**Data Structures - Lecture Notes**

**Hash Tables**

**A. Map ADT:**

A map Sometimes referred to as “dictionary” or “associative array” is a structure which stores pairs of values. Each pair contains two elements:

* **Key:** A unique value associates with each entry.
* **Value:** the value of the entry.

The keys are used to decide where the element is stored within the table. The most obvious example of a map ADT is a normal dictionary, the key in this case will be the word and the value will be the meaning of the word. The position of the word in a dictionary is determined by the word itself.

The map is usually used only for storing and retrieving information, so it is minimally associated with the following list of functions:

* Size: returns the number of elements stored.
* isEmpty: returns true if there are no elements stored.
* get(k): returns the entry with key k.
* put(k,v): add an entry with key k and value v to the map.
* remove(k): removes the entry with key k from map.

**Implementation:**

There are a number of different ways to implement a map:

***1. Linked List:***

All elements in a map can be stored in a linked list, but as we know to search for a certain value in a linked list you will need to go through the list from the start till you reach the required element. To insert an element (given that the keys are unique) you will have to search the list to make sure that the key does not already exist in the map. To delete an element, you need to locate it first before you delete it. This makes all the operations of the map run in O(n).

***2. Binary Search Trees:***

Storing the elements in a binary search tree sorted with key values will ensure that the operations run in an average time of O(log n).

***3. Hash Tables:***

The hash table is an implementation to the map ADT that performs all its operations in O(1) in average.

**B. Hash Table Implementation:**

A hash table, put simply, is an abstraction of an array that allows any value to be used as an index. While an array requires that indices be integers, a hash table can use a floating-point value, a string, another array, or even a structure as the index. This index is called the key, and the contents of the array element at that index is called the value. So a hash table is a data structure that stores key/value pairs and can be quickly searched by the key. Because insertion and removal are operations dependent on the speed of the search, they tend to be fast as well.

The idea behind hashing is to perform insertions, searches and deletion in a constant time (O(1)). This is achieved by using a "Hash Function", which is a function used to convert the key of an element to an index within the array.

So when inserting an element into the hash table, what you need to do is apply the hash function to the key of the element (the hash function in O(1)), the hash function will return an index within the array, you can next directly place the element inside the array (which takes O(1)). When searching for an element you search by the key, the hash function is again used to calculate the index within the array and when the index is located the value could be retrieved directly.

***A simple Example:***

Assume that we have an array of size 10 (index values are 0 to 9) and we want to store the data of employees. The key is the ID of the employee and the value is the name. Assume that the ID values are 1 to 10, we could use a hash function:

**Hash (ID) = ID -1**

So if we need to store and employee with ID = 5, we calculate the hash function as (5-1) which gives 4, and we store the element in the array cell with index 4.

**C. Terminology:**

The following are the most important terms related to hash tables:

* **Hash Function:** A function that takes a key and returns a number pointing to a specific position within an array indicating where the value should be stored.
* **Hash Value:** The value returned by the hash function.
* **Collision:** The case in which two different keys have the same hash value and hence need to be stored in the same position.
* **Bucket:** The name used to refer to a slot in the array.
* **Load Factor (ʎ):** It is an important performance indicator. It is calculated as (n/T) where n is the number of elements stored in the array and T is the array size.

**D. Hash Functions:**

The hashing approach requires that the hash function is a one-to-one mapping from each k to an index within the array. Such a function is known as a perfect hashing function: it maps each key to a distinct integer within some manageable range and enables us to achieve an O(1) search time.

Unfortunately, finding a perfect hashing function is not always possible. Let's say that we can find a hash function, h(k), which maps most of the keys onto unique integers, but maps a small number of keys on to the same integer. If the number of collisions (cases where multiple keys map onto the same integer), is sufficiently small, then hash tables work quite well and give O(1) search times.

So what makes a hash function good?

1. The hash function should be fully determined by the data being hashed.

This means that it uses only the key to derive its value, and it's value should change with the change in the key value. So a hash function like hash(key) = 17 is a very bad hash function because it returns the same value for all keys which causes a lot of collisions. Furthermore it does not use the key at all.

2. The hash function "uniformly" distributes the data across the entire set of possible hash values.

The hash function should distribute the values evenly over the array. It should not create groups of values close together (clusters) and it should not leave spaces unused in the array.

3. Must return a value within the table bounds.

Returning a value outside the table bounds makes the function unusable. Therefore, most hash functions calculate a hash value h then use h%T (where T is the table size) as an index to make sure that it never goes outside the bounds.

4. Given the same key, it should always return the same value.

For the hash table to work properly the hash function should always return the same value when given the same value for the key. This ensures that when you search the table for a value you will find it in the same position that it was inserted in.

**Commonly used Hash Functions:**

***1. Modular Hashing:***

Used for integer keys. It calculates the value of the key modulus (%) the table size. It is a good function since it depends on the key, ensures that the values are within the range of the table and it distributes the values evenly over the table. To make it perform even better and decrease the probability of collisions it is recommended that the table size be a prime number.

***2. Handling Strings:***

What if the key is a string? The hash function needs to convert the string into an integer value that falls within the bounds of the array. Most functions that deal with strings start by converting each character in the string into its ASCII code value then performing some operation on the result. One possible option is summing all the values and then taking the result modulus the table size.

***3. The folding method:***

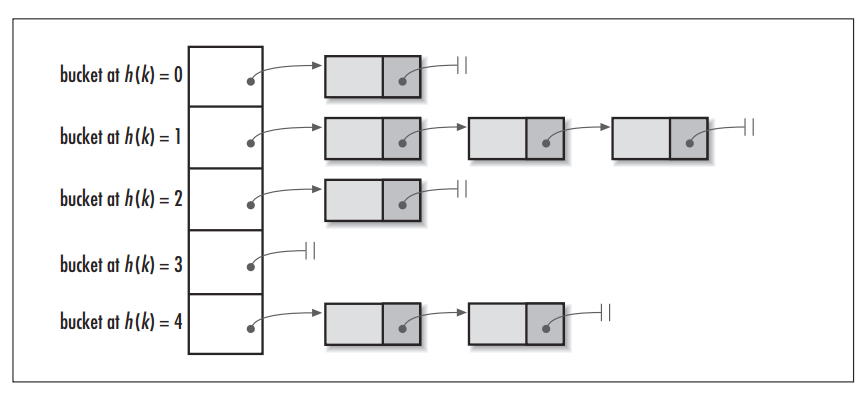
The folding method functions divides the key into parts and performs operations on each part individually. The previous method for handling strings can fall under this category.

**E. Collision Resolution:**

A hash function converts keys into array indices. The second component of a hashing algorithm is collision resolution: a strategy for handling the case when two or more keys to be inserted hash to the same index.

***1. Separate Chaining:***

Separate Chaining uses an array of linked lists as the hash table. When there is a collision, the colliding key can be pushed onto the list, thus preserving both values. When searching for a key, the index is hashed and then the list is searched, and deletion is as simple as deletion from a linked list. The structure is clarified in the Figure 1:



**Figure 1. Separate Chaining for Collision Resolution**

**Evaluation:**

***Advantages:***

* The table has no hard size limit. This means that you can store any number of values in the table, they can even be more than the size of the table. Which means also that you do not need prior knowledge about the number of values that you will need to store.
* It can handle an unlimited number of collisions.
* As more collisions happen the performance of all techniques will be less but it does not degrade as fast when using separate chaining.
* It is the only technique that can easily handle duplicate keys.
* Easy to delete records from the linked list.

***Disadvantages:***

* You will need to search the linked list to find the required key.
* It uses more memory than just using the table to store values.

***2. Open Addressing:***

When a collision occurs, open addressing algorithm calculates another location within the table to store the value. It uses three different techniques:

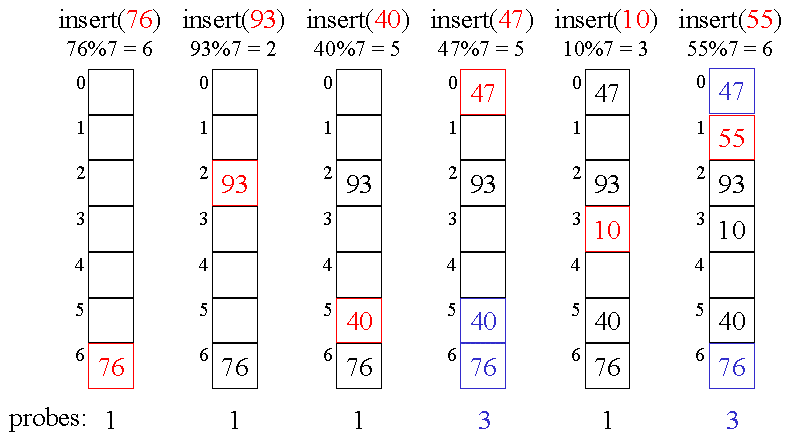
i. Linear Probing:

It is the simplest technique to handle collisions. When a collision is detected, the index is incremented by 1 until an empty bucket is found. The new place to insert the data is calculated as:

**NewHash(Key) = [Hash(Key)+ i] % T**

Where i is called "The Rehash Index" which is the number of times a collision has happened at the specific cell that has the index returned by the original hash function. and T is the table size.

Figure 2 describes the process: As an example, when you insert 47 the hash function will return 5 which is already taken, so the search moves to position 6 which is also taken. Since it reached the end of the table it goes back to 0 which is empty and that is where 47 is placed.



**Figure 2. Linear Probing**

The search follows the same process it first goes to the slot indicated by the hash function and if it did not find the value it searches the next slot and so on until either the value is found or it reaches and empty bucket.

Deleting may become difficult with this strategy since if you directly delete a value and leave the slot empty it may cause problems. When a search happens it could stop thinking that the value it is searching for does not exist because it has hit an empty slot. The solution would be not to permanently delete the value but just mark the space as deleted so that the search would recognize that there might be other values next.

**Clustering**

In linear probing, keys tend to cluster. That is, several parts of the table may become full quickly while others remain completely empty. Because linear probing expects a large number of empty buckets uniformly distributed among the used buckets, clusters will cause a large number of buckets to be searched before an empty bucket is found. This slows down search significantly, and in turn slows down insertion and removal as well.

ii. Quadratic Probing:

To attempt to solve the problem of clustering, you may use a non constant value to be added to the address in case of collision. The new place to insert the data is calculated as:

**NewHash(Key) = [Hash(Key)+ i2 ] % T**

This means that when a collision occurs the insert function will go to the next position (i2=1), then it go to a slot 4 spaces away (i2=4), then 9 and so on.

iii. Double Hashing

Double hashing uses two different hash functions. The first hash function is used as usual, and the second hash function is used to create a step size. The new place to insert the data is calculated as:

**NewHash(Key) = [Hash(Key)+ f(i)] % T**

**f(i) = i\*Hash2 (Key)**

Since the second hash function is dependent on the key (not only the rehash index) this means that clustering can be avoided and also reduces the probability of more collisions occurring. The only consideration you need to take care of is that the second hash function cannot be allowed to return 0 because this will cause the process to go into an infinite loop. This occurs because you add 0 to the original address so it gives the same value, and even with the increase in the value of i, it will be multiplied by 0 so no change will occur.

**F. Performance Evaluation:**

***1. Complexity:***

The hash table's best case performance is O(1). All the attempts to choose the right hash function and the right collision resolution technique aim at making the best case also the average case. Generally, the performance is between O(1) and O(log n). And the worst case performance will be O(n).

***2. Hash Function Effect:***

Choosing the wrong hash function can cause the performance to drop to the worst case of O(n). For example, a hash function that maps all keys to a single value means that you have to search all values to find a specific key (O(n)). Also one of the most important properties of a hash function is that it should be fast to calculate. A slow hash function can cause the whole performance of the table to be much worse.

***3. Collision Resolution Strategy Effect:***

Choosing the right collision resolution strategy can highly affect the performance of the hash table. In general, separate chaining is the best strategy for the following reasons:

* If you use a good hash function, the values will be distributed all over the table, which makes the chains (lists attached to each cell) short. So when you search you need only to search the list and this keeps the complexity to O(1).
* The performance of all techniques become worse when the number of values stored in the table increases, but for separate chaining the performance degrades more slowly.
* Separate chaining handles deletions very easily since all you need to do is delete and element from the list.
* It is the only technique that can handle duplicate keys.

The only disadvantage of separate chaining which might cause the developer to choose another technique for collision resolution is that it uses much more memory than other methods. So if space is an issue, separate chaining is not preferred.

***4. Load Factor Effect:***

As mentioned above, the load factor is the ratio between the number of values in the hash table and the capacity of the table. Even if you use a good hash function the load factor affects the performance of the hash table. All the mentioned collision resolution techniques perform equally well when the load factor is kept below 0.5. As the load factor increases the performance of all these techniques will degrade. Generally, it is recommended that the load factor be kept between 0.5 and 0.7. If the load factor goes above 0.7 it is recommended that the hash table be resized. The process of resizing the hash table is called "rehashing".

Rehashing:

When you need to resize the hash table, the typical process is to recalculate the index for each entry in the table again. The hash function can be different specially if you are using a modular function which includes the size of the table. Rehashing is a time consuming process that should not be repeated often. The technique is the same no matter what collision resolution strategy is being used.