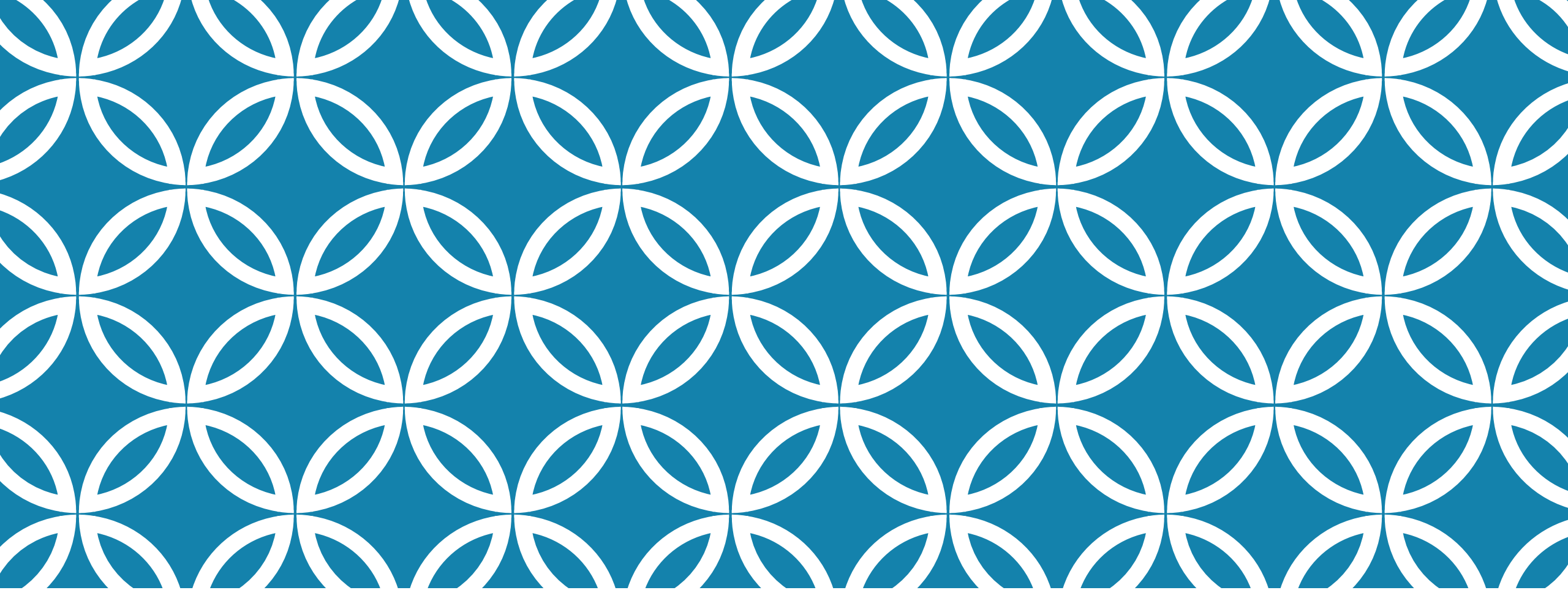


DATA STRUCTURE AND ALGORITHM

UNIT -1

Introduction of DSA

Prepared by: Prof. Kinjal Patel

DATA STRUCTURE

- Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently.
- Examples of Data Structures are arrays, Linked List, Stack, Queue, etc.
- Data Structures are widely used in almost every aspect of Computer Science i.e. operating System, Compiler Design, Artificial intelligence, Graphics and many more.
- Data Structures are the 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

BASIC TERMINOLOGY

Data: Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

Group Items: Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

Record: Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

File: A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

Attribute and Entity: An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

Field: Field is a single elementary unit of information representing the attribute of an entity.

NEED OF DATA STRUCTURES

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.

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.

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

in order to solve the above 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.

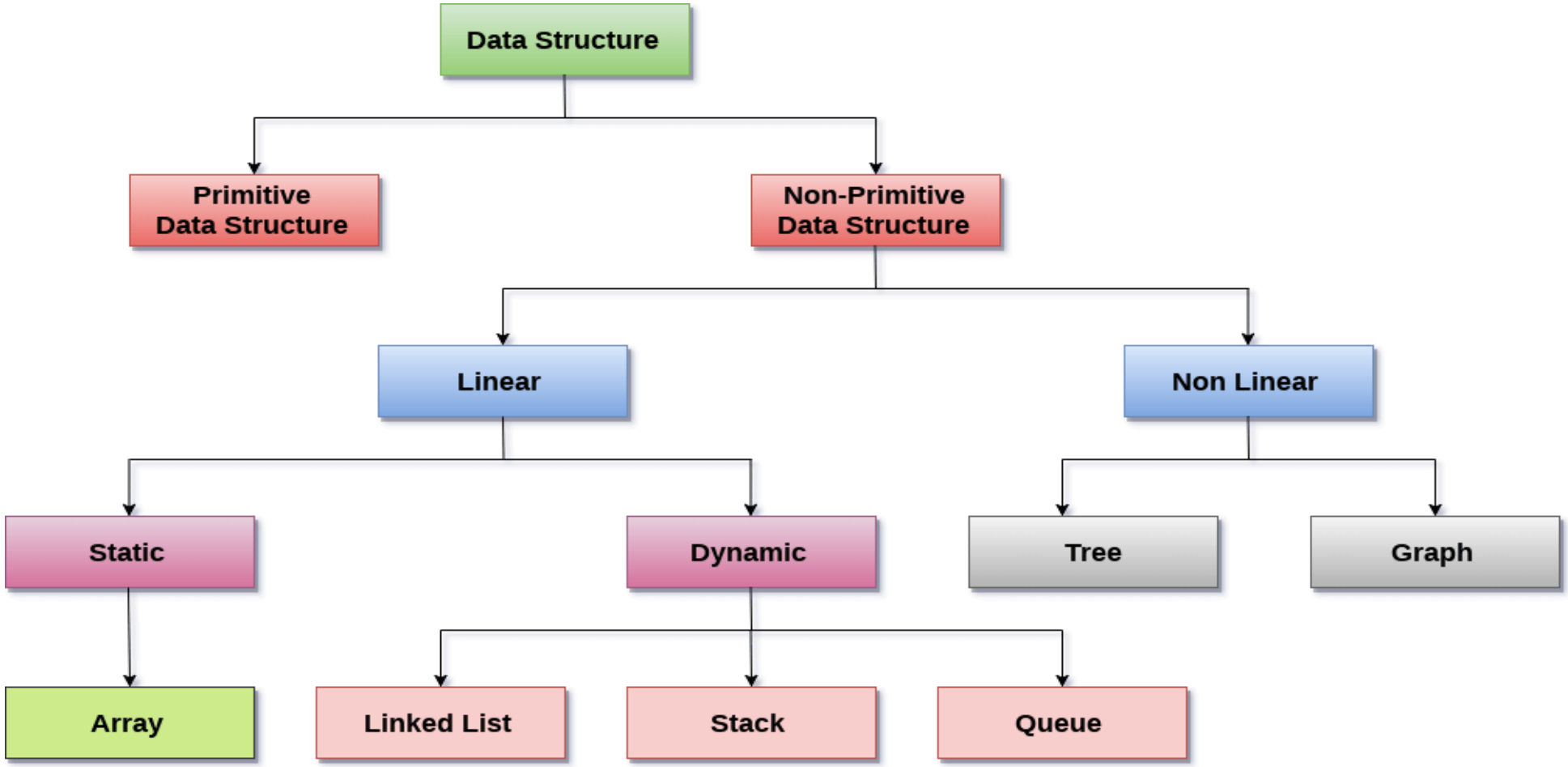
ADVANTAGES OF DATA STRUCTURES

Efficiency: search process efficient like ordered array, binary search tree or hash tables.

Reusability: Data structures are reusable, i.e. 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.

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.

DATA STRUCTURE CLASSIFICATION



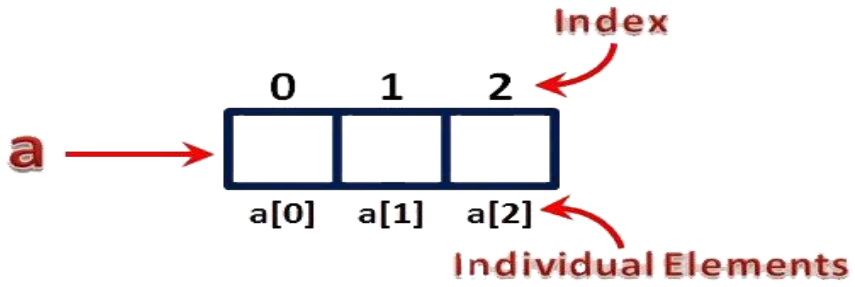
LINEAR DATA STRUCTURES

- A data structure is called linear if all of its elements are arranged in the linear order.
- The elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element

ARRAYS

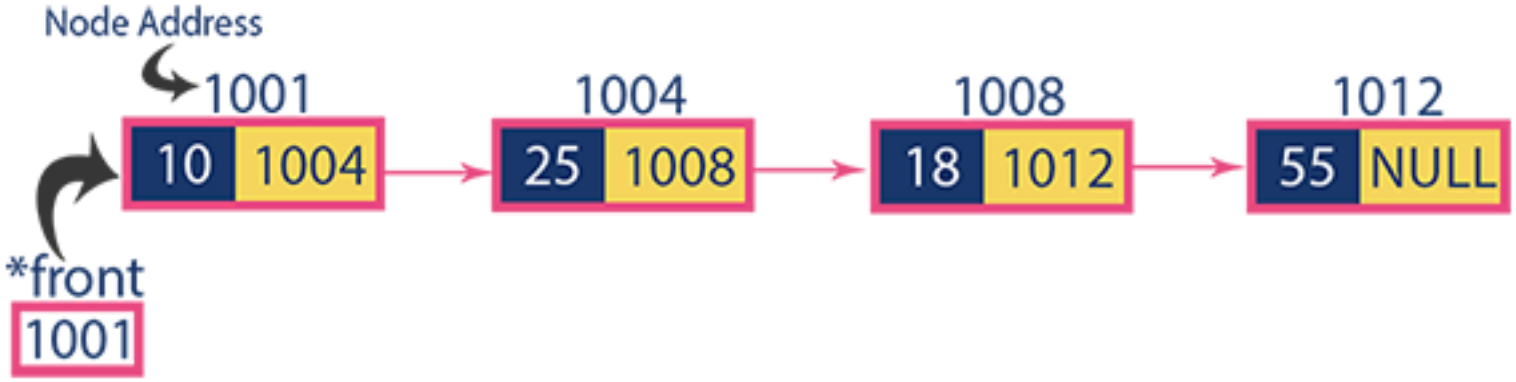
- An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.
- The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.
- The individual elements of the array age are:

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



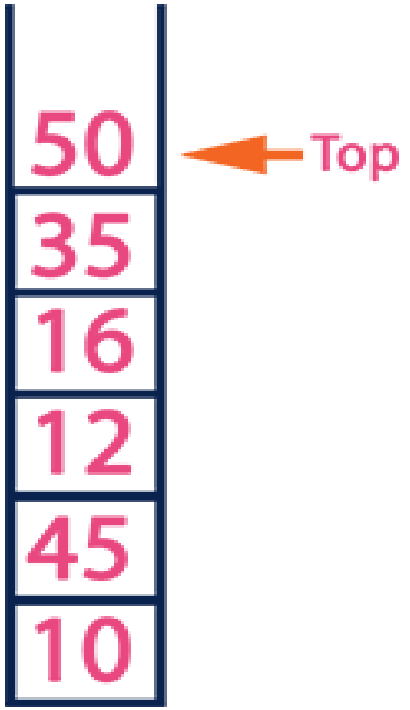
LINKED LIST

- Linked list is 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.



STACK

- Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.
- A stack 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

- 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.

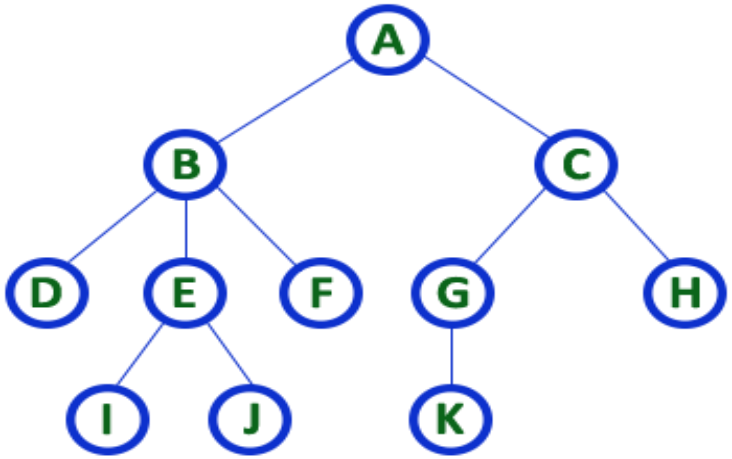
NON LINEAR DATA STRUCTURES

- This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

TREES

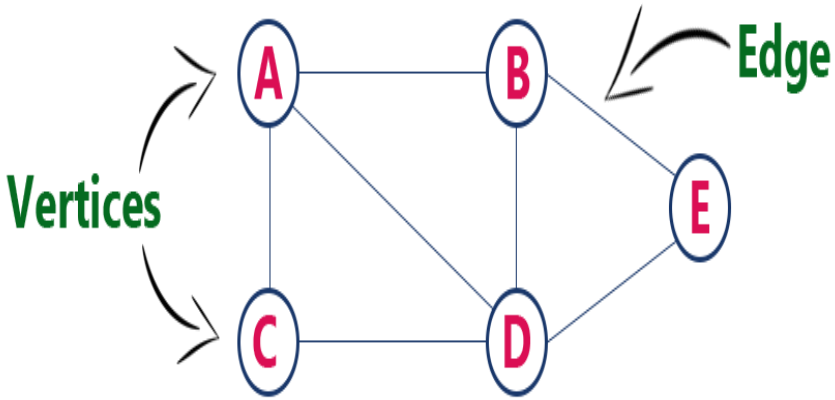
- Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.

- 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. Trees can be classified into many categories which will be discussed later in this tutorial.



GRAPHS

- Graphs can be defined as the pictorial representation of the set of elements (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 cannot have the one.



OPERATIONS ON DATA STRUCTURE

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

Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will divide that sum by the number of subjects i.e. 6, in order to find the average.

2) **Insertion:** Insertion can be defined as 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:** 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.

OPERATIONS ON DATA STRUCTURE

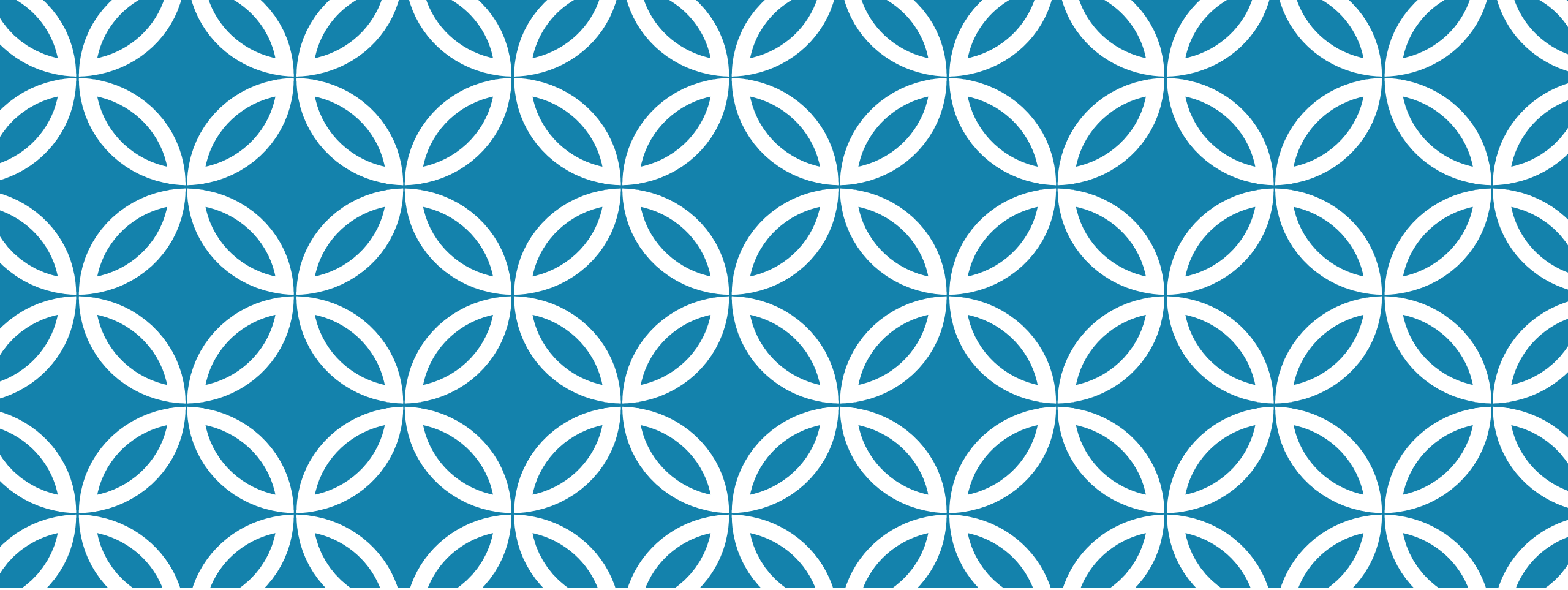
4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

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

ASSIGNMENT:

1. Define Data structure. What is the use of Data structure?
2. Draw the classification of DS.
3. Explain Linear DS with its type in brief.
4. Explain Non-linear DS with its type in brief.
5. Differentiate linear and non-linear DS.
6. WAP to make an integer array and enter 5 different values.
    a. Find max and min from array
    b. Display all elements.



DATA STRUCTURE AND ALGORITHM

UNIT -2

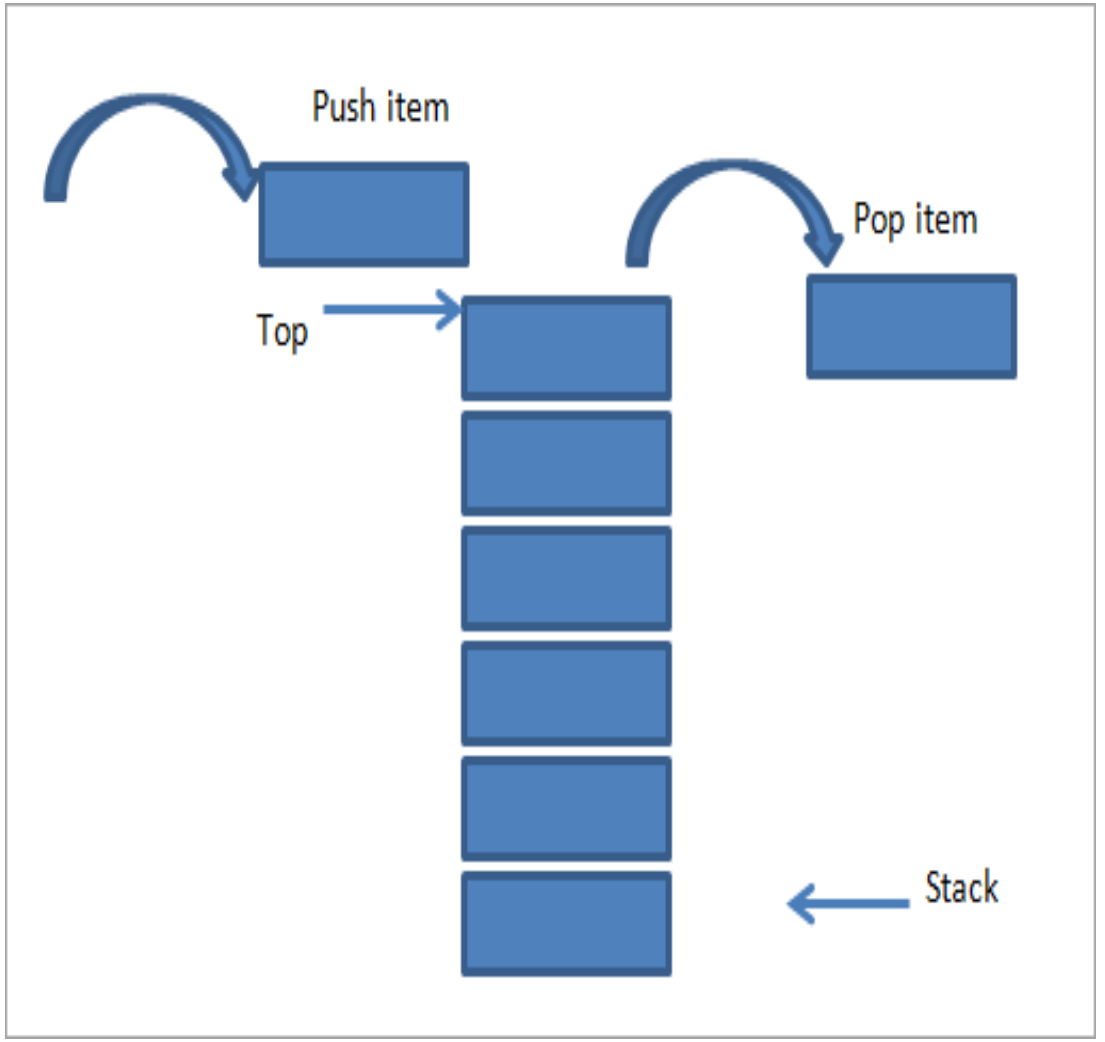
Linear Data Structures –STACK

Prepared by: Prof. Kinjal Patel.

STACK DATA STRUCTURE

- A stack is a linear **OR** non-primitive list in which insertion and deletion operations are performed at only one end of the list.
- A stack is **Last in First out (LIFO)** Structure.
- A stack is a linear data structure in which elements are added and remove from one end, called top of stack.
- **For example of stack** is, consider a **stack of plates on the counter in cafeteria,** during the time of dinner, customer take plates from top of the stack and waiter puts the washed plates on the top of the stack. So new plates are put on the top and old one yet bottom.
- Another example is **Railway Shunting System.**
- Application of stack is **Recursion, Stack Machine, Polish notation.**

- The operations on the stack are represented by using vector consisting of number of elements.
- The insertion operation is referred to as a "PUSH" operation and deletion operation is referred to as a "POP" operation.
- The most accessible element in the stack is known as a TOP element and the least accessible element in the stack is known as a BOTTOM element.
- Since, the insertion and deletion operation are performed at only one end of the stack the element which is inserted last in the stack is first to delete. So stack is also known as Last in First out (LIFO).



OPERATIONS ON STACK:

1) **Push:** The operations that **add an element to the top of the stack** is called **PUSH** operation. It is used to **insert an element** into the stack.
2) **Pop:** The operation that **delete top element from the top of stack** is called **POP**. it is used to delete an element from stack
3) **Peep:** it is used to **retrieve i^{th} element from the top of the stack.**
4) **Change:** it is used to **change value of the i^{th} element from the top of the stack.**

Implementation or Representation of Stack

1 Static / sequential / array representation

- ✓ In this case of array, stack is also collection of homogeneous elements .
- ✓ Therefore stack can be easily implemented using array.
- ✓ Two operation : PUSH & POP
- ✓ Element stored from stack[0] to stack[TOP]
- ✓ Stack[0] ->Bottom of STACK
- ✓ Stack [TOP]-> Top of STACK
- ✓ TOP =-1 It indicate Stack is Empty.

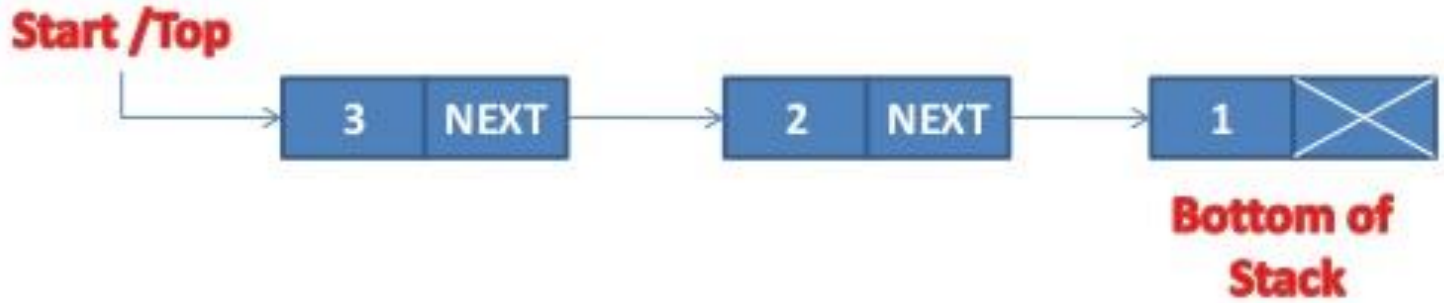
2 Dynamic / pointer/ linked Representation

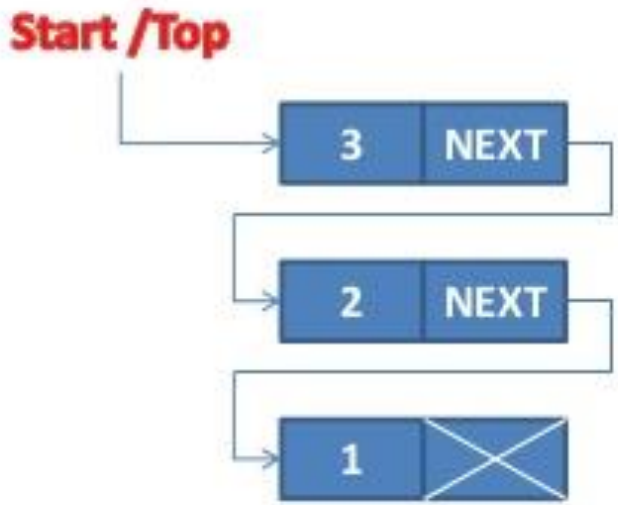
- ✓ In Dynamic or linked representation stack is collection of nodes. Where each node is divided in to two parts

Structure of Node

- ✓ Top of stack is represented by start pointer of linked list
- ✓ If START =NULL , it shows that stack is empty & TOP = NULL
- ✓ Last node Next part contain NULL Value, it Indicate bottom of stack

E.g.





Physical dynamic stack implementation

ALGORITHM TO INSERT AN ELEMENT IN STACK

In push operation, we can add elements to the top of the stack, so before push operation, user must check the stack, it should not be a full.

If stack is already full and when we try to add the elements then error occurs. It is called "**Stack Over Flow**" error

PUSH (S, N, TOP,VAL)

- Stack is represented by vector S which contains N elements.
- TOP is a pointer which points to the top element of the Stack.
- VAL is the element to be inserted.

(1) [Check for Stack Overflow]

If TOP ==N-1 then

Print "Stack Overflow"

Exit

(2) [Increment TOP pointer]

TOP=TOP+1

(3) [Insert new element at TOP]

S[TOP] = VAL

(4) [Finish]

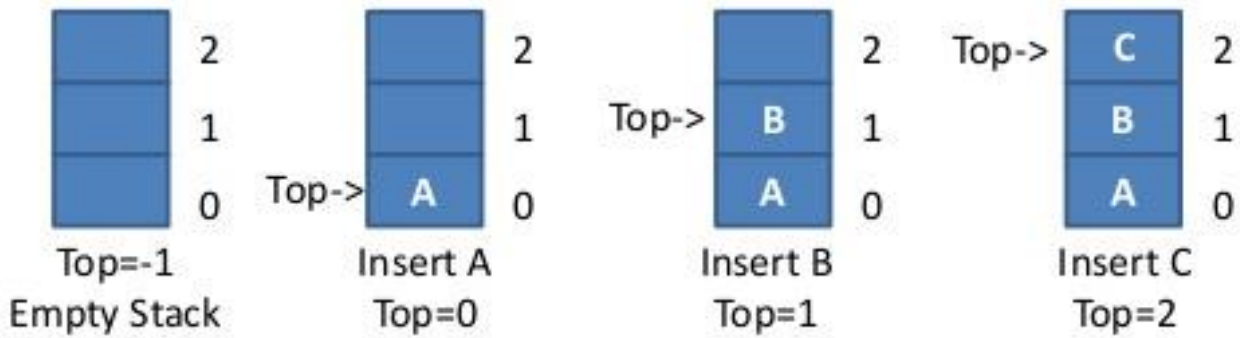
Exit

Stack Operation

1. PUSH Operation

✓ Used to insert element into Stack

e.g.



✓ Each time new element is inserted into stack, the value of Top is incremented by one

✓ Top=2 i.e. MAX-1. It shows stack is **Overflow**

ALGORITHM TO DELETE AN ELEMENT FROM STACK

In POP operation, we can delete or remove an elements from top of the stack, so before pop operation, user must check the stack, stack should not be a empty.

If the stack is empty, and we try to pop an element, then error occur. It is called "**Stack under Flow**" error

POP(S, TOP)

- Stack is represented by vector S which contains N elements.
- TOP is a pointer which points to the top element of the Stack.

(1) [Check for underflow]

If TOP<0 then

Print "Stack underflow"

Exit

(2)[Copy the element from the top of stack]

VAL= S[TOP]

(3) [Decrement the TOP pointer]

TOP=TOP-1

(4) [Return the deleted value]

Return VAL

(5) [Finish]

Exit

Stack Operation

2. POP Operation

✓ Used to Delete element From Stack

e.g.

- ✓ In POP Last inserted element is Deleted First [LIFO]
- ✓ After deleting element value of top is decreased by one
- ✓ When all elements from stack are removed (i.e. empty stack) we cannot removed anything from stack. This condition is called "**Underflow**"

ALGORITHM TO PEEP AN ELEMENT FROM STACK

Peep is an operation that returns the value of the i^{th} element from the top of the stack without deleting it from the stack

PEEP(S, TOP ,I)

- Stack is represented by vector S which contains N elements.
- TOP is a pointer which points to the top element of the Stack.
- I is the position of the element want to access.

(1) [Check for underflow]

If TOP - i +1 <0 then

Print "Stack underflow"

Exit

(2)[Return i^{th} element from the top of the stack]

return S[TOP - I +1]

(3) [Finish]

Exit

ALGORITHM TO CHANGE AN ELEMENT IN STACK

Change operation changes the value of the i^{th} element from the top of the stack to the value containing in VAL

CHANGE(S, TOP ,I ,VAL)

- Stack is represented by vector S which contains N elements.
- TOP is a pointer which points to the top element of the Stack.
- I is the position of the element want to change with new value VAL

(1) [Check for underflow]

If TOP- i +1 <0 then

Print "Stack underflow"

Exit

(2)[Replace the i^{th} element from the top of the stack with value VAL]

S[TOP- I +1] = VAL

(3) [Finish]

Exit

APPLICATION OF STACK

There are three main application of stack:

(1) Recursion

(2) Evaluate polish notation

(3) Stack Machine

(4) Tower of Hanoi

(5) Reversing a list

RECURSION

- **Recursion** means function call itself.
- It is the technique of defining a process in terms of itself.
- Stack is widely used in recursion because of its Last in First out Property.
- In terms of C language the function calling itself is known as recursion.
- There are two types of recursion:

    (1) Primitive Recursive function: Example Factorial Function

    (2) Non primitive Recursive function: Example **Fibonacci Series**

PRIMITIVE V/S NON PRIMITIVE RECURSIVE FUNCTION

```
int Fact(int n)
{
    if (n == 1)
        return 1;
    else
        return (n * Fact(n-1));
}
```

Primitive Recursive function

```
int Fibonacci(int num)
{
  if(num == 0)
    return 0;
  else if (num == 1)
    return 1;
   else
    return (Fibonacci(num - 1) + Fibonacci(num - 2));
}
```

Non Primitive Recursive function

EVALUATE POLISH NOTATION

- There are basically three types of polish notation:
- **(A) Infix (B) Prefix (C) Postfix**
- When the operator(**+**) exists between two operands(**A & B**) then it is known as Infix notation.(**e.g. A+B**)
- When the operator(**+**) are written before their operands(**A & B**) then it is known as Prefix notation(**Polish notation**). **(e.g. +AB)**
- When the operator(**+**) are written after their operands(**A & B**) then it is known as Postfix notation(**Reverse polish notation**). **(e.g. AB+).**
- Stack is widely used to convert infix notation into prefix or postfix notation.
- One of the major uses of stack is a polish notation or polish expression.
- The process of writing the operators of an expression either before their operands or after operands are called the polish notation.

RULES FOR CONVERTING INFIX NOTATION TO PREFIX/ POSTFIX

1) The operations with heights precedence are converted first and then after a portion of the expression have been converted to postfix.
2) It is to be treated as single operand.
3) Take only two operands at a time to convert in a postfix from like A+B→ AB+
4) Always, convert the parenthesis first
5) Convert postfix exponentiation second if there are more than one exponentiation in sequence then take order from right to left.
6) Convert in to postfix as multiplication and division operator, left to right.
7) Convert in postfix as addition and subtraction left to right.

CONVERT INFIX INTO POLISH NOTATION (PREFIX)

(A + B) * C

= (+ A B) * C

= * (+ A B) C

= * + A B C

Convert following infix notation to polish notation

1) (A + B) * (C + D)

2) (A-B)+C*A+B

3) (B*C/D)/(D/C+E)

4) A*B/(C+D-E)

1. (A + B) * (C + D) = (+ A B) * (+ C D) =*(+ A B) (+ C D) =* + AB + CD	**2. (A-B)+C*A+B** =-AB+C*A+B =-AB+*CA+B =+-AB*CA+B =++-AB*CAB
3. (B*C/D)/(D/C+E) =(*BC/D)/(D/C+E) =(/*BCD)/(D/C+E) =(/*BCD)/(/DC+E) =(/*BCD)/(+/DCE) =//*BCD+/DCE	**4. A*B/(C+D-E)** =A*B/(+CD-E) =A*B/(-+CDE) =*AB/-+CDE =/*AB-+CDE

CONVERTS FOLLOWING INFIX INTO REVERSE POLISH NOTATION (POSTFIX).

A * B / C +D

= [AB*]/C+D

= [AB*C/] +D

=AB*C/D+

Convert following infix notation to reverse polish notation

1) **C*B*A/D**
2) **(A-B)+C*A+B**
3) **(B*C/D)/(D/C+E)**
4) **A*B+C*D+E*F**
5) **A*B+A*(B*D+E*F)**
6) **A*B/(C+D-E)**
7) **A*A-B*B**

1. C*B*A/D = [CB*]*A/D = [CB*A*]/D =CB*A*D/	**2. (A-B)+C*A+B** = [AB-] +C*A+B = [AB-] + [CA*] +B = [AB-CA*+] +B =AB-CA*+B+	**3. (B*C/D)/(D/C+E)** = ([BC*]/D)/(DC/+E) = [BC*D/]/[DC/E+] =BC*D/DC/E+/
4. A*B+C*D+E*F =[AB*]+C*D+E*F =[AB*]+[CD*]+E*F =[AB*]+[CD*]+[EF*] =[AB*CD*+]+[EF*] =AB*CD*+EF*+	**5. A*B+A*(B*D+E*F)** =A*B+A*([BD*]+[EF*]) =A*B+A*([BD*EF*+]) =[AB*]+A*[BD*EF*+] =[AB*]+[ABD*EF*+*] =AB*ABD*EF*+*+	**6. A*B/(C+D-E)** =A*B/([CD+]-E) =A*B/[CD+E-] =[AB*]/[CD+E-] =AB*CD+E-/
7. A*A-B*B =[AA*]-B*B =[AA*]-[BB*] =AA*BB*-		

CONVERT INFIX TO POSTFIX WITHOUT PARENTHESIS

Symbol	Precedence	Rank
+, -	1	**-1**
*, /	2	**-1**
Variables	3	**1**
#	**0**	**-**

CONVERT INFIX TO POSTFIX WITHOUT PARENTHESIS A+B*C-D/E*H

Character scanned	Content of stack	Reverse police notation	Rank
	#		
A	# A		
+	# +	A	1
B	# + B	A	1
*	# + *	A B	2
C	# + * C	A B	2
-	# -	A B C * +	1
D	# - D	A B C * +	1
/	# - /	A B C * + D	2
E	# - / E	A B C * + D	2
*	# -*	A B C * + D E /	2
H	# - * H	A B C * + D E /	2
#	#	A B C * + D E / H * -	

CONVERT INFIX TO POSTFIX WITH PARENTHESIS

Symbol	Input precedence function F	Stack precedence function G	Rank function R
+, -	1	2	**-1**
*, /	3	4	**-1**
^	6	5	**-1**
Variables	7	8	**1**
(	9	0	**-**
)	**0**	**-**	**-**

CONVERT INFIX TO POSTFIX WITH PARENTHESIS (A+B^C^D)*(E+F/D))

Character scanned	Content of stack	Reverse police notation	Rank
	(		
(	((		
A	((A		
+	((+	A	1
B	((+B	A	1
^	((+^	AB	2
C	((+^C	AB	2
^	((+^^	ABC	3
D	((+^^D	ABC	3
)	(	ABCD^^+	1
*	(*	ABCD^^+	1
(	(*(	ABCD^^+	1
E	(*(E	ABCD^^+	1
+	(*(+	ABCD^^+E	2
F	(*(+F	ABCD^^+E	2
/	(*(+/	ABCD^^+EF	3
D	(*(+/D	ABCD^^+EF	3
)	(*	ABCD^^+EFD/+	2
)		ABCD^^+EFD/+*	1

(ii) (A + B) * C + D / (B + A * C) + D

Input Symbol	Content of stack	Reverse polish	Rank
	(		0
(	((		0
A	((A		0
+	((+	A	1
B	((+ B	A	1
)	(	A B +	1
*	(*	A B +	1
C	(* C	A B +	1
+	(+	A B + C *	1
D	(+ D	A B + C *	1
/	(+ /	A B + C * D	2
(	(+ / (	A B + C * D	2
B	(+ / (B	A B + C * D	2
+	(+ / (+	A B + C * D B	3
A	(+ / (+ A	A B + C * D B	3
*	(+ / (+ *	A B + C * D B A	4
C	(+ / (+ * C	A B + C * D B A	4
)	(+ /	A B + C * D B A C * +	3
+	(+	A B + C * D B A C * + / +	1
D	(+ D	A B + C * D B A C * + / +	1
)		A B + C * D B A C * + / + D +	1

Postfix expression is: **A B + C * D B A C * + / + D +**

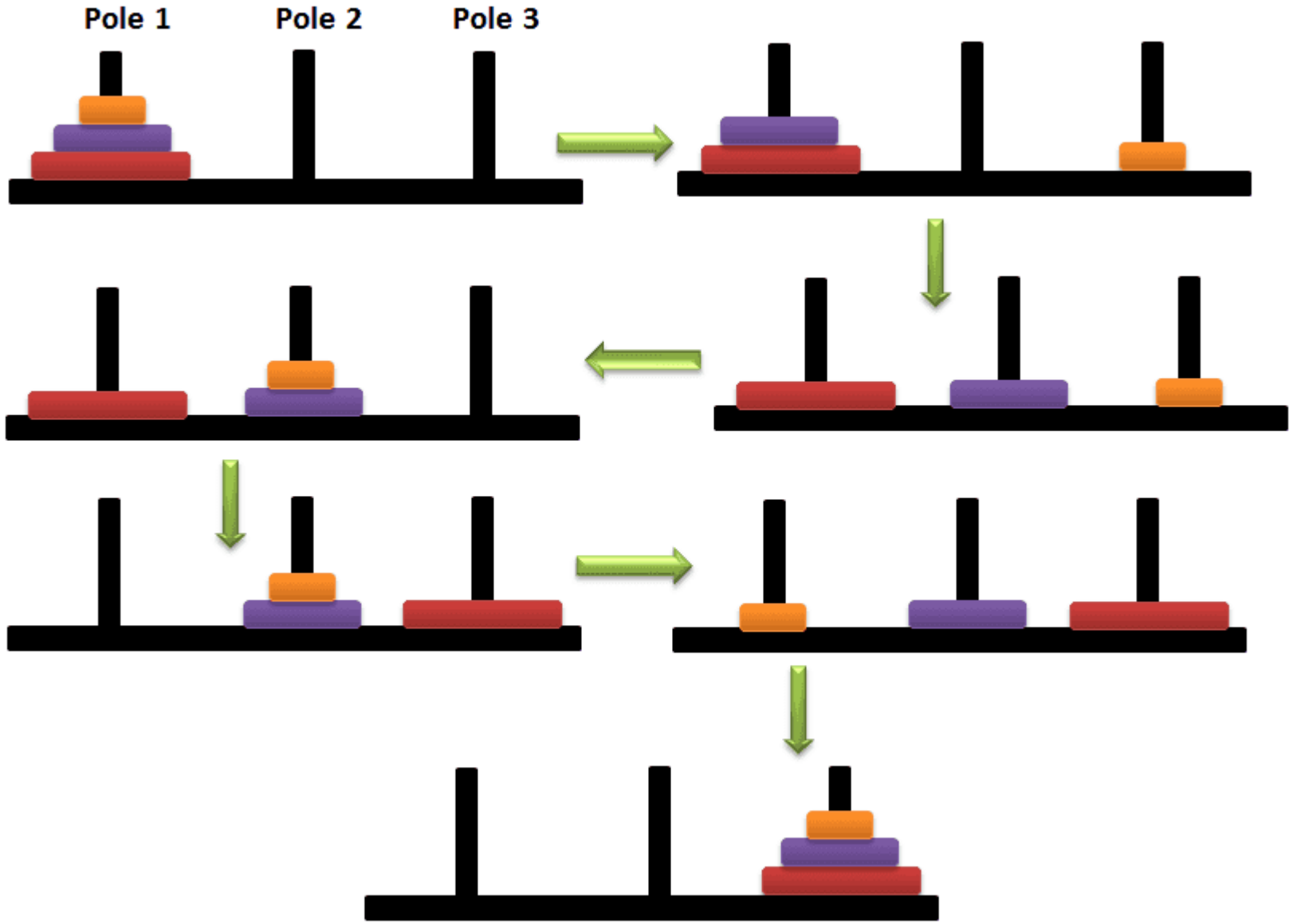
1. [Initialize stack]
   TOP ← 1
   S[TOP] ← '('
2. [Initialize output string and rank count]
   POLISH ← ' '
   RANK ← 0
3. [Get first input symbol]
   NEXT←NEXTCHAR (INFIX)
4. [Translate the infix expression]
   Repeat thru step 7 while NEXT != ' '
5. [Remove symbols with greater precedence from stack]
   IF TOP < 1
   Then write ('INVALID')
   EXIT
   Repeat while G (S[TOP]) > F(NEXT)
   TEMP ← POP (S, TOP)
   POLISH ← POLISH O TEMP
   RANK ← RANK + R(TEMP)
   IF RANK <1
   Then write 'INVALID')
   EXIT
6. [Are there matching parentheses]
   IF G(S[TOP]) != F(NEXT)
   Then call PUSH (S,TOP, NEXT)
   Else POP (S,TOP)
7. [Get next symbol]
   NEXT ← NEXTCHAR(INFIX)
8. [Is the expression valid]
   IF TOP != 0 OR RANK != 1
   Then write ('INVALID ')
   Else write ('VALID ')

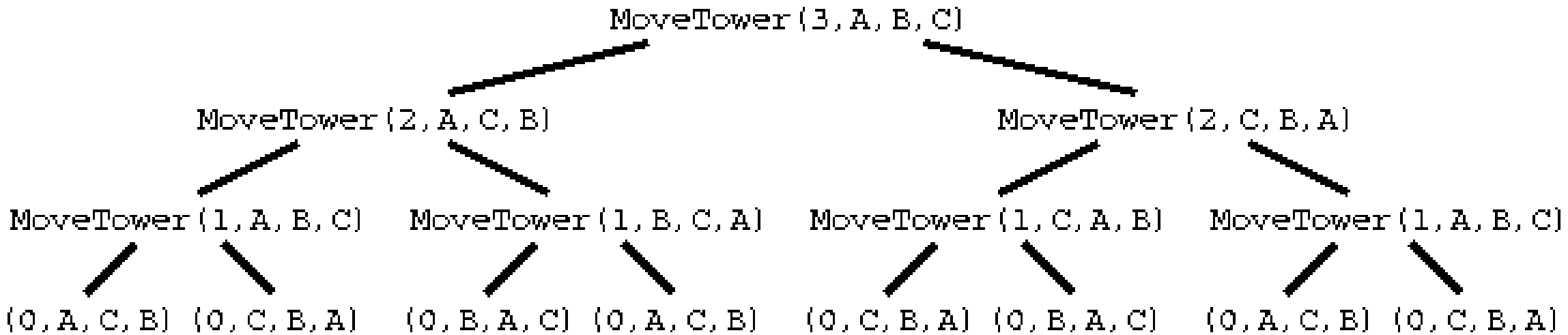
TOWER OF HANOI

Tower of Hanoi is a mathematical puzzle which consist of **3 poles** and number of **discs** of different sizes. Initially all the discs will be places in the single pole with the largest disc at the bottom and smallest on the top. We need to move all the disc from the first pole to the third pole with the smallest disc at the top and the largest at the

1. Only one disc can be moved at a time.
2. Larger disc cannot be placed on a smaller disc.

https://www.youtube.com/watch?v=fffbT41luB4


Pole 1
Pole 2
Pole 3


MoveTower(3, A, B, C)
MoveTower(2, A, C, B)
MoveTower(2, C, B, A)
MoveTower(1, A, B, C)
MoveTower(1, B, C, A)
MoveTower(1, C, A, B)
MoveTower(1, A, B, C)
(0, A, C, B)
(0, C, B, A)
(0, B, A, C)
(0, A, C, B)
(0, C, B, A)
(0, B, A, C)
(0, A, C, B)
(0, C, B, A)

ASSIGNMENT:

1. What is stack? What are the characteristics of stack.
2. Write algorithm for PUSH, POP, PEEK and CHANGE operation in stack.
3. Convert following expression into the prefix and postfix notation:

   (i) a+b*(c/d)-e

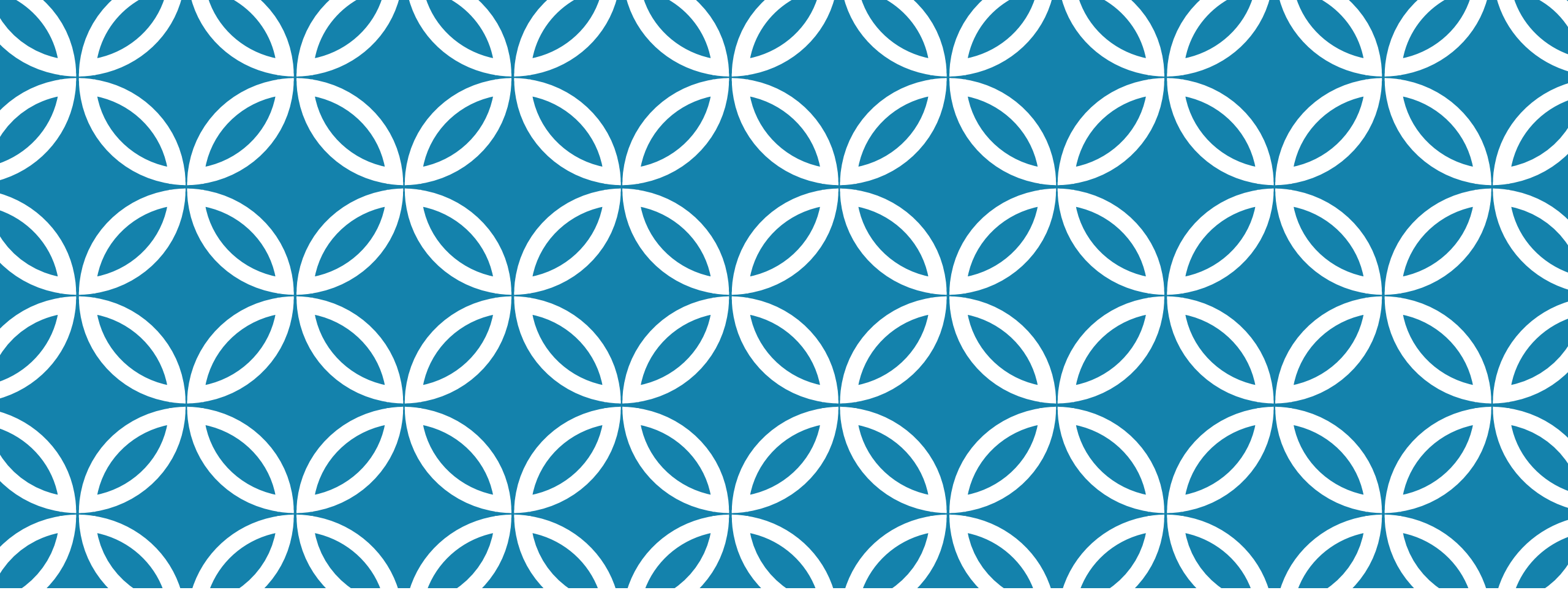
   (ii) (a*b) * (c^(d+e)-f)

4. Covert following expression into reverse polish notation using algorithm. Show all intermediate steps by table

   (i) A+B*C/D-E/F

   (ii) (A+B)^C-(D*E)/F)

4. Explain tower of Hanoi problem.



DATA STRUCTURE AND ALGORITHM

UNIT -2

Linear Data Structures –Queue

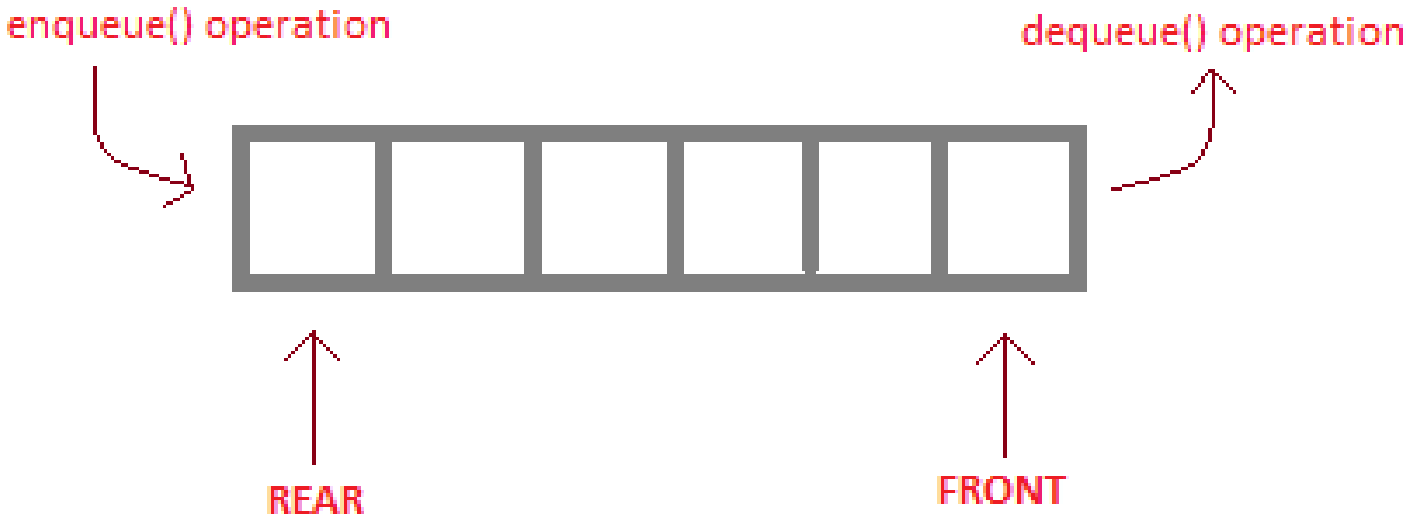
Prepared by: Prof. Kinjal Patel

QUEUE

- A linear list which permits deletion to be performed at one end of the list and insertion at the other end is called queue.
- The information in such a list is processed FIFO (first in first out) of FCFS (first come first served) pattern.
- Front is the end of queue from that deletion is to be performed.
- Rear is the end of queue at which new element is to be inserted.
- The process to add an element into queue is called **Enqueue**
- The process of removal of an element from queue is called **Dequeue.**

Examples of Queue

- A row of students at registration counter
- The bullet in a machine gun.(you cannot fire 2 bullets at the same time)
- Line of cars waiting to proceeds in some direction at traffic signal



enqueue( ) is the operation for adding an element into Queue.

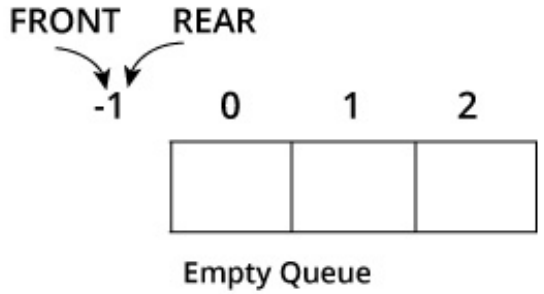
dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

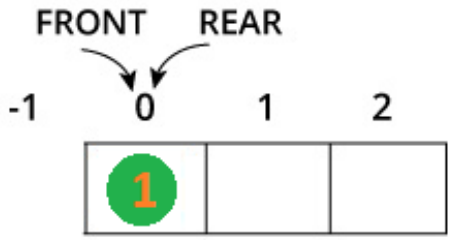
APPLICATION OF QUEUE

- Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
- Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
- Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
- Queues are used in operating systems for handling interrupts.
- A mailbox or port to save messages to communicate between two users or processes in a system is essentially a queue-like structure.
- Queue is widely used in Simulation.

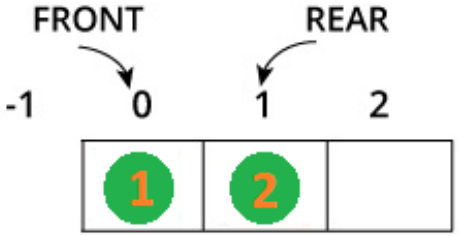
ENQUEUE



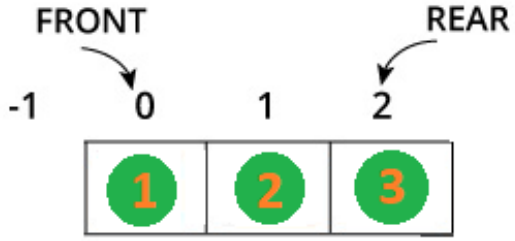
Empty Queue



EnQueue first element



EnQueue



EnQueue

http://www.cs.armstrong.edu/liang/animation/web/Queue.html

ALGORITHM TO INSERT AN ELEMENT IN QUEUE

Enqueue (Q, F, R, N, VAL)

- Queue is represented by vector Q which contains N elements.
- F is a pointer which points to the front end
  R is a pointer which points to the rear end
- VAL is the element to be inserted.

(1) [Check for Overflow]

If R >=N then

Print “Queue Overflow”

Exit

(2) [Increment rear pointer and check front pointer]

If F==0 then

F=R=1

else

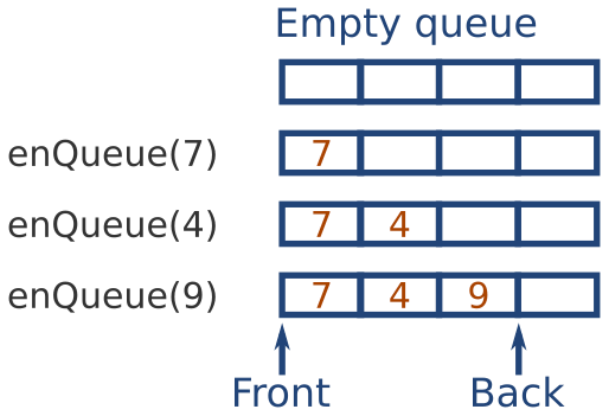
R=R+1

(3) [Insert new element at rear]

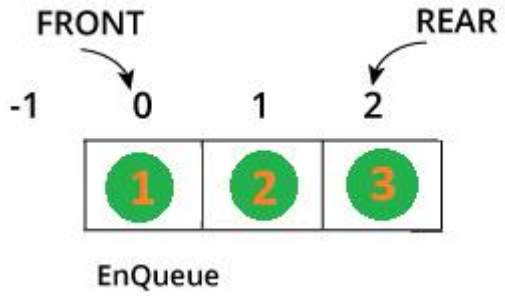
Q[R] = VAL

(4) [Finish]

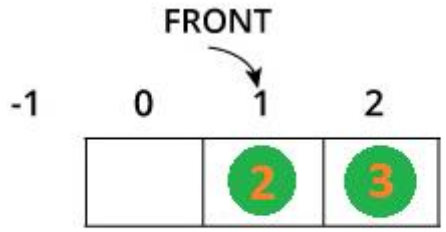
Exit



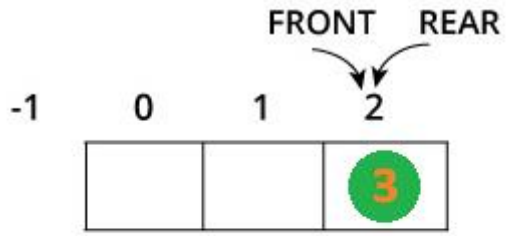
DQUEUE



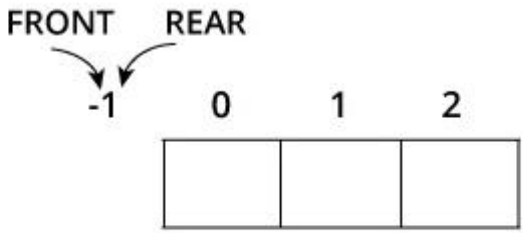
EnQueue



DeQueue



DeQueue last element



Empty Queue

ALGORITHM TO DELETE AN ELEMENT FROM QUEUE

Dqueue (Q, F, R)

- Queue is represented by vector Q which contains N elements.
- F is a pointer which points to the front end
  R is a pointer which points to the rear end

(1) [Check for underflow]

```
If F==0 AND R ==0 then
        Print "Queue underflow"
        Exit
```

(2)[Copy the front element]

```
Y= Q[F]
```

(3) [Increment the front pointer]

```
If F==R then
        F=R= 0
else
        F++
```

(4) [Return the deleted value]

```
Return Y
```

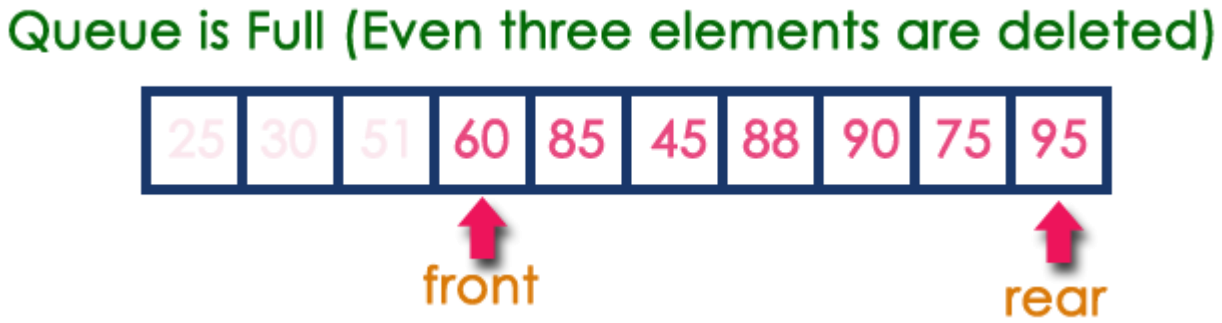
(5) [Finish]

```
Exit
```

LIMITATION OF SIMPLE QUEUE

normal Queue Data Structure, we can insert elements until queue becomes full. But once the queue becomes full, we can not insert the next element until all the elements are deleted from the queue.

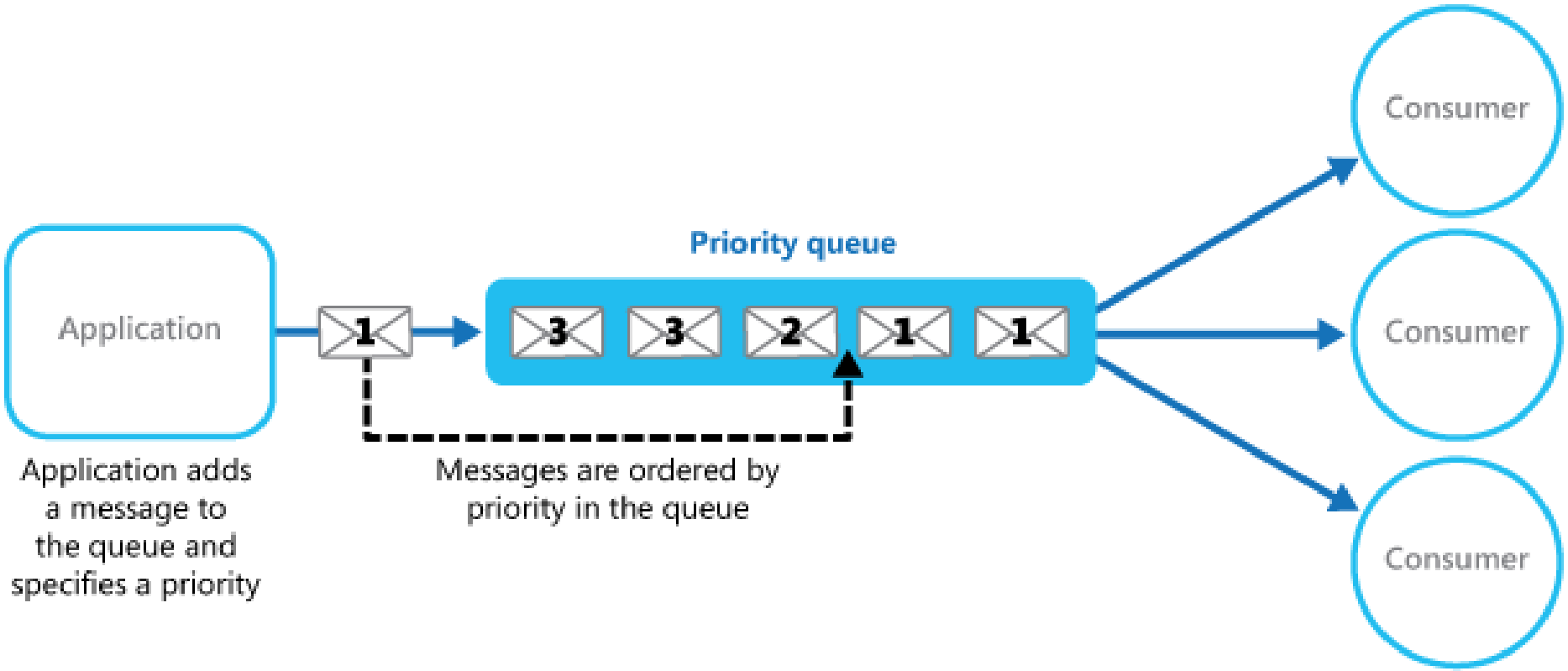
Queue is Full and we cannot insert the new element because '**rear**' is still at last position



DIFFERENCE BETWEEN STACK AND QUEUE

LIFO	FIFO
(1) In LIFO the insertion and deletion operation are performed at only one end.	(1) In FIFO the insertion and deletion operation are performed at two different ends.
(2) In LIFO the element which is inserted last is first to delete.	(2) In FIFO the element which is inserted first is first to delete.
(3) LIFO require only one pointer called TOP	(3) FIFO requires two pointers called front and rear.
(4) Example: piles of trays in cafeteria	(4) Example: students at registration counter
(5) In LIFO there is no wastage of memory space.	(5) In FIFO even if we have free memory space sometimes we cannot use that space to store elements.

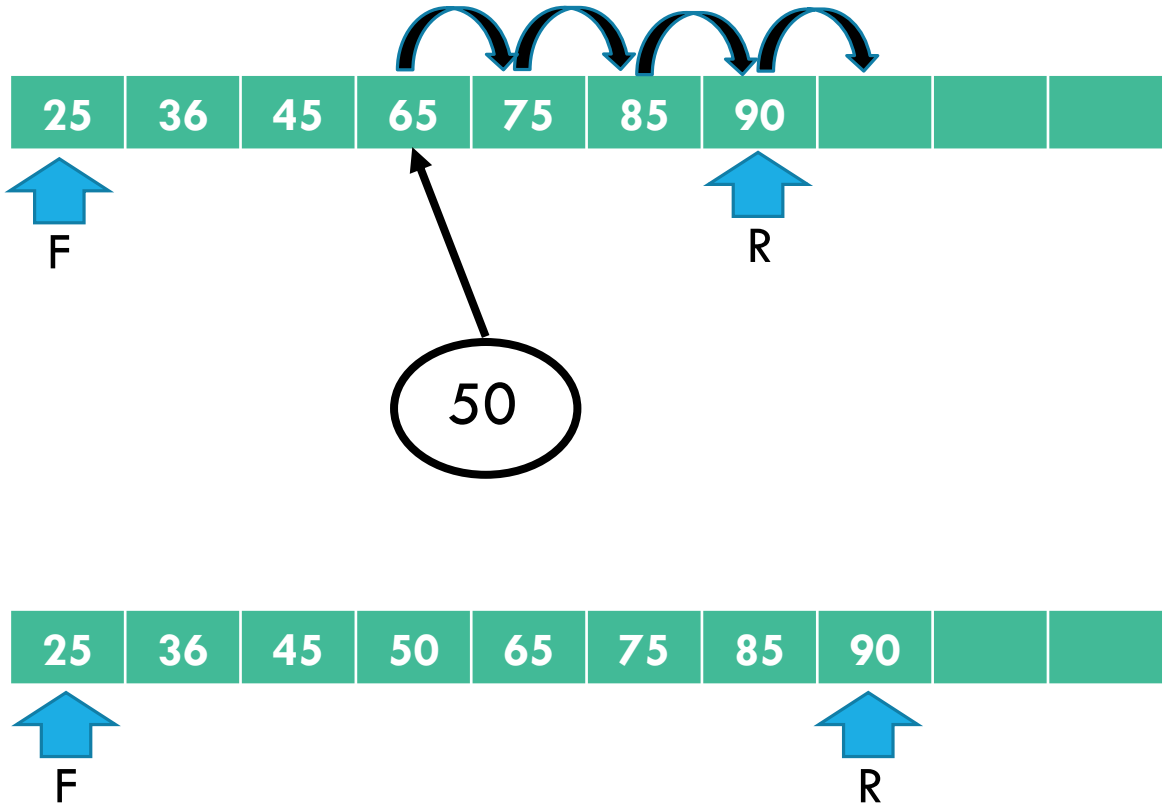
PRIORITY QUEUE



PRIORITY QUEUE

- Priority Queue is like a regular queue, but each element has a “priority” associated with it.
- In a priority queue, an element with high priority is served before an element with low priority.
- For this, it uses a comparison function which imposes a total ordering of the elements.
- The elements of the priority queue are ordered according to their priorities.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

PRIORITY ENQUEUE



ALGORITHM TO INSERT AN ELEMENT IN PRIORITY QUEUE

PEnqueue(Queue, F, R, X)

This procedure inserts a new element X into the priority queue.

Here, Less the value, higher the priority. Hence, the resultant queue will be in ascending order.

Queue is a one-dimensional array having maximum N elements.

(1) [Check if Overflow]

```
IF R == N THEN
        Write("Queue Overflow")
        Return
END IF
```

(2) [Find the position to insert the Element]

```
I = R
WHILE (Queue [I] > X)
DO
        Queue[ I+1 ] = Queue [I]
        I = I – 1
END DO
```

(3) [Insert the new Element and Increment Rear]

```
Queue [ I + 1 ] = X
R = R + 1
```

(4) [Finish]

```
Exit
```

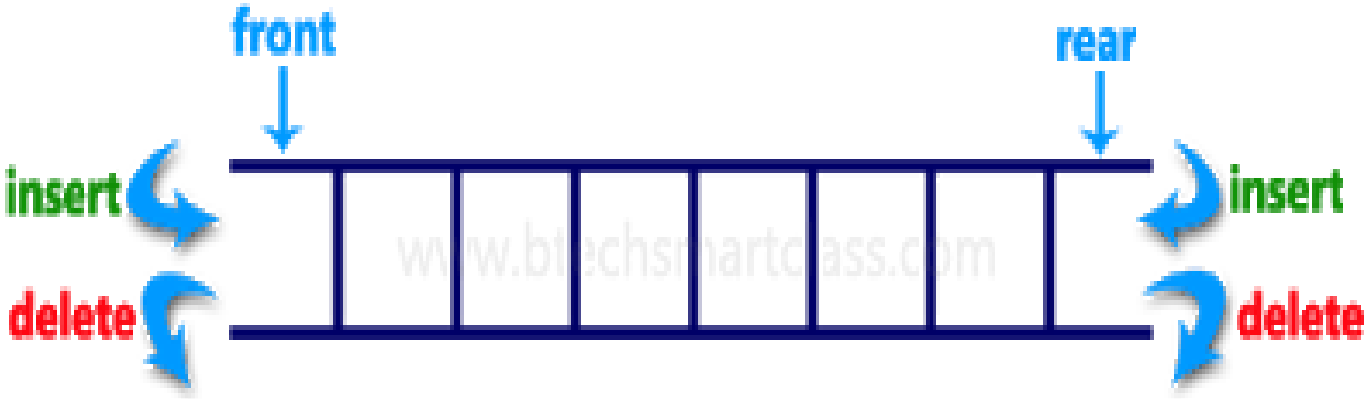
ALGORITHM TO DELETE AN ELEMENT FROM PRIORITY QUEUE

PDnqueue(Queue, F, R, X)

- This procedure removes an element from the priority queue.
- This procedure is SAME as SIMPLE queue, because it removes the FRONT element.
- Here, the Front element is having the Highest Priority.

DOUBLE ENDED QUEUE (DEQUE)

- Double ended queue is a more generalized form of queue data structure which allows insertion and removal of elements from both the ends.
- New items can be added at either the front or the rear. Likewise, existing items can be removed from either end.
- In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure.
- This data structure is hybrid in a way, that it does not employ LIFO or FIFO. It is up to the user, how the DQueue will be used.



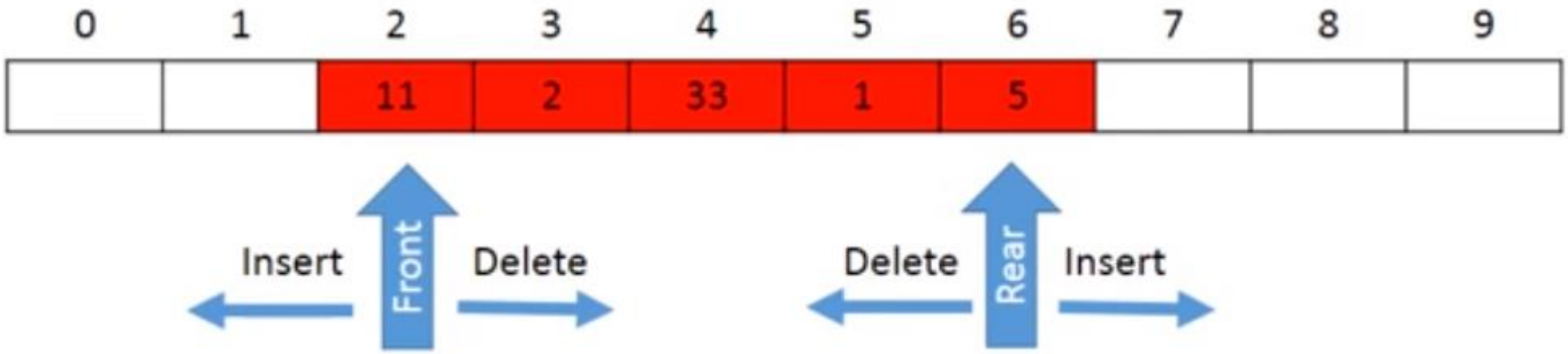
OPERATION ON DEQUE

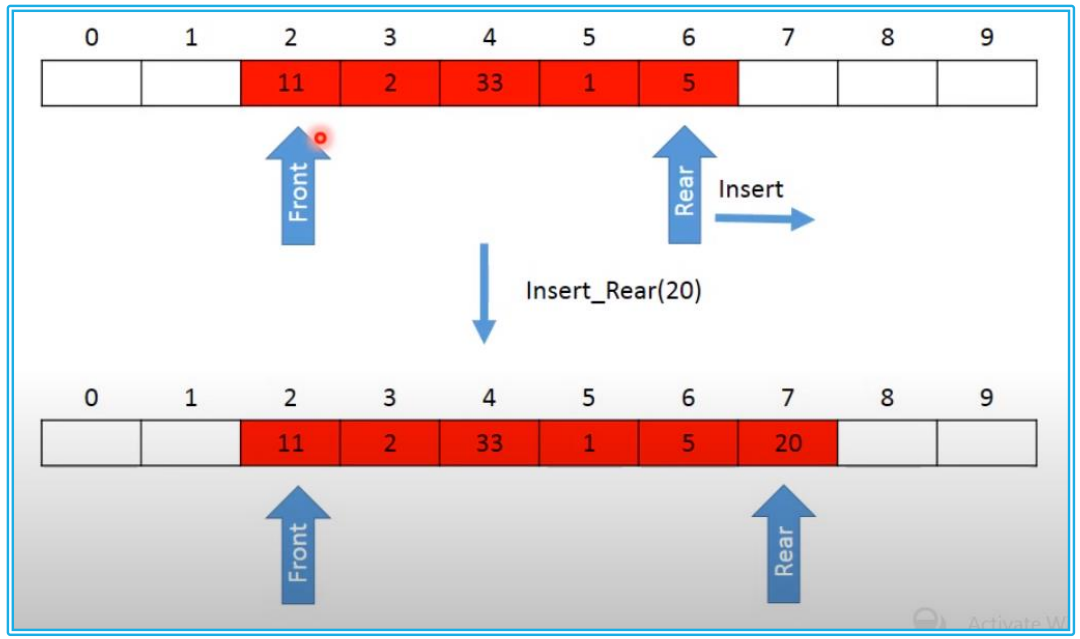
Enqueue_Front: A new operation added to the DeQue which allows inserting a new element at the Front.

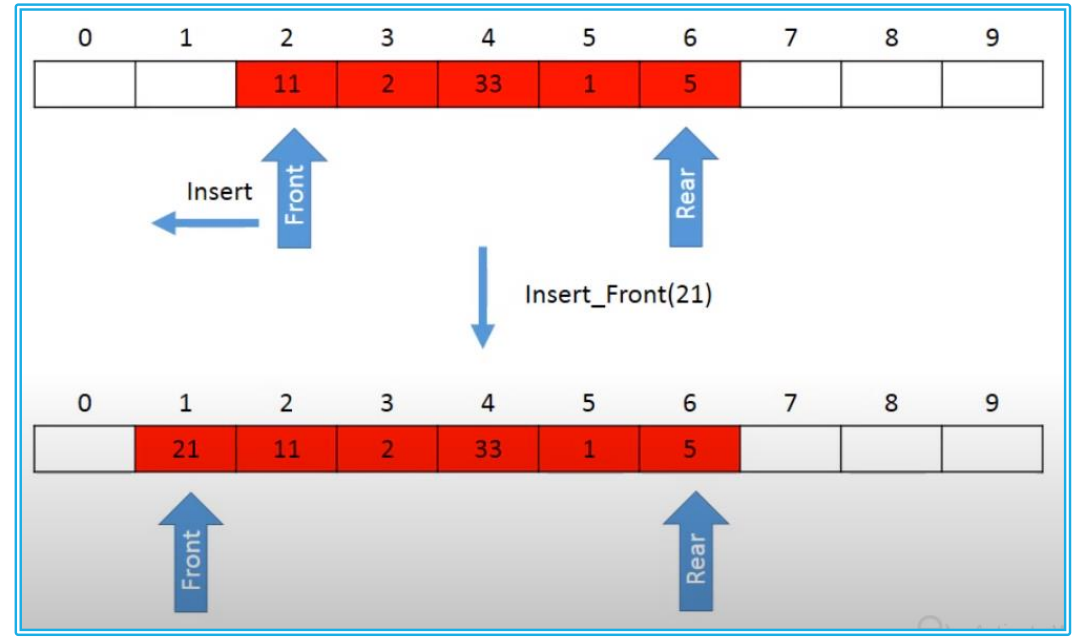
DeQueue_Front: **Same as SIMPLE QUEUE.**

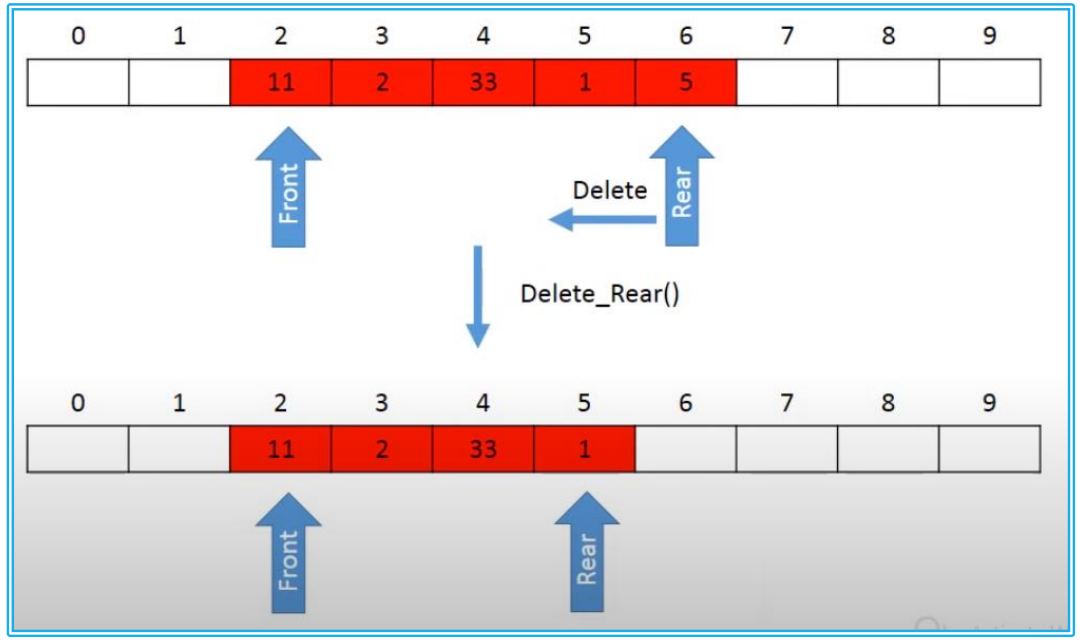
Enqueue_Rear: **Same as SIMPLE QUEUE.**

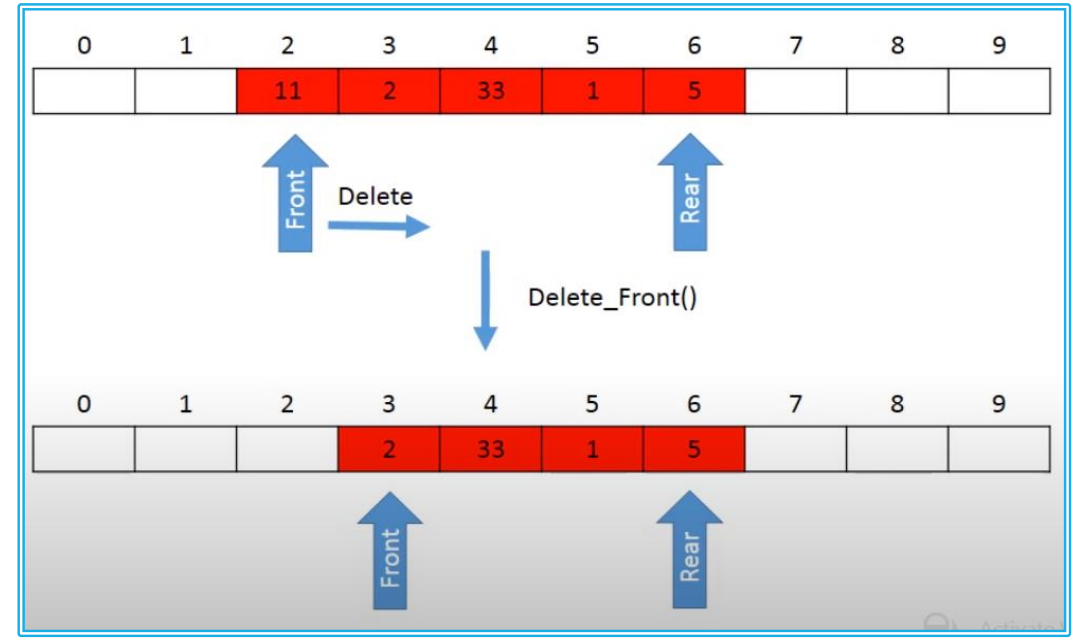
DeQueue_Rear: A new operation added to the DeQue which allows deletion from the Rear.

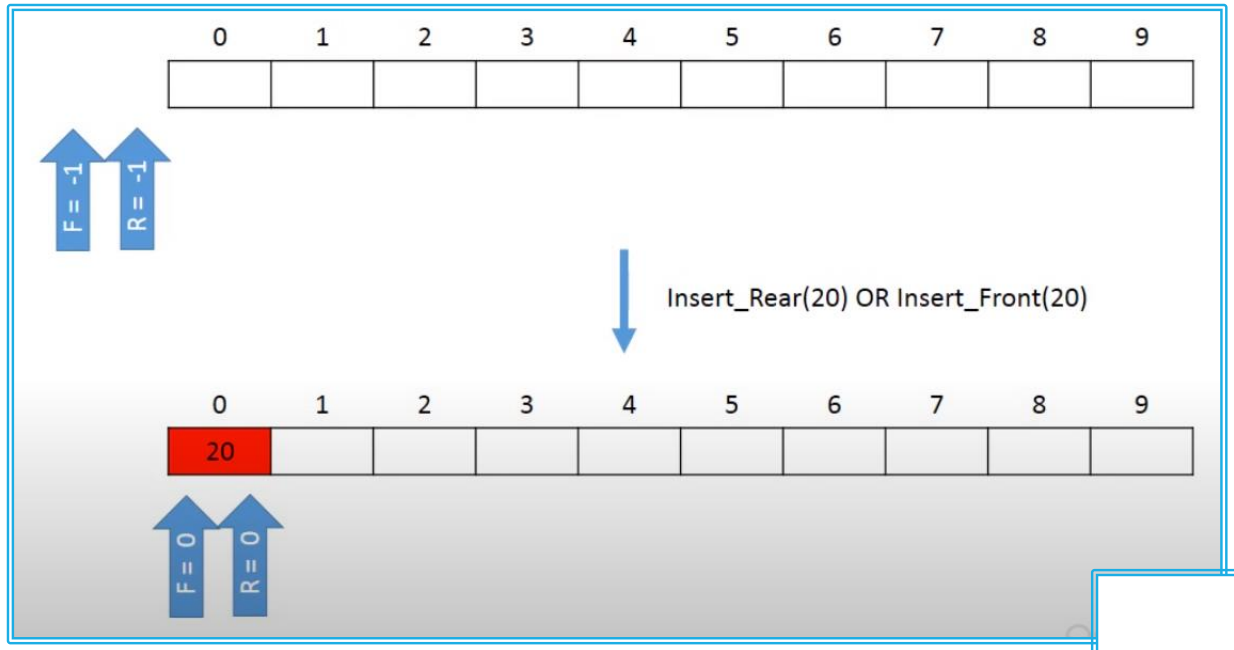


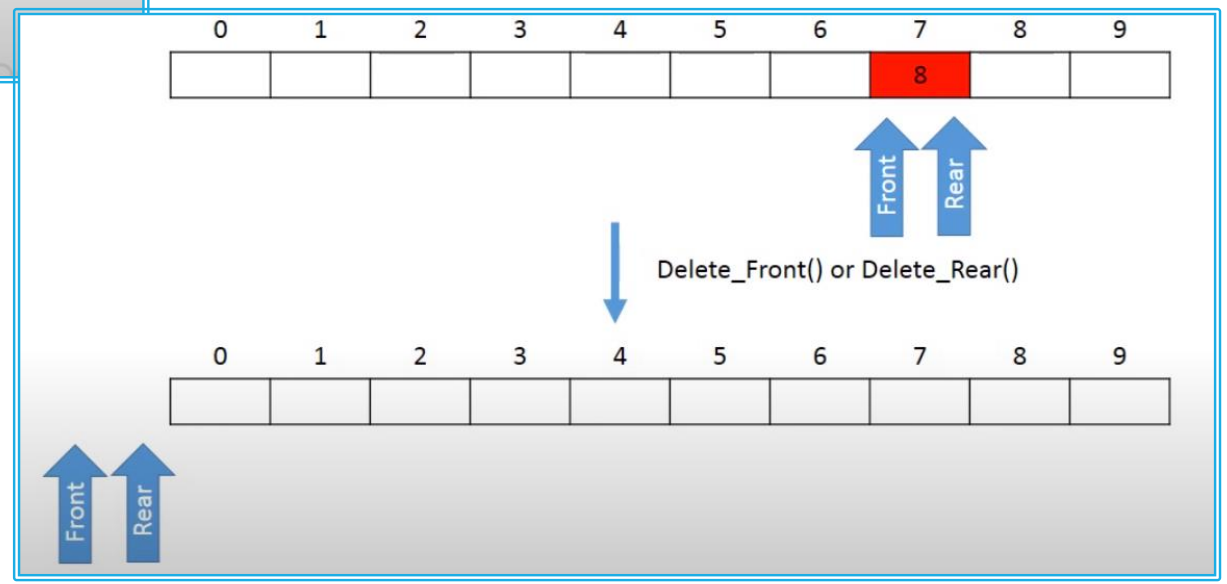

0 1 2 3 4 5 6 7 8 9
11 2 33 1 5
Front
Rear
Insert
Insert_Rear(20)
0 1 2 3 4 5 6 7 8 9
11 2 33 1 5 20
Front
Rear

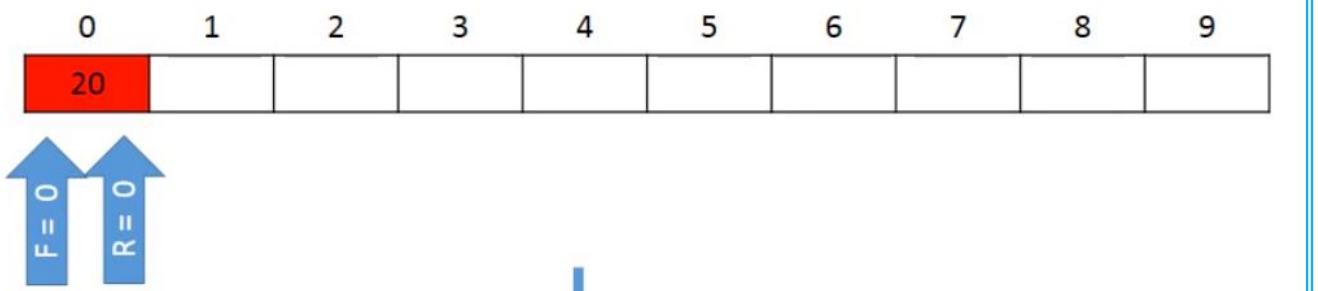

0 1 2 3 4 5 6 7 8 9
11 2 33 1 5
Insert
Front
Rear
Insert_Front(21)
0 1 2 3 4 5 6 7 8 9
21 11 2 33 1 5
Front
Rear


0 1 2 3 4 5 6 7 8 9
11 2 33 1 5
Front
Delete
Rear
Delete_Rear()
0 1 2 3 4 5 6 7 8 9
11 2 33 1
Front
Rear


0 1 2 3 4 5 6 7 8 9
11 2 33 1 5
Front
Delete
Rear
Delete_Front()
0 1 2 3 4 5 6 7 8 9
2 33 1 5
Front
Rear

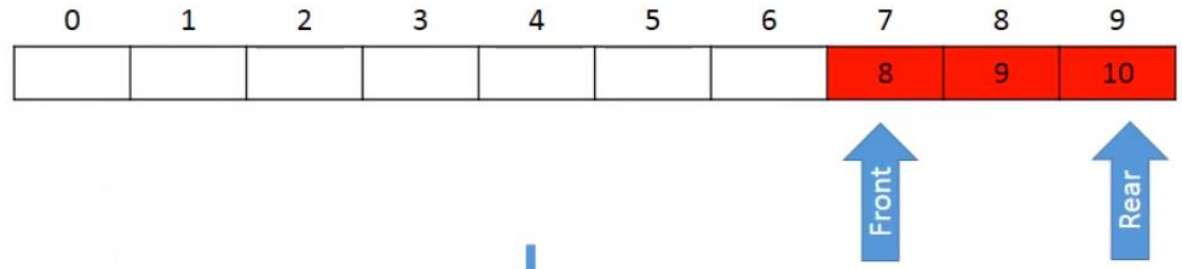

0 1 2 3 4 5 6 7 8 9
F = -1
R = -1
Insert_Rear(20) OR Insert_Front(20)
0 1 2 3 4 5 6 7 8 9
20
F = 0
R = 0


0 1 2 3 4 5 6 7 8 9
8
Front
Rear
Delete_Front() or Delete_Rear()
0 1 2 3 4 5 6 7 8 9
Front
Rear



Insert_Front(6)

Can’t insert at front if we are using simple queue and not circular queue.



Insert_Rear(6)

Can’t insert at Rear if we are using simple queue and not circular queue.

ALGORITHM TO INSERT AN ELEMENT IN DQUEUE AT FRONT

Enqueue_Front(DeQue, F, R, X)

This procedure inserts a new element X , at the Front of the DeQue. Here, Front will decremented by one.

```
(1) [Check if Enqueue possible]
        IF F==1 THEN
                Write("Enqueue at Front not

        possible…")
                Return
        END IF
(2) [Check if DeQue is Empty
Otherwise, Decrement the Front]
        IF F==0 THEN
                F=R=1
        ELSE
                F = F – 1
        END IF
(3) [Insert the Element]
        DeQue[Front] = X
(4) [Finish]
        Return
```

ALGORITHM TO DELETE AN ELEMENT FROM DQUEUE AT REAR

Dequeue_Rear(DeQue, F, R)

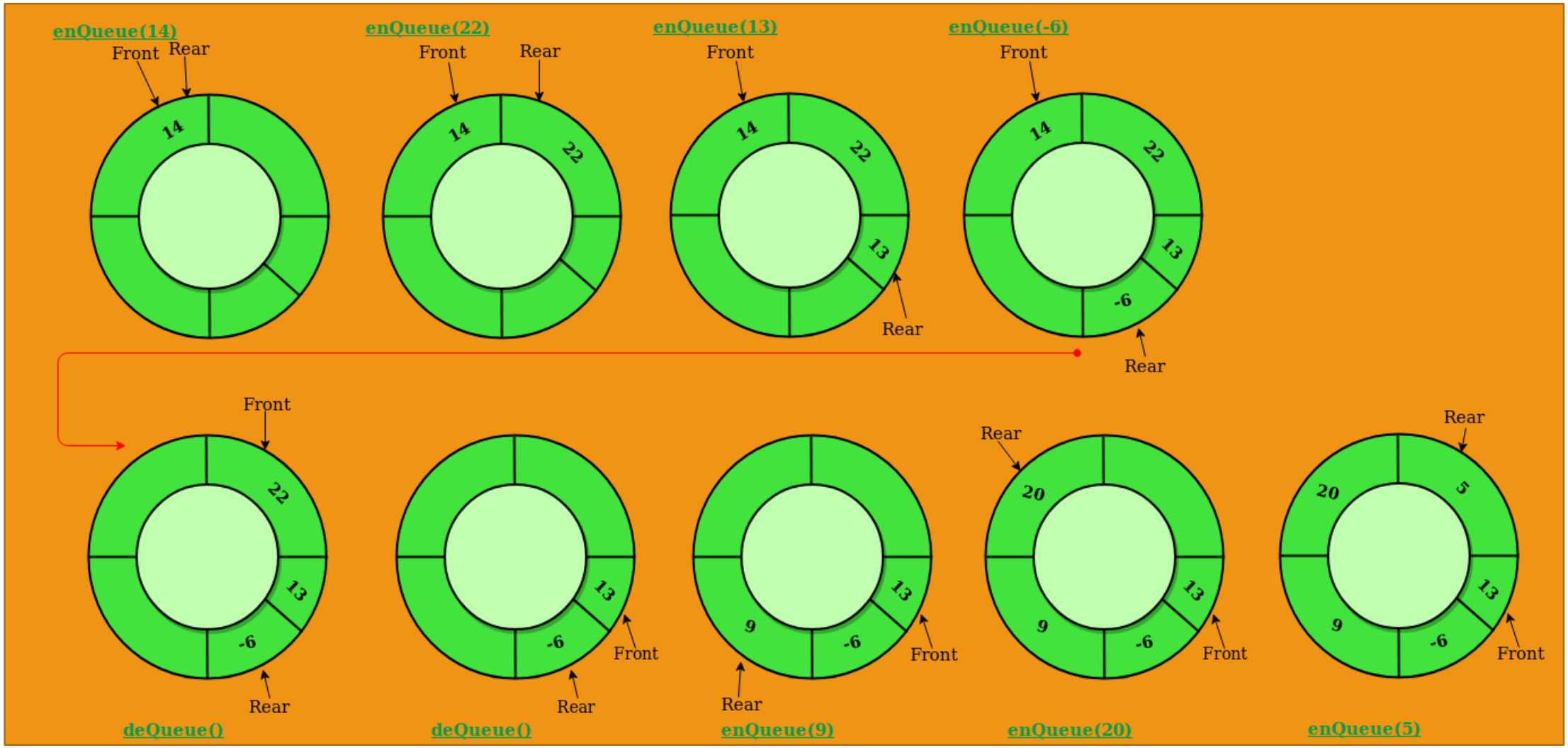
This procedure removes an element from the Rear of the DeQue. Here, Rear will decremented by one.

```
(1) [Check if Underflow]
        IF F==0 THEN
                Write("Deque Underflow...")
                Return(-1)
        END IF
(2) [Copy the Rear Element]
        Temp = DeQue[Rear]
(3) [Check if there is the Last element
Otherwise, Decrement the Rear]
        IF F==R THEN
                F=R=0
        ELSE
                R = R – 1
        END IF
(4) [Finish]
        Return
```

CIRCULAR QUEUE

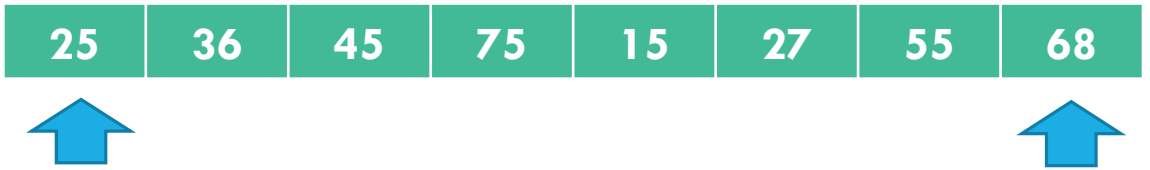
- A circular queue is a queue in which elements are arranged such that the first element in the queue follows the last element in the queue
- Circular queue is a linear data structure. It follows FIFO principle
- Disadvantage of simple queue is that even if we have a free memory space in a queue we can not use that free memory space to insert element.
- In circular queue the last node is connected back to the first node to make a circle.
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as “Ring buffer”.

CIRCULAR QUEUE

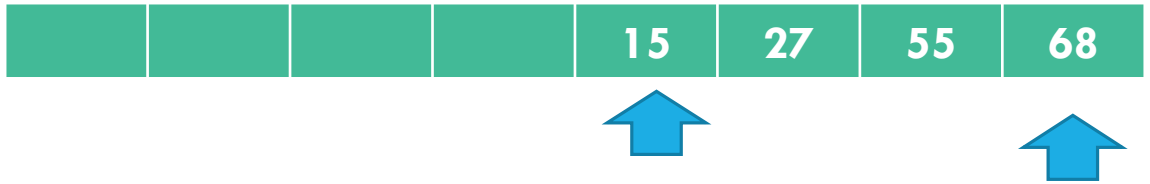


CIRCULAR QUEUE

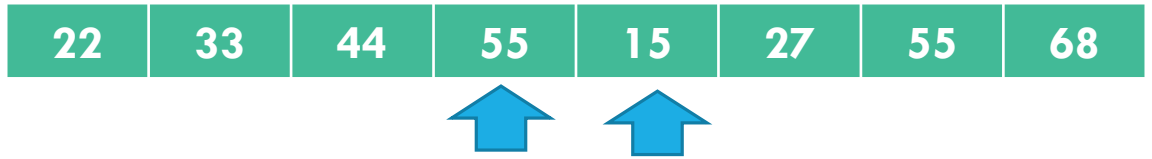
CEnqueue (25)
CEnqueue (36)
CEnqueue (45)
CEnqueue (75)
CEnqueue (15)
CEnqueue (27)
CEnqueue (55)
CEnqueue (68)
CDequeue()
CDequeue()
CDequeue()
Cenqueue(22)
Cenqueue(33)
Cenqueue(44)
Cenqueue(55)



F=1, R = 8



F=5, R = 8



CEnqueue (66)
CQueue Overflow

F=5, R = 4

Cdequeue()

F=6, R = 4

ALGORITHM TO INSERT AN ELEMENT IN CIRCULAR QUEUE

CEnqueue (Cqueue, F, R, N, VAL)

- Queue is represented by vector Cqueue which contains N elements.
- F and R are pointers, incremented in circular way.
- VAL is the element to be inserted.

(1) [Check for Overflow]

```
IF (F == 1 and R == N) OR (R == F – 1) THEN
        Write("Queue Overflow")
        Return
END IF
```

(2) [Increment Rear in Circular Way]

```
IF (R == N) THEN
        R = 1
ELSE
        R = R + 1
END IF
```

(3) [Reset Queue if First time Enqueue]

```
IF (F ==0) THEN
        F = 1
END IF
```

(4) [Insert the Element]

```
Cqueue[R] = VAL
```

(5) [Finish]

```
Exit
```

ALGORITHM TO DELETE AN ELEMENT FROM CIRCULAR QUEUE

CDqueue (Cqueue, F, R)

- Queue is represented by vector Cqueue which contains N elements.
- F and R are pointers, incremented in circular way.

(1) [Check Underflow]

```
IF (F == 0) THEN
        Write("Queue Underflow")
        Return
END IF
```

(2)[Copy the front element]

```
Temp= Cqueue[F]
```

(3) [Reset Queue if last element]

```
IF (F == R) THEN
        F = R = 0
END IF
```

(4) [Increment the Front in Circular Way]

```
IF F == N THEN
        F = 1
ELSE
        F = F + 1
```

(5) [Return the Copied Element]

```
Return (Temp)
```

(6) [Finish]

```
Exit
```

ALGORITHM TO DISPLAY ELEMENTS FROM CIRCULAR QUEUE

Cqueue_Display(Cqueue,F,R): This procedure traverses the circular queue and displays all its elements Queue is represented by vector Cqueue which contains N elements. F and R are pointers.

(1) [Check Queue Underflow]

IF F==0 THEN

Write ("Queue Underflow")

END IF

(2) [Traverse the Queue and display the Elements]

IF F < R THEN

For Count = F To R

Do

print Cqueue[Count];

End Do

Else

For Count=F To N (Size)

Do

print Cqueue[Count]

End Do

For Count=1 To R

Do

print Cqueue[Count]

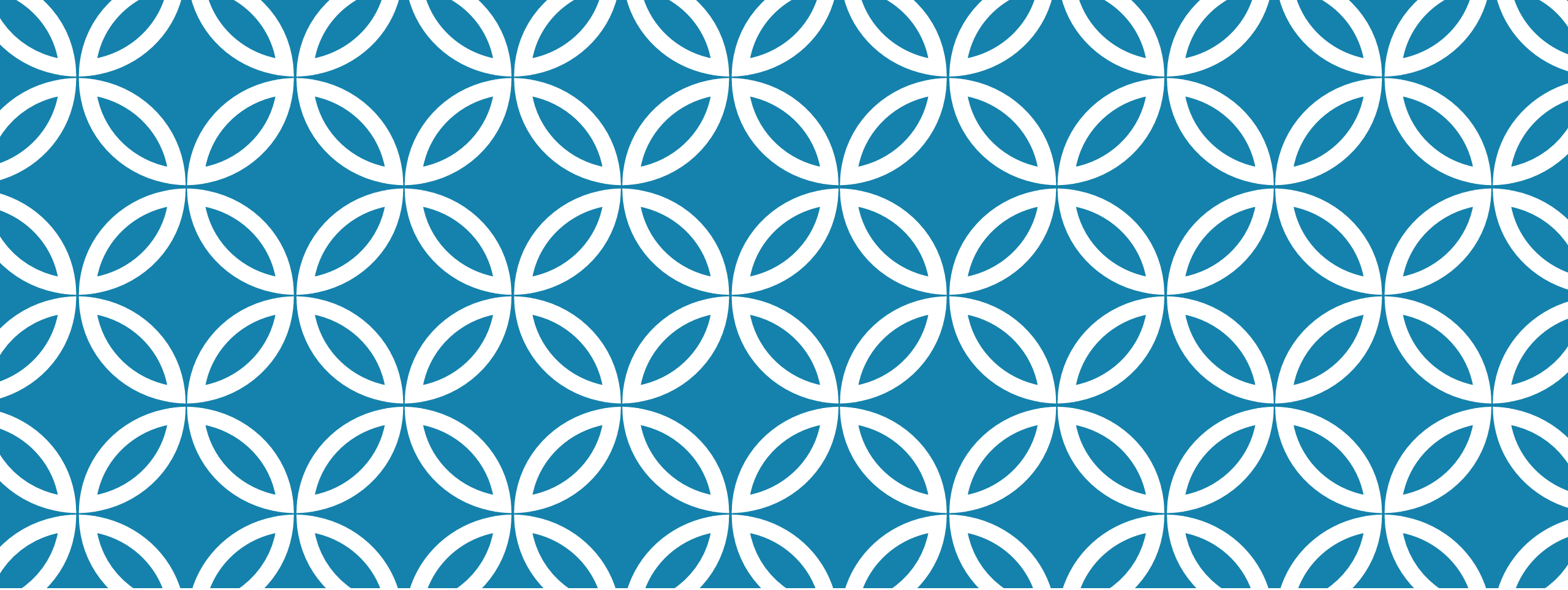
End Do.

(6) [Finish]

Exit

ASSIGNMENT:

1. Difference between Stack and queue.
2. What is Queue? Explain disadvantages of simple queue.
3. Explain Queue fundamentals with Queue Insertion & deletion algorithms.
4. What is mean by priority queue. Write algorithms of Pqueue.
5. What is double ended queue? Explain different operations of Dqueue by algoritham.
6. What is Circular queue? Compare circular queue with normal queue.
7. Write and explain algorithm to insert & delete element in circular queue.
8. List application of queue.



DATA STRUCTURE AND ALGORITHM

UNIT -2

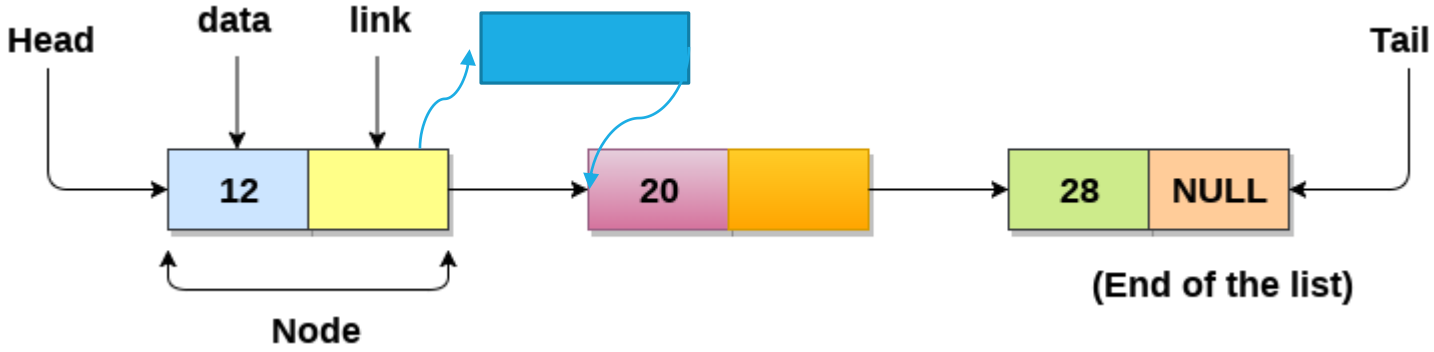
Linear Data Structures –Link List

Prepared by: Prof. Kinjal Patel

LINKED LIST

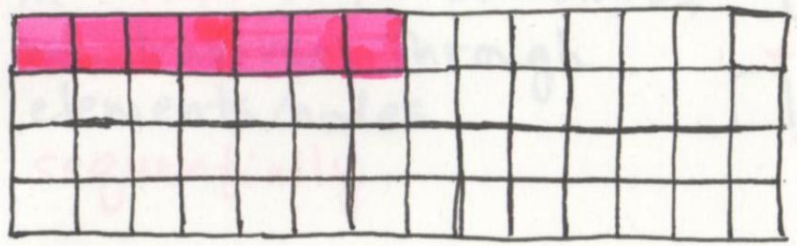
- A **linked list** is a linear collection of data elements, called **nodes,** where the linear order is given by means of **pointers**.Each **node** is divided into two parts:

- A linked list is a non-sequential collection of data items.
- The concept of a linked list is very simple, for every data item in the linked list, there is an associated pointer that would give the memory allocation of the next data item in the linked list.
- The data items in the linked list are not in a consecutive memory locations but they may be anywhere in memory.



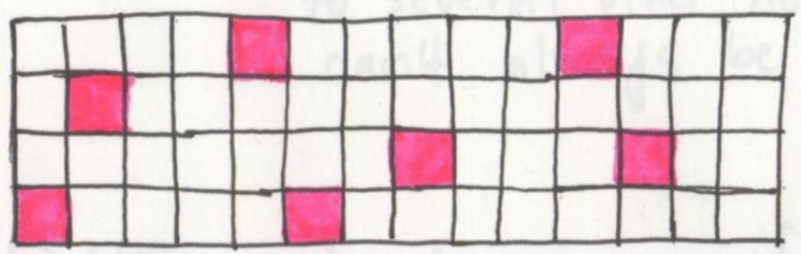
Memory Allocation

STATIC



Arrays need a contiguous block of memory.

DYNAMIC



Linked lists don't need to be contiguous in memory; they can grow dyndmically.

USES OF LINKED LIST

The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.

list size is limited to the memory size and doesn't need to be declared in advance.

Empty node can not be present in the linked list.

We can store values of primitive types or objects in the singly linked list.

WHY USE LINKED LIST OVER ARRAY?

Array contains following limitations:

- The size of array must be known in advance before using it in the program.
- Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
- All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array.

- Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
- Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
- Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
- Many complex applications can be easily carried out with linked lists.

DISADVANTAGES OF LINKED LISTS

- It consumes more space because every node requires a additional pointer to store address of the next node.
- It cannot be easily sorted
- More complex to create than an array
- Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
- Searching a particular element in list is difficult and also time consuming.

Types of Linked Lists:

- Single Linked List.
- Double Linked List.
- Circular Linked List.
- Circular Double Linked List.

COMPARISON BETWEEN ARRAY AND LINKED LIST

ARRAY	LINKED LIST
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

APPLICATION OF LINKED LIST

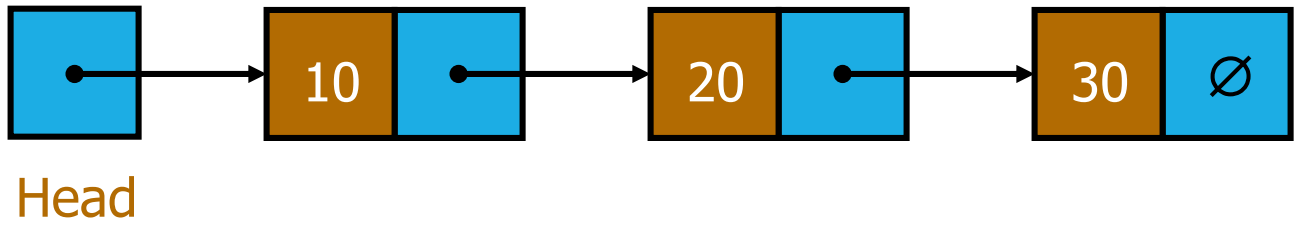
- Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non zero coefficient and exponents. For example:

$$P(x) = a_0 X^n + a_1 X^{n-1} + \ldots\ldots + a_{n-1} X + a_n$$

- Represent very large numbers and operations of the large number such as addition, multiplication and division.
- Linked lists are to implement stack, queue, trees and graphs.
- Implement the symbol table in compiler construction.

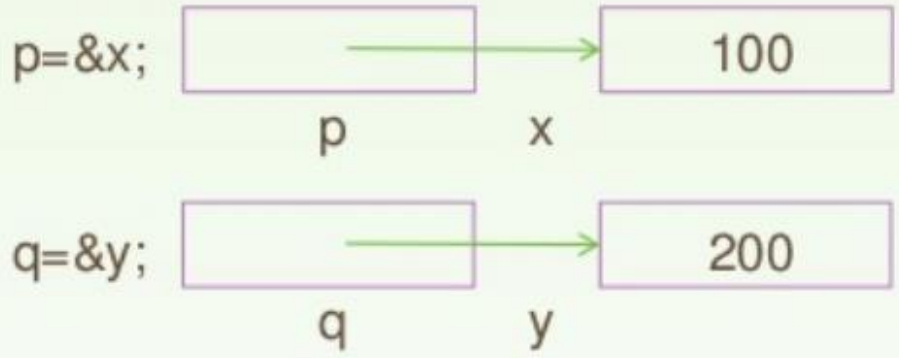
OPERATIONS OF LINKED LIST

- Insertion at beginning
- Insertion at end
- Insertion at after specifies value
- Deletion at beginning
- Deletion at end
- Deletion at after specifies value
- Traversing
- Searching
- Sorting
- Counting
- Reversing



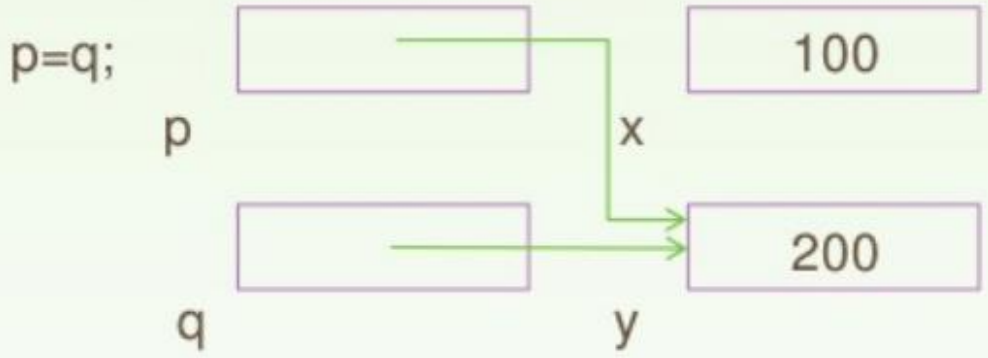
We need basic of pointer

Initialization:



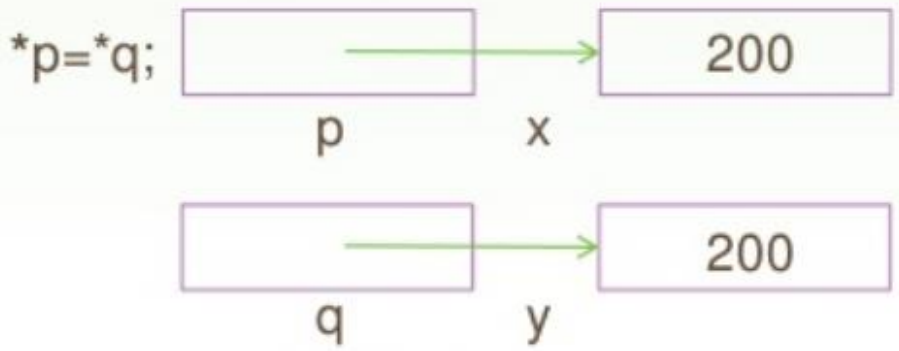
***p=100 and *q=200 and p< >q**

Assignment p=q



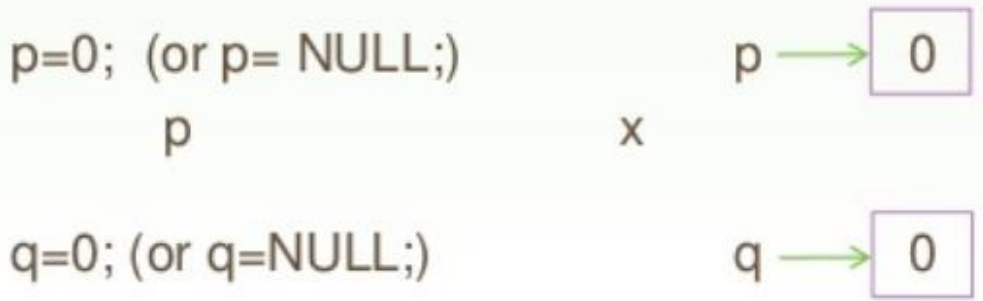
***p=*q=200 but x< >y**

Assignment *p=*q



x=y=200 but p< >q

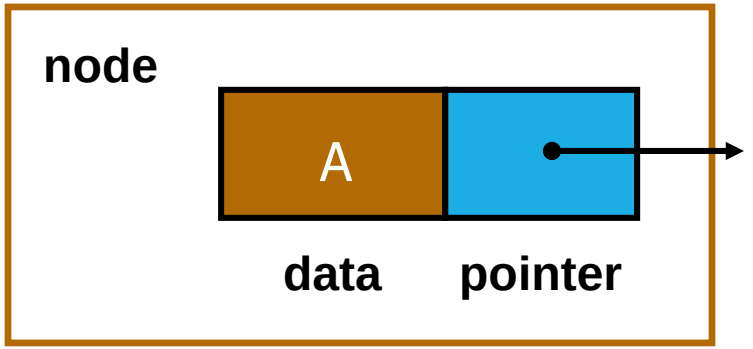
NULL pointers



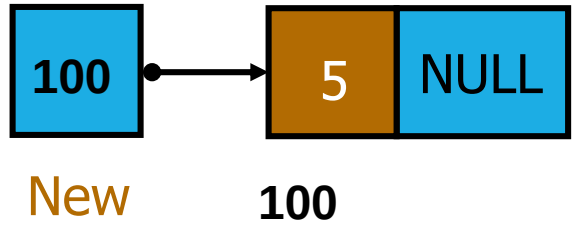
p and q points nothing

NODE STRUCTURE

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




```
struct node *New;
New=(struct node*) malloc(sizeof(struct node));
printf("\nENTER THE DATA: ");
scanf("%d",&New->data);
New->next=NULL;
```



TRAVERSE(DISPLAY) LL

Traverse(head) :This function display all node of linked list

(1) [Check LL is empty]

IF (head == NULL)

print “LL is empty”

Return

(2) [Display all node]

ptr=head

while (ptr != NULL)

DO

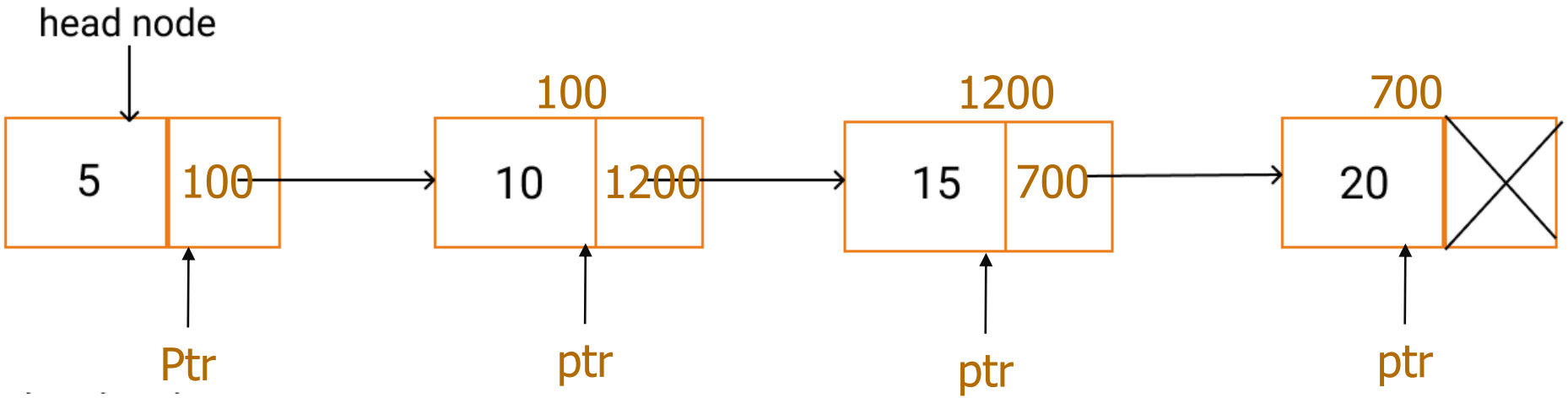
print ptr->data

ptr=ptr->next

END DO

(3) [Finish]

Exit


head node
5
100
100
10
1200
1200
15
700
700
20
Ptr
ptr
ptr
ptr

INSERT AT BEGINNING

Insert_first(head,val) :This function inserts a new node at the first position of linked list.

(1) [Create a new node with data]

ptr= new Node

ptr-> data =val

ptr-> next = NULL

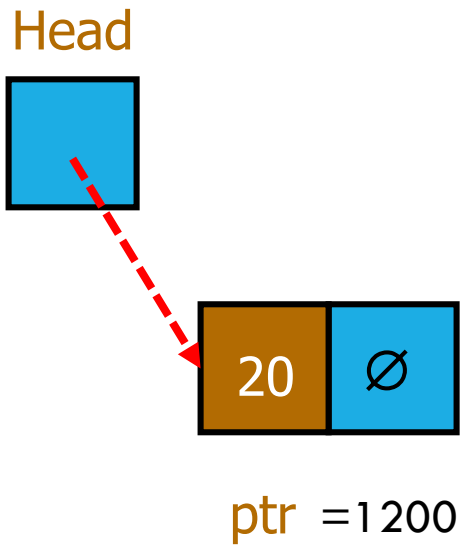
(2) [Insert at First]

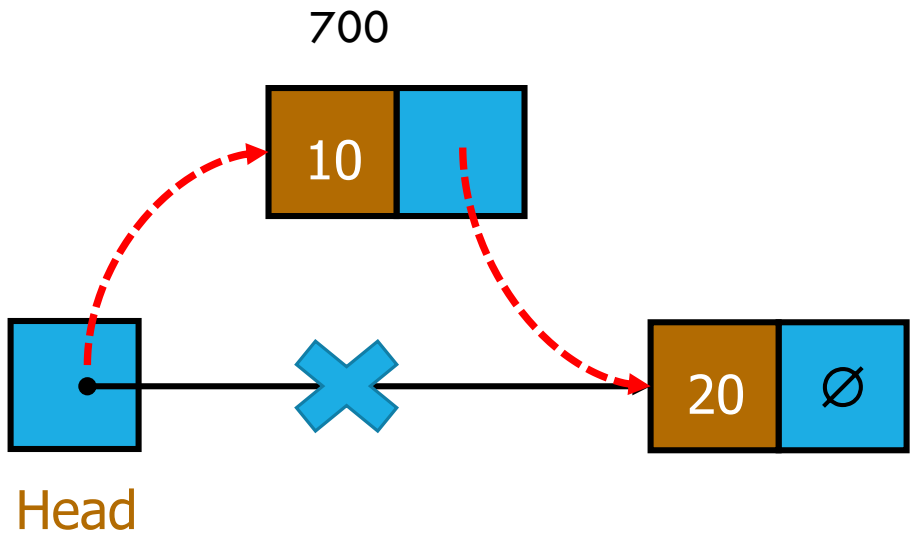
ptr->next=head

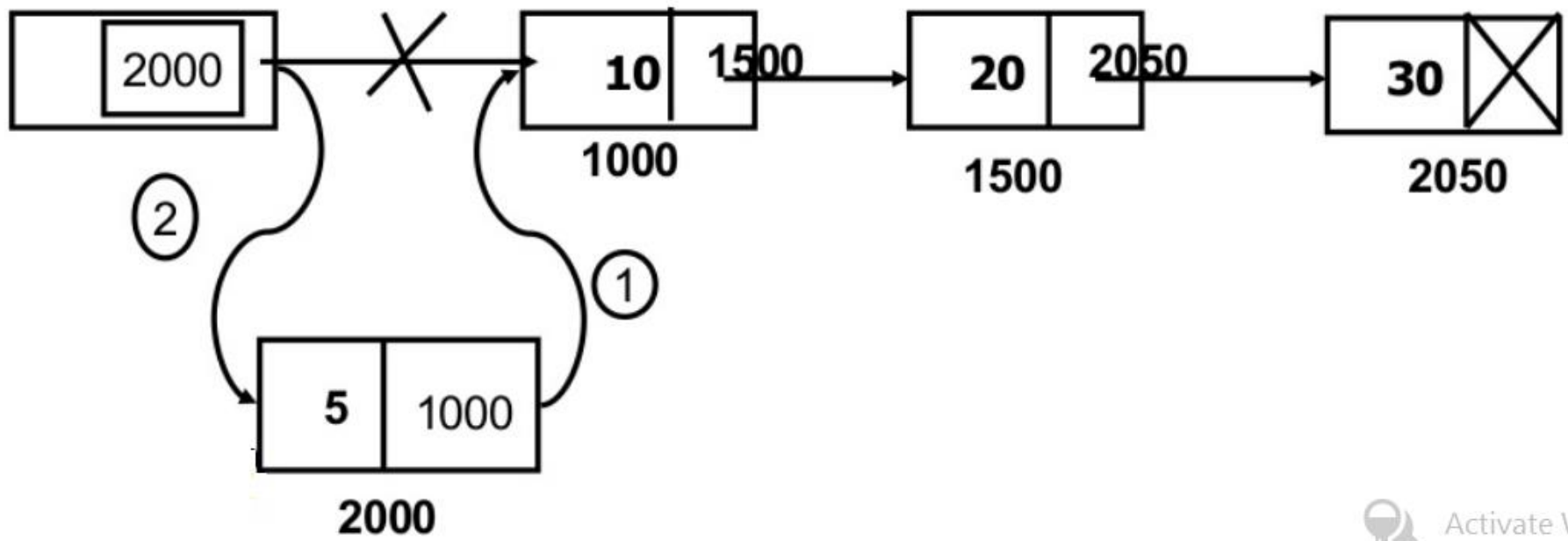
head= ptr

(3) [Finish]

Exit







INSERT AT END

Insert_end(head,val) :This function inserts a new node at the end of linked list.

(1) [Create a new node with data]

ptr= new Node

ptr-> data = val

ptr-> next = NULL

(2) [Insert at First if head is NULL]

IF (head==NULL) THEN

head= ptr

Return

(3) [Inset at last if head is NOT NULL]

IF (head != NULL) THEN

Temp = head

while (temp -> next != NULL)

DO

temp = temp -> next;

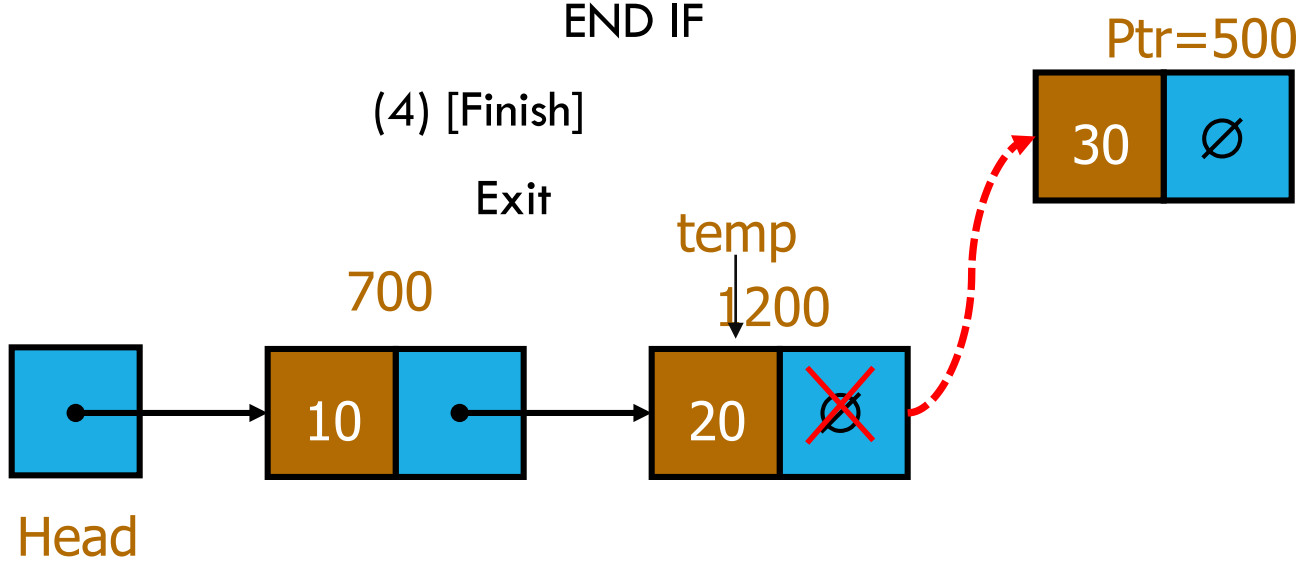
END DO

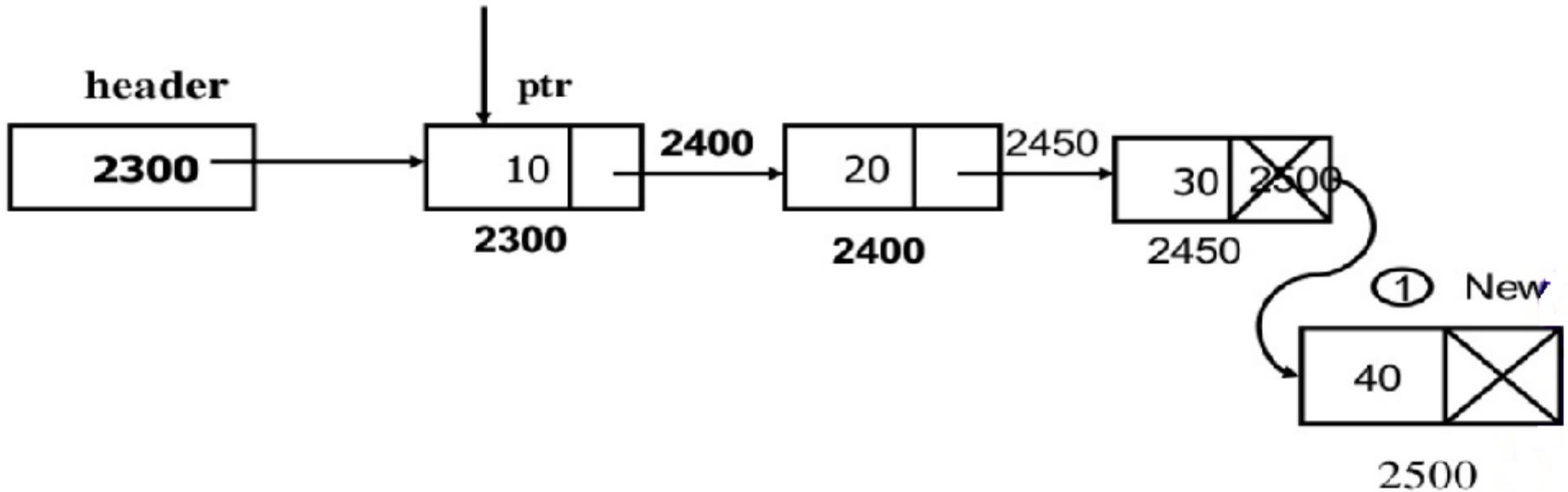
temp->next = ptr;

END IF

(4) [Finish]

Exit




header
ptr
2300
10
2400
20
2450
30
2500
2300
2400
2450
1
New
40
2500

INSERTION AT AFTER SPECIFIC KEY VALUE

Insert_after(head,key, val) :This function inserts a new node after key node in linked list.

(1) [Search for key]

temp = head

while(temp->data!= key && temp!= NULL)

DO

temp = temp->next

END DO

(2) [Check key found or not]

IF (temp == NULL)

Print “Key not found”

Return

(3) [Create a new node]

ptr = new Node

ptr-> data = val

ptr-> next = NULL

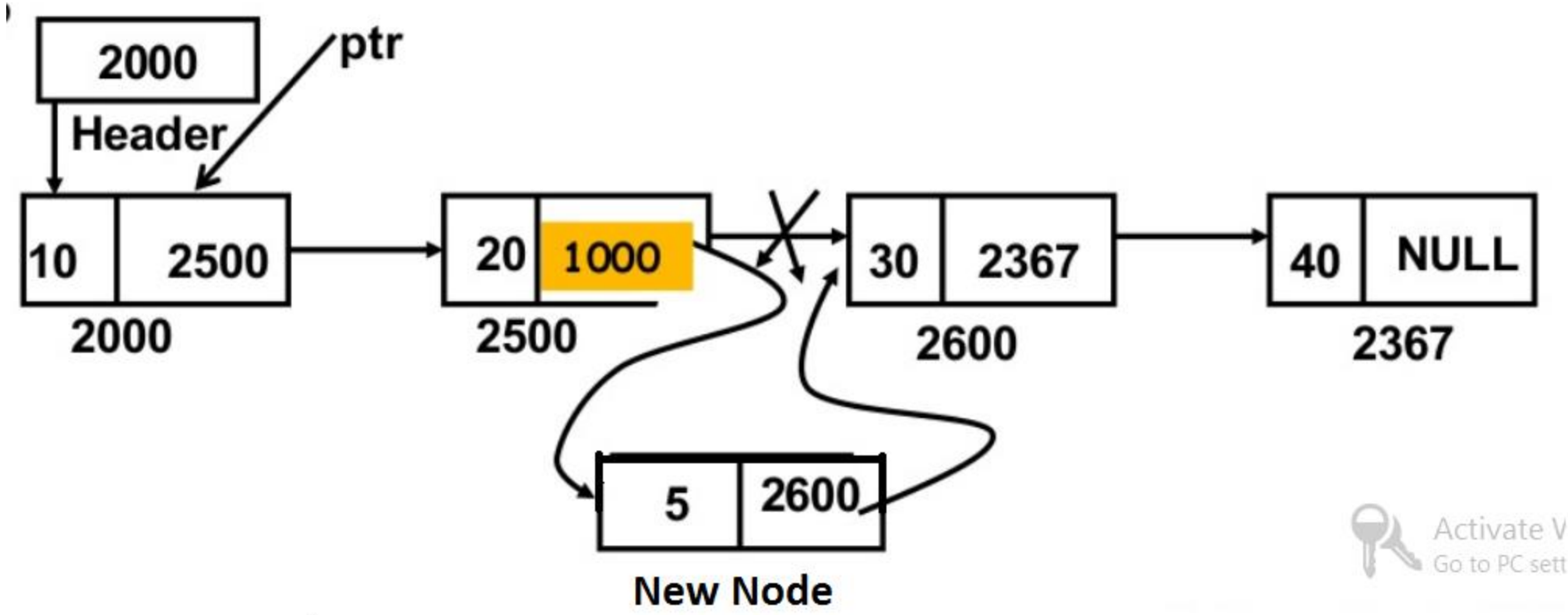
(4) [Insert the node after key node]

ptr->next=temp->next

temp->next=ptr

(5) [Finish]

Exit


2000
ptr
Header
10
2500
2000
20
1000
2500
30
2367
2600
40
NULL
2367
5
2600
New Node

DELETION AT BEGINNING

delete_first(head) :This function delete a node at the first position of linked list.

(1) [Check LL is empty]

IF (head == NULL)

print “LL is empty”

Return

(2) [delete first node]

ptr=head

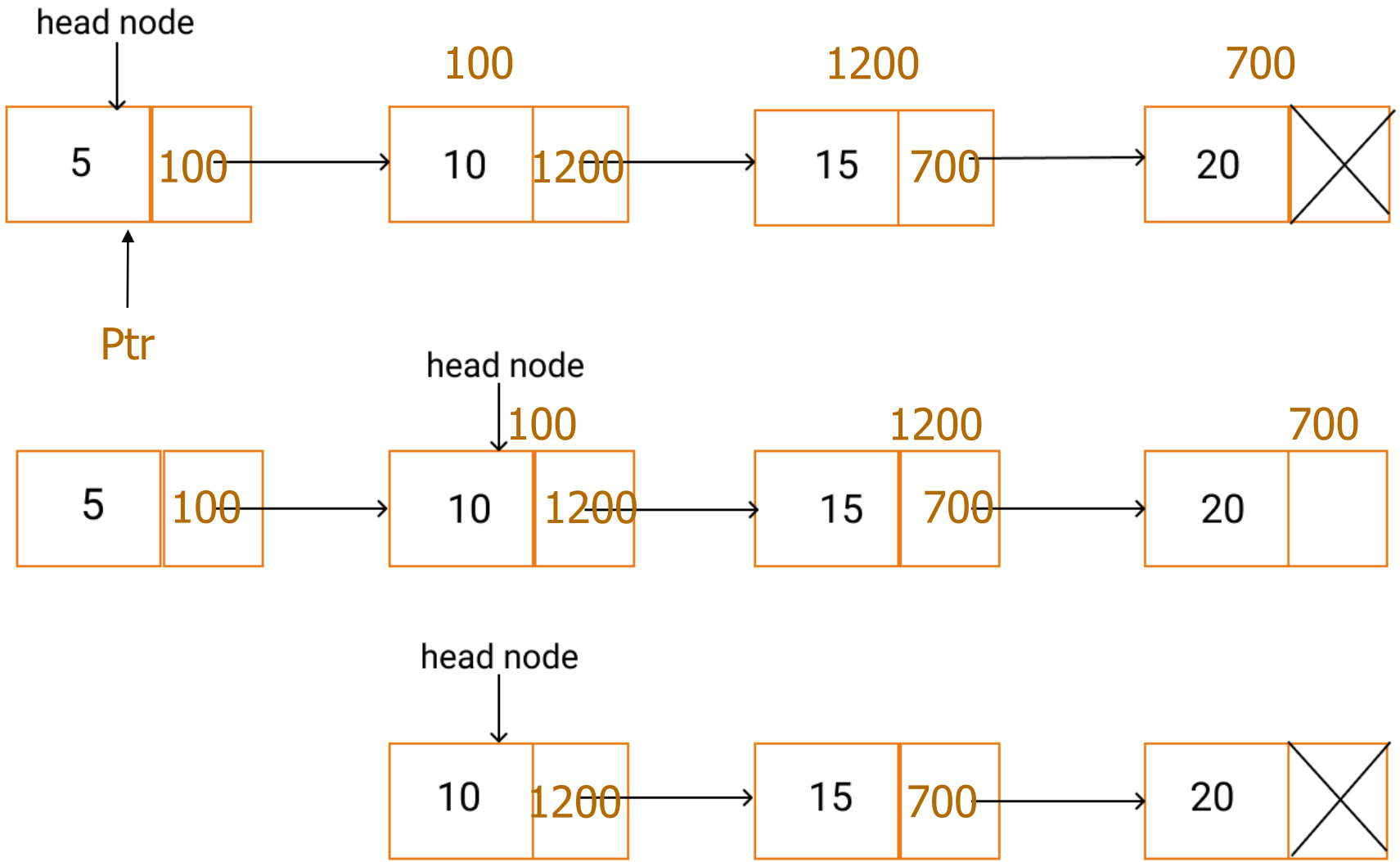
head= head->next

print “ptr->data deleted”

free node ptr

(3) [Finish]

Exit


head node
100
1200
700
5
100
10
1200
15
700
20
Ptr
head node
100
1200
700
5
100
10
1200
15
700
20
head node
10
1200
15
700
20

DELETION AT END

delete_end(head) :This function delete a node from end position of linked list.

(1) [Check LL is empty]

```
IF (head == NULL)
        print “LL is empty”
 Return
```

(2) [Delete node if LL contain only one node]

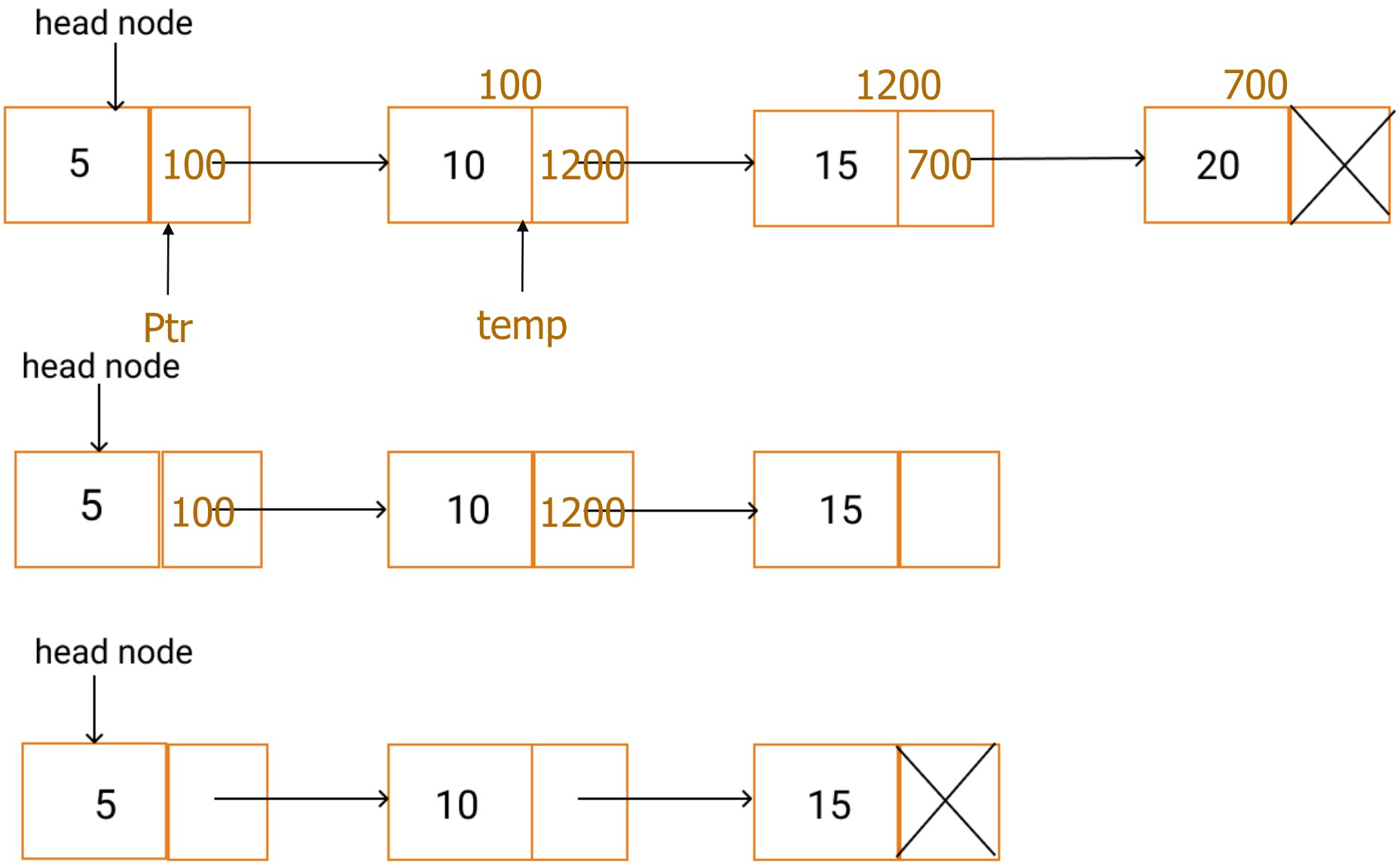
```
IF (head->next == NULL) THEN
    ptr=head
    head = NULL
    print “Node deleted and LL empty”
    free node ptr
END IF
Return
```

(3) [Delete node if more than one element]

```
ptr = head
while (ptr -> next->next != NULL)
   DO
        temp=ptr->next
        ptr = ptr ->next
   END DO
   ptr->next = NULL
   temp=temp->next
   free(temp)
   print “Node deleted”
```

(4) [Finish]

```
Exit
```


head node
5
100
100
10
1200
1200
15
700
700
20
Ptr
temp
head node
5
100
10
1200
15
head node
5
10
15

DELETION OF SPECIFIC KEY

delete_key(head,key) :This function delete a node from linked list by specific value

(1) [Check LL is empty]

```
IF (head == NULL)
        print “LL is empty”
Return
```

(2) [key in first node]

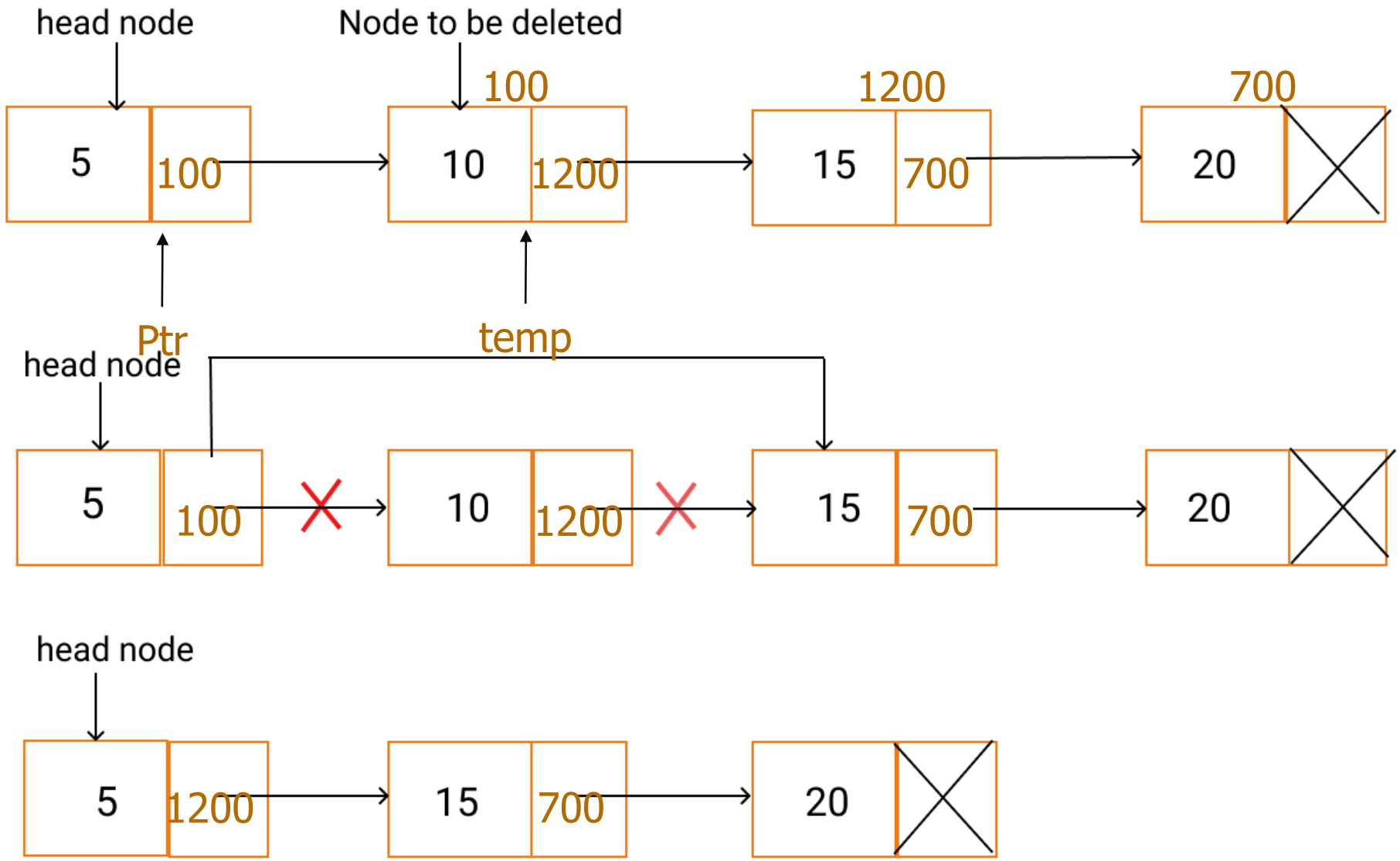
```
IF (head->data==key) THEN
        head=head->next;
END IF
Return
```

(3) [Delete node having KEY value]

```
ptr=ptr1=head
while (ptr->data!=key && ptr!=NULL)
    DO
        ptr1=ptr
        ptr=ptr->next
    END DO
    IF (ptr==NULL)
        print “KEY not found “
    ELSE
        ptr1->next=ptr->next
        free(ptr)
    END IF
```

(4) [Finish]

```
Exit
```


head node
Node to be deleted
100
1200
700
5
100
10
1200
15
700
20
Ptr
temp
head node
5
100
10
1200
15
700
20
head node
5
1200
15
700
20

SEARCH A KEY VALUE

search(head,key) :This function search a key from linked list by position

(1) [Check LL is empty]

```
IF (head == NULL)
        print “LL is empty”
END IF
Return
```

(2) [Travers the LL and Search key node]

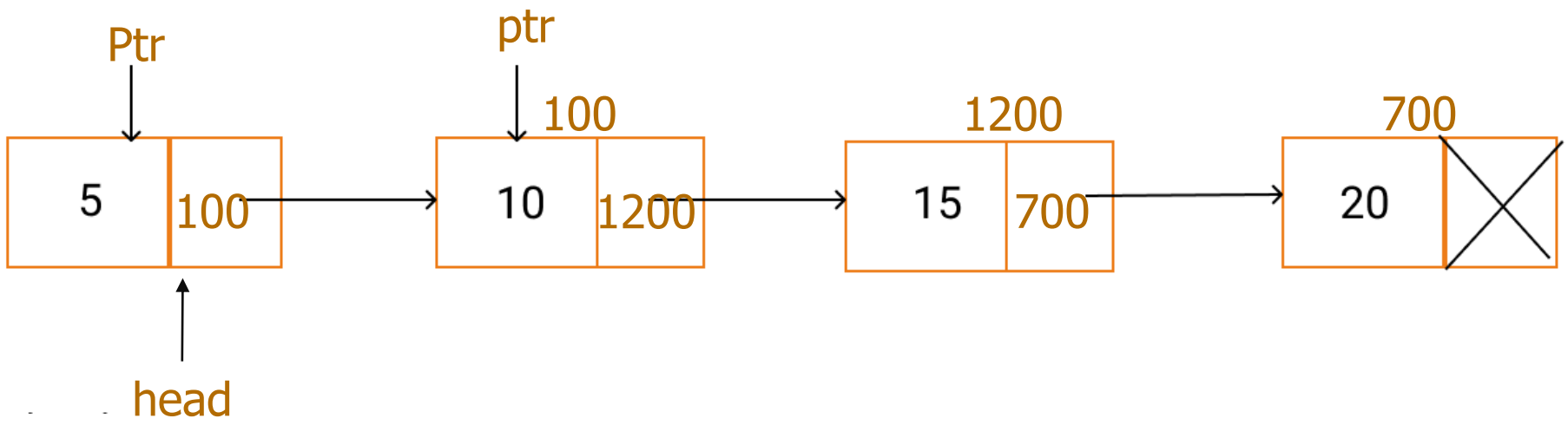
```
ptr = head
while (ptr->data!=key && ptr!=NULL)
    DO
        ptr=ptr->next
    END DO
```

(3) [check key found or not]

```
IF (ptr==NULL)
    print “KEY not found “
ELSE
    print “KEY found “
END IF
```

(4) [Finish]

```
Exit
```


Ptr
ptr
100
1200
700
5
100
10
1200
15
700
20
head

REVERSE LL

Reverse(head) :This function reverse linked list

(1) [Check LL is empty]

IF (head == NULL)

print “LL is empty”

END IF

Return

(2) [Travers the LL and change links]

current = head

prev = next = NULL

while (current != NULL)

DO

next = current->next

current ->next = prev

prev = current

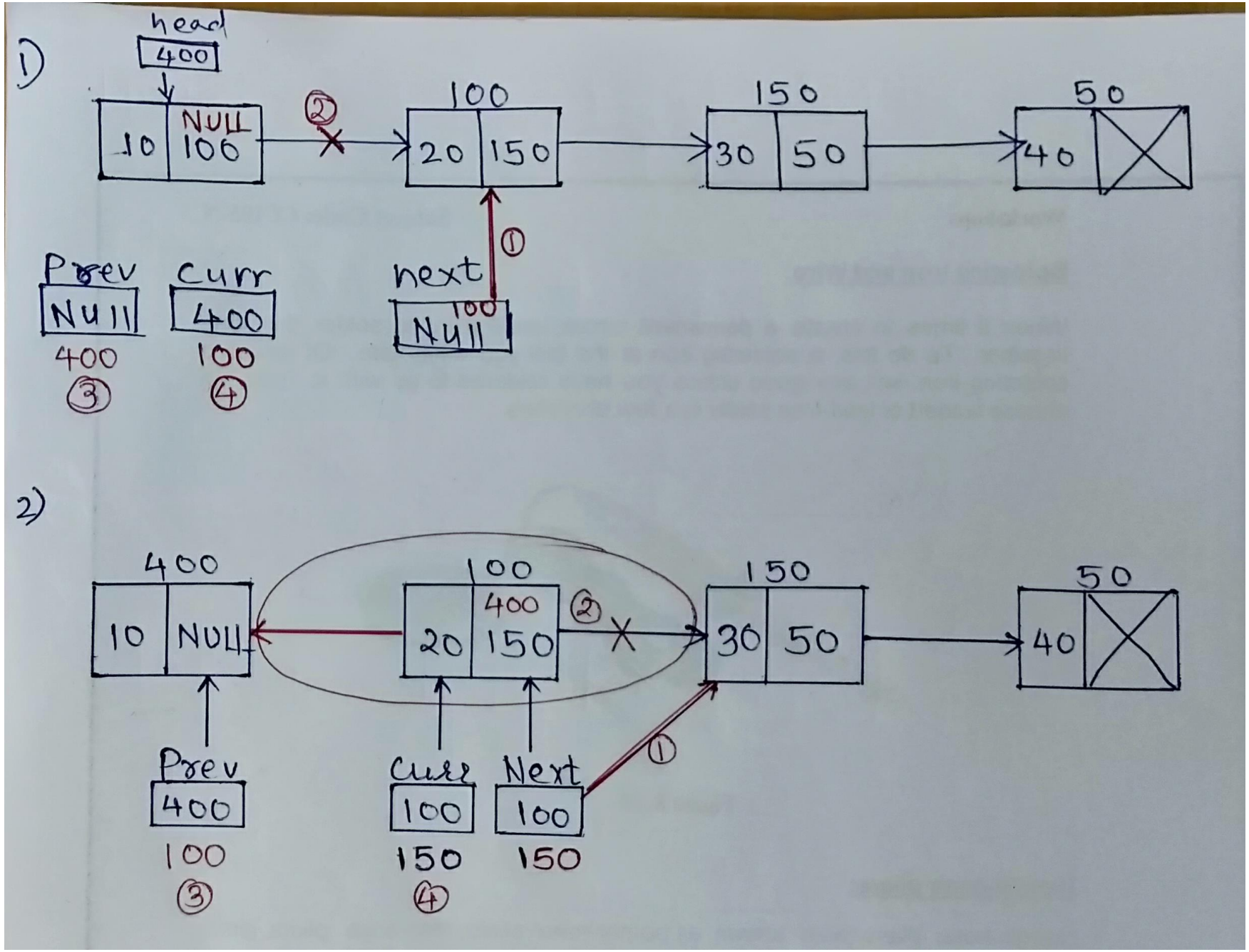
current = next

END DO

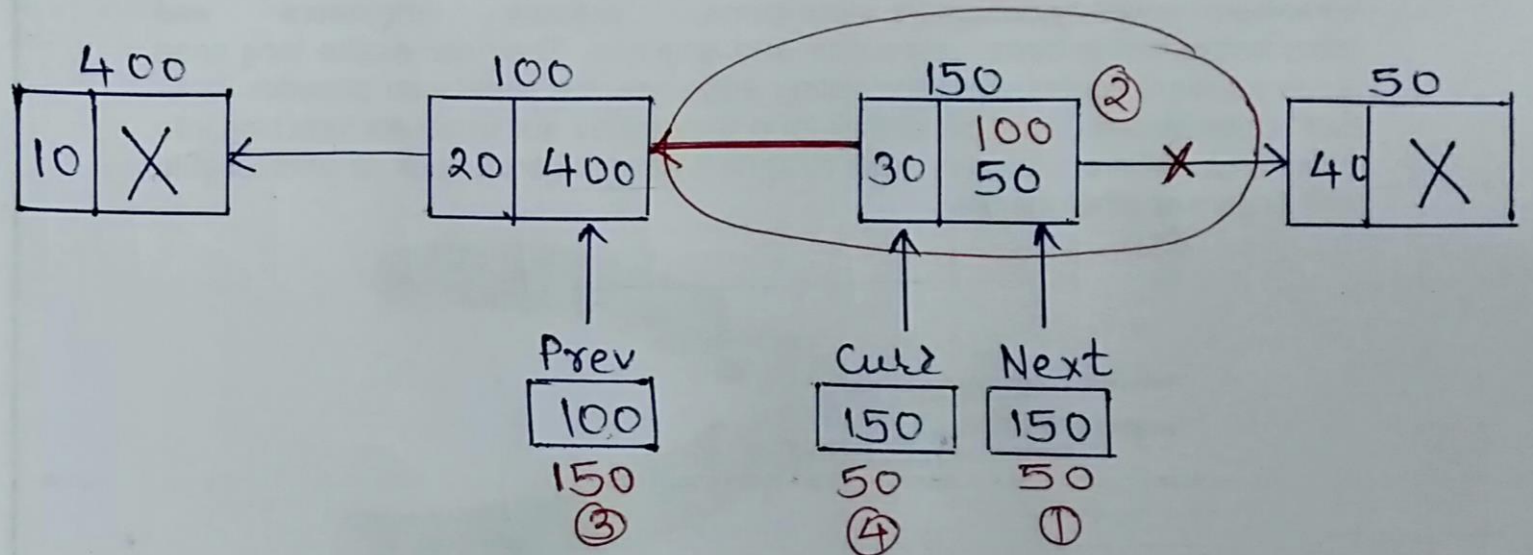
head=prev

(3) [Finish]

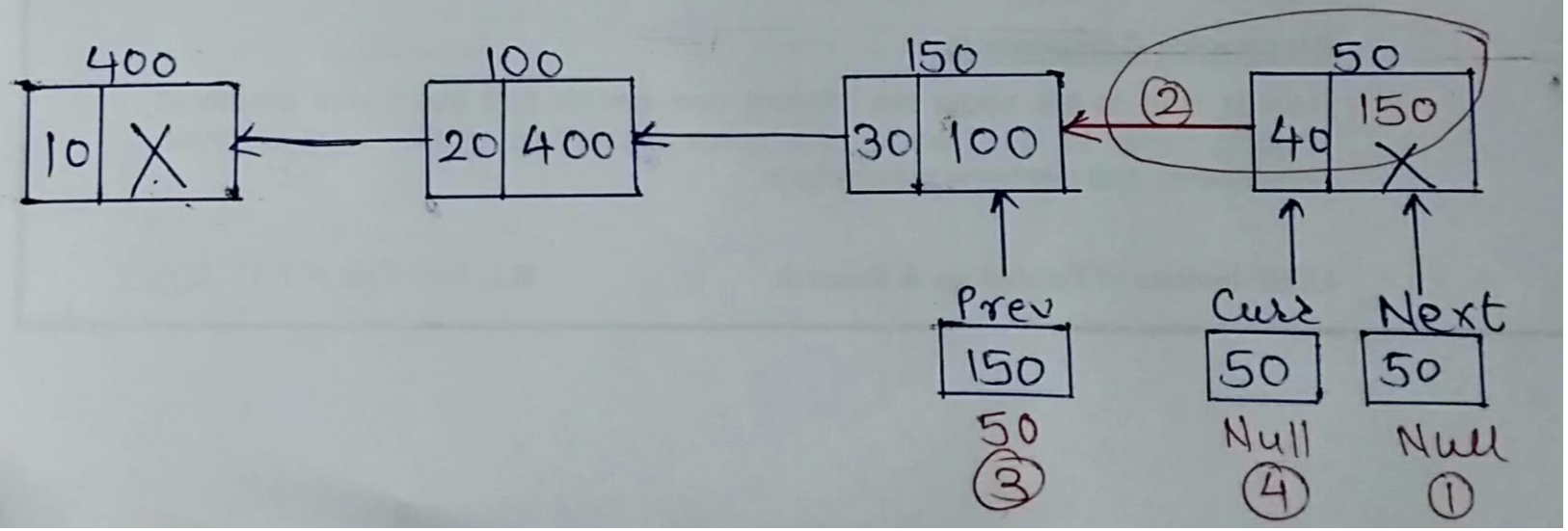
Exit


1)
head
400
10
NULL
100
100
20
150
150
30
50
50
40
2)
100
150
50
Prev
NULL
400
Curr
400
100
next
100
NULL
400
10
NULL
100
400
20
150
150
30
50
50
40
Prev
400
100
Curr
100
150
Next
100
150

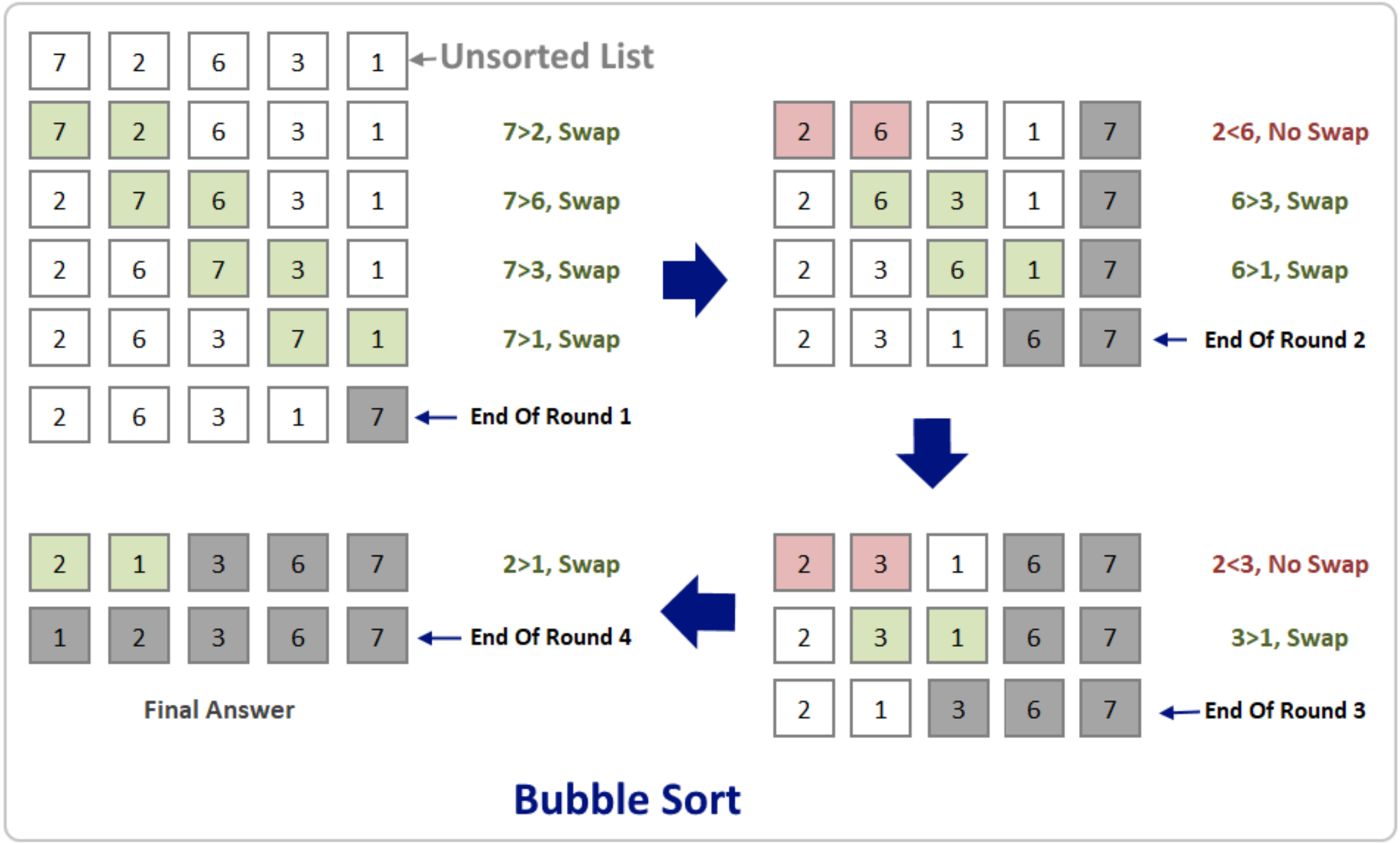
3)


400
10
X
100
20
400
150
100
30
50
②
50
40
X
Prev
100
150
③
Curr
150
50
④
Next
150
50
①

4)

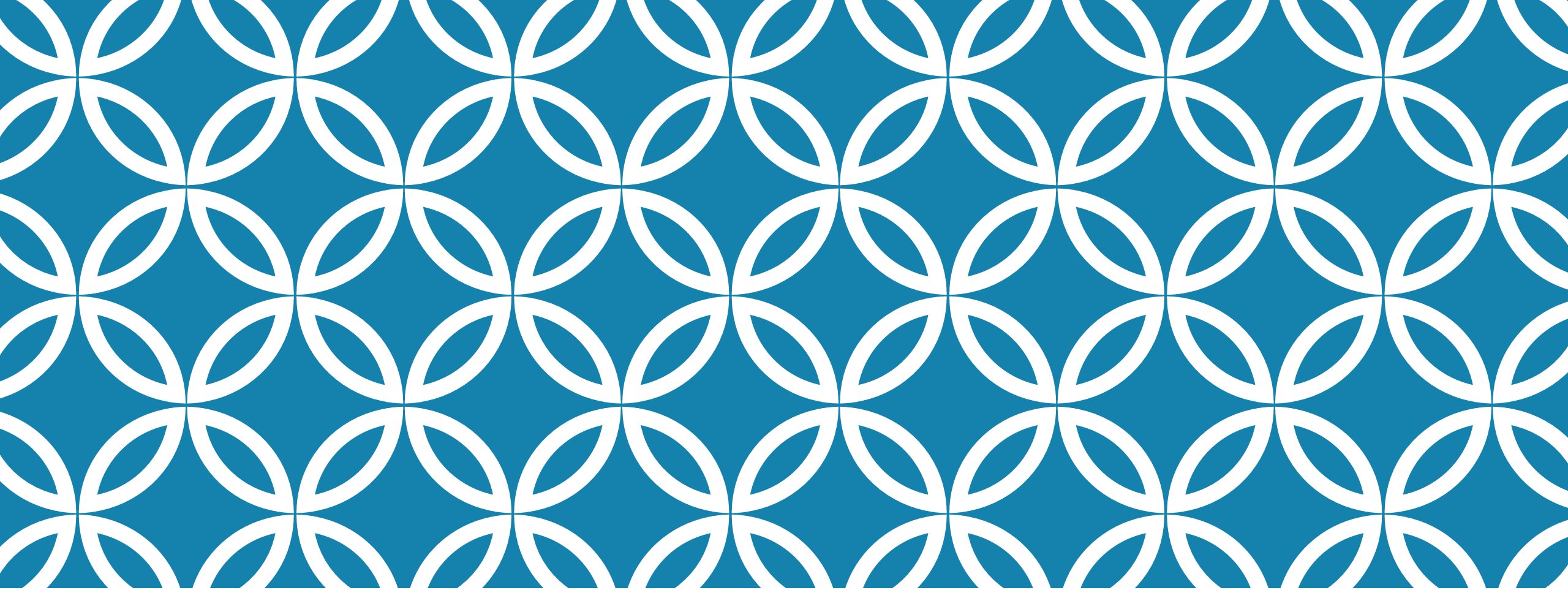

400
10
X
100
20
400
150
30
100
②
50
40
150
X
Prev
150
50
③
Curr
50
Null
④
Next
50
Null
①

SORT LL


7 2 6 3 1 ← Unsorted List
7 2 6 3 1 7>2, Swap
2 7 6 3 1 7>6, Swap
2 6 7 3 1 7>3, Swap
2 6 3 7 1 7>1, Swap
2 6 3 1 7 ← End Of Round 1
2 6 3 1 7 2<6, No Swap
2 6 3 1 7 6>3, Swap
2 3 6 1 7 6>1, Swap
2 3 1 6 7 ← End Of Round 2
2 3 1 6 7 2<3, No Swap
2 3 1 6 7 3>1, Swap
2 1 3 6 7 ← End Of Round 3
2 1 3 6 7 2>1, Swap
1 2 3 6 7 ← End Of Round 4
Final Answer
Bubble Sort

ASSIGNMENT:

1. What is the difference between Array and linked list.
2. What is LL? Explain about memory allocation in LL.
3. Write the algorithm for following operation for singly LL.
    1. Insert at first
    2. Insert at last
    3. Insert after specific key
    4. Delete at first
    5. Delete at last
    6. Delete a specific key
    7. Sort a LL
    8. Reverse



DATA STRUCTURE AND ALGORITHM

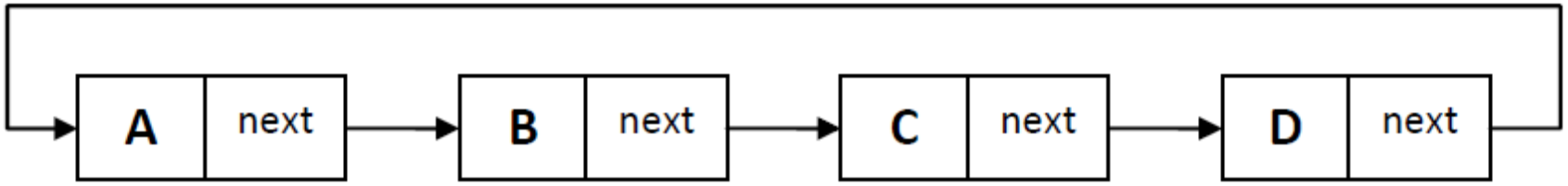
UNIT -2

Linear Data Structures –Link List

Prepared by: Prof. Kinjal Patel

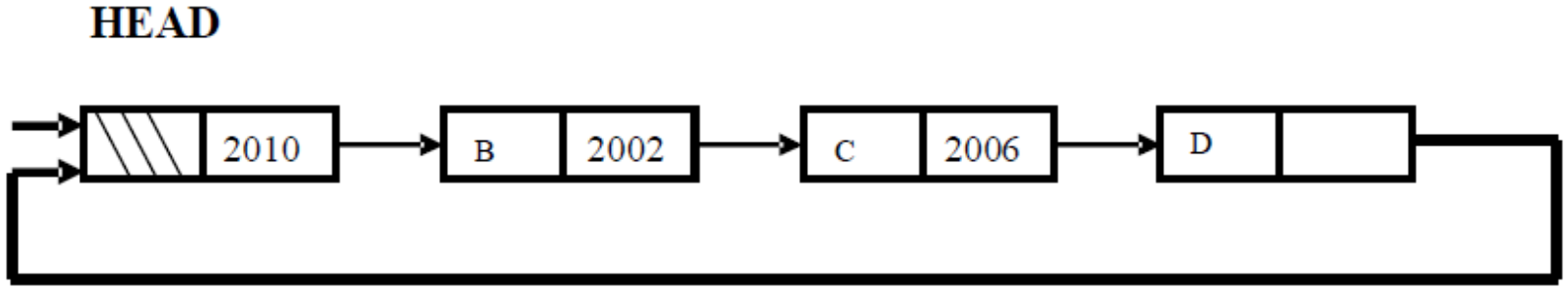
CIRCULAR LINKED LIST

- A list in which last node contains a link or pointer to the first node in the list is known as circular linked list.
- It dose not contain null pointer like singly linked list.
- Representation of circular linked list is shown below:



Circular Linked List

- To detect the end of the list in circular linked list we use one special node called the HEAD node.
- Such a circular linked list with HEAD node is shown below:



- Advantage of such a representation is that the list is never empty.

ADVANTAGE OF CIRCULAR LINKED LIST OVER SINGLY LINKED LIST :

(1) In singly linked list the last node contains NULL address. So if we are at middle of the list and want to access the first node we can not go backward in the list. Thus every node in the list is not accessible from given node. While in Circular linked list last node contains an address of the first node so every node is in the list is accessible from given node.

(2) Circular linked list every node is accessible from given node so there is no need to give address of the first node in the list.

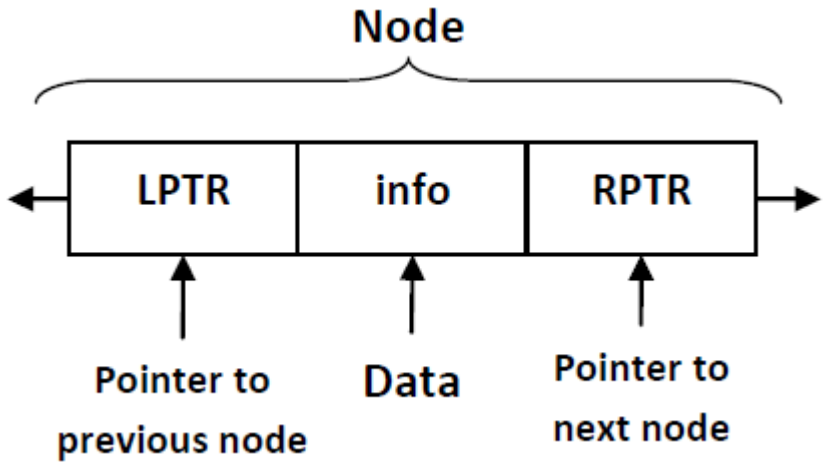
(3) Concatenation and splitting operations are also efficient in circular linked list as compared to the singly linked list.

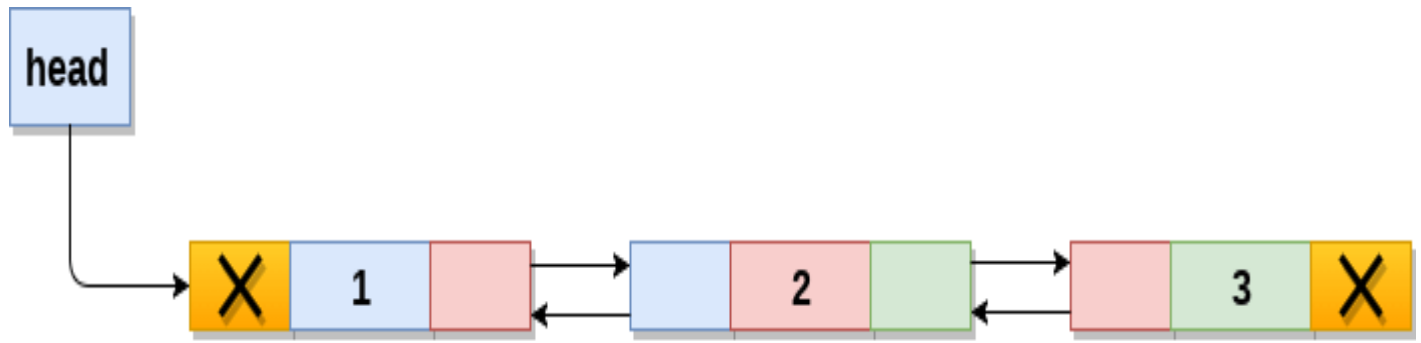
DISADVANTAGE:

1) Without some care in processing it is possible to get into the infinite loop. So we must able to detect end of the list.

DOUBLY LINKED LIST

- In Singly linked list we are able to traverse the list in only one direction. However in some cases it is necessary to traverse the list in both directions.
- This property of linked list implies that each node must contain two link fields instead of one link field.
- One link is used to denote the predecessor of the node and another link is used to denote the successor of the node.
- Thus each node in the list consist of three fields:
  1. Information
  2. LPTR
  3. RPTR





Doubly Linked List

INSERT AT BEGINNING

Insert_first(head,val) :This function inserts a new node at the first position of linked list.

(1) [Create a new node with data]

ptr= new Node

ptr-> data =val

ptr-> next = ptr->prev = NULL

(2) [Insert at First]

IF (head != NULL)

head->prev=ptr

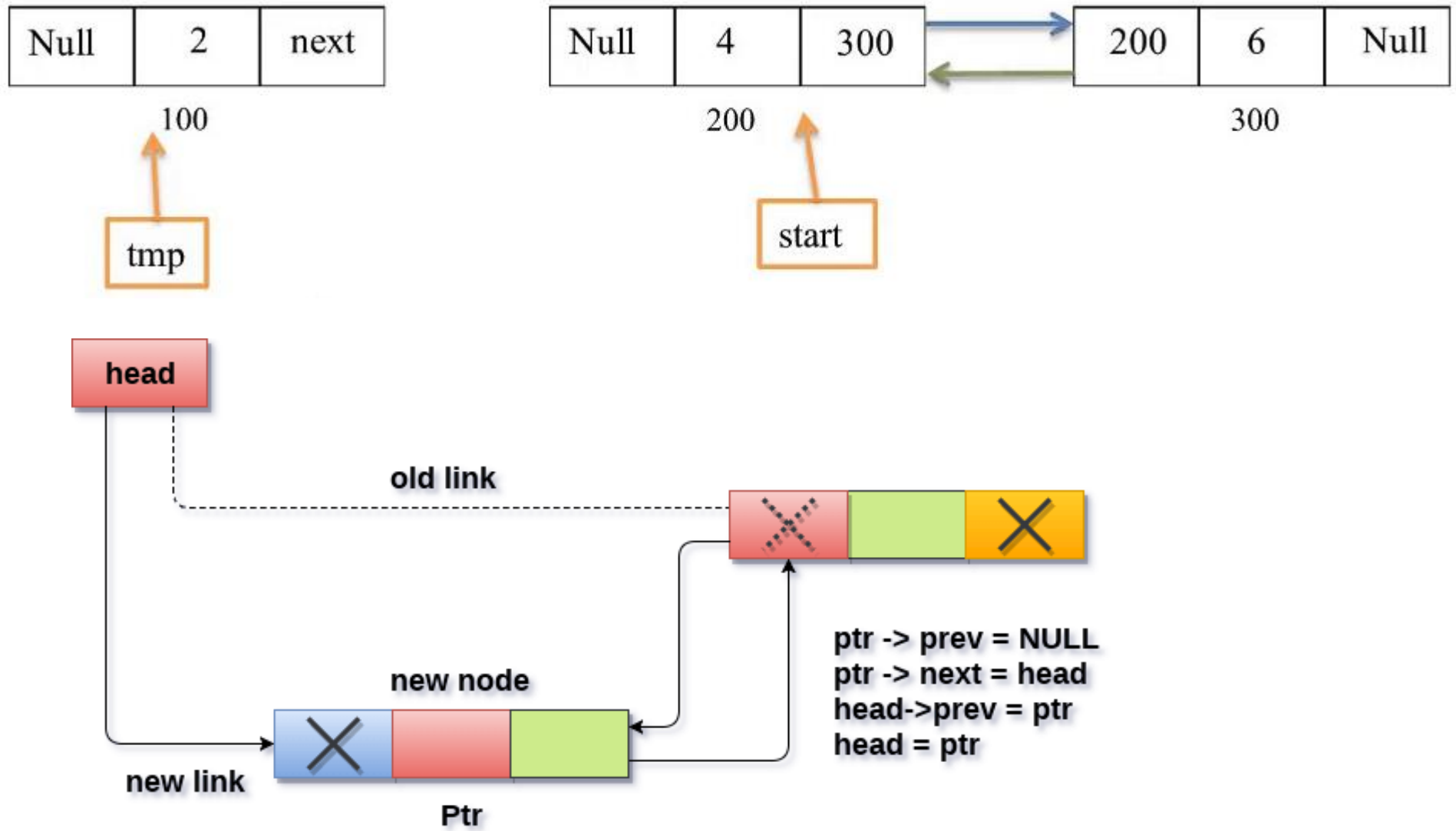
END IF

ptr->next=head

head= ptr

(3) [Finish]

Exit



Insertion into doubly linked list at beginning

INSERT AT END

Insert_end(head,val) :This function inserts a new node at the end of linked list.

(1) [Create a new node with data]

ptr= new Node

ptr-> data = val

ptr-> next = ptr->prev = NULL

(2) [Insert at First if head is NULL]

IF (head==NULL) THEN

head = ptr

Return

(3) [Inset at last if head is NOT NULL]

IF (head != NULL) THEN

temp = head

while (temp -> next != NULL)

DO

temp = temp -> next

END DO

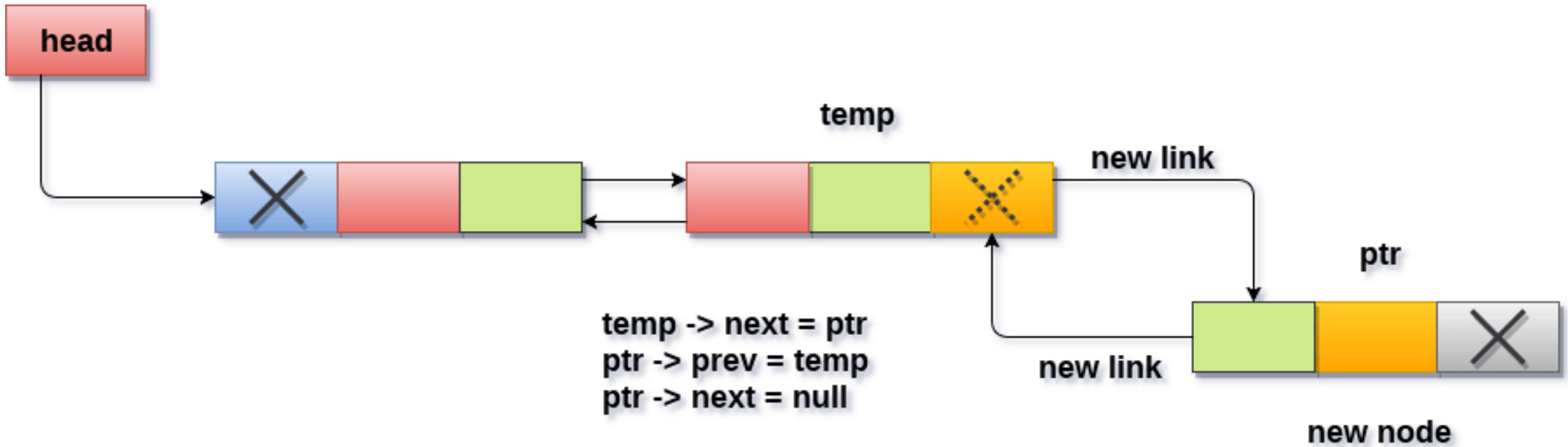
temp->next = ptr

ptr->prev=temp

END IF

(4) [Finish]

Exit



Insertion into doubly linked list at the end

INSERT AFTER SPECIFIC KEY

Insert_At(head,val,Key) :This function inserts a new node after specific key node in linked list.

(1) [Search for key]

temp = head

while(temp->data!= key && temp!= NULL)

DO

temp = temp->next

END DO

(2) [Check key found or not]

IF (temp == NULL)

Print “Key not found”

Return

(3) [Create a new node]

ptr = new Node

ptr-> data = val

ptr-> next = ptr->prev = NULL

(4) [Insert the node after key node]

temp->next->prev=ptr

ptr->prev = temp

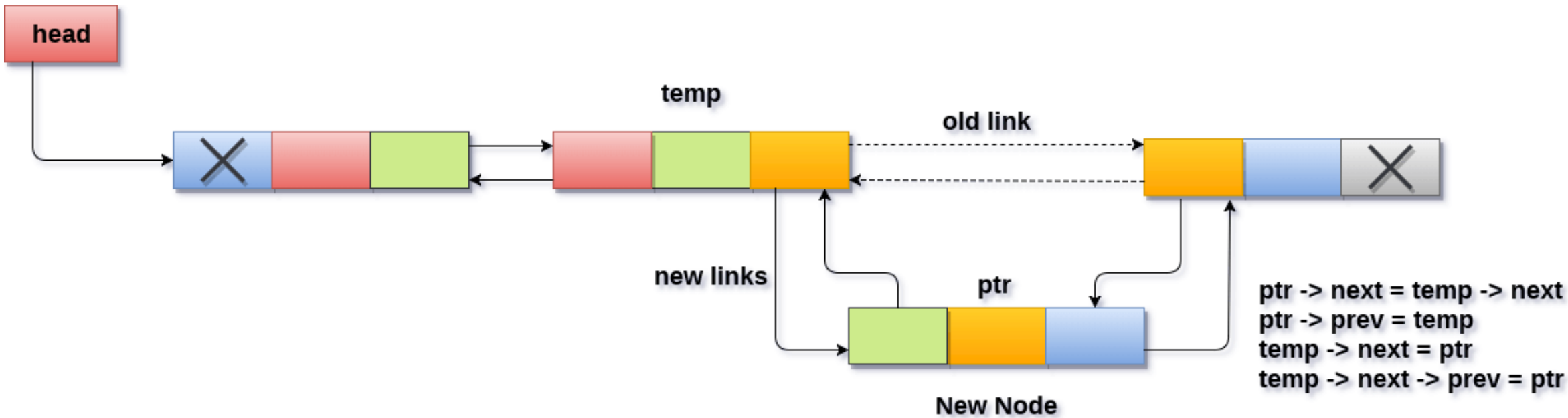
ptr->next=temp->next

temp->next=ptr

print “Node Inserted”

(5) [Finish]

Exit



Insertion into doubly linked list after specified node

DELETION OF SPECIFIC KEY

delete_key(head,key) :This function delete a node from linked list by specific value

(1) [Check LL is empty]

IF (head == NULL)

print “LL is empty”

Return

(2) [key in first node]

temp = head

IF (head->data==key) THEN

head=temp->next;

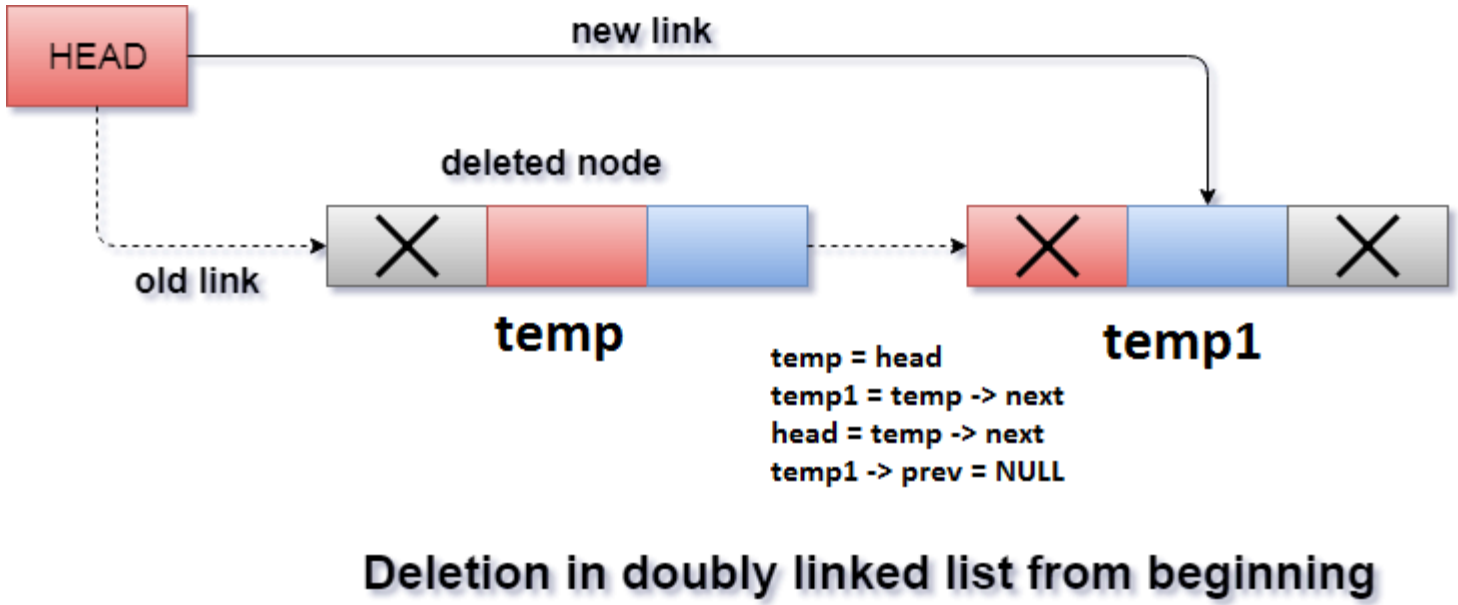
temp1=temp->next;

temp1->prev=NULL;

free(temp);

END IF

Return



Deletion in doubly linked list from beginning

DELETION OF SPECIFIC KEY

(3) [Find node having KEY value]

```
while (temp->data != val && temp!=NULL)
    DO
        temp = temp -> next
    END DO
```

(4) [Key not found]

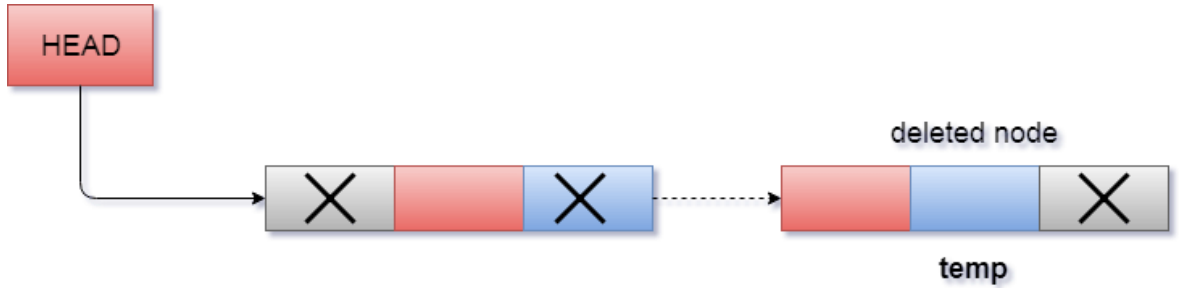
```
IF (temp == NULL)
    print "KEY not found "
    Return
```

(5) [Delete the node having KEY value]

```
IF (temp-> next ==NULL)
    temp -> prev -> next = NULL
ELSE
    ptr = temp -> prev
    temp -> next->prev = ptr
    ptr -> next = temp->next
END IF
free (temp)
```

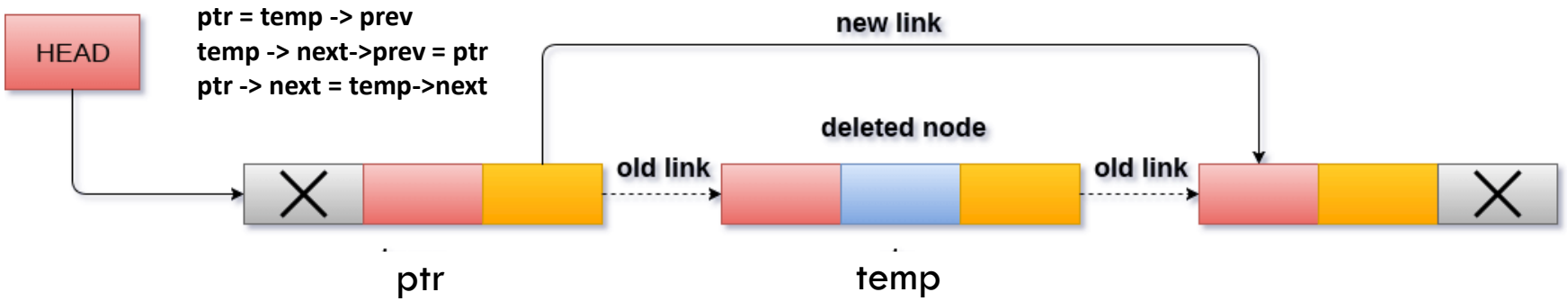
(5) [Finish]

```
Exit
```

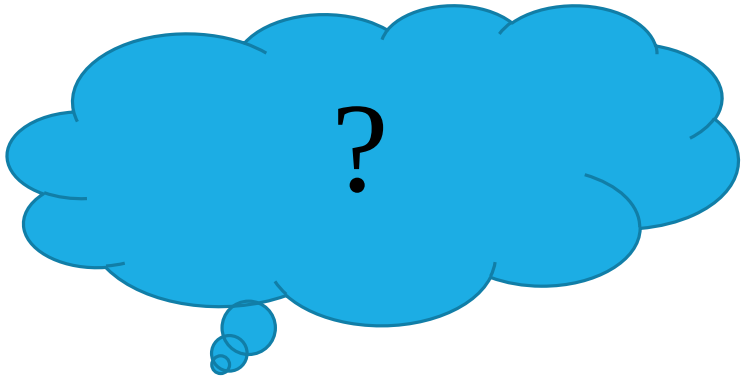


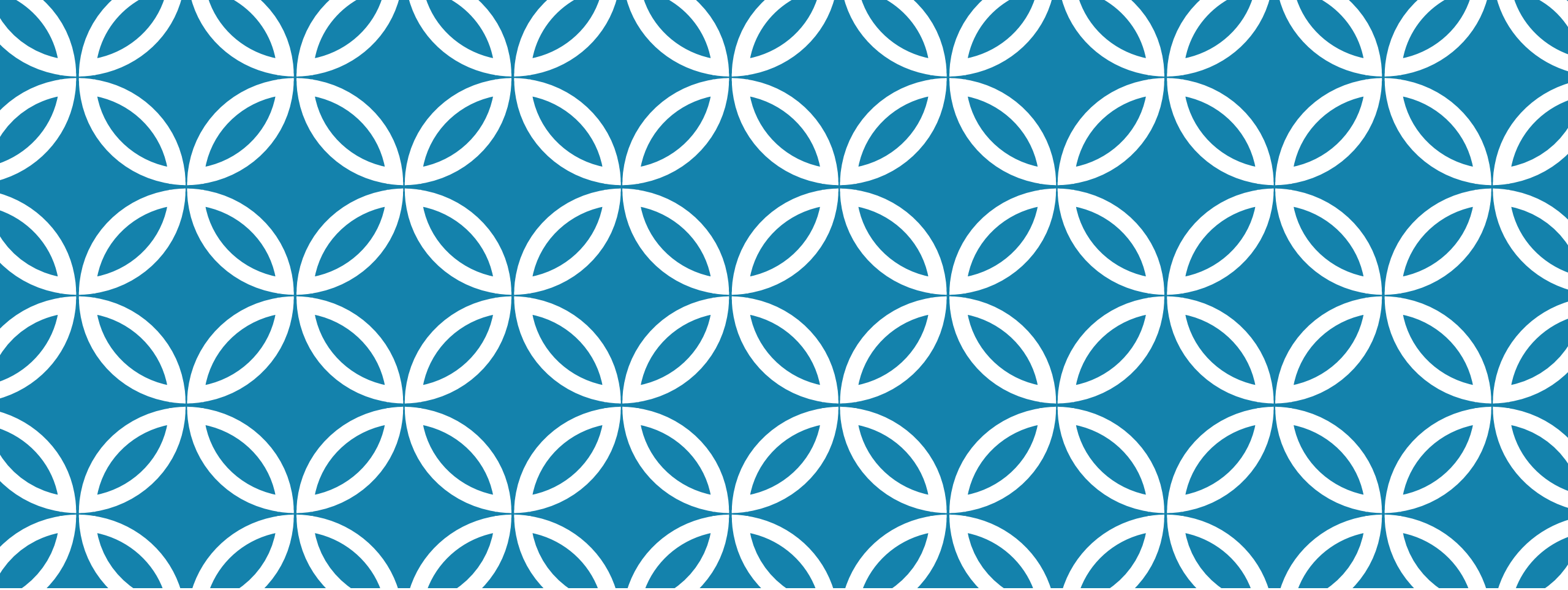
temp->prev->next = NULL
free(temp)

Deletion in doubly linked list at the end



IMPLEMENTATION OF DYNAMIC LINIER DATA STRUCTURE





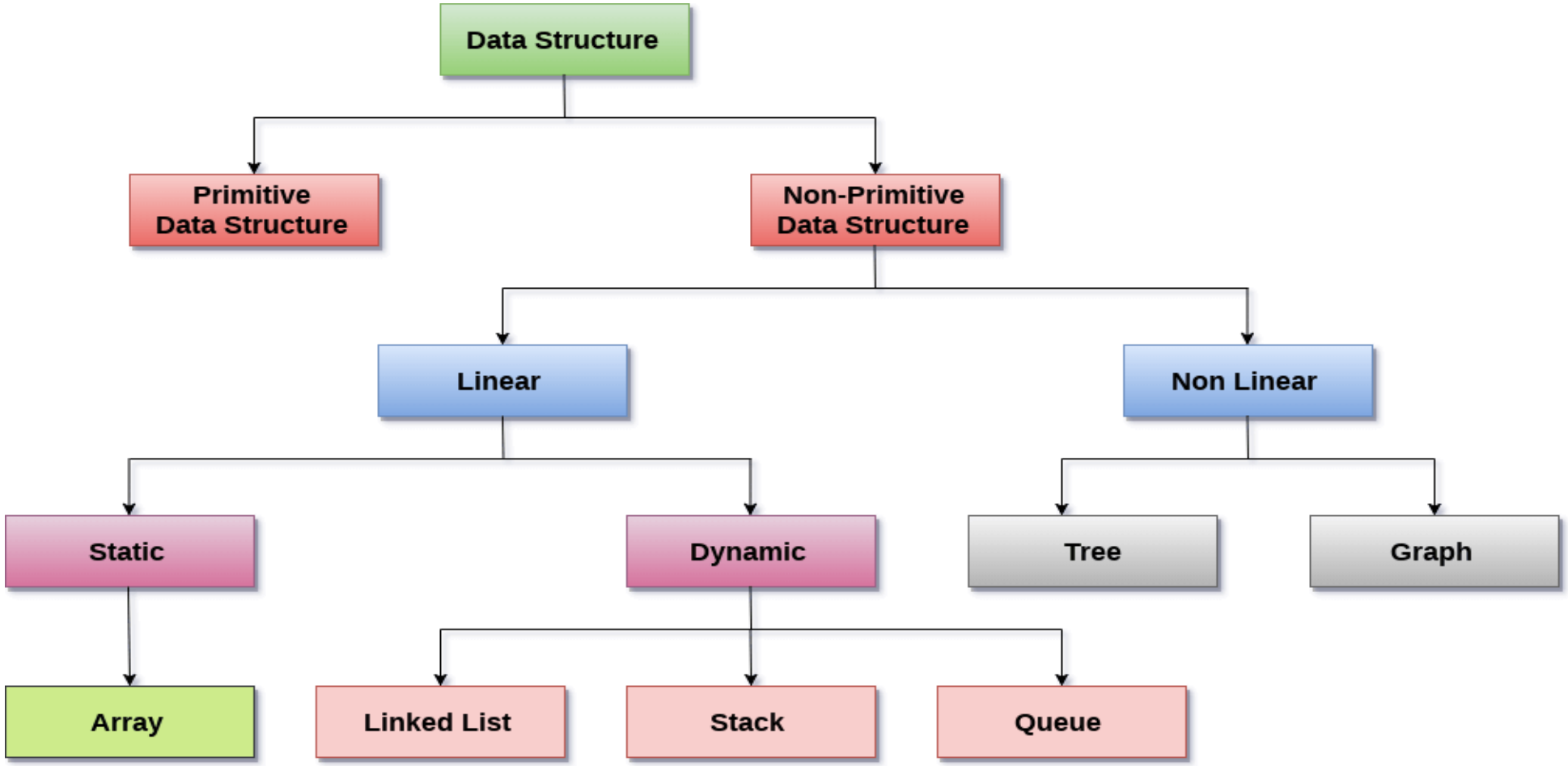
DATA STRUCTURE AND ALGORITHM

UNIT - 3

Non-linear Data structure

Prepared by: Prof. Kinjal Patel.

DATA STRUCTURE CLASSIFICATION

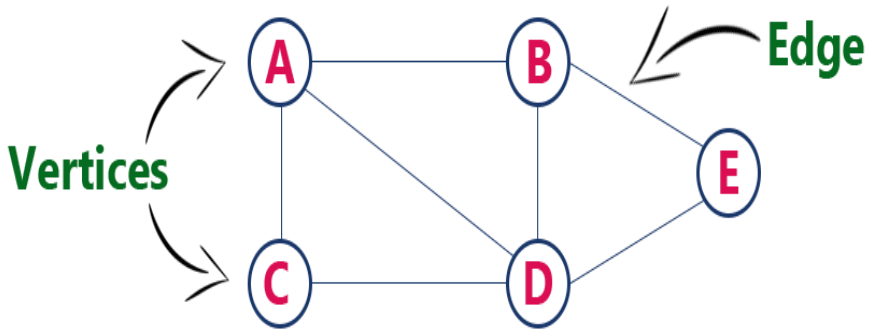


GRAPH TERMINOLOGY

Graph

- A graph G consist of a non-empty set V called the set of nodes (points, vertices) of the graph, a set E which is the set of edges and a mapping from the set of edges E to a set of pairs of elements of V.
- A graph as G=(V , E)
- Graph = Every edge of a graph G, we can associate a pair of nodes of the graph. If an edge X ∈ E is thus associated with a pair of nodes (u,v) where u, v ∈ V then we says that edge x connect u and v.
- This graph G can be defined as G = (V , E)
  Where V = {A,B,C,D,E}

  E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Vertex :- Individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

Adjacent Nodes :- Any two nodes which are connected by an edge in a graph are called adjacent node.

Edge :- An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex). For example, in above graph the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

Undirected Edge - An undirected edge is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

Directed Edge - A directed edge is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

Weighted Edge - A weighted edge is a edge with value (cost) on it.

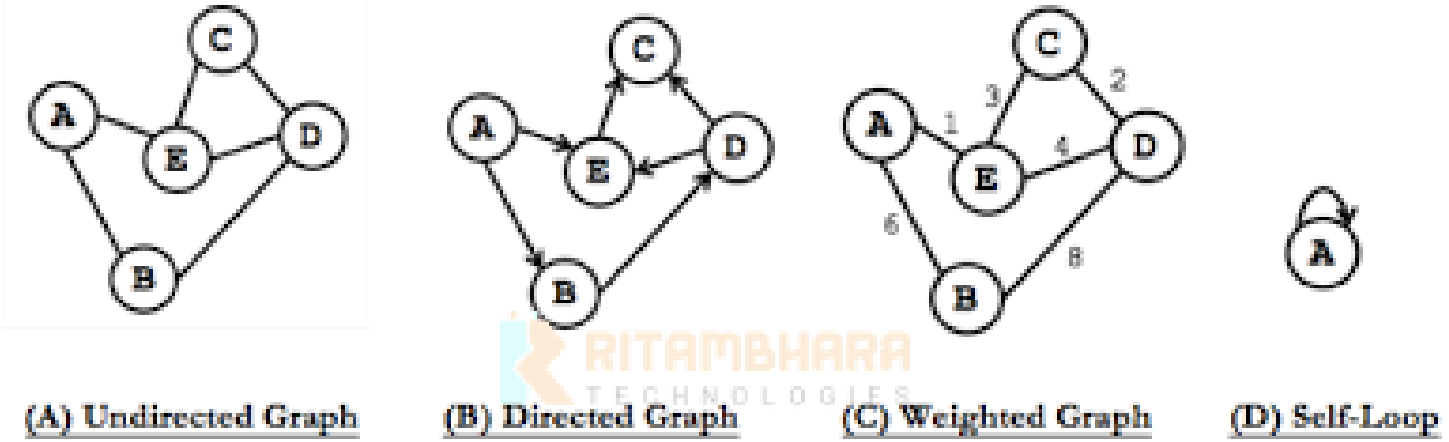
Undirected Graph : - A graph with only undirected edges is said to be undirected graph.

Directed Graph

- A graph with only directed edges is said to be directed graph.
- **Degree :** Total number of edges connected to a vertex is said to be degree of that vertex.
- **Indegree :** Total number of incoming edges connected to a vertex is said to be indegree of that vertex.
- **Outdegree :** Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Mixed Graph :- A graph with both undirected and directed edges is said to be mixed graph.

Self-loop Graph :- A graph which join a nod to itself is called loop or sling.



(A) Undirected Graph (B) Directed Graph (C) Weighted Graph (D) Self-Loop

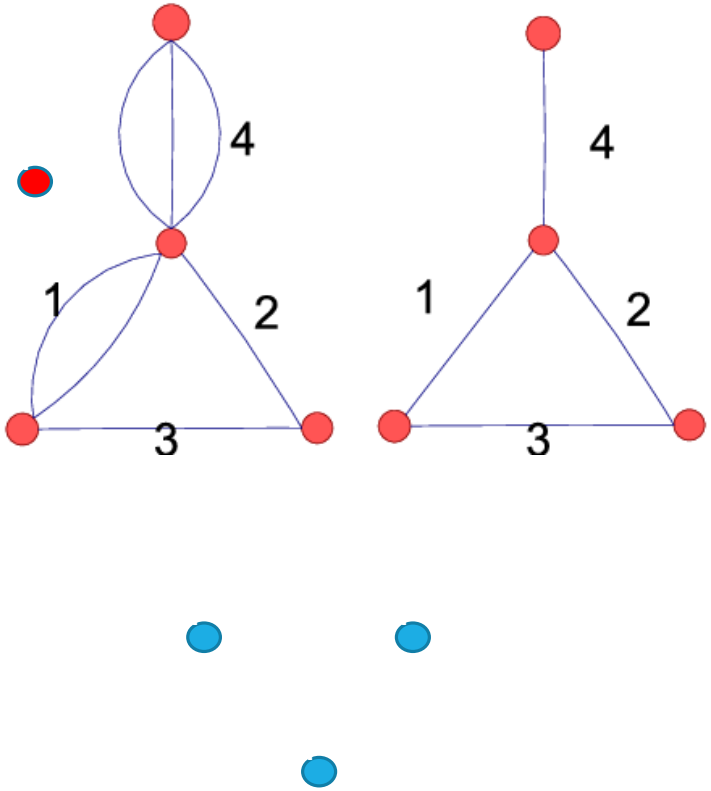
Parallel edges or Multiple edges :- If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.

Multigraph :- Any graph which contains some parallel edges is called multigraph.

Simple Graph :- A graph is said to be simple if there are no parallel and self-loop edges.

Weighted graph :- A graph in which weights are assigned to every edge is called a weighted graph

Isolated Node and Null graph :- A graph, a node which is not adjacent to any other node is called isolated node. A graph containing only isolated node is called a Null graph.



Path :- Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

Simple Path (Edge Simple) :- A path in a diagraph in which the edges are distinct is called simple path or edge simple.

Elementary Path (Node Simple) :- A path in which all the nodes through which it traverses are distinct is called elementary path.

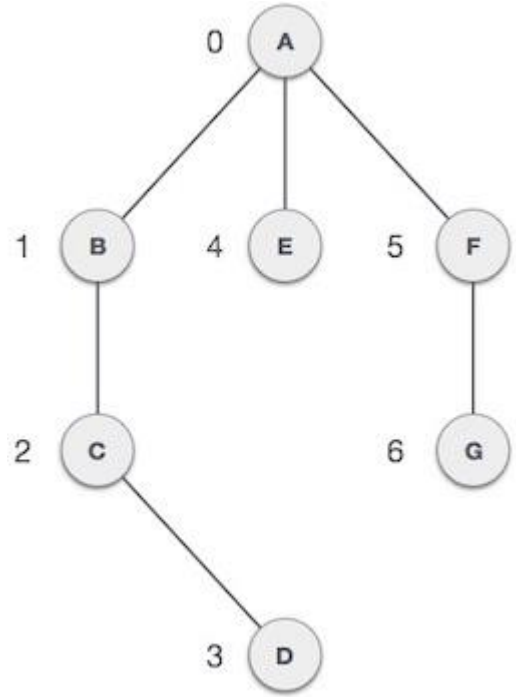
Cyclic graph: - It is a directed **graph** which contains a path from at least one node back to itself.

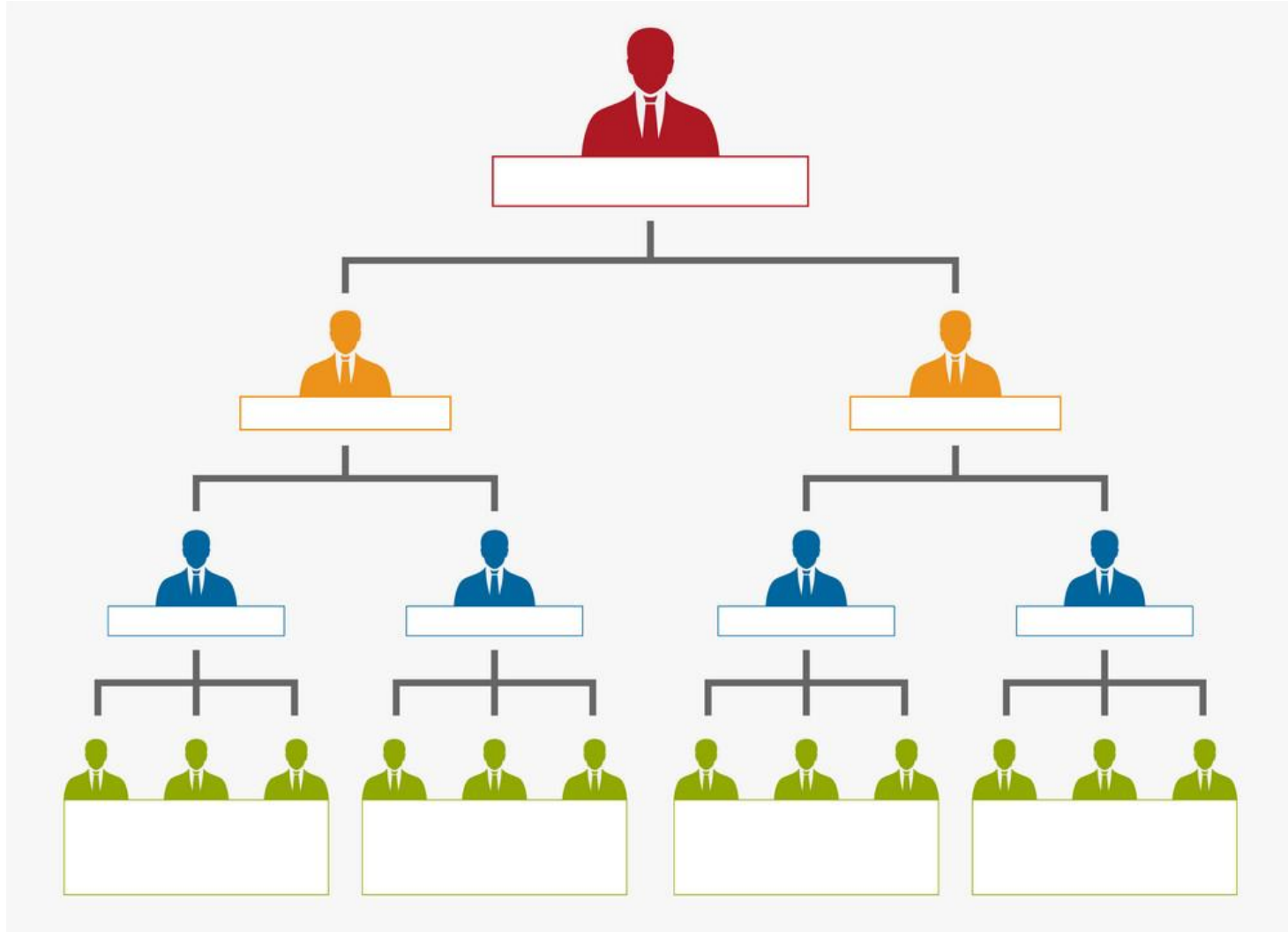
A**cyclic graph** : G**raph** without cycles.

Length of Path :- The number of edges appearing in the sequence of the path is called length of path.

Directed Tree :- A directed tree is an acyclic digraph which has one node called its root with in degree 0, while all other nodes have in degree 1.

A directed tree is an acyclic digraph.





TREE

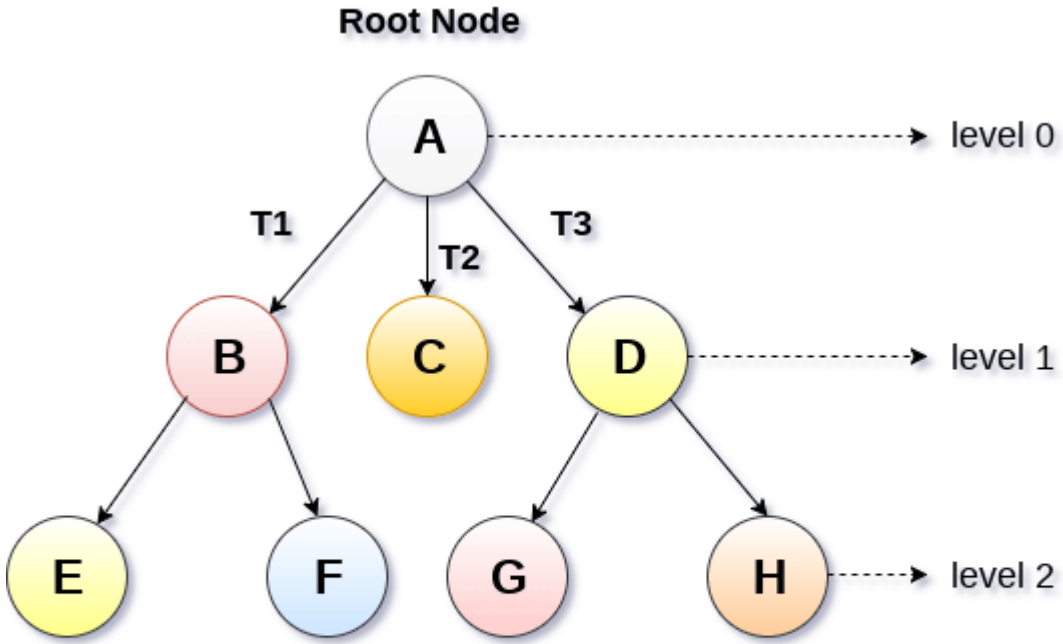
A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.

It can be a defined as a collection of nodes lined together to make a hierarchical structure.

Root Node :- The root node is the topmost node in the tree hierarchy. In other words, the root node is the one which doesn't have any parent.

Sub Tree :- If the root node is not null, the tree T1, T2 and T3 is called sub-trees of the root node.

Leaf Node :- The node of tree, which doesn't have any child node, is called leaf node. Leaf node is the bottom most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.



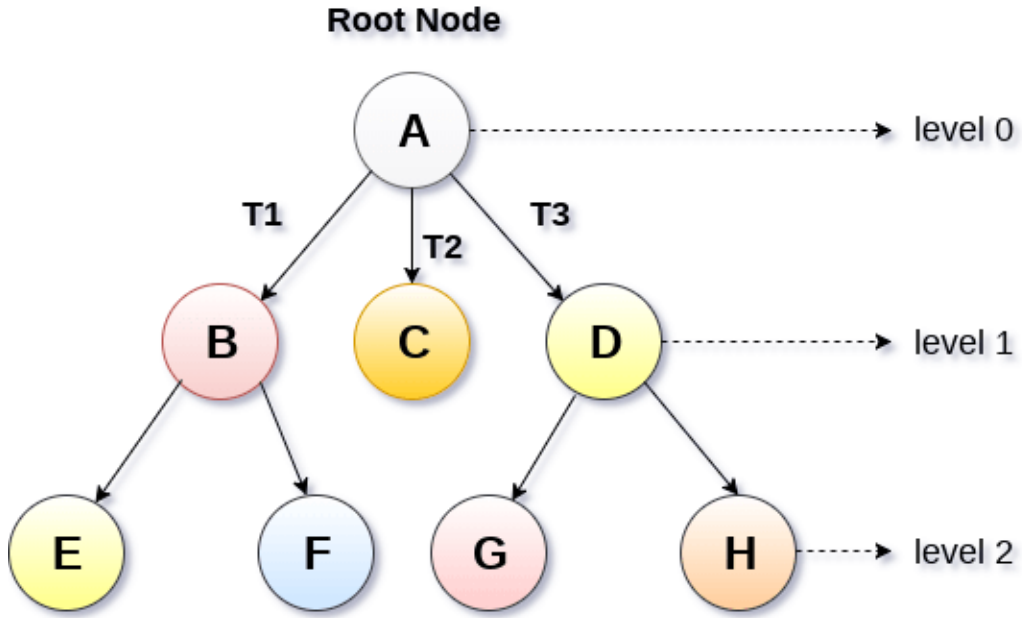
Tree

Path :- The sequence of consecutive edges is called path. Path to the node E is A⟶ B ⟶ E.

Ancestor node :- An ancestor of a node is any predecessor node on a path from root to that node. The root node doesn't have any ancestors. The node F have the ancestors, B and A.

Degree :- Degree of a node is equal to number of children (no of subtree), a node have. The degree of node B is 2. Degree of a leaf node is always 0 while in a complete binary tree, degree of each node is equal to 2.

Level Number :- Each node of the tree is assigned a level number in such a way that each node is present at one level higher than its parent. Root node of the tree is always present at level 0.



Tree

Depth of node :- Length of path from root to the node. Depth of node E is 2(no of edges)

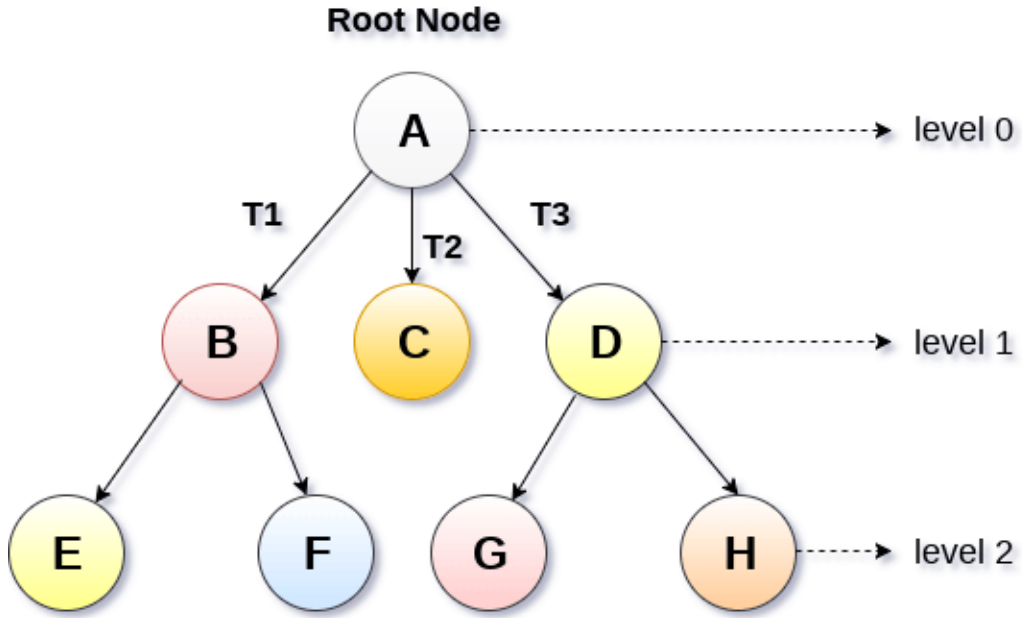
Height of node :- No. of edges in the longest path from the node to leaf node. Height of B is 1(no of edges)

Ordered Tree :- In a directed tree an ordering of the nodes at each level is prescribed then such a tree is called ordered tree.

Forest :- If we delete the root and its edges connecting the nodes at level 1, we obtain a set of disjoint tree. A set of disjoint tree is a forest.

M-ary tree: In a directed tree the outdegree of every node is less than or equal to m, then the tree is called an m-ary tree?

What about when m=2?



Tree

Linked List	Tree
(1) Linked list is an example of linear data structure.	**(1)** Tree is an example of non linear data structure.
(2) Searching time is more required in linked list. Because we must have to traverse each node sequentially in linked list even if the list is sorted.	**(2)** Searching time is less required in tree because we can search an element using binary search method.

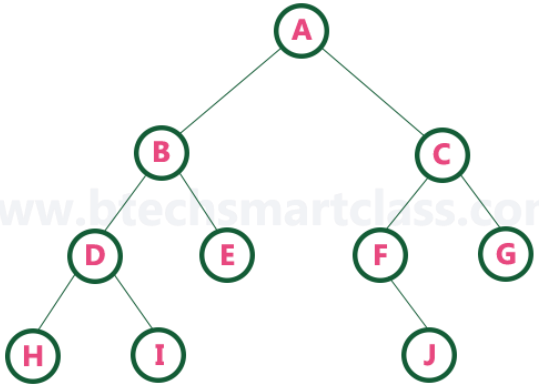
BINARY TREE

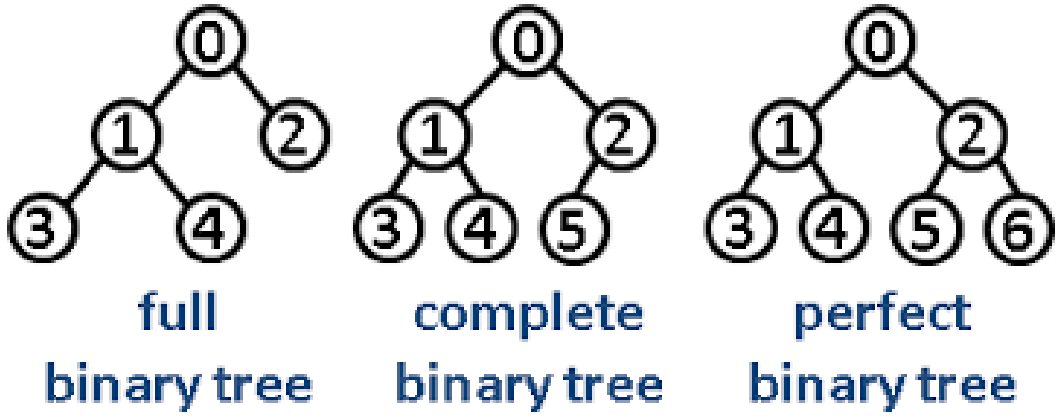
Binary tree: A tree in which every node can have a maximum of two children is called Binary Tree.

Strictly Binary Tree/Full Binary tree : A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree

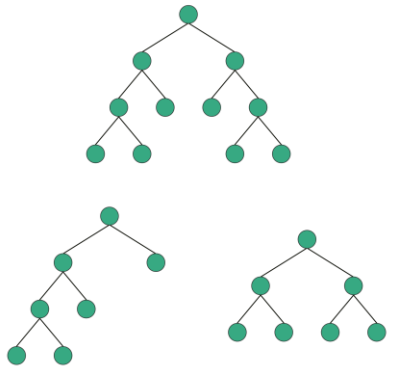
Complete Binary Tree : A **Binary Tree** is a complete **Binary Tree** if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.

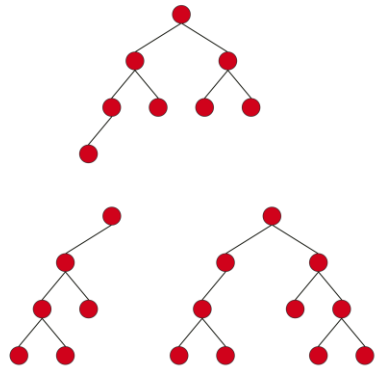
Perfect binary tree: A binary tree with all leaf nodes at the same depth. All internal nodes have exactly two children



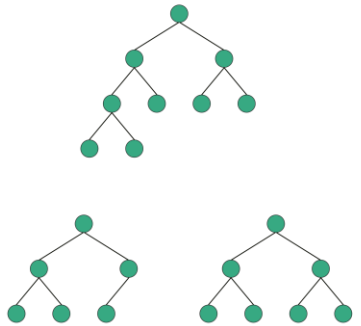


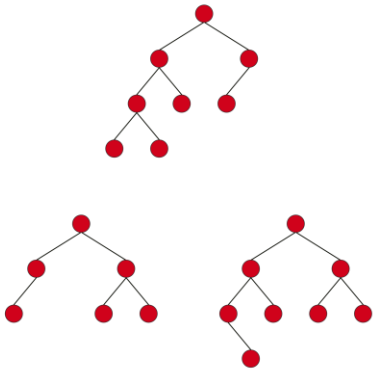
Strictly Binary Tree/Full Binary tree



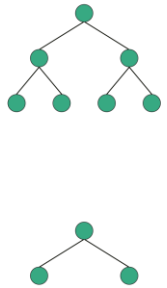


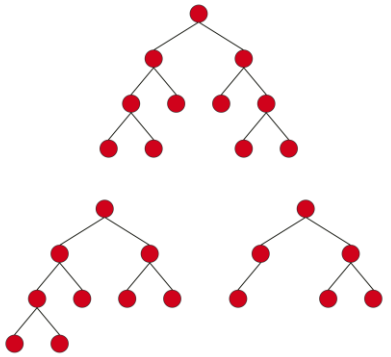
Complete Binary Tree





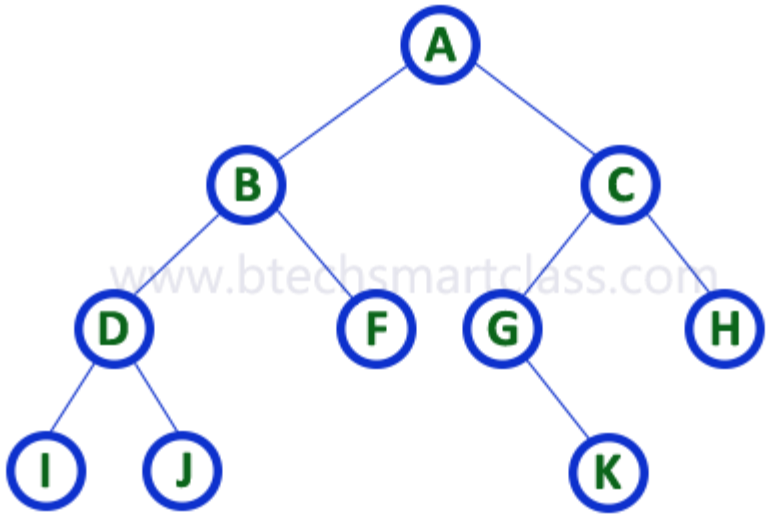
Perfect binary tree:





	Max nodes	Min nodes
Binary tree	$2^{h+1} - 1$	$h + 1$
Full Binary Tree	$2^{h+1} - 1$	$2*h + 1$
Complete binary tree	$2^{h+1} - 1$	2^{h}

ARRAY REPRESENTATION OF BINARY TREE



A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

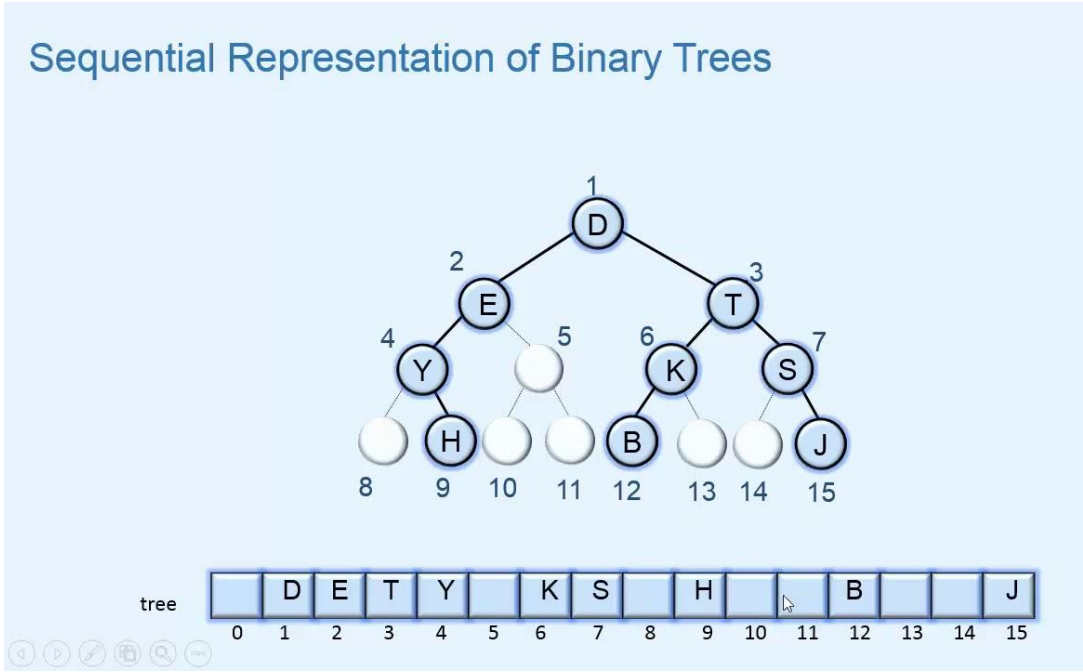
Case 1:

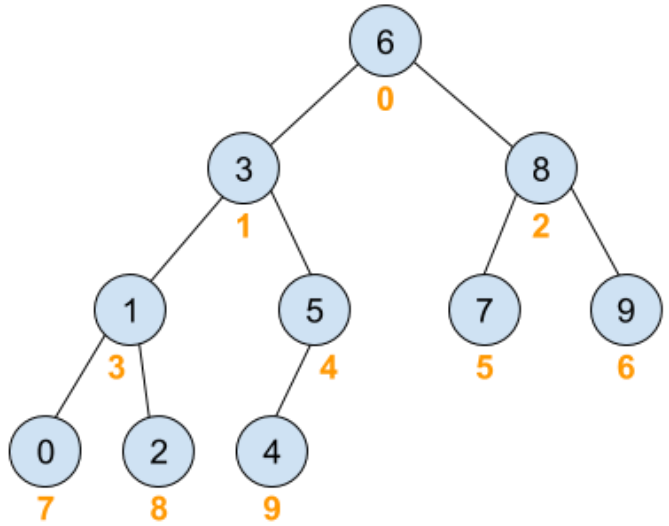
Node i having children

Left child = 2*i

Right child = 2*i + 1

Parent of i node = i/2





i	0	1	2	3	4	5	6	7	8	9
x	6	3	8	1	5	7	9	0	2	4

Case 2:

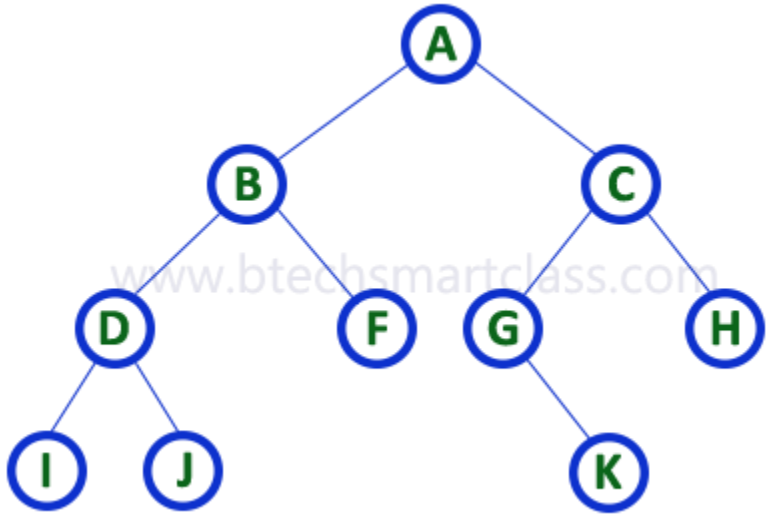
Node i having children

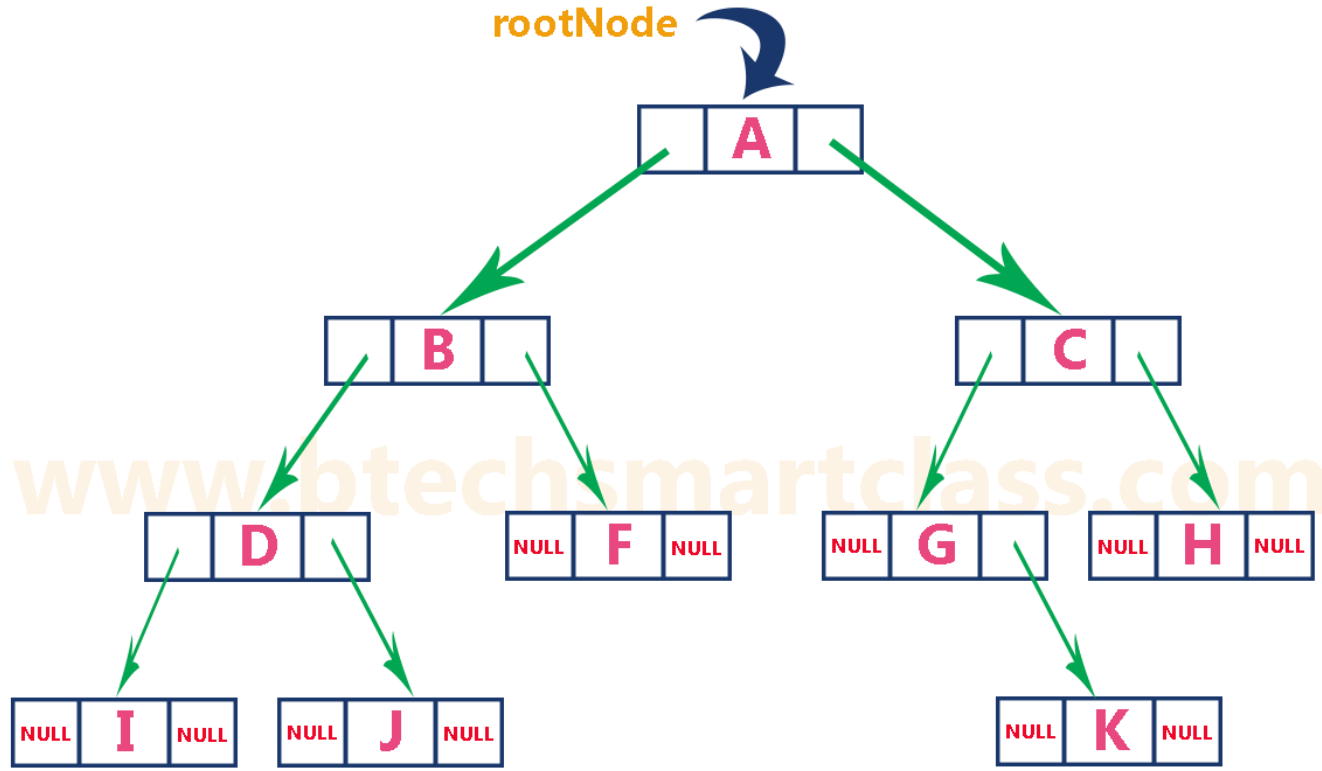
Left child = 2*i + 1

Right child = 2*i + 2

Parent of i node = (i-1)/2

LINK REPRESENTATION OF BINARY TREE





Left Child Address	Data	Right Child Address

TRAVERSAL OF BINARY TREE

SN	Traversal	Description
1	Pre-order Traversal	Traverse the root first then traverse into the left sub-tree and right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively.
2	In-order Traversal	Traverse the left sub-tree first, and then traverse the root and the right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively.
3	Post-order Traversal	Traverse the left sub-tree and then traverse the right sub-tree and root respectively. This procedure will be applied to each sub-tree of the tree recursively.

TRAVERSAL

In - Order Traversal(leftChild - root - rightChild)

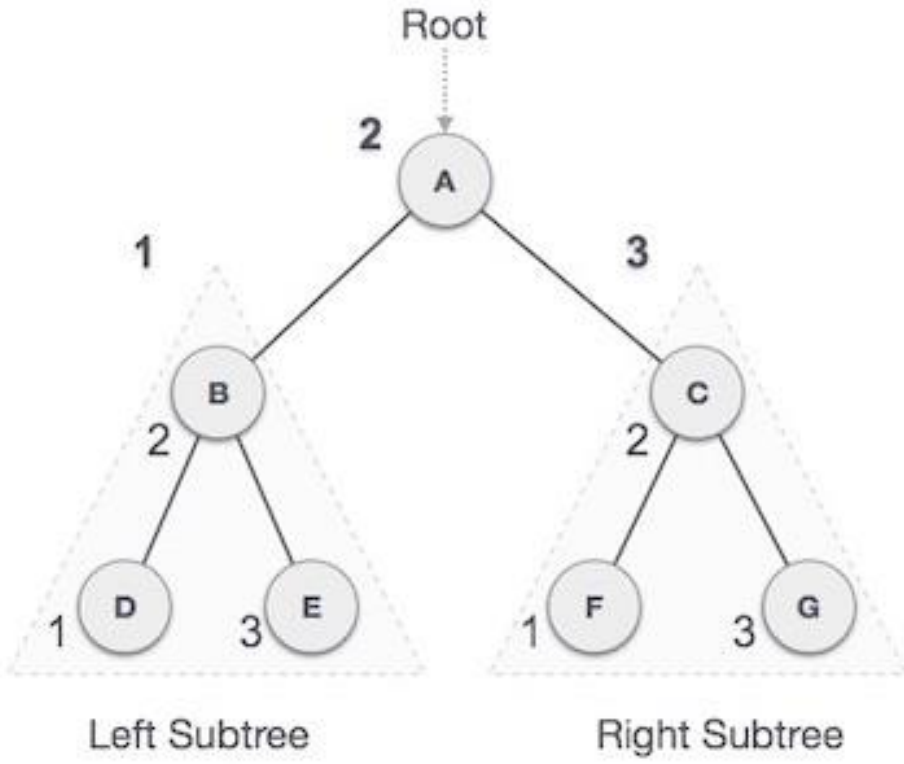
$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Pre - Order Traversal (root - leftChild - rightChild)

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Post - Order Traversal (leftChild - rightChild - root)

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$



TRAVERSAL

In - Order Traversal

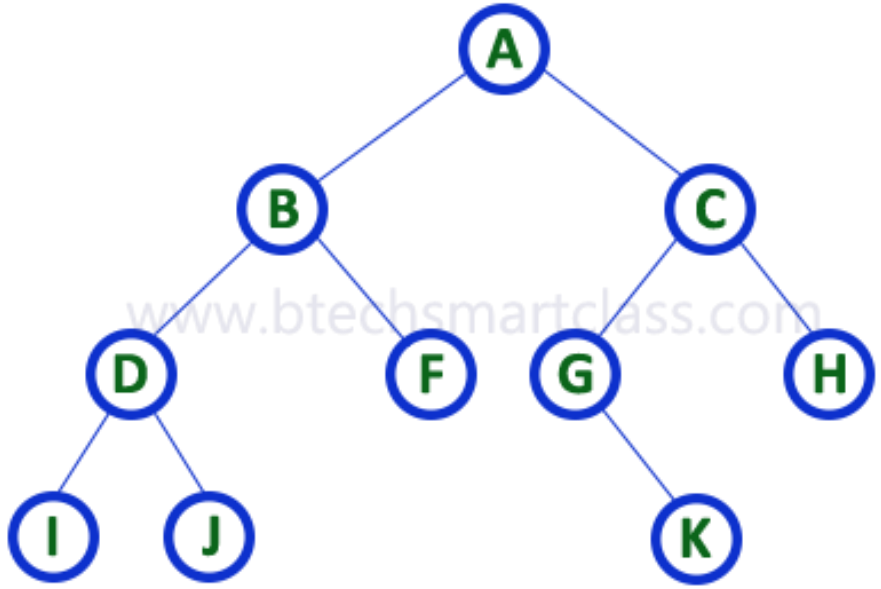
I - D - J - B - F - A - G - K - C - H

Pre - Order Traversal

A - B - D - I - J - F - C - G - K - H

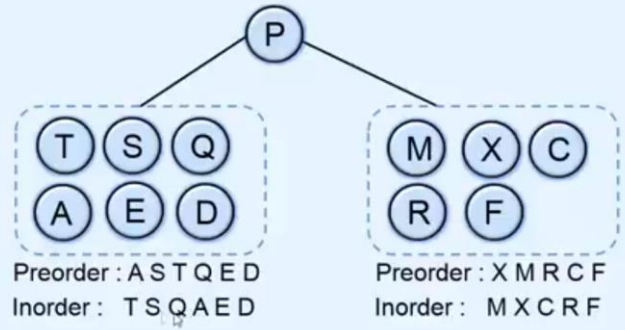
Post - Order Traversal

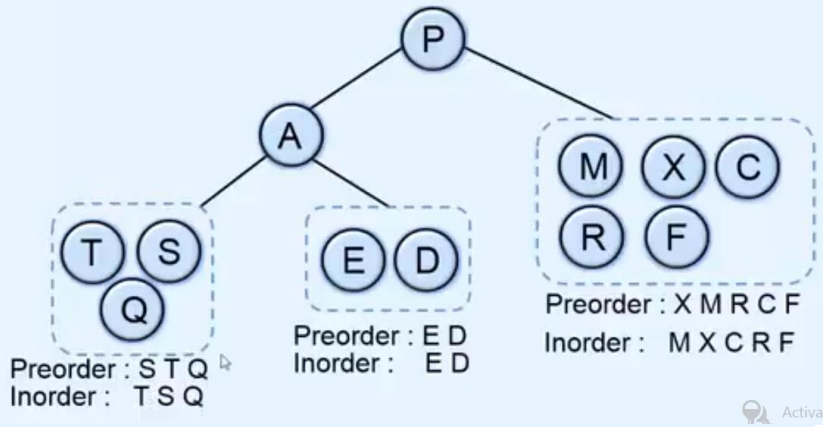
I - J - D - F - B - K - G - H - C - A

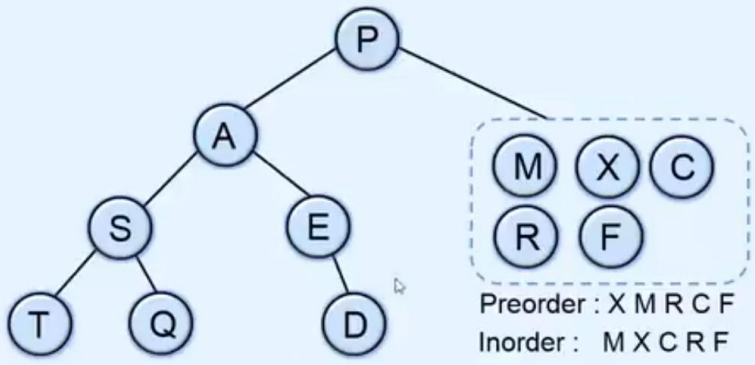


https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/BinaryTreeTraversal.html

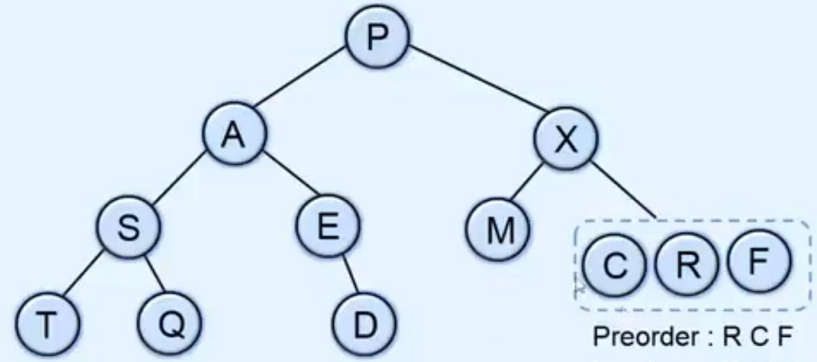
Preorder : P A S T Q E D X M R C F
Inorder : T S Q A E D P M X C R F



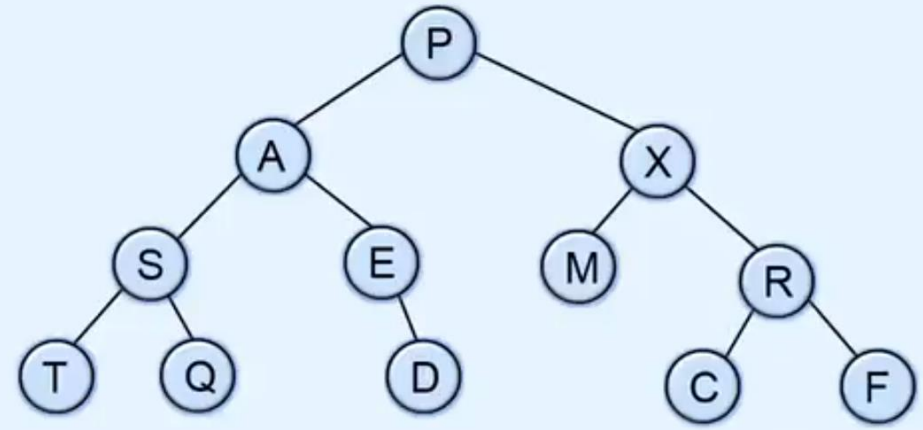




Preorder : P A S T Q E D X M R C F
Inorder : T S Q A E D P M X C R F



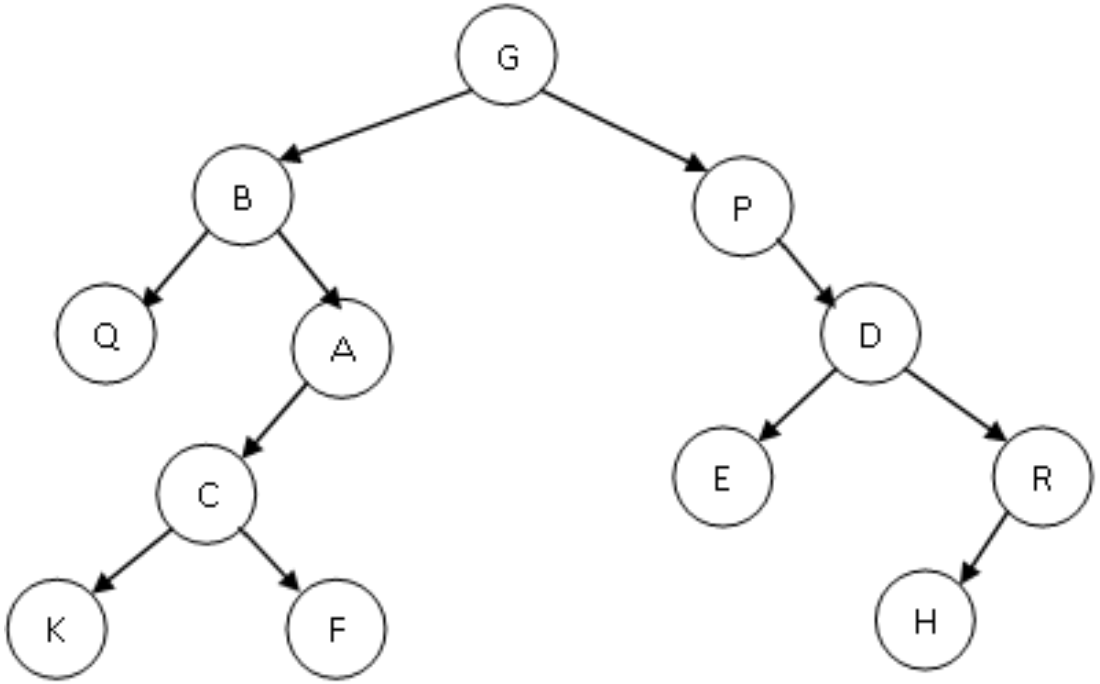
Preorder : P A S T Q E D X M R C F
Inorder : T S Q A E D P M X C R F

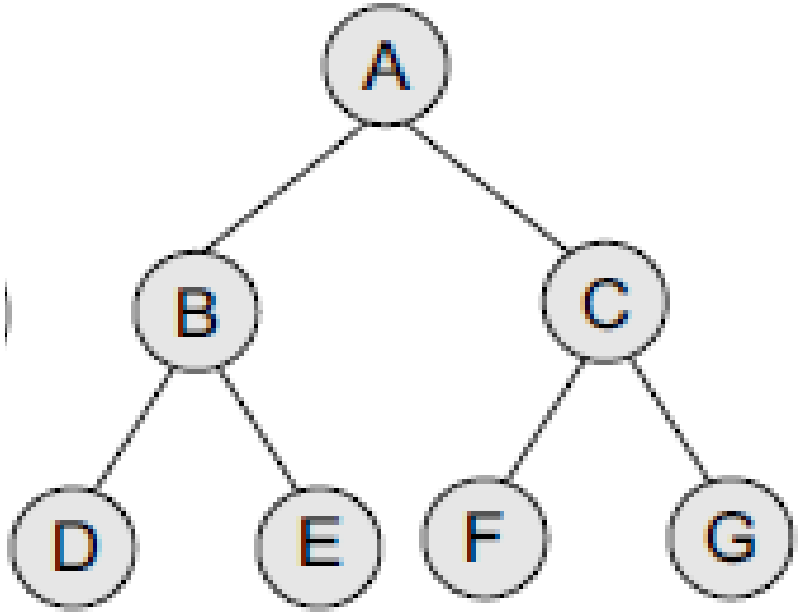


CONSTRUCT A TREE

Preorder : G B Q A C K F P D E R H

Inorder: Q B K C F A G P E D H R





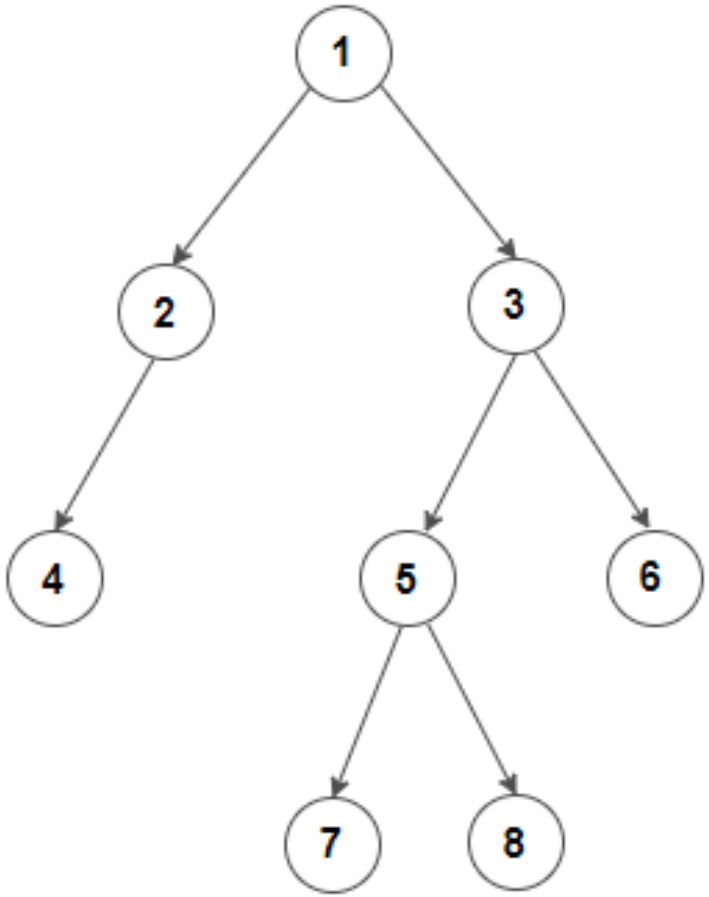
Preorder : A B D E C F G

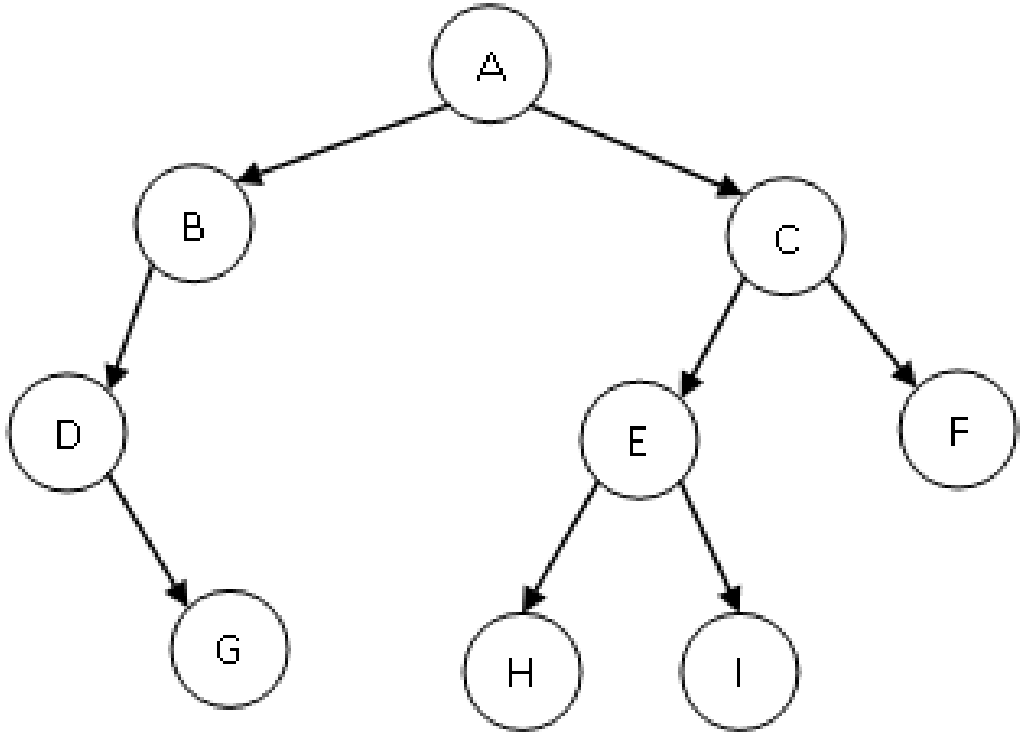
Inorder: D B E A F C G

CONSTRUCT BINARY TREE

Inorder traversal : 4-2-1-7-5-8-3-6

Preorder traversal : 1-2-4-3-5-7-8-6

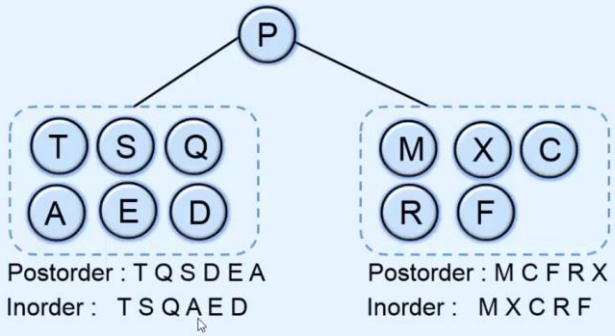




Inorder traversal : D G B A H E I C F

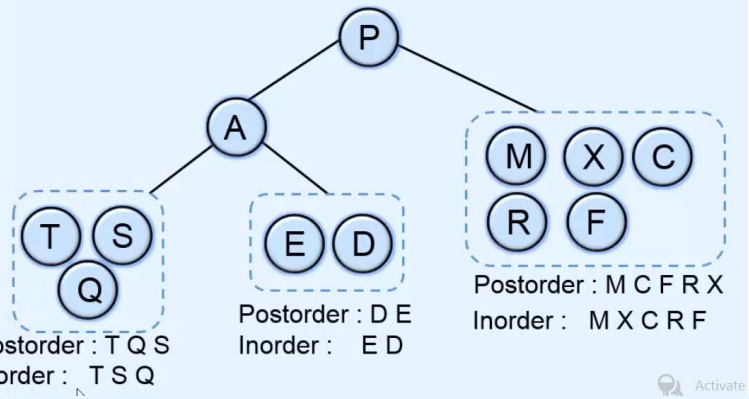
Preorder traversal : A B D G C E H I F

Postorder : T Q S D E A M C F R X P
Inorder : T S Q A E D P M X C R F



Postorder : T Q S D E A
Inorder : T S Q A E D

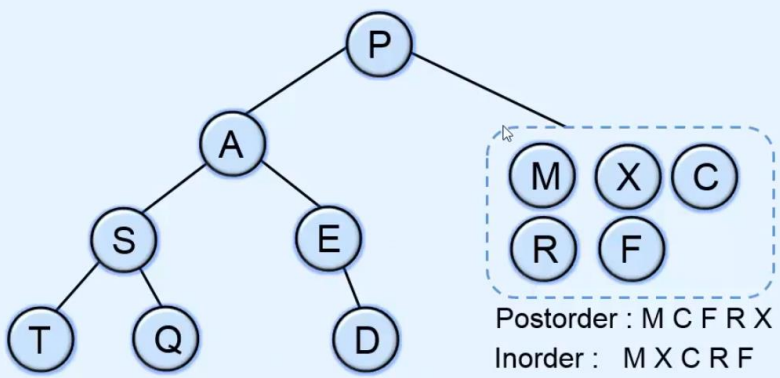
Postorder : M C F R X
Inorder : M X C R F



Postorder : T Q S
Inorder : T S Q

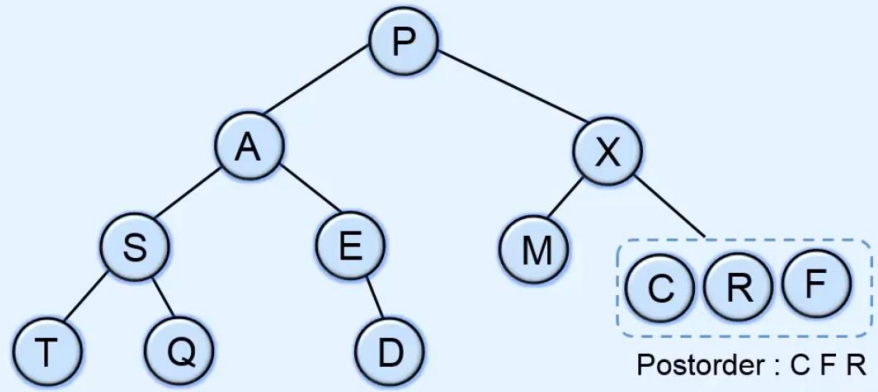
Postorder : D E
Inorder : E D

Postorder : M C F R X
Inorder : M X C R F



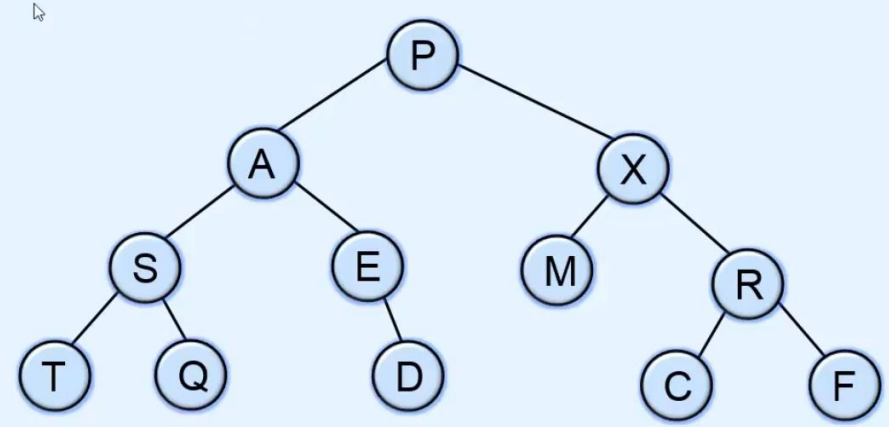
Postorder : M C F R X
Inorder : M X C R F

Postorder : T Q S D E A M C F R X P
Inorder : T S Q A E D P M X C R F



Postorder : C F R
Inorder : C R F

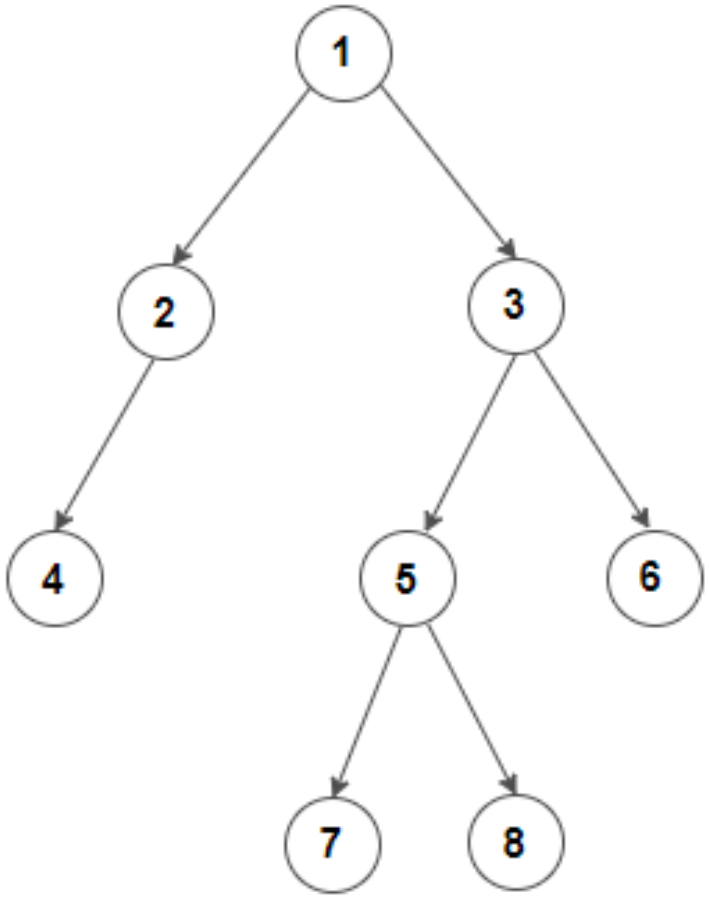
Postorder : T Q S D E A M C F R X P
Inorder : T S Q A E D P M X C R F

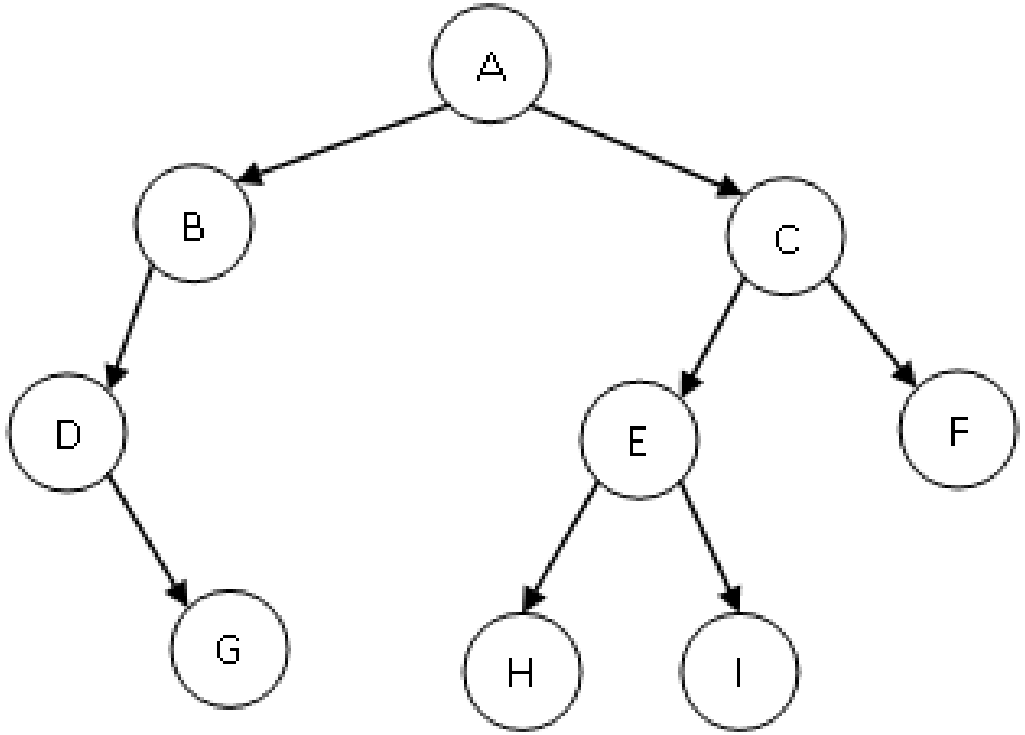


CONSTRUCT BINARY TREE

Inorder traversal : 4-2-1-7-5-8-3-6

Postorder traversal : 4-2-7-8-5-6-3-1



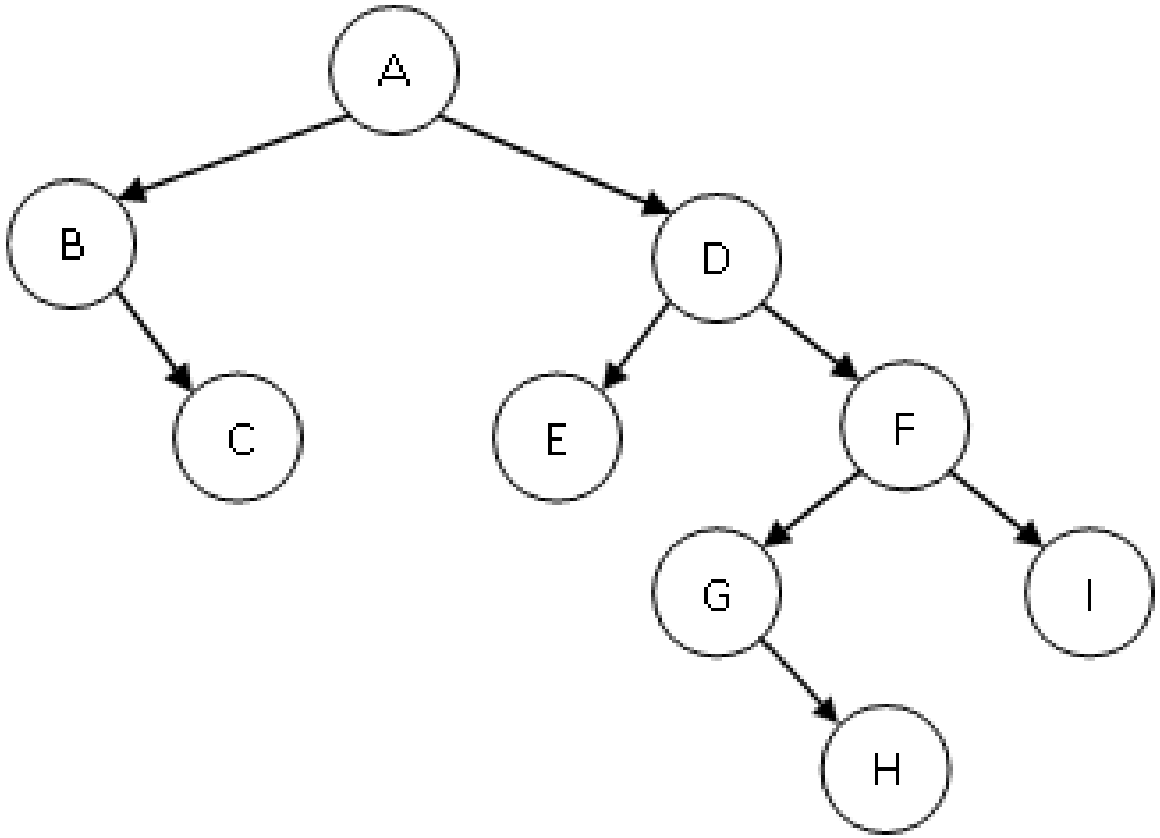


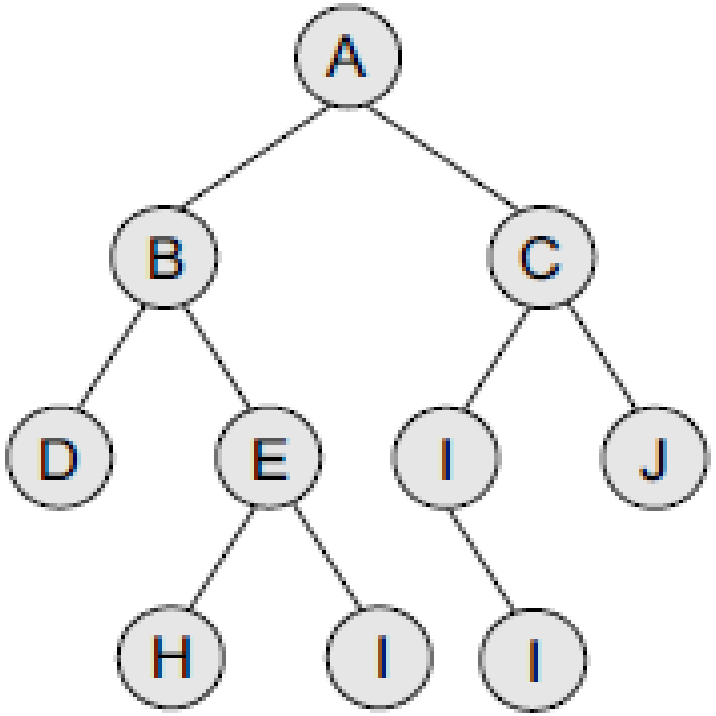
Inorder traversal : D G B A H E I C F

Postorder traversal : G D B H I E F C A

Postorder : C B E H G I F D A

Inorder: B C A E D G H F I





Postorder : D H I E B J F G C A

Inorder: D B H E I A F J C G

BINARY SEARCH TREES

- Binary search tree has following characteristics:

  (1) All the nodes to the left of root node have value less than the value of root node.
  (2) All the nodes to the right of root node have value greater than the value of root node.

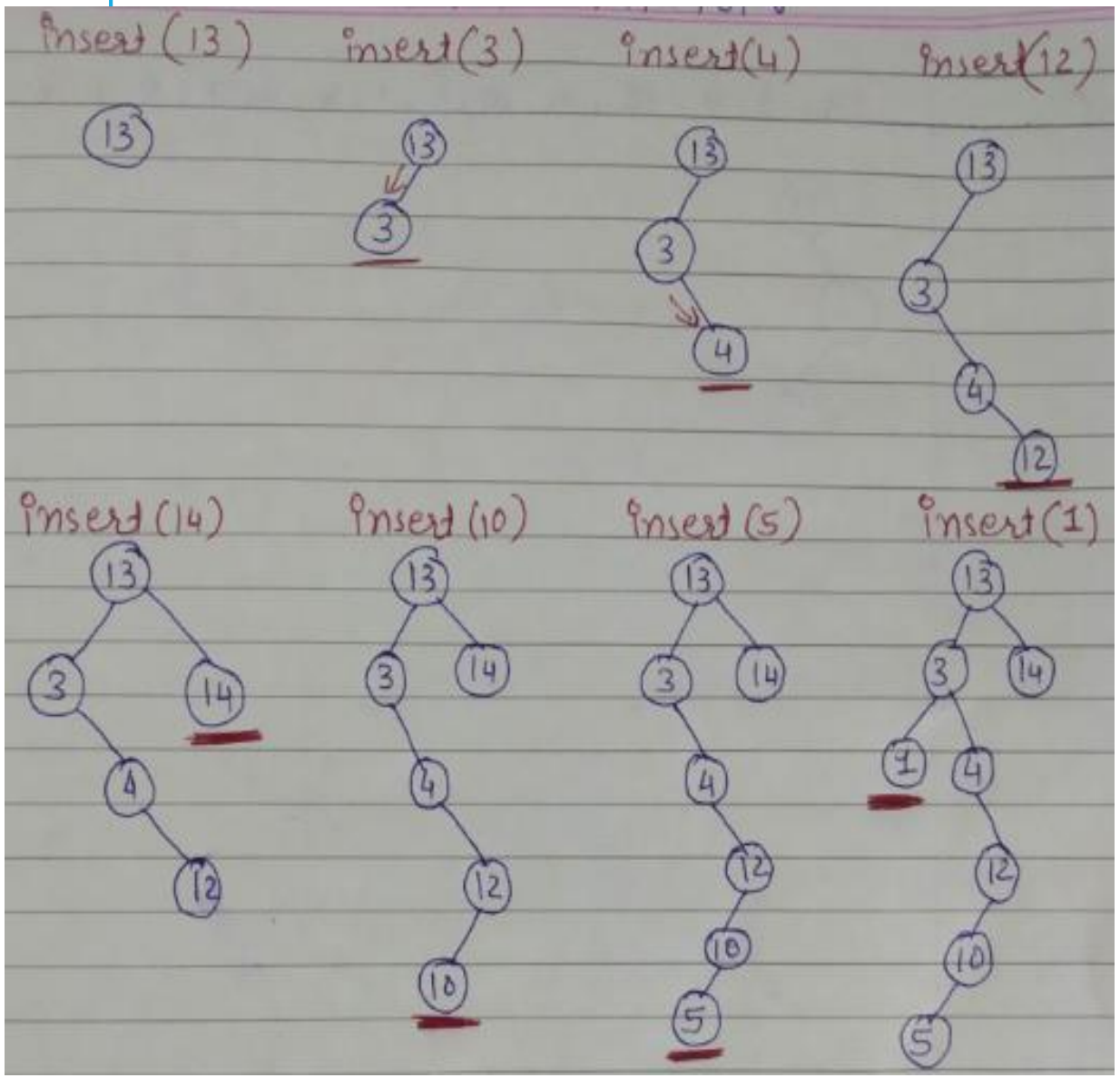
BINARY TREE	BINARY SEARCH TREE
BINARY TREE is a non linear data structure where each node can have almost two child nodes	BINARY SEARCH TREE is a node based binary tree which further has right and left subtree that too are binary search tree.
BINARY TREE is unordered hence slower in process of insertion, deletion and searching.	Insertion, deletion, searching of an element is faster in BINARY SEARCH TREE than BINARY TREE due to the ordered characteristics
IN BINARY TREE there is no ordering in terms of how the nodes are arranged	IN BINARY SEARCH TREE the left subtree has elements less than the nodes element and the right subtree has elements greater than the nodes element.

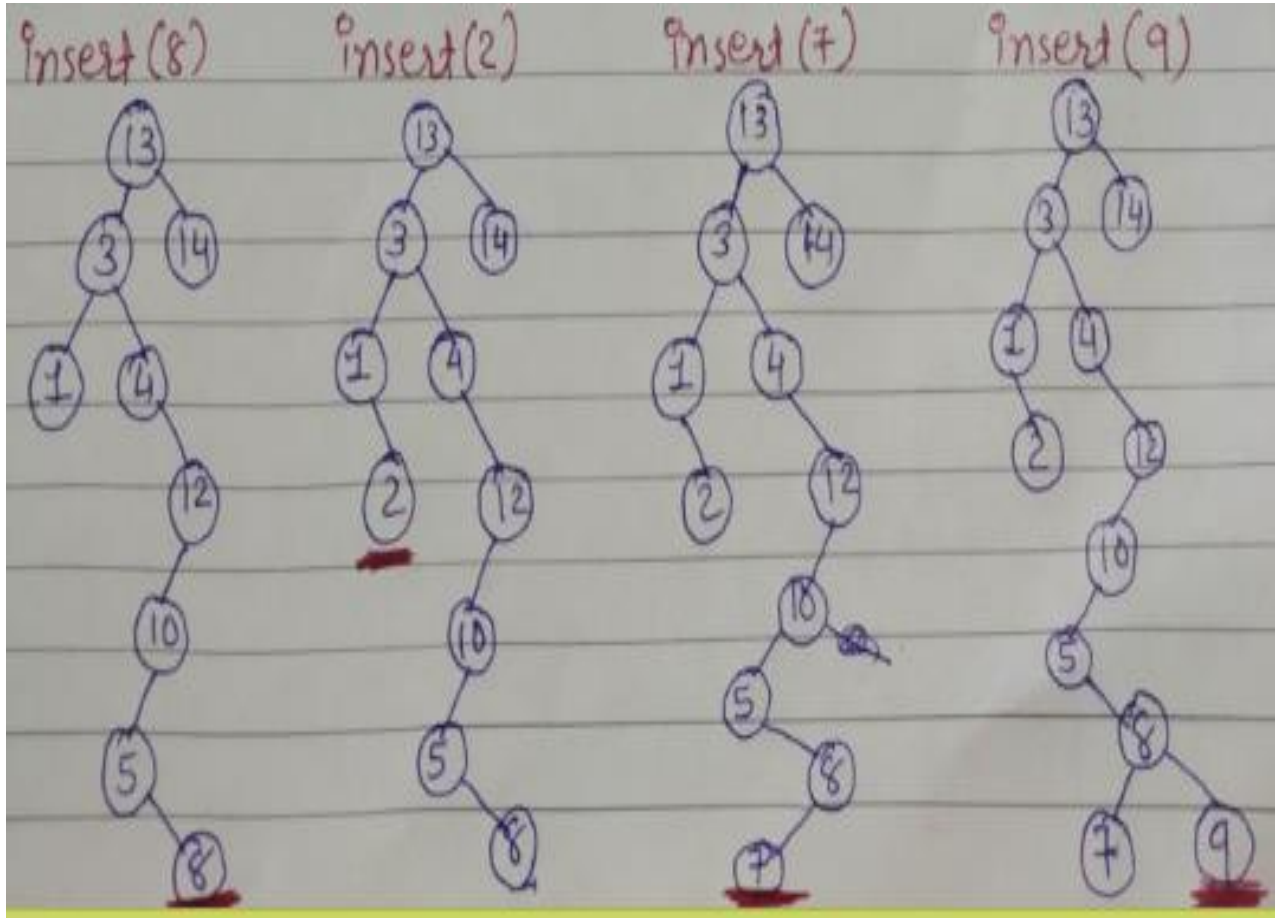
INSERTION IN BINARY SEARCH TREE

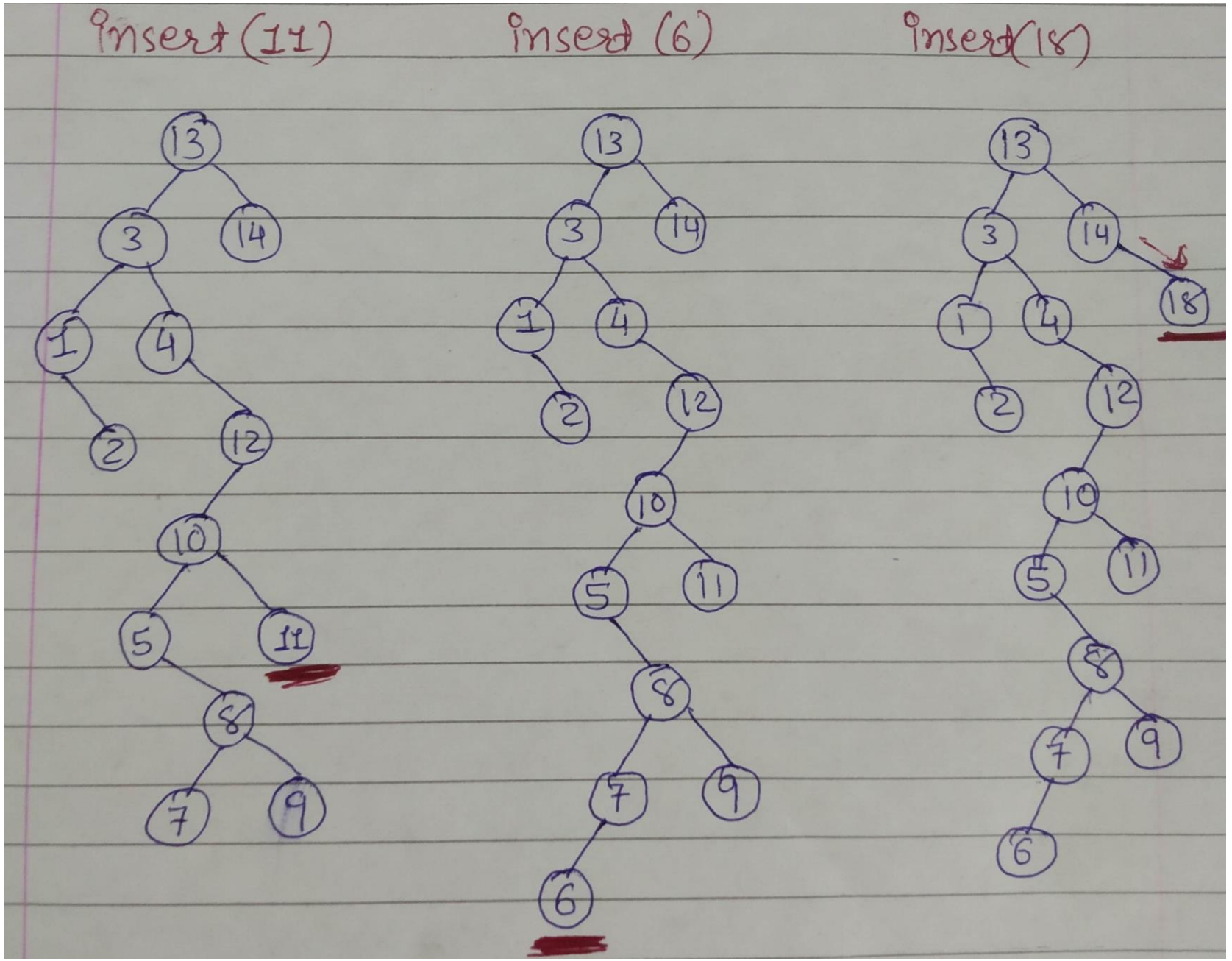
https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/BST.html#bst-insert

CREATE A BINARY SEARCH TREE FOR INSERTING THE FOLLOWING DATA.

13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18





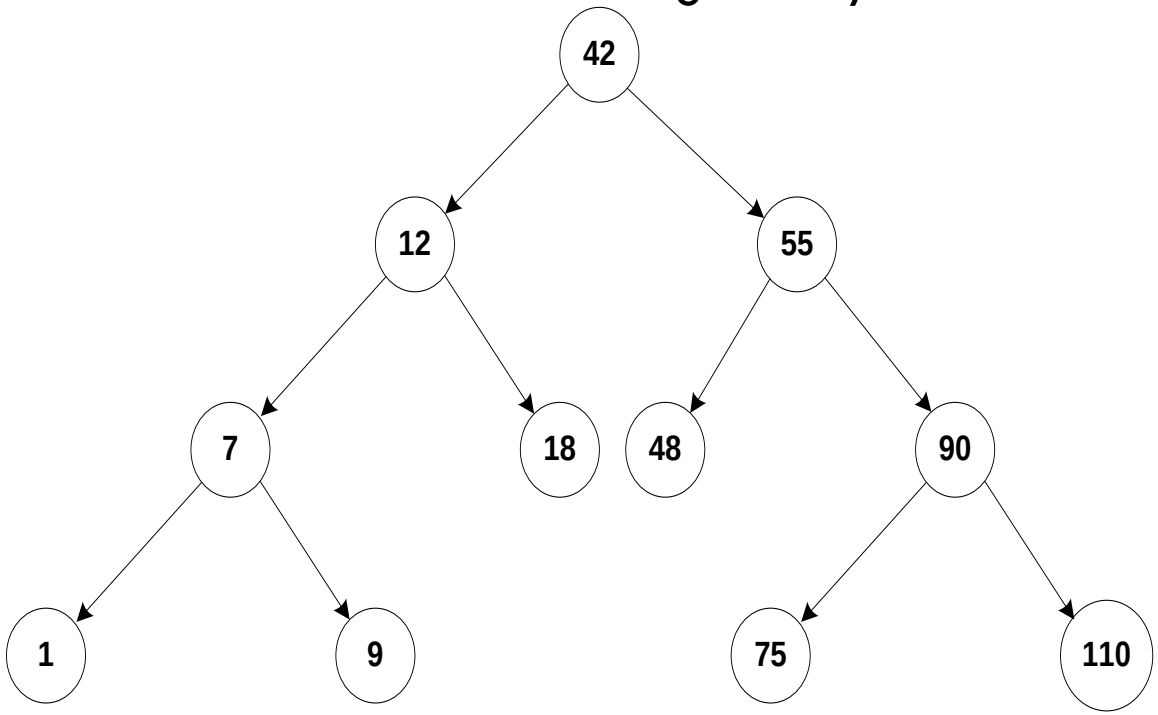

insert (11)
insert (6)
insert (18)
13
3
14
1
4
2
12
10
5
11
8
7
9
6
18

DELETION OPERATION ON BINARY SEARCH TREE

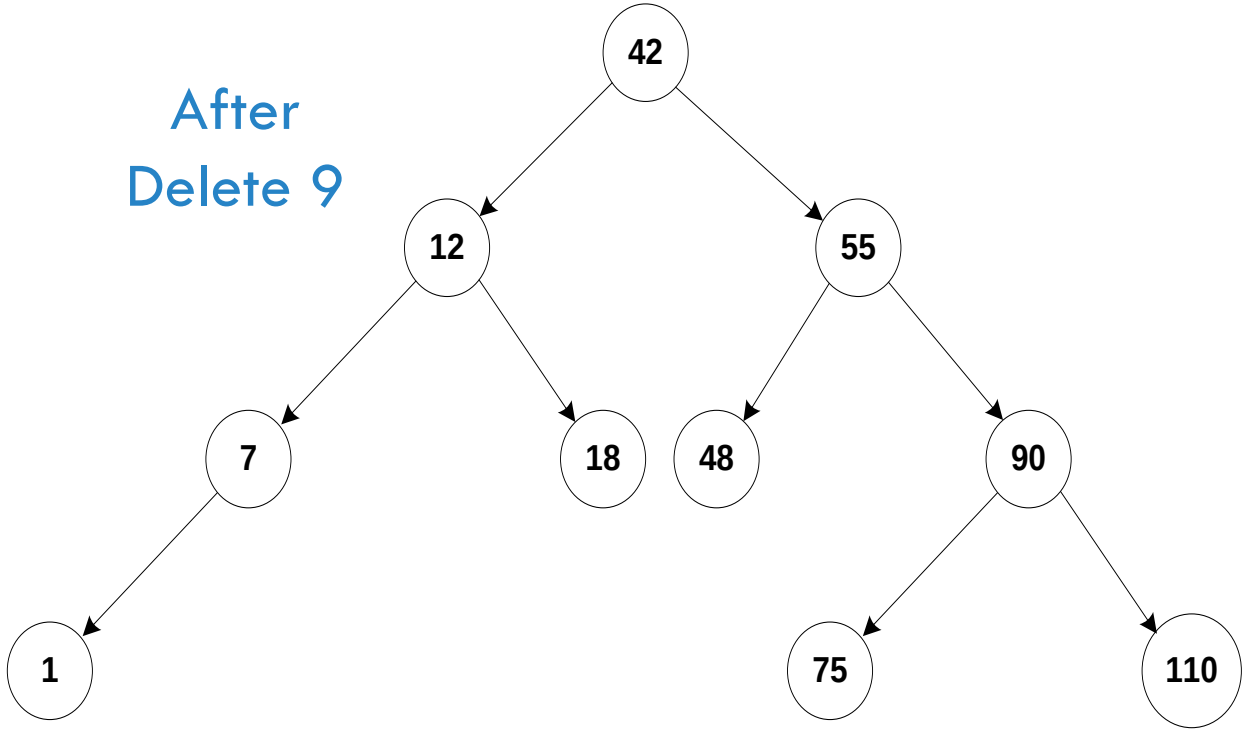
There are three possibility to delete node from tree

1) Node having 0 child : If a node to be deleted has empty left and right sub tree then a node is deleted directly

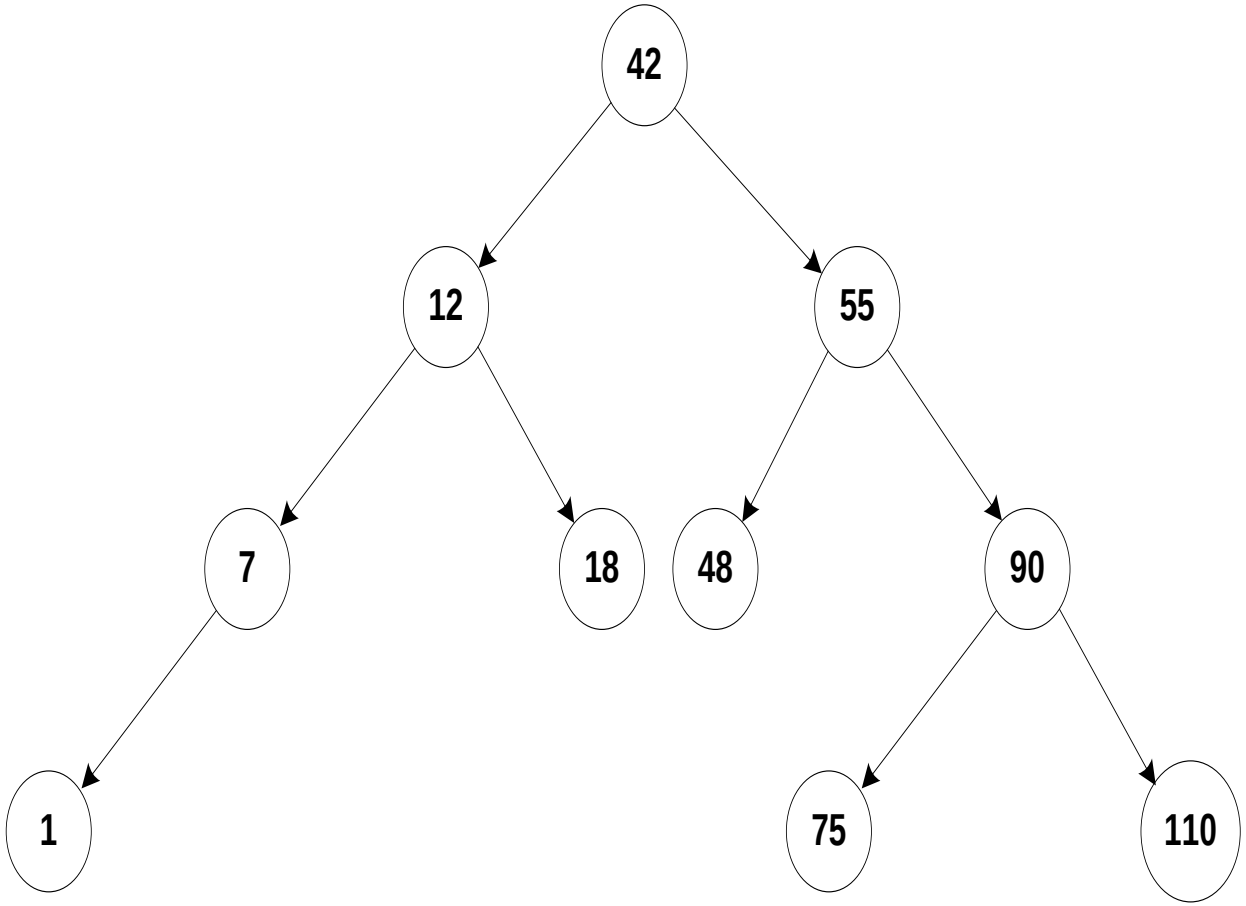
Consider the following binary search tree



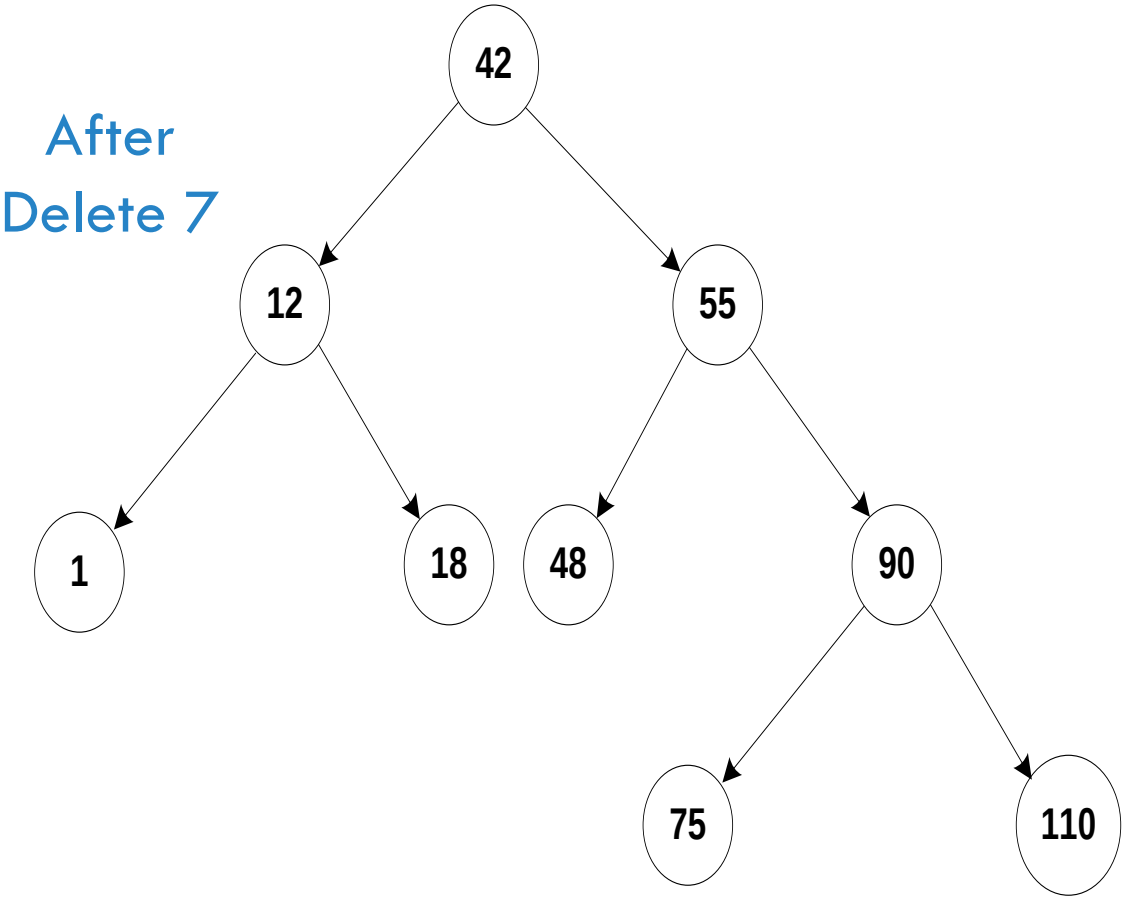
After
Delete 9



2) Node having 1 child : If a node to be deleted has only one left sub tree or right sub tree then the sub tree of the deleted node is linked directly with the parent of the deleted node.



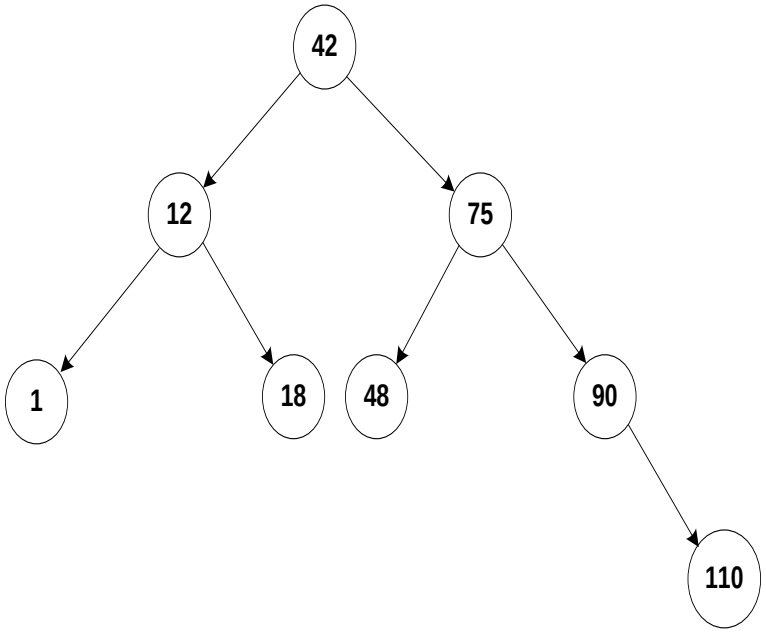
After
Delete 7



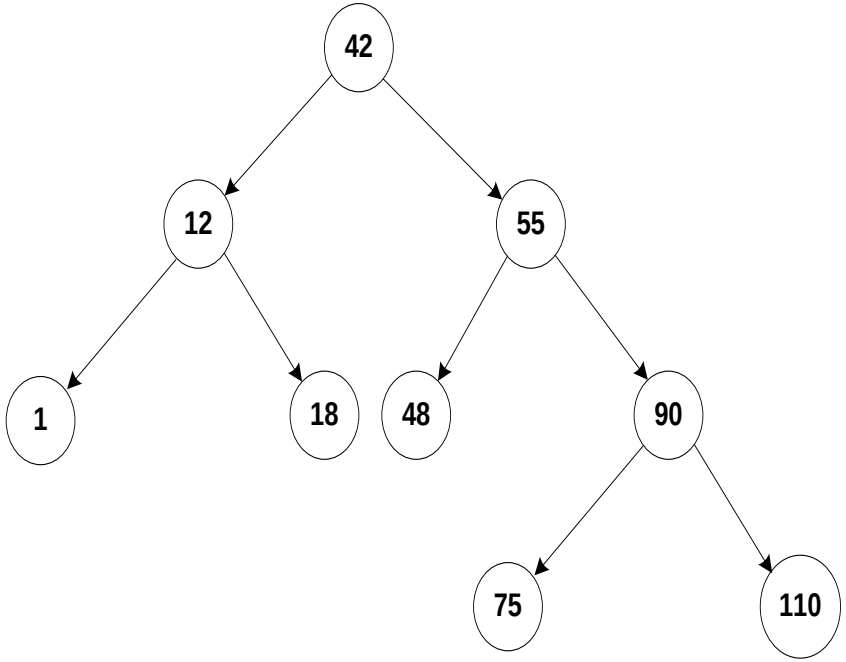
3) Node having 2 child : If a node to be deleted has left and right sub tree then we have to do following steps:

(a) In order successor : Replace next maximum of the deleted node from inorder traversal.

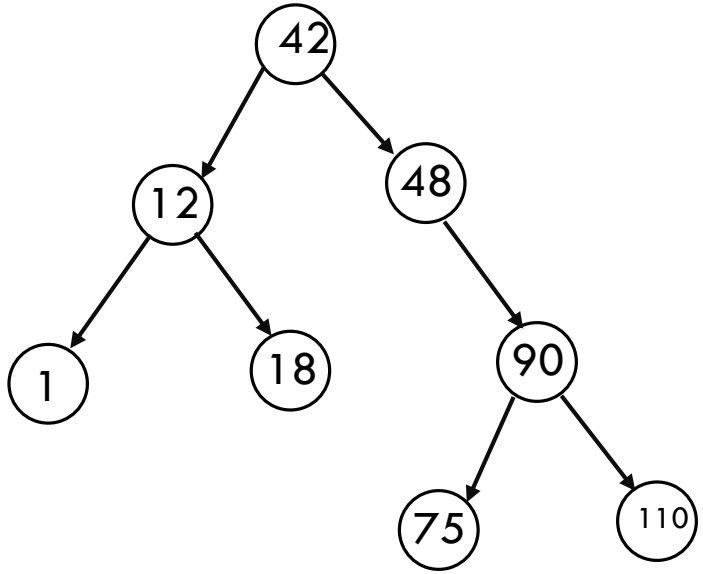
(b) In order predecessor : Replace with previous predecessor of the deleted node in inorder traversal.



In order successor



After
Delete 55

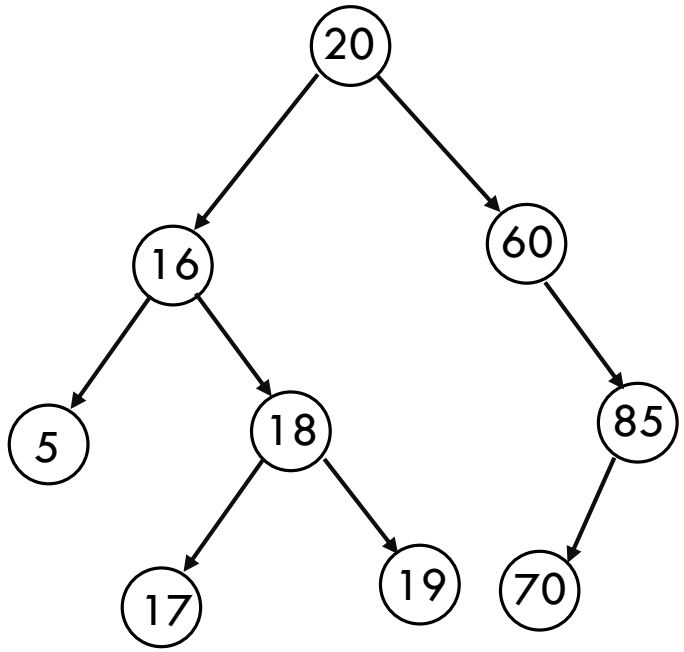


In order predecessor

CONSTRUCT BST FROM PREORDER

Preorder : 20,16,5,18,17,19,60,85,70

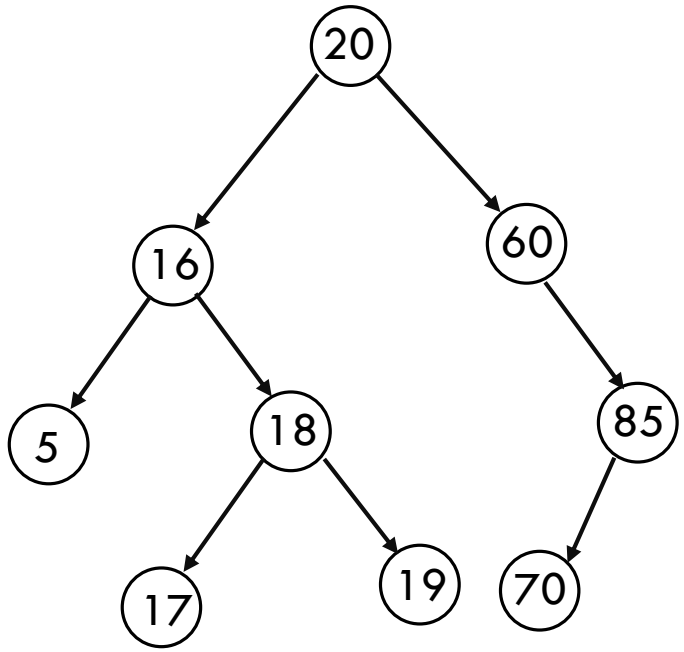
Inorder : 5,16,17,18,19,20,60,70,85



CONSTRUCT BST FROM POSTORDER

Postorder : 5,17,19,18,16,70,85,60,20

Inorder : 5,16,17,18,19,20,60,70,85



THREADED BINARY TREE

- Binary trees have a lot of wastage space , the leaf nodes each having 2 NULL pointer
- Here NULL entries can be replaced to store a pointer to the in-order predecessor or the in-order successor of the node
- These special pointers are called *threads*.
- Thread : A pointer to other node in the tree replacing Null link
- Binary trees containing threads are called *threaded trees. And this thread represent using arrow.*
- There are two type of threaded tree correspondence

    1)One way threading

    2)Two way threading

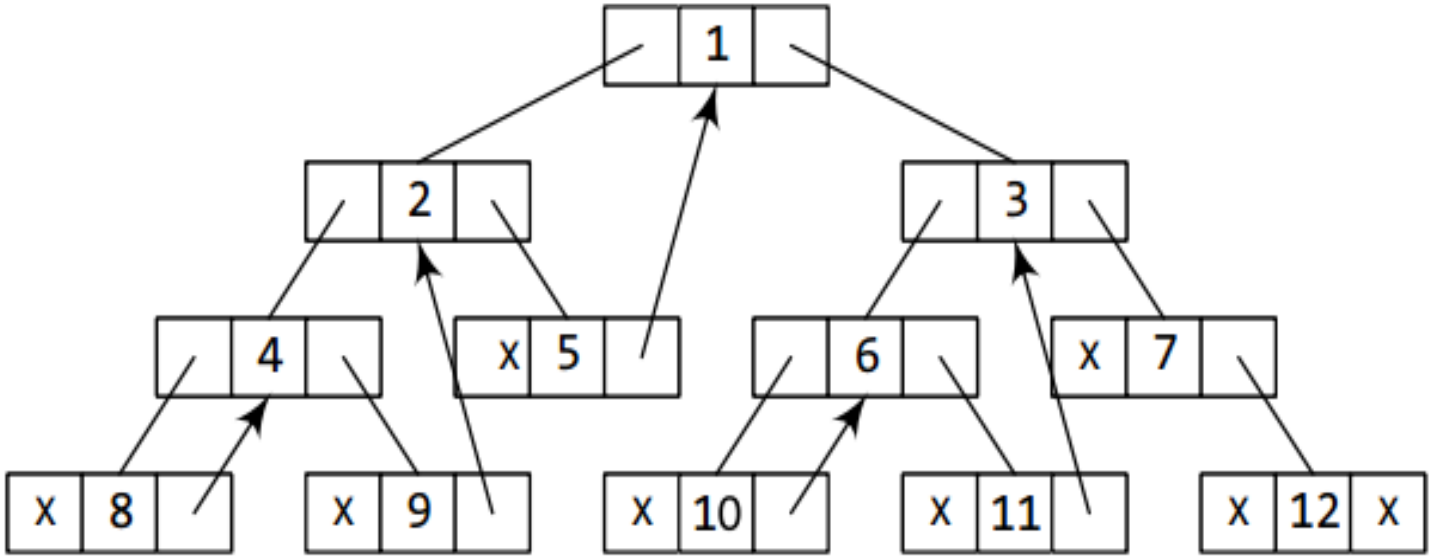
- Threaded binary trees is to make in-order traversal faster and do it without stack and without recursion

ONE WAY THREADING IN IN-ORDER TRAVERSAL

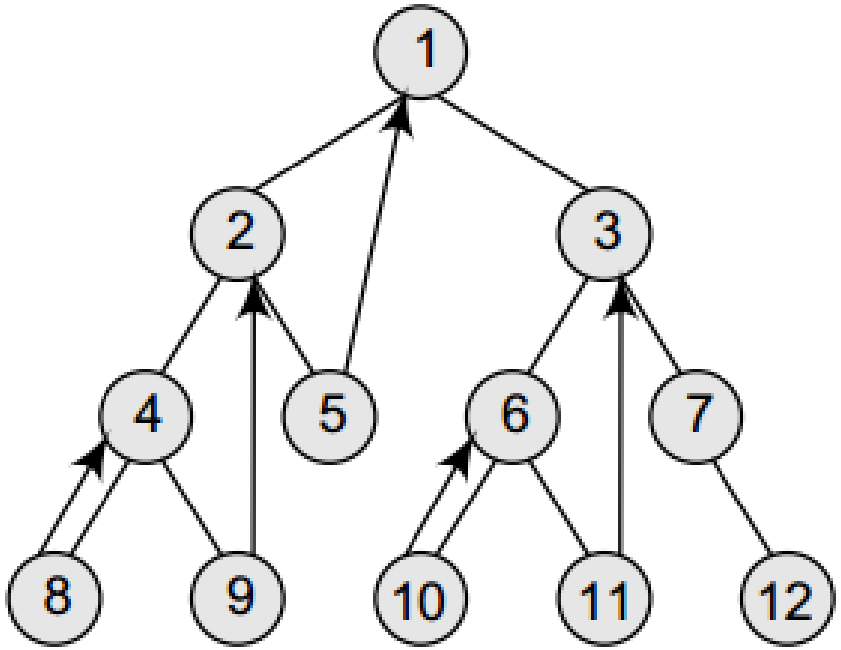
A one-way threaded tree is also called a single-threaded tree

Left-threaded binary tree- The node left field(NULL) is pointing to the in-order predecessor of the node.

Right-threaded binary tree- The node left field(NULL) is pointing to the in-order successor of the node.



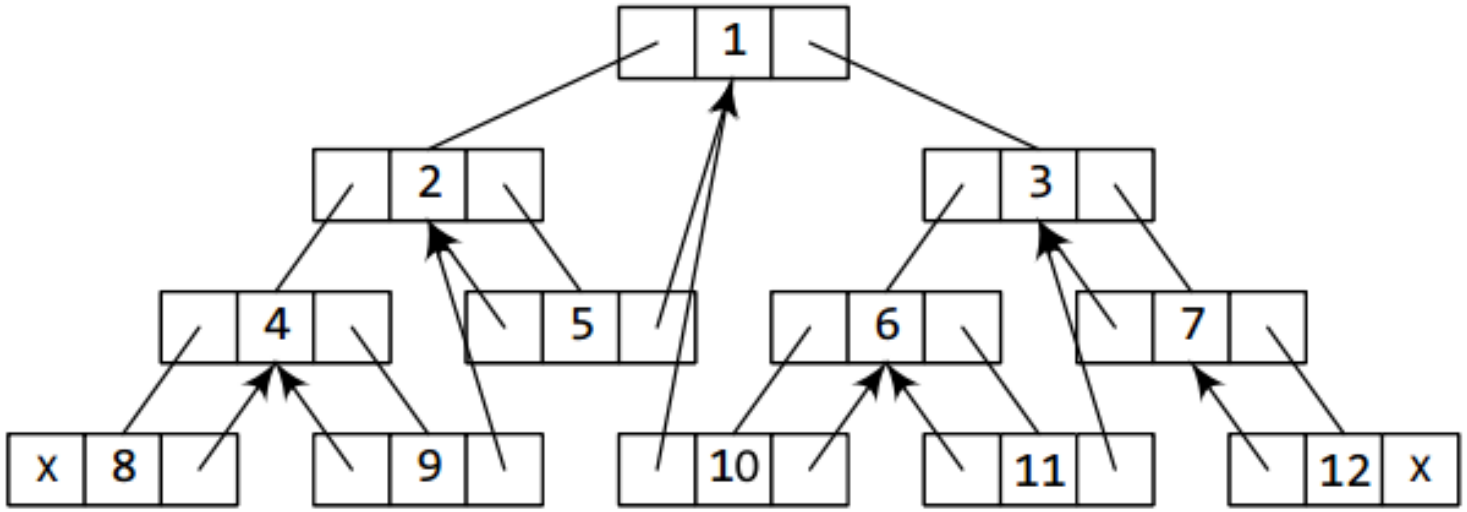
Linked representation of Right threaded binary tree



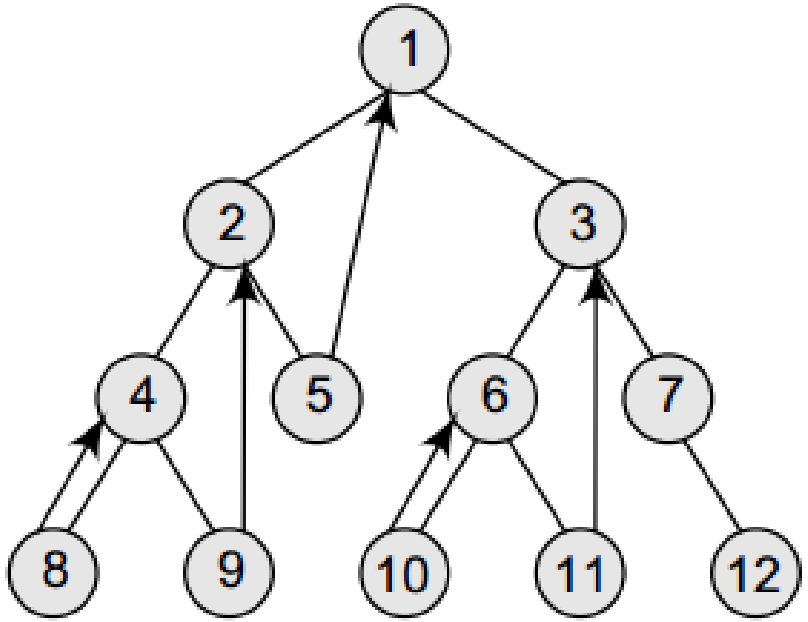
Right threaded binary tree

TWO WAY THREADING IN IN-ORDER TRAVERSAL

Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively.



Linked representation of Right threaded binary tree



Right threaded binary tree

ADVANTAGES OF THREADED BINARY TREE

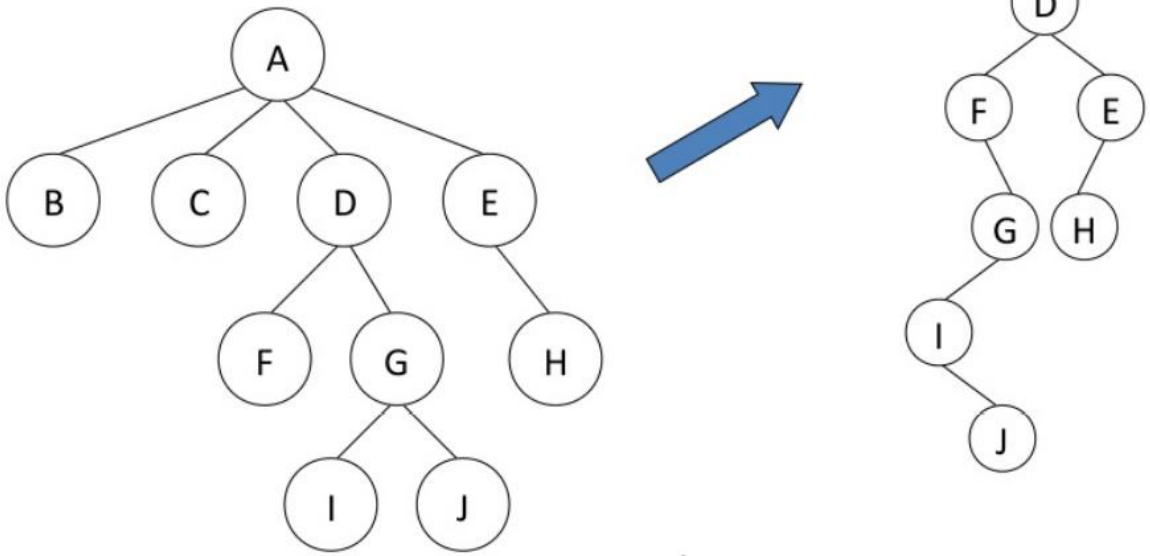
- It enables linear traversal of elements in the tree.
- Linear traversal eliminates the use of stacks which in turn consume a lot of memory space and computer time.
- It enables to find the parent of a given element without explicit use of parent pointers.
- Since nodes contain pointers to in-order predecessor and successor, the threaded tree enables forward and backward traversal of the nodes as given by in-order fashion.

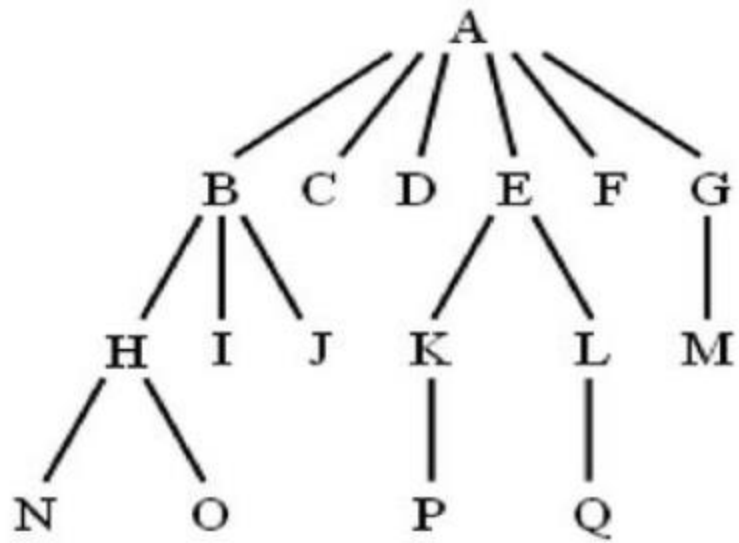
CONVERSION OF GENERAL TREES TO BINARY TREES

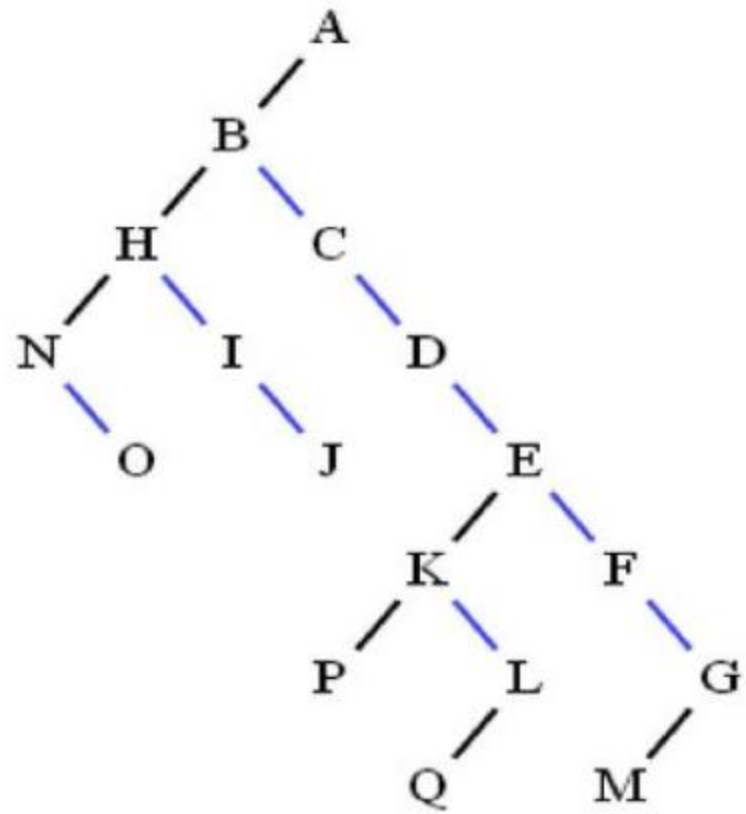
❖General tree (m array tree) or forest of tree can be converted into binary tree.

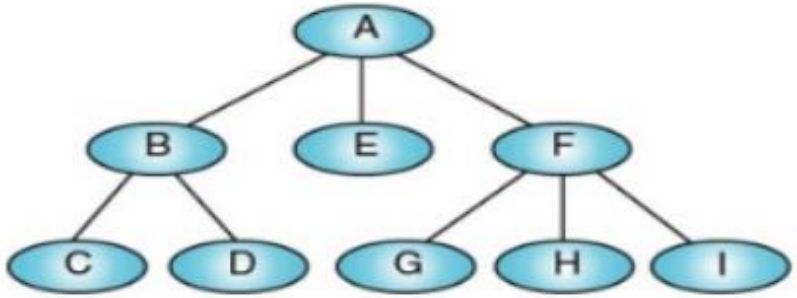
❖In this algorithm we must connect a parent to its left offspring and connect from left to right all the sibling at the same level within the same tree

- Binary tree left child
  = leftmost child
- Binary tree right child
  = right sibling

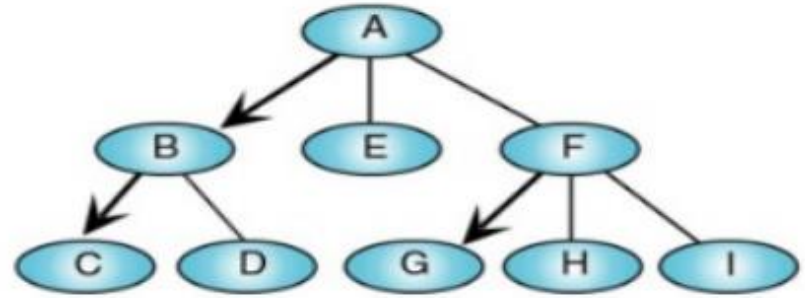



A
B C D E F G
H I J K L M
N O P Q

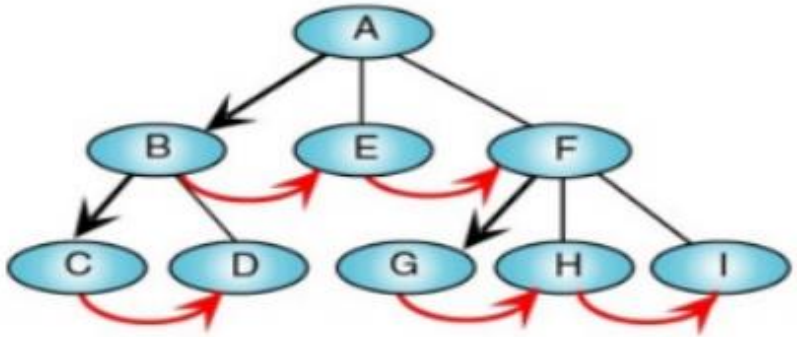

A
B
H C
N I D
O J E
K F
P L G
Q M



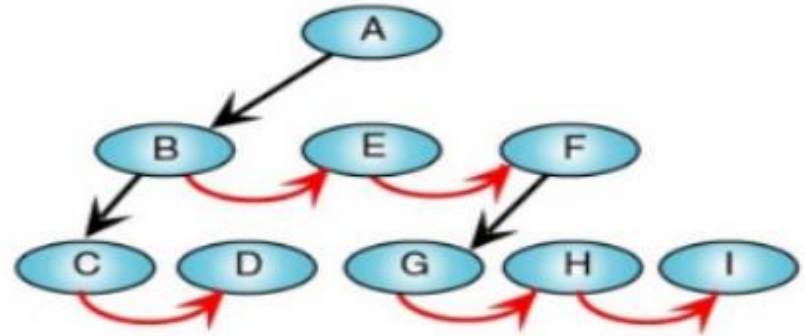
(a) The general tree



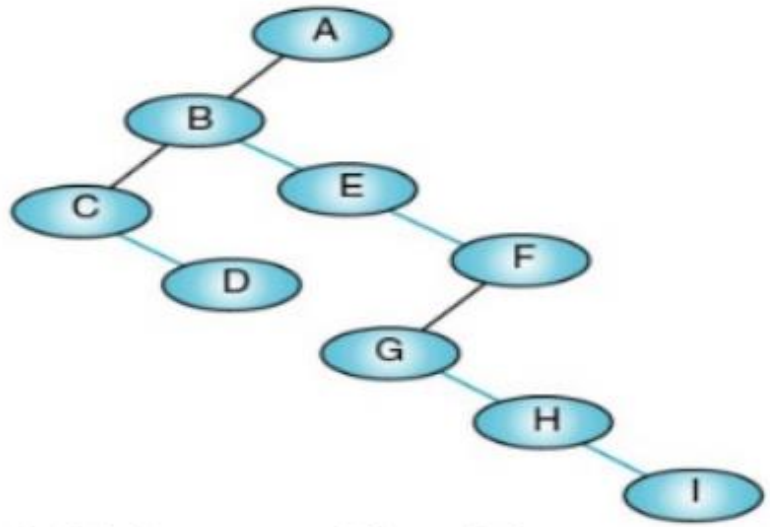
(b) Identify leftmost children



(c) Connect siblings



(d) Delete unneeded branches



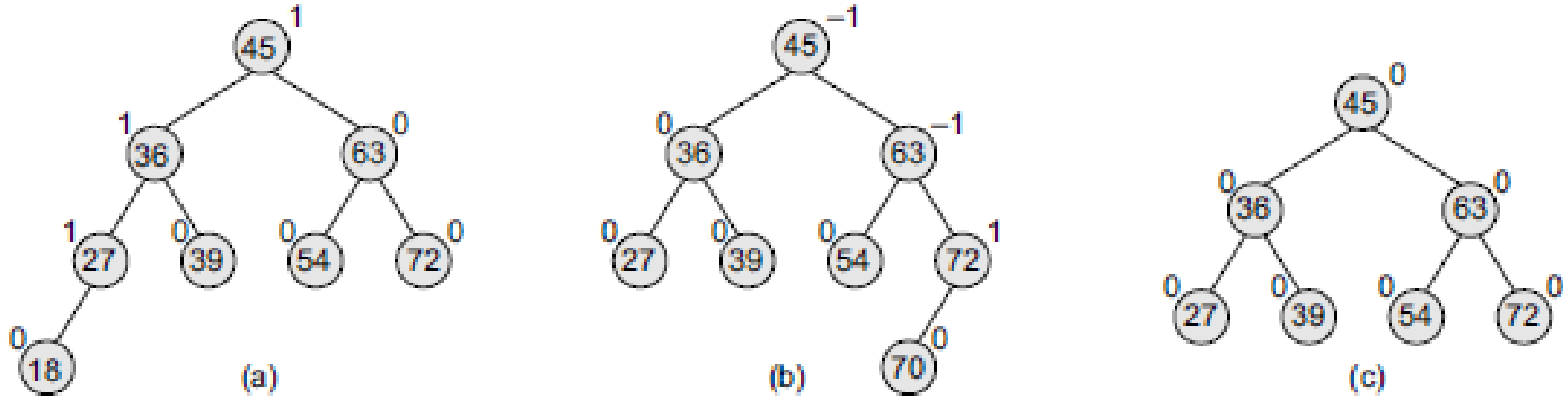
(e) The resulting binary tree

Height Balance tree: Height balance tree (self balancing tree) is a binary tree which automatically maintain height of tree, and its sub tree on each insertion and deletion of node. AVL tree, red-black tree are example of height balanced tree.

Weight Balance tree: A weight-balanced tree is a binary tree in which for each node. The number of nodes in the left sub tree is at least half and at most twice the number of nodes in the right sub tree.

AVL TREE

- **AVL tree** is a self-balancing Binary Search **Tree** (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.
- AVL tree is also known as a height-balanced tree.
- it takes O(log n) time to perform search, insert, and delete operations in an average case as well as the worst case
- AVL tree is the same as that of a binary search tree but its structure having an additional variable called the BalanceFactor
- *Balance factor = Height (left sub-tree) – Height (right sub-tree)*



(a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

- *Balance factor = Height (left sub-tree) – Height (right sub-tree)*
  - higher than that of the right sub-tree. Such a tree is therefore called as a *left-heavy tree*.
  - If the balance factor of a node is 0, then it means that the height of the left sub-tree (longest path in the left sub-tree) is equal to the height of the right sub-tree.
  - If the balance factor of a node is –1, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is therefore called as a *right-heavy tree*.

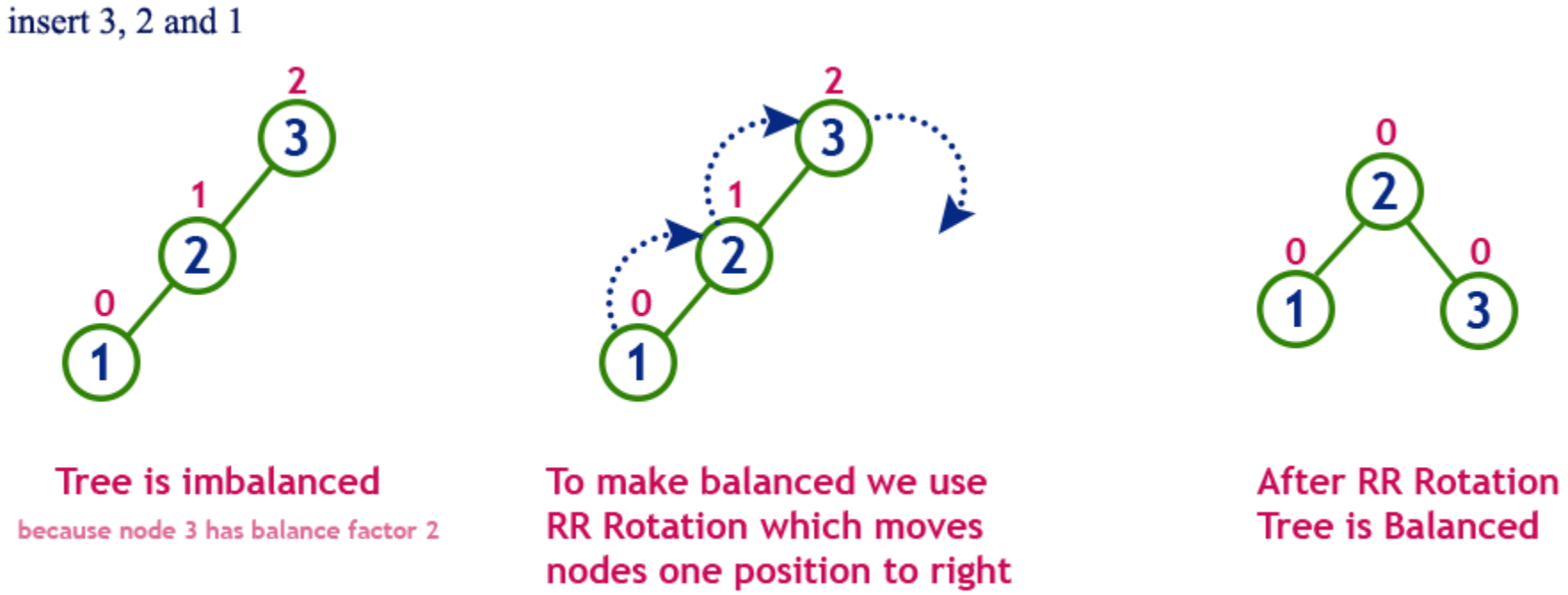
OPERATION ON AVL TREE

- Insertions and deletions from an AVL tree may disturb the balance factor of the nodes
- The tree is rebalanced by performing rotation at the critical node.
- AVL tree may perform the following four kinds of rotations
  - Left Left rotation
  - Right Right rotation
  - Left-Right rotation
  - Right-Left rotation

LEFT LEFT ROTATION

This rotation is performed when a new node is inserted at the left child of the left subtree.

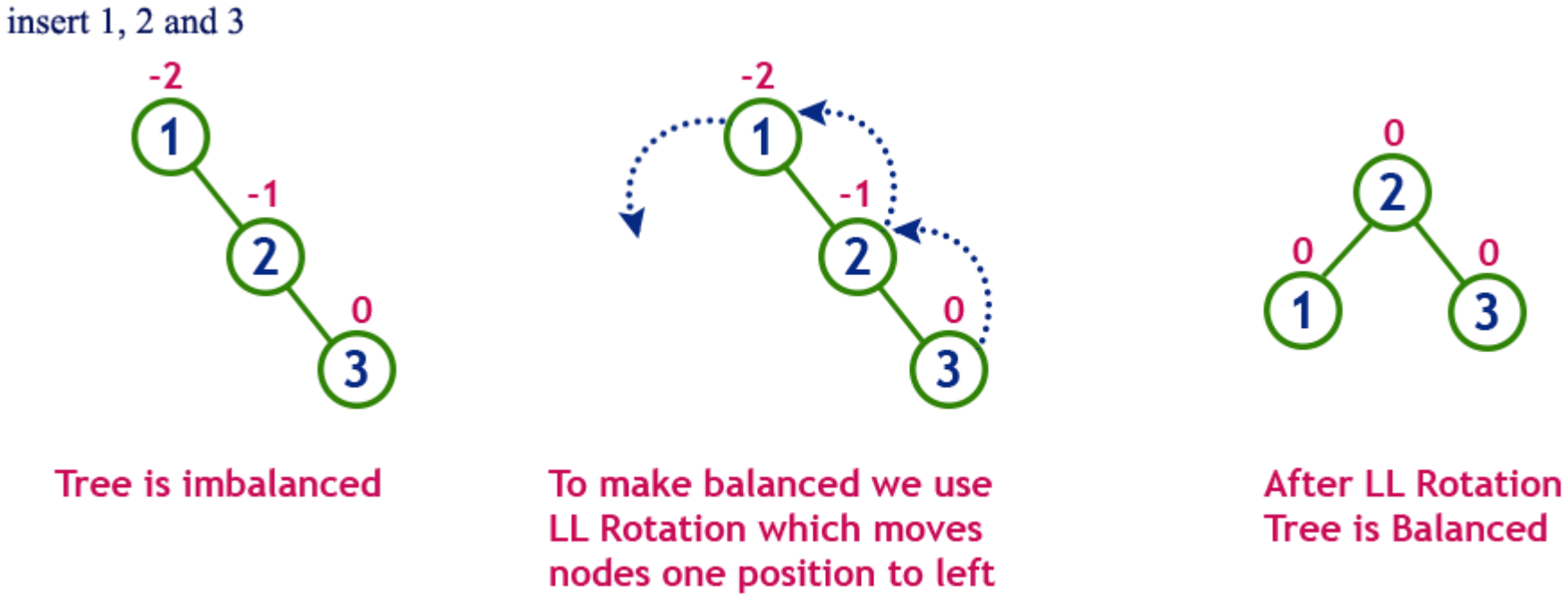
A single right rotation is performed. This type of rotation is identified when a node has a balanced factor as +2, and its left-child has a balance factor as +1.



RIGHT RIGHT ROTATION

This rotation is performed when a new node is inserted at the right child of the right subtree.

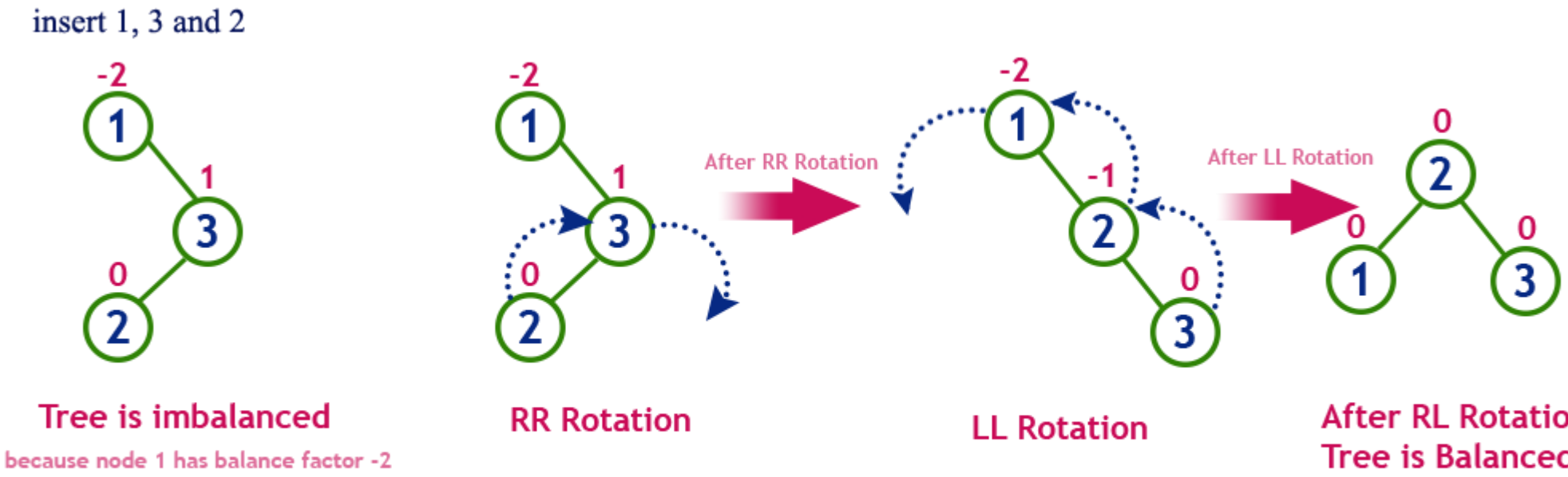
A single left rotation is performed. This type of rotation is identified when a node has a balanced factor as -2, and its right-child has a balance factor as -1.



RIGHT LEFT ROTATION

This rotation is performed when a new node is inserted at the left child of the right subtree.

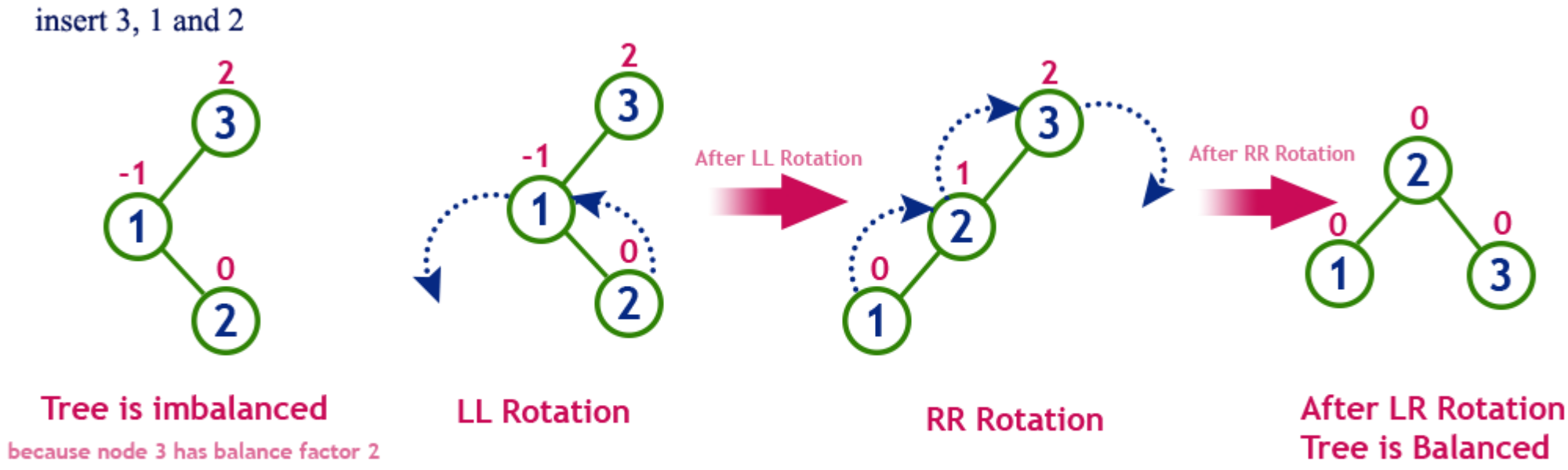
This rotation is performed when a node has a balance factor as –2, and its right-child has a balance factor as +1.



LEFT RIGHT ROTATION

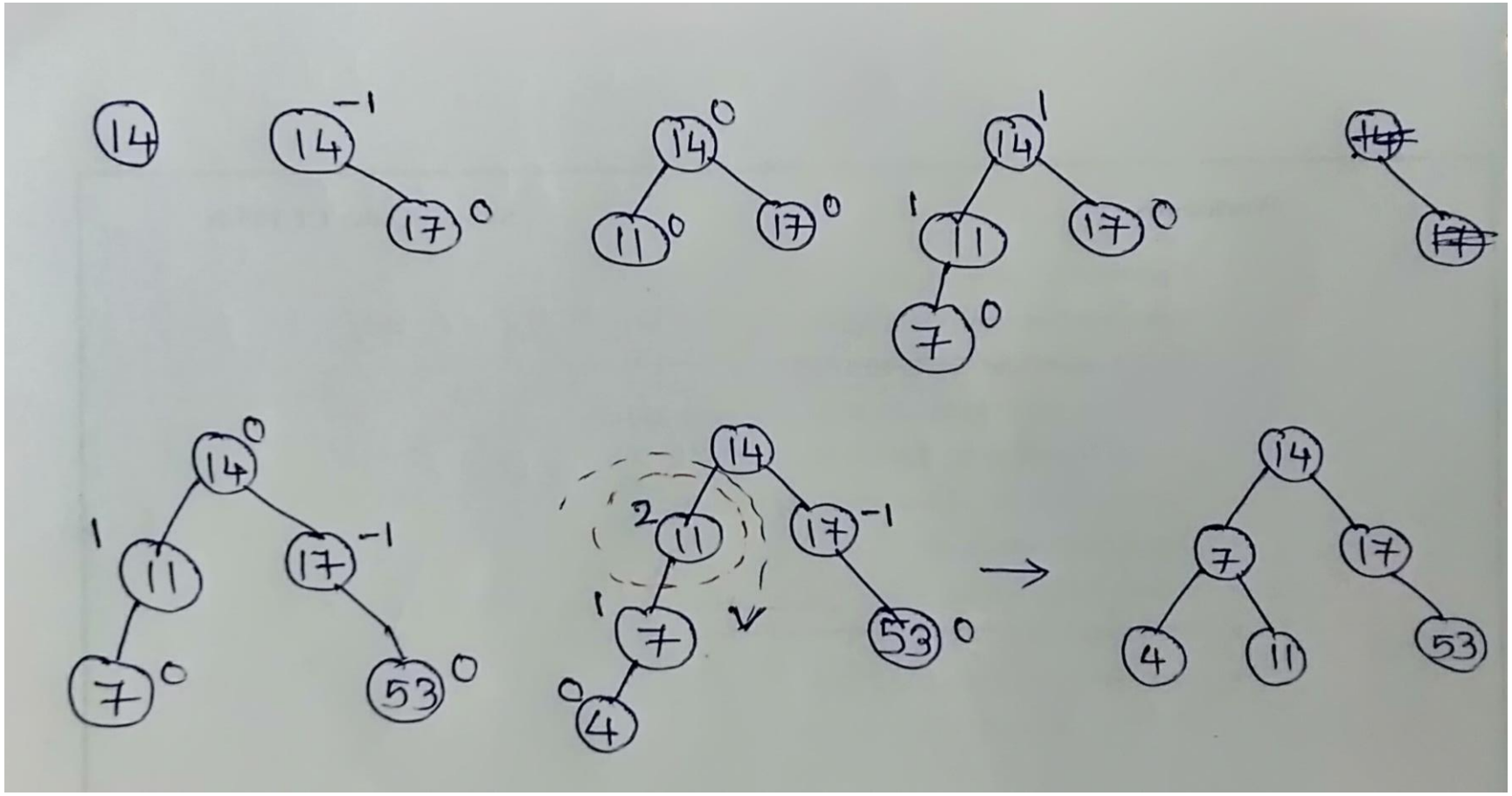
This rotation is performed when a new node is inserted at the right child of the left subtree.

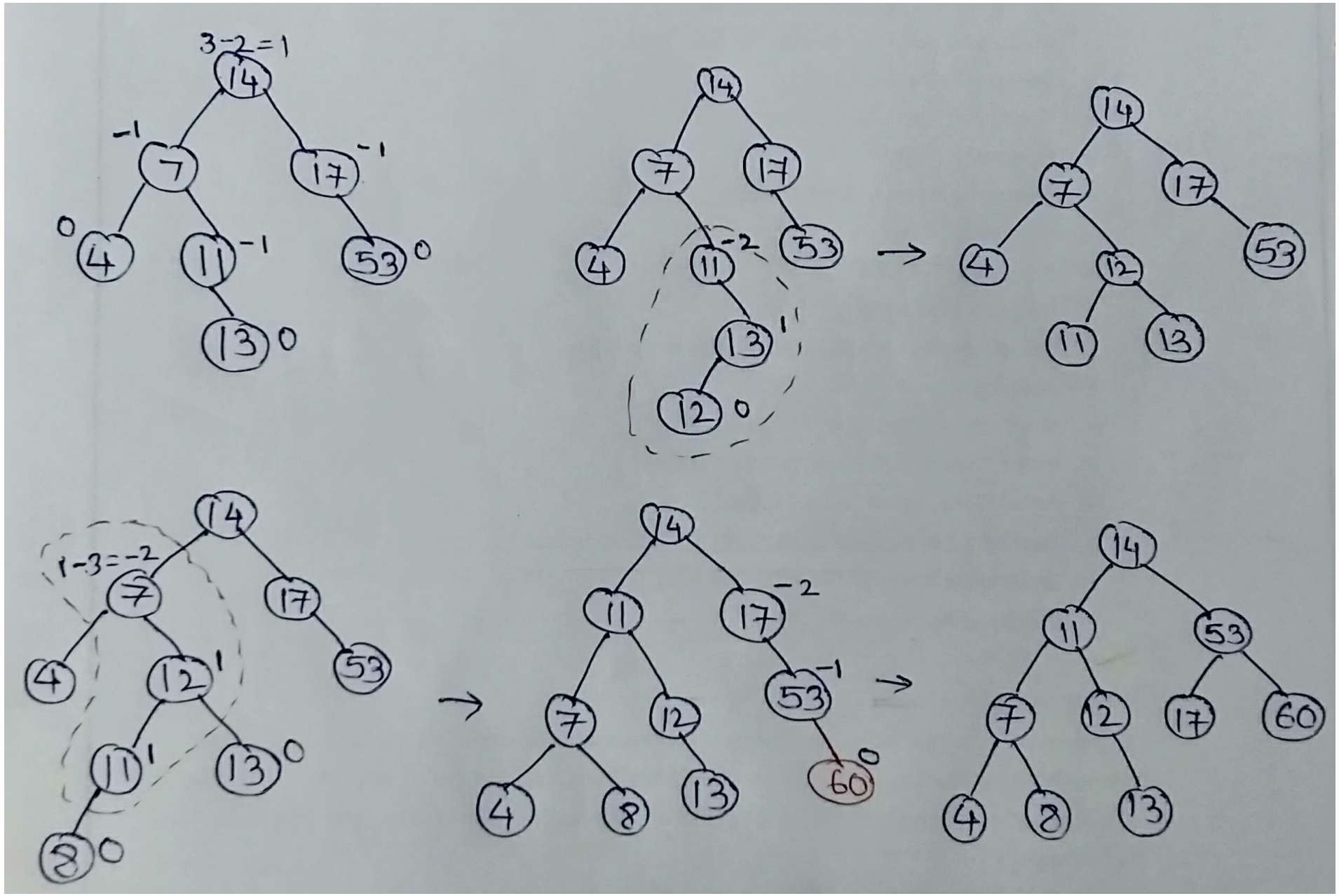
This rotation is performed when a node has a balance factor as +2, and its right-child has a balance factor as -1.

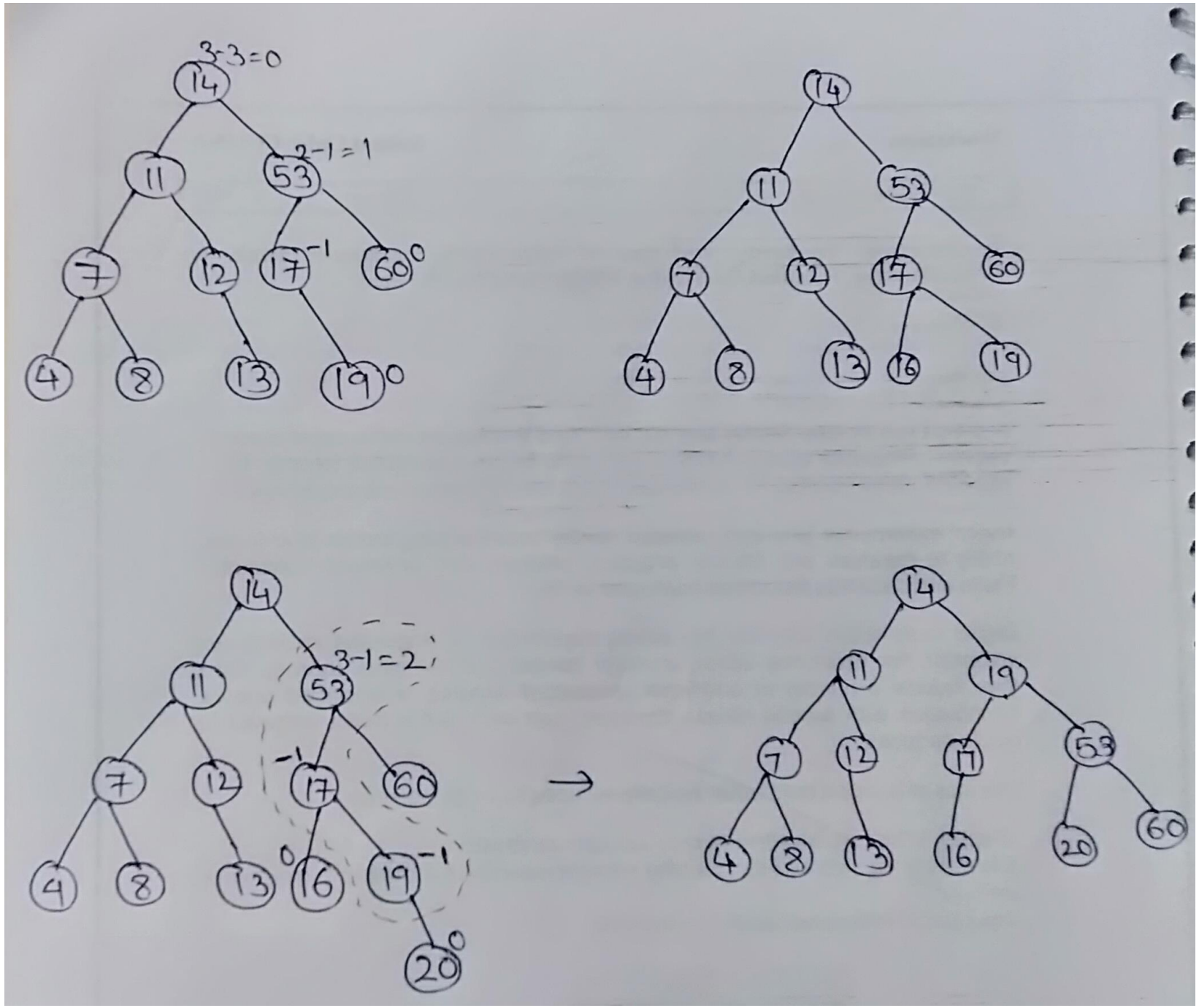


INSERTION IN AVL

Construct AVL tree by inserting the following data :
14,17,11,7,53,4,13,12,8,60,19,16,20




3-2=1
14
7
17
4
11
53
13
12
1-3=-2
8
60


3-3=0
14
11
53
2-1=1
7
12
17
-1
60
0
4
8
13
19
0
14
11
53
7
12
17
60
4
8
13
16
19
14
11
53
3-1=2
7
12
-1
17
60
4
8
13
0
16
19
-1
20
0
14
11
19
7
12
17
53
4
8
13
16
20
60

DELETION IN AVL

The deletion operation in AVL Tree is similar to deletion operation in BST.

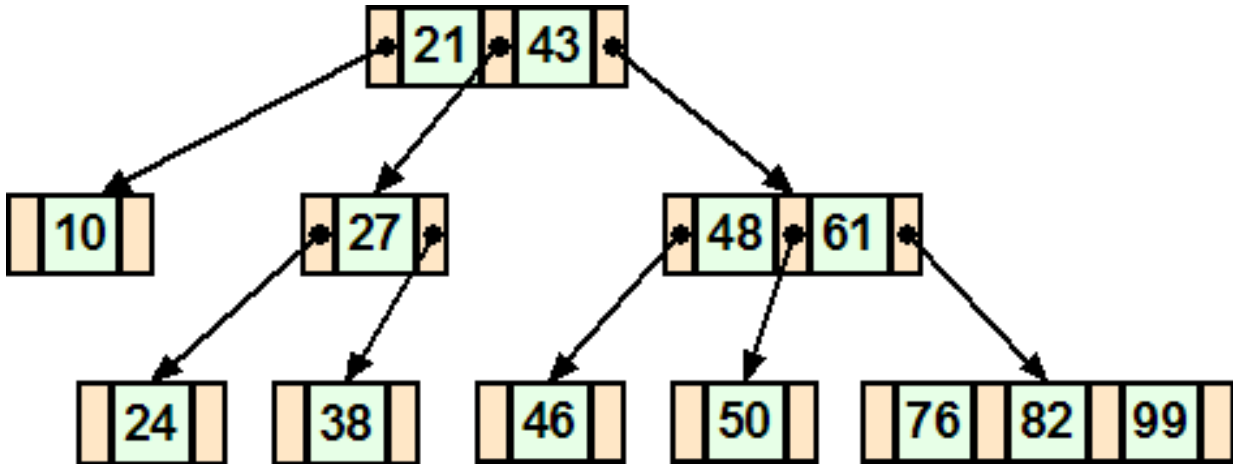
After every deletion operation, we need to check with the Balance Factor condition.

If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

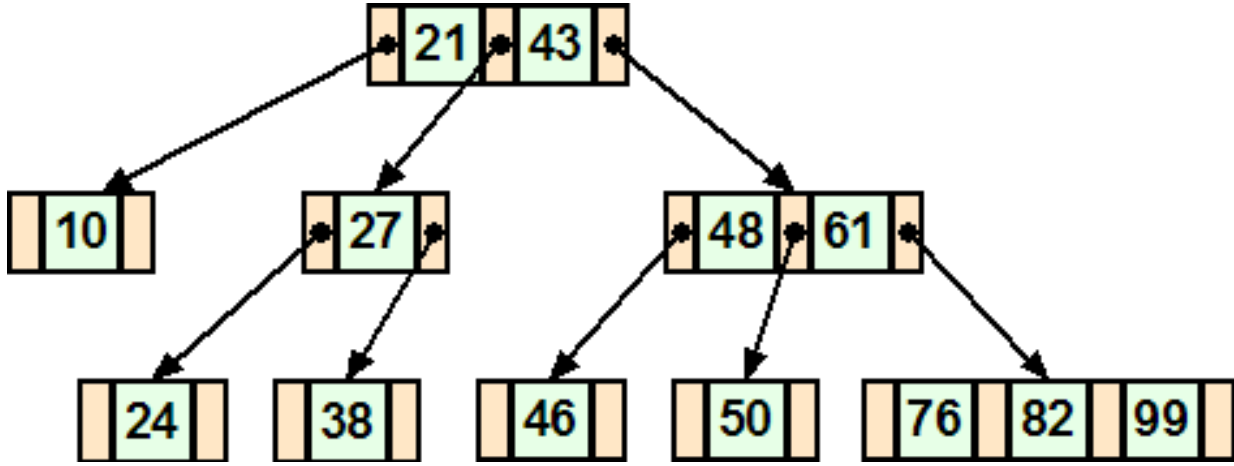
2-3 TREES

2-3 tree obeys the following definition:

1. A node contains one or two keys.
2. Every internal node has either two children (if it contains one key) or three children (if it contains two keys).
    1. 2-3 tree is a tree data structure in which every internal node (non-leaf node) has either one data element and two children or two data elements and three children.
3. All leaves are at the same level in the tree, so the tree is always height balanced.
4. Time complexity 0(log(n))



SEARCH IN 2-3 TREE



1. If **T** is empty, return False (key cannot be found in the tree).
2. If current node contains data value which is equal to **K**, return True.
3. If we reach the leaf-node and it doesn't contain the required key value **K**, return False.

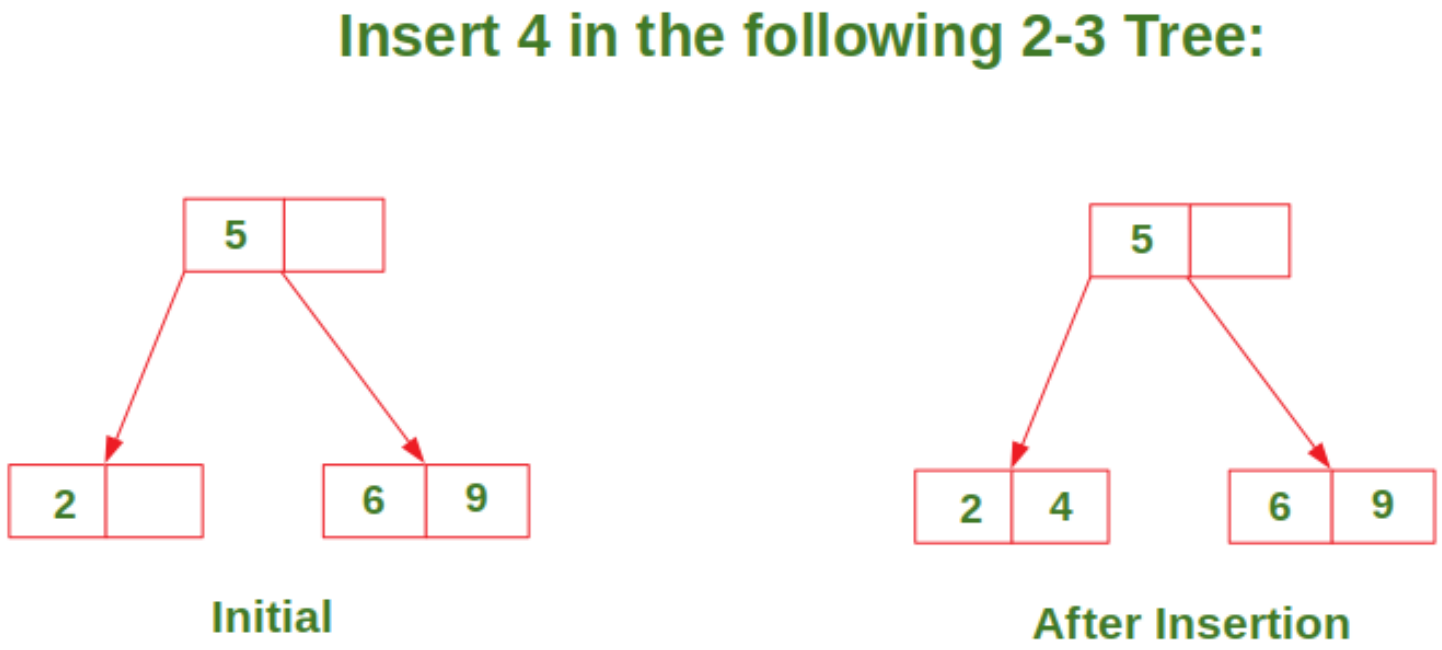
Recursion call:

1. If **K** < currentNode.leftVal, we explore the left subtree of the current node.
2. Else if currentNode.leftVal < **K** < currentNode.rightVal, we explore the middle subtree of the current node.
3. Else if **K** > currentNode.rightVal, we explore the right subtree of the current node

https://cs221viz.netlify.app/src/btree

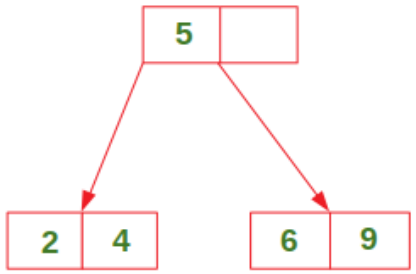
INSERTION

Case 1: Insert in a node with only one data element

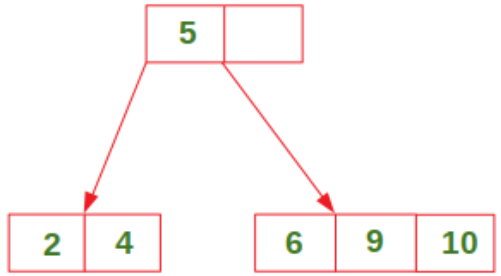


Case 2: Insert in a node with two data elements whose parent contains only one data element.

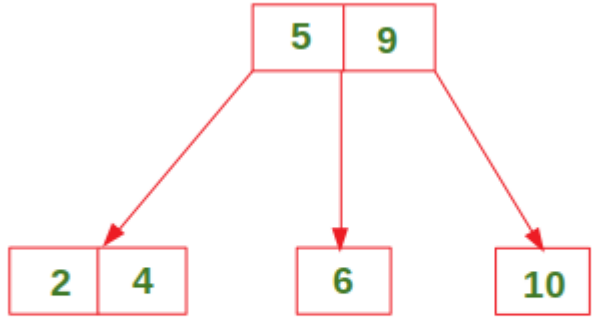
Insert 10 in the following 2-3 Tree:



Initial



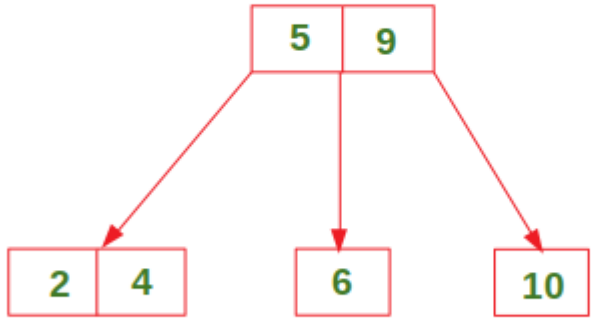
Temporary Node with 3 data elements



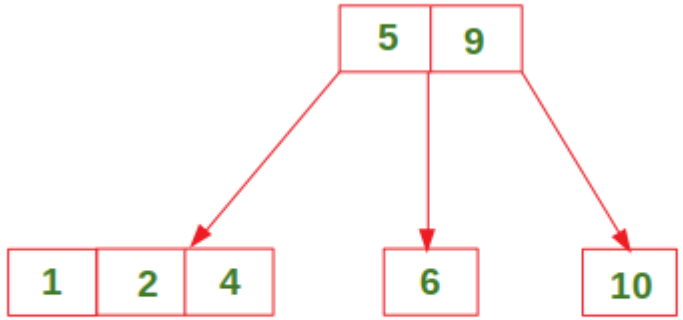
Move the middle element to parent and split the current Node

Case 3: Insert in a node with two data elements whose parent also contains two data elements.

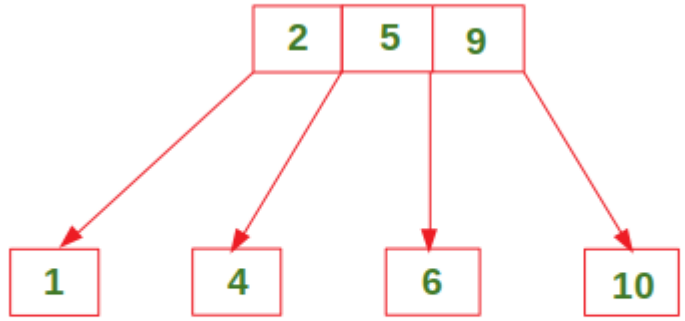
Insert 1 in the following 2-3 Tree:



Initial

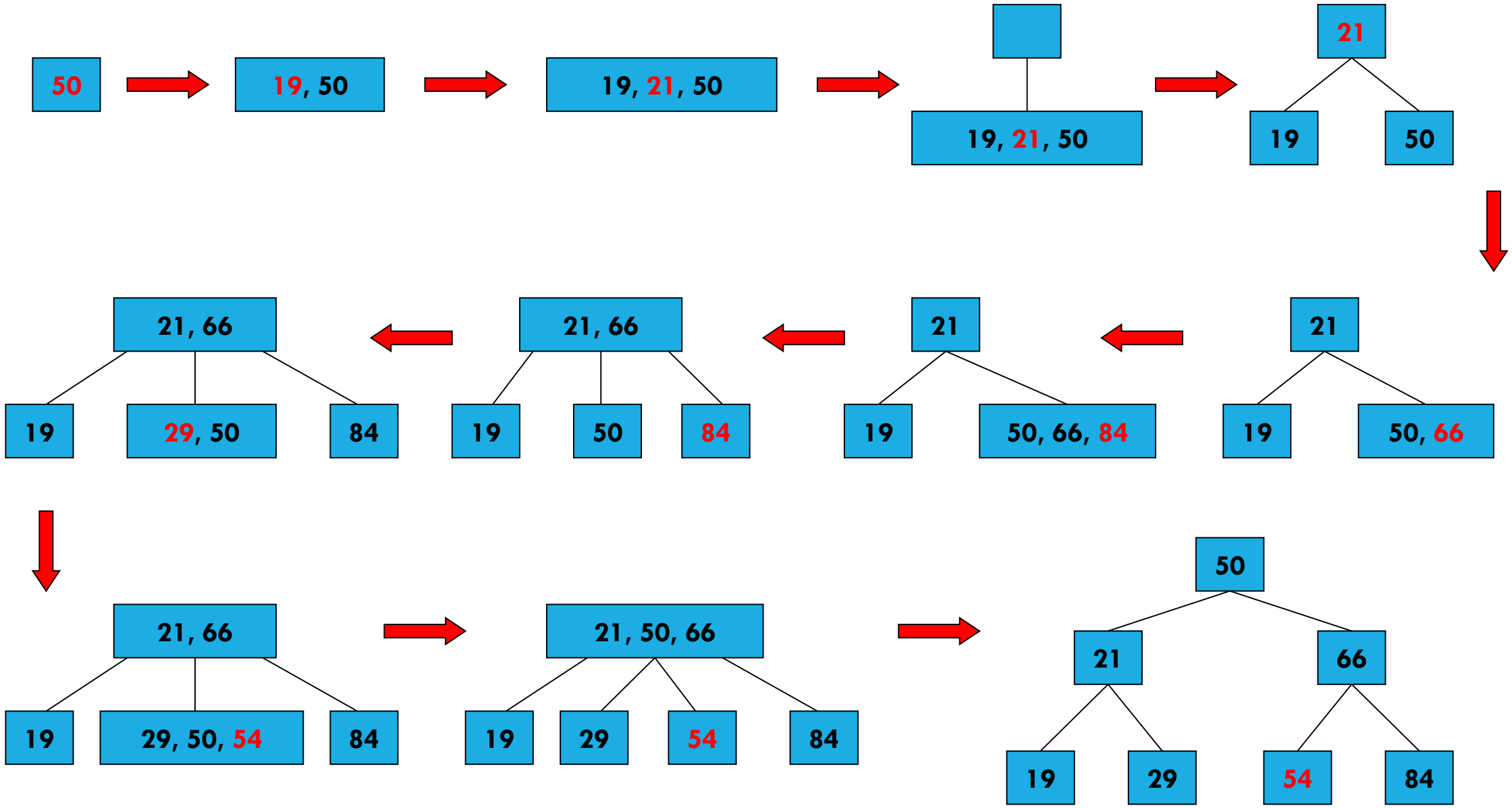


Temporary Node with 3 data elements



Move the middle element to the parent and split the current Node

Addition: 50, 19, 21, 66, 84, 29, and 54.



B-TREE

If order m = 5
Min node = (m/2) = 3
Max node = 5
Min keys = (ceiling)([m-1]/2) = 2
Max key = 4

It is also known as a height-balanced m-way tree.

B-tree of order m is defined to have the following properties:

1. Balanced m-way tree
2. Generalization of BST in which a node can have more then one key and more then two children.
3. It maintain sorted data.
4. All leaves are at the same level.
5. A B-Tree is defined by the term *minimum degree* 'm'.
6. Every node except root must contain at least (ceiling)([m-1]/2) keys. The root may contain minimum 1 key.
7. All nodes (including root) may contain at most m – 1 keys.
8. Number of children of a node is equal to the number of keys in it plus 1.
9. All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
10. Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(log n).

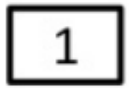
Create a B-tree of order 5 by inserting value from 1 to 20.

https://cs221viz.netlify.app/src/btree

Construct the B-Tree of order 4 for the following elements

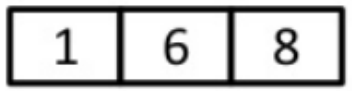
1, 6, 8, 2, 9, 12, 15, 7, 18, 3, 4, 20

Insert 1

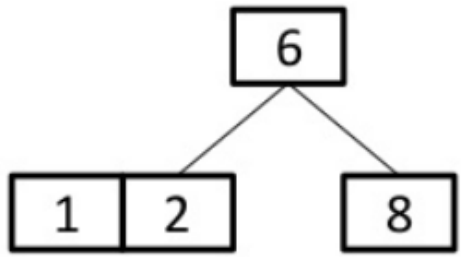


Insert 6

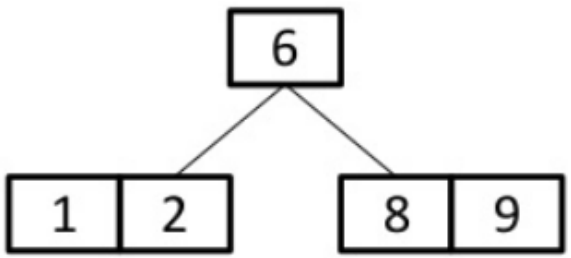
Insert 8



Insert 2



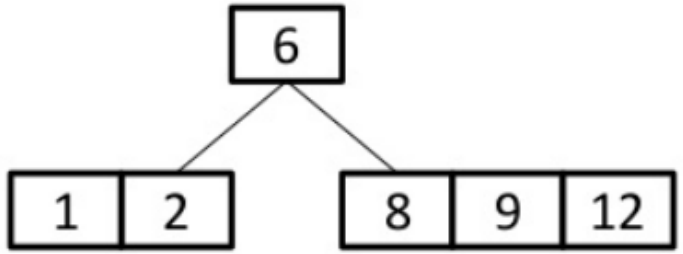
Insert 9



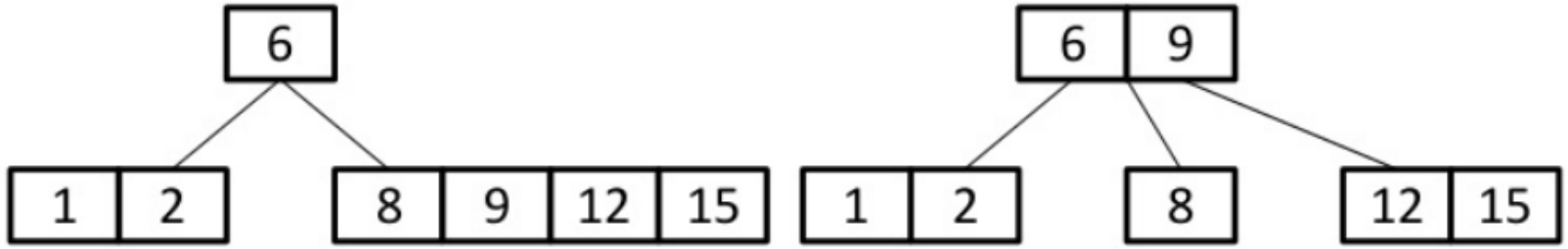
Construct the B-Tree of order 4 for the following elements

1, 6, 8, 2, 9, 12, 15, 7, 18, 3, 4, 20

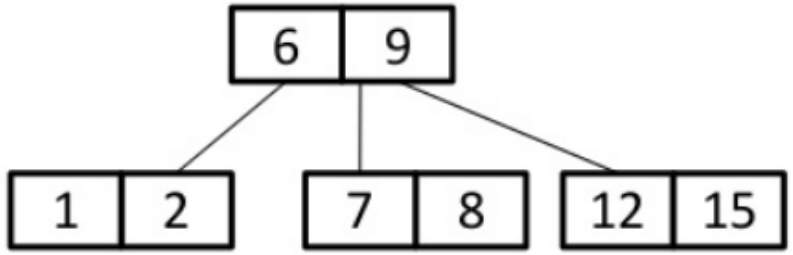
Insert 12



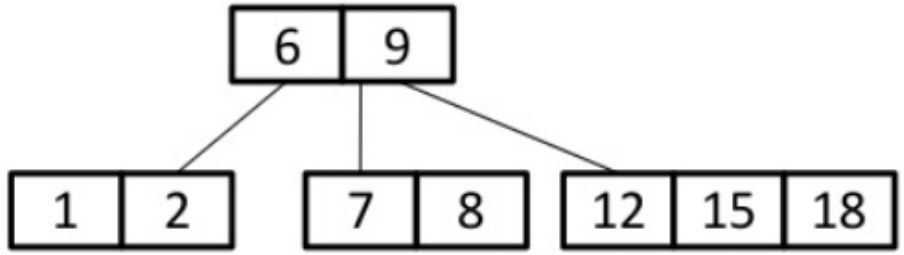
Insert 15



Insert 7



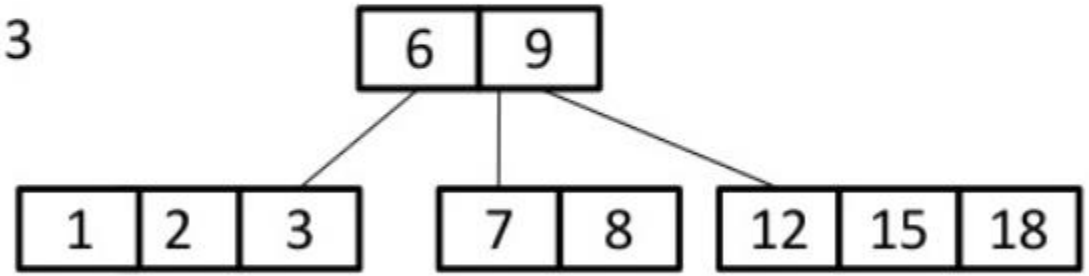
Insert 18



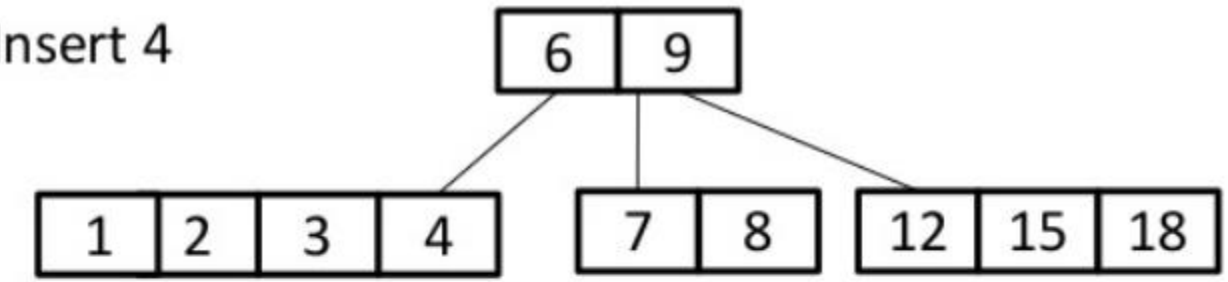
Construct the B-Tree of order 4 for the following elements

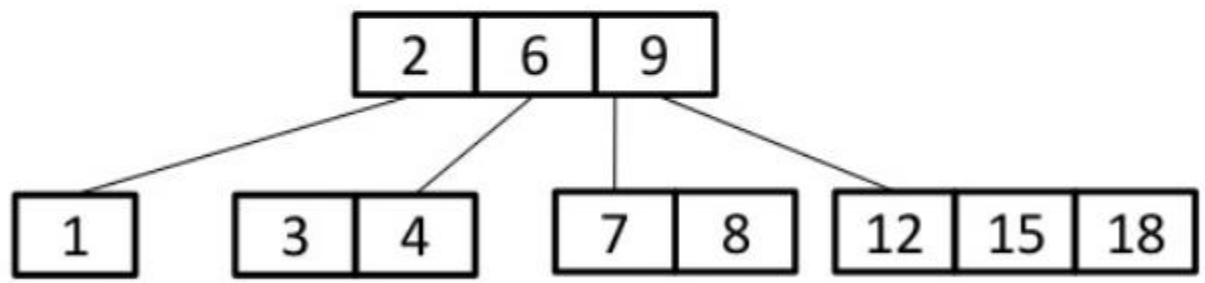
1, 6, 8, 2, 9, 12, 15, 7, 18, 3, 4, 20

Insert 3



Insert 4





APPLICATIONS OF TREES

- Trees are used to store simple as well as complex data. Here simple means an integer value, character value and complex data means a structure or a record.
- Trees are often used for implementing other types of data structures like hash tables, sets, and maps.
- A self-balancing tree, Red-black tree is used in kernel scheduling, to preempt massively multiprocessor computer operating system use. (We will study red-black trees in next chapter.)
- Another variation of tree, B-trees are prominently used to store tree structures on disc. They are used to index a large number of records
- B-trees are also used for secondary indexes in databases, where the index facilitates a select operation to answer some range criteria.
- Trees are an important data structure used for compiler construction.
- Trees are also used in database design.
- Trees are used in file system directories.
- Trees are also widely used for information storage and retrieval in symbol tables.

ASSIGNMENT

1. Construct Binary tree from preorder and Inorder.

Preorder : 1,2,4,8,9,10,11,5,3,6,7

Inorder : 8,4,10,9,11,2,5,1,6,3,7

2. Construct Binary tree from preorder and Inorder.

Preorder : P A S T Q E D X M R C F

Inorder : T S Q A E D P M X C R F

3. Construct Binary tree from postorder and Inorder.

Postorder : 9,1,2,12,7,5,3,11,4,8

Inorder : 9,5,1,7,2,12,8,4,3,11

4. Construct Binary tree from postorder and Inorder.

Postorder : T Q S D E A M C F R X P

Inorder : T S Q A E D P M X C R F

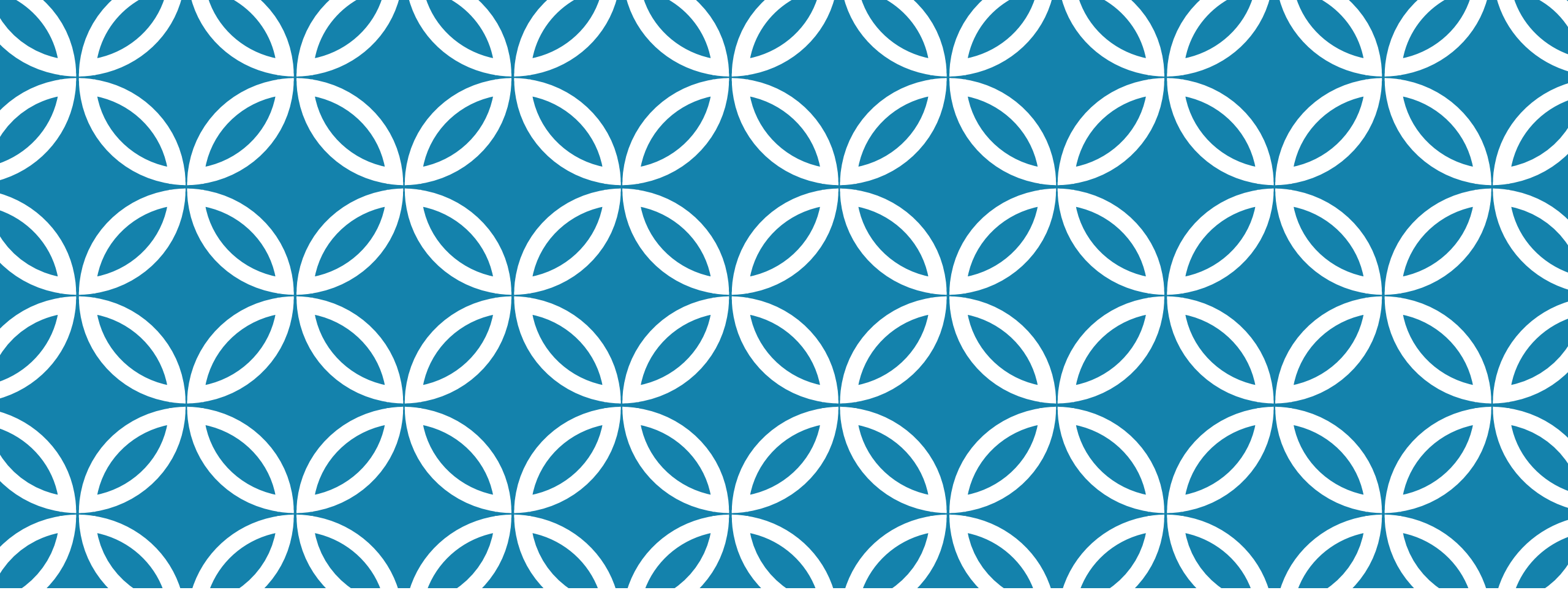
5 .Create a BST for the following data.

50,25,75,22,40,60,80,90,15,30

6. What is a binary search tree? Create a binary search tree for inserting the following data. 13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18.Explain deletion in the above tree
7. Generate a binary search tree for following numbers and perform in-order and post-order traversals: 7, 4, 2, 3, 6, 5, 12, 9, 8, 11, 19, 15, 20
8. Create the BST for the following data and do inorder, Pre–order and Post-order traversal of the tree. 40,60,15,4,30,70,65,10,95,25,34
9. The preorder traversal is as follow : 30,20,10,15,25,23,39,35,42

   Draw a BST for the given data and give inorder and postorder traversal for the same.
10. Create a BST for the following data : 14,10,17,12,10,11,20,12,18,25,20,8,22,11,23

    Explain deleting node 20 in the resultant BST.
11. Explain AVL tree with example.
12. Explain 2-3 trees with example.
13. Explain Threaded binary tree with example.



DATA STRUCTURE AND ALGORITHM

UNIT - 3

Non-linear Data structure-Graph

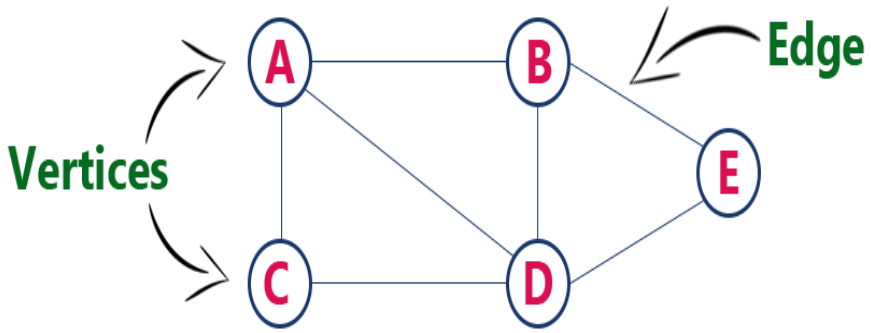
Prepared by: Prof. Kinjal Patel.

GRAPH TERMINOLOGY

Graph

- A graph G consist of a non-empty set V called the set of nodes (points, vertices) of the graph, a set E which is the set of edges and a mapping from the set of edges E to a set of pairs of elements of V.
- A graph as G=(V , E)
- Graph = Every edge of a graph G, we can associate a pair of nodes of the graph. If an edge X ∈ E is thus associated with a pair of nodes (u,v) where u, v ∈ V then we says that edge x connect u and v.
- This graph G can be defined as G = (V , E)
  Where V = {A,B,C,D,E}

  E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Vertex :- Individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

Adjacent Nodes :- Any two nodes which are connected by an edge in a graph are called adjacent node.

Edge :- An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex). For example, in above graph the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

Undirected Edge - An undirected edge is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

Directed Edge - A directed edge is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

Weighted Edge - A weighted edge is a edge with value (cost) on it.

APPLICATION OF GRAPH

In **computer science** graph is used to represent flow of computation

In **Google Map** graph is used to represent the direction from one place to another place where places are represent by node and rode is represent by edge.

In **Facebook** the friends suggestion algorithm use graph theory, where users are node and edge represent friendship, here it is undirected graph

In **world wide web,** the pages are node and hyperlink from one page to second page is the edge, here it is directed graph

In **operating system,** resource allocation is example of Graph. Where process and resources are node and allocation of resources to process is called edge

GRAPH-MATRIX REPRESENTATION OF GRAPHS

Graph G (V, E) can be represent by two ways

1) Using adjacency matrix

2) Using an adjacency List

1) Using Adjacency Matrix

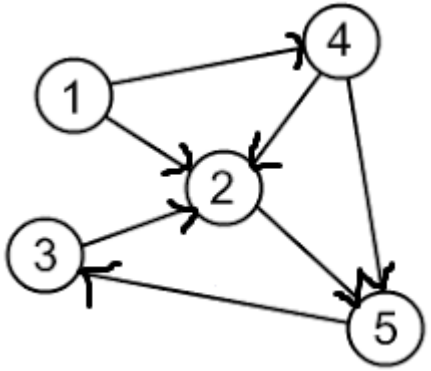
The adjacency matrix representation of a directed graph G consist of |v|*|v| matrix A=(a_{ij}), such that

$$aij = \begin{cases} 1 \; if (i,j) \in E \\ 0 \text{ otherwise} \end{cases}$$

The adjacency matrix representation of a undirected graph G consist of |v|*|v| matrix A=(a_{ij}), such that

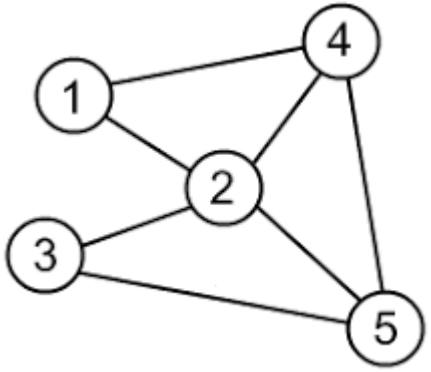
$$aij = \begin{cases} 1 \; if (i,j) \in E \quad \text{or } (j,i) \in E \\ 0 \text{ otherwise} \end{cases}$$

Directed Graph



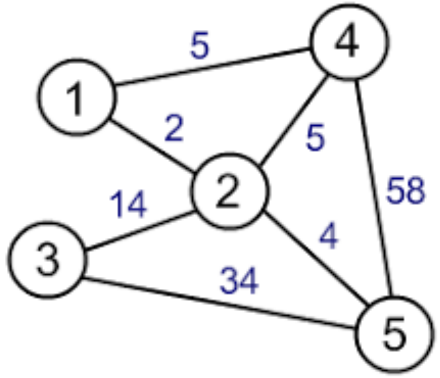
	1	2	3	4	5
1	0	1	0	1	0
2	0	0	0	0	1
3	0	1	0	0	0
4	0	1	0	0	1
5	0	0	1	0	0

Undirected Graph



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	1
3	0	1	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

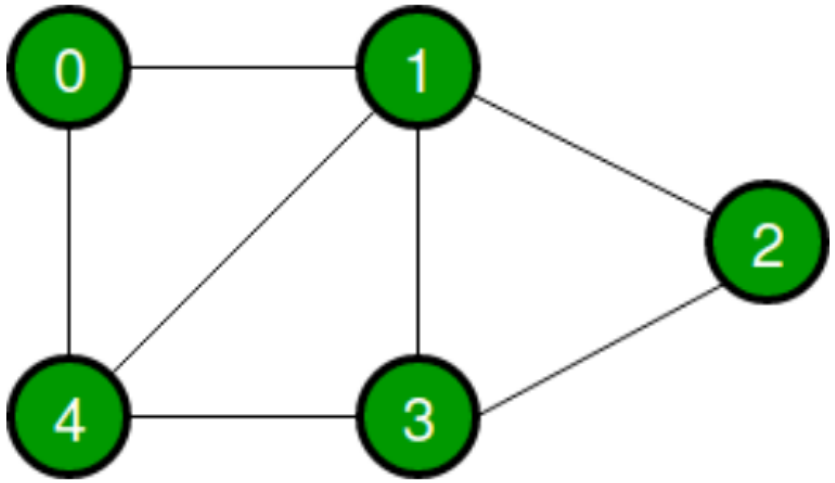
Weighted Graph

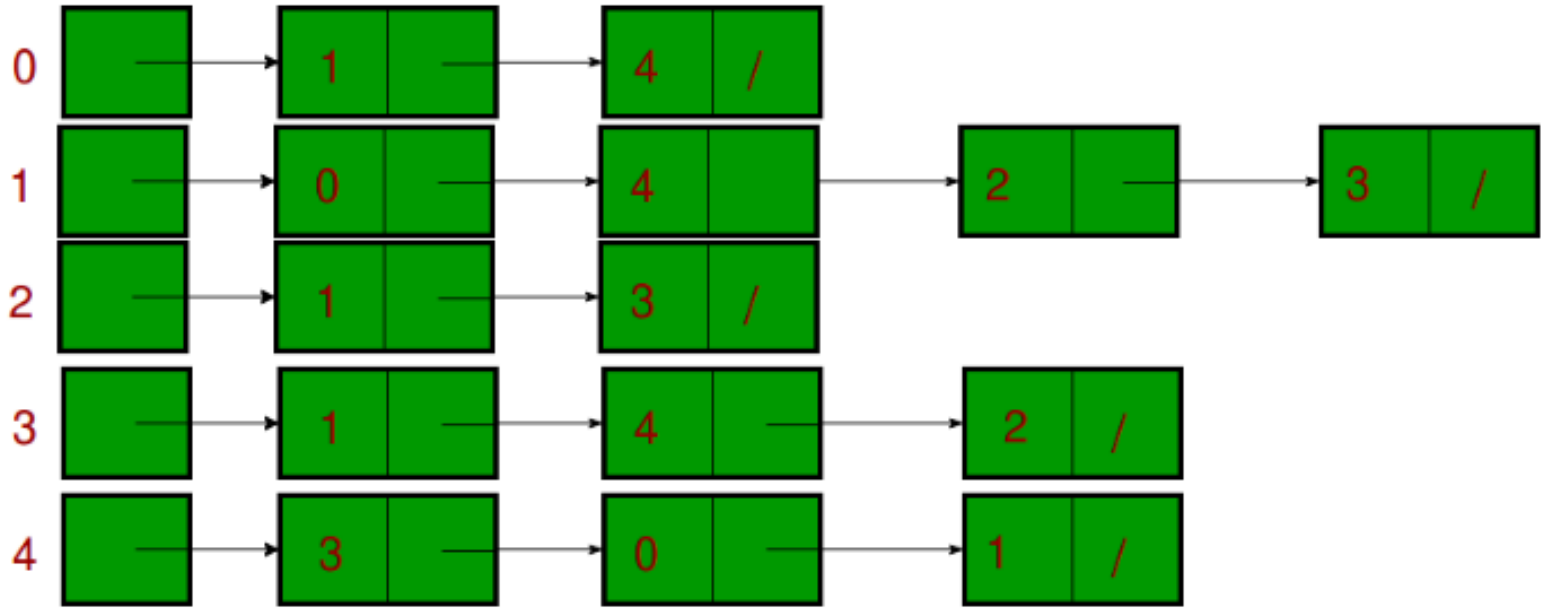


	1	2	3	4	5
1	0	2	0	5	0
2	2	0	14	5	4
3	0	14	0	0	34
4	5	5	0	0	58
5	0	4	34	58	0

Graph Representation Using Adjacency List

The Adjacency list representation of the graph G=(V,E) consist of an array Adj of |v| lists, one for each vertex in V.





TRAVERSAL OF GRAPH

There are two method to traverse a Graph G(v,e)

1) Breadth-First Search
2) Depth-First Search

BREADTH FIRST SEARCH

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes until it find the goal.

BFS traversal of a graph produces a **spanning tree (graph without loop)** as final result.

We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

Step 1 - Define a Queue of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

Step 3 - Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.

Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

Step 5 - Repeat steps 3 and 4 until queue becomes empty.

Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

ALGORITHM

Step 1: SET STATUS = 1 (ready state) for each node in G

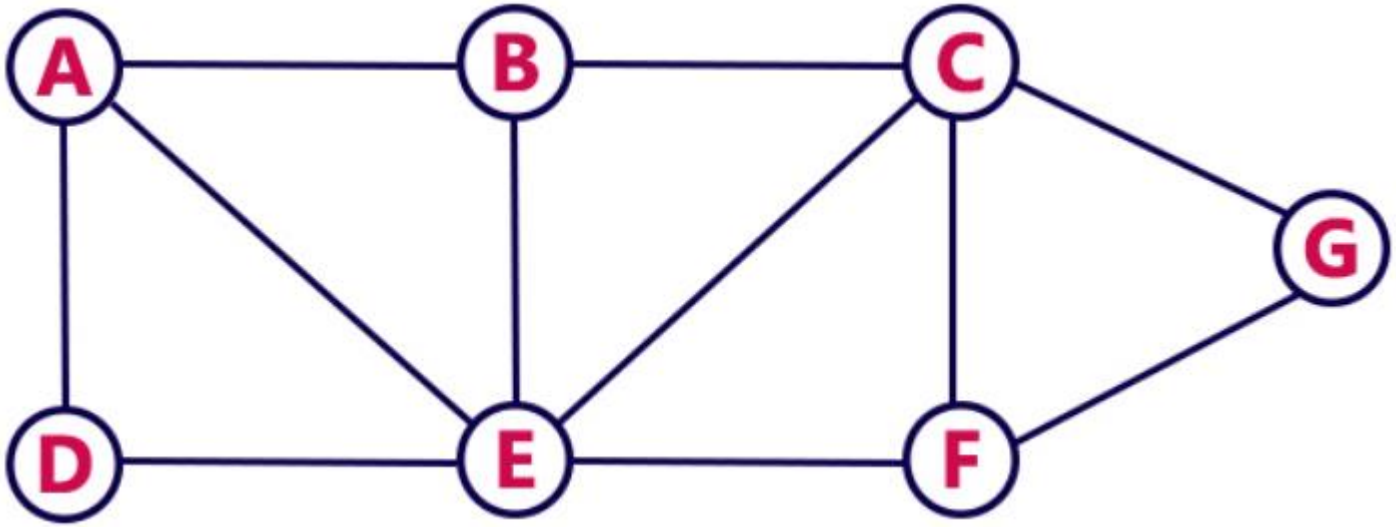
Step 2: Enqueue the starting node A and set its STATUS = 2
(waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3
(processed state).

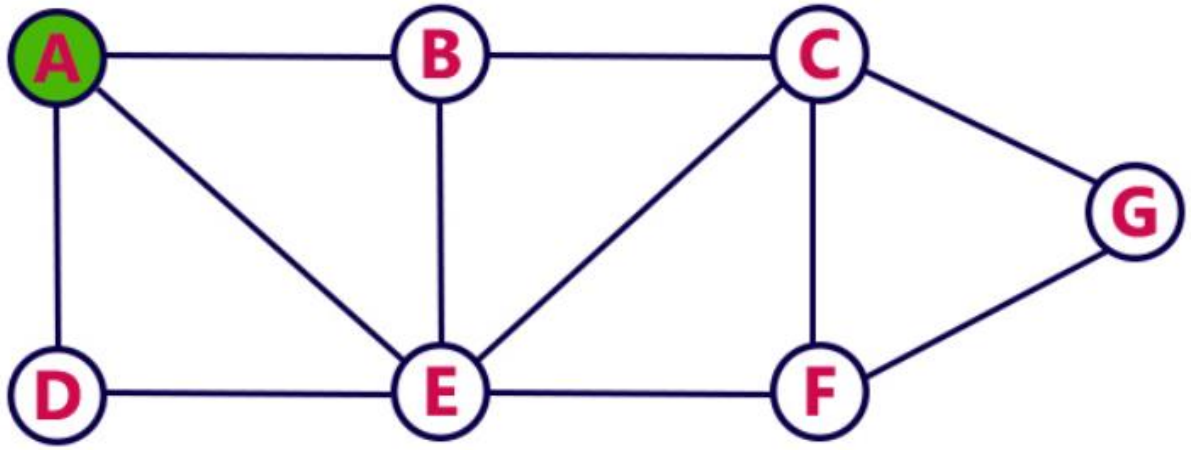
Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2
(waiting state) [END OF LOOP]

Step 6: EXIT

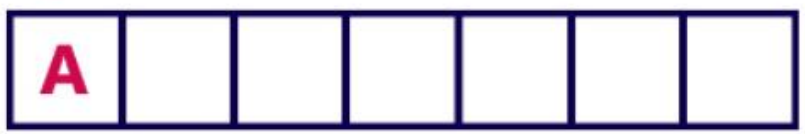


Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

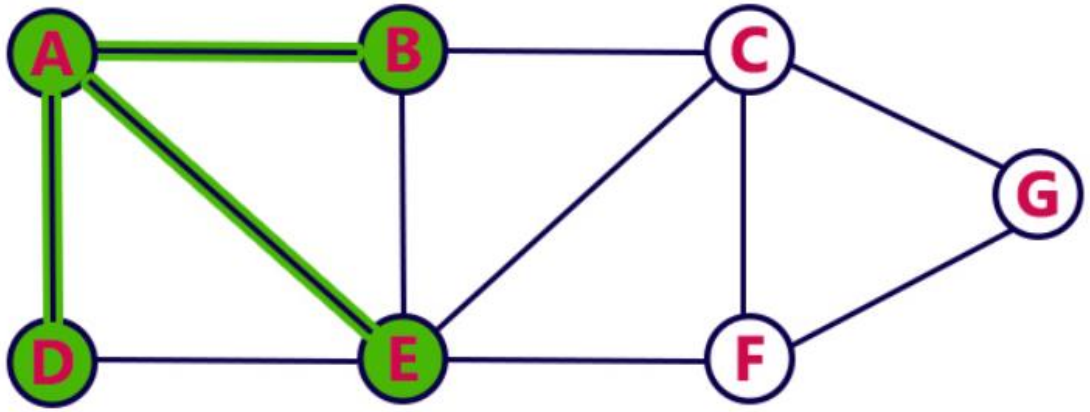


Queue



Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

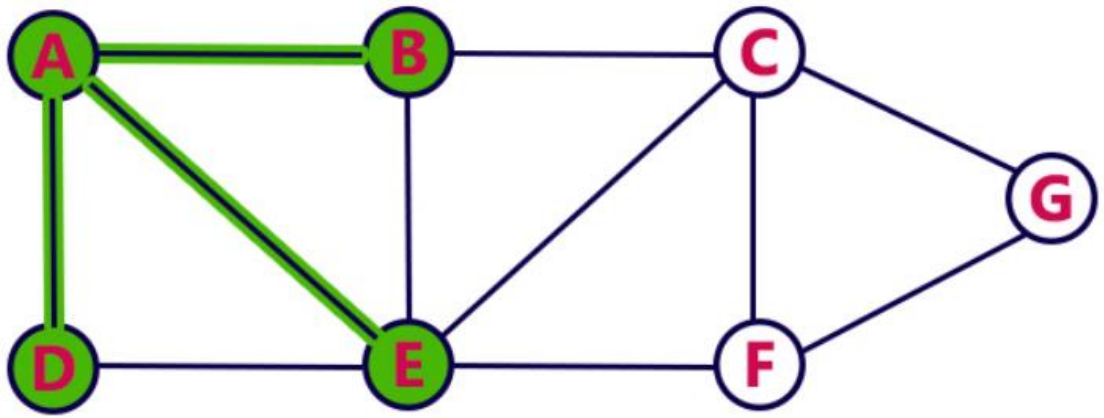


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

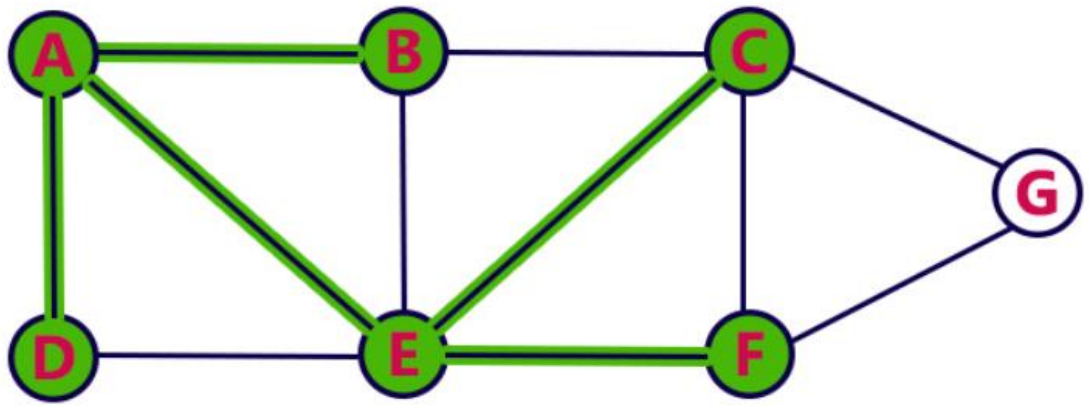


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

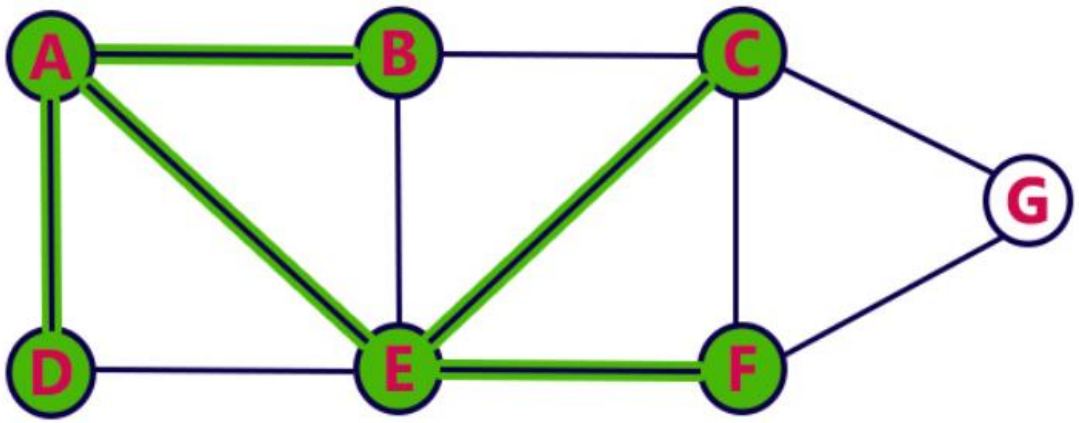


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

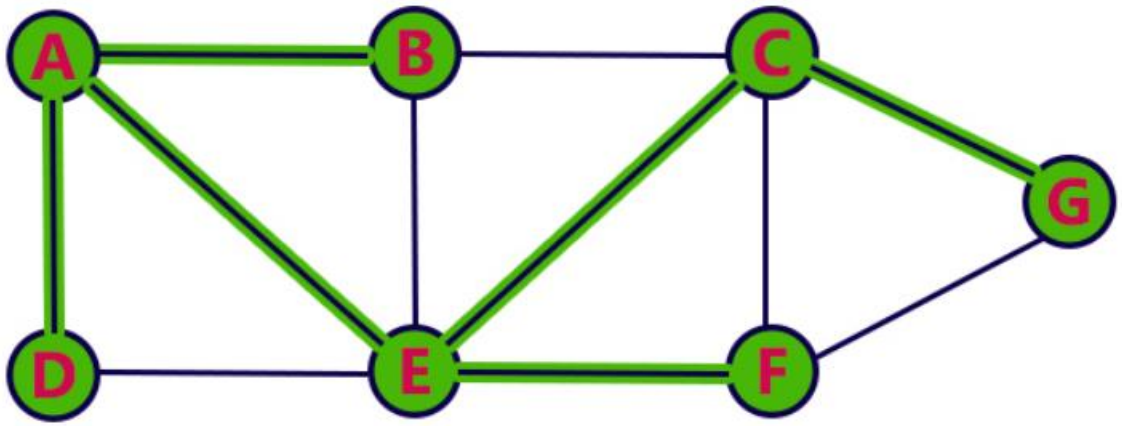


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

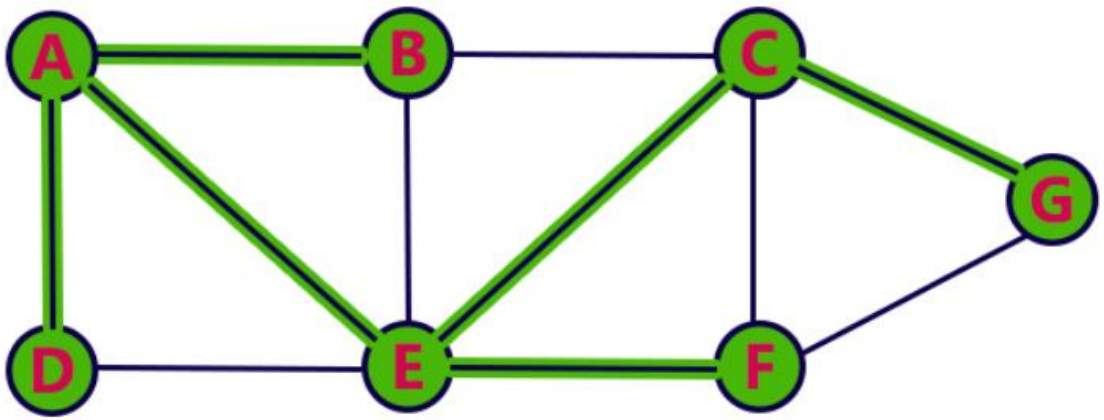


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

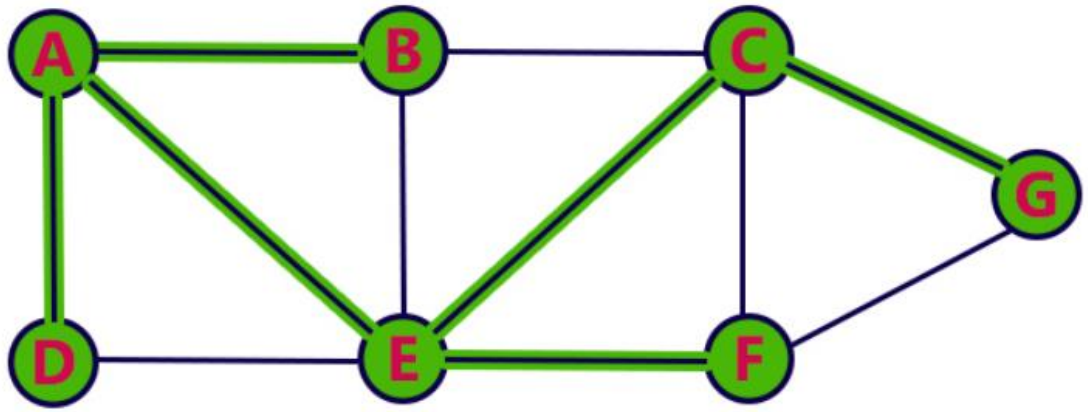


Queue

						G

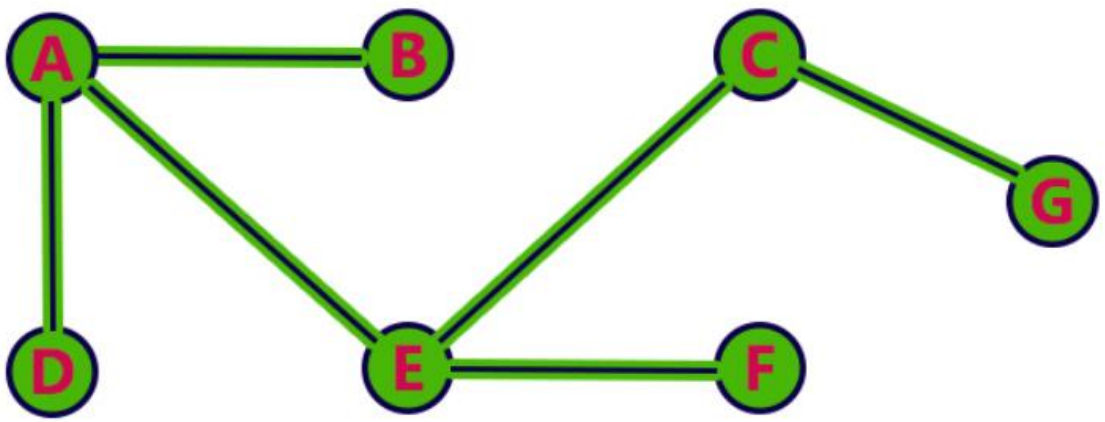
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Depth-First Search

- The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered.
- When a dead-end is reached, the algorithm **backtracks,** returning to the most recent node that has not been completely explored
- DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

ALGORITHM

Step 1: SET STATUS = 1 (ready state) for each node in G

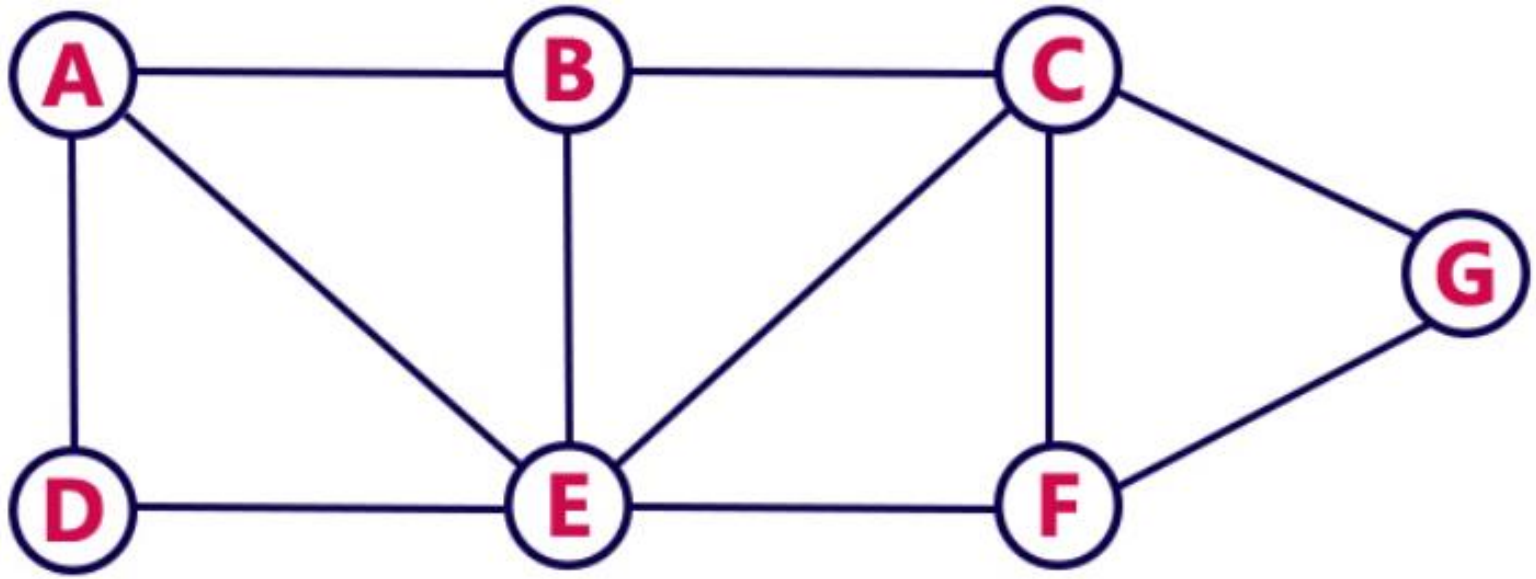
Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

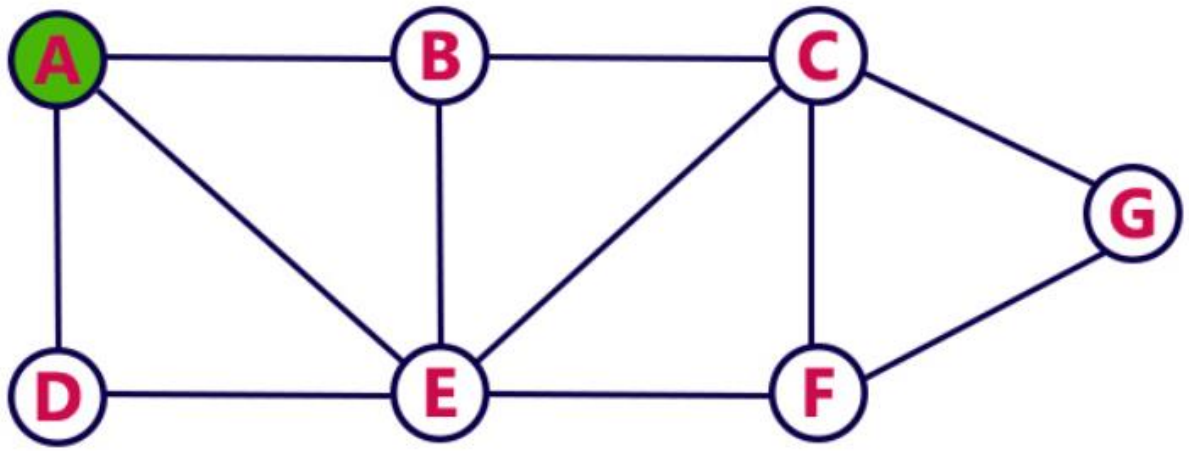
Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)
[END OF LOOP]

Step 6: EXIT



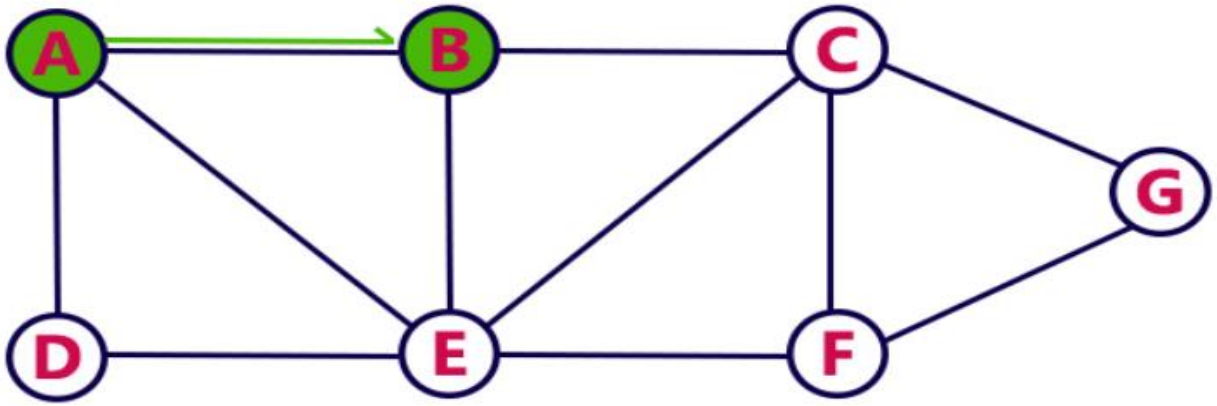
Step 1:

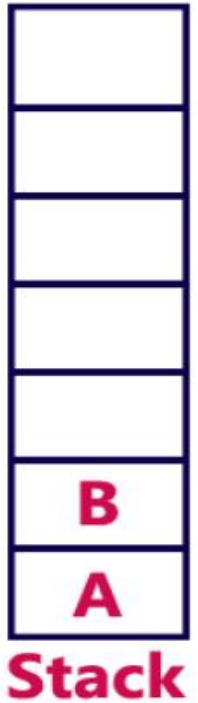
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



Step 2:

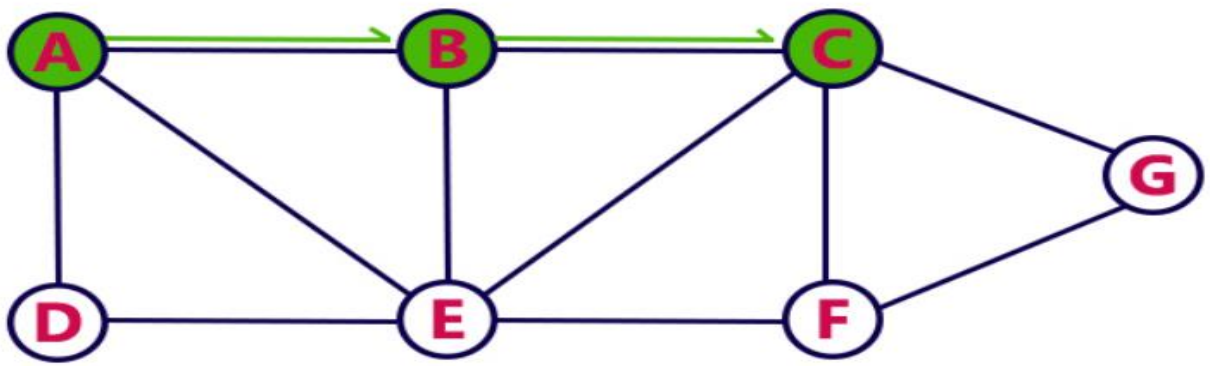
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

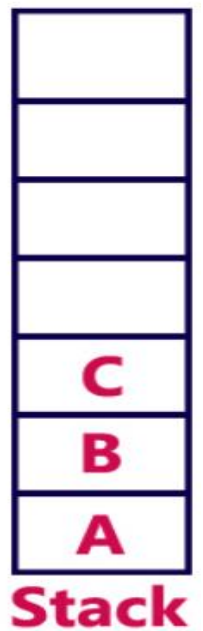




Step 3:

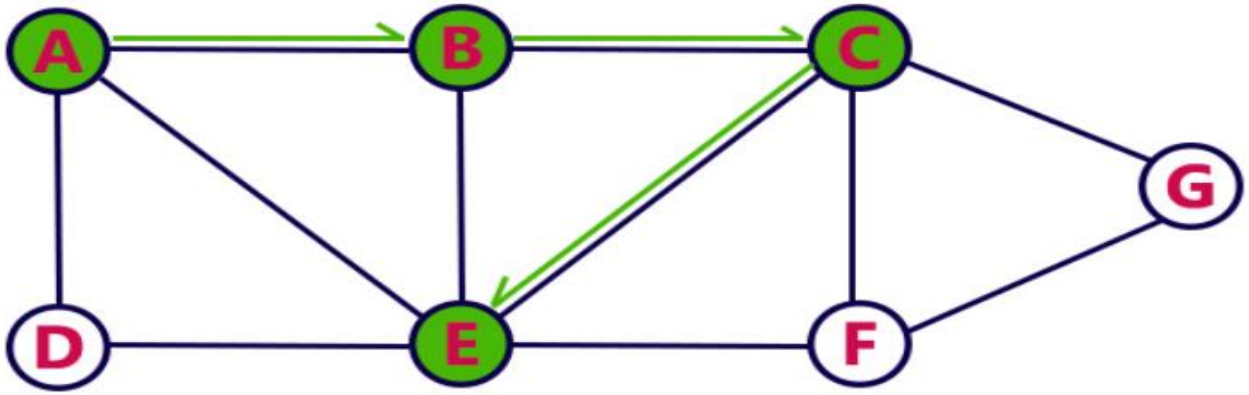
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

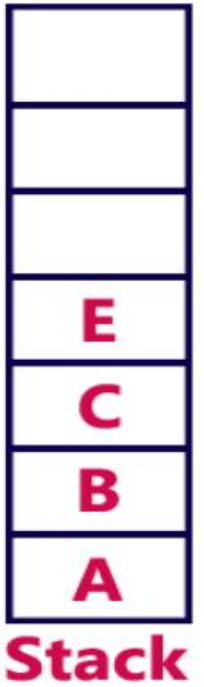




Step 4:

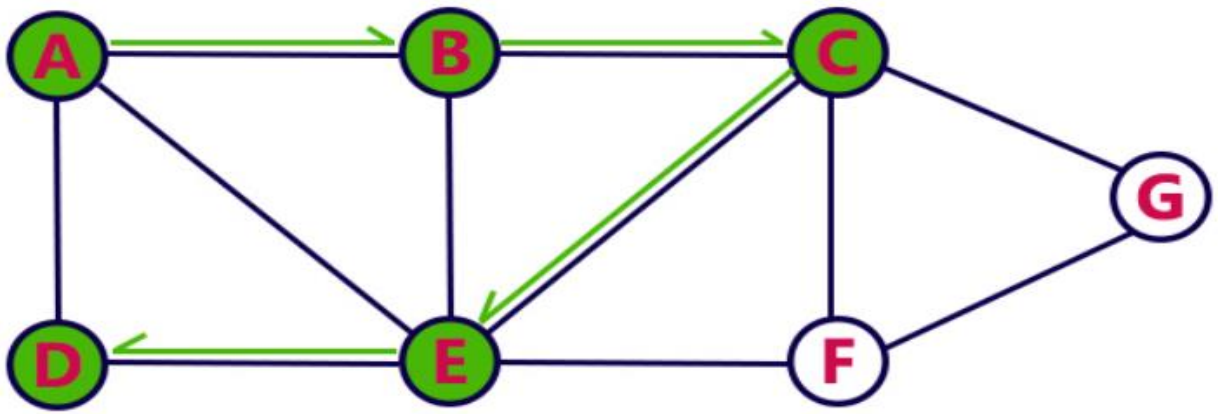
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack





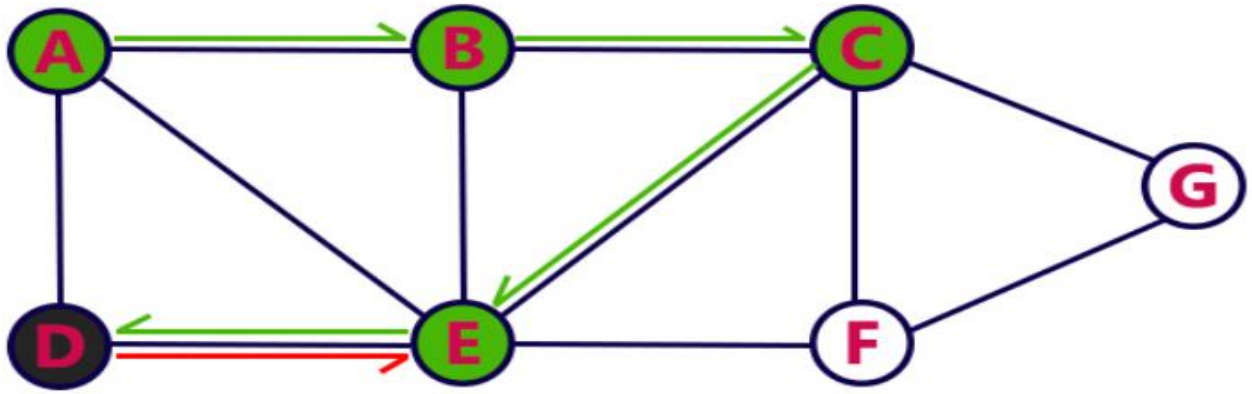
Step 5:

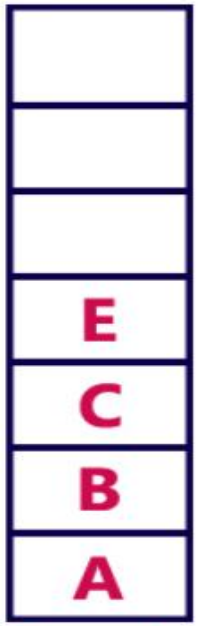
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



Step 6:

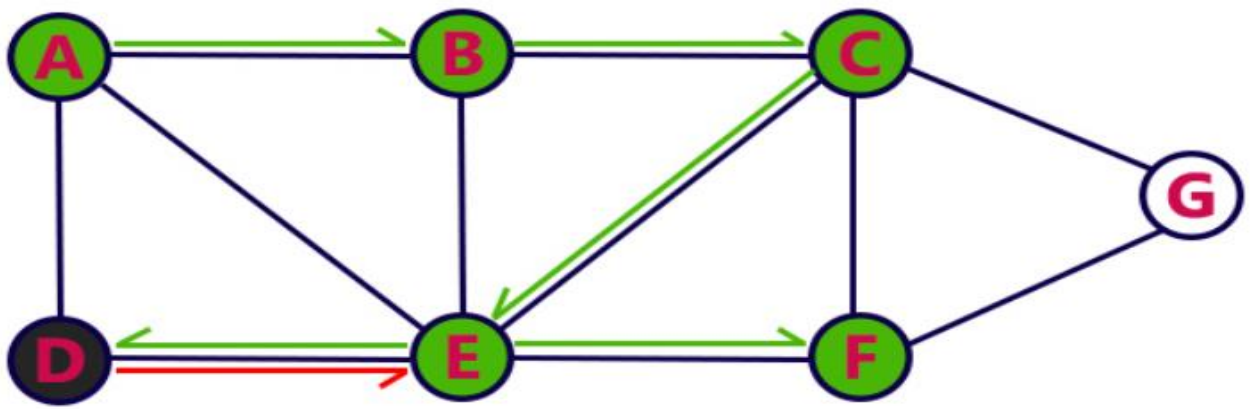
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

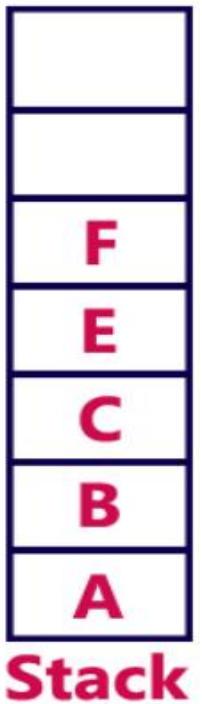




Step 7:

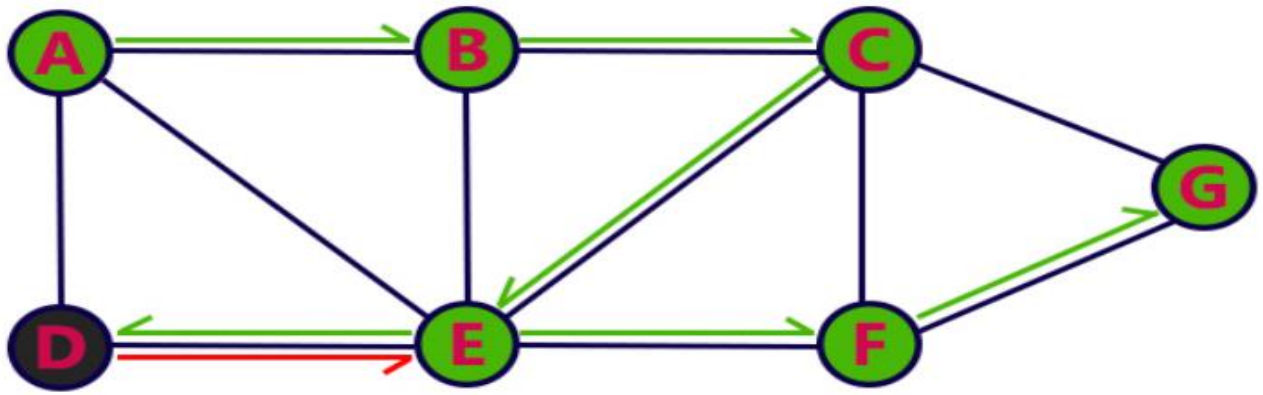
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

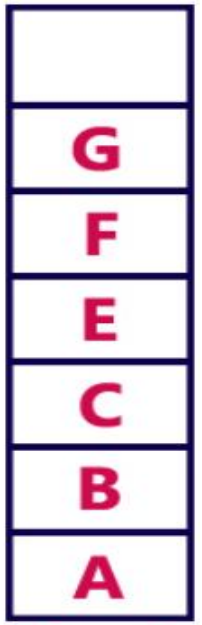




Step 8:

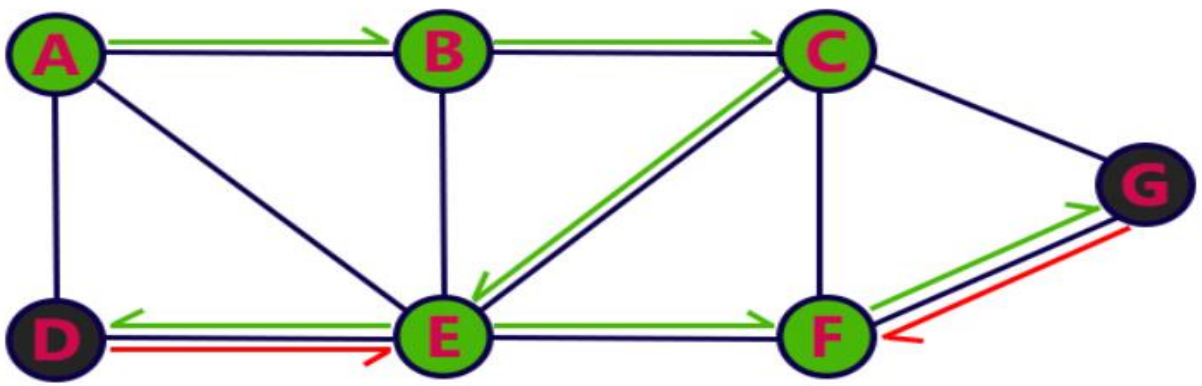
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

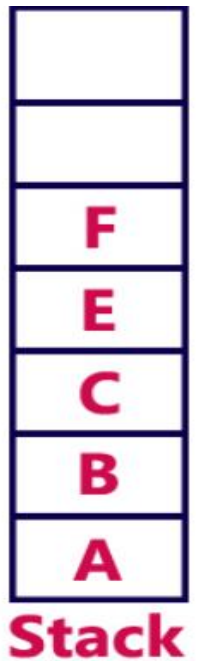




Step 9:

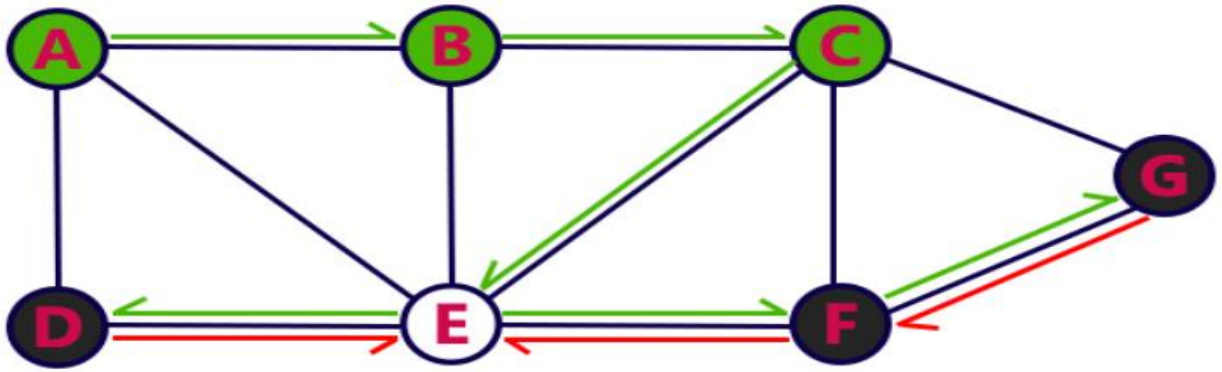
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

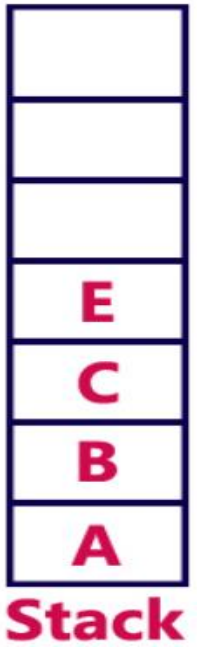




Step 10:

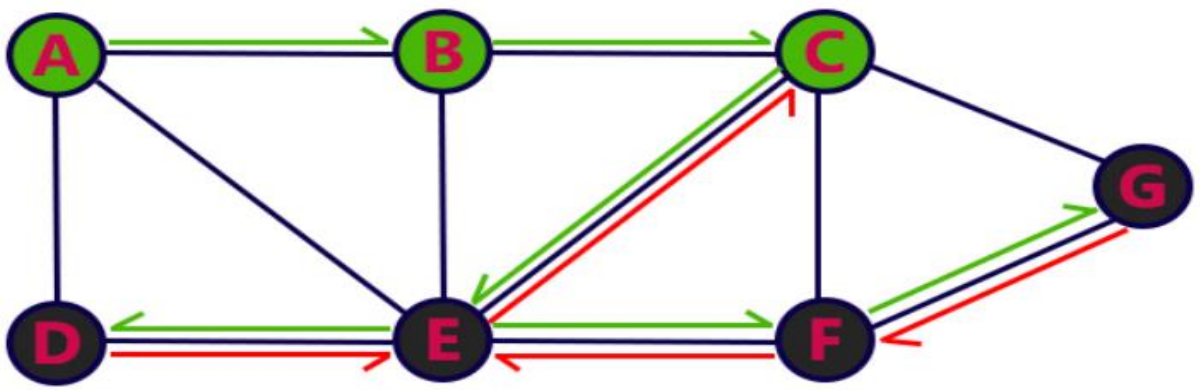
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

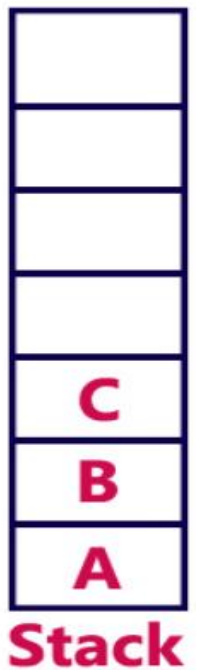




Step 11:

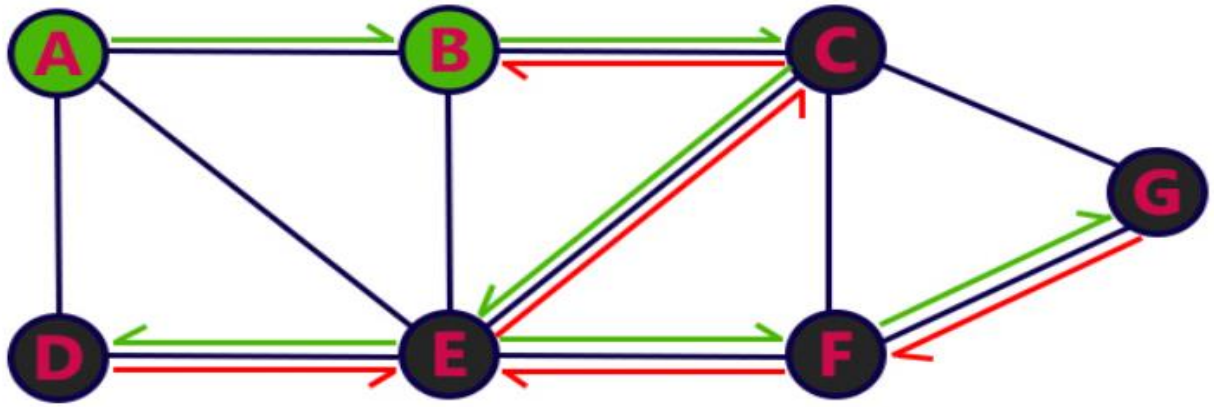
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

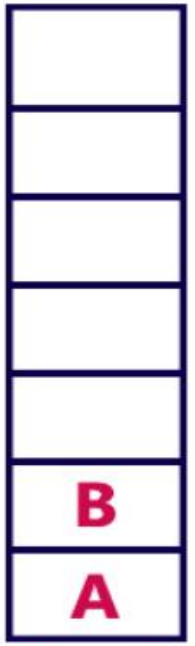




Step 12:

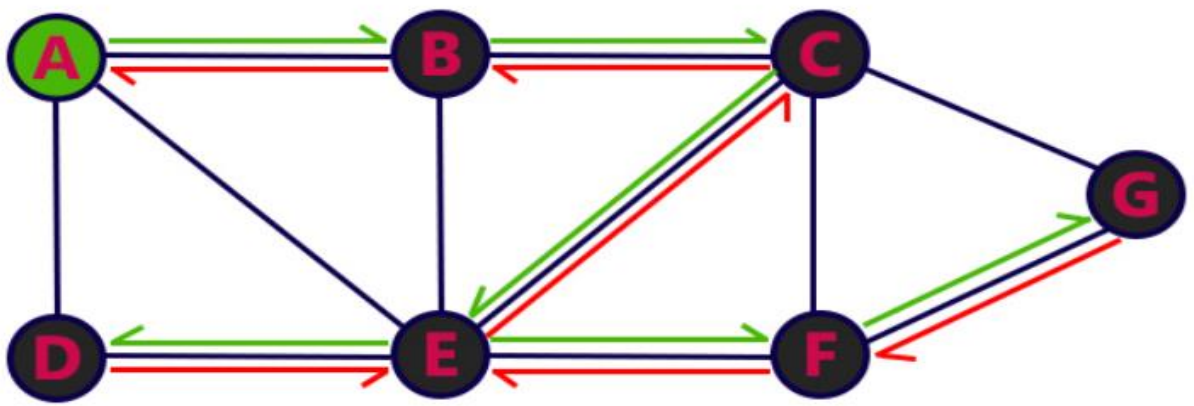
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

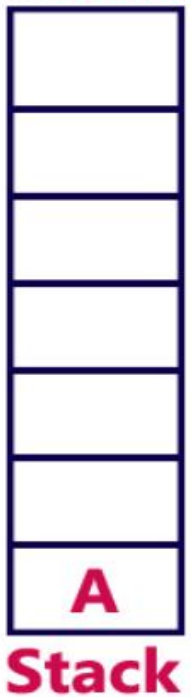




Step 13:

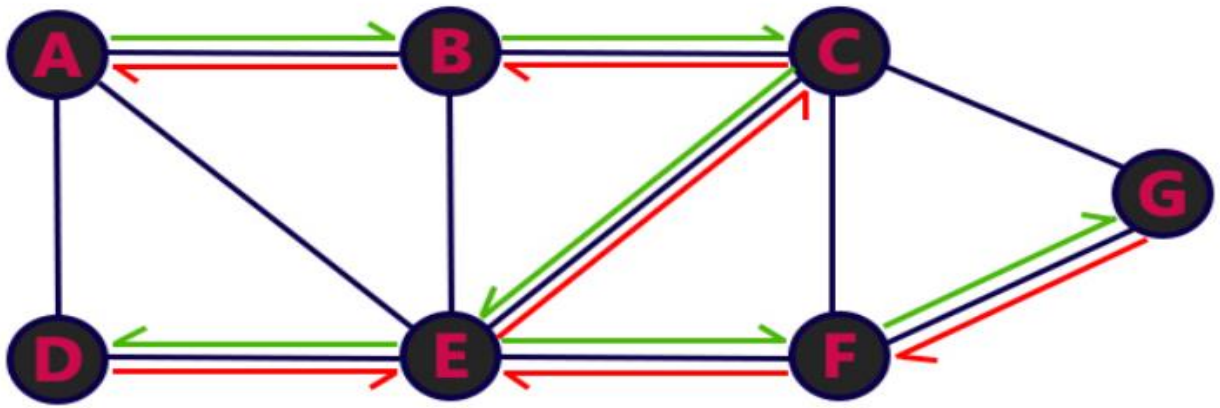
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

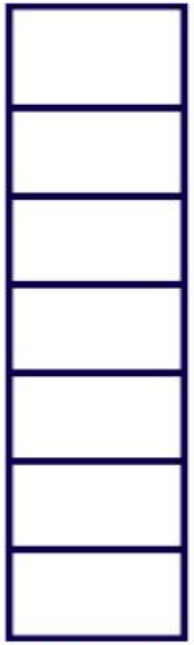




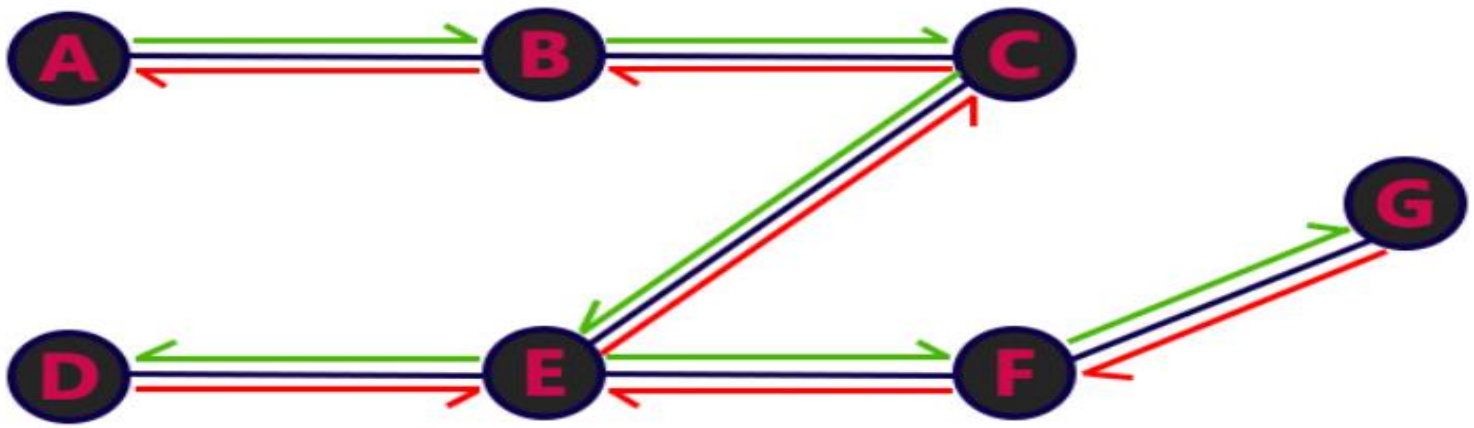
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.





- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



SPANNING TREES

A spanning tree of a connected, undirected graph G is a sub-graph of G which is a tree that connects all the vertices together

A graph G can have many different spanning trees

Use weight of edge in spanning tree by calculating the sum of the weights of the edges in that spanning tree.

A ***minimum spanning tree*** **(MST)** is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree

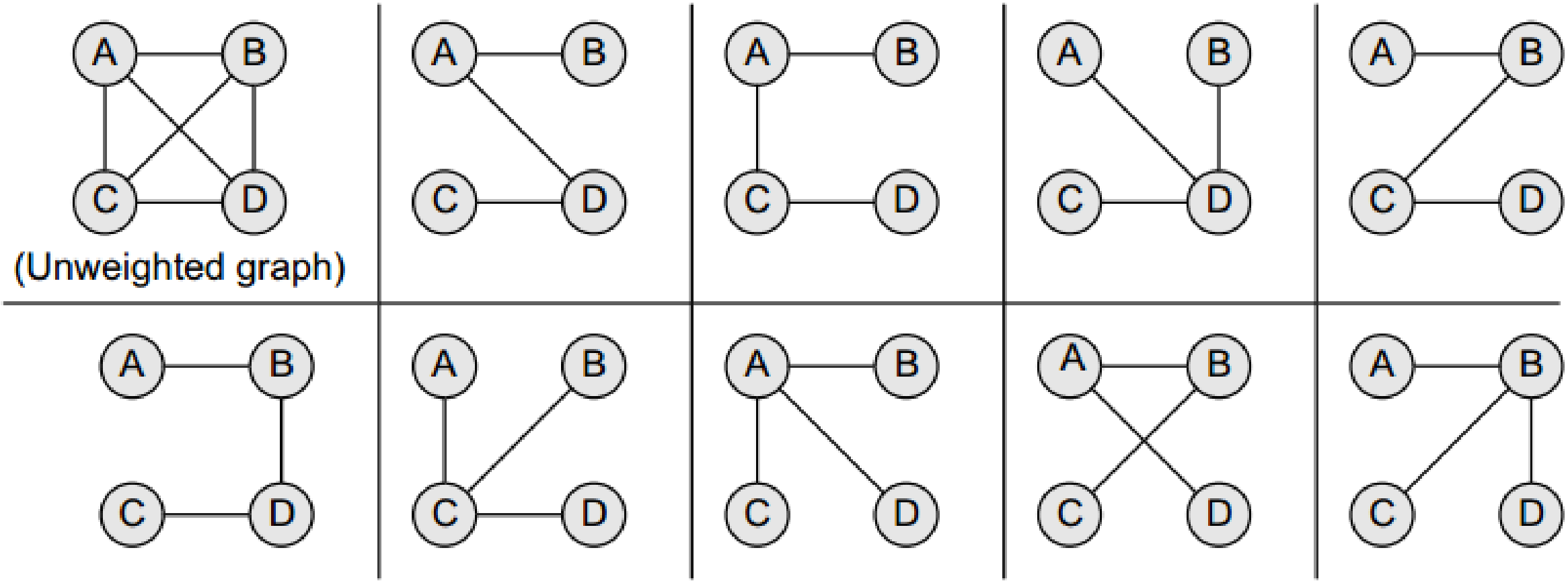
Applications of Minimum Spanning Trees

1. MSTs are widely used for designing networks. used to determine the least costly paths with no cycles in this network

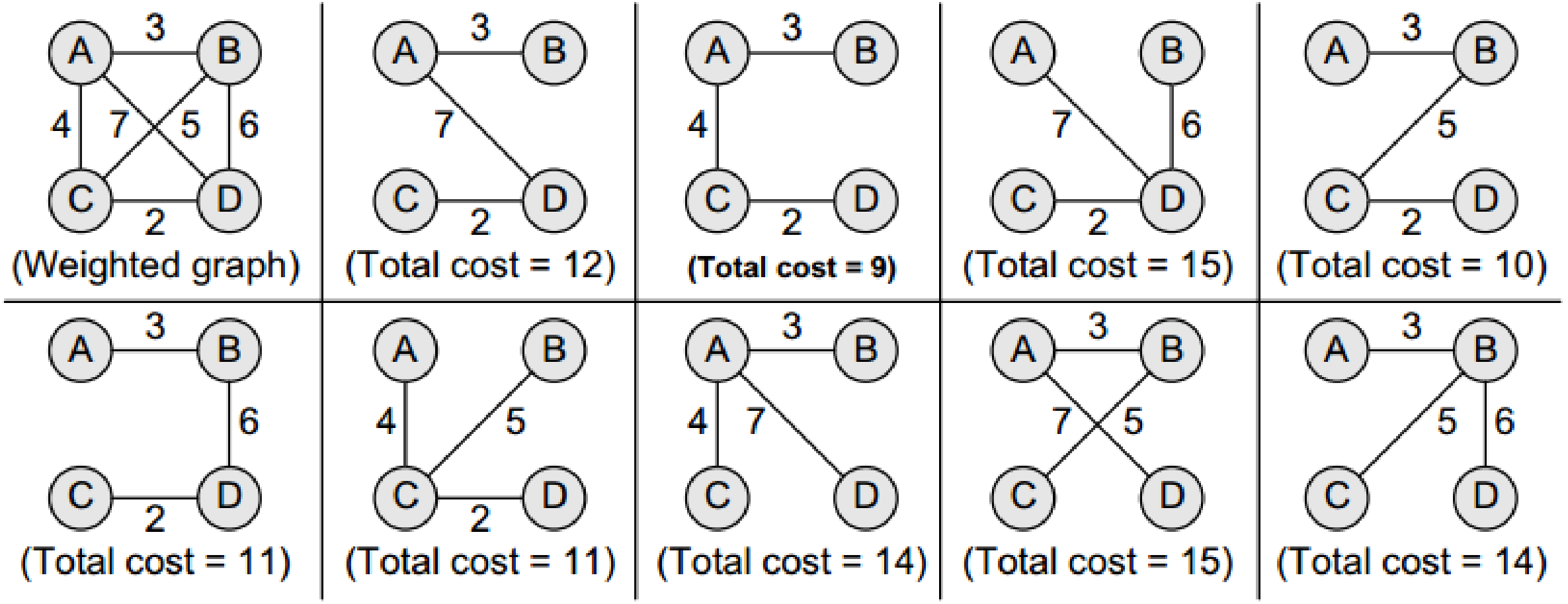
2. MSTs are used to find airline routes

3. MSTs are also used to find the cheapest way to connect terminals, such as cities, electronic components or computers via roads, airlines, railways, wires or telephone lines

4. MSTs are applied in routing algorithms for finding the most efficient path.



Unweighted graph and its spanning trees



Weighted graph and its spanning trees

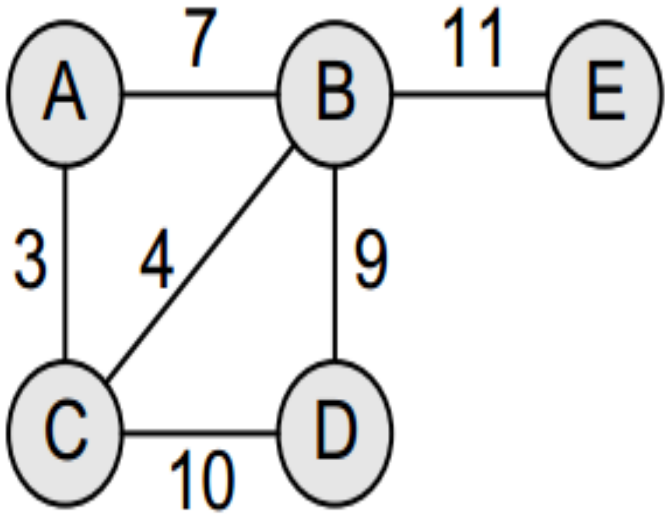
PRIM'S ALGORITHM

Prim's algorithm is a greedy algorithm that is used to form a **minimum spanning tree** for a **connected weighted undirected** graph

The algorithm builds a tree that includes every vertex and a subset of the edges in such a way that the total weight of all the edges in the tree is minimized

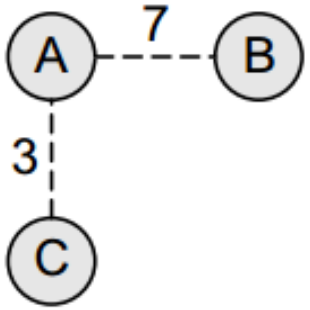
Algorithm steps:
Step 1: Select a starting vertex
Step 2: Repeat Steps 3 and 4 until there are adjacent vertices
Step 3: Select an edge **e** connecting the tree vertex and adjacent vertex that has minimum weight
Step 4: Add the selected edge and the vertex to the minimum spanning tree T
[END OF LOOP]
Step 5: EXIT

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/MCST.html#prim-s-algorithm

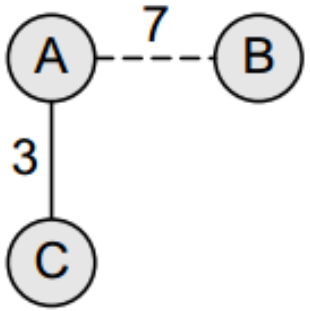


COST(MST)=3+4+9+11=27

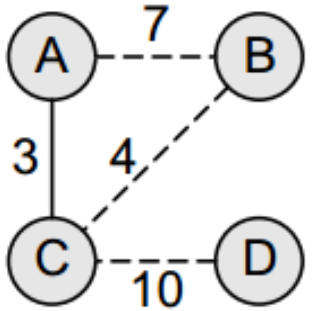
Step 1



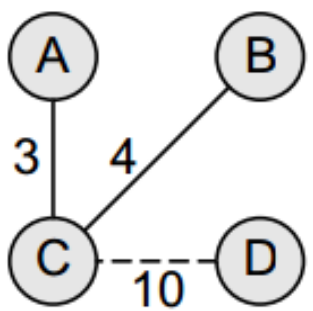
Step 2



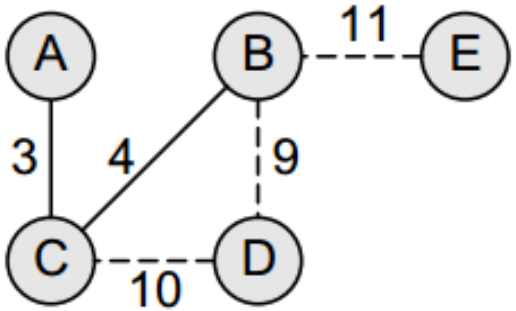
Step 3



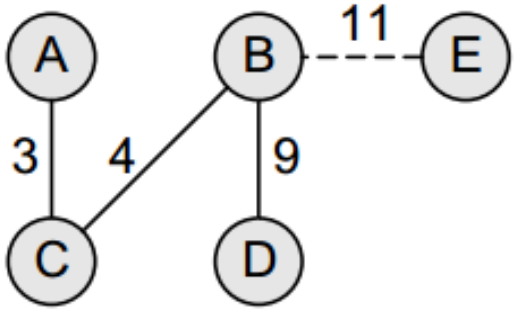
Step 4



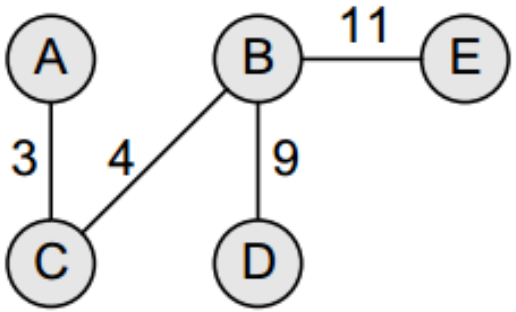
Step 5



Step 6

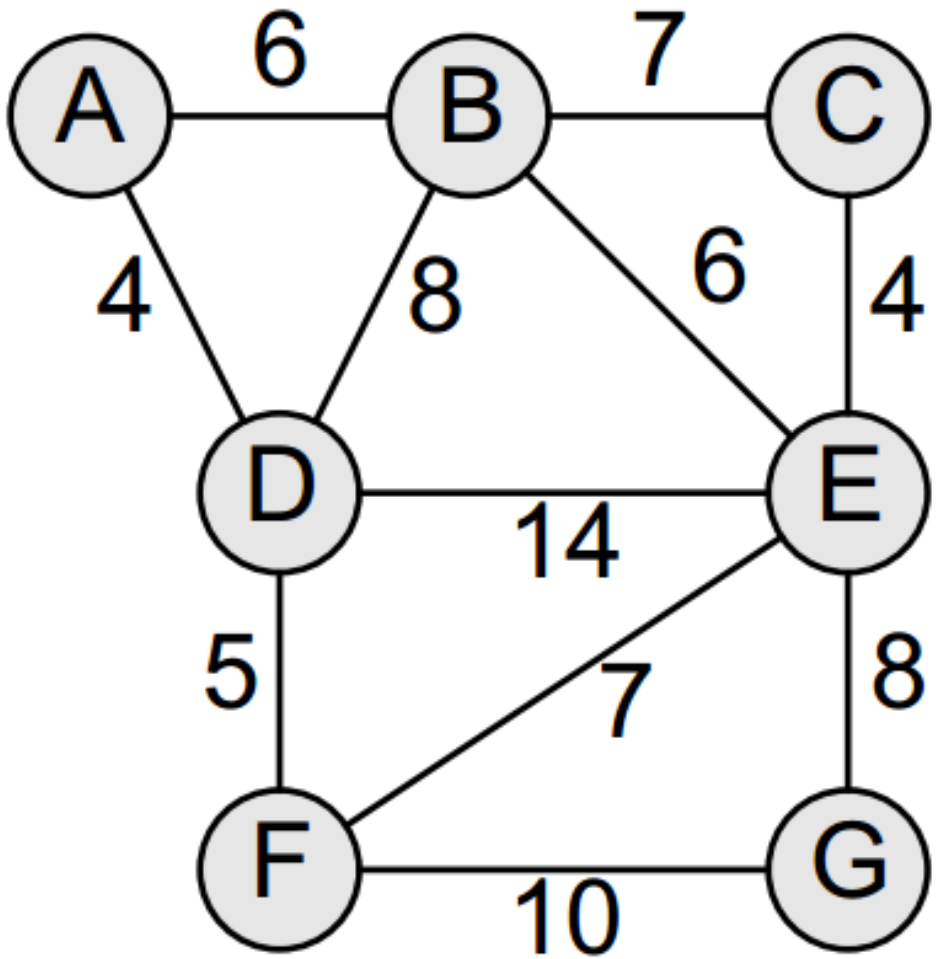


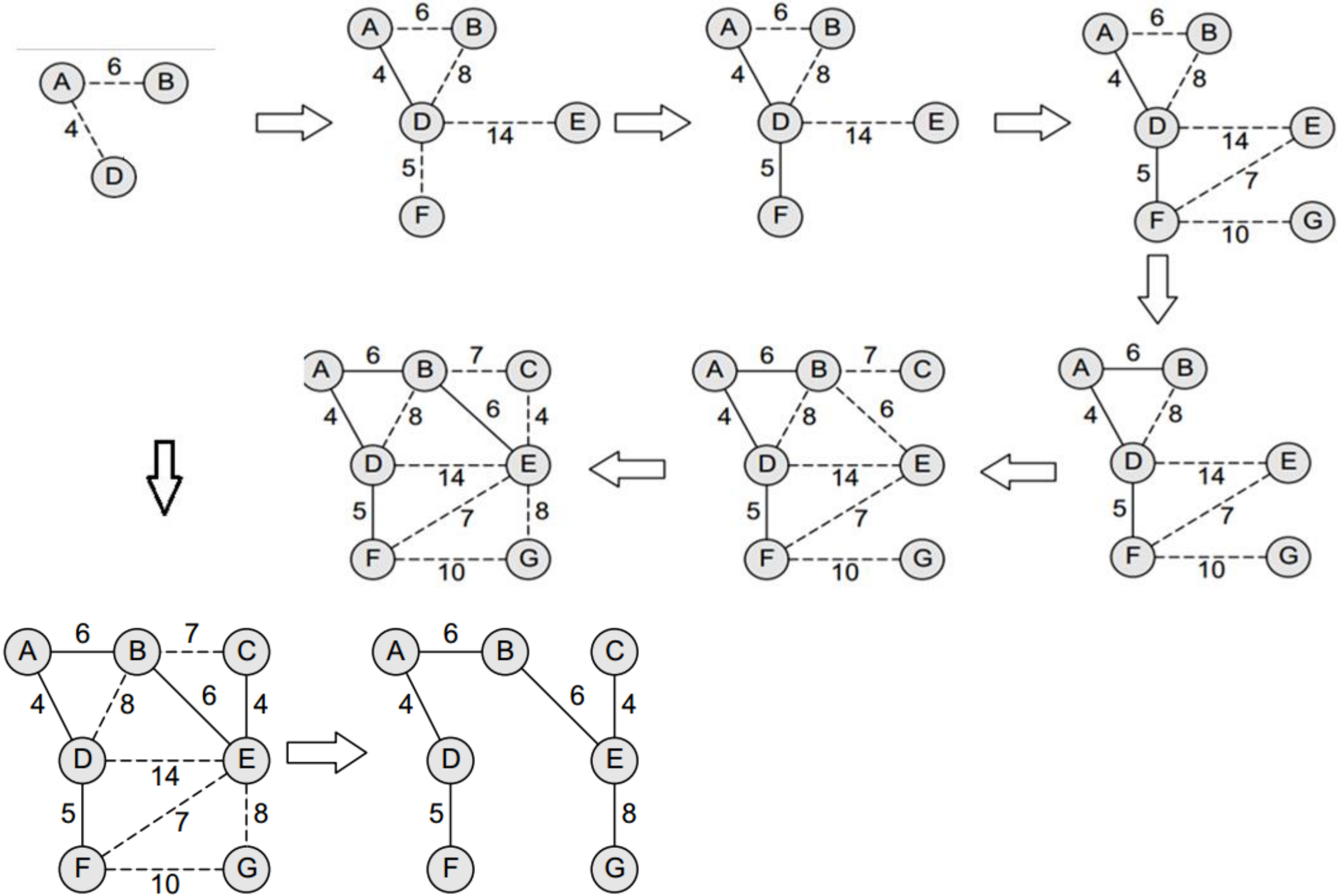
Step 7



Step 8

CREATE MST USING PRIMS ALGORITHM




A
B
C
D
E
F
G
6
4
8
14
5
7
10
4
6
7
8
6
4
14
7
8
10
5

KRUSKAL'S ALGORITHM

- Kruskal's algorithm is used to find the minimum spanning tree for a connected weighted graph
- The total weight of all the edges in the tree is minimized
- In the algorithm, we use a priority queue Q in which edges that have minimum weight takes a priority over any other edge in the graph
- **Algorithm**

Step 1: Create a forest in such a way that each graph is a separate tree.

Step 2: Create a priority queue Q that contains all the edges of the graph.

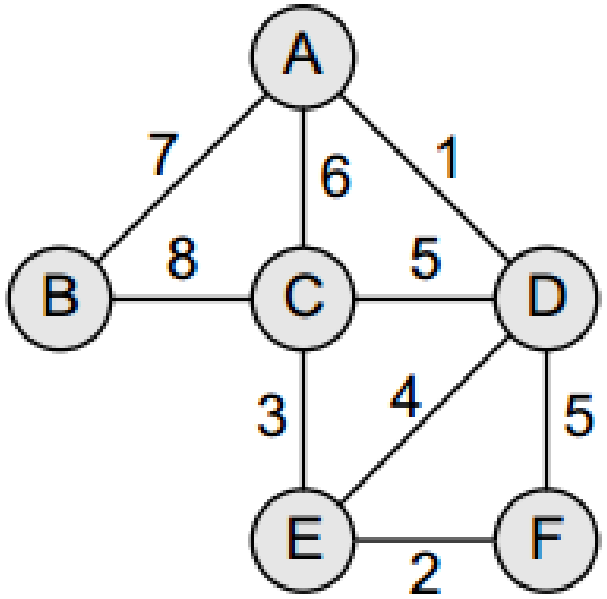
Step 3: Repeat Steps 4 and 5 while Q is NOT EMPTY

Step 4: Remove an edge from Q

Step 5: IF the edge obtained in Step 4 connects two different trees, then Add it to the forest (for combining two trees into one tree).
ELSE
Discard the edge

Step 6: END

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Kruskal.html

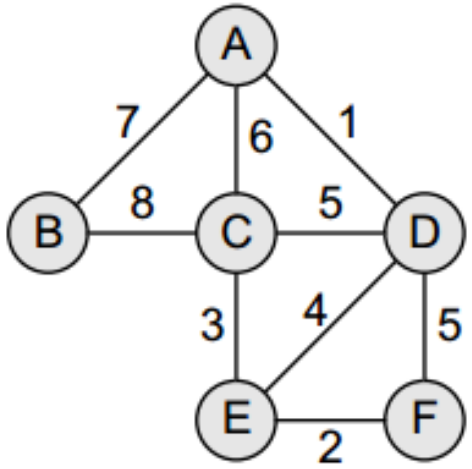


Initially, we have F = {{A}, {B}, {C}, {D}, {E}, {F}}

MST = {}

Q = {(A, D), (E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 1: Remove the edge `(A, D)` from `Q` and make the following changes:

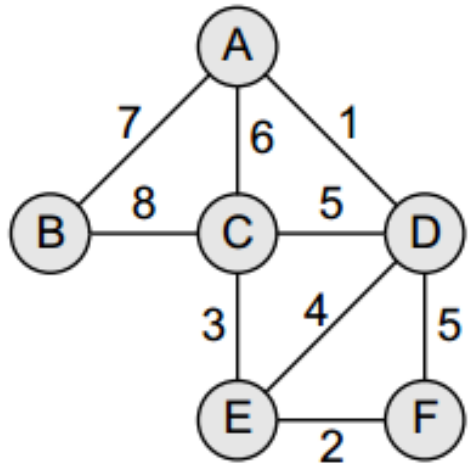


F = {{A, D}, {B}, {C}, {E}, {F}}

MST = {A, D}

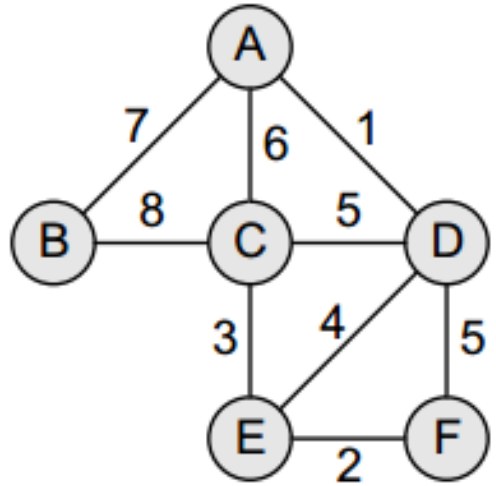
Q = {(E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 2: Remove the edge `(E, F)` from `Q` and make the following changes:



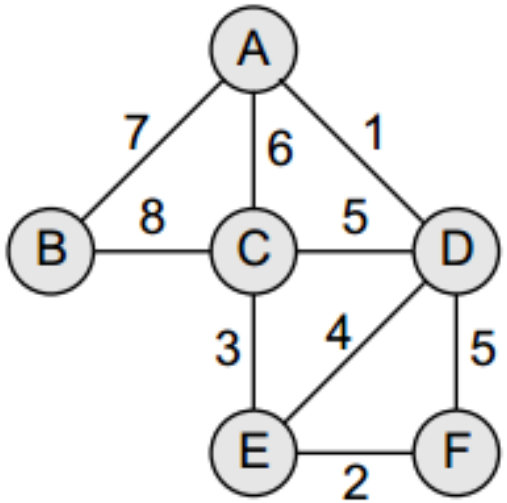
F = {{A, D}, {B}, {C}, {E, F}}
MST = {(A, D), (E, F)}
Q = {(C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 3: Remove the edge `(C, E)` from `Q` and make the following changes:



F = {{A, D}, {B}, {C, E, F}}
MST = {(A, D), (C, E), (E, F)}
Q = {(E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 4: Remove the edge (E, D) from Q and make the following changes:



F = {{A, C, D, E, F}, {B}}
MST = {(A, D), (C, E), (E, F), (E, D)}
Q = {(C, D), (D, F), (A, C), (A, B), (B, C)}

Step 5: Remove the edge (C, D) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST. Therefore,

$$F = \{\{A, C, D, E, F\}, \{B\}\}$$
$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$
$$Q = \{(D, F), (A, C), (A, B), (B, C)\}$$

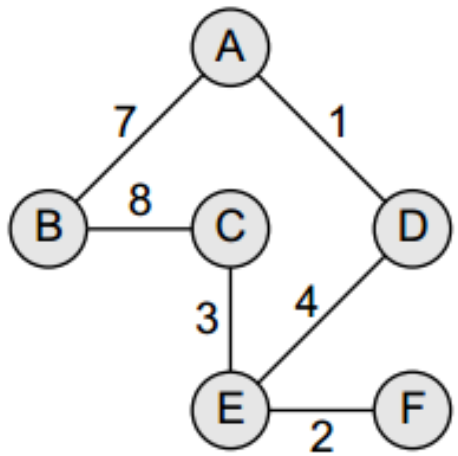
Step 6: Remove the edge `(D, F)` from `Q`. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting `(A, D, C, E, F)` to `B` will be added to the MST.

$$F = \{\{A, C, D, E, F\}, \{B\}\}$$
$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$
$$Q = \{(A, C), (A, B), (B, C)\}$$

Step 7: Remove the edge `(A, C)` from `Q`. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting `(A, D, C, E, F)` to `B` will be added to the MST.

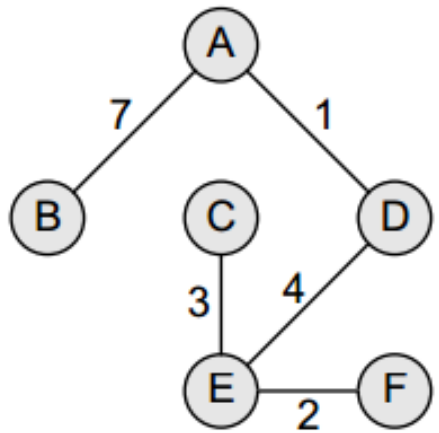
$$F = \{\{A, C, D, E, F\}, \{B\}\}$$
$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$
$$Q = \{(A, B), (B, C)\}$$

Step 8: Remove the edge `(A, B)` from `Q` and make the following changes:



F = {A, B, C, D, E, F}
MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}
Q = {(B, C)}

Step 9: The algorithm continues until Q is empty. Since the entire forest has become one tree, all the remaining edges will simply be discarded. The resultant MS can be given as shown below.



F = {A, B, C, D, E, F}

MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}

Q = {}

SHORTEST PATH ALGORITHM

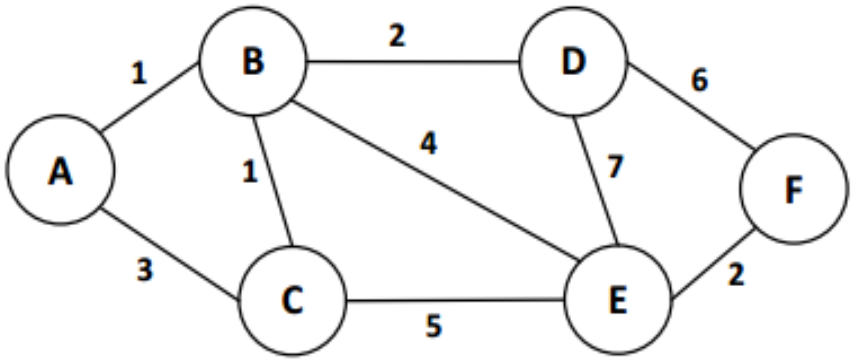
- The **shortest path** algorithm is used to find out **path** between two vertices (or nodes) in a **graph** such that the sum of the weights of its constituent edges is minimized.
- Dijkstra's algorithm is Shortest path Algorithm , given by a Dutch scientist Edsger Dijkstra in 1959
- Dijkstra's algorithm is used to find the length of an *optimal* path between two nodes in a graph
- The term *optimal* can mean anything, shortest, cheapest, or fastest.
- Dijkstra is greedy algorithm which work for a nonnegative edge weights
- It works only for connected graphs
- This algorithm works for both directed and undirected graphs

https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html

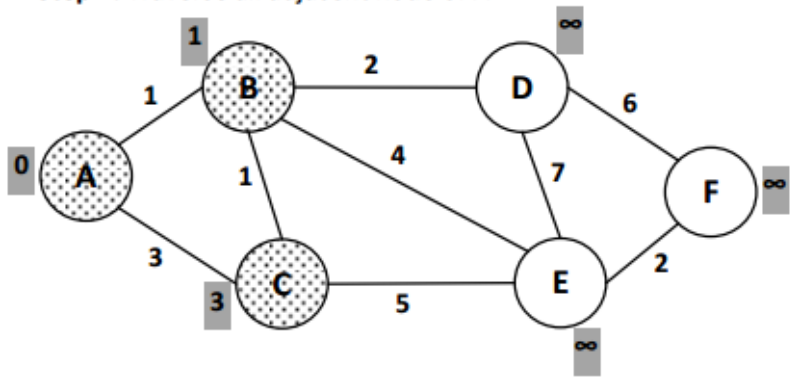
DIJKSTRA'S ALGORITHM

1. Select the source node also called the initial node
2. Define an empty set N that will be used to hold nodes to which a shortest path has been found.
3. Label the initial node with , and insert it into N.
4. Repeat Steps 5 to 7 until the destination node is in N or there are no more labelled nodes in N.
5. Consider each node that is not in N and is connected by an edge from the newly inserted node.
6. (a) If the node that is not in N has no label then SET the label of the node = the label of the newly inserted node + the length of the edge.

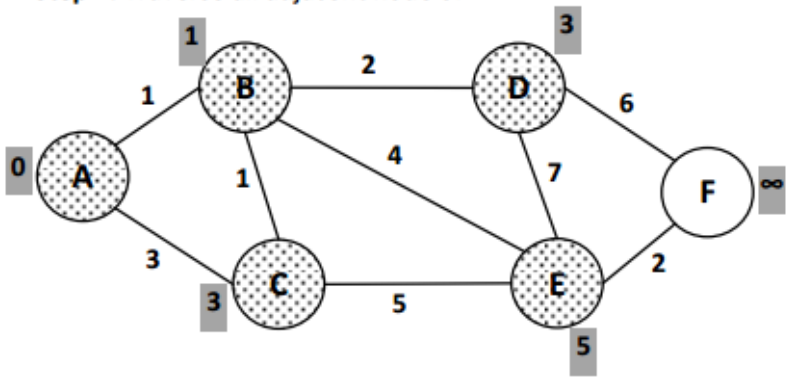
   (b) Else if the node that is not in N was already labelled, then SET its new label = minimum (label of newly inserted vertex + length of edge, old label)
7. Pick a node not in N that has the smallest label assigned to it and add it to N.



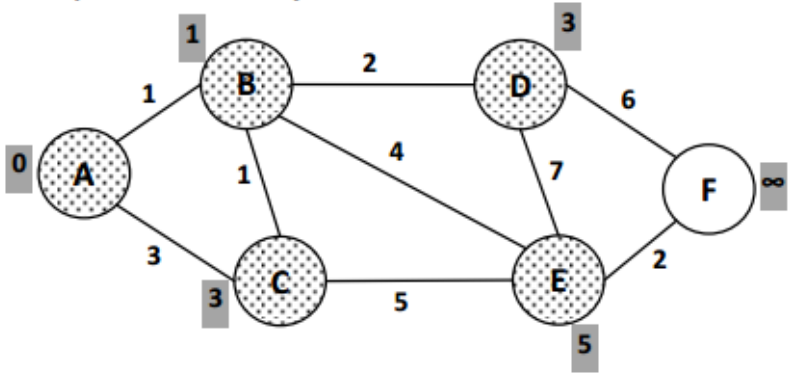
Step 1: Traverse all adjacent node of A



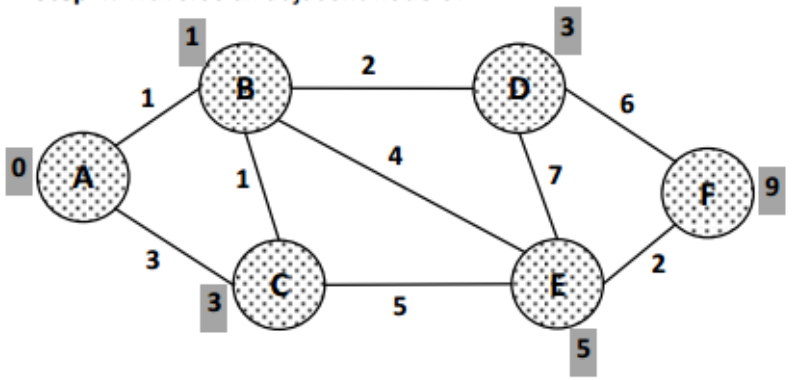
Step 2: Traverse all adjacent node of B



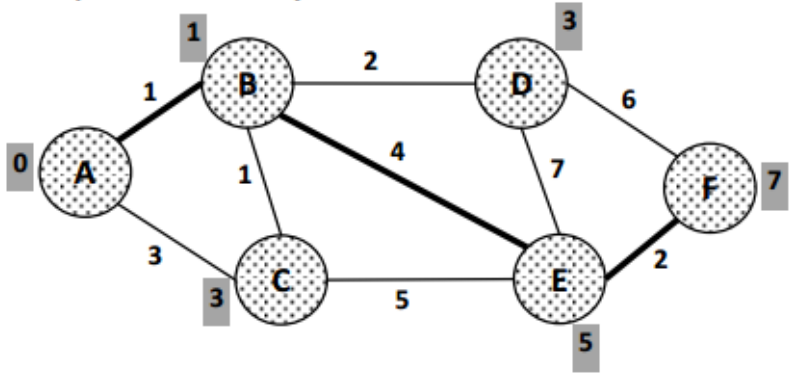
Step 3: Traverse all adjacent node of C



Step 4: Traverse all adjacent node of D



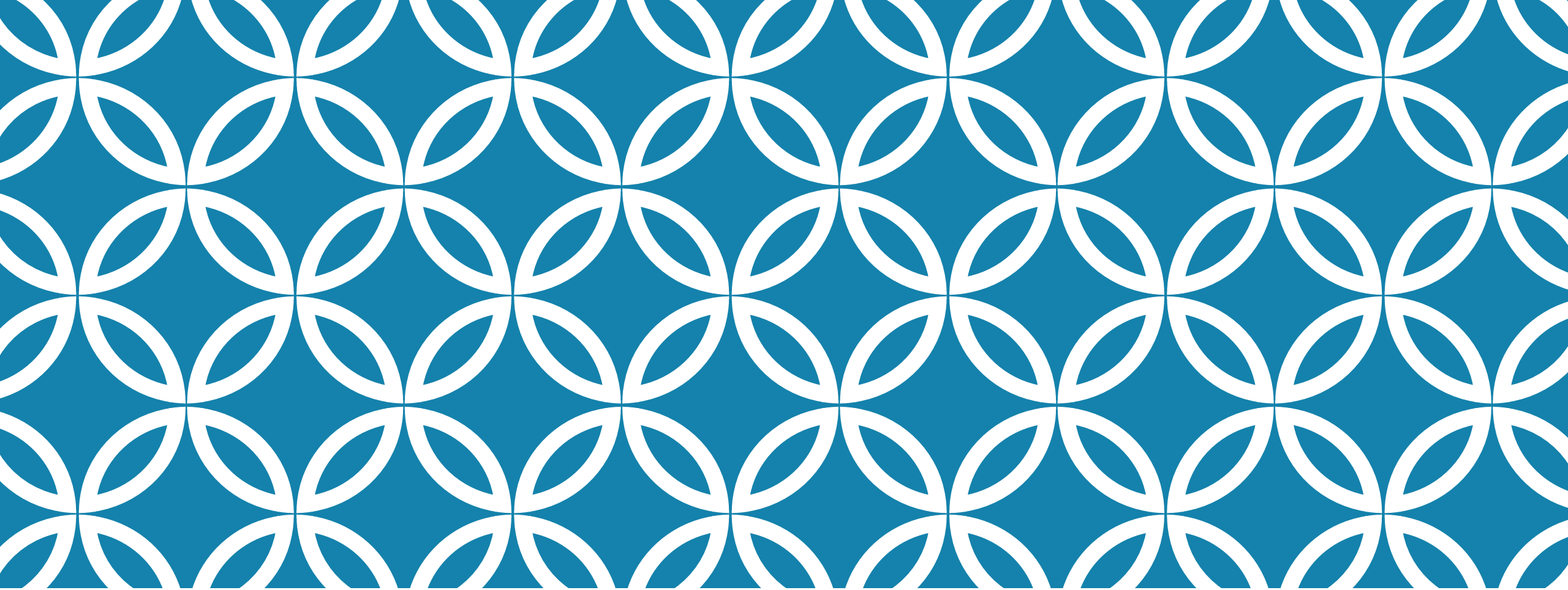
Step 5: Traverse all adjacent node of E



- Shortest path from node **A to F** is :
  A – B – E – F as shown in step 5
- Length of path is **7**

ASSIGNMENT

1. Define following word
    1) Connected Graph
    2) Cycle
    3) Degree of node
    4) Depth
    5) Directed Edge
2. What is BFS and DFS explain with example.
3. What is a graph? Discuss the Adjacency Matrix and Adjacency List representation of graphs with an example.
4. What is MST ? Explain with suitable example Kruskal's algorithm to find out MST.
5. Explain with suitable example Prim's algorithm to find out MST.



DATA STRUCTURE AND ALGORITHM

UNIT -4

Sorting and Searching

Prepared by: Prof. Kinjal Patel.

SORTING

- Sorting is the process of arranging the elements in some logical order.
- This logical order may be ascending or descending in case of numeric or dictionary order in case of alphanumeric values.

Types of Sorting:

- Insertion Sort
- Bubble sort
- Selection sort
- Quick Sort
- Merge Sort
- Heap Sort

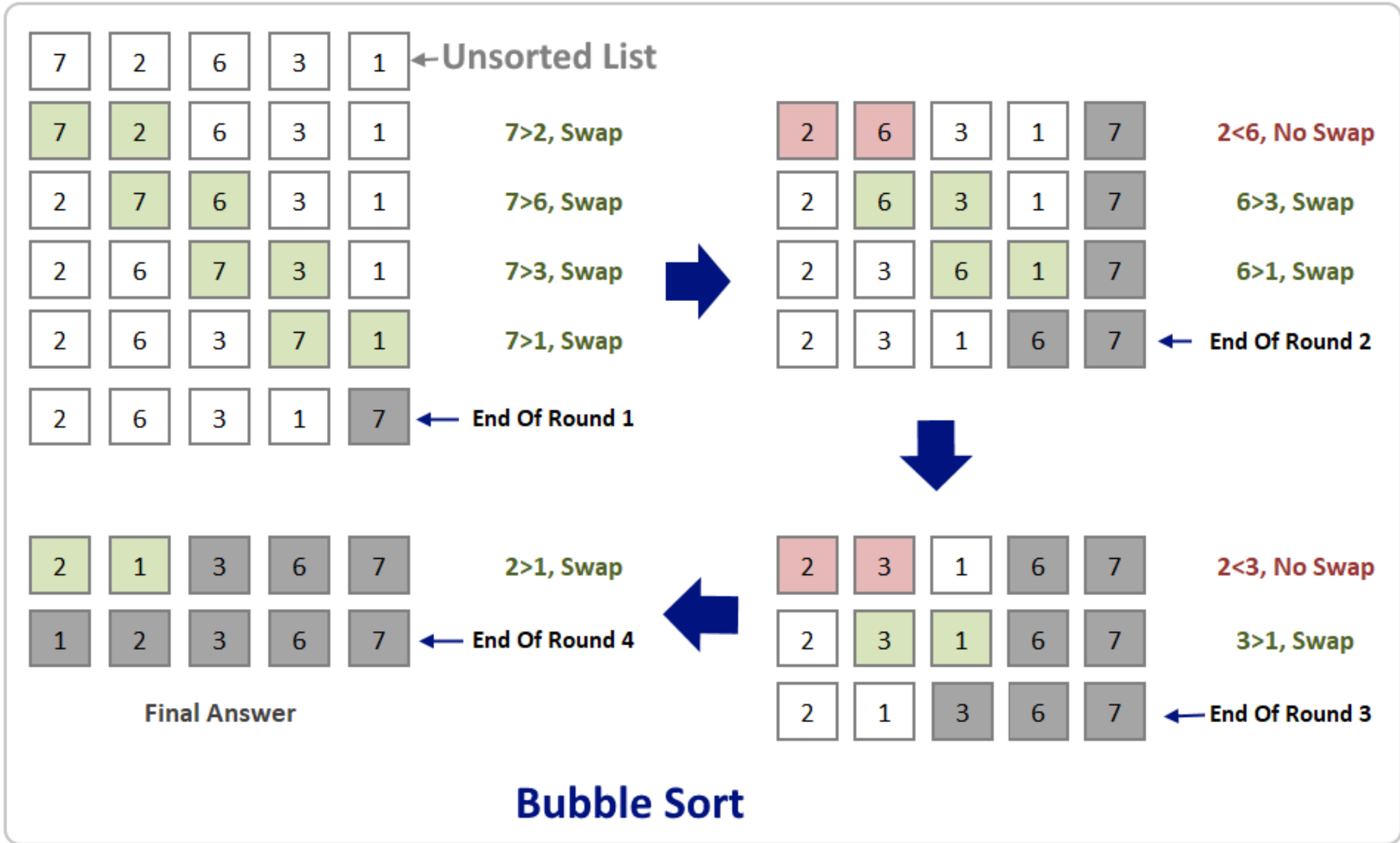
- https://www.hackerearth.com/practice/algorithms/sorting/insertion-sort/visualize/

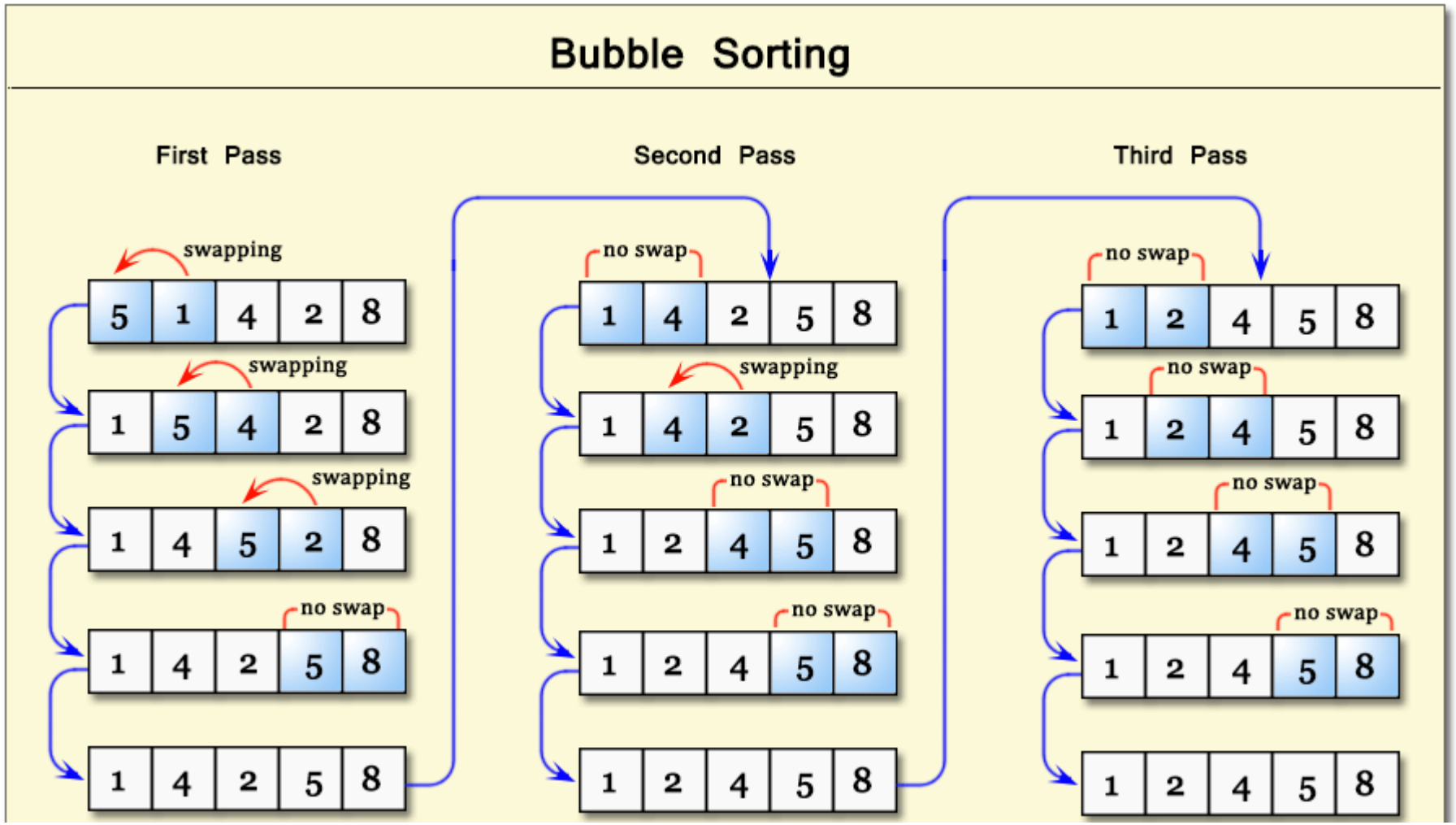
BUBBLE SORT

- Bubble sort is easy to understand and simple sorting technique.
- That's why it is the most widely known sorting method.
- But this sorting technique is not efficient in comparison to other sorting technique.
- In this technique, right elements of the list are comparing to left element of the list step by step up to the top position and finally It fix top position element.
- In bubble sort method total n-1 passes are required.
- During the first pass element 1 and 2 are compared and if they are out of order then they are interchanged. This process is repeated for elements 2 and 3 and so on.
- During first pass we required n-1 comparisons. During second pass we required n-2 comparisons and during i^{th} pass we required n-i comparisons. So total number of comparisons is given as:

$$\sum_{i=1}^{n-1} (n-i) = n(n-1)/2 = n2 - n$$

- Thus the order of comparisons is proportional to n^2 i.e $O(n^2)$


7 2 6 3 1 ← Unsorted List
7 2 6 3 1 7>2, Swap
2 7 6 3 1 7>6, Swap
2 6 7 3 1 7>3, Swap
2 6 3 7 1 7>1, Swap
2 6 3 1 7 ← End Of Round 1
2 6 3 1 7 2<6, No Swap
2 6 3 1 7 6>3, Swap
2 3 6 1 7 6>1, Swap
2 3 1 6 7 ← End Of Round 2
2 3 1 6 7 2<3, No Swap
2 3 1 6 7 3>1, Swap
2 1 3 6 7 ← End Of Round 3
2 1 3 6 7 2>1, Swap
1 2 3 6 7 ← End Of Round 4
Final Answer
Bubble Sort


Bubble Sorting
First Pass
Second Pass
Third Pass
swapping
swapping
swapping
no swap
5 1 4 2 8
1 5 4 2 8
1 4 5 2 8
1 4 2 5 8
1 4 2 5 8
no swap
swapping
no swap
no swap
1 4 2 5 8
1 4 2 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
no swap
no swap
no swap
no swap
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8

ALGORITHM OF BUBBLE SORT

BUBBLE_SORT (K,N)

- These function sorts elements of array K consist of N elements.
- PASS denotes the pass index.
- LAST denotes the last unsorted element.
- EXCHS counts the total number of exchange made during pass

1**[Initialize]**

LAST←N

2**[Loop on pass index]**

Repeat thru step 5 for PASS = 1, 2… N-1

3**[Initialize exchange counter for this pass]**

EXCHS←O

4**[Perform pair wise comparisons on unsorted elements]**

Repeat for I = 1, 2… LAST-1

If K [I] > K [I+1] then

K[I]↔K[I+1]

EXCHS←EXCHS+1

5**[Exchange made on this pass?]**

If EXCHS = 0 then

Write “Array already sorted”

Else

LAST←LAST-1

6**[Finished]**

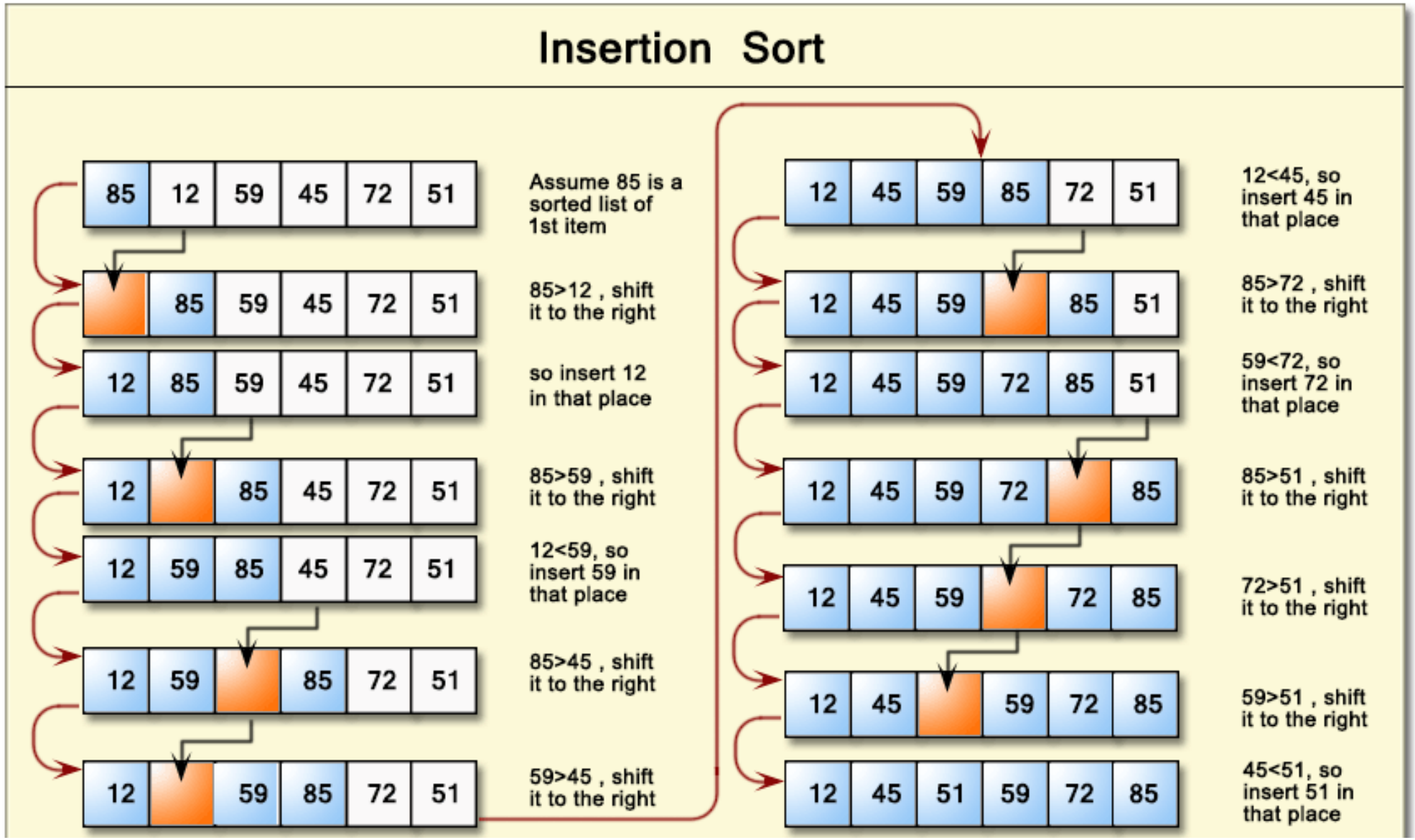
Return (Minimum number of pass required)

INSERTION SORT

- In this Algorithm One element from the list are picked and placed at its proper place by comparing it with previous element.
- The implementation of algorithm is simple and it is suitable for the small data set
- Insertion sort is an example of an **incremental** algorithm
- It builds the sorted sequence one number at a time
- This algorithm is suitable for playing card game.
- If there are *n* elements to be sorted. Then, this procedure is repeated *n-1* times to get sorted list of array.
- For each element A[i], if A[i] > A[i+1], swap the elements until A[i] ≤ *A*[*i*+1].
- In this algorithm to insert the last element we need at most n-1 comparisons and to insert the second last element we need at most n-2 comparisons and so on.
- So total number of comparisons is given as:

$$\sum_{i=1}^{n-1} (n-i) = n(n-1)/2$$

- Thus the order of comparisons is proportional to n^2 i.e $O(n^2)$


Insertion Sort
85 12 59 45 72 51
Assume 85 is a sorted list of 1st item
85 59 45 72 51
85>12 , shift it to the right
12 85 59 45 72 51
so insert 12 in that place
12 85 45 72 51
85>59 , shift it to the right
12 59 85 45 72 51
12<59, so insert 59 in that place
12 59 85 72 51
85>45 , shift it to the right
12 59 85 72 51
59>45 , shift it to the right
12 45 59 85 72 51
12<45, so insert 45 in that place
12 45 59 85 51
85>72 , shift it to the right
12 45 59 72 85 51
59<72, so insert 72 in that place
12 45 59 72 85
85>51 , shift it to the right
12 45 59 72 85
72>51 , shift it to the right
12 45 59 72 85
59>51 , shift it to the right
12 45 51 59 72 85
45<51, so insert 51 in that place

ALGORITHM OF INSERTION SORT

INSERTIONSORT(K,N)

- K is an array consist of N elements.

1. [Loop on Array]

   Repeat thru step 5 for I =2 to N

2. [Set temporary variable]

   TEMP = K [I]

   J= I - 1

3. [Compare elements]

   while (J >= 0 && K [J]> TEMP)

   K [J + 1] = K[J];

   J = J - 1

4. [Insert element to its proper place]

   K[J] = TEMP

5. [Finish]

   Exit

SELECTION SORT

In this method selection sort starts from first element and searches the entire array until it find smallest element. Then smallest value interchanges with the first element.

Now select second element and searches for the second smallest element from the array, if found then interchange with second element.

So in this method after pass 1 smallest value arranged at first position then after pass 2 second minimum will arrange at second position and so on.

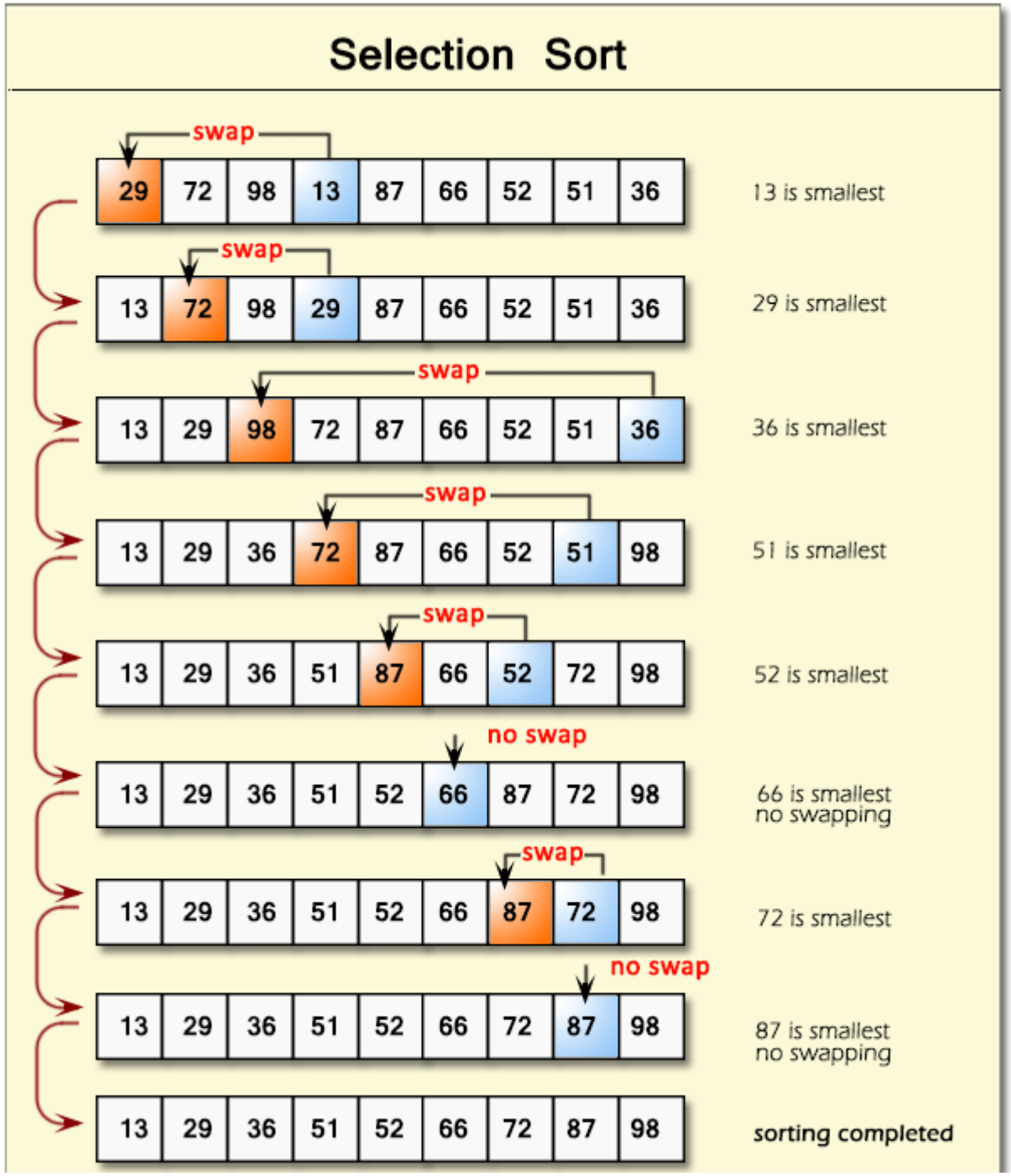
This process continues until all the elements in the array are arranged in ascending order.

During first pass we required n-1 comparisons. During second pass we required n-2 comparisons and during i^{th} pass we required n-i comparisons. So total number of comparisons is given as:

$$\sum_{i=1}^{n-1} (n-i) = n(n-1)/2$$

Thus the order of comparisons is proportional to n^2 i.e $O(n^2)$

Selection Sort



ALGORITHM OF SELECTION SORT

SELECTION_SORT(K,N)

- These function sorts elements of array K consist of N elements.
- PASS denotes the pass index and the position of the first element in the vector which is to be examined during particular pass.
- MIN_INDEX denotes the position of the smallest element encountered thus far in a particular pass.

1.[Loop on pass index]

Repeat thru step 4 for PASS = 1, 2, 3… N-1

2.[initialize minimum index]

MIN_INDEX ←PASS

3.[Make a pass and obtain element with the smallest value]

Repeat for I = PASS+1 to N

If K [I] < K [MIN_INDEX] then

MIN_INDEX←I

4.[Exchange elements]

If MIN_INDEX ≠ PASS then

K[PASS]⟷ K[MIN_INDEX]

5.[finished]

Return

https://visualgo.net/en/sorting

MERGE SORT

- Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm
- ***Divide*** means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements
- ***Conquer*** means sorting the two sub-arrays recursively using merge sort.
- ***Combine*** means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.
- The basic steps of a merge sort algorithm are as follows:
  - If the array is of length 0 or 1, then it is already sorted.
  - Otherwise, divide the unsorted array into two sub-arrays of about half the size.
  - Use merge sort algorithm recursively to sort each sub-array.
  - Merge the two sub-arrays to form a single sorted list.
- There are two types of Merge sort
  - **Simple Merge sort** means to merge two sorted list as one sorted list
  - **Two way Merge sort** means to first divide the unordered list in two sub list recursively and then Merge sorted sub list as sorted main list
- It needs Additional Memory to store temporary array.

SIMPLE MERGE SORT

MergeSortArray (A,LB,UB,MID)

- A is a linear array. LB and UB are lower bound and upper bound of an array.
- TEMP is a linear array.

[1] [Initialize variable]

I = LB

J = MID + 1

K = 1

[2] [Compare corresponding elements and output the smallest]

Repeat while I<=MID and J <= UB

If A [I] ≤ A [J] then

TEMP [K] ← A [I]

K = K +1

I = I + 1

Else

TEMP [K] ← A [J]

K = K +1

J = J + 1

[3] [Copy the remaining unprocessed elements in output area]

If I > MID then

Repeat while J ≤ UB

TEMP [K] ← A [J]

K = K + 1

J = J + 1

Else

Repeat while I <= MID

TEMP [K] ← A [I]

K = K +1

I = I + 1

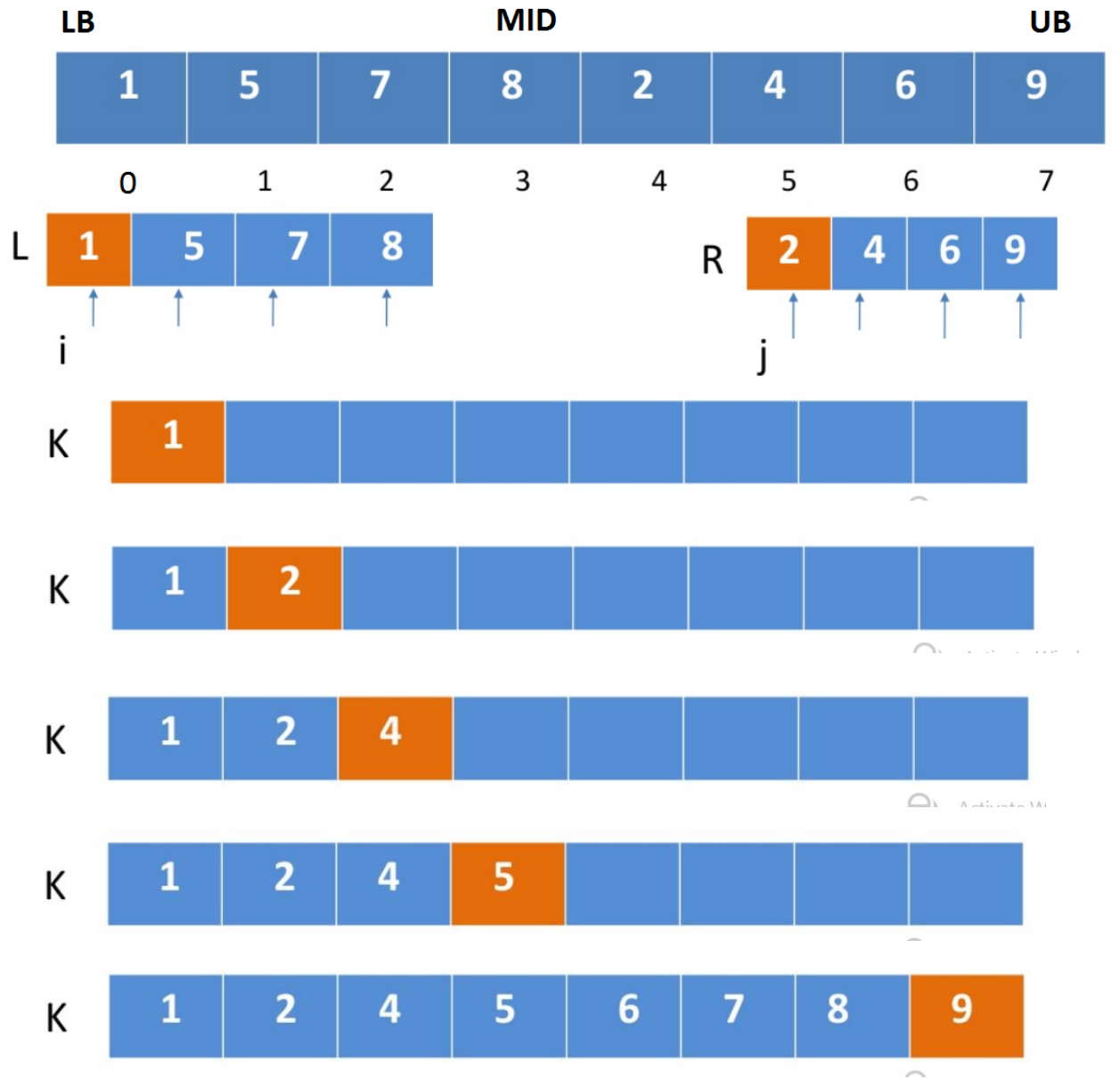
[4] [Copy elements in temporary vector into original vector]

Repeat for I = 1 to UB

A[I] = TEMP [I]

[5] [Finished]

Return


LB
MID
UB
1 5 7 8 2 4 6 9
0 1 2 3 4 5 6 7
L
1 5 7 8
R
2 4 6 9
i
j
K
1
K
1 2
K
1 2 4
K
1 2 4 5
K
1 2 4 5 6 7 8 9

TWO WAY MERGE SORT

MergeSort(A, LB, UB)

- A is a linear array. LB and UB are lower bound and upper bound of an array.

[1] [Compute MID point]

MID = (LB + UB)/2

[2] [Return if only one element]

IF (LB > UB)

Return

[3] [Recursive call for first sub list]

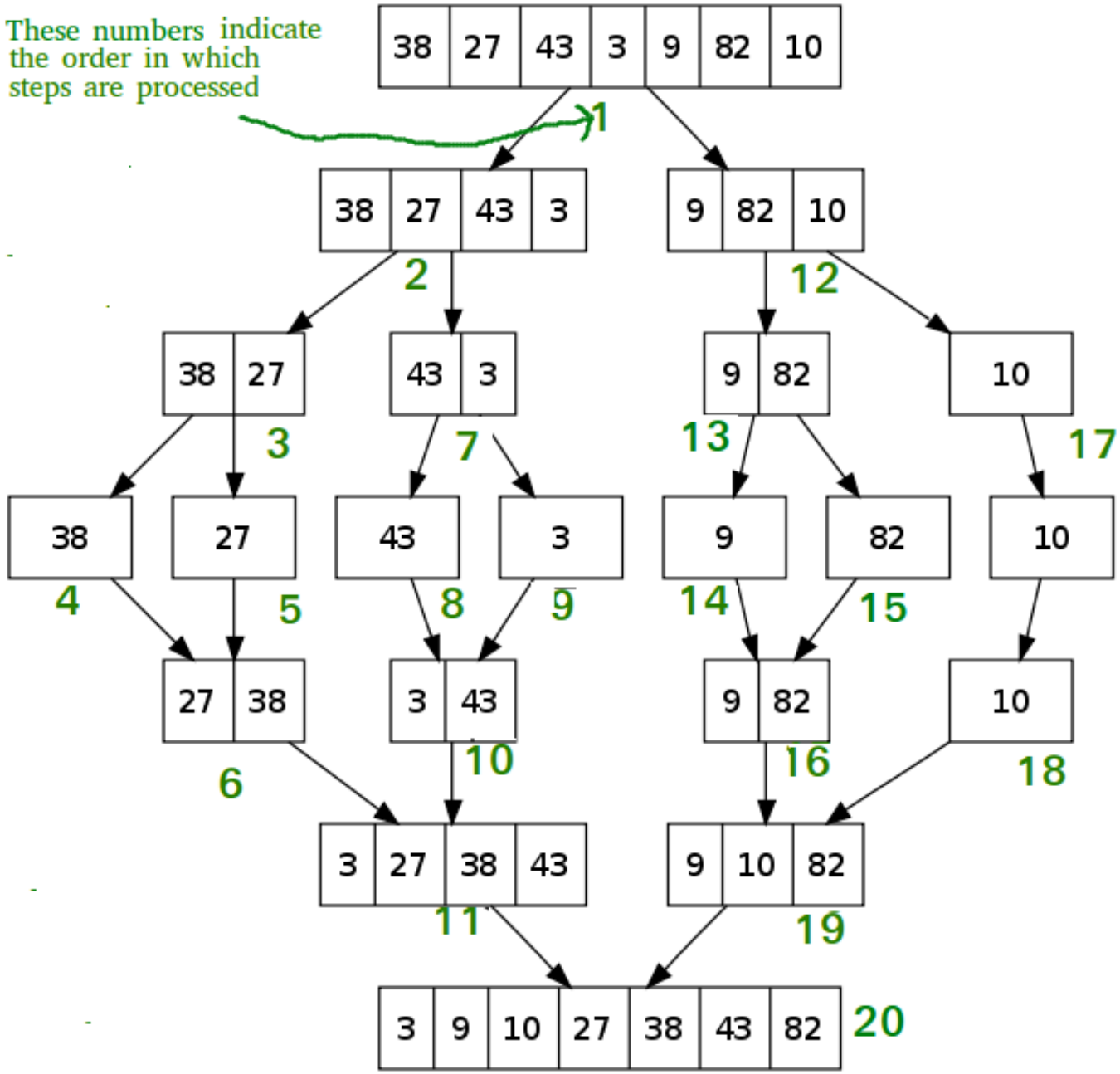
Call MergeSort(A,LB,MID)

[4] [Recursive call for second sublist]

Call MergeSort(A, MID+1, UB)

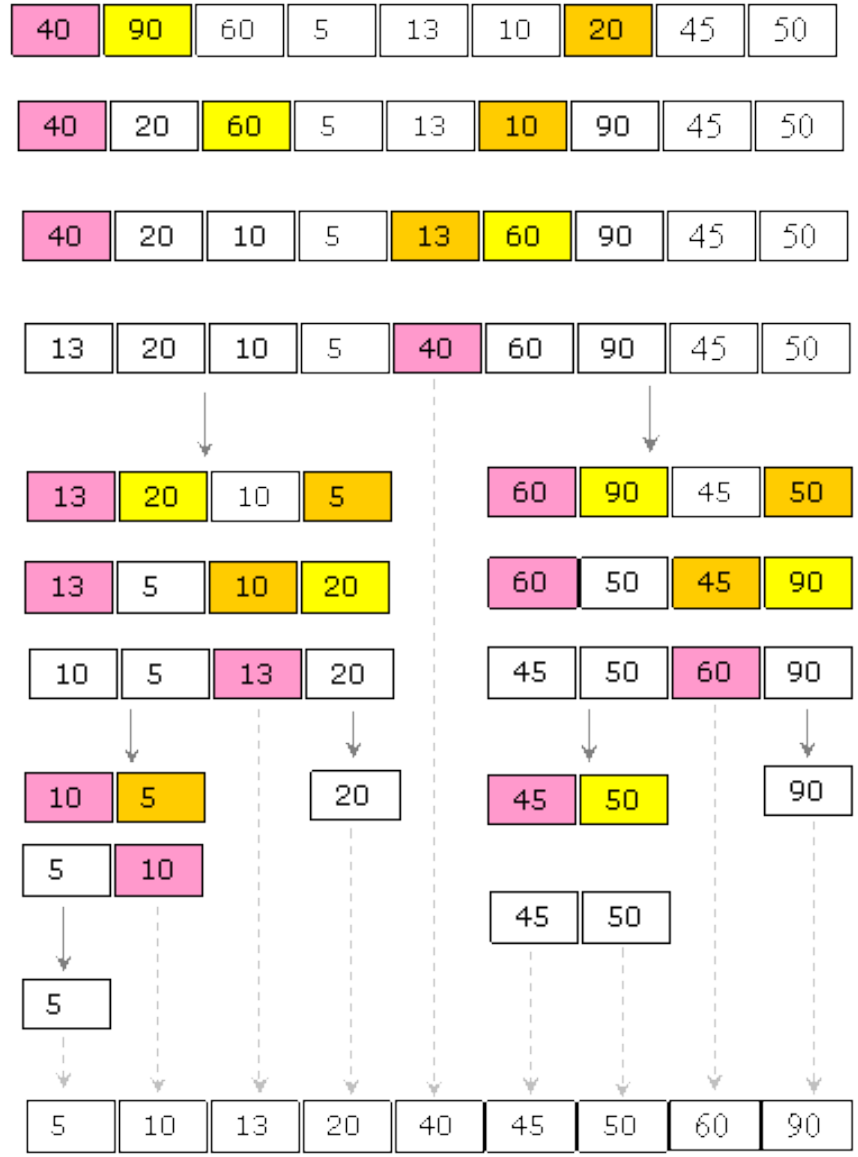
[5] [Merge two sublists]

Call MergeSortArray (A,LB,UB,MID)


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

QUICK SORT

- In this method we pick the first element (pivot element) from the array and place it in it's proper position in the array such that all the elements preceding the element having smaller value and all the elements following the element having larger value.
- Thus the table is divided into two sub tables.
- The same process is then applied to each of the sub tables and repeated until all records are placed in their final position.
- This is fastest sorting algorithm on an average
- The order of comparisons is proportional to n^2 i.e $O(n^2)$ in worst case
- The order of comparisons is proportional to $n \log_2 n$ i.e $O(n \log_2 n)$ in best case

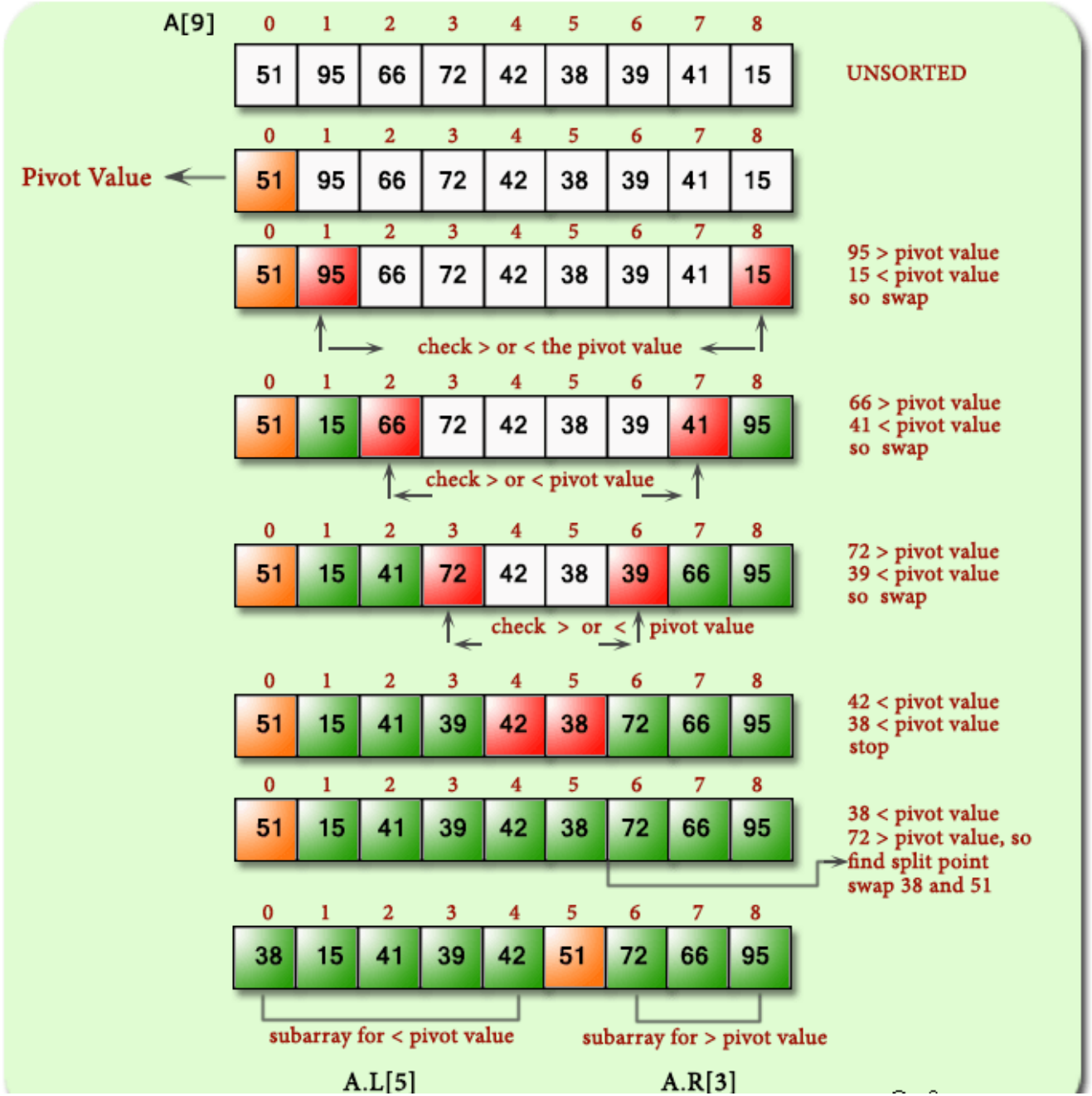


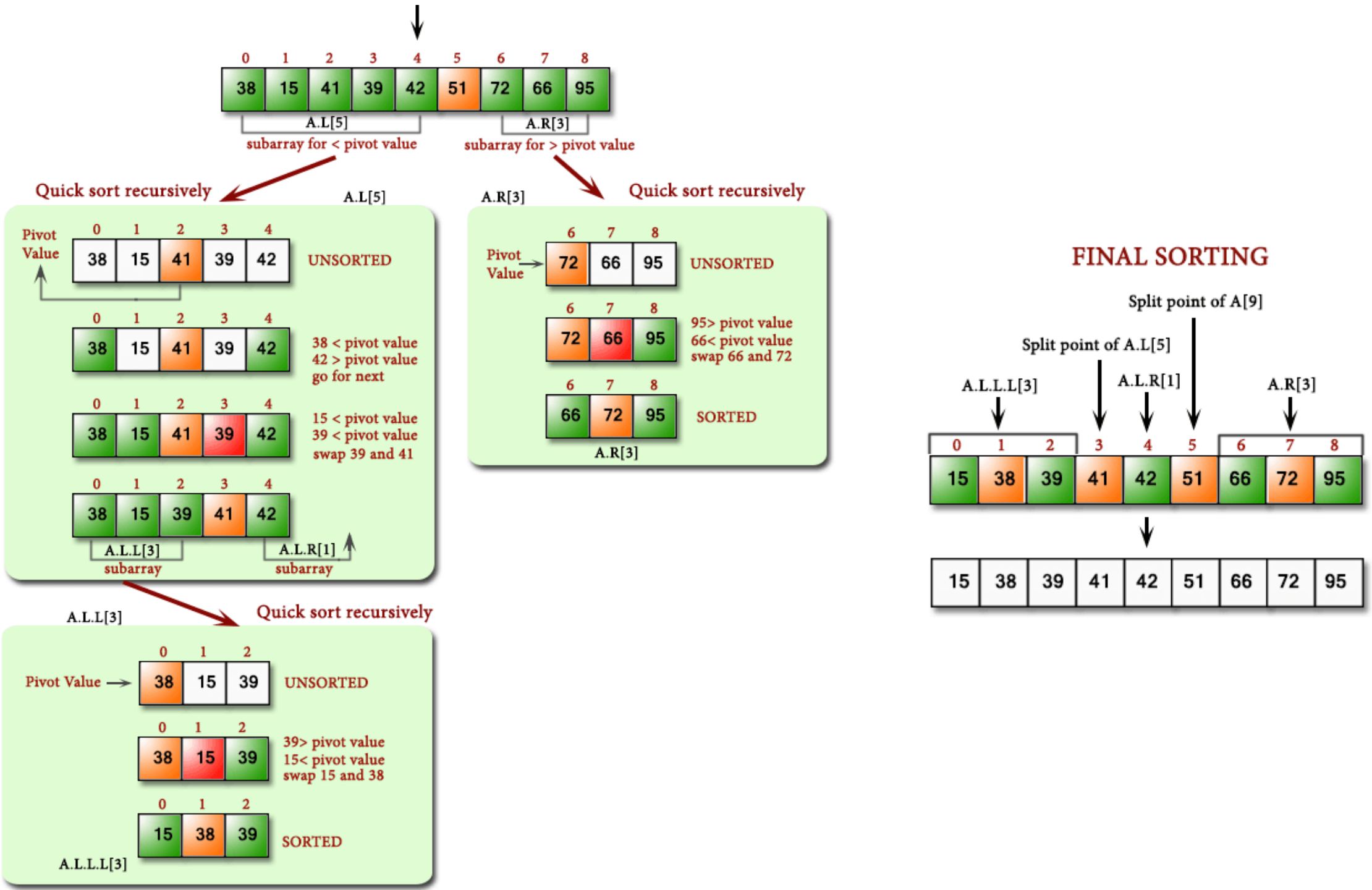
Pivot/Split element

Greater than pivot from start

Less than pivot from end

Quick Sort


A[9]
0 1 2 3 4 5 6 7 8
51 95 66 72 42 38 39 41 15
UNSORTED
Pivot Value
95 > pivot value
15 < pivot value
so swap
check > or < the pivot value
66 > pivot value
41 < pivot value
so swap
check > or < pivot value
72 > pivot value
39 < pivot value
so swap
check > or < pivot value
42 < pivot value
38 < pivot value
stop
38 < pivot value
72 > pivot value, so
find split point
swap 38 and 51
38 15 41 39 42 51 72 66 95
subarray for < pivot value
subarray for > pivot value
A.L[5]
A.R[3]


0 1 2 3 4 5 6 7 8
38 15 41 39 42 51 72 66 95
A.L[5]
subarray for < pivot value
A.R[3]
subarray for > pivot value
Quick sort recursively
A.L[5]
Pivot Value
UNSORTED
38 < pivot value
42 > pivot value
go for next
15 < pivot value
39 < pivot value
swap 39 and 41
A.L.L[3]
subarray
A.L.R[1]
subarray
A.R[3]
Quick sort recursively
Pivot Value
UNSORTED
95> pivot value
66< pivot value
swap 66 and 72
SORTED
A.R[3]
A.L.L[3]
Quick sort recursively
Pivot Value
UNSORTED
39> pivot value
15< pivot value
swap 15 and 38
SORTED
A.L.L.L[3]
FINAL SORTING
Split point of A[9]
Split point of A.L[5]
A.L.L.L[3]
A.L.R[1]
A.R[3]
15 38 39 41 42 51 66 72 95
15 38 39 41 42 51 66 72 95

ALGORITHM OF QUICK SORT

QUICK_SORT(K,LB,UB)

- These function sorts elements of array K consist of N elements.
- LB and UB denotes the lower and upper bound of the sub table being processed.
- FLAG is a logical variable.
- KEY is the value which is placed in its final position during pass.

1. [Initialize]
FLAG←true
2. [Perform Sort]
If LB < UB then
I←LB
J←UB
KEY←K [LB]
Repeat while FLAG
I←I+1
Repeat while K [I] < KEY
I←I+1
Repeat while K [J] > KEY
J←J-1
If I < J then
K[I] ⟷ K[J]
Else
FLAG←false
K[LB] ⟷ K[J]
Call QUICK_SORT (K, LB, J-1)
Call QUICK_SORT (K, J+1, UB)
3.[Finished]

HEAP SORT

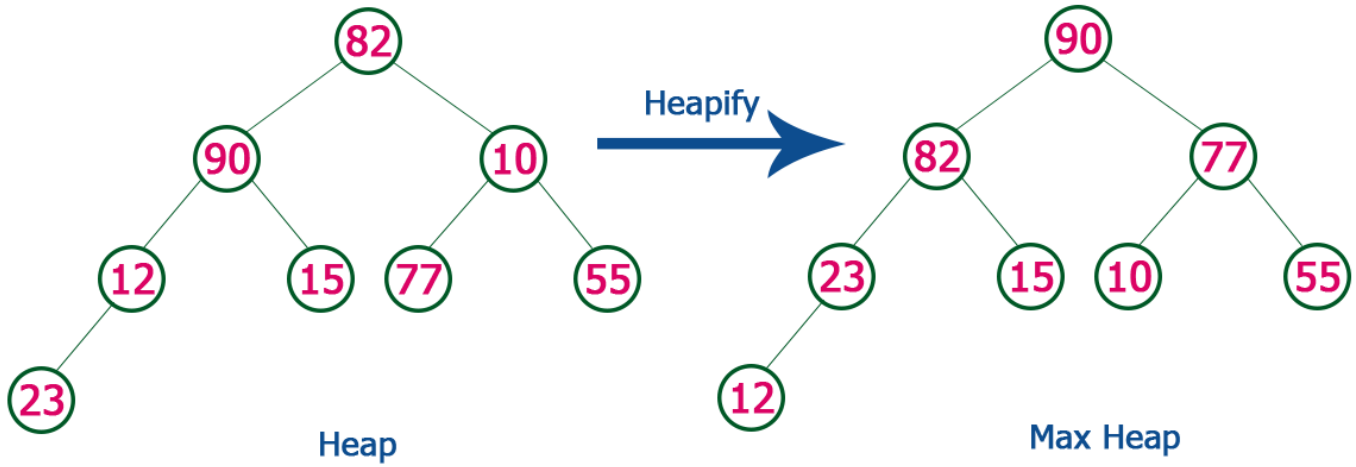
- Heap sort is one of the sorting algorithms used to arrange a list of elements in order. Heapsort algorithm uses one of the tree concepts called **Heap Tree**. In this sorting algorithm, we use **Max Heap** to arrange list of elements in Descending order and **Min Heap** to arrange list elements in Ascending order.

There are two phases involved in the sorting of elements using heap sort algorithm they are as follows:

- First, start with the construction of a heap by adjusting the array elements.
- Once the heap is created repeatedly eliminate the root element of the heap by shifting it to the end of the array and then store the heap structure with remaining elements.
- **Time Complexity:** O(n log n)

82, 90, 10, 12, 15, 77, 55, 23

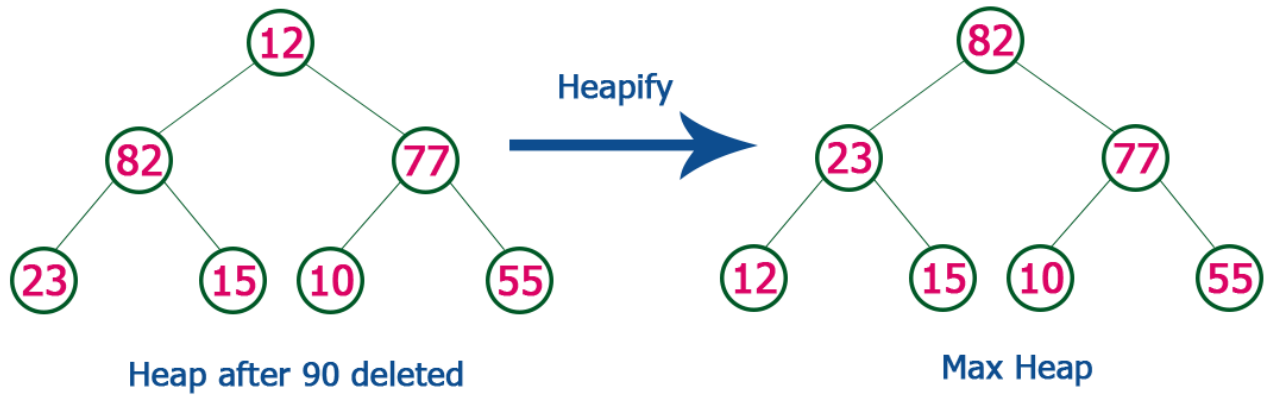
Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

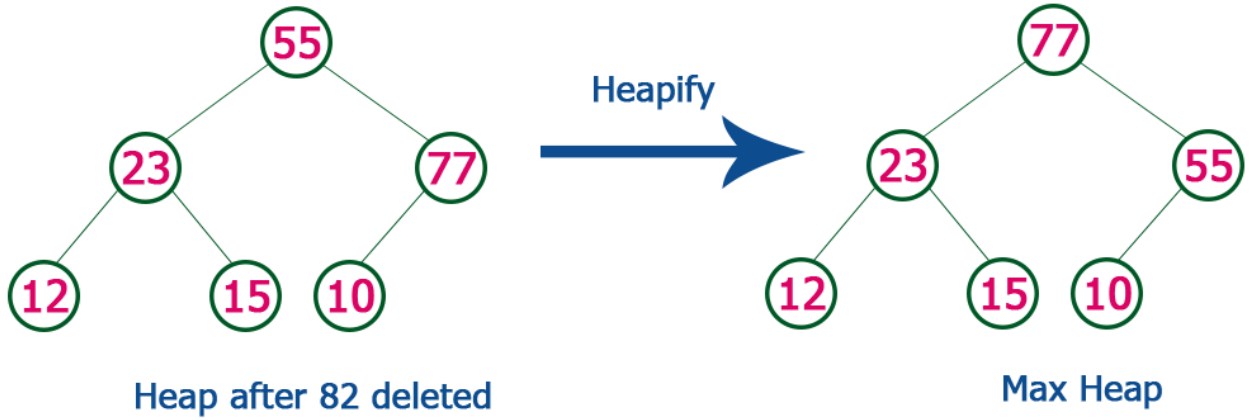
Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, **90**

Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.



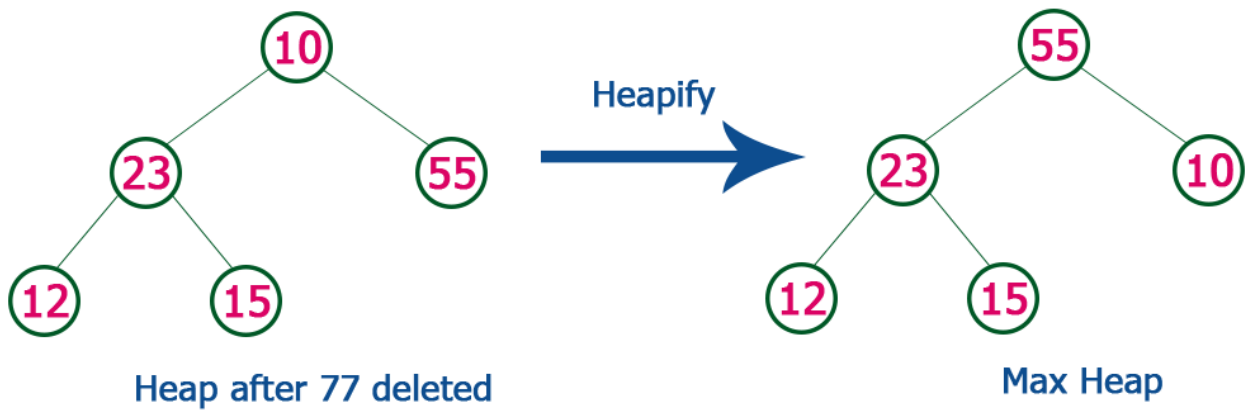
Heap after 82 deleted

Max Heap

list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, **82**, **90**

Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



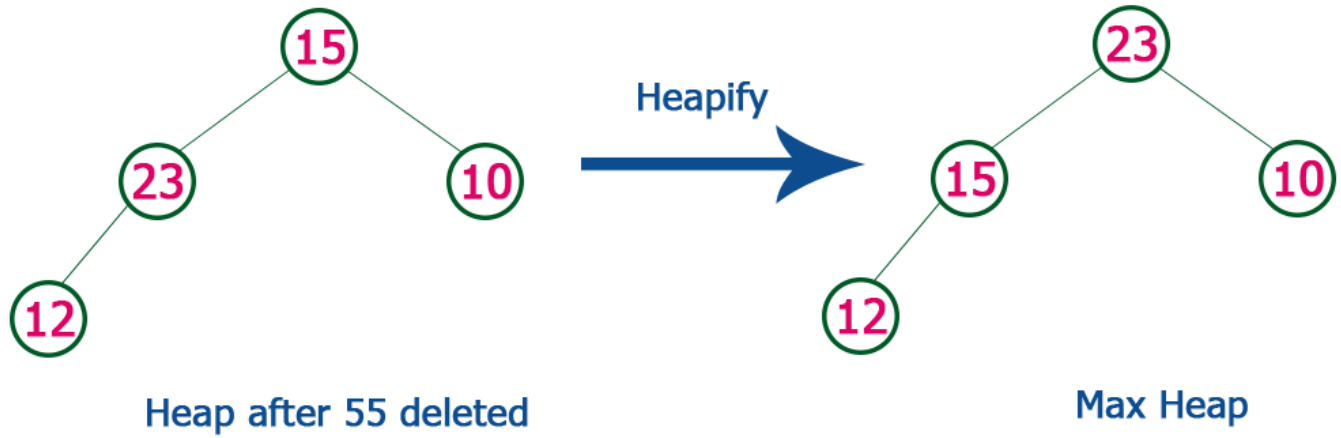
Heap after 77 deleted

Max Heap

list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, **77**, **82**, **90**

Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 55 with 15.

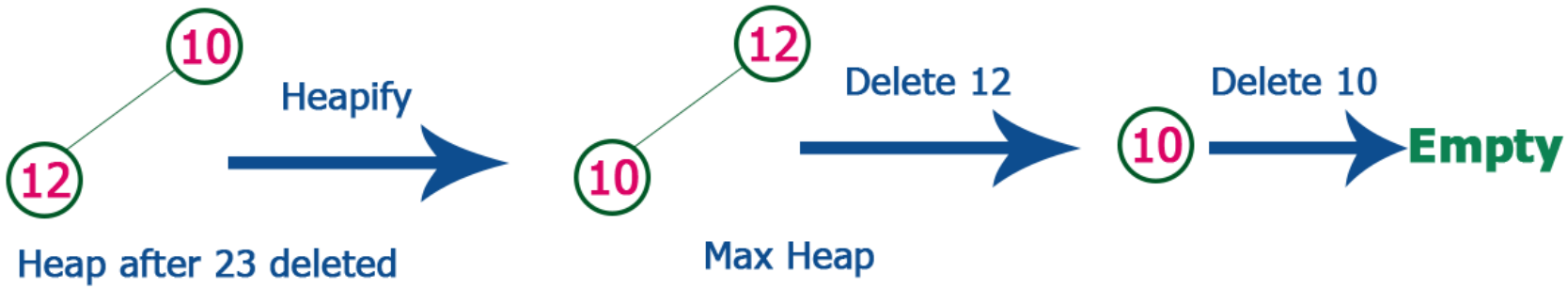
12, 15, 10, 23, **55, 77, 82, 90**

Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 23 with 12.

12, 15, 10, **23, 55, 77, 82, 90**

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after Deleting 15, 12 & 10 from the Max Heap.

10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

LINEAR SEARCH

- This is also known as sequential search
- This method is used to find out element in an unordered list.
- Suppose the list L consist of N elements and we want to find the element from the list.
- In this technique first the value of the element is compared with the value of the first element in the list, if match is found then the search is terminated.
- If match is not found then next element from the list is fetched and compared with the element.
- This process is repeated until match is found or all the element in the list is compared.
- Time complexity O(n)

ALGORITHM OF LINEAR SEARCH

LINEAR_SEARCH(K, N, X)

- These function searches the list K consist of N elements for the value of X.

1**[Initialize search]**

I←1

K [N+1] ←X

2**[Search the Vector]**

Repeat while K [I] ≠ X

I←I+1

3**[Successful Search?]**

If I = N+1 then

Write “Unsuccessful Search”

Return 0

Else

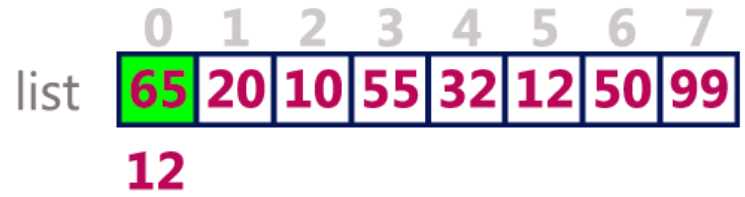
Write “Successful Search”

Return I

search element **12**

Step 1:

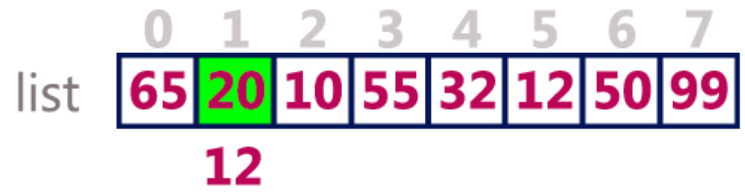
search element (12) is compared with first element (65)



Both are not matching. So move to next element

Step 2:

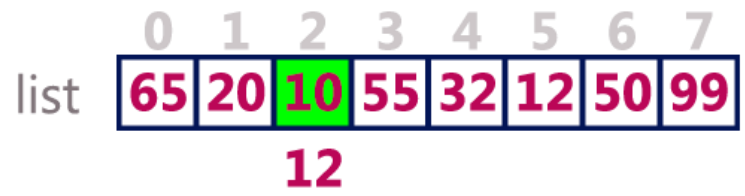
search element (12) is compared with next element (20)



Both are not matching. So move to next element

Step 3:

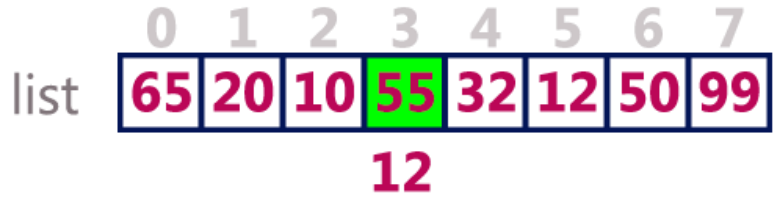
search element (12) is compared with next element (10)



Both are not matching. So move to next element

Step 4:

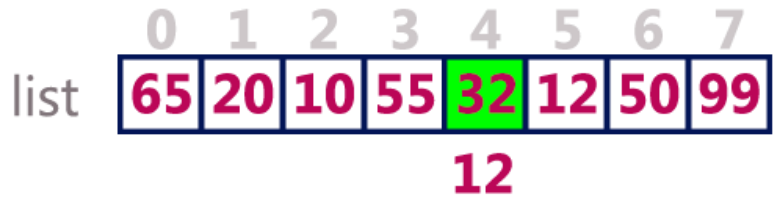
search element (12) is compared with next element (55)



Both are not matching. So move to next element

Step 5:

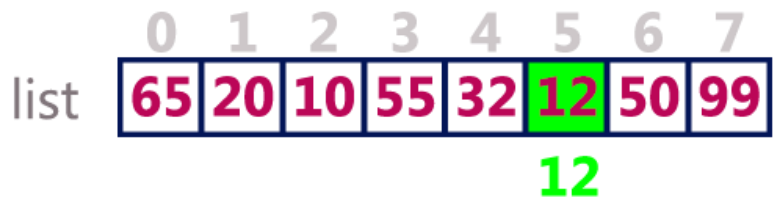
search element (12) is compared with next element (32)



Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)



Both are matching. So we stop comparing and display element found at index 5.

BINARY SEARCH

- This method is used to find out element in an ordered list.
- It is very efficient method to search element.
- Let low represents lower limit of the list and high represents the higher limit of the list.
- First we calculate the value of middle as:
- Mid = (Low + High)/2
- Now we compare the value of the middle element with the element to be searched.
- If the value of the middle element is greater then the element to be searched then the element will exist in lower half of the list. So take high = mid – 1 and find value of the middle in this new interval.
- If the value of the middle element is smaller then the element to be searched then the element will exist in upper half of the list. So take low = mid + 1 and find value of the middle in this new interval.
- This process is repeated until entire list is searched or the element is found.

ALGORITHM OF BINARY SEARCH

BINARY_SEARCH(K, N, X)

- These function searches the list K consist of N elements for the value of X.
- LOW, HIGH and MIDDLE denotes the lower, upper and middle limit of the list.

1**[Initialize]**

LOW←1

HIGH←N

2**[Perform Search]**

Repeat thru step 4 while LOW ≤ HIGH

3**[Obtain Index of midpoint interval]**

MIDDLE ←[(LOW + HIGH)/2]

4**[Compare]**

If X < K [MIDDLE] then

HIGH ← MIDDLE – 1

Else if X > K [MIDDLE] then

LOW←MIDDLE+1

Else

Write “Successful Search”

Return (MIDDLE)

5**[Unsuccessful Search]**

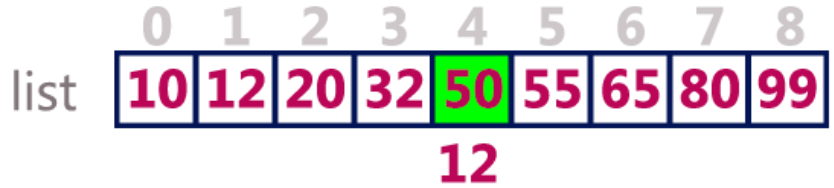
Write “Unsuccessful Search”

6**Return 0**

search element **12**

Step 1:

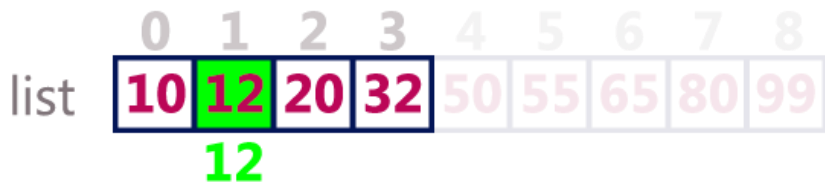
search element (12) is compared with middle element (50)



Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:

search element (12) is compared with middle element (12)

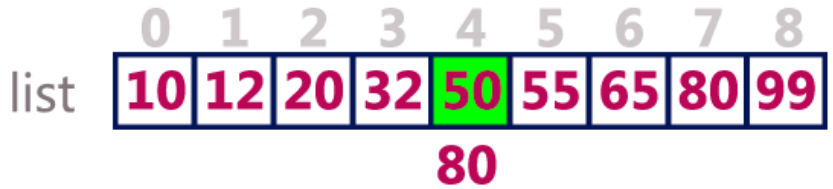


Both are matching. So the result is "Element found at index 1"

search element **80**

Step 1:

search element (80) is compared with middle element (50)



Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

Step 2:

search element (80) is compared with middle element (65)

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

Step 3:

search element (80) is compared with middle element (80)

Sequential Search	Binary Search
(1) Sequential Search is used to find elements from unordered list	(1) Binary Search is used to find elements from ordered list.
(2) This technique is less efficient	(2) This technique is more efficient
(3) The order of sequential search is O(N).	(3) The order of Binary Search is $O(Log_2N)$