Requirements:
1) In the “designs.jpg” file I designed the code in the “code_2” folder.

2) In the “code_2” folder there is a file called “main.cpp”. I have some tests for each function in “hash.h”.

```
6   int main() {
7       HashTable *ht = new HashTable();
8       string dog_hash = ht->hash("dog");
9       string hi_hash = ht->hash("hi");
10      string bye_hash = ht->hash("bye");
11      string zap_hash = ht->hash("zap");
12      string at_hash = ht->hash("at");
13      string it_hash = ht->hash("it");
14
15      cout << "------Inserting root node with value 41507------" << endl;
16      ht->insert(dog_hash);
17      cout << "------Displaying table in order------" << endl;
18      ht->in_order(ht->get_roots_address());
19
20      cout << "\n------Inserting nodes values: 809, 22505, 260116, 120------" << endl;
21      ht->insert(hi_hash);
22      ht->insert(bye_hash);
23      ht->insert(zap_hash);
24      ht->insert(at_hash);
25      cout << "------Displaying table in order------" << endl;
26      ht->in_order(ht->get_roots_address());
27
28      cout << "\n------Checking if the table contains 'zap'------" << endl;
29      bool result = ht->contains(zap_hash);
30      if (result == true) {
31          cout << "'zap' is in the hash table." << endl;
32      } else {
33          cout << "'zap' was not found." << endl;
34      }
```

3.1) To hash the string values I associate a number with each character. (a=01, z=26, j=10, b=02). I then concatenate those integer values. “hi” turns into 0809.

3.2) The insert function inserts the hash values into a binary search tree. This function can be found in the “code_2” folder in the file “hash.h”.

```
40          // non recursive add function. Tree is sorted in ascending numerical order from left to right.
41          void insert(string str_value) {
42              bool insert = false; // will set this to true when the value has been inserted
43              int new_value = stoi(str_value); // converts string to integer
44
45              bool result;
46              result = contains(str_value);
47              if (result == true) { // if there is a collision... make a linked list at the collision
48
49                  node *head;
50                  head = search(root, new_value);
51                  head->index = 1;
52                  node *tail = new node;
53                  tail->value = new_value;
54                  tail->next = nullptr;
55                  int count = 0;
56                  node *temp_ptr = head;
57                  while (temp_ptr->next != nullptr){ // will iteratre through linked list until the previous tail is found
58                      count += 1;
59                      temp_ptr = temp_ptr -> next;
60                  }
61                  temp_ptr->next = tail;
62                  tail->index = count;
63
64                  insert = true;
65              } else { // if there is no collision
66                  node *temp = root; // pointer to node that will be iterated through the tree
67                  if (root == nullptr){ // if there is no root node make the value the root
68                      root = new node;
69                      root->value = new_value;
70                      root->left = nullptr;
71                      root->right = nullptr;
72                      root->next = nullptr;
73                      root->index = 1;
74                      insert = true;
```

3.3) The “contains” function can be found in the “code_2” folder in the file “hash.h”.

```
104         // determines whether or not a hash value already exists in the binary search tree
105         bool contains(string hash_value) {
106             int int_hash_value = stoi(hash_value); // converts string to integer
107             node *collision_node = search(root, int_hash_value);
108             if (collision_node != nullptr) {
109                 return true;
110             }
111             return false;
112         }
```

4) My hash table deals with collisions by chaining. It will create a linked list at the colliding node. This can be seen in the insert function.

5) I compare collision frequencies and complexity in CollisionEffectOnComplexity.pdf.