DATA STRUCTURES

HASH TABLE

A hash table is a type of data structure that stores key-value pairs.

The key is sent to a hash function that performs arithmetic operations on it.

The result (commonly called the hash value or hash) is the index of the key-value pair in the hash table.

Search – Searches an element in a hash table.

Insert – inserts an element in a hash table.

delete – Deletes an element from a hash table.

Components of a hash table

1. Hash function

   a. the hash function determines the index of our key-value pair.
   b. Choosing an efficient hash function is a crucial part of creating a good hash table.
   c. You should always ensure that it's a one-way function, i.e., the key cannot be retrieved from the hash.
   d. Another property of a good hash function is that it avoids producing the same hash for different keys.

2. Array
   - The array holds all the key-value entries in the table.
   - The size of the array should be set according to the amount of data expected.

Types of Hashing

A collision occurs when two keys get mapped to the same index. There are several ways of handling collisions.

Linear probing

a. the hashing technique is used to create an already used index of the array.
b. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell.
c. This technique is called linear probing.
d. Hash Function:

```
int hashCode(int key){
   return key % SIZE;
}
```

Insert Operation:

```
void insert(int key,int data) {
   struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
   item->data = data;
   item->key = key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty or deleted cell
   while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {
      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   hashArray[hashIndex] = item;
}
```

Search Operation

```
struct DataItem *search(int key) {
   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] != NULL) {

      if(hashArray[hashIndex]->key == key)
         return hashArray[hashIndex];

      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   return NULL;
}
```

Delete Operation

```
struct DataItem* delete(struct DataItem* item) {
   int key = item->key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] !=NULL) {

      if(hashArray[hashIndex]->key == key) {
         struct DataItem* temp = hashArray[hashIndex];

         //assign a dummy item at deleted position
         hashArray[hashIndex] = dummyItem;
         return temp;
      }

      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   return NULL;
}
```

Time Complexity:

Insertion operation

- Best Case Time Complexity: O(1)
- Worst Case Time Complexity: O(N). This happens when all elements have collided and we need to insert the last element by checking free space one by one.
- Average Case Time Complexity: O(1) for good hash function; O(N) for bad hash function

Search operation

- Best Case Time Complexity: O(1)

- Worst Case Time Complexity: O(N)
- Average Case Time Complexity: O(1) for good hash function; O(N) for bad hash function
- Space Complexity: O(1) for search operation

Deletion operation

- Best Case Time Complexity: O(1)
- Worst Case Time Complexity: O(N)
- Average Case Time Complexity: O(1) for good hash function; O(N) for bad hash function
- Space Complexity: O(1) for deletion operation

Chaining Method

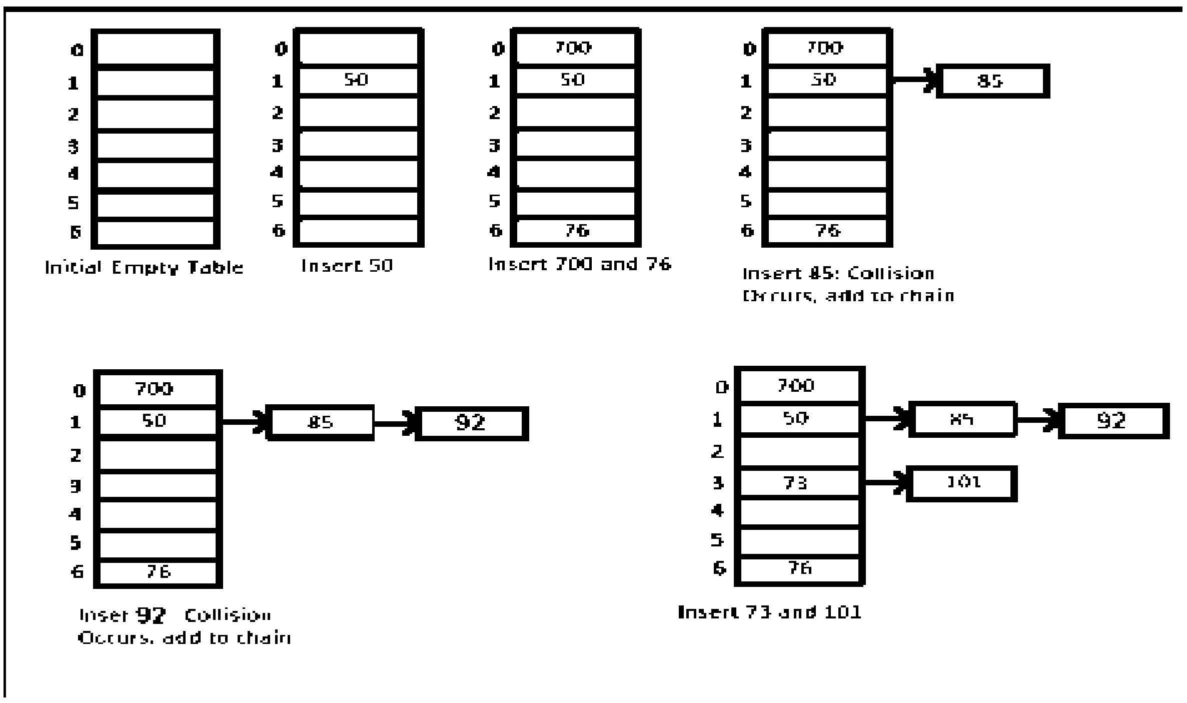
The hash table will be an array of linked lists. All keys mapping to the same index will be stored as linked list nodes at that index.

chaining is to implement the array as a linked list called a chain. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions.

when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.

this technique supposes our hash function h(x) ranging from 0 to 6. So for more than 7 elements, there must be some elements, that will be places inside the same room. For that we will create a list to store them accordingly. In each time we will add at the beginning of the list to perform insertion in O(1) time

Example: Let us consider a simple hash function as “key mod 7” and a sequence of keys as 50, 700, 76, 85, 92, 73, 101



Insert data into the separate chain

1. Declare an array of a linked list with the hash table size.

2. Initialize an array of a linked list to NULL.

3. Find hash key.

4. If chain[key] == NULL

    Make chain[key] points to the key node.

5. Otherwise(collision),

    Insert the key node at the end of the chain[key].

Searching a value from the hash table

1. Get the value

2. Compute the hash key.

3. Search the value in the entire chain. i.e. chain[key].

4. If found, print "Search Found"

5. Otherwise, print "Search Not Found"

Removing an element from a separate chaining

To remove an element from the hash table, We need to find the correct chain. i.e. chain[value%key].

After the chain found, we have to use linked list deletion algorithm to remove the element.

1. Get the value

2. Compute the key.

3. Using linked list deletion algorithm, delete the element from the chain[key].

Linked List Deletion Algorithm: Deleting a node in the linked list

4. If unable to delete, print "Value Not Found"

Advantages:

- Simple to implement.
- Hash table never fills up, we can always add more elements to the chain.
- Less sensitive to the hash function or load factors.
- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- The cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
- Wastage of Space (Some Parts of the hash table are never used)
- If the chain becomes long, then search time can become O(n) in the worst case
- Uses extra space for links

Time complexity:

Data Structures For Storing Chains:

1. Linked lists

- Search: O(l) where l = length of linked list
- Delete: O(l)
- Insert: O(l)
- Not cache friendly