Homework #3: Hash Tables

Name: 여승엽

Student ID: 20215414

1. Introduction

1.1. What is a hash table?

▶ A hash table is a data structure that allows mapping keys to values. It uses a hash function to convert a key into a hash value, which then serves as an index in an array to store and retrieve data. This enables fast access to data.

1.2. What is a hash function?

▶ A hash function is a function that maps data of arbitrary length to a fixed-length hash value. In a hash table, it's used to convert a key into a valid array index. A good hash function should distribute hash values evenly to minimize collisions between different keys.

1.3. Why use a hash table?

▶ Hash tables are used for the following reasons:

1. Fast Data Access: On average, data insertion, deletion, and retrieval can be performed with O(1) time complexity, making them highly efficient.
2. Efficient Memory Usage: They use an array to store data, and access data via hash values rather than directly storing key-value pairs, which allows for efficient memory management.
3. Diverse Applications: They are utilized in various fields such as database indexing, cache implementation, symbol tables, and dictionaries.

2. Implementation (Explain for core functions)

▶ **Key Members of the HashMap Class**

n: Represents the current number of entries (key-value pairs) stored in the hash table.

hash: The hash function object used to convert keys into hash values.

B: The bucket array (BktArray) that stores the actual data of the hash table. Each bucket is composed of a std::list that stores Entry objects.

▶Explanation of Core Functions

1. **put(const K& k, const V& v)**:

-> **Functionality**: Inserts a new key-value pair (k, v) into the hash table, or updates the value for key k to v if k already exists.

-> **Operation**: First, it uses the finder function to check if k already exists.
If k does not exist, it calls the inserter function to insert a new Entry(k, v) into the corresponding bucket and increments n.
If k exists, it updates the value of the existing Entry to v.

2. **find(const K& k)**:
-> **Functionality**: Returns an iterator (Iterator) to the Entry corresponding to key k in the hash table.
-> **Operation**: Calls the finder function to locate the entry for k.
If k is not found in the hash table, it returns the end() iterator.
If k exists, it returns an iterator pointing to that Entry.

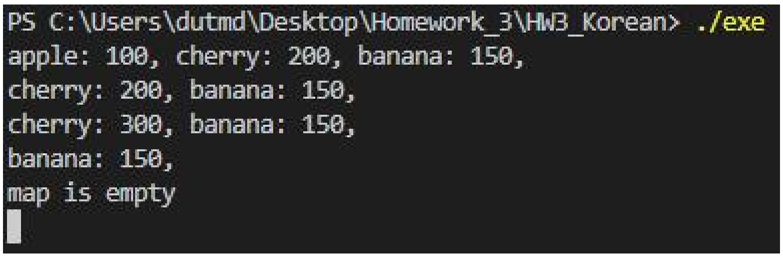
3. **erase(const K& k) or erase(const Iterator& p)**:
-> **Functionality**: Removes the entry corresponding to a specific key k or the entry pointed to by a given iterator p from the hash table.
-> **Operation**:
erase(const K& k): Uses the finder function to locate the entry for k, then calls the eraser function to remove that entry from the bucket and decrements n.
erase(const Iterator& p): Directly calls the eraser function to remove the entry pointed to by p and decrements n.

4. **begin()**:
-> **Functionality**: Returns an iterator pointing to the first entry in the hash table.
-> **Operation**: If the hash table is empty, it returns the end() iterator. Otherwise, it finds the first non-empty bucket and creates an iterator pointing to the first entry in that bucket, then returns it.

5. **end()**:
-> **Functionality**: Returns an iterator pointing to the "past-the-end" position of the hash table (it does not point to an actual entry).
-> **Operation**: Creates and returns an iterator pointing to the end of B (the bucket array).

3. Result (Results of the provided main.cpp)

3.1. Figures



```
PS C:\Users\dutmd\Desktop\Homework_3\HW3_Korean> ./exe
apple: 100, cherry: 200, banana: 150,
cherry: 200, banana: 150,
cherry: 300, banana: 150,
banana: 150,
map is empty
```

3.2. Explanations

▶ **First Output: apple: 100, cherry: 200, banana: 150,**

-> This output is the result of iterating through all entries in the map from map.begin() to map.end() and printing them.

-> Due to the nature of hash tables, the order of output may differ from the insertion order, as it depends on the hash function and the internal bucket array structure.

-> All three entries ("apple", "cherry", "banana") are present in the map with their respective values of 100, 200, and 150.

-> map.erase("apple");
This line deletes the entry corresponding to the key "apple" from the map.
Now, the map contains only two entries: "cherry": 200 and "banana": 150.

▶ **Second Output: cherry: 200, banana: 150,**

-> After "apple" is erased, this is the result of iterating and printing the remaining entries from map.begin() to map.end() again.

-> Only "cherry" and "banana" are correctly printed, while "apple" is no longer present.

-> map.put("cherry", 300);
Since the key "cherry" already exists in the map, the put function updates its value from 200 to 300. The map now contains two entries: "cherry": 300 and "banana": 150.

▶ **Third Output: cherry: 300, banana: 150,**

-> After the value of "cherry" is updated, this output shows the result of iterating and printing the entries from map.begin() to map.end() once more.

-> iter = map.find("cherry"); map.erase(iter);
First, map.find("cherry") finds an iterator to the entry corresponding to the key "cherry". Then, map.erase(iter) is used to delete the entry pointed to by that iterator ("cherry": 300) from the map.
At this point, the map contains only one entry: "banana": 150.

▶ **Fourth Output: banana: 150,**
-> After "cherry" is deleted, this output shows the remaining entries when iterating from map.begin() to map.end(). Only "banana" is correctly printed.

-> map.erase("banana");
This line deletes the entry corresponding to the key "banana" from the map.
Now, no entries remain in the map, making it empty.

▶ **Final Output: map is empty**
-> The if (map.empty()) condition evaluates to true, resulting in "map is empty" being printed.
-> This confirms that all entries have been successfully removed from the map

Appendix A. hashMap.h

```
#pragma once // Ensures this header file is included only once.

#include <iostream> // Includes iostream for standard input/output operations.
#include <list>       // Includes std::list, which will be used for buckets in the hash table.
#include <vector>    // Includes std::vector, which will be used for the array of buckets.

// Defines a hash function structure for strings.
struct stringHash {
    // Overloads the function call operator to allow the struct to be used like a function.
    std::size_t operator()(const std::string& key) const {
        std::size_t hash = 0; // Initializes a variable to store the hash value.
        for (char c : key) { // Iterates through each character in the input key.
            hash = (hash * 31) + c; // Calculates the hash value using a simple algorithm.
        }
        return hash; // Returns the calculated hash value.
    }
};

// Template class Entry to store a key-value pair.
template <typename K, typename V>
class Entry {
public:
    // Constructor: Initializes an Entry object with a key (k) and a value (v).
    Entry(const K& k = K(), const V& v = V())
        : _key(k), _value(v) { } // Uses a member initializer list to initialize _key and _value.

    // Constant member function to return the key.
    const K& key() const { return _key; }

    // Constant member function to return the value.
    const V& value() const { return _value; }

    // Member function to set the key.
    void setKey(const K& k) { _key = k; }

    // Member function to set the value.
    void setValue(const V& v) { _value = v; }

private:
    K _key;    // Member variable to store the key of the Entry.
    V _value; // Member variable to store the value of the Entry.
};

// Template class HashMap. Takes key type (K), value type (V), and hash function type (H).
template <typename K, typename V, typename H>
class HashMap {
public:
    typedef Entry<const K, V> Entry; // Defines Entry type to represent a (key,value) pair.
    // Declaration of the inner Iterator class for HashMap.
    class Iterator;

public:
    // Constructor: Sets the initial capacity of the hash map's buckets. Default is 100.
    HashMap(int capacity = 100);

    // Returns the number of entries (key-value pairs) currently stored in the hash map.
    int size() const;

    // Returns true if the map is empty.
    bool empty() const;

    // Finds an entry with key k and returns an iterator to it.
    Iterator find(const K& k);

    // Inserts or replaces a (k,v) pair in the map.
    Iterator put(const K& k, const V& v);

    // Removes an entry with key k.
    void erase(const K& k);

    // Erases the entry at position p.
    void erase(const Iterator& p);

    // Returns an iterator to the first entry in the map.
    Iterator begin();

    // Returns an iterator to the end entry (past-the-end) of the map.
    Iterator end();

protected:
    typedef std::list<Entry> Bucket;       // Defines 'Bucket' as a list to hold entries.
    typedef std::vector<Bucket> BktArray; // Defines 'BktArray' as a vector to hold buckets.
    // Utility functions for HashMap operations.
    Iterator finder(const K& k);                     // Internal utility function to find a given key (k).
    Iterator inserter(const Iterator& p, const Entry& e); // Internal utility function to insert an entry (e) before position (p).
    void eraser(const Iterator& p);                  // Internal utility function to remove the entry pointed to by iterator (p).

    typedef typename BktArray::iterator BItor; // Defines BItor as an iterator type for the bucket array.
    typedef typename Bucket::iterator EItor;   // Defines EItor as an iterator type for a bucket (list).

    // Static utility function to advance an iterator (p) to the next entry within a bucket.
    static void nextEntry(Iterator& p) { ++p.ent; }
    // Static utility function to check if an iterator (p) has reached the end of its current bucket.
    static bool endOfBkt(const Iterator& p) { return p.ent == p.bkt->end(); }

private:
    int n;          // The number of entries currently stored in the hash map.
    H hash;          // The hash function object used to compare keys.
    BktArray B;     // The bucket array where the actual data is stored.

public:
    // Definition of the HashMap::Iterator class.
    class Iterator {
    private:
        EItor ent;              // The position of the entry within its bucket.
        BItor bkt;              // The position of the bucket within the bucket array.
        const BktArray* ba;     // A pointer to the bucket array this iterator belongs to.

    public:
        // Iterator constructor: Initializes the iterator with references to the bucket array (a),
        // bucket position (b), and entry position (q).
        Iterator(const BktArray& a, const BItor& b, const EItor& q = EItor())
```

```
                : ent(q), bkt(b), ba(&a) { }


            // Overloads the dereference operator to return a reference to the Entry pointed to by the iterator.
            Entry& operator*() const;
            // Overloads the equality operator to compare if two Iterator objects point to the same location.
            bool operator==(const Iterator& p) const;
            // Overloads the pre-increment operator to advance the iterator to the next entry.
            Iterator& operator++();


            friend class HashMap; // Grants the HashMap class access to the private members of Iterator.
        };
};
```

Appendix B. hashMap.cpp

```
#include "hashMap.h" // Includes the definition of the HashMap class.

#include <iostream> // Includes iostream for standard input/output operations.
#include <list>       // Includes the std::list container.
#include <vector>    // Includes the std::vector container.

// HashMap constructor implementation:
// Initializes the bucket array B with the given capacity and sets the number of entries n to 0.
template <typename K, typename V, typename H>
HashMap<K, V, H>::HashMap(int capacity) : n(0), B(capacity) {}

// size() function implementation:
// Returns the current number of entries (n) in the HashMap.
template <typename K, typename V, typename H>
int HashMap<K, V, H>::size() const { return n; }

// empty() function implementation:
// Checks if the HashMap is empty (i.e., if the number of entries is 0).
template <typename K, typename V, typename H>
bool HashMap<K, V, H>::empty() const { return size() == 0; }
```

```
// Iterator::operator*() implementation:
// Returns a reference to the Entry object pointed to by the iterator.
template <typename K, typename V, typename H>
typename HashMap<K, V, H>::Entry&
HashMap<K, V, H>::Iterator::operator*() const {
    return *ent; // Dereferences the internal entry iterator (ent) and returns the value.
}

// Iterator::operator==() implementation:
// Compares if two Iterator objects point to the same location.
template <typename K, typename V, typename H>
bool HashMap<K, V, H>::Iterator::operator==(const Iterator& p) const {
    // If the bucket array pointer (ba) or the bucket iterator (bkt) are different, the iterators are not equal.
    if (ba != p.ba || bkt != p.bkt) return false;
    // If both iterators are at the end of the bucket array, they are considered equal.
    else if (bkt == ba->end()) return true;
    // Otherwise, equality is determined by comparing the internal entry iterators (ent).
    else return (ent == p.ent);
}

// Iterator::operator++() implementation:
// Advances the iterator to the next entry.
template <typename K, typename V, typename H>
typename HashMap<K, V, H>::Iterator& HashMap<K, V, H>::Iterator::operator++() {
    ++ent; // Move to the next entry within the current bucket.
    if (endOfBkt(*this)) { // Check if the end of the current bucket has been reached.
        ++bkt; // Move to the next bucket in the bucket array.
        while (bkt != ba->end() && bkt->empty()) // Find the next non-empty bucket.
            ++bkt;
        if (bkt == ba->end()) return *this; // If the end of the bucket array is reached, return the current iterator.
        ent = bkt->begin(); // Set the entry iterator to the beginning of the newly found non-empty bucket.
    }
    return *this; // Return the modified iterator itself.
}

// end() function implementation:
// Returns an iterator pointing to the end (past-the-end) of the HashMap.
template <typename K, typename V, typename H>
typename HashMap<K, V, H>::Iterator HashMap<K, V, H>::end() {
    return Iterator(B, B.end()); // Creates and returns an iterator pointing to the end of the bucket array (B).
}

// begin() function implementation:
// Returns an iterator pointing to the first entry in the HashMap.
template <typename K, typename V, typename H>
typename HashMap<K, V, H>::Iterator HashMap<K, V, H>::begin() {
    if (empty()) return end(); // If the HashMap is empty, return the end() iterator.
    BItor bkt = B.begin();      // Start searching from the beginning of the bucket array.
    while (bkt->empty()) ++bkt; // Find the first non-empty bucket.
    return Iterator(B, bkt, bkt->begin()); // Create and return an iterator pointing to the first entry of the found bucket.
}

// finder() function implementation:
// Finds the entry corresponding to the given key (k) and returns an iterator to its position.
// If the key is not found, it returns an iterator to the end of the respective bucket (endOfBkt).
template <typename K, typename V, typename H>
typename HashMap<K, V, H>::Iterator HashMap<K, V, H>::finder(const K& k) {
    int i = hash(k) % B.size();         // Calculate the hash index (i) for the key (k) by modulo with bucket array size.
    BItor bkt = B.begin() + i;          // Get an iterator to the i-th bucket.
    Iterator p(B, bkt, bkt->begin()); // Create an iterator (p) pointing to the beginning of the i-th bucket.
    // Search for key (k) by iterating through the bucket until the end of the bucket is reached or the key is found.
    while (!endOfBkt(p) && (*p).key() != k)
        nextEntry(p); // Advance iterator p to the next entry.
    return p; // Return the final position (either the entry or the end of the bucket).
}
```

```
// find() function implementation:
// Finds the entry for the given key (k) and returns an iterator to it.
template <typename K, typename V, typename H>
typename HashMap<K, V, H>::Iterator HashMap<K, V, H>::find(const K& k) {
    Iterator p = finder(k); // Use the finder utility function to look for key (k).
    if (endOfBkt(p))          // If the key was not found (i.e., p points to the end of a bucket),
        return end();         // Return the HashMap's end() iterator.
    else
        return p; // If the key was found, return the iterator (p) to its position.
}

// inserter() function implementation:
// Inserts a new entry (e) at the position pointed to by the iterator (p).
template <typename K, typename V, typename H>
typename HashMap<K, V, H>::Iterator HashMap<K, V, H>::inserter(const Iterator& p, const Entry& e) {
    Eltor ins = p.bkt->insert(p.ent, e); // Insert the new entry (e) before the position p.ent in p.bkt (current bucket).
    n++; // Increment the count of entries in the HashMap by 1.
    return Iterator(B, p.bkt, ins); // Create and return an iterator pointing to the newly inserted entry.
}

// put() function implementation:
// Inserts a (k, v) pair into the HashMap, or replaces the value if key (k) already exists.
template <typename K, typename V, typename H>
typename HashMap<K, V, H>::Iterator HashMap<K, V, H>::put(const K& k, const V& v) {
    Iterator p = finder(k); // Search for key (k) using the finder utility function.
    if (endOfBkt(p)) { // If key (k) was not found (i.e., p points to the end of a bucket),
        return inserter(p, Entry(k, v)); // Insert a new entry (k, v) at the end of the bucket and return its position.
    } else { // If key (k) was found,
        p.ent->setValue(v); // Replace the value of the existing entry (pointed to by p.ent) with the new value (v).
        return p; // Return the position of the updated entry.
    }
}

// eraser() function implementation:
// Removes the entry pointed to by the iterator (p) from its bucket.
template <typename K, typename V, typename H>
void HashMap<K, V, H>::eraser(const Iterator& p) {
    p.bkt->erase(p.ent); // Erase the entry pointed to by p.ent from its bucket
(p.bkt).
    n--; // Decrement the number of entries in the HashMap by 1.
}

// erase(const Iterator& p) function implementation:
// Deletes the entry pointed to by iterator p. Internally calls the eraser utility.
template <typename K, typename V, typename H>
void HashMap<K, V, H>::erase(const Iterator& p) {
    eraser(p); // Calls the eraser function to perform the actual deletion.
}

// erase(const K& k) function implementation:
// Deletes the entry corresponding to the given key (k) from the map.
template <typename K, typename V, typename H>
void HashMap<K, V, H>::erase(const K& k) {
    Iterator p = finder(k); // Find the key (k) to be erased using the finder utility
function.
    if (endOfBkt(p))            // If the key was not found (i.e., p points to the end of
a bucket),
        // throw NonexistentElement("Erase of nonexistent"); // A throw could be
added here for erasing a non-existent element. (Currently commented out)
        return; // The assignment example seems to do nothing instead of
throwing an error.
    eraser(p); // If the key was found, call the eraser function to remove the
entry.
}

// Explicit instantiation of the HashMap template for std::string keys, int values,
and stringHash.
// This ensures that the code for this specific template specialization is generated,
// allowing it to be used without linking errors when the template definition is in a
.cpp file.
template class HashMap<std::string, int, stringHash>;
```