we Resolve Collisions?

When two keys map to the same table location, we have a *collision*. Two classes of approaches to collision resolution are commonly used:

- In *separate chaining*, each table location stores a list or vector of the keys (and values) that are hashed to that location.
- In *open addressing*, when a table location already stores a key (and its associated value, if any), a different table location is sought in order to store the new value.

20.3 Collision Resolution: Separate Chaining

- Each table location stores a list of keys (and values) hashed to it. Thus, the hashing function really just selects the list to check.
- 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 number of items stored.

20.4 Collision Resolution: Open Addressing

- Here are three different open addressing variations to handle a collision during an *insert* operation:
 - *Linear probing*: If i is the hash location then the following sequence of table locations is tested (“probed”):
 $(i+1)\%N, (i+2)\%N, (i+3)\%N, \dots$
until an empty location is found.
 - *Quadratic probing*: If i is the hash location then the following sequence of table locations is tested:
 $(i+1)\%N, (i+2^2)\%N, (i+3^2)\%N, (i+4^2)\%N, \dots$
More generally, the j^{th} “probe” of the table is $(i + c_1j + c_2j^2) \bmod N$ 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.
- The *erase* function must mark a location as “formerly occupied”. If a location is marked empty, instead, *find* may fail. 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.

20.5 A Set As a Hash Table

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

```
template < typename KeyType, typename HashFunc >
class cs2hashset {
    ...
```
- We use separate chaining for collision resolution. Hence the main data structure inside the class is:

```
std::vector< std::list<KeyType> > m_table;
```
- We will use automatic resizing to handle the possibility that our table is too full. This resize is analogous to the automatic reallocation that occurs inside the vector `push_back` function.

20.6 Function Objects

- Our hash function is implemented as an object with a function call operator.
- The basic form of the function call operator is shown below. Any specific number of arguments can be used.

```
class name {
public:
    // ...

    return_type operator() ( *** args *** );
};
```

- Here is an example of a templated function object implementing the less-than comparison operation. This is the default 3rd argument to `std::sort`.

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

- Constructors of functions objects can be used to specify parameters for use in comparisons. For example

```
class between_values {
private:
    int x, y;
public:
    between_values(int in_x, int in_y) : x(in_x), y(in_y) {}
    bool operator() (int z) { return x <= z && z <= y; }
}
```

This can be used in combination with `find_if`. For example if `v` is a vector of integers, then

```
between_values low_range(-99, 99);
if (std::find_if(v.begin(), v.end(), low_range) != v.end())
    std::cout << "A value between -99 and 99 is in the vector.\n";
```

20.7 Our Hash Function Object

```
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;
    }
};
```

The type `hash_string_obj` is one of the template parameters to the declaration of a `cs2hashset`. E.g.,

```
cs2hashset<std::string, hash_string_obj> my_hashset;
```

20.8 Hash Set Iterators

- Iterators move through the hash table in the order the values are stored rather than the ordering imposed by (say) an `operator<`. The order depends on the hash function and the table size.
 - Hence the increment operators must move to the next entry in the current 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 `cs2hashset` 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 `cs2hashset`, this does not mean an iterator automatically knows about any particular `cs2hashset`.
- The iterator must also store:
 - 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 `cs2hashset` class as a friend, but the reverse is unnecessary.

20.9 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 `cs2hashset` 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.

20.10 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.

20.11 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 list 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.

20.12 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.

20.13 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()`

20.14 Hash Table Iterator Invalidation

- Any insert operation invalidates *all* `cs2hashset` iterators because the insert operation could cause a resize of the table.
- The erase function only invalidates an iterator that references the current object.

20.15 Summary

- The algorithmic steps are not particularly difficult.
- The use of C++ is more sophisticated than recent examples.
- The standard library is exploited to simplify the implementation.
- This imposes some hidden costs in the implementation, the most substantial being in `resize` where
 - lists are copied and then reallocated, and
 - hash function values are recomputed.

```

#define cs2hashset_h_
#define cs2hashset_h_

// The set class as a hash table instead of a binary search tree. The
// primary external difference between cs2set and cs2hashset 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>

// The cs2hashset is templated over both the type of key and the type
// of the hash function, a function object.
template < typename KeyType, typename HashFunc >
class cs2hashset {
private:
    typedef typename std::list<KeyType>::iterator hash_list_itr;
public:
    // =====
    // THE ITERATOR CLASS
    // Defined as a nested class and does not have separately templated.
    class iterator {
    public:
        friend class cs2hashset; // allows access to private variables
    private:
        // ITERATOR REPRESENTATION
        cs2hashset* 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 cs2hashset only
        iterator(cs2hashset* hs) : m_hs(hs), m_index(-1) {}
        iterator(cs2hashset* 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& 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 --;
}
}
// 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.
cs2hashset(unsigned int init_size = 10) : m_table(init_size), m_size(0) {}

// Copy constructor just uses the member function copy constructors.
cs2hashset(const cs2hashset<KeyType, HashFunc>& old)
: m_table(old.m_table), m_size(old.m_size) {}

~cs2hashset() {}

cs2hashset& operator=(const cs2hashset<KeyType, HashFunc>& old) {
    if (&old != this)
        *this = old;
}

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 class
}

// 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 class
}

// 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 class
}
};
#endif

```