Name: Michael Beaver

Course: CS 355

Semester: Fall 2012

Assignment Number: 8

Assignment Type: Homework 6

Assignment Description: You will create a template Hash Table

Assignment Due Date: Tuesday, October 23, 2012 (first half of class)

To Be Included in Portfolio: YES

Total Grade: Implementation (60), Test Cases (20), and Analysis (20)

Write a template Hash Table. This hash table should have the following operations:

1. Constructor: The constructor should take in as a parameter the number of keys to be stored in the hash table. You should dynamically create your array such that the size of the hash table is roughly twice the size of the parameter and is prime.
2. Dynamic Memory Methods: Copy constructor, Assignment Operator, and Destructor to handle dynamic array
3. Show Fill: Print a “picture” of the table to demonstrate how the table is filled. You may simply print X’s where slots are taken as opposed to the actual data values.

*void ShowFill()const;*

1. Print: Print the location values and data of slots taken.

*void ShowContents()const;*

1. Insert: Insert a key by h(x) = x % tablesize; Return the number of slots hit before a slot is found. Should return 1 if hits actual hashed slot.

*int Insert(T key);*

1. Remove: Remove a key. Return the number of slots hit before finding the slot to remove. If not found, return 0 or the negative of the number of slots that had to be searched before removing.

*int Remove (T key);*

1. Search: Search for a key. Return the number of slots hit before finding the slot of the key. If not found, return 0 or the negative of the number of slots that had to be searched before removing.

*int Search(T key);*

1. Collision: Taking in the original hashed slot, and the number of times you have tried to find a slot, determine the next slot for the new key. Use linear probing for collision handling.

*int NewSlot(T HashVal, int trynumber)*

Write a driver that mimics the driver I gave you for the BST. Remove any items that no longer make sense. The Collision method will be tested through Insert and no extra call is needed in the driver. Add ‘Z’ to the menu for ShowFill. There can be a private method called IsPrime written for use in the constructor.

Test Case Requirements Met:

\_\_\_\_ created at least one test case for each method

\_\_\_\_ test cases showed methods were correct

Analysis Requirements Met:

\_\_\_\_ Clear and correct communication

\_\_\_\_ Reasonable/correct answers and justifications

Name:

Course: CS 355

Semester: Fall 2012

Assignment Number: 8

Assignment Type: Homework 6 – Test Cases

Assignment Description: Create Test cases for each of the methods. You should show enough test cases to demonstrate the method works. Be sure to think about special cases. Try to break your code.

Assignment Due Date: Tuesday, October 23, 2012 (first half of class)

To Be Included in Portfolio: YES

Test Case 1

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Test Case 2

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Name: Michael Beaver

Course: CS 355

Semester: Fall 2012

Assignment Number: 8

Assignment Type: Homework 6 - Analysis

Assignment Description: Carefully answer the questions below. Be sure you answer in complete sentences and with correct grammar. The space provided is not an indicator for the space needed to answer the question.

Assignment Due Date: Tuesday, October 30, 2012 (beginning of class)

To Be Included in Portfolio: YES

Question 1: Create a separate driver that tests your Hash Table with a data set of size 10, 50, 100, 1000, 10,000. Making use of the integer values returned from the insert, remove, and search methods, justify that those routines have runtimes of O(1). Feel free to show tables in Excel or generate them with your new statistical driver.

First, let us define the table fill ratio as R = F/N, where F is the number of filled slots and N is the table size. Also, it should be noted that all numbers were pseudo-random and modulo by the data set size (e.g., rand() % 50 + 1).

**Insert**

The Insert method will have a runtime of O(1) given that the table fill ratio is relatively low. For example, for the data set size of 10, the table size is 23. If we insert 10 values, the table fill ratio becomes 10/23 ≈ 0.43, which means less than half of the table is filled. Hence, if N is 23, then the Insert method will not be O(N) unless the table is nearly full or full.

Considering the linear probing collision handling technique, the runtime will correlate to size of the primary clusters. If a value is hashed to a slot, but there is a cluster around that slot of size X, then it will take another X - 1 probes to find the available slot. Hence, it is possible that linear probing could have a runtime of O(X), where X is the cluster size.

However, the overall runtime of the Insert method will not be O(N) because the entirety of the table is not processed. Rather, a subset of the table is processed. Various tests have shown that the number of slots processed during insertion will not approach the table size, but the number of slots processed should approach the specified number of keys to be hashed. Of course, the number of slots processed will approach the table size if more keys than specified are inserted. Once that happens, the runtime will gradually approach O(N), as the number of slots processed approaches the size of the table N.

This table is an excerpt from an insertion test (using pseudo-random values) on a data set size of 50. The table is of size 101.

|  |  |  |  |
| --- | --- | --- | --- |
| Insertion # | Value to Insert | Slots Hit | Comments |
| . . . | . . . | . . . | . . . |
| 39 | 29 | 10 | This value had to probe 10 slots. R = 38/101 before insertion. |
| 40 | 42 | 5 | This value had to probe 5 slots. R = 39/101 before insertion. |
| 41 | 43 | 9 | This value had to probe 9 slots. R = 40/101 before insertion. |
| 42 | 5 | 1 | Pure constant runtime. R = 41/101 before insertion. |
| 43 | 28 | 12 | This value had to probe 12 slots. R = 42/101 before insertion. |
| 44 | 8 | 4 | This value had to probe 4 slots. R = 43/101 before insertion. |
| 45 | 36 | 17 | This value had to probe 17 slots. R = 44/101 before insertion. |
| **46** | **20** | **34** | **This value had to probe 34 slots! R = 45/101 before insertion.** |
| 47 | 25 | 30 | This value had to probe 30 slots. R = 46/101 before insertion. |
| 48 | 32 | 24 | This value had to probe 24 slots. R = 47/101 before insertion. |
| . . . | . . . | . . . | . . . |

Interestingly, the value 20 had to probe 34 slots before finding an empty slot. At the time of insertion, the primary cluster must have had size X = 34. Hence, the linear probing had to linearly traverse 34 slots before finding an available slot. While this example illustrates that linear probing can have runtimes of O(X), it also shows that the runtime of Insert will not be O(N). In this example, N = 101, and by the end of insertion, S = 50; thus, R = 50/101 ≈ 0.5, which is approximately half the table.

Consider the relatively extreme case that the value 20 had to probe 34 slots before insertion. Clearly the runtime of Insert is not O(N) because N slots were not traversed. In fact, only approximately one-third of the table was processed. In the absolute worst case that the entire table is filled (i.e., 101 values), then the runtime would be O(N); however, we are assuming that the user will only insert the specified number of keys (i.e., 50).

Clearly the runtime of the algorithm is not logarithmic relative to the size of the table; likewise, only a portion of the table will ever be processed. Hence, unless the user inserts more keys than specified, the runtime will never approach O(N). Once the table fill ratio crosses a certain threshold, then the runtime will start to experience a linear runtime relative to N due to the linear probing collision handling scheme. However, a general insertion should happen in constant time relative to the table size.

The ample room provided by the size of the table helps to ensure the constant runtime. If the table’s size was exactly the specified number of keys or smaller, then the runtime would most likely extend to O(N). As the number of values inserted increases, the slots available decreases. Hence, as more collisions occur, the runtime will approach O(N) as more slots are touched before insertion.

**Search**

The same basic reasoning for Insert’s constant runtime holds for Search’s constant runtime as well. Of note is how Search actually works. The Search algorithm hashes a value, and it tests the original hash location for equality. If the value is not stored in the original location, then it performs linear probing to find the value in the primary cluster. If the algorithm reaches an available slot before finding the value, then the value is obviously not in the table. For such a case, the runtime is O(X), where X is the size of the primary cluster, because the cluster has to be traversed before finding the available slot. Obviously, the entire table may be traversed and the value may not be found. In that worst case, the runtime would definitely be O(N), where N is the size of the table. However, a general search will have constant runtime relative to the table size.

This table is an excerpt from a search test (using pseudo-random values) on a data set size of 100. The table is of size 211.

|  |  |  |  |
| --- | --- | --- | --- |
| Search # | Value to Find | Slots Hit | Comments |
| . . . | . . . | . . . | . . . |
| 91 | 94 | 1 | Pure constant runtime. The value is at its original hash slot. |
| 92 | 78 | 5 | Hit 5 slots before finding the value. The cluster size may not be 5. |
| **93** | **47** | **0** | **Value not found.** |
| 94 | 37 | 1 | Pure constant runtime. The value is at its original hash slot. |
| **95** | **12** | **0** | **Value not found.** |
| **96** | **28** | **0** | **Value not found.** |
| 97 | 74 | 37 | Hit 37 slots before finding the value. The cluster size may not be 37. |
| 98 | 13 | 1 | Pure constant runtime. The value is at its original hash slot. |
| **99** | **44** | **0** | **Value not found.** |
| **100** | **2** | **0** | **Value not found.** |

Clearly the values 94, 37, and 13 were hashed to their original locations, so searching for them happens in pure constant runtime. Of note are values 47, 12, 28, 44, and 2. These values were not found, as indicated by the number of slots hit being equal to zero. Because only 100 values were inserted, it reasonable to rule out a runtime of O(N), where N = 211, for “finding” these values. Obviously the entire table was not searched, so the runtime cannot be O(N). The only other alternative is that the values were not found at their original hash locations, so they were searched for by using linear probing. Once an empty slot was found, it was determined that the value was not in the table. Therefore, the runtime for finding these values is at most O(X), where X is the size of the respective primary clusters.

However, even a runtime of O(X) is less than the runtime O(N) since X < N—unless, of course, the primary cluster takes up the entire table. Hence, in relation to the size of the table, probing the primary clusters is relatively constant. Thus, searching for the highlighted values in the table happened in constant time in relation to the size of the table. Therefore, any search of a table that is not full and has a reasonable fill ratio R will have constant runtime. If a majority of the table is filled, then the runtime will obviously increase. However, if only the specified number of keys are inserted and searched, then the runtime will be constant in relation to the size of the table.

**Remove**

The same basic reasoning for the constant runtime of the Insert and Search methods also applies to the Remove method. Indeed, the Remove algorithm is similar to the Search algorithm; the only obvious difference is that the found value is deleted from the table in the Remove algorithm.

In fact, if a value is stored in its original hash slot, the runtime of removing the value will always be constant. If the value is stored in part of a primary cluster, then part or the entire cluster will have to be traversed. Assuming the cluster is of size X, the runtime of the traversal of the cluster will be O(X). Hence, unless the cluster is the size of the table, the runtime will never be O(N) because X < N. The following table helps to illustrate this point.

This table is an excerpt from a removal test (using pseudo-random values) on a data set size of 10,000. The table is of size 20,011.

|  |  |  |  |
| --- | --- | --- | --- |
| Remove # | Value to Remove | Slots Hit | Comments |
| . . . | . . . | . . . | . . . |
| 9,914 | 6417 | 8 | Hit 8 slots before removal. |
| 9,915 | 5667 | 0 | Value not found. |
| 9,916 | 1752 | 0 | Value not found. |
| 9,917 | 1084 | 1949 | Hit 1949 slots before removal |
| 9,918 | 76 | 0 | Value not found. |
| 9,919 | 3750 | 0 | Value not found. |
| 9,920 | 440 | 0 | Value not found. |
| 9,921 | 7388 | 4 | Hit 4 slots. |
| **9,922** | **2020** | **3944** | **Hit 3944 slots before removal!** |
| 9,923 | 2963 | 368 | Hit 368 slots before removal. |
| . . . | . . . | . . . | . . . |

The Remove method hit an amazing 3944 slots before removing the value 2020. If we assume the primary cluster is of size 3944, then the cluster is 3944/20011 ≈ 0.20 ≈ 20% of the table. That is a relatively small percentage of the table; granted, it is a huge cluster, assuming it is not of a greater size than 3944.

Regardless, the runtime is O(X), where X is the size of the primary cluster. Because X < N when the user only inserts the specified number of keys, it is safe to conclude that not all table slots were touched during removal. Hence, it is reasonable to suggest that in any general case—bearing in mind the conditions that the user only inserts the specified number of keys and the table size is at least twice the number of keys to be inserted and prime—not all of the slots in the table will be hit before removing a value. Therefore, the runtime of the Remove method cannot be O(N); rather, the runtime is constant relative to the table size.