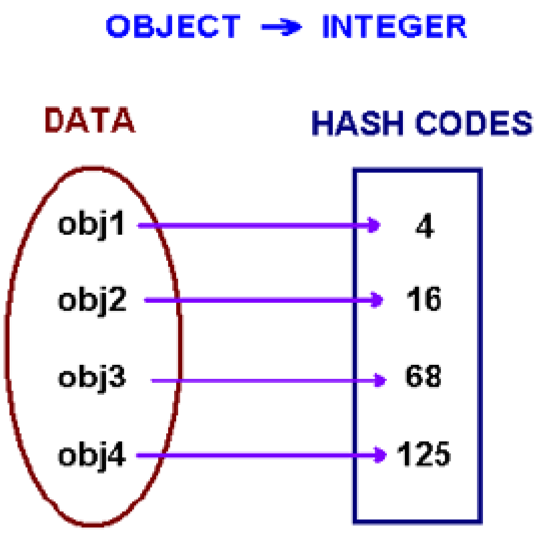
Source: https://www.andrew.cmu.edu/course/15-121/lectures/Hashing/hashing.html

# Concept of Hashing

## Introduction

The problem at hands is to speed up searching. Consider the problem of searching an array for a given value. If the array is not sorted, the search might require examining each and all elements of the array. If the array is sorted, we can use the binary search, and therefore reduce the worse-case runtime complexity to O(log n). We could search even faster if we know in advance the index at which that value is located in the array. Suppose we do have that magic function that would tell us the index for a given value. With this magic function our search is reduced to just one probe, giving us a constant runtime O(1). Such a function is called a **hash function** . A hash function is a function which when given a key, generates an address in the table.



The example of a hash function is a *book call number*. Each book in the library has a *unique* call number. A call number is like an address: it tells us where the book is located in the library. Many academic libraries in the United States, uses Library of Congress Classification for call numbers. This system uses a combination of letters and numbers to arrange materials by subjects.

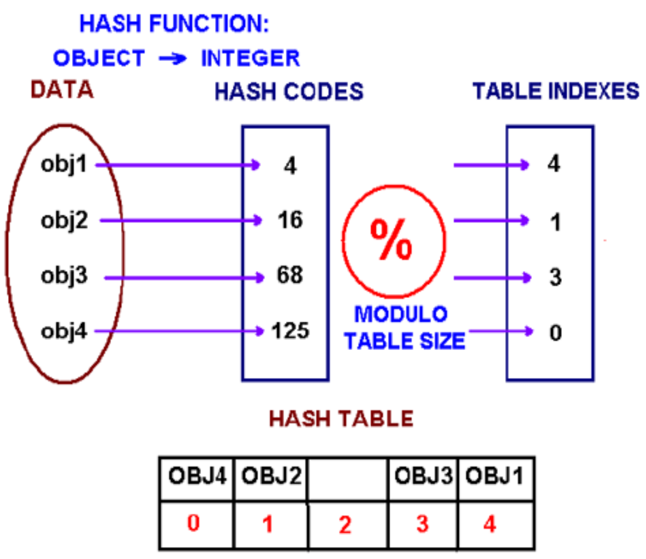
A hash function that returns a unique hash number is called a **universal hash function**. In practice it is extremely hard to assign unique numbers to objects. The later is always possible only if you know (or approximate) the number of objects to be proccessed.

Thus, we say that our hash function has the following properties

- it always returns a number for an object.
- two equal objects will always have the same number
- two unequal objects not always have different numbers

The precedure of storing objets using a hash function is the following.

> Create an array of size *M*. Choose a hash function *h*, that is a mapping from objects into integers *0, 1, ..., M-1*. Put these objects into an array at indexes computed via the hash function *index = h(object)*. Such array is called a **hash table**.



How to choose a hash function? One approach of creating a hash function is to use Java's *hashCode()* method. The hashCode() method is implemented in the Object class and therefore each class in Java inherits it. The hash code provides a numeric representation of an object (this is somewhat similar to the toString method that gives a text representation of an object). Conside the following code example

```
        Integer obj1 = new Integer(2009);
        String obj2 = new String("2009");
        System.out.println("hashCode for an integer is " + obj1.hashCode());
        System.out.println("hashCode for a string is " + obj2.hashCode());
```

It will print

```
        hashCode for an integer is 2009
        hashCode for a string is 1537223
```

The method hasCode has different implementation in different classes. In the String class, hashCode is computed by the following formula

        s.charAt(0) * $31^{n-1}$ + s.charAt(1) * $31^{n-2}$ + ... + s.charAt(n-1)

where *s* is a string and *n* is its length. An example

        "ABC" = 'A' * $31^{2}$ + 'B' * 31 + 'C' = 65 * $31^{2}$ + 66 * 31 + 67 = 64578

Note that Java's *hashCode* method might return a negative integer. If a string is long enough, its hashcode will be bigger than the largest integer we can store on 32 bits CPU. In this case, due to integer overflow, the value returned by *hashCode* can be negative.

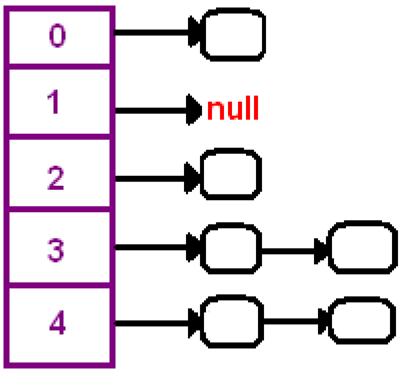
Review the code in HashCodeDemo.java.

## Collisions

When we put objects into a hashtable, it is possible that different objects (by the *equals()* method) might have the same hashcode. This is called a **collision**. Here is the example of collision. Two different strings ""Aa" and "BB" have the same key: .

```
        "Aa" = 'A' * 31 + 'a' = 2112
        "BB" = 'B' * 31 + 'B' = 2112
```

How to resolve collisions? Where do we put the second and subsequent values that hash to this same location? There



are several approaches in dealing with collisions. One of them is based on idea of putting the keys that collide in a linked list! A hash table then is an array of lists!! This technique is called a *separate chaining* collision resolution.

The big attraction of using a hash table is a constant-time performance for the basic operations `add`, `remove`, `contains`, `size`. Though, because of collisions, we cannot guarantee the constant runtime in the worst-case. Why? Imagine that all our objects collide into the same index. Then searching for one of them will be equivalent to searching in a list, that takes a liner runtime. However, we can guarantee an expected constant runtime, if we make sure that our lists won't become too long. This is usually implemnted by maintaining a *load factor* that keeps a track of the average length of lists. If a load factor approaches a set in advanced threshold, we create a bigger array and *rehash* all elements from the old table into the new one.

Another technique of collision resolution is a *linear probing*. If we cannoit insert at index k, we try the next slot k+1. If that one is occupied, we go to k+2, and so on. This is quite simple approach but it requires new thinking about hash tables. Do you always find an empty slot? What do you do when you reach the end of the table?

## HashSet

In this course we mostly concern with using hashtables in applications. Java provides the following classes HashMap, HashSet and some others (more specialized ones).

HashSet is a regular set - all objects in a set are distinct. Consider this code segment

```
String[] words = new String("Nothing is as easy as it looks").split(" ");

HashSet<String> hs = new HashSet<String>();

for (String x : words) hs.add(x);

System.out.println(hs.size() + " distinct words detected.");
System.out.println(hs);
```

It prints "6 distinct words detected.". The word "as" is stored only once.

HashSet stores and retrieves elements by their content, which is internally converted into an integer by applying a hash function. Elements from a HashSet are retrieved using an Iterator. The order in which elements are returned depends on their hash codes.

Review the code in HashSetDemo.java.

The following are some of the HashSet methods:

- set.add(key) -- adds the key to the set.
- set.contains(key) -- returns true if the set has that key.
- set.iterator() -- returns an iterator over the elements

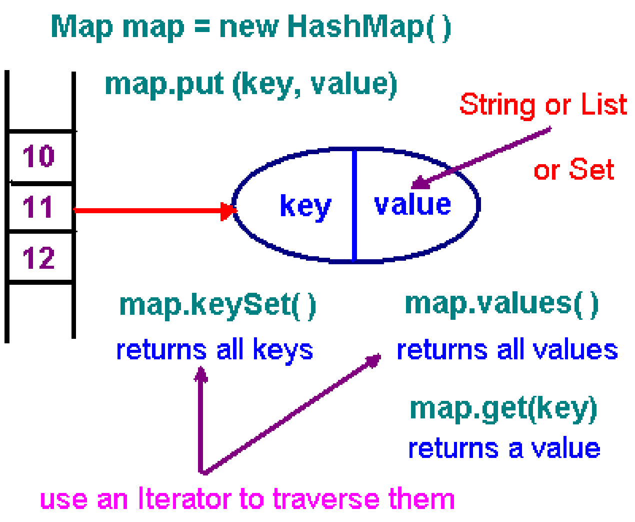
## Spell-checker

You are implement a simple spell checker using a hash table. Your spell-checker will be reading from two input files. The first file is a dictionary located at the URL *http://www.andrew.cmu.edu/course/15-121/dictionary.txt* . The program should read the dictionary and insert the words into a hash table. After reading the dictionary, it will read a list of words from a second file. The goal of the spell-checker is to determine the misspelled words in the second file by looking each word up in the dictionary. The program should output each misspelled word.

See the solution here Spellchecker.java.

## HashMap

HashMap is a collection class that is designed to store elements as key-value pairs. Maps provide a way of looking up one thing based on the value of another.



```
Iterator itr = map.keySet().iterator();
while( itr.has.Next())
{
   Object key = itr.next();
   System.out.println(map.get(key));
}
```

We modify the above code by use of the HashMap class to store words along with their frequencies.

```
String[] data = new String("Nothing is as easy as it looks").split(" ");

HashMap<String, Integer> hm = new HashMap<String, Integer>();

for (String key : data)
{
  Integer freq = hm.get(key);
  if(freq == null) freq = 1; else freq ++;
  hm.put(key, freq);
}
```

```
        System.out.println(hm);
```

This prints `{as=2, Nothing=1, it=1, easy=1, is=1, looks=1}`.

HashSet and HashMap will be printed in no particular order. If the order of insertion is important in your application, you should use *LinkeHashSet* and/or *LinkedHashMap* classes. If you want to print dtata in sorted order, you should use *TreeSet* and or *TreeMap* classes

Review the code in SetMapDemo.java.

The following are some of the HashMap methods:

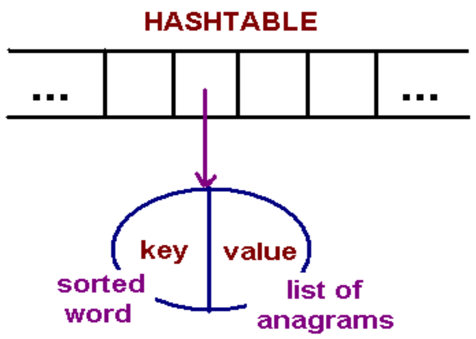
- map.get(key) -- returns the value associated with that key. If the map does not associate any value with that key then it returns null. Referring to "map.get(key)" is similar to referring to "A[key]" for an array A.
- map.put(key,value) -- adds the key-value pair to the map. This is similar to "A[key] = value" for an array A.
- map.containsKey(key) -- returns true if the map has that key.
- map.containsValue(value) -- returns true if the map has that value.
- map.keySet() -- returns a set of all keys
- map.values() -- returns a collection of all value

## Anagram solver

An anagram is a word or phrase formed by reordering the letters of another word or phrase. Here is a list of words such that the words on each line are anagrams of each other:

```
        barde, ardeb, bread, debar, beard, bared

        bears, saber, bares, baser, braes, sabre
```

In this program you read a dictionary from the web site at *http://www.andrew.cmu.edu/course/15-121/dictionary.txt* and build a **Map( )** whose key is a sorted word (meaning that its characters are sorted in alphabetical order) and whose values are the word's anagrams.



See the solution here Anagrams.java.

Source: https://www.geeksforgeeks.org/hashing-data-structure/

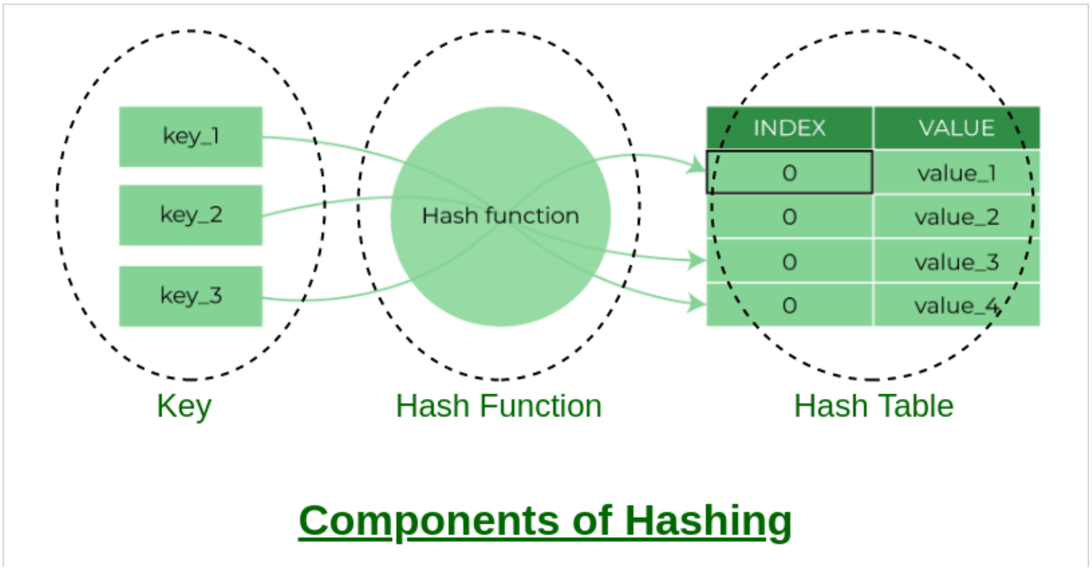
# Hashing Data Structure

- Last Updated : 19 Apr, 2023

## What is Hashing?

Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.

Let a hash function H(x) maps the value **x** at the index **x%10** in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.



Hashing Data Structure

Source: https://www.section.io/engineering-education/hashing-in-data-structures/

## EngEd Author Bio

[Anubhav Bansal]

Anubhav is passionate about Computer Science. He is a hard worker and a rational thinker who loves to logically deconstruct a problem to find innovative solutions. With a multi disciplinary approach in life, he always gives emphasis on being a team player and recognises how reliability can lead to success.

# Hashing in Data Structures

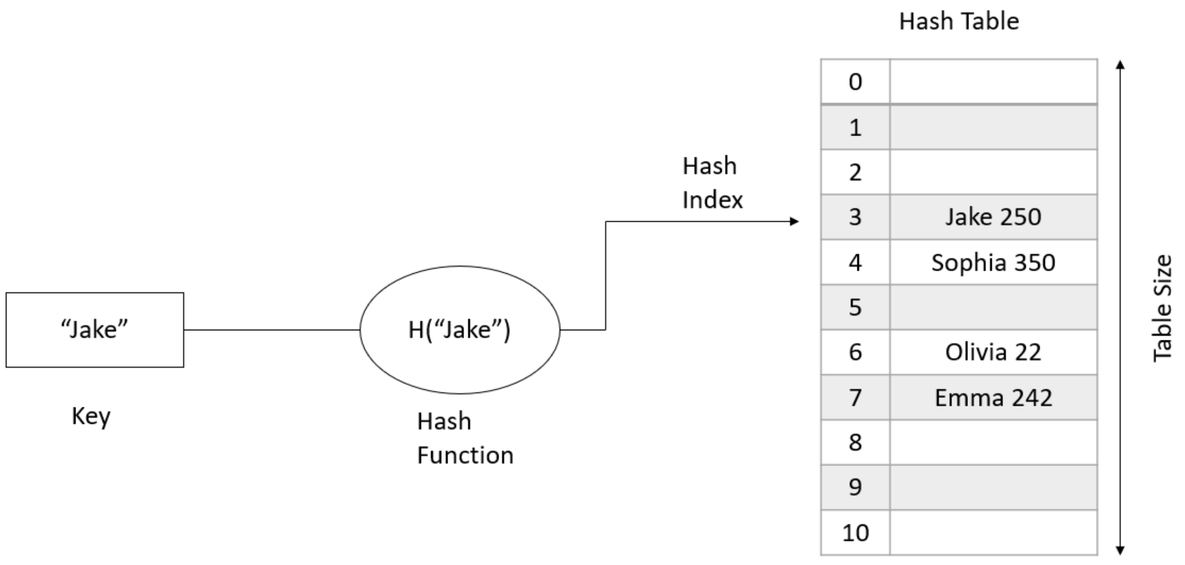
***June 20, 2021***

- Topics:
- Languages

Hashing is the process of converting an input of any length into a fixed size string or a number using an algorithm. In hashing, the idea is to use a hash function that converts a given key to a smaller number and uses the small number as an index in a table called a hash table.

## Hashing in Data Structures

We generate a hash for the input using the hash function and then store the element using the generated hash as the key in the hash table.



**Hash Table**: The hash table is a collection of key-value pairs. It is used when the searching or insertion of an element is required to be fast.

Operation in hash function:

- **Insert** - T[ h(key) ] = value;
  - It calculates the hash, uses it as the key and stores the value in hash table.
- **Delete** - T[ h(key) ] = NULL;
  - It calculates the hash, resets the value stored in the hash table for that key.
- **Search** - return T[ h(key) ];
  - It calculates the hash, finds and returns the value stored in the hash table for that key.

**Hash Collision**: When two or more inputs are mapped to the same keys as used in the hash table. Example: h("John") == h( "joe")

A collision cannot be completely avoided but can be minimized using a 'good' hash function and a bigger table size.

The chances of hash collision are less if the table size is a prime number.

## How to choose a Hash Function

- An efficient hash function should be built such that the index value of the added item is distributed equally across the table.
- An effective collision resolution technique should be created to generate an alternate index for a key whose hash index corresponds to a previously inserted position in a hash table.
- We must select a hash algorithm that is fast to calculate.

## Characteristics of a good Hash Function

- **Uniform Distribution**: For distribution throughout the constructed table.

- **Fast**: The generation of hash should be very fast, and should not produce any considerable overhead.

## Collision Hashing Techniques

1. **Open Hashing (Separate Chaining)**: It is the most commonly used collision hashing technique implemented using Lined List. When any two or more elements collide at the same location, these elements are chained into a single-linked list called a chain. In this, we chain all the elements in a linked list that hash to the same slot.

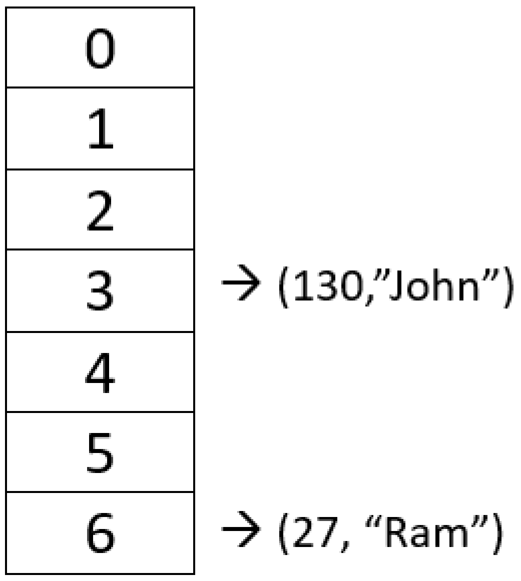
Let’s consider an example of a simple hash function.

```
h(key) = key%table size
```

In a hash table with the size 7

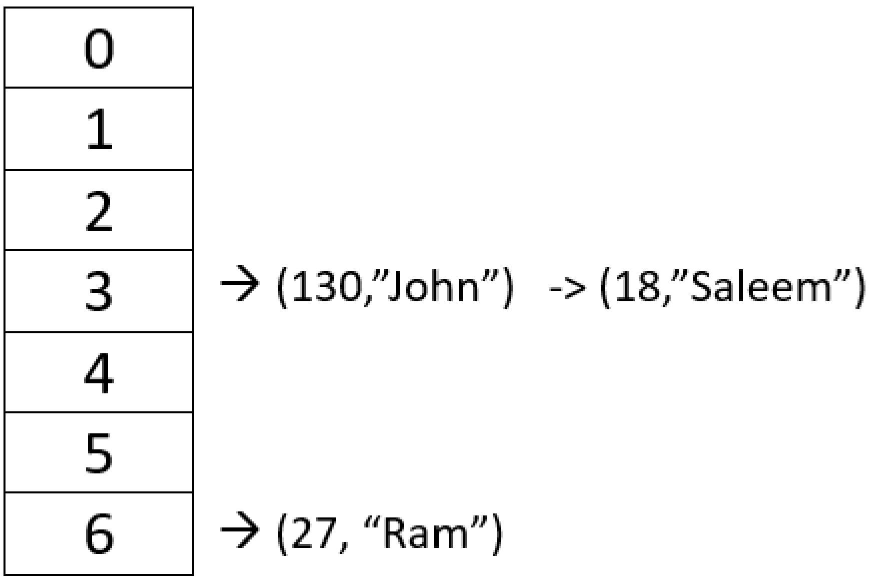
h(27) = 27%7 = 6

h(130) = 130%7 = 4



If we insert a new element (18, “Saleema”), that would also go to the 4th index.

h(18) = 18%7 = 4



;

For separate chaining, the worst-case scenario is when all of the keys will get the same hash value and will be inserted in the same linked list. We can avoid this by using a good hash function.

2. **Closed Hashing (Open Addressing)**: In this, we find the "next" vacant bucket in Hash Table and store the value in that bucket.
    1. **Linear Probing**: We linearly go to every next bucket and see if it is vacant or not.

        ```
        rehash(key) = (n+1)%tablesize
        ```

    2. **Quadratic Probing**: We go to the 1st, 4th, 9th … bucket and check if they are vacant or not.

        ```
        rehash(key) = (n+ k<sup>2</sup> ) % tablesize
        ```

3. **Double Hashing**: Here we subject the generated key from the hash function to a second hash function.

    ```
    h2(key) != 0 and h2 != h1
    ```

**Load Factor**: This is a measurement of how full a hash table may become before its capacity is increased.

The hash table's load factor, T, is defined as:

- N = number of elements in T - Current Size
- M = size of T - Table Size
- e = N/M - Load factor

Generally, if the load factor is greater than 0.5, we increase the size of the bucket array and rehash all the key-value pairs again.

## How Hashing gets O(1) complexity?

Given the above examples, one would wonder how hashing may be O(1) if several items map to the same place…

The solution to this problem is straightforward. We use the load factor to ensure that each block, for example, (linked list in a separate chaining strategy), stores the maximum amount of elements fewer than the load factor on average. Also, in practice, this load factor is constant (generally 10 or 20). As a result, searching in 10 or 20 elements become constant.

If the average number of items in a block exceeds the load factor, the elements are rehashed with a larger hash table size.

### Rehashing

When the load factor gets "too high" (specified by the threshold value), collisions would become more common, so rehashing comes as a solution to this problem.

- We increase the size of the hash table, typically, doubling the size of the table.
- All existing items must be reinserted into the new doubled size hash table.

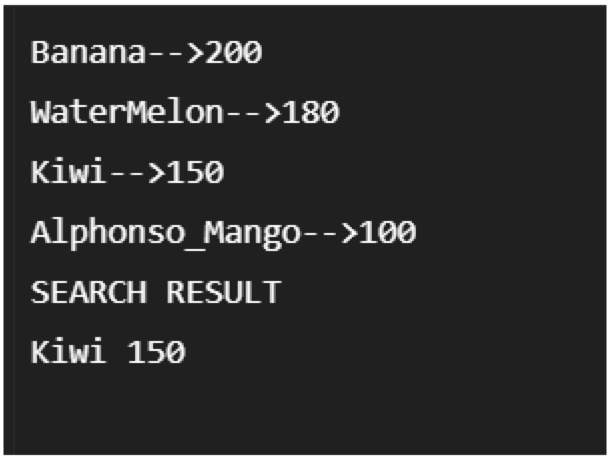
Now let's deep dive into the code. I will implement everything in code that we have learned till now.

```
#include<iostream>
using namespace std;

class node{
    public:
    string name;
    int value;
    node* next;
    node(string key,int data){
        name=key;
        value=data;
        next=NULL;
    }
};

class hashmap{
    node** arr;
    int ts;
    int cs;

    int hashfn(string key){
        int ans=0;
        int mul=1;
        for(int i=0; key[i]!='\0';i++){
            ans =  (ans + ((key[i]/ts)*(mul%ts))%ts);
            mul *= 37;
            mul %=ts;
        }
        ans = ans %ts;
        return ans;
    }
    void reHash(){
        node** oldarr=arr;
        int oldts=ts;
        arr= new node*[2*ts];
        ts *= 2;
        cs=0;

        for(int i=0;i<ts;i++){
            arr[i]=NULL;
        }

        //insert in new table
        for(int i=0;i<oldts;i++){
            node* head = oldarr[i];
            while(head){
                insert(head->name,head->value);
                head=head->next;
            }
        }
        delete []oldarr;


    }

    public:
    hashmap(int s=7){
        arr = new node*[s];
        ts=s;
        cs=0;
        for(int i=0;i<s;i++){
            arr[i]=NULL;
        }

    }

    void insert(string key, int data){
        int i=hashfn(key);
        node* n=new node(key,data);
        n->next=arr[i];
        arr[i]=n;
        cs++;

        if(cs/(1.0*ts)  > 0.6){
            reHash();
        }

    }

    node* search(string key){
        int i=hashfn(key);
        node*head= arr[i];
        while(head){
            if(head->name==key){
               return head;
                break;
            }
            head=head->next;
        }
        if(head==NULL){
            cout<<"not exist";
        }
        return NULL;
    }
    void print(){
        for(int i=0;i<ts;i++){
            node* head= arr[i];
            while(head){
                cout<<head->name<<"-->"<<head->value;
                head=head->next;
                 cout<<endl;
            }

        }
    }

    int& operator[](string key){
        node* ans=search(key);
        if(ans){
            return ans->value;
        }
        else{
            int agrbagevalue;
            insert(key,agrbagevalue);
            ans= search(key);
            return ans->value;
        }

    }

    void Delete(string key){
        int i=hashfn(key);
        node* head= arr[i];
        node* trail=NULL;

            while(head){

                    //first
                    if(head->name==key && trail==NULL){
                        arr[i]=NULL;
                        delete head;
                    }
                    //in end
                    if(head->name==key && trail!=NULL && head->next==NULL){
                        trail->next=NULL;
                        delete head;
                    }
                    //mid
                    if(head->name==key && head->next!=NULL){
                        node*ptr =head;
                        head=head->next;
                        delete ptr;
                        arr[i]=head;

                    }
                    trail =head;
                    head=head->next;
            }
        return;
    }

};

int main(){
    hashmap h;
    h.insert("Alphonso_Mango",100);
    h.insert("Kiwi",150);
    h.insert("Banana",200);
    h.insert("WaterMelon",180);
    h.print();

    node* x=  h.search("Kiwi");
    cout<<"SEARCH RESULT"<<endl;
    cout<<x->name<<" "<<x->value;

}
```

The complete execution of the above code can be found here.

=======

## Output



```
Banana-->200

WaterMelon-->180

Kiwi-->150

Alphonso_Mango-->100

SEARCH RESULT

Kiwi 150
```

- In the output, you can see we have inserted different key-value pairs into the custom hash table build by us. Try to experiment with these values and you will find collisions also and rehashing too if there are many collisions.
- Using the search function, we can find the value of the key in O(1) time complexity.

## Conclusion

With the help of hash tables, we can insert, search and delete in O(1) time which is a great achievement. Hash tables are widely used for making our code more efficient and faster. Here are some problems of data structures you can try and think the approach using hash tables.

---