

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.

```
1 void HashTable::insert(const string& x) {
2     int h = (98 * hash_code(x) + 460) % 997;
3     h = h % buckets.size();
4     if (h < 0) { h = -h; }
5
6     Node* current = buckets[h];
7     while (current != nullptr)
8     {
9         if (current->data == x) { return; }
10        // Already in the set
11        current = current->next;
12    }
13    Node* new_node = new Node;
14    new_node->data = x;
15    new_node->next = buckets[h];
16    buckets[h] = new_node;
17    current_size++;
18
19    double load_factor = (1.0 * current_size) / buckets.size();
20
21    if(load_factor > 1) {
22        vector<Node*> tmp = buckets;
23
24        for (int i = 0; i < buckets.size() / 2; i++) {
25            tmp.push_back(nullptr);
26        }
27
28        Node* previous = nullptr;
29        for (Iterator iter = this->begin(); !iter.equals(this->end()); iter.next())
30        {
31            delete previous;
32            previous = iter.current;
33            h = (98 * hash_code(iter.get()) + 460) % 997;
34            h = h % tmp.size();
35            if (h < 0) { h = -h; }
36
37            Node* tcurrent = tmp[h];
38            while (tcurrent != nullptr) {
39                if (tcurrent->data == iter.get()) { return; }
```

```

39         // Already in the set
40         tcurrent = tcurrent->next;
41     }
42     Node* tnew_node = new Node;
43     tnew_node->data = iter.get();
44     tnew_node->next = tmp[h];
45     tmp[h] = tnew_node;
46 }
47
48 buckets = tmp;
49 }
50
51 if(load_factor < 0.5) {
52     vector<Node*> tmp = buckets;
53
54     int num_buckets = buckets.size();
55     for (int i = 0; i < num_buckets; i++) {
56         buckets.push_back(nullptr);
57     }
58
59     Node* previous = nullptr;
60     for (Iterator iter = this->begin(); !iter.equals(this->end()); iter.next())
61     {
62         delete previous;
63         previous = iter.current;
64         h = (98 * hash_code(iter.get()) + 460) % 997;
65         h = h % tmp.size();
66         if (h < 0) { h = -h; }
67
68         Node* tcurrent = tmp[h];
69         while (tcurrent != nullptr) {
70             if (tcurrent->data == iter.get()) { return; }
71             // Already in the set
72             tcurrent = tcurrent->next;
73         }
74         Node* tnew_node = new Node;
75         tnew_node->data = iter.get();
76         tnew_node->next = tmp[h];
77         tmp[h] = tnew_node;
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**

```

1 int h = (3 * hash_code<T>(x) + 5) % 99173;
2 h = h % buckets.size();
3 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

```
1 #ifndef HASHTABLE_H
2 #define HASHTABLE_H
3
4 #include <string>
5 #include <vector>
6
7 using namespace std;
8
9 /**
10  Computes the hash code for a string.
11  @param str a string
12  @return the hash code
13 */
14 template <class T>
15 int hash_code(const T& str);
16
17 template <class T>
18 class HashTable;
19
20 template <class T>
21 class Iterator;
22
23 template <class T>
24 class Node
25 {
26 private:
27     T data;
28     Node<T>* next;
29
30 friend class HashTable<T>;
31 friend class Iterator<T>;
32 };
33
34 template <class T>
35 class Iterator
36 {
37 public:
38     /**
39      Looks up the value at a position.
40      @return the value of the node to which the iterator points
41      */
42     T get() const;
43     /**
44      Advances the iterator to the next node.
45      */
46     void next();
47     /**
48      Compares two iterators.
49      @param other the iterator to compare with this iterator
50      @return true if this iterator and other are equal
```

```

51     */
52     bool equals(const Iterator& other) const;
53 private:
54     const HashTable<T>* container;
55     int bucket_index;
56     Node<T>* current;
57
58 friend class HashTable<T>;
59 };
60
61 /**
62     This class implements a hash table using separate chaining.
63 */
64 template <class T>
65 class HashTable
66 {
67 public:
68     /**
69         Constructs a hash table.
70         @param nbuckets the number of buckets
71     */
72     HashTable(int nbuckets);
73
74     /**
75         Tests for set membership.
76         @param x the potential element to test
77         @return 1 if x is an element of this set, 0 otherwise
78     */
79     int count(const T& x);
80
81     /**
82         Adds an element to this hash table if it is not already present.
83         @param x the element to add
84     */
85     void insert(const T& x);
86
87     /**
88         Removes an element from this hash table if it is present.
89         @param x the potential element to remove
90     */
91     void erase(const T& x);
92
93     /**
94         Returns an iterator to the beginning of this hash table.
95         @return a hash table iterator to the beginning
96     */
97     Iterator<T> begin() const;
98
99     /**
100         Returns an iterator past the end of this hash table.
101         @return a hash table iterator past the end
102     */
103     Iterator<T> end() const;
104
105     /**
106         Gets the number of elements in this set.
107         @return the number of elements
108     */
109     int size() const;

```

```

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

```

Listing 3: hashtable.h

```

1 #include<iostream>
2
3 #include "hashtable.h"
4
5 template<class T>
6 int hash_code(const T& str)
7 {
8     int h = 0;
9     for (int i = 0; i < str.length(); i++)
10    {
11        h = 31 * h + str[i];
12    }
13    return h;
14 }
15
16 template <class T>
17 HashTable<T>::HashTable(int nbuckets)
18 {
19     for (int i = 0; i < nbuckets; i++)
20     {
21         buckets.push_back(nullptr);
22     }
23     current_size = 0;
24     collisions = 0;
25 }
26
27 template <class T>
28 int HashTable<T>::count(const T& x)
29 {
30     int h = (3 * hash_code<T>(x) + 5) % 99173;
31     h = h % buckets.size();
32     if (h < 0) { h = -h; }
33
34     Node<T>* current = buckets[h];
35     while (current != nullptr)
36     {
37         if (current->data == x) { return 1; }
38         current = current->next;
39     }
40     return 0;

```

```

41 }
42
43 template <class T>
44 void HashTable<T>::insert(const T& x)
45 {
46     int h = (3 * hash_code<T>(x) + 5) % 99173;
47     h = h % buckets.size();
48     if (h < 0) { h = -h; }
49
50     Node<T>* current = buckets[h];
51     if (current != nullptr) collisions++;
52     while (current != nullptr)
53     {
54         if (current->data == x) { return; }
55         // Already in the set
56         current = current->next;
57     }
58     Node<T>* new_node = new Node<T>;
59     new_node->data = x;
60     new_node->next = buckets[h];
61     buckets[h] = new_node;
62     current_size++;
63 }
64
65 template <class T>
66 void HashTable<T>::erase(const T& x)
67 {
68     int h = (3 * hash_code<T>(x) + 5) % 99173;
69     h = h % buckets.size();
70     if (h < 0) { h = -h; }
71
72     Node<T>* current = buckets[h];
73     Node<T>* previous = nullptr;
74     while (current != nullptr)
75     {
76         if (current->data == x)
77         {
78             if (previous == nullptr)
79             {
80                 buckets[h] = current->next;
81             }
82             else
83             {
84                 previous->next = current->next;
85             }
86             delete current;
87             current_size--;
88             return;
89         }
90         previous = current;
91         current = current->next;
92     }
93 }
94
95 template <class T>
96 int HashTable<T>::size() const
97 {
98     return current_size;
99 }

```

```

100
101 template <class T>
102 HashTable<T>::~~HashTable() {
103     Node<T>* previous = nullptr;
104     for (Iterator<T> iter = this->begin(); !iter.equals(this->end()); iter.
        next()) {
105         delete previous;
106         previous = iter.current;
107     }
108 }
109
110 template <class T>
111 HashTable<T>& HashTable<T>::operator=(HashTable ht) {
112     buckets = ht.buckets;
113     current_size = ht.current_size;
114     collisions = ht.collisions;
115     return *this;
116 }
117
118 template <class T>
119 HashTable<T>::HashTable(HashTable &ht) {
120     buckets = ht->buckets;
121     current_size = ht->current_size;
122     collisions = ht->current_size;
123 }
124
125 template <class T>
126 Iterator<T> HashTable<T>::begin() const
127 {
128     Iterator<T> iter;
129     iter.current = nullptr;
130     iter.bucket_index = -1;
131     iter.container = this;
132     iter.next();
133     return iter;
134 }
135
136 template <class T>
137 Iterator<T> HashTable<T>::end() const
138 {
139     Iterator<T> iter;
140     iter.current = nullptr;
141     iter.bucket_index = buckets.size();
142     iter.container = this;
143     return iter;
144 }
145
146 template <class T>
147 int HashTable<T>::getCollisions() {
148     return collisions;
149 }
150
151 template <class T>
152 T Iterator<T>::get() const
153 {
154     return current->data;
155 }
156
157 template <class T>

```

```

158 bool Iterator<T>::equals(const Iterator& other) const
159 {
160     return current == other.current;
161 }
162
163 template <class T>
164 void Iterator<T>::next()
165 {
166     if (bucket_index >= 0 && current->next != nullptr)
167     {
168         // Advance in the same bucket
169         current = current->next;
170     }
171     else
172     {
173         // Move to the next bucket
174         do
175         {
176             bucket_index++;
177         }
178         while (bucket_index < container->buckets.size()
179             && container->buckets[bucket_index] == nullptr);
180         if (bucket_index < container->buckets.size())
181         {
182             // Start of next bucket
183             current = container->buckets[bucket_index];
184         }
185         else
186         {
187             // No more buckets
188             current = nullptr;
189         }
190     }
191 }

```

Listing 4: hashtable.cpp