Lecture -1

What is a Data structure?

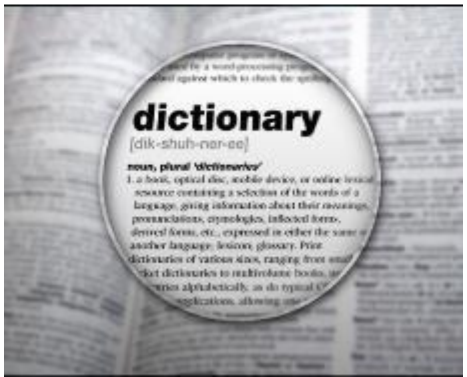
Data structure is one of the most **fundamentals** subject in Computer Science & in-depth understanding of this topic is very important especially when you are into **development/programming** domain where you **build efficient software systems & applications.** **Definition-**
In computer science, a data structure is a **data organization**, **management** and **storage** format that enables **efficient access** and **modification**.
In Simple words-
Data Structure is a **way** in which data is stored on a computer.

Data Structure is a way to **store** and organize data so that it can be used efficiently.

Our Data Structure tutorial includes all topics of Data Structure such as Array, Pointer, Structure, Linked List, Stack, Queue, etc.

Why do we need Data structures?

- Each Data Structure allows data to be stored in specific manner.
- Data Structure allow efficient data search and retrieval.
- Specific data structures are decided to work for specific problems.
- It allows to manage large amount of data.



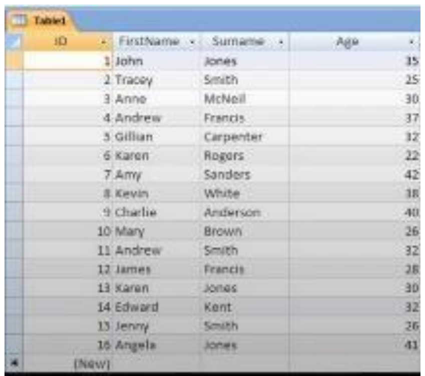
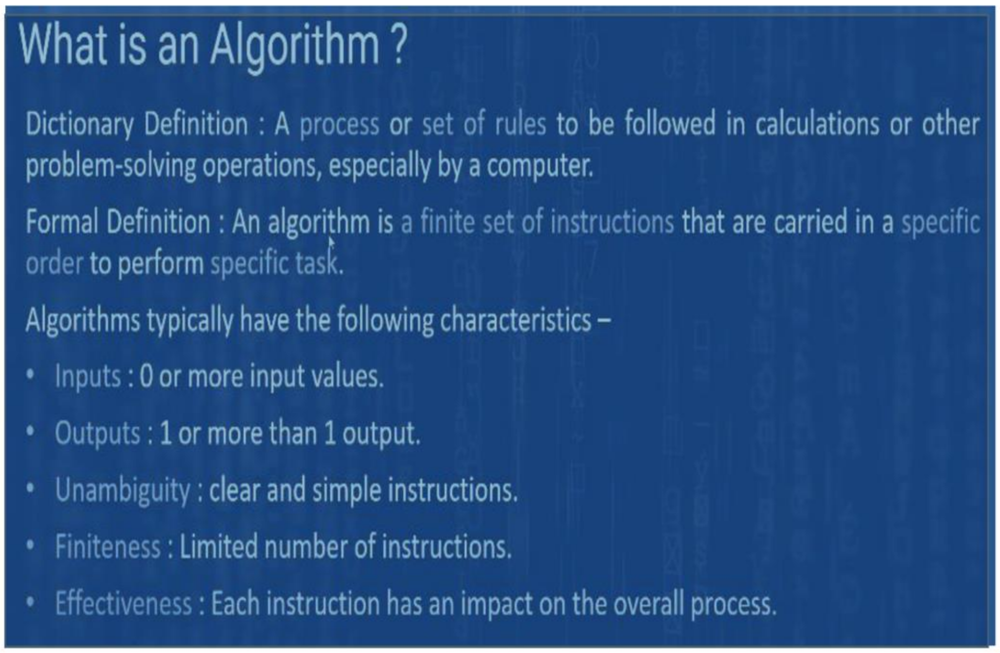


Table1

ID	FirstName	Surname	Age
1	John	Jones	35
2	Tracey	Smith	25
3	Anne	McNeil	30
4	Andrew	Francis	37
5	Gillian	Carpenter	32
6	Karen	Rogers	22
7	Amy	Sanders	42
8	Kevin	White	38
9	Charlie	Anderson	40
10	Mary	Brown	26
11	Andrew	Smith	32
12	James	Francis	28
13	Karen	Jones	30
14	Edward	Kent	32
15	Jenny	Smith	26
16	Angela	Jones	41
(New)			

Note: The data structure is not any programming language like C, C++, java, etc. It is a set of algorithms that can be use in any programming language to structure the data in the memory.

Algorithms



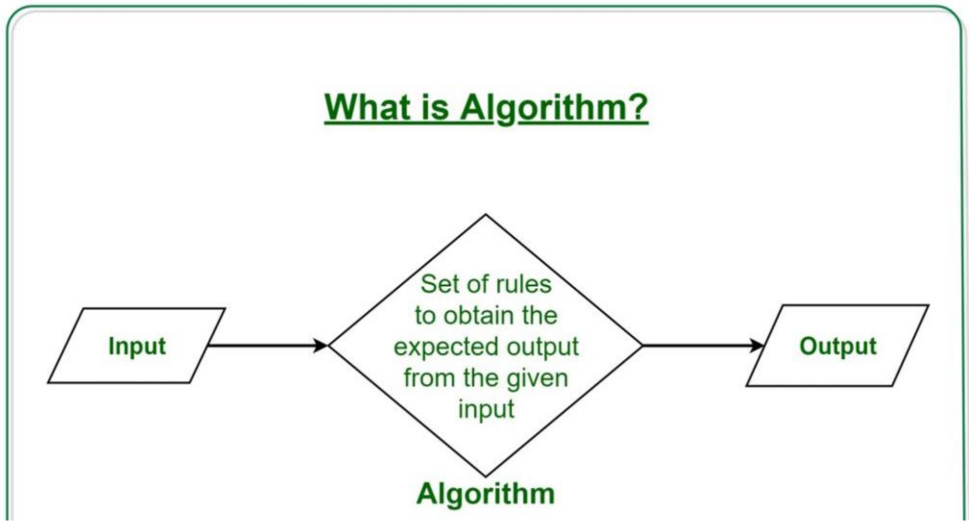
What is an Algorithm ?

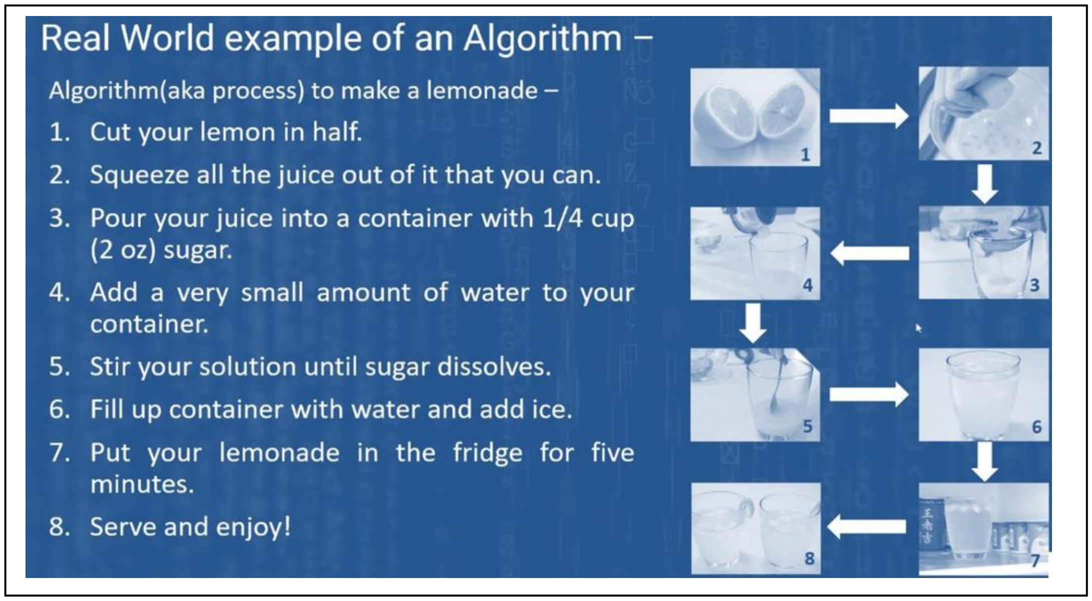
Dictionary Definition : A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

Formal Definition : An algorithm is a finite set of instructions that are carried in a specific order to perform specific task.

Algorithms typically have the following characteristics –

- Inputs : 0 or more input values.
- Outputs : 1 or more than 1 output.
- Unambiguity : clear and simple instructions.
- Finiteness : Limited number of instructions.
- Effectiveness : Each instruction has an impact on the overall process.

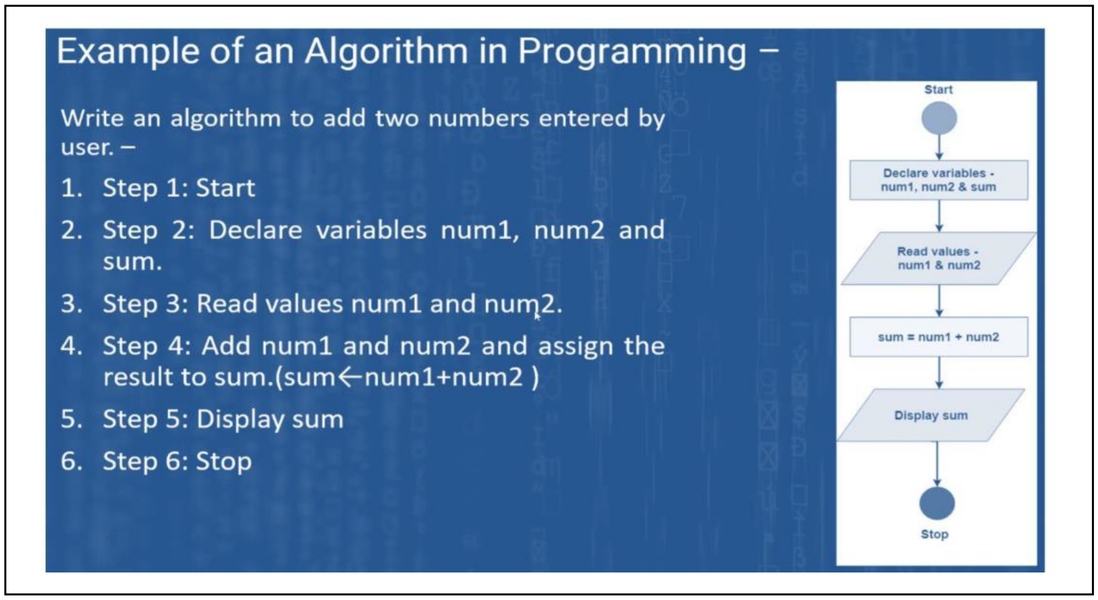

What is Algorithm?
Input
Set of rules to obtain the expected output from the given input
Output
Algorithm



Real World example of an Algorithm –

Algorithm(aka process) to make a lemonade –

1. Cut your lemon in half.
2. Squeeze all the juice out of it that you can.
3. Pour your juice into a container with 1/4 cup (2 oz) sugar.
4. Add a very small amount of water to your container.
5. Stir your solution until sugar dissolves.
6. Fill up container with water and add ice.
7. Put your lemonade in the fridge for five minutes.
8. Serve and enjoy!



Example of an Algorithm in Programming –

Write an algorithm to add two numbers entered by user. –

1. Step 1: Start
2. Step 2: Declare variables num1, num2 and sum.
3. Step 3: Read values num1 and num2.
4. Step 4: Add num1 and num2 and assign the result to sum.(sum←num1+num2)
5. Step 5: Display sum
6. Step 6: Stop

Start
Declare variables - num1, num2 & sum
Read values - num1 & num2
sum = num1 + num2
Display sum
Stop

program = Algorithm + Data Structure

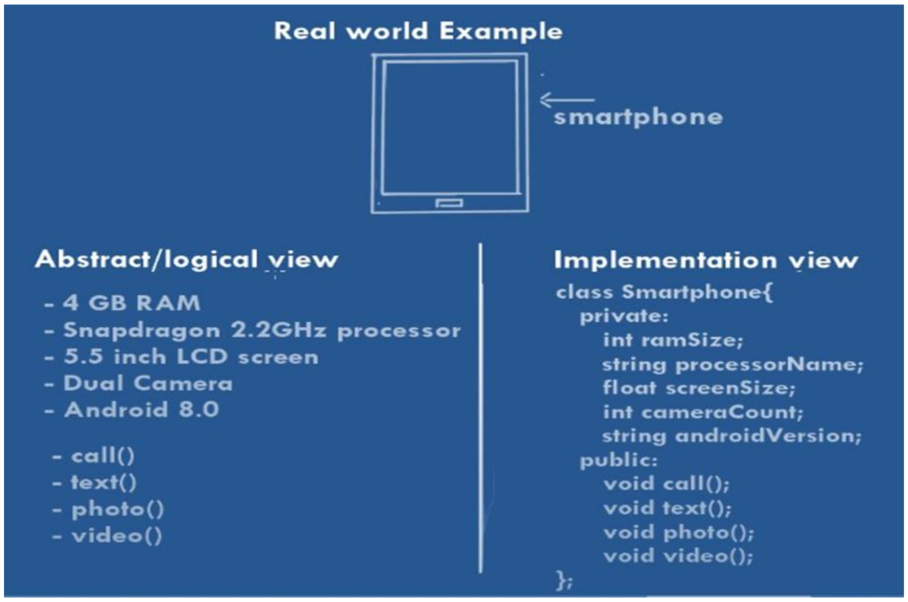
What is Abstract Data Type?

Definition: ADTs are entities that are definitions of data and operations but do not have implementation details.

Two ways of looking at Data Structures-

- Mathematical/Logical/Abstract Models/Views

- Implementation



Real world Example

smartphone

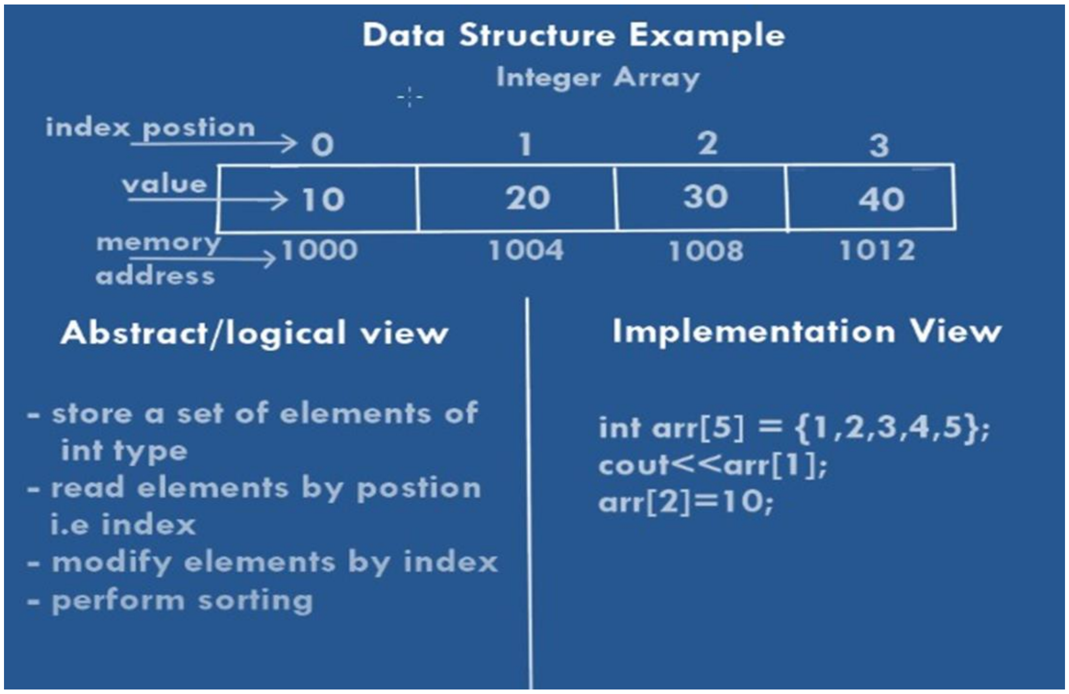
Abstract/logical view

- 4 GB RAM
- Snapdragon 2.2GHz processor
- 5.5 inch LCD screen
- Dual Camera
- Android 8.0

- call()
- text()
- photo()
- video()

Implementation view

```
class Smartphone{
  private:
    int ramSize;
    string processorName;
    float screenSize;
    int cameraCount;
    string androidVersion;
  public:
    void call();
    void text();
    void photo();
    void video();
};
```



Data Structure Example

Integer Array

Abstract/logical view

- store a set of elements of int type
- read elements by postion i.e index
- modify elements by index
- perform sorting

Implementation View

```
int arr[5] = {1,2,3,4,5};
cout<<arr[1];
arr[2]=10;
```

Use of the Algorithms:

Algorithms play a crucial role in various fields and have many applications. Some of the key areas where algorithms are used include:

1. **Computer Science:** Algorithms form the basis of computer programming and are used to solve problems ranging from simple sorting and searching to complex tasks such as artificial intelligence and machine learning.
2. **Mathematics:** Algorithms are used to solve mathematical problems, such as finding the optimal solution to a system of linear equations or finding the shortest path in a graph.
3. **Operations Research**: Algorithms are used to optimize and make decisions in fields such as transportation, logistics, and resource allocation.
4. **Artificial Intelligence:** Algorithms are the foundation of artificial intelligence and machine learning, and are used to develop intelligent systems that can perform tasks such as image recognition, natural language processing, and decision-making.
5. **Data Science:** Algorithms are used to analyze, process, and extract insights from large amounts of data in fields such as marketing, finance, and healthcare.

Representation of algorithm can written By:-

In natural language (English) / pseudo-code / diagrams (Flow chart) / etc.

Pseudo- code:-

A mixture of natural language and high – level programming concepts that describes the main ideas behind a generic implementation of a data structure or algorithm**. Pseudo- code** is more structured than usual language but less formal than a programming language.

Ex.:- Algorithm to find the maximum number in array input: An array

```
with n integers
output: The Maximum element in A
currentMax ← A[0]
for i ← 1 to n-1 do
if currentMax < A[i] then
currentMax ← A[i]return
currentMax
```

Ex: An algorithm to find sum n numbers for N range

```
1- Start
2- Read N
3- Sum =0
4- For I= 1 to N
 sum = sum + I
   next I
5- print  Sum
6- End
```

What Makes a Good Algorithm?

Suppose you have two possible algorithms or data structures that basically do the same thing; which is better?

- Faster
- Less space
- Easier to code
- Easier to maintain

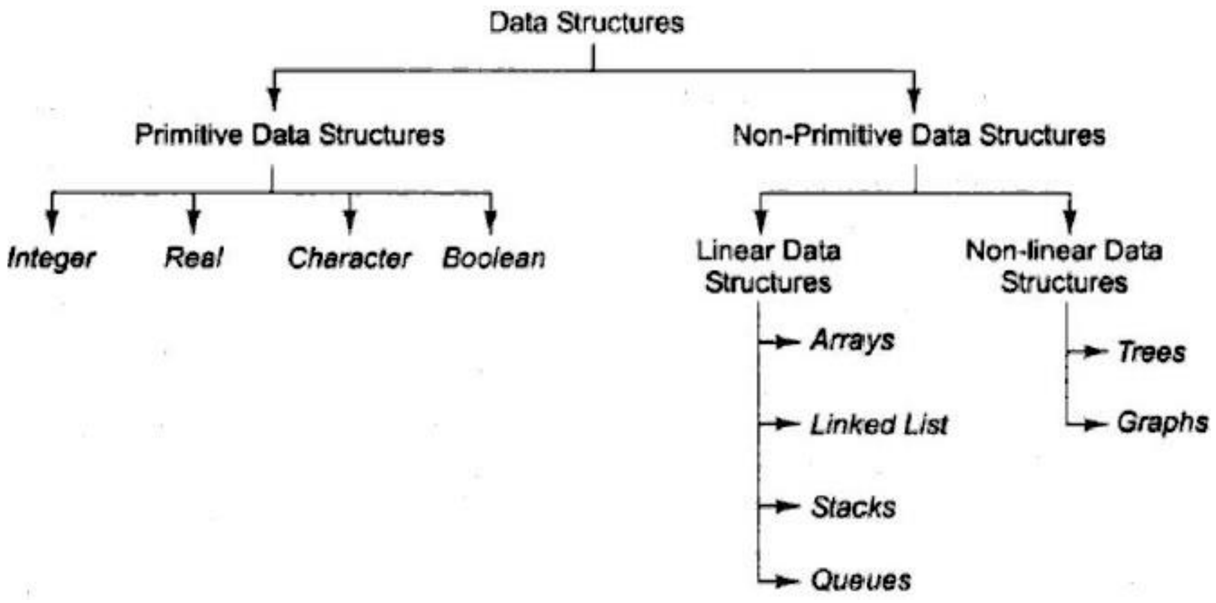


Fig.1 Classifications of data structures

Classification of data structure

Data structures are broadly divided into two:

1. Primitive data structures: These are the basic data structures and are directly operated upon by the machine instructions, which is in a primitive level. They are integers, floating point numbers, characters, string constants, pointers etc. These primitive data structures are the basis for the discussion of more sophisticated (non-primitive) data structures.
2. Non-primitive data structures: It is a more sophisticated data structure emphasizing on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items. Array, list, files, linked list, trees and graphs fall in this category.
3.

How to choose the suitable data structure:-

For each set of data, there are different methods to organize these data in a particular data structure. To choose the suitable data structure, we must use the following criteria:-

1- Data size and the required memory.

2- The dynamic nature of the data.

3- The required time to obtain any data element from the data structure.

4- The programming approach and the algorithm that will be used to manipulate these data.

Assignment -1-

- **Write an algorithm for the following**

  a- Print the even& odd numbers in given range

Array

Array is a linear data structure that stores a collection of elements of the same data type. Elements are allocated contiguous memory, allowing for constant-time access. Each element has a unique index number.

Array is a collection of items of the same variable type that are stored at contiguous memory locations. It is one of the most popular and simple data structures used in programming

Operations on Array:

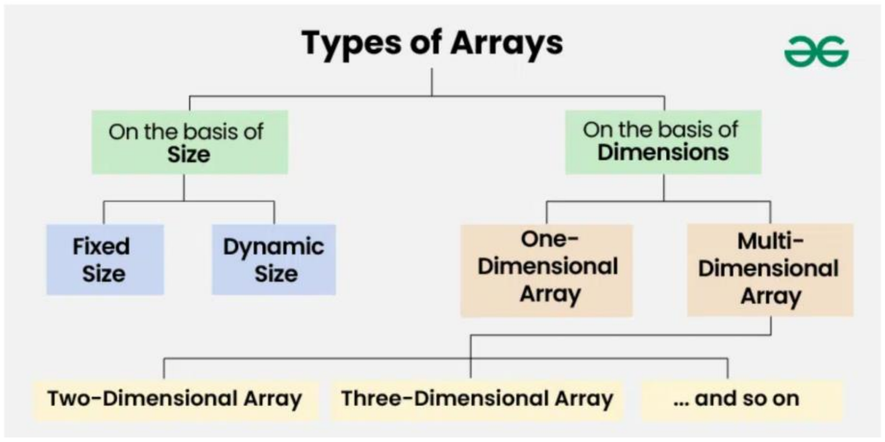
- **Traversal**: Iterating through the elements of an array.
- **Insertion**: Adding an element to the array at a specific index.
- **Deletion**: Removing an element from the array at a specific index.
- **Searching**: Finding an element in the array by its value or index.

- **Types of Arrays**:
  - **One-dimensional array**: A simple array with a single dimension.
  - **Multidimensional array**: An array with multiple dimensions, such as a matrix.
- **Applications of Array**:
  - Storing data in a sequential manner
  - Implementing queues, stacks, and other data structures
  - Representing matrices and tables

Types of Arrays

There are majorly 4 types of arrays

1. Fixed Size Array
2. Dynamic Sized Array
3. 1-Dimensional Array
4. Multi-Dimensional Array

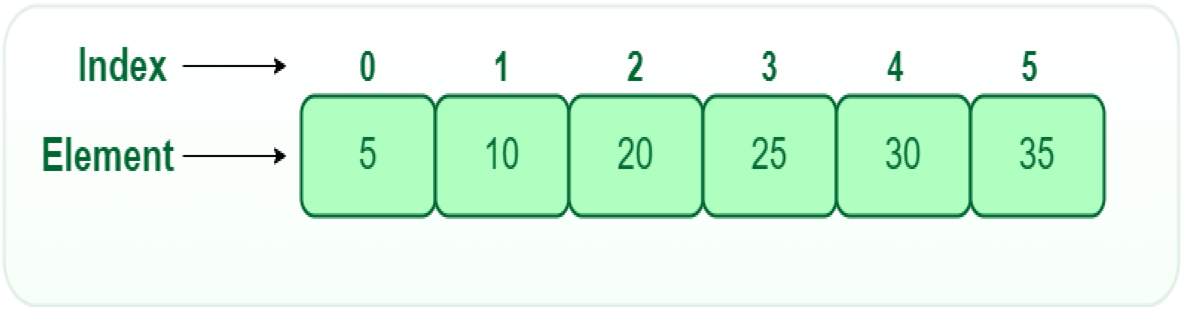


Types of Arrays on the basis of Dimensions

There are majorly three types of arrays on the basis of dimensions:

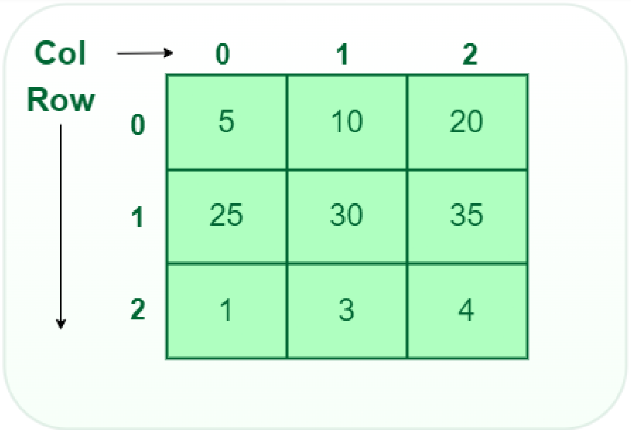
1. One-dimensional array (1-D arrays):

You can imagine a 1d array as a row, where elements are stored one after another.



Two-dimensional (2D) array:

Multidimensional arrays can be considered as an array of arrays or as a matrix consisting of rows and columns.



Advantages of Array

- Arrays allow **random access** to elements. This makes accessing elements by position faster.
- Arrays have **better cache locality** which makes a pretty big difference in performance.
- Arrays **represent multiple data items of the same type** using a single name.
- Arrays are used to implement the other data structures like linked lists, stacks, queues, trees, graphs, etc.

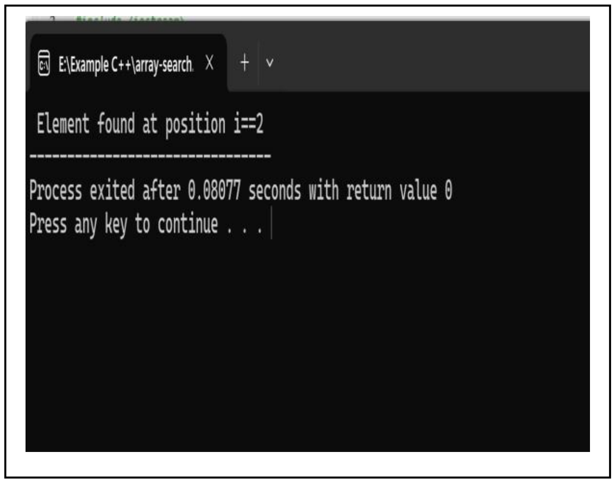
Disadvantages of Array

- As arrays have a fixed size, once the memory is allocated to them, it cannot be increased or decreased, making it impossible to store extra data if required. An array of fixed size is referred to as a static array.
- Allocating less memory than required to an array leads to loss of data.
- An array is homogeneous in nature so, a single array cannot store values of different data types.
- Arrays store data in contiguous memory locations, which makes deletion and insertion very difficult to implement. This problem is overcome by implementing linked lists, which allow elements to be accessed sequentially.

```
// search in unsorted array
#include <iostream>
using namespace std;

// Function to implement search operation
int findElement(int arr[], int n, int key)
{
    int i;
    for (i = 0; i <= n; i++)
        if (arr[i] == key)
            return i;


}
int main()
{
    int arr[] = { 12, 34, 10, 6, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    // Using a last element as search element
    int key = 10;
        // Function call
    int position = findElement(arr, n, key);
       if (position <=n)
      cout <<" Element found at position i=="<<position;
    else
        cout << "Element not Found ";
          return 0;
}
```



```
 Element found at position i==2
--------------------------------
Process exited after 0.08077 seconds with return value 0
Press any key to continue . . .
```

** Find min, max in an array

```
#include <algorithm>
using namespace std;
int main() {
      // Input array
    int a[] = { 1, 423, 6, 46, 34, 23, 13, 53, 4 };
    int n = sizeof(a) / sizeof(a[0]);
      sort(a, a + n);
  for(int i=0;i<=n;i++)
  cout<<a[i]<<"  ";
  cout<<endl;
    cout << "min=  " << a[0] << " max=  " << a[n - 1] << endl;
      return 0;
}
```

- Note: **sort()** is an STL function that is used to sort the given range in desired order. It provides a simple and efficient way to sort the data in C++ but it only works on data structure which have random access to its elements such as vectors and arrays. It is defined inside <algorithm> header file.

- **The sizeof operator yields the size (in bytes) of its operand**, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is an integer. If the type of the operand is a variable length array type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant.

Array Operations

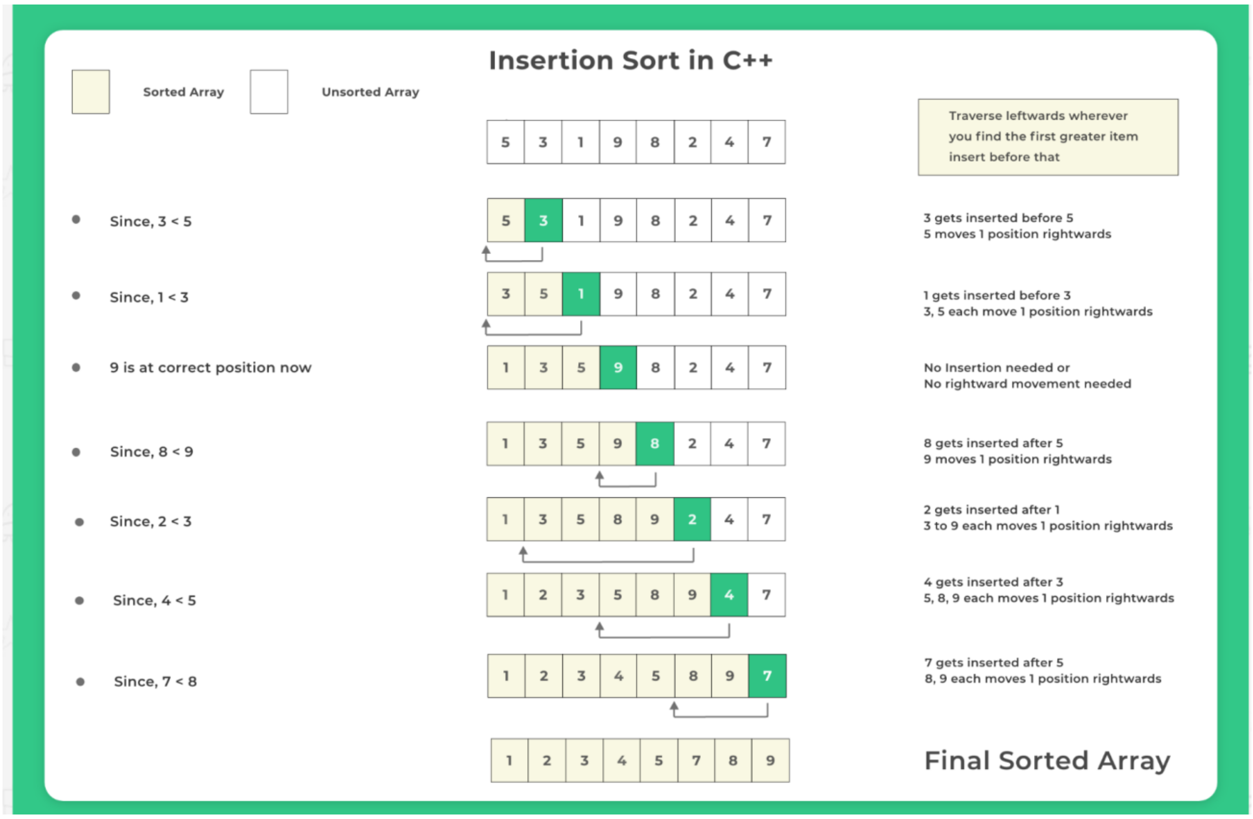
Inserting Elements in an Array

Searching Elements in an Array

Deleting Elements in an Array

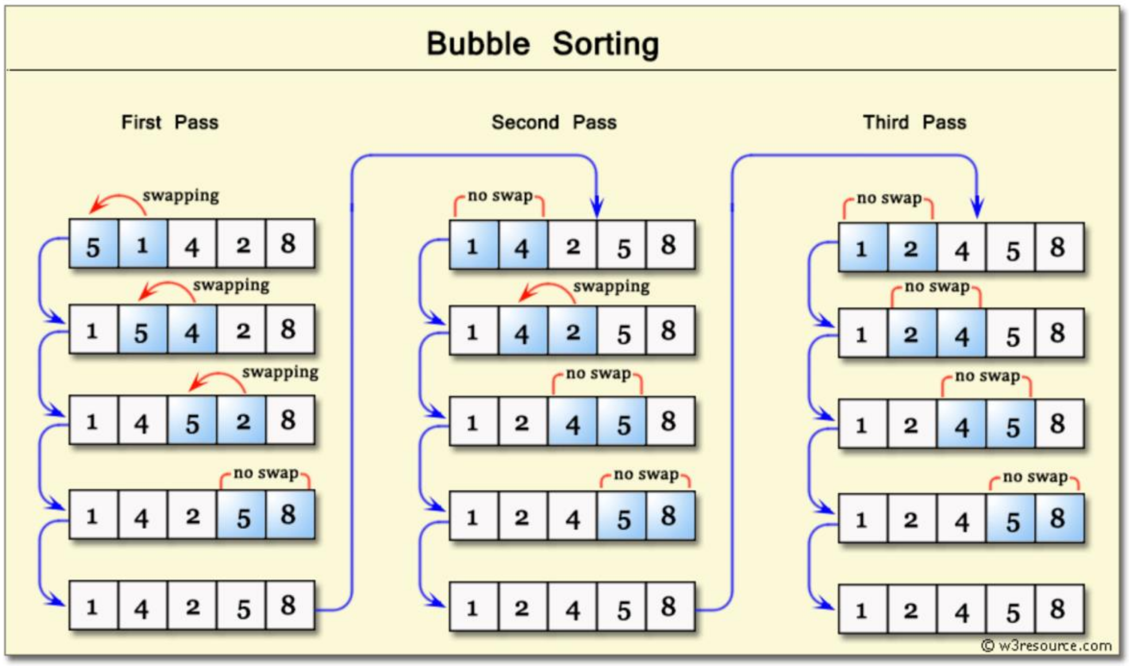
Insertion Sort in C++

If there are n element and it requires (n-1) pass to sort them then, at each pass we insert current element in the appropriate position so that the element are in order. Some characteristics of insertion sort in C++.



In Bubble Sort algorithm

- traverse from left and compare adjacent elements and the higher one is placed at right side.
- In this way, the largest element is moved to the rightmost end at first.
- This process is then continued to find the second largest and place it and so on until the data is sorted.



```
#include <iostream>
using namespace std;
void bubbleSort(int arr[], int n)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false)
        break;
    }
}
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << " " << arr[i];
}
int main()
{
    int arr[] = { 64, 34, 25, 12, 22, 11, 90 };
    int N = sizeof(arr) / sizeof(arr[0]);
    for (int i = 0; i < N - 1; i++)
    cout<<arr[i]<<" ";
    cout<<endl;
    bubbleSort(arr, N);
    cout << "Sorted array: \n";
    printArray(arr, N);
    return 0;
}
```

LINKED LIST DATA STRUCTURE

A linked list is a linear collection of specially designed data elements, called nodes, linked to one another by means of pointers. Each node is divided into two parts: the first part contains the information of the element, and the second part contains the address of the next node in the linked list. Address part of the node is also called linked or next field. Following Fig 5:1 shows a typical example of node.

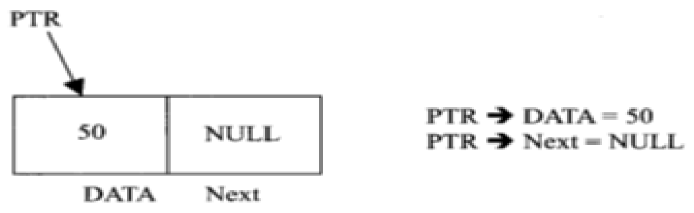
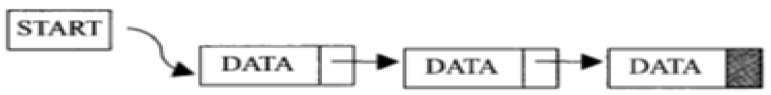


Fig. 5.1. Nodes.



The figure above shows a schematic diagram of a linked list with 3 nodes. Each node is pictured with two parts. The left part of each node contains the data items and the right part represents the address of the next node; there is an arrow drawn from it to the next node. The next pointer of the last node contains a special value, called the NULL pointer, which does not point to any address of the node. That is NULL pointer indicates the end of the linked list. START pointer will hold the address of the1st node in the list START = NULL if there is no list (*i.e.*; NULL list or empty list).

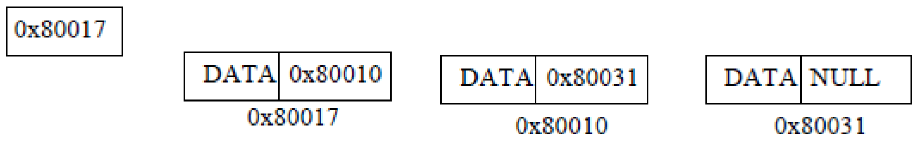


Fig. Linked List representation in memory.

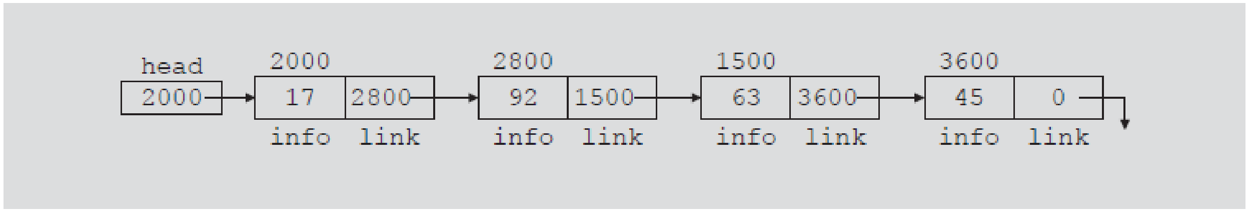
Explanation:

Because each node of a linked list has two components, we need to declare each node as a class or **struct**. The data type of each node depends on the specific application—that is, what kind of data is being processed. However, the link component of each node is a pointer. The data type of this pointer variable is the node type itself. For the previous linked list, the definition of the node is as follows. (Suppose that the data type is int.)

```
struct nodename
      {
        int info;
        nodename *link;
       };
```

The variable declaration is as follows:
nodename *head;

Linked List: Some Properties
To better understand the concept of a linked list and a node, some important properties of linked lists are described next.
Consider the linked list in in the following figure



Linked list with four nodes
This linked list has four nodes. The address of the first node is stored in the pointer head.
Each node has two components: info, to store the info, and link, to store the address of the next node.
For simplicity, we assume that info is of type int.
Suppose that the first node is at location 2000, the second node is at location 2800, the third node is at location 1500, and the fourth node is at location 3600.
The Table below shows the values of head and some other nodes in the list shown in this figure .

	Value	Explanation
head	2000	
head->info	17	Because **head** is 2000 and the **info** of the node at location 2000 is 17
head->link	2800	
head->link->info	92	Because **head->link** is 2800 and the **info** of the node at location 2800 is 92

Suppose that current is a pointer of the same type as the pointer head. Then the statement

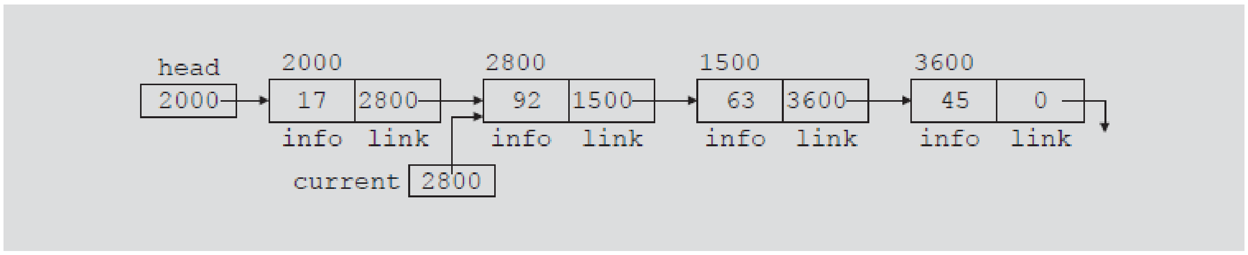
    current = head;

copies the value of head into current. Now consider the following statement:

    current = current->link;

This statement copies the value of current->link, which is 2800, into current.

Therefore, after this statement executes, current points to the second node in the list. (When working with linked lists, we typically use these types of statements to advance a pointer to the next node in the list.) See the figure .



List after the statement current = current->link; executes

The table shows the values of current, head, and some other nodes in Figure

TABLE Values of current, head, and some of the nodes of the linked list in Figure

	Value
`current`	2800
`current->info`	92
`current->link`	1500
`current->link->info`	63
`head->link->link`	1500
`head->link->link->info`	63
`head->link->link->link`	3600
`current->link->link->link`	0 (that is, NULL)
`current->link->link->link->info`	Does not exist (run-time error)

TRAVERSING A LINKED LIST

The basic operations of a linked list are as follows:

- Search the list to determine whether a particular item is in the list
- Insert an item in the list, display the elements of the list
- Delete an item from thelist.

These operations require the list to be traversed. That is, given a pointer to the first node of the list, we must step through the nodes of the list.

Suppose that the pointer **head** points to the first node in the list, and the link of the last node is **NULL**. We cannot use the pointer **head** to traverse the list because if we use the **head** to traverse the list, we would lose the nodes of the list. This problem occurs because the links are in only one direction. The pointer **head** contains the address of the first node, the first node contains the address of the second node, the second node contains the address of the third node, and so on. If we move **head** to the second node, the first node is lost (unless we save a pointer to this node). If we keep advancing **head** to the next node, we will lose all the nodes of the list (unless we save a pointer to each node before advancing **head**, which is impractical because it would require additional computer time and memory space to maintain the list). Therefore, we always want **head** to point to the first node. It now follows that we must traverse the list using another pointer of the same type. Suppose that **current** is a pointer of the same type as **head**. The following code traverses the list:

```
current = head;
while (current != NULL)
{
      //Process current
      current = current->link;
}
```

For example, suppose that **head** points to a linked list of numbers. The following code outputs the data stored in each node:

```
current = head;
while (current != NULL)
{
       cout << current->info << " ";
       current = current->link;
}
```

LINKED LIST ALGORITHMS

This section discusses the algorithms of linked list data structures. Consider the following definition of a node. (For simplicity, we assume that the info type is **int**.)

```
struct nodename
{
        int info;
        nodename *link;
};
```

We will use the following variable declaration:

```
        nodename *head, *p, *q, *newNode;
```

ALGORITHM FOR INSERTING A NODE

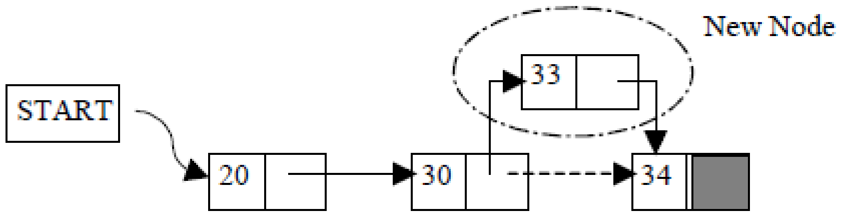


Fig. 5.14. Insertion of New Node

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.

<u>Insert a Node at the beginning</u>

```
1. Input DATA to be inserted
2. Create a NewNode
3. NewNode -> DATA = DATA
4. If (SATRT equal to NULL)
    (a) NewNode -> Link = NULL
5. Else
    (a) NewNode -> Link = START
6. START = NewNode
7. Exit
```

Insert a Node at the end

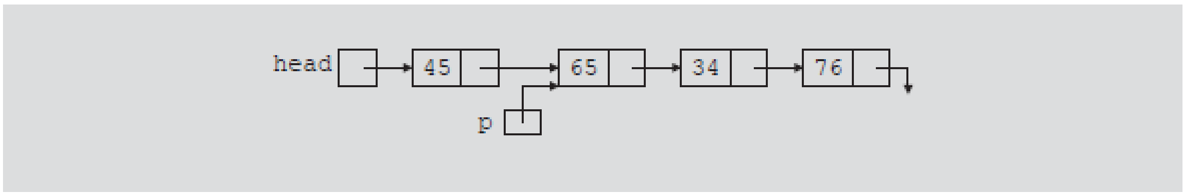
```
1. Input DATA to be inserted
2. Create a NewNode
3. NewNode -> DATA = DATA
4. NewNode -> Next = NULL
5. If (SATRT equal to NULL)
        (a) START = NewNode
6. Else
       (a) TEMP = START
       (b) While (TEMP -> Next not equal to NULL)
              (i) TEMP = TEMP -> Next
7. TEMP -> Next = NewNode
8. Exit
```

Insert a Node at any specified position

```
1. Input DATA and POS to be inserted
2. initialize TEMP = START; and k = 0
3. Repeat the step 3 while( k is less than POS)

        (a) TEMP = TEMP -> Next
        (b) If (TEMP is equal to NULL)
                (i) Display "Node in the list less than the position"
                (ii) Exit
        (c) k = k + 1
4. Create a New Node
5. NewNode -> DATA = DATA
6. NewNode -> Next = TEMP -> Next
7. TEMP -> Next = NewNode
8. Exit
```

Consider the linked list shown in this figure



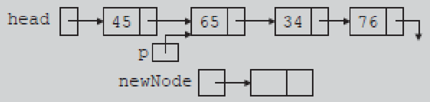
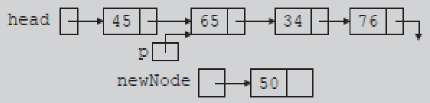
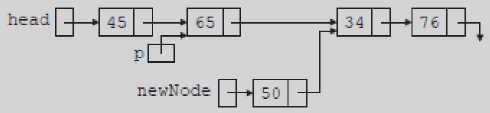
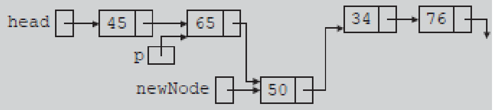
Suppose that **p** points to the node with **info 65**,

and a new node with **info 50** is to be created andinserted after **p**.

Consider the following statements:

```
newNode = new nodename;          //create newNode
newNode->info = 50;              //store 50 in the new node
newNode->link = p->link;
p->link = newNode;
```

The table shows the effect of these statements.

Statement	Effect
`newNode = new nodeType;`	
`newNode->info = 50;`	
`newNode->link = p->link;`	
`p->link = newNode;`	

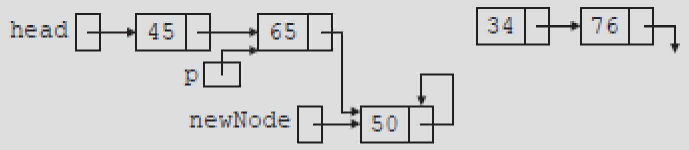
Note that the sequence of statements to insert the node, that is,

```
newNode->link = p->link;
p->link = newNode;
```

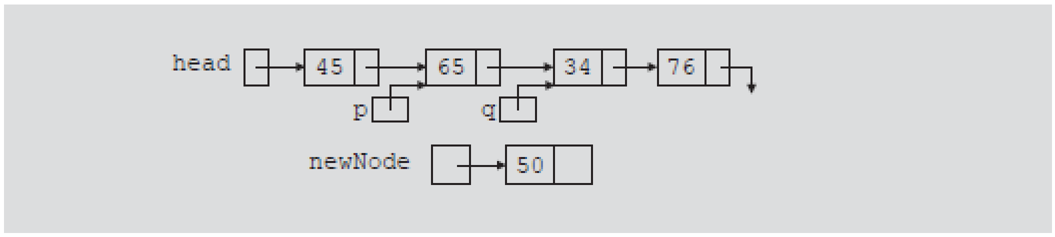
is very important because to insert **newNode** in the list we use only one pointer, **p**, to adjust the links of the nodes of the linked list. Suppose that we reverse the sequence of the statements and execute the statements in the following order:

```
p->link    =    newNode;
newNode->link = p->link;
```

The Figure shows the resulting list after these statements execute.



From Figure above, it is clear that **newNode** points back to itself and the remainder of the list is lost. Using two pointers, we can simplify the insertion code somewhat. Suppose **q** points to the node with **info 34**. (See Figure below)



List with pointers p and q

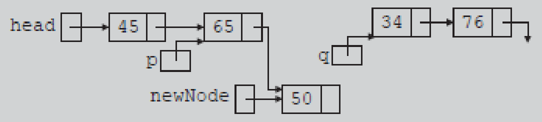
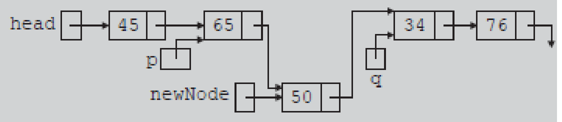
The following statements insert **newNode** between **p** and **q**:

newNode->link = q;
p->link = newNode;

The order in which these statements execute does not matter. To illustrate this, suppose that we execute the statements in the following order:

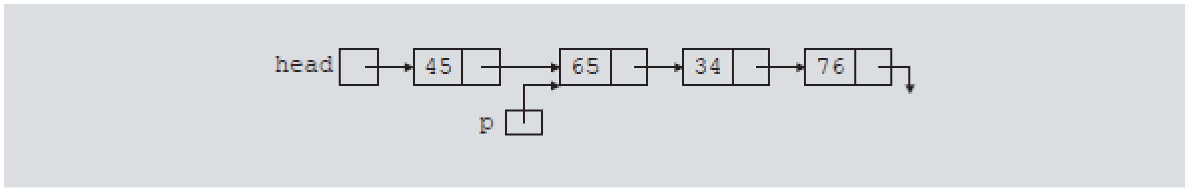
p->link = newNode;
newNode->link = q;

Table below shows the effect of these statements.

Statement	Effect
`p->link = newNode;`	
`newNode->link = q;`	

ALGORITHM FOR DELETING A NODE

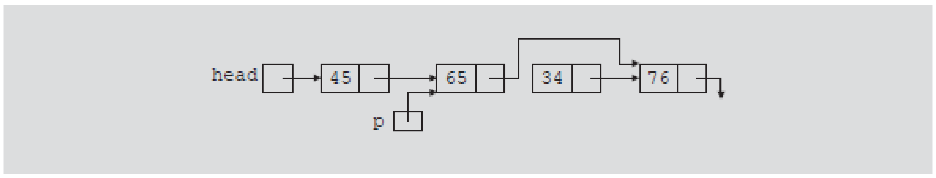
Consider the linked list shown in Figure



Suppose that the node with **info 34** is to be deleted from the list. The following statement removes the node from the list:

```
p->link = p->link->link;
```

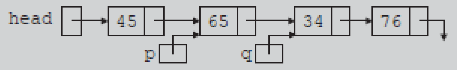
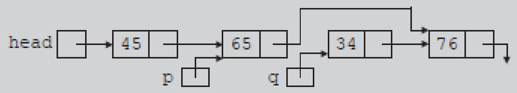
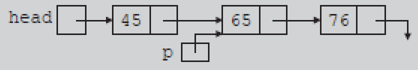
The Figure shows the resulting list after the preceding statement executes.

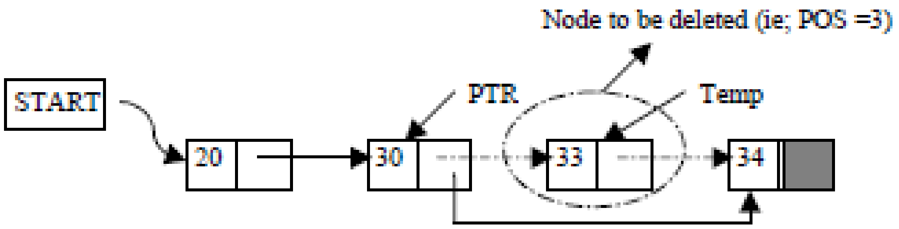


From the figure it is clear that the node with **info 34** is removed from the list.
However, the memory is still occupied by this node and this memory is inaccessible; that is, this node is dangling. To deallocate the memory, we need a pointer to this node. The following statements delete the node from the list and deallocate the memory occupied by this node:

```
q = p->link;
p->link = q->link;
delete q;
```

The table shows the effect of these statements.

Statement	Effect
`q = p->link;`	
`p->link = q->link;`	
`delete q;`	



Deletion of a Node

Suppose START is the first position in linked list. Let DATA be the element to be deleted. TEMP, HOLD is a temporary pointer to hold the node address.

```
1. Input the DATA to be deleted
2. if ((START -> DATA) is equal to DATA)
    (a) TEMP = START
    (b) START = START -> Next
    (c) Set free the node TEMP, which is deleted
    (d) Exit
3. HOLD = START
4. while ((HOLD -> Next -> Next) not equal to NULL))
    (a) if ((HOLD -> NEXT -> DATA) equal to DATA)
        (i) TEMP = HOLD -> Next
        (ii) HOLD -> Next = TEMP -> Next
        (iii) Set free the node TEMP, which is deleted
        (iv) Exit
    (b) HOLD = HOLD -> Next
5. if ((HOLD -> next -> DATA) == DATA)
    (a) TEMP = HOLD -> Next
    (b) Set free the node TEMP, which is deleted
    (c) HOLD -> Next = NULL
    (d) Exit
6. Disply "DATA not found"
7. Exit
```

ALGORITHM FOR SEARCHING A NODE

Suppose START is the address of the first node in the linked list and DATA is the information to be searched. After searching, if the DATA is found, POS will contain the corresponding position in the list.

```
1. Input the DATA to be searched
2. Initialize TEMP = START; POS =1;
3. Repeat the step 4, 5 and 6 until (TEMP is equal to NULL)
4. If (TEMP → DATA is equal to DATA)
    (a) Display "The data is found at POS"
    (b) Exit
5. TEMP = TEMP → Next
6. POS = POS+1
7. If (TEMP is equal to NULL)
    (a) Display "The data is not found in the list"
8. Exit
```

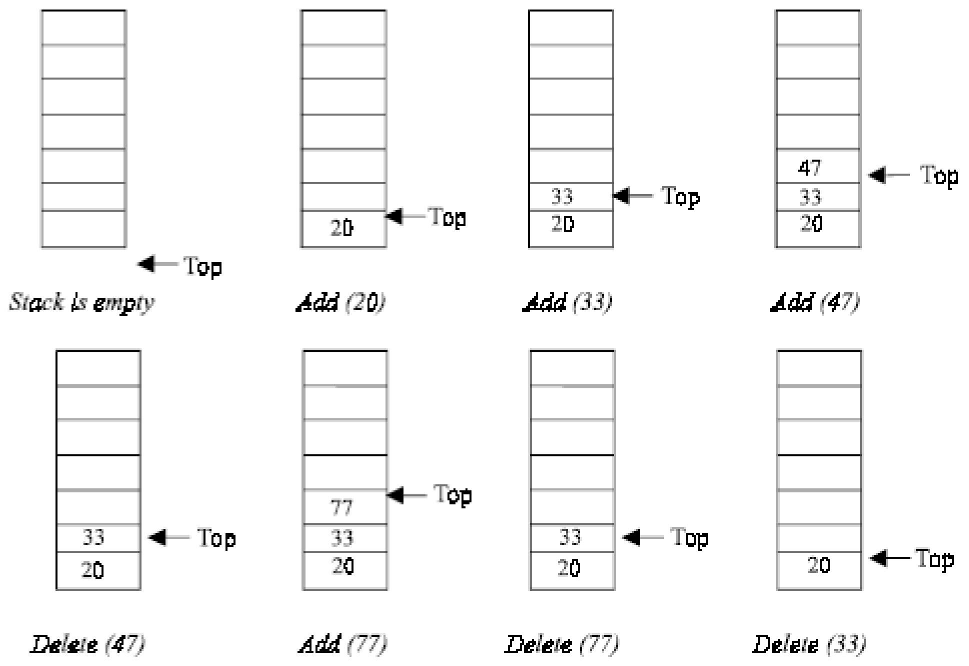
ALGORITHM FOR DISPLAY ALL NODES

Suppose START is the address of the first node in the linked list. Following algorithm will visit all nodes from the START node to the end.

```
1. If (START is equal to NULL)
        (a) Display "The list is Empty"
        (b) Exit
2. Initialize TEMP = START
3. Repeat the step 4 and 5 until (TEMP == NULL )
4. Display "TEMP → DATA"
5. TEMP = TEMP → Next
6. Exit
```

The Stack data structure

A stack is one of the most important and useful non-primitive linear data structure in computer science. It is an ordered collection of items into which new data items may be added/inserted and from which items may be **deleted at only one end**, called the top of the stack. As all the addition and deletion in a stack is done from



the top of the stack, the last added element will be first removed from the stack. That is why the stack is also called Last-in-First-out (LIFO). Notethat the most frequently accessible element in the stack is the top most elements, whereas the least accessible element is the bottom of the stack. The operation of the stack can be illustrated as In Figure the insertion (or addition) operation is referred to as push, and the deletion (or remove) operation as pop. A stack is said to be empty or underflow, if the stack contains no elements. At this point the top of the stack is present at the bottom of the stack. And it is overflow when the stack becomes full, i.e., no other elements can be pushed onto the stack. At this point the top pointer is at the highest location of the stack.

OPERATIONS PERFORMED ON STACK

The primitive operations performed on the stack are as follows:

PUSH: The process of adding (or inserting) a new element to the top of the stack is called PUSH operation. Pushing an element to a stack will add the new element at the top. After everypush operation the top is incremented by one. If the array is full and no new element can be accommodated, then the stack **overflow** condition occurs.

POP: The process of deleting (or removing) an element from the top of stack is called POP operation. After every pop operation the stack is decremented by one. If there is no element in the stack and the pop operation is performed then the stack **underflow** condition occurs.

STACK IMPLEMENTATION

Stack can be implemented in two ways:

1. Static implementation (using arrays)

2. Dynamic implementation (linked list)

Static implementation uses arrays to create stack. Static implementation using arrays is a very simple technique but is not a flexible way, as the size of the stack has to be declared during the program design, because after that, the size cannot be varied (i.e., increased or decreased). Moreover static implementation is not an efficient method when resource optimization is concerned (i.e., memory utilization).

For example a stack is implemented with array size 50. That is before the stack operation begins, memory is allocated for the array of size 50. Now if there are only few elements (say 30) to be stored in the stack, then rest of the statically allocated memory (in this case 20) will be wasted, on the other hand if there are more number of elements to be stored in the stack (say 60) then we cannot change the size array to increase its capacity. The above said limitations can be overcome by dynamically implementing (is also called linkedlist representation) the stack using pointers.

STACK USING ARRAYS

Implementation of stack using arrays is a very simple technique. Algorithm for pushing (or add or insert) a new element at the top of the stack and popping (or delete) an element from the stackis given below.

Algorithm for push

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack.

Let DATA is the data item to be pushed.

```
1. If TOP = SIZE – 1, then:
    (a) Display “The stack is in overflow condition”
    (b) Exit
2. TOP = TOP + 1
3. STACK [TOP] = ITEM
4. Exit
```

```
void push(void)
{
        int x;
        if(top==max-1) // Condition for checking If Stack is Full
        {
                cout<<"\n stack overflow\n";
                return;
        }
        cout<<"enter a no: ";
        cin>>x;
        a[++top]=x; //increment the top and inserting element
        }
```

Algorithm for pop

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. DATA is the popped (or deleted) data item from the top of the stack.

```
1. If TOP is equal to -1, then
    (a) Display “The Stack is empty”
    (b) Exit
2. DATA = STACK[TOP]
3. TOP = TOP – 1
4. Exit
```

```
void pop() {
    if (top == -1)
     {
        cout << "Stack Underflow" << endl;
     }
    int x = A[top];
    top--;
}
```

Algorithm for display

```
1. If TOP is equal to -1, then
    (a) Display “the stack is empty”
    (b) exit
2. i  from TOP to 0
    (a) display Array[i]
3. Exit
```

```
void display(void)
{
        if(top==-1)
        {
               cout <<"stack is empty\n";
        }
        cout<<"\n elements of Stack are : ";
        for(int i=0;i<=top;i++)
        {
                cout << a[i] << "   ";
        }
          cout << endl << endl;
        return;
 }
```

STACK USING LINKED LIST

we have discussed the implementation of stack using array, i.e., static memory allocation. Implementation issues of the stack (Last In First Out - LIFO) using linked list is illustrated in following figures.

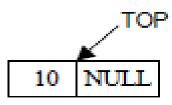


Fig. 5.11. push (10)

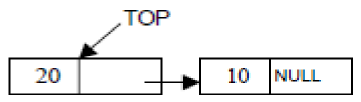


Fig. 5.12. push (20)

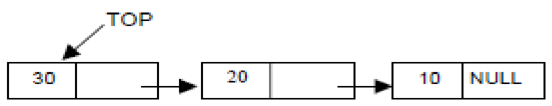


Fig. 5.13. push (30)

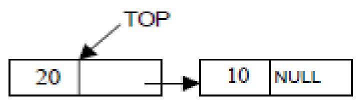


Fig. 5.14. X = pop() (ie; X = 30)

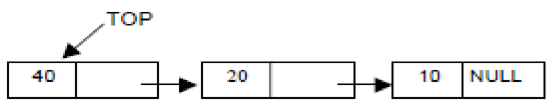


Fig. 5.15. push (40)

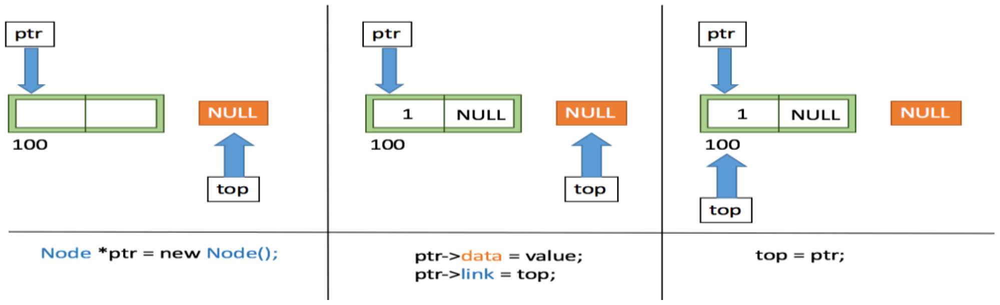
```
Struct Node{
 Int data;
Node *link;
};
```

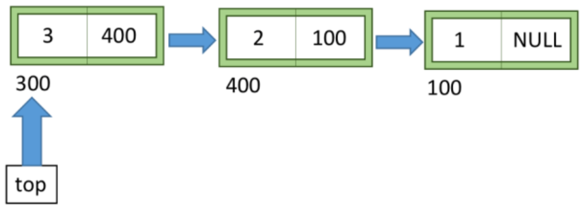
Algorithm for push operation

1. The push operation would be similar to inserting a node at starting of the linked list
2. So initially when the Stack (Linked List) is empty, the top pointer will be NULL. Let's suppose we have to insert the values 1, 2 & 3 in the stack.
3. So firstly we will create a new Node using the new operator and return its address in temporary pointer ptr.
4. Then we will insert the value 1 in the data part of the Node : ptr->data = value

   and make link part of the node equal to top : ptr->link=top.

5. Finally we will make top = ptr to point it to the newly created node which will now be the starting of the linked list and top of our stack.
6. Similarly we can push the values 2 & 3 in the stack which will give us a linked list of three nodes with top pointer pointing to the node containing value 3.





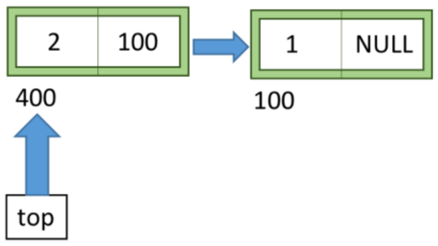
Suppose TOP is a pointer, which is pointing towards the top most element of the stack. TOP isNULL when the stack is empty. DATA is the data item to be pushed.

1. Input the DATA to be pushed
2. Creat a New Node
3. NewNode → DATA = DATA
4. NewNode → Next = TOP
5. TOP = NewNode
6. Exit

```
void push (int value)
{
  Node *ptr = new Node();
  ptr->data = value;
  ptr->link = top;
  top = ptr;
}
```

Algorithm for pop operation

1. The pop operation would be similar to deleting a node from the starting of a linked list.
2. So we will take a temporary pointer ptr and equate it to the top pointer.
3. Then we will move the top pointer to the next node i.e. top = top->link
4. Finally, we will delete the node using delete operator and pointer ptr i.e delete(ptr)



Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. TEMP is pointer variable to hold any nodes address. DATA is the information on the node which is just deleted.

```
1. if (TOP is equal to NULL)
    (a) Display "The stack is empty"
2. Else
    (a) TEMP = TOP
    (b) Display "The popped element TOP → DATA"
    (c) TOP = TOP → Next
    (d) TEMP → Next = NULL
    (e) Free the TEMP node
3. Exit
```

```
void pop ( )
{
 if ( isempty() )
  cout<<"Stack is Empty";
 else
 {
  Node *ptr = top;
  top = top -> link;
  delete(ptr);
 }
}
```

Algorithm for display operation

```
1. if (TOP is equal to NULL)
     (a) display “the stack is empty”
     (b) exit
2. else
     (a) temp = top
     (b) while temp is not equal to null
           (b.1) display temp->info
           (b.2) temp = temp->link
3. exit
```

```
void displayStack()
{
 if ( isempty() )
  cout<<"Stack is Empty";
 else
 {
  Node *temp=top;
  while(temp!=NULL)
  {
   cout<<temp->data<<" ";
   temp=temp->link;
  }
  cout<<"\n";
 }
 }
```

DOUBLY LINKED LIST

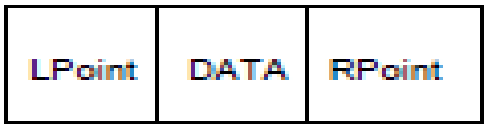
A doubly linked list is one in which all nodes are linked together by multiple links which help in accessing both the successor (next) and predecessor (previous) node for any arbitrary node within the list. Every nodes in the doubly linked list has three fields:

LeftPointer,

RightPointer

DATA.

Fig. below shows a typical doubly linked list.



LPoint will point to the node in the left side (or previous node) that is LPoint will hold the address of the previous node. RPoint will point to the node in the right side (or nex node) that is RPoint will hold the address of the next node. DATA will store the information of the node.

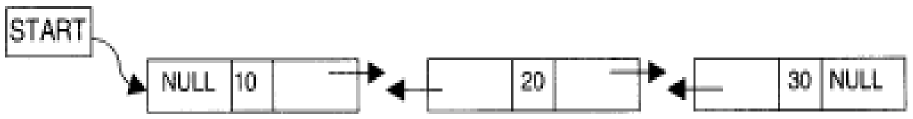
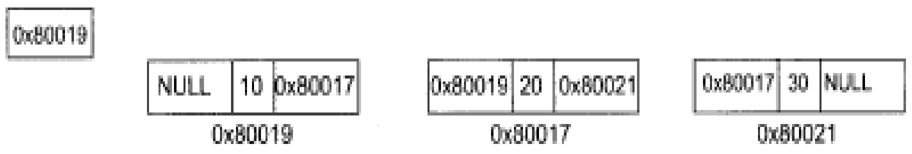


Fig. 5.25. Doubly Linked List



Representation of doubly linked list

A node in the doubly linked list can be represented in memory with the following declarations.

```
Struct Node
{
    Int DATA;
    Node *next;
    Node *prev;
};
```

All the operations performed on singly linked list can also be performed on doubly linked list.

Following figure will illustrate the insertion and deletion of nodes.

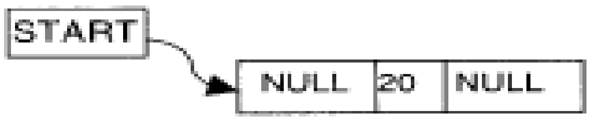


Fig. 5.27. Add(20)

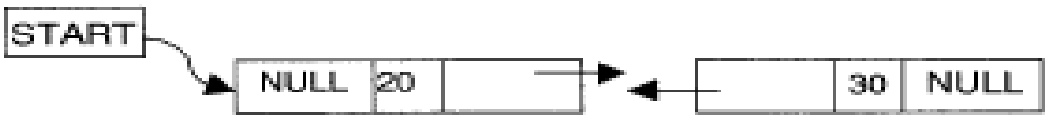
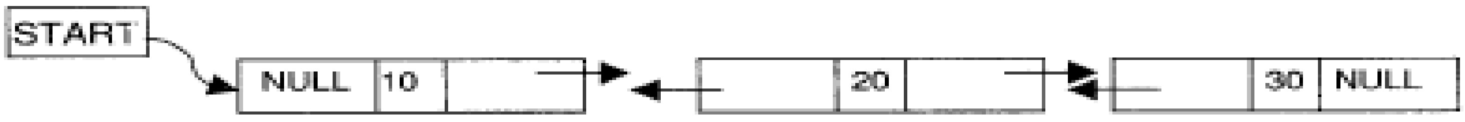
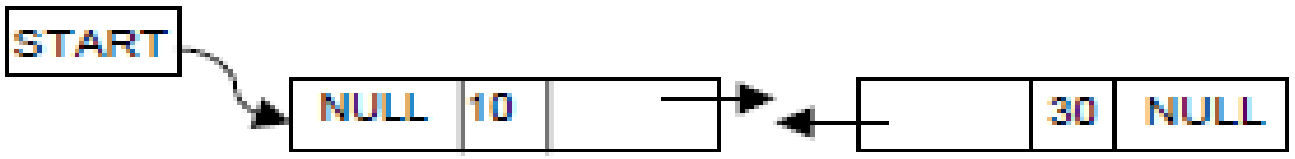


Fig 5.28. Insert (30) at the end





Algorithm for creating a doubly linked list (inserting at the end)

```
1. Input the DATA to be inserted
2. TEMP to create a new node
3. TEMP->info=DATA
4. TEMP->next=NULL
5. if (START is equal to NULL)
    5.a. TEMP->prev=NULL
    5.b. START=TEMP
    5.c. exit
6. else
    6.a. HOLD =START
    6.b. while(HOLD->next not equal to NULL)
        6.b.1 HOLD=HOLD->next
    6.c. HOLD->next=TEMP
    6.d. TEMP->prev=HOLD
7. exit
```

Algorithm for inserting a node at the beginning

```
1. Input the DATA to be inserted
2. TEMP to create a new node
3. TEMP->prev=NULL
4. TEMP->info=DATA
5. TEMP->next=START
6. if (START is equal to NUUl)
        6.a. START=TEMP
        6.b. exit
7. START->prev=TEMP
8. START=TEMP
9. exit
```

Algorithm for inserting a node at a specific position

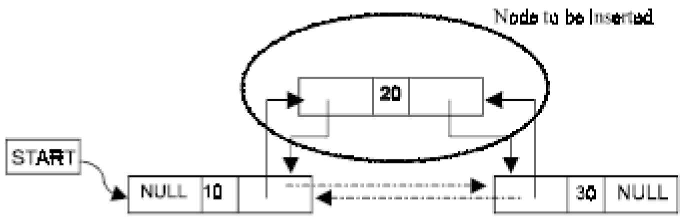
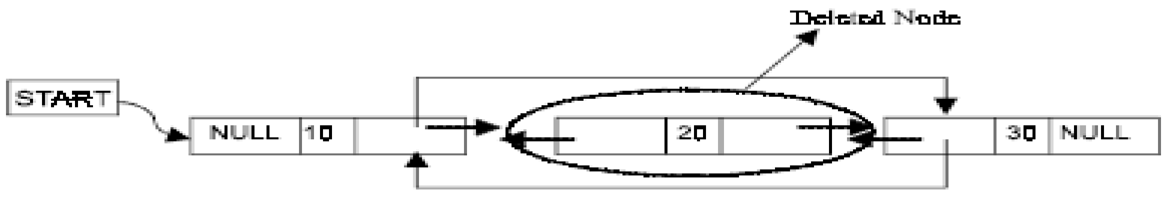


Fig. 5.31. Insert a node at the 2nd position

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 NewNode is to be inserted. TEMP is a temporary pointer to hold the node address.

```
1. Input the DATA and POS
2. Initialize TEMP = START; i = 0
3. Repeat the step 4 if (i less than POS) and (TEMP is not equal to NULL)
4. TEMP = TEMP ->next; i = i +1
5. If (TEMP not equal to NULL) and (i equal to POS)
    (a) Create a New Node
    (b) NewNode -> DATA = DATA
    (c) NewNode -> next = TEMP -> next
    (d) NewNode -> prev = TEMP
    (e) (TEMP -> next) -> prev = NewNode
    (f ) TEMP -> next = New Node
6. Else
    (a) Display “Position NOT found”
7. Exit
```

Algorithm for deleting a node



Suppose START is the address of the first node in the linked list. Let DATA be the Element to be deleted. TEMP, HOLD is the temporary pointer to hold the address of the node.

```
1. Input the DATA to be deleted
2. if ((START->DATA)is equal to DATA)
      (a) TEMP = START
      (b) START = STAR->next
      (c) START->prev=NULL
      (d) set free the node TEMP , which is deleted
      (e) Exit
3. HOLD = START
4. while((HOLD->next->next) not equal to NULL)
      (a)if (HOLD->next->DATA) equal to DATA)
            (a1) TEMP = HOLD->next
            (a2)HOLD->next=TEMP->next
            (a3) TEMP->next->prev=HOLD
            (a4) set free the node TEMP, which is deleted
            (a5) Exit
      (b) HOLD=HOLD->next
5. if ((HOLD->next->DATA)==DATA)
      (a) TEMP=HOLD->next
      (b) set free the node TEMP, which is deleted
      (c) HOLD->next=NULL
      (d) Exit
```

```
6. Display "DATA not found"
7. Exit
```

Algorithm for displaying the doubly linked list

```
1. if (START is equal to NULL)
    1.a. display "The list is empty"
    1.b. exit
2. TEMP=START
3. while TEMP is not equal to NULL
    3.a. display TEMP->info
    3.b. TEMP=TEMP->next
4. Exit
```

Advantages:

- The doubly linked list can be traversed in forward as well as backward directions, unlike singly linked list which can be traversed in the forward direction only.
- Delete operation in a doubly-linked list is more efficient when compared to singly list when a given node is given. In a singly linked list, as we need a previous node to delete the given node, sometimes we need to traverse the list to find the previous node. This hits the performance.
- Insertion operation can be done easily in a doubly linked list when compared to the singly linked list.

Disadvantages:

- As the doubly linked list contains one more extra pointer i.e. previous, the memory space taken up by the doubly linked list is larger when compared to the singly linked list.
- Since two pointers are present i.e. previous and next, all the operations performed on the doubly linked list have to take care of these pointers.

Assignment

1. write an algorithm and a function to display a doubly linked list in reverse order.
2. write an algorithm and a function to count the number of elements in the doubly linked list.
3. write an algorithm and function for searching a number in doubly linked list.

Difference Between Singly and Doubly Linked List

Feature	**Singly Linked List**	**Doubly Linked List**
Direction of Traversal	Only forwards. You can traverse from the head to the end of the list.	Both forwards and backwards. You can traverse from the head to the tail and vice versa.
Memory Usage	Less memory per node (one pointer per node).	More memory per node (two pointers per node).
Insertions/Deletions	Efficient at the beginning; requires traversal from the head for other positions.	Efficient at both the beginning and the end; easier insertions and deletions in the middle without full traversal.
Complexity	Simpler and requires less code to manage compared to doubly linked lists.	More complex due to additional pointers, requiring more management and care in code.
Use Case	Suitable when memory is a concern and only forward traversal is needed.	Preferred when frequent operations require elements to be accessed from both ends.
Operations	Typically slower for operations that involve elements at the end of the list.	Faster for operations involving the end of the list due to the tail pointer.
Head and Tail Management	Only head pointer is used.	Both head and tail pointers are used, providing immediate access to both ends of the list.

الفرق بين استخدام class و struct في C++

في لغة C++ ، كلا من class وstruct يستخدمان لتعريف أنواع بيانات مخصصة

(user-defined data Types).

ومع ذلك، هناك اختلاف رئيسي بينهما:

الاختلاف الرئيسي:

- **الوصول إلى الأعضاء:**
  - **struct:** بشكل افتراضي، جميع أعضاء struct تكون عامة (public) ويمكن الوصول إليها من أي مكان في البرنامج.
  - **class:** بشكل افتراضي، جميع أعضاء class تكون خاصة (private) ولا يمكن الوصول إليها إلا من داخل الكلاس نفسه أو من خلال الدوال الأعضاء (member functions).

Stack applications

Expression

An application of stack is calculation of postfix expression. There are basically three types of notation for an expression (mathematical expression; An expression is defined as the number of operands or data items combined with several operators.)

1. Infix notation: A+B

2. Prefix notation: +AB

3. Postfix notation: AB+

The infix notation is what we come across in our general mathematics, where the operator is written in-between the operands. For example: The expression to add two numbers A and B is written in infix notation as: A + B
Note that the operator '+' is written in between the operands A and B.

The prefix notation is a notation in which the operator(s) is written before the operands, it is also called **polish notation**. The same expression when written in prefix notation looks like: + A B
As the operator '+' is written before the operands A and B, this notation is called prefix (pre means before).

In the postfix notation the operator(s) are written after the operands, so it is called the postfix notation (post means after), it is also known as **suffix notation** or **reverse polish notation**. The above expression if written in postfix expression looks like:
A B +

The prefix and postfix notations are not really as awkward to use as they might look. For example, a C function to return the sum of two variables A and B (passed as argument) is called or invoked by the instruction: add(A, B)
Note that the operator add (name of the function) precedes the operands A and B.

Because the postfix notation is most suitable for a computer to calculate any expression (due to its reverse characteristic), and is the universally accepted notation for designing Arithmetic and Logical Unit (ALU) of the CPU (processor). Therefore it is necessary to study the postfix notation. Moreover the postfix notation is the way computer looks towards arithmetic expression, any expression entered into the computer is first converted into postfix notation, stored in stack and then calculated. In the preceding sections we will study the conversion of the expression from one notation to other.

Advantages of using postfix notation

Human beings are quite used to work with mathematical expressions in infix notation, which is rather complex. One has to remember a set of nontrivial rules while using this notation and it must be applied to expressions in order to determine the final value. These rules include precedence,

BODMAS (Order of Operations), and associativity.

Using infix notation, one cannot tell the order in which operators should be applied.

Whenever an infix expression consists of more than one operator, the precedence rules (BODMAS) should be applied to decide which operator (and operand associated with that operator) is evaluated first.

But in a postfix expression operands appear before the operator, so there is no need for operator precedence and other rules. As soon as an operator appears in the postfix expression during scanning of postfix expression the topmost operands are popped off and are calculated by applying the encountered operator. Place the result back onto the stack; likewise at the end of the whole operation the final result will be there in the stack.

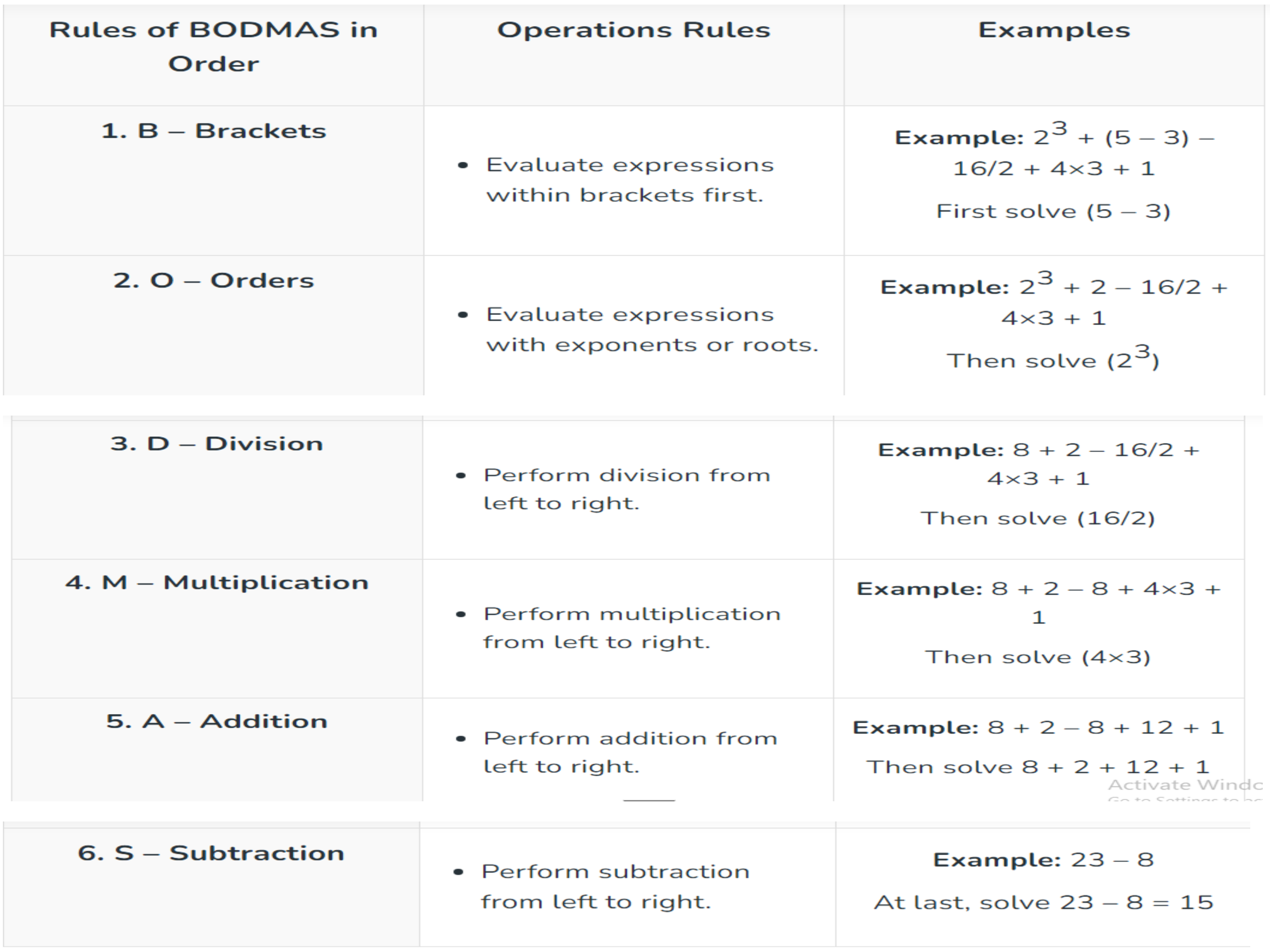
Notation Conversions

Let A + B * C be the given expression, which is an infix notation.

To calculate this expression for values 4, 3, 7 for A, B, respectively we must follow certain rule (called BODMAS in general mathematics) in order to have the right result. For example:

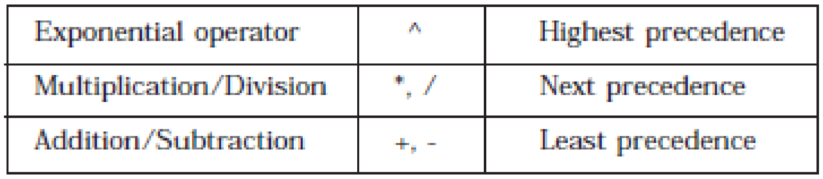
A + B * C = 4 + 3 * 7 = 7 * 7 = 49

The answer is not correct; multiplication is to be done before the addition, because multiplication has higher precedence over addition. This means that an expression is calculated according to the operator's precedence not the order as they look like. The error in the above calculation occurred, since there were no braces to define the precedence of the operators. Thus expression A + B * C can be interpreted as A + (B * C). Using this alternative method we can convey to the computer that multiplication has higher precedence over addition.

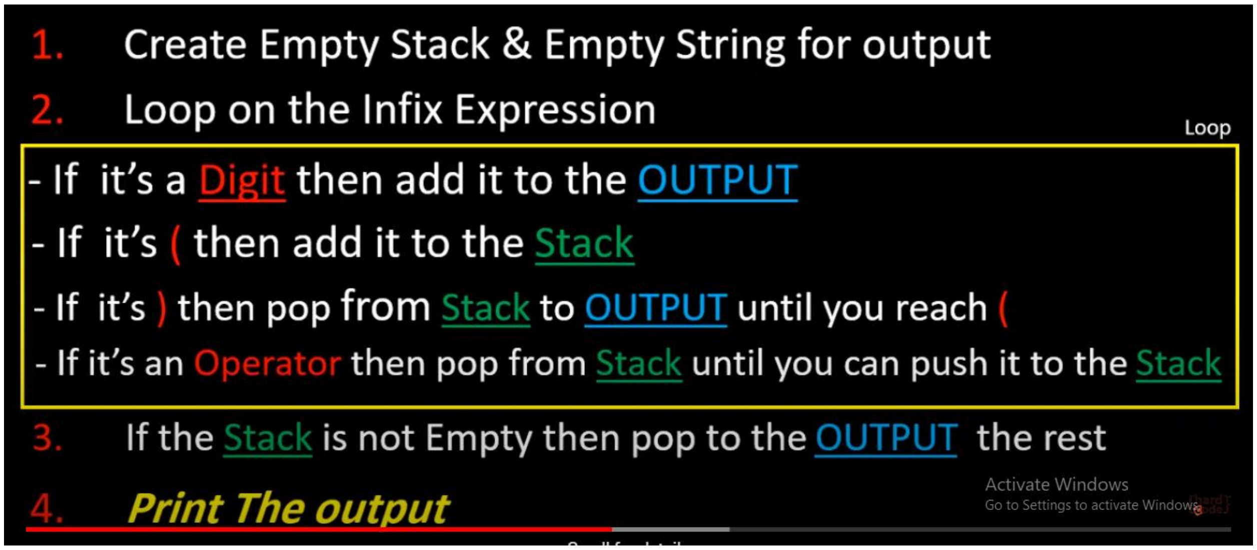


Rules of BODMAS in Order	Operations Rules	Examples
1. B – Brackets	• Evaluate expressions within brackets first.	**Example:** 2^3 + (5 – 3) – 16/2 + 4×3 + 1 First solve (5 – 3)
2. O – Orders	• Evaluate expressions with exponents or roots.	**Example:** 2^3 + 2 – 16/2 + 4×3 + 1 Then solve (2^3)
3. D – Division	• Perform division from left to right.	**Example:** 8 + 2 – 16/2 + 4×3 + 1 Then solve (16/2)
4. M – Multiplication	• Perform multiplication from left to right.	**Example:** 8 + 2 – 8 + 4×3 + 1 Then solve (4×3)
5. A – Addition	• Perform addition from left to right.	**Example:** 8 + 2 – 8 + 12 + 1 Then solve 8 + 2 + 12 + 1
6. S – Subtraction	• Perform subtraction from left to right.	**Example:** 23 – 8 At last, solve 23 – 8 = 15

Operator precedence



Exponential operator	^	Highest precedence
Multiplication/Division	*, /	Next precedence
Addition/Subtraction	+, -	Least precedence



1. Create Empty Stack & Empty String for output
2. Loop on the Infix Expression

Loop

- If it's a Digit then add it to the OUTPUT
- If it's (then add it to the Stack
- If it's) then pop from Stack to OUTPUT until you reach (
- If it's an Operator then pop from Stack until you can push it to the Stack

3. If the Stack is not Empty then pop to the OUTPUT the rest
4. *Print The output*

The method of converting infix expression A + B * C to postfix form is:

A + B * C Infix Form

A + (B * C) Parenthesized expression

A + (B C *) Convert the multiplication

A (B C *) + Convert the addition

A B C * + Postfix form

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to light.
2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in the above expression B * C is parenthesized first before A + B.
3. The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
4. Once the expression is converted to postfix form, remove the parenthesis.

Problem 1. Give postfix form for A + [(B + C) + (D + E) * F] / G

Solution. Evaluation order is

A + { [(BC +) + (DE +) * F] / G}

A + { [(BC +) + (DE +) F *] / G}

A + { [(BC +) (DE + F *)+] / G} .

A + [BC + DE + F *+ G /]

ABC + DE + F * + G / +

Problem 2.

Give postfix form for (A + B) * C / D + E ^ A / B Solution. Evaluation order is

[(AB +) * C / D] + [(EA ^) / B]

[(AB +) * C / D] + [(EA ^) B /]

[(AB +) C * D /] + [(EA ^) B /]

(AB +) C * D / (EA ^) B / +

AB + C * D / EA ^ B / +   Postfix Form

Algorithm

Suppose P is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Q. Besides operands and operators, P (infix notation) may also contain left and right parentheses. We assume that the operators in P consists of only exponential (^), multiplication (*), division (/), addition (+) and subtraction (-).

The algorithm uses a stack to temporarily hold the operators and left parentheses.

The postfix expression Q will be constructed from left to right using the operands from P and operators, which are removed from stack. We begin by pushing a left parenthesis onto stack and adding a right parenthesis at the end of P. the algorithm is completed when the stack is empty.

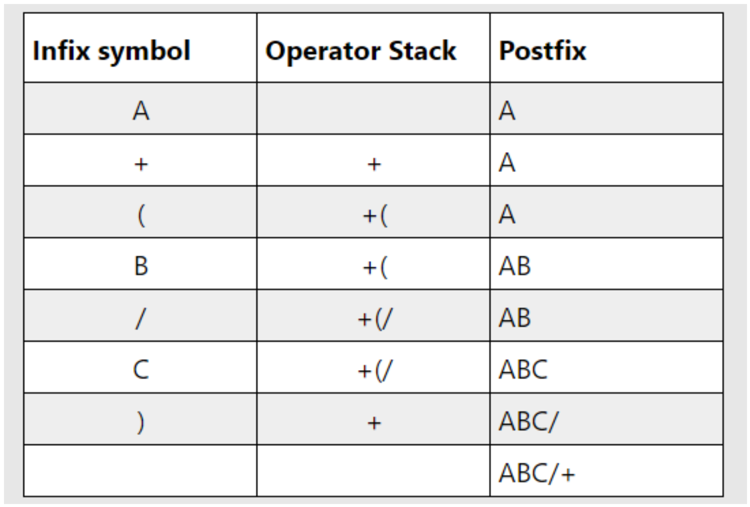
1. Scan P from left to right and repeat Steps 3 to 6 for each element of P until the stack is empty.
2. If an operand is encountered, add it to Q.
3. If a left parenthesis is encountered, push it onto stack.
4. If an operator ⊗ is encountered, then:
   (a) Repeatedly pop from stack and add to Q each operator (on the top of stack), which has the same precedence as, or higher precedence than ⊗.
   (b) Add ⊗ to stack.
5. If a right parenthesis is encountered, then:
   (a) Repeatedly pop from stack and add to Q (on the top of stack until a left parenthesis is encountered.
   (b) Remove the left parenthesis. [Do not add the left parenthesis to stack.]
6. Exit.

Note. Special character ⊗ is used to symbolize any operator in P.

Consider the following arithmetic infix expression P

P = A + (B / C)

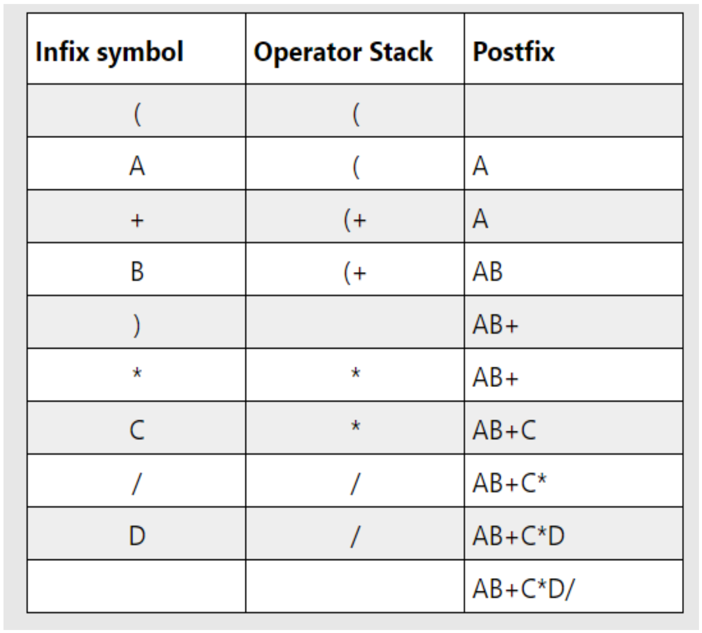
Figure below shows the character (operator, operand or parenthesis) scanned, status of the stack and postfix expression Q of the infix expression P.



Infix symbol	Operator Stack	Postfix
A		A
+	+	A
(	+(	A
B	+(	AB
/	+(/	AB
C	+(/	ABC
)	+	ABC/
		ABC/+

Converting infix to postfix expression

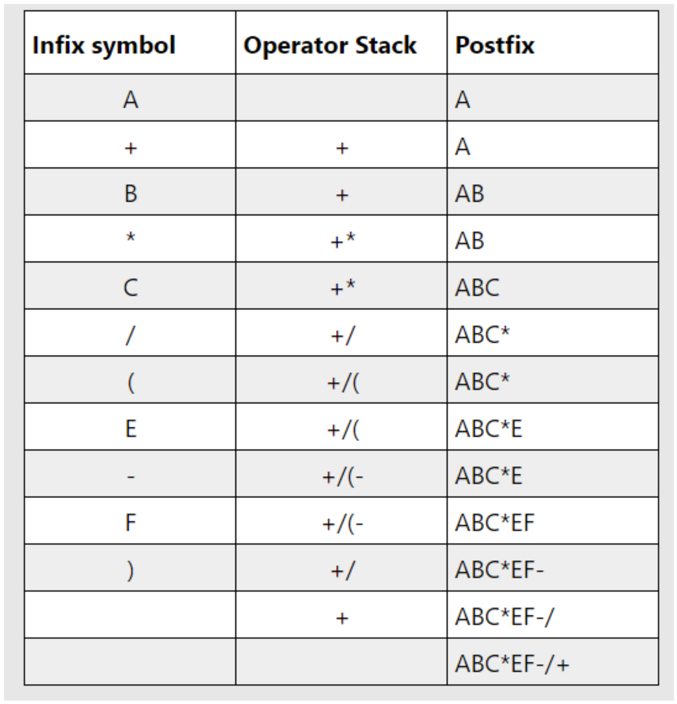
(A + B) * C / D



Infix symbol	Operator Stack	Postfix
(	(	
A	(	A
+	(+	A
B	(+	AB
)		AB+
*	*	AB+
C	*	AB+C
/	/	AB+C*
D	/	AB+C*D
		AB+C*D/

Postfix Form

Convert A+B*C/(E-F) to postfix +A/*BC-EF



Infix symbol	Operator Stack	Postfix
A		A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
/	+/	ABC*
(	+/(	ABC*
E	+/(	ABC*E
-	+/(-	ABC*E
F	+/(-	ABC*EF
)	+/	ABC*EF-
	+	ABC*EF-/
		ABC*EF-/+

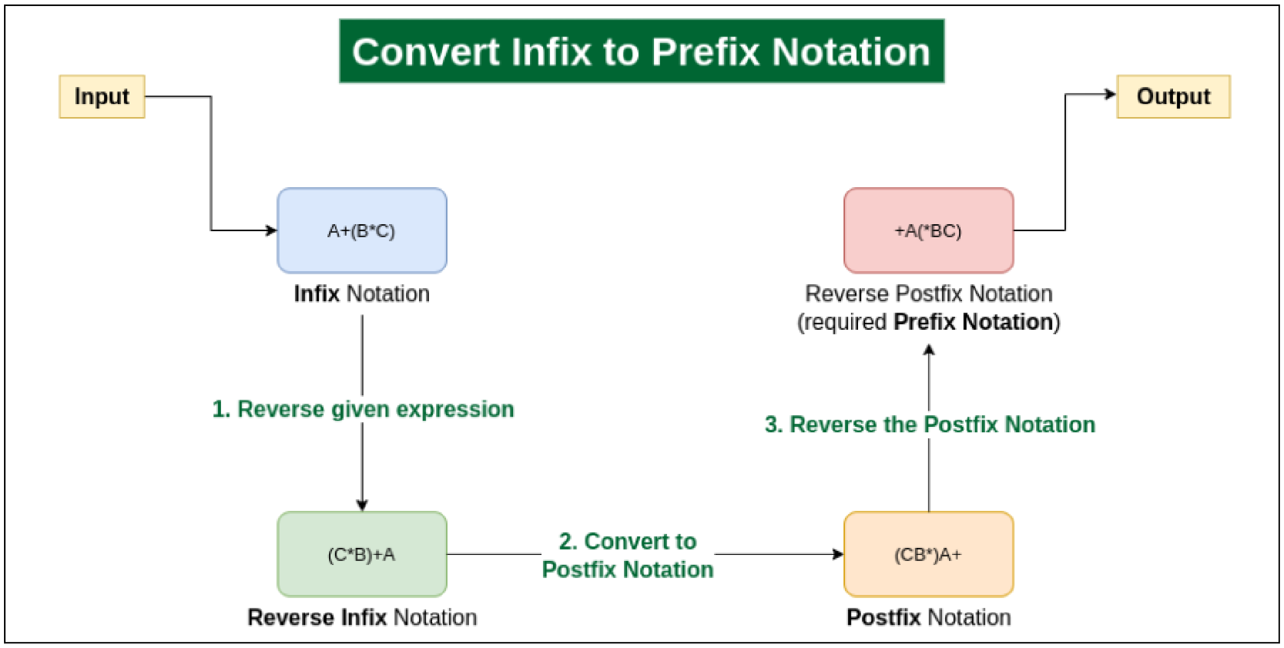
Evaluating postfix expression

Following algorithm finds the RESULT of an arithmetic expression P written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm

1. Scan P from left to right and repeat Steps 3 and 4 for each element of P.
2. If an operand is encountered, put it on STACK.
3. If an operator ⊗ is encountered, then:
   (a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
   (b) Evaluate B ⊗ A.
   (c) Place the result on to the STACK.
4. Result equal to the top element on STACK.
5. Exit.

Infix to prefix


Convert Infix to Prefix Notation
Input
A+(B*C)
Infix Notation
1. Reverse given expression
(C*B)+A
Reverse Infix Notation
2. Convert to Postfix Notation
(CB*)A+
Postfix Notation
3. Reverse the Postfix Notation
+A(*BC)
Reverse Postfix Notation (required Prefix Notation)
Output

A+B*C/(E-F)

- 1. Reversed string: (F-E)/C*B+A
- 2. Postfix of (F-E)/C*B+A: FE-C/B*A+
- 3. Reversed string of FE-C/B*A+: +A*B/C-EF

Input String	Output Stack	Operator Stack
(F-E)/C*B+A		(
(F-E)/C*B+A	F	(
(F-E)/C*B+A	F	(-
(F-E)/C*B+A	FE	(-
(F-E)/C*B+A	FE-	
(F-E)/C*B+A	FE-	/
(F-E)/C*B+A	FE-C	/
(F-E)/C*B+A	FE-C/	*
(F-E)/C*B+A	FE-C/B	*
(F-E)/C*B+A	FE-C/B*	+
(F-E)/C*B+A	FE-C/B*A	+
(F-E)/C*B+A	FE-C/B*A+	

A+B*C

- 1. Reversed string: C*B+A
- 2. Postfix of C*B+A: CB*A+

Input String	Output Stack	Operator Stack
C*B+A	C	
C*B+A	C	*
C*B+A	CB	*
C*B+A	CB*	+
C*B+A	CB*A	+
C*B+A	CB*A+	

- 3. Reversed string of CB*A+: +A*BC

The Queues

A queue is logically a *first in first out (FIFO or first come first serve)* linear data structure. The concept of queue can be understood by our real life problems. For example a customer come and join in a queue to take the train ticket at the end (rear) and the ticket is issued from the front end of queue. That is, the customer who arrived first will receive the ticket first. It means the customers are serviced in the order in which they arrive at the service center.

It is a homogeneous collection of elements in which new elements are added at one end called *rear*, and the existing elements are deleted from other end called *front*.

The basic operations that can be performed on queue are

1. Insert (or add) an element to the queue (push)

2. Delete (or remove) an element from a queue (pop)

- Push operation will insert (or add) an element to queue, at the rear end, by incrementing the array index.
- Pop operation will delete (or remove) from the front end by decrementing the array index and will assign the deleted value to a variable.
- Total number of elements present in the queue is **(rear-front)+1**, when implemented using arrays. Following figure will illustrate the basic operations on queue.

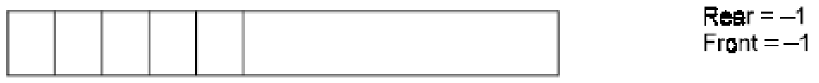


Fig. 4.1. Queue is empty.

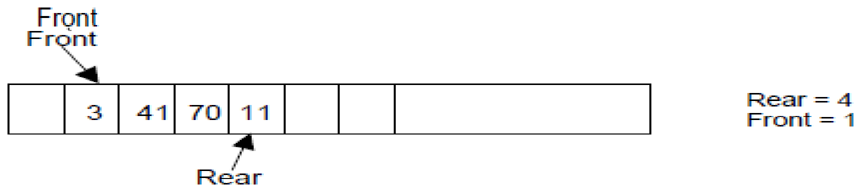


Fig. 4.7. push(11)

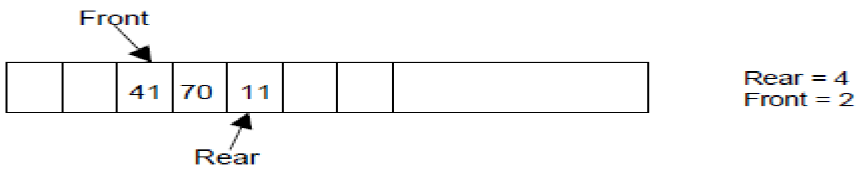


Fig. 4.8. x = pop() (*i.e.*; x = 3)

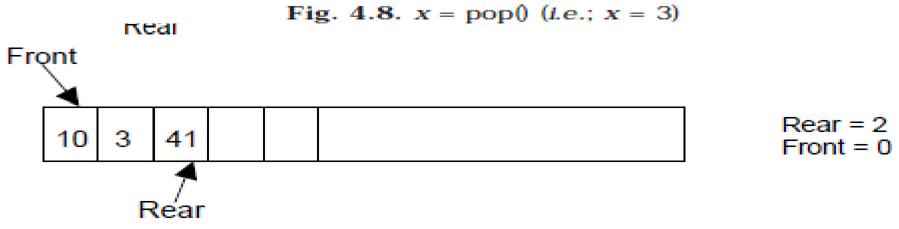


Fig. 4.4. push(41)

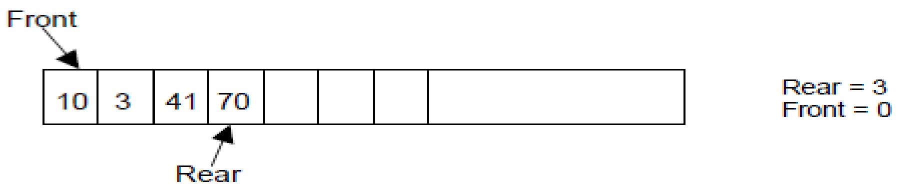


Fig. 4.5. push(70)

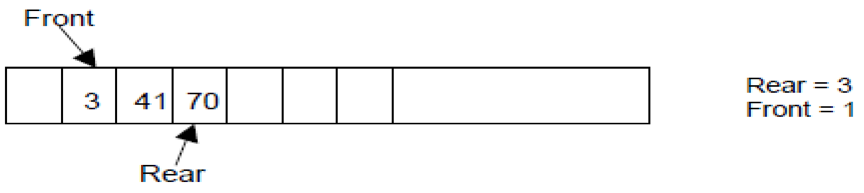


Fig. 4.6. x = pop() (*i.e.*; x = 10)

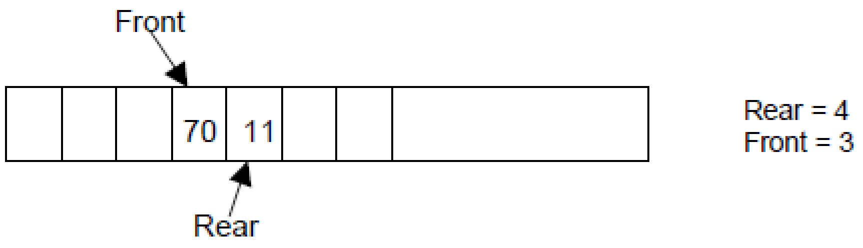
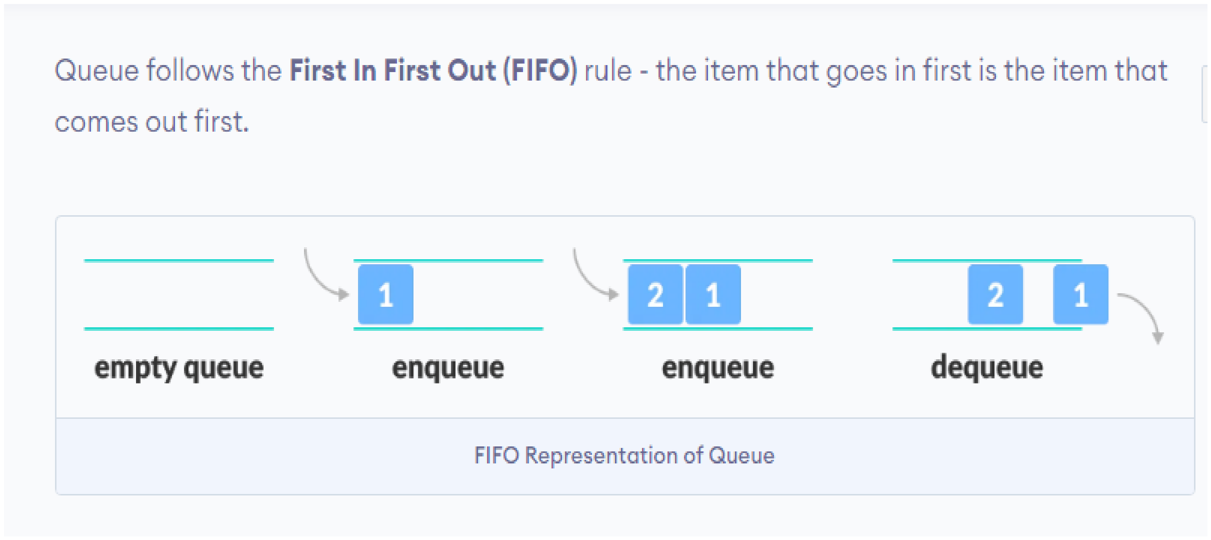


Fig. 4.9. x = pop() (*i.e.*, x = 41)

- **Operation on Queue:**
    - Enqueue: Adds an element to the rear of the queue
    - Dequeue: Removes an element from the front of the queue
    - Peek: Retrieves the front element without removing it
    - IsEmpty: Checks if the queue is empty
    - IsFull: Checks if the queue is full


Queue follows the **First In First Out (FIFO)** rule - the item that goes in first is the item that comes out first.

FIFO Representation of Queue

Queue can be implemented in two ways:

1. Using arrays (static)
2. Using pointers (Linked List) (dynamic)

Implementation of queue using pointers will be discussed later. Let us discuss underflow and overflow conditions when a queue is implemented using arrays.

If we try to pop (or delete or remove) an element from queue when it is empty, **underflow** occurs. It is not possible to delete (or take out) any element when there is no element in the queue.

Suppose maximum size of the queue (when it is implemented using arrays) is 50. If we try to push (or insert or add) an element to queue, **overflow** occurs. When queue is full it is naturally not possible to insert any more elements.

ALGORITHM FOR QUEUE OPERATIONS

Let Q be the array of some specified size say SIZE

1- INSERTING AN ELEMENT INTO THE QUEUE

```
1. Initialize front= –1, rear = –1
2. Input the value to be inserted and assign to variable "data"
3. If (rear = = SIZE-1)
    (a) Display "Queue overflow"
    (b) Exit
4. Else
    (a) Rear = rear +1
5. Q[rear] = data
6. Exit
```

2- DELETING AN ELEMENT FROM QUEUE

```
1. If (rear< front) or (front and rear is equal to -1)
       (a) Front = –1, rear = –1
       (b) Display “The queue is empty”
       (c) Exit
2. Else
       (a) Data = Q[front]
3. Front = front +1
4. Exit
```

3- DISPLAY THE ELEMENTS OF QUEUE

```
1. If (rear< front) or (front and rear is equal to -1)
       (a) Display “The queue is empty”
       (b) Exit
2. Else
       (a) i=front to rear
             (1) display Queue[i]
       (b) Exit
3. Exit
```

QUEUE USING LINKED LIST

Queue is a First In First Out [FIFO] data structure. We have discussed the implementation of stack using array, ie; static memory allocation. Implementation issues of the stack (Last In First Out - LIFO) using linked list is illustrated in the following figures.

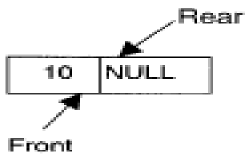


Fig. 5.16. push (10)

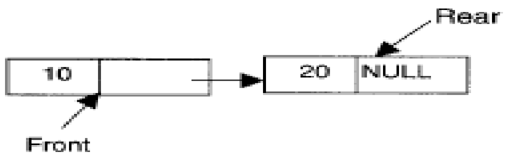


Fig. 5.17. push (20)

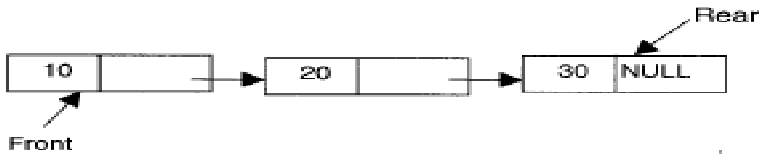


Fig. 5.18. push (30)

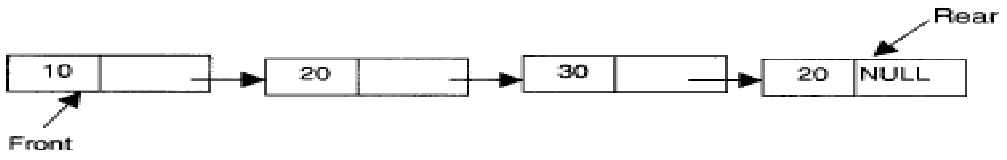


Fig. 5.19. push (40)

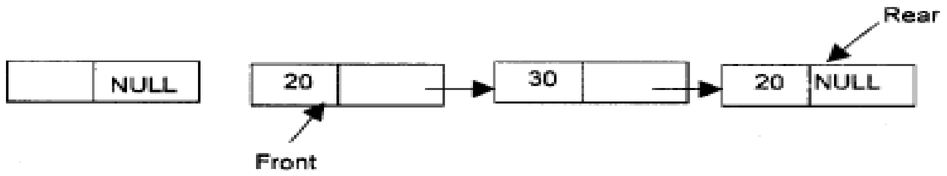


Fig. 5.20. X = pop() (*i.e.*; X = 10)

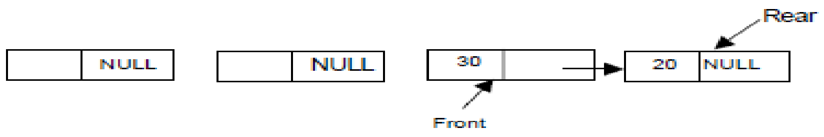


Fig. 5.21. X = pop() (*i.e.*; X = 20)

ALGORITHM FOR PUSHING AN ELEMENT TO A QUEUE

REAR is a pointer in queue where the new elements are added.

FRONT is a pointer, which is pointing to the queue where the elements are popped.

DATA is an element to be pushed.

```
1. Input the DATA element to be pushed
2. Create a New Node
3. NewNode → DATA = DATA
4. NewNode → Next = NULL
5. If(front is equal to NULL and rear is equal to NULL)
    (a) front = rear = NewNode
    (b) exit
6. rear → next = NewNode
7. rear = NewNode
7. Exit
```

ALGORITHM FOR POPPING AN ELEMENT FROM A QUEUE

REAR is a pointer in queue where the new elements are added.

FRONT is a pointer, which is pointing to the queue where the elements are popped.

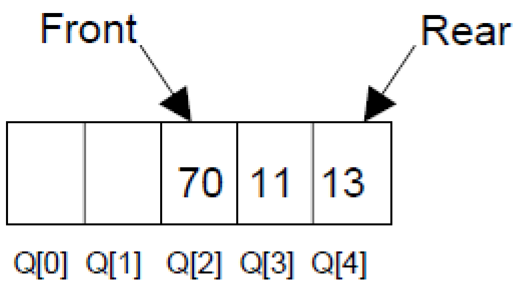
DATA is an element popped from the queue.

```
1. declare temp = FRONT
2. If (FRONT is equal to NULL)
       (a) Display “The Queue is empty”
3. Else if (FRONT is equal to REAR)
       (a) FRONT = REAR = NULL
4. Else
       (a) FRONT = FRONT → next
5. delete temp
6. Exit
```

Lecture-8

CIRCULAR QUEUE

Suppose a queue Q has maximum size 5, say 5 elements pushed and 2 elementspopped.



Now if we attempt to add more elements, even though 2 queue cells are free, the elements cannot be pushed. Because in a queue, elements are always inserted at the rear end and hence rear points to last location of the queue array Q[4]. That is queue is full (overflow condition) though it is empty. This limitation can be overcomeif we use **circular queue**.

In circular queues the elements Q[0],Q[1],Q[2]..... Q[n – 1] is represented in a circular fashion with Q[1] following Q[n]. A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the lastlocation at the queue is full.

Suppose Q is a queue array of 6 elements. Push and pop operation can beperformed on circular. The following figures will illustrate the same.

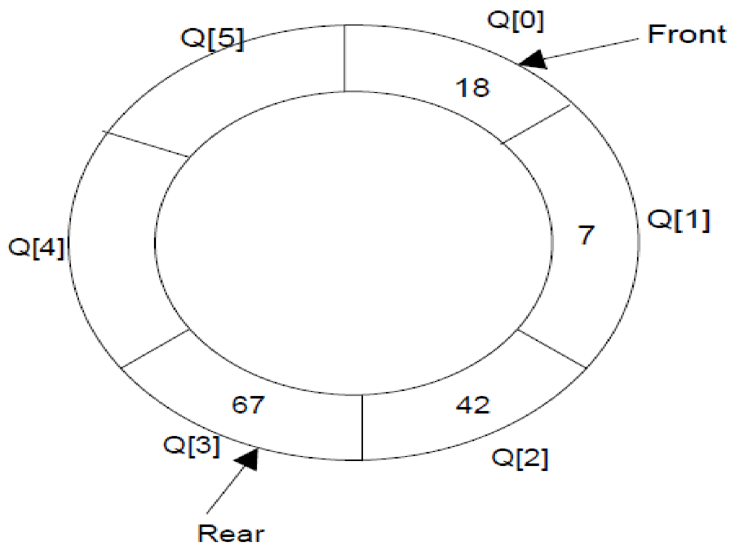


Fig. 4.11. A circular queue after inserting 18, 7, 42, 67.

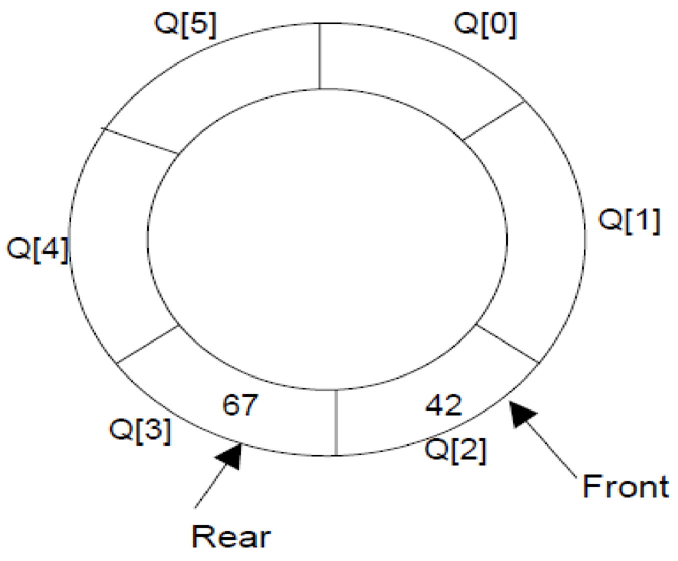


Fig. 4.12. A circular queue after popping 18, 7

After inserting an element at last location Q[5], the next element will be inserted at

the very first location (i.e., Q[0]) **that is circular queue is one in which the firstelement comes just after the last element.**

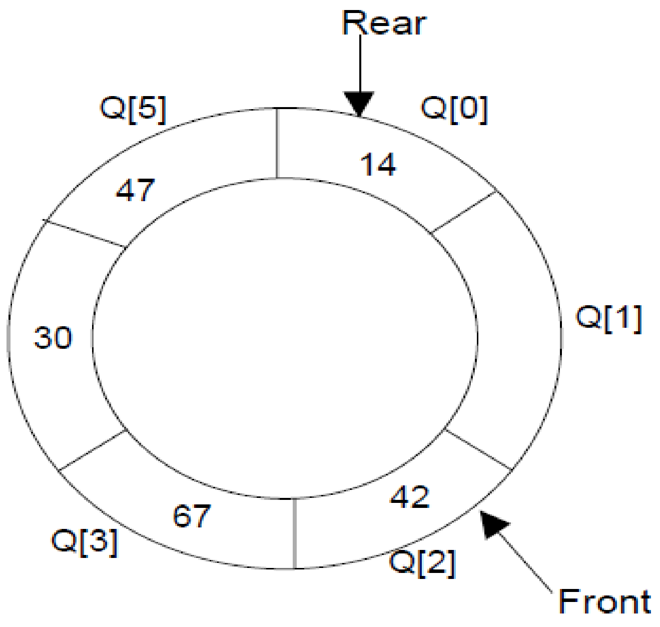
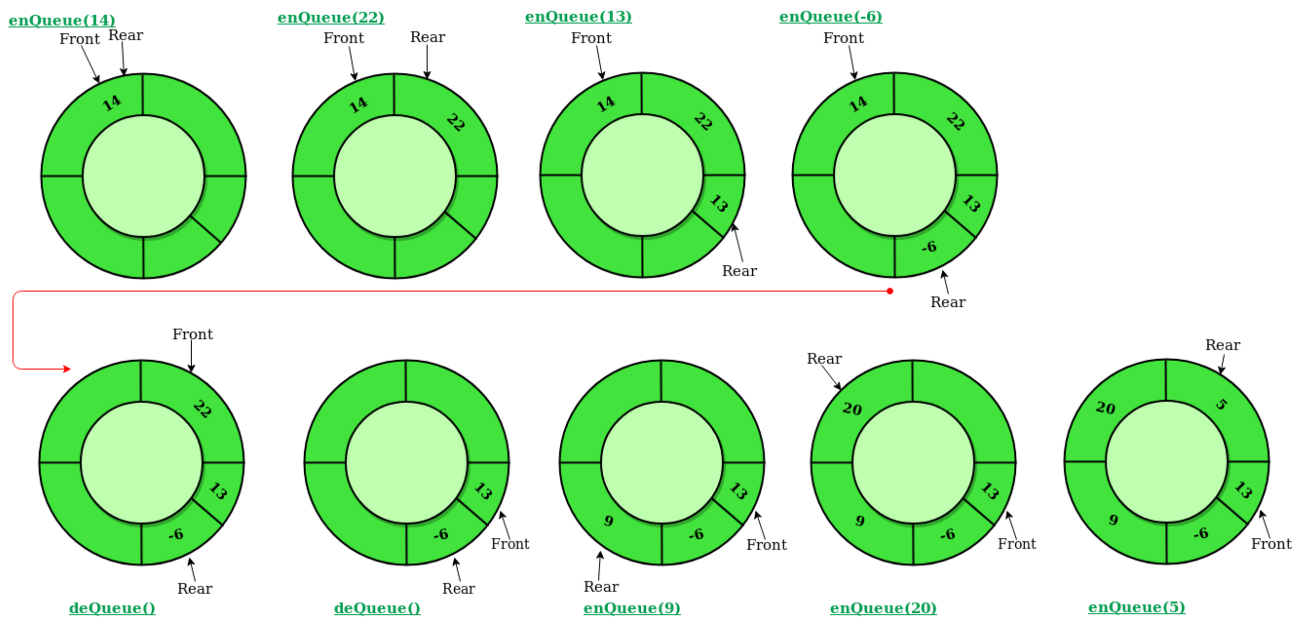


Fig. 4.13. A circular queue after pushing 30, 47, 14



- At any time the position of the element to be inserted will be calculated by the relation

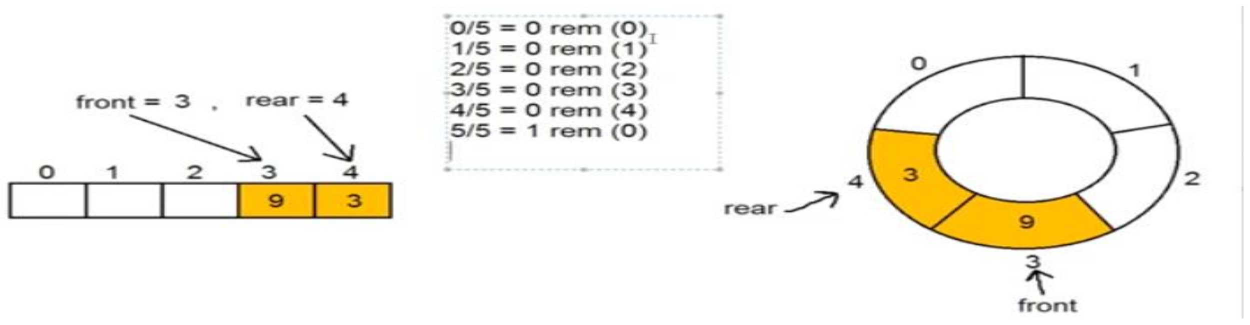
Rear = (Rear + 1) % SIZE

- After deleting an element from circular queue the position of the front end is calculated by the relation

Front= (Front + 1) % SIZE

- After locating the position of the new element to be inserted, rear, compare it withfront.

If (rear = front), the queue is full and cannot be inserted anymore.



CIRCULAR QUEUE ALGORITHMS

Let Q be the array of some specified size say SIZE. FRONT and REAR are two pointers where the elements are deleted and inserted at two ends of the circular queue.DATA is the element to be inserted.

Inserting an element to circular Queue

```
1. If [(rear+1) % Size = = front]
      (a) Display "Queue is full"
      (b) Exit
2. Input the value to be inserted and assign to variable "DATA"
3. If (front is equal to – 1)
      (a) front = 0
4. rear = (rear + 1) % Size
5. Q[rear] = DATA
6. Exit
```

Deleting an element from a circular queue

```
1. If (FRONT is equal to – 1)
      (a) Display "Queue is empty"
      (b) Exit
2. If (REAR is equal to FRONT)
      (a) FRONT = –1
      (b) REAR = –1
3. Else
      (a) DATA = Q[FRONT]
      (b) FRONT = (FRONT +1) % SIZE
4. Exit
```

Algorithm display element of a circular queue

```
display()
{
    int i=front;
    if(front==-1 && rear==-1)
    {
        Cout<<"\n Queue is empty..";
    }
    else
    {
        Cout<<"\nElements in a Queue are :";
        while(i<=rear)
        {
            Cout<<"%d,", queue[i]);
            i=(i+1)%max;
        }
    }
}
```

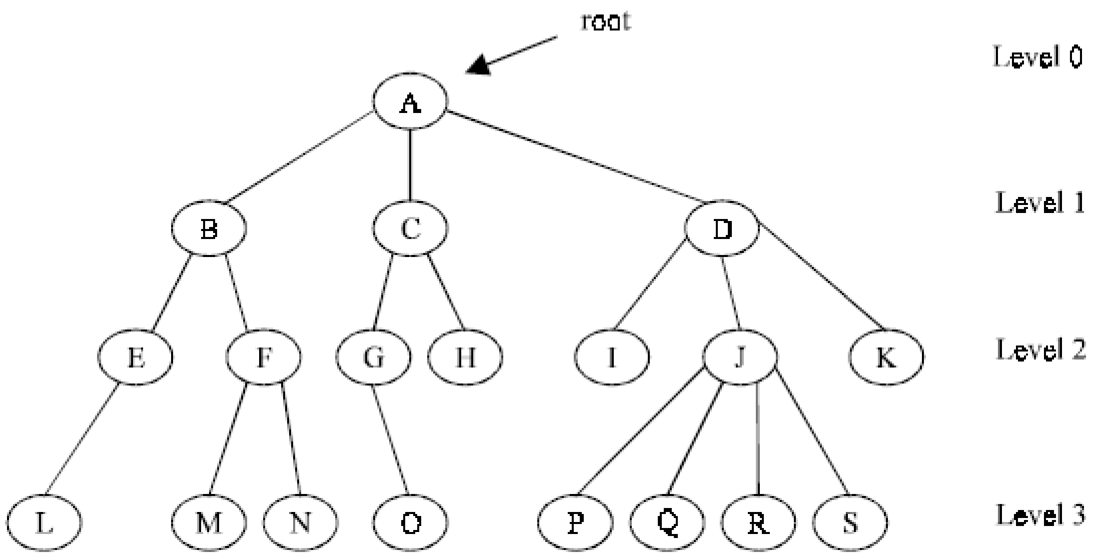
Assignment

- Program to find the odd, even number in circular queue
- Program to find the positive, negative number in circular queue

Non Linear Data Structures

The Trees

One of the important non-liner data structure in computer science is Trees. Many real life problems can be represented and solved using trees.
Trees are very flexible, versatile and powerful non-liner data structure that can be used to represent data items possessing hierarchical relationship between the grand father and his children and grandchildren as so on.



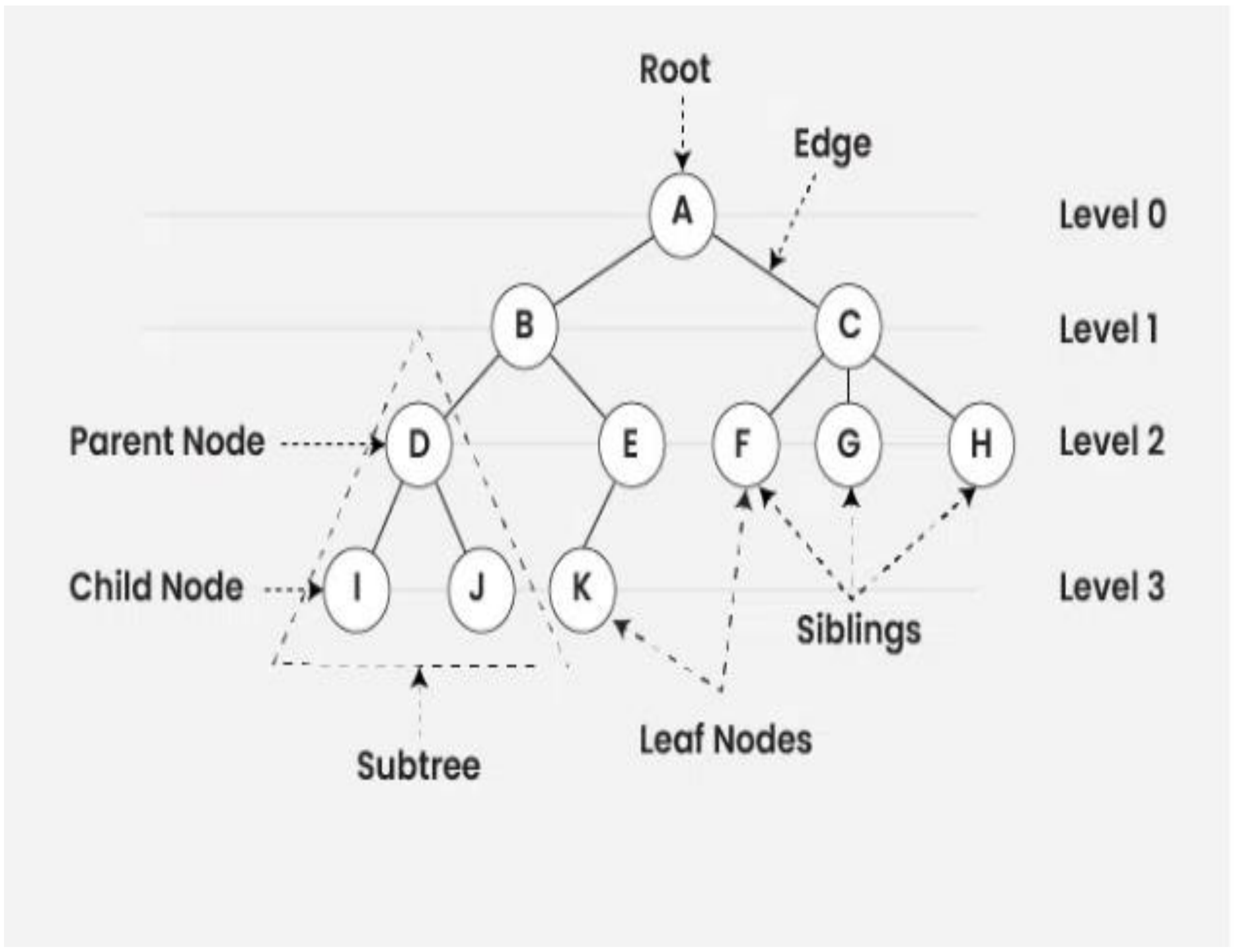
The Tree data structure is similar to Linked Lists in that each node contains data and can be linked to other nodes.

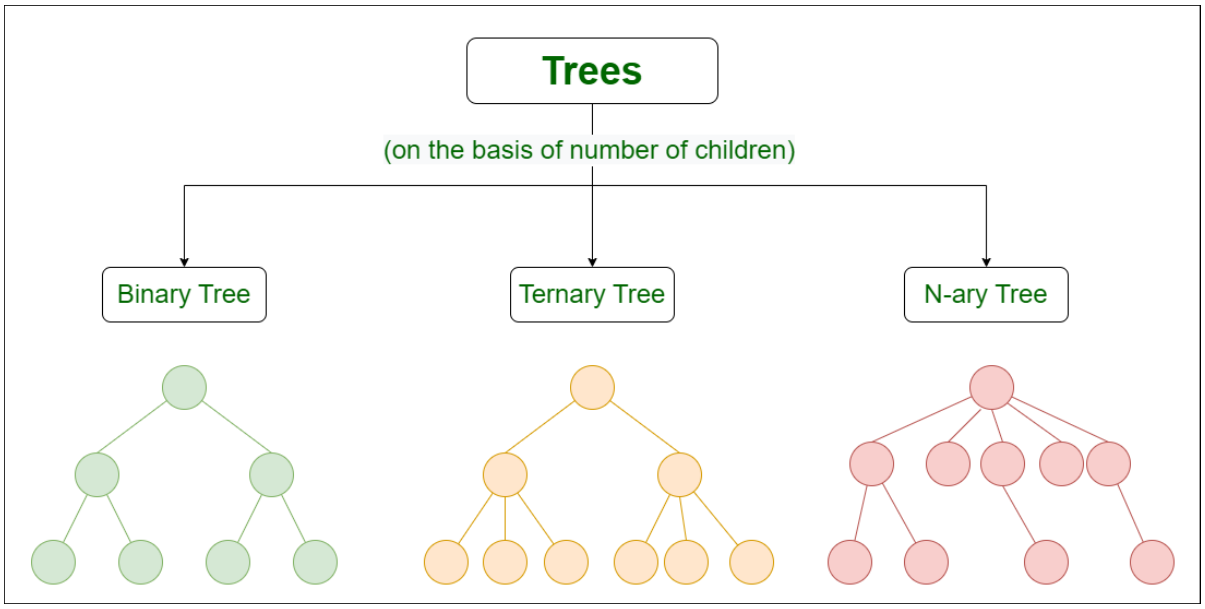
We have previously covered data structures like Arrays, Linked Lists, Stacks, and Queues. These are all linear structures, which means that each element follows directly after another in a sequence. Trees however, are different. In a Tree, a single element can have multiple 'next' elements, allowing the data structure to branch out in various directions.

A tree is an ideal data structure for representing hierarchical data. A tree can be theoretically defined as a finite set of one or more data items (or nodes) such that:

1. There is a special node called the root of the tree.
2. Removing nodes (or data item) are partitioned into number of mutually exclusive (i.e., disjoined) subsets each of which is itself a tree, are called sub tree.

BASIC TERMINOLOGIES





Basic Terminologies In Tree Data Structure:

- **Parent Node:** The node which is an immediate predecessor of a node is called the parent node of that node. **{B}** is the parent node of **{D, E}**.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: **{D, E}** are the child nodes of **{B}.**
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A**}** is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. **{I, J, K, F, G, H}** are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. **{A,B}** are the ancestor nodes of the node **{E}**
- **Sibling:** Children of the same parent node are called siblings. **{D,E}** are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree**: Any node of the tree along with its descendant.

- **Level –** Count nodes in a path to reach a destination node. Example- Level of node D is 2 as nodes A and B form the path.
- **Height –** Number of edges to reach the destination node, Root is at height 0.
- **Degree of a node –** Number of children of a particular parent. Example- Degree of A is 2 and Degree of C is 3. Degree of I is 0.

Types of Trees

Trees are a fundamental data structure in computer science, used to represent hierarchical relationships. This tutorial covers several key types of trees.

Binary Trees: Each node has up to two children, the left child node and the right child node. This structure is the foundation for more complex tree types like Binay Search Trees and AVL Trees.

Binary Search Trees (BSTs): A type of Binary Tree where for each node, the left child node has a lower value, and the right child node has a higher value.

AVL Trees: A type of Binary Search Tree that self-balances so that for every node, the difference in height between the left and right subtrees is at most one. This balance is maintained through rotations when nodes are inserted or deleted.

Binary trees

A binary tree is a tree in which no node can have more than two children. Typically these children are described as “left child” and “right child” of the parent node.

A binary tree T is defined as a finite set of elements, called nodes, such that:

1- T is empty (i.e., if T has no nodes called the null tree or empty tree).

2- T contains a special node R, called the root of T, and the remaining nodes of T form an ordered pair of disjoined binary trees T1 and T2, and they are called left and right sub tree of R. if T1 is non empty then its root is called the left successor of R, similarly if T2 is non empty then its root is called the right successor of R.

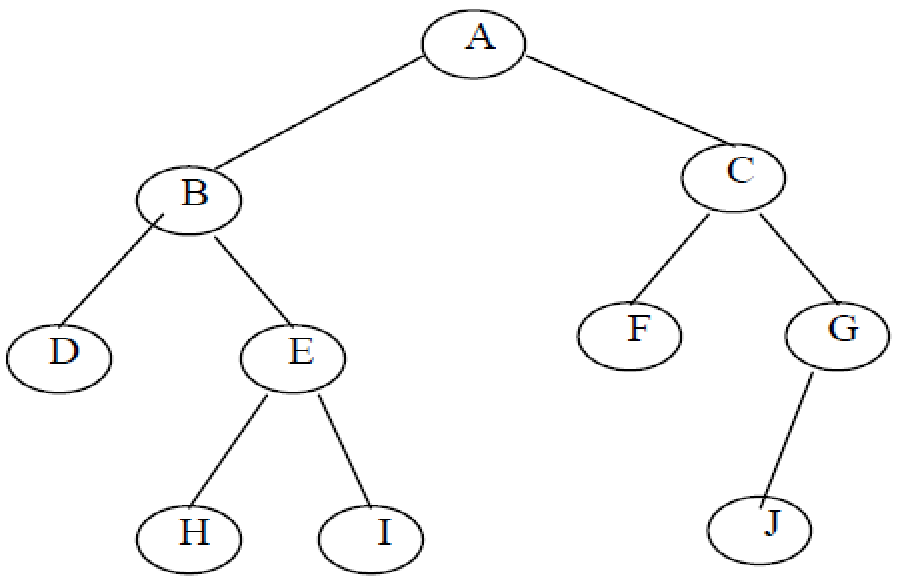


Fig. 8.3. Binary tree

Consider a binary tree T in Fig. 8.3. Here ‘A’ is the root node of the binary tree T. Then ‘B’ is the left child of ‘A’ and ‘C’ is the right child of ‘A’ i.e., ‘A’ is a father of ‘B’ and ‘C’. The node ‘B’ and ‘C’ are called brothers, since they are left and right child of the same father. If a node has no child then it is called a leaf node. Nodes D, H,I,F,J are leaf node in Fig. 8.3.

Types of Binary Trees

Types of Binary Tree On the basis of the completion of levels:

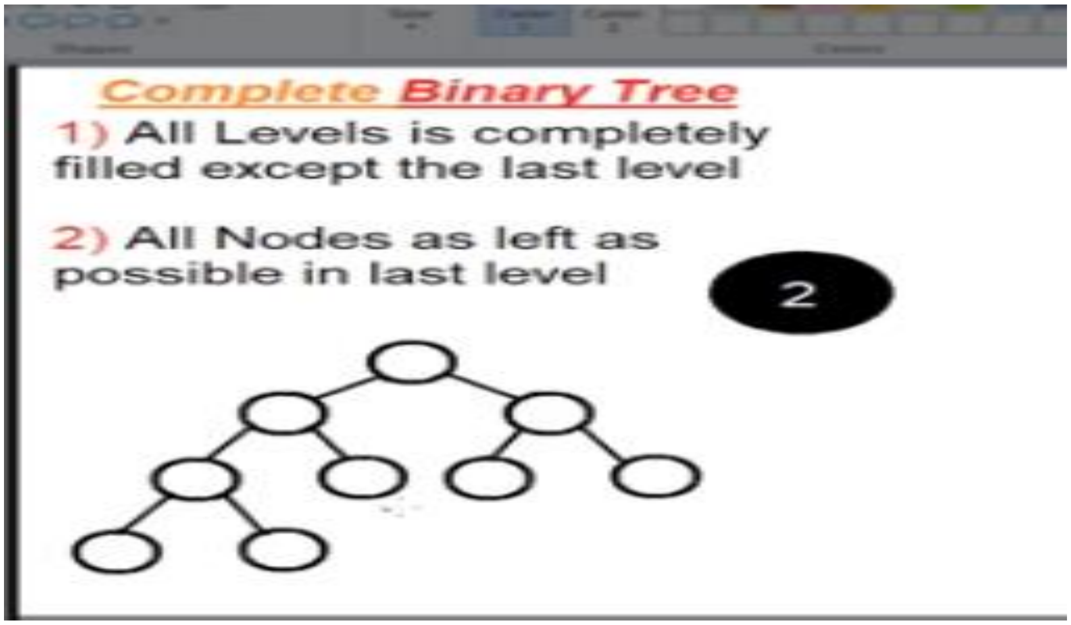
1. Complete Binary Tree
2. Perfect Binary Tree
3. Balanced Binary Tree

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.

A complete binary tree is just like a full binary tree, but with two major differences:

- Every level except the last level must be completely filled.
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.

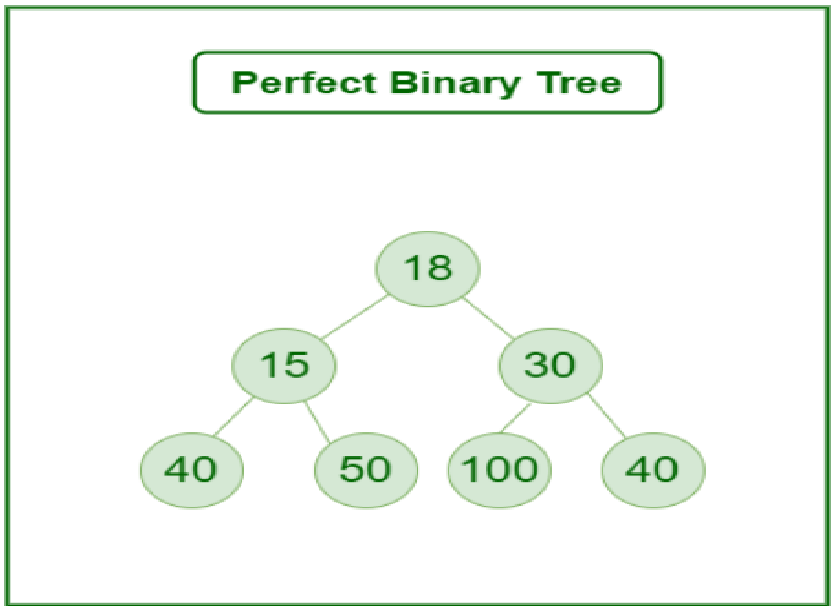

Complete Binary Tree
1) All Levels is completely filled except the last level
2) All Nodes as left as possible in last level
2

Perfect Binary Tree

A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.

The following are examples of Perfect Binary Trees.

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



Perfect Binary Tree

Strictly binary:- The tree is said to be **strictly binary** tree, if every non-leaf made in a binary tree has non-empty left and right sub trees. The tree in Fig. 8.4 is strictly binary tree, whereas the tree in Fig. 8.3 is not. That is every node in the strictly

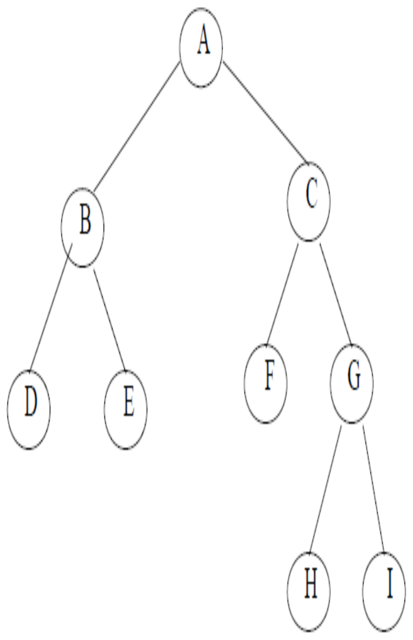
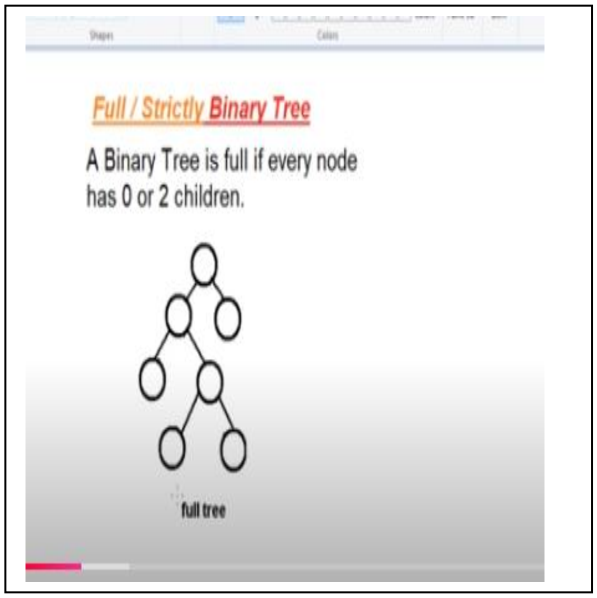
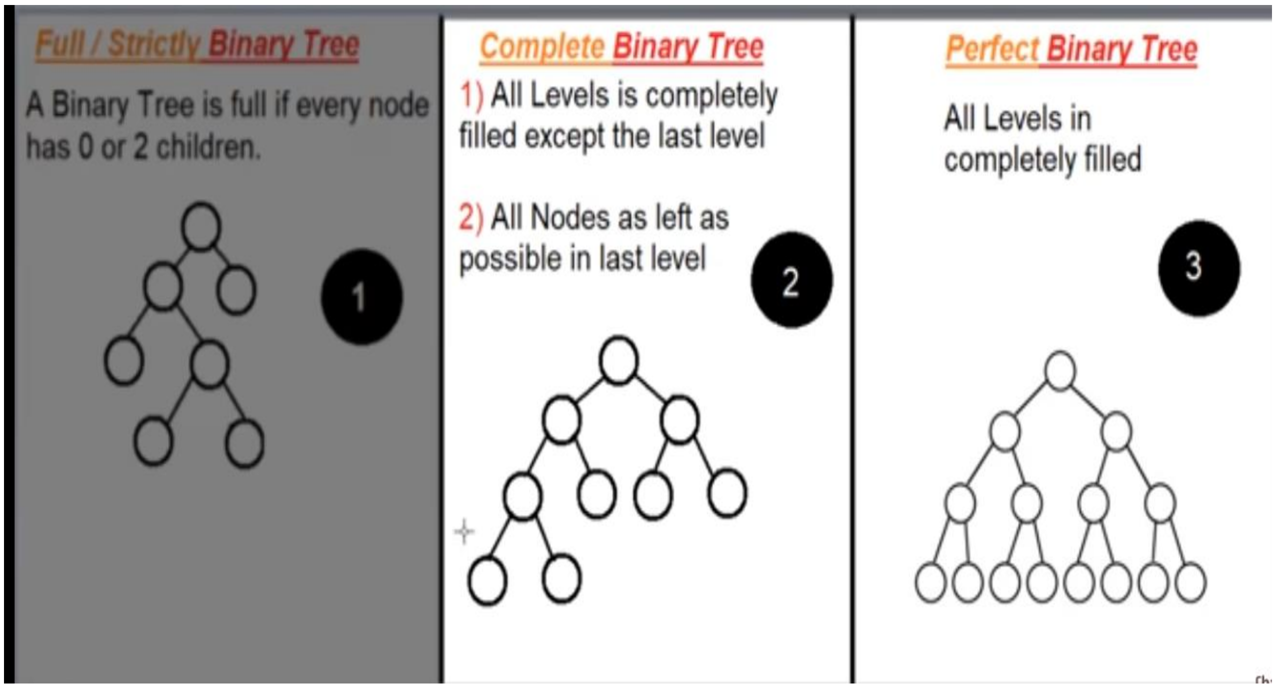


Fig. 8.4. Strictly binary tree



Full / Strictly Binary Tree

A Binary Tree is full if every node has 0 or 2 children.



Full / Strictly Binary Tree

A Binary Tree is full if every node has 0 or 2 children.

Complete Binary Tree

1) All Levels is completely filled except the last level

2) All Nodes as left as possible in last level

Perfect Binary Tree

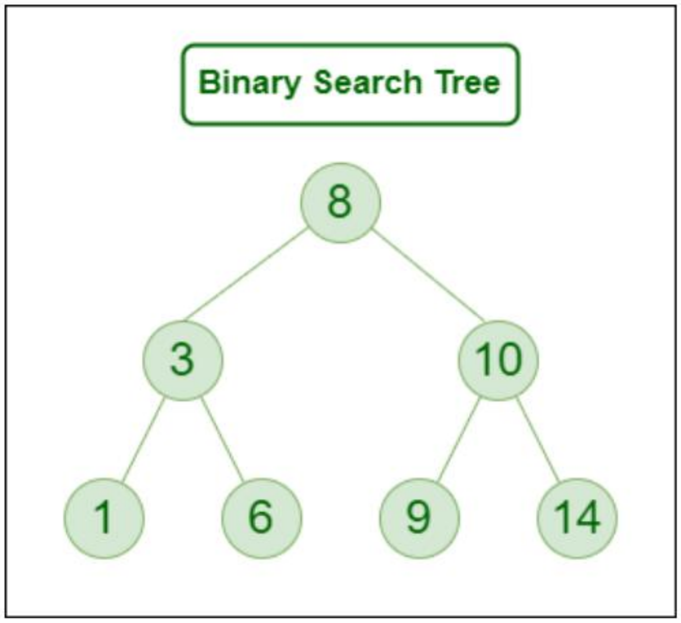
All Levels in completely filled

Special Types of Trees in Data Structure based on the nodes' values:

1. Binary Search Tree

A **binary Search Tree** is a node-based binary tree data structure that has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



Binary Search Tree

The main application of a 2-tree is to represent and compute any algebraic expression using binary operation.

For example, consider an algebraic expression E.

$$E = (a + b) / ((c - d)*e)$$

E can be represented by means of the extended binary tree T as shown in Fig. 8.5. Each variable or constant in E appears as an internal node in T whose left and right sub tree corresponds to the operands of the operation.

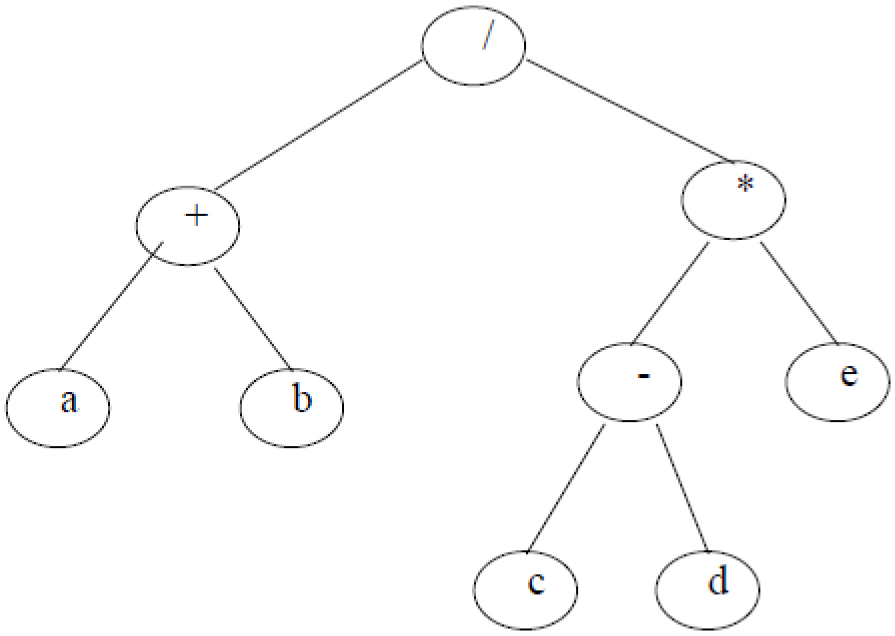


Fig. 8.5. Expression tree

If a binary tree has only left sub trees, then it is called left skewed binary tree. Fig.8.7 (a) is a left skewed binary tree.

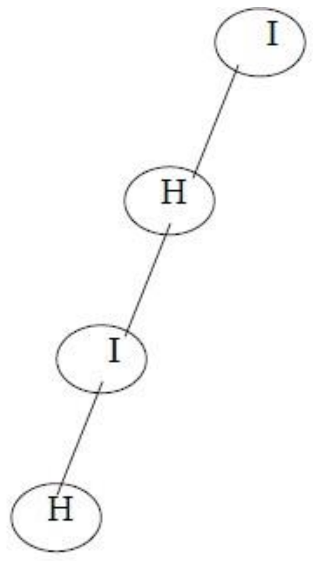


Fig. 8.7(*a*). Left skewed

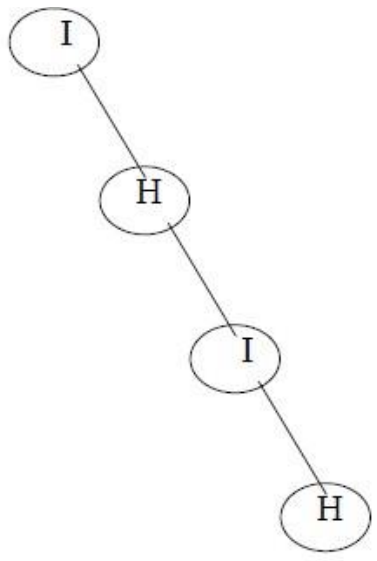


Fig. 8.7(*b*). Right skewed

If a binary tree has only right sub trees, then it is called right skewed binary tree. Fig. 8.7(b) is a right skewed مائل او منحرفbinary tree.

Binary Tree Traversal

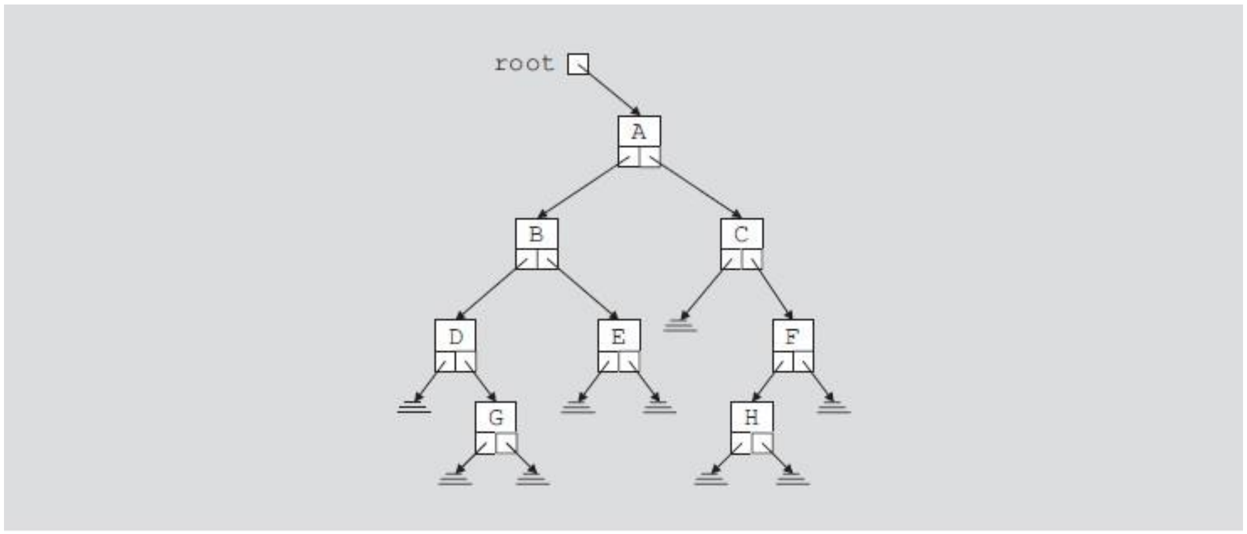


FIGURE 11-4 Binary tree

The item insertion, deletion, and lookup operations require that the binary tree be traversed. Thus, the most common operation performed on a binary tree is to traverse the binary tree, or visit each node of the binary tree. As you can see from the diagram of a binary tree, the traversal must start at the root node because there is a pointer to the root node. For each node,we have two choices:

- Visit the node first.
- Visit the sub trees first.

These choices lead to three different traversals of a binary tree—

Inorder,

preorder

postorder

Inorder Traversal(LVR)

In an inorder traversal, the binary tree is traversed as follows:

1. Traverse the left subtree.
2. Visit the node.
3. Traverse the right subtree.

Preorder Traversal(VLR)

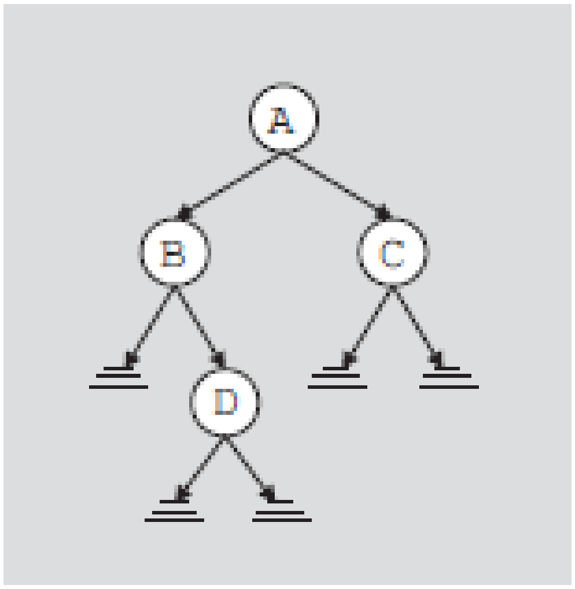
In a preorder traversal, the binary tree is traversed as follows:

1. Visit the node.
2. Traverse the left subtree.
3. Traverse the right subtree.

Postorder Traversal(LRV)

In a postorder traversal, the binary tree is traversed as follows:

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the node.

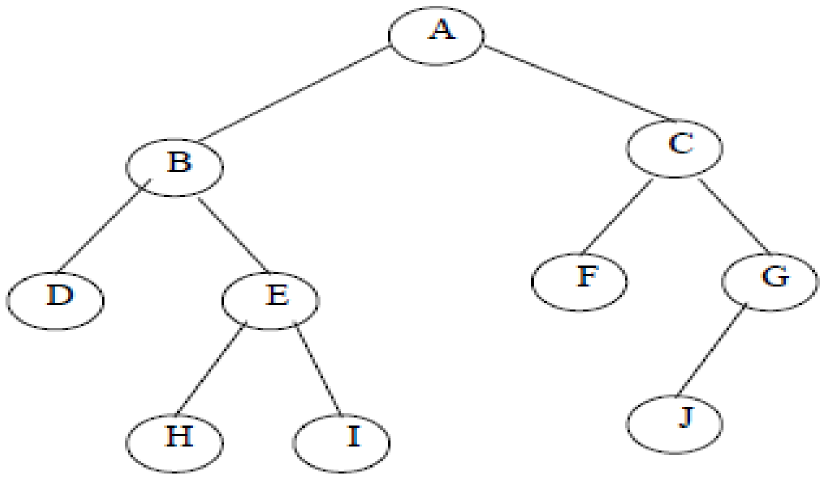


Inorder sequence: *B D A C*

Similarly, the preorder and postorder traversals output the nodes in the following order:

Preorder sequence: *A B D C*

Postorder sequence: *D B C A*



The preorder traversal of a binary tree in Fig. is A, B, D, E, H, I, C, F, G, J.

The in order traversal of a binary tree in Fig. is D, B, H, E, I, A, F, C, J, G.

The post order traversal of a binary tree in Fig. 8.12 is D, H, I, E, B, F, J, G, C, A

Preorder: The node we are on right now is visited first and then we visit left child (as well as the whole left subtree) followed by its right subtree (as well as the whole right subtree). I denote Preorder traversal by VLR : (V)isit the current node, then visit its (L)eft child, and then visit (R)ight child node.

So, for the below tree, let's see what the Preorder traversal would look like this:

29 -> 20 -> 12 -> 7 -> 15 -> 26 -> 31 -> 30

```
       VLR
        29
     /      \
    VLR      VLR
    20       31
   /  \       /
  VLR  VLR  VLR
  12   26   30
 /  \
VLR   VLR
7    15
```

Inorder: The (L)eft childnode and left subtree of the current node is visited first, followed by the (V)isiting the current node and then visit the (R)ight childnode and right subtree. Let's denote Inorder traversal by LVR.

For the tree below, the Inorder traversal would look like:

7 -> 12 -> 15 -> 20 -> 26 -> 29 -> 30 -> 31.

```
       LVR
        29
     /      \
    LVR      LVR
    20       31
   /  \       /
  LVR  LVR  LVR
  12   26   30
 /  \
LVR  LVR
7    15
```

One interesting property of Inorder traversal is that, for a BST it gives the items in sorted order (increasing order).

Postorder: Here the current node is visited at the end after visiting its left subtree followed by its right subtree. I'd denote Postorder traversal as LVR.

Postorder traversal of the below tree would look like:

7 -> 15 -> 12 -> 26 -> 20 -> 30 -> 31 -> 29

```
       LRV
        29
     /      \
    LRV      LRV
    20       31
   /  \       /
  LRV  LRV  LRV
  12   26   30
 /  \
LRV   LRV
7    15
```

https://yongdanielliang.github.io/animation/web/BST.html

Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

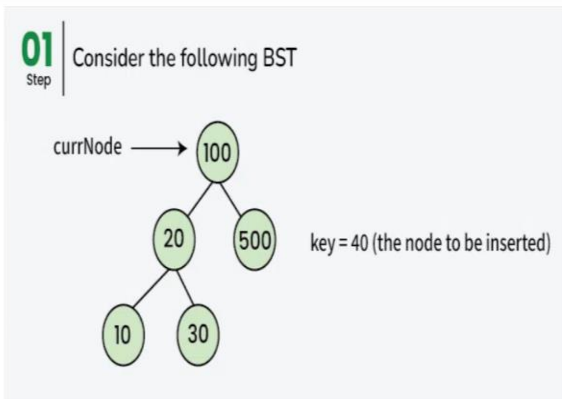
```
1.START
2. Check whether the tree is empty or not
3. If the tree is empty, search is not possible
4. Otherwise, first search the root of the tree.
5. If the key does not match with the value in the root,
   search its subtrees.
6. If the value of the key is less than the root value,
   search the left subtree
7. If the value of the key is greater than the root value,
   search the right subtree.
8. If the key is not found in the tree, return unsuccessful search.
       9. END
```

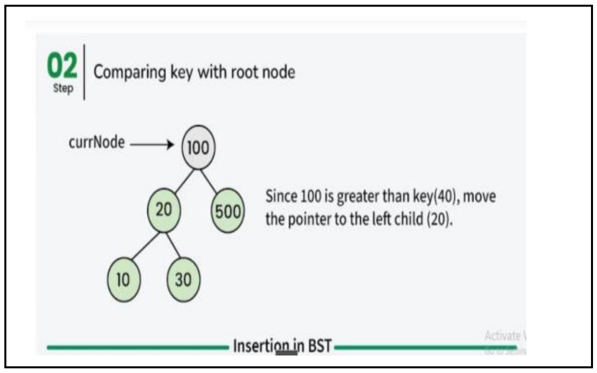
Insertion Operation

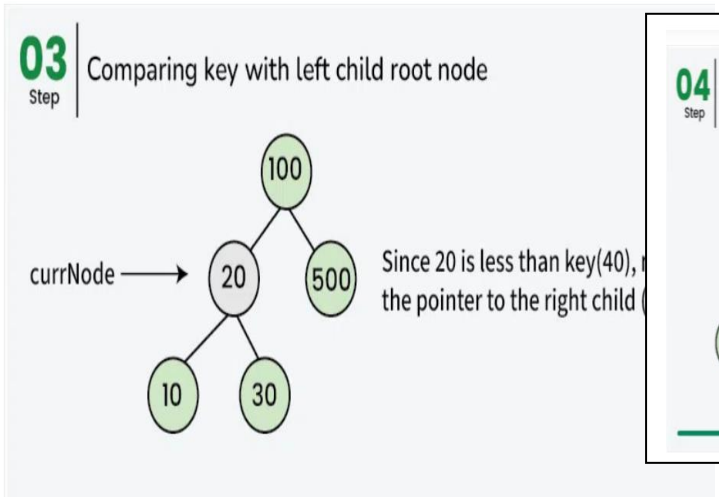
Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

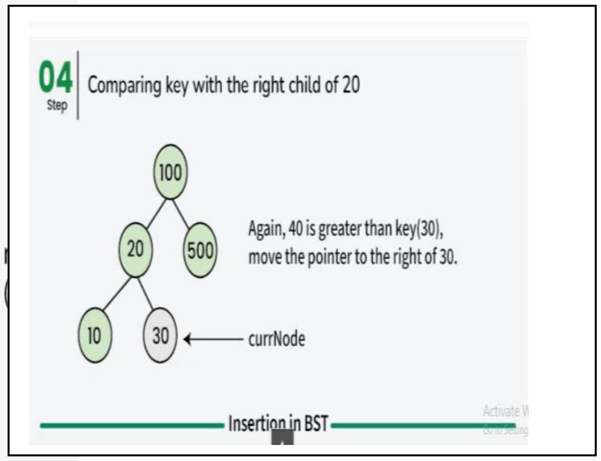
Algorithm

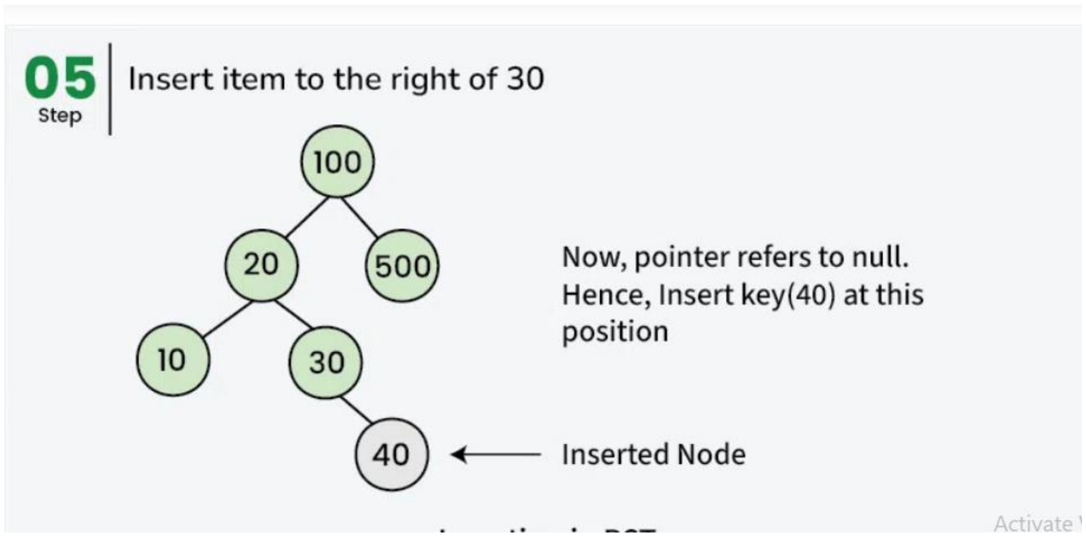
```
1. START
2. If the tree is empty, insert the first element as the root node of the
   tree. The following elements are added as the leaf nodes.
3. If an element is less than the root value, it is added into the left
   subtree as a leaf node.
4. If an element is greater than the root value, it is added into the right
   subtree as a leaf node.
5. The final leaf nodes of the tree point to NULL values as their
   child nodes.
6. END
```


01 Step
Consider the following BST
currNode
100
20
500
10
30
key = 40 (the node to be inserted)


02 Step
Comparing key with root node
currNode
100
20
500
10
30
Since 100 is greater than key(40), move the pointer to the left child (20).
Insertion in BST


03 Step
Comparing key with left child root node
100
currNode
20
500
10
30
Since 20 is less than key(40),
the pointer to the right child


04 Step
Comparing key with the right child of 20
100
20
500
Again, 40 is greater than key(30), move the pointer to the right of 30.
10
30
currNode
Insertion in BST


05 Step
Insert item to the right of 30
100
20
500
10
30
40
Now, pointer refers to null. Hence, Insert key(40) at this position
Inserted Node