EC-200 Data Structures

Lab Manual 12

**Course Instructor:** Lecturer Anum Abdul Salam

**Lab Engineer:** Lab Engineer Ayesha Batool

**Student Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Degree/ Syndicate: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Trait** | **Obtained Marks** | **Maximum Marks** |
| **R1** | **Application Functionality 20%** |  | 20 |
| **R2** | **Specification & Data structure implementation**  **30%** |  | 30 |
| **R3** | **Reusability**  **10%** |  | 10 |
| **R4** | **Input Validation**  **10%** |  | 10 |
| **R5** | **Efficiency**  **20%** |  | 20 |
| **R6** | **Delivery**  **10%** |  | 10 |
| **R7** | **Plagiarism above 80%** |  | 1 |
|  | **Total** |  | 10 |

**Total Marks = O**𝒃𝒕𝒂𝒊𝒏𝒆𝒅 𝑴𝒂𝒓𝒌𝒔 (∑6𝟏 𝑹𝒊 ∗ 𝑹7)

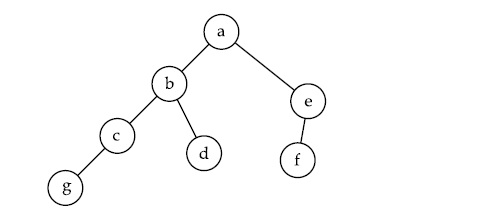
# Lab Manual # 12: Hash Tables & STL

## Lab Objective:

To Implement Hash Table ADT.

## Lab Description:

Hash tables are another type of data structures. Before going into the details of hash table, we will see why we need hash tables? In binary search trees complexity of operations (insert, search, delete) is said to be O(logN). This is only possible if tree is balanced. If the BST is not balanced, complexity of these operations is not O(logN).



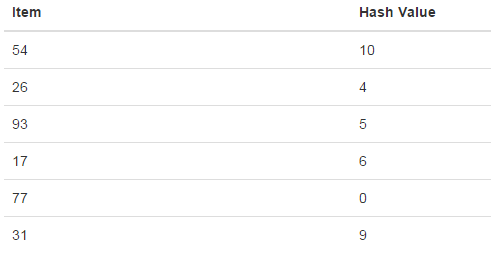
**Figure 12.1:** Unbalanced BST

An alternate to reduce complexity is hash table.

**Hash Tables**

A hash table is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a slot, can hold an item and is named by an integer value starting at 0. For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on. Initially, the hash table contains no items so every slot is empty. We can implement a hash table by using a list with each element initialized to the special value.

The mapping between an item and the slot where that item belongs in the hash table is called the **hash function**. The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and m-1. Assume that we have the set of integer items 54, 26, 93, 17, 77, and 31. Our first hash function, sometimes referred to as the “remainder method,” simply takes an item and divides it by the table size, returning the remainder as its hash value (h(item)=item%11). Figure 11.2 gives all of the hash values for our example items. Note that this remainder method (modulo arithmetic) will typically be present in some form in all hash functions, since the result must be in the range of slot names.



**Figure 12.2:** Mapping values in hash table using hash function Item%size

Now when we want to search for an item, we simply use the hash function to compute the slot name for the item and then check the hash table to see if it is present. This searching operation is O(1), since a constant amount of time is required to compute the hash value and then index the hash table at that location. If everything is where it should be, we have found a constant time search algorithm.

You can probably already see that this technique is going to work only if each item maps to a unique location in the hash table. For example, if the item 44 had been the next item in our collection, it would have a hash value of 0 (44%11==0). Since 77 also had a hash value of 0, we would have a problem. According to the hash function, two or more items would need to be in the same slot. This is referred to as a collision (it may also be called a “clash”). Clearly, collisions create a problem for the hashing technique. We will discuss them in detail later.

**What is the hash Function?**

Given a collection of items, a hash function that maps each item into a unique slot is referred to as a perfect hash function. If we know the items and the collection will never change, then it is possible to construct a perfect hash function.

**Data** **Index Where data is to be placed**

**Hash Function**

**Figure 12.3:** Hash Function

**Collision Resolution**

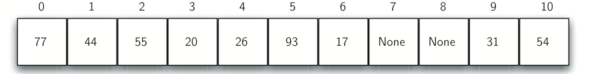
When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called collision resolution. As we stated earlier, if the hash function is perfect, collisions will never occur. However, since this is often not possible, collision resolution becomes a very important part of hashing. Some approaches being used in collision resolution are

1. Chaining
2. re-hashing,
3. using neighboring slots (linear probing),
4. quadratic probing,

**Linear Probing**

Start at the original hash value position and then move in a sequential manner through the slots until we encounter the first slot that is empty. Note that we may need to go back to the first slot (circularly) to cover the entire hash table. By systematically visiting each slot one at a time, we are performing an open addressing technique called linear probing.

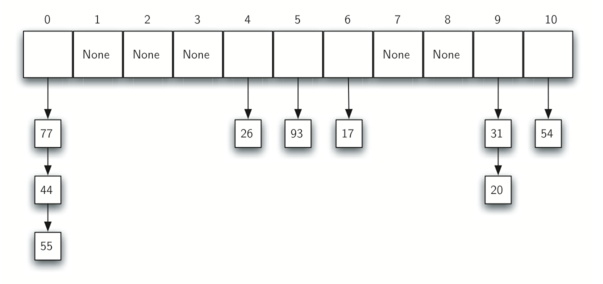
When we attempt to place 44 into slot 0, a collision occurs. Under linear probing, we look sequentially, slot by slot, until we find an open position. In this case, we find slot 1. Again, 55 should go in slot 0 but must be placed in slot 2 since it is the next open position. The final value of 20 hashes to slot 9. Since slot 9 is full, we begin to do linear probing. We visit slots 10, 0, 1, and 2, and finally find an empty slot at position 3.



**Figure 12.4:** Linear Probing

**Chaining**

An alternative method for handling the collision problem is to allow each slot to hold a reference to a collection (or chain) of items. Chaining allows many items to exist at the same location in the hash table. When collisions happen, the item is still placed in the proper slot of the hash table. As more and more items hash to the same location, the difficulty of searching for the item in the collection increases. Figure 11.6 shows the items as they are added to a hash table that uses chaining to resolve collisions.



**Figure 12.5:** Chaining

**Re-hashing or double hashing**

Double hashing uses a secondary hash function ***d***(***k***) and handles collisions by placing an item in the first available cell of the series.

**Functions in Hash Function ADT**

1. **boolisEmpty()** Returns true if the hash table is empty. Otherwise, returns false
2. **boolisFull()** Returns true if the hash table is full. Otherwise, returns false
3. **void insert (const DT &newDataItem)** Inserts newDataItem into the appropriate list in the hash table. The location (index) in the hash table is determined by the key and the hash function.
4. **bool remove (KF searchkey)** Searches the hash table for the data item with the key searchKey. If the data item is found, then removes the data item and returns true. Otherwise, returns false.
5. **bool retrieve (KF searchkey, DT &dataItem)**Searches the hash table for the data item with the key searchKey. If the data item is found, then copies the data item to dataItem and returns true. Otherwise, returns false.
6. **void clear()** Removes all data items in the hash table.
7. **voidshowStructure()** Outputs the data items in a hash table. If the hash table is empty, outputs, "Empty hash table". This is meant for testing/debugging purposes.

**Applications of Hash tables**

1. For driver's license record. With a hash table, you could quickly get information about the driver (i.e. name, address, age) given the license number.
2. For internet search engines.
3. For telephone book databases. You could make use of a hash table implementation to quickly look up John Smith's telephone number.
4. For electronic library catalogs. Hash Table implementations allow for a fast find among the millions of materials stored in the library.
5. For implementing passwords for systems with multiple users. Hash Tables allow for a fast retrieval of the password which corresponds to a given username.

## LAB TASKS:

1. Create a Website login system. System should provide an interactive panel to register the users. User can login the system if password matched. Insert passwords into the Hash Table retrieve one user's Password structure from the Hash Table compare retrieved user password to input password and print "Authentication failure" or "Authentication successful". Add in a line to insert passwords into the table. Build and Run your program.

**Note:**

* **Use STL in the Lab Task (Restricted).**
* **This task will be considered as Assignment # 03.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.** | **Operations** | **Expected Results** | **Results/Status** |
| **1.** | Login user “User1” | Not Registered |  |
| **2.** | Sign up user  Name: user1  Password: user1 | Registered Successfully |  |
| **3.** | Login user “User1”  Name: user1  Password: user1 | Login Successful |  |
| **4.** | Login user “User1”  Name: user1  Password: check | Incorrect Password |  |
| **5.** | Sign up user  Name: user1  Password: hello | User Already registered |  |
| **6.** | Sign up user  Name: user2  Password: user2 | Registered Successfully |  |
| **7.** | Login user “User2”  Name: user2  Password: user2 | Login Successful |  |
| **8.** | Display registered users | User1  User2 |  |