# BLG335E, Analysis of Algorithms I, Fall 2016 Project Report 4

**Name: Buğra Ekuklu**

**Student ID: 150120016**

---

## Part A. Questions on Hash Tables (20 points)

**1)** Why do we use Hash Tables as a data structure in our problems? Please explain briefly. **(5 points)**

**In traditional linear containers, elements of the data structure are exposed via raw indexes. For instance, in vector data structure, we would access an element with an index, which is generally an unsigned integer. The hash table data structure implements a new abstraction layer on top of integer-indexed linear containers, thus it makes available to access the elements by the keys. The naïve implementation provided by this assignment is a superficial example of hash tables. With an implementation of nodes, the hash table would allow access to the value with a key. Remember that, the value and key could be simple data types (i.e. integers), or complex data structures. A typical usage of hash tables would be dictionaries, where string keys access values, which are objects.**

**2)** Consider a hash table consisting of M = 11 slots, and suppose nonnegative integer key values are hashed into the table using the hash function `h1()` :

```
int h1 (int key) {
int x = (key + 7) * (key + 7);
x = x / 16;
x = x + key;
x = x % 11;
return x;
}
```

Suppose that collisions are resolved by using linear probing. The integer key values listed below are to be inserted, in the order given. Show the home slot (the slot to which the key hashes, before any probing), the probe sequence (if any) for each key, and the final contents of the hash table after the following key values have been inserted in the given order: **(10+5 points)**

| Key Value | Home Slot | Probe Sequence |
|---|---|---|
| 43 | 1 | h(43, 0) = 1 |
| 23 | 2 | h(23, 0) = 2 |
| 1 | 5 | h(1, 0) = 5 |
| 0 | 3 | h(0, 0) = 3 |
| 15 | 1 | h(15, 3) = 4 |
| 31 | 0 | h(31, 0) = 0 |
| 4 | 1 | h(4, 5) = 6 |
| 7 | 8 | h(7, 0) = 8 |
| 11 | 9 | h(11, 0) = 9 |
| 3 | 9 | h(3, 1) = 10 |

Final Hash Table:

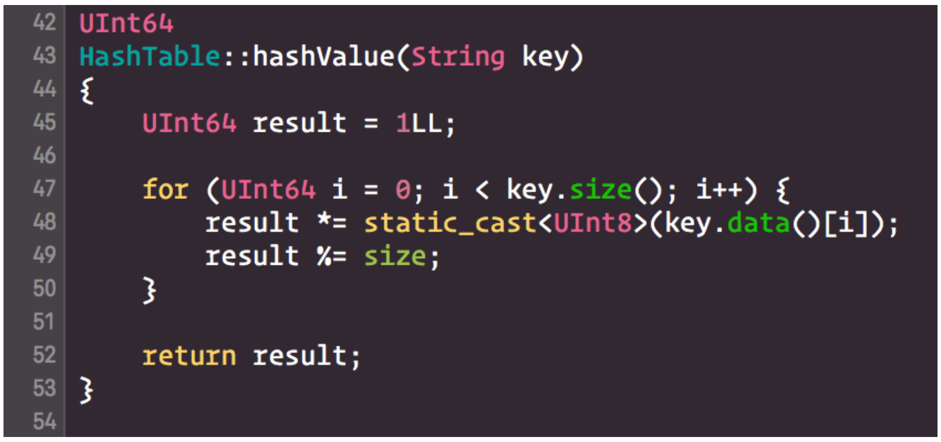
| Slot | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Contents | 31 | 43 | 23 | 0 | 15 | 1 | 4 | | 7 | 11 | 3 |

## Part B. Implementation and Report (80 points)

To compile the program, copy and paste the following command to the shell and run:

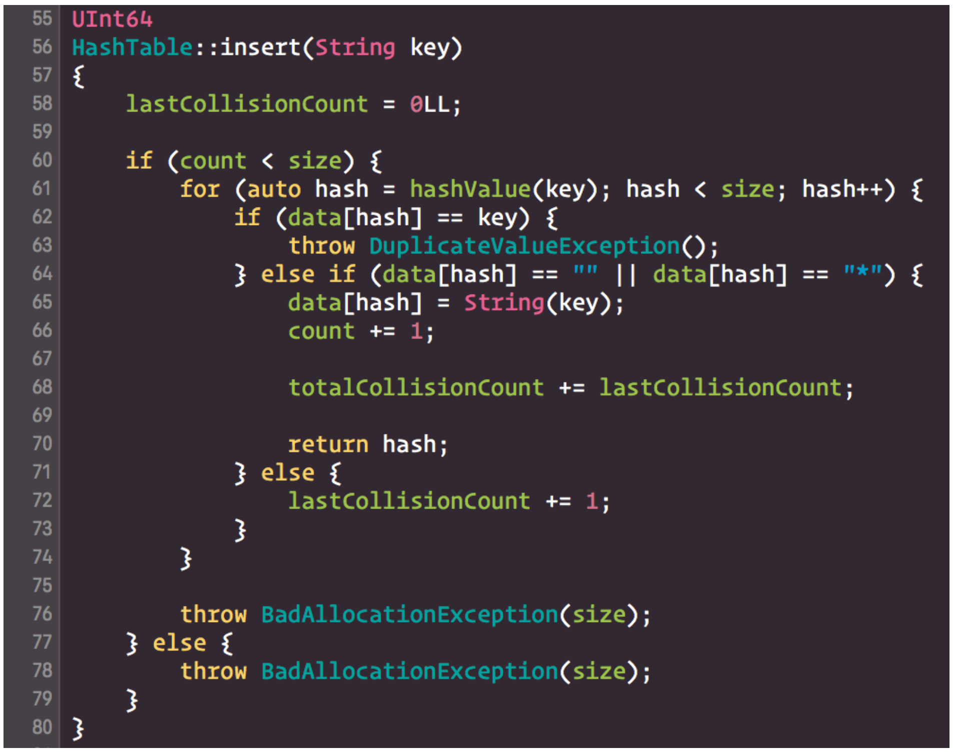
```
g++ AoAHW4ApplicationDelegate.cpp \
Bootstrapping/Application.cpp \
Containers/Array.cpp \
Containers/HashTable.cpp \
Controllers/SpellCheckController.cpp \
Controllers/HashTableOperationController.cpp \
ErrorHandling/RangeException.cpp \
ErrorHandling/BadAllocationException.cpp \
ErrorHandling/BufferOverrunException.cpp \
ErrorHandling/DuplicateValueException.cpp \
Protocols/Hashable.cpp \
Models/SpellCheck.cpp \
Supporting\ Files/main.cpp -std=c++11 -O3 -o main
```

The result is multiplied by the ASCII value of the ith character and afterwards the remainder is calculated. Remainder calculation per loop run prevents overflow.

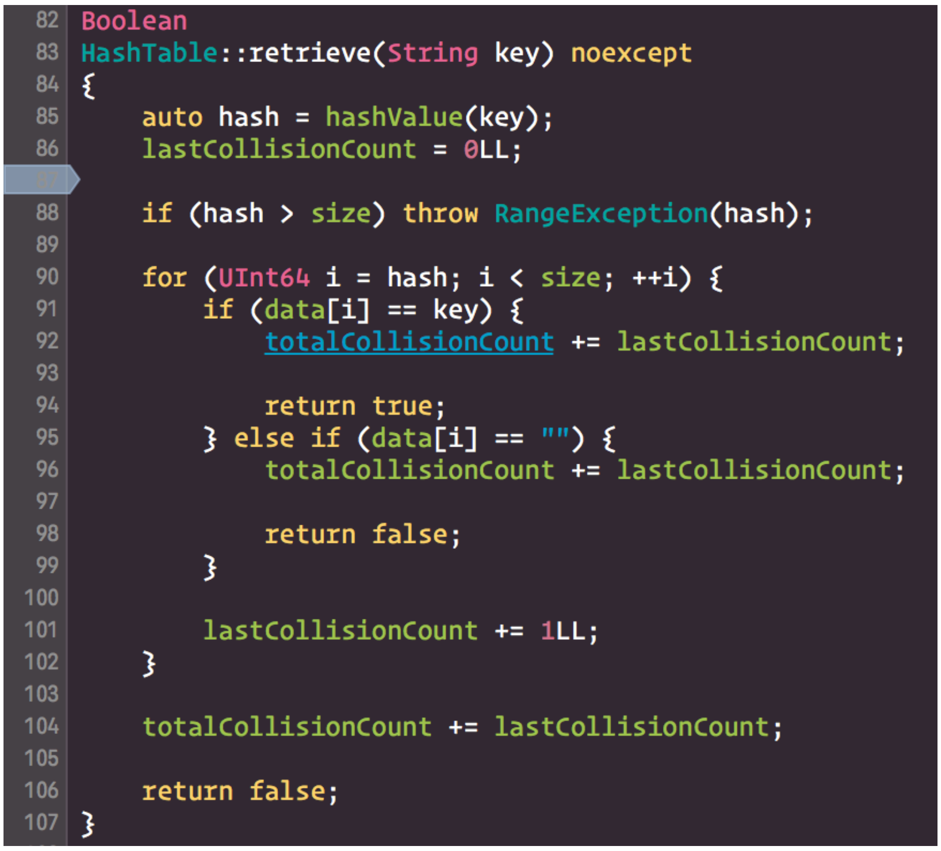


```
UInt64
HashTable::hashValue(String key)
{
    UInt64 result = 1LL;

    for (UInt64 i = 0; i < key.size(); i++) {
        result *= static_cast<UInt8>(key.data()[i]);
        result %= size;
    }

    return result;
}
```

In case of insertion, if the number of elements in the table surpasses the size of the table, it throws an allocation exception. After fetching the hash value of the key, it starts to doing linear search in the internal array. If there is a buffer overrun, it will throw an allocation exception.

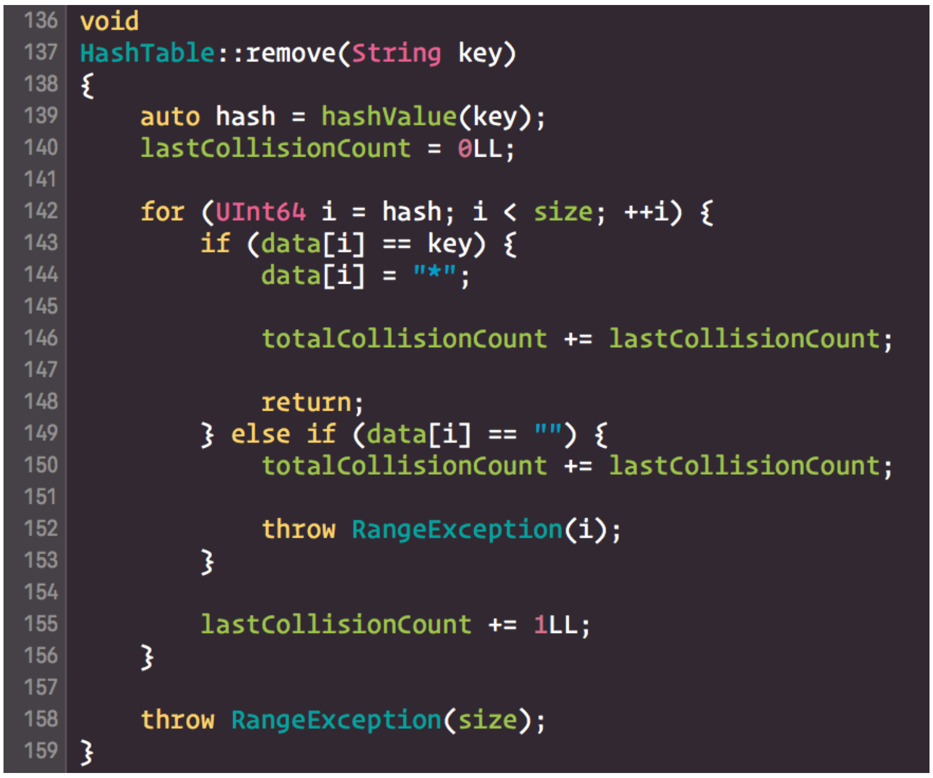

```cpp
UInt64
HashTable::insert(String key)
{
    lastCollisionCount = 0LL;

    if (count < size) {
        for (auto hash = hashValue(key); hash < size; hash++) {
            if (data[hash] == key) {
                throw DuplicateValueException();
            } else if (data[hash] == "" || data[hash] == "*") {
                data[hash] = String(key);
                count += 1;

                totalCollisionCount += lastCollisionCount;

                return hash;
            } else {
                lastCollisionCount += 1;
            }
        }

        throw BadAllocationException(size);
    } else {
        throw BadAllocationException(size);
    }
}
```

To retrieve a key in the hash table, it looks up for the hash value, and starts to do linear search in the internal array. If there is a corresponding value, it will return true. In case of buffer overrun, it will return false, indicating value does not exist in the hash table.



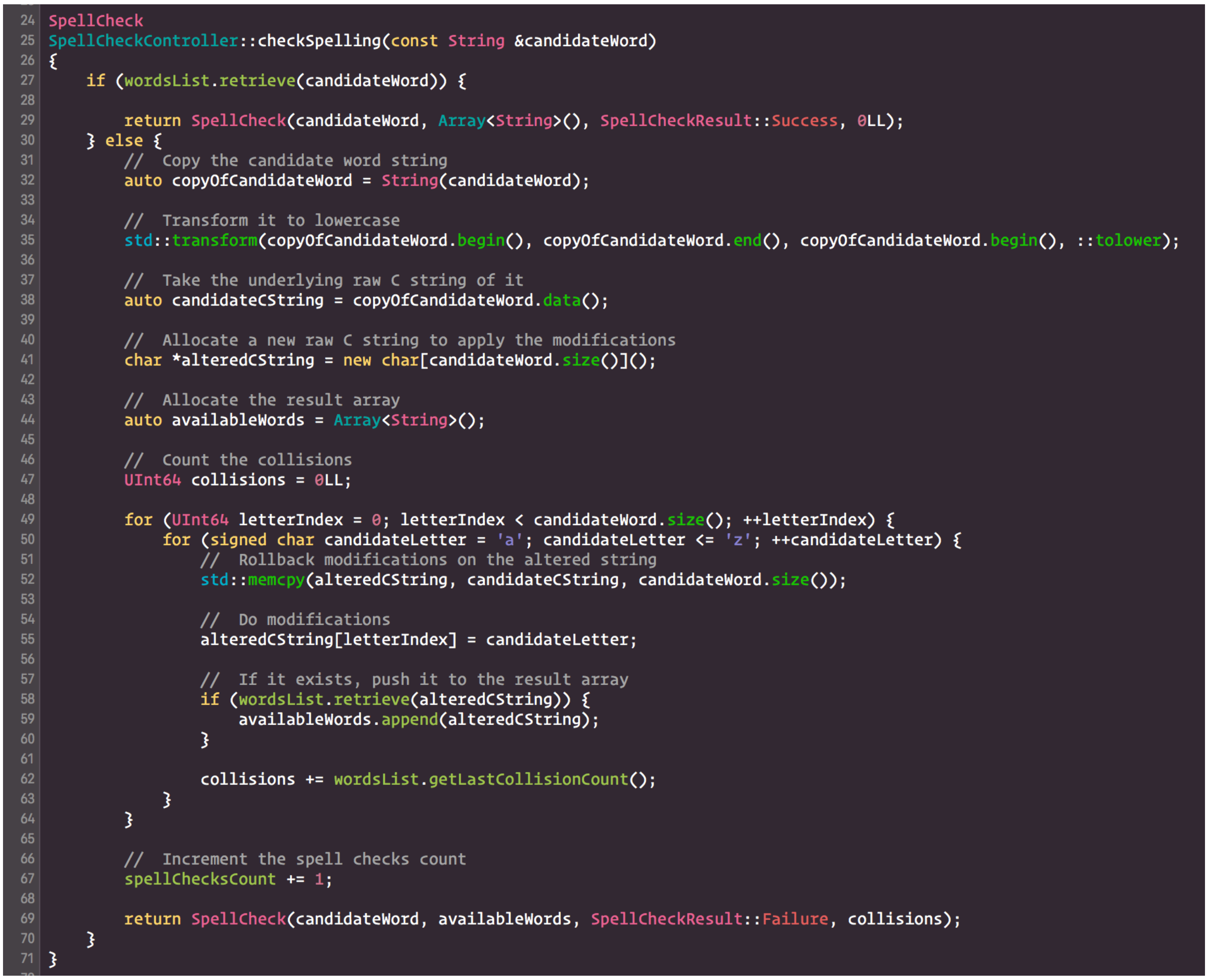
```
Boolean
HashTable::retrieve(String key) noexcept
{
    auto hash = hashValue(key);
    lastCollisionCount = 0LL;

    if (hash > size) throw RangeException(hash);

    for (UInt64 i = hash; i < size; ++i) {
        if (data[i] == key) {
            totalCollisionCount += lastCollisionCount;

            return true;
        } else if (data[i] == "") {
            totalCollisionCount += lastCollisionCount;

            return false;
        }

        lastCollisionCount += 1LL;
    }

    totalCollisionCount += lastCollisionCount;

    return false;
}
```

To remove a value from the hash table, we look up for the hash value of the key first, and after doing linear search, we mark it with an asterisk.



```
void
HashTable::remove(String key)
{
    auto hash = hashValue(key);
    lastCollisionCount = 0LL;

    for (UInt64 i = hash; i < size; ++i) {
        if (data[i] == key) {
            data[i] = "*";

            totalCollisionCount += lastCollisionCount;

            return;
        } else if (data[i] == "") {
            totalCollisionCount += lastCollisionCount;

            throw RangeException(i);
        }

        lastCollisionCount += 1LL;
    }

    throw RangeException(size);
}
```

If the retrieve function does not return a truthy value, then we start to search for similar words found in the same hash table. The spell checker controller checks for words with one wrong character.



```cpp
SpellCheck
SpellCheckController::checkSpelling(const String &candidateWord)
{
    if (wordsList.retrieve(candidateWord)) {

        return SpellCheck(candidateWord, Array<String>(), SpellCheckResult::Success, 0LL);
    } else {
        //  Copy the candidate word string
        auto copyOfCandidateWord = String(candidateWord);

        //  Transform it to lowercase
        std::transform(copyOfCandidateWord.begin(), copyOfCandidateWord.end(), copyOfCandidateWord.begin(), ::tolower);

        //  Take the underlying raw C string of it
        auto candidateCString = copyOfCandidateWord.data();

        //  Allocate a new raw C string to apply the modifications
        char *alteredCString = new char[candidateWord.size()]();

        //  Allocate the result array
        auto availableWords = Array<String>();

        //  Count the collisions
        UInt64 collisions = 0LL;

        for (UInt64 letterIndex = 0; letterIndex < candidateWord.size(); ++letterIndex) {
            for (signed char candidateLetter = 'a'; candidateLetter <= 'z'; ++candidateLetter) {
                //  Rollback modifications on the altered string
                std::memcpy(alteredCString, candidateCString, candidateWord.size());

                //  Do modifications
                alteredCString[letterIndex] = candidateLetter;

                //  If it exists, push it to the result array
                if (wordsList.retrieve(alteredCString)) {
                    availableWords.append(alteredCString);
                }

                collisions += wordsList.getLastCollisionCount();
            }
        }

        //  Increment the spell checks count
        spellChecksCount += 1;

        return SpellCheck(candidateWord, availableWords, SpellCheckResult::Failure, collisions);
    }
}
```