COMSC-200
Lab 11

Ryan Jacoby

29 November 2020

1 E15.16

Implementation of **HashTable::***insert* to keep the load factor between 0.5 and 1. New function doubles or halves bucket ammount and re-hashes all of the data from the old buckets.

```
void HashTable::insert(const string& x) {
   int h = (98 * hash_code(x) + 460) % 997;
   h = h % buckets.size();
   if (h < 0) { h = -h; }

   Node* current = buckets[h];
   while (current != nullptr)
   {
      if (current-> data == x) { return; }
         // Already in the set
      current = current->next;
   }
   Node* new_node = new Node;
   new_node->data = x;
   new_node->next = buckets[h];
   buckets[h] = new_node;
   current_size++;

   double load_factor = (1.0 * current_size) / buckets.size();

   if(load_factor > 1) {
      vector<Node*> tmp = buckets;

      for (int i = 0; i < buckets.size() / 2; i++) {
         tmp.push_back(nullptr);
      }

      Node* previous = nullptr;
      for (Iterator iter = this->begin(); !iter.equals(this->end()); iter.next())
    {
         delete previous;
         previous = iter.current;
         h = (98 * hash_code(iter.get()) + 460) % 997;
         h = h % tmp.size();
         if (h < 0) { h = -h; }

         Node* tcurrent = tmp[h];
         while (tcurrent != nullptr) {
            if (tcurrent-> data == iter.get()) { return; }
```

```
               // Already in the set
            tcurrent = tcurrent->next;
         }
         Node* tnew_node = new Node;
         tnew_node->data = iter.get();
         tnew_node->next = tmp[h];
         tmp[h] = tnew_node;
      }

      buckets = tmp;
   }

   if(load_factor < 0.5) {
      vector<Node*> tmp = buckets;

      int num_buckets = buckets.size();
      for (int i = 0; i < num_buckets; i++) {
         buckets.push_back(nullptr);
      }

      Node* previous = nullptr;
      for (Iterator iter = this->begin(); !iter.equals(this->end()); iter.next())
    {
         delete previous;
         previous = iter.current;
         h = (98 * hash_code(iter.get()) + 460) % 997;
         h = h % tmp.size();
         if (h < 0) { h = -h; }

         Node* tcurrent = tmp[h];
         while (tcurrent != nullptr) {
            if (tcurrent-> data == iter.get()) { return; }
               // Already in the set
            tcurrent = tcurrent->next;
         }
         Node* tnew_node = new Node;
         tnew_node->data = iter.get();
         tnew_node->next = tmp[h];
         tmp[h] = tnew_node;
      }

      buckets = tmp;
   }

}
```

Listing 1: HashTable::insert

2 E15.17

The following code replaces the code to determine h in **HashTable::***erase*, **HashTable::***insert*, and **HashTable::***count*

```
int h = (3 * hash_code<T>(x) + 5) % 99173;
h = h % buckets.size();
if (h < 0) { h = -h; }
```

Listing 2: h code

3 E15.18

There were less collisions using MAD; there were 74684 collisions while using no compressions and 74507 collisions while using MAD hash compression.

4 P15.11 and p15.13

```
#ifndef HASHTABLE_H
#define HASHTABLE_H

#include <string>
#include <vector>

using namespace std;

/**
    Computes the hash code for a string.
    @param str a string
    @return the hash code
*/
template <class T>
int hash_code(const T& str);

template <class T>
class HashTable;

template <class T>
class Iterator;

template <class T>
class Node
{
private:
    T data;
    Node<T>* next;

friend class HashTable<T>;
friend class Iterator<T>;
};

template <class T>
class Iterator
{
public:
    /**
        Looks up the value at a position.
        @return the value of the node to which the iterator points
    */
    T get() const;
    /**
        Advances the iterator to the next node.
    */
    void next();
    /**
        Compares two iterators.
        @param other the iterator to compare with this iterator
        @return true if this iterator and other are equal
```

```cpp
    */
    bool equals(const Iterator& other) const;
private:
    const HashTable<T>* container;
    int bucket_index;
    Node<T>* current;

friend class HashTable<T>;
};

/**
    This class implements a hash table using separate chaining.
*/
template <class T>
class HashTable
{
public:
    /**
        Constructs a hash table.
        @param nbuckets the number of buckets
    */
    HashTable(int nbuckets);

    /**
        Tests for set membership.
        @param x the potential element to test
        @return 1 if x is an element of this set, 0 otherwise
    */
    int count(const T& x);

    /**
        Adds an element to this hash table if it is not already present.
        @param x the element to add
    */
    void insert(const T& x);

    /**
        Removes an element from this hash table if it is present.
        @param x the potential element to remove
    */
    void erase(const T& x);

    /**
        Returns an iterator to the beginning of this hash table.
        @return a hash table iterator to the beginning
    */
    Iterator<T> begin() const;

    /**
        Returns an iterator past the end of this hash table.
        @return a hash table iterator past the end
    */
    Iterator<T> end() const;

    /**
        Gets the number of elements in this set.
        @return the number of elements
    */
    int size() const;

    int getCollisions();

    ~HashTable();
    HashTable<T>& operator=(HashTable ht);
    HashTable(HashTable &ht);

private:
    vector<Node<T>*> buckets;
    int current_size;
    int collisions;

friend class Iterator<T>;
};

#endif
```

Listing 3: hashtable.h

```cpp
#include<iostream>

#include "hashtable.h"

template<class T>
int hash_code(const T& str)
{
    int h = 0;
    for (int i = 0; i < str.length(); i++)
    {
        h = 31 * h + str[i];
    }
    return h;
}

template <class T>
HashTable<T>::HashTable(int nbuckets)
{
    for (int i = 0; i < nbuckets; i++)
    {
        buckets.push_back(nullptr);
    }
    current_size = 0;
    collisions = 0;
}

template <class T>
int HashTable<T>::count(const T& x)
{
    int h = (3 * hash_code<T>(x) + 5) % 99173;
    h = h % buckets.size();
    if (h < 0) { h = -h; }

    Node<T>* current = buckets[h];
    while (current != nullptr)
    {
        if (current->data == x) { return 1; }
        current = current->next;
    }
    return 0;
}

template <class T>
void HashTable<T>::insert(const T& x)
{
    int h = (3 * hash_code<T>(x) + 5) % 99173;
    h = h % buckets.size();
    if (h < 0) { h = -h; }

    Node<T>* current = buckets[h];
    if (current != nullptr) collisions++;
    while (current != nullptr)
    {
        if (current-> data == x) { return; }
            // Already in the set
        current = current->next;
    }
    Node<T>* new_node = new Node<T>;
    new_node->data = x;
    new_node->next = buckets[h];
    buckets[h] = new_node;
    current_size++;
}

template <class T>
void HashTable<T>::erase(const T& x)
{
    int h = (3 * hash_code<T>(x) + 5) % 99173;
    h = h % buckets.size();
    if (h < 0) { h = -h; }

    Node<T>* current = buckets[h];
    Node<T>* previous = nullptr;
    while (current != nullptr)
    {
        if (current->data == x)
        {
            if (previous == nullptr)
            {
            buckets[h] = current->next;
            }
            else
            {
            previous->next = current->next;
            }
            delete current;
            current_size--;
            return;
        }
        previous = current;
        current = current->next;
    }
}

template <class T>
int HashTable<T>::size() const
{
    return current_size;
}
```

```cpp

template <class T>
HashTable<T>::~HashTable() {
        Node<T>* previous = nullptr;
        for (Iterator<T> iter = this->begin(); !iter.equals(this->end()); iter.next()) {
            delete previous;
            previous = iter.current;
        }
}

template <class T>
HashTable<T>& HashTable<T>::operator=(HashTable ht) {
    buckets = ht.buckets;
    current_size = ht.current_size;
    collisions = ht.collisions;
    return *this;
}

template <class T>
HashTable<T>::HashTable(HashTable &ht) {
    buckets = ht->buckets;
    current_size = ht->current_size;
    collisions = ht->current_size;
}

template <class T>
Iterator<T> HashTable<T>::begin() const
{
    Iterator<T> iter;
    iter.current = nullptr;
    iter.bucket_index = -1;
    iter.container = this;
    iter.next();
    return iter;
}

template <class T>
Iterator<T> HashTable<T>::end() const
{
    Iterator<T> iter;
    iter.current = nullptr;
    iter.bucket_index = buckets.size();
    iter.container = this;
    return iter;
}

template <class T>
int HashTable<T>::getCollisions() {
    return collisions;
}

template <class T>
T Iterator<T>::get() const
{
    return current->data;
}

template <class T>
```

```cpp
bool Iterator<T>::equals(const Iterator& other) const
{
    return current == other.current;
}

template <class T>
void Iterator<T>::next()
{
    if (bucket_index >= 0 && current->next != nullptr)
    {
        // Advance in the same bucket
        current = current->next;
    }
    else
    {
        // Move to the next bucket
        do
        {
            bucket_index++;
        }
        while (bucket_index < container->buckets.size()
            && container->buckets[bucket_index] == nullptr);
        if (bucket_index < container->buckets.size())
        {
            // Start of next bucket
            current = container->buckets[bucket_index];
        }
        else
        {
            // No more buckets
            current = nullptr;
        }
    }
}
```

Listing 4: hashtable.cpp