BTCOC303

DATA-STRUCTURES

UNIT-I INTRODUCTION

Data, Data structure, Abstract data types, representation of Information, characteristics of Algorithms, Program, Analysing program, Arrays & Hash Table concept of sequential organization, linear & non-linear data structure, storage representation, array processing sparse matrices, transpose of sparse matrices, Hash tables, Direct address tables, Hash table, Hash functions Open Addressing, Perfect Hashing.

★ Data :-

Data is defined as a collection of individual facts or statistics. (While "datum" is technically the singular form of "data" it is not commonly used in everyday language.) Data can come in the form of text, observations, figures, Images, numbers, graphs, or symbols. For example, data might include individual prices, weights, addresses, ages, names, temperatures, dates, or distances.

Data is a raw form of knowledge & on its own, doesn't carry any significance or

purpose of other words, you have to interpret data for it to have meaning. Data can be simple and may even seem useless until it analysed, organised, & Interpreted.

★ Data Types:-

A data type is the most basic and the most common classification of data. It is this through which the compiler gets to know the form or the type of information that will be used throughout the code. So basically, data type is a type of information transmitted between the programmer and the compiler where the programmer informs the compiler about what type of data is to be stored and also tells how much space it requires in the memory. Some basic examples are int, String etc. It is the type of any variable used in the code.

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

Void main()
 {
    int a;
    a=5;

    float b;
    b = 5.0;

    char c;
    c = 'A';

    char d[10];
    d = "example";
 }
```

As seen from the explained above we come to know that in the above code, the variable 'a' is of data type integer which is denoted by int a. So the variable 'a' will be used as an integer type variable throughout the processes of the code. And, in the same way, the variables 'b', 'c' and 'd' are of type float, character and string respectively. And all these are kinds of data types.

* Data Structure :-

→ Data structure is a representation of the logical relationship existing between individual elements of data.
→ Data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.
→ We can also define data structure as a mathematical or logical model of a particular organization of data items.
→ The representation of particular data structure in the main memory of a computer is called a storage structure.
→ The storage structure representation in auxilary memory is called as file structure.
→ It is defined as the way of storing and manipulating data in organized form so that it can be used efficiently.
→ Data structure mainly specifies the following four things:-
  - Organization of Data.
  - Accessing Methods.
  - Degree of Associativity.
  - processing alternatives for information.

→ Algorithm + Data structure = Program.

→ Data structure study covers the following points:-

- Amount of memory requires to store.
- Amount of time requires to process.
- Representation of data in memory.
- Operations performed on that data.

A data structure is a collection of different forms and different types of data that has a set of specific operations that can be performed. It is a collection of data types. It is a way of organizing the items in terms of memory, and also the way of accessing each item through some defined logic. Some examples of data structures are stack, queues, linked lists, binary trees and many more.

Data structures performs some special operations only like insertion, deletion and traversal. For example you have to store data for many employees where each employee has his name, employee id and mobile number. So this kind of ~~information~~ data requires complex data management. Which means it requires data structure comprised of multiple primitive data types. So, data structures are one of the most important aspect when implementing coding concepts in real-world applications.

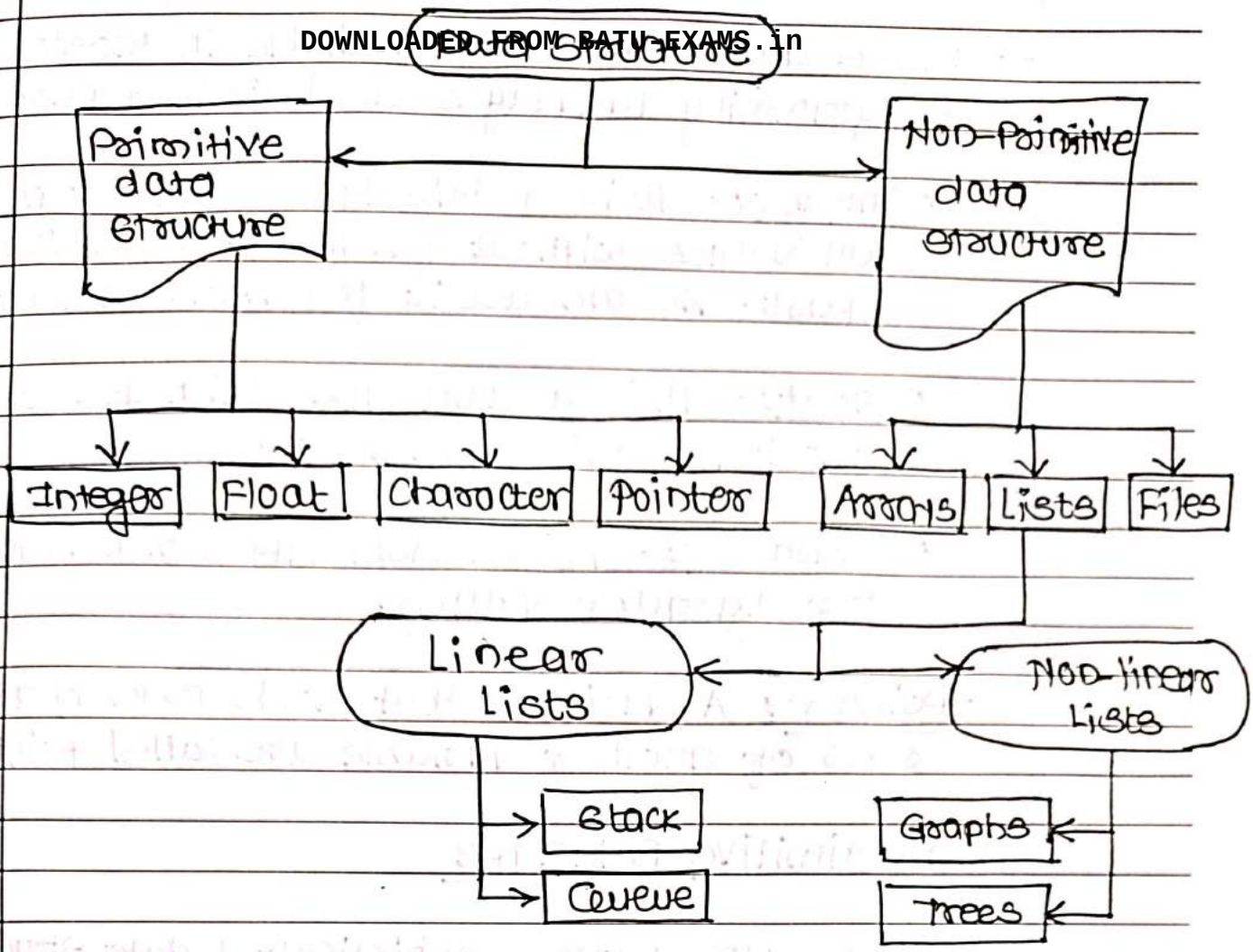


figure:- Classification of data structures.

Data structures are normally classified into two categories.

1. Primitive Data structure.
2. Non-Primitive data structure.

Data Types

A particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.

Primitive Data Structure

- Primitive data structure are basic structures and are directly operated upon by machine instructions.
- Primitive data structure have different representations on different computers.
- Integers, floats, character & pointers are examples of primitive data structures.

- These data types are available in most programming languages as built in types.

→ Integer:- It is a data type which allows all values without ~~function part~~ fraction part. We can use it for whole number.

→ Float:- It is a data type which use for storing fractional numbers.

→ Character:- It is a data type which is used for character values.

- Pointer: A variable that holds memory address of another variable are called pointer.

Non-primitive Data Type

- These are more sophisticated data structure.
- These are derived from primitive data structure.
- The non-primitive data structure emphasize on structuring of a group of homogeneous or heterogeneous data items.
- Examples of non-primitive data types are Array, List, and File etc.
- A Non-primitive data types is further divided into Linear and Non-Linear data structure.

→ Array:- an Array is fixed-sized sequence collection of elements of the same data types.

→ List:- An ordered set containing variable number of elements is called as Lists.

→ File:- A file is collection of logically related information. It can be viewed as a large list of records consisting of various fields.

Linear Data Structures

- A data structure is said to be linear, if its elements are connected in linear fashion by means of logically or in sequence memory locations.
- There are two ways to represent a linear data structure in memory.
  - → static memory allocation.
  - → Dynamic memory allocation.
- The possible operations on the linear data structure are: Traversal, Insertion, Deletion, Searching, Sorting and Merging.

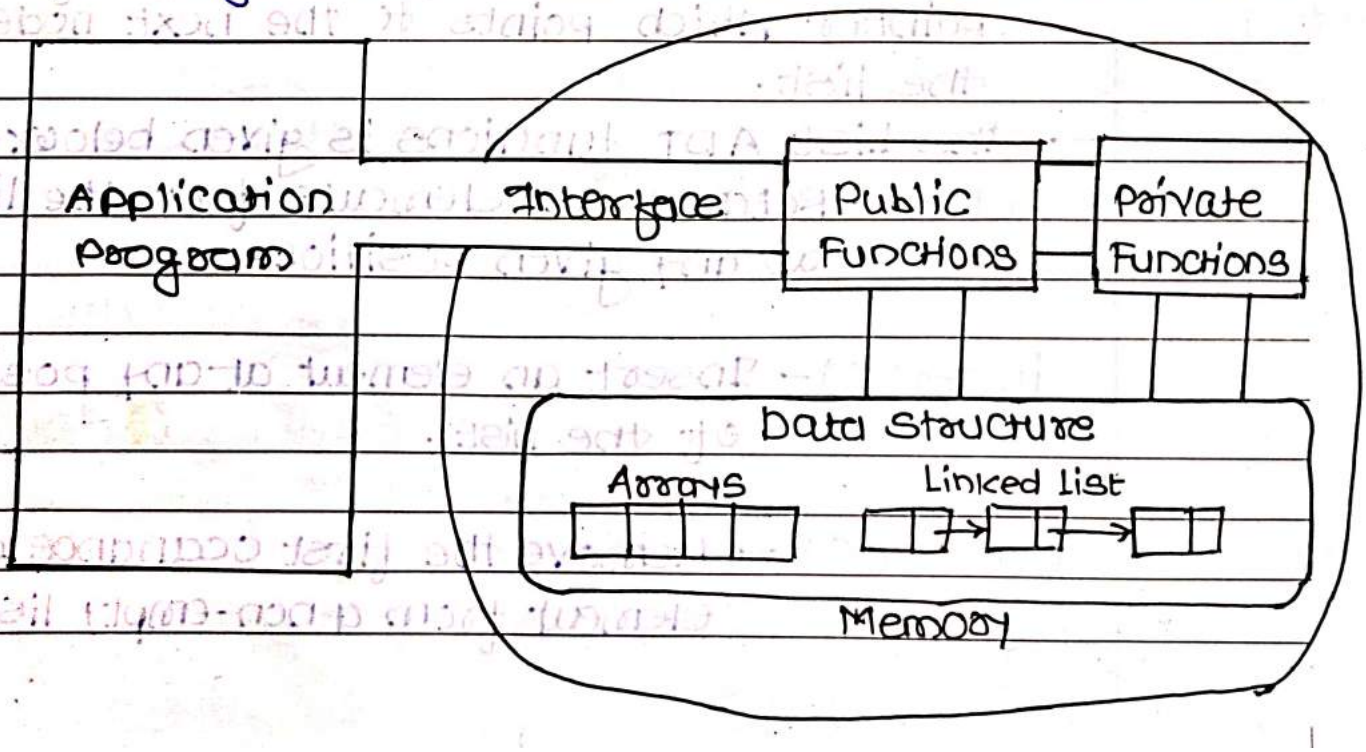
* Abstract Data Types :- (ADT)

- Abstract Data Type (ADT) is a type (or class) whose behaviour is defined by a set of values and a set of operations.

- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.

- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation - independent view.

- The process of providing only essentials and hiding the details is known as abstraction.



- The user of data type does not need to know how that data type is implemented. for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.

- So a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type. Now We'll define three ADTs namely List ADT, Stack ADT, Queue ADT.

1. List ADT :-

10	20	30	40	50	60

- The data is generally stored in key sequence in a list which has a head structure consisting of count, pointers, and address of compare function needed to compare the data in the list.
- The data node contains the pointers to a data structure and self-referential pointer which points to the next node in the list.
- The List ADT functions is given below:-

get() - Returns an elements from the list at any given position.

insert() - Insert an element at any position of the list.

remove() - Remove the first occurance of any element from a non-empty list.

removeAt() - Remove the element at a specified location from a non-empty list.

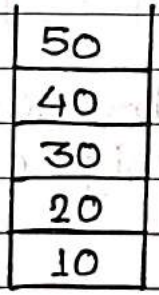
replace() - Replace an element at any position by another element.

size() - Return the number of elements in the list.

isEmpty() - Return true if the list is empty, otherwise return false.

isFull() - Return true if the list is full, otherwise return false.

2. **Stack ADT :-**



- In stack ADT implementation instead of data being stored in each node, the pointer to data is stored.
- The program allocates memory for the data and address is passed to the stack ADT.
- The head node and the data nodes are encapsulated in the ADT. The calling functions can only see the pointer to the stack.
- The stack head structure also contains a pointer to top and count of number of entries currently in stack.

push() - Insert an element at one end of the stack called top.

pop() - Remove and return the element at the top of stack, if it is not empty.

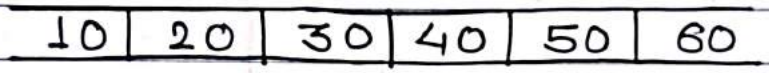
peek() — Return the element at the top of the stack without removing it, if the stack is not empty.

size() — Return number of elements in the stack.

isEmpty() — Return true if the stack is empty, otherwise return false.

isFull() — Return true if the stack is full, otherwise return false.

3. Queue ADT :-



— The queue abstract data type (ADT) follows the basic design of the stack abstract data type.

— Each node contains a void pointers to the data and the link pointer to the next element in the .queue. The program's responsibility is to allocate memory for storing the data.

enqueue() - Insert an element at the end of the queue.

dequeue() — Remove and return the first element of the queue, if the queue is not empty.

peek() — Return the element of the queue without removing it, if the queue is not empty.

size() — Returns the no. of elements in queue.

isEmpty() — Returns true if the queue is empty, Otherwise return false.

isFull() — Return true if the queue is Full, otherwise return false.

* Features of ADT :-

→ Abstract data types (ADT) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include :-

- **Abstraction** - The user does not need to know the implementation of the data structure only essentials are provided.

- **Better Conceptualization** - ADT gives us a better conceptualization of the real world.

- **Robust** - The program is robust and has the ability to catch errors.

- **Encapsulation** - ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance & modification of the data structure.

- **Data Abstraction** - ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.

- **Data Structure Independence** - ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.

- **Information Hiding** - ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors & misuse of the data.

- Modularity - ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

Overall, ADTs provide a powerful tool for organizing and manipulating data in structured and efficient manner.

Abstract data types (ADTs) have several advantages and disadvantages that should be considered when deciding to use them in software development.

* Advantages of ADT :-

- Encapsulation - ADTs provide a way to encapsulate data and operations into a single unit, making it easier to manage and modify the data structure.
- Abstraction - ADTs allow users to work with data structures without having to know the implementation details, which can simplify programming and reduce errors.
- Data structure Independence - ADTs can be implemented using different data structures, which can make it easier to adapt to changing needs and requirements.
- Information Hiding - ADTs can protect the integrity of data by controlling access and ~~unauthorized~~ preventing unauthorized modifications.
- Modularity - ADTs can be combined with other ADTs to form more complex data structures, which can increase flexibility & modularity in programming.

* Disadvantages of ADT :-

- **Overhead** - Implementing ADTs can add overhead in terms of memory and processing, which can affect performance.
- **Complexity** - ADTs can be complex to implement, especially for large and complex data structures.
- **Learning Curve** - Using ADTs requires knowledge of their implementation and usage, which can take time and effort to learn.
- **Limited Flexibility** - Some ADTs may be limited in their functionality or may not be suitable for all types of data structures.
- **Cost** :- Implementing ADTs may require additional resources and investment, which can increase the cost of development.

* What is an Algorithm ?

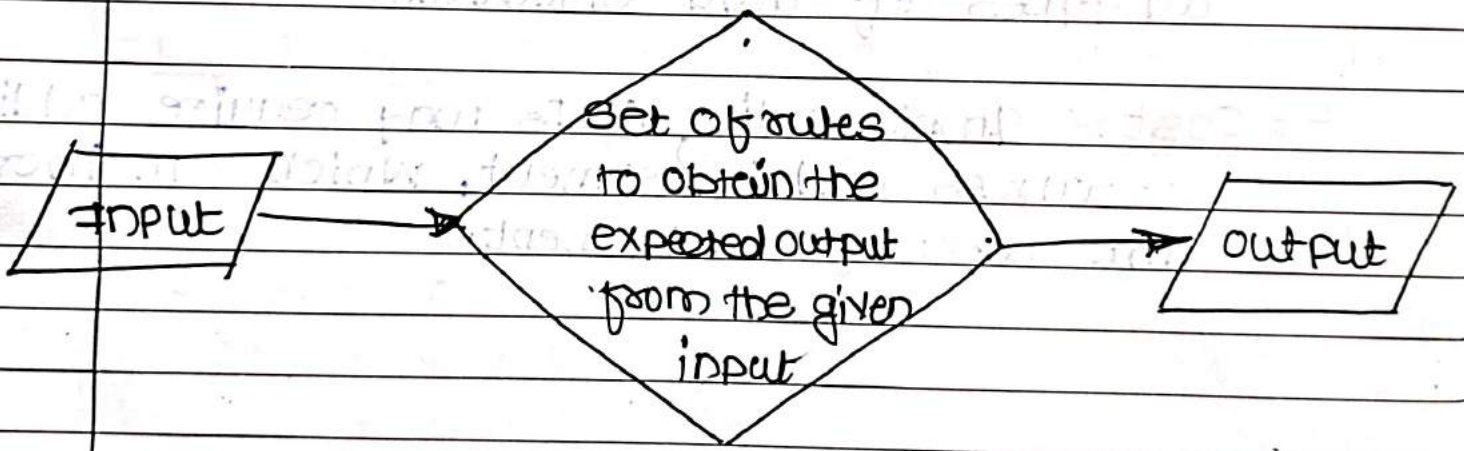
— The word Algorithm means "A set of finite rules or instructions to be followed in calculations or other problem solving operations."

or

"A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations."

Therefore, Algorithm refers to a sequence of finite steps to solve a particular problem.

Algorithm can be simple and complex depending on what you want to achieve.



Algorithm

* Characteristics of Algorithm :-

— Well-defined Inputs.
— Well-defined outputs.
— Clear and Unambiguous
— Finite-steps
— Language Independent
— Feasible.

① Well-defined Inputs – If an algorithm says to take inputs, it should be well-defined inputs. It may or may not take inputs.

② Well-defined outputs – The algorithm must clearly define what output will be yeilded and it should be well-defined as well. It should take at least 1 output.

③ Clear and Unambiguous – The algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.

④ Finiteness – The algorithm must be finite, i.e. it should terminate after a finite time.

⑤ Feasible – The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything.

⑥ Language Independent – The algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

* Properties of Algorithm:-

- It should terminate after a finite time.
- It should produce at least one output.
- It should take zero or more ~~output~~ input.
- It should be deterministic means giving the same output for the same input case.

— Every step in the algorithm must be effective i.e. every step should do some work.

✱ Advantages of Algorithms:-

— It is easy to understand.
— An algorithm is step-wise representation of a solution to a given problem.
— In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

✱ Disadvantages of Algorithm:-

— Writing an algorithm takes a long time so it is time-consuming.
— Understanding complex logic through algorithms can be very difficult.
— Branching and Looping statements are difficult to show in Algorithms (imp).

✱ How to Design an Algorithm?

In order to write an algorithm, the following things are needed as a pre-requisite:

1. The problem that is to be solved by this algorithm i.e. clear problem definition.
2. The constraints of the problem must be considered while solving the problem.
3. The input to be taken to solve the problem.
4. The output to be expected when the problem is solved.
5. The solution to this problem, is within the given constraints.

Then the algorithm is written with the help of the parameters such that it solves the problem.

★ Example:- Consider the example to add three numbers and print the sum.

Step 1: Fulfilling the pre-requisites
As discussed above, in order to write an algorithm, its pre-requisites must be fulfilled.

1. The problem that is to be solved by this algorithm: Add 3 numbers and print their sum.
2. The constraints of the problem that must be considered while solving the problem: The three numbers to be added.
3. The input to be taken to solve the problem: The three numbers to be added.
4. The output to be expected when the problem is solved: The sum of the three numbers taken as the input i.e. single integer value.
5. The solution to this problem, in the given constraints: The solution consist of adding the 3 numbers. It can be done with the help of '+' operator, or bit-wise, or any other method.

Step 2: Designing the algorithm
Now let's design the algorithm with the help of the above pre-requities:
Algorithm to add 3 numbers and print their sum:

1. START
2. Declare 3 integer variables num1, num2, num3.
3. Take the three numbers, to be added, as inputs in variables num1, num2, num3 respectively.
4. Declare an integer variable sum to store the resultant sum of the 3 numbers.
5. Add the 3 numbers and store the result in the variable sum.
6. Print the value of the variable sum.
7. END.

Step3: Testing the algorithm by implementing it

In order to test the algorithm, let's implements it in C language.

```
// C program to add three numbers with the
// help of above designed algorithm.

#include <stdio.h>

int main()
{
  // variables to take the input of the 3 numbers.
  int num1, num2, num3;

  // variable to store the resultant sum.
  int sum;

// Take the 3 numbers as input.
  printf("Enter the 1st number: ");
  scanf("%d", &num1);
  printf("\n", num1);

  printf("\nEnter the 2nd number: ");
  scanf("%d", &num2);
  printf("\n", num2);

  Printf("\n Enter the 3rd number: ");
  scanf("%d", &num3);
  printf("\n", num3);
// calculate the sum using + operator and
// store it in variable sum.
  sum = num1 + num2 + num3;

// Print the sum
  printf("\n Sum of the 3 numbers is: %d", sum);

 return 0;
}
```

output :-

Enter the 1st number: 5

Enter the 2nd number: 7

Enter the 3rd number: 3

sum of the 3 numbers is: 15

* How to analyze an Algorithm?

For a standard algorithm to be good, it must be efficient. Hence the efficiency of an algorithm must be checked and maintained. It can be in two stages:

1. Priory Analysis :- "priori" means "before". Hence priori analysis means checking the algorithm before its implementation. In this, the algorithm is checked when it is written in the form of theoretical steps. This efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation. This is done usually by the algorithm designer. This analysis is independent of the type of hardware and language of the compiler. It gives the approximate answers for the complexity of the program.

2. Posterior Analysis:- "Posterior" means "after". Hence posterior analysis means checking the algorithm after its implementation. In this, the algorithm is checked by implementing it in any programming language and executing it. This analysis helps to get the actual and real analysis report about correctness (for every possible input/s if it shows/returns correct output or not), space required, time consumed etc. That is, it is dependent on the language of the compiler and the type of hardware used.

What is Algorithm complexity and how to find it?
An algorithm is defined as complex based on the amount of space and Time it consumes. Hence the complexity of an algorithm refers to the measures of the time that it will need to execute and get the expected output, and the space it it will need to store all the data (input, temporary data, and output). Hence these two factors define the efficiency of an algorithm.

The two factors of Algorithm complexity are:

- Time Factor - Time is measured by counting the number of key operations such as comparisions in the sorting algorithm.
- Space Factor - Space is measured by counting the maximum memory space required by the algorithm to run/execute.

Therefore, the complexity of an algorithm can be divided into two types:-

1. Space Complexity :-

The space complexity of an algorithm refers to the amount of memory required by the algorithm to store the variables and get the result. This can be for inputs, temporary operations, or outputs.

How to calculate Space Complexity?

The space complexity of an algorithm is calculated by determining the following 2 components:

- Fixed Part - This refers to the space that is definitely required by the algorithm. For example, input variables, output variables, program size etc.
- Variable Part - This refers to the space that can be different based on the implementation of the algorithm. For example, temporary variables, dynamic memory allocation, recursion stack space, etc.

Therefore Space Complexity $S(P)$ of any algorithm P is $S(P) = C + SP(I)$, where C is the fixed part and $S(I)$ is the variable part of the algorithm, which depends on instance characteristic I.

2. Time Complexity :-

The time complexity of an algorithm refers to the amount of time that is required by the algorithm to execute and get the result. This can be for normal operations, conditional if-else statements, loop statements, etc.

How to calculate time complexity?

The time complexity of an algorithm is also calculated by determining the 2 components:

- Constant time part – Any instruction that is executed just once comes in this part. For example, input, output, if-else, switch, arithmatic operations etc.

- Variable time part – Any instruction that is executed more than once, say n times, comes in this part. For example, loops, recursion, etc.

Therefor time complexity T(P) of any algorithm P is T(P) = C + TP(I), where C is the Constant time part and TP(I) is the variable part of the algorithm, which depends on the instance characteristic I.

★ How to express an Algorithm?

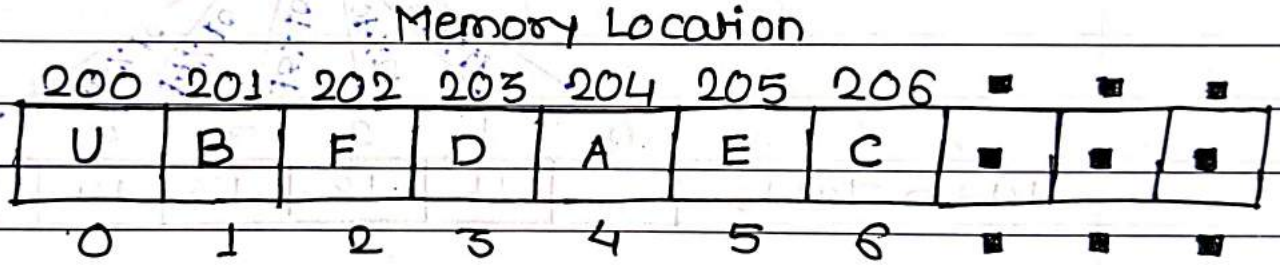
1. Natural Language – Here we express the algorithm in natural language English language. It is too hard to understand the algorithm from it.

2. Flow chart – Here we express the Algorithm by making graphical/pictorial representation of it. It is easier to understand than Natural Language.

3. Pseudo Code – Here we express the Algorithm in the form of annotations and informative text written in plain English which is very much similar to real code but as it has no syntax like any of the programming language, it can't be compiled or interpreted by the computer. It is the best way to express an algorithm because it can be understood by even a layman with some school level programming knowledge.

* Array Data Structure :-

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This make it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the ~~name~~ name of array).



The above image can be looked as top-level view of a staircase where you are at the base of the staircase. Each element can be uniquely identified by their index in the array (in a similar way you could identify your friends by the step on which they were on in the above example).

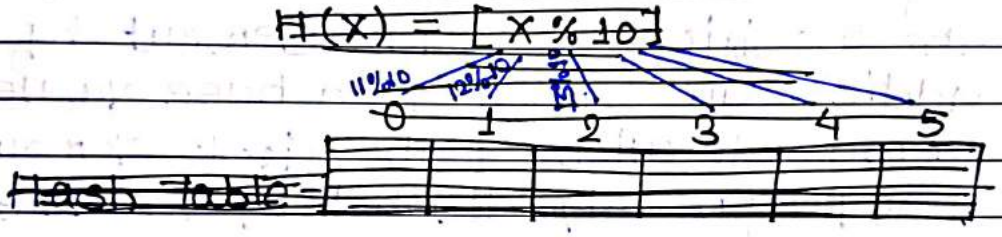
* Hashing Data Structure

— Hashing is a technique or process of mapping key, 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 hash function used.

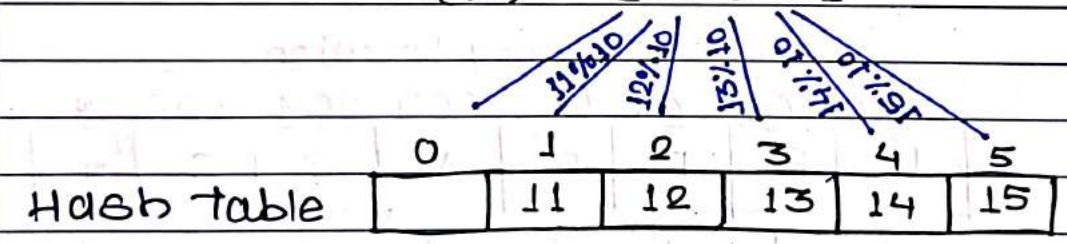
— Let a hash function H(x) maps the value X at the index X%10 in an Array. For example, if the list of value is [11, 12, 13, 14, 15] it will be stored at positions [1, 2, 3, 4, 5] in the array or Hash table respectively.

List = [11, 12, 13, 14, 15]

H(X) = [X % 10]



H(X) = [X % 10]



* Hash Table :-

— Hash table is one of the most important data structure that uses a special function known as a hash function that maps a given value with a key to access the element faster.

— A Hash table is a data structure that store some information, and the information has basically two main components, i.e. key and value. The hash table can be implemented with

the help of an associative array. The efficiency of mapping depends upon the efficiency of the hash function used for mapping.

- For example, suppose the key value is John and the value is the phone number, so when we pass the key value in the hash function shown as below:

```
Hash(key) = index;
```

When we pass the key in the hash function, then it gives the index.

```
Hash(John) = 3;
```

The above example adds the John at the index 3.

Drawback of Hash function

A Hash function assigns each value with unique key. Sometimes hash table uses an imperfect hash function that causes a collision because the hash function generates the same key of two different values.

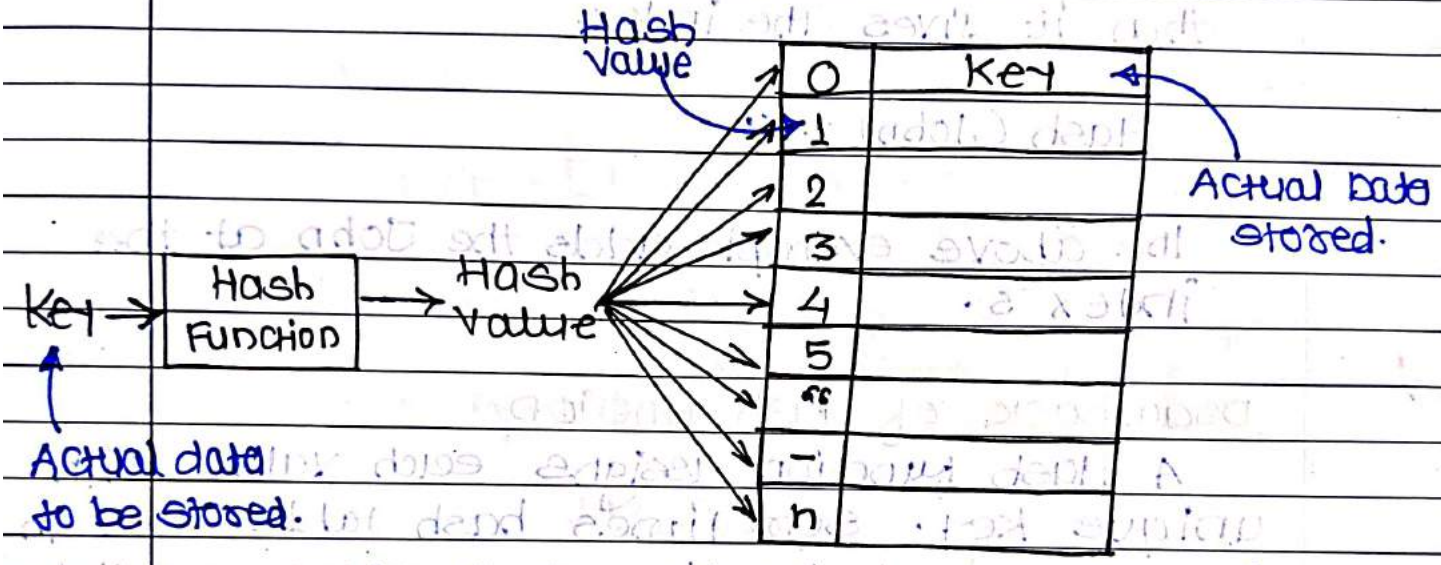
* Hashing:-

Hashing is one of the searching techniques that uses a constant time. The time complexity in hashing is O(1). Till now, we read the two techniques for searching linear search and binary search. The worst time complexity in linear search is O(n), and O(log n) in binary search. In both the searching techniques, searching depends upon the number of elements but we want the technique that takes a constant time. So, hashing technique

time that provides constant time.

In Hashing technique, the hash table and hash function are used. Using the hash function, we can calculate the address of at which the value can be ~~stored~~ stored.

The main idea behind the hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value could be stored. It can be written as:



There are three ways of calculating the hash function:

- Division method
- Folding method
- Mid Square method

In the division method, the hash function can be defined as:

$$h(K_i) = K_i \% m;$$

where m is size of the hash table.

For example, if the key value is 6 and the size of the hash table is 10, When we apply the hash function to key 6 then index would be

$$h(6) = 6 \% 10 = 6$$

The index is 6 at which the value is stored.

* STORAGE REPRESENTATION *

Data structure is the way of storing data in computer's memory so that it can be used easily and efficiently. There are different data-structures used for the storage of data. It can also be defined as a mathematical or logical model of a particular organization of data items. The representation of particular data structure in the main memory of a computer is called as storage structure. For Example: Array, stack, Queue, tree, Graph etc.

* Operations on different Data structure:-

There are different types of operations that can be performed for the manipulation of data in every data structure. Some operations are below:-

(1) Traversing:- Traversing a data structure means to visit the element stored in it. It visits data in a systematic manner. This can be done with any type of DS.

// C program to traverse the array

```
#include <stdio.h>

// Function to traverse and print the array
void printArray (int* arr, int n)
{
    int i;
    printf ("Array: ");
    for (i=0; i<n; i++)
    {
        printf ("%d", arr[i]);
    }
    printf ("\n");
}

// Driver Program
int main ()
{
    int arr[] = {2, -1, 5, 6, 0, -3};
    int n = sizeof (arr) / sizeof (arr[0]);
    printArray (arr, n);
    return 0;
}
```

Output:-

2, -1, 5, 6, 0, -3

<2> Searching:-

Searching means to find a particular element in the given data-structure. It is considered as successful when the required element is found. Searching is the operation which we can performed on data-structures like array, linked-list, tree, graph etc.

```
#include<stdio.h>
#include<conio.h>

int main()
{
  int a[10000], i, n, key;

  printf("\n Enter the Size of array: ");
  scanf("%d", &n);
  printf("\n Enter the elements in array: ");
  for(i=0; i<n; i++)
  {
     scanf("%d", &a[i]);
  }
  printf("Enter the key: ");
  scanf("%d", &key);

  for(i=0; i<n; i++)
  {
    if(a[i]==key)
    {
       printf("element found");
       return(0); 0;
    }
  }
  printf("element not found");
}
```

Output:-

```
Enter Size of the array: 5
Enter elements in array: 4
6
2
1
3
Enter the key: 2
element found.
```

(3) Insertion :-

It is the operation which we apply on all the data-structures. Insertion means to add an elements in the given data structure. The operation of insertion is successful when the required element is added to the required data-structure. It is unsuccessful in some cases when the size of the data structure is full and when there is no space in the data-structure to add any additional element. The insertion has the same name as an insertion in the data structure as an array, linked-list, graph, tree. In stack, this operation is called Push. In the queue, this operation is called Enqueue.

```
#include<stdio.h>
int main(){
    int arr[40], pos, i, size, value;
    printf("\n Enter no. of elements in an array:");
    scanf("%d", &size);
    printf("\n Enter %d elements are:", size);
    for (i=0; i< size; i++)
        scanf("%d", &arr[i]);
    printf("\n
```

```
#include<stdio.h>
#include<conio.h>
int main()
{
  int arr[10], i, element;
  printf("\n Enter 5 array elements:");
  for (i=0; i<5; i++)
     scanf("%d", &arr[i]);
  printf("\n Enter elements to insert: ");
  scanf("%d", &element);
  arr[i] = element;
  printf("\n The New Array is: \n")
  for (i=0; i<6; i++)
```

```
        printf ("%d ", arr[i]);
    getch();
    return 0;
}
```

(4) Deletion :-

- It is the operation which we apply on all the data-structure. Deletion means to delete an element in the given data structure. The operation of deletion is successful when the required element is deleted from the data structure. The deletion has the same name as a deletion in the data-structure as an array, linked-list, graph, tree, etc. In stack, this operation is called pop, In queue, this operation is called Dequeue.

```
#include<stdio.h>
#include<conio.h>

int main()
{
 // declaration of the int type variable
 int arr[50];
 int pos,i, num // declare int type variable.

 printf("\n Enter the no.of elements in an array: \n");
 scanf ("%d", &num);

 printf("\n Enter %d elements in array: \n", num);

// use for loop to insert elements one by one in array
 for (i=0; i<num; i++)
  {
    printf ("arr [%d]= ", i);
    scanf( "%d", &arr[i]);
  }

// enter the position of the element to be deleted
 printf("Define the position of the array element
          where you want to delete: \n");
 scanf( "%d", &pos);
// check whether the deletion is possible or not
 if (pos >= num+1)
  {
    printf("\n Deletion is not possible in the array.");
  }
 else
  {
// use for loop to delete the element & update the index
    for (i=pos-1; i<num-1; i++)
     {
```

```
        arr[i] = arr[i+1];//assign arr[i+1] to arr[i]
    }

    Printf("In the resultant array is: In");

    //display the final array
    for( i=0; i<num-1; i++)
    {
        printf ("arr[%d]=", i);
        Printf ("%d In", arr[i]);
    }
}

return 0;
}
```

Output:-

```
Enter the no. of Elements in an array: 8

Enter 8 elements in array:
arr[0] = 3
arr[1] = 6
arr[2] = 42
arr[3] = 15
arr[4] = 10
arr[5] = 5
arr[6] = 8
arr[7] = 12

Define the position of the array element where
you want to delete: 5

The resultant array is:
arr[0] = 3
arr[1] = 6
arr[2] = 2
```

arr[3] = 15
arr[4] = 5
arr[5] = 8
arr[6] = 12

☆ Some other Methods:-

① Create:-

It reserves memory for program elements by declaring them. The creation of data structure can be done during
1. Compile-time
2. Run-time
You can use malloc function.

② Selection:-

It select specific data from present data. You can any select specific data by giving condition in loop.

③ Update:-

It updates the data in the data structure. You can also update any specific data by giving some condition in loop like selection approach.

④ Sort:-

Sorting data in a particular order (ascending or descending).
We can take the help of many sorting algorithms to sort data in less time. Example: bubble sort which takes O(n^2) time to sort data. There are many algorithms present like merge sort, insertion sort, selection sort, quick sort etc.

⑤ Merge:-

Merging data of two different orders in a specific order may ascend or descend.

We use merge sort to merge sort data.

⑥ Split Data :-
Dividing data into different sub-parts to make the process complete in less time.

* Array Processing. Sparse Matrices :-

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have 0 value, then it is called a sparse matrix.

Why to use sparse matrix instead of simple Matrix?

— Storage - There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.

— Computing Time - Computing time can be saved by logically designing a data structure traversing only non-zero elements.

Example :-

0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0

Representing a sparse matrix by 2D arrays leads to wastage of lots of memory as zeros in matrix are of no use in most of cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with triples - (Row, Column, value).

Sparse Matrix Representations can be done in many ways following are two common representations:-

1. Array Representation
2. Linked-list Representation

Using Array -

2D array is used to represent a sparse matrix in which there are three rows named as

- Row:- Index of row, where non-zero elements is located.
- Column:- Index of column, where non-zero elements is located.
- Value:- Value of the non-zero elements located at index - (rows, column)

Row \ Column	0	1	2	3	4
0	0	0	3	0	4
1	0	0	5	7	0
2	0	0	0	0	0
3	0	2	6	0	0

⇒

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Given two sparse matrices, perform operations such as add, multiply or transpose of the matrices in their sparse from itself. The result should consist of three sparse matrices, one obtained by adding the two input matrices, one by multiplying the two matrices and one obtained by transpose of the first matrix.

Example: Note that most entries of matrices will be zero as matrices are sparse.

Input :-

Matrix 1 : (4×4)

Row	Column	Value
1	2	10
1	4	12
3	3	5
4	1	15
4	2	12

Matrix 2 : (4×4)

Row	Column	Value
1	3	8
2	4	23
3	3	9
4	1	20
4	2	25

Output :-

Result of Addition : (4×4)

Row	Column	Value
1	2	10
1	3	8
1	4	12
2	4	23
3	3	14
4	1	35
4	2	37

Result of Multiplication : (4×4)

Row	Column	Value
1	1	240
1	2	300
1	4	230
3	3	45
4	3	120
4	4	276

Result of transpose of the first matrix: (4x4)

Row	Column	Value
1	4	15
2	1	10
2	4	12
3	3	5
4	1	12

The sparse matrix used anywhere in the program is sorted according to its row values. Then elements with the same row values are further sorted according to their column values.

Now to Add the matrices, we simply traverse through both matrices element by element and insert the smaller element (one with smaller row and col value) into the resultant matrix. If we come across an element with the same row and column value, we simply add their values and insert the added data into resultant matrix.

To transpose a matrix, we can simply change every column value to the row value and vice-versa, however, in this case, the resultant matrix won't be sorted as we require. Hence initially determine the number of elements less than the current element's column being inserted in order to get the exact index of the resultant matrix where the current element should be placed. This is done by maintaining an array index[] whose ith value indicates the number of elements in the matrix less than the column i.

To Multiply the matrices, we first calculate transpose of the second matrix to simplify our comparisions and maintain the sorted order. So, the resultant matrix is obtained

by traversing through the entire length of both matices and summing the appropriate multiplied values.

Any row value equal to x in the first matrix and row value equal to y in the second matrix (transposed one) will contribute towards result[x][y]. This is obtained by multiplying all such elements having col value in both matrices and adding only those with the rows as x in first matrix and row as y in the second transposed matrix to get the result[x][y].

For example: Consider 2 matrices:

Row	Col	Val
1	2	10
1	3	12
2	1	1
2	3	2

Row	Col	Val
1	1	2
1	2	5
2	2	1
3	1	8

The resulting matrix after multiplication will be obtained as follows:-

The transpose of second matrix

Row	Col	Val
1	2	10
1	3	12
2	1	1
2	3	2

Row	Col	Val
1	1	2
1	3	8
2	1	5
2	2	1

Summation of multiplied values:-

result [1] [1] = A[1][3] * B[1][3] = 12 * 8 = 96
result [1] [2] = A[1][2] * B[2][2] = 10 * 1 = 10
result [2] [1] = A[2][1] * B[1][1] + A[2][3] * B[3][1] = 2*1 + 2*8 = 18
result [2] [2] = A[2][1] * B[2][2] = 1 * 5 = 5

Any other element can not be obtained by any combination of row in Matrix A and ROW in Matrix B.

Hence the final resultant matrix will be:

Row	Col	value
1	1	96
1	2	10
2	1	18
2	2	5

* Hash Tables

Hash tables are the data structures which favour efficient storage and retrieval of data elements which are linear in nature.

Dictionaries:-

- Dictionaries is a collection of data elements uniquely identified by field called key. A diction directory supports operations of search, insert and delete.
- A ~~directory~~ dictionary supports both sequential and random access is the process in which the data elements of the dictionary are ordered and accessed according to the order of the process in which the data elements of the dictionary are not accessed according to particular order.

- Hash tables are ideal data structures for dictionaries.

Hash Search :-

- Hash selection is a search in which the key, through an algorithmic function, determines the location of the data.
- Hashing it is a key to address transformation in which the key map to addresses in a list.

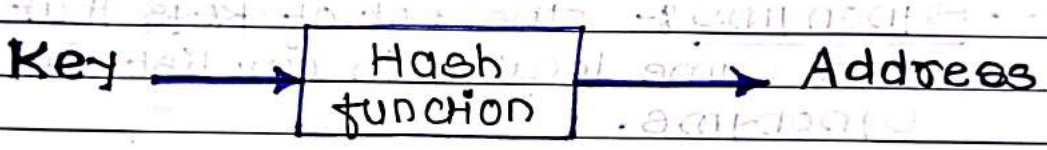


fig:- Hash Search

Hash Function :-

- A hash function is a mathmatical function which maps a given key of the dictionary to its corresponding location in the storage table (Known as hash table).
- The process of mapping the keys to their respective position in hash table is called as Hashing.

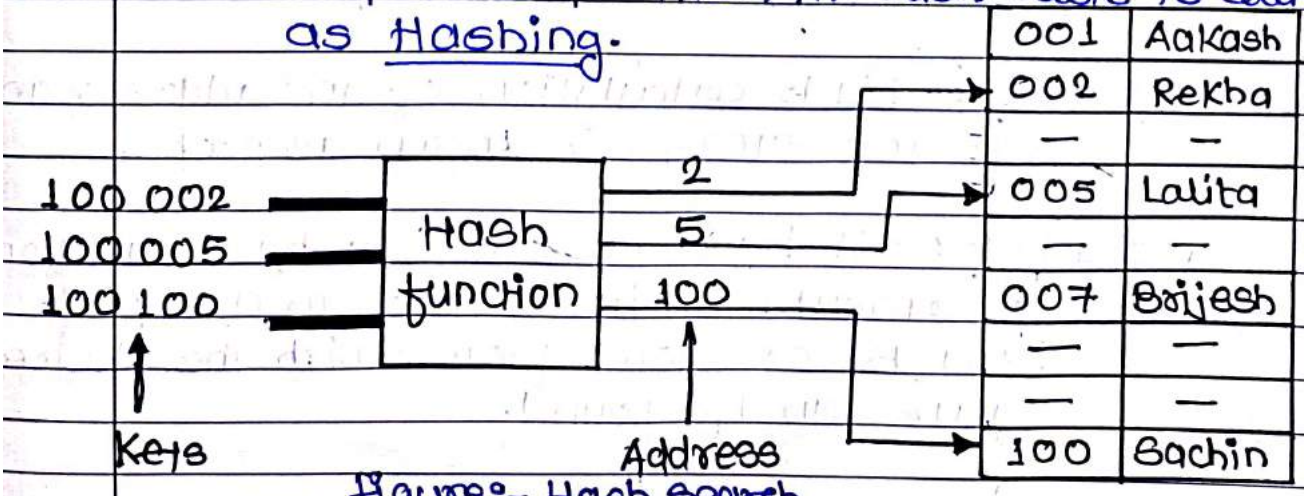


Figure:- Hash Search

- The choice of the hash function plays a significant role in the performance of the hash table. It is therefore essential that a hash function satisfies following characteristics:-

Characteristics of Hash Functions:

- Easy and quick to compute.
- Even distribution of keys across the hash table.
- A hash function must minimize collision.

Basic Definitions of Hashing:

- **Synonyms:-** The set of keys that hash to the same location in our list is called as synonyms.

- **Collision:-** Collision is the event that occurs when a hashing algorithm produce an address for an insertion key and that address is already occupied.

- **Home Address:-** The address produced by the hashing algorithms is known as Home address.

- **Prime Area:-** The memory that contains all of the home addresses is known as the prime area.

- **Probe:-** Each calculation of an address and test for success is known as probe.

- **Bucket:-** A hash table uses a hash function to compute an index into an array of Buckets or slots, from which the desired value can be found.

- **Overflow :-** An overflow occurs when the home bucket for a new pair (key, element) is full.

- **Open Hashing :-** In open hashing, key are stored in linked lists attached to cell of a hash table.

- **Closed Hashing :-** In closed hashing, all keys are stored in linked lists attached to cell of hash table.

- **Load Density / Load Factor :-** The loading density or loading factor of a hash table is

$$a = n/(sb)$$

  - s is number of slots.
  - b is number of buckets.

★ Issues in Hashing :-

Following are some basic issues which are consider while hashing :-

- Computing the hash function.
- **Collision Resolution :-** Algorithm and data structure to handle two keys that hash to the same index.
- **Equality Test :-** Method for checking whether two keys are equal.

* Properties of Good Hashing Function :-

Hash functions should have following properties:
- Fast computation of the hash value (O(1)).
- Hash value should be distributed (nearly) uniformly:
  - → Every hash value (cell in the hash table) has equal probability.
  - → This should hold even if keys are non-uniformly distributed.
- The goal of a hash function is: 'disperse' the key in an apparently random way.
- A hash function must minimize collisions.

* Forms of Hashing Data Structure

<1> Linear Open Addressing :-

- It allows any number of records to be stored, because the space is dynamic.

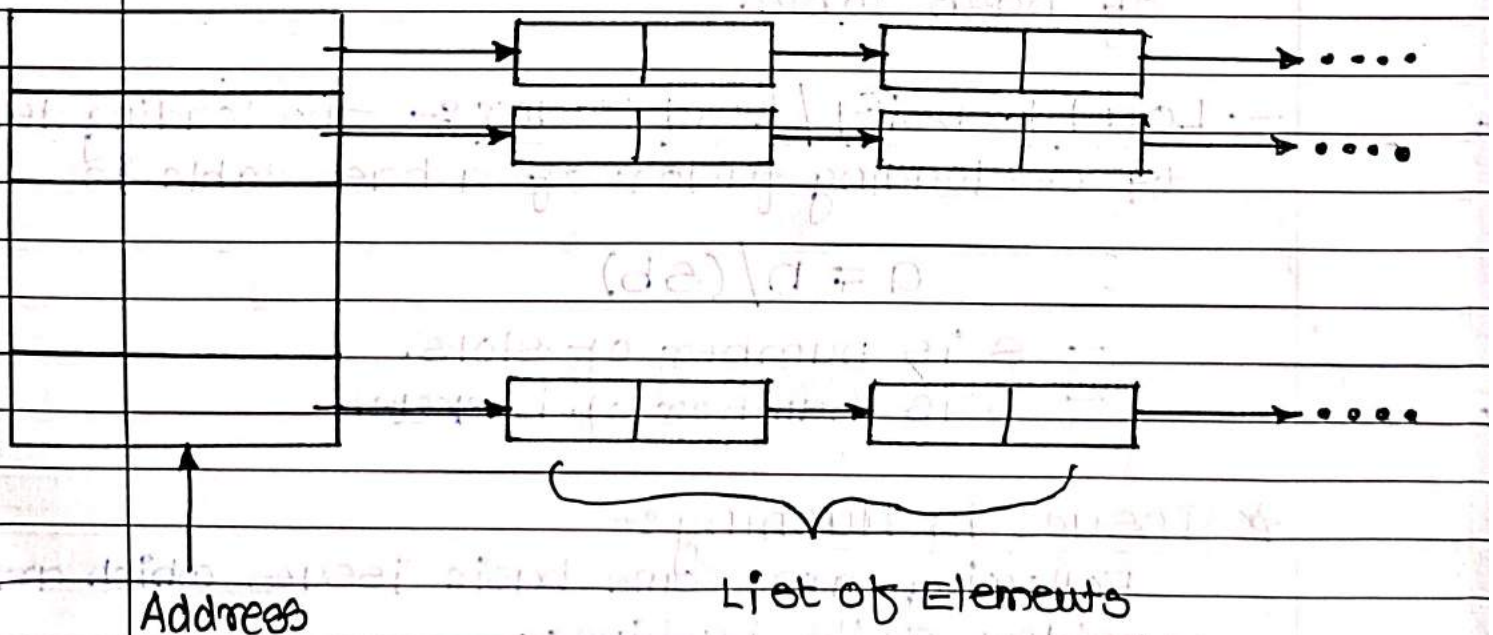


figure:- Linear Open Addressing

<2> Linear Closed Addressing:-

It uses a fixed space for storage and hence this limits the size of hash table.

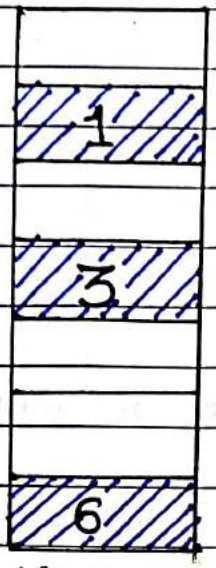


figure:- Linear closed addressing.

In this case maximum 7 elements can be stored as array size is only 7 and that is fixed.

☆ Direct Address Tables

— Direct Address Table is a data structure that has the capability of mapping records to their corresponding key using arrays. In direct address tables, records are placed using their key values directly as indexes. They facilitate fast searching, insertion and deletion operations.

— We can understand the concept using the following example. We create an array of size equal to maximum value plus one (assuming 0 based index) and then use values as indexes. For example, in diagram which is on next page key 21 is used directly as index.

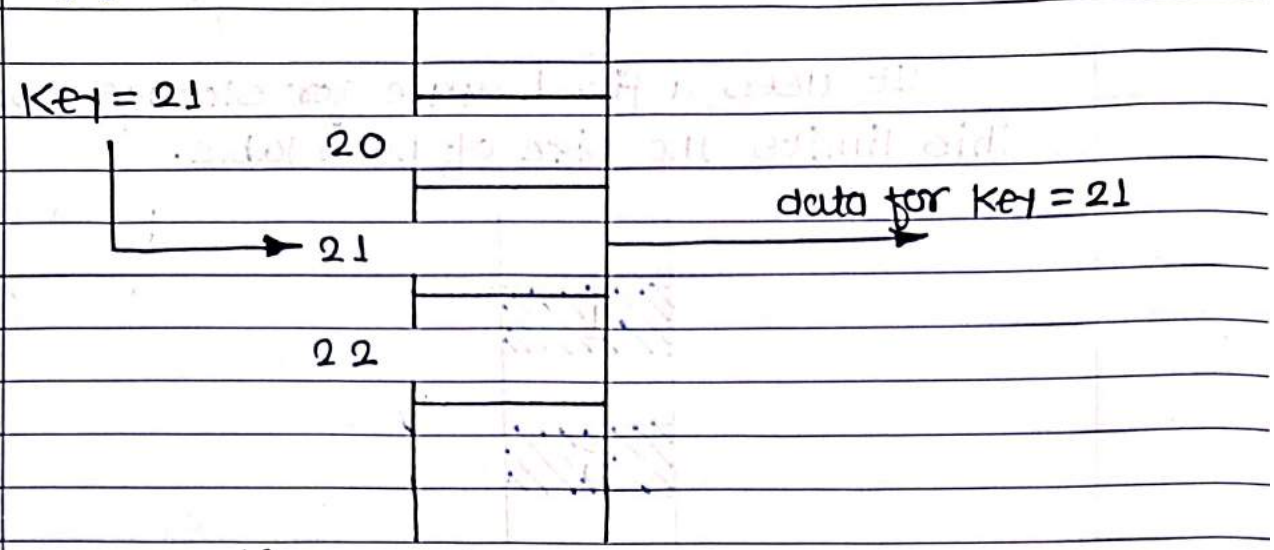


figure:- Direct address table.

Advantages:—

— Searching in O(1) Time: Direct address tables use arrays which are random access data structure, so, the key values (which are also the index of the array) can be easily used to search the records in O(1) time.

— Insertion in O(1) Time: We can easily insert an element in an array in O(1) time. The same thing follows in a direct address table also.

— Deletion in O(1) Time: Deletion of an element takes O(1) time in an array. Similarly, to delete an element in direct address table we need O(1) time.

Limitations:—

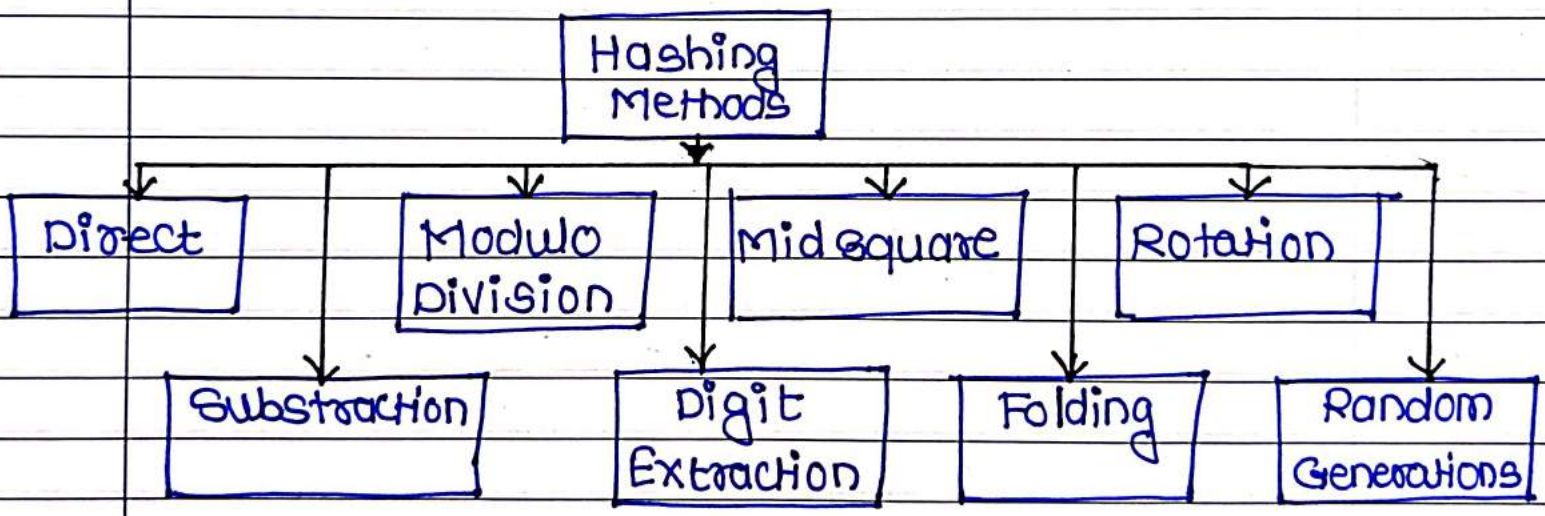
— Prior knowledge of maximum key value.
— Practically useful only if the maximum value is very less.
— It causes wastage of memory space if there is a significant difference between total records and maximum value.

Hashing can overcome these limitations of direct address tables.

How to handle collisions?

Collisions can be handles like hashing. We can either use chaining or open addressing to handle collisions. The only difference from hashing here is, we do not use hash function to find the index. We rather directly use values as indexes.

✱ HASH FUNCTIONS



(1) Direct Hashing:-

- In direct hashing, address for a key is generated without any algorithmic manipulation. Therefore the data structure must contain an address for every possible key.

BATU-EXAM

Made by batuexams.com

at MET Bhujbal Knowledege City