7.1 Introduction to data structure, list, linked lists and trees

1. Data types,
2. Data structures and abstract data types;
3. Time and space analysis of algorithms (big oh, omega and theta notations),
4. Linear data structure (stack and queue implementation);
5. Stack application: infix to postfix conversion, and evaluation of postfix expression,
6. Array implementation of lists;
7. Stack and queues as list; and static list structure,
8. Static and dynamic list structure;
9. Dynamic implementation of linked list;
10. Types of linked list: singly linked list, doubly linked list, and circular linked list;
11. Basic operations on linked list: creation of linked list, insertion of node in different positions, and deletion of nodes from different positions;
12. Doubly linked lists and its applications,
13. Concept of tree, operation in binary tree,
14. Tree search, insertion/deletions in binary tree,
15. Tree traversals (pre-order, post-order and in-order),
16. Height, level and depth of a tree,
17. AVL balanced trees.

Data Types

- A data type in programming, is a classification that specifies which type of value a variable has and what type of mathematical, relational or logical operations can be applied to it.
- A data type is the collection of values and a set of operations on the values.
- For example:
  - A string is a data type that is used to classify text.
  - An integer is a data type that is used to classify whole numbers.

Data Structures

- Data is a collection of raw facts on computer in digital form.
- A method of organizing information so that the information can be stored and retrieved efficiently.
- It is about rendering data elements in terms of relationship for better organization and storage.
- The structure not only stores data, but also supports operations for accessing and manipulating the data.
- Computer programmers decide which data structures to use based on the nature of the data and the processes that need to be performed on that data.
- Data structure affects the design of both structural & functional aspects of a program.
- Program=Algorithm + Data Structure

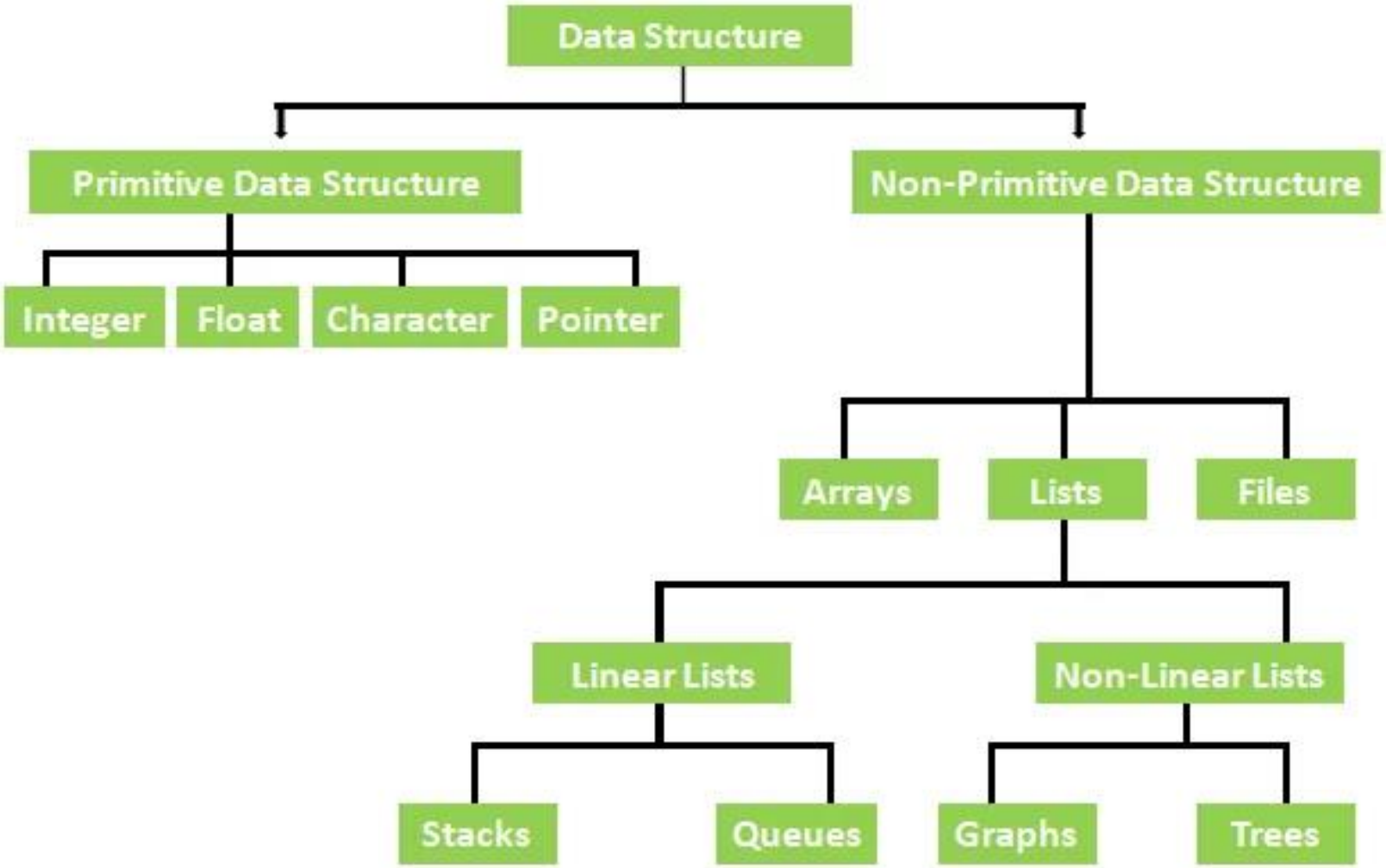
Need of Data Structures

- As applications are getting complexed and amount of data is increasing day by day, there may arise the following problems:
- **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.

Data Structure Operations

- **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.
- **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.
- **Insertion:** We can also insert new element in a data structure. If the size of data structure is **n** then we can only insert **n-1** data elements into it.
- **Updation:** We can also update the element, i.e., we can replace the element with another element.
- **Deletion:** We can also perform the delete operation to remove the element from the data structure.
- **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.

TYPES OF DATA STRUCTURE



There are two types of data structures:

i. Primitive data structure:

- These are the structures which are supported at the machine level, they can be used to make non-primitive data structures.
- They are primary data types, that are already defined or can be represented by one keyword.
- These are integral and are pure in form. They have predefined behavior and specifications.

- Examples: Integer, float, character, pointers.

- The pointers, however don't hold a data value, instead, they hold memory addresses of the data values. These are also called the reference data types.

ii. Non-primitive data structure:

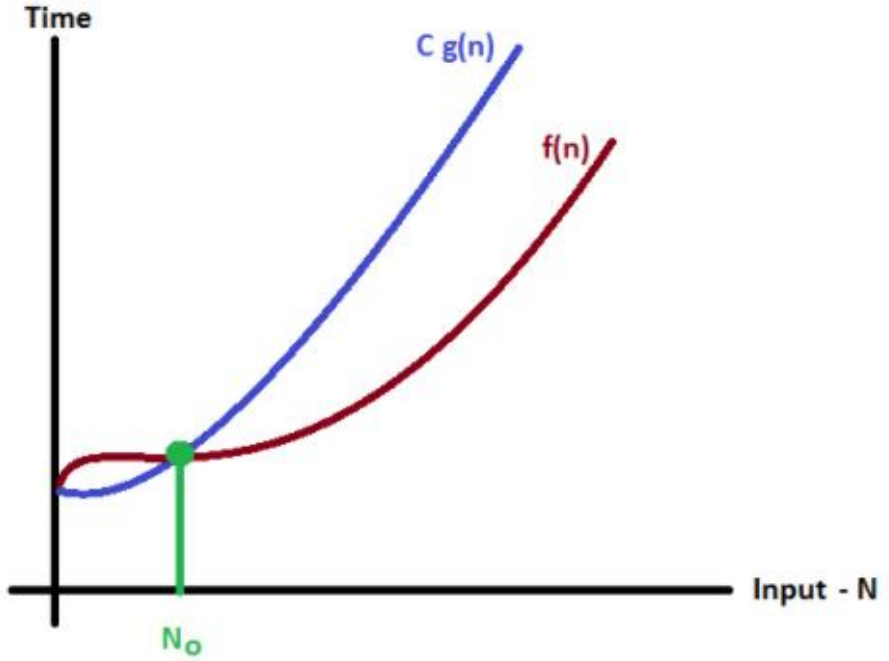
- The non-primitive data structures cannot be performed without the primitive data structures.
- It is a type of data structure that can store the data of more than one type.
- Although, they too are provided by the system itself yet they are derived data structures and cannot be formed without using the primitive data structures.

Asymptotic Notations

- While analyzing the complexity of any algorithm in terms of time and space, we can never provide an exact number to define the time required and the space required by the algorithm, instead we express it using some standard notations, also known as **Asymptotic Notations**.
- They are the expressions that are used to represent the complexity of an algorithm.
- They are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.
- Usually, the time required by an algorithm comes under three types:
- **Worst case:** It defines the input for which the algorithm takes a huge time.
- **Average case:** It takes average time for the program execution.
- **Best case:** It defines the input for which the algorithm takes the lowest time
- The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:
- **Big oh Notation (O)**
- **Omega Notation (Ω)**
- **Theta Notation (θ)**

1. Big oh Notation (O) :

- Big Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.
- It always indicates the maximum time required by an algorithm for all input values. It describes the worst case of an algorithm time complexity.
- **Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as O(g(n)).**
- **f(n) = O(g(n))**
- This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n). In this case, we are calculating the growth rate of the function which eventually calculates the worst time complexity of a function, i.e., how worst an algorithm can perform.
- In below graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.



Example : f(n)=2n+3 , g(n)=n

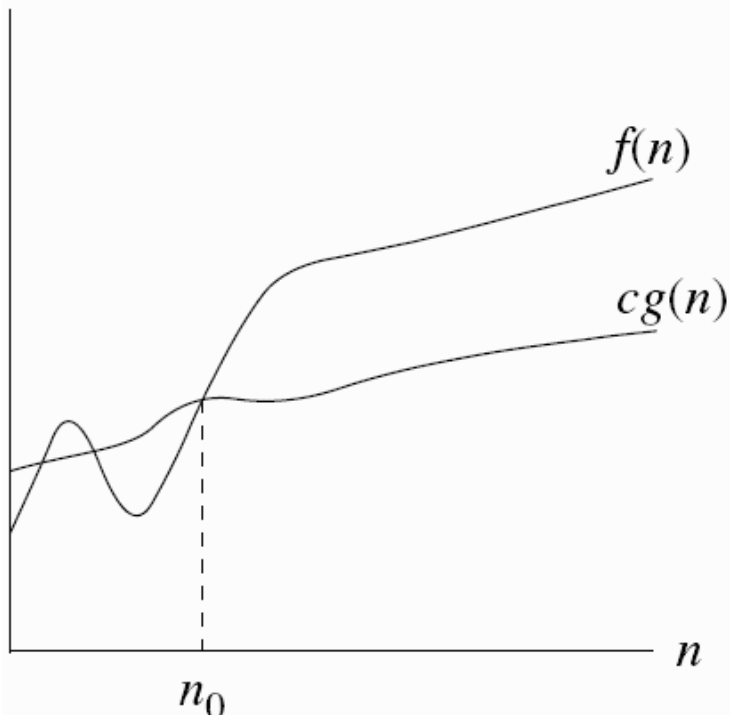
- Now, we have to find **Is f(n)=O(g(n))?**
- To check f(n)=O(g(n)), it must satisfy the given condition: **f(n)<=c.g(n)**
- First, we will replace f(n) by 2n+3 and g(n) by n.

→2n+3 <= c.n

- Let's assume c=5, n=1 then: 2*1+3<=5*1 i.e, 5<=5
- For n=1, the above condition is true.
- If n=2 then 2*2+3<=5*2 i.e, 7<=10
- For n=2, the above condition is true.
- We know that for any value of n, it will satisfy the above condition, i.e., 2n+3<=c.n. If c=5, then it will satisfy the condition 2n+3<=c.n. We can take any value of n starting from 1, it will always satisfy. Therefore, we can say that for some constants c and for some constants n0, it will always satisfy 2n+3<=c.n. As it is satisfying the above condition, so f(n) is big oh of g(n) or we can say that f(n) grows linearly. Therefore, it concludes that c.g(n) is the upper bound of the f(n).
- **The idea of using big o notation is to give an upper bound of a particular function, and eventually it leads to give a worst-time complexity. It provides an assurance that a particular function does not behave suddenly as a quadratic or a cubic fashion, it just behaves in a linear manner in a worst-case.**

2. Big Omega Notation (Ω)

- It basically describes the best-case scenario which is opposite to the big oh notation.
- It is the formal way to represent the lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity.
- It determines what is the fastest time that an algorithm can run.
- **Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω(g(n)).**
- **f(n) = Ω(g(n))**
- If we required that an algorithm takes at least certain amount of time without using an upper bound, we use big- Ω notation. It is used to bound the growth of running time for large input size.

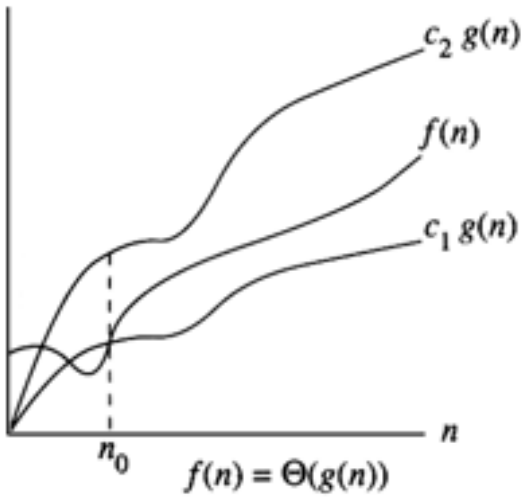


Example of Big omega notation:

- If f(n) = 2n+3, g(n) = n,
- Is f(n)= **Ω** (g(n))?
- It must satisfy the condition: **f(n)>=c.g(n)**
- To check the above condition, we first replace f(n) by 2n+3 and g(n) by n.
- **2n+3>=c*n**
- Suppose c=1
- **2n+3>=n** (This equation will be true for any value of n starting from 1).
- Therefore, it is proved that g(n) is big omega of 2n+3 function.

3. Theta Notation (θ)

- The theta notation mainly describes the average case scenarios.
- It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm.
- Big theta is mainly used when the value of worst-case and the best-case is same.
- It is the formal way to express both the upper bound and lower bound of an algorithm running time.
- **Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If C_1 g(n) <= f(n) <= C_2 g(n) for all n >= n_0, C_1 > 0, C_2 > 0 and n_0 >= 1. Then we can represent f(n) as Θ(g(n)).**
- **f(n) = Θ(g(n))**



- In above graph after a particular input value n_0, always C_1 g(n) is less than f(n) and C_2 g(n) is greater than f(n) which indicates the algorithm's average bound.

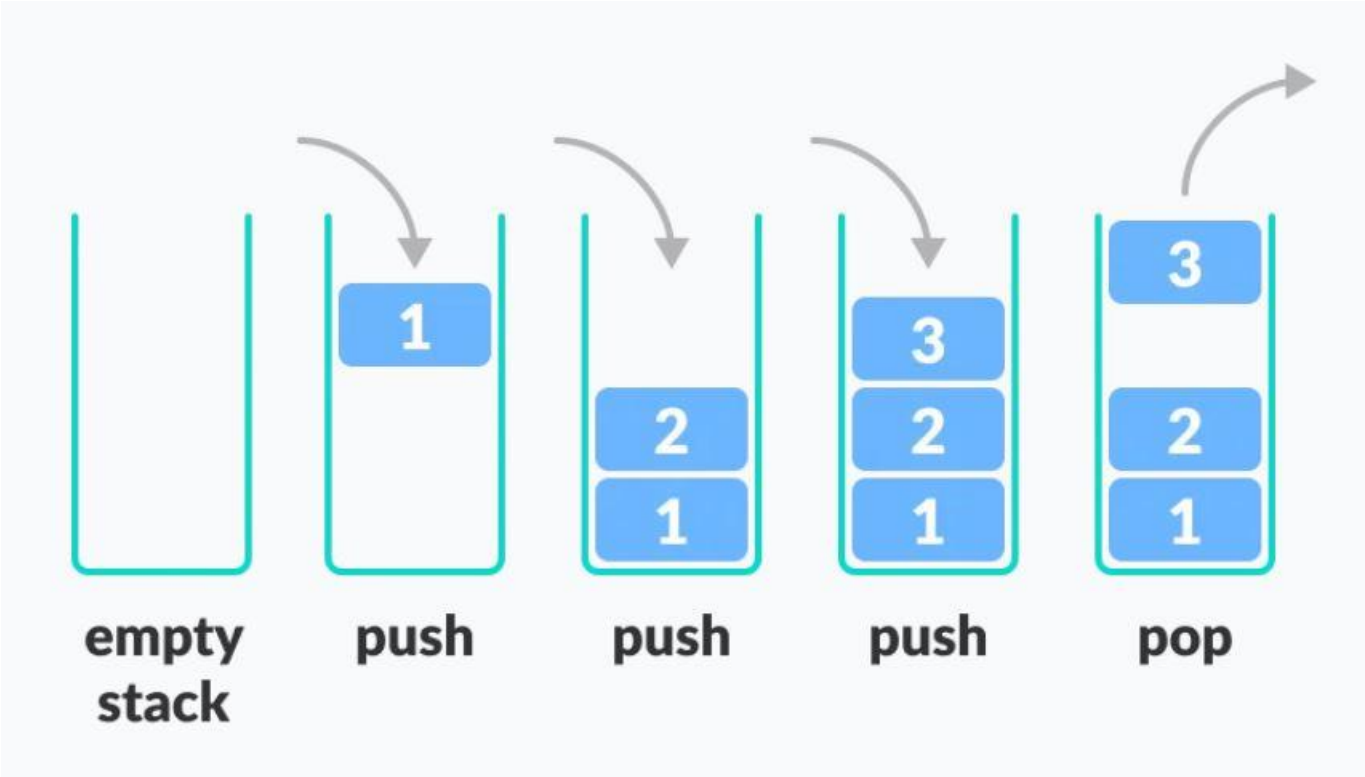
Example: f(n)=2n+3, g(n)=n

- As c1.g(n) should be less than f(n) so c1 has to be 1 whereas c2.g(n) should be greater than f(n) so c2 is equal to 5. The c1.g(n) is the lower limit of the of the f(n) while c2.g(n) is the upper limit of the f(n).
- c1.g(n)<=f(n)<=c2.g(n)
- Replace g(n) by n and f(n) by 2n+3
- c1.n <=2n+3<=c2.n
- If c1=1, c2=2, n=1 then 1*1 <=2*1+3 <=2*1
- **1** <= **5** <= **2** // for n=1, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)
- **If n=2**

- 1*2<=2*2+3<=2*2
- 2<=7<=4 // for n=2, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)
- Therefore, we can say that for any value of n, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n). Hence, it is proved that f(n) is big theta of g(n). So, this is the average-case scenario which provides the realistic time complexity.

Stack :

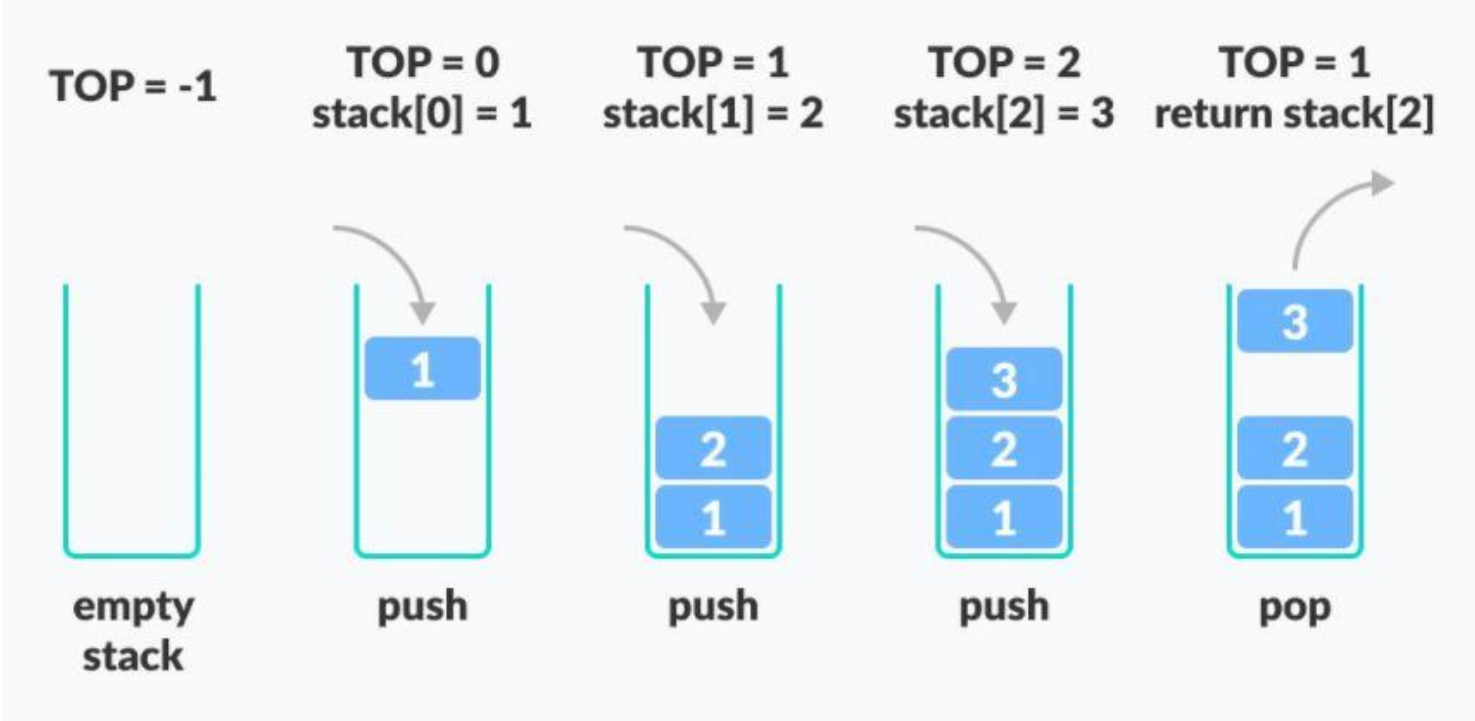
- Stack is a linear data structure in which the insertion and deletion operations are performed at only one end.
- In a stack, adding and removing of elements are performed at a single position which is known as "**top**". That means, a new element is added at top of the stack and an element is removed from the top of the stack.
- In stack, the insertion and deletion operations are performed based on **LIFO** (Last In First Out) principle.
- In a stack, the insertion operation is performed using a function called **"push"** and deletion operation is performed using a function called **"pop"**.



Working of Stack Data Structure

The operations work as follows:

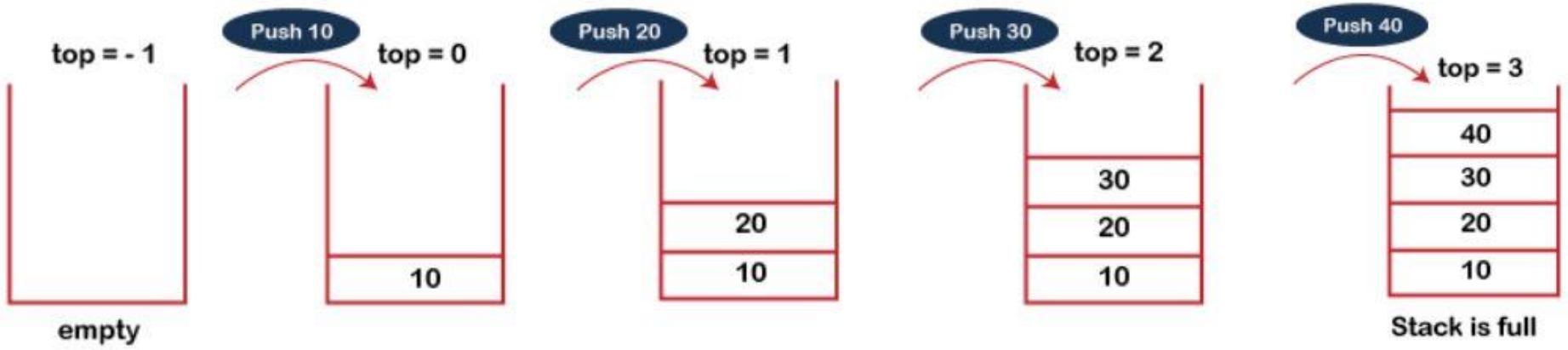
1. A pointer called TOP is used to keep track of the top element in the stack.
2. When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP==-1.
3. On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
4. On popping an element, we return the element pointed to by TOP and reduce its value.
5. Before pushing, we check if the stack is already full
6. Before popping, we check if the stack is already empty



PUSH operation

The steps involved in the PUSH operation is given below:

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**. (**stack[top] = value**).
- The elements will be inserted until we reach the ***max*** size of the stack.



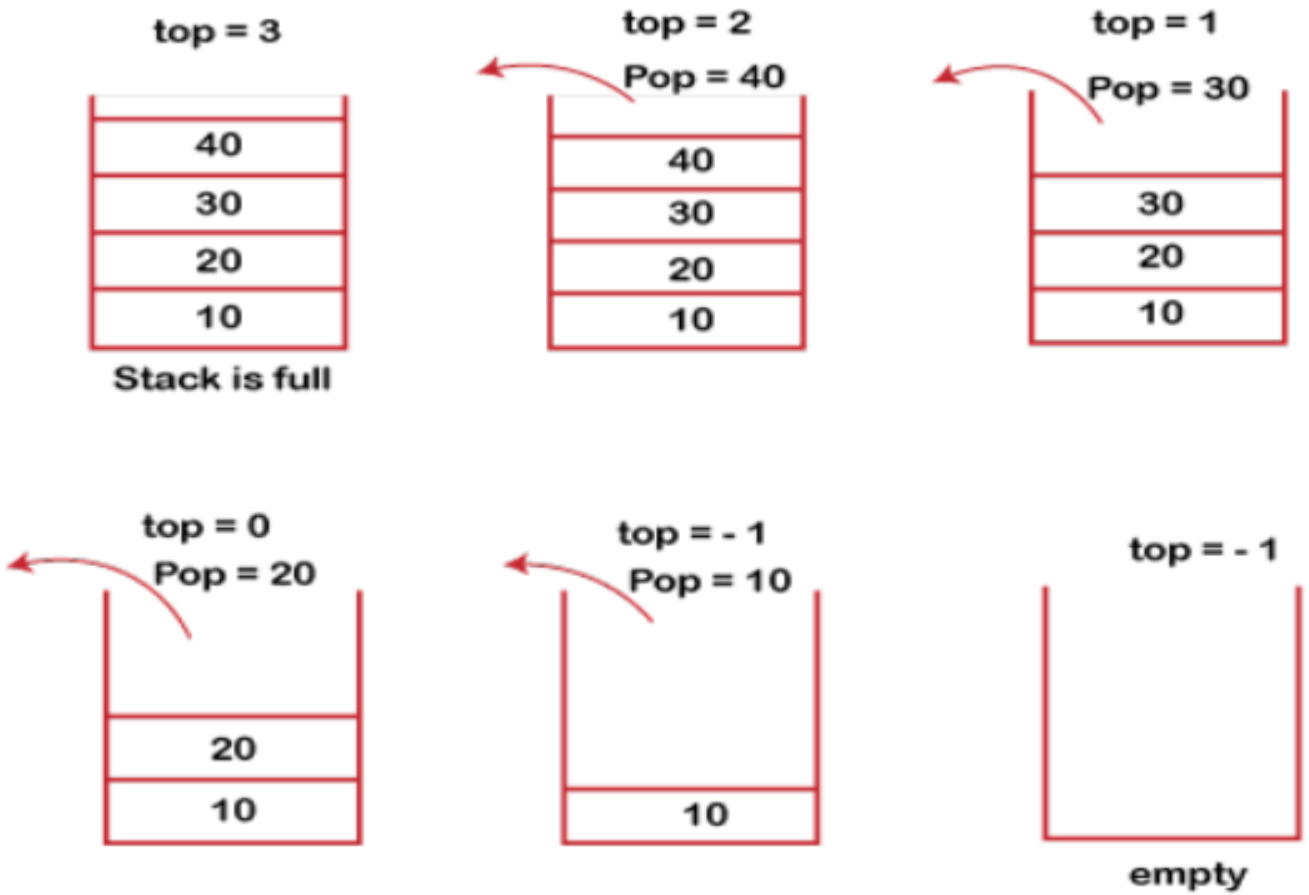
ALGORITHM FOR PUSH OPERATION

1. START
2. Check for stack overflow as if TOP==MAXSIZE-1 then
   print "Stack overflow" and exit the program
   else
   Increase the value of TOP by 1
   set TOP=TOP+1
3. Read element to be inserted.
4. set stack[TOP]=element //Insert item in new top position
5. STOP

POP operation

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the ***top***
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



ALGORITHM FOR POP OPERATION

1. START
2. Check for stack underflow as if TOP<0 then

   print "Stack underflow" and exit the program

   else

   remove the top element and set the element to the variable as

   Set Element=stack[TOP]

   Decrease TOP by 1 As

   Set TOP=TOP-1
3. Print "element" as deleted item from stack
4. STOP

Applications of Stack Data Structure

i. **In browsers** - The back button in a browser saves all the URLs you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack, and the previous URL is accessed.

ii. **In compilers** - Compilers use the stack to calculate the value of expressions like 2+4 /5 *(7-9) by converting the expression to prefix or postfix form.

iii. **To reverse a word -** Stack is also used for reversing a string/word. For example, we want to reverse "**Tesla**", so we can achieve this with the help of a stack.
First, we push all the characters of the string in a stack until we reach the null character.
After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.

iv. **DFS(Depth First Search):** This search is implemented on a Graph, and Graph uses the stack data structure.

v. **Expression conversion:** Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below: (Infix to prefix),(Infix to postfix),(prefix to postfix),(prefix to infix),(postfix to infix),(postfix to prefix).

vi. **Backtracking:** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.

Expressions:

- In any programming language, if we want to perform any calculation or to frame a condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.
- An expression is a collection of operators and operands that represents a specific value.
- **operator** is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.
- **Operands** are the values on which the operators can perform the task.

Expression Types

- Based on the operator position, expressions are divided into three types. They are as follows :

1. **Infix Expression**
2. **Postfix Expression**
3. **Prefix Expression**

-

1. Infix Expression:

- In infix expression, operator is used in between the operands.
- It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices.
- An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.
- The general structure of an Infix expression is as follows :

Operand1 Operator Operand2

- E.g : A + B, (A +B) * (C+D)

2. Postfix Expression:

- In postfix expression, operator is used after operands. We can say that "**Operator follows the Operands**".
- The general structure of Postfix expression is as follows :

Operand1 Operand2 Operator

- E.g: A B +, A B + C D + ∗

3. Prefix Expression :

- In prefix expression, operator is used before operands. We can say that "**Operands follows the Operator**".
- The general structure of Prefix expression is as follows:

Operator Operand1 Operand2

- E.g: + A B, ∗ + A B + C D

Algorithm for Infix to Postfix

1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then directly print it to the result (Output).
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
4. If the reading symbol is right parenthesis ')', pop it from the stack and print the operator until left parenthesis is found.
5. If incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If incoming symbol has lower precedence than the top of the stack, pop and print the top. Then test the incoming operator against the new top of the stack.
7. If incoming symbol has equal precedence than the top of the stack, use associativity rule.

⇒Associativity Left to Right, then pop and print the top of the stack and then push the incoming operator.

⇒Associativity Right to Left, then push the incoming operator.

8. At the end of the expression, pop and print all operators of stack.

Example : (A + B) * (C - D)

A + B ➔ Postfix : AB+

Rules

1. Priorities of operators:
- ^ = Highest priority
- *,/ = Next priority
- +,- = Lowest priority

2. No two operators of same priority can stay together in stack column.

3. Lowest priority can not be placed before highest priority.

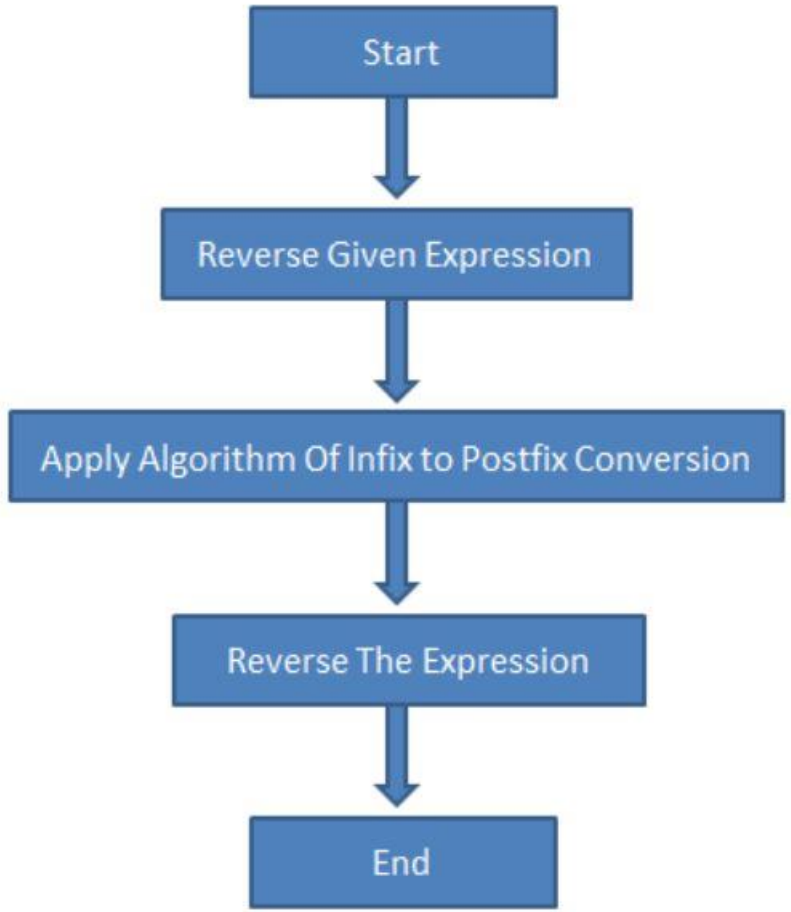
Q=((A-(B+C))*D)^(E+F)			
S.N	Symbol	Stack	Postfix
1	(	(	
2	(	((	
3	A	((	A
4	-	((-	A
5	(	((-(	A
6	B	((-(	AB
7	+	((-(+	AB
8	C	((-(+	ABC
9	)	((-	ABC+
10	)	(	ABC+-
11	*	(*	ABC+-
12	D	(*	ABC+-D
13	)		ABC+-D*
14	^	^	ABC+-D*
15	(	^(	ABC+-D*
16	E	^(	ABC+-D*E
17	+	^(+	ABC+-D*E
18	F	^(+	ABC+-D*EF
19	)	^	ABC+-D*EF+
			ABC+-D*EF+^

Q=A-B/C*D+E			
S.N	Symbol	Stack	Postfix
1	A		A
2	-	-	A
3	B	-	AB
4	/	-/	AB
5	C	-/	ABC
6	*	-*	ABC/
7	D	-*	ABC/D
8	+	+	ABC/D*-
9	E	+	ABC/D*-E
			ABC/D*-E+

Q=((A+B)*(C-D))			
S.N	Symbol	Stack	Postfix
1	(	(	
2	(	((	
3	A	((	A
4	+	((+	A
5	B	((+	AB
6	)	(	AB+
7	*	(*	AB+
8	(	(*(	AB+
9	C	(*(	AB+C
10	-	(*(-	AB+C
11	D	(*(-	AB+CD
12	)	(*	AB+CD-
13	)		AB+CD-*

Rules for the conversion of infix to prefix expression:

- First, reverse the infix expression given in the problem.
- Scan the expression from left to right.
- Whenever the operands arrive, print them.
- If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
- If the incoming operator has higher precedence than the TOP of the stack, push the incoming operator into the stack.
- If the incoming operator has the same precedence with a TOP of the stack, push the incoming operator into the stack.
- If the incoming operator has lower precedence than the TOP of the stack, pop, and print the top of the stack. Test the incoming operator against the top of the stack again and pop the operator from the stack till it finds the operator of a lower precedence or same precedence.
- If the incoming operator has the same precedence with the top of the stack and the incoming operator is ^, then pop the top of the stack till the condition is true. If the condition is not true, push the ^ operator.
- When we reach the end of the expression, pop, and print all the operators from the top of the stack.
- If the operator is ')', then push it into the stack.
- If the operator is '(', then pop all the operators from the stack till it finds) opening bracket in the stack.
- If the top of the stack is ')', push the operator on the stack.
- At the end, reverse the output.


Start
Reverse Given Expression
Apply Algorithm Of Infix to Postfix Conversion
Reverse The Expression
End

Expression : (A+ B^C)*D + E^5

Step 1 : Reverse the given expression : 5^E+D*)C^B+A(

Step 2 : Make every ‘(‘ as ‘)’ and every ‘)’ as ‘(‘.

5^E+D*(C^B+A)

Step 3 : Convert expression to postfix Expression

Step 4 : Reverse the postfix expression as :

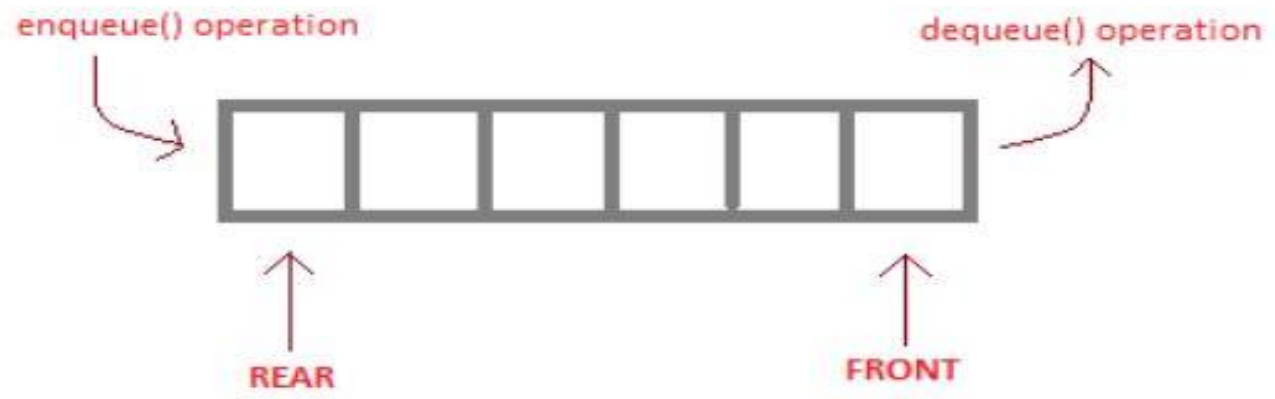
5E^DCB^A+*+

Result: Prefix Expression : **+*+A^BCD^E5**

S.N	Symbol	Stack	Postfix
1	5		5
2	^	^	5
3	E	^	5E
4	+	+	5E^
5	D	+	5E^D
6	*	+ *	5E^D
7	(	+ * (	5E^D
8	C	+ * (	5E^DC
9	^	+ * (^	5E^DC
10	B	+ * (^	5E^DCB
11	+	+ * (+	5E^DCB^
12	A	+ * (+	5E^DCB^A
13	)	+ *	5E^DCB^A+
			5E^DCB^A+*+

Queue :

- A queue is a data structure used for storing data (similar to Linked Lists and Stacks).
- In queue, the order in which data arrives is important.
- In general, a queue is a line of people or things waiting to be served in sequential order starting at the beginning of the line or sequence.
- Queue is an ordered list in which insertions are done at one end (rear/tail) and deletions are done at other end (front/head).
- The first element to be inserted is the first one to be deleted. Hence, it is called First in First out (FIFO) or Last in Last out (LILO) list.



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

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

QUEUE DATA STRUCTURE

Queue

The queue has the following conditions:

- If FRONT = 0, then the queue is empty.
- If REAR = size of the queue, then the queue is full.
- If FRONT = REAR, then there is at least one element in the queue.
- If you want to know the total number of elements in the queue, then you use this formula (REAR – FRONT) +1.

What are the steps to Enqueue Operation into a Data Structure Queue?

- Two pointers – **front** and **rear** are maintained by Queues and hence the operations are different from that of Stacks.
- The steps to enqueue (insert) data into a queue are -

1. Check if the queue is full.
2. If the queue is full, produce overflow error and exit.
3. If the queue is not full, increment rear pointer to point the next empty space.
4. Add data element to the queue location, where the rear is pointing.
5. Return success.

Algorithm:

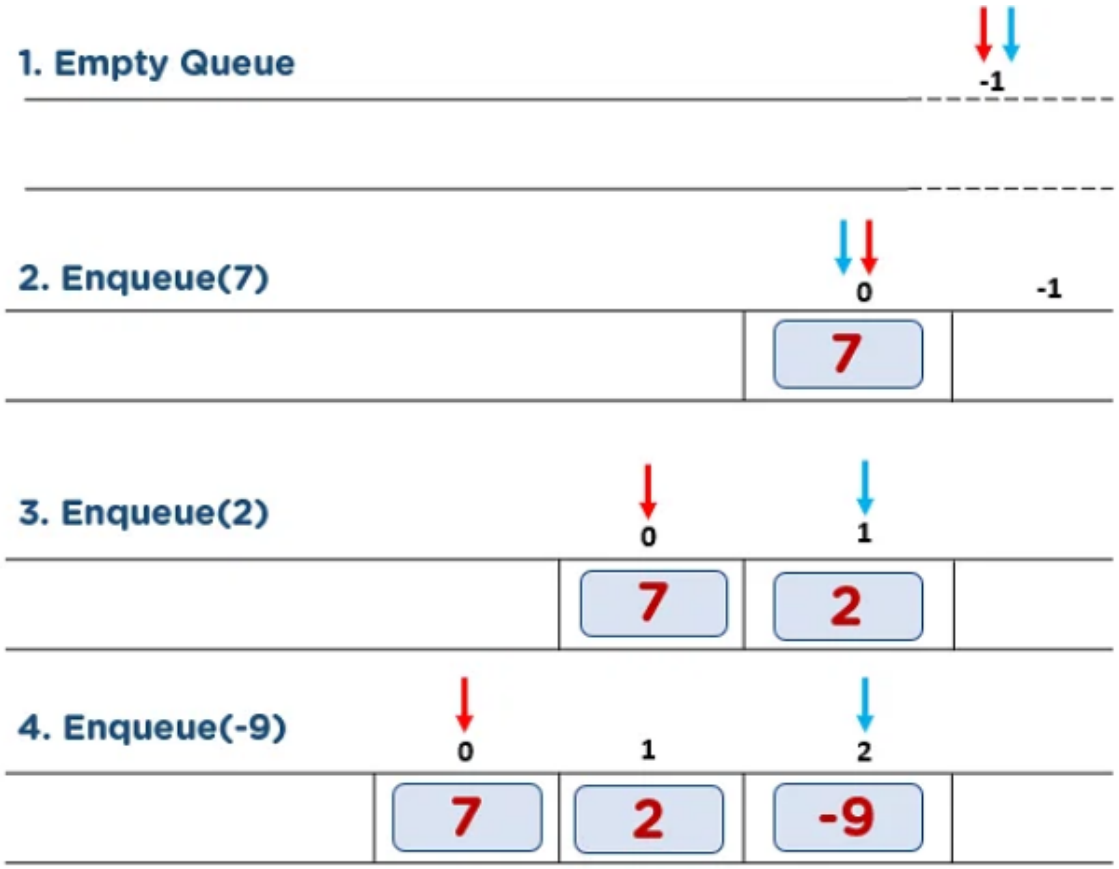
1. Initialize front=-1 ,rear=-1 [Create an empty queue]
2. Input the value to be inserted and assign to variable “data”.
3. for the first element, set the value of FRONT to 0
4. if(rear>=MAXSIZE-1) then display “Queue overflow” and exit.

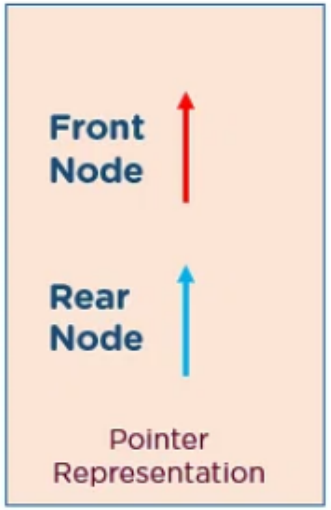
   else

   rear=rear+1
5. Queue[rear]=data
6. Exit

Enqueue Operation

Step 1: Check if the queue is full.
Step 2: If the queue is full, Overflow error.
Step 3: If the queue is not full, increment the rear pointer to point to the next available empty space.
Step 4: Add the data element to the queue location where the rear is pointing.
Step 5: Here, we have successfully added 7, 2, and -9.





What are the steps to Dequeue Operation into a Data Structure Queue?

1. Check if the queue is empty.
2. If the queue is empty, produce underflow error and exit.
3. If the queue is not empty, access the data where **front** is pointing.
4. Increment **front** pointer to point to the next available data element.
5. Return success.

Algorithm:

1. If (rear<front) or if (front==-1 and rear==-1) [Checking for empty queue]

   Display queue is empty and exit.

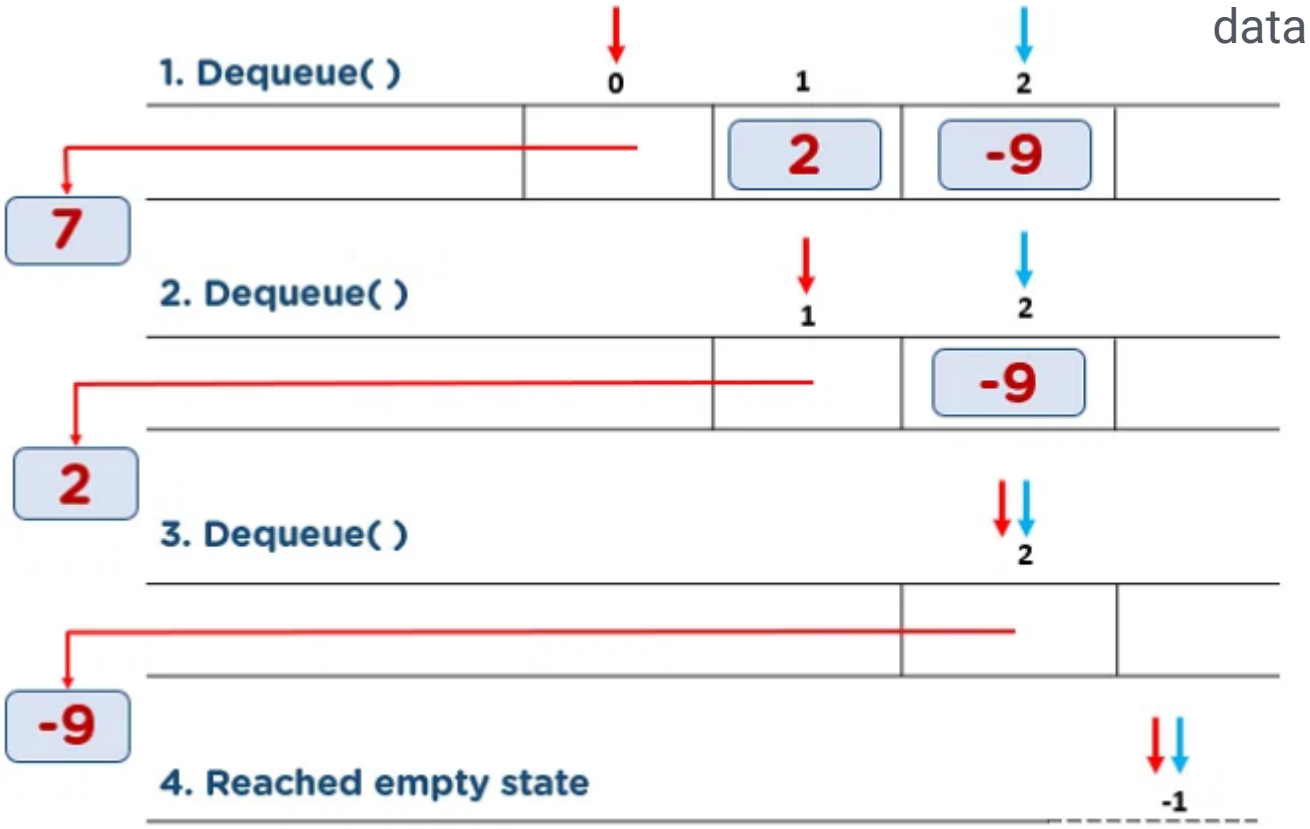
2. Else data=queue[front]

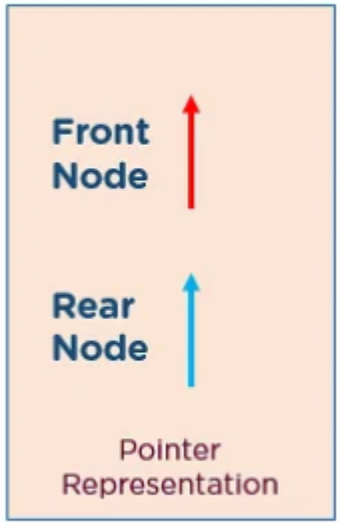
3. front=front+1

4. Exit

Dequeue Operation

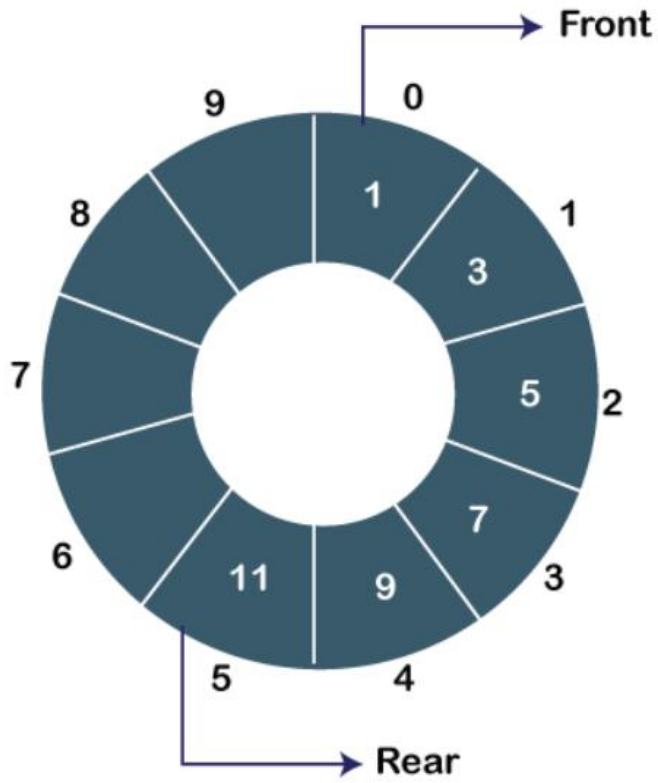
Obtaining data from the queue comprises two subtasks: access the data where the front is pointing and remove the data after access.
Step 1: Check if the queue is empty.
Step 2: If the queue is empty, Underflow error.
Step 3: If the queue is not empty, access the data where the front pointer is pointing.
Step 4: Increment front pointer to point to the next available data element.
Step 5: Here, we have removed 7, 2, and -9 from the queue data structure



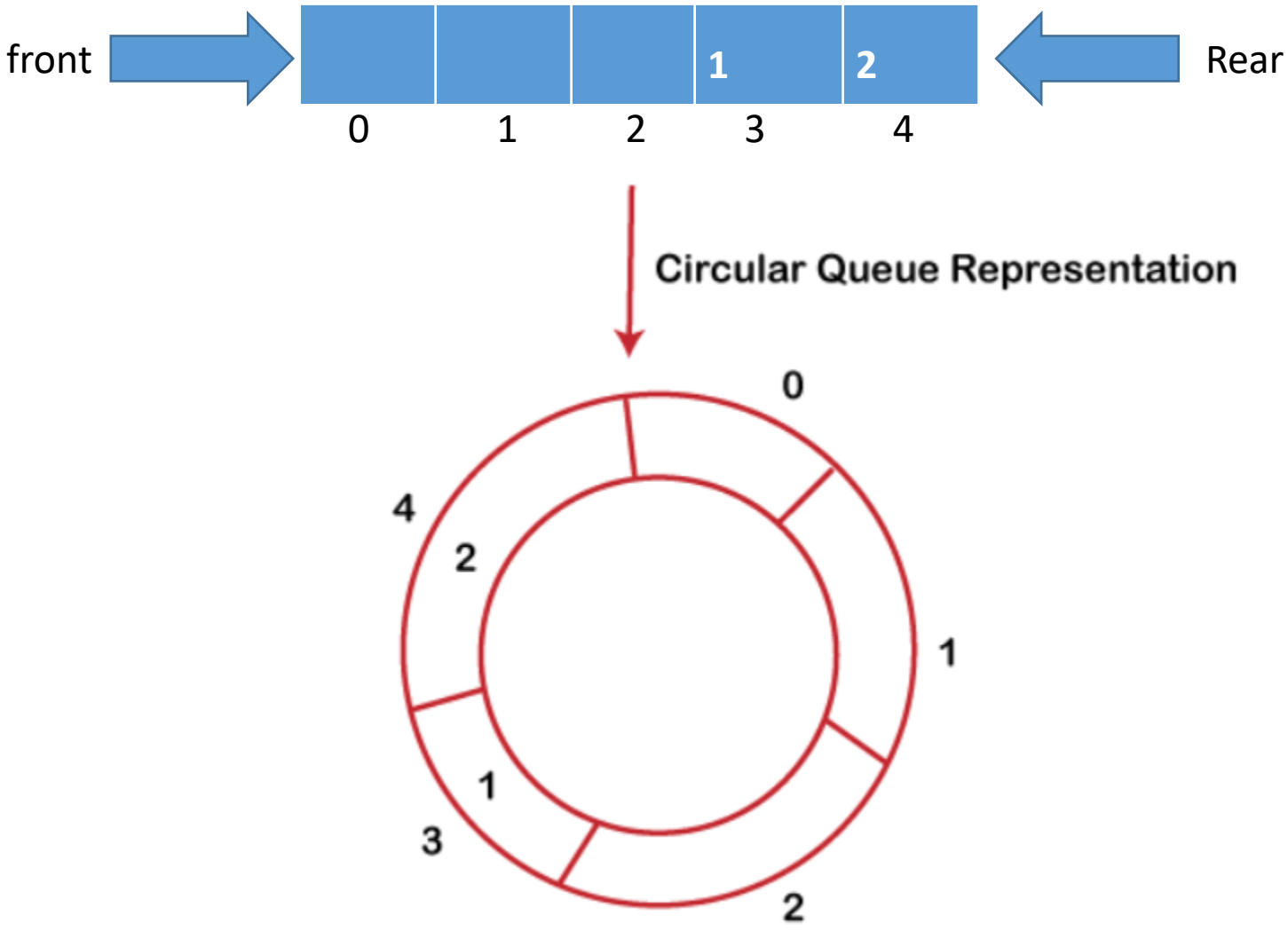


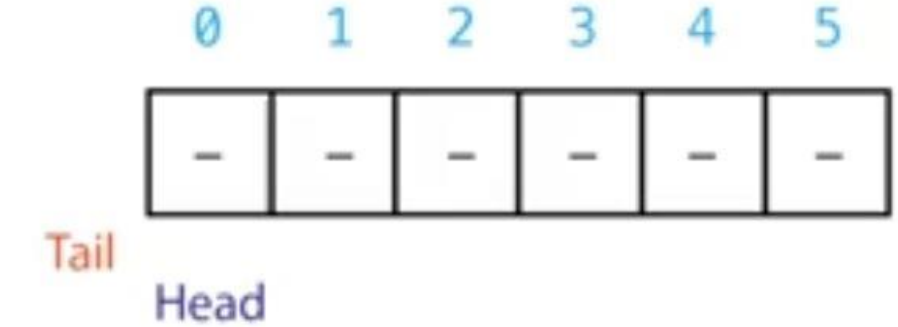
Circular Queue:

- A circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.
- In Circular Queue, all the nodes are represented as circular.
- It is similar to the linear Queue except that the last element of the queue is connected to the first element.
- It is also known as **Ring Buffer** as all the ends are connected to another end.

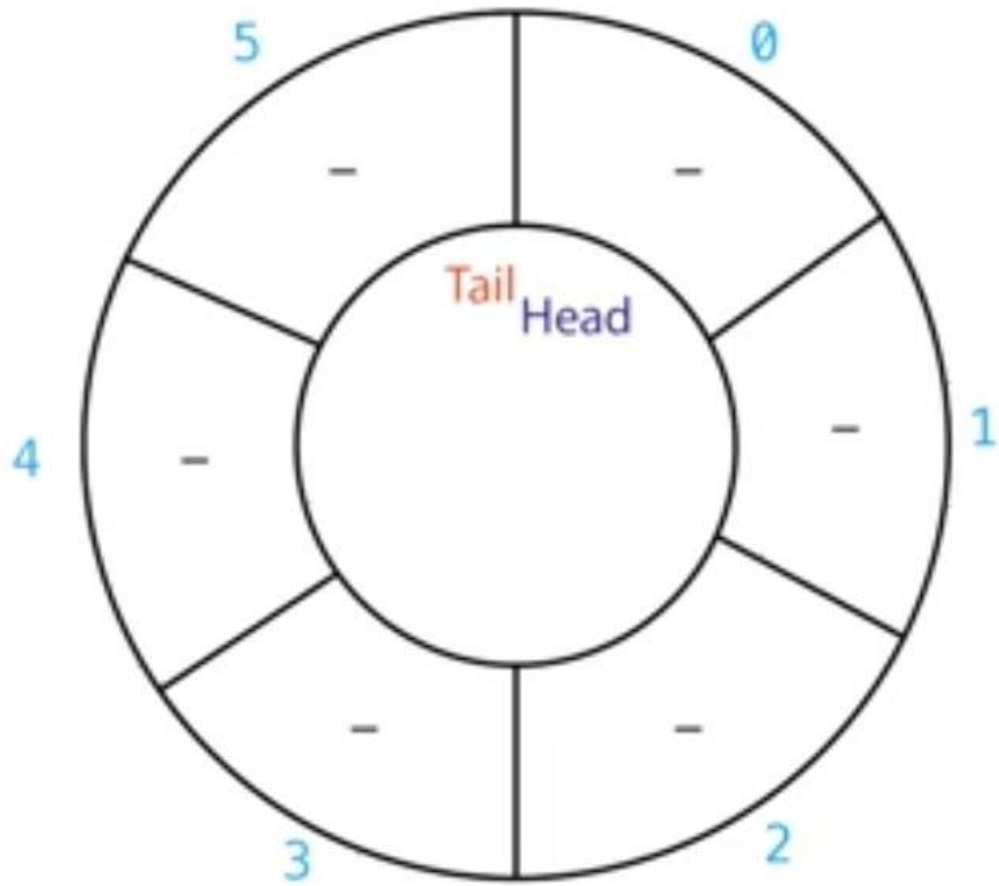


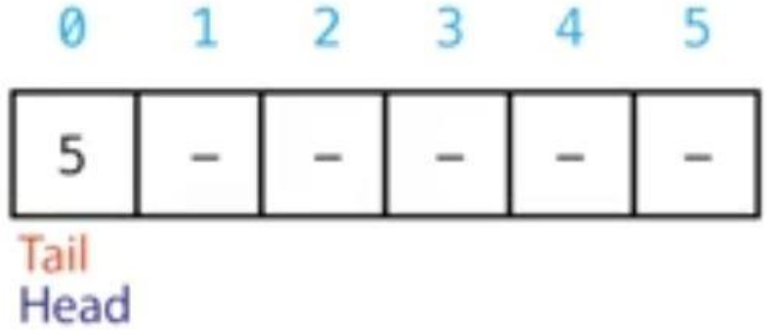
- There was one limitation in the array implementation of queue. If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.

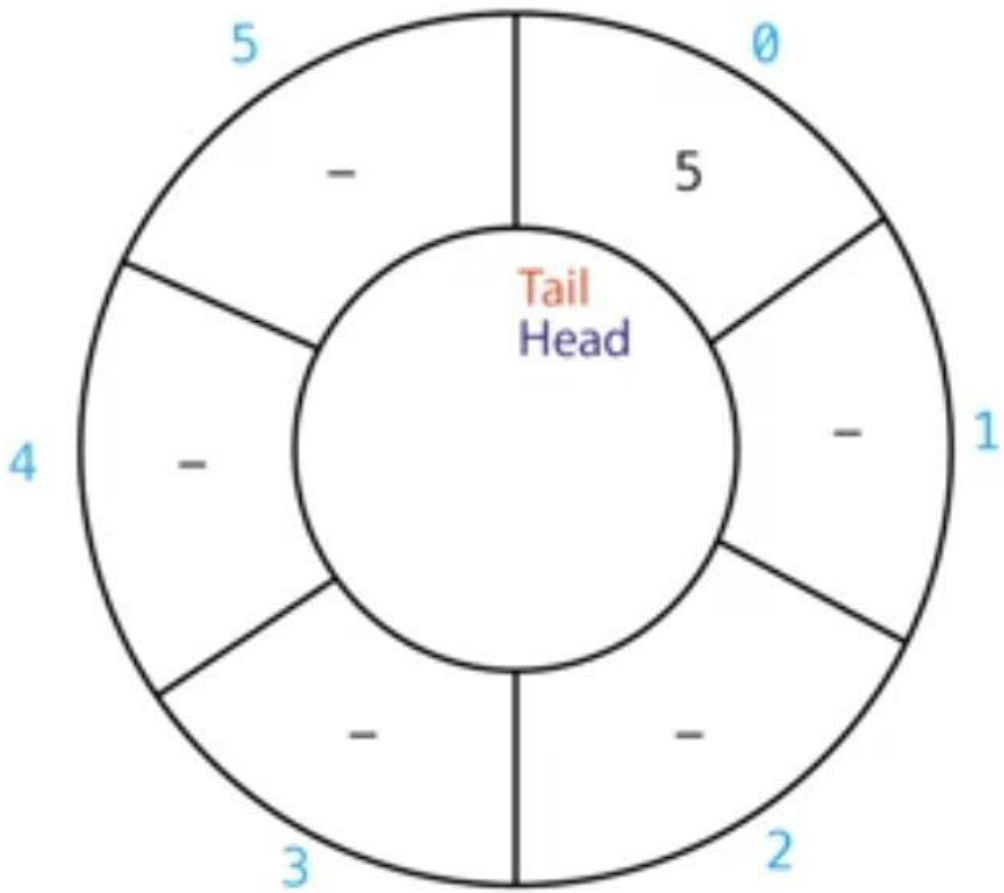




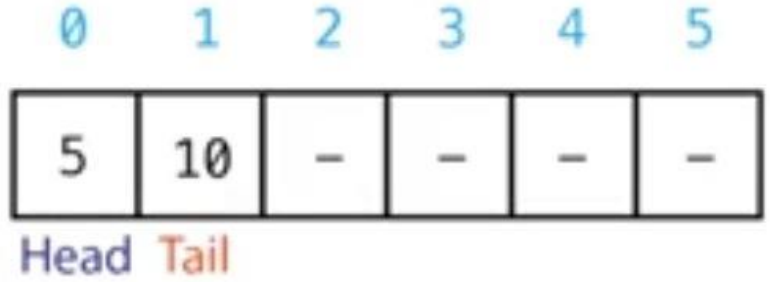

```
Q = circularQueue(6)
```

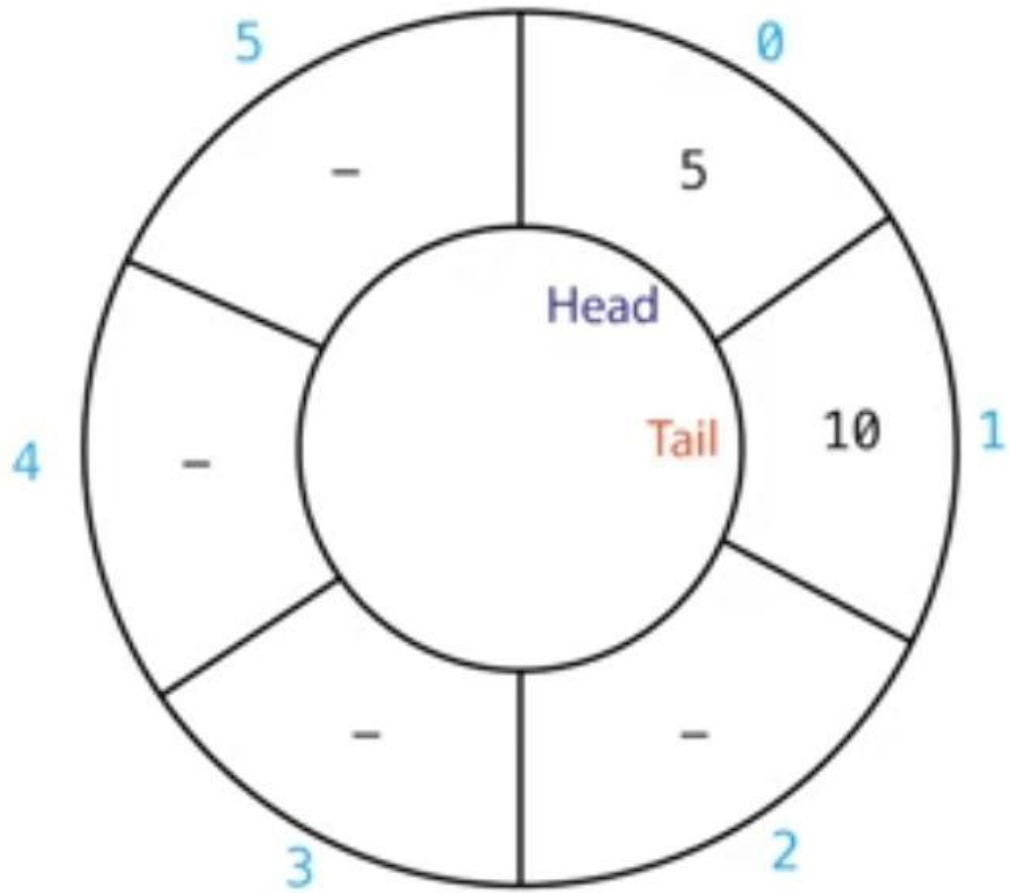




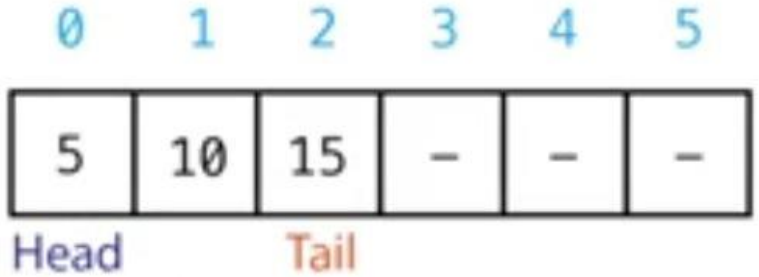


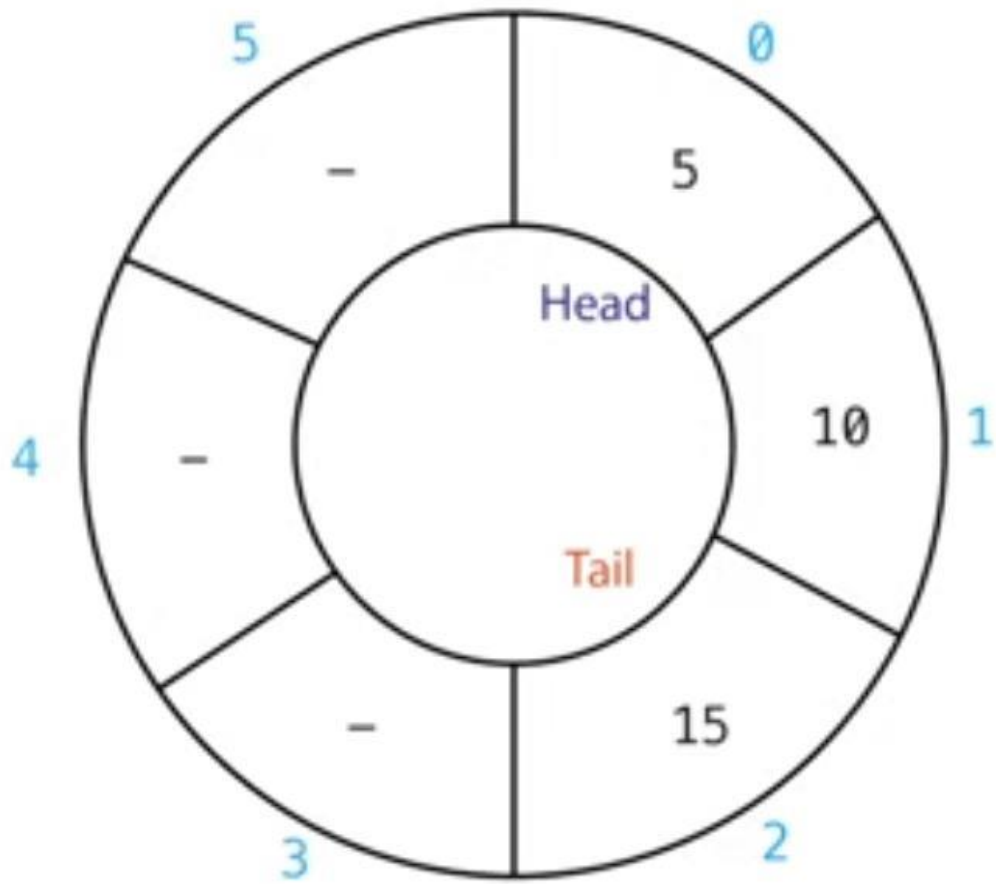

```
Q = circularQueue(6)
Q.Enqueue(5)
```



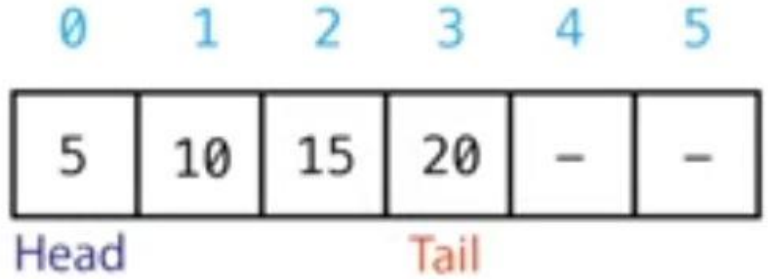


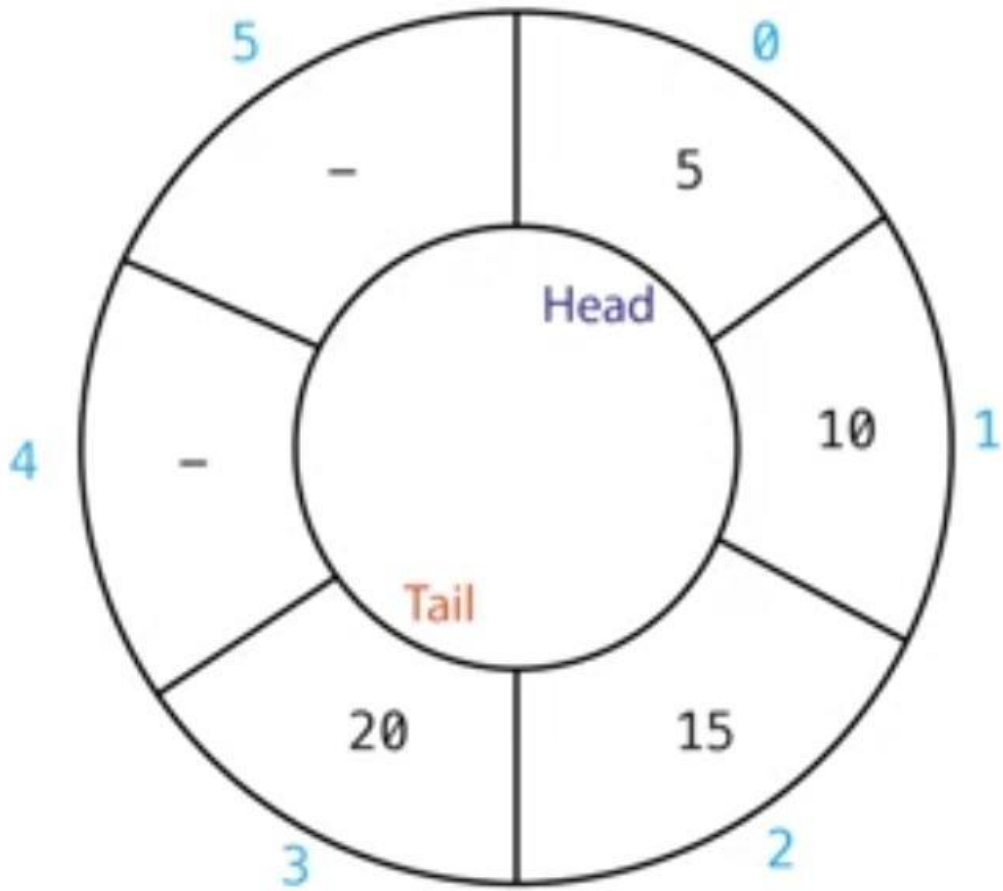

```
Q = circularQueue(6)
Q.Enqueue(5)
Q.Enqueue(10)
```



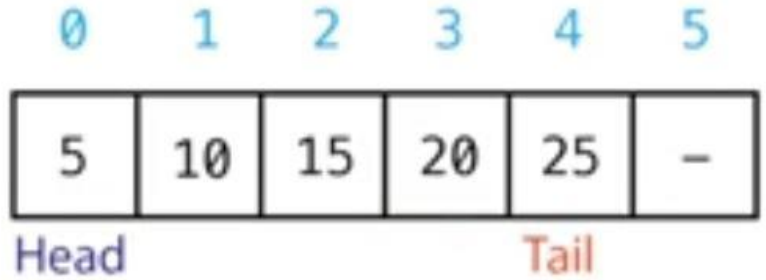


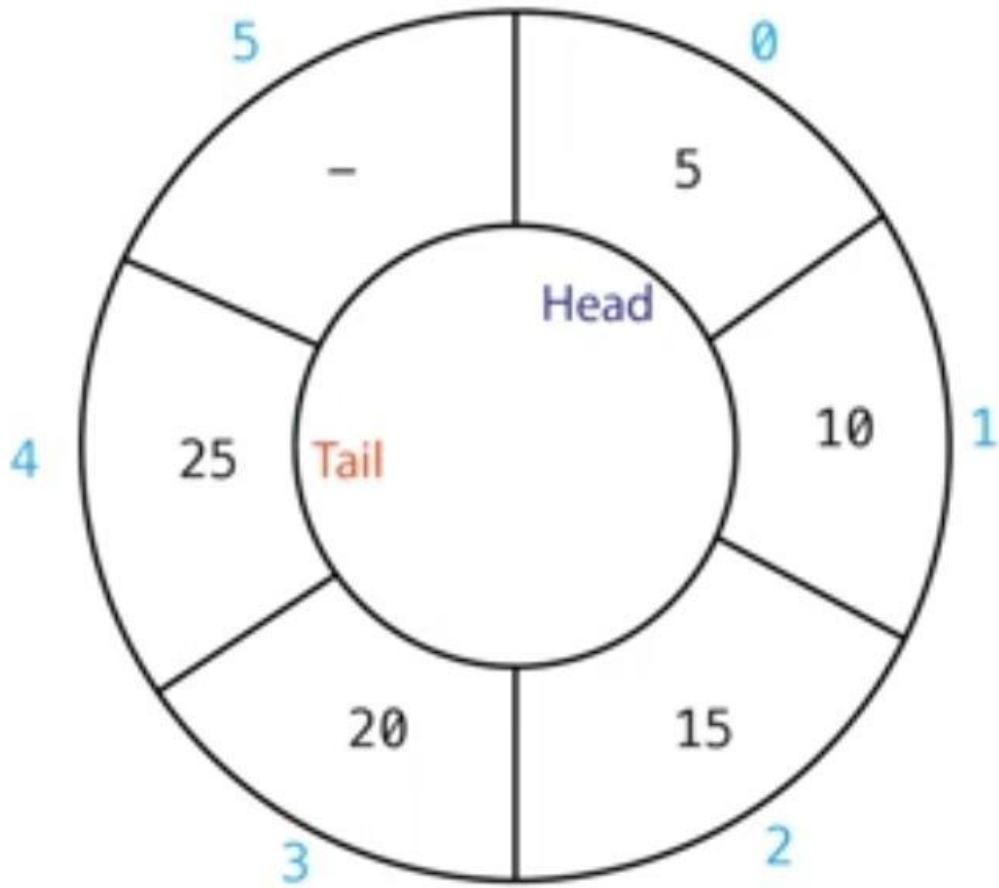

```
Q = circularQueue(6)
Q.Enqueue(5)
Q.Enqueue(10)
Q.Enqueue(15)
```



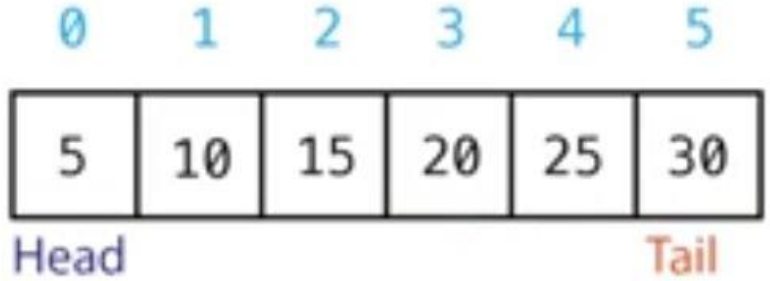


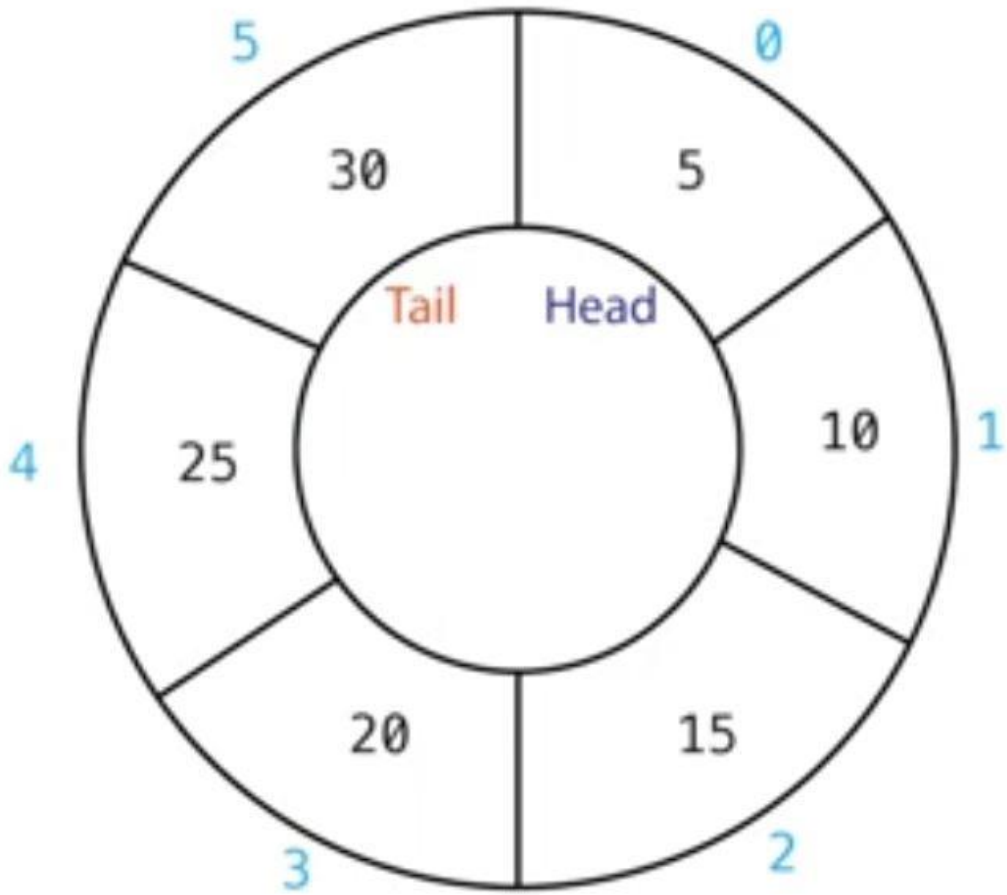

```
Q = circularQueue(6)
Q.Enqueue(5)
Q.Enqueue(10)
Q.Enqueue(15)
Q.Enqueue(20)
```



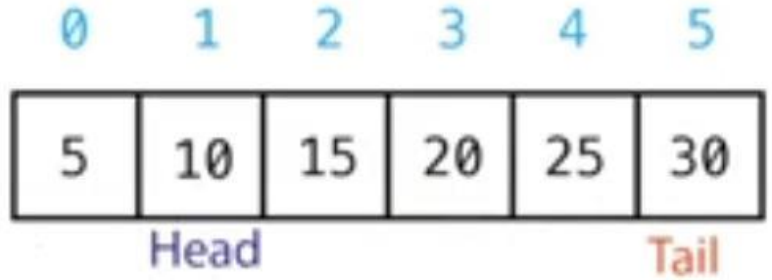


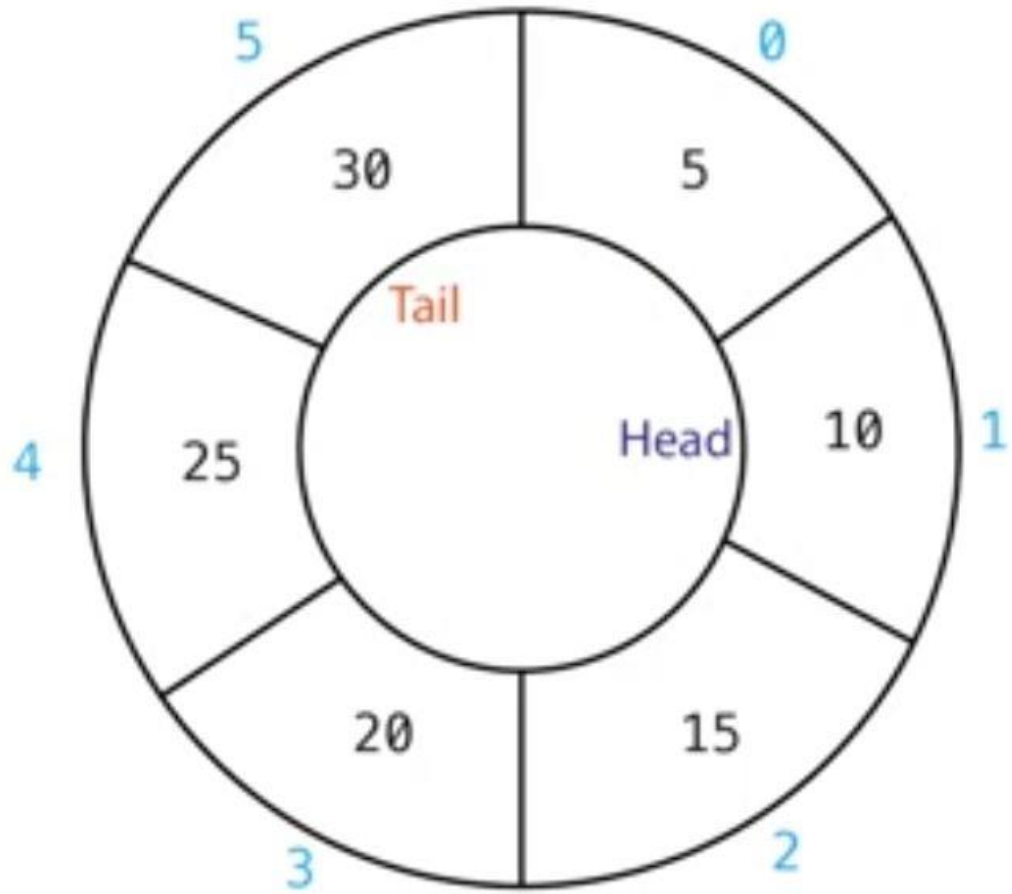

```
Q = circularQueue(6)
Q.Enqueue(5)
Q.Enqueue(10)
Q.Enqueue(15)
Q.Enqueue(20)
Q.Enqueue(25)
```





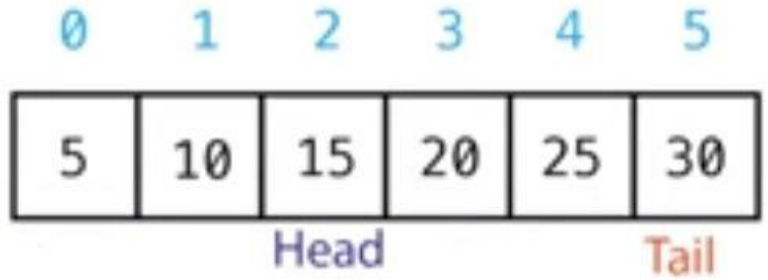

```
Q = circularQueue(6)
Q.Enqueue(5)
Q.Enqueue(10)
Q.Enqueue(15)
Q.Enqueue(20)
Q.Enqueue(25)
Q.Enqueue(30)
```

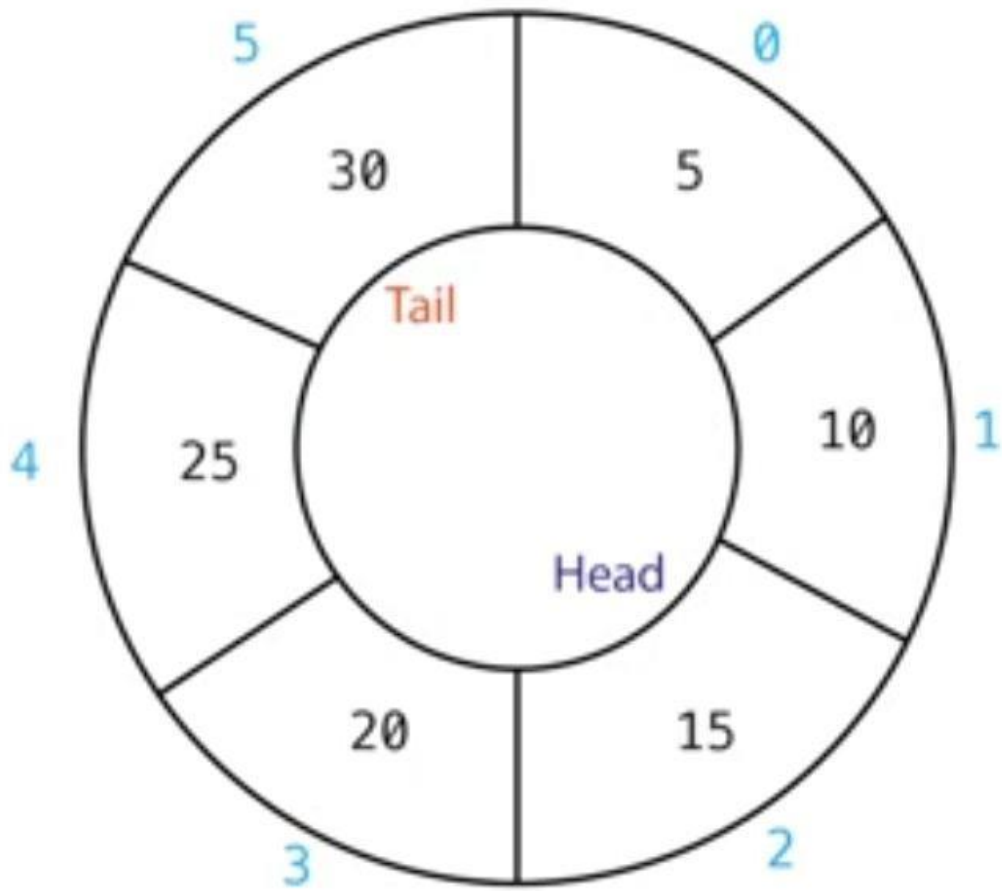





```
Q = circularQueue(6)
Q.Enqueue(5)
Q.Enqueue(10)
Q.Enqueue(15)
Q.Enqueue(20)
Q.Enqueue(25)
Q.Enqueue(30)
Q.Dequeue()
```

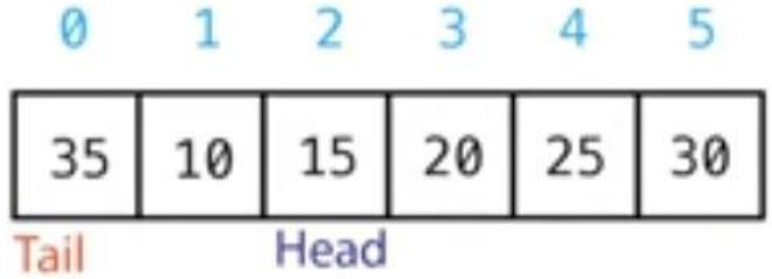
returns 5

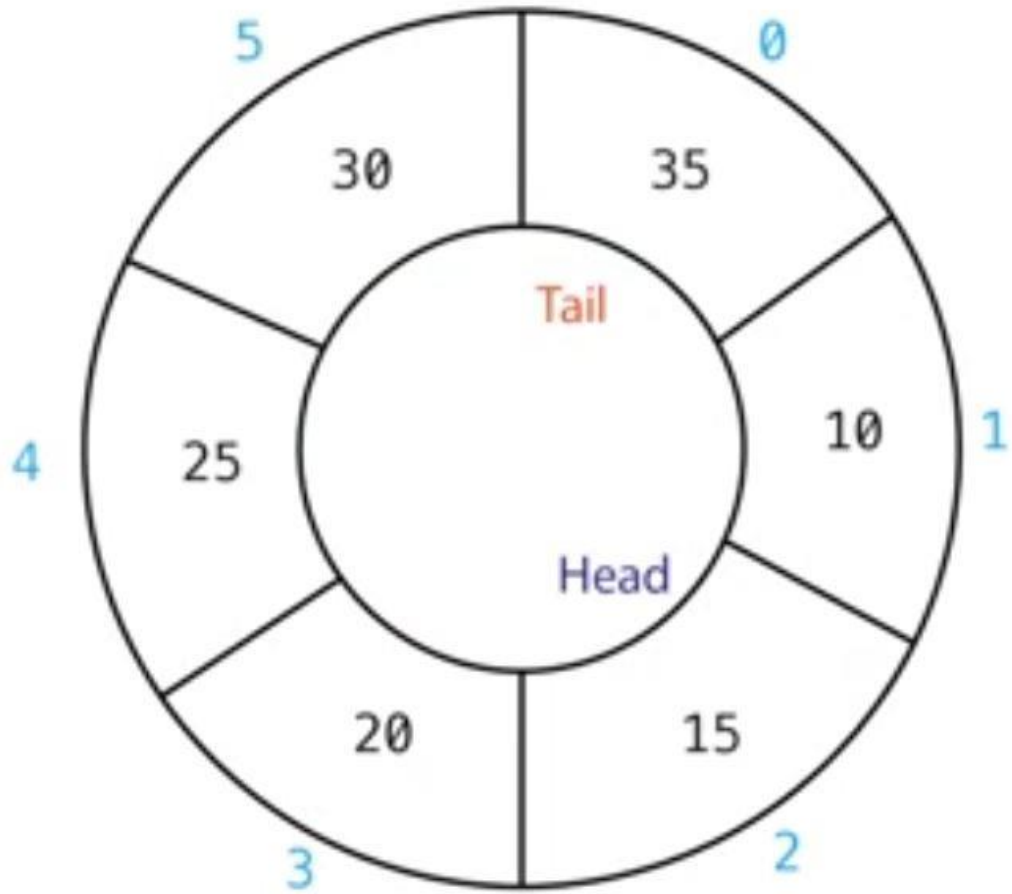




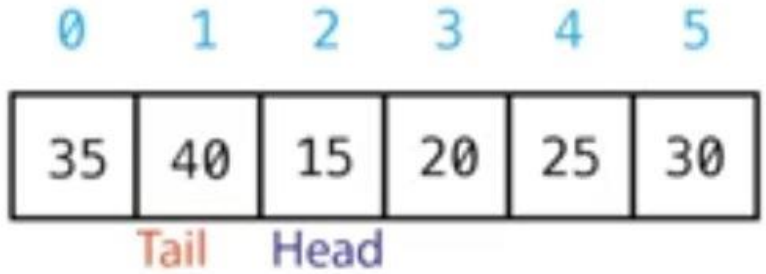

```
Q = circularQueue(6)
Q.Enqueue(5)
Q.Enqueue(10)
Q.Enqueue(15)
Q.Enqueue(20)
Q.Enqueue(25)
Q.Enqueue(30)
Q.Dequeue()
Q.Dequeue()
```

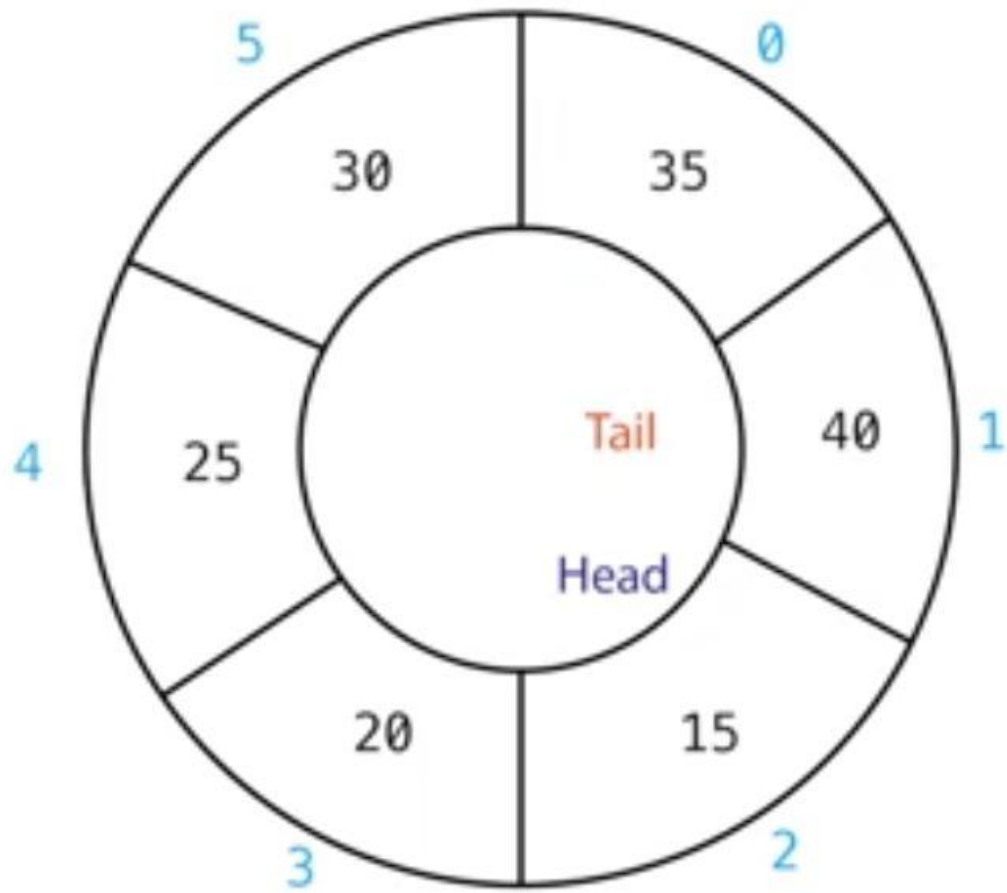
returns 10






```
Q = circularQueue(6)
Q.Enqueue(5)
Q.Enqueue(10)
Q.Enqueue(15)
Q.Enqueue(20)
Q.Enqueue(25)
Q.Enqueue(30)
Q.Dequeue()
Q.Dequeue()
Q.Enqueue(35)
```






```
Q = circularQueue(6)
Q.Enqueue(5)
Q.Enqueue(10)
Q.Enqueue(15)
Q.Enqueue(20)
Q.Enqueue(25)
Q.Enqueue(30)
Q.Dequeue()
Q.Dequeue()
Q.Enqueue(35)
Q.Enqueue(40)
```

3. Priority Queue :

- priority queue is another special type of Queue data structure in which each element has some priority associated with it.
- Based on the priority of the element, the elements are arranged in a priority queue.
- It behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue.
- If two elements has same priority then they are processed according to the order in which they were added to the queue.
- The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.
- The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

Implementation of the Priority Queue in Data Structure

We can implement the priority queues in one of the following ways:

- Linked list
- Binary heap
- Arrays
- Binary search tree

Note : The binary heap is the most efficient method for implementing the priority queue in the data structure.

Types of Priority Queue:

1. **Ascending Priority Queue:** In this type of priority queue, elements can be inserted into any order but only the smallest element can be removed.
2. **Descending Priority Queue:** In this type of priority queue, elements can be inserted into any order but only the largest element can be removed.

Representation of priority queue

We will create the priority queue by using the list given below in which **INFO** list contains the data elements, **PRIORITY** list contains the priority numbers of each data element available in the **INFO** list, and POINTER basically contains the address of the next node.

address	INFO	PRIORITY	POINTER
0	200	2	4
1	400	4	2
2	500	4	6
3	300	1	0
4	100	2	5
5	600	3	1
6	700	4	

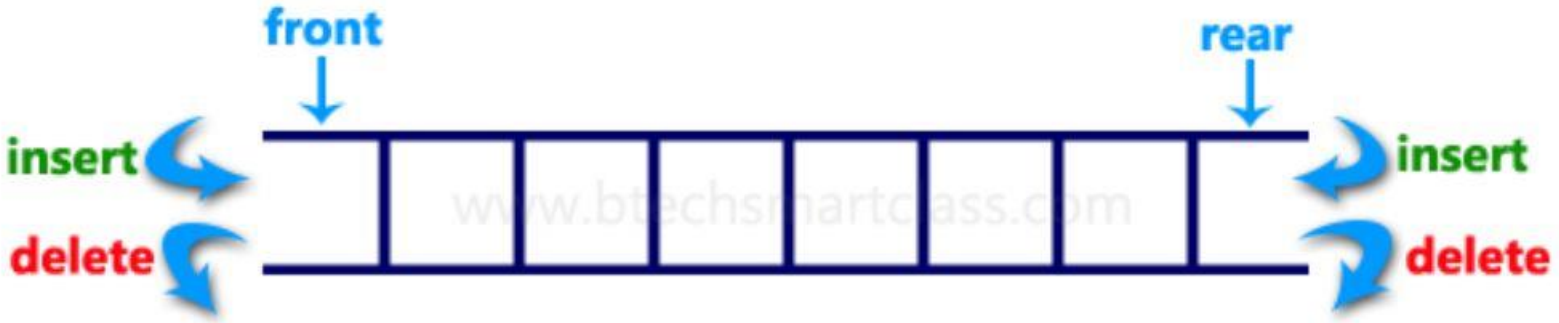
Lowest value = Highest priority

Applications of Priority Queue:

- used in operating system like priority scheduling, load balancing and interrupt handling.
- used in Dijkstra's shortest path algorithm.
- used in data compression techniques like Huffman code.
- Used in heap sort.

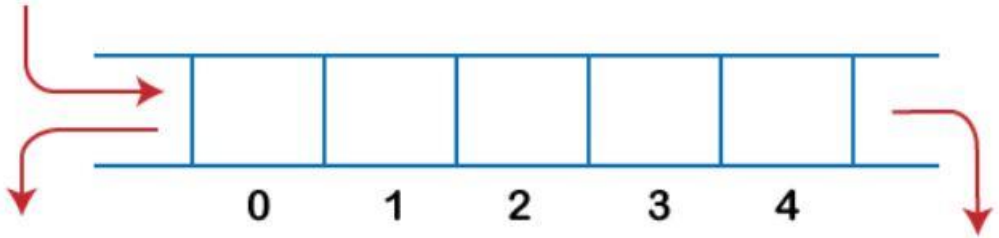
4. Deque (Double Ended Queue) :

- **Deque** is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.
- Deque can be used both as **stack** and **queue** as it allows the insertion and deletion operations on both ends.

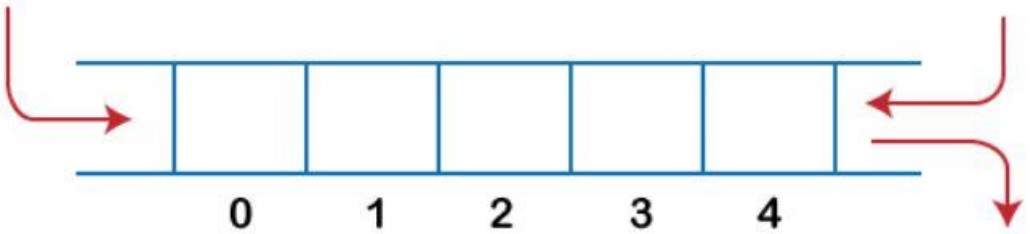


Types of Deque

1. **Input-restricted queue:** The input-restricted queue means that some restrictions are applied to the insertion. In input-restricted queue, the insertion is applied to one end while the deletion is applied from both the ends.



2. **Output-restricted queue:** The output-restricted queue means that some restrictions are applied to the deletion operation. In an output-restricted queue, the deletion can be applied only from one end, whereas the insertion is possible from both ends.



Applications of Deque

- can be used as a **stack** and **queue**; therefore, it can perform both redo and undo operations.
- Undo-redo operations in software applications.
- Implementing task scheduling for multiple processors (multiprocessor scheduling) [A-steal job scheduling algorithm].
- used as a palindrome checker means that if we read the string from both ends, then the string would be the same.

Operations on Deque

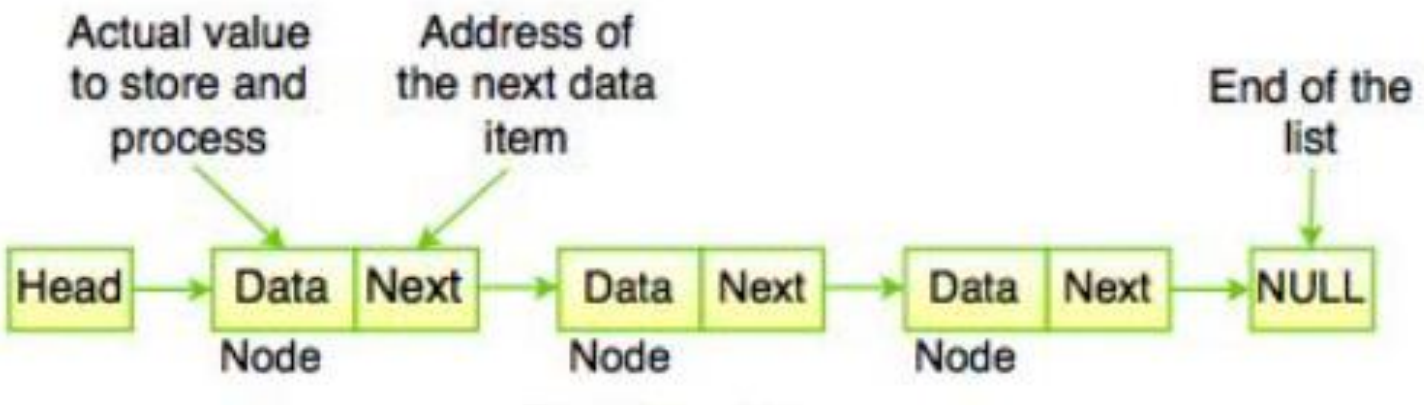
- Insert at front
- Delete from end
- insert at rear
- delete from rear

Linked List

- When we want to work with an unknown number of data values, we use a linked list data structure to organize that data.
- It is a collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.

A linked list has the following properties:

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



Limitations of Array

- 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. Using linked list is useful because:

- It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
- Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Advantages of using Linked List

- The chances of memory wastage are minimum because memories are allocated dynamically as the requirement of the user.
- Insertions and deletions can be done easily. It does not need movement of elements for insertion and deletion.
- It can be extended or reduced according to requirements.
- Elements may or may not be stored in consecutive memory locations so if we do not have consecutive memory available, even then we can store the data in computer.
- It is less expensive.
- Data structures such as stack and queues can be easily implemented using linked list.

Linked Lists ADT

The following operations make linked lists an ADT:

Main Linked Lists Operations

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

Auxiliary Linked Lists Operations

- Delete List: removes all elements of the list (disposes the list)
- Count: returns the number of elements in the list.
- Find n^{th} node from the end of the list.

Operations of Linked List

1. Traversing a linked list.
2. Append a new node (to the end) of a list
3. Prepend a new node (to the beginning) of the list
4. Inserting a new node to a specific position on the list
5. Deleting a node from the list
6. Updating a node in the list

Types of Linked List

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List
4. Doubly Circular Linked List

1. **Singly Linked List:**

- Each node has a single link to another node.
- Each node stores the contents of the node and a reference to the next node in the list.
- It does not store any pointer any reference to the previous node.
- It has successor and predecessor. First node does not have predecessor while last node does not have successor. Last node have successor reference as NULL.
- It has two successive nodes linked together in linear way and contains address of the next node to be followed.
- It has only single link for the next node.
- In this type of linked list, only forward sequential movement is possible, no direct access is allowed.

Singly Linked List Insertion

Insertion into a singly-linked list has three cases:

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

Insert a Node at the beginning of Linked List

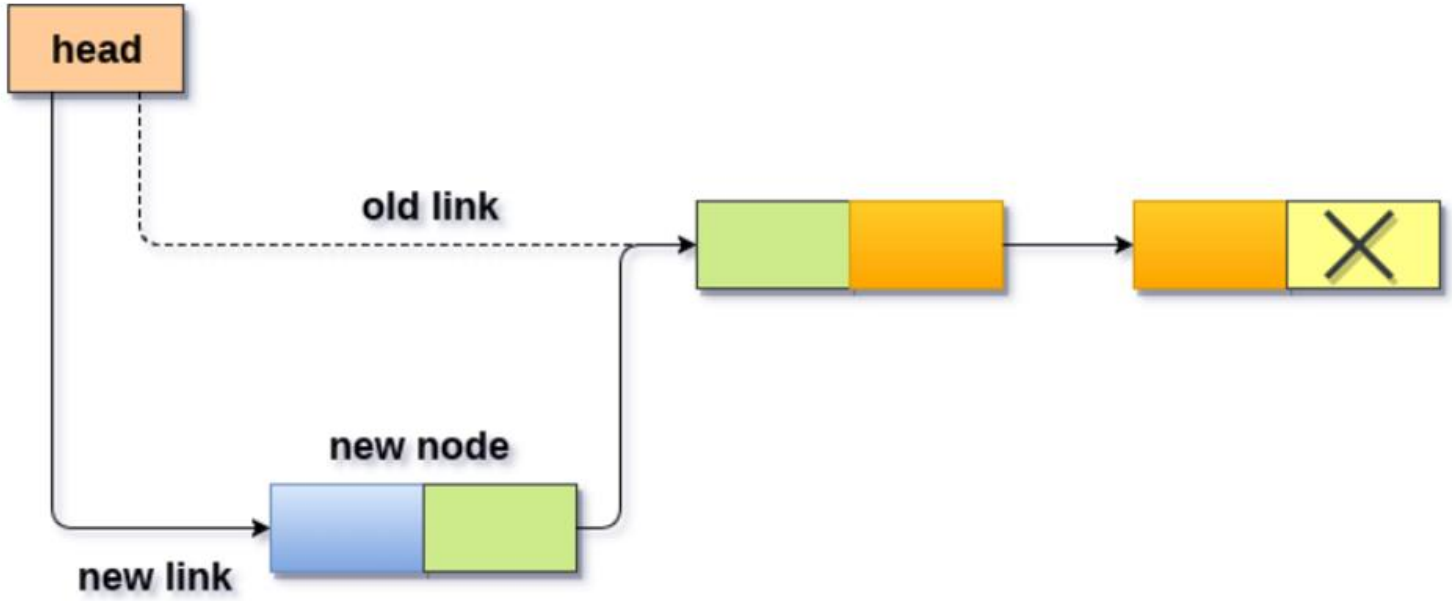
- Suppose START is the first position in linked list.
- Let DATA be the element to be inserted in the new node.
- POS is the position where the new node is to be inserted.
- TEMP is a temporary pointer to hold the node address.

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode → DATA = DATA (given from user)
4. If (START==NULL)

   (a) NewNode → next = NULL

   Else

   (a) NewNode → next = START
5. START = NewNode
6. Exit



Insert a Node at the end of Linked List

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode → DATA = DATA
4. NewNode → Next = NULL
5. If (START == NULL)

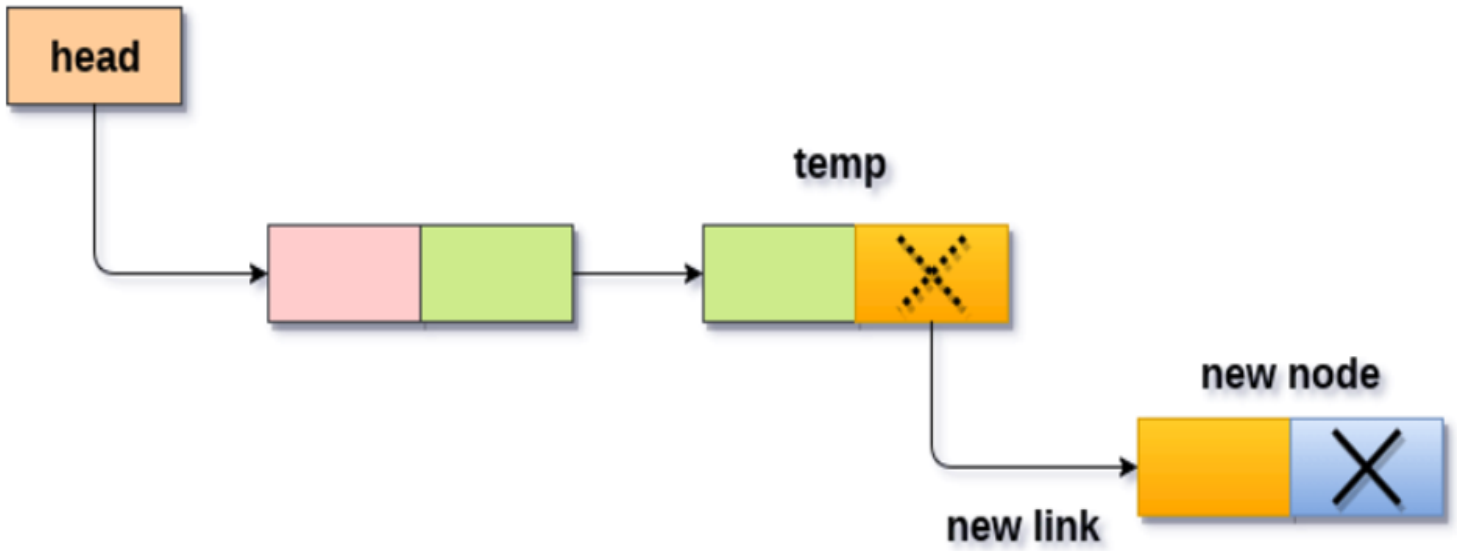
   START = NewNode

   Else

   (a) TEMP = START

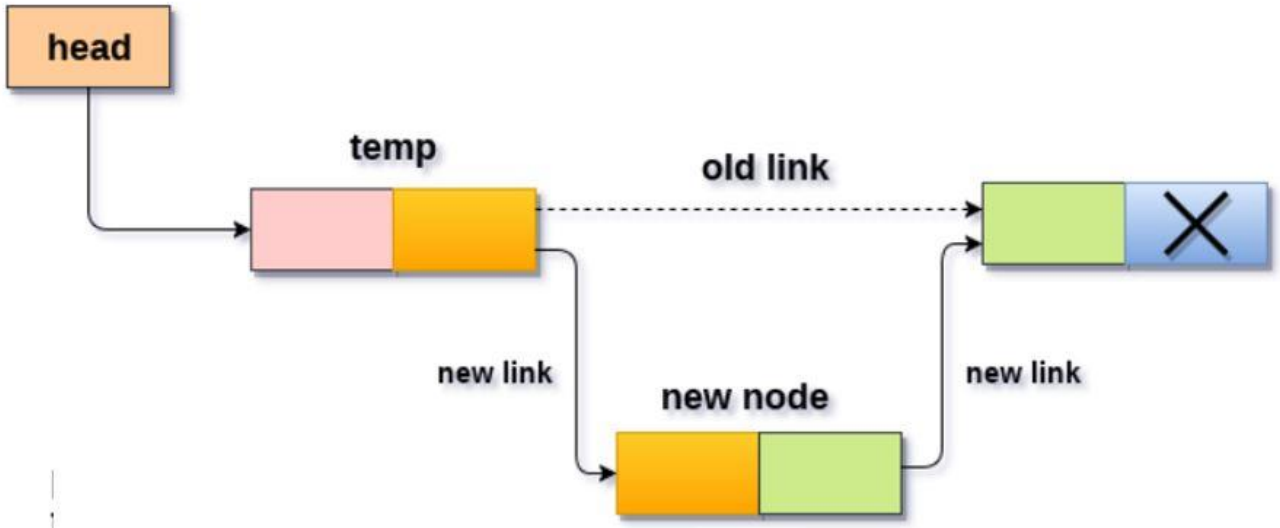
   (b) While (TEMP → Next !=NULL)

   TEMP = TEMP → Next
6. TEMP → Next = NewNode
7. Exit



Inserting a new node at the middle of the list (random location)

1. Input DATA and POS to be inserted
2. Initialize TEMP = START; and i = 1
3. for(i=1;i<POS;i++)
   (a) TEMP = TEMP->Next
   (b) If (TEMP == NULL)
      i. Display “Less Number of nodes than POS”
      ii. Exit
4. Create a New Node
5. NewNode → DATA = DATA
6. NewNode → Next = TEMP → Next
7. TEMP → Next = NewNode
8. Exit



Singly Linked List Deletion

Similar to insertion, here we also have three cases.

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

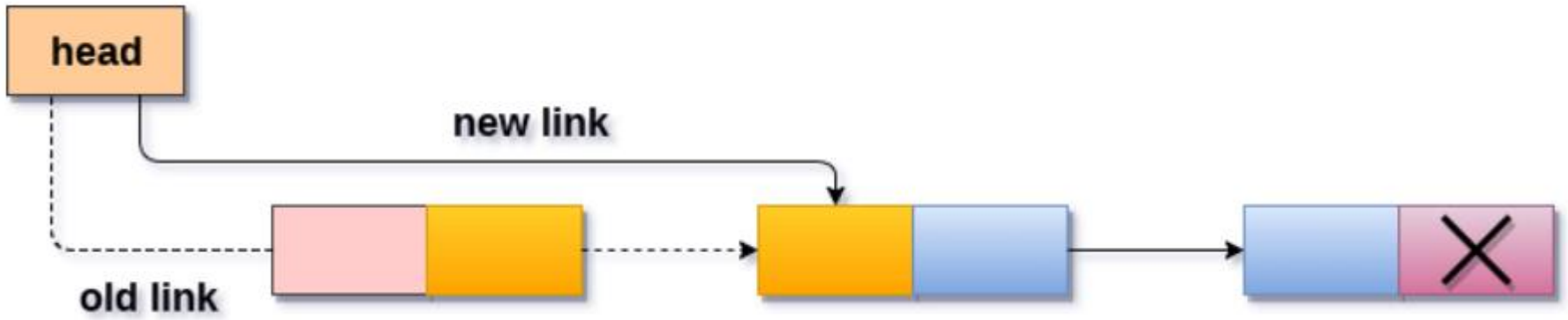
Deleting the first node in Singly Linked List

1. If START== NULL

   print(‘ No Nodes in the list’)

   Else

2. Temp= START
3. START=START→ Next;
4. temp→ Next= NULL
5. free(temp)



Deleting the last node in Singly Linked List

1. If START== NULL

   print(‘ No Nodes in the list’)

   Else

2. Initialize: Temp= START

   While (Temp→ Next !=NULL)

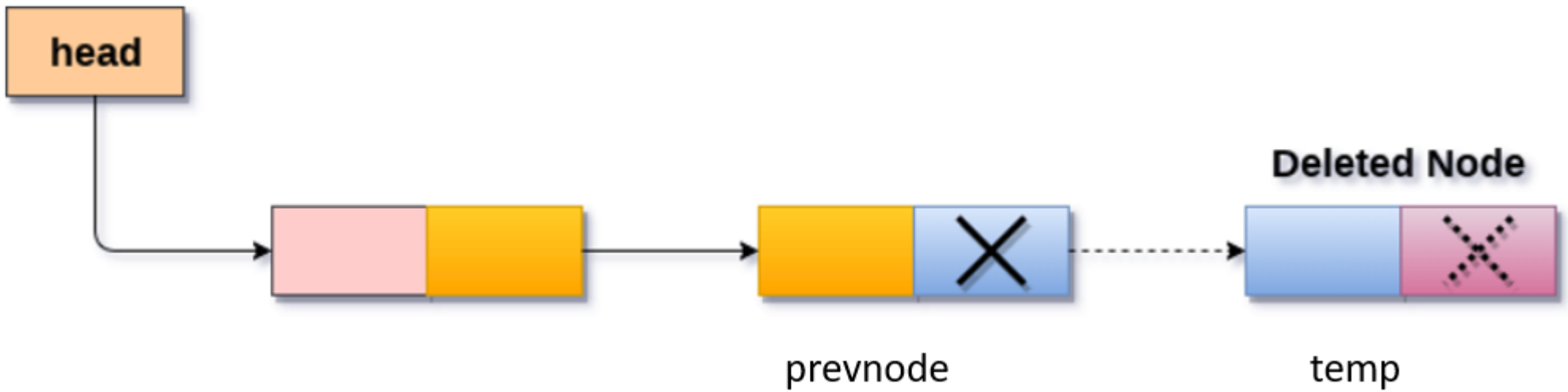
   {

   prevNode=Temp

   Temp=Temp→ Next

   }

3. prevnode→ Next= NULL
4. free(temp)



Deleting at specific position at Singly Linked List

1. If START== NULL

   print(‘ No Nodes in the list’)

   Else

2. Temp= START,i=1

   for(i=1;i<POS;i++)

   {

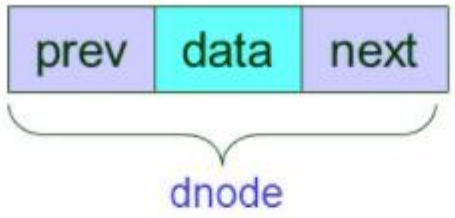
   Temp=Temp→ Next

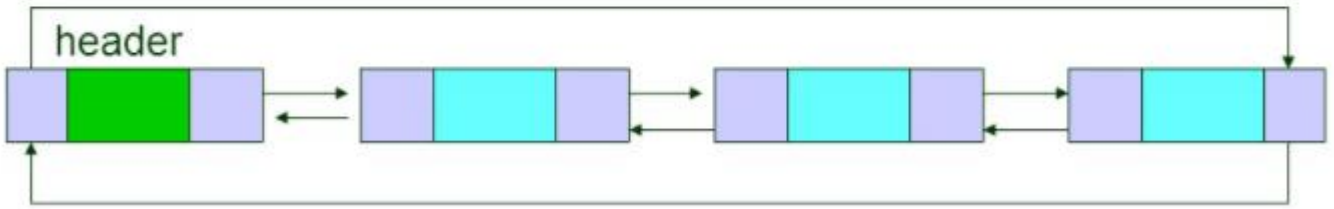
   }

3. Nextnode= Temp→ Next
4. Temp→ Next = Nextnode→ Next
5. free(nextnode)

2. Doubly Linked List

- It is complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.
- It consists of three parts—data, a pointer to the next node (next pointer), and a pointer to the previous node (previous pointer).
- In double linked list, the first node must be always pointed by head.
- Always the previous field of the first node must be NULL.
- Always the next field of the last node must be NULL.





Doubly Linked List Insertion

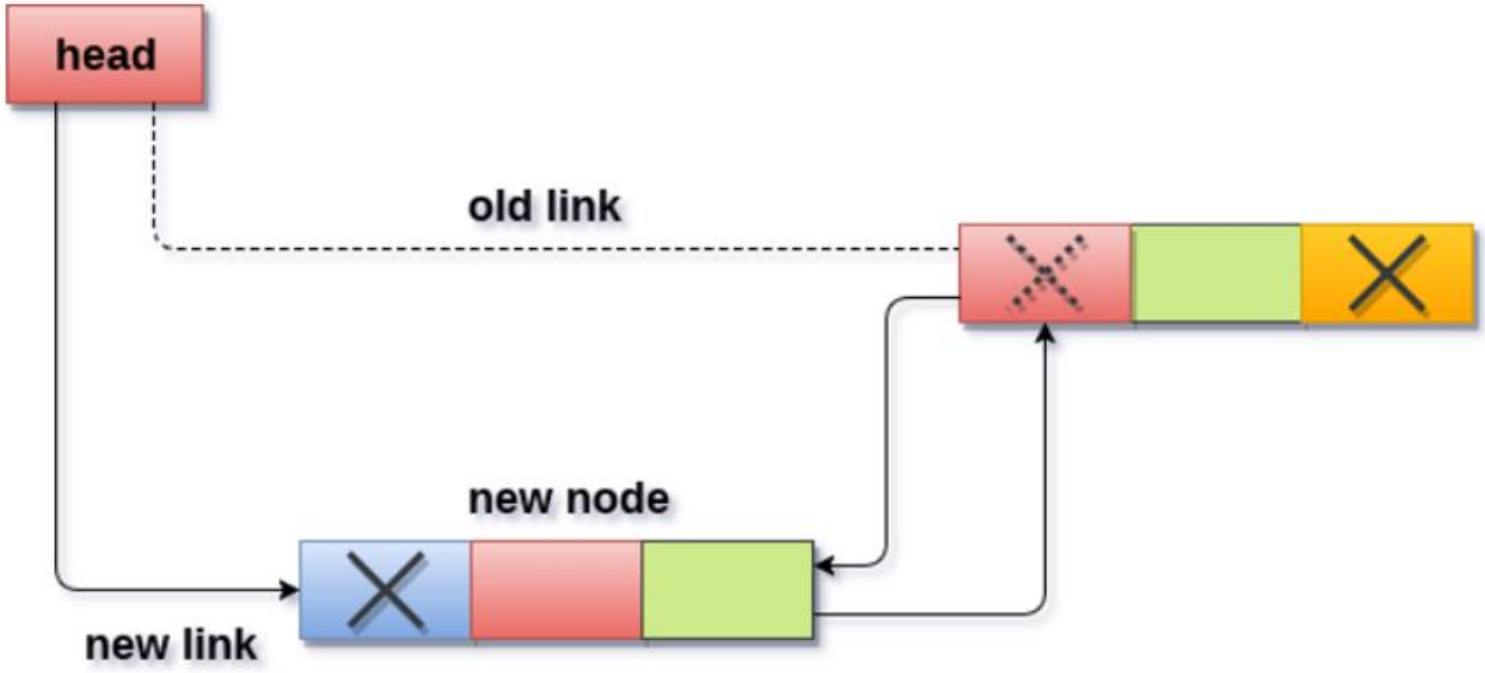
Insertion into a doubly-linked list has three cases (same as singly linked list):

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

Insert a Node at the beginning of Linked List

- Suppose START is the first position in linked list.
- Let DATA be the element to be inserted in the new node.
- POS is the position where the new node is to be inserted.
- TEMP is a temporary pointer to hold the node address.

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode → DATA = DATA (given from user)
4. NewNode → prev = NULL
5. NewNode → Next = NULL [optional]
6. START → prev = NewNode
7. NewNode → Next = START
8. START = NewNode



Doubly Linked List Deletion

Similar to insertion, here we also have three cases.

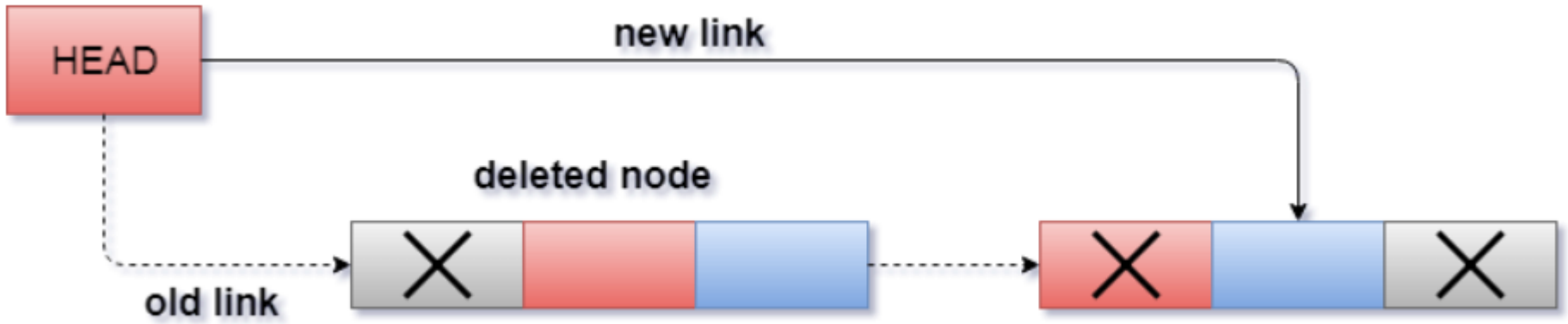
- Deleting the first node
- Deleting the last node
- Deleting an intermediate node.

Deleting the first node in Doubly Linked List

1. If START== NULL

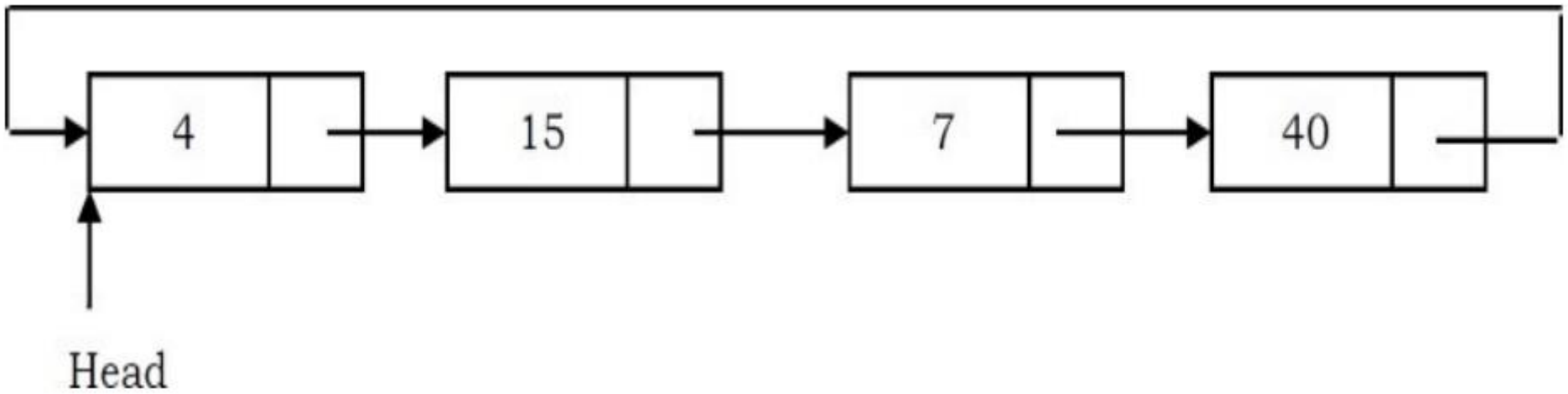
   print(‘ List is Empty’)

   Else
2. Temp= START
3. START=START→ Next;
4. START → prev= NULL
5. free(temp)
6. Exit



Circular Linked List / Circular singly linked list

- It is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.
- In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.



Circular Linked List Insertion

Insertion into a circular linked list has three cases:

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

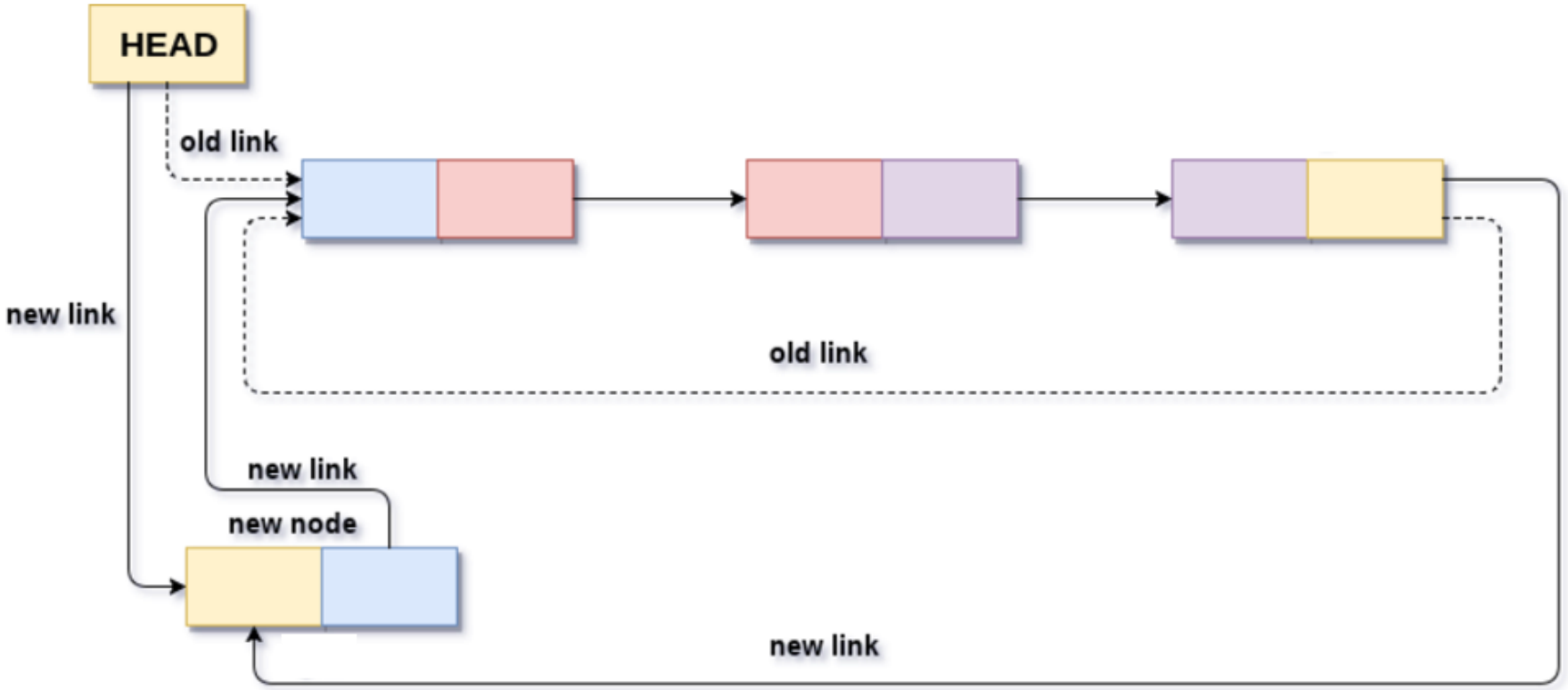
Inserting a new node at the beginning in Circular linked list

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode → DATA = DATA (given from user)
4. Temp = START
5. While (Temp→ Next != START)

   {

   Temp= Temp → Next

   }
6. NewNode → next = START
7. Temp → Next = NewNode
8. Start = NewNode
9. Exit



Circular Linked List Deletion

Similar to insertion, here we also have three cases.

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

Deleting the first node in Circular Linked List

1. If START== NULL

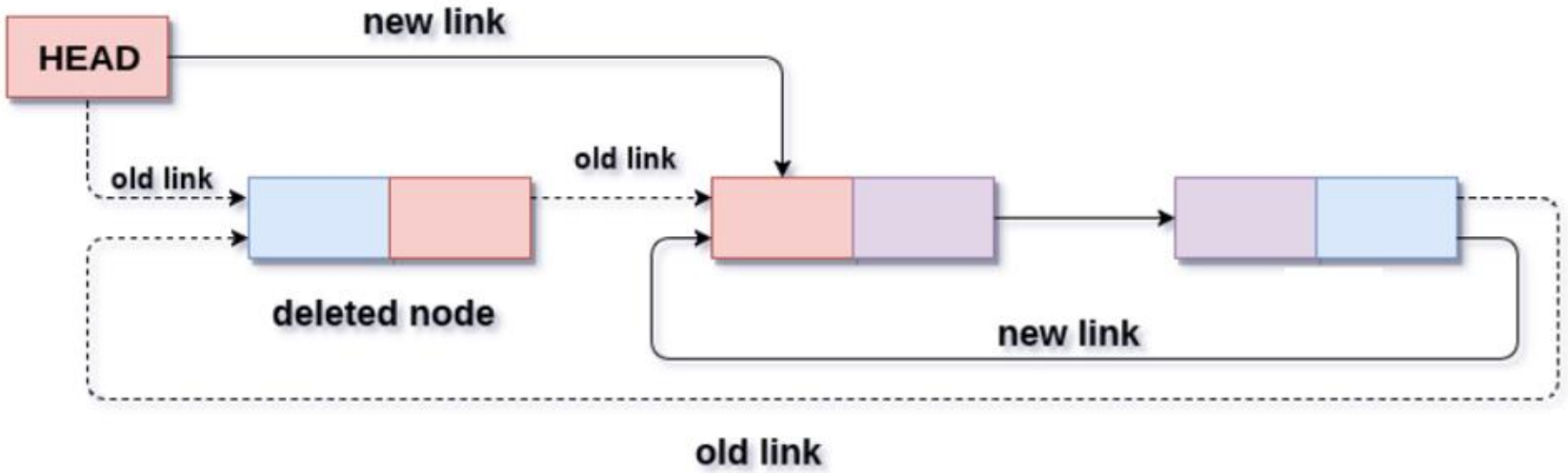
   print(‘ Empty List’)

   Else
2. Temp= START
3. While (Temp → Next != START)

   {

   Temp = Temp → Next

   }
4. Temp → Next = Start → Next
5. START = Temp → Next
6. Exit



Circular Linked List Deletion

Similar to insertion, here we also have three cases.

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

Deleting the first node in Circular Linked List

1. If START== NULL

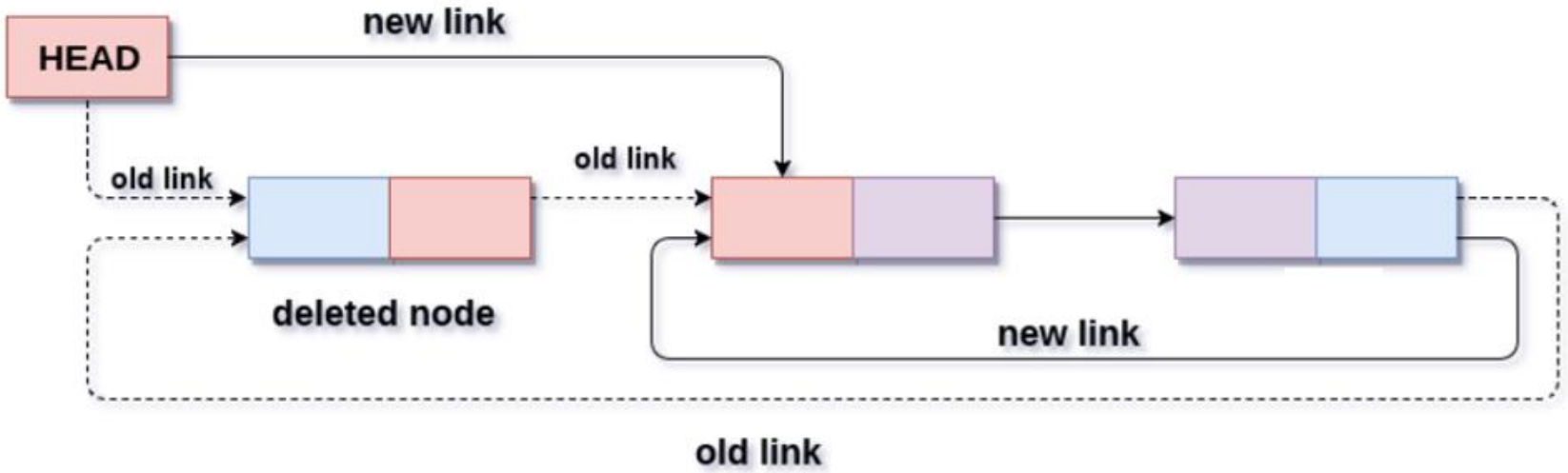
   print(‘ Empty List’)

   Else
2. Temp= START
3. While (Temp → Next != START)

   {

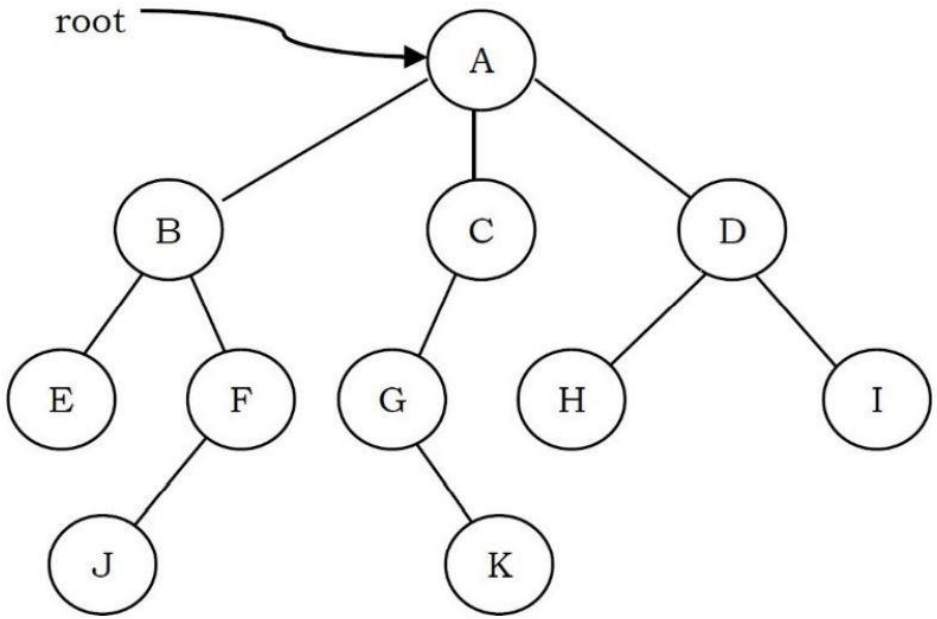
   Temp = Temp → Next

   }
4. Temp → Next = Start → Next
5. START = Temp → Next
6. Exit

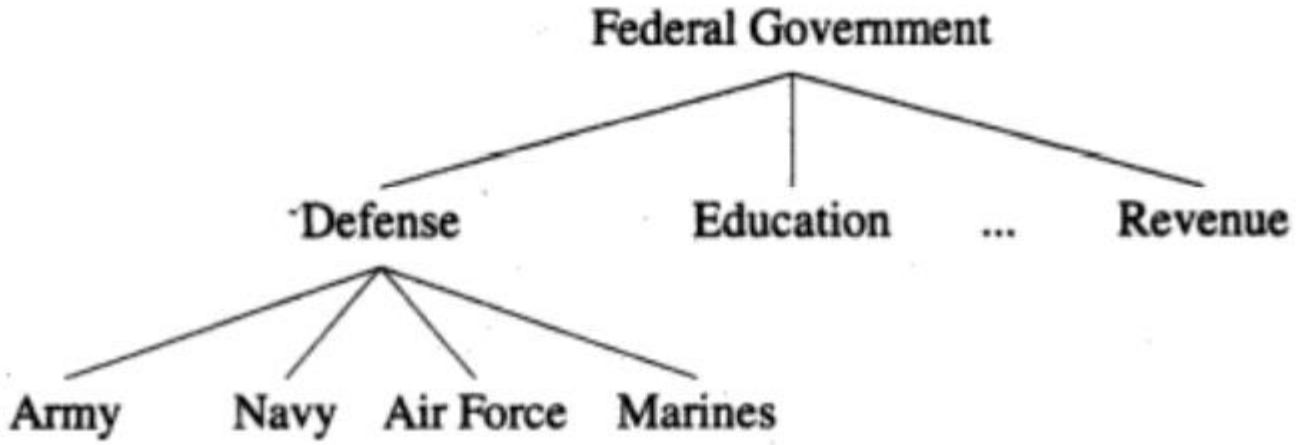


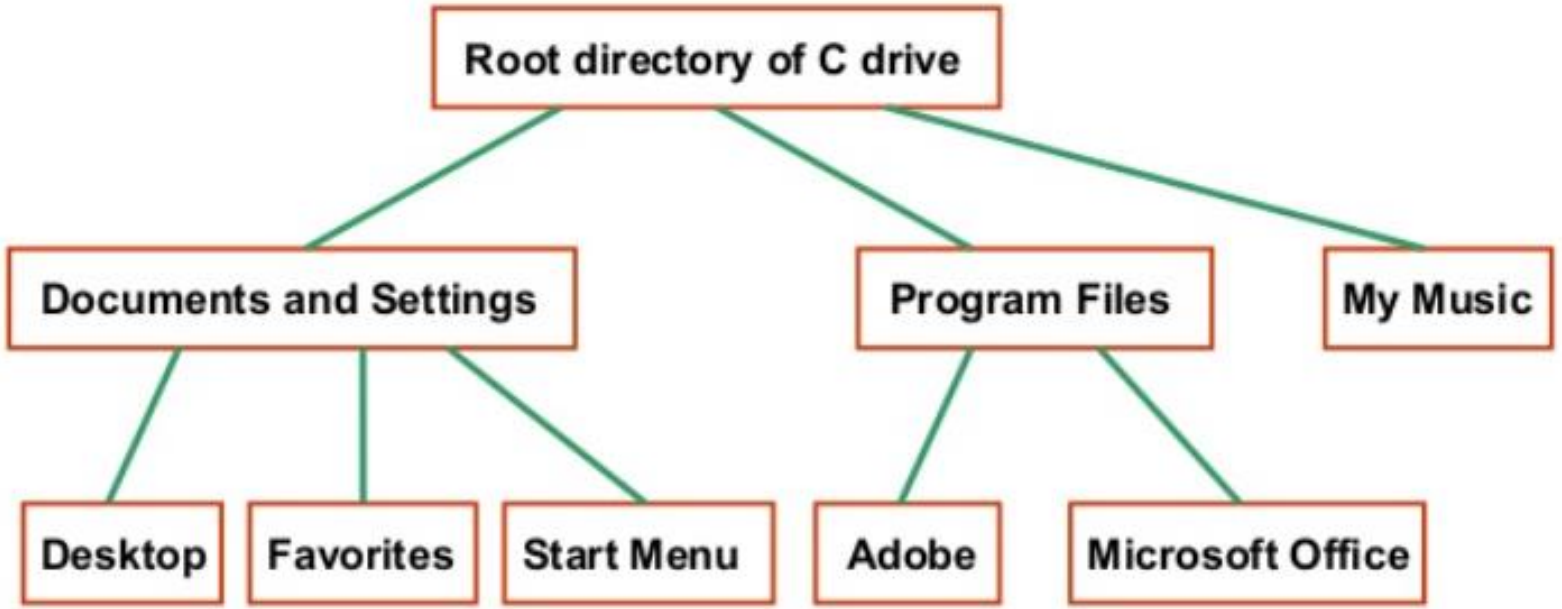
Tree:

- Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- It is a collection of nodes that are linearly connected and don't have any cyclic relations .
- It represents the nodes connected by edges.



Tree Example : Modules of federal Government

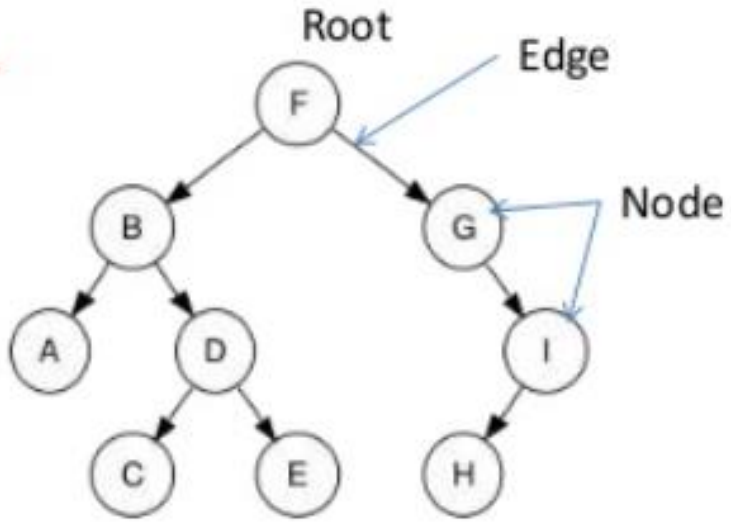




Properties :

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

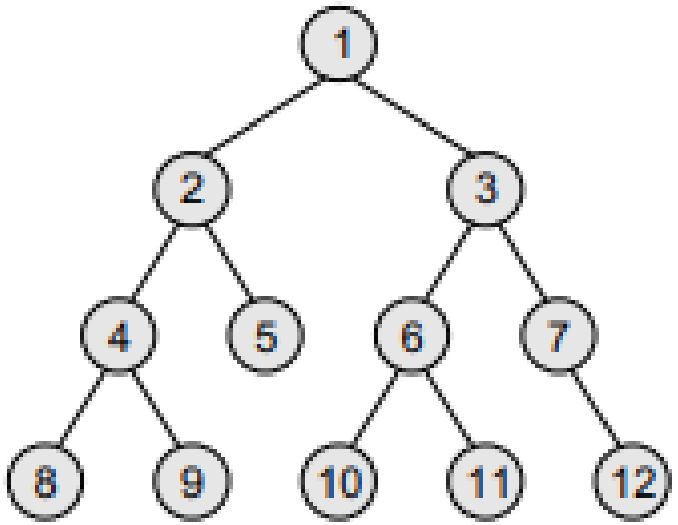
Tree Terminology :-



1. Node: Each Elements of Tree is called node
2. Edges: The Lines Connecting the nodes are called Edges.
3. Parent Node: The Immediate predecessor of node is called parent node.
4. Grand Parent: Parent of Parent
5. Child Node: All the immediate successor of a node are its child node.
6. Root Node: The node does not having any parent.
7. Leaf Node: The node does not having any child called Leaf node or Terminal node.
8. Level: Level of Node is distance of that node from root.
   (Rot node at distance 0 from itself hence is at level 0)
9. Height: Total No. of level in tree (is equal to depth of tree)
10. Degree of Node: No. of children it has.
11. Siblings: children of the same parent node (two or more node have same parent are called sibling or brother

Binary Tree :

- It is a tree in which every node can have a maximum of two children.
- In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.
- One is known as left child and the other is known as right child.

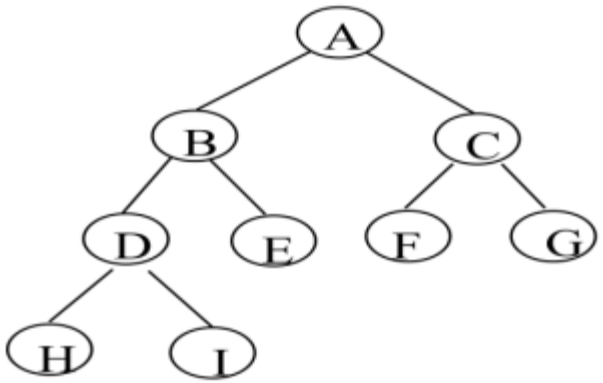


Types of binary tree

I. Strictly Binary Tree
II. Complete Binary Tree
III. Almost Complete Binary Tree

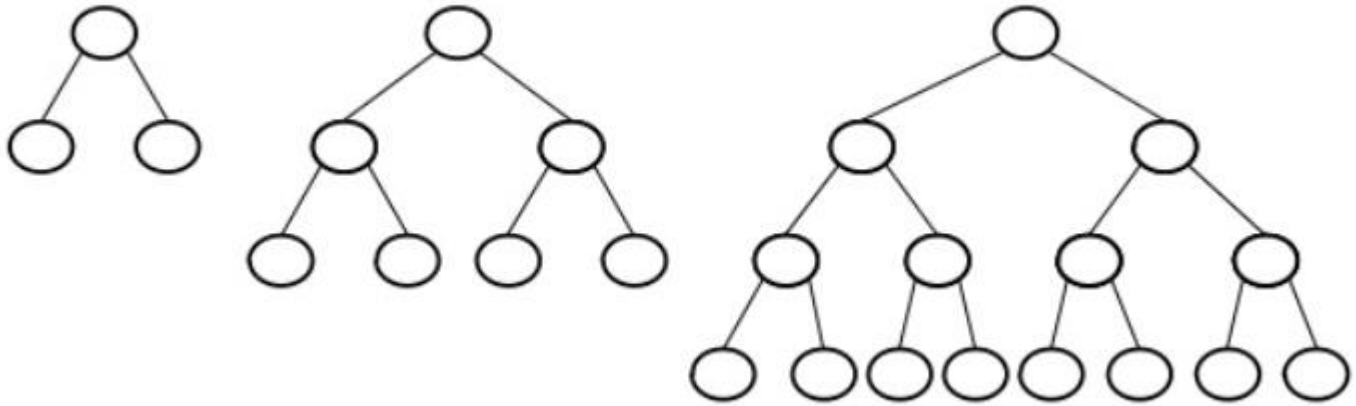
I. Strictly Binary Tree:

- every node should have exactly two children or none.
- also called as Full Binary Tree or Proper Binary Tree.



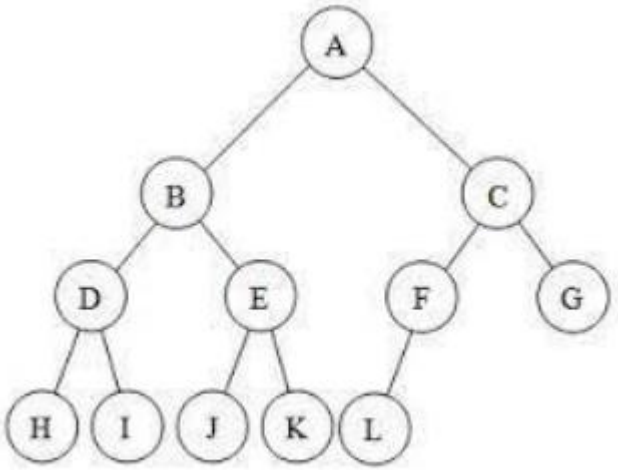
II. Complete Binary Tree :

- A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.
- In it, all the nodes must have exactly two children and at every level of complete binary tree there must be 2 level number of nodes.
- For example at level 2 there must be 2^2 = 4 nodes and at level 3 there must be 2^3 = 8 nodes.
- Also called perfect binary tree.



III. Almost Complete Binary Tree :

- A Binary Tree is almost complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible.
- A binary tree of depth d is an almost binary tree if:
  - Any node ‘nd’ at level less than‘d-1’ has two sons.
  - For any node ‘nd’ in the tree with a right descendant at level d, nd must have a left son and every left descendant of nd is either a leaf at level d or has two sons.



Binary Tree Traversal :

- Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way.

There are three popular methods of traversal:

i. Inorder traversal

ii. Preorder traversal

iii. Postorder traversal

i. Inorder traversal (LeftChild - Root - RightChild) :

-The Inorder traversal of a nonempty binary tree is defined as follows:

- Traverse the left sub-tree in Inorder
- Visit the root node
- Traverse the right sub-tree in Inorder

```
C function for inorder traversing:

While ( Root != NULL)
{
        INORDER(Root-> LEFT)
        Write Root-> DATA
        INORDER(Root -> RIGHT)
}
```

ii. Preorder traversal (Root - LeftChild - RightChild) :

The preorder traversal of a nonempty binary tree is defined as follows:

- Visit the root node
- Traverse the left sub-tree in preorder
- Traverse the right sub-tree in preorder

```
C function for Preorder traversing:

While ( Root != NULL)
{
        Write Root-> DATA
        PREORDER(Root-> LEFT)
        PREORDER(Root -> RIGHT)
}
```

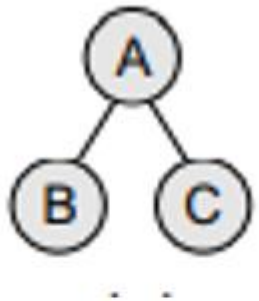
iii. Postorder traversal (LeftChild – RightChild - Root) :

The postorder traversal of a nonempty binary tree is defined as follows:

- Traverse the left sub-tree in postorder
- Traverse the right sub-tree in postorder
- Visit the root node

C function for Postorder traversing:

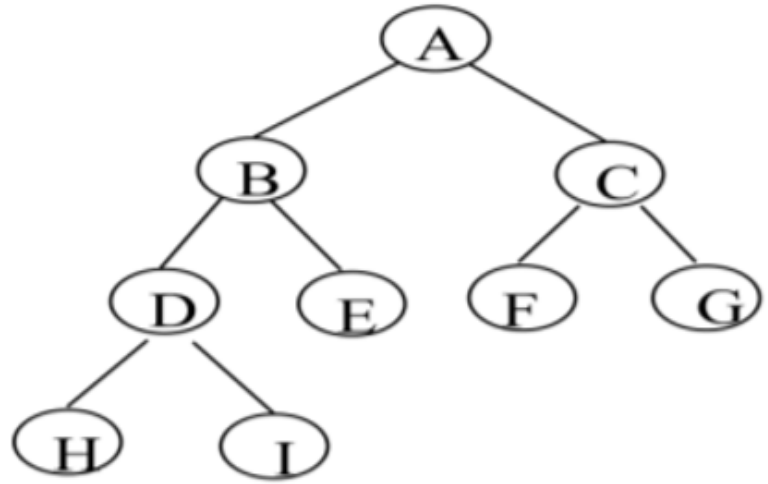
```
While ( Root != NULL)
{
        POSTORDER(Root-> LEFT)
        POSTORDER(Root -> RIGHT)
        Write Root-> DATA
}
```



Inorder: BAC

Preorder: ABC

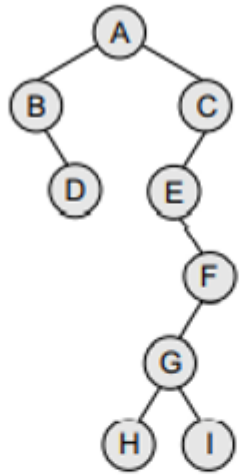
Postorder: BCA



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

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

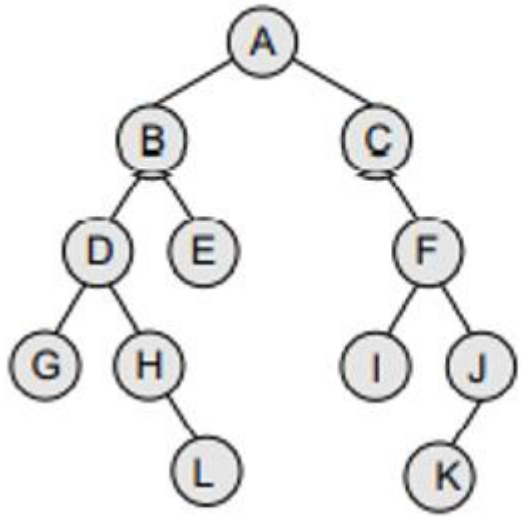
Postorder: H-I-D-E-B-F-G-C-A



Inorder: BDAEHGIFC

Preorder: ABDCEFGHI

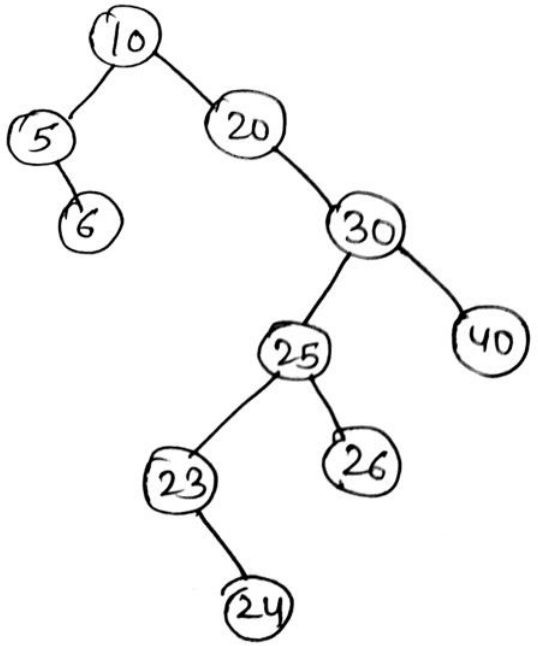
Postorder: DBHIGFECA



Inorder: GDHLBEACIFKJ

Preorder: ABDGHLECFIJK

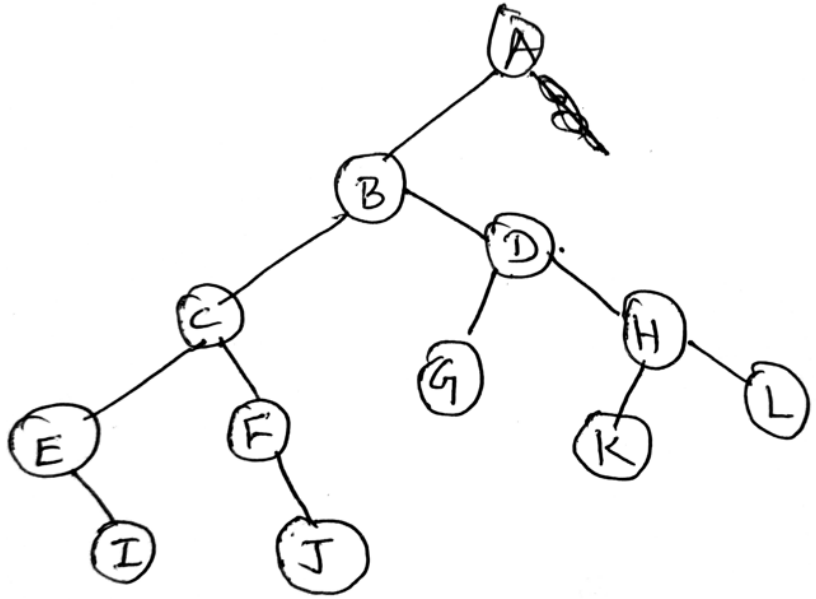
Postorder: GLHDEBIKJFCA



Preorder: 10,5,6,20,30,25,23,24,26,40

Post order: 6,5,24,23,26,25,40,30,20,10

In order: 5,6,10,20,23,24,25,26,30,40



Preorder: ABCEIFJDGHKL

Post order: IEJFCGKLHDBA

In order: EICFJBGDKHLA

Construction of binary tree from given inorder and preorder sequence

- In preorder sequence, leftmost element (first element) is the root of the tree.
- In Inorder sequence, the root of the tree is in between the left subtree and right subtree.

Algorithmic Steps:

Step 1: Scan the preorder sequence from left to right. Pack an element ‘X’ from preorder and increment the preorder index value.

Step 2: Find the picked element ‘X’ in the inorder sequence.

Step 3: Build the tree using ‘X’ as a root node and before ‘X’ as left subtree of ‘X’ and after ‘X’ as right subtree of ‘X’

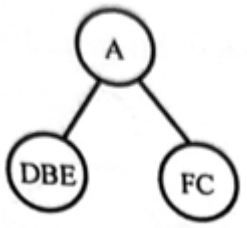
Step 4: Repeat step 1 to 3 for each symbol in preorder sequence.

Construct the binary tree considering the following sequences:

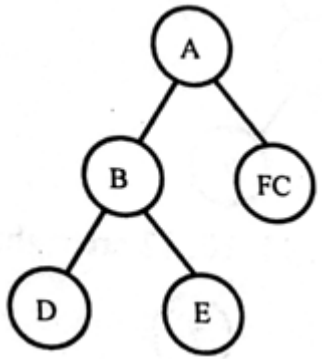
Inorder: DBEAFC

Preorder: ABDECF

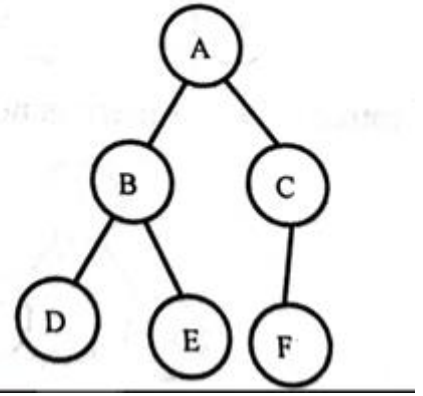
Solution: In a preorder sequence, A is first node. So, A is root node of tree. From inorder sequence, we can conclude that DBF are left subtree of A and FC are right subtree of A.



B is the second element of preorder sequence. Here, we can write that from inorder sequence, D is the left subtree of B and E is the right subtree of B.



Similary, F is the left subtree of C. Final binary tree is :



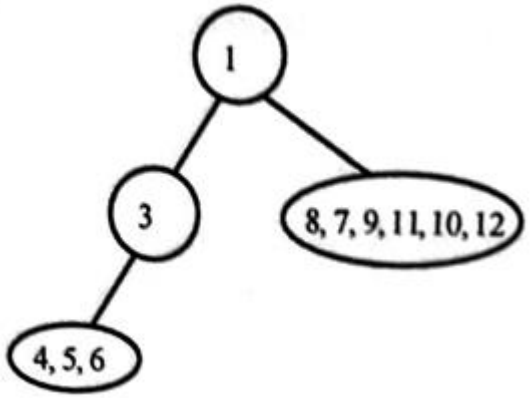
Construct the binary tree considering the following sequences:

Inorder: 4,5,6,3,1,8,7,9,11,10,12

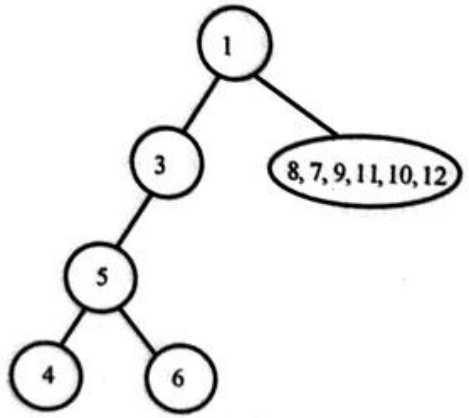
Preorder: 1,3,5,4,6,7,8,9,10,11,12

Solution: 1 is the first element of preorder.

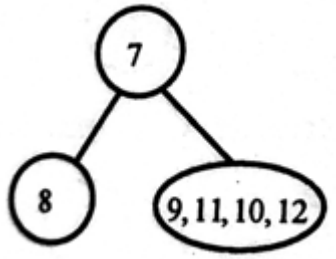
- 3 is the second element of preorder. So, from inorder sequence we can write: 4,5,6 are left subtree of 3 and there is no any right subtree of 3.



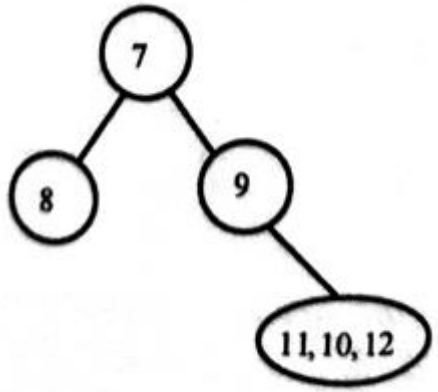
- Similarly, for 5, 4 is the left subtree and 6 is the right subtree.



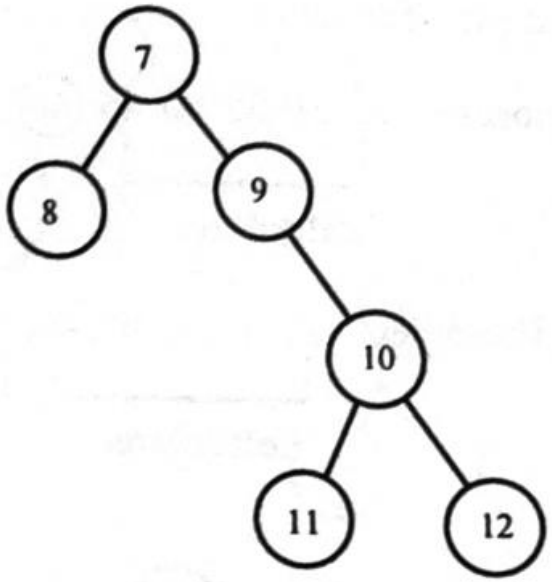
- 8 is left subtree 0f 7 and 9,11,10,12 are right subtree of 7.



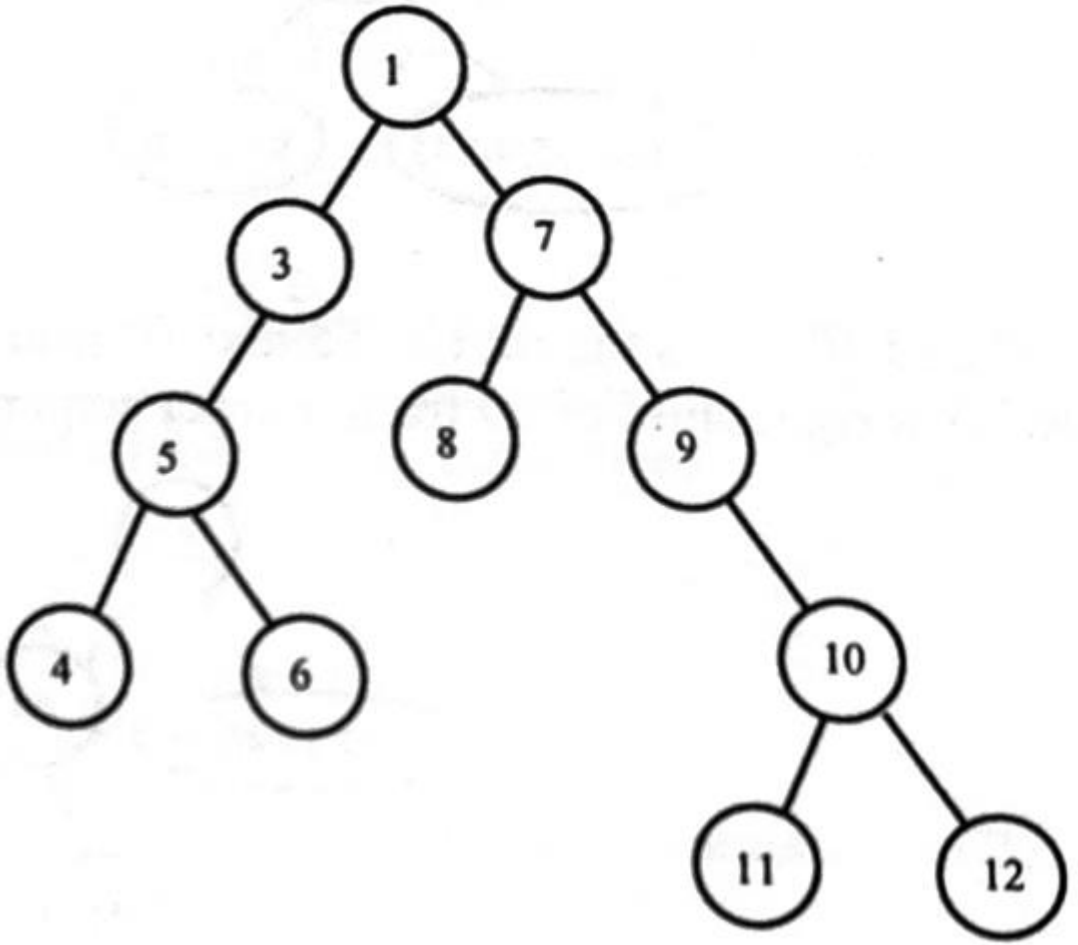
- 10,11,12 are right subtree of 9 and there is no any left subtree of 9.



- 11 is the left subtree of 10 and 12 is the right subtree of 10.



- The final binary tree is:



Construction of binary tree from given inorder and postorder sequence

- In postorder sequence, rightmost element (last element) is the root of the tree.
- In Inorder sequence, the root of the tree is in between the left subtree and right subtree.

Algorithmic Steps:

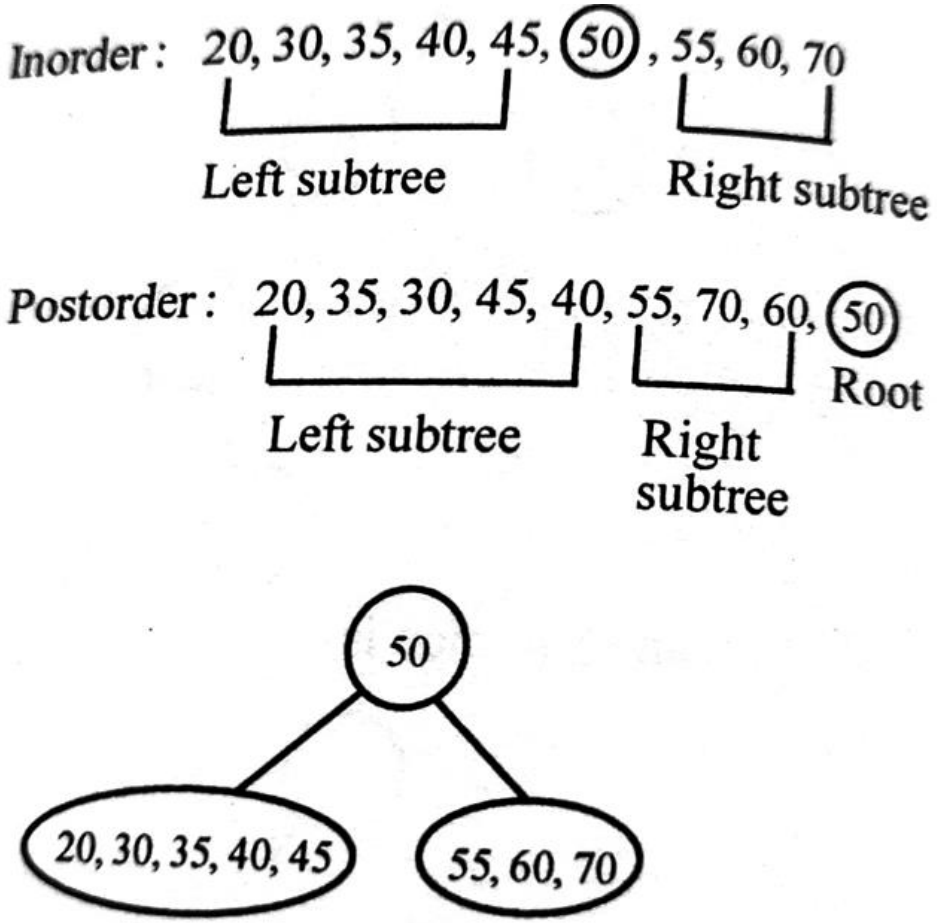
- Step 1: Scan the postorder sequence from right to left. last element from the postorder sequence is the root of the tree.
- Step 2: Search the same element in inorder sequence when the element is found in inorder sequence, then all the elements present before the node found are left child or subtree and all the elements present after the node found are right children or subtree of node found.
- Step 3: Repeat step 1 to 3 for each symbol in postorder sequence.

Construct the binary tree considering the following sequences:

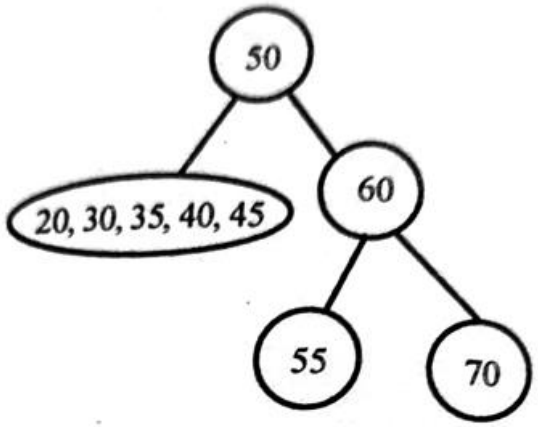
Inorder: 20, 30, 35,40,45,50,55,60,70

Postorder: 20,35,30,45,40,55,70,60,50

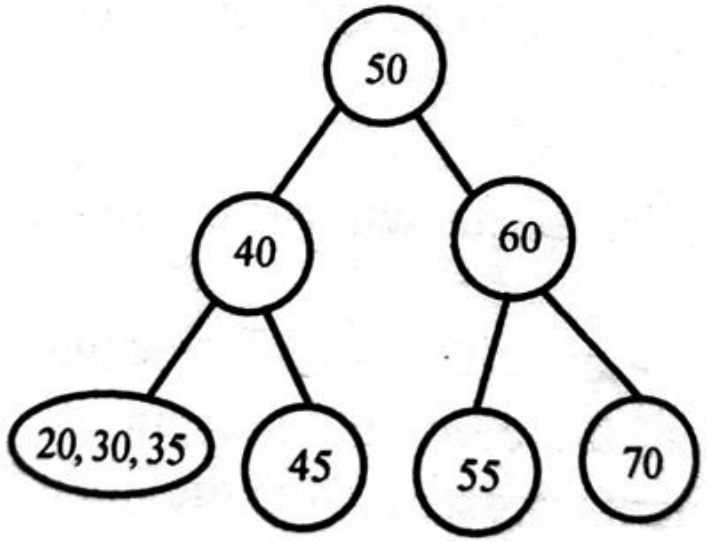
Solution: 50 is the root node of the tree.



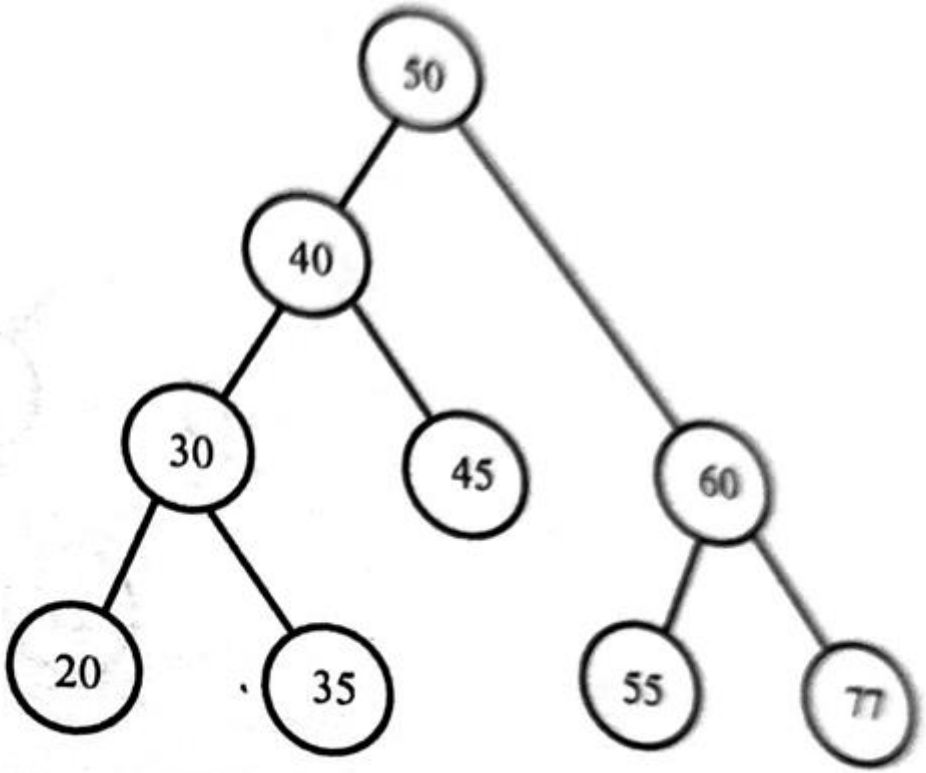
- For 55,60 and 70, 60 is the root of 55 and 70 from postorder sequence and 55 is left child and 70 is right child of 60 from inorder sequence.



- Similarly, 20, 30, 35 are left childs of 40 and 45 is the right child of 40.



- Similarly, 20 is the left child of 30 an 35 is the right child of 30. Hence, final binary tree is:

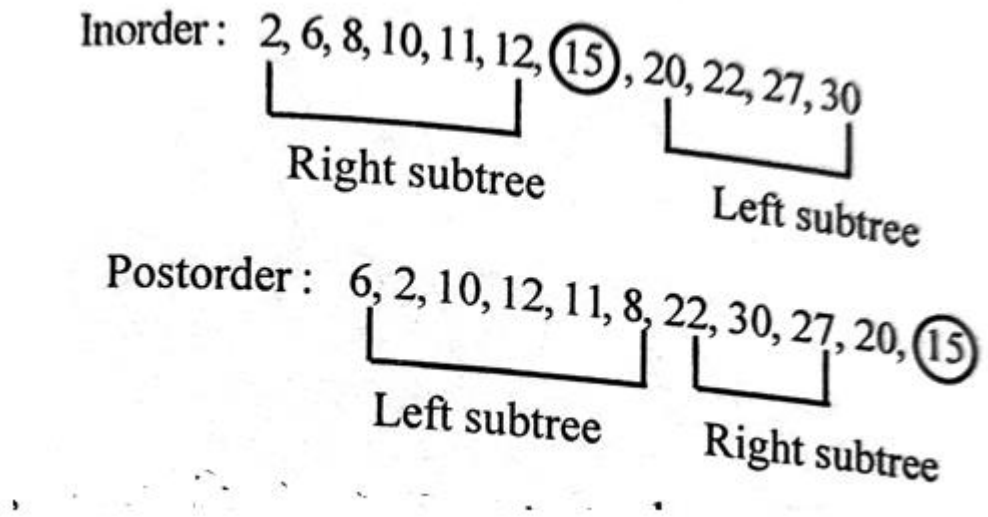


- Construct the binary tree considering the following sequences:

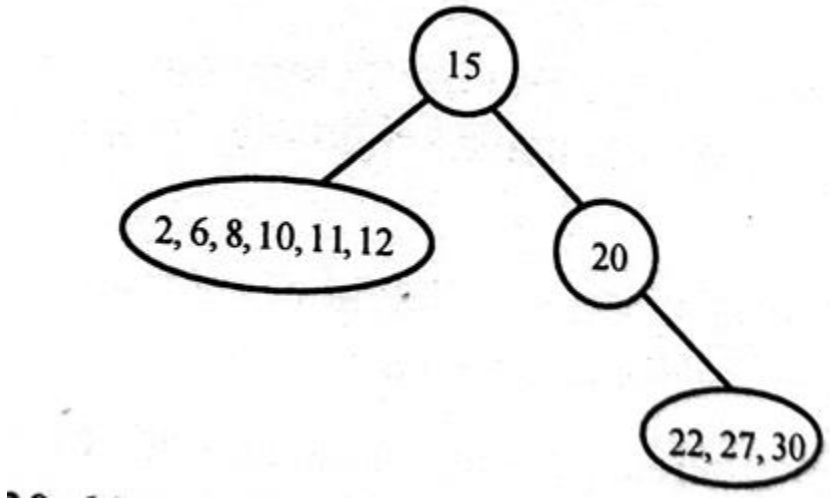
Inorder: 2,6,8,10,11,12,15,20,22,27,30

Postorder: 6,2,10,12,11,8,22,30,27,20,15

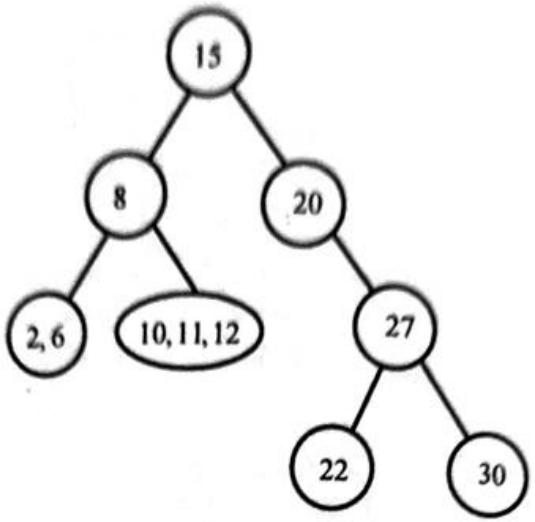
Solution: 15is the root node of the tree.



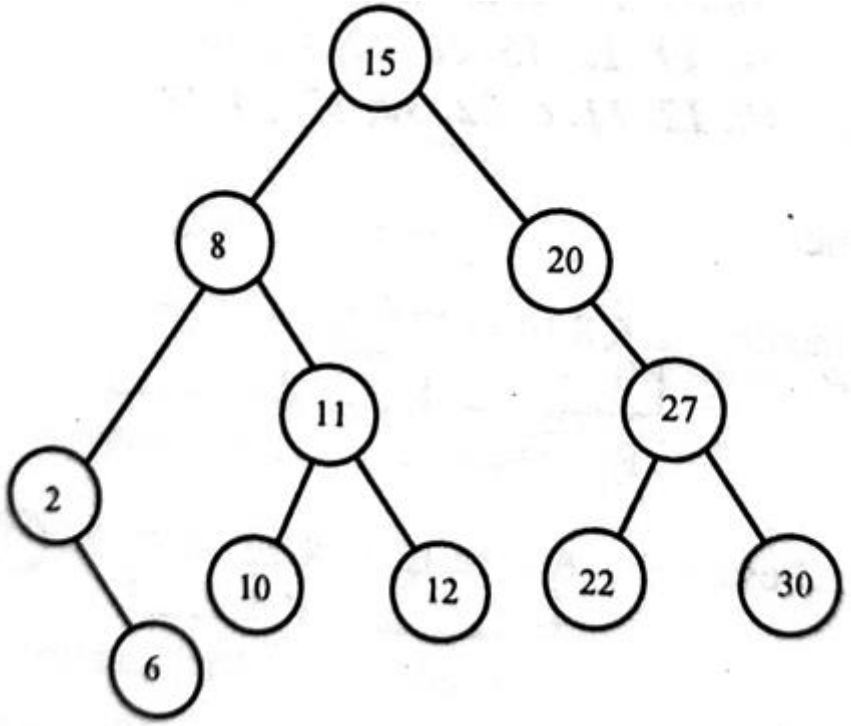
- For 22,30,27 and 20, there is no left subtree of 20 element and 22,27,30 are right subtree of 20.



- For 22,27 and 30,22 is the left child and 30 is the right child of 27.
- For 2,6,8,10,11 and 12; 2 ad 6 are left child of 8 and 10, 11 and 12 are right child.



- Similarly, final binary tree of given sequence is:



Binary Search Tree

- A binary Search Tree is a binary tree that satisfied the following conditions:

1. The data elements of left sub tree are smaller than the root of the tree.

2. The data elements of right sub tree are greater than or equal to the root of the tree.

3. The left sub tree and right sub tree are also the binary search trees, i.e. they must also follow the above two rules.

Operations on Binary search Tree

Following operations can be done in BST:

- Search(k, T): Search for key k in the tree T. If k is found in some node of tree then return true otherwise return false.
- Insert(k, T): Insert a new node with value k in the info field in the tree T such that the property of BST is maintained.
- Delete(k, T):Delete a node with value k in the info field from the tree T such that the property of BST is maintained.
- FindMin(T), FindMax(T): Find minimum and maximum element from the given nonempty BST.

Insertion in Binary Search Tree

- In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node.
- The insertion operation is performed as follows...

1. Create a NewNode with given value and set its left and right to NULL .

2. Check whether tree is Empty.

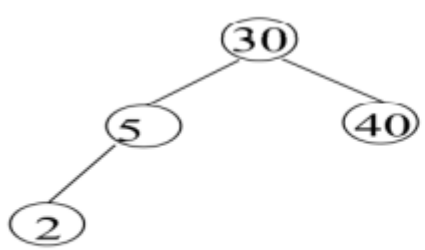
3: If the tree is Empty, then set root to NewNode.

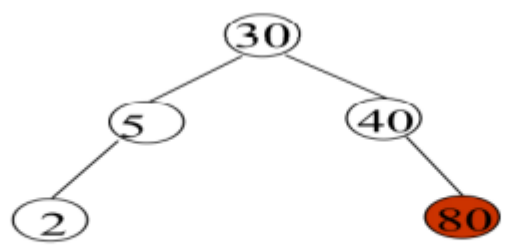
4: If the tree is Not Empty, then check whether value of NewNode is smaller or larger than the node (here it is root node).

5: If NewNode is smaller than or equal to the node, then move to its left child. If NewNode is larger than the node, then move to its right child.

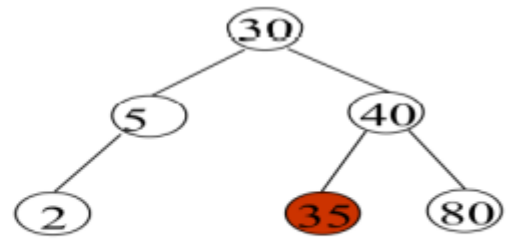
6: Repeat the above until we reach a node (i.e, reach to NULL) where search terminates.

7: After reaching a last node, then insert the NewNode as left child if NewNode is smaller or equal to that node else insert it as right child.



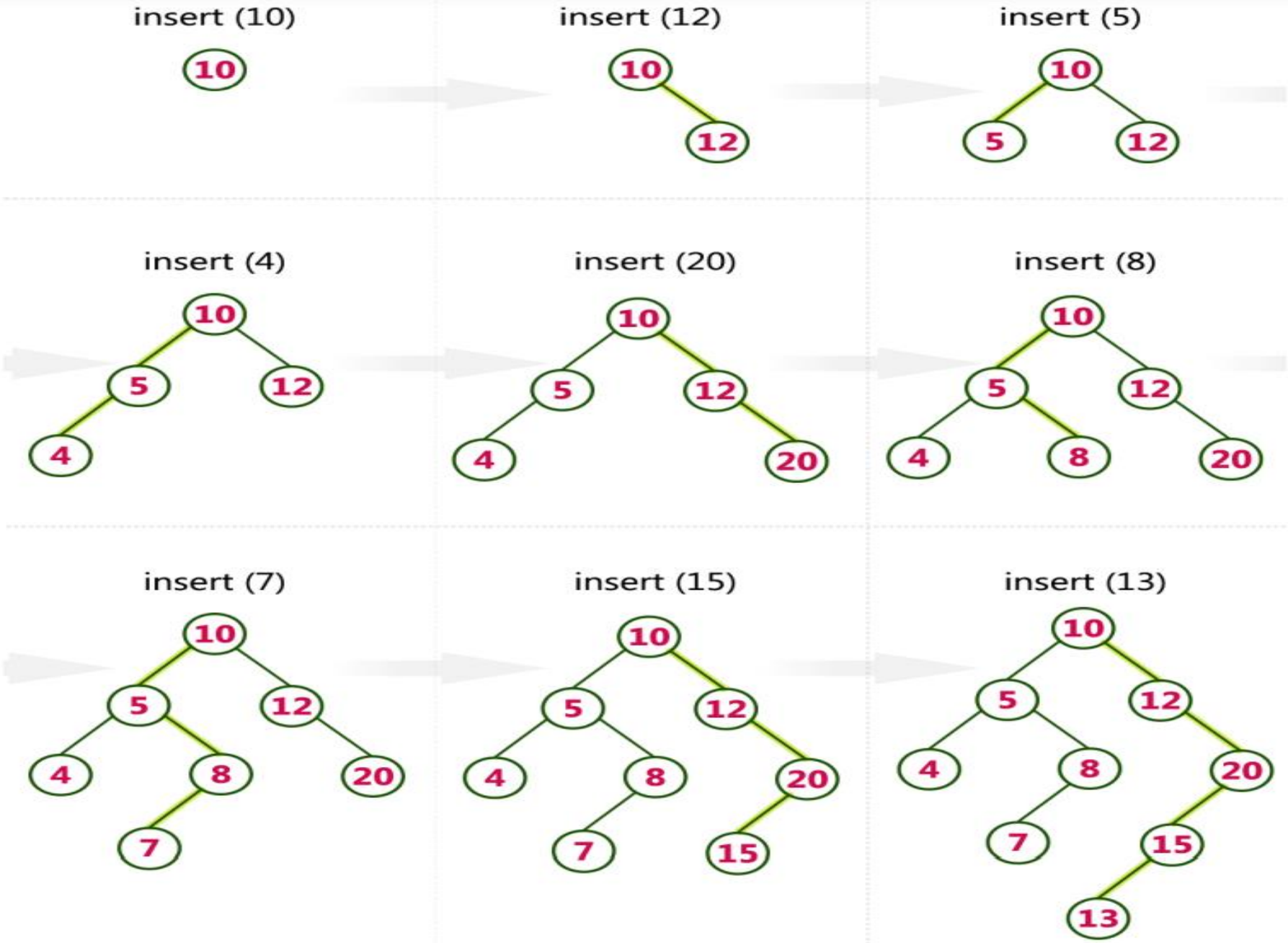


Insert 80

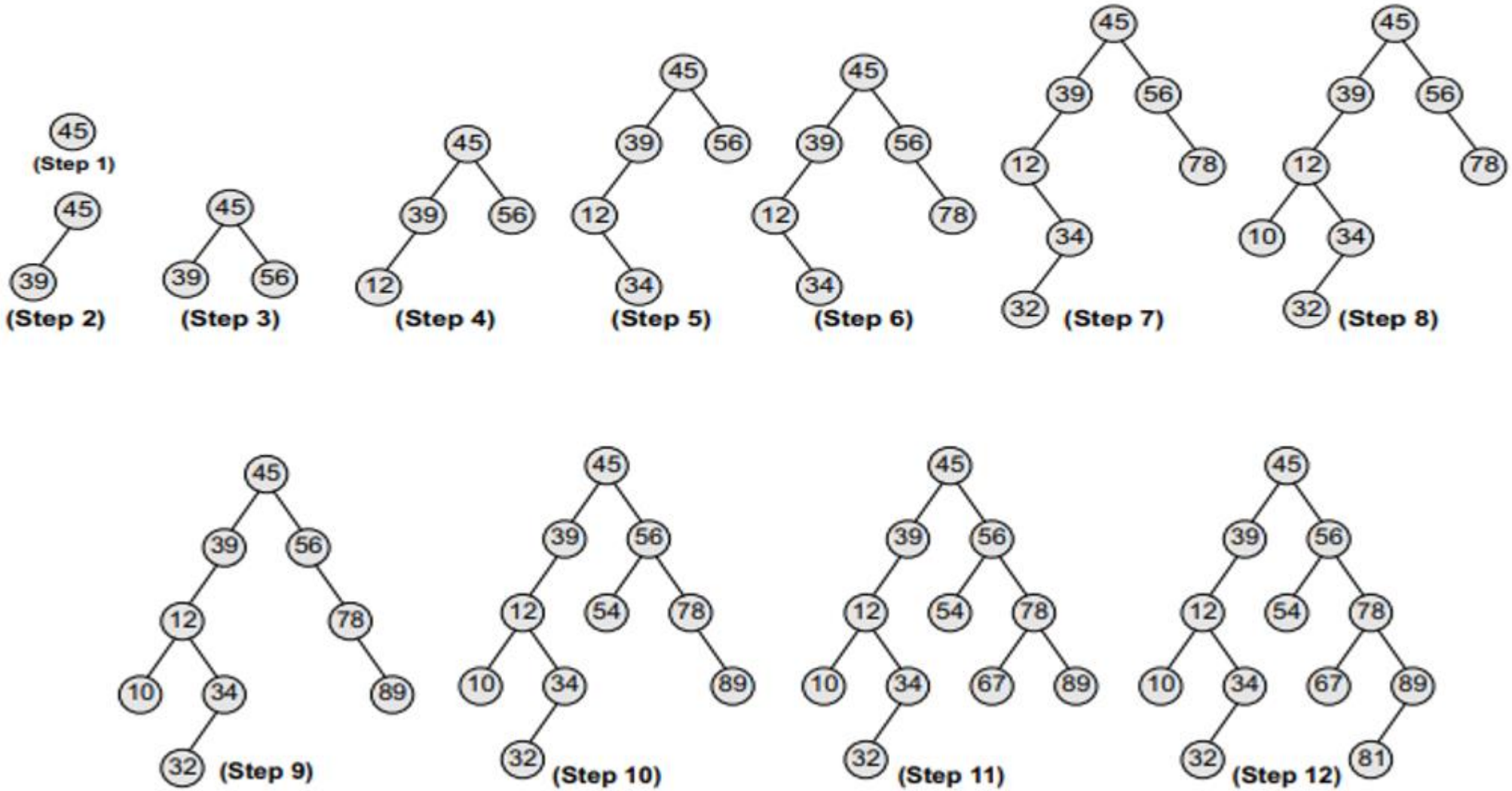


Insert 35

- Construct Binary search Tree : 10,12,5,4,20,8,7,15 and 13



Construct binary search tree : 45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81



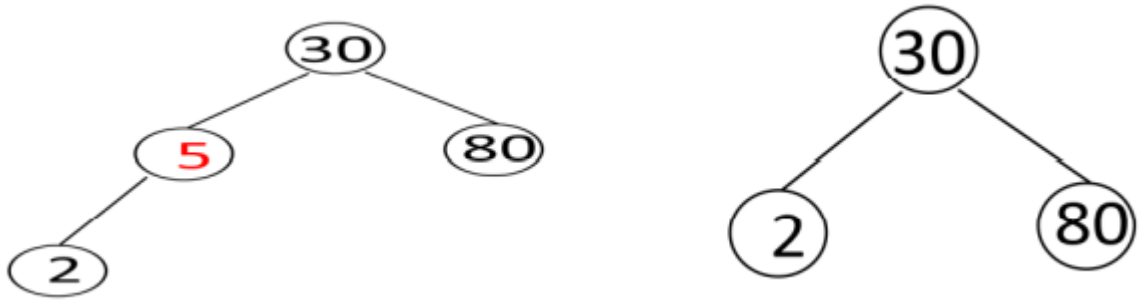
Deletion Operation in BST

Case 1: Deleting a leaf node(a node with no children)

- Find the node to be deleted using search operation
- Delete the node using free function (if it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

- Find the node to be deleted using search operation
- If it has only one child then create a link between its parent node and child node.
- Delete the node using free function and terminate the function.



Case 3: Deleting a node with two children

Step 1 - Find the node to be deleted using search operation.

Step 2 - If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.

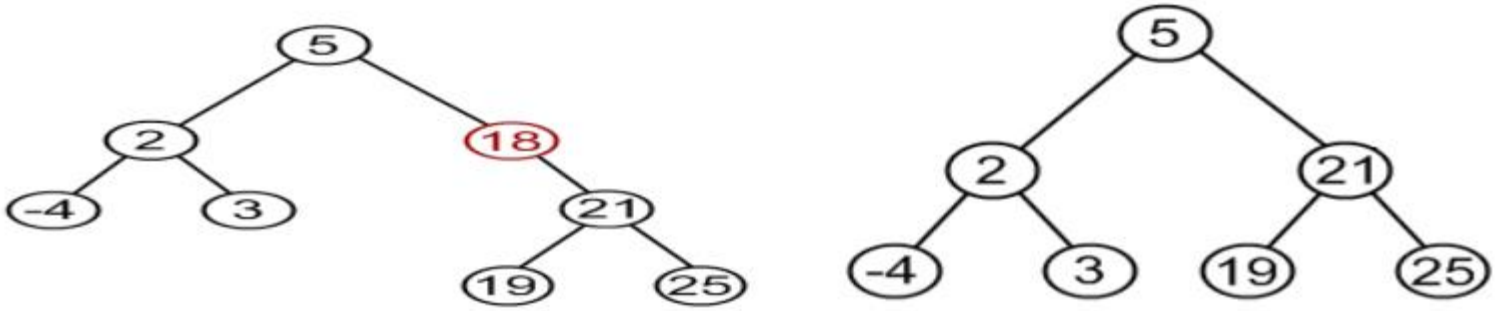
Step 3 - Swap both deleting node and node which is found in the above step.

Step 4 - Then check whether deleting node came to case 1 or case 2 or else goto step 2

Step 5 - If it comes to case 1, then delete using case 1 logic.

Step 6- If it comes to case 2, then delete using case 2 logic.

Step 7 - Repeat the same process until the node is deleted from the tree.



Balanced Search Tree

- A balanced binary tree is the one with every node not having much of difference in height of left and right subtree.
- Searching and Traversing will be much more efficient if the tree is balanced.
- For making a balanced search tree, different types of balancing trees are used. Some of them are:
- Different types of balancing tree
  - **AVL balanced tree**
  - **B-tree**
  - **Red Black tree**

AVL Tree

- G. M. Adelson-Velskii and E. M. Landis introduced a binary tree structure that is balanced with respect to the height of sub-trees.
- Height balanced tree
- AVL tree is self balancing binary search tree in which
  - the balance factor of every node is either -1, 0, or +1
  - the balance factor is calculated as: $H_L - H_R = -1, 0, \text{or } 1$

where, H_L and H_R are the heights of left and right subtrees for any given node

- the left subtree and right subtree should be again AVL
- if the difference is more than one, then the tree is rebalanced by applying certain rules of rotation

Rules of Rotation

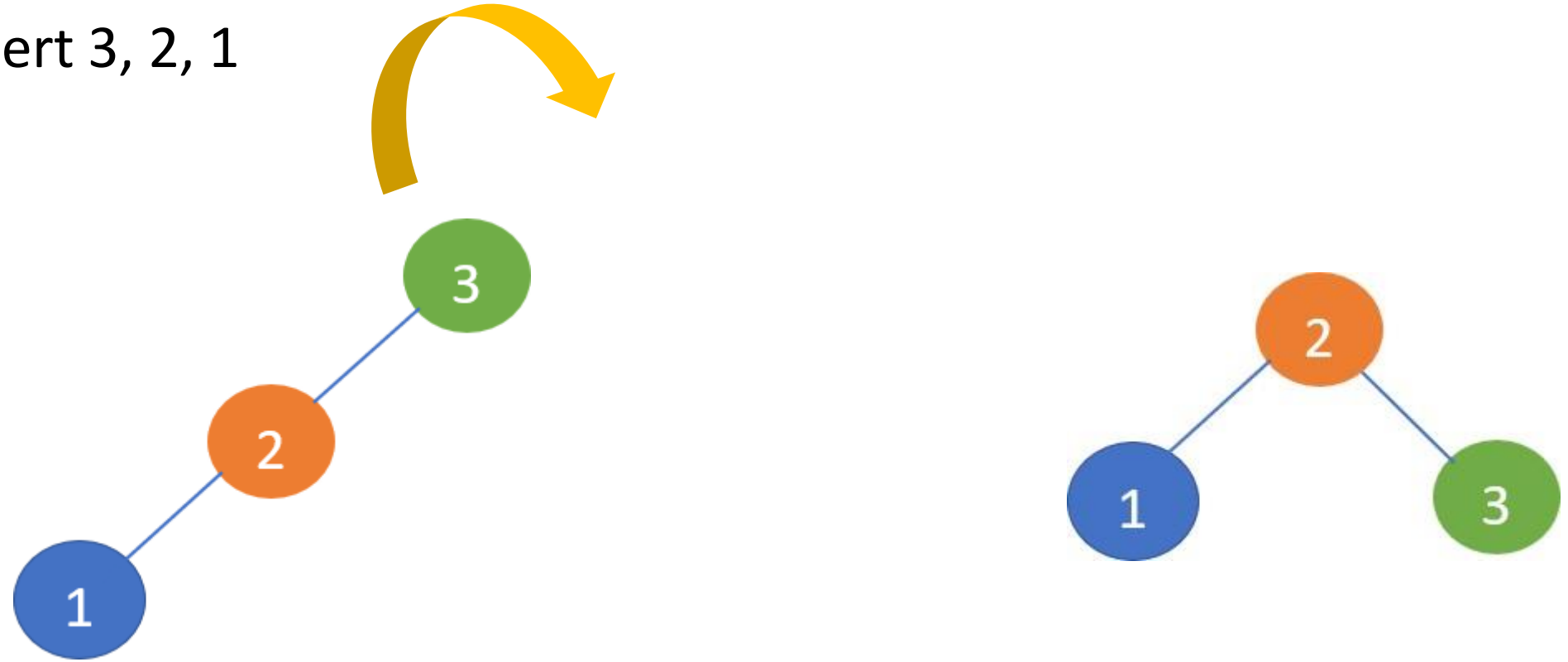
- The 4 kinds of rotation are:
  - **left – left rotation**
    - a single ‘left’ rotation
  - **right – right rotation**
    - a single ‘right’ rotation
  - **left – right rotation**
    - a ‘left’ rotation followed by a ‘right’ rotation
  - **right – left rotation**
    - a ‘right’ rotation followed by a ‘left’ rotation

left - left rotation (right of right)

Insert 1, 2, 3

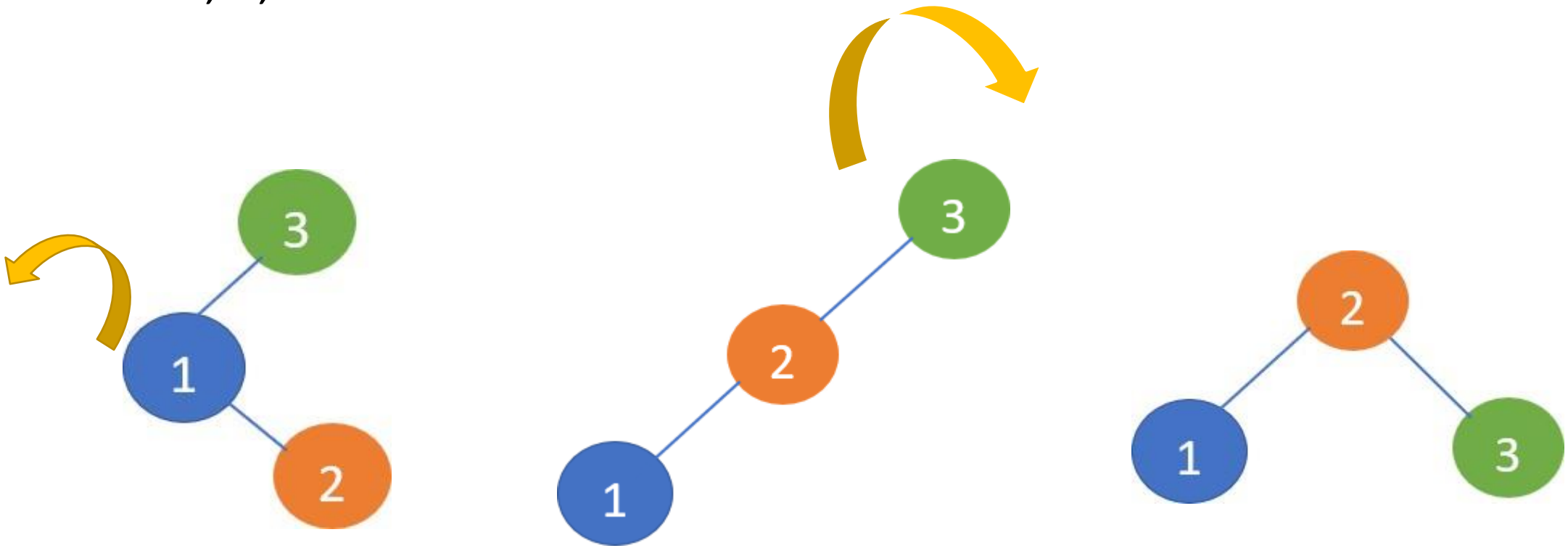
right - right rotation (left of left)

Insert 3, 2, 1



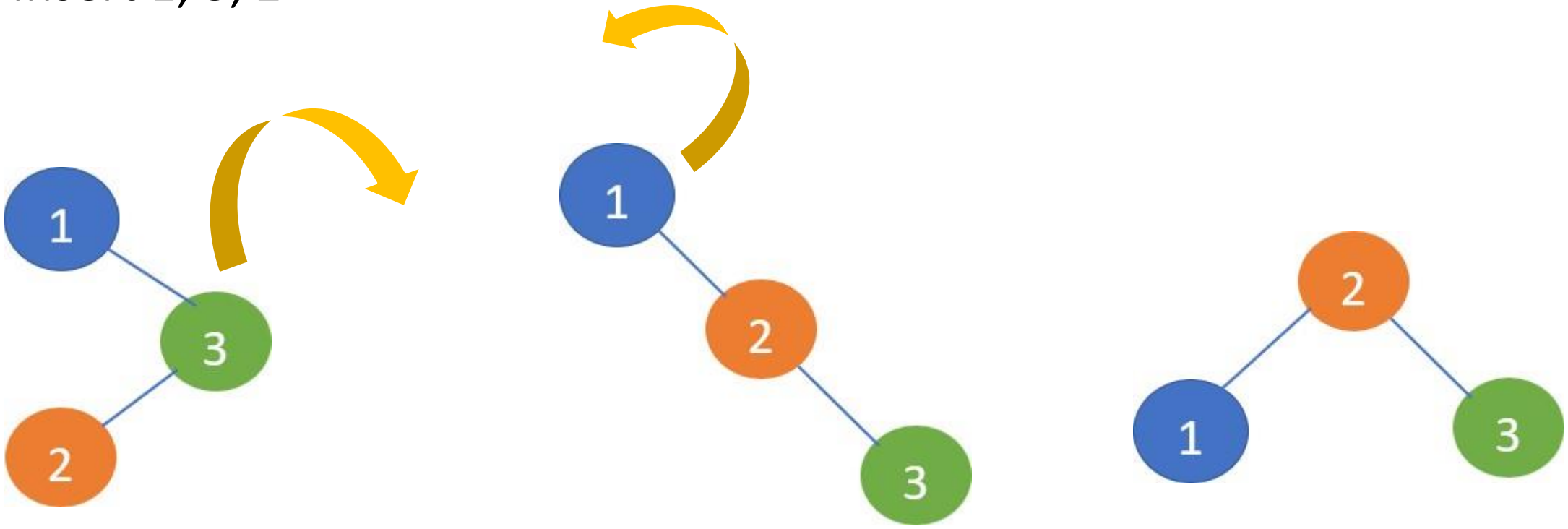
left - right rotation (right of left)

Insert 3, 1, 2

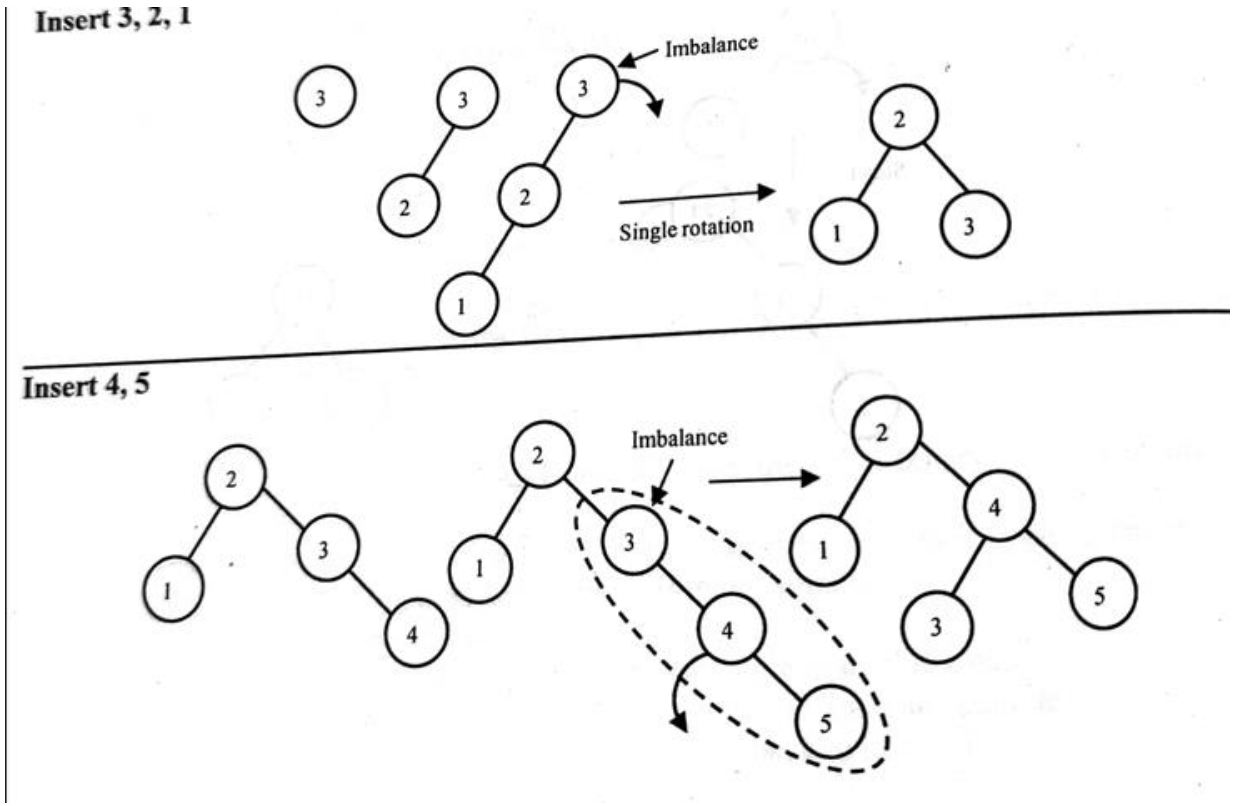


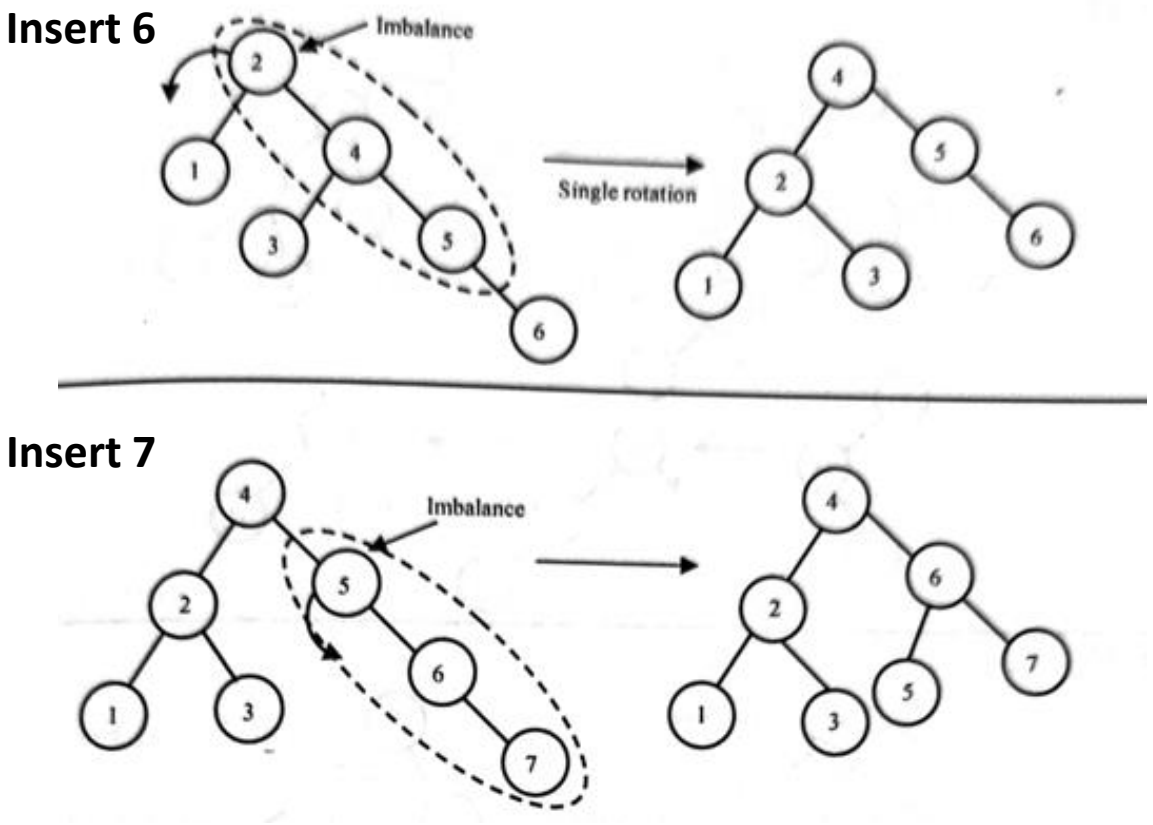
right - left rotation (left of right)

Insert 1, 3, 2

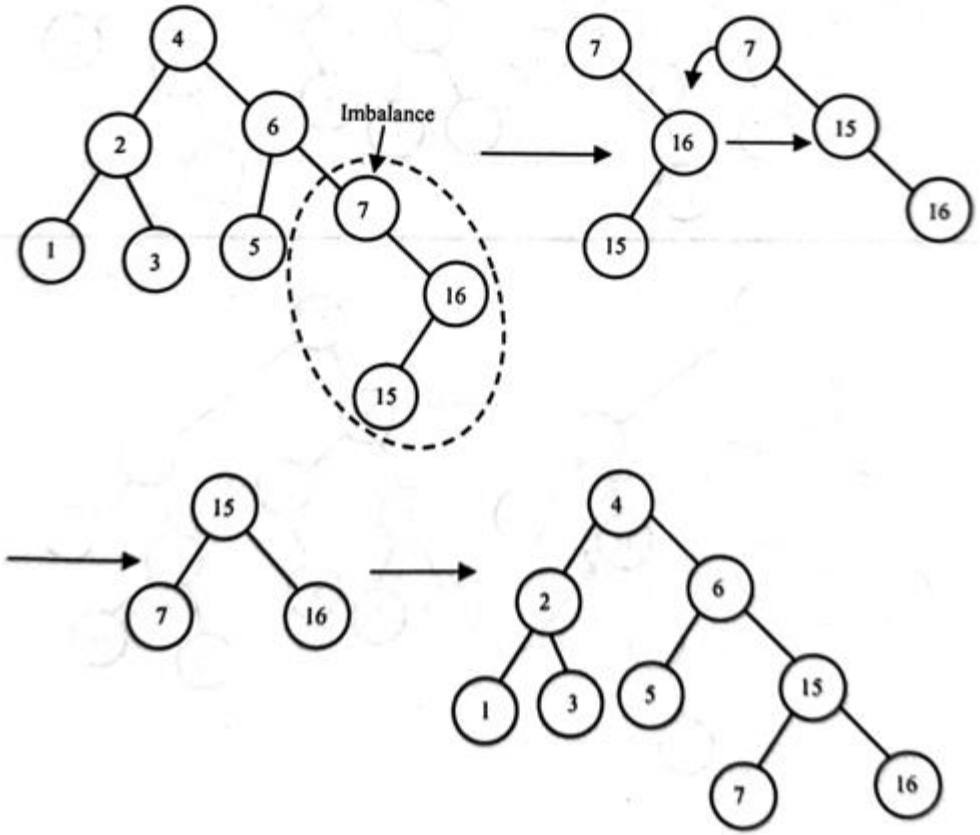


1. Sequentially Insert the given data 3,2,1,4,5,6,7,16,15,14,13,12,11 into AVL tree.

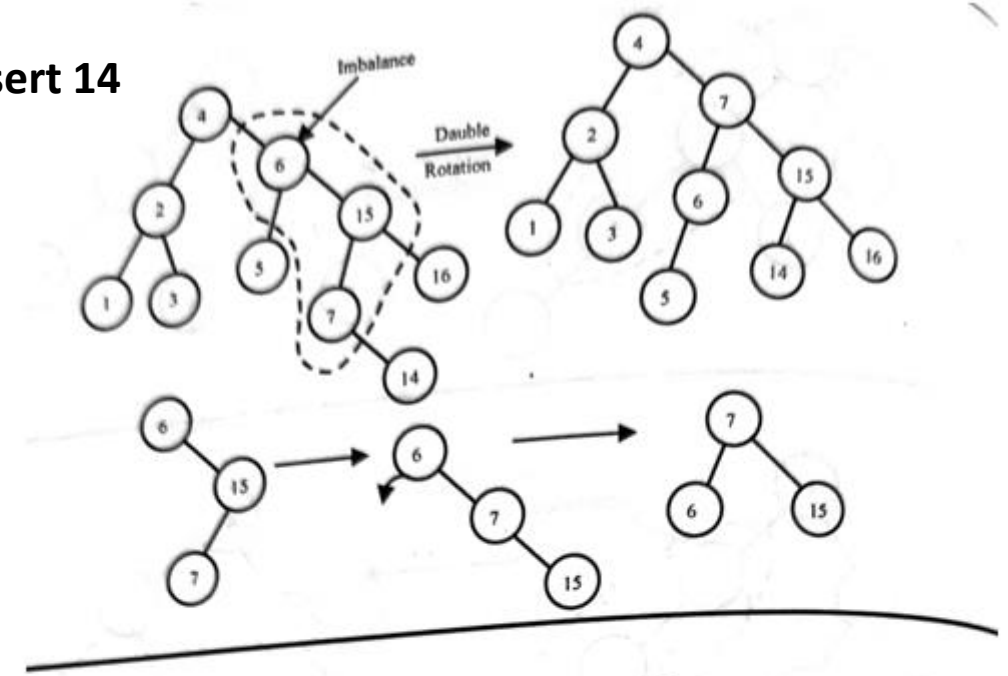




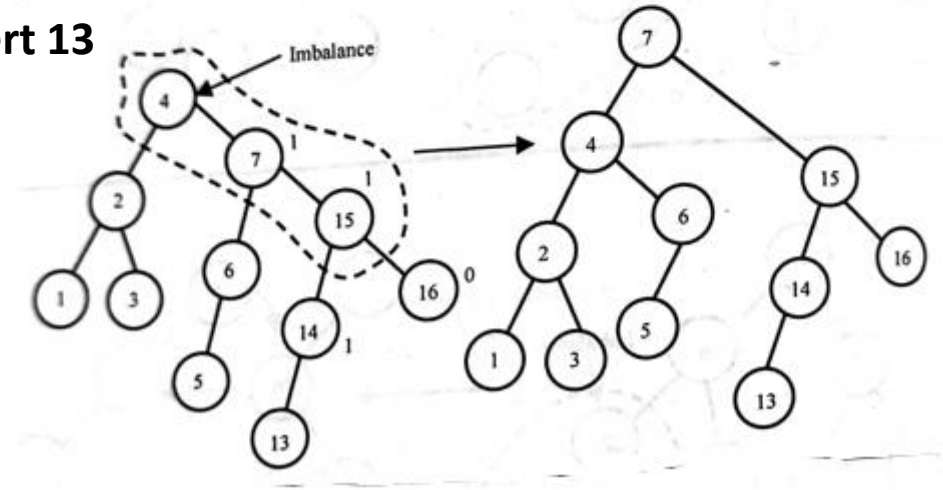
Insert 15 and 16



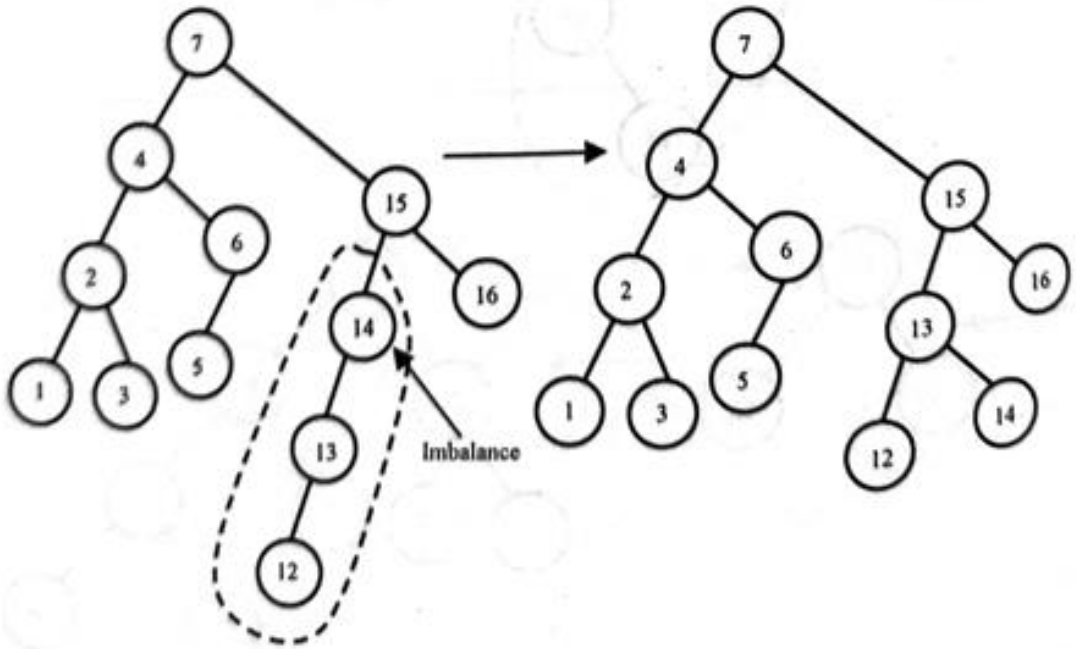
Insert 14



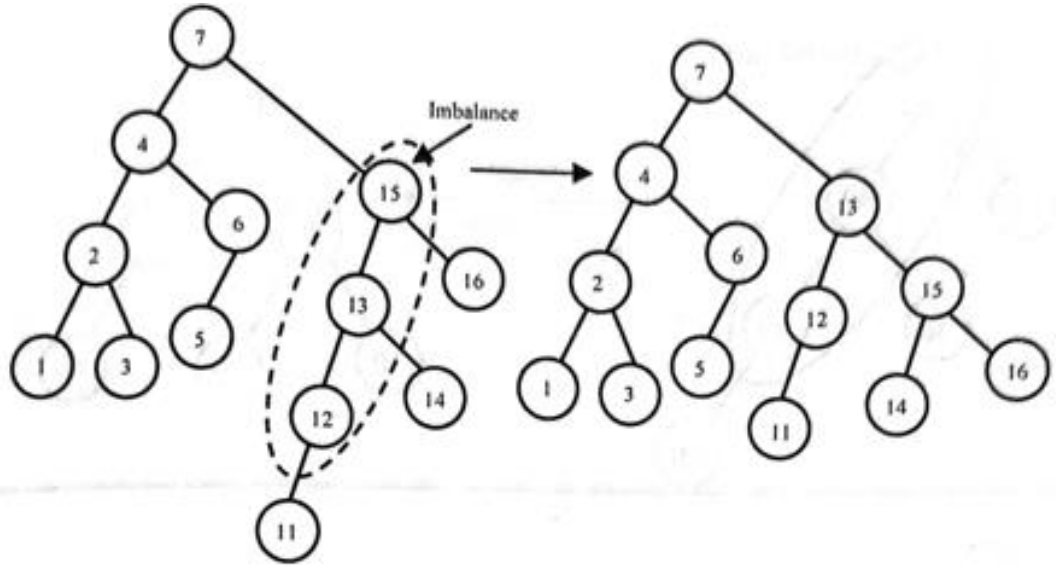
Insert 13



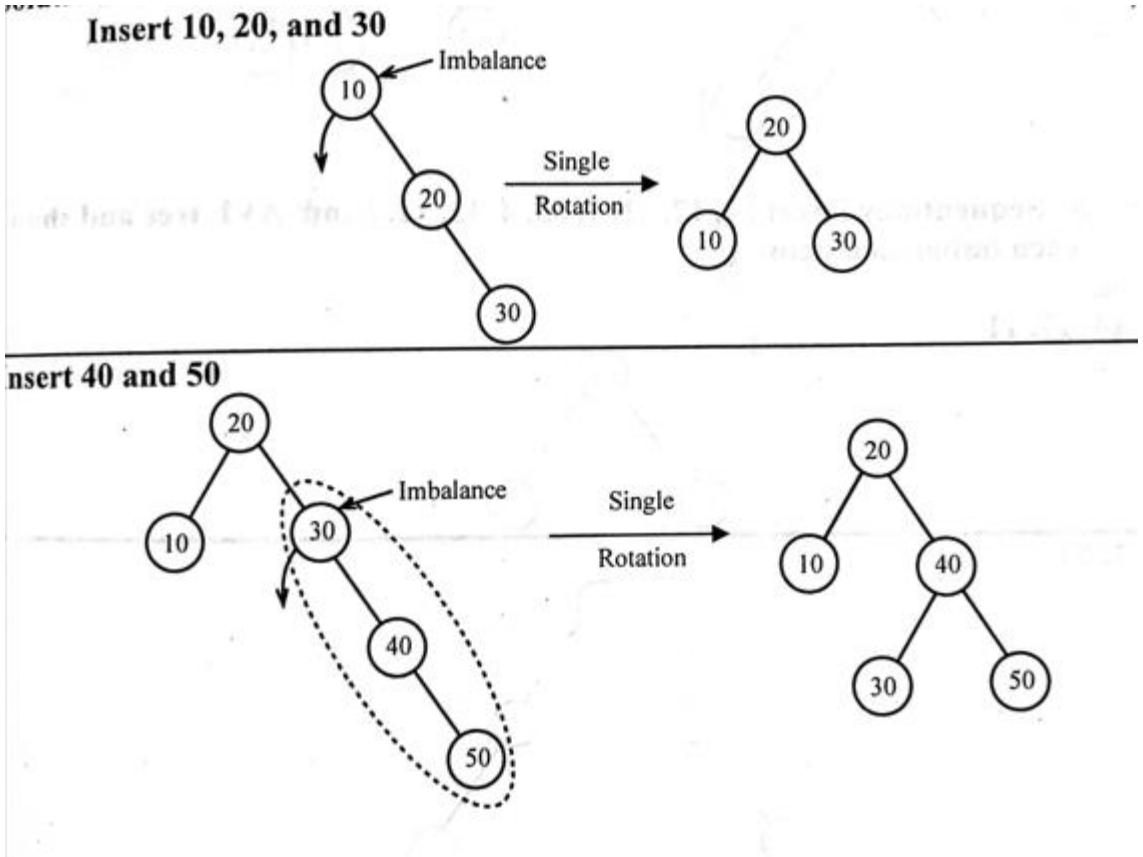
Insert 12

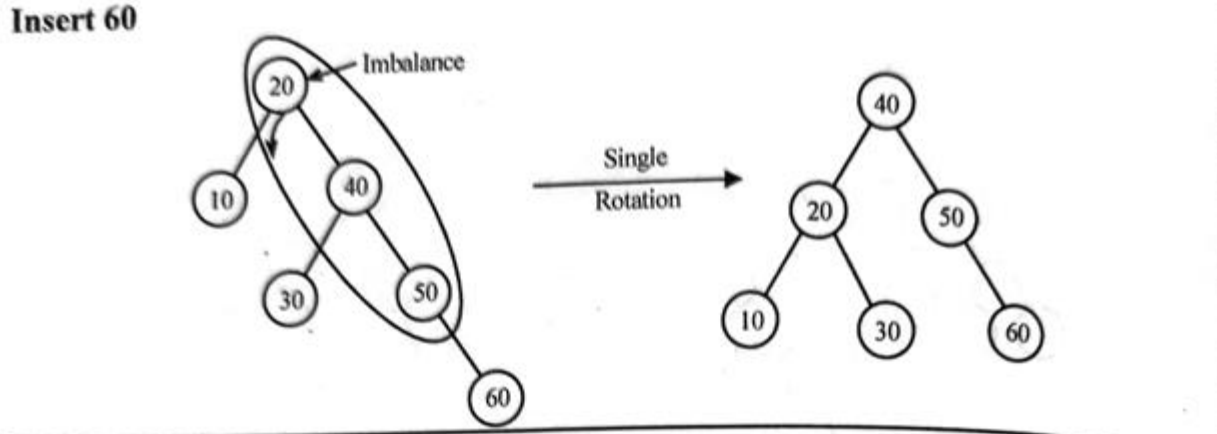


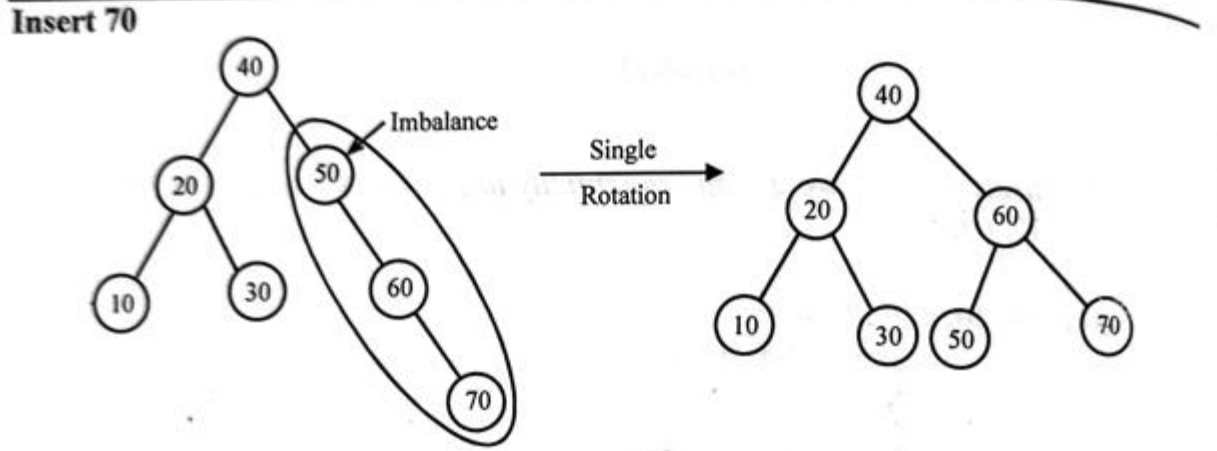
Insert 11



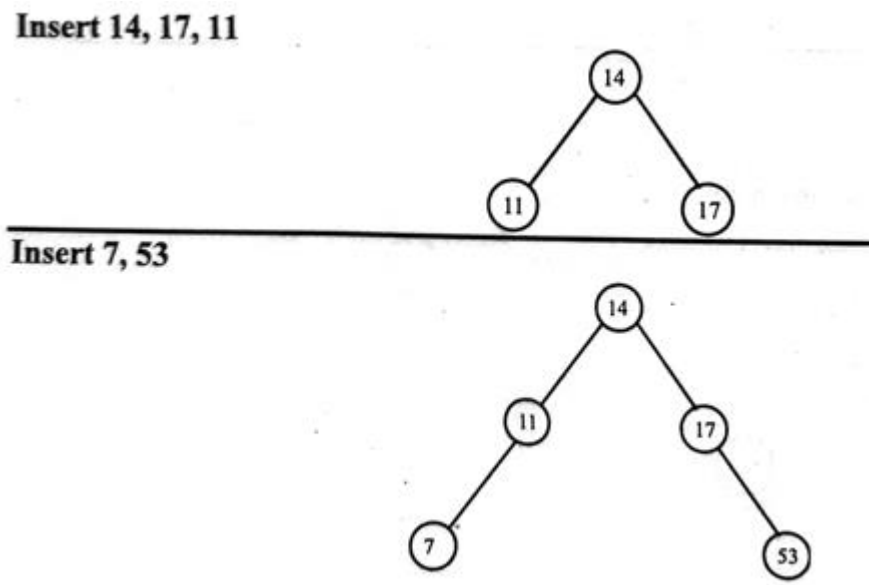
2. Insert the data sequentially to create AVL tree: 10,20,30,40,50,60,70

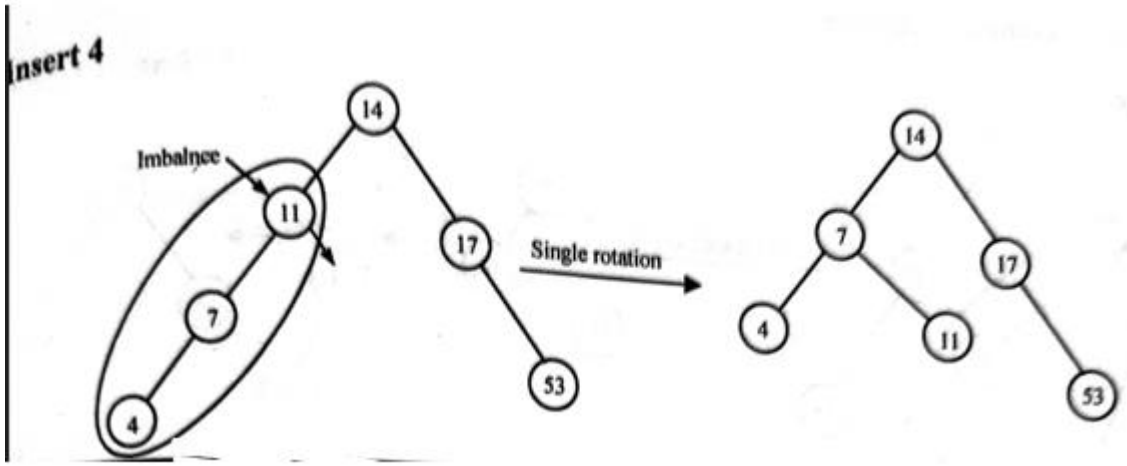


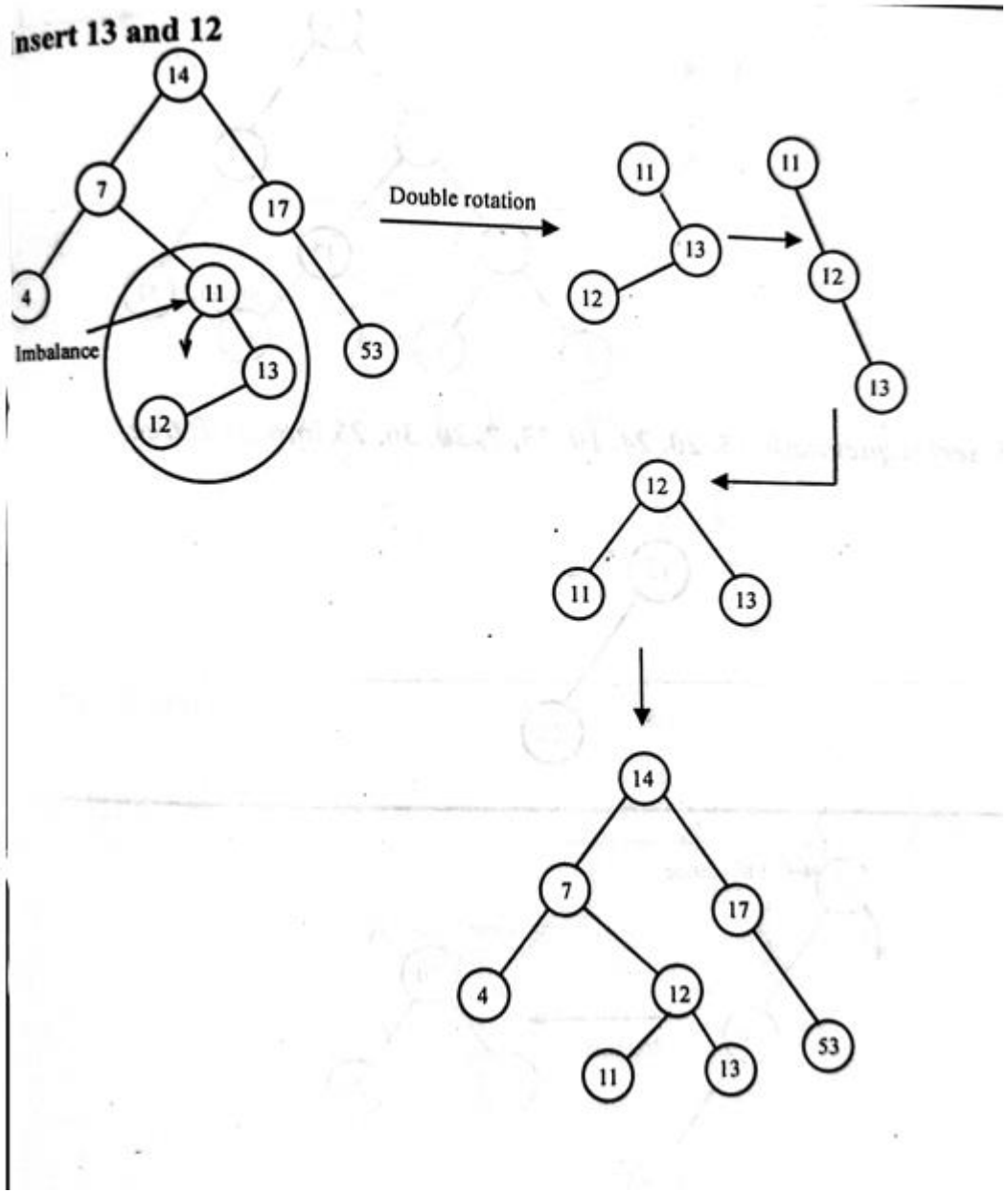




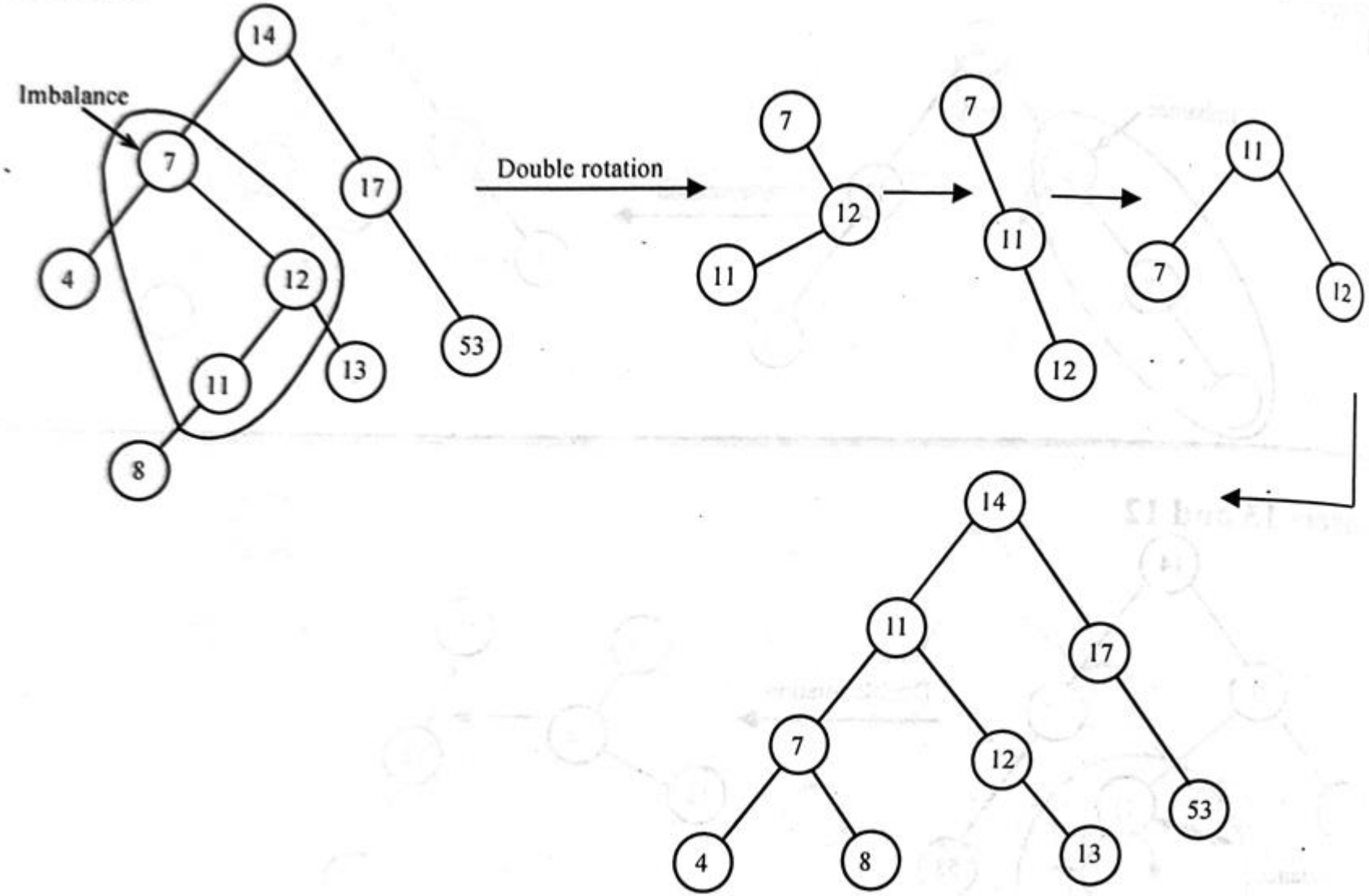
3. Sequentially insert 14,17,11,7,53,4,13,12,8 into AVL tree and show each imbalance steps.



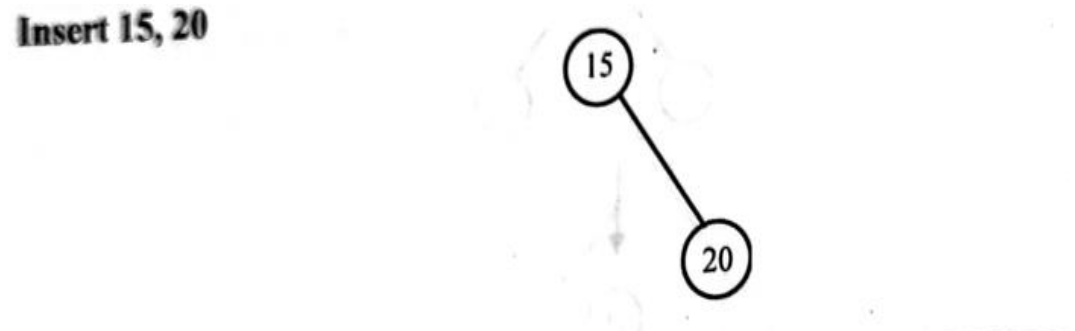


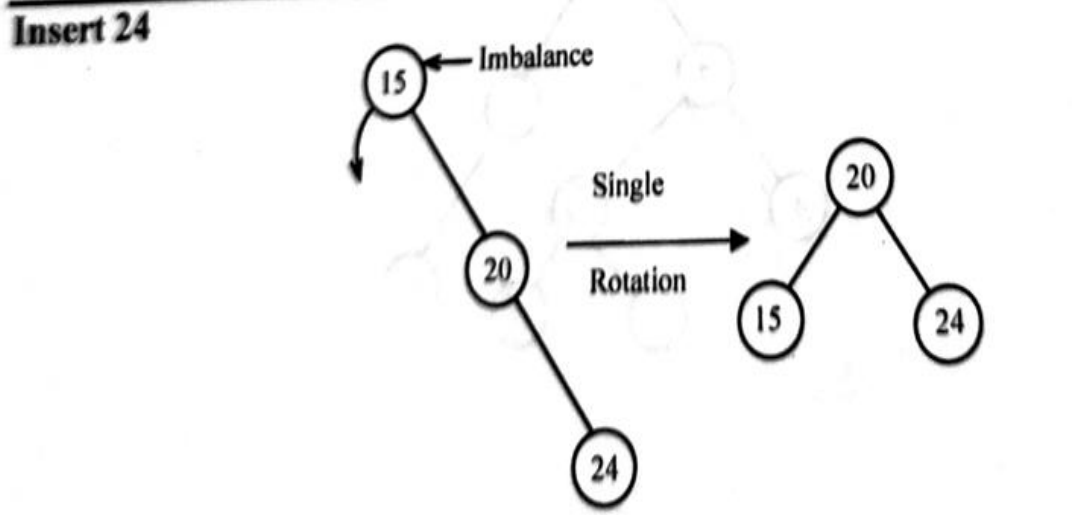


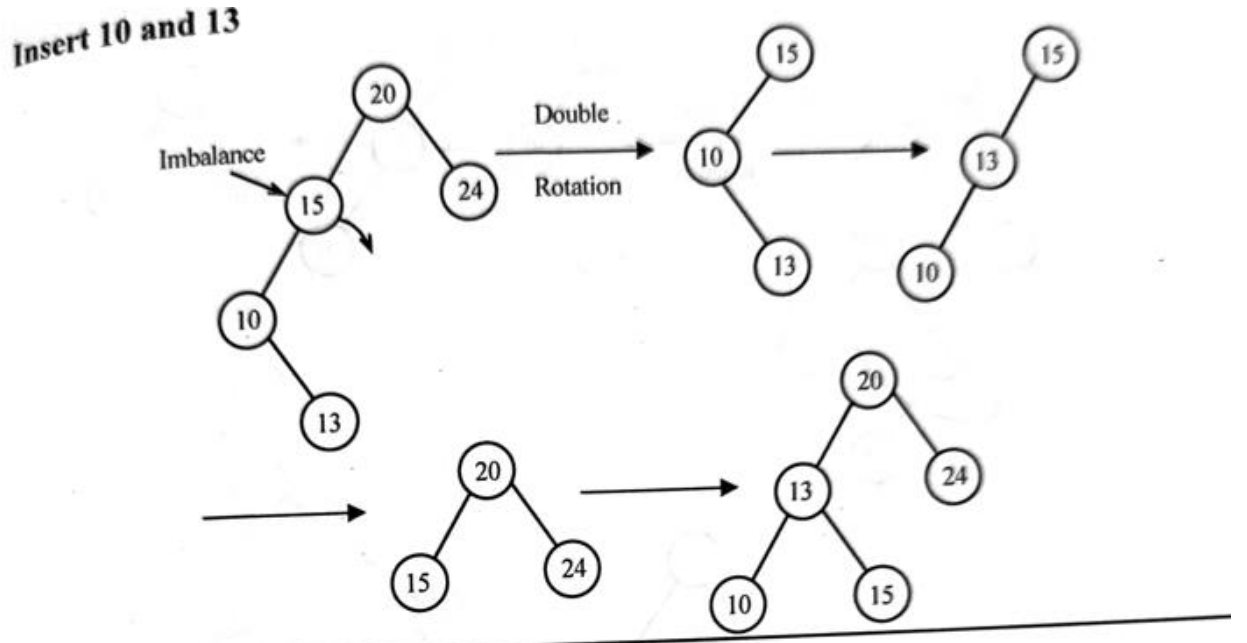
Insert 8

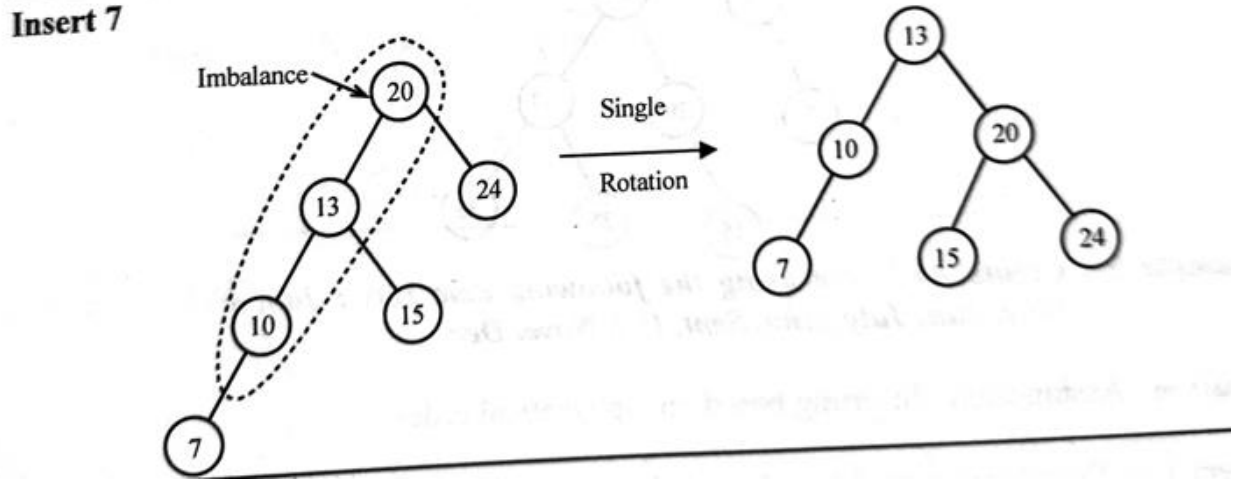

Imbalance
14
7
17
4
12
53
11
13
8
Double rotation
7
12
11
7
11
12
11
7
12
14
11
17
7
12
53
4
8
13

4. Insert sequentially 15,20,24,10,13,7,30,36,25 into AVL tree.

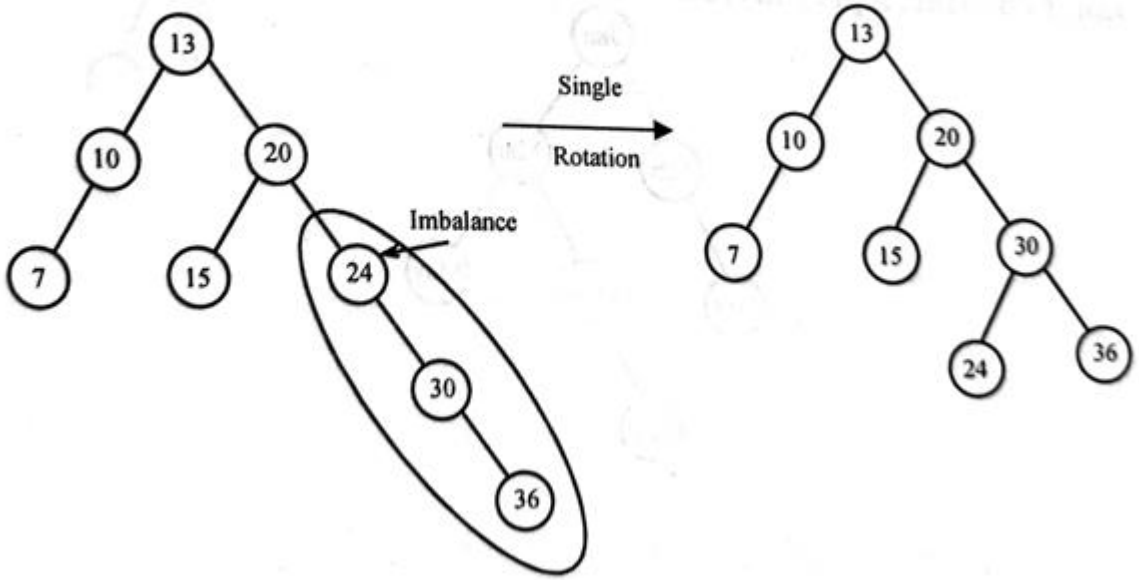




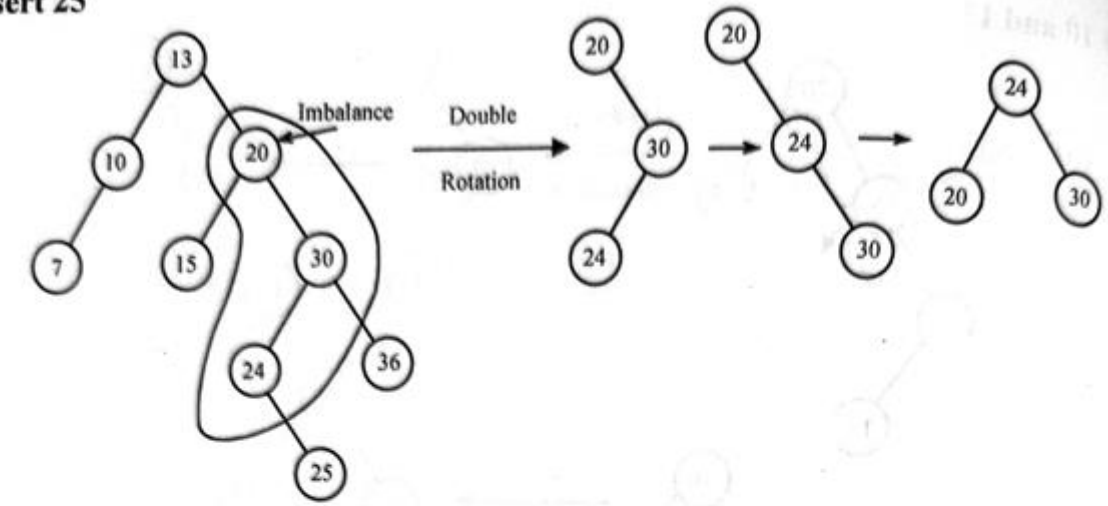


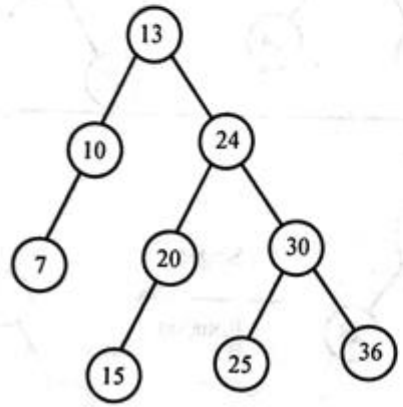


Insert 30 and 36



Insert 25





MCQs Practice

of

Introduction to data structures, list, linked list and trees

Q. In a stack, if a user tries to remove an element from an empty stack it is called _____

A. Underflow

B. Empty collection

C. Overflow

D. Garbage Collection

Q. Pushing an element into stack already having five elements and stack size of 5, then stack becomes _____

A. Overflow

B. Crash

C. Underflow

D. User flow

Q. Circular Queue is also known as _____

A. Ring Buffer

B. Square Buffer

C. Rectangle Buffer

D. Curve Buffer

Q. If the elements “A”, “B”, “C” and “D” are placed in a queue and are deleted one at a time, in what order will they be removed?

A. ABCD

B. DCBA

C. DCAB

D. ABDC

Q. A data structure in which elements can be inserted or deleted at/from both ends but not in the middle is?

A. Queue

B. Circular queue

C. Dequeue

D. Priority queue

Q. Which of the following is not the type of queue?

A. Ordinary queue

B. Single ended queue

C. Circular queue

D. Priority queue

Q. Which of these is not an application of a linked list?

A. To implement file systems

B. For separate chaining in hash-tables

C. To implement non-binary trees

D. Random Access of elements

Q. What differentiates a circular linked list from a normal linked list?

A. You cannot have the ‘next’ pointer point to null in a circular linked list

B. It is faster to traverse the circular linked list

C. You may or may not have the ‘next’ pointer point to null in a circular linked list

D. Head node is known in circular linked list

Q. Which of the following application makes use of a circular linked list?

A. Undo operation in a text editor

B. Recursive function calls

C. Allocating CPU to resources

D. Implement Hash Tables

Q. What does ‘stack overflow’ refer to?

A. accessing item from an undefined stack

B. adding items to a full stack

C. removing items from an empty stack

D. index out of bounds exception

Q. Which of the following data structures can be used for parentheses matching?

A. n-ary tree

B. queue

C. priority queue

D. stack

Q. In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into a NONEMPTY queue?

A. Only front pointer

B. Only rear pointer

C. Both front and rear pointer

D. No pointer will be changed

Q. In linked list implementation of a queue, from where is the item deleted?

A. At the head of link list

B. At the centre position in the link list

C. At the tail of the link list

D. Node before the tail

Q. Out of the following operators (|, *, +, &, $), the one having lowest priority is _____

A. +

B. $

C. |

D. &

Q. Out of the following operators (^, *, +, &, $), the one having highest priority is _____

A. +

B. $

C. ^

D. &

Q. What data structure is used when converting an infix notation to prefix notation?

A. Stack

B. Queue

C. B-Trees

D. Linked-list

Q. How many stacks are required for applying evaluation of infix expression algorithm?

A.one

B. two

C. three

D.four

Q. Evaluation of infix expression is done based on precedence of operators

A. True

B. False

Q. What is the maximum number of children that a binary tree node can have?

A. 0

B. 1

C. 2

D. 3

Q. How many common operations are performed in a binary tree?

A.1

B.2

C.3

D.4

Q. How many orders of traversal are applicable to a binary tree (In General)?

A.1

B.4

C.2

D.3

Q. To obtain a prefix expression, which of the tree traversals is used?

A. Level-order traversal

b) Pre-order traversal

C. Post-order traversal

D. In-order traversal

Q. Consider the following data. The pre order traversal of a binary tree is A, B, E, C, D. The in order traversal of the same binary tree is B, E, A, D, C. The level order sequence for the binary tree is _____

A. A, C, D, B, E

B. b) A, B, C, D, E

C. c) A, B, C, E, D

D. D, B, E, A, C

Q. The number of edges from the root to the node is called _____ of the tree

A. Height

B. Depth

C. Length

D. Width

Q. What is a full binary tree?

A. Each node has exactly zero or two children

B. Each node has exactly two children

C. All the leaves are at the same level

D. Each node has exactly one or two children

Q. The number of edges from the node to the deepest leaf is called _____ of the tree.

A. Height

B. Depth

C. length

D.width

Q. Given an empty AVL tree, how would you construct AVL tree when a set of numbers are given without performing any rotations?

A. just build the tree with the given input

B. find the median of the set of elements given, make it as root and construct the tree

C. use trial and error

D.use dynamic programming to build the tree