346:210-810

Introduction to Computer Science II

Project #3
An Application of Linked Lists: Hash Tables

James R. Daehn
Department of Computer Science
University of Akron

Delivered: 30 Mar 2012
Due: 15 Apr 2012

***Note**: Unless otherwise cited, the contents of this project is an adaptation of Project 5.2 Application of Linked Lists: Hash Tables found in Nyhoff's lab manual that accompanies his textbook on data structures.*[1]

1 Introduction

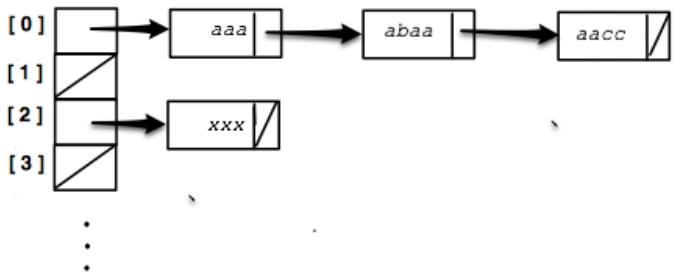
In this project, you will be using a *dynamic array* of linked lists to implement a hash table. To process the linked lists, you will use the `LinkedList` template class developed in the text book.

A ***hash table*** is a data structure in which the location of an item is determined as a function of the item rather than by a sequence of trial-and-error comparisons; it is used when fast searching is needed. Ideally, search time is $O(1)$; i.e., it is constant – independent of the number of items to be searched. There are a number of ways that a hash table may be implemented. In this project, we will use the method known as chaining, in which the hash table is implemented using an array (or vector) of linked lists.

When an item is to be inserted into the table, a hashing function h is applied to determine where it is to be placed in the table; for example a common one is

$$h(item) \quad = \quad item \,\%\, size \tag{1}$$

where the table size is usually taken to be a prime number in order to scatter ("hash") the items throughout the table. For nonnumeric items, numeric codes – e.g., ASCII – are used. The linked list at location $h(item)$ is searched for the item. If it not found, the item is inserted into the linked list at this location.



In many situations, hash tables turn out to be more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets.[2]

2 Project Requirements

2.1 `LinkedList` Modifications

This project requires *two* modifications to the `LinkedList` template class found in your book (and coded for you in your project folder). The first modification is to modify the `insertNode()` member function such that it only inserts a new item if and only if it is not already in the list. That is, a class invariant is that the list contains zero or more items in ascending order with only one occurrence of each item.

❑ After implementing and testing your modified `insertNode()` member function, commit your changes to `LinkedList.h`.

Since a hash table is a data structure that is used to quickly *find* items, and since our implementation of a hash table employs a linked list as a way to

resolve collisions, your second modification to the `LinkedList` template class will be to add a `findNode()` operation. The prototype for this `LinkedList` member function is shown in Figure 1.

```
/* findNode(const T& value) const
 *
 * Precondition: none
 * Postcondition: findNode returns true if value is somewhere
 * in the linked, false otherwise. Both the linked list
 * and the value remain unchanged.
 */
bool findNode(const T& value) const;
```

Figure 1: Prototype for new `LinkedList` member function.

❑ After implementing and testing your `findNode()` member function, commit your changes to `LinkedList.h`.

2.2 `HashTable` Specification and Implementation

Your next steps are to design and implement a `HashTable` class for processing hash tables. To facilitate your efforts, a blank header file `HashTable.h` and implementation file `HashTable.cpp` are found in your project folder.

1. Design a `HashTable` class for processing hash tables. The table shall hold thirteen `LinkedList<string>` objects. The `HashTable` class should provide *at least* the following member functions:

   (a) Default constructor in which you shall *dynamically* create the storage for the thirteen lists.

   (b) Destructor in which you shall free the dynamic array.

   (c) Insert an item into the hash table as previously described.

   (d) Find an item in the hash table where if the item is found, return true, otherwise return false.

   (e) Display each index in the hash table and a list of all items stored at that location.

   For this particular problem, the hash table is to store strings and use the following hash function:

```
h(str) = (sum of ASCII codes of first 3 chars of str) % table_size
```

For strings with fewer than three characters, just use whatever characters are in the string.

2. Write a driver program to test your class. It should read several strings, storing each in the hash table. After all the strings are input, it should display the hash table by listing the indices and, for each index, a list of all the words in the linked list at that location.

3. After you have thoroughly test your class, use it to process the strings in the text file `usconst.txt`

3 Deliverables

In addition to checking in your work at regular intervals, please hand in:

1. Print outs of all source files (`.cpp` and `.h`)

4 Grading

Your project will be evaluated by the following categories:

Category	Earned	Possible
Correctness of (`LinkedList`) class modifications		20
Correctness of (`HashTable`) class		25
Design and structure of the (`HashTable`) class		15
Documentation (including specifications of functions)		10
Style (white space, alignment, etc.)		5
Driver program (checks all operations)		25
Total		**100**

5 Bonus

Analyze the performance of your hash table. Time how long it takes to construct the hash table and then search for different words in `usconst.txt`. Next, time how long it takes to construct a linked list that stores the words of `usconst.txt` and then search for the same words. For a good test, take

a look at the list that is created and see what word is first; last and then pick a word you know isn't in the file.

To time your code, consider the following code:

```
#include <ctime>

int main() {
    const int PRECISION = 6;

    clock_t startTime = clock();

    // do some processing...

    clock_t endTime = clock();

    // NOTE: The following expression is only split across
    // several lines for space considerations...

    double elapsedTime = static_cast<double>(endTime);
    elapsedTime = elapsedTime - static_cast<double>(startTime);
    elapsedTime = elapsedTime / static_cast<double>(CLOCKS_PER_SEC);

    // print out the elapsed time, in seconds of
    // said processing...
    cout << Elapsed time: ";
    cout << fixed << setprecision(PRECISION);
    cout << elapsedTime;
    cout << " seconds!" << endl;

    return 0;
}
```

NOTE: On a real fast machine, you may need to increase your precision to display a very short time interval.

References

[1] Larry Nyhoff. *Lab Manual to Accompany ADTs, Data Structures, and Problem Solving with C++.* Pearson Prentice Hall, Second edition, 2006.

[2] wikipedia.org. Hash table. http://en.wikipedia.org/wiki.