Overview

The data structures you have implemented up to this point are all useful and widely used. However, their average performance for insertion and retrieval is generally O(N), or at best O($\log_2$N). As N becomes large—*large* is a relative term, depending on current hardware configuration and performance, data record size and a number of other factors, but let's say hundreds of thousands or millions of records—O(N) becomes a poor choice. Even O($\log_2$N) performance can be unacceptable when handling many simultaneous queries or processing large reports. How does searching, inserting, and retrieving in O(1) sound? That is the possibility that the Hash Table ADT tries to offer. Hash tables are among the fastest data structures for most operations and come closest to offering O(1) performance. Consequently, a hash table is the preferred data structure in many contexts. For instance, most electronic library catalogs are based on hash tables.

The goal of the ideal hash table is to come up with a unique mapping of each key onto a specific location in an array. The mapping of the key to a specific location in an array is handled by the hash operation. A hash operation will accept the key as input and return an integer that is used as an index into the hash table.

How does this work? The simplest case occurs when the key is an integer. Then the hash function could simply return an integer. For instance, given the key 3, the hash function would return the index value 3 to place the record in the hash table position 3. The key 1 would be used to place the record in hash table position 1.

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*
Key		1		3			

But what about a key value of 8? The array used to implement the hash table does not have a valid position 8, so some set of operations must be performed on the key in order to map it to an index value that is valid for the array. A simple solution to this problem is to perform a modulus operation with the table size on the key value. Using the example of 8 for the table of size 7 above, the hash function would calculate 8 modulus 7 to produce an index value of 1.

Unfortunately, it is easy for the hash calculation to generate the same index value for more than one key. This is called a **collision**. Using a key of 10 in the example above, the hash calculation would produce 3—calculated as 10 modulus 7 (`10%7` in C++). But position 3 already has a key associated with it—the key 3. There are a number of methods that can be used to resolve the collision. The one we use is called **chaining**. When using chaining, all the keys that generate a particular index are connected to that array position in a chain. One way to implement chaining is by associating a list with each table entry. Using this approach, position 0 in the hash table would have a list of all data items for which the hash operation produces an index of 0, position 1 would have a list of all data items associated with index 1, and so on through index 6. The key values 1, 3, 7, 8, 10, and 13 would produce the following chains associated with the indexes 0, 1, 3, and 6.

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*
Key	7	1		3			13
		8		10			

Generating an index for other key types is more complicated than generating an index for integers. For instance, if the key for a record is a string, the string could be mapped to an integer by adding up the ASCII values of each of the characters in the string. Given a last name of "smith", the function could calculate a value of 115 ('s') + 109 ('m') + 105 ('i') + 116 ('t') + 104 ('h') = 549. Real numbers can be mapped to integers by simply truncating the noninteger part.

Note that these are simple examples of hash operations intended as an introduction to hash tables. A more detailed explanation would go into great detail about how to take a key and move the bits around to produce a high-quality key that will ensure a fairly uniform distribution of data items throughout the table. See In-lab 3 for more detail.

Hash Table ADT

Data Items

The data items in a hash table are of generic type DT.

Structure

The hash table is an array of singly linked lists. The list into which a data item is placed is determined by the index calculated using the data item's hash operation. The placement within a particular list is determined by the chronological order in which the data items are inserted into the list—the earliest insertion takes place at the head of the list, the most recent at the end of the list. The ordering within a particular list is *not* a function of the data contained in the hash table data items. You interact with each list by using the standard list operations.

Operations

```
HashTbl ( int initTableSize ) throw ( bad_alloc )
```

Requirements:
None

Results:
Constructor. Creates the empty hash table.

```
~HashTbl ()
```

Requirements:
None

Results:
Destructor. Deallocates (frees) the memory used to store a list.

```
void insert ( const DT &newDataItem ) throw ( bad_alloc )
```

Requirements:
Hash table is not full.

Results:
Inserts `newDataItem` into the appropriate list. If a data item with the same key as `newDataItem` already exists in the list, then updates that data item's nonkey fields with `newDataItem`'s nonkey fields. Otherwise, it inserts it at the end of that list.

```
bool remove ( KF searchKey )
```

Requirements:
None

Results:
Searches the hash table for the data item with key `searchKey`. If the data item is found, then removes the data item and returns `true`. Otherwise, returns `false`.

```
bool retrieve ( KF searchKey, DT &dataItem )
```

Requirements:
None

Results:
Searches the hash table for the data item with key `searchKey`. If the data item is found, then copies the data item to `dataItem` and returns `true`. Otherwise, returns `false` with `dataItem` undefined.

```
void clear ()
```

Requirements:
None

Results:
Removes all data items in the hash table.

```
bool isEmpty () const
```

Requirements:
None

Results:
Returns `true` if a hash table is empty. Otherwise, returns `false`.

```
bool isFull () const
```

Requirements:
None

Results:
Returns `true` if a hash table is full. Otherwise, returns `false`.

```
void showStructure () const
```

Requirements:
None

Results:
Outputs the data items in a hash table. If the hash table is empty, outputs "Empty hash table". Note that this operation is intended for testing/debugging purposes only. It supports only list data items that are one of C++'s predefined data types (`int`, `char`, and so forth).

Laboratory 14: Prelab Exercise

Name _____ Date _____

Section _____

You can implement a hash table in many ways. We have chosen to implement the hash table using chaining to resolve collisions. The singly linked list ADT provides a simple way of dealing with a chain of data items and is an opportunity to use one of your ADTs to implement another ADT. Your instructor may choose to let you use one of the STL (Standard Template Library) lists instead.

Step 1: Implement the operations in the Hash Table ADT using an array of lists to store the list data items. You need to store the number of hash table slots (`tableSize`) and the actual hash table itself (`dataTable`). Base your implementation on the following declarations from the file *hashtbl.h*. An implementation of the `showStructure` operation is given in the file *show14.cpp*. If you are using an STL list, modify the `showStructure` operation to work with that STL list.

```
template < class DT, class KF >
class HashTbl
{
    public:
        HashTbl ( int initTableSize );
        ~HashTbl ();

        void insert ( const DT &newDataItem) throw ( bad_alloc );
        bool remove ( KF searchKey );
        bool retrieve ( KF searchKey, DT &dataItem );
        void clear ();

        bool isEmpty () const;
        bool isFull () const;

        void showStructure () const;
    private:
        int tableSize;
        List<DT> *dataTable;
};
```

Step 2: Save your implementation of the Hash Table ADT in the file *hashtbl.cpp*. Be sure to document your code.

The following program was adapted from the Lab 4 (Ordered List ADT) prelab. It reads in account numbers and balances for a set of accounts. It then tries retrieving records using the account numbers as the keys. The primary change is that the `Account` struct needs to have a `hash()` function added to be usable with the hash table. We also removed the code outputting the accounts in ascending order based on their account numbers because an ordered traversal of the hash table is not something supported by this Hash Table ADT.

```
// lab14-example1.cpp
#include <iostream>
#include <cmath>
#include "hashtbl.cpp"

using namespace std;

struct Account
{
    int acctNum;              // (Key) Account number
    float balance;            // Account balance

    int getKey () const { return acctNum; }
    int hash(int key) const { return abs( key ); }
};

void main()
{
    HashTbl<Account,int> accounts(11);    // List of accounts
    Account acct;                         // A single account
    int searchKey;                        // An account key

    // Read in information on a set of accounts.

    cout << endl << "Enter account information for 5 accounts: "
         << endl;

    for ( int i = 0; i < 5; i++ )
    {
        cin >> acct.acctNum >> acct.balance;
        accounts.insert(acct);
    }

    // Checks for accounts and prints records if found

    cout << endl;
    cout << "Enter account number: ";
    while ( cin >> searchKey )
    {
       if ( accounts.retrieve(searchKey,acct) )
           cout << acct.acctNum << " " << acct.balance << endl;
       else
           cout << "Account " << searchKey << " not found." << endl;
    }
};
```

Laboratory 14: In-lab Exercise 1

Name _____ Date _____

Section _____

One possible use for a hash table is to store computer user login usernames and passwords. Your program should load username/password sets from the file *password.dat* and insert them into the hash table until the end of file is reached on *password.dat*. There is one username/password set per line, as shown in the following example.

```
jack
broken.crown
jill
tumblin'down
mary         contrary
bopeep       sheep!lost
```

Your program will then present a login prompt, read one username, present a password prompt, read the password, and then print either "Authentication successful" or "Authentication failure", as shown in the following examples.

Login: `jack`

Password: `broken.crown`

Authentication successful

Login: `jill`

Password: `tumblingdown`

Authentication failure

This authentication loop is to be repeated until the end of input data (EOF) is reached on the console input stream (`cin`).

Step 1: Prepare a test plan that specifies how you will validate that your program works correctly.

Step 2: Create a program that will read in the usernames and passwords from *password.dat* and then allow the user to try authenticating usernames and passwords as shown so long as the user enters more data. Store your program in the file *login.cpp*.

Create an appropriate `struct` to hold the username/password sets in the hash table.

Step 3: Run your program and test according to your test plan. If you discover mistakes, correct them and execute your test plan again.

Test Plan for the Login Authentication Program

Test Case	*Expected Result*	*Checked*

Laboratory 14: In-lab Exercise 2

Name _____ Date _____

Section _____

A hash table insertion or retrieval with no collisions is an O(1) operation. Collisions reduce the O(1) behavior to something less desirable. There are two ways to reduce collisions:

- Increase the size of the table. As the table size increases, the statistical probability of collisions for unique keys decreases. The problem with arbitrarily increasing the table size is that the amount of physical memory is finite and to declare a wildly large table in the hope of reducing collisions wastes memory.
- Enhance the quality of the `hash()` function so that it produces fewer collisions. Ideally, unique keys have unique indexes into the hash table and no collisions. This is called a **perfect hash**. The problem with generating perfect hash tables is that the hash function must be carefully crafted to avoid collisions.

  A **minimal perfect hash** is a hash table with the following two properties:

- The minimal property—the memory allocated to store the keywords is exactly large enough to hold the needed number of keys and no more. For *n* keys, there are exactly *n* table entries.
- The perfect property—locating a table entry requires at most one key comparison. There are no collisions. Consequently, no collision resolution is required.

Software developers like minimal perfect hash tables for specific sets of strings because of the performance boost. For instance, it is very helpful if a C++ compiler can perform an O(1) lookup on a string to determine whether it is a C++ reserved word.

Step 1: Develop a `hash()` function implementation that will produce a minimal perfect hash for the following seven C++ reserved words.

- `double`
- `else`
- `if`
- `return`
- `switch`
- `void`
- `while`

Use the following `struct` to hold the strings.

```
struct Identifier
{
    string ident;
    string getKey() const
           { return ident; }
    int hash( string key ) const
           { return . . . }
};
```

Step 2: Prepare a test plan that specifies how you will verify that your `hash()` function works correctly to generate a minimal perfect hash table.

Step 3: Implement a test program using the above `struct` to demonstrate that you have indeed developed a minimal perfect hash for the given seven C++ identifiers. Save your program in the file *perfect.cpp*. Use the provided `showStructure()` function to display the hash table after all the data has been entered.

Step 4: Execute your test program and consider the results according to your test plan. If you discover mistakes in your implementation of the `hash()` function, correct them and execute your test plan again.

Test Plan for the `hash()` function

Hash Formula	*Expected Result*	*Checked*