A REPORT OF ONE MONTH TRAINING

on

COMPETITIVE CODING

at

GURU NANAK DEV ENGINEERING COLLEGE, LUDHIANA

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE AWARD OF THE DEGREE OF

BACHELOR OF TECHNOLOGY

(Computer Science and Technology Engineering)

16/07/2021 to 16/08/2021

SUBMITTED BY:

NAME: KANAV DUA

UNIVERSITY ROLL NO.: 1905011

DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY ENGINEERING

GURU NANAK DEV ENGINEERING COLLEGE, LUDHIANA

(An Autonomous college under UGC ACT)

CANDIDATE'S DECLARATION

I “KANAV DUA” hereby declare that I have undertaken one-month training at **“Guru Nanak Dev Engineering College, Ludhiana”** during a period from ……. to ……. in partial fulfilment of requirements for the award of the degree of B.Tech (Computer Science and Technology Engineering) at GURU NANAK DEV ENGINEERING COLLEGE, LUDHIANA. The work which is being presented in the training report submitted to the Department of Electronics and Communication Engineering at GURU NANAK DEV ENGINEERING COLLEGE, LUDHIANA is an authentic record of training work.

Signature of the Student

The one-month industrial training Viva–Voce Examination of_____ has been held on _____ and accepted.

Signature of Internal Examiner Signature of External Examiner

ABSTRACT

During the training, I have been taught about competitive coding in which I learned basic data structures and algorithms and also learned how to implement all these concepts with the help of programming language i.e. C++. Here I have revised basic C++ programming concepts i.e. functions, pointers, OOPS, and reference variables to implement the concept of data structure and algorithm. In the data structure, I have learned about Array, Linked List, Stack, Queues and Hashing. In the algorithm section, I have learned about the analysis and design of algorithms, asymptotic notation, basic recursion, searching and sorting. All these concepts are the backbone to develop any software and building a career in software development in the IT sector.

ACKNOWLEDGEMENT

I am highly grateful to **Dr. Sehijpal Singh,** Principal, Guru Nanak Dev Engineering College (GNDEC), Ludhiana, for providing this opportunity to carry out the one-month training on **Competitive Coding**. The Encouragement received from **Dr. Parminder Singh,** Head, Department of Electronics and Communication Engineering, GNDEC Ludhiana has been of great help in carrying the present work and is acknowledged with reverential thanks.

I humbly acknowledge my gratitude to **T&P Cell** as well as the coordinator **Prof. Kuldeepak Singh** and other faculty members of IT department also for helping me, answering my doubts all along and guiding me. They helped me right from the beginning of to the end in every aspect. I am sincerely grateful to all of them for sharing their truthful and illuminating views on a number of issues related to the training.

Last, but not the least, my thanks are also due to all others who remained behind the scene but whose work has been consulted and referred to in this training.

CONTENTS

CHAPTER – 1

INTRODUCTION TO PROGRAMMING WITH CPP

1. FUNCTIONS

A function in C++ refers to a group of statements that takes input, processes it, and returns an output. The idea behind a function is to combine common tasks that are done repeatedly. If you have different inputs, you will not write the same code again. You will simply call the function with a different set of data called parameters.

Each C++ program has at least one function, the main() function. You can divide your code into different functions. This division should be such that every function does a specific task.

Need of functions - There are numerous benefits associated with the use of functions. These include:

- Each function puts related code together. This makes it easier for programmers to understand code.
- Functions make programming easier by eliminating code repetition.
- Functions facilitate code reuse. You can call the same function to perform a task at different sections of the program or even outside the program.

There are two basic types of function :

- Built in function
- User defined function

Built in function - These functions are part of the compiler package. These are part of standard library made available by the compiler. For example, exit(), sqrt(), pow(), strlen() etc. are library functions (built-in functions). Built-in functions helps you, to use function without declaring and defining it. To use built-in functions in a program, you have to only include that header file where the required function is defined. Here is an example. This program uses built-in function named strlen() of string.h header file to find the length of the string.

```
#include<iostream.h>
```

```
#include<string.h>
#include<conio.h>
void main()
{
  clrscr();
  char str[80];
  cout<<"Enter any string (line):\n";
  cin.getline(str, 80);
  int len = strlen(str);
  cout<<"\nLength of the string is: "<<len;
  getch();
}
```

User defined function -The user-defined functions are created by you i.e., the programmer. These functions are created as per requirements of your program. Here is an example, demonstrating user-defined functions in C++.

```
#include<iostream.h>
#include<conio.h>
void start_msg(void);    // function declaration
void main()
{
  clrscr();
  start_msg();     // function calling
  int num;
  cout<<"Enter a number: ";
```

```
   cin>>num;
   cout<<"You entered: "<<num;
   getch();
}

void start_msg(void)   // function definition
{
   cout<<"Welcome to gndec ludhiana\n";
   cout<<"This is C++ Function Types Tutorial\n\n";
}
```

Function Definition - The general form of a C++ function definition is as follows −

```
return_type function_name( parameter list ) {
   body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

- **Return Type** − A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.
- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

2. POINTERS

Pointers are symbolic representations of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. Iterating over elements in arrays or other data structures is one of the main use of pointers.

The address of the variable you're working with is assigned to the pointer variable that points to the same data type (such as an int or string).

Syntax:

datatype *var_name;

int *ptr; // ptr can point to an address which holds int data

How to use a pointer?

- Define a pointer variable
- Assigning the address of a variable to a pointer using the unary operator (&) which returns the address of that variable.

- Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.

```
#include <bits/stdc++.h>
using namespace std;
void geeks()
{
    int var = 20;

    // declare pointer variable
    int* ptr;

    // note that data type of ptr and var must be same
    ptr = &var;

    // assign the address of a variable to a pointer
    cout << "Value at ptr = " << ptr << "\n";
    cout << "Value at var = " << var << "\n";
    cout << "Value at *ptr = " << *ptr << "\n";
}
// Driver program
int main()
{
  geeks();
  return 0;
```

}

3. OBJECT ORIENTED PROGRAMMING (OOPS)

OOPs, or Object-oriented programming is an approach or a programming pattern where the programs are structured around objects rather than functions and logic. It makes the data partitioned into two memory areas, i.e., data and functions, and helps make the code flexible and modular. Object-oriented programming mainly focuses on objects that are required to be manipulated. In OOPs, it can represent data as objects that have attributes and functions.

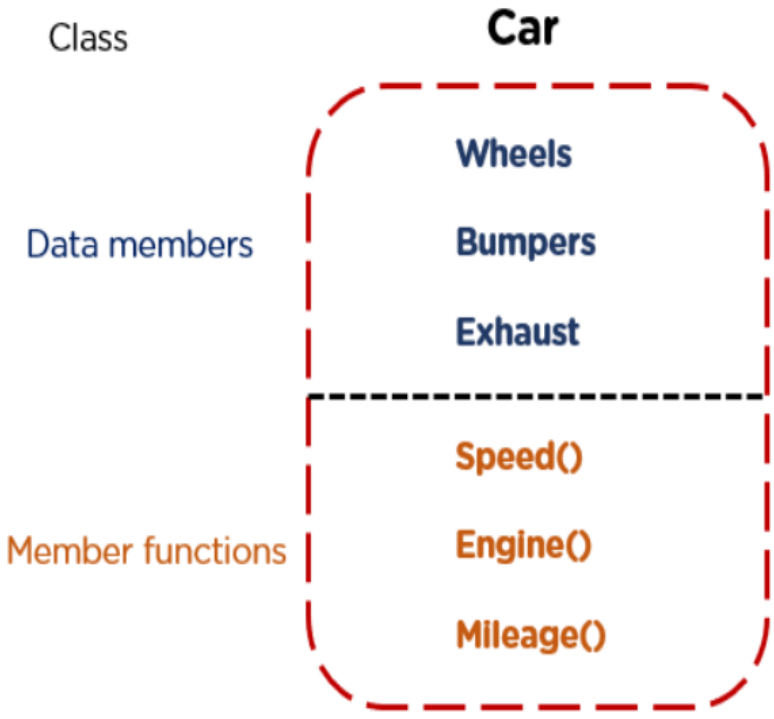
There are some basic concepts that act as the building blocks of OOPs.

- Classes & Objects
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Object -An Object can be defined as an entity that has a state and behavior, or in other words, anything that exists physically in the world is called an object. It can represent a dog, a person, a table, etc.

Classes - Class can be defined as a blueprint of the object. It is basically a collection of objects which act as building blocks.

A class contains data members (variables) and member functions. These member functions are used to manipulate the data members inside the class.



Abstraction - Abstraction helps in the data hiding process. It helps in displaying the essential features without showing the details or the functionality to the user. It avoids unnecessary information or irrelevant details and shows only that specific part which the user wants to see.

Encapsulation -The wrapping up of data and functions together in a single unit is known as encapsulation. It can be achieved by making the data members' scope private and the member function's scope public to access these data members. Encapsulation makes the data non-accessible to the outside world.

Inheritance -Inheritance is the process in which two classes have an is-a relationship among each other and objects of one class acquire properties and features of the other class. The class which inherits the features is known as the child class, and the class whose features it inherited is called the parent class. For example, Class Vehicle is the parent class, and Class Bus, Car, and Bike are child classes.

Polymorphism -Polymorphism means many forms. It is the ability to take more than one form. It is a feature that provides a function or an operator with more than one definition. It can be implemented using function overloading, operator overload, function overriding, virtual function.

4. POINTERS AND REFERENCES

Pointers	References
A variable that stores the address of another variable and it is known as a pointer. It can be dereferenced using the (*) operator in order to access the memory location towards which the pointer points.	A reference variable is really an alias, or another kind of name for a variable that already exists. Similar to a pointer, a reference is similarly created by storing an object's address.
The address of the variable is stored mainly using pointers.	It always refers to an existing variable.
Null values can be assigned to the pointers.	Null values cannot be assigned to the references.

CHAPTER 2

ALGORITHM FUNDAMENTALS AND BASIC OPERATIONS

2.1 Introduction to DS and algorithms

What is Data Structure and Algorithms?

A data structure is defined as a particular way of storing and organizing data in our devices to use the data efficiently and effectively. The main idea behind using data structures is to minimize the time and space complexities. An efficient data structure takes minimum memory space and requires minimum time to execute the data.

Algorithm is defined as a process or set of well-defined instructions that are typically used to solve a particular group of problems or perform a specific type of calculation. To explain in simpler terms, it is a set of operations performed in a step-by-step manner to execute a task.

Applications of Data Structure and Algorithms

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms −

- Search − Algorithm to search an item in a data structure.
- Sort − Algorithm to sort items in a certain order.
- Insert − Algorithm to insert item in a data structure.
- Update − Algorithm to update an existing item in a data structure.
- Delete − Algorithm to delete an existing item from a data structure

2.2 Analyze and design algorithms

What is meant by Algorithm Analysis?

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis −

- **Worst-case** − The maximum number of steps taken on any instance of size
- **Best-case** − The minimum number of steps taken on any instance of size
- **Average case** − An average number of steps taken on any instance of size
- **Amortized** − A sequence of operations applied to the input of size averaged over time.

2.3 Asymptotic notation:

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

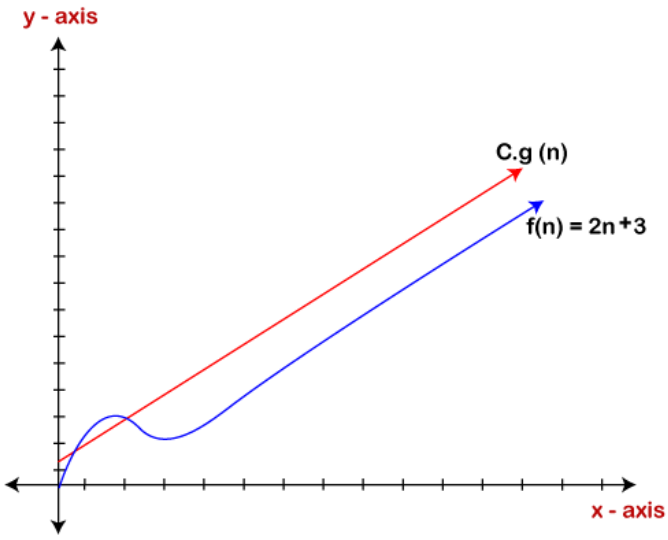
There are mainly three asymptotic notations:

a. Big-O notation
b. Omega notation
c. Theta notation

2.3.1 Big-O notation:

- Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function.
- This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation. It is represented as shown below:



The idea of using big o notation is to give an upper bound of a particular function, and eventually it leads to give a worst-time complexity. It provides an assurance that a particular function does not behave suddenly as a quadratic or a cubic fashion, it just behaves in a linear manner in a worst-case.

2.3.2 Omega notation

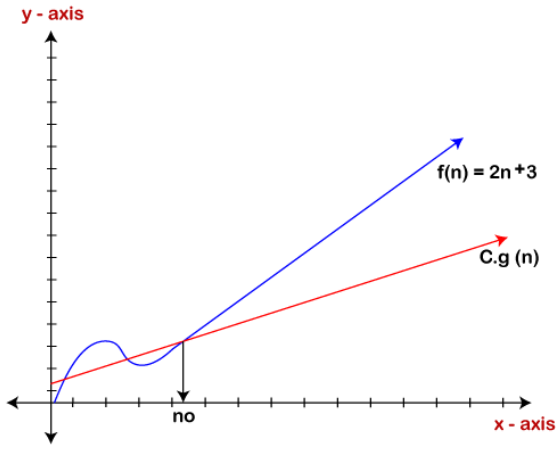
- It basically describes the best-case scenario which is opposite to the big o notation.
- It is the formal way to represent the lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity.
- It determines what the fastest time that an algorithm can run is.

If we required that an algorithm takes at least certain amount of time without using an upper bound, we use big- Ω notation i.e. the Greek letter "omega". It is used to bind the growth of running time for large input size.

If **f(n)** and **g(n)** are the two functions defined for positive integers,

then **f(n) = Ω (g(n))** as **f(n) is Omega of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:

f(n)>=c.g(n) for all n≥no and c>0



As we can see in the above figure that g(n) function is the lower bound of the f(n) function when the value of c is equal to 1. Therefore, this notation gives the fastest running time. But, we are not more interested in finding the fastest running time, we are interested in calculating the worst-case scenarios because we want to check our algorithm for larger input that what is the worst time that it will take so that we can take further decision in the further process.

2.3.3 Theta notation

- The theta notation mainly describes the average case scenarios.
- It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm.
- Big theta is mainly used when the value of worst-case and the best-case is same.
- It is the formal way to express both the upper bound and lower bound of an algorithm running time.

Let's understand the big theta notation mathematically:

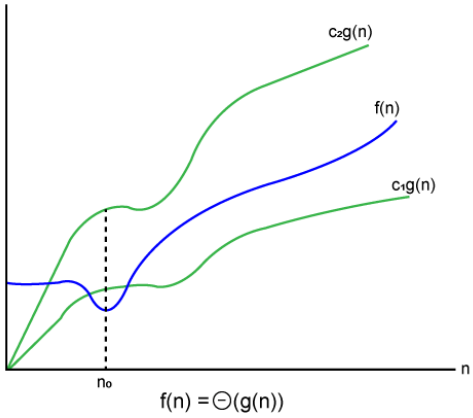
Let f(n) and g(n) be the functions of n where n is the steps required to execute the program then:

f(n)= θg(n)

The above condition is satisfied only if when

c1.g(n)<=f(n)<=c2.g(n)

where the function is bounded by two limits, i.e., upper and lower limit, and f(n) comes in between. The condition **f(n)= θg(n)** will be true if and only if c1.g(n) is less than or equal to f(n) and c2.g(n) is greater than or equal to f(n). The graphical representation of theta notation is given below:



2.4. Basic recursion algorithms on memory level.

What is Recursion?

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc. A

recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems. Many more recursive calls can be generated as and when required. It is essential to know that we should provide a certain case in order to terminate this recursion process. So we can say that every time the function calls itself with a simpler version of the original problem.

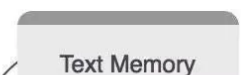
Need of Recursion

Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write. It has certain advantages over the iteration technique which will be discussed later. A task that can be defined with its similar subtask, recursion is one of the best solutions for it. For example; The Factorial of a number.

Memory Allocation in Recursion

When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to the calling function and a different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

When a recursive function is called for the first time, a space is set aside in the memory to execute this call and the function body is executed. Then a second call to a function is made; again a space is set for this call, and so on. In other words, memory spaces for each function call are arranged in a stack. Each time a function is called; its memory area



is placed on the top of the stack and is removed when the execution of the call is completed.

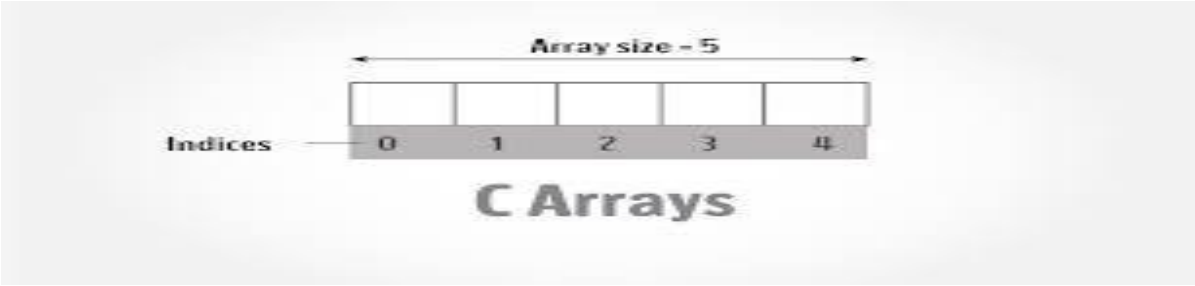
CHAPTER 3

INTRODUCTION TO LINEAR DATA STRUCTURES

It is a type of data structure where the arrangement of the data follows a linear trend. The data elements are arranged linearly such that the element is directly linked to its previous and the next elements. As the elements are stored linearly, the structure supports single-level storage of data. And hence, traversal of the data is achieved through a single run only.

3.1 ARRAY

The array is that type of structure that stores homogeneous elements at memory locations which are contiguous. The same types of objects are stored sequentially in an array. The main idea of an array is that multiple data of the same type can be stored together. Before storing the data in an array, the size of the array has to be defined. Any element in the array can be accessed or modified and the elements stored are indexed to identify their locations.An array can be explained with the help of a simple example of storing the marks for all the students in a class. Suppose there are 20 students, then the size of the array has to be mentioned as 20. Marks of all the students can then be stored in the created array without the need for creating separate variables for marks for every student. Simple traversal of the array can lead to the access of the elements.



CODE :

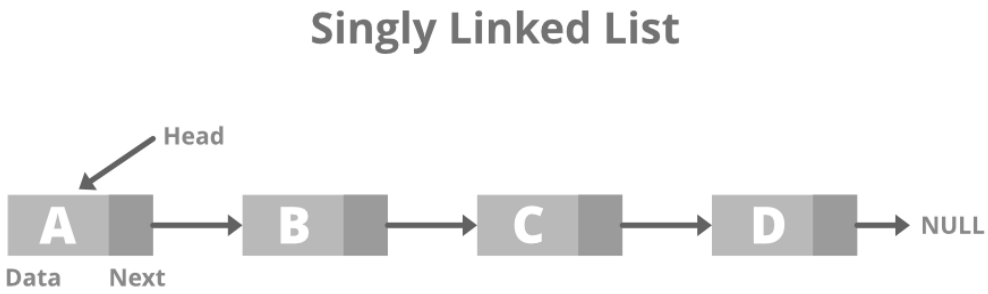
```
#include <iostream>

using namespace std;

int main()  {

int arr[5]={10, 0, 20, 0, 30};

for (int i = 0; i < 5; i++)    {

cout<<arr[i]<<"\n";

  }

}
```

LINKED LIST: The linked list is that type of data structure where separate objects are stored sequentially. Every object stored in the data structure will have the data and a reference to the

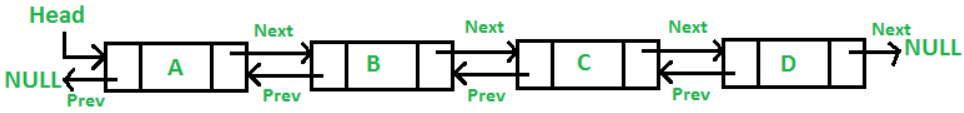
next object. The last node of the linked list has a reference to null. The first element of the linked list is known as the head of the list. There are many differences between a linked list to the other types of data structures. These are in terms of memory allocation, the internal structure of the data structure, and the operations carried on the linked list.

Getting to an element in a linked list is a slower process compared to the arrays as the indexing in an array helps in locating the element. However, in the case of a linked list, the process has to start from the head and traverse through the whole structure until the desired element is reached. In contrast to this, the advantage of using linked lists is that the addition or deletion of elements at the beginning can be done very quickly. There are three types of linked lists:

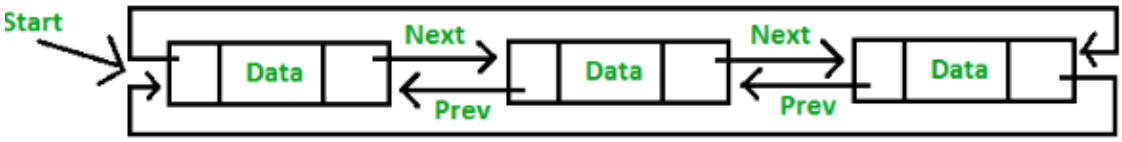
- **Single Linked List:** This type of structure has the address or the reference of the next node stored in the current node. Therefore, a node which at the last has the address and reference as a NULL. Example: A->B->C->D->E->NULL.



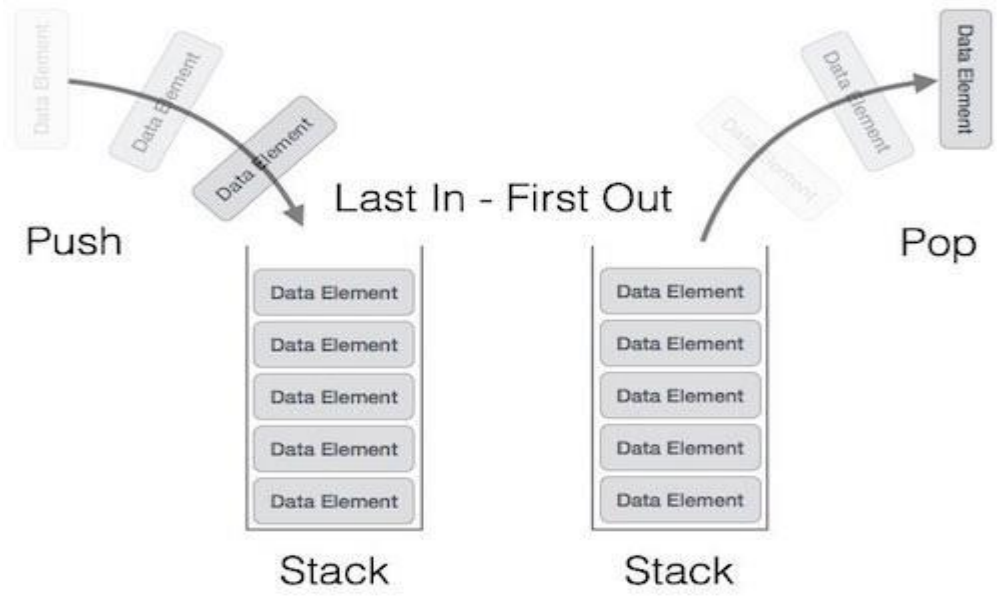
- **A Double Linked List:** As the name suggests, each node has two references associated with it. One reference directs to the previous node while the second reference points to the next node. Traversal is possible in both directions as reference is available for the previous nodes. Also, explicit access is not required for deletion. Example: NULL<-A<->B<->C<->D<->E->NULL.



- **Linked List circular**: The nodes in a circular linked list are connected in a way that a circle is formed. As the linked list is circular there is no end and hence no NULL. This type of linked list can follow the structure of both singly or doubly. There is no specific starting node and any node from the data can be the starting node. The reference of the last node points towards the first node. Example: A->B->C->D->E.



3.2 STACKS: The stack is another type of structure where the elements stored in the data structure follow the rule of LIFO (last in, first out) or FILO (First In Last Out). Two types of operations are associated with a stack i.e. push and pop. Push is used when an element has to be added to the collection and pop is used when the last element has to be removed from the collection. Extraction can be carried out for only the last added element.

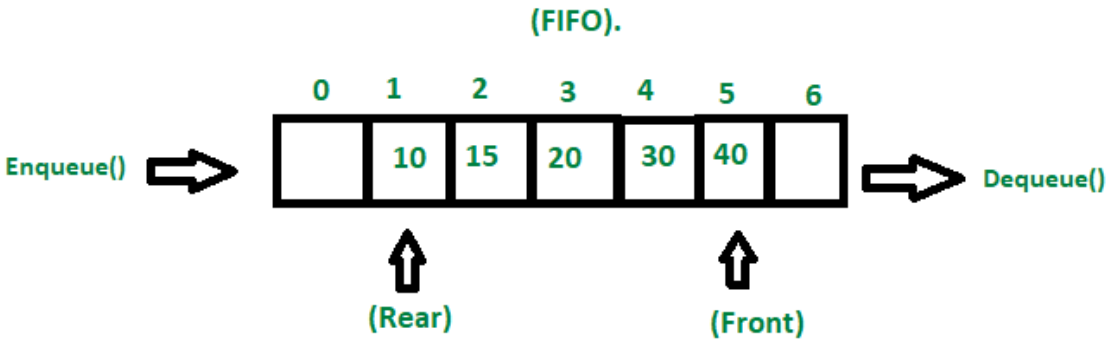


CODE TO PERFORM PUSH AND POP FUNCTIONS:

```
void push(int val) {
   if(top>=n-1)
   cout<<"Stack Overflow"<<endl;
   else {
      top++;
      stack[top]=val;
   }
}
void pop() {
   if(top<=-1)
   cout<<"Stack Underflow"<<endl;
   else {
      cout<<"The popped element is "<< stack[top] <<endl;
      top--;
   }
}
```

3.3 Queue: Queue is the type of data structure where the elements to be stored follow the rule of First In First Out (FIFO). The particular order is followed for performing the required

operations over the elements. The difference of a queue from that of a stack lies in the removal of an element, where the most recently added object is removed first in a stack. Whereas, in the case of a queue, the element that was added first is removed first. Both the end of the data structure is used for the insertion and the removal of data. The two main operations governing the structure of the queue are enqueue, and dequeue. Enqueue refers to the process where inserting an element is allowed to the collection of data and dequeue refers to the process where removal of elements is allowed, which is the first element in the queue in this case. Example of the queue: Similar to those queues made while waiting for the bus or anywhere, the data structure too follows the same pattern. We can imagine a person waiting for the bus and standing at the first position as the person that came to the queue first. This person will be the first one who will get onto a bus, i.e. exit the queue. Queues are applied when multiple users are sharing the same resources and they have to be served on the basis of who has come first on the server.



```
void Insert() {
   int val;
   if (rear == n - 1)
   cout<<"Queue Overflow"<<endl;
   else {
      if (front == - 1)
      front = 0;
      cout<<"Insert the element in queue : "<<endl;
      cin>>val;
      rear++;
      queue[rear] = val;
   }
}
void Delete() {
   if (front == - 1 || front > rear) {
      cout<<"Queue Underflow ";
      return ;
   } else {
      cout<<"Element deleted from queue is : "<< queue[front] <<endl;
      front++;;
   }
}
```

In the above program, the function Insert() inserts an element into the queue. If the rear is equal to n-1, then the queue is full and overflow is displayed. If front is -1, it is incremented by 1. Then rear is incremented by 1 and the element is inserted in index of rear. In the function Delete(), if there are no elements in the queue then it is underflow condition. Otherwise the element at front is displayed and front is incremented by one

CHAPTER 4

HASH TABLE

4.1 Notion of Hashing and Collision

Hashing: Hashing is a technique of mapping a large set of arbitrary data to tabular indexes using a hash function. It is a method for representing dictionaries for large datasets. This is usually represented by a shorter, fixed-length value or key that represents and makes it easier to find or employ the original string. The most popular use for hashing is the implementation of hash tables. Hashing is a popular technique for storing and retrieving data as fast as possible. The main reason behind using hashing is that it gives optimal results as it performs optimal searches.

Why to use Hashing? : If you observe carefully, in a balanced binary search tree, if we try to search , insert or delete any element then the time complexity for the same is O(logn). Now there might be a situation when our applications want to do the same operations in a faster way i.e. in a more optimized way and here hashing comes into play. In hashing, all the above operations can be performed in O(1) i.e. constant time. It is important to understand that the worst case time complexity for hashing remains O(n) but the average case time complexity is O(1).

Let a hash function H(x) maps the value 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.

Collision:

a **hash collision** or **clash** is when two pieces of data in a hash table share the same hash value. The hash value in this case is derived from a hash function which takes a data input and returns a fixed length of bits.

Although hash algorithms have been created with the intent of being collision resistant, they can still sometimes map different data to the same hash (by virtue of the pigeonhole principle). Malicious users can take advantage of this to mimic, access, or alter data.

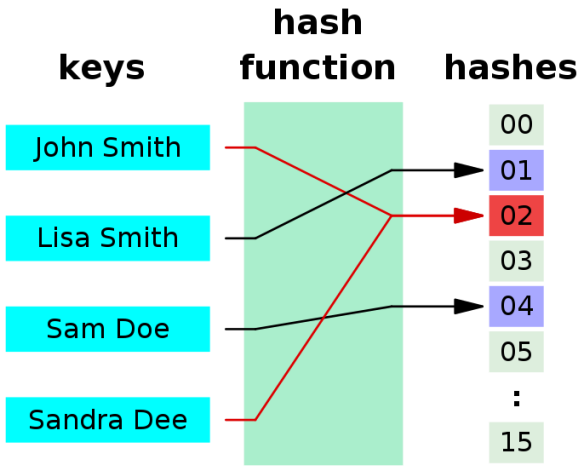


Fig: Hash Collision

4.2 Hash Functions

A **hash function** is any function that can be used to map data of arbitrary size to fixed-size values. A hash function takes a group of characters (called a key) and maps it to a value of a certain length (called a hash value or hash). The hash value is representative of the original string of characters, but is normally smaller than the original. Hashing is used with a database to enable items to be retrieved more quickly. Hashing can also be used in the encryption and decryption of digital signatures. The hash function transforms the digital signature, then both the hash value and signature are sent to the receiver. The receiver uses the same hash function to generate the hash value and then compares it to that received with the message. If the hash values are the same, it is likely that the message was transmitted without errors.

One example of a hash function is called folding. This takes an original value, divides it into several parts, then adds the parts and uses the last four remaining digits as the hashed value or key.

Another example is called digit rearrangement. This takes the digits in certain positions of the original value, such as the third and sixth numbers, and reverses their order. It then uses the number left over as the hashed value.

A good hash function should have the following properties:

1. Efficiently computable.
2. Should uniformly distribute the keys (Each table position equally likely for each key)

The two heuristic methods are **hashing by division** and **hashing by multiplication** which are as follows:

1. **The mod method:**

- In this method for creating hash functions, we map a key into one of the slots of table by taking the remainder of key divided by table_size. That is, the hash function is

h(key) = key **mod** table_size

i.e. key % table_size

- Suppose **r** = 256 and **table_size** = 17, in which r % table_size i.e. 256 % 17 = 1.
- So for **key = 37599**, its hash is

37599 % 17 = 12

- But for **key = 573**, its hash function is also

573 % 17 = 12

1. **The multiplication method:**
    - In multiplication method, we multiply the key **k** by a constant real number **c** in the range **0 < c < 1** and extract the fractional part of **k * c**.
    - Then we multiply this value by table_size **m** and take the floor of the result. It can be represented as

h(k) = floor (m * (k * c mod 1))

or

h(k) = floor (m * frac (k * c))

4.3 Method of Collision Handling

There are mainly two methods to handle collision:

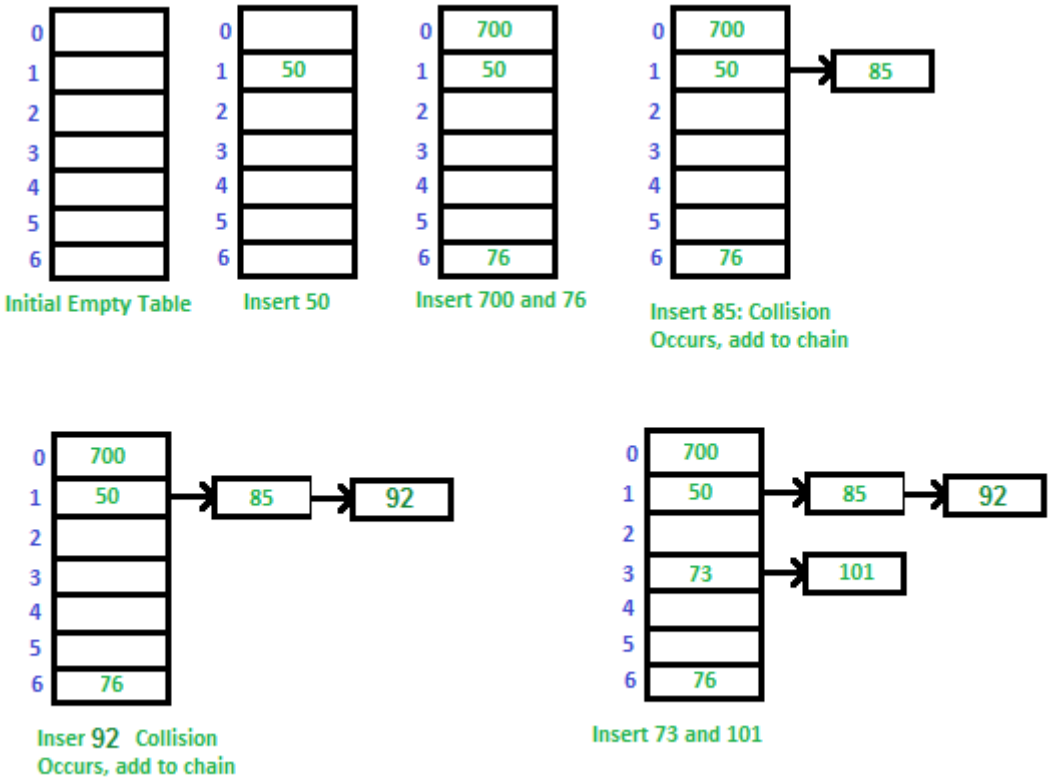
- Separate Chaining
- Open Addressing

Separate Chaining: The idea behind separate 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.

The **linked list** data structure is used to implement this technique. So what happens is, 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.

Here, all those elements that hash into the same slot index are inserted into a linked list. Now, we can use a key K to search in the linked list by just linearly traversing. If the intrinsic key for any entry is equal to K then it means that we have found our entry. If we have reached the end of the linked list and yet we haven't found our entry then it means that the entry does not exist. Hence, the conclusion is that in separate chaining, if two different elements have the same hash value then we store both the elements in the same linked list one after the other.

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



Open Addressing:

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the **hash table** itself. So at any point, the size of the table must be greater

than or equal to the total number of keys (Note that we can increase table size by copying old data if needed). This approach is also known as closed hashing. This entire procedure is based upon probing. We will understand the types of probing ahead:

- **Insert(k):** Keep probing until an empty slot is found. Once an empty slot is found, insert k.
- **Search(k):** Keep probing until the slot's key doesn't become equal to k or an empty slot is reached.
- **Delete(k)**: **Delete operation is interesting**. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted". The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

CHAPTER 5

SEARCHING AND SORTING INTRODUCTION

SORTING

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order. The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats.

In-place Sorting and Not-in-place Sorting

Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called **in-place sorting**. Bubble sort is an example of in-place sorting.

However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called **not-in-place sorting**. Merge-sort is an example of not-in-place sorting.

5.1 Bubble sort

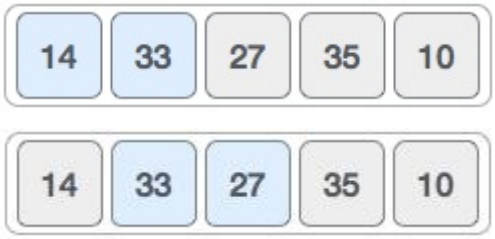
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

Working:

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare



33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.

The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

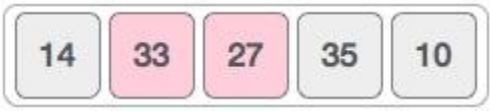
We swap these values. We find that we have reached the end of the array. After one iteration, the

array should look like this −

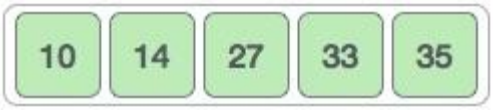
To be precise, we are now showing how an array should look like after each iteration. After the

second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



C program to implement bubble sort :-

```
#include <bits/stdc++.h>

using namespace std;

void bubbleSort(int arr[], int n)

{

	int i, j;

	for (i = 0; i < n - 1; i++)

		for (j = 0; j < n - i - 1; j++)

			if (arr[j] > arr[j + 1])

				swap(arr[j], arr[j + 1]);

}

void printArray(int arr[], int size)

{

	int i;

	for (i = 0; i < size; i++)

		cout << arr[i] << " ";

	cout << endl;

}

int main()

{

	int arr[] = { 5, 1, 4, 2, 8};
```

```
    int N = sizeof(arr) / sizeof(arr[0]);

    bubbleSort(arr, N);

    cout << "Sorted array: \n";

    printArray(arr, N);

    return 0;

}
```

5.2 Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Characteristics of Insertion Sort:

This algorithm is one of the simplest We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

- ***Second Pass:***

- *Now, move to the next two elements and compare them*

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.*Here, 12 is*

greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Th These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list.

Code to implement insertion sort

```
#include <bits/stdc++.h>

using namespace std;
```

```
void insertionSort(int arr[], int n)
{
	int i, key, j;
	for (i = 1; i < n; i++)
	{
		key = arr[i];
		j = i - 1;
		while (j >= 0 && arr[j] > key)
		{
			arr[j + 1] = arr[j];
			j = j - 1;
		}
		arr[j + 1] = key;
	}
}
void printArray(int arr[], int n)
{
	int i;
	for (i = 0; i < n; i++)
	cout << arr[i] << " ";
	cout << endl;
```

```
}

int main()

{

        int arr[] = { 12, 11, 13, 5, 6 };

        int N = sizeof(arr) / sizeof(arr[0]);

        insertionSort(arr, N);

        printArray(arr, N);

        return 0;

}
```

5.3 Merge sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following −

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly the same manner as they were broken down.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this −

Code to implement Merge Sort:

```
#include <iostream>
using namespace std;
void merge(int array[], int const left, int const mid,int const right)
{
        auto const subArrayOne = mid - left + 1;
        auto const subArrayTwo = right - mid;
        auto *leftArray = new int[subArrayOne],*rightArray = new int[subArrayTwo];
        for (auto i = 0; i < subArrayOne; i++)
                leftArray[i] = array[left + i];
        for (auto j = 0; j < subArrayTwo; j++)
                rightArray[j] = array[mid + 1 + j];
        auto indexOfSubArrayOne = 0, // Initial index of first sub-array
        indexOfSubArrayTwo = 0; // Initial index of second sub-array
        int indexOfMergedArray
                = left; // Initial index of merged array
        while (indexOfSubArrayOne < subArrayOne
                && indexOfSubArrayTwo < subArrayTwo) {
                if (leftArray[indexOfSubArrayOne]
                        <= rightArray[indexOfSubArrayTwo]) {
```

```
                array[indexOfMergedArray]
                        = leftArray[indexOfSubArrayOne];
                indexOfSubArrayOne++;
        }
        else {
                array[indexOfMergedArray]
                        = rightArray[indexOfSubArrayTwo];
                indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
}
while (indexOfSubArrayOne < subArrayOne) {
        array[indexOfMergedArray]
                = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
}
while (indexOfSubArrayTwo < subArrayTwo) {
        array[indexOfMergedArray]
                = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
```

```
            indexOfMergedArray++;
        }
        delete[] leftArray;
        delete[] rightArray;
}
void mergeSort(int array[], int const begin, int const end)
{
        if (begin >= end)
            return; // Returns recursively
        auto mid = begin + (end - begin) / 2;
        mergeSort(array, begin, mid);
        mergeSort(array, mid + 1, end);
        merge(array, begin, mid, end);
}
void printArray(int A[], int size)
{
        for (auto i = 0; i < size; i++)
            cout << A[i] << " ";
}
// Driver code
int main()
```

```
{
        int arr[] = { 12, 11, 13, 5, 6, 7 };
        auto arr_size = sizeof(arr) / sizeof(arr[0]);
        cout << "Given array is \n";
        printArray(arr, arr_size);
        mergeSort(arr, 0, arr_size - 1);
        cout << "\nSorted array is \n";
        printArray(arr, arr_size);
        return 0;
}
```

5.4 Searching of element

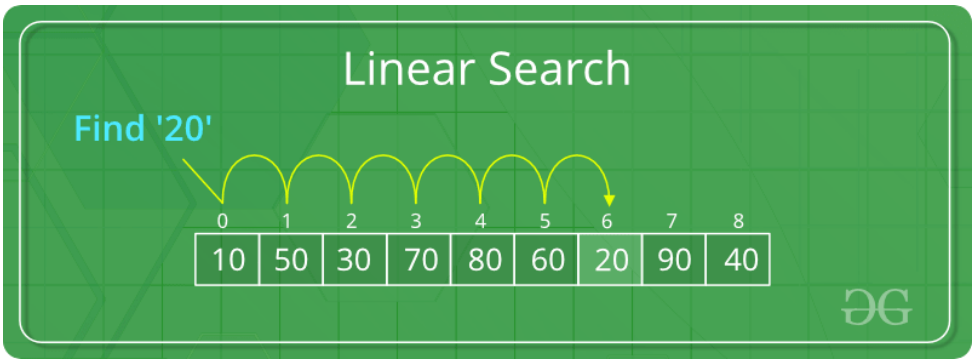
Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

1. **Sequential Search**: In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.

   Timecomplexity: O(N)

   Auxiliary Space: O(1)

   Linear Search to find the element “20” in a given list of numbers

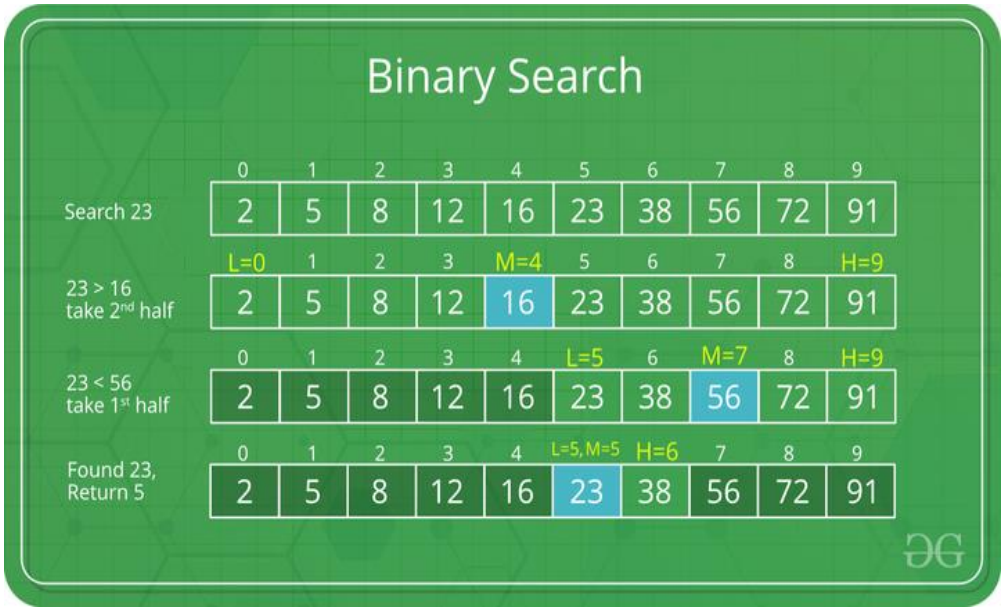


2. **Interval Search**: These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: Binary Search.

   TimeComplexity: O(logn)

   Auxiliary Space: O(log n)

Binary Search to find the element "23" in a given list of numbers



Binary Search

REFERENCES

1. https://www.geeksforgeeks.org/ …accessed on Aug. 2022.

2. https://www.programiz.com/ ….accessed on Sept. 2022.

3. https://www.tutorialspoint.com/ …accessed on Oct. 2022