What is **Data**?

Data is a set of facts or values (such as alphabets, numbers, symbols, or a mixture of them). Data can be analyzed or used in an effort to gain knowledge, information or make decisions; and represented in a form suitable for processing by computer.

What is **Information**?

Information is a processed, interpreted, organized, structured or presented data, so to make the data meaningful or useful.

What is data **structure**?

Data structure is a way of organizing and storing data so that operations can be performed efficiently and easily. It represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.

Not all data structures can perform all the operations efficiently in term of time and storage space, So, that led to develop different types of data structures.
An example of the important of data structure is, if you need to find a specific book in an unorganized library, that task would take an enormous amount of time. Just like a library organizes their books, data need to be organized so operations can be performed efficiently.
There are many operations that could perform on data as follows:

- Accessing
- Inserting
- Deleting
- Finding
- Sorting
- Analyzing

Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, if have some data which has, player's name as a "string" and age 26 as an integer data type, it can organize this data as a record like Player record, which will have both player's name and age in it. Now it can collect and store player's records in a file or database as a data structure as follows. Player

Player Record	
Name	Age
Jack	20
Taylor	22

Basic types of Data Structures
Anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as Primitive Data Structures.
Then also have some complex Data Structures, which are used to store large and connected data. Some example of Abstract Data Structure (ADS) are:

- Linked List
- Tree
- Graph
- Stack
- Queue

All these data structures allowed to perform different operations on data. These data structures have been selected based on which type of operation is required. Will look into these data structures in more details in the later lessons.

The data structures can be classified based on the following characteristics:

Characteristic	**Description**
Linear	In Linear data structures, the data items are arranged in a linear sequence. Example: **Array**
Non-Linear	In Non-Linear data structures, the data items are not in sequence. Example: **Tree**, **Graph**
Homogeneous	In homogeneous data structures, all the elements are of same type. Example: **Array**
Non- Homogeneous	In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: **Structures**
Static	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: **Array**
Dynamic	Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: **Linked List created using pointers**

What is Array?

An array is a data structure that contains a group of elements. Typically these elements are all of the same data type, such as an integer or string. Arrays are commonly used in computer programs to organize data so that a related set of values can be easily sorted or searched.

The important characteristics of an array are:

- Each element has the same data type (although they may have different values).
- The entire array is stored contiguously in memory (that is, there are no gaps between elements see figure 1).

Arrays can have more than one dimension. A one-dimensional array is called a **vector** ; a two-dimensional array is called a **matrix**.

1. One-dimensional

In C++ we define an array like follows:

Char X[N]; where X is the name of the array and N is the number of elements in the array.

The address of the first element in the array is the Base Address(BA) for that array, the BA is used to find each element location of the array in memory.

Memory representation of one-dimensional array:

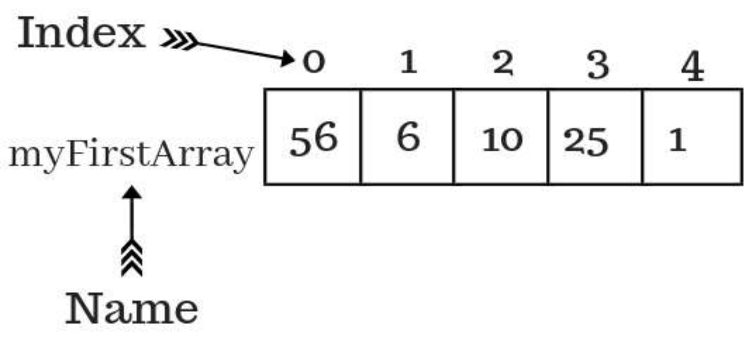


Figure1: memory representation of a vector

How to find an element of an array location in memory?

Before that we need to know the fundamental size of each data type in C++. The following table shows the sizes of the data types.

Type	Size (Byte)
Integer	4
Short integer	2
Long Long integer	8
Char	1
Float	4
Double	8

Now to find the location of A[i], we do the following:

Loc(A[i])= BA + i * size.

Exercise 1) Find the location of A[7] from the int A[50], the BA=50.

Loc(A[7])= 50 + 7 * 4

=50 + 28

= 78

Note: The fourth element in the array is not Loc(X[4]), it's X[3].

Exercise 2) Let's consider that we have Char X[100], Find the fourth element location, if we got the BA is 500.

Loc(X[3])= 500 + 3 * 1

= 500 +3

= 503

Exercise 3) Find Loc(M[3]) of Double M[10]. The BA is 75.

Loc(M[3])= 75 + 3 * 8

=75 + 24

= 99

2-**Two Dimension Array**: it is a Data Structure contains a group of data elements that can be represented as M*N, where the M number of rows of that array and N number of columns. Each element can be accessed via two indexes, which are:

$0 <= i < M$

$0 <= j < N$

For example A[3][5], i=3 and j=5, which means the element located in the 4th row and 6th column.

How to calculate an element address of two-dimension array in memory?

There are two methods to calculate an element address in two-dimension array.

1- **Row-Wise Method**: In this method, the programming language represents an array in memory row by row starting by the BA(Based Address). If the above example considered and got the BA=700, the first element A[0][0] address =700. The calculation the address of A[i][j] using **ROW-WISE** method will be according to the following:

Loc(A[i][j]) = BA + (N * i + j) * size
N = total number of array columns.

Exercise 4) Find the address of the element **A[3][2]** of the array **Float A[8][5]**, **BA**=1000.

Sol: ***Loc(A[3][2]) = 1000 + (5 * 3 + 2) * 4***

=1000 + (17) * 4

=1000 + 68 = 1068

Exercise 5) We have got the array Long Long int **T[5][7]** and the **BA**=900, calculate the address of the **T[3][5]** using row-wise method.

Sol: ***Loc(T[i][j]) = BA + (N * i + j) * size***

Loc(T[3][5]) = 900 + (7 * 3 + 5) * 8

= 900 + (26) * 8

= 900 + 208 = 1108

2- **Column-Wise Method**: Array is represented in memory column by column starting from BA. The calculation the address of **A[i][j]** using **column-wise** method as follows:

Loc(A[i][j]) = BA + (M * j + i) * size
M = total number of array rows.

Exercise **6**) Re-calculate the addresses of the previous examples using the column-wise method.

Sol exercise 4 in column-wise method: (Find the address of the element **A[3][2]** of the array **Float A[8][5]**, **BA**=1000.)

Loc(A[3][2]) = 1000 + (8 * 2 + 3) * 4

= 1000 + (19) * 4

= 1000 + 76 = 1076

Structures:

A ***structure*** is a group of data elements grouped together under one name. These data elements, known as members, can have different types and different lengths. Data structures can be declared in C++ using the following syntax:

```
struct struct_name {
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
}
```

Where struct_name is a name for the structure. Within braces {}, there is a list with the data members, each one is specified with a type and a valid identifier as its name.

For example:

```
#include<iostream>
Using namespace std;
struct student {
  string  name;
  float mark1;
  float mark2;
  float avg;
};
int main(){
float sum=0;
student ali;
cin >> ali.mark1;
cin >>ali.mark2;
sum = ali.mark1 + ali.mark2;
ali.avg = sum/2;
cout << ali.avg;
}
```

This declares a structure type, called student, and defines it having three members: mark1, mark2 and avg, each of a different fundamental type. This declaration creates a new type (student), which is then used to declare one object (variable) of this type: **Ali**. Note how once student is declared, it is used just like any other type.

This struct is designed and implement for one student only; to implement for a group of students, let's say a 100 student, we create an array for that group as the defined struct above, just change the bit in the main function.

```
int main() {
student std[100];
}
```

Calculate the location of a structure member in the memory:

To calculate location of a structure member, we use the same previously mentioned methods for one and two dimensions arrays.

Exercise 7) Find the location of the member bk[6] of Book bk[10] when BA=25 and OS=16 bit.

```
 struct Book {
  char title;
  short int  ISBN;
  char author;
}
Book bk[10];
```

We use the following:

Loc(bk[i])= BA + i * size

bk[6] = 25 + 6 * (1 + 2 + 1)

= 25 + 6 * 4

= 25 + 24 **= 49**

Exercise 8) Calculate the member's location std[4] of Student std[20], BA=200.

```
struct Student {
  char name[20];
  int  age;
  float avg;
} std[20];
```

Loc(std[i])= BA + i * size

Loc(std[4]) = 200 + 4 * (20 + 4 + 4)

= 200 + 4 * 28

= 200 + 112 = **312**

In case it requires to calculate a member of an object like ***std[4].avg***, it needs to calculate the size of shifted bytes.

Shifted_bytes = 20 + 4 = 24.

Loc(std[i].avg)= BA + i * size + shifted_bytes

Exercise 9) Having the following structure, BA=500:

```
struct Student {
  char name[15];
  int  age;
  char gender;
  float avg;
} S[30];
```

```
Find the following locations:
1-S[7]
2-S[3].age
3-S[6].name[12]
4-S[6].gender
5-S[6].avg
```

Sol:

1- ***Loc(S[i])= BA + i * size***

Loc(std[7])= 500 + 7 * (15 + 4 + 1 + 4)

= 500 + 7 * 24

= 500 + 168 = **668**

2- ***Loc(S[3].age)= BA + i * size + shifted_bytes***

= 500 + 3 * 24 + (15)

= 500 + 72 + 15

= 500 + 87 = **587**

3- ***Loc(S[6].name[12])= BA + i * size + shifted_bytes***

= 500 + 6 * 24 +(12)

= 500 + 144 + 12

= 500 + 156 = **656**

You solve 4 and 5.

Exercise 10) Calculate the location of A[1][2], BA=6, as having the following structure:

```
struct Book {
  char title;
  long long int  No;
  float S_no;
}
```

```
Book A[3][4];
```

Here, finding the member location is depending on which method we're using:

For the **column-wise** method:

Loc(A[i][j]) = BA + (M * j + i) * size

Or for the **row-wise** method:

Loc(A[i][j]) = BA + (N * i + j) * size

Nested Structure:

```
struct Name {
  char N1;
  int  N2;
 }
struct Book {
  Name title;
  int  No;
  float S_no;
}
Book X[10];
```

Find Loc(`X[5]`), BA=10.
Sol:

Loc(x[i])= BA + i * size

Loc(x[5])= 10 + 5 * (4 + 4 + (4 + 1))

=10 + 5 * 13

= 10 + 65 = 75

What is Stack?

A stack is a data structure consisting of a set of homogeneous elements and is based on the principle of **Last In First Out (LIFO)**. It is a commonly uses two major operations, namely push and pop. Push and Pop are carried out on the topmost element, which is the item most recently added to the stack. The push operation adds an element to the stack while the pop operation removes an element from the top position. The stack concept is used in programming and memory organization in computers.

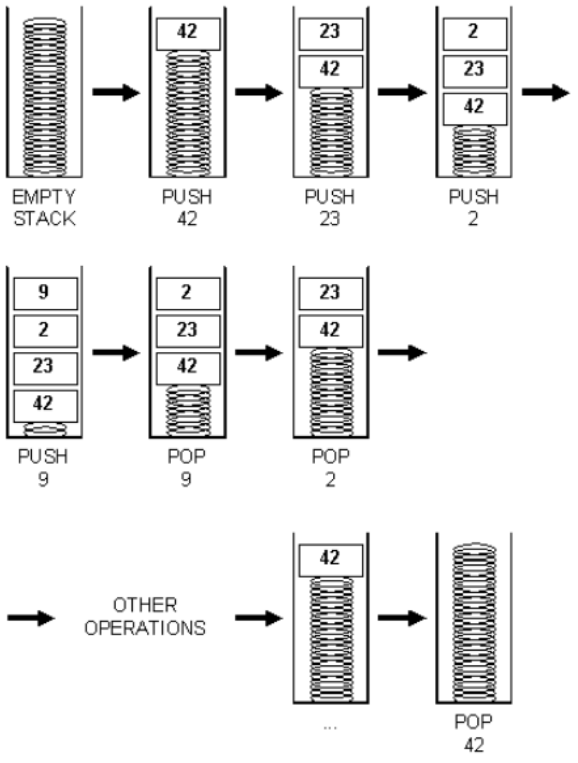
A stack represents a sequence of objects or elements in a linear data structure format. The stack consists of a bounded bottom and all the operations are carried out on the top position. Whenever an element is added to the stack by the push operation, the **top** pointer is increment by one, and when an element is popped out from the stack, the **top** pointer is decrement by one. A pointer to the top position of the stack is also known as the stack pointer.

A stack may be fixed in size or may have dynamic implementation where the size is allowed to change. In the case of bounded capacity stacks, trying to add an element to an already full stack causes a stack overflow exception. Similarly, a condition where a pop operation tries to remove an element from an already empty stack is known as underflow.

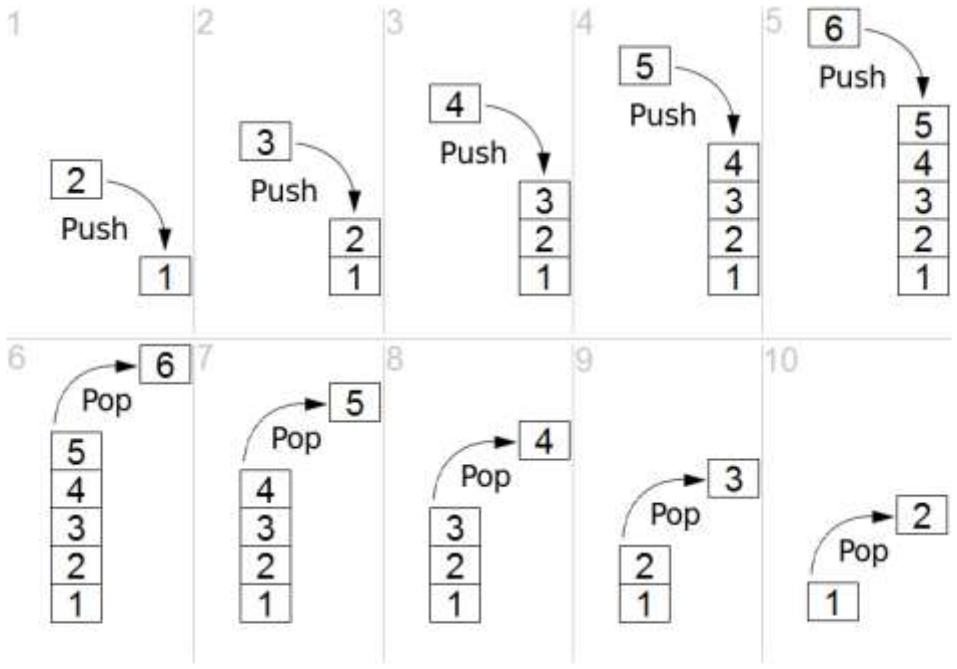
A stack is considered to be a restricted data structure as only a limited number of operations are allowed (Push & Pop). Besides the push and pop operations, certain implementations may allow for advanced operations such as:

- Peek — View the topmost item in the stack.
- Duplicate — Copy the top item's value into a variable and push it back into the stack.
- Swap — Swap the two topmost items in the stack.

Stack Example1



Stack Example2



Implementation of Stack using array in C++:

To implement a stack using array in C++, two variables need to be defined first.

1- Max: Size of stack.
2- Top: it is a pointer pointing to the first element in the stack.

```
#include<iostream>
using namespace std;
#define max 5
int stack1[max], top=-1;
```

- **Push function.**
  1- Check the stack is not overflow(`top<max-1`).
  2- Increase the top pointer by 1(`top +=1`).
  3- Add the new item into the stack (`stack[top]=item`)

  As follows:

```
void push(int x)
{
if (top>=max-1){
cout<<"The stack is overflow.\n";
exit(1);
}
else
{
 top +=1;
    stack1[top]=x;
 cout<<x<<" is inserted into the stack.\n";
} }
```

- **Pop function.**

1- Check the stack is not underflow (`top != -1`).
2- Take the top item of the stack (`item=stack[top]`).
3- Decrease the top pointer by 1(`top -=1`).

As follows:

```
int pop()
{ int item;
    if (top <0)
    {
        cout<<"The stack is underflow.\n";
       exit(1);
    }
    else
    {
        item= stack1[top];
       top -=1;
    }
 return item;
}
```

- **Main function.**

1- Push or pop all the desire items into and from the stack.

As follows:

```
int main()
{
    int y,l;
    for(int i=0; i<max;i++)
    {
    cout <<"Enter an integer no. to be pushed into the stack\n";
    cin >>y;
    push(y);
    }
  for(int j=0; j<max+1;j++){
  l=pop();
 cout <<"The element that popped out from the stack is "<<l<<"\n";
    }
}
```

H.W.) Write a C++ code to design a stack that contain 10 integer numbers, then find the summation of all the negative numbers in the stack.

Stack operations:

Push -> Stacking (S).

Pop-> Unstacking(U).

Ex10) If a stack gets the following input stream from **right to left** (R,N,Y,B,M), find the output after applying the following group of stack operations(left->right).

1- SSUUSUSUSU

Input: M B Y N R

Operations: S S U U S U S U S U

Output: B M Y N R

2- SSSUSUUSUU

Input: M B Y N R

Operations: S S S U S U U S U U

Output: Y N B R M

Ex11) If a stack gets the following input stream from **left to right** (1,2,3,4,5), demonstrate which one of the following outputs is correct according to stack mechanism.

a- 2 4 5 3 1 (left->right)

Input: 1 2 3 4 5

Operations: S S U S S U S U U U

Output: 2 4 5 3 1 The output is correct.

Input stream **left to right** (1,2,3,4,5):

b- 4 2 3 1 5

Input: 1 2 3 4
Operations: S S S S U U U U
Output: 4 3 2 1 This answer is incorrect.

c- 4 5 1 2 3

Input:
Operations:
Output:

d- 4 3 5 2 1

Input:
Operations:
Output:

H.W.) Write a C++ code to find the maximum value in a stack of 10 integer numbers. Push the 10 numbers first into the stack.

H.W.) Write a C++ code to create two arrays as an output of a stack, the first array contains lowercase letters, while the second array stores the uppercase letters.

What are Pointers?

Pointers are symbolic address of a memory location. As known every variable is a memory location and every memory location have its address defined which can be accessed using ampersand (&) operator which denotes an address in memory.

So, **Pointer** can be defined as a variable whose value is the address of another variable **or** (variable that stores the address of another variable). Like any variable or constant, it must declare a pointer before it can work with it. The general form of a pointer variable declaration is :

```
type *var-name;
```

Consider the following which will print the address of the variables defined:

```
void main () {
   int  var1;

   cout << "Address of var1 variable: ";
   cout << &var1 << endl;

}
```

When the above code is compiled and executed, it produces the following result :

```
Address of var1 variable: 0xbfebd5c0
```

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-**name is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch     // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Using Pointers in C++

There are few important operations, which we will do with the pointers very frequently.

a-define a pointer variable.

b-Assign the address of a variable to a pointer.

c-Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

Following example makes use of these operations:

```
int main(){
int x=100;
int *p;         //a
p=&x;           //b
cout<<p<<"\n";
cout<<*p<<"\n"; //c
}
```

Using pointers in functions:

```
#include<iostream>
using namespace std;
int *read_a(){
int *A;
int a;
cin >>a;
A =&a;
return A;
}
int *read_b(){
int *B;
int b;
cin >>b;
B =&b;
return B;
}
int main(){
int *p;
p = read_a();
cout<<p<<"\n"<<*p;   }
```

Example) write a C++ program to implement stack using structures with pointers.

H.W.) Write a C++ code to push 10 integer values into a stack, then create two stacks (one contains the even values of the first stack & the second contains the odd values). Use structures to implement the three stacks.

Example) Write a C++ program to merge two stacks (size of each stack is 5 float elements) into an array.

H.W.) Write a C++ code to do the following:

1- create two stacks (stack1[10] & stack2[15])

2- Switch the elements of the theses stacks.

H.W.) Write a C++ **function** to find the number of elements in a stack (the number of the elements is unknown).

H.W.) Write a C++ code to push 10 float values into a stack then write a function that calculates the average of the stack elements, then push the average into the stack.

What is the output of the following code(push & pop functions are already implemented):

```
#define max 10
int main(){
int x1,y;
for(int i=2;i<max;i+=2)
{
cout<<"i's value is "<<i<<endl;
x1=(2*i)+2;
push(x1);
}
for(int j=2;j<max;j+=2)
{
y=pop();
cout<<y;
}
```

Implementing Stack using Structures:

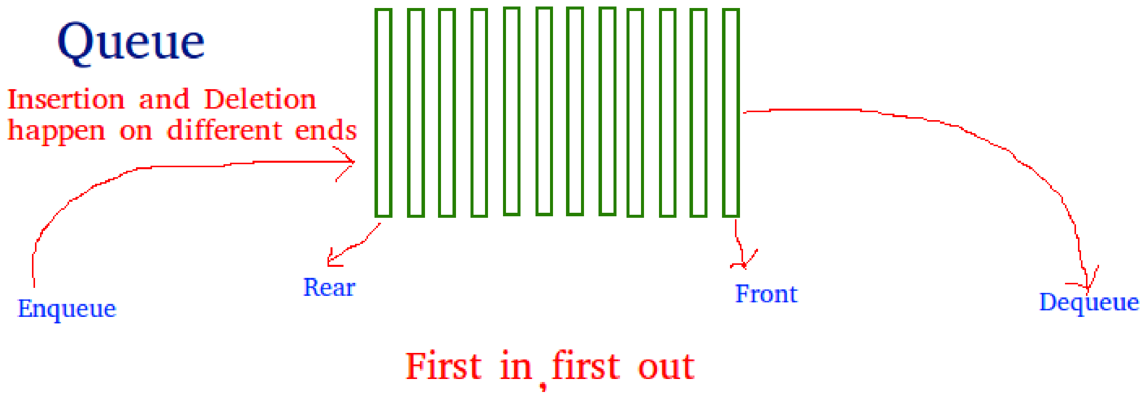
To avoid re-write push & pop functions for each stack that is needed, it is vital to write a re-usable function (write functions and apply it to all implemented stacks). To achieve this aim, implementing stack using structures gives the solution. This task is performed with pointers.

1-Queue

What is Queue:

Queue is another type of computers data structure. It is a linear collection of items. Unlike Stack, queue has two ends, in which each element is inserted from one end called the **REAR**, and the removal of existing element takes place from the other end called as **FRONT**. This makes queue as **FIFO** (First in First Out) data structure, which means that element inserted first will be removed first. Which is exactly how queue system works in real world. If you go to a ticket counter to buy tickets, and you are first in the queue, then you will be the first one to get the tickets. Same is the case with Queue data structure. Data inserted first, will leave the queue first. The following figure shows how queue works in the real-world.

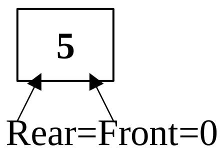
The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



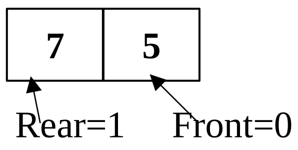
Queue example:

Here is an illustration on how queue works, and the changes on the values of **Rear** and **Front** pointer

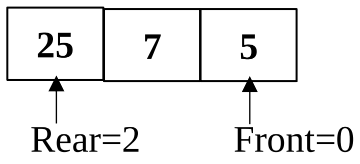
1. When queue is empty, both Rear = Front = -1
2. Enqueue(5)



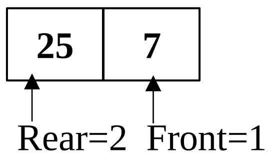
3. Enqueue(7)



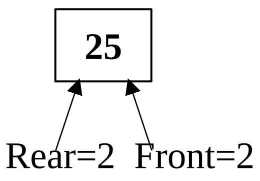
4. Enqueue(25)



5. X=Dequeue(), X=5.



6. X=Dequeue(), X=7.



7. X=Dequeue(), X=25.
   Rear = Front = -1

Note*: When a queue has only one item, the values of Rear and Front are the same. In other words, if Rear and front pointer are equal, means queue contains one item only.*

Queue Applications:

Queue is used whenever it needs to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real-world scenario, Call Centre phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e. First come first served.

Enqueue Algorithm:

1. Check if the queue is not full. If it is, then raise an error and exit.
2. Check if the queue is empty, if it is, then Rear = Front == 0.
3. If none of the above cases, then Rear +=1.
4. Insert item into the queue.

Enqueue Function

```
#define max 10
int front=-1, rear=-1;
int q[max];
void enqueue(int x){
    //if the queue is full
    if (rear>=max-1){
        cout<<"The queue is full.\n";
        exit(1);
    }
    //if the queue is empty
    else if(rear ==-1 && front==-1){
        rear =0;
        front =0;
         cout<<"The queue was empty and new item will be
enqueued\n";
    }
    else {
        rear ++;
    }
    q[rear]=x;
    cout<<"The item "<<x<<" is enqueued\n";     }
```

Dequeue Algorithm:
Accessing data from the queue is a process of two tasks − access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation

1. Check if the queue is not empty. If it is, then raise an error and exit.
2. Check if Rear == Front, if it is, means the queue has only one item, dequeuer the item the reset the pointers, Rear = Front= -1.
3. If none of the above cases, then, dequeue the item and increment Front by one (Front +=1).

```
int dequeue(){
    int z;
    //if the queue is empty
    if(front == -1 ){
        cout<<"The queue is empty.\n";
        exit(1);
    }
    //means last item in the queue.
    else if (front == rear)|| (front > rear){
        z=q[front];
        cout<<"\nfront = "<<front;
        cout<<"\nrear = "<<rear<<"\n";
        front=-1;
        rear=-1;
        //cout<<"\nLast item in the queue.\n";
        }
    else{
        z=q[front];
       //cout<<"\nThe "<<q[front]<<" item in dequeued.\n";
        front ++;
    }
return z;
}
```

Exercise: Show the contents of queue and the positions of **Front** & **Rear** pointers after performing the following enqueue and dequeue operations. Queue size=5.

[Queue initialization, enqueue(10), enqueue(20), dequeue(), enqueue(30), enqueue(40), dequeue(), enqueue(50), dequeue(), dequeue(), dequeue()].

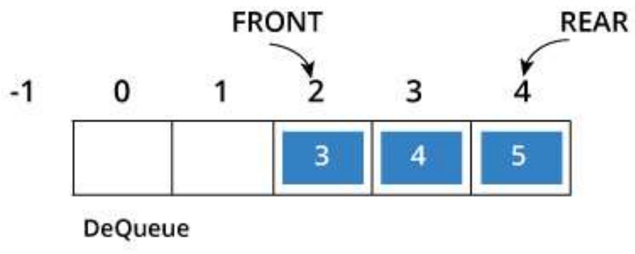
H.W) Show the contents of queue and the positions of **Front** & **Rear** pointers after performing the following enqueue and dequeue operations. Queue size=5.

[Queue initialization, enqueue(A), dequeue(), enqueue(B), enqueue(C), enqueue(D), dequeue(), dequeue(), enqueue(E), dequeue()]

2- Circular Queue

Why Circular Queue?

There is a weak point in Queue data structure, which is raised after a few of enqueueing and dequeuing operations, the size of the queue is reduced. The indexes 0 and 1 can only be used after the queue is reset when all the elements have been dequeued. The following figure shows the normal queue problem.



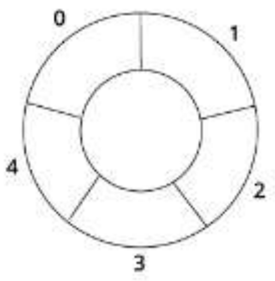
How Circular Queue Works

Circular Queue works by the process of circular increment i.e. when we try to increment any variable and we reach the end of queue, we start from the beginning of queue by modulo division with the queue size.

i.e.

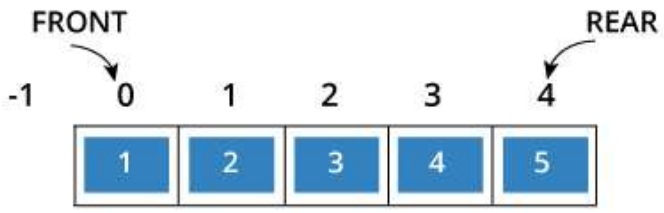
```
if REAR + 1 == 5 (overflow!)

REAR = (REAR + 1)%5 = 0 (start of queue)
```

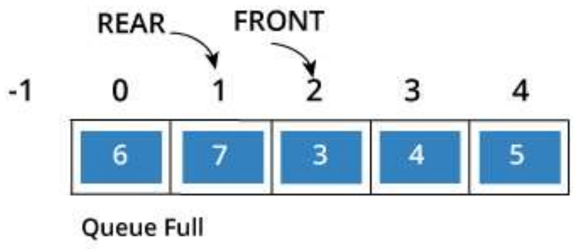


To check for full queue has a new additional case:

- **Case 1**: FRONT = 0 && REAR == SIZE – 1



- **Case 2**: FRONT = REAR + 1



Enqueue Function

```
#define max 10
int front=-1, rear=-1;
int q[max];
void enqueue(int x){
    if( (rear>=max-1 && front==0) || (front == rear+1) ){
        cout<<"The queue is full\n";
        exit(0);
}
    else{
        if(front==-1){
            front=0;
            rear=(rear+1)%max;
            q[rear]=x;
            cout<<"The queue was empty. A new item is added
"<<x<<" \n";
        }
        else{
            rear=(rear+1)%max;
            q[rear]=x;
            cout<<"A new item "<< x << " is enqueued\n";
        }
}
}
```

Dequeue Function

```
int dequeue(){
    int z;
    //if the queue is empty
    if(front == -1 ){
        cout<<"The queue is empty.\n";
        exit(1);
    }
    else {
         z=q[front];
         if (front == rear) {
             front=-1;
             rear=-1;
          }
          else{ front = (front+1)% max;}
              }
return z;}
```

Lab exercise: Implement a Queue of 10 integer values using C++, then enqueue 10 values and dequeue the inserted items.

H.W: Implement a Queue of 15 characters using C++, then dequeue all the inserted items except the content of no. 6 cell.

Linked List (القوائم الموصولة):

What is Linked list?

linked list is a linear data structure, unlike arrays, elements are not stored at contiguous memory locations as can be shown in **figure1**, they are linked using pointers as shown in the **figure2**:

Node1	Var.	file	Node2	Var.	Var.	Var.	Node3

Figure1: RAM(Random Access Memory).

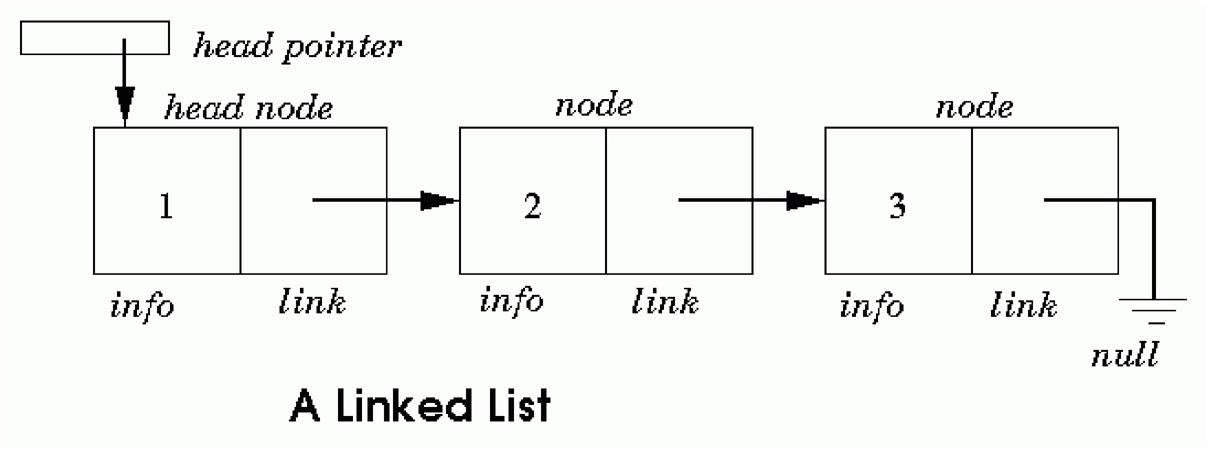


Figure 2: Example of a linked list with 3 nodes

The elements of a linked list are called the nodes. A node has two fields i.e. **data** and **next (Link)**. The data field contains the data being stored in that specific node. It can be a single variable or many variables presenting in data section of node. The next field contains the address of the next node, so this is the place where the link between nodes is established.

No matter how many nodes are present in the linked list, the very first node is called **Head** and the last node is called **Tail**. If there is just one node created then it is called both head and tail.

A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.

Applications of linked list in real world:

1. Image viewer – Previous and next images are linked, hence can be accessed by next and previous button.
2. Previous and next page in web browser: the access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
3. Music Player: Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements. If you want to access a particular item then you have to start at the head and follow the references until you get to that item.

Linked List vs. Arrays:

Advantages over arrays
1) Dynamic size
2) Ease of insertion/deletion

Drawbacks(disadvantages):
1) Random access is not allowed. It have to access elements sequentially starting from the first node. So it cannot do binary search with linked lists efficiently with its default implementation.
2) Extra memory space for a pointer is required with each element of the list.

Insert a Node at the beginning of a Linked list:

- First we need to define node structure.

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    struct Node *link;
};
Node *head = NULL; //Create a head pointer.
```

- Then implement the Insert function.

```
void insert(int new_data) {
    //Here we create & reserve a place in memory for the new node.
Node *new_node= new Node();
new_node->data = new_data; //fill the new_node with data.
new_node->link = head;   //make the head node linked to the new_node.
head = new_node; //the new_node will be the head now.
}
```

To print the whole list, we need to implement the display function as follows:

```
void display() {
    struct Node *ptr;
    ptr = head;
    while (ptr != NULL) {
        cout<< ptr->data <<" "<<"\n";
        ptr = ptr->link;
    }
}
```

To delete a node at the beginning of linked list, we need to implement the delete function as follows:

```
void del(){
    if (head == NULL){
        cout<<"The list is empty\n";
        exit(0);
    }
    else{
    //define ptr to point to the head
    Node *ptr=head;
//then make the head pointer pointing to the next node(the head->link)
    head = head->link;
    //delete the head node
    free(ptr);
    }
}
```

Lab exercise: Implement a linked list and let user enter 5 integer values, then print the list. Once you've done, delete the first two nodes and print the list again.

H.W.) Implement a linked list that contains 10 characters, then make the display function print the upper-case characters only.

- **Implementation of inserting function at the end of a linked list**:

```
void insert_tail(int x){
    Node *ptr;
    ptr=head;
    Node *new_node=new Node;
    new_node->data=x;
    if(ptr==NULL){
        new_node->link =head;
        }
    else{
        while(ptr->link !=NULL){
            ptr=ptr->link;
        }
        ptr->link=new_node;
        new_node->link=NULL;
        }
}
```

- **Implementation of delete function from the end of a linked list.**

```
void del_tail(){
    Node *ptr;
    Node *last;
    ptr=head;
    while(ptr->link !=NULL){
        last=ptr;
        ptr=ptr->link;
    }
    cout<<"Last node is "<<ptr->data<<"\n";
    free(ptr);
    last->link=NULL;
}
```

- To delete the whole list.

```
head=NULL;
```

- **Implementation of delete function of a specific node in a linked list.**

```
void del_pos(int pos){
    Node *ptr;
    Node *before;
    Node *after;
    ptr=head;
    for(int i=0; i<pos-1; i++){
        if(ptr->link !=NULL){
            before=ptr;
            ptr=ptr->link;
        }
    }
    after=ptr->link;
    before->link = after;
    free(ptr);
}
```

Lab exercise: Implement a function that calculate the summation of all even numbers in a linked list.

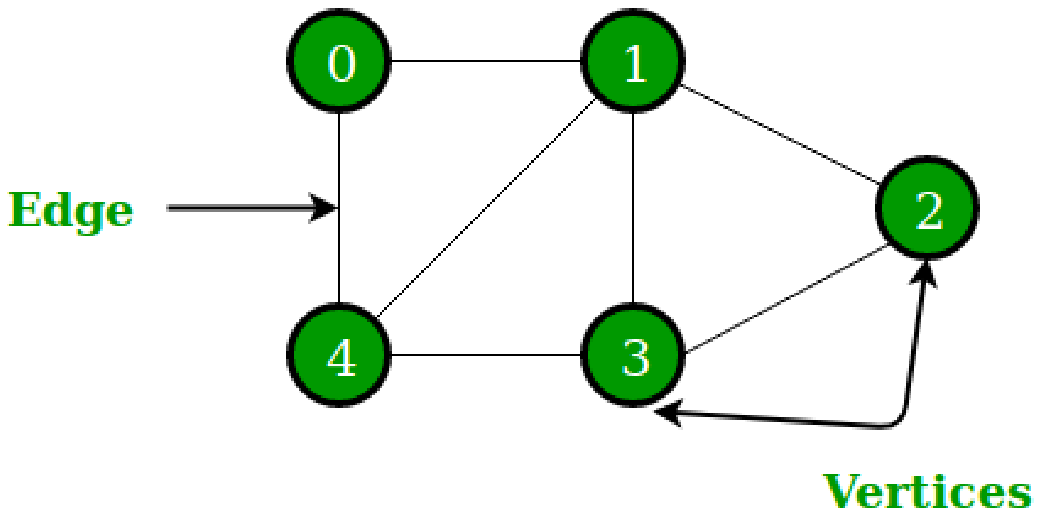
H.W.) Implement a function to find the fifth node in a linked list, then add 50 to its data field.

H.W.) Implement a function to find the number of even and odd numbers in a linked list.
H.W.) Implement a function to find the number of nodes in a linked list.
H.W.) Implement a function to find the number of positive values and negative values in a linked list.

Graphs:

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as:

A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.



In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {{0,1}, {1,2}, {2,3}, {3,4}, {0,4}, {1,4}, {1,3}}. So, graph G=(V,E)
Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

Path: Path represents a sequence of edges between two vertices. In the above example, path= **{(0,1), (1,2)}** represents a path from 0 to 2, and path=**{(0,1), (1,3), (3,2)}** represents the same path.

Graph Representation:

Following two are the most commonly used representations of a graph.
1-Adjacency Matrix: Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.

2-Adjacency List: An array of lists is used. Size of the array is equal to the number of vertices.

- **Fully Connected Graph**

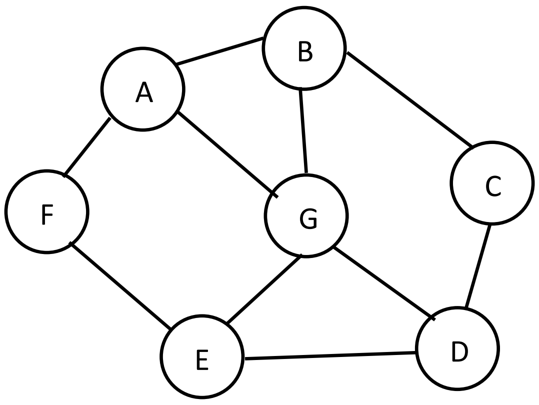
A graph can be considered as a fully connected graph if all its vertices are connected with each other with an edge.

- **Semi Connected Graph**

A graph can be considered as a semi-connected graph if al least one vertex is not connected with another vertex with an edge.

Graphs types:

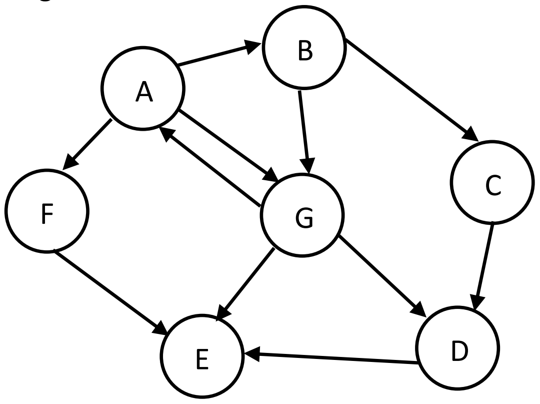
1- **Undirected Graph:** A graph with undirected edges named undirected graph, see the following graph. In this type, the edges are represented as unordered pair {A,G} because the edge is bi-directional



The Adjacency matrix for the above undirected graph is as follows:

	A	B	C	D	E	F	G
A	0	1	0	0	0	1	1
B	1	0	1	0	0	0	1
C	0	1	0	1	0	0	0
D	0	0	1	0	1	0	1
E	0	0	0	1	0	1	1
F	1	0	0	0	1	0	0
G	1	1	0	1	1	0	0

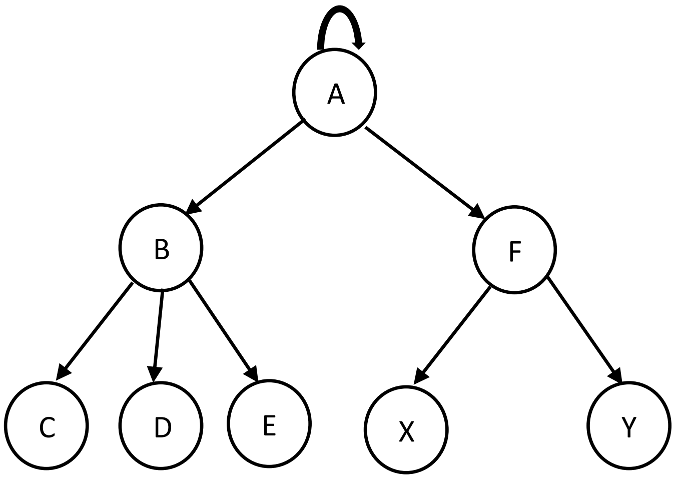
2- **Directed Graph:** A graph with directed edges named directed graph or digraph, see the following graph. Here, the edges are represented as ordered pair (A,G) $\neq$ (G,A) because the edge is un-directional.



The Adjacency matrix for the above directed graph is as follows:

	A	B	C	D	E	F	G
A	0	1	0	0	0	1	1
B	0	0	1	0	0	0	1
C	0	0	0	1	0	0	0
D	0	0	0	0	1	0	0
E	0	0	0	0	0	0	0
F	0	0	0	0	1	0	0
G	1	0	0	1	0	0	0

Example: We have the following graph:



1- Is this graph is directed or un-directed?

2- Is this graph fully or semi connected graph?

3- Path length?

4- Represent the graph using Adjacency Matrix after transforming it to un-directed graph.

H.W.: Represent the graph using Adjacency Matrix.

Tree:

A tree is a collection of entities called nodes. Nodes are connected by edges. Each node contains a value or data, this data can be data of any type. Each node contains a link to some other nodes that can be called its children. Nodes may or may not have a child node .

Tree is a non-linear data structure; it is a hierarchical data structure. The topmost node in the tree is called root of tree.

Now some vocabulary that used for tree data structure.

- **Root**: The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent**: Any node(Y) has one edge upward to another node(X), Node X called parent node.
- **Child**: The node below a given node connected by its edge downward is called its child node.
- **Sibling**: Nodes with the same parent node is called sibling.
- **Leaf**: The node which does not have any child node is called the leaf node.
- **Height**: It is the length of the longest path to a leaf
- **Subtree**: Subtree represents the descendants of a node that could be a separated tree.
- **Levels**: Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **Node degree**: It represents the number of children of that nodes.
- **Tree degree**: The highest node degree of that tree.

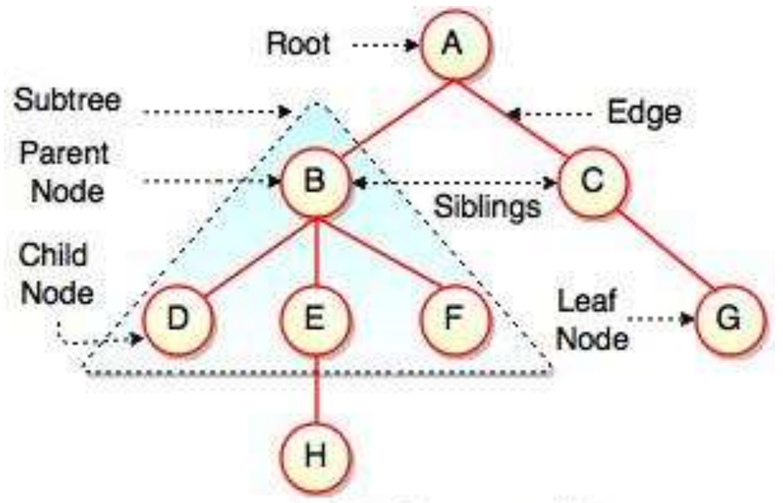
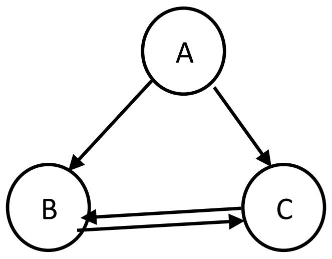
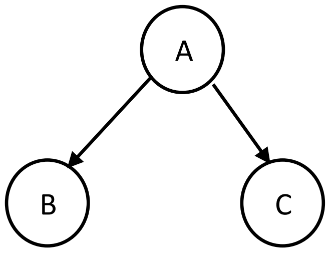


Fig. Structure of Tree

Tree is a graph with no cycle.

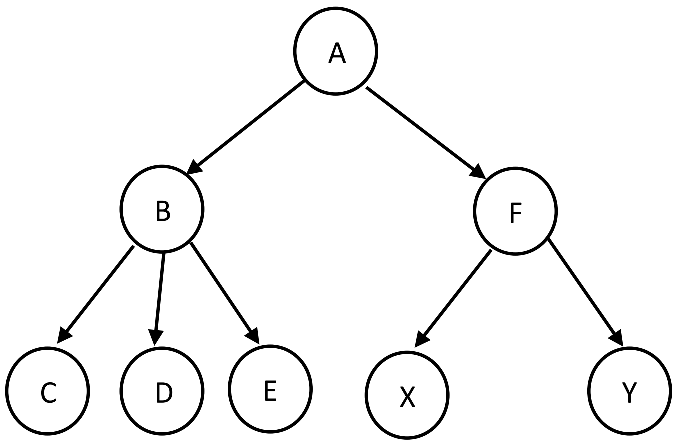


Not a Tree (Graph)



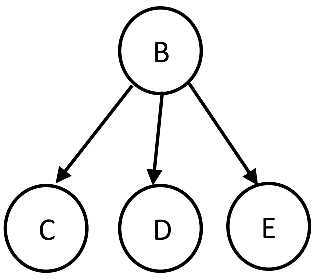
Tree

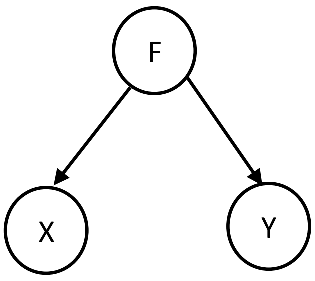
We can convert a graph to a tree by removing all the cycles from a given graph.



Root= A
Leafs= C, D, E, X and Y

Subtree =



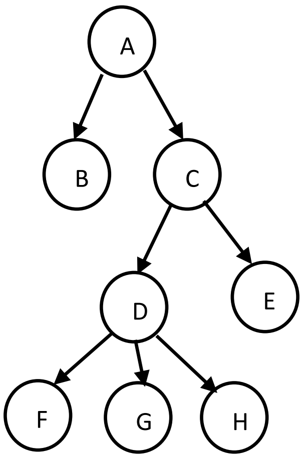


Nodes	Node degree	Level
A		
B		
F		
C		
D		
E		
X		
Y		

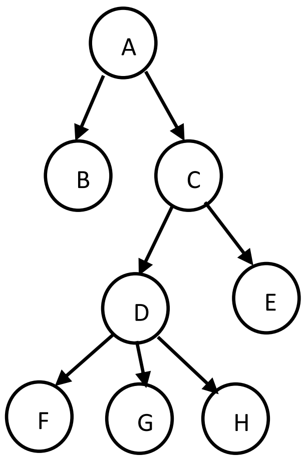
Tree Traverse: the process of visiting (checking and/or updating) each node in a tree data structure, exactly once.

1- Level-by-Level Traversing:
   a. Top-Down Traversing: Traversing starts from left to right for each level.

Top-Down=A B C D E F G H

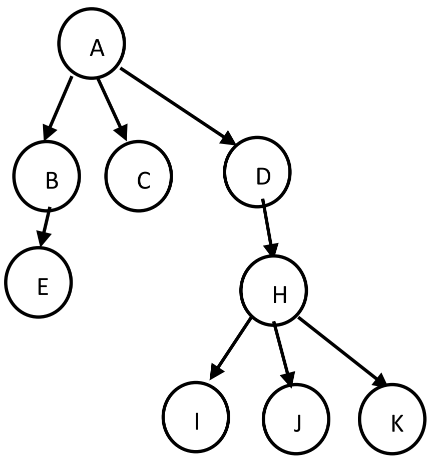


b. Bottom-Up Traversing: Traversing starts from left to right form last level up to the first.



Bottom-Up= F G H D E B C A

NLP Pre-order Traversing: The process is called left-to-right traversal.

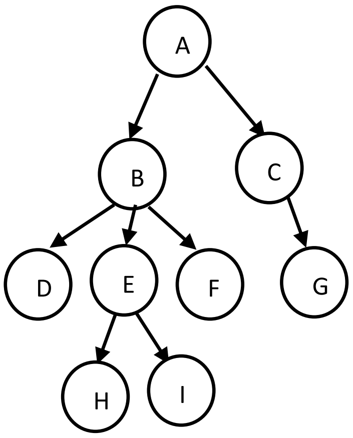


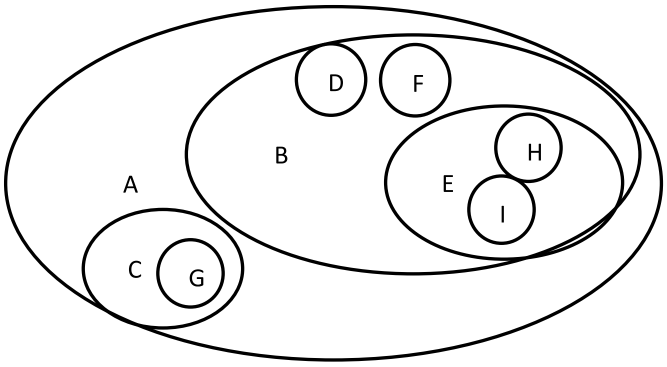
NLP=A B E C D H I J K

Venn Diagram: Is a type of graphic organiser. It is a way of organising complex relationships visually. They allow abstract ideas to be more visible.

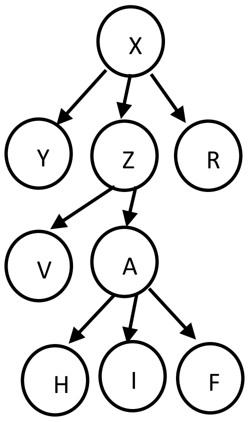
Representing Trees using Venn diagram.

1-



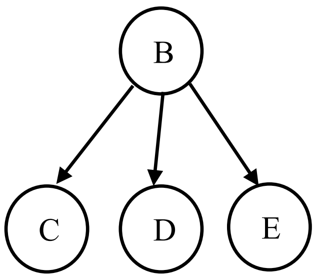


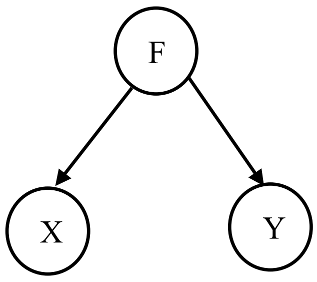
2- What is the Venn diagram for the following tree.



Binary Tree: is a tree data structure in which each node has at most two children, which are referred to as the **left** child and the **right** child.

Binary Tree: is a tree data structure in which each node has at most two children, which are referred to as the **left** child and the **right** child.





Not a binary Tree

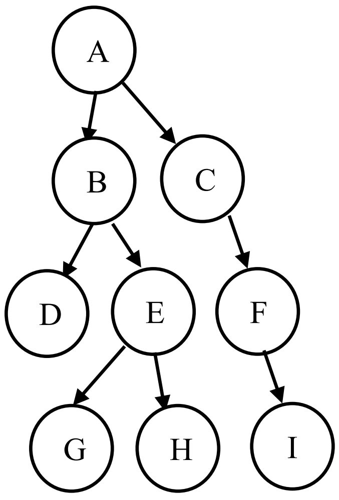
Binary Tree

Array Representation of Binary Tree:

Maximum number of nodes in each level could be calculated as 2^L, where L is the level. So, the maximum number of nodes in level 0 = 2^0=1 and so on.

The maximum number of nodes in a binary tree can be calculated as follows: 2^{h+1}-1, where h is tree height.

Maximum no. of nodes=?

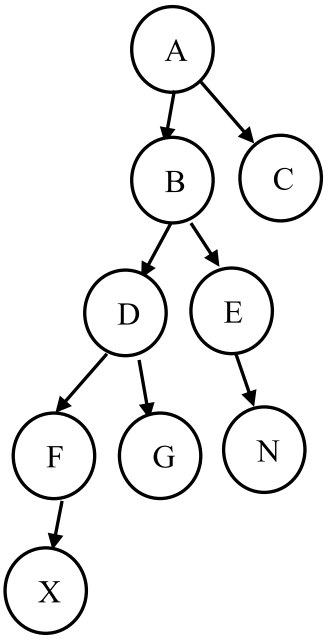


The size of array of a binary tree == maximum number of nodes.

1. Root node stores in index1 of array.
2. Left child located in 2*I in the array, where I is the parents index in array.
3. Right child located in 2*I+1 in the array.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Example1: represent the following tree into an array.



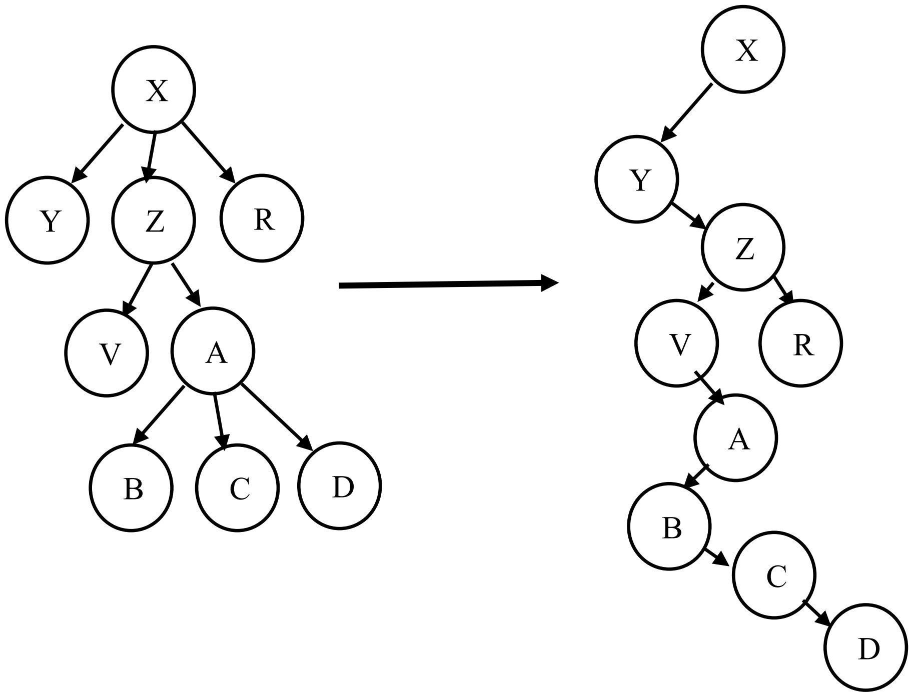
H.W.: Convert the following array into a binary tree.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
L	M	X	A	\o	\o	B	S	R	\o	\o	\o	\o	\o	N	\o	T	P	F	\o	\o	\o	\o	\o	\o	\o	\o	\o	\o	K	\o

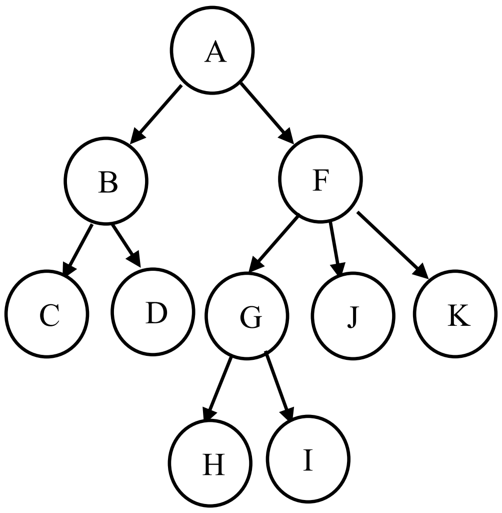
A general tree could be converted into a binary tree using the following algorithm.

1. All node in general tree will be nodes in the binary tree.
2. Root of the general tree is the root of the binary tree.
3. Find branch parent t left most child.
4. Connect sibling of each node **Left** to **Right** child.
5. Delete all edges from the parent node to its children nodes.

Example2: Convert the following tree to a binary tree.



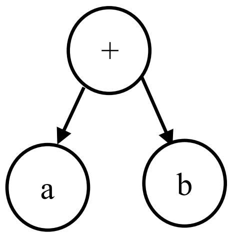
Example3: Convert the following tree to a binary tree.



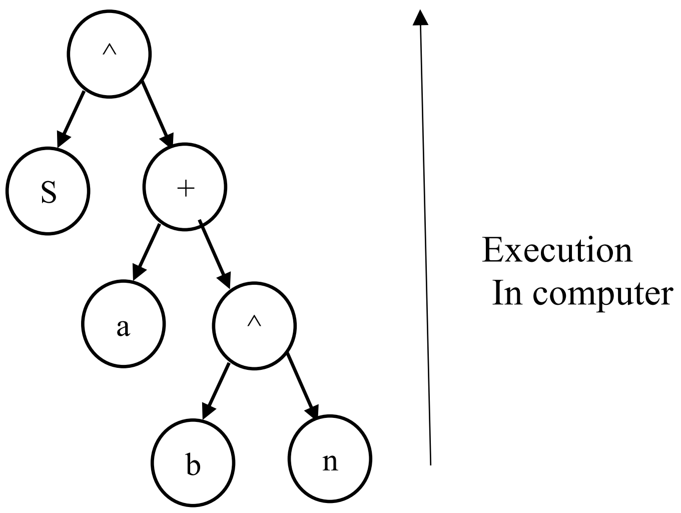
Converting Arithmetic expression into a binary tree:

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand

Example 4: a+b



Example 5: S^(a+b^n)



Example 6: A=B*C+(8+D*E)/(f*2)

Example 7: X=(a+b/3)$^{\wedge 2}$ + 4/7*b

H.W.:**1)** X=2*(a-b/c) **2)** a+b-((c+d)*e) **3)** a+b*(c+d)