Jeff Caldwell

CS 3311

Assignment 02

November 20, 2021

Implementing a Hash Table in C++

Basic Implementation

To implement a basic hash table, a `HashTable` class was created using C++. This class contained two private data members. The first, `buckets`, a static constant integer variable initialized to `10`, would determine the number of buckets the hashtable would allow. The second member was a C++ list of type `pair<int, string>`, initialized to hold all data passed into the hash table.

Public methods were included to implement the basic functionality of a hash table such as `insertItem()`, `findItem()`, `removeItem()`, and `isEmpty()`. Finally, a simple mod-10-based hashing function was included to place items into buckets that coincided with the given item's `id`.

Testing

To test the `HashTable` class, a set of twelve values was passed to one of the class's overloaded `insertItem()` methods. Four of those items contained `id` properties which would collide.

```
    h.insertItem(110, "XYZ");
    h.insertItem(121, "ABC");
    h.insertItem(221, "DEF");
    h.insertItem(222, "HJK");
    h.insertItem(333, "FUI");
    h.insertItem(444, "RYU");
    h.insertItem(555, "ODU");
    h.insertItem(556, "GHI");
    h.insertItem(666, "JKL");
    h.insertItem(777, "ERW");
    h.insertItem(888, "UTI");
    h.insertItem(999, "FDH");
```

The simplicity of the hashing function and the hardcoded dataset used to test the `HashTable` class guaranteed two collisions. Those collisions were then represented in the terminal output as being linked with `-->` characters.

Final output of `HashTable` testing, which includes validation that the table is empty at the beginning of execution, the addition of all twelve test values, printed output representation of the table with collisions, the removal of colliding values, and a final validation that the table is not empty at the end of execution, is included below.

```
Table empty. Beginning insertion.

Full Table:
[0] --> { ID: 110, tag: XYZ }
[1] --> { ID: 121, tag: ABC } --> { ID: 221, tag: DEF }
[2] --> { ID: 222, tag: HJK }
[3] --> { ID: 333, tag: FUI }
[4] --> { ID: 444, tag: RYU }
[5] --> { ID: 555, tag: ODU }
[6] --> { ID: 556, tag: GHI } --> { ID: 666, tag: JKL }
[7] --> { ID: 777, tag: ERW }
[8] --> { ID: 888, tag: UTI }
[9] --> { ID: 999, tag: FDH }

Items with colliding IDs:
{121, ABC}
{221, DEF}
{556, GHI}
{666, JKL}

Removing items with colliding IDs 121 & 556

Items after removal of one colliding item:
{666, JKL}
{221, DEF}

Table after colliding item removal:
[0] --> { ID: 110, tag: XYZ }
[1] --> { ID: 221, tag: DEF }
[2] --> { ID: 222, tag: HJK }
[3] --> { ID: 333, tag: FUI }
[4] --> { ID: 444, tag: RYU }
[5] --> { ID: 555, tag: ODU }
[6] --> { ID: 666, tag: JKL }
[7] --> { ID: 777, tag: ERW }
[8] --> { ID: 888, tag: UTI }
[9] --> { ID: 999, tag: FDH }

Table not empty. Operations successful.
```

Appendix

HashTable class

```cpp
class HashTable {
  private:
    static const int buckets = 10;
    list<pair<int, string>> table[buckets];

  public:
    bool isEmpty() const;
    int hashFunction(int bucketSize);
    void insertItem(int id, string tag);
    void insertItem(pair<int, string> val);
    void removeItem(int id);
    pair<int, string> findItem(int id);

    void printItem(pair<int, string> it);
    void printTable();
};
```

hashFunction()

```cpp
/**
 * @brief Simple mod hash function that returns the
 *        last digit of a given integer
 *
 * @param id
 * @return int
 */
int HashTable::hashFunction(int id) {
  return (id % buckets);
}
```

insertItem()

```cpp
/**
 * @brief Creates and inserts a pair into the table into
 *        the "row" that coincides with the hash
 *        of the item's ID
 *
 * @param id
 * @param tg
 */
void HashTable::insertItem(int id, string tg) {
  int index = hashFunction(id);
  pair<int, string> item;
  item.first = id;
```

```
    item.second = tg;
    table[index].push_back(item);
  }
```

overloaded insertItem()

```
  /**
   * @brief Inserts a given pair into the hashtable
   *
   * @param val
   */
  void HashTable::insertItem(pair<int, string> val) {
    insertItem(val.first, val.second);
  }
```

removeItem()

```
  /**
   * @brief Removes item whose ID matches the given ID
   *
   * @param id
   */
  void HashTable::removeItem(int id) {
    int index = hashFunction(id);
    list<pair<int, string> >::iterator i;

    for(i = table[index].begin(); i != table[index].end(); i++) {
      if(i->first == id) {
        break;
      }
    }

    if(i != table[index].end()) {
      table[index].erase(i);
    }
  }
```

findItem()

```
  /**
   * @brief Returns a pair whose ID matches the given ID
   *
   * @param id
```

```
 * @return pair<int, string>
 */
pair<int, string> HashTable::findItem(int id) {
  int index = hashFunction(id);
  list<pair<int, string> >::iterator i;
  pair<int, string> myRec;

  for(i = table[index].begin(); i != table[index].end(); i++) {
    if(i->first == id) {
      myRec.first = i->first;
      myRec.second = i->second;
    }
  }
    return myRec;
}
```

Main function (driver code)

```
// Driver/test code
int main(int argc, const char** argv) {
  HashTable h;

  if(h.isEmpty()) {
    cout << "\nTable empty. Beginning insertion." << endl;
  } else {
    cout << "\nTable not empty. There's a problem!." << endl;
  }

  h.insertItem(110, "XYZ");
  h.insertItem(121, "ABC");
  h.insertItem(221, "DEF");
  h.insertItem(222, "HJK");
  h.insertItem(333, "FUI");
  h.insertItem(444, "RYU");
  h.insertItem(555, "ODU");
  h.insertItem(556, "GHI");
  h.insertItem(666, "JKL");
  h.insertItem(777, "ERW");
  h.insertItem(888, "UTI");
  h.insertItem(999, "FDH");

  cout << "\nFull Table: " << endl;
  h.printTable();

  cout << "\nItems with colliding IDs: " << endl;

  h.printItem(h.findItem(121));
  h.printItem(h.findItem(221));
```

```
    h.printItem(h.findItem(556));
    h.printItem(h.findItem(666));

    cout << "\nRemoving items with colliding IDs 121 & 556" << endl;

    h.removeItem(121);
    h.removeItem(556);

    cout << "\nItems after removal of one colliding item: " << endl;
    h.printItem(h.findItem(666));
    h.printItem(h.findItem(221));

    cout << "\nTable after colliding item removal: " << endl;
    h.printTable();

    if(!h.isEmpty()) {
      cout << "\nTable not empty. Operations successful." << endl;
    } else {
      cout << "\nTable empty. There's a problem!" << endl;
    }

    return 0;
  }
```