## Introduction to Data Structure

- Computer is an electronic machine which is used for data processing and manipulation.
- When programmer collects such type of data for processing, he would require to store all of them in computer's main memory.
- In order to make computer work we need to know
  - Representation of data in computer.
  - Accessing of data.
  - How to solve problem step by step.
- For doing this task we use data structure.

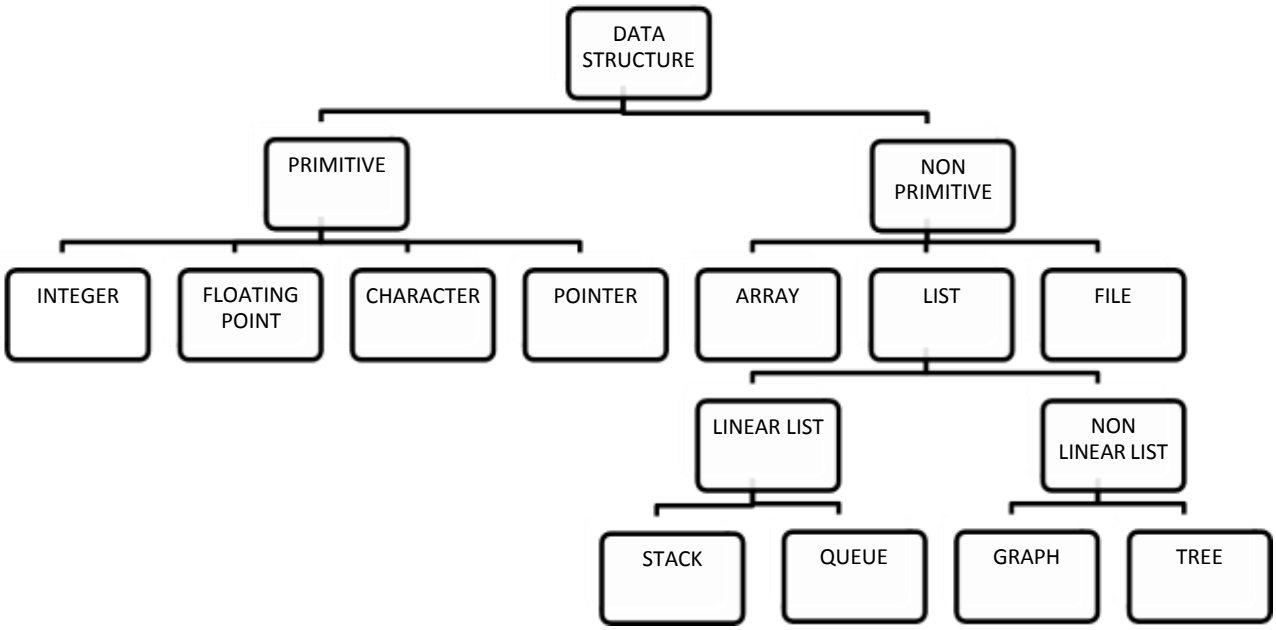
---

## What is Data Structure?

- **Data structure** is a representation of the logical relationship existing between individual elements of data.
- Data Structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.
- We can also define data structure as a mathematical or logical model of a particular organization of data items.
- The representation of particular data structure in the main memory of a computer is called as **storage structure.**
- The storage structure representation in auxiliary memory is called as **file structure**.
- It is defined as the way of storing and manipulating data in organized form so that it can be used efficiently.
- Data Structure mainly specifies the following four things
  - Organization of Data
  - Accessing methods
  - Degree of associativity
  - Processing alternatives for information
- Algorithm + Data Structure = Program
- Data structure study covers the following points
  - Amount of memory require to store.
  - Amount of time require to process.
  - Representation of data in memory.
  - Operations performed on that data.

---

## Classification of Data Structure




Data Structures are normally classified into two broad categories

1. Primitive Data Structure
2. Non-primitive data Structure

## Data types

A particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.

## Primitive Data Structure

- Primitive data structures are basic structures and are directly operated upon by machine instructions.
- Primitive data structures have different representations on different computers.
- Integers, floats, character and pointers are examples of primitive data structures.
- These data types are available in most programming languages as built in type.
  - Integer: It is a data type which allows all values without fraction part. We can use it for whole numbers.
  - Float: It is a data type which use for storing fractional numbers.
  - Character: It is a data type which is used for character values.

Pointer: A variable that holds memory address of another variable are called pointer.

## Non primitive Data Type

- These are more sophisticated data structures.
- These are derived from primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.
- Examples of Non-primitive data type are Array, List, and File etc.
- A Non-primitive data type is further divided into Linear and Non-Linear data structure
  - **Array:** An array is a fixed-size sequenced collection of elements of the same data type.
  - **List:** An ordered set containing variable number of elements is called as Lists.
  - **File:** A file is a collection of logically related information. It can be viewed as a large list of records consisting of various fields.

## Linear data structures

- A data structure is said to be Linear, if its elements are connected in linear fashion by means of logically or in sequence memory locations.
- There are two ways to represent a linear data structure in memory,
  - Static memory allocation
  - Dynamic memory allocation
- The possible operations on the linear data structure are: Traversal, Insertion, Deletion, Searching, Sorting and Merging.
- Examples of Linear Data Structure are Stack and Queue.
- Stack: Stack is a data structure in which insertion and deletion operations are performed at one end only.
  - The insertion operation is referred to as 'PUSH' and deletion operation is referred to as 'POP' operation.
  - Stack is also called as Last in First out (LIFO) data structure.
- Queue: The data structure which permits the insertion at one end and Deletion at another end, known as Queue.
  - End at which deletion is occurs is known as FRONT end and another end at which insertion occurs is known as REAR end.
  - Queue is also called as First in First out (FIFO) data structure.

## Nonlinear data structures

- Nonlinear data structures are those data structure in which data items are not arranged in a sequence.
- Examples of Non-linear Data Structure are Tree and Graph.
- **Tree:** A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement.
  - Trees represent the hierarchical relationship between various elements.
  - Tree consist of nodes connected by edge, the node represented by circle and edge lives connecting to circle.

- **Graph:** Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.
  - A tree can be viewed as restricted graph.
  - Graphs have many types:
    - Un-directed Graph
    - Directed Graph
    - Mixed Graph
    - Multi Graph
    - Simple Graph
    - Null Graph
    - Weighted Graph

---

## Difference between Linear and Non Linear Data Structure

| Linear Data Structure | Non-Linear Data Structure |
|---|---|
| Every item is related to its previous and next time. | Every item is attached with many other items. |
| Data is arranged in linear sequence. | Data is not arranged in sequence. |
| Data items can be traversed in a single run. | Data cannot be traversed in a single run. |
| Eg. Array, Stacks, linked list, queue. | Eg. tree, graph. |
| Implementation is easy. | Implementation is difficult. |

---

## Operation on Data Structures

Design of efficient data structure must take operations to be performed on the data structures into account. The most commonly used operations on data structure are broadly categorized into following types

1. **Create**
   The create operation results in reserving memory for program elements. This can be done by declaration statement. Creation of data structure may take place either during compile-time or run-time. malloc() function of C language is used for creation.

2. **Destroy**
   Destroy operation destroys memory space allocated for specified data structure. free() function of C language is used to destroy data structure.

3. **Selection**
   Selection operation deals with accessing a particular data within a data structure.

4. **Updation**
   It updates or modifies the data in the data structure.

5. **Searching**
   It finds the presence of desired data item in the list of data items, it may also find the locations of all elements that satisfy certain conditions.

6. **Sorting**
   Sorting is a process of arranging all data items in a data structure in a particular order, say for example, either in ascending order or in descending order.

7. **Merging**
   Merging is a process of combining the data items of two different sorted list into a single sorted list.

8. **Splitting**
   Splitting is a process of partitioning single list to multiple list.

9. **Traversal**
   Traversal is a process of visiting each and every node of a list in systematic manner.

---

## Time and space analysis of algorithms

**Algorithm**

- An essential aspect to data structures is algorithms.
- Data structures are implemented using algorithms.
- An algorithm is a procedure that you can write as a C function or program, or any other language.
- An algorithm states explicitly how the data will be manipulated.

**Algorithm Efficiency**

- Some algorithms are more efficient than others. We would prefer to choose an efficient algorithm, so it would be nice to have metrics for comparing algorithm efficiency.
- The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process.
- Usually there are natural units for the domain and range of this function. There are two main complexity measures of the efficiency of an algorithm
- **Time complexity**
  - **Time Complexity** is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.

- "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

- **Space complexity**
  - **Space complexity** is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.
  - We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.
  - We can use bytes, but it's easier to use, say, number of integers used, number of fixed-sized structures, etc. In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit.
  - Space complexity is sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time.

---

## Worst Case Analysis

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched is not present in the array. When x is not present, the search () functions compares it with all the elements of array [] one by one. Therefore, the worst case time complexity of linear search would be.

## Average Case Analysis

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed. So we sum all the cases and divide the sum by (n+1).

## Best Case Analysis

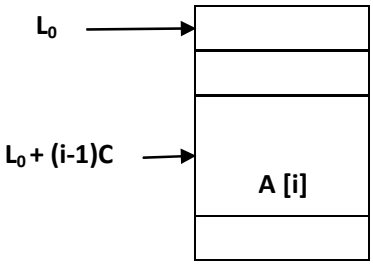
In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in worst case is constant (not dependent on n). So time complexity in the best case would be.

---

# Explain Array in detail

## One Dimensional Array

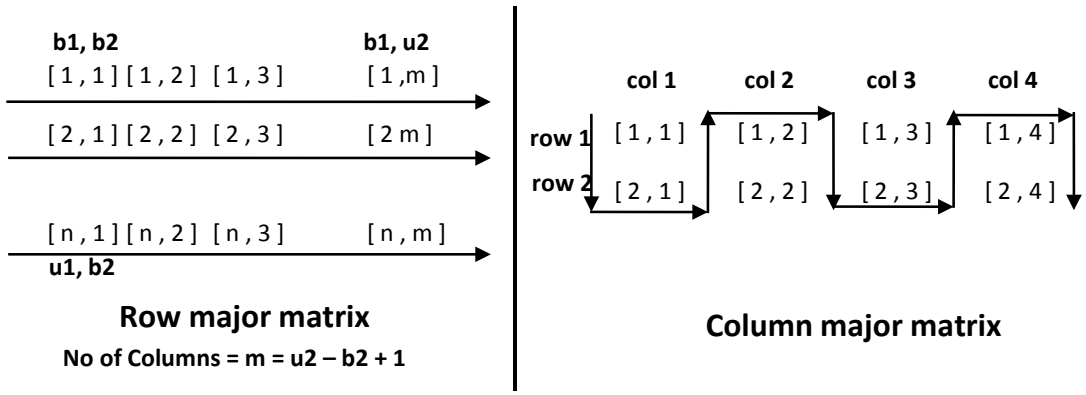
- Simplest data structure that makes use of computed address to locate its elements is the one-dimensional array or vector; number of memory locations is sequentially allocated to the vector.
- A vector size is fixed and therefore requires a fixed number of memory locations.
- Vector A with subscript lower bound of "one" is represented as below….




- $L_0$ is the address of the first word allocated to the first element of vector A.
- C words are allocated for each element or node
- The address of $A_i$ is given equation **Loc ($A_i$) = $L_0$ + C (i-1)**
- Let's consider the more general case of representing a vector A whose lower bound for it's subscript is given by some variable b. The location of Ai is then given by **Loc ($A_i$) = $L_0$ + C (i-b)**

## Two Dimensional Array

- Two dimensional arrays are also called table or matrix, two dimensional arrays have two subscripts
- Two dimensional array in which elements are stored column by column is called as column major matrix
- Two dimensional array in which elements are stored row by row is called as row major matrix
- First subscript denotes number of rows and second subscript denotes the number of columns
- Two dimensional array consisting of two rows and four columns as above Fig is stored sequentially by columns : A [ 1, 1 ], A [ 2 , 1 ], A [ 1 , 2 ], A [ 2 , 2 ], A [ 1 , 3 ], A [ 2 , 3 ], A [ 1, 4 ], A [ 2, 4 ]
- The address of element A [ i , j ] can be obtained by expression **Loc (A [ i , j ]) = $L_0$ + (j-1)*2 + i-1**
- In general for two dimensional array consisting of n rows and m columns the address element A [ i , j ] is given by **Loc (A [ i , j ]) = $L_0$ + (j-1)*n + (i – 1)**
- In row major matrix, array can be generalized to arbitrary lower and upper bound in its subscripts, assume that b1 ≤ I ≤ u1 and b2 ≤ j ≤u2




- For row major matrix : **Loc (A [ i , j ]) = $L_0$ + ( i – b1 ) *(u2-b2+1) + (j-b2)**

## Applications of Array

1. Symbol Manipulation (matrix representation of polynomial equation)
2. Sparse Matrix

## Symbol Manipulation using Array

- We can use array for different kind of operations in polynomial equation such as addition, subtraction, division, differentiation etc…
- We are interested in finding suitable representation for polynomial so that different operations like addition, subtraction etc… can be performed in efficient manner
- Array can be used to represent Polynomial equation

- **Matrix Representation of Polynomial equation**

| | **Y** | **$Y^2$** | **$Y^3$** | **$Y^4$** |
|---|---|---|---|---|
| **X** | $X\ Y$ | $X\ Y^2$ | $X\ Y^3$ | $X\ Y^4$ |
| **$X^2$** | $X^2\ Y$ | $X^2\ Y^2$ | $X^2\ Y^3$ | $X^2\ Y^4$ |
| **$X^3$** | $X^3\ Y$ | $X^3\ Y^2$ | $X^3\ Y^3$ | $X^3\ Y^4$ |
| **$X^4$** | $X^4\ Y$ | $X^4\ Y^2$ | $X^4\ Y^3$ | $X^4\ Y^4$ |

e.g. **$2x^2+5xy+Y^2$**
is represented in matrix form as below

| | | **Y** | **$Y^2$** | **$Y^3$** | **$Y^4$** |
|---|---|---|---|---|---|
| | 0 | 0 | **1** | 0 | 0 |
| **X** | 0 | **5** | 0 | 0 | 0 |
| **$X^2$** | **2** | 0 | 0 | 0 | 0 |
| **$X^3$** | 0 | 0 | 0 | 0 | 0 |
| **$X^4$** | 0 | 0 | 0 | 0 | 0 |

e.g. **$x^2+3xy+Y^2+Y-X$**
is represented in matrix form as below

| | | **Y** | **$Y^2$** | **$Y^3$** | **$Y^4$** |
|---|---|---|---|---|---|
| | 0 | 0 | **1** | 0 | 0 |
| **X** | **-1** | **3** | 0 | 0 | 0 |
| **$X^2$** | **1** | 0 | 0 | 0 | 0 |
| **$X^3$** | 0 | 0 | 0 | 0 | 0 |
| **$X^4$** | 0 | 0 | 0 | 0 | 0 |

- Once we have algorithm for converting the polynomial equation to an array representation and another algorithm for converting array to polynomial equation, then different operations in array (matrix) will be corresponding operations of polynomial equation

## What is sparse matrix? Explain

- An mXn matrix is said to be sparse if "many" of its elements are zero.
- A matrix that is not sparse is called a dense matrix.
- We can device a simple representation scheme whose space requirement equals the size of the non-zero elements.

- **Example:-**
  - The non-zero entries of a sparse matrix may be mapped into a linear list in row-major order.
  - For example the non-zero entries of 4X8 matrix of below fig.(a) in row major order are 2, 1, 6, 7, 3, 9, 8, 4, 5

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 2 | 0 | 0 | 1 | 0 |
| 0 | 6 | 0 | 0 | 7 | 0 | 0 | 3 |
| 0 | 0 | 0 | 9 | 0 | 8 | 0 | 0 |
| 0 | 4 | 5 | 0 | 0 | 0 | 0 | 0 |

Fig (a) 4 x 8 matrix

| **Terms** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
|---|---|---|---|---|---|---|---|---|---|
| **Row** | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 4 | 4 |
| **Column** | 4 | 7 | 2 | 5 | 8 | 4 | 6 | 2 | 3 |
| **Value** | 2 | 1 | 6 | 7 | 3 | 9 | 8 | 4 | 5 |

Fig (b) Linear Representation of above matrix

- To construct matrix structure we need to record
  (a) Original row and columns of each non zero entries
  (b) No of rows and columns in the matrix
- So each element of the array into which the sparse matrix is mapped need to have three fields: row, column and value
- A corresponding amount of time is saved creating the linear list representation over initialization of two dimension array.

A =

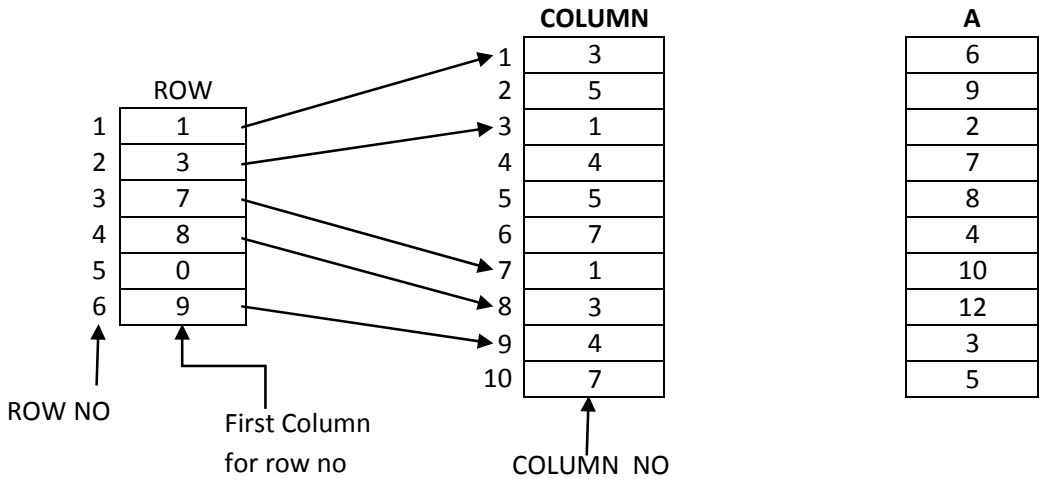
| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 6 | 0 | 9 | 0 | 0 |
| 2 | 0 | 0 | 7 | 8 | 0 | 4 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 12 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 3 | 0 | 0 | 5 |

- Here from 6X7=42 elements, only 10 are non zero. A[1,3]=6, A[1,5]=9, A[2,1]=2, A[2,4]=7, A[2,5]=8, A[2,7]=4, A[3,1]=10, A[4,3]=12, A[6,4]=3, A[6,7]=5.
- One basic method for storing such a sparse matrix is to store non-zero elements in one dimensional array and to identify each array elements with row and column indices fig (c).

| | **ROW** | **COLUMN** | **A** |
|---|---|---|---|
| **1** | 1 | 3 | 6 |
| **2** | 1 | 5 | 9 |

| | | | |
|---|---|---|---|
| **3** | 2 | 1 | 2 |
| **4** | 2 | 4 | 7 |
| **5** | 2 | 5 | 8 |
| **6** | 2 | 7 | 4 |
| **7** | 3 | 1 | 10 |
| **8** | 4 | 3 | 12 |
| **9** | 6 | 4 | 3 |
| **10** | 6 | 7 | 5 |

**Fig (c )**

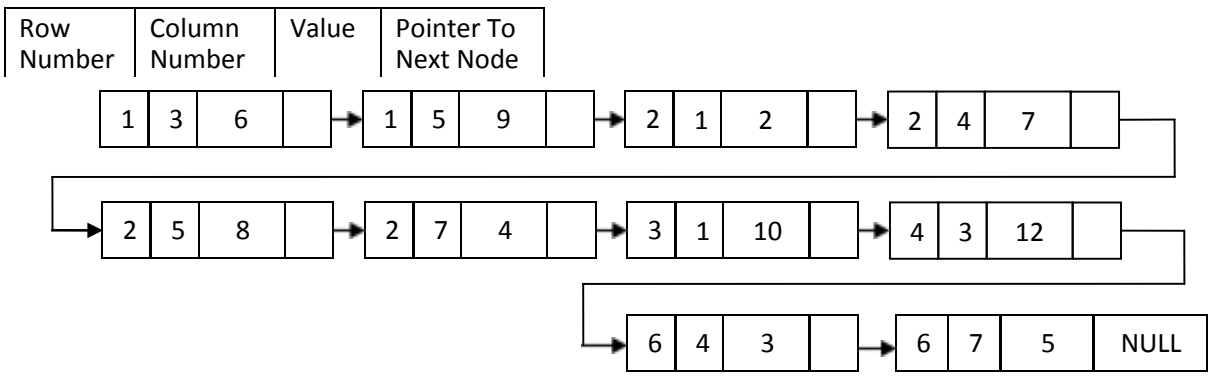



**Fig(d)**

- A more efficient representation in terms of storage requirement and access time to the row of the matrix is shown in fid (d). The row vector changed so that its $i^{th}$ element is the index to the first of the column indices for the element in row I of the matrix.

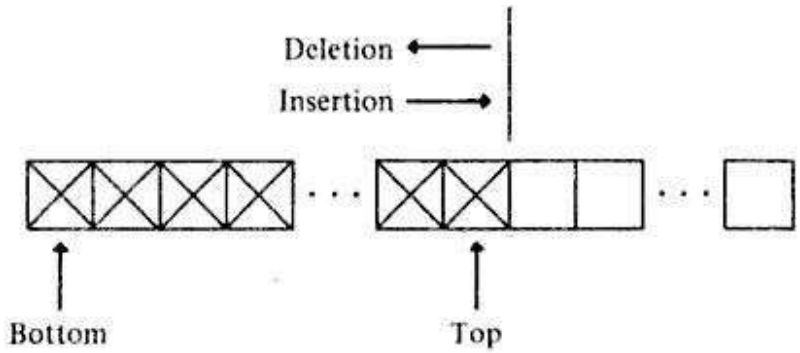
## Linked Representation of Sparse matrix

Typical node to represent non-zero element is



## Write algorithms for Stack Operations – PUSH, POP, PEEP

- A linear list which allows insertion and deletion of an element at one end only is called stack.
- The insertion operation is called as **PUSH** and deletion operation as **POP**.
- The most and least accessible elements in stack are known as top and bottom of the stack respectively.
- Since insertion and deletion operations are performed at one end of a stack, the elements can only be removed in the opposite orders from that in which they were added to the stack; such a linear list is referred to as a LIFO (last in first out) list.




Alternative representation of a stack

- A pointer TOP keeps track of the top element in the stack. Initially, when the stack is empty, TOP has a value of "one" and so on.
- Each time a new element is inserted in the stack, the pointer is incremented by "one" before, the element is placed on the stack. The pointer is decremented by "one" each time a deletion is made from the stack.

## Applications of Stack

- Recursion
- Keeping track of function calls
- Evaluation of expressions
- Reversing characters
- Servicing hardware interrupts
- Solving combinatorial problems using backtracking.

### Procedure : PUSH (S, TOP, X)

- This procedure inserts an element x to the top of a stack which is represented by a vector S containing N elements with a pointer TOP denoting the top element in the stack.

1. [Check for stack overflow]
   If TOP ≥ N
   Then write ('STACK OVERFLOW')
   Return
2. [Increment TOP]
   TOP ←TOP + 1
3. [Insert Element]
   S[TOP] ←X
4. [Finished]
   Return

**Function : POP (S, TOP)**

- This function removes the top element from a stack which is represented by a vector S and returns this element. TOP is a pointer to the top element of the stack.

1. [Check for underflow of stack]
   If TOP = 0
   Then Write ('STACK UNDERFLOW ON POP')
   Take action in response to underflow
   Return
2. [Decrement Pointer]
   TOP ← TOP – 1
3. [Return former top element of stack]
   Return (S[TOP + 1])

## Function : PEEP (S, TOP, I)

- This function returns the value of the $i^{th}$ element from the TOP of the stack which is represented by a vector S containing N elements. The element is not deleted by this function.

1. [Check for stack Underflow]
   If TOP - I +1 ≤ 0
   Then Write ('STACK UNDERFLOW ON PEEP')
   Take action in response to Underflow
   Exit
2. [Return $I^{th}$ element from top of the stack
   Return (S[TOP – I + 1])

## Write an algorithm to change the $i^{th}$ value of stack to value X

**PROCEDURE : CHANGE (S, TOP, X, I)**

- This procedure changes the value of the $I^{th}$ element from the top of the stack to the value containing in X. Stack is represented by a vector S containing N elements.

**1.** [Check for stack Underflow]
   If TOP – I + 1 ≤ 0
   Then Write ('STACK UNDERFLOW ON CHANGE')
   Return

**2.** [Change $I^{th}$ element from top of the stack]
   S[TOP – I + 1] ← X

**3.** [Finished]
   Return

## Write an algorithm which will check that the given string belongs to following grammar or not. L={wcw$^R$ | w Є {a,b}*}(Where w$^R$ is the reverse of w)

**Algorithm : RECOGNIZE**

- Given an input string named STRING on the alphabet {a, b, c} which contains a blank in its rightmost character position and function NEXTCHAR which returns the next symbol in STRING, this algorithm determines whether the contents of STRING belong to the above language. The vector S represents the stack, and TOP is a pointer to the top element of the stack.

**1.** [Initialize stack by placing a letter 'c' on the top]
TOP ← 1
S [TOP] ← 'c'

**2.** [Get and stack symbols either 'c' or blank is encountered]
NEXT ← NEXTCHAR (STRING)
Repeat while NEXT ≠ 'c'
If NEXT = ' '
Then Write ('Invalid String')
Exit
Else Call PUSH (S, TOP, NEXT)
NEXT ← NEXTCHAR (STRING)

**3.** [Scan characters following 'c'; Compare them to the characters on stack]
Repeat While S [TOP] ≠ 'c'
NEXT ← NEXTCHAR (STRING)
X ← POP (S, TOP)
If NEXT ≠ X
Then Write ('INVALID STRING')
Exit

**4.** [Next symbol must be blank]
If NEXT ≠ ' '
Then Write ('VALID STRING')
Else Write ('INVALID STRING')

**5.** [Finished]
Exit

## Write an algorithm for push, pop and empty operations on stack. Using above functions write an algorithm to determine if an input character string is of the form $a^ib^i$ where i>=1 i.e. no of a should be equal to no of b

**Algorithm RECOGNIZE**

- Given an input string named STRING on alphabet 'a' and 'b' which contain blank (' ') on right most character function NEXTCHAR which returns the next symbol from STRING. This algorithm determines if an input string is of form $a^ib^i$ where i>1 i.e no of 'a' should be equal to no of 'b'. the vector S represent the stack and TOP is the pointer to the top element of stack. Counter is a counter B for 'b' occurrence.

**1.** [Initialize stack and counter]
TOP ← 0
COUNTER_B ← 0

**2.** [Get and stack character 'a' from whole string and count the occurrence of 'b' ]
NEXT ← NEXTCHAR(STRING)
Repeat while NEXT != ' '
IF NEXT = 'a'
Then PUSH (S,TOP,NEXT)
Else COUNTER_B ← COUNTER_B+1
NEXT ←NEXTCHAR(STRING)

**3.** [Pop the stack until empty and decrement the COUNTER_B]
Repeat while TOP != 0
POP (S,TOP)
COUNTER_B← COUNTER_B-1

**4.** [Check for grammar]
If COUNTER_B != 0
Then write ('INVALID STRING')
Else write ('VALID STRING')

---

## What is recursion? Write a C program for GCD using recursion.

- A procedure that contains a procedure call to itself or a procedure call to second procedure which eventually causes the first procedure to be called is known as recursive procedure.
- There are two important conditions that must be satisfied by any recursive procedure
  a. Each time a procedure calls itself it must be nearer in some sense to a solution
  b. There must be a decision criterion for stopping the process or computation
- There are two types of recursion
  - Primitive Recursion: this is recursive defined function. E.g. Factorial function
  - Non-Primitive Recursion: this is recursive use of procedure. E.g. Find GCD of given two nunbers

***C program for GCD using recursion***

```
#include<stdio.h>
int Find_GCD(int, int);

void main()
{
      int n1, n2, gcd;
      scanf("%d %d",&n1, &n2);
      gcd = Find_GCD(n1, &n2);
      printf("GCD of %d and %d is %d", n1, n2, gcd);
}

int Find_GCD(int m, int n)
{
      int gcdVal;
      if(n>m)
      {
            gcdVal = Find_GCD(n,m);
      }
      else if(n==0)
      {
            gcdVal = m;
      }
      else
      {
            gcdVal = Find_GCD(n, m%n);
      }
      return(gcdVal);
}
```

---

## Write an algorithm to find factorial of given no using recursion

**Algorithm: FACTORIAL**

Given integer N, this algorithm computes factorial of N. Stack A is used to store an activation record associated with each recursive call. Each activation record contains the current value of N and the current return address RET_ADDE. TEMP_REC is also a record which contains two variables PARAM & ADDRESS.TOP is a pointer to the top element of stack A. Initially return address is set to the main calling address. PARAM is set to initial value N.

1. [Save N and return Address]
   CALL PUSH (A, TOP, TEMP_REC)
2. [Is the base criterion found?]
   If N=0
   then FACTORIAL← 1
   GO TO Step 4
   Else PARAM← N-1
   ADDRESS← Step 3
   GO TO Step 1
3. [Calculate N!]
   FACTORIAL← N * FACTORIAL
4. [Restore previous N and return address]
   TEMP_REC←POP(A,TOP)
   (i.e. PARAM←N, ADDRESS←RET_ADDR)
   GO TO ADDRESS

---

## Give difference between recursion and iteration

| Iteration | Recursion |
|---|---|
| In iteration, a problem is converted into a train of steps that are finished one at a time, one after another | Recursion is like piling all of those steps on top of each other and then quashing them all into the solution. |
| With iteration, each step clearly leads onto the next, like stepping stones across a river | In recursion, each step replicates itself at a smaller scale, so that all of them combined together eventually solve the problem. |
| Any iterative problem is solved recursively | Not all recursive problem can solved by iteration |
| It does not use Stack | It uses Stack |

---

## Write an algorithm to convert infix expression to postfix expression.

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

**Algorithm : REVPOL**

- Given an input string INFIX containing an infix expression which has been padded on the right with ')' and whose symbol have precedence value given by above table, a vector S used as a stack and a NEXTCHAR which when invoked returns the next character of its argument. This algorithm converts INFIX into reverse polish and places the result in the string POLISH. The integer variable TOP denotes the top of the stack. Algorithm PUSH and POP are used for stack manipulation. The integer variable RANK accumulates the rank of expression. Finally the string variable TEMP is used for temporary storage purpose.

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

## Trace the conversion of infix to postfix form in tabular form.

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

| Input Symbol | Content of stack | Reverse polish | Rank |
|---|---|---|---|
| | ( | | 0 |
| ( | ( ( | | 0 |
| A | ( ( | | 0 |
| + | ( ( + | A | 1 |
| B | ( ( + B | A | 1 |
| * | ( ( + * | A B | 2 |
| C | ( ( + * C | A B | 2 |
| / | ( ( + / | A B C * | 2 |
| D | ( ( + / D | A B C * | 2 |
| - | ( ( - | A B C * D / + | 1 |
| E | ( ( - E | A B C * D / + | 1 |
| + | ( ( + | A B C * D / + E - | 1 |
| F | ( ( + F | A B C * D / + E - | 1 |
| / | ( ( + / | A B C * D / + E – F | 2 |
| G | ( ( + / G | A B C * D / + E – F | 2 |
| / | ( ( + / | A B C * D / + E – F G / | 2 |
| ( | ( ( + / ( | A B C * D / + E – F G / | 2 |
| H | ( ( + / ( H | A B C * D / + E – F G / | 2 |
| + | ( ( + / ( + | A B C * D / + E – F G / H | 3 |
| I | ( ( + / ( + I | A B C * D / + E – F G / H | 3 |
| ) | ( ( + / | A B C * D / + E – F G / H I + | 3 |
| ) | ( | A B C * D / + E – F G / H I + / + | 1 |
| ) | | A B C * D / + E – F G / H I + / + | 1 |

Postfix expression is: **A B C * D / + E – F G / H I + / +**

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

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

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

## Convert the following string into prefix: A-B/(C*D^E)

**Step-1 : reverse infix expression**

) E ^ ) D * C ( ( / B - A

**Step-2 : convert '(' to ')' and ')' to '(' and append extra ')' at last**

( E ^ ( D * C ) ) / B - A

**Step-3 : Now convert this string to postfix**

| Input Symbol | Content of stack | Reverse polish | Rank |
|---|---|---|---|
| | ( | | 0 |
| ( | ( ( | | 0 |
| E | ( ( E | | 0 |
| ^ | ( ( ^ | E | 1 |
| ( | ( ( ^ ( | E | 1 |
| D | ( ( ^ ( D | E | 1 |
| * | ( ( ^ ( * | E D | 2 |
| C | ( ( ^ ( * C | E D | 2 |
| ) | ( ( ^ | E D C * | 2 |
| ) | ( | E D C * ^ | 1 |
| / | ( / | E D C * ^ | 1 |
| B | ( / B | E D C * ^ | 1 |
| - | ( - | E D C * ^ B / | 1 |
| A | ( - A | E D C * ^ B / | 1 |
| ) | | E D C * ^ B / A - | 1 |

**Step 4 : Reverse this postfix expression**

**- A / B ^ * C D E**

## Translate the following string into Polish notation and trace the content of stack: (a + b ^ c ^ d) * ( e + f / d )

**Step-1 : reverse infix expression**

) d / f + e ( * ) d ^ c ^ b + a (

**Step-2 : convert '(' to ')' and ')' to '(' and append extra ')' at last**

( d / f + e ) * ( d ^ c ^ b + a ) )

**Step-3 : Now convert this string to postfix**

| Input symbol | Content of stack | Reverse polish | Rank |
|---|---|---|---|
| | ( | | 0 |
| ( | ( ( | | 0 |
| d | ( ( d | | 0 |
| / | ( ( / | d | 1 |
| f | ( ( / f | d | 1 |
| + | ( ( + | d f / | 1 |
| e | ( ( + e | d f / | 1 |
| ) | ( | d f / e + | 1 |
| * | ( + | d f / e + | 1 |
| ( | ( * ( | d f / e + | 1 |
| d | ( * ( d | d f / e + | 1 |
| ^ | ( * ( ^ | d f / e + d | 2 |
| c | ( * ( ^ c | d f / e + d | 2 |
| ^ | ( * ( ^ ^ | d f / e + d c | 3 |
| b | ( * ( ^ ^ b | d f / e + d c | 3 |
| + | ( * ( + | d f / e + d c b ^ ^ | 2 |
| a | ( * ( + a | d f / e + d c b ^ ^ | 2 |
| ) | ( * | d f / e + d c b ^ ^ a + | 2 |
| ) | | d f / e + d c b ^ ^ a + * | 1 |

**Step 4 : Reverse this postfix expression**

*** + a ^ ^ b c d + e / f d**

## Write an algorithm for evaluation of postfix expression and evaluation the following expression showing every status of stack in tabular form.

## (i) 5 4 6 + * 4 9 3 / + * (ii) 7 5 2 + * 4 1 1 + / -

**Algorithm: EVALUAE_POSTFIX**

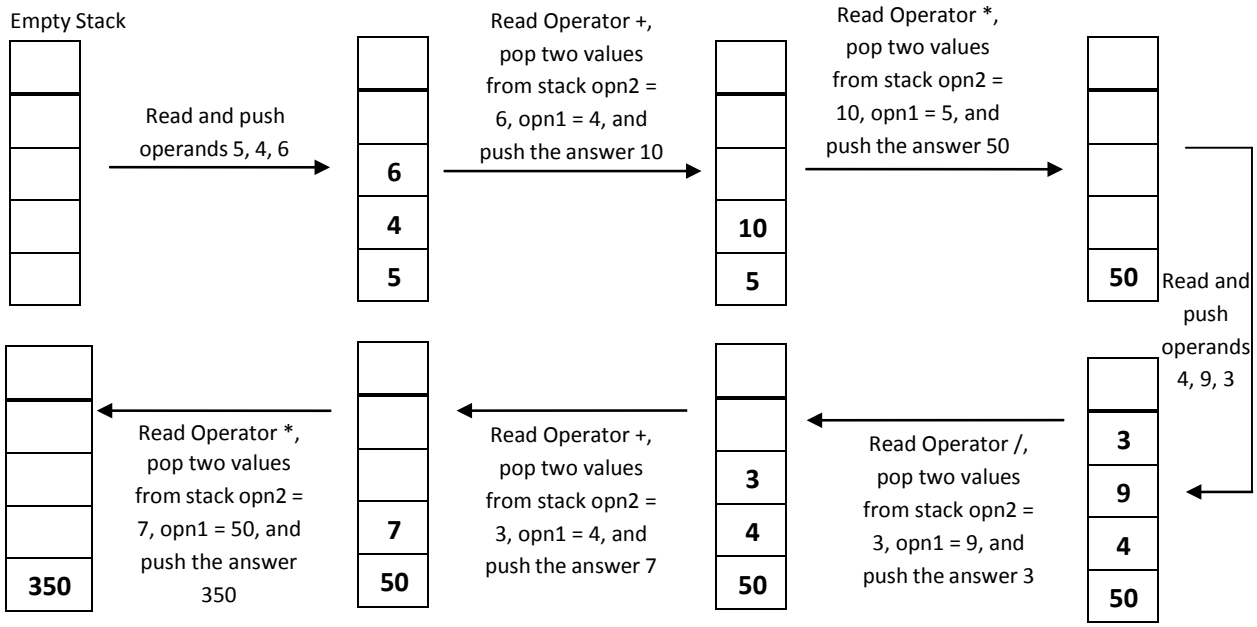
- Given an input string POSTFIX representing postfix expression. This algorithm is going to evaluate postfix expression and put the result into variable VALUE. A vector S is used as a stack PUSH and POP are the function used for manipulation of stack. Operand2 and operand1 are temporary variable TEMP is used for temporary variable NEXTCHAR is a function which when invoked returns the next character. PERFORM_OPERATION is a function which performs required operation on OPERAND1 AND OPERAND2.

**1.** [Initialize stack and value]
TOP ← 1
VALUE ← 0

**2.** [Evaluate the prefix expression]
Repeat until last character
TEMP ← NEXTCHAR (POSTFIX)
If TEMP is DIGIT
Then PUSH (S, TOP, TEMP)
Else OPERAND2 ← POP (S, TOP)
OPERAND1 ← POP (S, TOP)
VALUE ← PERFORM_OPERATION(OPERAND1, OPERAND2, TEMP)
PUSH (S, POP, VALUE)

**3.** [Return answer from stack]
Return (POP (S, TOP))

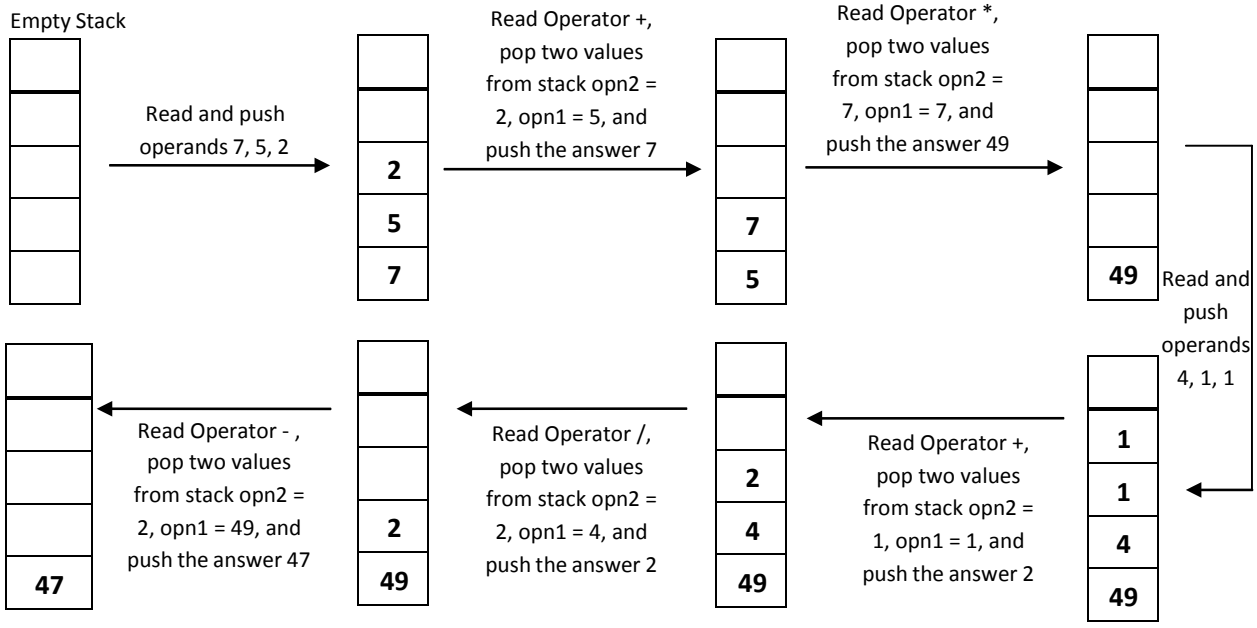
***Evaluate (i): 5 4 6 + * 4 9 3 / + ****




Poped value **350** is the answer

---

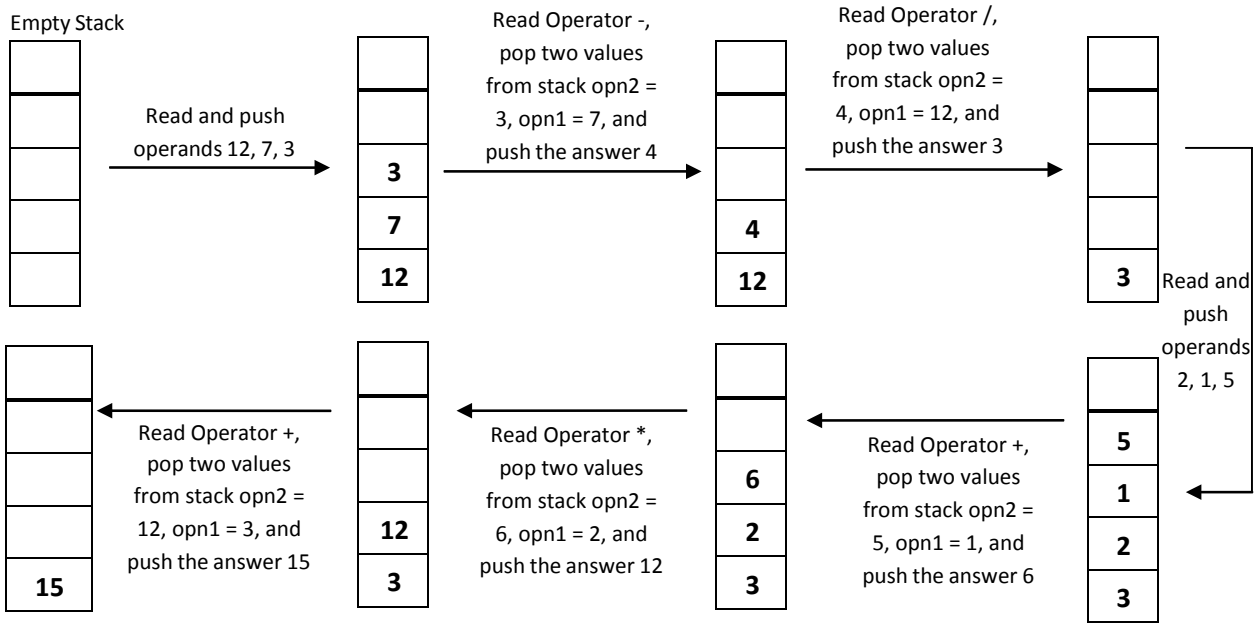
***Evaluate (ii) : * 7 5 2 + * 4 1 1 + / -***




Poped value **47** is the answer

## Consider the following arithmetic expression P, written in postfix notation. Translate it in infix notation and evaluate. P: 12, 7, 3, -, /, 2, 1, 5, +, *, +

Same Expression in infix notation is : **( 12 / ( 7 – 3 ) ) + ( ( 5 + 1 ) * 2 )**




Poped value **15** is the answer

---

## Explain Difference between Stack and Queue.

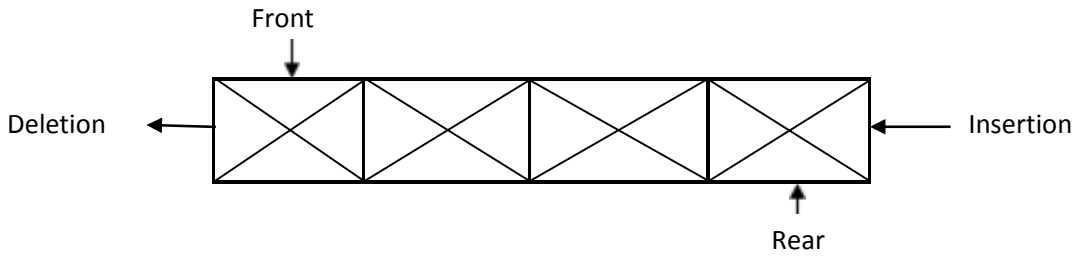
| **Stack** | **Queue** |
|---|---|
| A Linear List Which allows insertion or deletion of an element at one end only is called as Stack | A Linear List Which allows insertion at one end and deletion at another end is called as Queue |
| Since insertion and deletion of an element are performed at one end of the stack, the elements can only be removed in the opposite order of insertion. | Since insertion and deletion of an element are performed at opposite end of the queue, the elements can only be removed in the same order of insertion. |
| Stack is called as Last In First Out (LIFO) List. | Queue is called as First In First Out (FIFO) List. |
| The most and least accessible elements are called as TOP and BOTTOM of the stack | Insertion of element is performed at FRONT end and deletion is performed from REAR end |
| Example of stack is arranging plates in one above one. | Example is ordinary queue in provisional store. |
| Insertion operation is referred as PUSH and deletion operation is referred as POP | Insertion operation is referred as ENQUEUE and deletion operation is referred as DQUEUE |
| Function calling in any languages uses Stack | Task Scheduling by Operating System uses queue |

## Explain following:

## (i) Queue (ii) Circular Queue (iii) DQUEUE (iv) Priority Queue

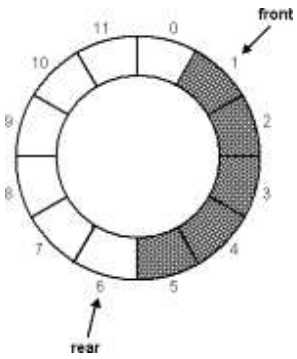
### (i) Queue

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



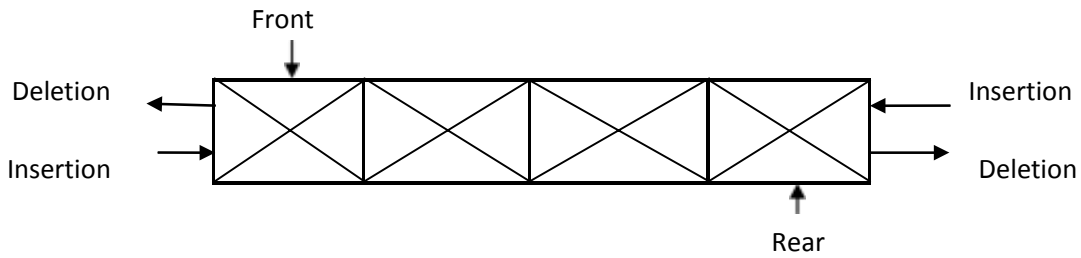

### (ii) Circular Queue

- A more suitable method of representing simple queue which prevents an excessive use of memory is to arrange the elements Q[1], Q[2]….,Q[n] in a circular fashion with Q[1] following Q[n], this is called circular queue
- In a standard queue data structure re-buffering problem occurs for each **dequeue** operation. To solve this problem by joining the front and rear ends of a queue to make the queue as a circular queue
- Circular queue is a linear data structure. It follows FIFO principle.
- In circular queue the last node is connected back to the first node to make a circle.
- Circular linked list fallow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as “Ring buffer”.




**(iii) Dequeue**

- A dequeue (double ended queue ) is a linear list in which insertion and deletion are performed from the either end of the structure.
- There are two variations of Dqueue
  - Input restricted dqueue- allows insertion at only one end
  - Output restricted dqueue- allows deletion from only one end
- Such a structure can be represented by following fig.




**(iv) Priority Queue**

- A queue in which we are able to insert remove items from any position based on some property (such as priority of the task to be processed) is often referred as priority queue.
- Below fig. represent a priority queue of jobs waiting to use a computer.
- Priorities of 1, 2, 3 have been attached with jobs of real time, online and batch respectively. Therefore if a job is initiated with priority i,it is inserted immediately at the end of list of other jobs with priorities i. Here jobs are always removed from the front of queue

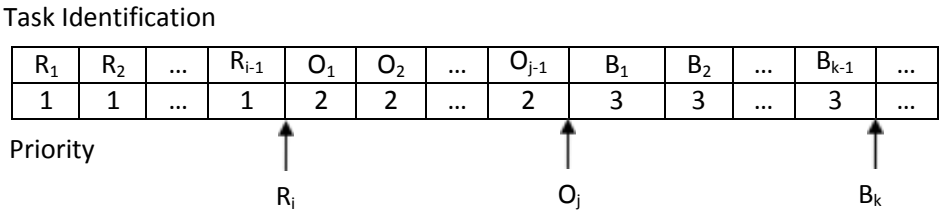



**Fig (a) : Priority Queue viewed as a single queue with insertion allowed at any position.**

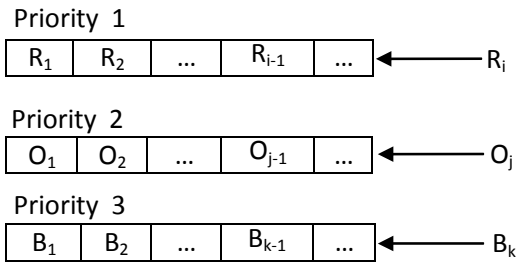



**Fig (b) : Priority Queue viewed as a Viewed as a set of queue**

## Write algorithms of basic primitive operations for Queue

**Procedure: QINSERT_REAR (Q, F, R, N,Y)**

- Given F and R pointers to the front and rear elements of a queue respectively. Queue Q consisting of N elements. This procedure inserts Y at rear end of Queue.

1. [Overflow]
   IF R >= N
   Then write ('OVERFLOW')
   Return
2. [Increment REAR pointer]
   R ← R + 1
3. [Insert element ]
   Q[R] ← Y
4. [Is front pointer properly set]
   IF F=0
   Then F ← 1
   Return

**Function: QDELETE_FRONT (Q, F, R)**

- Given F and R pointers to the front and rear elements of a queue respectively. Queue Q consisting of N elements. This function deleted and element from front end of the Queue.

1. [Underflow]
   IF F= 0
   Then write ('UNDERFLOW')
   Return(0) (0 denotes an empty Queue)
2. [Decrement element]
   Y ← Q[F]
3. [Queue empty?]
   IF F=R
   Then F← R← 0
   Else F← F+1 (increment front pointer)
4. [Return element]
   Return (Y)

## Write algorithms of basic primitive operations for Circular Queue

**Procedure: CQINSERT (F, R, Q, N, Y)**

- Given F and R pointers to the front and rear elements of a circular queue respectively. Circular queue Q consisting of N elements. This procedure inserts Y at rear end of Circular queue.

**1.** [Reset Rear Pointer]
If R = N
Then R← 1
Else R ← R + 1

**2.** [Overflow]
If F = R
Then Write ('Overflow')
Return

**3.** [Insert element]
Q[R] ← Y

**4.** [Is front pointer properly set?]
If F = 0
Then F ← 1
Return

**Function CQDELETE (F, R, Q, N)**

- Given F and R pointers to the front and rear elements of a Circular queue respectively. Circular Queue Q consisting of N elements. This function deleted and element from front end of the Circular Queue. Y is temporary pointer variable.

1. [Underflow?]
   If F = 0
   Then Write ('UNDERFLOW')
   Return (0)
2. [Delete Element]
   Y ← Q[F]
3. [Queue Empty?]
   If F = R
   Then F ← R ← 0
   Return (Y)
4. [Increment front pointer]
   If F = N
   Then F ← 1
   Else F ← F + 1
   Return (Y)

## Write algorithms of basic primitive operations for DQueue

### Procedure DQINSERT_FRONT (Q, F, R, N,Y)

- Given F and R pointers to the front and rear elements of a queue, a queue consisting of N elements and an element Y, this procedure inserts Y at the front of the queue.

1. [Overflow]
   IF F = 0
   Then write ('EMPTY')
   Return
   IF F=1
   Then write ('OVERFLOW')
   Return
2. [Decrement front pointer]
   F ← F-1
3. [Insert element ]
   Q[F] ← Y
   Return

### Procedure DQDELETE_REAR (Q, F, R)

- Given F and R pointers to the front and rear elements of a queue. And a queue Q to which they correspond, this function deletes and returns the last element from the front end of a queue. And Y is temporary variable.

**1.** [Underflow]
IF R= 0
Then write ('UNDERFLOW')
Return(0)

**2.** [Delete element]
Y ← Q[R]

**3.** [Queue empty?]
IF R=F
Then R← F← 0
Else R← R-1 (decrement front pointer)

**4.** [Return element]
Return (Y)

**PROCEDURE DQUEUE_DISPLAY (F,R,Q)**

- Given F and Rare pointers to the front and rear elements of a queue, a queue consist of N elements. This procedure display Queue contents

```
1. [Check for empty]
   IF      F >= R
    Then   write ('QUEUE IS EMPTY')
           Return
2. [Display content]
   FOR     (I=FRONT; I<=REAER; I++)
           Write (Q[I])
3. [Return Statement]
   Return
```

---

## Consider the following queue, where queue is a circular queue having 6 memory cells. Front=2, Rear=4

**Queue: _, A, C, D, _, _**

**Describe queue as following operation take place:**

**F is added to the queue**

**Two letters are deleted**

**R is added to the queue**

**S is added to the queue**

**One letter is deleted**

| **Positions** | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|
| Initial Position of Queue, **Front=2, Rear=4** | | A | C | D | | |
| F is added to queue, **Front=2, Rear=5** | | A | C | D | F | |
| Two letters are deleted, **Front=4, Rear=5** | | | | D | F | |
| R is added to the queue, **Front=4, Rear=6** | | | | D | F | R |
| S is added to the queue, **Front=4, Rear=1** | S | | | D | F | R |
| One letter is deleted, **Front=5, Rear=1** | S | | | | F | R |

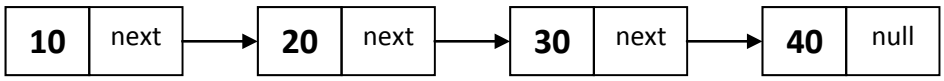
## 1. Linear Data Structure and their linked storage representation.

There are many applications where sequential allocation method is unacceptable because of following characteristics

- Unpredictable storage requirement
- Extensive manipulation of stored data

The linked allocation method of storage can result in both efficient use of computer storage and computer time.

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

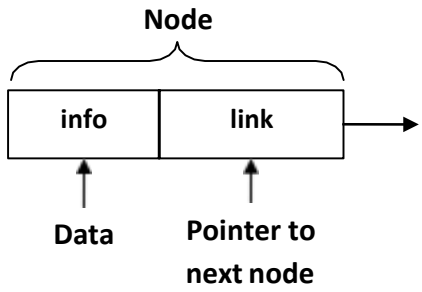



A Linked List

---

## 2. What is linked list? What are different types of linked list? OR Write a short note on singly, circular and doubly linked list. OR Advantages and disadvantages of singly, circular and doubly linked list.

- A linked list is a collection of objects stored in a list form.
- A linked list is a sequence of items (objects) where every item is linked to the next.
- A linked list is a non-primitive type of data structure in which each element is dynamically allocated and in which elements point to each other to define a linear relationship.
- Elements of linked list are called nodes where each node contains two things, data and pointer to next node.
- Linked list require more memory compared to array because along with value it stores pointer to next node.
- Linked lists are among the simplest and most common data structures. They can be used to implement other data structures like stacks, queues, and symbolic expressions, etc…




```
// C Structure to represent a node
struct node
{
      int info
      struct node *link
};
```

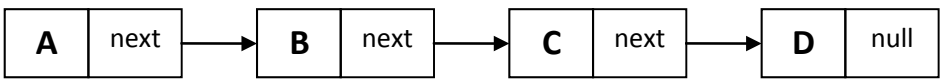
## Operations on linked list

- Insert
  - Insert at first position
  - Insert at last position
  - Insert into ordered list
- Delete
- Traverse list (Print list)
- Copy linked list

## Types of linked list

### Singly Linked List

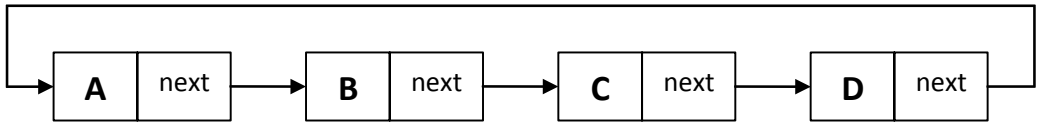
- It is basic type of linked list.
- Each node contains data and pointer to next node.
- Last node's pointer is null.
- Limitation of singly linked list is we can traverse only in one direction, forward direction.




Singly Linked List

### Circular Linked List

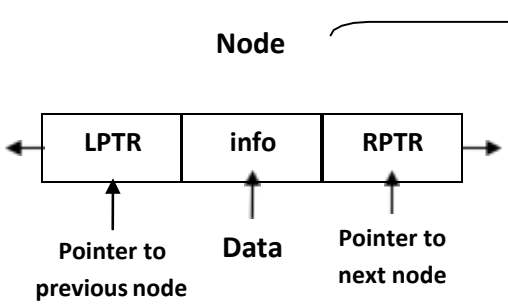
- Circular linked list is a singly linked list where last node points to first node in the list.
- It does not contain null pointers like singly linked list.
- We can traverse only in one direction that is forward direction.
- It has the biggest advantage of time saving when we want to go from last node to first node, it directly points to first node.
- A good example of an application where circular linked list should be used is a timesharing problem solved by the operating system.




Circular Linked List

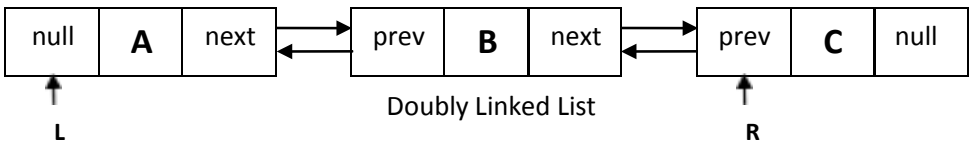
**Doubly Linked list**

- Each node of doubly linked list contains data and two pointers to point previous (LPTR) and next (RPTR) node.




```
// C Structure to represent a node
struct node
{
      int info
      struct node *lptr;
      struct node *rptr;
};
```

- Main advantage of doubly linked list is we can traverse in any direction, forward or reverse.
- Other advantage of doubly linked list is we can delete a node with little trouble, since we have pointers to the previous and next nodes. A node on a singly linked list cannot be removed unless we have the pointer to its predecessor.
- Drawback of doubly linked list is it requires more memory compared to singly linked list because we need an extra pointer to point previous node.
- L and R in image denote left most and right most nodes in the list.
- Left link of L node and right link of R node is NULL, indicating the end of list for each direction.




Doubly Linked List

---

## 3. Discuss advantages and disadvantages of linked list over array.

### Advantages of an array

1. We can access any element of an array directly means random access is easy
2. It can be used to create other useful data structures (queues, stacks)

3. It is light on memory usage compared to other structures

### Disadvantages of an array

1. Its size is fixed
2. It cannot be dynamically resized in most languages
3. It is hard to add/remove elements
4. Size of all elements must be same.
5. Rigid structure (Rigid = Inflexible or not changeable)

### Advantages of Linked List

1. **Linked lists are dynamic data structures:** That is, they can grow or shrink during execution of a program.
2. **Efficient memory utilization:** Here memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated (free) when it is no longer needed.
3. **Insertion and deletions are easier and efficient:** Linked list provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. **Elements of linked list are flexible:** It can be primary data type or user defined data types

### Disadvantages of Linked List

1. Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
2. It cannot be easily sorted
3. We must traverse 1/2 the list on average to access any element
4. More complex to create than an array
5. Extra memory space for a pointer is required with each element of the list

---

## 3. What are the advantages and disadvantages of stack and queue implemented using linked list over array?

Advantages and disadvantages of stack & queue implemented using linked list over array is described below,

### Insertion & Deletion Operation

- Insertion and deletion operations are known as push and pop operation in stack and as insert and delete operation in queue.
- In the case of an array, if we have n-elements list and it is required to insert a new element between the first and second element then n-1 elements of the list must be moved so as to make room for the new element.
- In case of linked-list, this can be accomplished by only interchanging pointers.
- Thus, insertion and deletions are more efficient when performed in linked list then array.

## Searching a node

- If a particular node in a linked list is required, it is necessary to follow links from the first node onwards until the desired node is found.
- Where as in the case of an array, directly we can access any node

## Join & Split

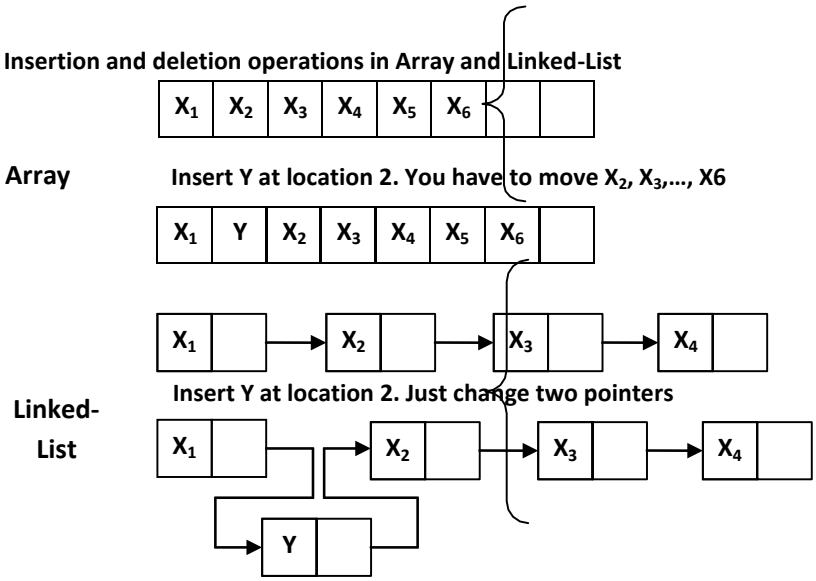
- We can join two linked list by assigning pointer of second linked list in the last node of first linked list.
- Just assign null address in the node from where we want to split one linked list in two parts.
- Joining and splitting of two arrays is much more difficult compared to linked list.

## Memory

- The pointers in linked list consume additional memory compared to an array

## Size

- Array is fixed sized so number of elements will be limited in stack and queue.
- Size of linked list is dynamic and can be changed easily so it is flexible in number of elements

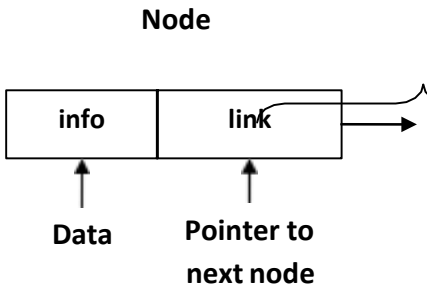


## 4. Write following algorithms for singly linked list.

### 1) Insert at first position
### 2) Insert at last position
### 3) Insert in Ordered Linked list
### 4) Delete Element
### 5) Copy Linked List

Few assumptions,

- We assume that a typical element or node consists of two fields namely; an information field called INFO and pointer field denoted by LINK. The name of a typical element is denoted by NODE.




```
// C Structure to represent a node
struct node
{
      int info
      struct node *link
};
```

**Function: INSERT( X, First )**

Given X, a new element and FIRST is a pointer to the first element of a linked linear list. Typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack; NEW is a temporary pointer variable. This function inserts a new node at the first position of linked list. This function returns address of FIRST node.

**1 [Underflow?]**
IF AVAIL = NULL
Then Write ("Availability Stack Underflow")
Return(FIRST)

**2 [Obtain address of next free Node]**
NEW←AVAIL

**3 [Remove free node from Availability Stack]**
AVAIL←LINK(AVAIL)

**4 [Initialize fields of new node and its link to the list]**
INFO (NEW) ← X
LINK (NEW) ← FIRST

**5 [Return address of new node]**
Return (NEW)

**When INSERT is invoked it returns a pointer value to the variable FIRST**

**FIRST ← INSERT (X, FIRST)**

**Function: INSEND( X, First ) (Insert at end)**

Given X, a new element and FIRST is a pointer to the first element of a linked linear list. Typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack; NEW is a temporary pointer variable. This function inserts a new node at the last position of linked list. This function returns address of FIRST node.

**1 [Underflow?]**
IF AVAIL = NULL
Then Write ("Availability Stack Underflow")
Return(FIRST)

**2 [Obtain address of next free Node]**
NEW←AVAIL

**3 [Remove free node from Availability Stack]**
AVAIL←LINK(AVAIL)

**4 [Initialize field of NEW node]**
INFO (NEW) ← X
LINK (NEW) ← NULL

**5 [Is the list empty?]**
If FIRST = NULL
then Return (NEW)

**6 [Initialize search for a last node]**
SAVE ← FIRST

**7 [Search for end of list]**
Repeat while LINK (SAVE) ≠ NULL
SAVE ← LINK (SAVE)

**8 [Set link field of last node to NEW)**
LINK (SAVE) ← NEW

**9 [Return first node pointer]**
Return (FIRST)

**When INSERTEND is invoked it returns a pointer value to the variable FIRST**

**FIRST ← INSERTEND (X, FIRST)**

**Insert a node into Ordered Linked List**

- There are many applications where it is desirable to maintain an ordered linear list. The ordering is in increasing or decreasing order on INFO field. Such ordering results in more efficient processing.
- The general algorithm for inserting a node into an ordered linear list is as below.
  1. Remove a node from availability stack.
  2. Set the field of new node.
  3. If the linked list is empty then return the address of new node.
  4. If node precedes all other nodes in the list then inserts a node at the front of the list and returns its address.
  5. Repeat step 6 while information contain of the node in the list is less than the information content of the new node.
  6. Obtain the next node in the linked list.
  7. Insert the new node in the list and return address of its first node.

**Function: INSORD( X, FIRST )**

Given X, a new element and FIRST is a pointer to the first element of a linked linear list. Typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack; NEW & SAVE are temporary pointer variables. This function inserts a new node such that linked list preserves the ordering of the terms in increasing order of their INFO field. This function returns address of FIRST node.

**1 [Underflow?]**
IF AVAIL = NULL
Then Write ("Availability Stack Underflow")
Return(FIRST)

**2 [Obtain address of next free Node]**
NEW←AVAIL

**3 [Remove free node from Availability Stack]**
AVAIL←LINK(AVAIL)

**4. [Is the list is empty]**
If FIRST = NULL
then LINK (NEW) ← NULL
Return (NEW)

**5. [Does the new node precede all other node in the list?]**
If INFO(NEW) ≤ INFO (FIRST)
then LINK (NEW) ← FIRST
Return (NEW)

**6. [Initialize temporary pointer]**
SAVE ← FIRST

**7. [Search for predecessor of new node]**
Repeat while LINK (SAVE) ≠ NULL and INFO (NEW) ≥ INFO (LINK (SAVE))
SAVE ← LINK (SAVE)

**8. [Set link field of NEW node and its predecessor]**
LINK (NEW) ← LINK (SAVE)
LINK (SAVE) ← NEW

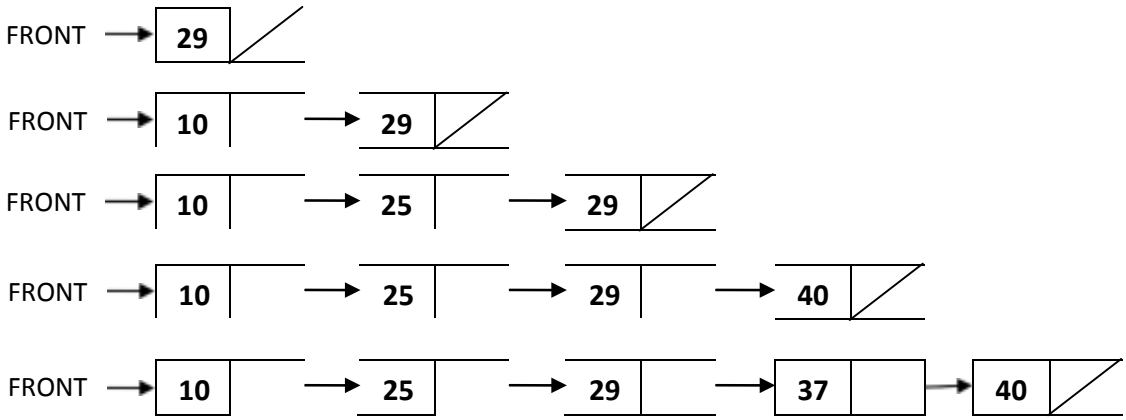
**9. [Return first node pointer]**
Return (FIRST)

**When INSERTORD is invoked it returns a pointer value to the variable FIRST**

**FIRST ← INSERTORD (X, FIRST)**

By repeatedly involving function INSORD; we can easily obtains an ordered liner list for example the sequence of statements.

FRONT ← NULL
FRONT ← INSORD (29, FRONT)
FRONT ← INSORD (10, FRONT)
FRONT ← INSORD (25, FRONT)
FRONT ← INSORD (40, FRONT)
FRONT ← INSORD (37, FRONT)




**Trace of construction of an ordered linked linear list using function INSORD**

---

### Algorithm to delete a node from Linked List

- Algorithm that deletes node from a linked linear list:-
  1. If a linked list is empty, then write under flow and return.
  2. Repeat step 3 while end of the list has not been reached and the node has not been found.
  3. Obtain the next node in list and record its predecessor node.
  4. If the end of the list has been reached then write node not found and return.
  5. Delete the node from list.
  6. Return the node into availability area.

**Procedure: DELETE( X, FIRST)**

Given X, an address of node which we want to delete and FIRST is a pointer to the first element of a linked linear list. Typical node contains INFO and LINK fields. SAVE & PRED are temporary pointer variables.

1. **[Is Empty list?]**
   If FIRST = NULL
   then write ('Underflow')
   return
2. **[Initialize search for X]**
   SAVE ← FIRST
3. **[Find X]**
   Repeat thru step-5 while SAVE ≠ X and LINK (SAVE) ≠ NULL
4. **[Update predecessor marker]**
   PRED ← SAVE
5. **[Move to next node]**
   SAVE ← LINK (SAVE)
6. **[End of the list]**
   If SAVE ≠ X
   then write ('Node not found')
   return
7. **[Delete X]**
   If X = FIRST (if X is first node?)
   then FIRST ← LINK (FIRST)
   else LINK (PRED) ← LINK (X)
8. **[Free Deleted Node]**
   Free (X)

---

**Function: COPY (FIRST)**

- FIRST is a pointer to the first node in the linked list, this function makes a copy of the list.
- The new list is to contain nodes whose information and pointer fields are denoted by FIELD and PTR, respectively. The address of the first node in the newly created list is to be placed in BEGIN. NEW, SAVE and PRED are points variables.
- A general algorithm to copy a linked list
  1. If the list is empty then return null

2. If the availability stack is empty then write availability stack underflow and return else copy the first node.
3. Report thru step 5 while the old list has not been reached.
4. Obtain next node in old list and record its predecessor node.
5. If availability stack is empty then write availability stack underflow and return else copy the node and add it to the rear of new list.
6. Set link of the last node in the new list to null and return.

1. **[Is Empty List?]**
   If FIRST = NULL
   then return (NULL)

2. **[Copy first node]**
   NEW ⇦ NODE
   New ← AVAIL
   AVAIL ← LINK (AVAIL)
   FIELD (NEW) ← INFO (FIRST)
   BEGIN ← NEW

3. **[Initialize traversal]**
   SAVE ← FIRST

4. **[Move the next node if not at the end if list]**
   Repeat thru step 6 while (SAVE) ≠ NULL

5. **[Update predecessor and save pointer]**
   PRED ← NEW
   SAVE ← LINK (SAVE)

6. **[Copy node]**
   If AVAIL = NULL
   then write ('Availability stack underflow')
   Return (0)
   else NEW ← AVAIL
   AVAIL ← LINK (AVAIL)
   FIELD (NEW) ← INFO (SAVE)
   PTR (PRED) ← NEW

7. **[Set link of last node and return]**
   PTR (NEW) ← NULL
   Return (BEGIN)

## 5. Write following algorithms for circular link list

1) **Insert at First Position**
2) **Insert at Last Position**
3) **Insert in Ordered Linked List**
4) **Delete Element**

### PROCEDURE: CIRCULAR_LINK_INSERT_FIRST (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary pointer variable. This procedure inserts value X at the first position of Circular linked linear list.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Initialize fields of new node and its link to the list]**
   INFO (NEW) ← X
   If FIRST = NULL
   then LINK (NEW) ← NEW
   FIRST ← LAST ← NEW
   else LINK (NEW) ← FIRST
   LINK (LAST) ← NEW
   FIRST ← NEW
   Return

## PROCEDURE: CIR_LINK_INSERT_END (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary pointer variable. This procedure inserts value X at the last position of Circular linked linear list.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Initialize fields of new node and its link to the list]**
   If FIRST = NULL
   then LINK (NEW) ← NEW
   FIRST ← LAST ← NEW
   else LINK(NEW) ← FIRST
   LINK(LAST) ← NEW
   LAST ← NEW
   Return

## PROCEDURE: CIR_LINK_INSERT_ORDER (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary pointer variable. This procedure inserts value X such that linked list preserves the ordering of the terms in increasing order of their INFO field.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Copy information content into new node]**
   INFO (NEW) ← X
3. **[Is Linked List is empty?]**
   If FIRST = NULL
   then LINK (NEW) ← NEW
   FIRST ← LAST ← NEW
   Return
4. **[Does new node precedes all other nodes in List?]**
   If INFO (NEW) ≤ INFO (FIRST)
   then LINK (NEW) ← FIRST
   LINK (LAST) ← NEW
   FIRST ← NEW
   Return
5. **[Initialize Temporary Pointer]**
   SAVE ← FIRST
6. **[Search for Predecessor of new node]**
   Repeat while SAVE ≠ LAST and INFO(NEW) ≥ INFO(LINK(SAVE))
   SAVE ← LINK(SAVE)
7. **[Set link field of NEW node and its Predecessor]**
   LINK(NEW) ← LINK(SAVE)
   LINK(SAVE) ← NEW
   If SAVE = LAST
   then LAST ← NEW
8. **[Finish]**
   Return

## PROCEDURE: CIR_LINK_DELETE (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. SAVE & PRED are temporary pointer variables. This procedure deletes a node whose address is given by pointer variable X.

1. **[Is Empty List?]**
   If FIRST = NULL
   then write ('Linked List is Empty')
   Return
2. **[Initialize Search for X]**
   SAVE ← FIRST
3. **[Find X]**
   Repeat thru step 5 while SAVE ≠ X and SAVE ≠ LAST
4. **[Update predecessor marker]**
   PRED ← SAVE
5. **[Move to next node]**
   SAVE ← LINK (SAVE)
6. **[End of Linked List]**
   If SAVE ≠ X
   then write('Node not found')
   return
7. **[Delete X]**
   If X = FIRST
   then FIRST ← LINK (FIRST)
   LINK (LAST) ← FIRST
   else LINK (PRED) ← LINK(X)
   If X = LAST
   then LAST ← PRED
8. **[Free Deleted Node]**
   Free (X)

**6. Write an algorithm to perform each of the following operations on Circular singly linked list using header node**

**1) add node at beginning**

**2) add node at the end**

**3) insert a node containing x after node having address P**

**4) delete a node which contain element x**

### FUNCTION: CIR_LINK_HEAD_INSERT_FIRST (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary pointer variable. HEAD is the address of HEAD node. This procedure inserts value X at the first position of Circular linked linear list.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Initialize fields of new node and its link to the list]**
   INFO (NEW) ← X
   LINK (NEW) ← LINK (HEAD)
   LINK (HEAD) ← NEW

---

### FUNCTION: CIR_LINK_HEAD_INSERT_LAST (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary points variable. HEAD is the address of HEAD node. This procedure inserts value X at the last position of Circular linked linear list.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Initialize fields of new node and its link to the list]**
   INFO (NEW) ← X
   LINK (NEW) ← HEAD
   LINK (LAST) ← NEW
   LAST ← NEW

---

## FUNCTION: CIR_LINK_HEAD_INSERT_AFTER_Node-P (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary pointer variable. HEAD is the address of HEAD node. This procedure insert a node after a node having address P.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Initialize fields of new node and its link to the list]**
   INFO (NEW) ← X
   LINK (NEW) ← LINK (P)
   LINK (P) ← NEW
   If P = LAST
   then LAST ← NEW

## PROCEDURE: CIR_LINK_HEAD_DELETE (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. SAVE & PRED are temporary pointer variables. HEAD is the address of HEAD node. This procedure deletes a node whose value is X.

1. **[Is Empty List?]**
   If FIRST = NULL
   then write ('Underflow)
   return
2. **[Initialize Search for X]**
   SAVE ← FIRST
3. **[Find X]**
   Repeat thru step 5 while INFO(SAVE) ≠ X and SAVE ≠ LAST
4. **[Update Predecessor]**
   PRED ← SAVE
5. **[Move to next node]**
   SAVE ← LINK(SAVE)
6. **[End of the List]**
   If INFO (SAVE) ≠ X
   then write('Node not Found')
   return
7. **[Delete node X]**
   If INFO (FIRST) = X
   then LINK (HEAD) ← LINK(FIRST)
   else LINK (PRED) ← LINK(SAVE)
   If SAVE = LAST
   then LAST ← PRED
8. **[Free Deleted Node]**
   Free (X)

## 7. Write following algorithms for doubly link list

### 1) Insert
### 2) Insert in Ordered Linked List
### 3) Delete Element

**PRDCEDURE: DOUBINS (L, R, M, X)**

Given a doubly link list whose left most and right most nodes addressed are given by the pointer variables L and R respectively. It is required to insert a node whose address is given by the pointer variable NEW. The left and right links of nodes are denoted by LPTR and RPTR respectively. The information field of a node is denoted by variable INFO. The name of an element of the list is NODE. The insertion is to be performed to the left of a specific node with its address given by the pointer variable M. The information to be entered in the node is contained in X.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Copy information field]**
   INFO (NEW) ← X
3. **[Insert into an empty list]**
   If R = NULL
   then LPTR (NEW) ← RPTR (NULL) ← NULL
   L ← R ← NEW
   Return
4. **[Is left most insertion ?]**
   If M = L
   then LPTR (NEW) ←NULL
   RPTR (NEW) ← M
   LPTR (M)← NEW
   L ← NEW
   Return
5. **[Insert in middle]**
   LPTR (NEW)← LPTR (M)
   RPTR (NEW) ← M
   LPTR (M) ← NEW
   RPTR (LPTR (NEW)) ← NEW
   Return

## PROCEDURE: DOUBINS_ORD (L, R, M, X)

Given a doubly link list whose left most and right most nodes addressed are given by the pointer variables L and R respectively. It is required to insert a node whose address is given by the pointer variable NEW. The left and right links of nodes are denoted by LPTR and RPTR respectively. The information field of a node is denoted by variable INFO. The name of an element of the list is NODE. The insertion is to be performed in ascending order of info part. The information to be entered in the node is contained in X.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[ Copy information field]**
   INFO (NEW) ← X
3. **[Insert into an empty list**]
   If R = NULL
   then LPTR (NEW) ← RPTR (NULL) ← NULL
   L ← R ← NEW
   return
4. **[Does the new node precedes all other nodes in List? ]**
   If INFO(NEW) ≤ INFO(L)
   then RPTR (NEW) ← L
   LPTR(NEW)← NULL
   LPTR (L) ← NEW
   L ← NEW
   Return
5. **[ Initialize temporary Pointer]**
   SAVE ← L
6. **[Search for predecessor of New node]**
   Repeat while RPTR(SAVE) ≠ NULL and INFO(NEW) ≥ INFO(RPTR(SAVE))
   SAVE ← RPTR (SAVE)
7. **[Set link field of new node and its predecessor]**
   RPTR (NEW) ← RPTR(SAVE)
   LPTR (RPTR(SAVE)) ← NEW
   RPTR (SAVE) ← NEW
   LPTR (NEW) ← SAVE

   If SAVE = R
   then RPTR(SAVE) ← NEW

## PROCEDURE: DOUBDEL (L, R, OLD)

Given a doubly linked list with the addresses of left most and right most nodes are given by the pointer variables L and R respectively. It is required to delete the node whose address is contained in the variable OLD. Node contains left and right links with names LPTR and RPTR respectively.

1. **[ Is underflow ?]**
   If R=NULL
   then write (' UNDERFLOW')
   return

2. **[Delete node]**
   If L = R (single node in list)
   then L ← R ← NULL
   else If OLD = L (left most node)
   then L ← RPTR(L)
   LPTR (L) ← NULL
   else if OLD = R (right most)
   then R ← LPTR (R)
   RPTR (R) ← NULL
   else RPTR (LPTR (OLD)) ← RPTR (OLD)
   LPTR (RPTR (OLD)) ← LPTR (OLD)

3. **[ FREE deleted node]**
   FREE (OLD)

## 8. Write the implementation procedure of basic primitive operations of the stack using: (i) Linear array (ii) linked list.

**Implement PUSH and POP using Linear array**

```
#define MAXSIZE 100
int stack[MAXSIZE];
int top=-1;

void push(int val)
{
      if(top >= MAXSIZE)
          printf("Stack is Overflow");
      else
          stack[++top] = val;
}

int pop()
{
     int a;
     if(top>=0)
     {
          a=stack[top];
          top--;
          return a;
     }
     else
     {
          printf("Stack is Underflow, Stack is empty, nothing to POP!");
          return -1;
     }
}
```

**Implement PUSH and POP using Linked List**

```
#include<stdio.h>
#include<malloc.h>

struct node
{
    int info;
    struct node *link;
} *top;

void push(int val)
{
      struct node *p;
      p = (struct node*)malloc(sizeof(struct node));
      p → info = val;
      p → link = top;
      top =  p;
      return;
}

int pop()
{
      int val;
      if(top!=NULL)
      {
            val = top → info;
            top=top →link;
            return val;
      }
      else
      {
            printf("Stack Underflow");
            return -1;
      }
}
```

---

## 9. Write the implementation procedure of basic primitive operations of the Queue using: (i) Linear array (ii) linked list

**Implement Enqueue(Insert)and Dequeue(Delete)using Linear Array**

```
# include <stdio.h>
# define MAXSIZE 100
int queue[MAXSIZE], front = -1, rear = -1;
void enqueue(int val)
{
      if(rear >= MAXSIZE)
      {
            printf("Queue is overflow") ;
            return ;
      }
      rear++;
      queue [rear] = val;
      if(front == -1)
      {
            front++;
      }
}
int dequeue()
{
      int data;
      if(front == -1)
      {
            printf("Queue is underflow") ;
            return -1;
      }
      data = queue [front];
      if(front == rear)
      {
            front = rear = -1;
      }
      else
      {
            front++;
      }
      return data;
}
```

**Implement Enqueue(Insert)and Dequeue(Delete)using Linked List**

```
#include<stdio.h>
#include<malloc.h>

struct node
{
    int info;
    struct node *link;
} *front, *rear;


void enqueue(int val)
{
      struct node *p;
      p = (struct node*)malloc(sizeof(struct node));
      p → info = val;
      p → link = NULL;
      if (rear == NULL || front == NULL)
      {
            front = p;
      }
      else
      {
            rear → link = p;
            rear = p;
      }
}

int dequeue()
{
    struct node *p;
    int val;
    if (front == NULL || rear == NULL)
    {
        printf("Under Flow");
        exit(0);
    }
    else
    {
        p = front;
        val = p → info;
        front = front → link;
        free(p);
    }
    return (val);
}
```

**10. Write an algorithm to implement ascending priority queue using singular linear linked list which has insert() function such that queue remains ordered list. Also implement remove() function**

```
struct node
{
      int priority;
      int info;
      struct node *link;
}*front = NULL;


remove()
{
      struct node *tmp;
      if(front == NULL)
            printf("Queue Underflow\n");
      else
      {
            tmp = front;
            printf("Deleted item is %d\n",tmp->info);
            front = front->link;
            free(tmp);
      }
}/*End of remove()*/
```

```
insert()
{
      struct node *tmp,*q;
      int added_item,item_priority;
      tmp = (struct node *)malloc(sizeof(struct node));
      printf("Input the item value to be added in the queue : ");
      scanf("%d",&added_item);
      printf("Enter its priority : ");
      scanf("%d",&item_priority);
      tmp->info = added_item;
      tmp->priority = item_priority;
      /*Queue is empty or item to be added has priority more than
first item*/
      if( front == NULL || item_priority < front->priority )
      {
            tmp->link = front;
            front = tmp;
      }
      else
      {
            q = front;
            while( q->link != NULL &&
                        q->link->priority <= item_priority )
            {
                  q=q->link;
            }
            tmp->link = q->link;
            q->link = tmp;
      }/*End of else*/
}/*End of insert()*/
```

```
display()
{
      struct node *ptr;
      ptr = front;
      if(front == NULL)
            printf("Queue is empty\n");
      else
      {
            printf("Queue is :\n");
            printf("Priority Item\n");
            while(ptr != NULL)
            {
                  printf("%5d %5d\n",ptr->priority,ptr->info);
                  ptr = ptr->link;
            }
      }/*End of else */
}/*End of display() */
```

# 1. Discuss following

## 1. Graph

- A graph G consist of a non-empty set V called the set of nodes (points, vertices) of the graph, a set E which is the set of edges and a mapping from the set of edges E to a set of pairs of elements of V.
- It is also convenient to write a graph as G=(V,E).
- Notice that definition of graph implies that to every edge of a graph G, we can associate a pair of nodes of the graph. If an edge X ∈ E is thus associated with a pair of nodes (u,v) where u, v ∈ V then we says that edge x connect u and v.

## 2. Adjacent Nodes

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

## 3. Directed & Undirected Edge

- In a graph G=(V,E) an edge which is directed from one end to another end is called a directed edge, while the edge which has no specific direction is called undirected edge.

## 4. Directed graph (Digraph)

- A graph in which every edge is directed is called directed graph or digraph.

## 5. Undirected graph

- A graph in which every edge is undirected is called undirected graph.

## 6. Mixed Graph

- If some of the edges are directed and some are undirected in graph then the graph is called mixed graph.

## 7. Loop (Sling)

- An edge of a graph which joins a node to itself is called a loop (sling).

## 8. Parallel Edges

- In some directed as well as undirected graphs, we may have certain pairs of nodes joined by more than one edges, such edges are called Parallel edges.

## 9. Multigraph

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

## 10. Weighted Graph

- A graph in which weights are assigned to every edge is called weighted graph.