WHAT IS DATA STRUCTURE ?

DEFINITION

- A particular way of storing and organizing data in a computer so that it can be used efficiently.
- A special format for organizing and storing data.

TYPES

- ❖Arrays
- ❖linked lists
- ❖Stacks
- ❖Queues
- ❖Trees
- ❖Graphs

TYPES OF DATA STRUCTURE

Depending on the organization of the elements, data structures are classified into two types:-

1) LINEAR DATA STRUCTURES

Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially.

Examples: ***Linked Lists, Stacks and Queues.***

2) NON – LINEAR DATA STRUCTURES

Elements of this data structure are stored/accessed in a non-linear order.

Examples: ***Trees and graphs.***

OBJECTIVES OF DATA STRUCTURES

- It provides an efficient way of storing and organizing data in the computer so that it can be used efficiently.
- Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artificial intelligence, Graphics and many more.
- Main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way.
- It plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.
- It is the building blocks of any program or the software.

REQUIREMENTS OF DATA STRUCTURES

As applications are getting complex and amount of data is increasing day by day, there may arise so many problems.

(1) *Processor Speed*

To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

(2) *Data Search*

Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

(3) *Multiple Requests*

If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process.

To solve these problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

ADVANTAGES OF DATA STRUCTURES

(1) Efficiency

❖*Efficiency of a program depends upon the choice of data structures.*

❖**Example : suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. Hence, using array may not be very efficient here.**

❖**There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.**

(2) Reusability

"Data structures are reusable"

❖**MEANING : Once we have implemented a particular data structure, we can use it at any other place.**

❖**Implementation of data structures can be compiled into libraries which can be used by different clients.**

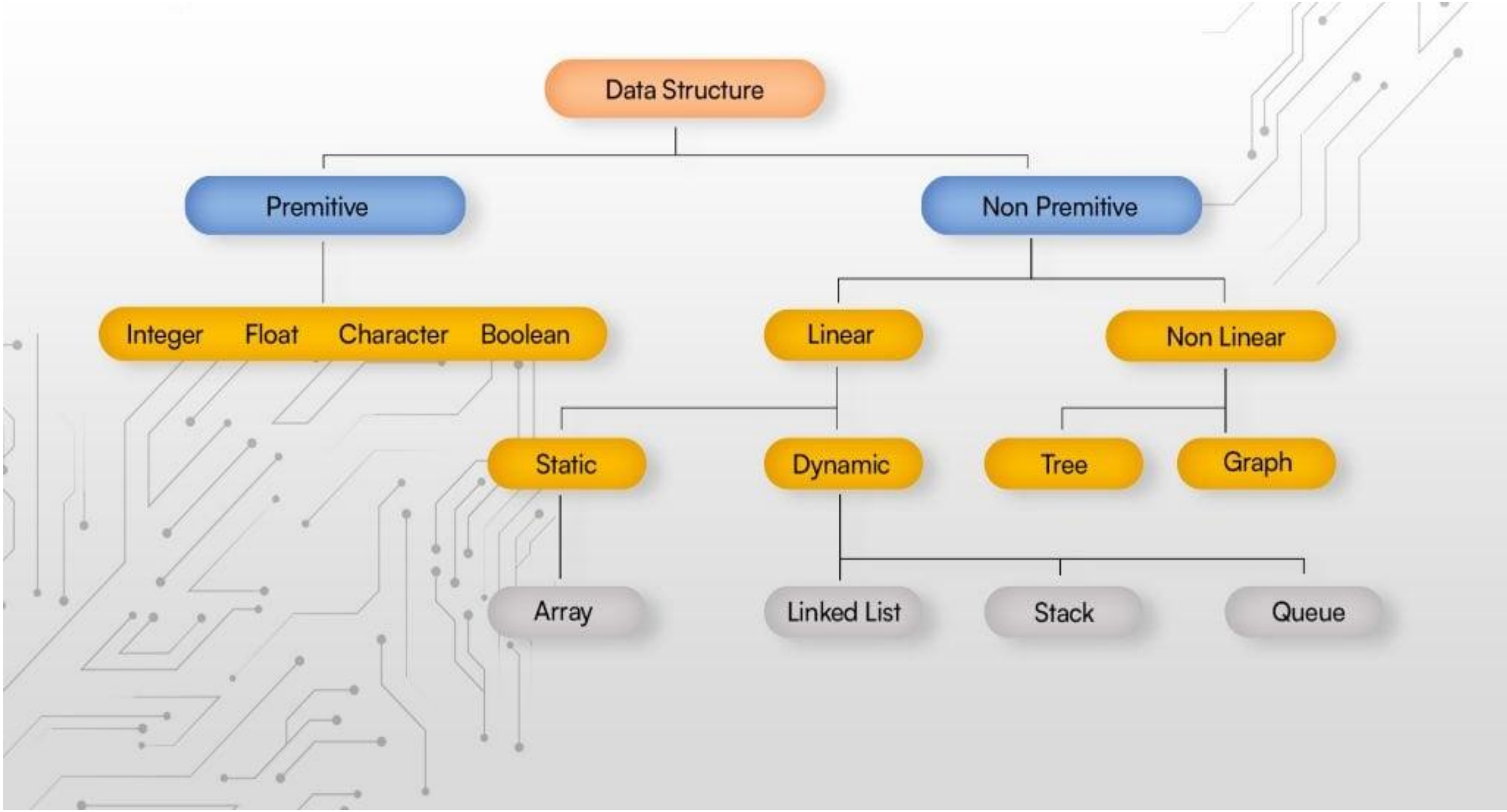
(3) Abstraction

❖*Data structure is specified by the ADT which provides a level of abstraction.*

❖**The client program uses the data structure through interface only, without getting into the implementation details.**

COMPARISON CHART

COMPARISON	LINEAR DATA STRUCTURE	NON-LINEAR DATA STRUCTURE
Basic	The data items are arranged in an orderly manner where the elements are attached adjacently.	It arranges the data in a sorted order and there exists a relationship between the data elements.
Traversing of the data	The data elements can be accessed in one time.	Traversing of data elements in one go is not possible.
Ease of implementation	Simpler	Complex
Levels involved	Single level	Multiple level
Examples	Array, queue, stack, linked list, etc.	Tree and graph.
Memory utilization	Ineffective	Effective


Data Structure
Premitive
Non Premitive
Integer
Float
Character
Boolean
Linear
Non Linear
Static
Dynamic
Tree
Graph
Array
Linked List
Stack
Queue

ARRAY

❖**DEFINITION :-** An array is a collection of similar type of data items and each data item is called an element of the array.

❖**DATA TYPES :-** The data type of the element may be any valid data type like char, int, float or double.

❖**SUBSCRIPT :-** The elements of array share the same variable name but each one carries a different index number known as subscript.

❖**TYPES :-** The array can be one dimensional, two dimensional or multidimensional.

❖**EXAMPLES :**

A[0], A[1], A[2], A[3],..... A[98], A[99].

LINKED LIST & STACK

IILM UNIVERSITY

- A linear data structure which is used to maintain a list in the memory.
- It can be seen as the collection of nodes stored at non-contiguous memory locations.
- Each node of the list contains a pointer to its adjacent node.

- A linear list in which insertion and deletions are allowed only at one end, called top.
- It is an abstract data type (ADT), can be implemented in most of the programming languages.
- It is named as stack because it behaves like a real-world stack.
- For example: - piles of plates or deck of cards etc.

Queue & Graph

❖Queue is a linear list in which elements can be inserted only at one end called rear and deleted only at the other end called front.

❖It is an abstract data structure, similar to stack.

❖Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

❖DEFINITION : The pictorial representation of the set of elements.

❖EDGES : It is represented by vertices connected by the links known as edges.

❖A graph is different from tree in the sense that a graph can have cycle while the tree can not have the one.

TREE

- Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes.
- The bottommost nodes in the hierarchy are called leaf node.
- The topmost node is called root node.
- Each node contains pointers to point adjacent nodes.
- Tree data structure is based on the parent-child relationship among the nodes.
- Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node.

OPERATIONS ON DATA STRUCTURES

1)Traversing :-

- *Every data structure contains the set of data elements.*
- *It means visiting each element of the data structure in order to perform some specific operation like searching or sorting.*

2) Insertion:-

- **DEFINITION** : *The process of adding the elements to the data structure at any location.*
- *If the size of data structure is n then we can only insert n-1 data elements into it.*

3) Deletion :-

- **DEFINITION** : *The process of removing an element from the data structure is called Deletion.*
- *We can delete an element from the data structure at any random location.*
- *If we try to delete an element from an empty data structure then underflow occurs.*

4) Searching :-

The process of finding the location of an element within the data structure.

5) Sorting :-

- *The process of arranging the data structure in a specific order is known as Sorting.*
- *There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.*

6) Merging :-

When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging.

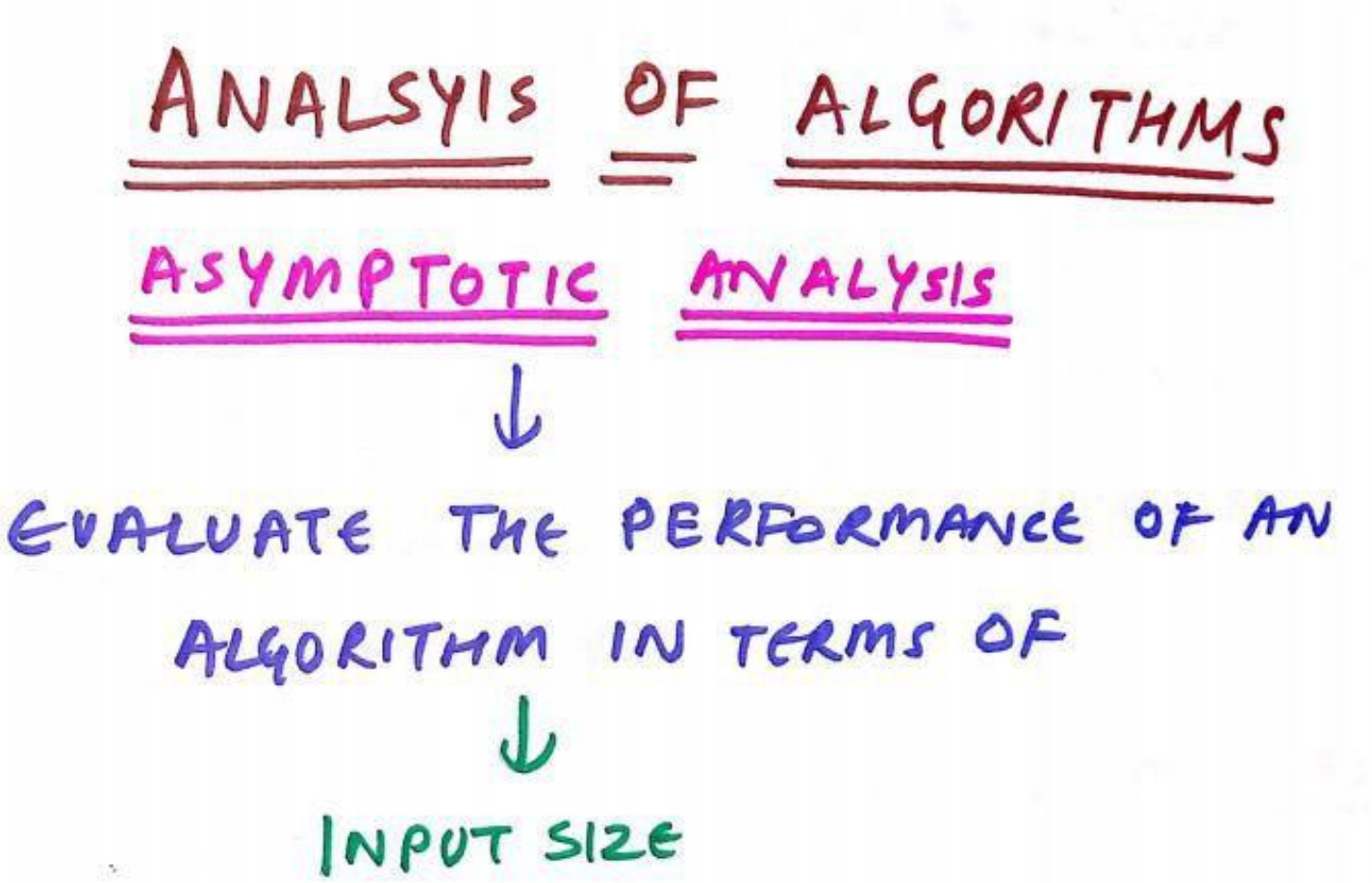
Asymptotic Analysis

Time complexity: It is a way of representing the amount of time needed by a program to run to the completion.

Space complexity: It is the amount of memory space required by an algorithm, during a course of its execution. Space complexity is required in situations when limited memory is available and for the multi user system.

1. In mathematical analysis, asymptotic analysis of algorithm is a method of defining the mathematical boundation of its run-time performance.
2. Using the asymptotic analysis, we can easily conclude about the average case, best case and worst case scenario of an algorithm.
3. It is used to mathematically calculate the running time of any operation inside an algorithm.

Worst case: It defines the input for which the algorithm takes the huge time.
Average case: It takes average time for the program execution.
Best case: It defines the input for which the algorithm takes the lowest time.



Measures actual running time OR Measure how does the time or space taken by an algorithm increases with input size

ASYMPTOTIC NOTATION

1. BIG Oh (O) → WORST CASE → UPPER BOUND
2. Omega (Ω) → BEST CASE → LOWER BOUND
3. Theta (Θ) → AVERAGE CASE → TIGHT BOUND

SINGLE REVISION PAGE - COMPLEXITIES

IILM UNIVERSITY

1. **O(1) – Constant Time Constant time means the running time is constant, it's *not affected by the input size*. (Loop Running constant amount of times)**
2. **O(n) – Linear Time When an algorithm accepts n input size, it would perform n operations.**
3. **O(log n) – Logarithmic Time Algorithm that has running time O(log n) is slight faster than O(n). Commonly, algorithm divides the problem into sub problems with the same size. Example: binary search algorithm, binary conversion algorithm.**
4. **O(n log n) – Linearithmic Time This running time is often found in "divide & conquer algorithms" which divide the problem into sub problems recursively and then merge them in n time. Example: Merge Sort algorithm.**
5. **O(n^2) – Quadratic Time Look Bubble Sort algorithm!**
6. **O(n^3) – Cubic Time It has the same principle with O(n^2).**
7. **O(2^n) – Exponential Time It is very slow as input get larger, if n = 1000.000, T(n) would be 2^1000.000. Brute Force algorithm has this running time.**
8. **O(n!) – Factorial Time THE SLOWEST => Example : Travel Salesman Problem (TSP)**

ARRAY

What is Array ??

An array is a collection of items stored at contiguous memory locations.

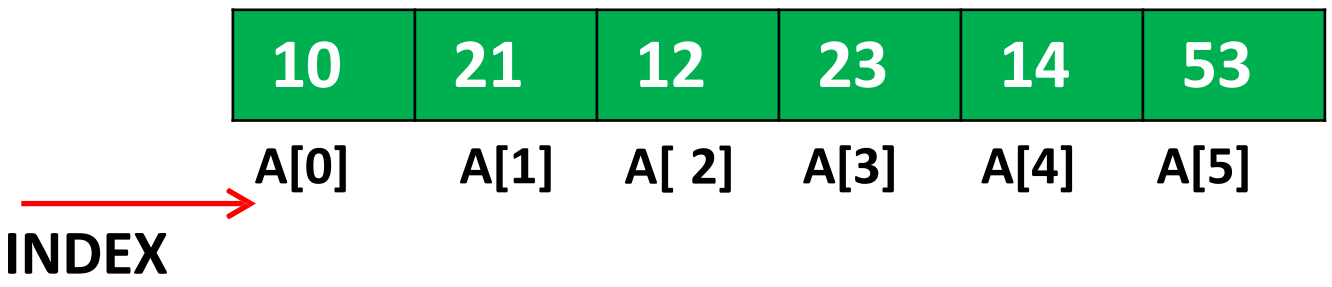
Element – Each item stored in an array is called an element.

Index – Each location of an element in an array has a numerical index, which is used to identify the element.

IMPORTANT POINTS

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index.
- For example, we can fetch an element at index 6 as 9.

Array Representation



HELPS IN EASY CALCULATION

Stores multiple items of the same type together.

This makes it easier to calculate the position of each element by simply adding an offset to a base value.

Each element can be uniquely identified by their index in the array.

Most of the data structures make use of arrays to implement their algorithms.

Arrays can be declared in various ways in different languages.

IILM UNIVERSITY

ARRAY

Operations On Array

1. **Traverse –** print all the array elements one by one.
2. **Insertion –** Adds an element at the given index.
3. **Deletion –** Deletes an element at the given index.
4. **Search –** Searches an element using the given index or by the value.
5. **Update –** Updates an element at the given index.

Space Complexity for worst case is O(n)

Declaration

```
int arr[10]; char arr[10]; float arr[5]
```

Accessing Elements of an array through :

1. Base Address of the array.
2. Size of an element in bytes.
3. Which type of indexing, array follows.

Time Complexity

Algorithm	Average Case	Worst Case
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	O(n)	O(n)
Deletion	O(n)	O(n)

Program

```
#include <stdio.h>
void main ()

{
   int marks[6] = {16,18,28,36,66,72);
   int i;  float avg;
   for (i=0; i<6; i++ )
   {
      avg = avg + marks[i];
   }
   printf(avg);
}
```

Advantages

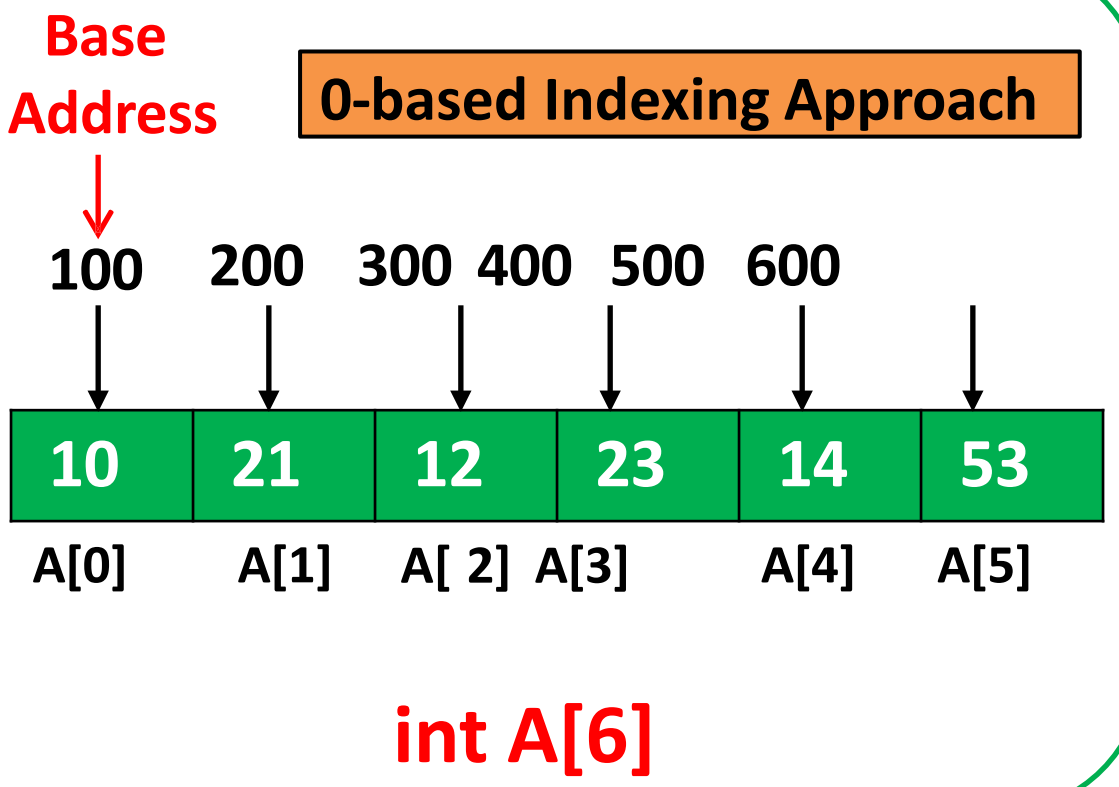
- Array provides the single name for the group of variables of the same type therefore, *it is easy to remember the name of all the elements of an array.*
- *Traversing an array is a very simple process,* we just need to increment the base address of the array in order to visit each element one by one.

•Any element in the array can be directly accessed by using the index.

Memory Allocation

- All the data elements of an array are stored at *contiguous locations* in the main memory.
- The name of the array represents the *base address or the address of first element* in the main memory.
- Each element of the array is represented by a *proper indexing.*

1. 0 (zero - based indexing) : The first element of the array will be arr[0].
2. 1 (one - based indexing) : The first element of the array will be arr[1].
3. n (n - based indexing) : The first element of the array can reside at any random index number.



Accessing Elements of an Array

1. Base Address of the array.
2. Size of an element in bytes.
3. Which type of indexing, array follows.

Array Declaration

- *The base address of the array is 100th byte.*
- *This will be the address of A[0].*
- *Here, the size of int is 4 bytes therefore each element will take 4 bytes in the memory.*
- *In 0 based indexing, If the size of an array is n then the maximum index number, an element can have is n-1. However, it will be n if we use 1 based indexing.*

1.Array declaration by specifying size

```
int arr1[10];
```

2.Array declaration by initializing elements

```
arr[] = { 10, 20, 30, 40 };
```

3.Array declaration by specifying size and initializing elements

```
arr[6] = { 10, 20, 30, 40 };
```

Address Calculation in single (one) Dimension Array:

Array of an element of an array say “A[I]” is calculated using the following formula:

Address of A [I] = B + W * (I – LB)

B = Base Address
W = Storage Size of one element stored in the array (in byte) I = Subscript of element whose address is to be found
LB = Lower limit / Lower Bound of subscript, if not specified assume 0
(zero)

Given the base address of an array B[1300…..1900] as 1020 and size of each element is 2 bytes in the memory. Find the address of B[1700].

Solution:

The given values are: B = 1020, LB = 1300, W = 2, I = 1700

Address of A [I] = B + W * (I – LB)

= 1020 + 2 * (1700 – 1300)

= 1020 + 2 * 400

= 1020 + 800

= 1820 [Ans]

Address Calculation in Double (Two) Dimensional Array:

Addre
ss of an element of any array say "A[I][J]" is calculated in two forms as given:
(1) Row Major System (2) Column Major System

Row Major System:

The address of a location in Row Major System is calculated using the following formula: Address of A [I][J] = B + W * [N * (I – Lr) + (J – Lc)]

Column Major System:

The address of a location in Column Major System is calculated using the following formula:
Address of A [I][J] Column Major Wise = B + W * [(I – Lr) + M * (J – Lc)]
Where,
B = Base address

I = Row subscript of element whose address is to be found
J = Column subscript of element whose address is to be found W = Storage Size of one element stored in the array (in byte)
Lr = Lower limit of row/start row index of matrix, if not given assume 0 (zero)
Lc = Lower limit of column/start column index of matrix, if not given assume 0 (zero)

M = Number of row of the given matrix

N = Number of column of the given matrix

An array X [-15…….10, 15……40] requires one byte of storage. If beginning location is 1500 determine the location of X [15][20].

Solution:

As you see here the number of rows and columns are not given in the question. So they are calculated as:

Number or rows say M = (Ur – Lr) + 1 = [10 – (- 15)] +1 = 26
Number or columns say N = (Uc – Lc) + 1 = [40 – 15)] +1 = 26

(i) Column Major Wise Calculation of above equation

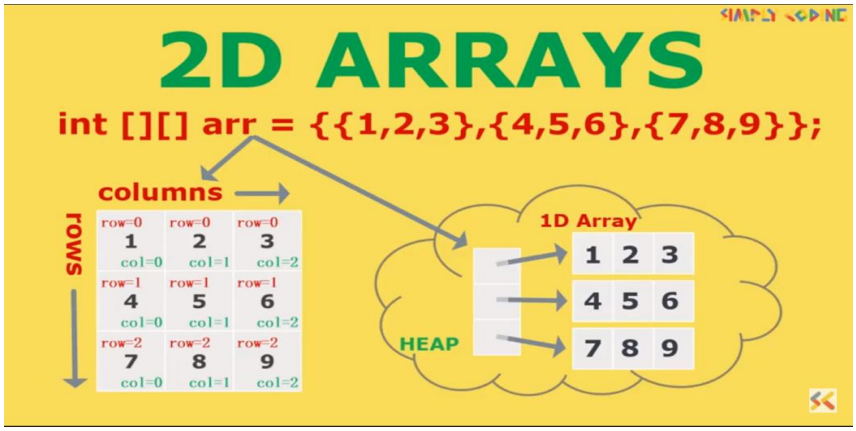
The given values are: B = 1500, W = 1 byte, I = 15, J = 20, Lr = -15, Lc = 15, M = 26

Address of A [I][J] = B + W * [(I – Lr) + M * (J – Lc)]

= 1500 + 1 * [(15 – (-15)) + 26 * (20 – 15)] = 1500 + 1 * [30 + 26 * 5] = 1500 + 1 * [160] = 1660 [Ans]

(ii)Row Major Wise Calculation of above equation
The given values are: B = 1500, W = 1 byte, I = 15, J = 20, Lr = -15, Lc = 15, N = 26

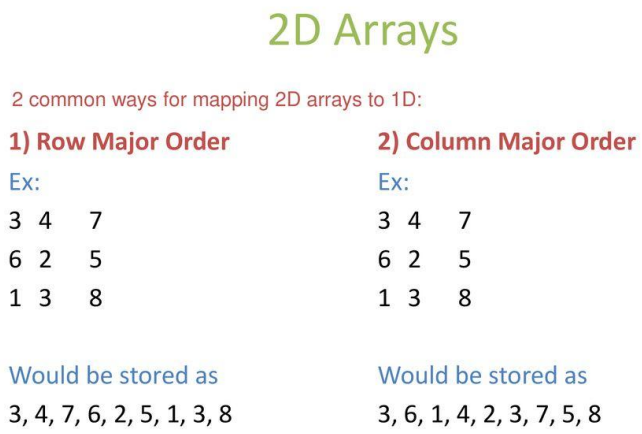
Address of A [I][J] = B + W * [N * (I – Lr) + (J – Lc)]
= 1500 + 1* [26 * (15 – (-15))) + (20 – 15)]
= 1500 + 1 * [26 * 30 + 5] = 1500 + 1 * [780 + 5] = 1500 + 785
= 2285 [Ans]



Column-Major Order:

- Stores all elements of a column in contiguous memory locations, followed by elements of the next column.
- Formula to compute the address of an element A[i][j] in a 2D array:

Address=Base Address+[(j×No. of Rows)+i]×Element Size



Applications of Arrays

Arrays are widely used in programming for various applications:

1. **Storing Data**: Arrays are used to store large amounts of data in an organized manner (e.g., storing student grades).
2. **Matrix Operations**: Arrays are essential for performing operations like addition, subtraction, and multiplication of matrices.
3. **Sorting and Searching**: Algorithms like Bubble Sort, Merge Sort, and Binary Search work efficiently with arrays.

4. **Graphics Processing**: 2D and 3D arrays represent pixels in images or grids in video games.
5. **Dynamic Programming**: Arrays store intermediate results to optimize problems using techniques like memoization.
6. **Data Structures Implementation**: Arrays serve as the underlying data structure for implementing stacks, queues, and heaps.

Sparse Matrices: A Detailed Explanation

A **sparse matrix** is a special type of matrix where most of the elements are zero. This property makes it inefficient to store all the elements directly in a normal array because most of the space would be wasted storing zeros. Instead, we use specialized techniques to store and work with sparse matrices efficiently.

Why Use Sparse Matrices?

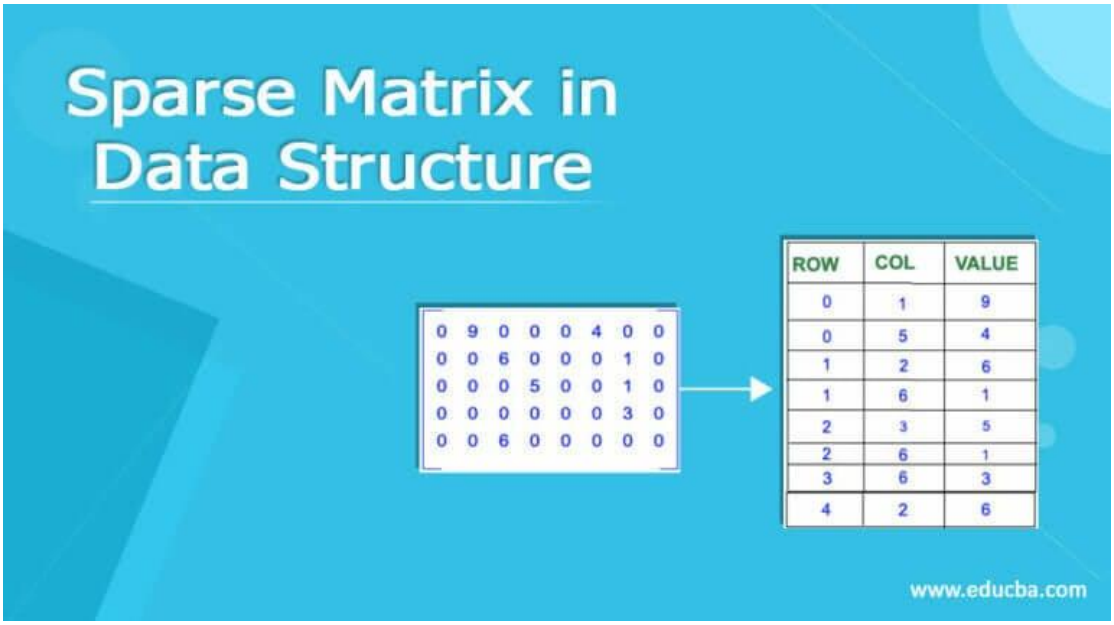
1. **Save Memory**: By storing only non-zero elements, you reduce the memory needed for large matrices.
2. **Speed Up Computations**: Operations on sparse matrices (like addition or multiplication) can skip over zero elements, saving time.
3. **Real-World Scenarios**: Sparse matrices appear in:
    1. Large graphs (adjacency matrices of sparse graphs).
    2. Machine learning (storing datasets with missing values as zeros).
    3. Physics and engineering simulations.

Representation of Sparse Matrices

To store sparse matrices efficiently, we use specific formats. Here are the most common ones:

1. Triplet Representation

- Stores non-zero elements as a list of their row index, column index, and value.
- Format: [row, column, value]



Operations on Sparse Matrices

Efficient operations are designed to take advantage of sparse structures:

Addition:

1. Add only the non-zero elements.
2. Skip positions where both matrices have zero.

Multiplication:

1. Focus only on rows and columns with non-zero elements.
2. Reduces computational complexity significantly.

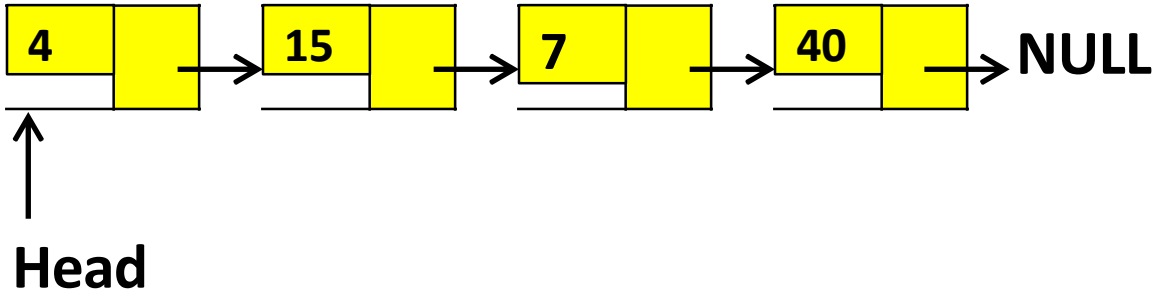
Transpose

1. Switch row and column indices of non-zero elements

What is a Linked List?

A data structure used for storing collections of data.

LINKED LIST



PROPERTIES

- Successive elements are connected by pointers
- The last element points to NULL
- Can grow or shrink in size during execution of a program
- Can be made just as long as required (until systems memory exhausts)
- Does not waste memory space (but takes some extra memory for pointers). It allocates memory as list grows.

Why Linked List is required ?

Both linked lists and arrays are used to store collections of data, and since both are used for the same purpose, we need to differentiate their usage. That means in which cases arrays are suitable and in which cases linked lists are suitable.

What is ADT ?

ADT : Abstract Data Types

Combination of data structures with their operations is called ADT.

Parts of ADT

1. **Declaration of data**
2. **Declaration of operations**

EXAMPLES

Linked Lists, Stacks, Queues, Priority Queues, Binary Trees, Hash Tables, Graphs, etc.

LINKED LIST

Operations make linked lists an ADT

MAIN LINKED LISTS OPERATIONS

- **Insert:** inserts an element into the list
- **Delete:** removes and returns the specified position element from the list

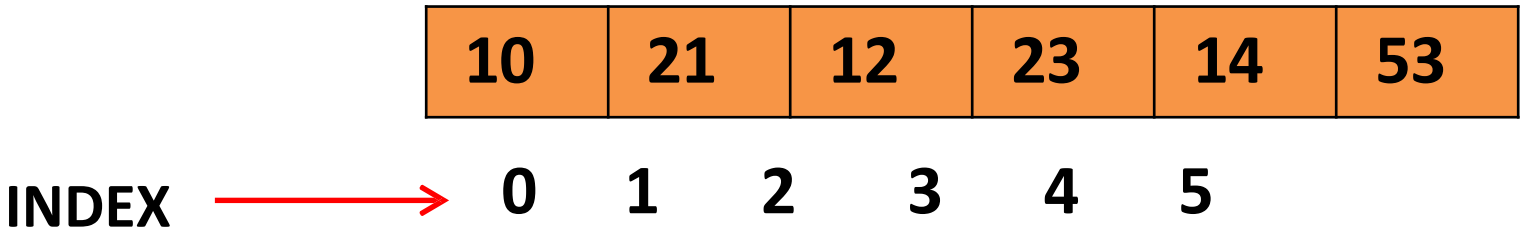
AUXILIARY LINKED LISTS OPERATIONS

- **Delete List :** removes all elements of the list
- **Count :** returns the no of elements in the list
- **Find** nth node from the end of the list

LINKED LIST

ROLE OF ARRAYS

One memory block is allocated for the entire array to hold the elements of the array.
The array elements can be accessed in constant time by using the index of the particular element as the subscript.



Advantages of Arrays

- Simple and easy to use
- Faster access to the elements

Disadvantages of Arrays

- **Preallocates** all needed memory up front and wastes memory space for indices in the array that are empty.
- **Fixed Size :** The size of the array is static.
- **One block allocation :** To allocate the array itself at the beginning, sometimes it may not be possible to get the memory for the complete array.
- **Complex position-based insertion :** To insert an element at a given position, we may need to shift the existing elements. This will create a position for us to insert the new element at the desired position. If the position at which we want to add an element is at the beginning, then the shifting operation is more expensive.

LINKED LIST

LINKED LIST SUPERIOR THAN ARRAYS

LINKED LIST ADVANTAGES

1)Dynamic size

2)Ease of insertion/deletion

3)Expand in constant time

4) We can start with space for just one allocated element and add on new elements
easily without the need to do any copying and reallocating.

LINKED LIST DISADVANTAGES

1) Extra memory space for a pointer is required with each element of the list.

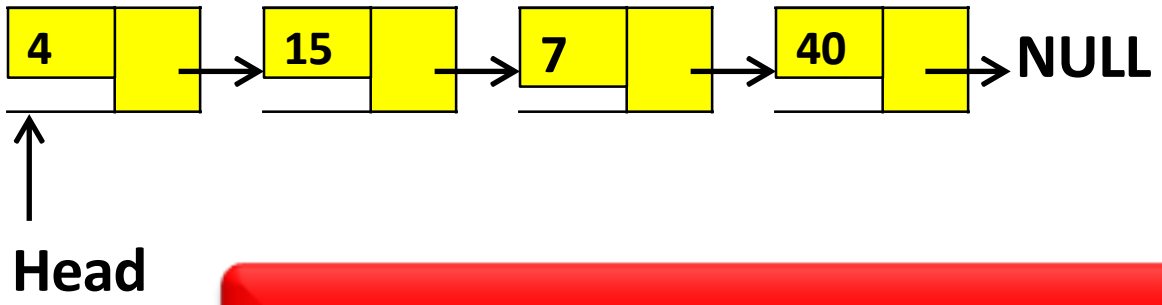
2) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

COMPARISON	ARRAY	LINKED LIST
Basic	It is a consistent set of a fixed number of data items.	It is an ordered set comprising a variable number of data items.
Size	Specified during declaration.	No need to specify; grow and shrink during execution.
Storage Allocation	Element location is allocated during compile time.	Element position is assigned during run time.
Order of the elements	Stored consecutively	Stored randomly
Accessing the element	Direct or randomly accessed, i.e., Specify the array index or subscript.	Sequentially accessed, i.e., Traverse starting from the first node in the list by the pointer.
Insertion and deletion of element	Slow relatively as shifting is required.	Easier, fast and efficient.
Searching	Binary search and linear search	linear search
Memory required	less	More
Memory Utilization	Ineffective	Efficient

SINGLE LINKED LIST (SLL)

S L L

- ❖General Linked List = Single Linked List
- ❖It consists of no. of nodes in which each node has a next pointer to the following element.
- ❖The link of the last node is NULL.



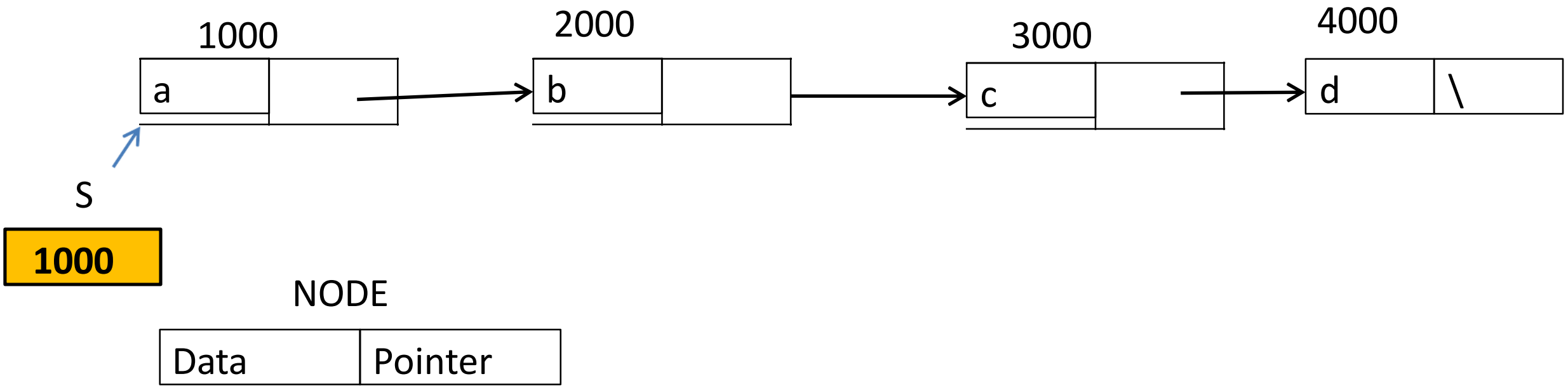
BASIC OPERATIONS

- Traversing the list
- Inserting an item in the list
- Deleting an item from the list

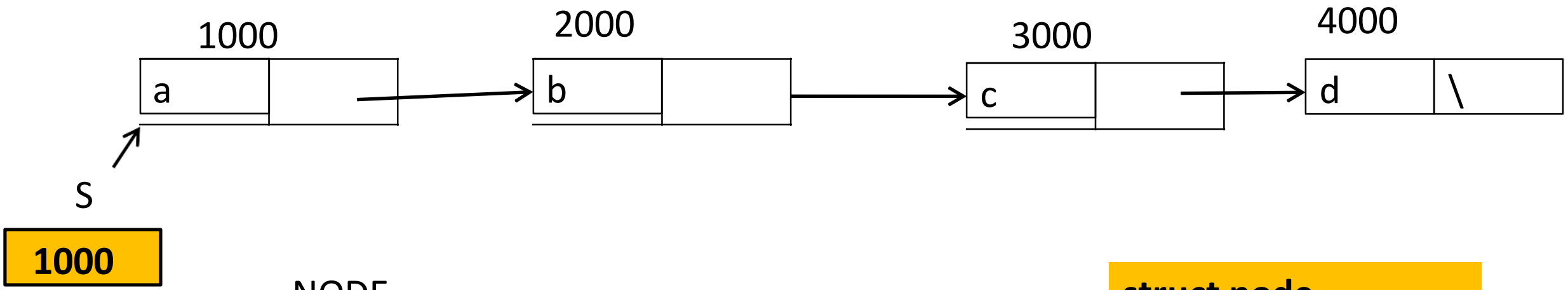
TYPE DECLARATION

```
struct ListNode
{
int  data;
 struct ListNode *next;
};
```

Linked list is a linear data structure which consists group of nodes in a sequence. Each node holds its own data and address of next node, hence forming chain like structures.



Linked list is a linear data structure which consists group of nodes in a sequence. Each node holds its own data and address of next node, hence forming chain like structures.



NODE

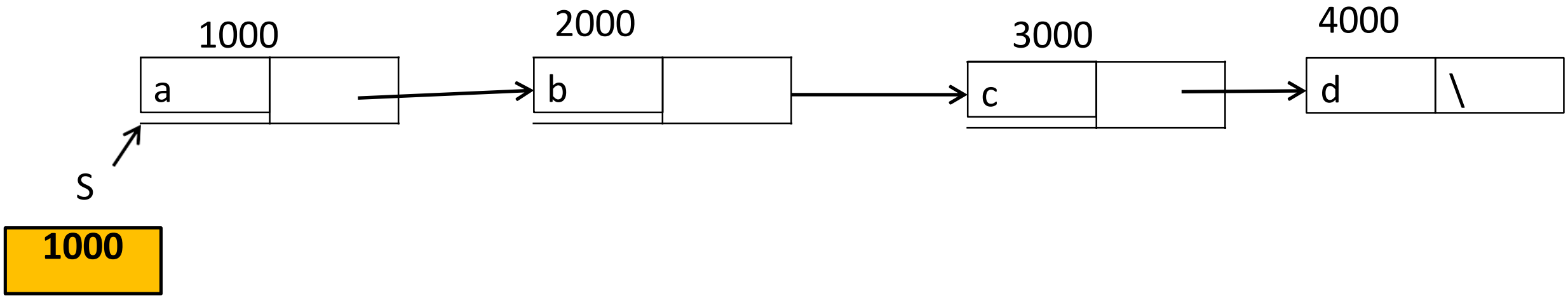
Data	Pointer

```
struct node
{
int data ;
struct node *next ;
}
```

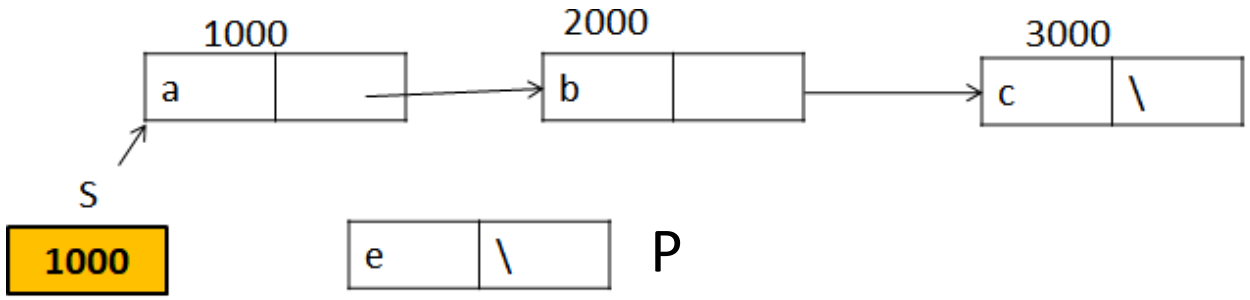
S =1000 , S → data = a , S → Next = 2000

Printing Data of Every Node in Linked List

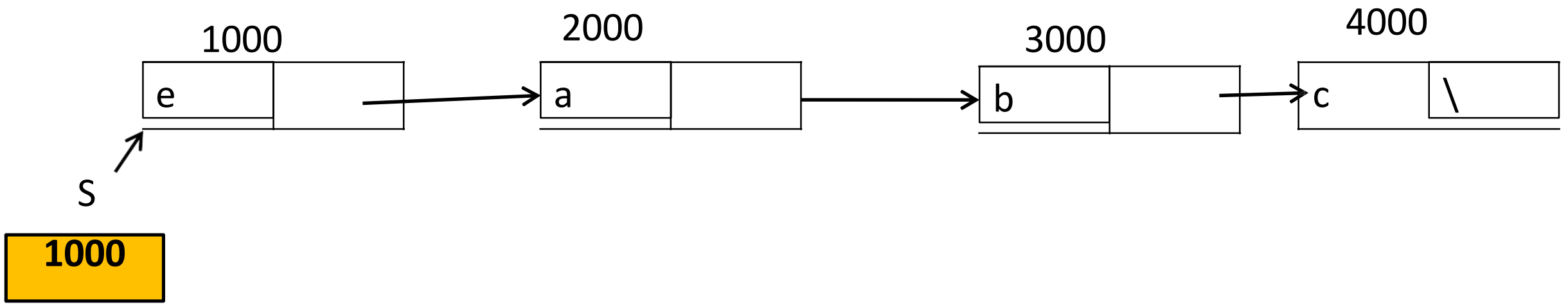
```
Print( Struct Node *S)
{
While (S!= NULL)
{
Printf (“%d” , S -> Data) ; S
= S -> Next ;
}
}
```



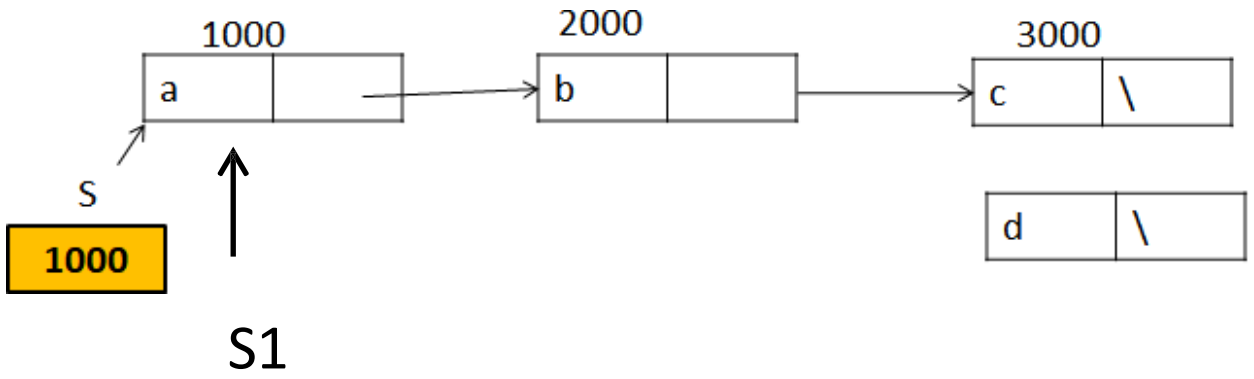
INSERTING NODE AT STARTING OF LINKED LIST



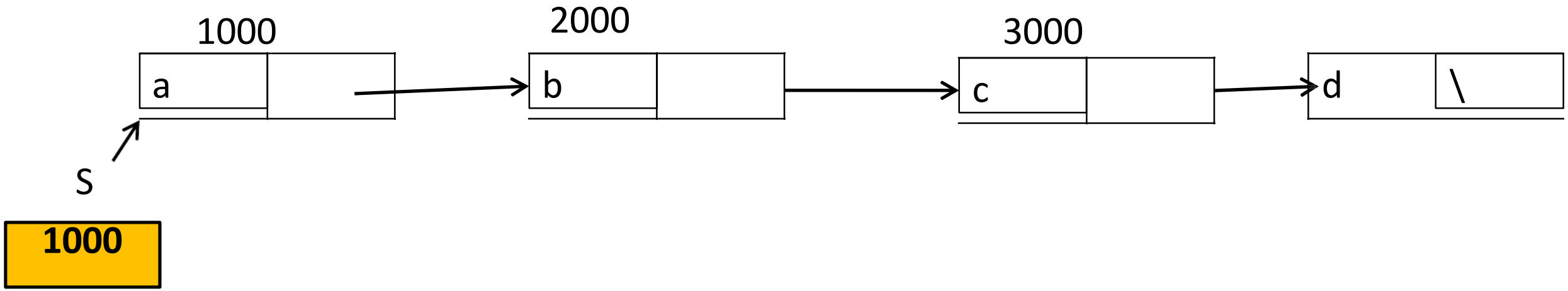

```
Insert_Start (Struct Node *S , Struct
node *P)
{
If (S == null) S = P ;
Else
{
P → next = S ; S = P ;
}
}
```



INSERTING NODE AT END OF LINKED LIST




```
Insert_End (Struct Node *S , Struct node
*P)
{
If (S == null) S = P ;
Else
{
S1 = S;
While (S1 -> next != NULL) S1 =
S1 -> next ;
S1 -> next = P ;
}
}
```



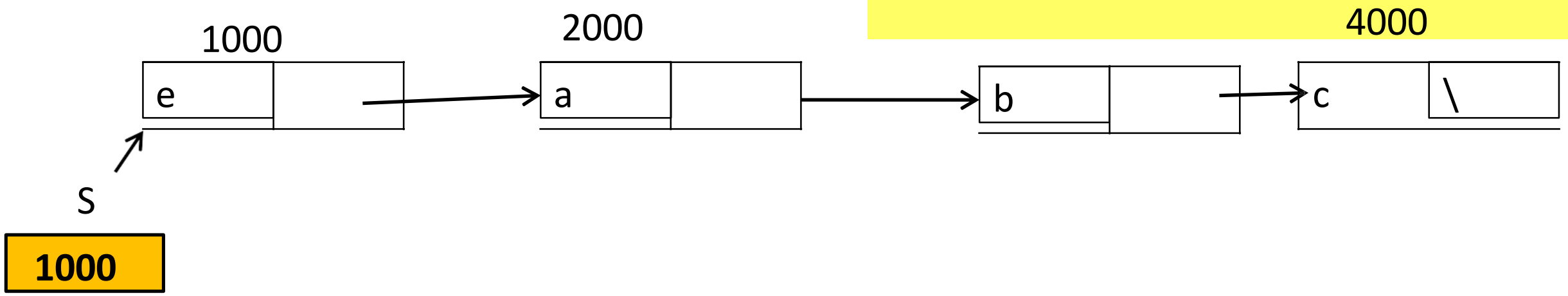
SEARCHING IN LINKED LIST

Search f => Return false

Search b =>

```
search(struct node *s , int x)
{
if (s == null)
return 0 ; else
{
struct node *current = s ;
while (current ! = null)
{
if (current -> data == x)
return true ;
current = current -> next;
}
return false ;
}
```

3000

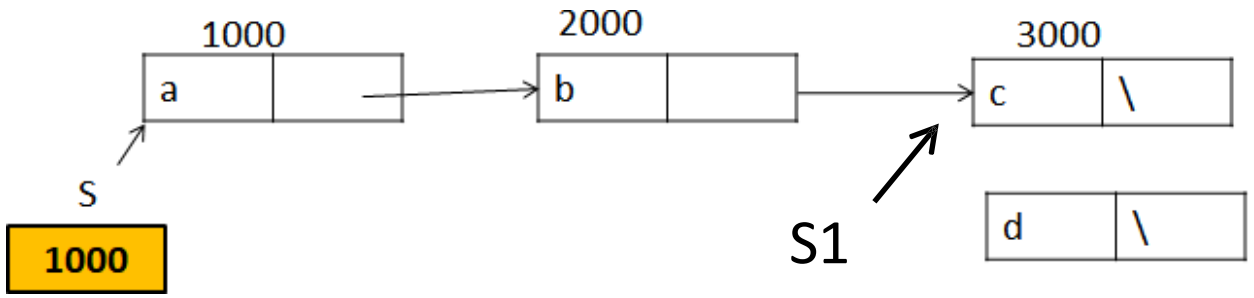


DRAWBACKS OF SINGLE LINKED LIST

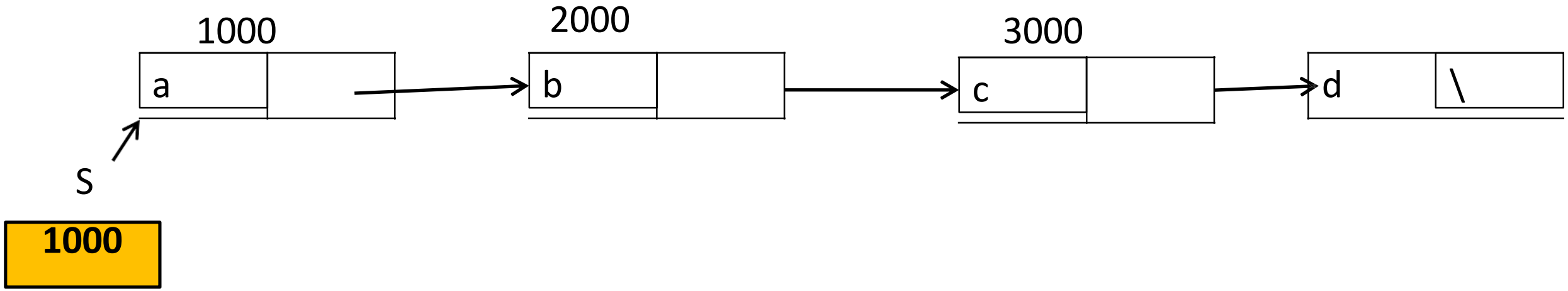
We Cannot go back because every node contain only one pointer and that pointer also contain next node address.

So to eliminate this drawback , We will use Doubly Linked list

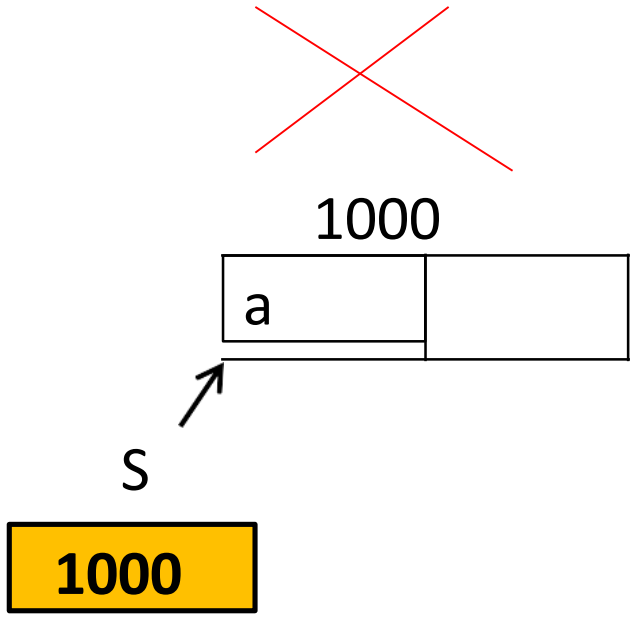
INSERTING NODE AT END OF LINKED LIST




```
Insert_End (Struct Node *S , Struct node
*P)
{
If (S == null) S = P ;
Else
{
S1 = S;
While (S1 -> next != NULL) S1 =
S1 -> next ;
S1 -> next = P ;
}
}
```



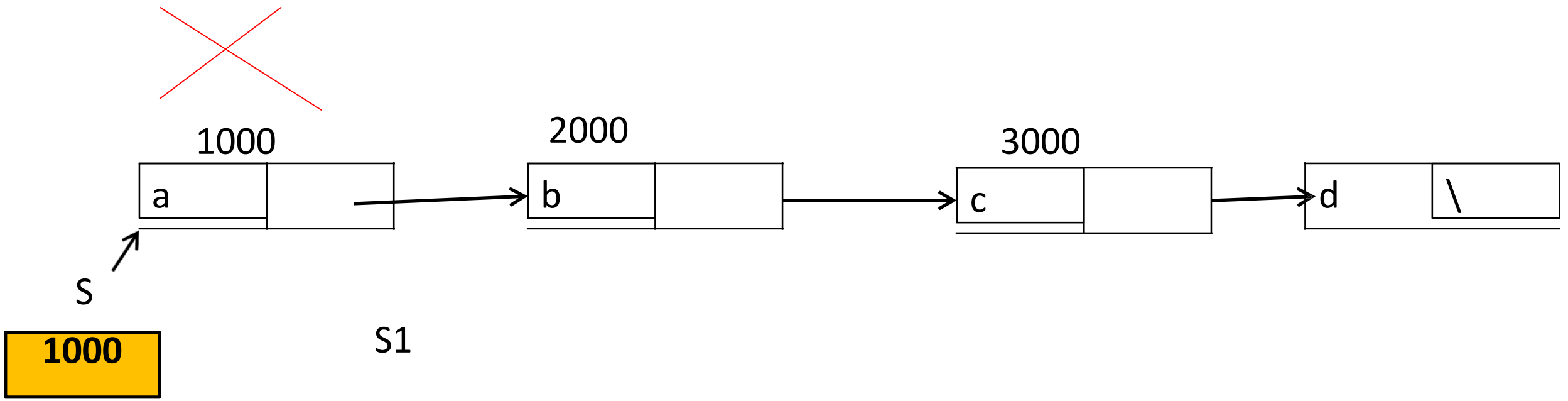
DELETING NODE AT BEGINNING OF LINKED LIST



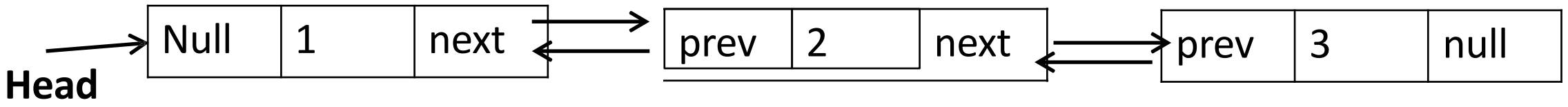

```
Delete_Start (Struct Node *S , Struct
node *P)
{
If (S->next == null)
Free(S);
S = Null ;
}
Else
{
S1 = S-> Next
S-> next = Null ;
Free(S);
S = S1;
}
}
```

DELETING NODE AT BEGINNING OF LINKED LIST

```
Delete_Start (Struct Node *S , Struct
node *P)
{
If (S->next == null)
Free(S);
S = Null ;
}
Else
{
S1 = S-> Next
S-> next = Null ;
Free(S);
S = S1;
}
}
```

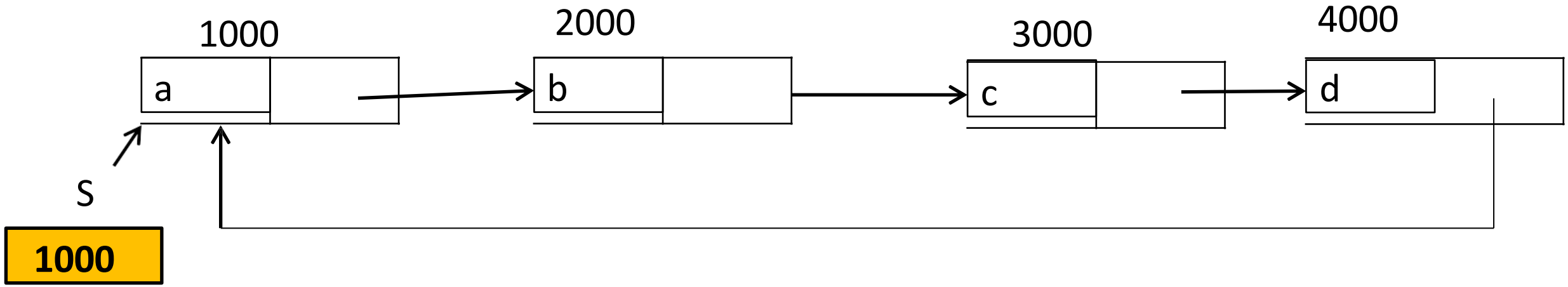


In Doubly linked list , a single node will have 2 pointers and a value . One pointer will point to the next node and other pointer will point to previous node.




```
Struct Node
{
Int data ;
Struct node *next ;
Struct node *prev ;
}
```

CIRCULAR LINKED LISTS



Time complexity for the following operations on Singly Linked Lists of n nodes:

Add a new node to the beginning of list: O(1)
Add a new node to the end: O(n)
Add a new node after k'th node: O(n)
Search a node with a given data: O(n)
Add a new node after a node with a given data: O(n)
Add a new node before a node with a given data: O(n)
Traverse all nodes: O(n)
Delete a node from the beginning: O(1)
Delete a node from the end: O(n)
Delete a node with a given data: O(n)
Delete the k'th node: O(n)
Modify the data of all nodes in a linked list: O(n)

LINKED LIST

COMPLEXITY SRP 5

PARAMETERS	LINKED LIST	ARRAY	DYNAMIC ARRAY
Indexing	O(n)	O(1)	O(1)
Insertion/Deletion at Beginning	O(1)	O(n), if array is not full	O(n)
Insertion at ending	O(n)	O(1), if array is not full	O(1), if array is not full O(n), if array is full
Deletion at ending	O(n)	O(1)	O(n)
Insertion in middle	O(n)	O(n), if array is not full	O(n)
Deletion in middle	O(n)	O(n), if array is not full	O(n)
Wasted space	O(n) (for pointers)	0	O(n)

Singly Linked List Insertion

Insertion into a singly-linked list has three cases:

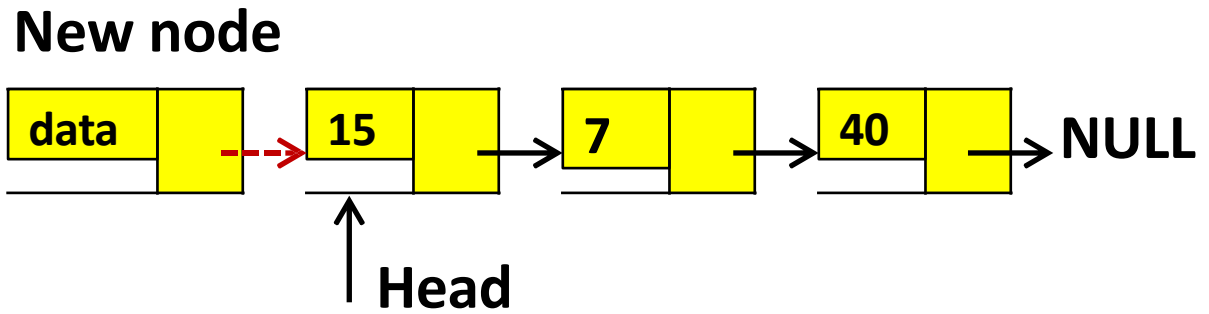
- Inserting a new node before the head (at the beginning)
- Inserting a new node after the tail (at the end of the list)
- Inserting a new node at the middle of the list (random location)

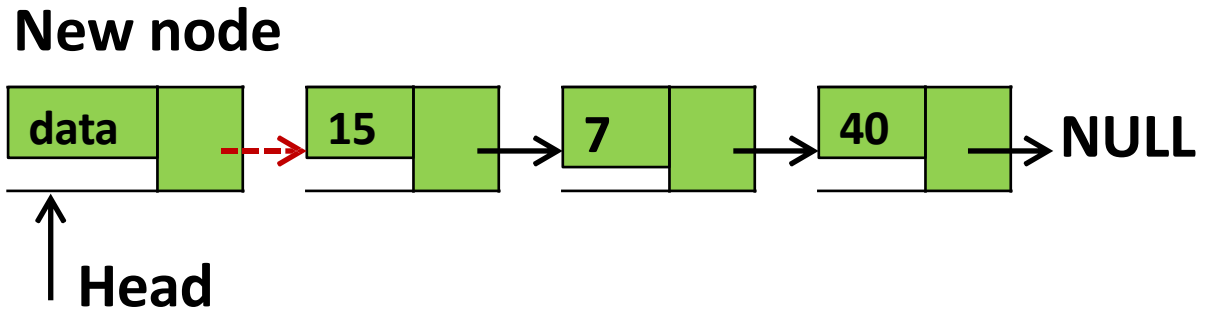
Note: To insert an element in the linked list at some position *p*, assume that after inserting the element the position of this new node is *p*.

INSERTING A NODE IN SLL AT THE BEGINNING

- **Here, a new node is inserted before the current head node.**
- **Only one next pointer be modified.** **STEPS REQUIRED :**

1) **Update the next pointer of new node, to point to the current node.**
2) **Update head pointer to point to the new node.**





INSERTING A NODE IN SLL AT THE ENDING

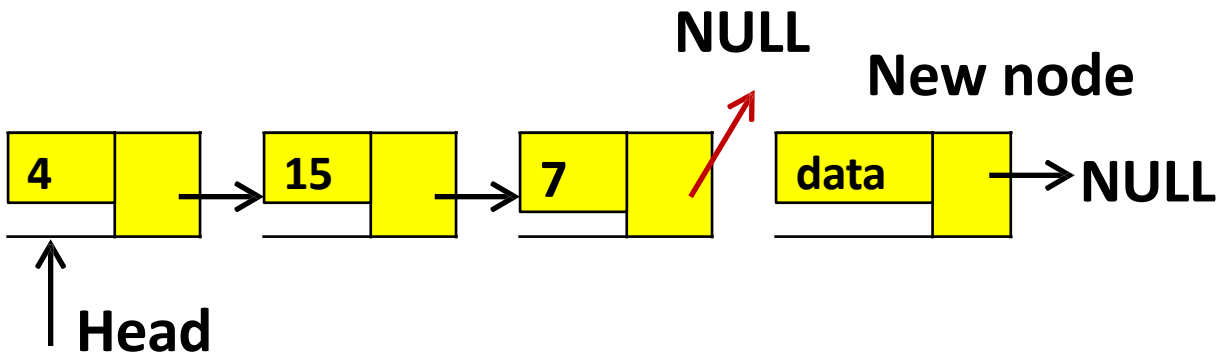
Here, we need to modify two next pointers

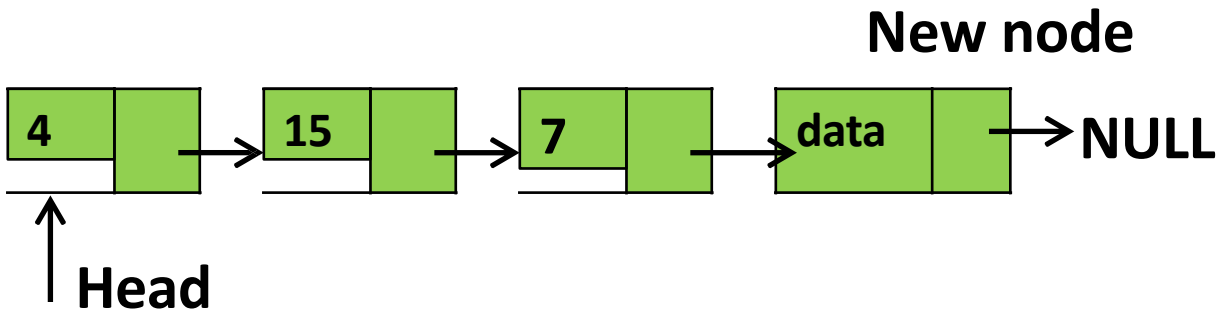
1. Last node's next pointer
2. New node's next pointer

STEPS REQUIRED :

1) New node's next pointer points to NULL.
2) Last node's next pointer points to the new node.

SLL

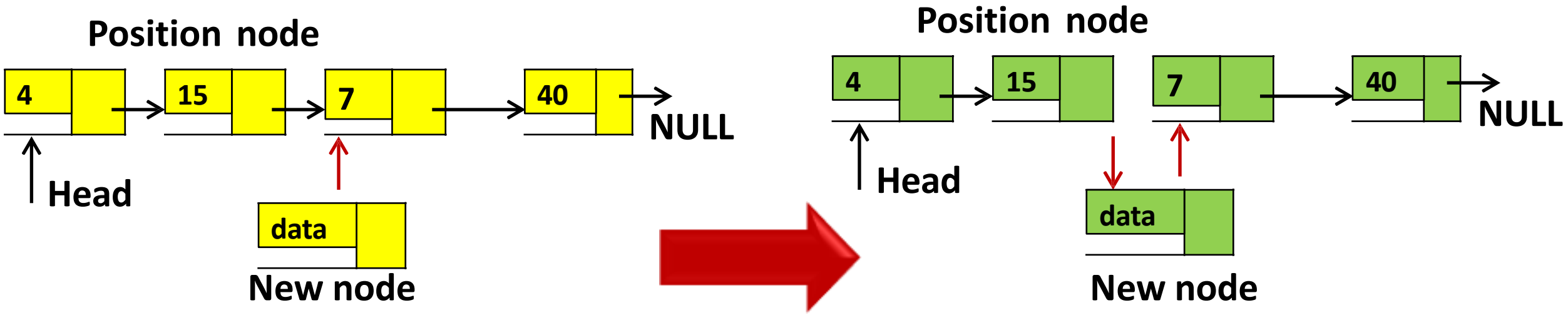




Inserting a Node in Singly Linked List at the Middle

Let us assume that we are given a position where we want to insert the new node. In this case also, we need to modify two next pointers.

- If we want to add an element at position 3 then we stop at position 2. That means we traverse 2 nodes and insert the new node. For simplicity let us assume that the second node is called *position* node. The new node points to the next node of the position where we want to add this node.



Let us write the code for all three cases. We must update the first element pointer in the calling function, not just in the called function. For this reason we need to send a double pointer. The following code inserts a node in the singly linked list.

Singly Linked List Deletion

Similar to insertion, here we also have three cases.

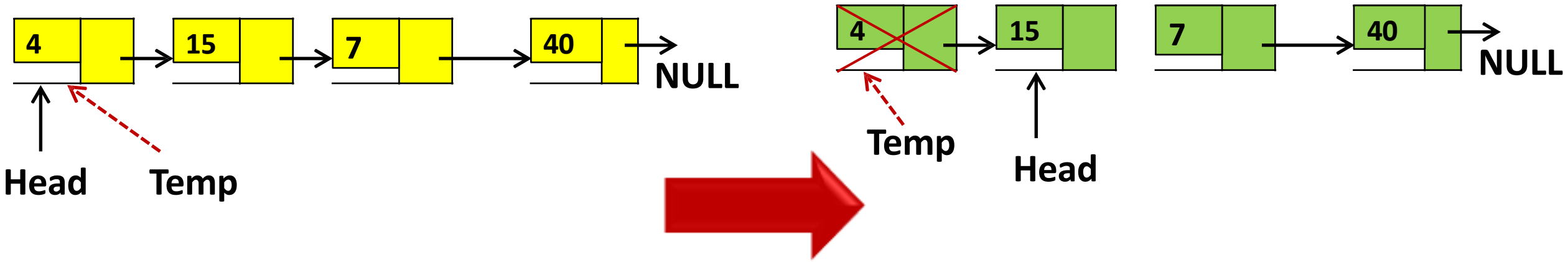
- Deleting the first node
- Deleting the last node
- Deleting an intermediate node.

SLL

Deleting the First Node in Singly Linked List

First node (current head node) is removed from the list. It can be done in two steps:

- Create a temporary node which will point to the same node as that of head.



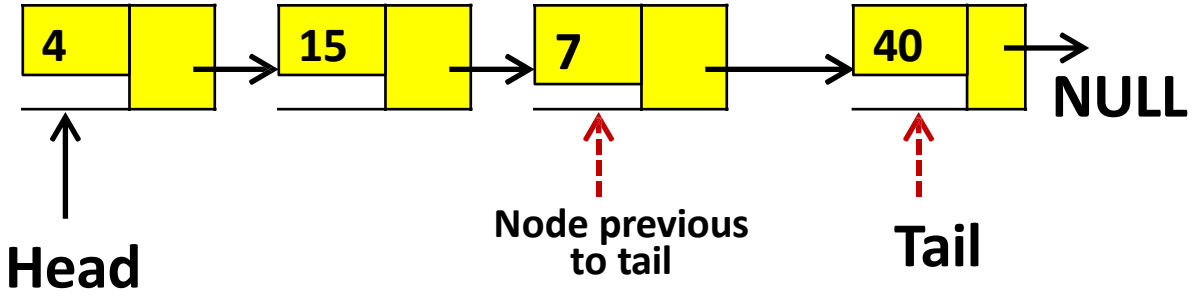
- Now, move the head nodes pointer to the next node and dispose of the temporary node.

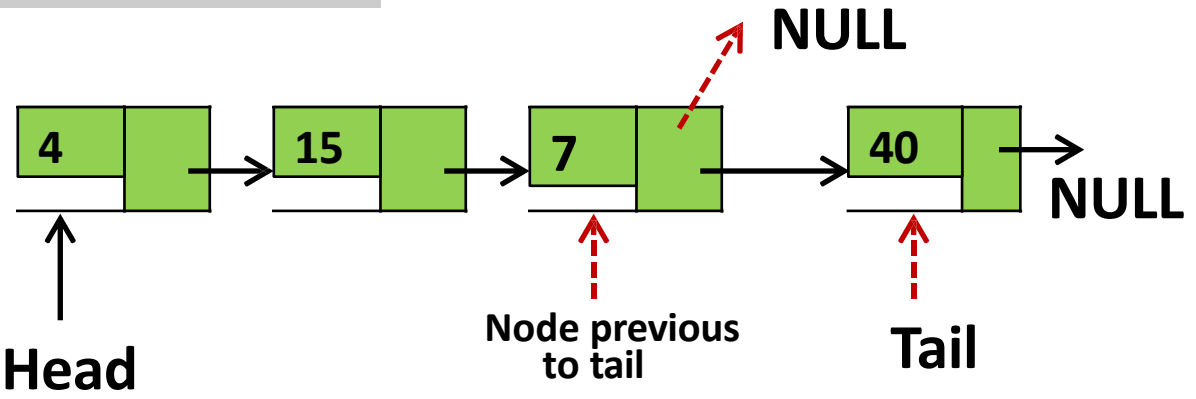
Deleting the Last Node in Singly Linked List

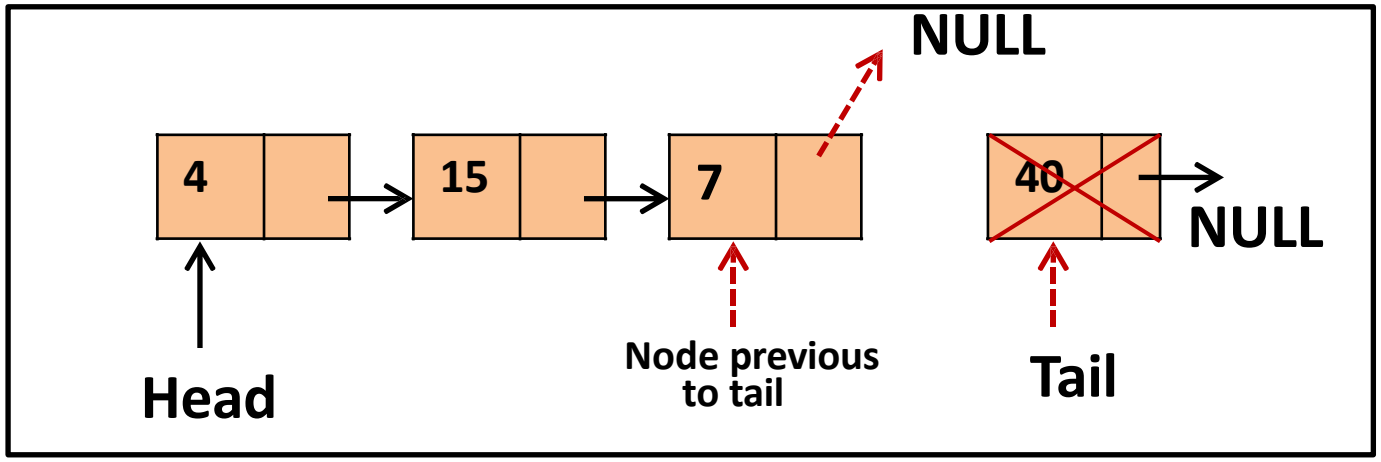
In this case, the last node is removed from the list. This operation is a bit trickier than removing the first node, because the algorithm should find a node, which is previous to the tail. It can be done in three steps:

- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the *tail* node and the other pointing to the node *before* the tail node.

SLL







- Update previous node's next pointer with NULL.

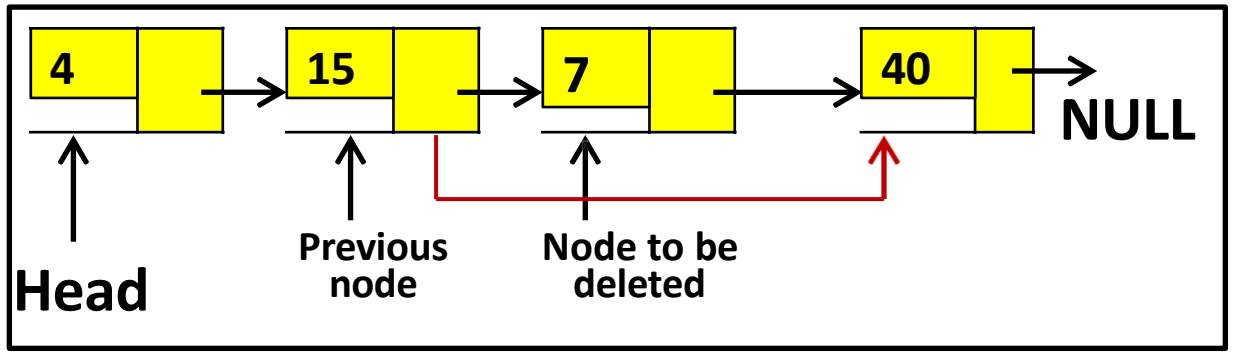
- Dispose of the tail node.

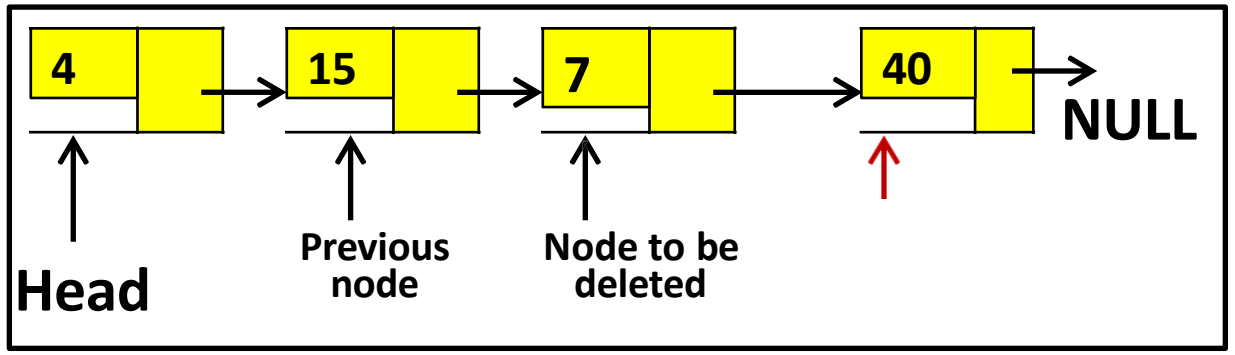
Deleting an Intermediate Node in Singly Linked List

SLL

In this case, the node to be removed is *always located between* two nodes. Head and tail links are not updated in this case. Such a removal can be done in two steps:

- Similar to the previous case, maintain the previous node while traversing the list. Once we find the node to be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.





- Dispose of the current node to be deleted.

Time Complexity: O(n). In the worst case, we may need to delete the node at the end of the list.
Space Complexity: O(1), for one temporary variable.

SLL

Deleting Singly Linked List

This works by storing the current node in some temporary variable and freeing the current node. After freeing the current node, go to the next node with a temporary variable and repeat this process for all nodes.

Time Complexity: O(*n*), for scanning the complete list of size n.
Space Complexity: O(1), for creating one temporary variable.

STACK - DEFINITION

- A stack is an Abstract Data Type (ADT), commonly used in most programming languages.
- It is named stack as it behaves like a real-world stack.
- For example – a deck of cards or a pile of plates, etc.

Programming and Data Structures	1 Marks: 4 2 Marks: 3	Queues, C Programming, Programing Code	Medium	10

STACK

- A real-world stack allows operations at one end only.
- For example, we can place or remove a card or plate from the top of the stack only.
- Likewise, Stack ADT allows all data operations at one end only.
- At any given time, we can only access the top element of a stack.

STACK - WORKING

- This feature makes it LIFO data structure.
- LIFO stands for Last-in-first-out.
- Here, the element which is placed (inserted or added) last, is accessed first.
- In stack terminology,

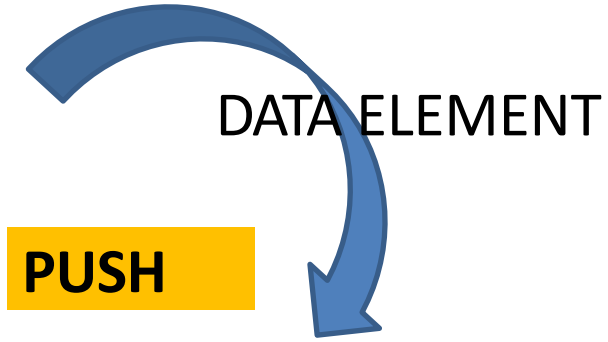
a) Insertion operation is called PUSH operation

b) Removal operation is called POP operation.

STACK REPRESENTATION

- A stack can be implemented by means of Array, Structure, Pointer, and Linked List.
- Stack can either be a fixed size one or it may have a sense of dynamic resizing.
- Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

STACK REPRESENTATION



DATA ELEMENT

PUSH

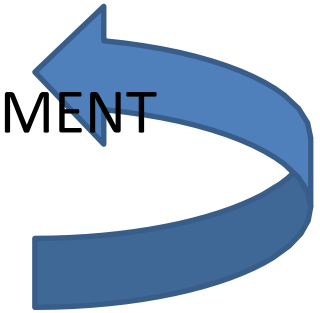
DATA ELEMENT
DATA ELEMENT
DATA ELEMENT
DATA ELEMENT
DATA ELEMENT
DATA ELEMENT

STACK

LAST IN FIRST OUT

DATA ELEMENT

POP



DATA ELEMENT
DATA ELEMENT
DATA ELEMENT
DATA ELEMENT
DATA ELEMENT
DATA ELEMENT

STACK

BASIC OPERATIONS

- At all times, we maintain a pointer to the last Pushed data on the stack.
- As this pointer always represents the top of the stack, hence named top.
- The top pointer provides top value of the stack without actually removing it.

PUSH OPERATION

What is PUSH Operation ?

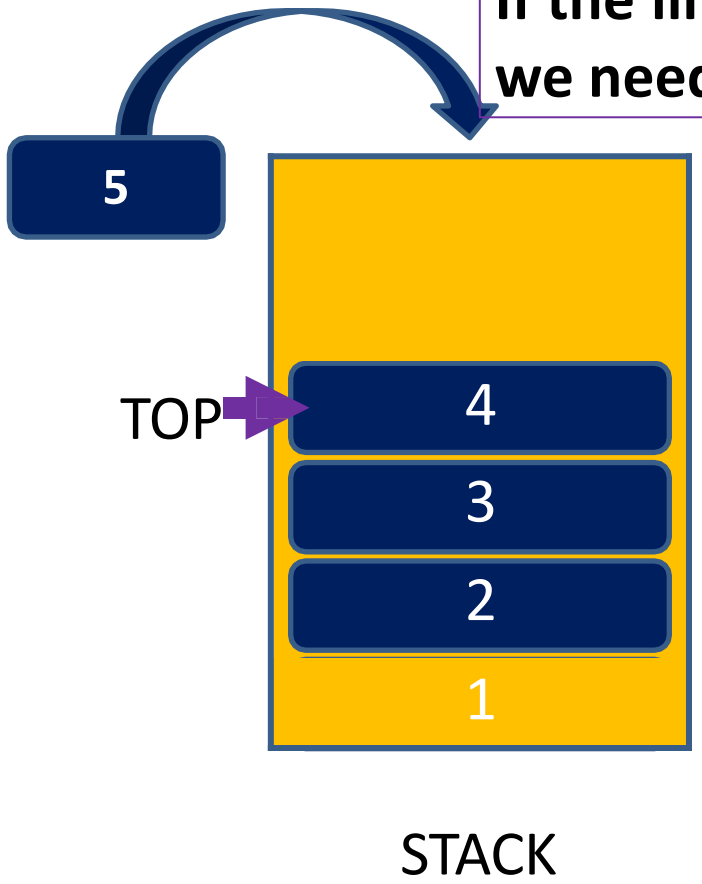
The process of putting a new data element onto stack is known as a Push Operation.

Push operation involves a series of steps –

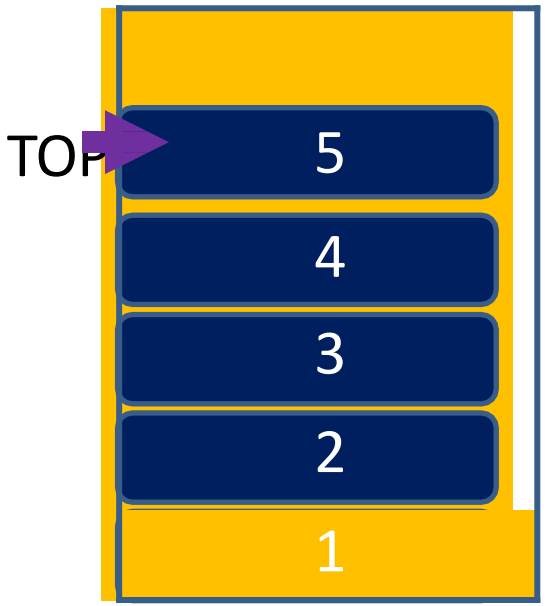
Step 1 – Checks if the stack is full.
Step 2 – If the stack is full, produces an error and exit.
Step 3 – If the stack is not full, increments top to point next empty space.
Step 4 – Adds data element to the stack location, where top is pointing.
Step 5 – Returns success.

PUSH OPERATION

If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.



STACK



STACK

POP OPERATION

- **What is POP Operation ?**

→ ***Accessing the content while removing it from the stack, is known as a Pop Operation.***

- **In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value.**
- **But in linked-list implementation, pop() actually removes data element and deallocates memory space.**

POP OPERATION

A Pop operation may involve the following steps –

Step 1 – Checks if the stack is empty.

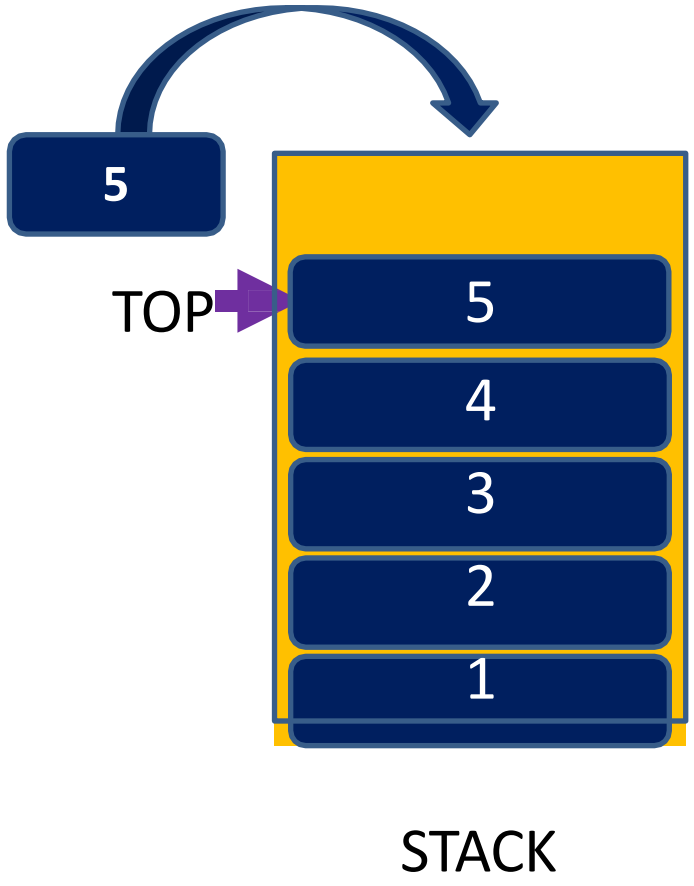
Step 2 – If the stack is empty, produces an error and exit.

Step 3 – If the stack is not empty, accesses the data element at which top is pointing.

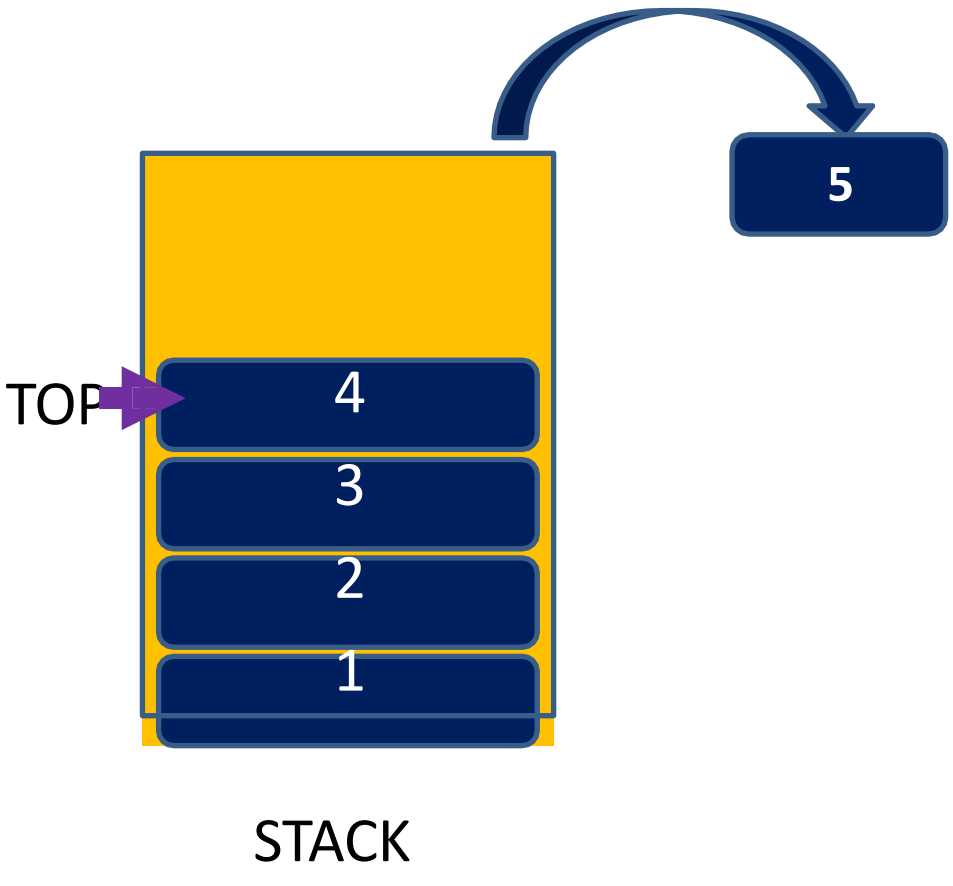
Step 4 – Decreases the value of top by 1.

Step 5 – Returns success.

POP OPERATION



STACK



STACK

STACK

What is Stack ??

Stack = ordered list

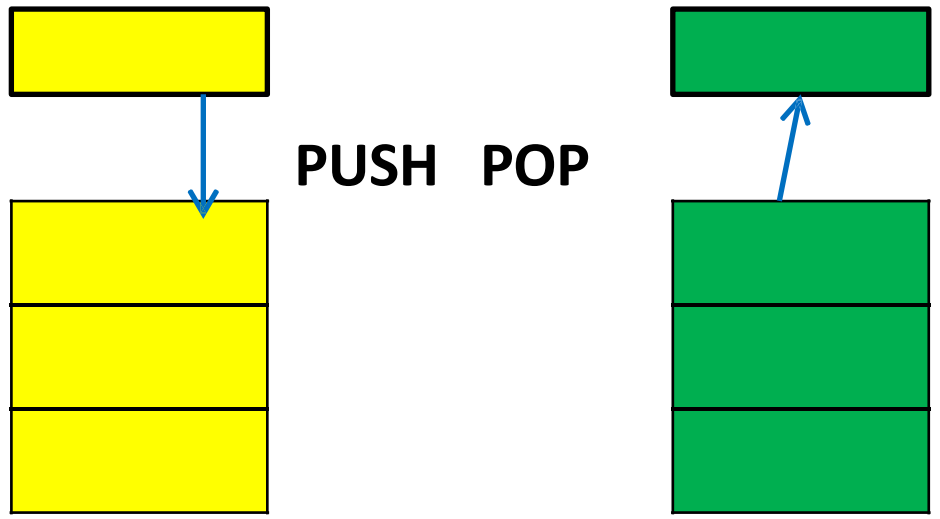
Stack = Insertion and deletion can be performed only at one end that is called top.

Stack = Recursive data structure having pointer to its top element.

Stacks = Last-In-First-Out (LIFO) lists.

Operations on Stack

1. Push : Adding an element onto the stack
2. Pop : Removing an element from the stack
3. Peek : Look all the elements of stack without removing them.



Applications of Stack

1. Recursion
2. Expression evaluations and conversions
3. Parsing
4. Browsers
5. Editors
6. Tree Traversals

GATE 1995

The postfix expression for the infix expression A + B * (C + D) / F + D * E is:

a) AB + CD + *F/D + E*

b) ABCD + *F/+DE*+

c) A *B + CD/F *DE++

d) A + *BCD/F* DE++

A + B * (CD+) / F + DE*

A + B * CD+ F / +DE*

A + BCD+ *F / +DE *

ABCD+ *+F/+ DE *+

ALGORITHM FOR POP OPERATION

Implementation of this algorithm in C, is very easy.

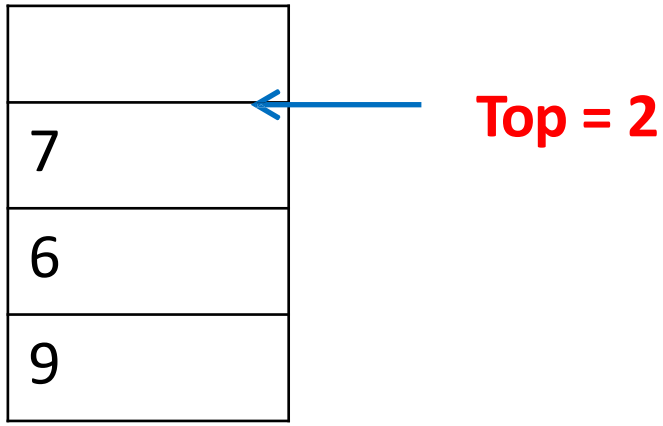
```
begin procedure pop: stack
if stack is empty
return null
endif
data ← stack[top]
top ← top - 1 return
data
end procedure
```

```
int pop(int data)
{ if(!isempty()) { data =
stack[top]; top = top - 1;
return data; }  else {
printf("Could not retrieve data, Stack is empty.\n");
} }
```

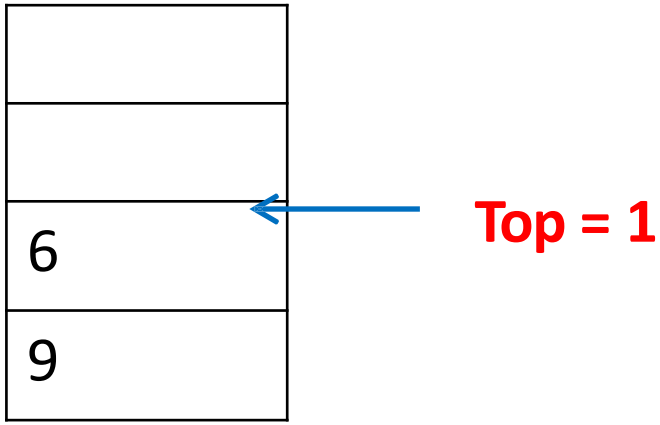
STACK

The stack is called empty if it doesn't contain any element inside it. At this stage, the value of variable top is -1.

Value of top will get increased by 1 every time when we add any element to the stack. In the following stack, After adding first element, top = 2.



Value of top will get decreased by 1 whenever an element is deleted from the stack.
In the following stack, after deleting 7 from the stack, top = 1.



STACK

TOP POSITION	STATUS OF STACK
-1	Empty
0	Only one element in the stack
N-1	Stack is full
N	Overflow

Array implementation of Stack = Stack is formed by using the array.

All the operations regarding the stack are performed using arrays

Adding an element into the top of the stack is referred to as push operation.

Push operation involves following two steps :

1. **Increment the variable Top so that it can now refer to the next memory location.**
2. **Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.**

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

```
begin
   if top = n then stack full
   top = top + 1
   stack (top) : = item;
end
```

Time Complexity : O(1)

```
void push (int val,int n) //n is size of the stack
{
   if (top == n )
   printf("\n Overflow");
   else
   {
   top = top +1;
   stack[top] = val;
   }

}
```

Top=1

Deletion of an element from a stack

(POP OPERATION)

The value of the variable top will be decremented by 1 whenever an item is deleted from the stack.

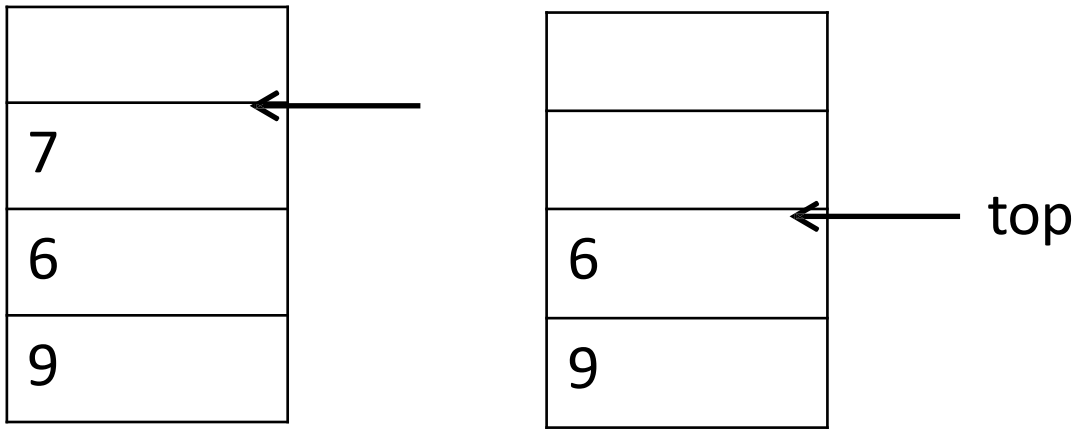
The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm :

```
begin
  if top = 0 then stack empty;
  item := stack(top);
  top = top - 1;
end;
```

Time Complexity : O(1)

```
int pop ()
{
  if(top == -1)
  {
    printf("Underflow");
    return 0;
  }
  else
  {
    return stack[top - - ];
  }
}
```



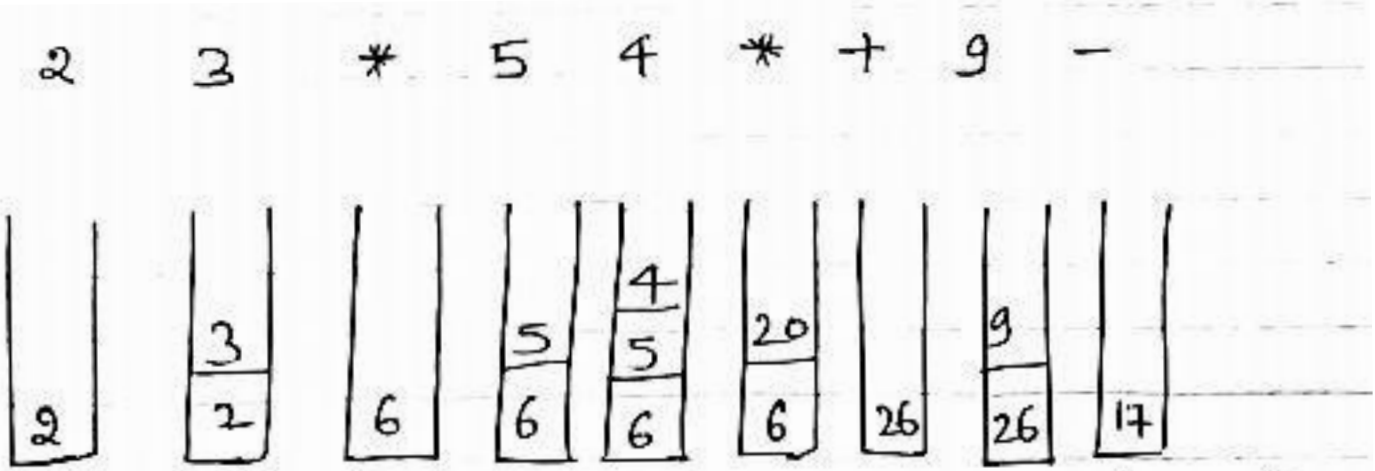
```
Evaluate Postfix (expr)

{ Create a stack S
  for i ← 0 to length (expr)-1

  { if (expr[i] is operand)

        push (expr[i])
    else if (expr[i] is operator)

    { op2 ← pop()
      op1 ← pop()
      res ← Perform (expr[i], op1, op2)
      Push (res)
    }
```



Infix to Postfix

$\{(A*B) + (C*D)\} - e$

$\Rightarrow \{(AB*) + (CD*)\} - e$

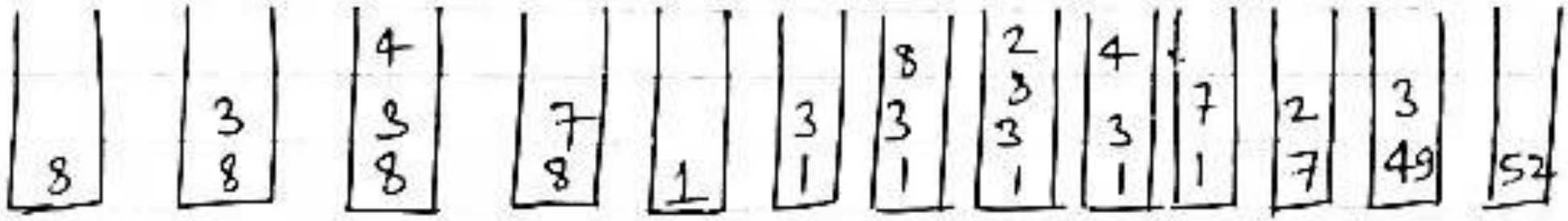
$\Rightarrow \{(AB*)(CD*)+\} - e$

$\Rightarrow \{(AB*)(CD*)+\} e -$

$\Rightarrow AB*CD*+e-$

Value of postfix expression : (STACK)

8 3 4 + − 3 8 2 / + * 2 $ 3 +



Ans = 52

To convert the infix expression
(((A+B) *(C-D))/E)-(F*(G+H)/I+J*K) to postfix notation using a stack,

Step-by-Step Conversion

1. Identify Operators & Precedence
   - Parentheses `()` have the highest precedence.
   - Multiplication `*` and Division `/` have the same precedence.
   - Addition `+` and Subtraction `-` have the lowest precedence.
   - Operators with the same precedence are evaluated from left to right.

2. Break Down the Expression
(((A+B) * (C-D)) / E) - (F * (G+H) / I + J * K)

3. Convert Parentheses First
- (A + B) → AB+

- (C - D) → CD-
- ((A+B) * (C-D)) → AB+CD-*
- (((A+B) * (C-D)) / E) → AB+CD-*E/
- (G + H) → GH+
- (F * (G+H)) → FGH+*
- (F * (G+H) / I) → FGH+*I/
- (J * K) → JK*
- ((F * (G+H) / I) + (J * K)) → FGH+*I/JK*+
- (((A+B) * (C-D)) / E) - ((F * (G+H) / I) + (J * K))
- `AB+CD-*E/FGH+*I/JK*+-`

Final Postfix Expression
AB+CD-*E/FGH+*I/JK*+-

SEARCHING

BINARY SEARCH

LINEAR SEARCH

- Linear Search - SIMPLE & SEQUENTIAL SEARCH
- Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

LS LINEAR SEARCH

ALGORITHM

Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

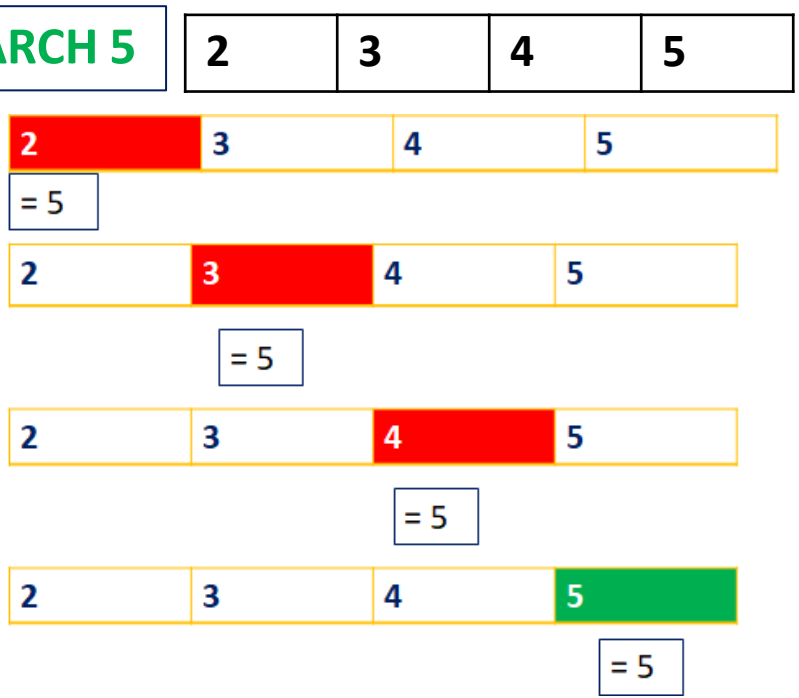
Step 7: Print element not found

Step 8: Exit

Time Complexity :- O(n)

SRP 13

SEARCH 5



Problem: Given an array arr[] of n elements, write a function to search a given element x in arr[].

Solution :

Input : arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}
x = 110;
Output : 6
Element x is present at index 6

Input : arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}
x = 175;
Output : -1

Element x is not present in arr[].

LS LINEAR SEARCH

A simple approach is to do linear search :

1. Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
2. If x matches with an element, return the index.
3. If x doesn't match with any of elements, return -1

```
// code for linearly search x in arr[]. If x
// is present then return its location, otherwise
// return -1
int search(int arr[], int n, int x)
{
  int i;
  for (i = 0; i < n; i++)
      if (arr[i] == x)
       return i;
  return -1;
}
```

Linear search is rarely used practically.

Why ?

Because other search algorithms such as the binary search algorithm and hash tables allow significantly faster searching comparison to Linear search.

BS BINARY SEARCH

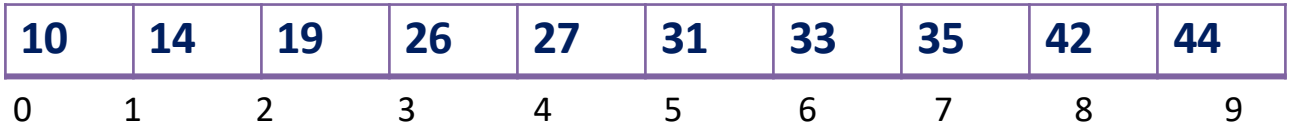
- BS = FAST + O(log n) + DIVIDE-CONQUER + SORTED
- It looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned.
- If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item.
- Otherwise, the item is searched for in the sub-array to the right of the middle item.
- This process continues on the sub-array as well until the size of the subarray reduces to zero.

- The Idea Of Binary Search is to use the information that the array is sorted and reduce the time complexity to O(Log n).
- Compare x with the middle element.
- If x matches with middle element, we return the mid index.
- Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
- Else (x is smaller) recur for the left half.

How BS Works?

- For a binary search to work, it is mandatory for the target array to be sorted.
- The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

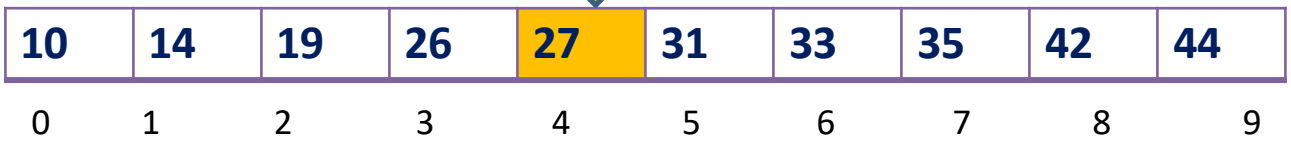
BINARY SEARCH



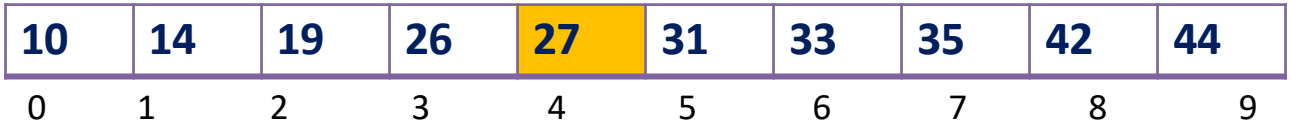
First, we shall determine half of the array by using this formula –

$$mid = low + (high - low) / 2$$

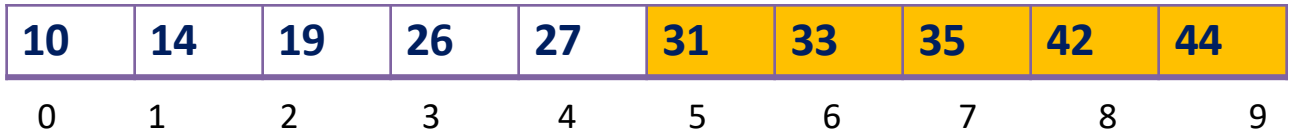
Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



BINARY SEARCH



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match.
As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



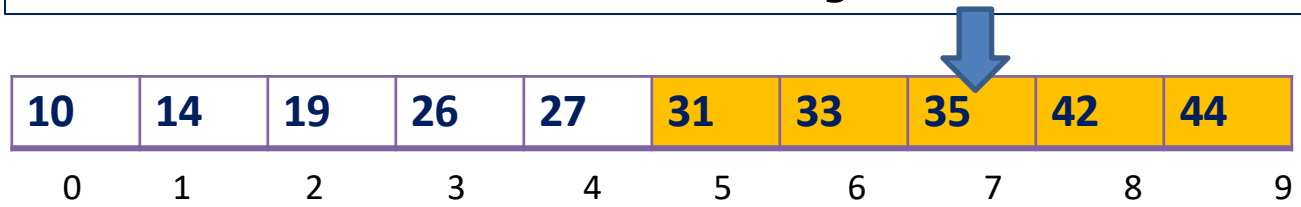
BINARY SEARCH

We change our low to mid + 1 and find the new mid value again.

low = mid + 1

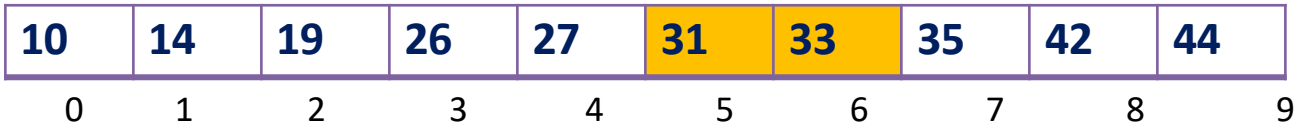
mid = low + (high - low) / 2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

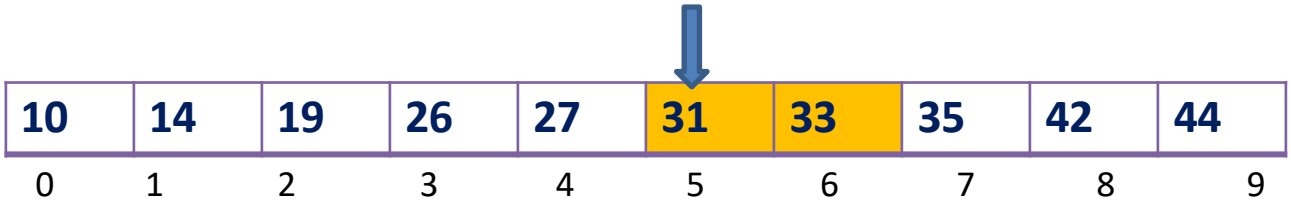


BINARY SEARCH

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

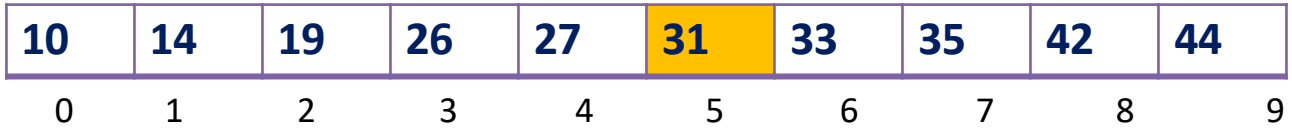


Hence, we calculate the mid again. This time it is 5.



BINARY SEARCH

We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.
Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

BINARY SEARCH TC

$$T(n) = T(n/2) + c$$

- Solved either using Recurrence Tree method or Master method.
- It falls in case II of Master Method and solution of the recurrence is O(Log n) .
- Auxiliary Space: O(1) in case of iterative implementation.
- In case of recursive implementation, O(Logn) recursion call stack space.

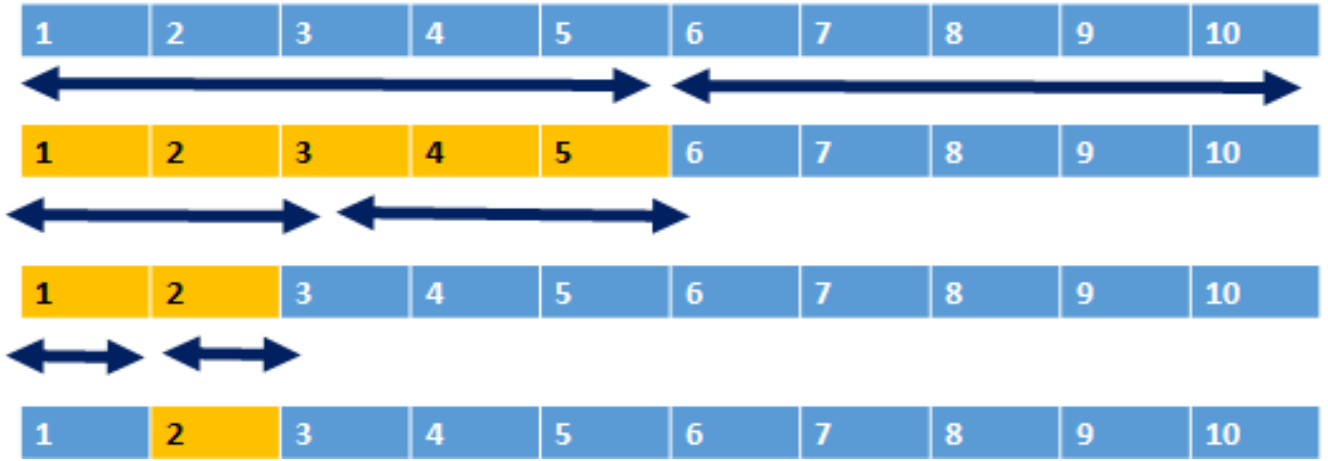
BINARY SEARCH Vs LINEAR SEARCH

	BINARY SEARCH	LINEAR SEARCH
1.	Input data needs to be sorted.	Input data not needs to be sorted.
2.	BS access data randomly.	LS does the sequential access.
3.	Time complexity = O(log n)	Time complexity = O(n)
4.	Performs ordering comparisons	Performs equality comparisons
5.	Jumping takes place	Scans one item at a time, without jumping to any item
6.	cut down your search to half as soon as you find middle of a sorted list.	Time taken to search elements keep increasing as the number of elements are increased.
7.	The middle element is looked to check if it is greater than or less than the value to be searched.	This type of checking does not occurs.

Positioning in Binary Search

In binary search, if the desired data is not found then the rest of the list is divided in two parts, lower and higher.
The search is carried out in either of them.

Even when the data is sorted, binary search does not take advantage to probe the position of the desired data.



Algorithm Binary_Search(a, n, key)

// Given array a[1 : n] of elements in non-decreasing order,
// n>=0, determine whether the key element is present or not.
// if so return index such that key = a[index], else return 0.

```
{
    low:=1; high:=n;
    while(low<=high) do
    {
        mid:=(low+high)/2;
        if(key == a[mid]) then
            return mid;
        else if(key < a[mid]) then
            high:=mid-1;
        else
            low:=mid+1;
    }
```

Date _____

Page _____

Implementation of stack using C:

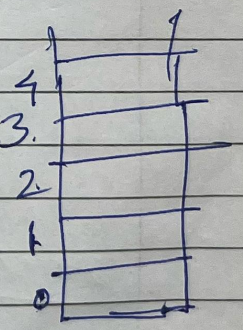
```
Void main()
{
    push();
    pop();
    peek();
    display();
}
```

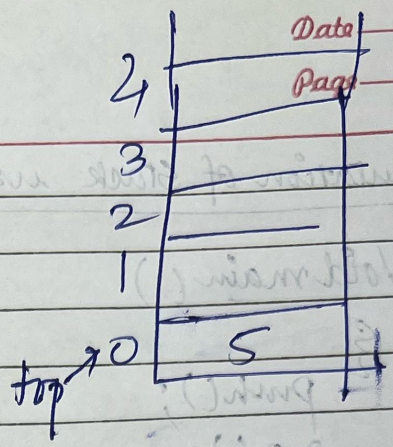
switch case statement

```
#define N 5;        (macro defination)
int stack[N];       (array name stack)
int top = -1;  ---> N (size)
                    if not insind macro define size.

voidpush()
{
printf("Enter data:");
scanf("%d", &x);
  if(top == N-1)
  {
      printf("overflow");
  }
  else
  {
      top++;
      stack[top] = x;
  }
}
```

if top=4 → overflow condition.






```
void Pop()
{
  int item;
  if(top==-1)
  {
    printf("underflow");
  }
  else
  {
    item=stack[top];
    top--;
    printf("%d", item);
  }
```

Infix
expres

Date
Page

Infix to Postfix operation :

5+1
p+q
a+5
} expressions

(a-1) + 5
op op

operands itself may be a expression.

Infix expression <operands> <operators> <operands>

5+1*6 infix

= 6*6
= 36

5+1*6
5+6
=11

Evaluation of expression follows some associative rule

① ( ) { } [ ] → R-L
② ^ → L-R
③ * /
④ + − L-R

Date
Page

5+1*6

Evaluation $\boxed{5+6=11}$

But if we want to evaluate first $(5+1)*6$

$6*6=36$

prefix

<operator> <operands> <operand>

Postfix

<operand> <operand> <operator>

5+1

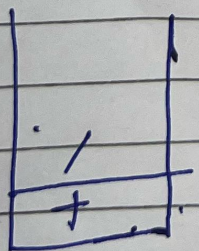
5 1 +

A+B/C

Postfix exp

ABC/+

stack



(incoming has higher precedence pushed into the stack

Date
Page

convert infix to Post fix

A - B / C * D + E

(1)

SORTING

BUBBLE SORT, INERTION SORT ,
MERGE SORT , HEAP SORT

Sorting Algorithms

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

In-place Sorting and Not-in-place Sorting

- The algorithm that do not require any extra space and sorting is said to happen **in-place sorting**.
- Ex : Bubble Sort , Insertion & Selection.
- Sorting which uses equal or more space is called **not-in-place sorting**.
- Ex : Merge-sort .

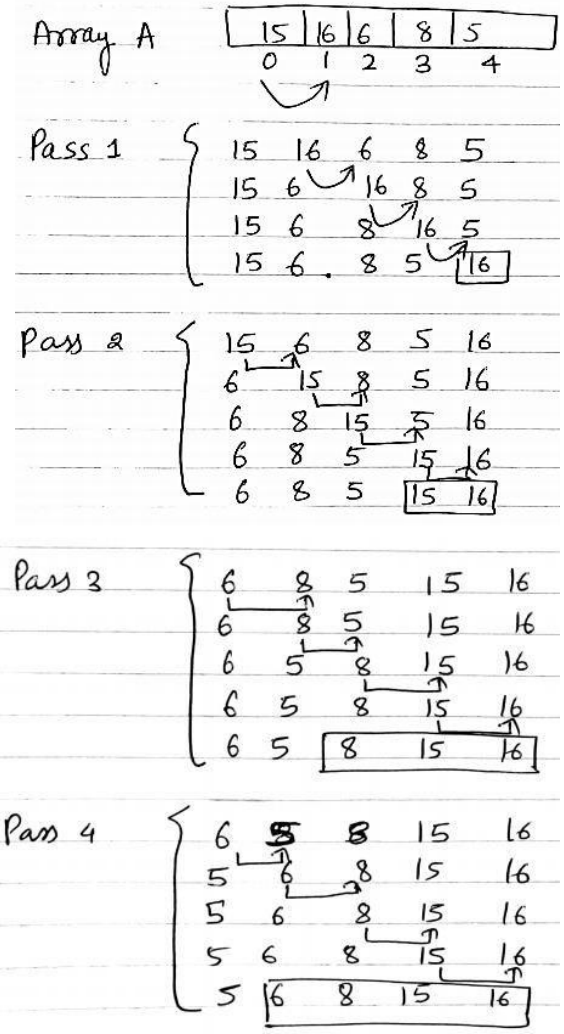
Stable and Not Stable Sorting

- If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting**.
- Ex : Insertion , Bubble & Merge Sort.
- If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.

Adaptive and Non-Adaptive Sorting Algorithm

- While sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.
- A non-adaptive algorithm is one which does not take into account the elements which are already sorted.
- Adaptive sorting algorithms:
  1. Bubble Sort
  2. Insertion Sort
  3. Quick Sort
- Non-adaptive sorting algorithms:
  1. Selection Sort
  2. Merge Sort
  3. Heap Sort

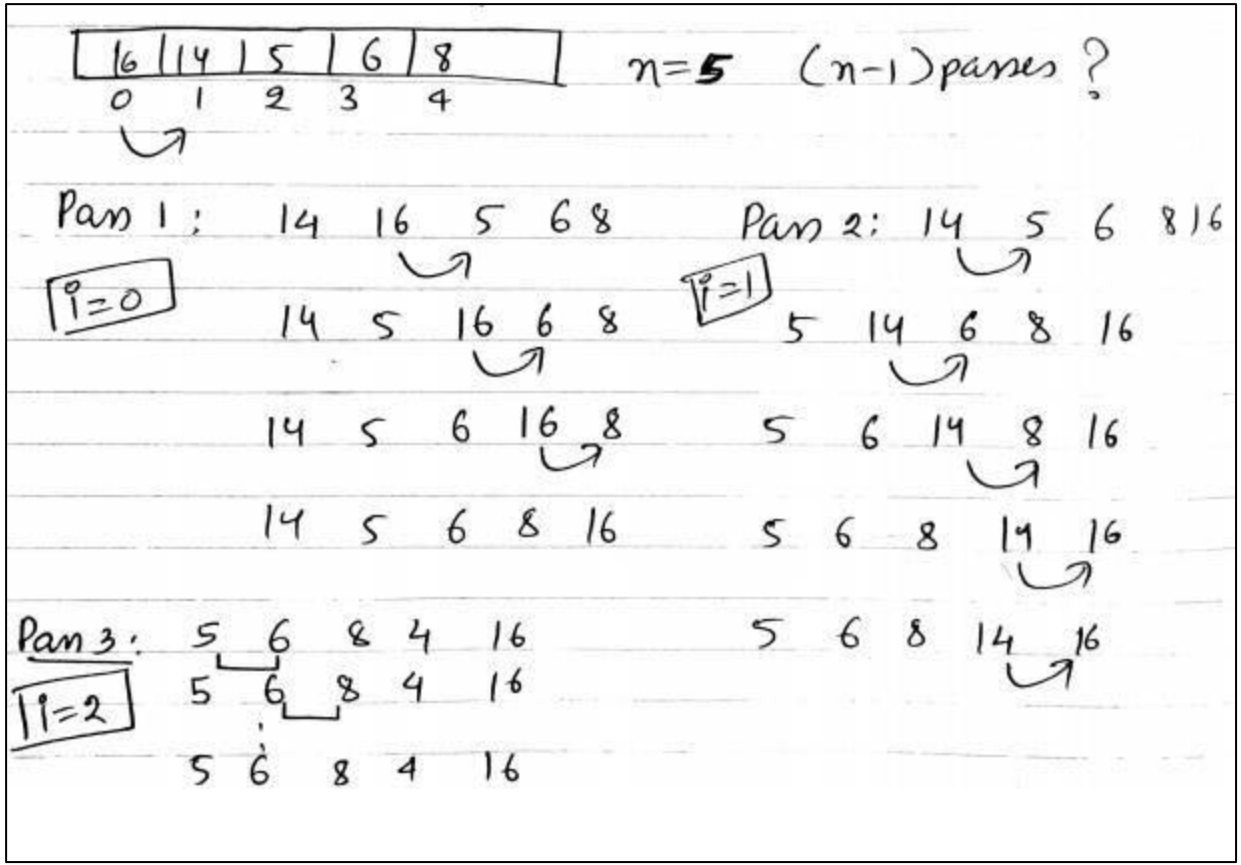
1. An online algorithm is one that can process its input piece-by-piece in a serial fashion, i.e., in the order that the input is fed to the algorithm, without having the entire input available from the start.
2. In contrast, an offline algorithm is given the whole problem data from the beginning and is required to output an answer which solves the problem at hand.
3. Selection sort repeatedly selects the minimum element from the unsorted remainder and places it at the front, which requires access to the entire input; it is thus an offline algorithm.
4. On the other hand, insertion sort considers one input element per iteration and produces a partial solution without considering future elements. Thus insertion sort is an online algorithm.
5. Note that insertion sort produces the optimum result, i.e., a correctly sorted list.
6. For many problems, online algorithms cannot match the performance of offline algorithms.
7. If the ratio between the performance of an online algorithm and an optimal offline algorithm is bounded, the online algorithm is called competitive.



Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Sorting In Place: Yes
Stable: Yes

```
for (i=0; i<n-1; i++)
{
for (j=0; j<n-1; j++)
{
if (A[j]>A[j+1])
{ temp = A[j];
  A[j] = A[j+1];
  A[j+1] = temp;
}
}
}
```


16 14 5 6 8
0 1 2 3 4
n=5 (n-1) passes ?
Pass 1:
i=0
14 16 5 6 8
14 5 16 6 8
14 5 6 16 8
14 5 6 8 16
Pass 2:
i=1
14 5 6 8 16
5 14 6 8 16
5 6 14 8 16
5 6 8 14 16
5 6 8 14 16
Pass 3:
i=2
5 6 8 4 16
5 6 8 4 16
5 6 8 4 16

Modified Bubble Sort

```
for (i=0; i<n-1; i++)          // loop for passes
{   flag = 0;
  for (j=0; j<n-1-i; j++)      // loop for comparisons
  { if (A[j] > A[j+1])
    {
      temp = A[j];             // Code of
      A[j] = A[j+1];           // swapping
      A[j+1] = temp;
      flag = 1;                // swapping has been
                               // completed.
    }
  }
  if (flag == 0)               // Already swapped
    break;                     // Break out of the loop.
}
```

i=2, $2 < n-1 \Rightarrow 2 < 4$ (Yes)
flag = 0

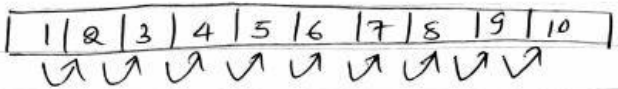
for (j=0; j< n-1-i; j++) // $0 < 4-2$
// $0 < 2$ (Yes)

A[0] > A[1]

⋮

so on...

Best Case → Already sorted



In one pass only, sorting is done.

```
for (i=0; i<n-1; i++)    // i=0 (one time) Constant
{ for (j=0; j<n-1-i; j++) // n-1 times
                          n-1 comparisons.
```

So, Complexity = $O(n)$

Worst Case → Array given in descending order and you have to sort in ascending order.

```
for (i=0; i<n-1; i++)      // (n-1) times
 {
   for (j=0; j<n-1-i; j++) // (n-1) times
```

$TC = O(n^2)$

INSERTION SORT

5	4	10	1	6	2
0	1	2	3	4	5

0	1	2	3	4	5
5	4	10	1	6	2

n = 6

Sorted list ← Unsorted list

temp = 4

4	5	10	1	6	2

Sorted ← Unsorted

temp = 10

4	5	10	1	6	2

Sorted | Unsorted

No Right shift

temp = 1

10 > 1 (Right shift)

4	5	10	1	6	2

←

4	5	?	10	6	2

temp = 1

5 > 1 (Right)

4 > 1 (Right

4	?	5	10	6	2

1	4	5	10	6	2

Sorted | Unsorted

temp = 6

10 > 6 (Right)

temp = 2

10 > 2 (Right)

1	4	5	6	10	2

Sorted | Unsorted

1	4	5	6	?	10

temp = 2

1	4	5	?	6	10

6 > 2 (Right)

1	4	?	5	6	10

5 > 2 (Right)

1	?	4	5	6	10

4 > 2 (Right)

1 ≯ 2 (No shift)

1	2	4	5	6	10

→ SORTED

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

Sorting In Place: Yes
Stable: Yes
Online: Yes
Uses: Insertion sort is used when number of elements is small.
It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

```
for (i=1 ; i<n ; i++)
{   temp = a[i];
    j = i-1 ;
while (j>=0 && a[j] >temp)
{   a[j+1] = a[j];
    j--;
}
  a[j+1] = temp;
}
```

0	1	2	3	4	5
5	4	10	1	6	2

n=6

For i=2, 2<6

temp = a[2] = 10

j = i-1 = 2-1 = 1

j=1

while (j>=0 && a[1] > 10)

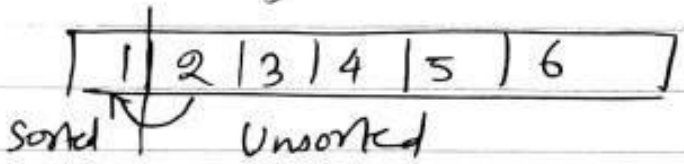
1st condn – TRUE

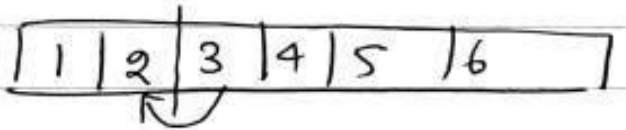
2nd condn – 5 > 10 (NO)

No entry to loop

a[2] = 10 ⇐ a[j+1]=temp;

Already sorted → Best case O(n)

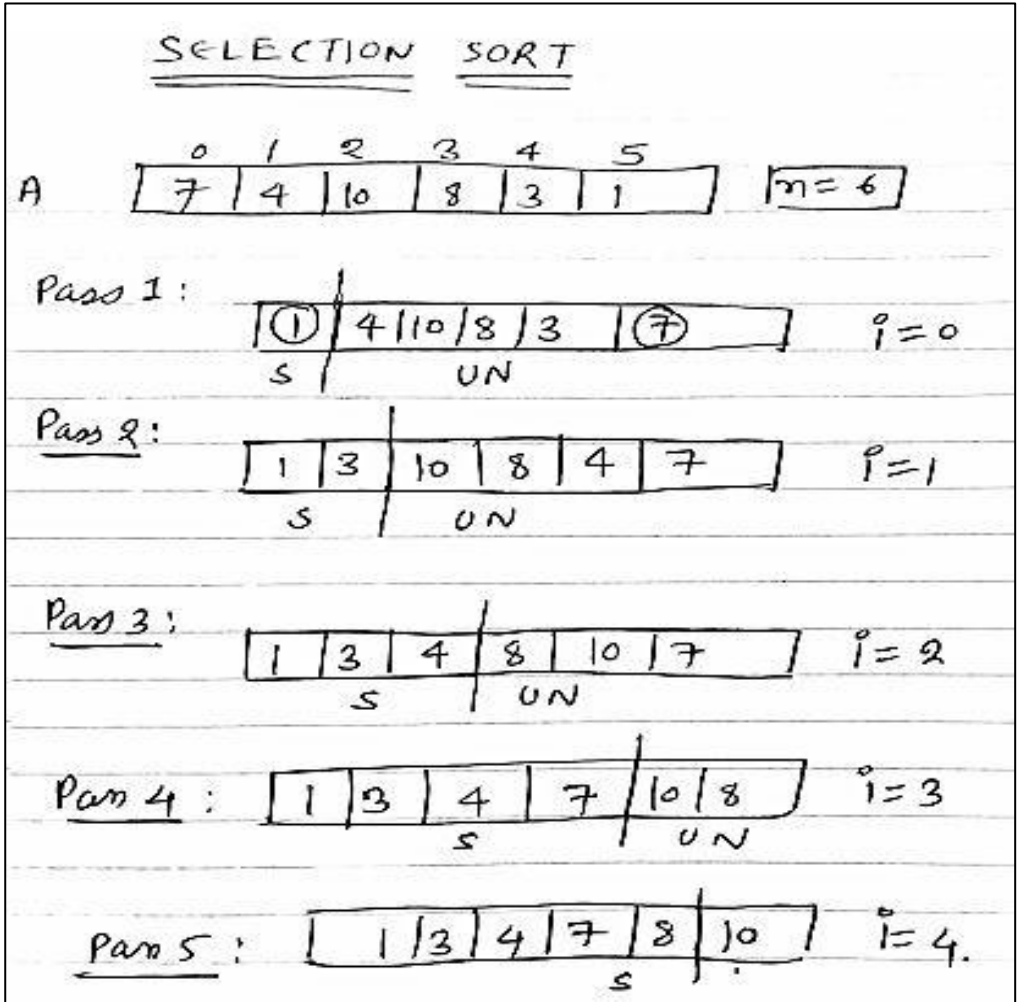




No need to compare 3 to 2 & 1
Just compare it with 2 only

Worst case → $O(n^2)$.

SELECTION SORT



The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.

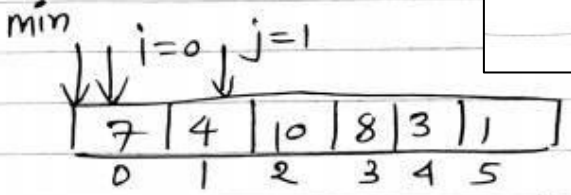
2) Remaining subarray which is unsorted.

Time Complexity: $O(n^2)$ as there are two nested loops.
Auxiliary Space: O(1)
The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

```
for (i=0; i<n-1; i++)
{ int min = i;
  for (j= i+1; j<n; j++)
  { if (a[j] < a[min])
    { min = j;
    }
  }
  if (min != i)
  { swap(a[i], a[min]);
  }
}
```

Complexity

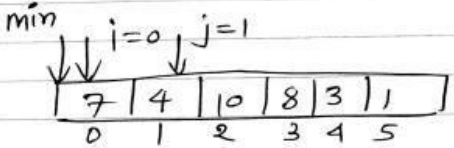
$O(n^2)$



i=0

min 0

3



i=0

min 0

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

n=6

j = 1, 1<6 Yes

if (a[j] < a[min]) ⇒ a[1] < a[0]

⇒ 4 < 7 (Yes)

then min = j

min = 1 Now j++

Selection sorting algorithm makes minimum number of memory writes

Quick Sort Preferred For Arrays

Merge Sort is preferred for Linked Lists

QUICK SORT – CACHE FRIENDLY.

QUICK SORT – GOOD LOCALITY OF REFERENCE

QUICK SORT – TAIL RECURSIVE

QUICK SORT – BEST SORTING ALGORITHM

MERGE SORT : For processing large amounts of data. .

Selection Sort : Stable With O (N) Extra Space Or When Using Linked Lists

Shell Sort : Small Code Size, No Use Of Call Stack, Reasonably Fast

BUBBLE SORT : EFFICIENT FOR SMALL LISTS AND MOSTLY SORTED LISTS.

Insertion is Expensive in Insertion Sort.

SHELLSORT IS A VARIANT OF INSERTION SORT THAT IS MORE EFFICIENT FOR LARGER LISTS.

Selection Sort : Useful Where Swapping is Expensive. Selection Sort < Insertion Sort

SELECTION SORT : INEFFICIENT FOR LARGE LISTS.

Merge Sort : it only requires sequential access, not random access. Involves a large number of copies in simple implementations.
Heap sort > Selection Sort
Quick Sort : Fastest Algorithm

Why quicksort is better than mergesort ?

1. **Auxiliary Space**
2. **Worst Case**
3. **Locality Of Reference**
4. **Merge sort is better for large data structures**

- **Heap sort** is a comparison based sorting technique based on Binary Heap data structure.
- It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.
- A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap.
- The heap can be represented by binary tree or array.

Time complexity of heapify is O(Logn).
Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).

HEAP SORT

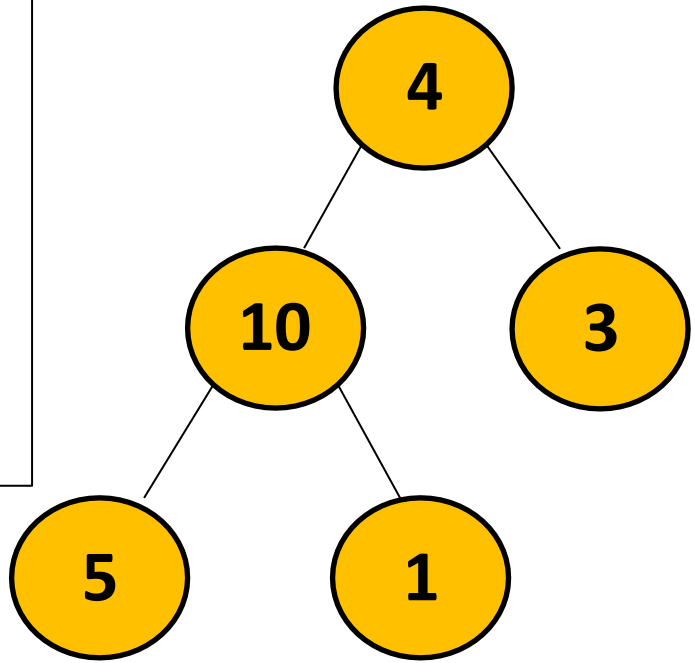
1. In this algorithm , we first build the heap using the given element.
2. We create a max heap to sort the elements in ascending order.
3. Once the heap is created , we swap the root node with the last node and delete the last node from the heap.

```
void heapSort(int arr[], int n)
{
   // Build heap (rearrange array)
   for (int i = n / 2 - 1; i >= 0; i--)
      heapify(arr, n, i);  // Creates max heap
   // One by one extract an element from heap
   for (int i=n-1; i>=0; i--)
   {
      // Swaps first and last node
      swap(arr[0], arr[i]);
      // creates max heapify on the reduced heap
      heapify(arr, i, 0);
   }
}
```

HEAP SORT

1. In this algorithm , we first build the heap using the given element.
2. We create a max heap to sort the elements in ascending order.
3. Once the heap is created , we swap the root node with the last node and delete the last node from the heap.

Index	0	1	2	3	4
I/P Data	4	10	3	5	1

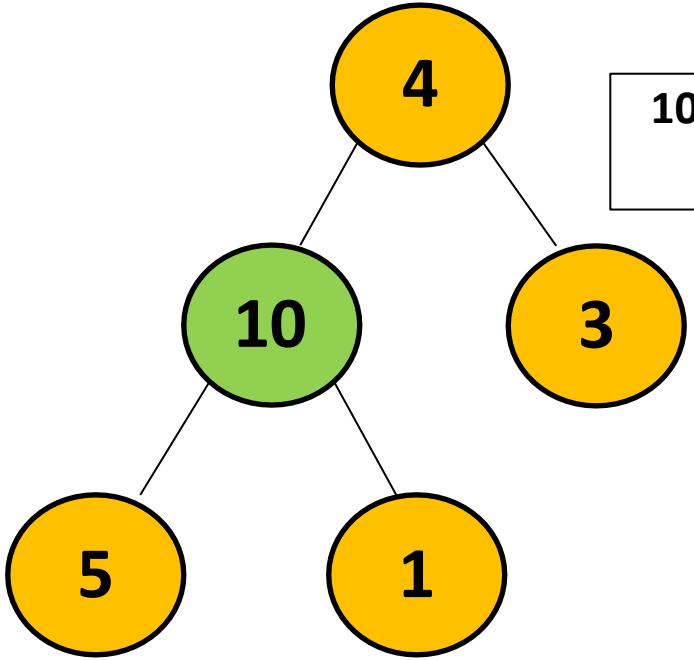


Heap is build and now convert it into max heap

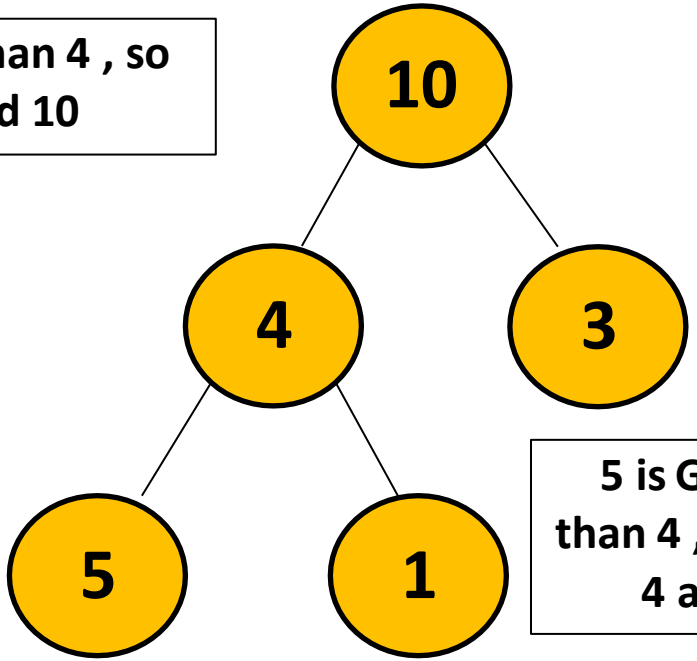
Heap is build and now convert it into max heap

Index	0	1	2	3	4
I/P Data	4	10	3	5	1

Index	0	1	2	3	4
I/P Data	10	4	3	5	1

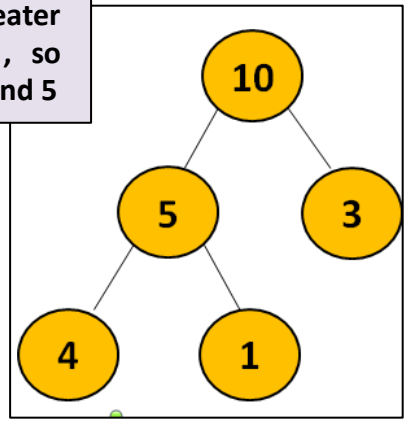


10 is Greater than 4 , so swap 4 and 10



5 is Greater than 4 , so swap 4 and 5

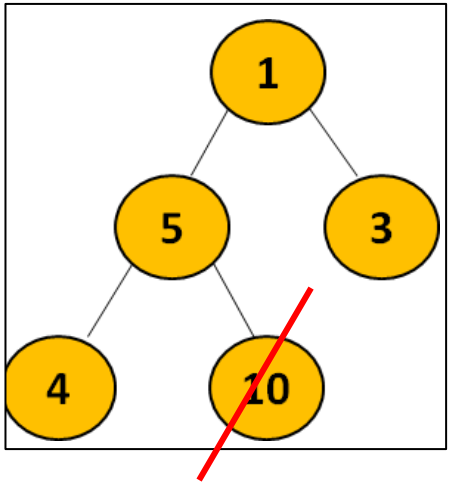
5 is Greater than 4 , so swap 4 and 5



Index	0	1	2	3	4
I/P Data	4	10	3	5	1

Index	0	1	2	3	4
I/P Data	10	4	3	5	1

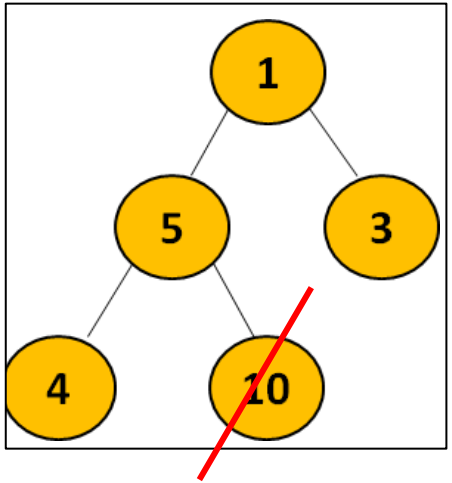
Index	0	1	2	3	4
I/P Data	10	5	3	4	1

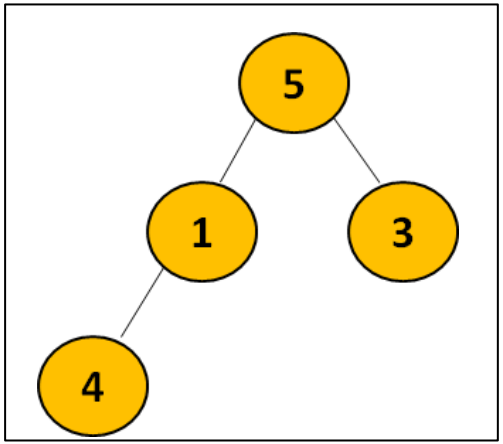


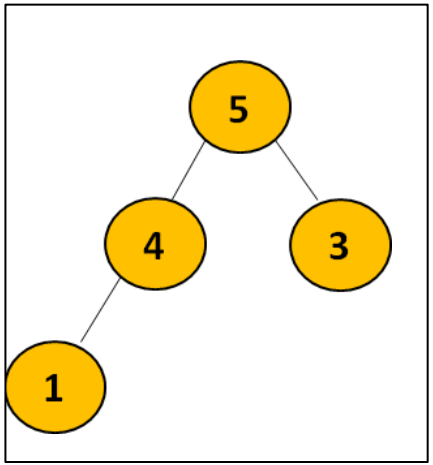
Swap 1st and last node and delete the last node

Index	0	1	2	3	4
I/P Data	1	5	3	4	10

Swap 1 and 5



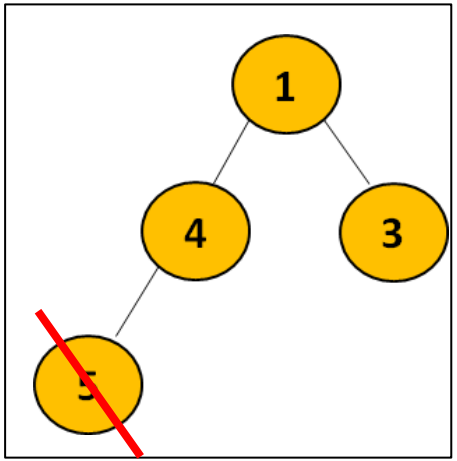


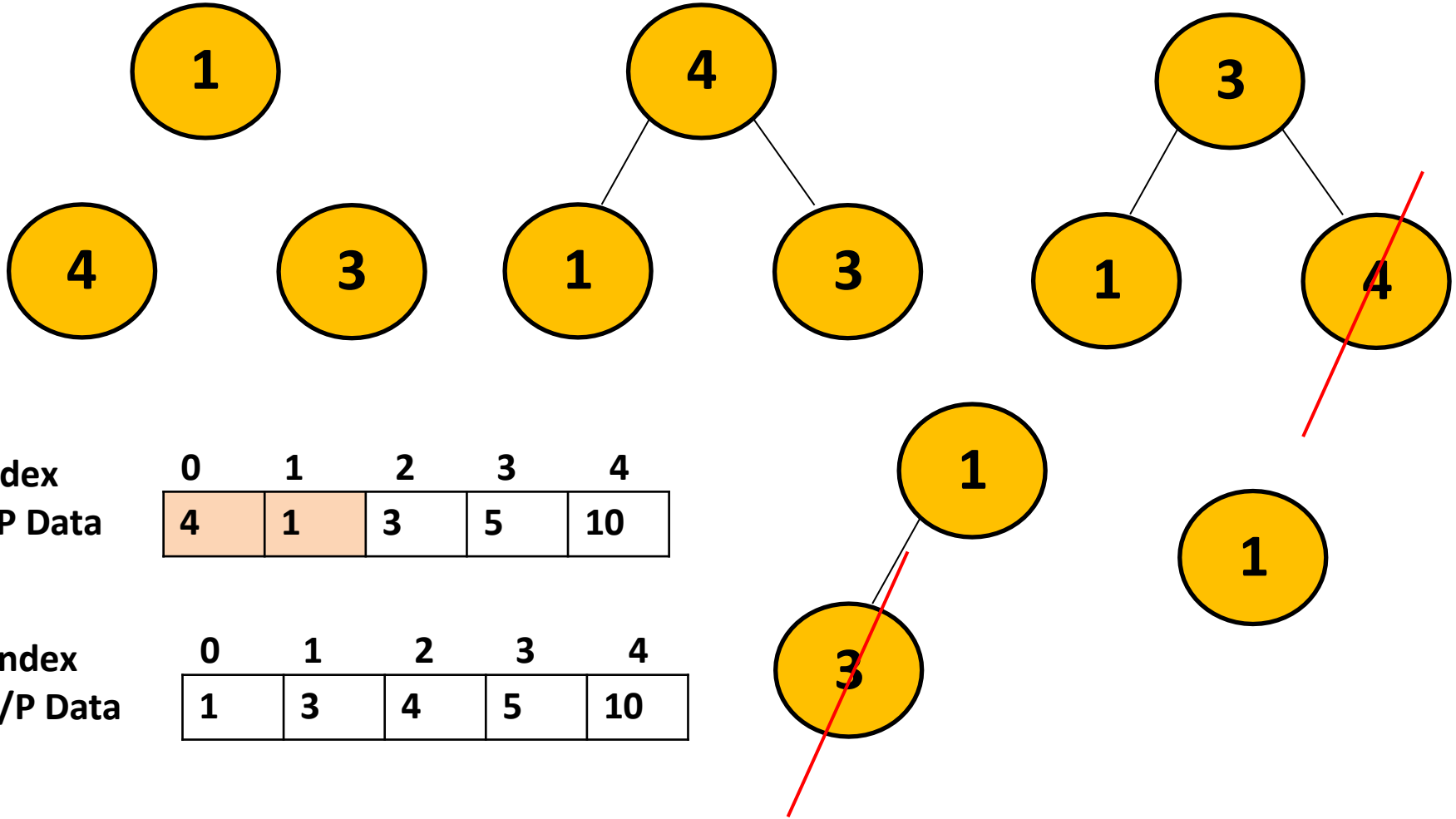


Index	0	1	2	3	4
I/P Data	5	1	3	4	10

Index	0	1	2	3	4
I/P Data	5	4	3	1	10

Index	0	1	2	3	4
I/P Data	1	4	3	5	10




1
4
3
4
1
3
3
1
4
1
3
Index
0
1
2
3
4
I/P Data
4
1
3
5
10
Index
0
1
2
3
4
I/P Data
1
3
4
5
10

SORTING

SRP

SRP 21 | 22

Sorting Algorithms	Description
Bubble Sort	Repeatedly moving the largest element to the highest index of the array. It comprises of comparing each element to its adjacent element and replace them accordingly.
Heap Sort	In the heap sort, Min heap or max heap is maintained from the array elements deending upon the choice and the elements are sorted by deleting the root element of the heap.
Insertion Sort	As the name suggests, insertion sort inserts each element of the array to its proper place. It is a very simple sort method which is used to arrange the deck of cards while playing bridge.
Merge Sort	Merge sort follows divide and conquer approach in which, the list is first divided into the sets of equal elements and then each half of the list is sorted by using merge sort. The sorted list is combined again to form an elementary sorted array.
Quick Sort	Quick sort is the most optimized sort algorithms which performs sorting in O(n log n) comparisons. Like Merge sort, quick sort also work by using divide and conquer approach.
Radix Sort	In Radix sort, the sorting is done as we do sort the names according to their alphabetical order.
Selection Sort	Selection sort finds the smallest element in the array and place it on the first place on the list, then it finds the second smallest element in the array and place it on the second place. This process continues until all the elements are moved to their correct ordering. It carries running time O(n2) which is worst than insertion sort.
Shell Sort	Shell sort is the generalization of insertion sort which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.

Sort	Average	Best	Worst
Bubble sort	O(n^2)	O(n^2)	O(n^2)
Modified Bubble sort	O(n^2)	O(n)	O(n^2)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Insertion Sort	O(n^2)	O(n)	O(n^2)
Heap Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))
Merge Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))
Quicksort	O(n*log(n))	O(n*log(n))	O(n^2)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)
Radix Sort	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)
Shell Sort	O(n)	O((nlogn)^2)	O((nlogn)^2)
Counting Sort	O(n+k)	O(n+k)	O(n+k)
Randomized Quick Sort	O(nlogn)	O(nlogn)	O(nlogn)

39 SHORT TRICKS

13

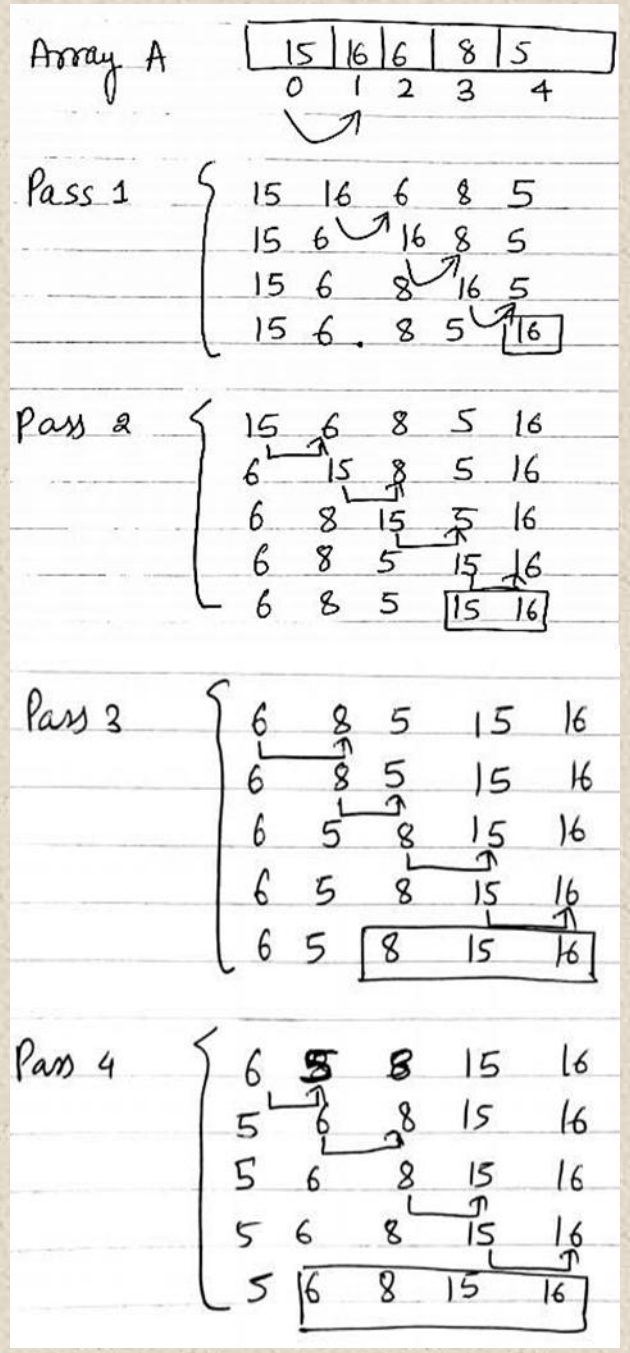
SORTING ALGORITHMS

Where is Heap Sort used practically?

- Although QuickSort works better in practice, the advantage of HeapSort worst case upper bound of O(nLogn).
- MergeSort also has upper bound as O(nLogn) and works better in practice when compared to HeapSort. But MergeSort requires O(n) extra space.
- HeapSort is not used much in practice, but can be useful in real time embedded systems where less space is available.

Can QuickSort be implemented in O(nLogn) worst case time complexity?

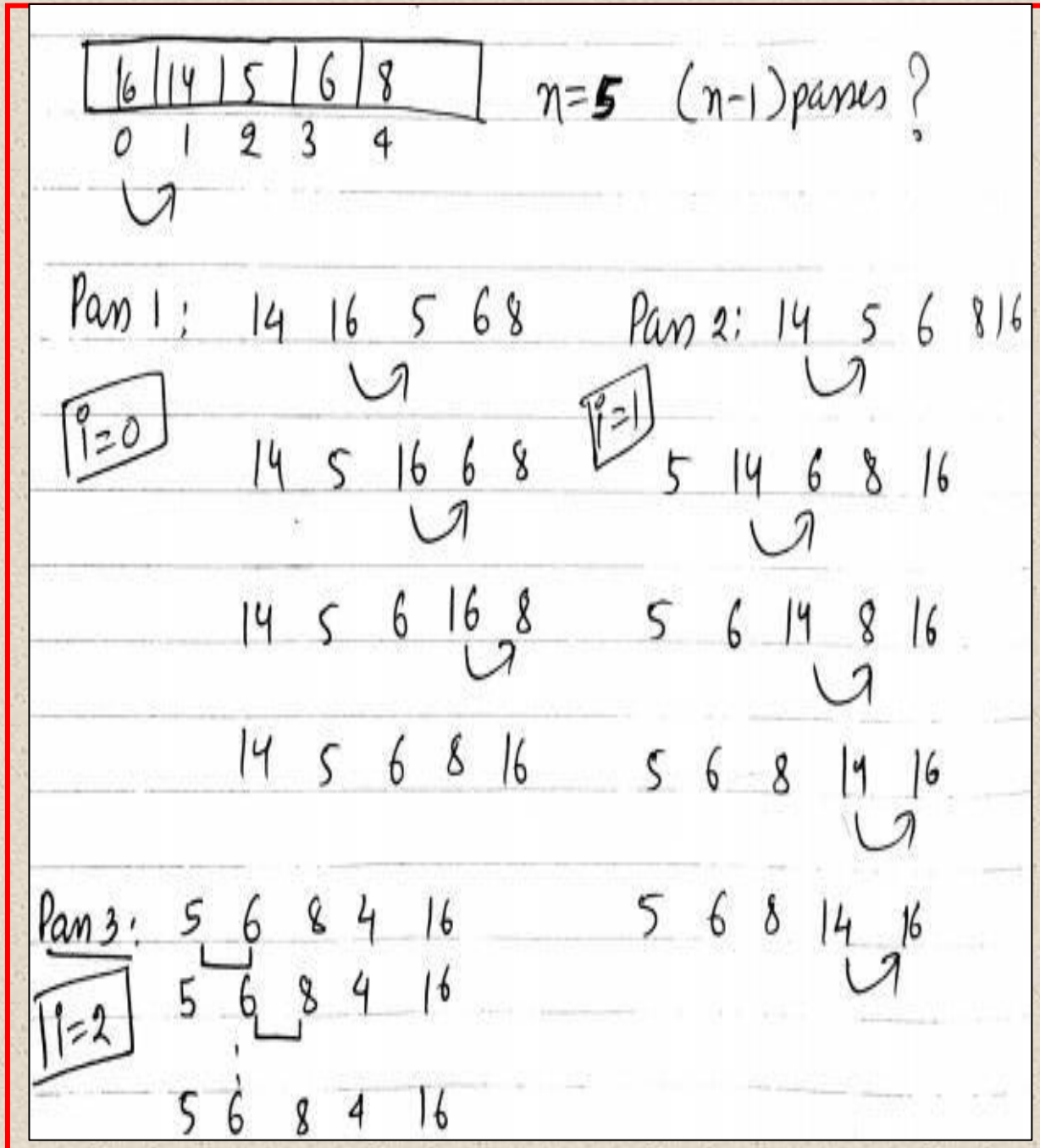
1. The worst case time complexity of a typical implementation of QuickSort is $O(n^2)$.
2. The worst case occurs when the picked pivot is always an extreme (smallest or largest) element. This happens when input array is sorted or reverse sorted and either first or last element is picked as pivot.
3. The answer is yes, we can achieve O(nLogn) worst case.
4. The idea is based on the fact that the median element of an unsorted array can be found in linear time. So we find the median first, then partition the array around the median element.



Simple Bubble sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.

```
for (i=0; i<n-1; i++)
{
for (j=0; j<n-1; j++)
{
if (A[j]>A[j+1])
{ temp = A[j];
  A[j] = A[j+1];
  A[j+1] = temp;
}
}
}
```



In simple bubble sort for 'n' elements, (n-1) comparison and (n-1) Passes are required.

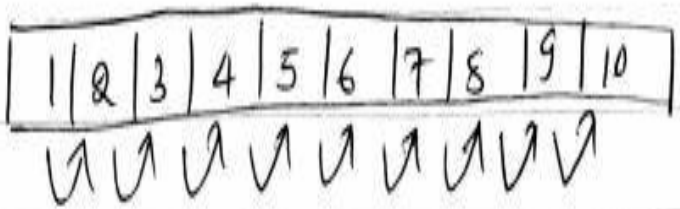
Modified and Optimized Bubble Sort

```
Modified Bubble Sort
for (i=0; i<n-1; i++)        // loop for passes
{   flag = 0;
  for (j=0; j<n-1-i; j++)    // loop for comparisons
  { if (A[j] > A[j+1])
    {
      temp = A[j];           // code of
      A[j] = A[j+1];         // swapping
      A[j+1] = temp;
      flag = 1;              // swapping has been
                             // completed.
    }
}
 if (flag==0)                // Already swapped
    break;                   // Break out of the loop.
}
```

```
i=2,  2 < n-1 ⇒ 2 < 4 (Yes)
flag = 0
for (j=0; j< n-1-i; j++)     // 0 < 4-2
                             // 0 < 2 (Yes)
A[0] > A[1]
  :
so on...
```

In modified Bubble sort
Reducing unwanted comparison
As we have seen a number of passes
Increases comparison decreases.
It is clearly visible inner loop is responsible for comparison so we are altering the inner loop here

Best Case → Already sorted



In one pass only, sorting is done.

```
for (i=0; i<n-1; i++)   // i=0 (one time) Constant

{ for (j=0; j<n-1-i; j++)  // n-1 times
                           n-1 comparisons.
```

So, Complexity = O(n)

Time complexity

Worst Case → Array given in descending order and you have to sort in ascending order.

```
for (i=0; i<n-1; i++)      // (n-1) times
{
   for (j=0; j<n-1-i; j++) // (n-1) times
```

TC = $O(n^2)$

	Time		
Sort	Average	Best	Worst
Bubble sort	O(n^2)	O(n^2)	O(n^2)
Modified Bubble sort	O(n^2)	O(n)	O(n^2)

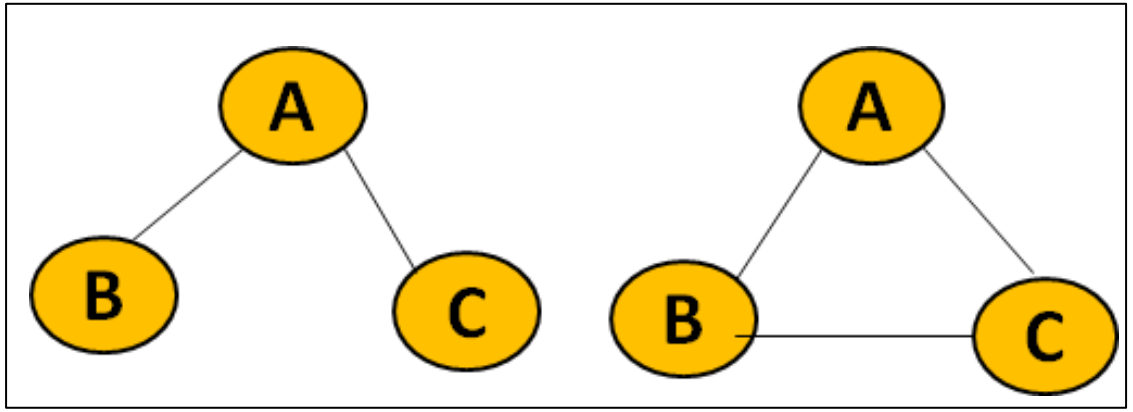
TREE DATA STRUCTURES

Tree is a non-linear data structure which organizes data in a hierarchical structure and this is a recursive definition.

- **A tree is a connected graph without any circuits.**
- **If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree.**

Properties-

1. **There is one and only one path between every pair of vertices in a tree.**
2. **A tree with n vertices has exactly (n-1) edges.**
3. **A graph is a tree if and only if it is minimally connected.**
4. **Any connected graph with n vertices and (n-1) edges is a tree.**



SRP (TREE)

1. Node: Each data item in a tree.

4. Degree of a Tree: Maximum degree of a node in a tree. Example: Degree of above tree = 3

8. Siblings: D and G are siblings of parent Node B.

9. Internal nodes: All nodes those have children nodes are called as internal nodes.

A TREE DIAGRAM

5. Height Of Tree = Height Of Root Node = 3

11. Depth of a node - number of edges from the node to the tree's root node.
A root node -depth of 0.
Ex: D(J) = 2

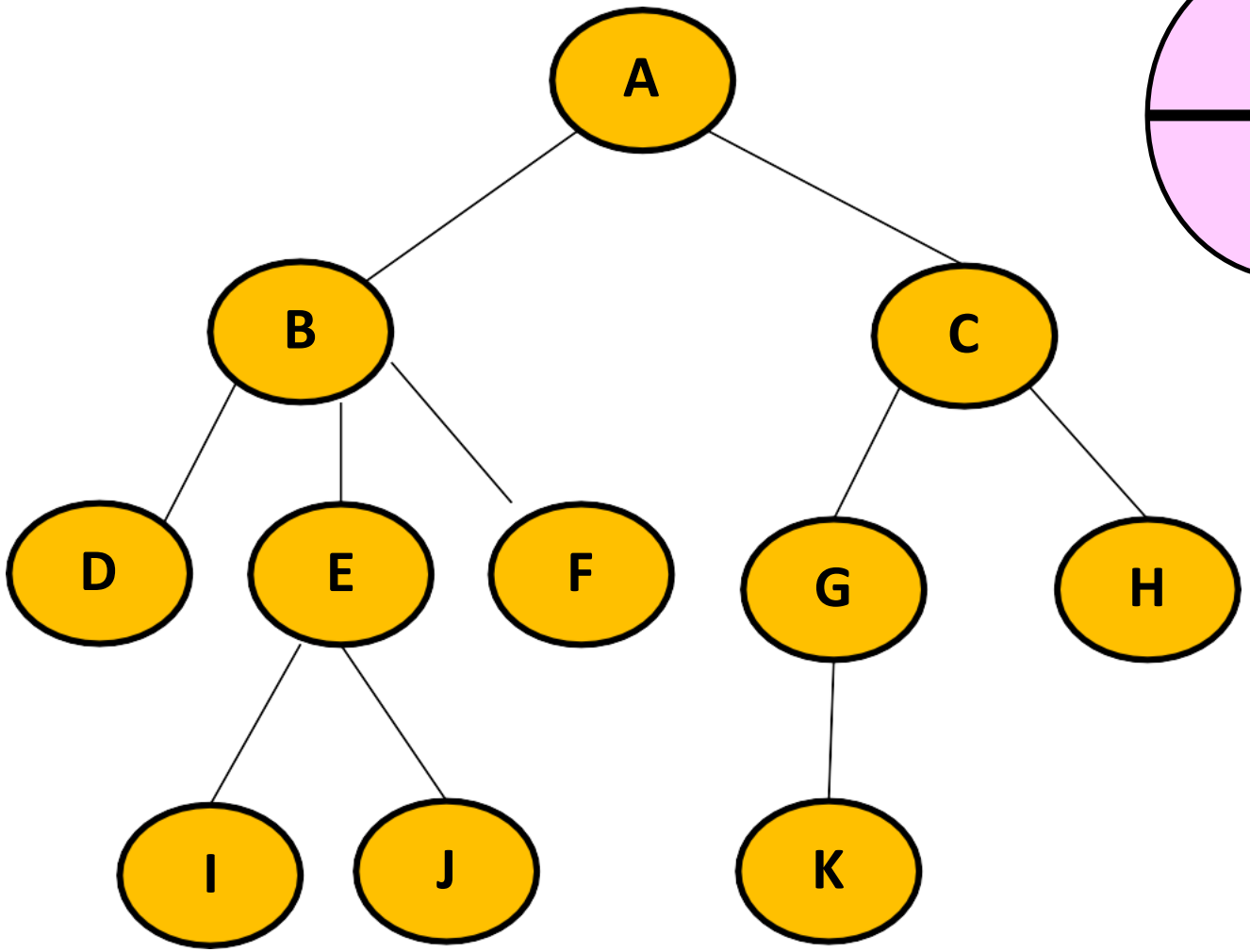
2. Height of a node number of edges on the *longest path* from the node to a leaf.
A leaf node will have a height of 0. Ex : H(E) = 2

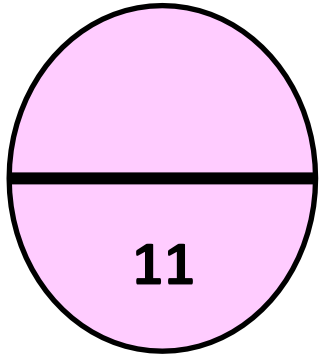
6. Non-terminal Node: Any node except root node whose degree is not zero.

3. Degree of a Node: The total number of children associated with a node is called as degree of that node.
Example:
Degree of A = 3,
Degree of E = 2

7. Forest:
Set of disjoint trees.

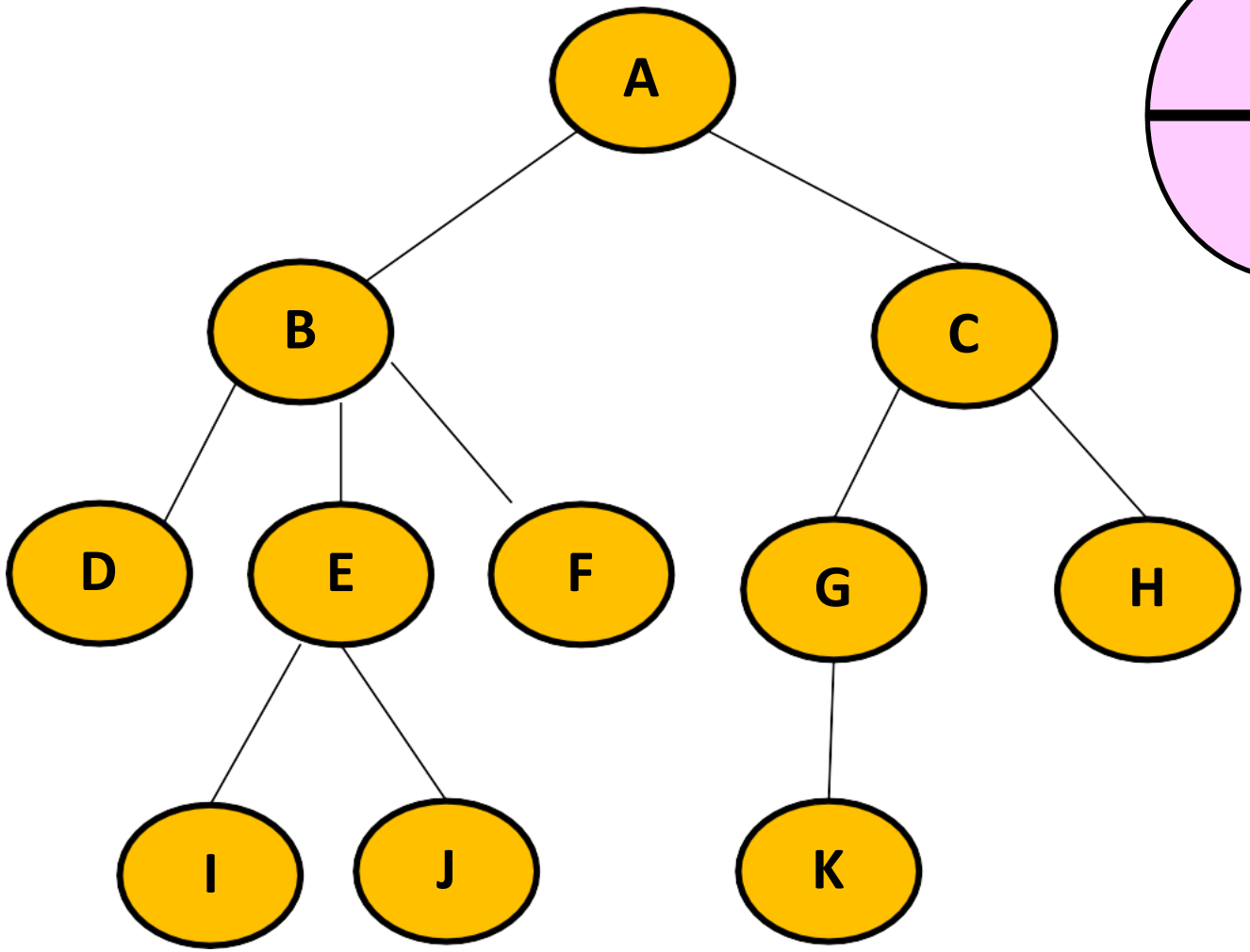
10. Leaf nodes: Those nodes, which have no child, are called leaf nodes.

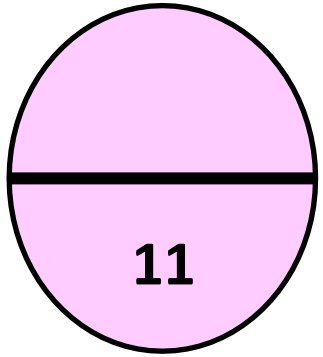




TELL ME YOUR SCORE

1. Degree of node A =
2. Degree of node B =
3. Degree of node C =
4. Degree of node D =
5. Degree of node E =
6. Degree of node F =
7. Degree of node G =
8. Degree of node H =
9. Degree of node I =
10. Degree of node J =
11. Degree of node K =





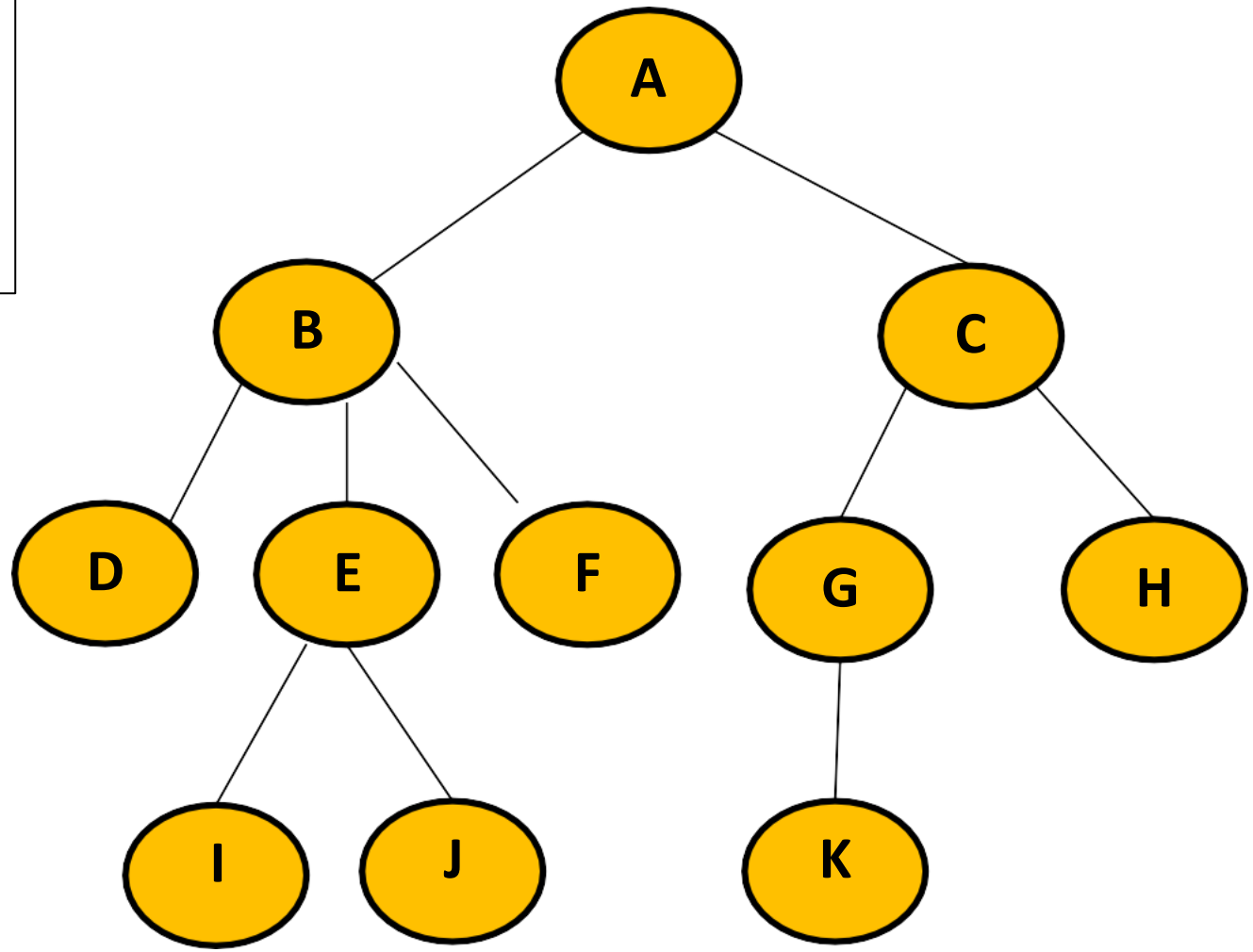
TELL ME YOUR SCORE

1. Degree of node A = 2
2. Degree of node B = 3
3. Degree of node C = 2
4. Degree of node D = 0
5. Degree of node E = 2
6. Degree of node F = 0
7. Degree of node G = 1
8. Degree of node H = 0
9. Degree of node I = 0
10. Degree of node J = 0
11. Degree of node K = 0

Here, nodes A, B, C, E and G are internal nodes.

INTERNAL NODES =

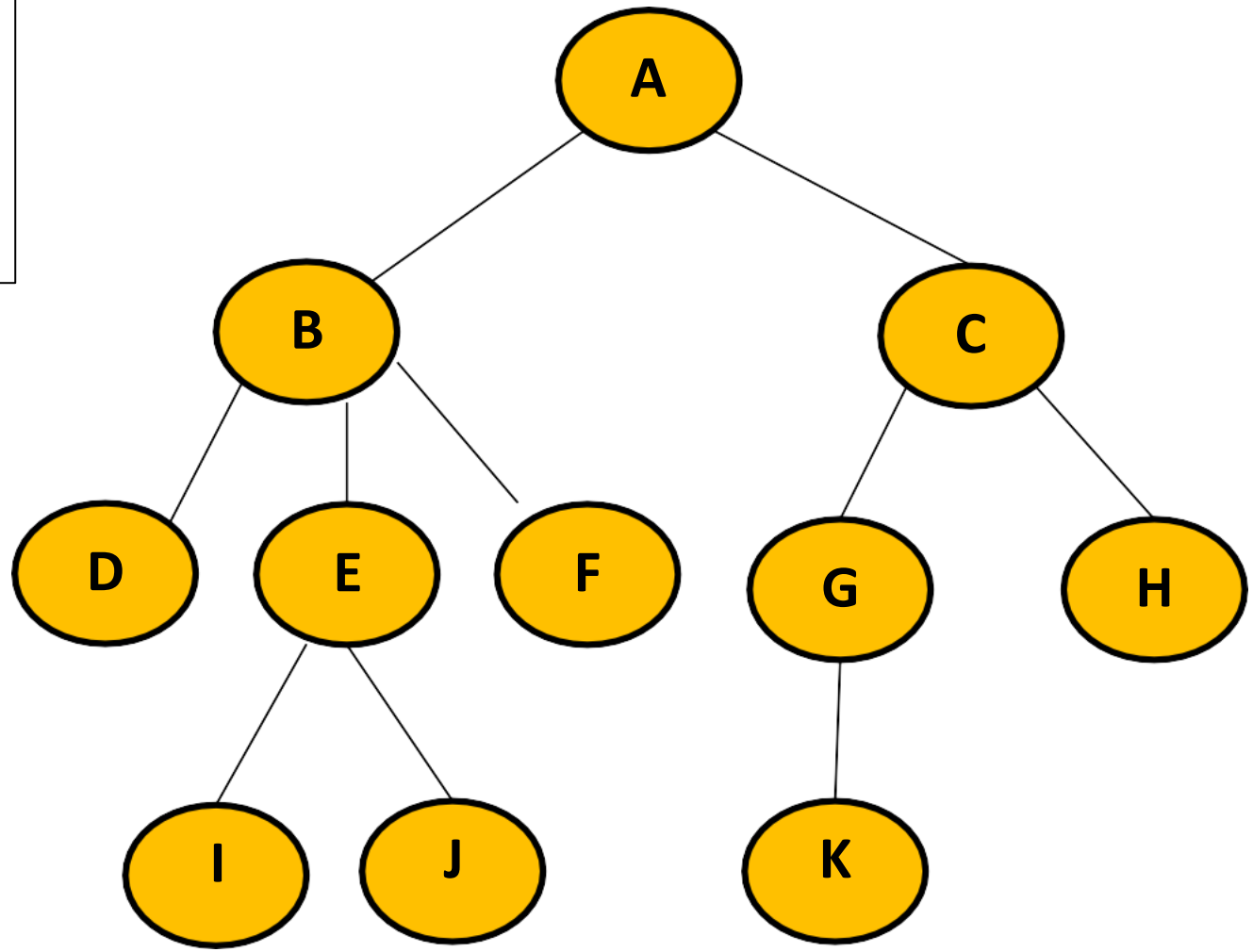
1. The node which has at least one child is called as an internal node.
2. Internal nodes are also called as non-terminal nodes.
3. Every non-leaf node is an internal node.



Here, nodes D, I, J, F, K and H are leaf nodes.

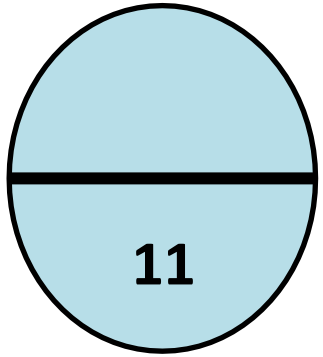
LEAF NODE =

1. The node which does not have any child is called as a leaf node.
2. Leaf nodes are also called as external nodes or terminal nodes.

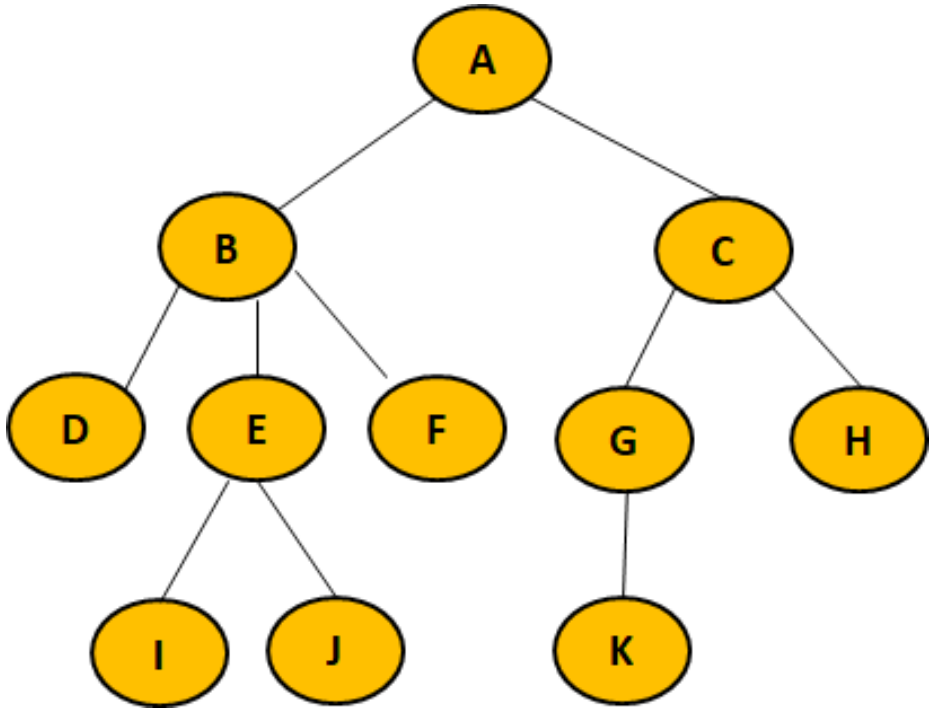


HEIGHT =

1. Total number of edges that lies on the longest path from any leaf node to a particular node is called as height of that node.
2. Height of a tree is the height of root node.
3. Height of all leaf nodes = 0



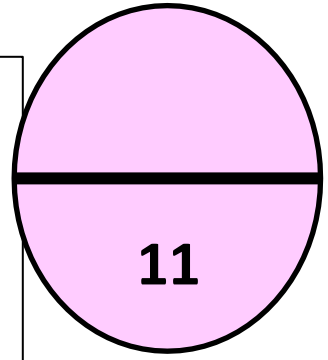
TELL ME YOUR SCORE



1. Height of node A =
2. Height of node B =
3. Height of node C =
4. Height of node D =
5. Height of node E =
6. Height of node F =
7. Height of node G =
8. Height of node H =
9. Height of node I =
10. Height of node J =
11. Height of node K =

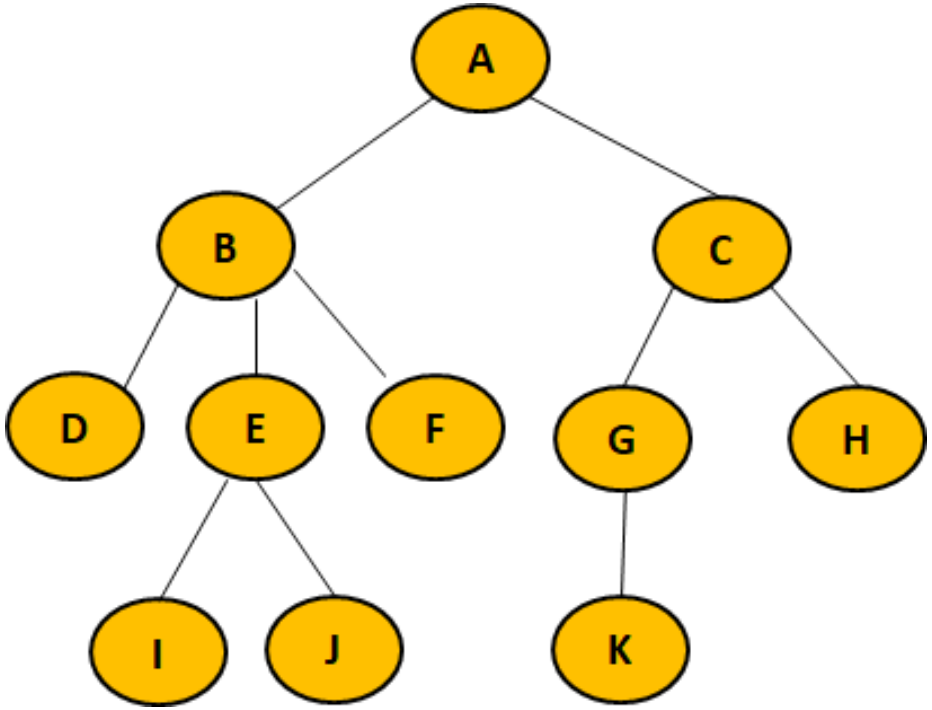
Depth-

1. Total number of edges from root node to a particular node is called as depth of that node.
2. Depth of a tree is the total number of edges from root node to a leaf node in the longest path.
3. Depth of the root node = 0
4. The terms “level” and “depth” are used interchangeably.



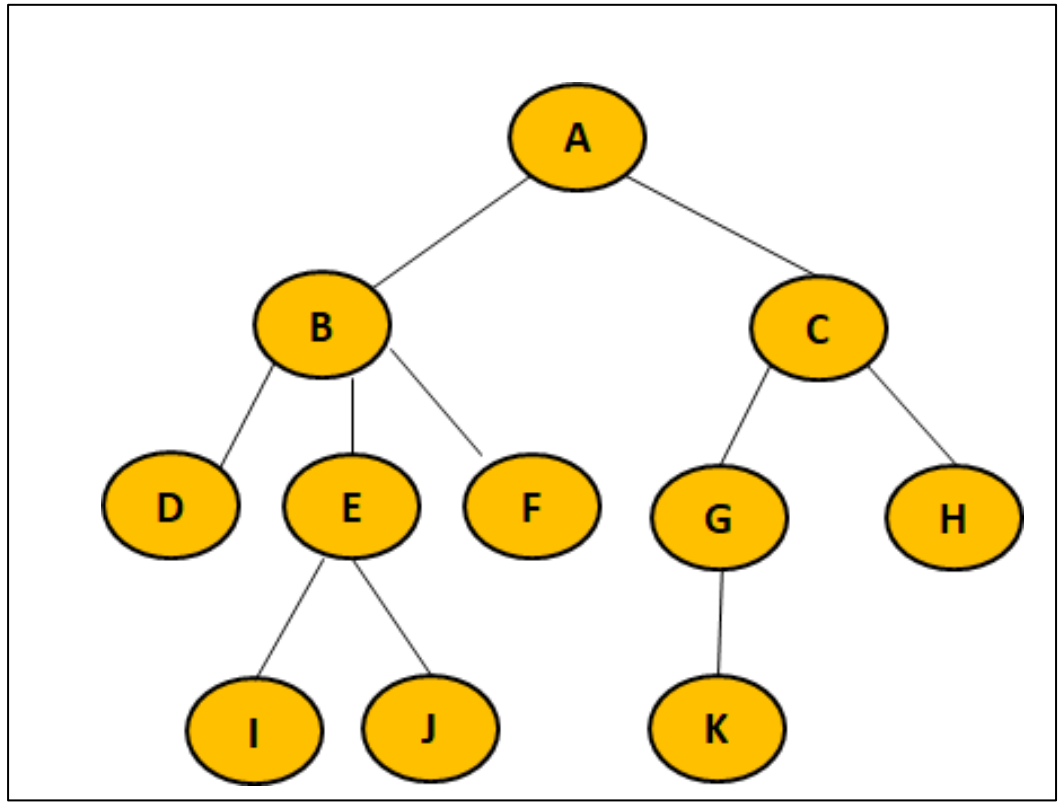
TELL ME YOUR SCORE

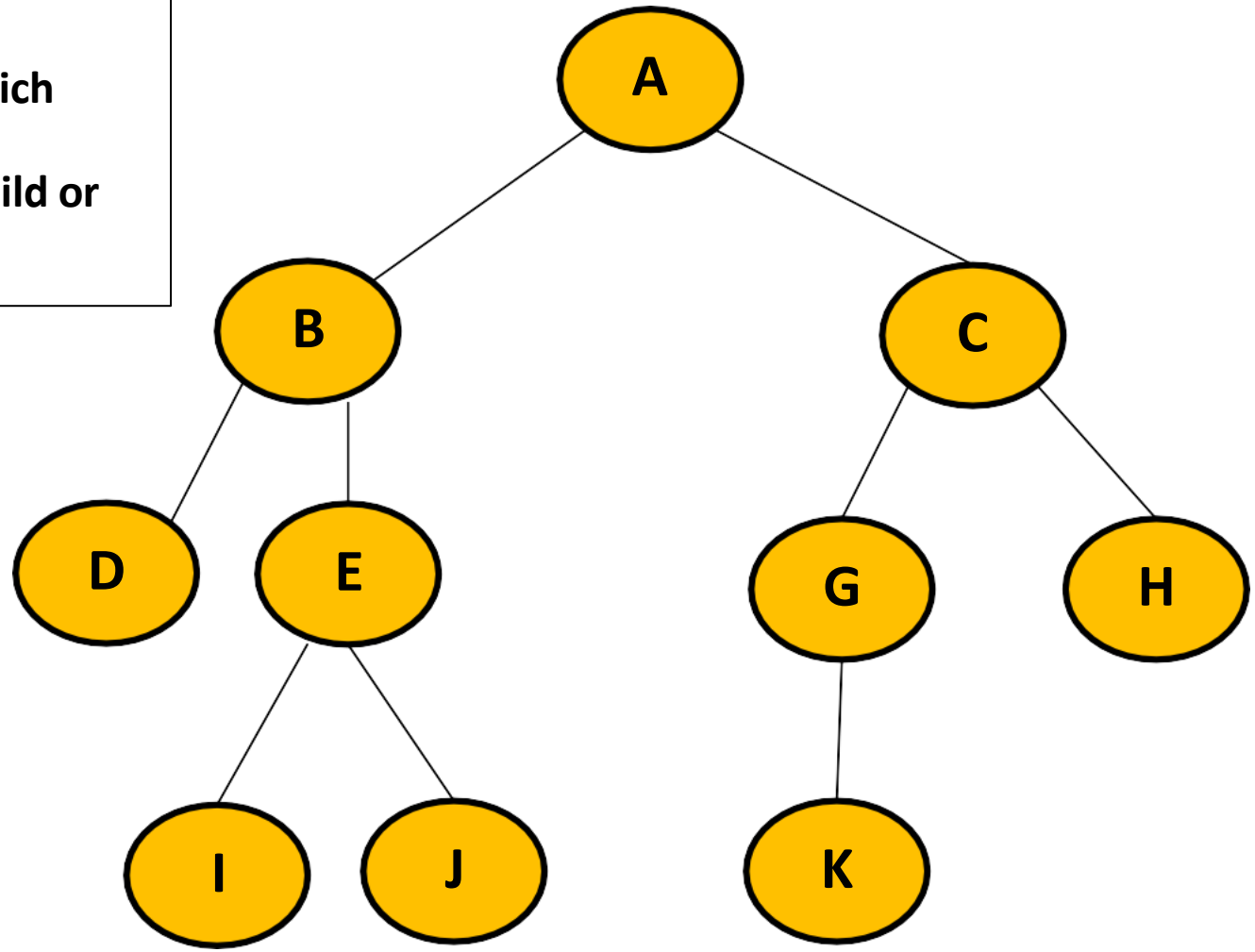
1. Depth of node A =
2. Depth of node B =
3. Depth of node C =
4. Depth of node D =
5. Depth of node E =
6. Depth of node F =
7. Depth of node G =
8. Depth of node H =
9. Depth of node I =
10. Depth of node J =
11. Depth of node K =



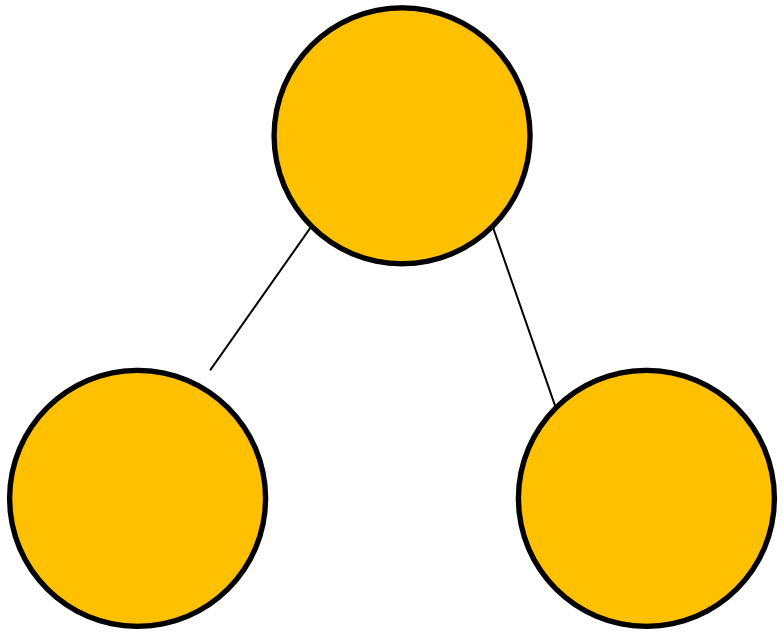
BINARY TREE

1. Binary tree is a special tree data structure in which each node can have at most 2 children.
2. Thus, in a binary tree, Each node has either 0 child or 1 child or 2 children.





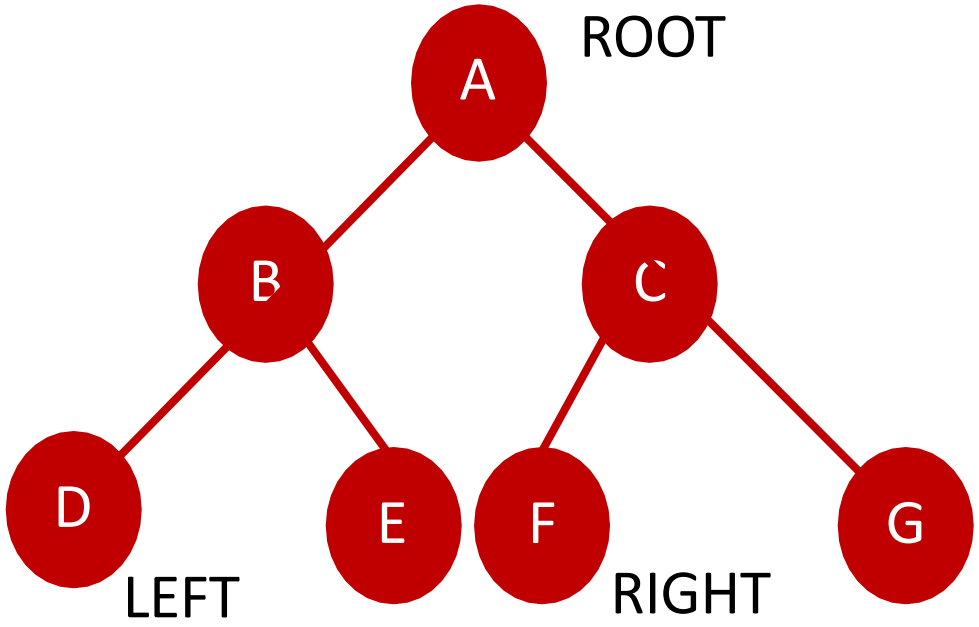
A binary tree is unlabeled if its nodes are not assigned any label.



Number of binary trees possible with n unlabeled nodes

$= {}^{2 \times n}C_n / (n + 1)$

Number of binary trees possible with 3 unlabeled nodes ?



SRP3

PREORDER = ROOT, LEFT , RIGHT
INORDER = LEFT,ROOT,RIGHT
POSTORDER= LEFT , RIGHT , ROOT
Breadth First Or LEVEL ORDER = TOP TO BOTTOM & LEFT TO RIGHT

A , B , D ,E ,C, F , G - PRE ORDER

D , B, E , A , F , C , G - Inorder

D , B, E , F , G, C , A- Postorder

A , B , C, D , E , F, G – level order

SHORTCUT

Maximum number of nodes in an AVL Tree of height H = $2^{H+1} - 1$

SRP (TREE)

Minimum number of nodes in a binary tree of height H = H + 1

Minimum number of nodes in an AVL Tree of height H is given by a recursive relation-
N(H) = N(H-1) + N(H-2) + 1

N(0) = 1
N(1) = 2

Maximum number of nodes in a binary tree of height H = $2^{H+1} - 1$

leaf nodes in a Binary Tree = Degree-2 nodes + 1

Minimum height of an AVL Tree using N nodes = floor($\log_2 N$)

Maximum number of nodes at any level 'L' in a binary tree = 2^L

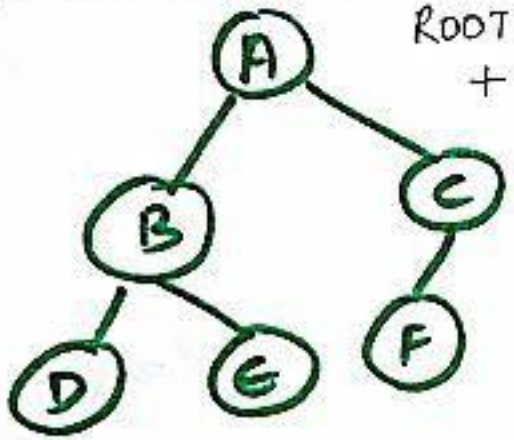
Maximum height of an AVL Tree using N nodes is calculated using recursive relation-
N(H) = N(H-1) + N(H-2) + 1

If there are n nodes in an AVL Tree, maximum height can not exceed $1.44\log_2 n$.

In other words, Worst case height of an AVL Tree with n nodes = $1.44\log_2 n$.

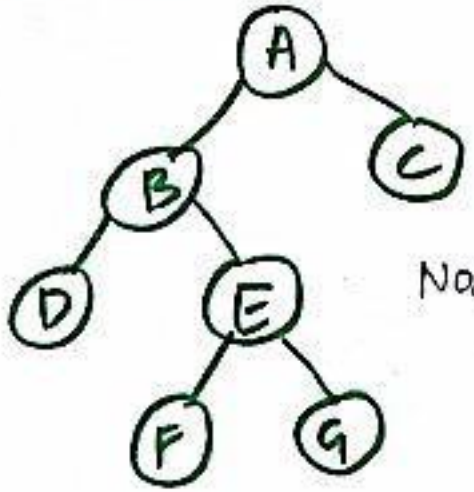
Types OF BINARY TREE

1 ROOTED BT



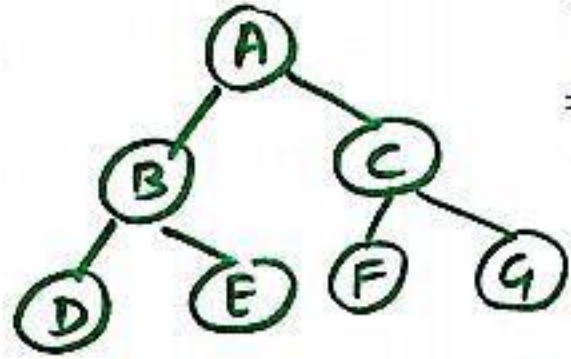
ROOT NODE
+ ATMOST 2
children

2 FULL/STRICTLY BT



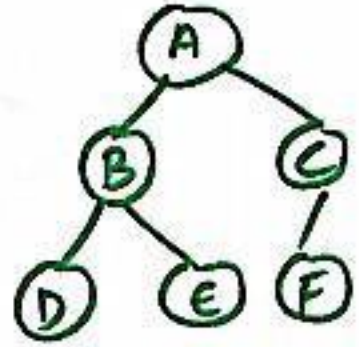
Node = Either
0 or 2
children.

3 COMPLETE/PERFECT BT

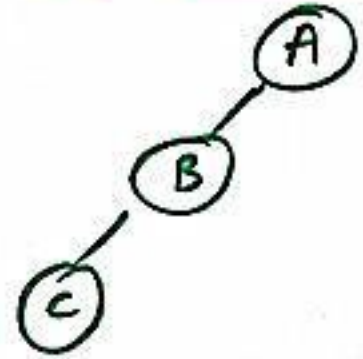


Every Internal Node
= exactly 2 children
& all leaf nodes
at same level.

4 Almost Complete BT



5 Skewed BT

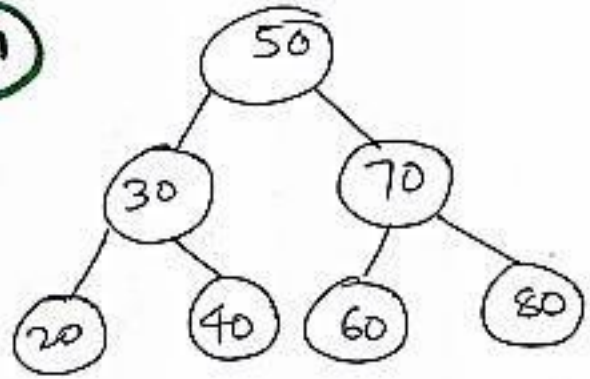


BINARY SEARCH TREE

DELETION

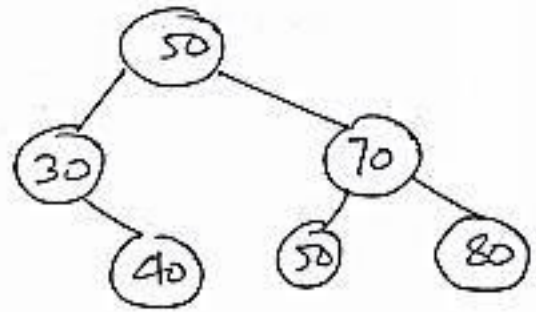
1. Node - No child
2. Node - 1 child
3. Node - 2 child

Case (1)



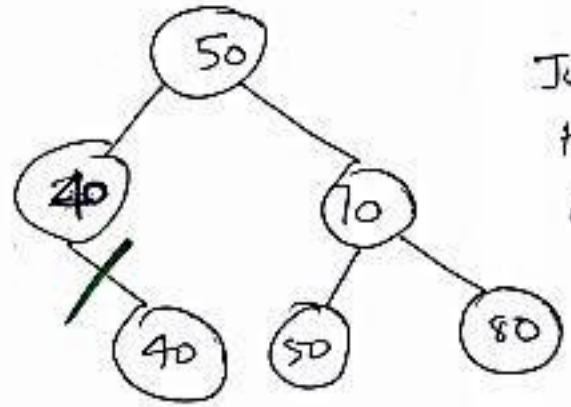
Delete '20'

Just Delete



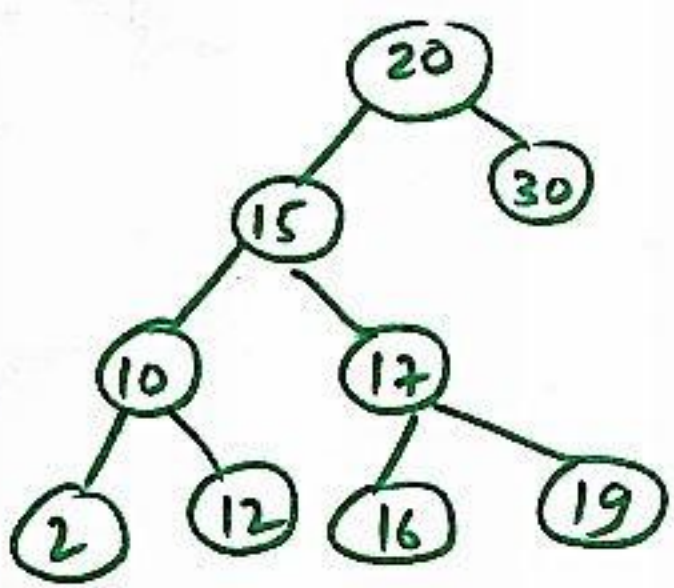
Case (2):

Delete '30'

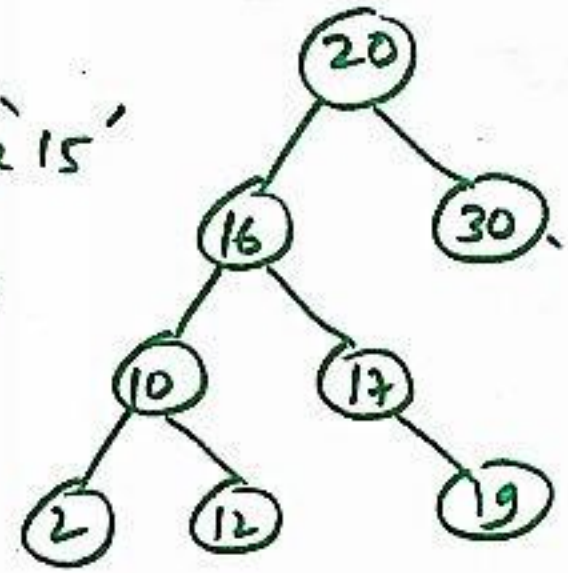


Just make the child as parent

Case 3: 2 children

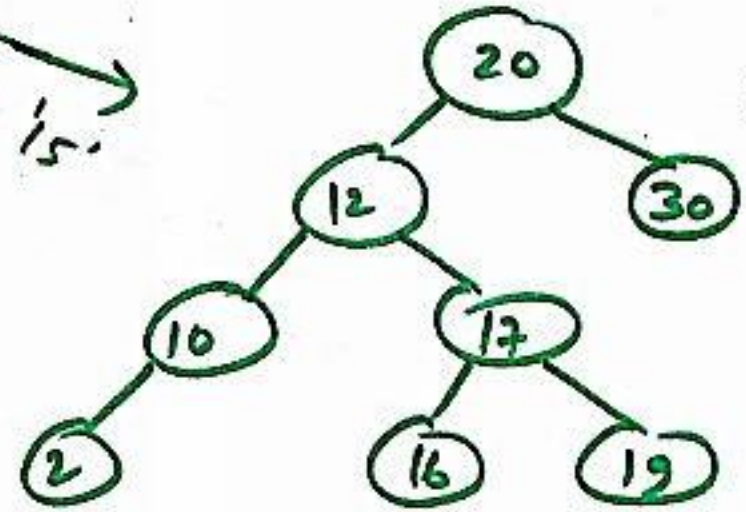


Delete '15'



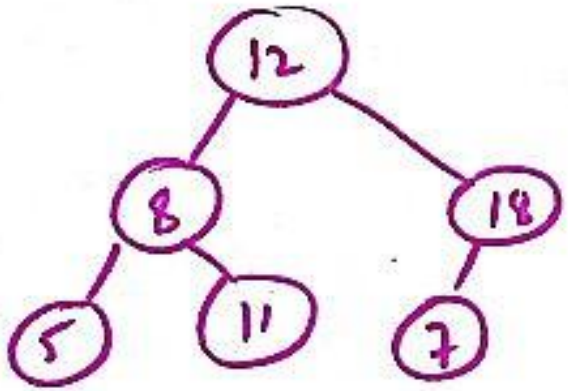
(1) Visit Right subtree of deleting node, Choose least Element & replace.

Delete '15'



(2) Visit left Choose greatest Element

AVL TREE [Self BALANCING BST]

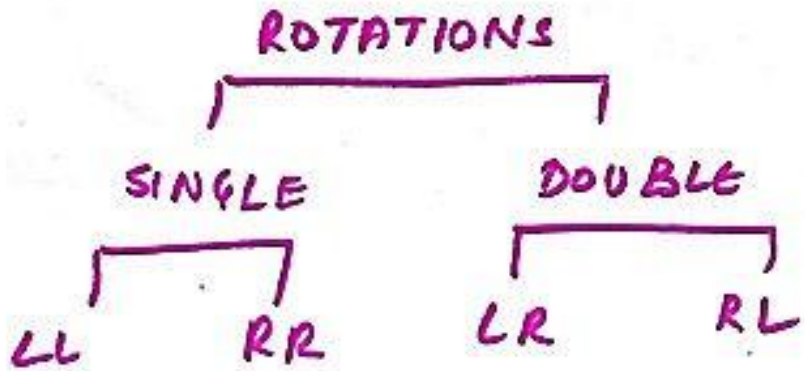


left Subtree − Right Subtree = At most 1.

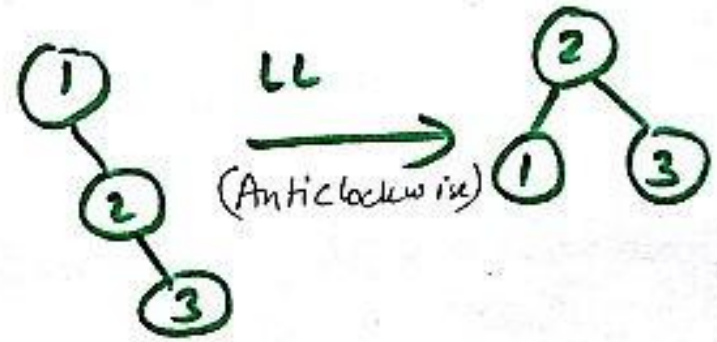
[1, 0, −1].

ROTATIONS → To make AVL Balanced.

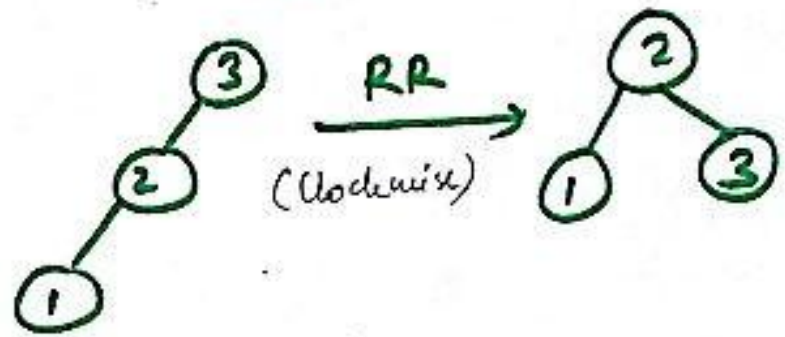
HOW? By checking Balancing factor.



Case 1:



Case 2:



A binary search tree contains the numbers 1, 2, 3, 4, 5, 6, 7, 8. When the tree is traversed in preorder and the values in each node printed out. The sequence of values obtained in 5, 3, 1, 2, 4, 6, 8, 7, if tree is traversed in postorder. The sequence obtained would be:

(a) 8, 7, 6, 5, 4, 3, 2, 1

(b) 1, 2, 3, 4, 8, 7, 6, 5

(c) 2, 1, 4, 3, 6, 7, 8, 5

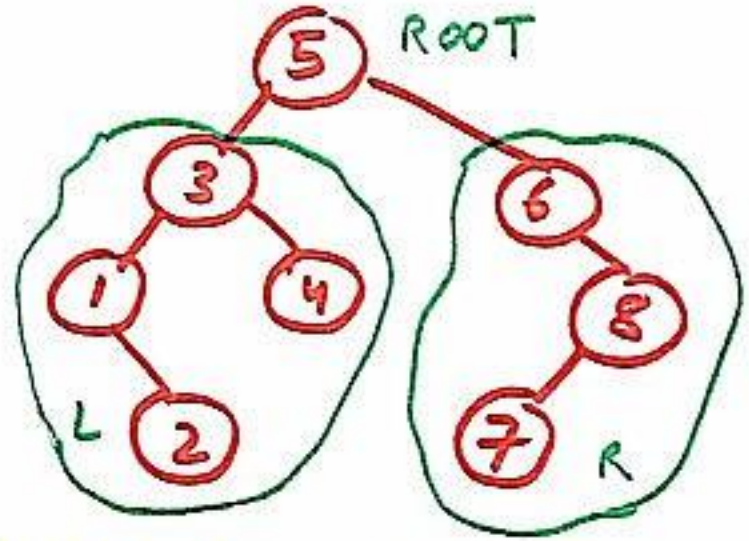
(d) 2, 1, 4, 3, 7, 8, 6, 5

GATE 2005

BINARY SEARCH TREE

PREORDER : 5, 3, 1, 2, 4, 6, 8, 7

ROOT, L, R



POSTORDER: ?

LRRoot

Left → 2, 1

Right → 4

Root → 3

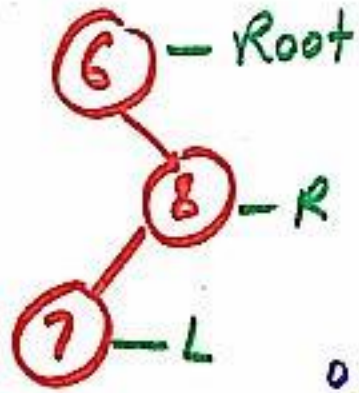
∴ 2143 From Left subtree.

↓

786 from Right subtree

↓

5 Root



∴ 2143786 5 ‼ (D)

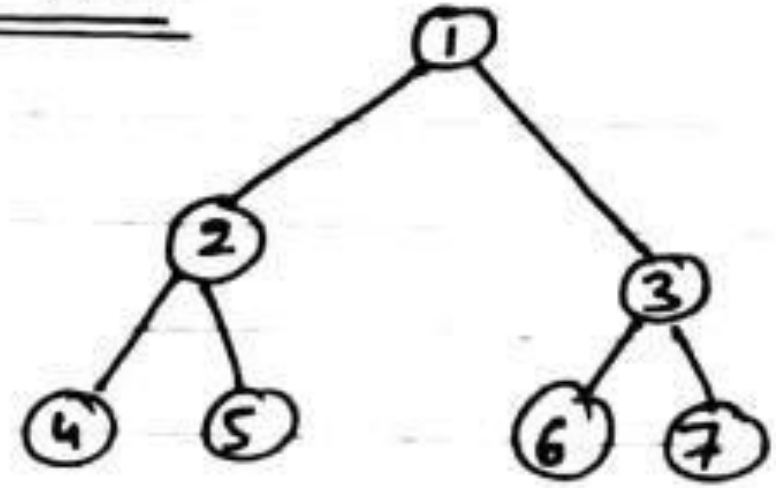
Graph Traversal refer to the process of visiting every vertex in a graph.

1. **Breadth First Traversal**
2. **Depth First Traversal**

TRAVERSAL IN BINARY TREE

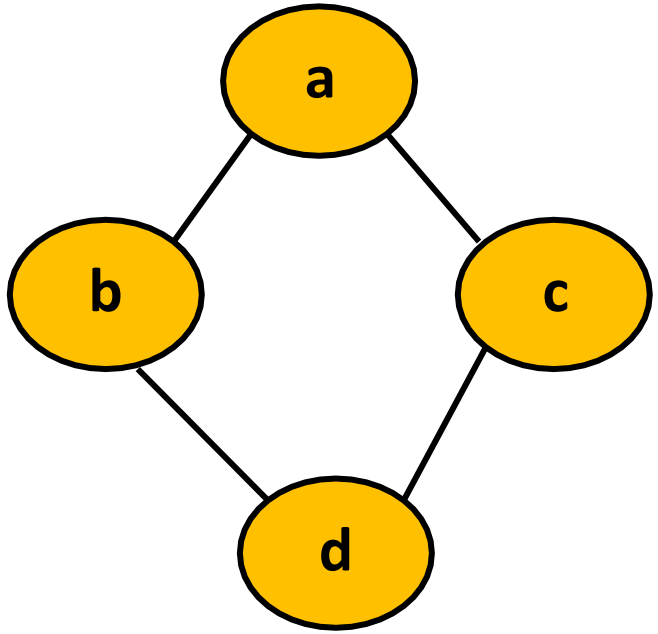
(a) BFS → {1, 2, 3, 4, 5, 6, 7}
→ Level-Order

(b) DFS → {1, 2, 4, 5, 3, 6, 7}
→ Pre-order



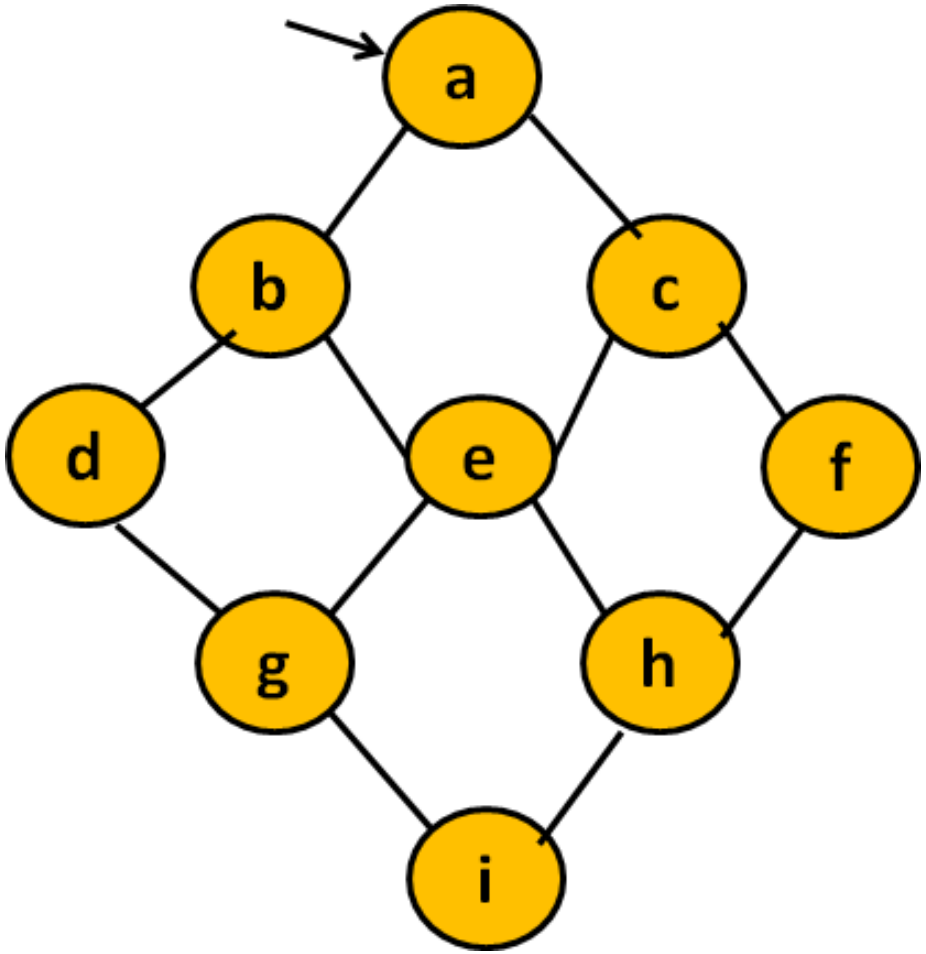
BREADTH FIRST TRVERSAL

1. Implemented using QUEUE Data Structure. => FIFO
2. In BFS , nodes are visited level by level.
3. BFS also known as level order traversal.
4. Time Complexity of BFS : O(V+E)



EXAMPLE OF BFS :

1. Enqueue starting vertex
2. Enqueue all adjacent vertices
3. Dequeue & print
4. Repeat until over



O/P :

QUEUE =

APPLICATIONS OF BREADTH FIRST TRAVERSAL :

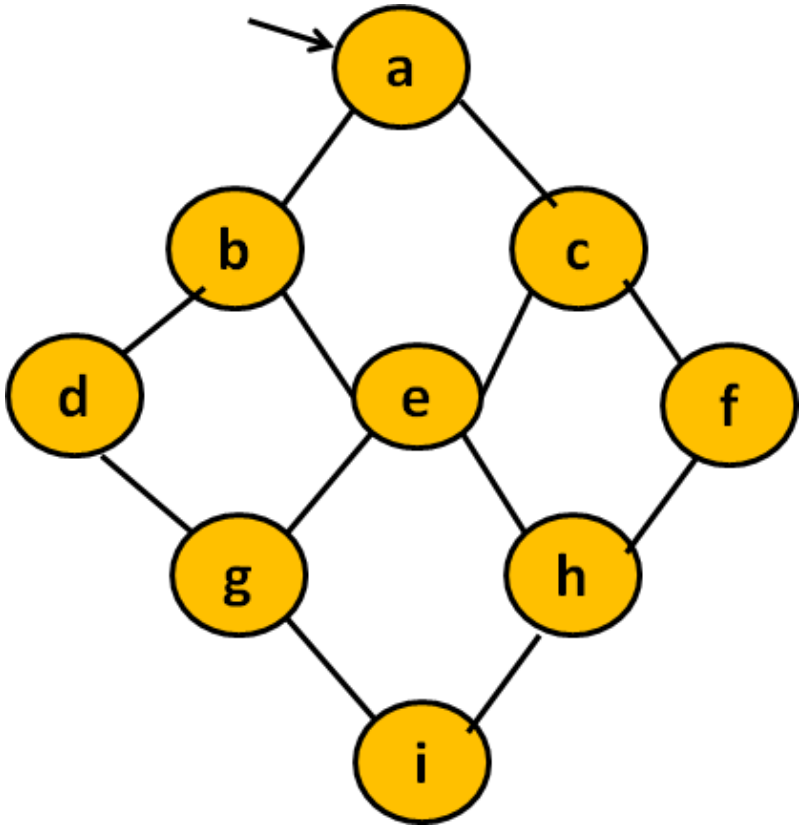
1. We can verify given graph connected or not.
2. We can find number of connected components
3. We can check given graph contain cycle or not
4. We can find shortest path from given source to every vertex in the given unweighted graph.
5. We can verify given graph bipartite or not.

DEPTH FIRST TRAVERSAL

DFS is implemented using Stack as Data Structure

Time Complexity of DFS : O(V+E)

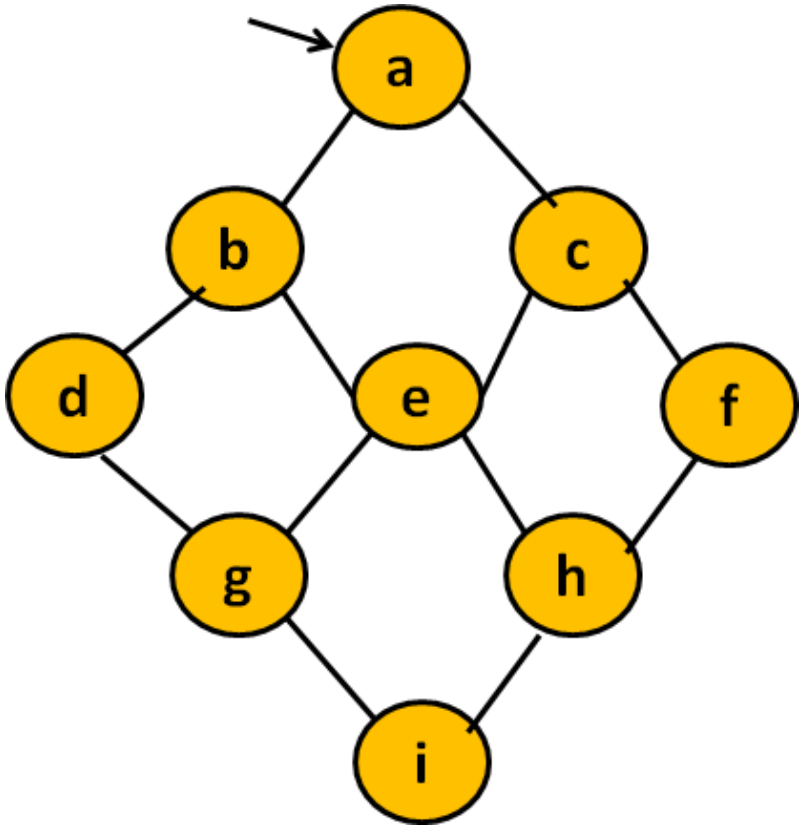
1. Push starting vertex
2. Push any one adjacent vertex
3. Print
4. Back Traverse if required



DFS EXAMPLE

STACK

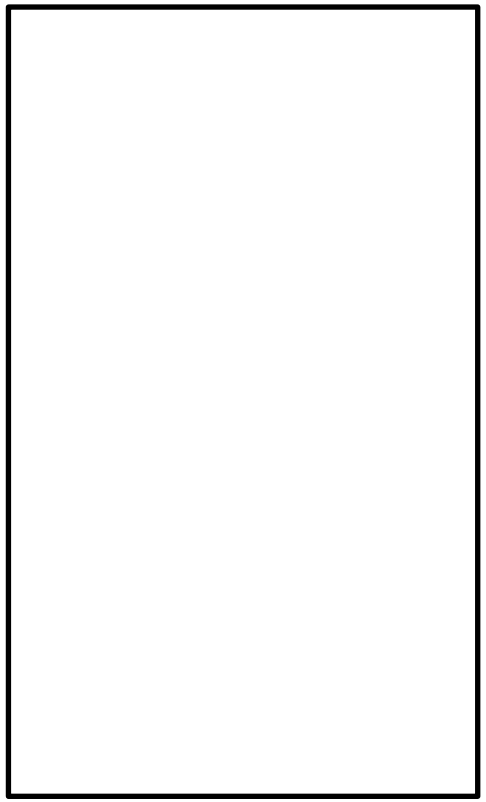
1. Push starting vertex
2. Push any one adjacent vertex
3. Print
4. Back Traverse if required

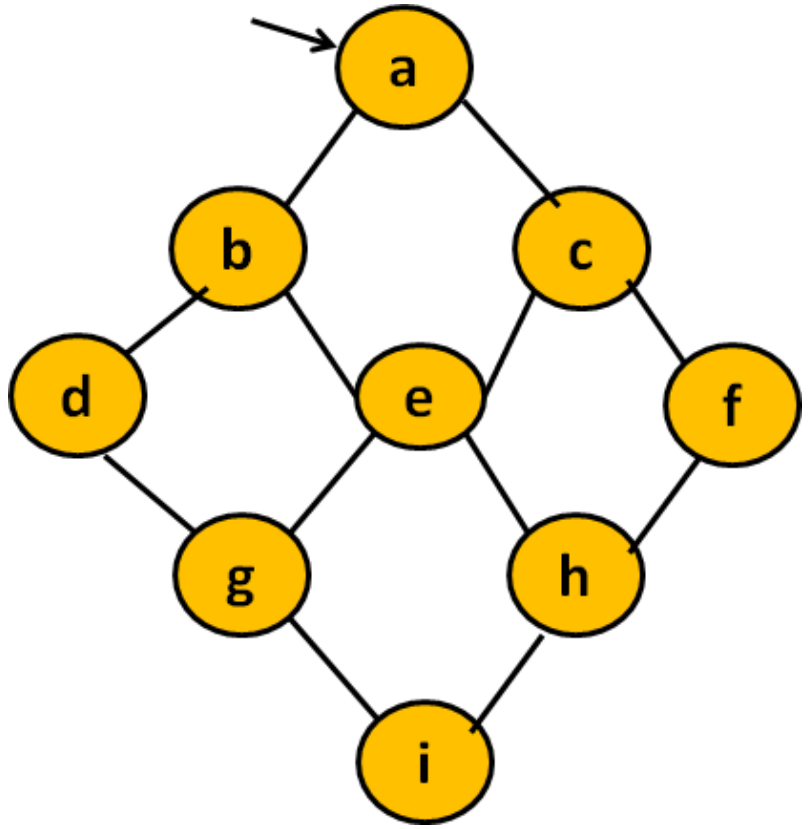


O/P :

STACK

1. Push starting vertex
2. Push any one adjacent vertex
3. Print
4. Back Traverse if required (POP)





O/P : a b d g i h f c e

DFS EXAMPLE

BFS	DFS
1. Breadth First Search	2. Depth First Search
2. Uses Queue DS	2. Uses Stack DS
3. Used to find single source shortest path in an unweighted graph.	3. Not like that.
4. We reach a vertex with min. no. of edges from a source vertex.	4. We might traverse through more edges to reach a destination vertex from a source.
5. More suitable for searching vertices which are closer to the given source.	5. More suitable when there are solutions away from source.
6. Considers all nighbours first. So not suitable for desision making trees used in games or puzzles.	6. Make decision then explore all paths. If desicion leads to win situation, then stop, So, suitable for game or puzzle problems.
7. Time Complexity = O(V+E)	7. Time Complexity = O(V+E)
8. Used in Branch & Bound.	8. Used in Backtracking
9. Act as level-order in BT.	9. Act as pre-order in BT.

What is Minimum Spanning Tree?

Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together.

1. A single graph can have many different spanning trees.
2. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree.

How many edges does a minimum spanning tree has?

A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

KRUSKAL'S MINIMUM SPANNING TREE

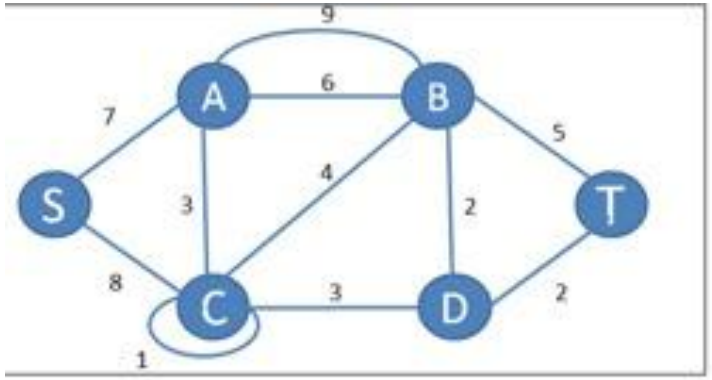
- The graph must be weighted, connected and undirected.
- If the edges are already sorted, then there is no need to construct min heap.
- So, deletion from min heap time is saved.
- In this case, time complexity of Kruskal's Algorithm = O(E + V)
- Kruskal's Algorithm is preferred when the graph is sparse i.e. when there are less number of edges in the graph like E = O(V) or when the edges are already sorted or can be sorted in linear time.

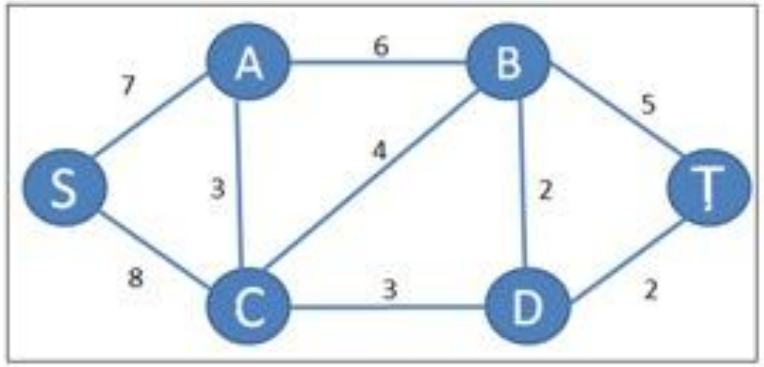
Will both Prim's Algorithm and Kruskal's Algorithm always produce the same Minimum Spanning Tree (MST) for any given graph?

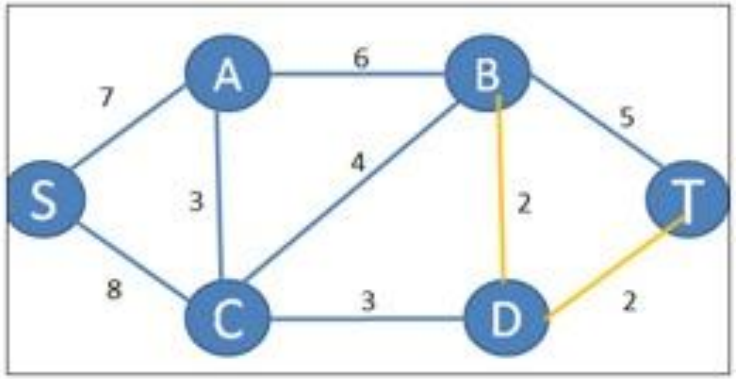
If all the edge weights are distinct, then both the algorithms are guaranteed to find the same i.e. unique MST.

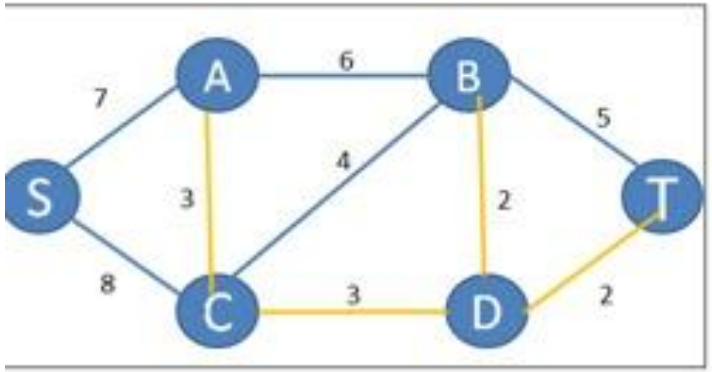
If all the edge weights are not distinct, then both the algorithms may not always produce the same i.e. unique MST but the cost of the MST produced would always be same in both the cases.

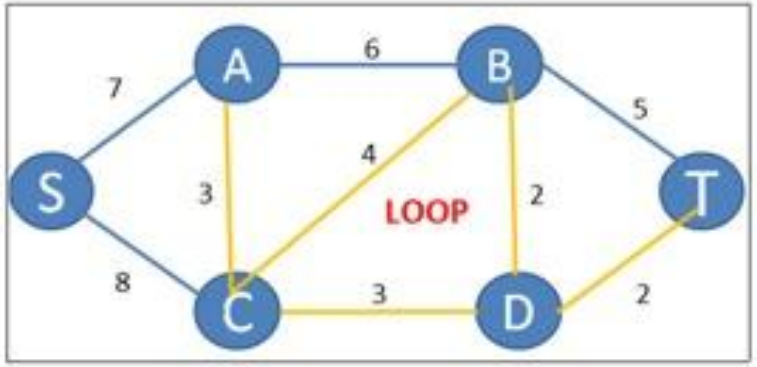
Worst case time complexity of Kruskal's Algorithm = O(ElogV) or O(ElogE).

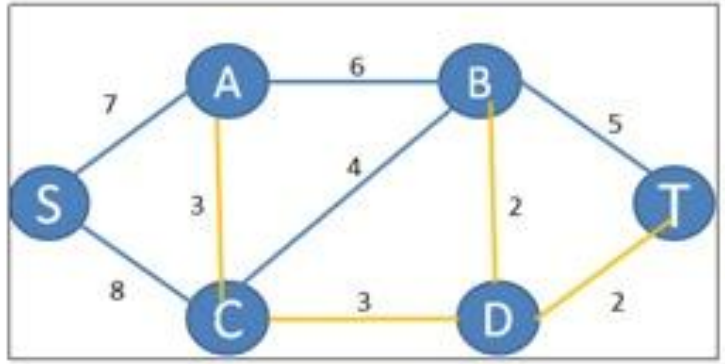


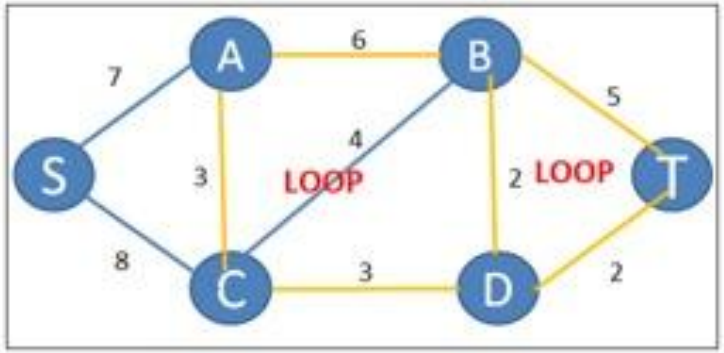


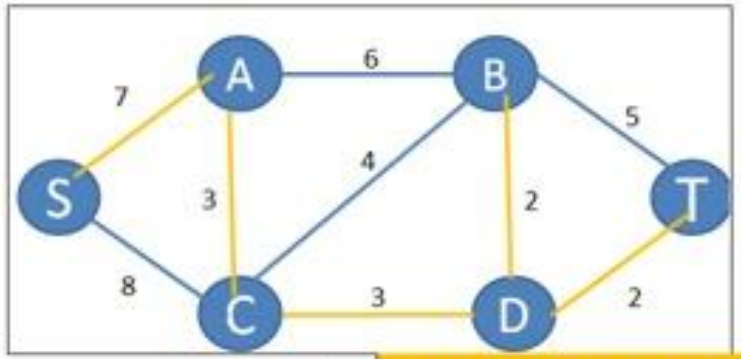


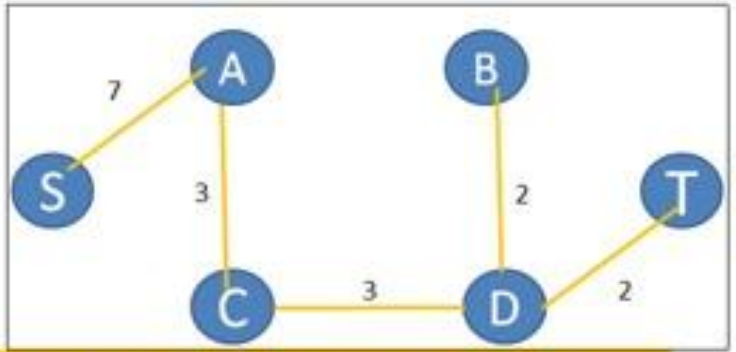












Stopping Condition = MST [Edges = Nodes - 1]

WEIGHT OF Kruskal's MST = 7 + 3+ 3+ 2+ 2 = 17

SHORTEST PATH PROBLEMS

FLOYD WARSHALL

BASIC TO ADVANCED CONCEPTS

Shortest path problem = finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized.

- **The single-source shortest path problem,** in which we have to find shortest paths from a source vertex *v* to all other vertices in the graph.
- **The single-destination shortest path problem**, in which we have to find shortest paths from all vertices in the directed graph to a single destination vertex *v*.
- This can be reduced to the **single-source shortest path** problem by reversing the arcs in the directed graph.
- **The all-pairs shortest path problem,** in which we have to find shortest paths between every pair of vertices *v*, *v'* in the graph

Floyd–Warshall algorithm solves all pairs shortest paths.

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem.

The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Time Complexity-

Floyd Warshall Algorithm consists of three loops over all nodes.

The inner most loop consists of only operations of a constant complexity.

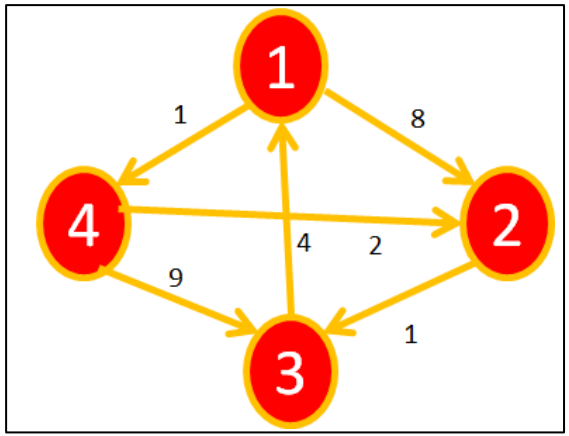
Hence, the asymptotic complexity of Floyd-Warshall algorithm is $O(n^3)$, where n is the number of nodes in the given graph.

When Floyd- Warshall Algorithm is used?

Floyd-Warshall Algorithm is best suited for dense graphs since its complexity depends only on the number of vertices in the graph.

For sparse graphs, Johnson's Algorithm is more suitable.

Floyd Warshall (Dynamic Programming)	O(V^3) There are three nested loops that run through the vertices of the graph.



D0

	1	2	3	4
1	0	8	∞	1
2	∞	0	1	∞
3	4	∞	0	∞
4	∞	2	9	∞

D1

	1	2	3	4
1	0	8	∞	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	9	0

D2

	1	2	3	4
1	0	8	9	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	3	0

D3

	1	2	3	4
1	0	8	9	1
2	5	0	1	∞
3	4	12	0	5
4	7	2	3	0

D4

	1	2	3	4
1	0	3	4	1
2	5	0	1	∞
3	4	7	0	5
4	7	2	3	0

1. Using Floyd-Warshall Algorithm, find the shortest path distance between every pair of vertices. Distance Between 3 to 2 ?

ANSWER (A)

I. Minimum Spanning Tree of G is always unique – MST will always be distinct if the edges are unique **so Correct**

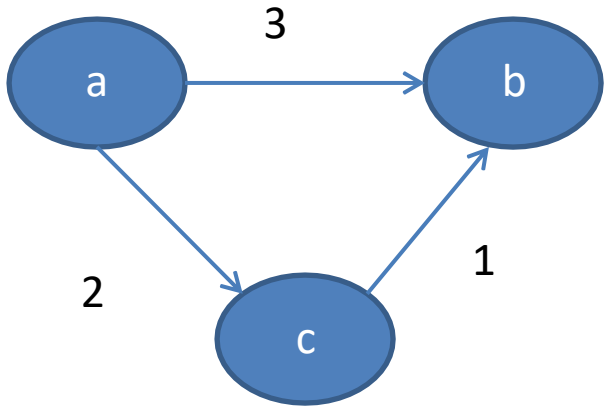
II. Shortest path between any two vertices of G is always unique – Shortest path between any two vertices can be same **so incorrect**

Therefore**, option A** is correct

Background Explanation

We know that minimum spanning tree of a graph is always unique if all the weight are distinct, so statement 1 is correct.
Now statement 2 , this might not be true in all cases.

There are two shortest paths from a to b (one is direct and other via node c) So statement 2 is false
Hence option a is correct.



Shortest path problem = finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized.

- **The single-source shortest path problem,** in which we have to find shortest paths from a source vertex *v* to all other vertices in the graph.
- **The single-destination shortest path problem,** in which we have to find shortest paths from all vertices in the directed graph to a single destination vertex *v*.
- **The all-pairs shortest path problem,** in which we have to find shortest paths between every pair of vertices *v*, *v'* in the graph

- **Dijkstra doesn't work for Graphs with negative weight** edges, Bellman-Ford works for such graphs.
- Bellman-Ford is also simpler than Dijkstra and suites well **for distributed systems.**

Floyd-Warshall Algorithm is best suited for dense graphs since its complexity depends only on the number of vertices in the graph.
For sparse graphs, Johnson's Algorithm is more suitable.

Dijkstra's algorithm solves the single-source shortest path problem with non-negative edge weight.
Bellman–Ford algorithm solves the single-source problem if edge weights may be negative.
Floyd–Warshall algorithm solves all pairs shortest paths.

DJIKSTRA (GREEDY ALGORITHM)	O(V^2) = Matrix Representation O(ELogV) adjacency list representation (Fibonacci Heap)
Bellman Ford (Dynamic Programming)	O(VE)
Floyd Warshall (Dynamic Programming)	O(V^3) There are three nested loops that run through the vertices of the graph.