# 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; }
9
            // Already in the set
        current = current->next;
12
     }
     Node * new_node = new Node;
     new_node->data = x;
14
     new_node->next = buckets[h];
     buckets[h] = new_node;
16
     current_size++;
17
18
     double load_factor = (1.0 * current_size) / buckets.size();
19
20
21
     if(load_factor > 1) {
22
        vector < Node *> tmp = buckets;
23
        for (int i = 0; i < buckets.size() / 2; i++) {</pre>
24
            tmp.push_back(nullptr);
27
        Node* previous = nullptr;
28
        for (Iterator iter = this->begin(); !iter.equals(this->end()); iter.next())
29
            delete previous;
30
31
            previous = iter.current;
            h = (98 * hash\_code(iter.get()) + 460) \% 997;
32
            h = h % tmp.size();
33
            if (h < 0) { h = -h; }
34
35
            Node* tcurrent = tmp[h];
36
            while (tcurrent != nullptr) {
               if (tcurrent-> data == iter.get()) { return; }
```

```
// Already in the set
39
40
               tcurrent = tcurrent->next;
            }
41
42
            Node* tnew_node = new Node;
            tnew_node->data = iter.get();
43
            tnew_node->next = tmp[h];
44
            tmp[h] = tnew_node;
45
         }
46
47
48
         buckets = tmp;
49
50
      if(load_factor < 0.5) {</pre>
         vector < Node *> tmp = buckets;
53
54
         int num_buckets = buckets.size();
         for (int i = 0; i < num_buckets; i++) {</pre>
55
            buckets.push_back(nullptr);
56
         }
57
58
         Node* previous = nullptr;
         for (Iterator iter = this->begin(); !iter.equals(this->end()); iter.next())
61
            delete previous;
            previous = iter.current;
62
            h = (98 * hash\_code(iter.get()) + 460) \% 997;
63
            h = h \% tmp.size();
64
            if (h < 0) { h = -h; }
65
66
            Node* tcurrent = tmp[h];
67
            while (tcurrent != nullptr) {
68
               if (tcurrent-> data == iter.get()) { return; }
69
                   // Already in the set
                tcurrent = tcurrent->next;
            }
            Node * tnew_node = new Node;
74
            tnew_node->data = iter.get();
            tnew_node ->next = tmp[h];
            tmp[h] = tnew_node;
76
         }
78
79
         buckets = tmp;
     }
80
81
82 }
```

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; }</pre>
```

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

```
1 #ifndef HASHTABLE_H
2 #define HASHTABLE_H
4 #include <string>
5 #include <vector>
7 using namespace std;
9 /**
      Computes the hash code for a string.
      Oparam str a string
12
      Oreturn the hash code
13 */
14 template <class T>
int hash_code(const T& str);
17 template <class T>
18 class HashTable;
20 template <class T>
21 class Iterator;
23 template <class T>
24 class Node
26 private:
      T data;
27
      Node <T>* next;
30 friend class HashTable <T>;
31 friend class Iterator<T>;
32 };
33
34 template <class T>
35 class Iterator
36 {
37 public:
38
           Looks up the value at a position.
39
           Oreturn the value of the node to which the iterator points
40
      */
41
      T get() const;
42
43
           Advances the iterator to the next node.
44
45
46
      void next();
47
           Compares two iterators.
48
           Oparam other the iterator to compare with this iterator
49
           Oreturn true if this iterator and other are equal
```

```
const HashTable <T>* container;
      int bucket_index;
     Node <T>* current;
56
58 friend class HashTable <T>;
59 };
60
61 /**
This class implements a hash table using separate chaining.
63 */
64 template <class T>
65 class HashTable
66 {
67 public:
68
          Constructs a hash table.
69
          Oparam nbuckets the number of buckets
70
       */
71
      HashTable(int nbuckets);
72
73
          Tests for set membership.
75
          @param x the potential element to test
76
          Oreturn 1 if x is an element of this set, 0 otherwise
77
78
       int count(const T& x);
79
       /**
81
           Adds an element to this hash table if it is not already present.
82
           Oparam x the element to add
83
84
       void insert(const T& x);
85
86
       /**
          Removes an element from this hash table if it is present.
88
          Oparam x the potential element to remove
89
90
       void erase(const T& x);
91
92
       /**
          Returns an iterator to the beginning of this hash table.
94
          Oreturn a hash table iterator to the beginning
95
96
       Iterator <T> begin() const;
97
98
99
           Returns an iterator past the end of this hash table.
           Oreturn a hash table iterator past the end
101
102
       Iterator <T> end() const;
103
104
          Gets the number of elements in this set.
106
           Oreturn the number of elements
107
108
    int size() const;
109
```

```
110
       int getCollisions();
111
112
113
       ~HashTable();
       HashTable <T>& operator = (HashTable ht);
114
       HashTable(HashTable &ht);
115
116
117 private:
       vector < Node < T > * > buckets;
118
       int current_size;
       int collisions;
121
122 friend class Iterator <T>;
123 };
124
125 #endif
```

Listing 3: hashtable.h

```
#include < iostream >
3 #include "hashtable.h"
5 template < class T>
6 int hash_code(const T& str)
7 {
      int h = 0;
      for (int i = 0; i < str.length(); i++)</pre>
9
10
           h = 31 * h + str[i];
11
12
13
      return h;
14 }
16 template <class T>
17 HashTable <T>::HashTable(int nbuckets)
      for (int i = 0; i < nbuckets; i++)</pre>
19
20
           buckets.push_back(nullptr);
23
      current_size = 0;
24
      collisions = 0;
25 }
26
27 template <class T>
28 int HashTable <T>::count(const T& x)
29 {
      int h = (3 * hash_code < T > (x) + 5) % 99173;
30
      h = h % buckets.size();
31
       if (h < 0) { h = -h; }</pre>
32
33
      Node < T > * current = buckets[h];
       while (current != nullptr)
36
           if (current->data == x) { return 1; }
37
           current = current->next;
38
      }
39
  return 0;
```

```
41 }
42
43 template <class T>
44 void HashTable <T>::insert(const T& x)
       int h = (3 * hash_code < T > (x) + 5) % 99173;
46
      h = h % buckets.size();
47
      if (h < 0) { h = -h; }
48
49
      Node < T > * current = buckets[h];
51
      if (current != nullptr) collisions++;
      while (current != nullptr)
52
53
           if (current-> data == x) { return; }
54
               // Already in the set
55
           current = current->next;
57
      Node <T>* new_node = new Node <T>;
58
      new_node->data = x;
59
      new_node ->next = buckets[h];
60
      buckets[h] = new_node;
61
       current_size++;
62
63 }
65 template <class T>
66 void HashTable <T>::erase(const T& x)
67 {
      int h = (3 * hash_code < T > (x) + 5) % 99173;
68
      h = h % buckets.size();
69
      if (h < 0) { h = -h; }
71
      Node < T > * current = buckets[h];
72
      Node < T > * previous = nullptr;
73
       while (current != nullptr)
74
75
76
           if (current->data == x)
77
               if (previous == nullptr)
78
79
               buckets[h] = current->next;
80
81
               else
82
               {
83
               previous ->next = current ->next;
84
85
               delete current;
86
               current_size --;
87
               return;
           previous = current;
           current = current->next;
91
92
93 }
94
95 template <class T>
96 int HashTable <T>::size() const
97 {
98
      return current_size;
99 }
```

```
101 template <class T>
102 HashTable <T>::~HashTable() {
           Node < T > * previous = nullptr;
           for (Iterator<T> iter = this->begin(); !iter.equals(this->end()); iter.
104
      next()) {
               delete previous;
                previous = iter.current;
106
           }
107
108 }
110 template <class T>
111 HashTable <T>& HashTable <T>::operator = (HashTable ht) {
   buckets = ht.buckets;
       current_size = ht.current_size;
113
       collisions = ht.collisions;
114
       return *this;
115
116 }
117
118 template <class T>
HashTable <T>::HashTable (HashTable &ht) {
       buckets = ht->buckets;
       current_size = ht->current_size;
122
       collisions = ht->current_size;
123 }
124
125 template <class T>
126 Iterator <T> HashTable <T>::begin() const
       Iterator <T> iter;
       iter.current = nullptr;
129
      iter.bucket_index = -1;
130
       iter.container = this;
131
       iter.next();
132
       return iter;
133
134 }
135
136 template <class T>
137 Iterator <T> HashTable <T>::end() const
138 {
       Iterator <T> iter;
139
       iter.current = nullptr;
140
       iter.bucket_index = buckets.size();
141
       iter.container = this;
142
143
       return iter;
144 }
146 template <class T>
int HashTable <T>::getCollisions() {
148
       return collisions;
149 }
150
151 template <class T>
152 T Iterator <T>::get() const
153 {
       return current ->data;
155 }
156
157 template <class T>
```

```
bool Iterator<T>::equals(const Iterator& other) const
       return current == other.current;
161 }
162
163 template <class T>
void Iterator <T>::next()
165
       if (bucket_index >= 0 && current->next != nullptr)
166
167
168
           // Advance in the same bucket
           current = current->next;
169
       }
170
       else
171
       {
172
           // Move to the next bucket
173
174
           do
           {
175
                bucket_index++;
176
177
           while (bucket_index < container->buckets.size()
178
                && container->buckets[bucket_index] == nullptr);
179
           if (bucket_index < container->buckets.size())
181
                // Start of next bucket
182
                current = container ->buckets[bucket_index];
183
           }
184
           else
185
           {
186
                // No more buckets
187
                current = nullptr;
188
           }
189
       }
190
191 }
```

Listing 4: hashtable.cpp