Program 1.1: Implementation of Stack using Array

Theory:

• A stack is a container of objects that are inserted and removed according to the last-in-first-out (LIFO) principle.
• Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
• Inserting an item is known as "pushing" onto the stack. "Popping" off the stack is synonymous with removing an item

A stack is an abstract data type(ADT) that supports
two main methods:-

- push(o)

Inserts object o onto top of stack

- pop()

Removes the top object of stack and returns it; if stack is empty an error occurs

• The following support methods should also be defined:

- size()

Returns the number of objects in stack

- isEmpty()

Return a boolean indicating if stack is empty.

- isFull()

Return a Boolean indicating if stack is full.

- peek():

return the top object of the stack, without removing it; if the stack is empty an error occurs.

- display()

display the elements of stack from top of the stack.

Program 1.2: Implementation of two stacks in an array

Theory:

Implementation of *two Stacks* should use only one array, i.e., both stacks should use the same array for storing elements. Following functions must be supported.

push1(int x) –> pushes x to first stack
push2(int x) –> pushes x to second stack

pop1() –> pops an element from first stack and return the popped element
pop2() –> pops an element from second stack and return the popped element

Method 1: Divide the space into two equal halves
A simple way to implement two stacks is to divide the array in two halves and assign the half half space to two stacks, i.e., use arr[0] to arr[n/2] for stack 1, and arr[n/2+1] to arr[n-1] for stack 2 where arr[] is the array to be used to implement two stacks and size of array be n.
The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the array size is 6 and we push 3 elements to stack1 and do not push anything to second stack2. When we push 4th element to stack1, there will be overflow even if we have space for 3 more elements in array.

Method 2: (A space efficient implementation)
This method efficiently utilizes the available space. It doesn't cause an overflow if there is space available in arr[]. The idea is to start two stacks from two extreme corners of arr[]. stack 1 starts from the leftmost element, the first element in stack1 is pushed at index 0. The stack 2 starts from the rightmost corner, the first element in stack 2 is pushed at index (n-1). Both stacks grow (or shrink) in opposite direction.

Post Lab Questions:
1) What do you mean by a pointer pointing to structure?
2) Write syntax for creating an array of 20 float values in a static and dynamic manner

Program 2.1: Evaluation of postfix expression

Theory:

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are –

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression.

a) Infix Notation

We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

b) Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

c) Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

The following table briefly tries to show the difference in all three notations –

Sr. No.	Infix Notation	Prefix Notation	Postfix Notation
1)	a + b	+ a b	a b +
2)	(a + b) ∗ c	∗ + a b c	a b + c ∗
3)	a ∗ (b + c)	∗ a + b c	a b c + ∗
4)	a / b + c / d	+ / a b / c d	a b / c d / +

Sr. No.	Infix Notation	Prefix Notation	Postfix Notation
5)	(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗
6)	((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

Algorithm to evaluate a postfix expression:

1) Scan the given expression from left to right and read each character.
2) If the read character is operand, push it to Stack.
3) If the read character is operator (+ , - , * , / etc.,), perform pop operation twice and store the two popped operands in variables operand2 and operand1 respectively. Then perform operation indicated by read character on operand1 and operand2
4) Push the output of step 3 back to stack.
5) After reaching to the end of postfix expression perform a pop operation on stack and display the popped value as final result.

Example: For the postfix Expression 4 5 + 7 2 - * working of algorithm is as shown in figure 2.1

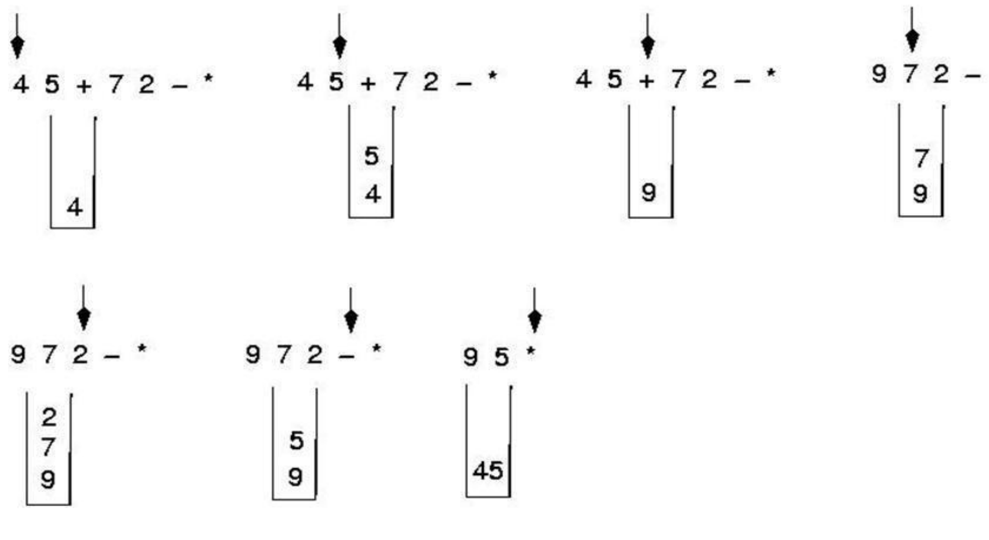


Figure 2.1 Evaluation of postfix expression

Program 2.2: Conversion of an expression from Infix to Postfix (reverse polish notation)

Theory:

Advantage of Postfix Expression over Infix Expression:

An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence).Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

Algorithm to convert expression from Infix to Postfix:

X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

```
1) Push “(“onto Stack, and add “)” to the end of X.
2) Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
3) If an operand is encountered, add it to Y.
4) If a left parenthesis is encountered, push it onto Stack.
5) If an operator is encountered, then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the
       same precedence as or higher precedence than operator.
    2. Add operator to Stack.
   [End of If]
6) If a right parenthesis is encountered, then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left
       Parenthesis is encountered.
    2. Remove the left Parenthesis.
   [End of If]
7) END.
```

Consider the Infix Expression: A+ (B*C-(D/E^F)*G)*H, where ^ is an exponential operator. Working of algorithm is as shown in figure 2.2

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack , that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack , that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Figure 2.2 Conversion of an expression from Infix to Postfix

Resultant Postfix Expression is: ABC*DEF^/G*-H*+

Program 2.3: Conversion of an expression from Infix to Prefix (Polish notation)

Theory:

Algorithm for Infix to prefix conversion

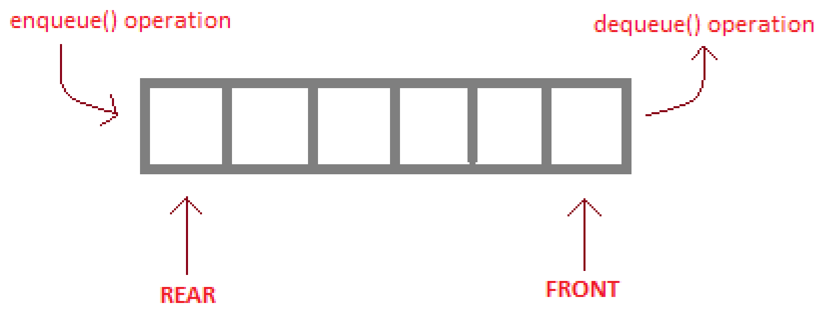
1) Read Infix expression from user.
2) Reverse the infix expression.
3) Make every '(' as ')' and every ')' as '(' to adjust parenthesis after reversal.
4) Convert expression to postfix form using infix to postfix algorithm.
5) Reverse the postfix expression generated in step 4 and print it.

Post Lab Questions:
1) Explain recursion as an application of stack.

Program 3.1: Static implementation of linear queue.

Theory:
Queue is also an abstract data type or a linear data structure, in which the elements are inserted from one end called **REAR**(also called tail), and existing elements are deleted from the other end called **FRONT**(also called head). This makes queue as FIFO(First in First Out) data structure, which means that element inserted first will also be removed first.

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



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

QUEUE DATA STRUCTURE

Algorithm:

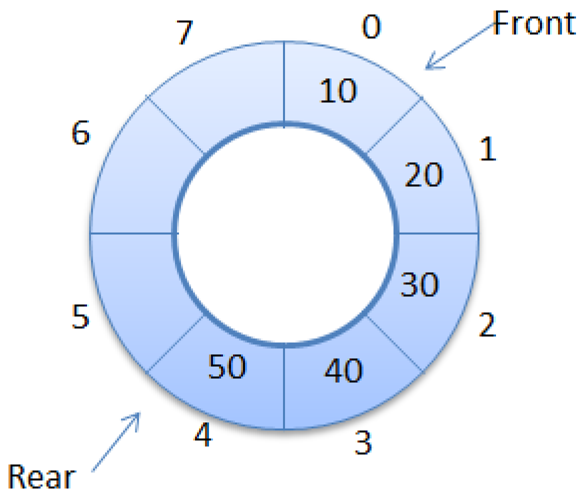
```
1) START
2) Initialize front to 0 and rear to -1
3) Algorithm for enqueue()

   1.   If (REAR  = MAX-1) then
        a. Display "Queue overflow";
        b. Return;
   2.   Otherwise
        a. REAR := REAR + 1;
        b. QUEUE(REAR) := ITEM;
   3.   Return;
4)Algorithm for dequeue()

   1.   If (REAR < FRONT) then
```

```
        a. Display “Queue underflow”;
        b. Return -1;
     2.    Otherwise
        a. ITEM := QUEUE(FRONT);
        b. FRONT := FRONT + 1;
     3.    Return ITEM;
5) Algorithm for display()
     1. if(REAR<FRONT)
        a. Display “Queue is empty”;
        b. Return;
     2.    Otherwise
        For  i=front till rear print QUEUE(i)
6) STOP
```

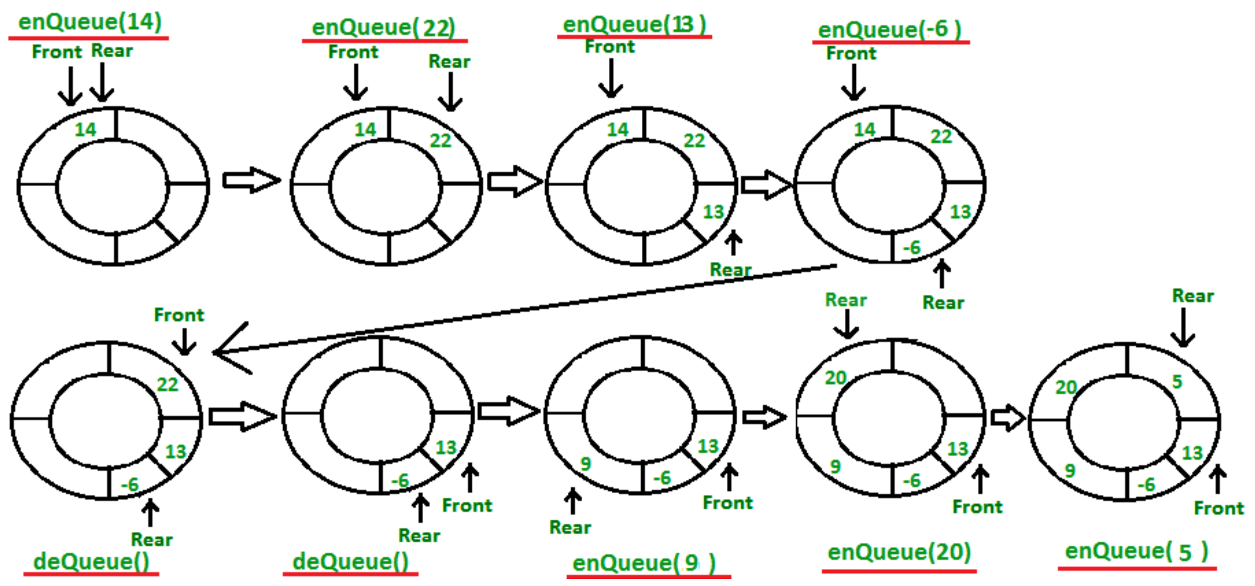
Program 3.2: Static implementation of circular queue

Theory:

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called **'Ring Buffer'**.



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.



Operations on Circular Queue:

- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.
- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.
- **display()-**This function is used to display element of the circular queue.

Algorithm:

```
1) START
  2) Initialize front and rear to -1
  3) Algorithm for enqueue(ITEM)
    i.     If (REAR  = MAX-1 and FRONT= 0) OR (REAR+1=FRONT) then
              Display " Queue overflow"
           Else if FRONT=-1 AND REAR=-1
                  a)FRONT=0
                  b) REAR=0
                  c) QUEUE(REAR) := ITEM;
           Else if REAR=MAX-1 then
                  a)   REAR=0
                  b)  QUEUE(REAR) := ITEM;
           else
              a)REAR := REAR + 1;
              b)  QUEUE(REAR) := ITEM;

  4)Algorithm for dequeue()

    i.     If ( FRONT=-1 AND REAR=-1) then
      a. Display "Queue underflow";
      b. Return -1;
    ii.   else  if ( FRONT=REAR) Then
          a)   x=QUEUE(FRONT)
          b)   FRONT=REAR=-1
          c)   Return x
    iii.   Else if (FRONT= MAX-1) then
              a)   x=QUEUE(FRONT)
              b)   FRONT=0
              c)   Return x
           Else
   x=QUEUE(FRONT)
              a)   FRONT++
              b)   Return x
              c)
 5) Algorithm for display()
    1. if(REAR=-1 AND FRONT=-1)
      a. Display "Queue is empty";
      b. Return;
    2. If (FRONT<REAR)
      Print element from FRONT index till REAR index
  3. if (FRONT>REAR)
      a. Print elements from FRONT index till MAX -1 index
      b. Print elements from 0 th index till REAR index

          6) STOP
```

Post Lab Questions:

1) Describe applications of queue data structures.

2) Given a queue Q containing n elements, transfer these items on to a stack S (initially empty) such that front element of Q appears at the top of the stack.

Expt No: 4

Aim: Implementation of Linked List

Theory:

A **linked list** is a data structure consisting of group of nodes which together represent a sequence. Under the simplest form, each node is composed of a data and a reference (in other words, a *link*) to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence.

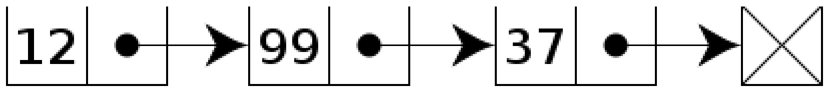


Fig: A linked list whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to a terminator used to signify the end of the list.

Advantages of linked list over array:

1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached. Whereas linked list are dynamic and memory is created as and when needed.

2) Inserting a new element in an array of elements is expensive; because room has to be created for the new elements and to create room existing elements needs to be shifted.

3) Deletion is also expensive with arrays until unless some special techniques are used.

4) So, Linked list provides following two advantages over arrays:
   a) Dynamic size
   b) Ease of insertion/deletion

Drawbacks of Linked lists:

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

2) Random access is not allowed. Nodes in a linked list must be read in order from the beginning as linked lists are inherently sequential access.

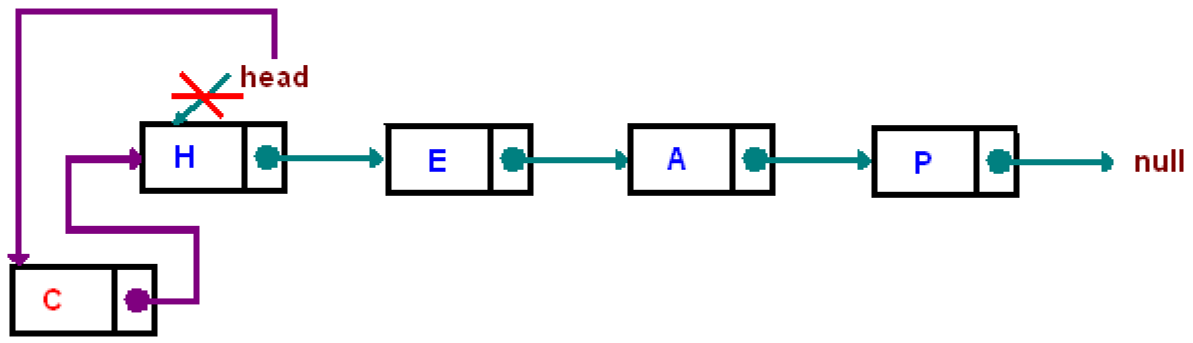
3) Difficulties arise in linked lists when it comes to traverse them in reverse direction. Singly linked lists are extremely difficult to navigate backwards, and while doubly linked lists are somewhat easier to read, memory is wasted in allocating space for a back pointer.

4) Arrays have better cache locality that can make a pretty big difference in performance.

Linked List Operations

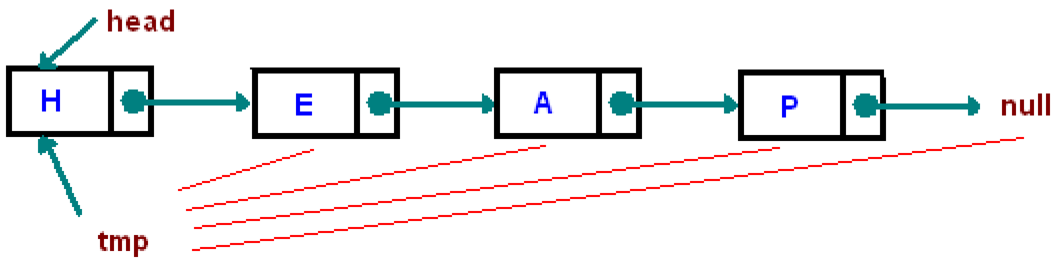
1) Add new node in the beginning Insert_First()

The function creates a node and adds it in the beginning of the list.



2) Traversing

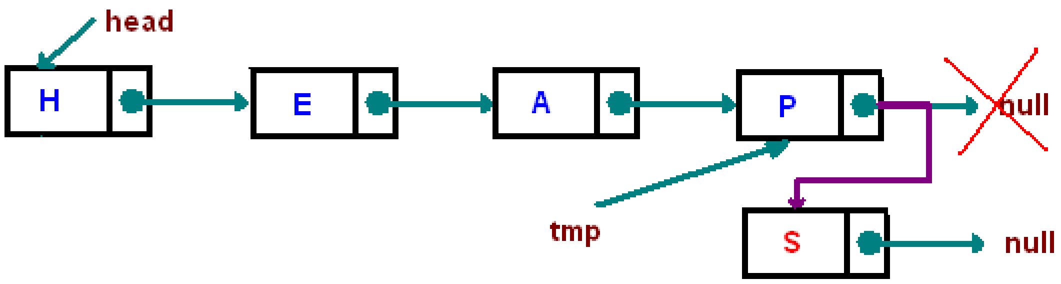
Start with the head and access each node until you reach null. Do not change the head reference.



```
Node q = head;
while(q != null)
  q = q ->next;
```

3)addLast

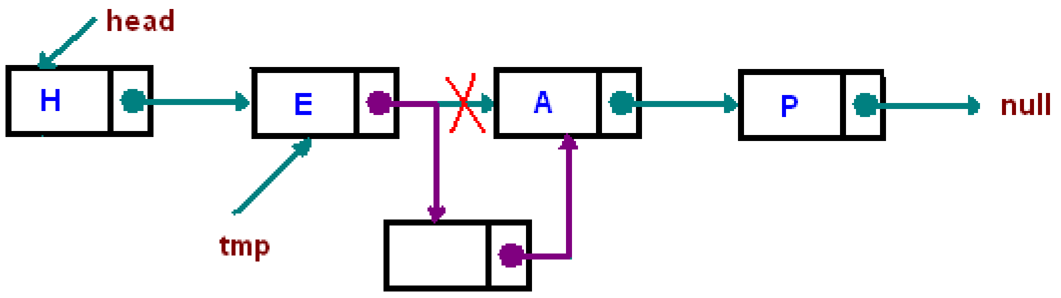
The method appends the node to the end of the list. This requires traversing, but make sure you stop at the last node



1. Traverse until last node is found.
2. Attach last node to newly created node by putting address of newly created node in next part of last node. Now newly created node becomes last node

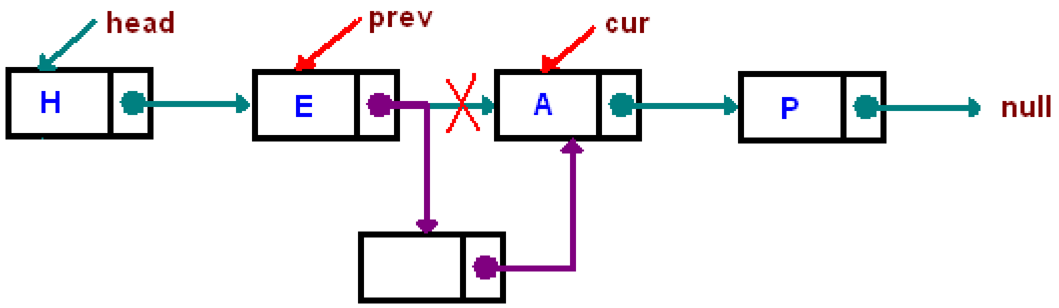
4) Inserting "after"

Find a node containing "key" and insert a new node after it. In the picture below, we insert a new node after "e":



5) Inserting "before"

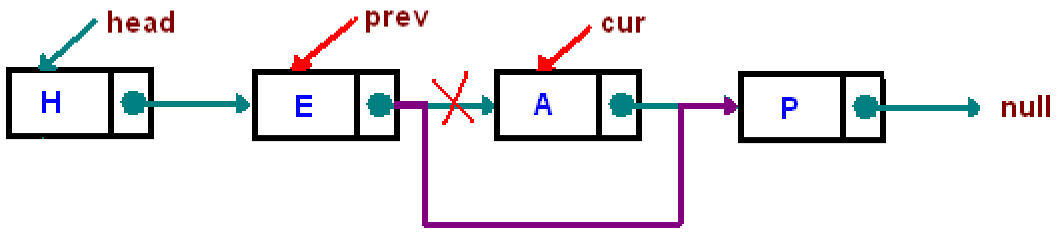
Find a node containing "key" and insert a new node before that node. In the picture below, we insert a new node before "a":



For the sake of convenience, we maintain two references prev and cur. When we move along the list we shift these two references, keeping prev one step before cur. We continue until cur reaches the node before which we need to make an insertion. If cur reaches null, we don't insert, otherwise we insert a new node between prev and cur.

6) Deletion

Find a node containing "key" and delete it. In the picture below we delete a node containing "A"



The algorithm is similar to insert "before" algorithm. It is convinient to use two references prev and cur. When we move along the list we shift these two references, keeping prev one step before cur. We continue until cur reaches the node which we need to delete. There are three exceptional cases we need to take care of:

1. list is empty
2. delete the head node
3. node is not in the list

Supporting methods are :

7) Display

Traverse all nodes of Linked List, display data part of nodes

8) Count : Traverse all nodes of Linked List, while traversing count the nodes, return count

9) Reverse. 3 pointers to accomplish reversal.

```
p = null;
q = start node of LL
until end node is reached using q
        set r as next node of q
        make q next part as p // q points to its previous node p
        set q as p
        set r as q
```

Post Lab questions:

1) Write a function to remove all nodes having duplicate information from a linked list

Program 5.1: Sparse matrix implementation using linked list

Theory:

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 values**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Example:

```
0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0
```

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

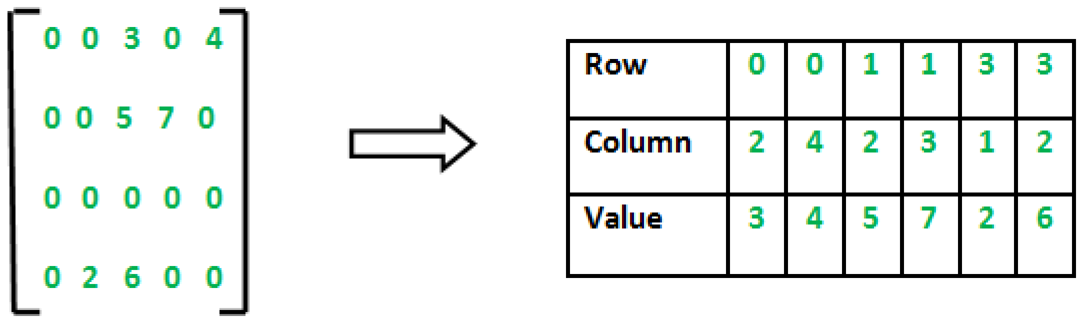
Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

Method 1: Using Arrays

2D array is used to represent a sparse matrix in which there are three rows named as

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)

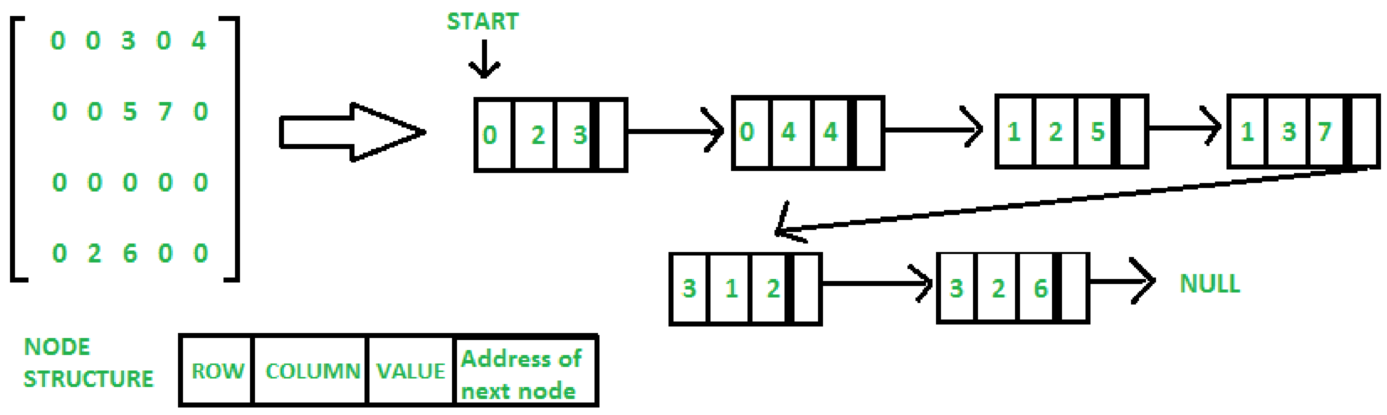


Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Method 2: Using Linked Lists

In linked list, each node has four fields. These four fields are defined as:

- Row: Index of row, where non-zero element is located
- Column: Index of column, where non-zero element is located
- Value: Value of the non zero element located at index – (row,column)
- Next node: Address of the next node



Program 5.2: Polynomial Manipulation using linked list

Theory:

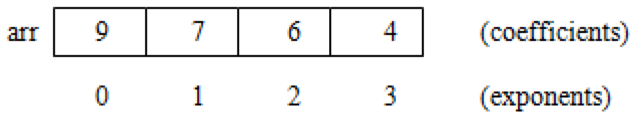
Polynomial Manipulation

- Representation
- Addition
- Subtraction

Representation of a Polynomial: A polynomial is an expression that contains more than two terms. A term is made up of coefficient and exponent. An example of polynomial is

$P(x) = 4x^3+6x^2+7x+9$

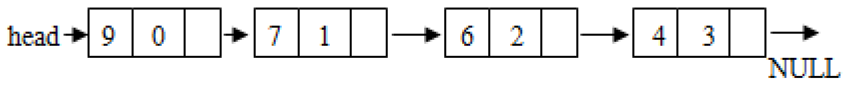
A polynomial thus may be represented using arrays or linked lists. Array representation assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the subscript of the array beginning with 0. The coefficients of the respective exponent are placed at an appropriate index in the array. The array representation for the above polynomial expression is given below:



A polynomial may also be represented using a linked list. A structure may be defined such that it contains two parts- one is the coefficient and second is the corresponding exponent. The structure definition may be given as shown below:

```
 struct polynomial
{
int coefficient;
int exponent;
struct polynomial *next;
};
```

Thus the above polynomial may be represented using linked list as shown below:

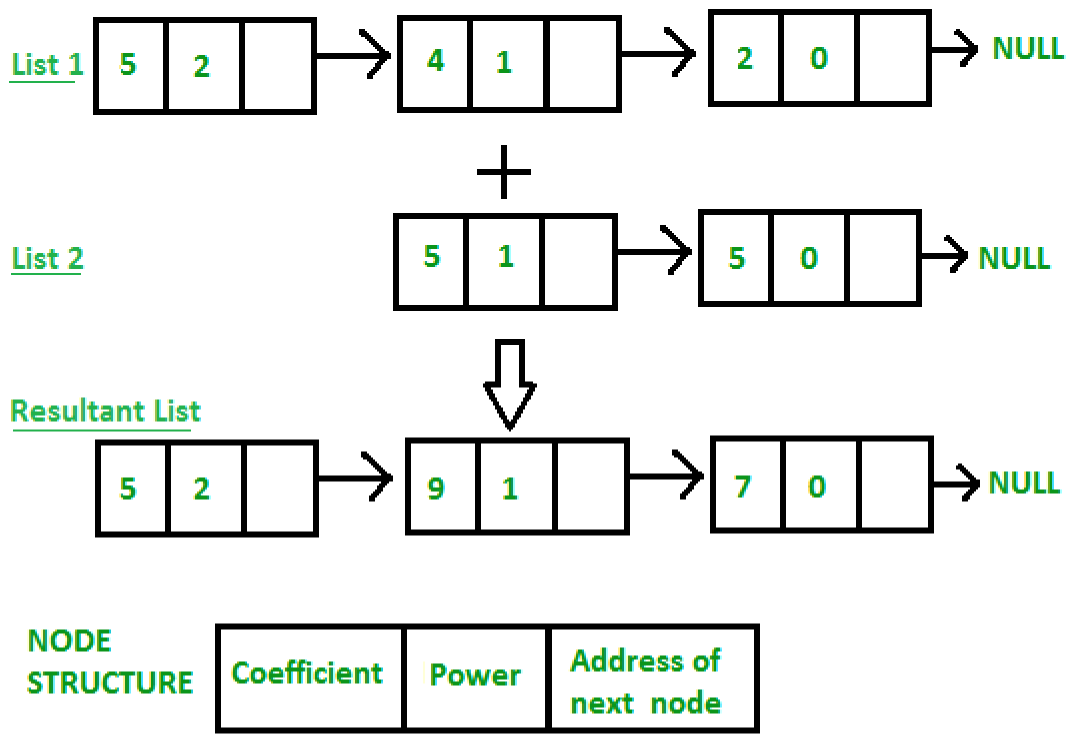


Addition of two Polynomials:

For adding two polynomials using arrays is straightforward method, since both the arrays may be added up element wise beginning from 0 to n-1, resulting in addition of two polynomials. Addition of two polynomials using linked list requires comparing the exponents, and wherever the exponents are found to be same, the coefficients are added up. For terms with different exponents, the complete term is simply added to the result thereby making it a part of addition result.

Example:

```
Input:
    1st polynomial = 5x^2 + 4x^1 + 2x^0
    2nd polnomial = 5x^1 + 5x^0
Output:
      5x^2 + 9x^1 + 7x^0
Input:
    1st polynomial = 5x^3 + 4x^2 + 2x^0
    2nd polynomial = 5x^1 + 5x^0
Resultant polynomial:
      5x^3 + 4x^2 + 5x^1 + 7x^0
```



Expt No: 6.1

Aim: Dynamic implementation of Stack

Theory:

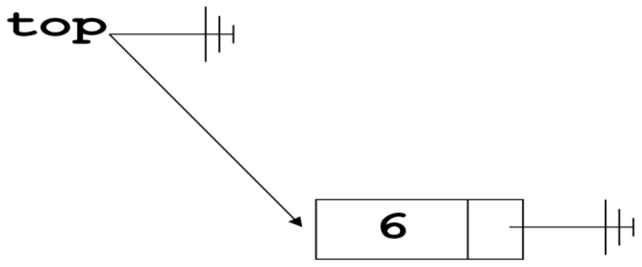
The disadvantage of a stack implemented using an array is that its size is fixed and needs to be specified at compile time. This stack implementation is often impractical. To make the size of the stack to be dynamic and determined at run-time, we need to implement it using a linked list. By doing this, the size of the stack can shrink or grow on demand during the program execution. A stack implemented using a linked list is also known as a linked stack.

- ➔ Number of elements in stack is not restricted to certain size.
- ➔ Dynamic memory creation, memory will be assigned to stack when a new node is pushed into stack, and memory will be released when an element being popped from the stack.
- ➔ Stack using linked list implementation can be empty or contains a series of nodes.
- ➔ each node in a stack must contain at least 2 attributes:

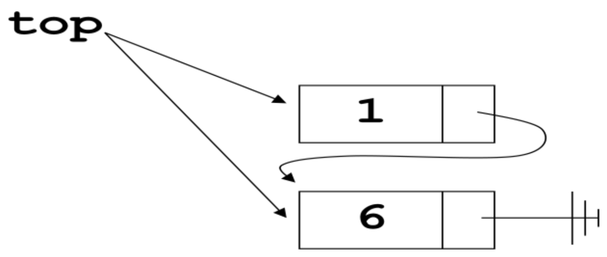
  data – to store information in the stack.

  pointer next (store address of the next node in the stack)

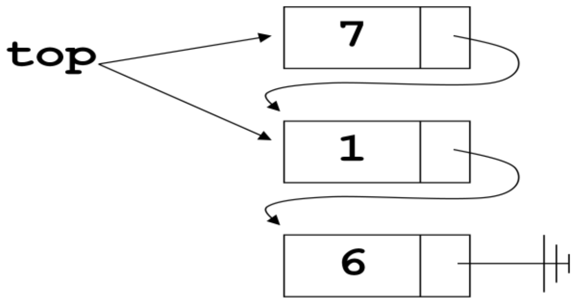
1. Initially top is Null
2. Then pushing first node pointed by top



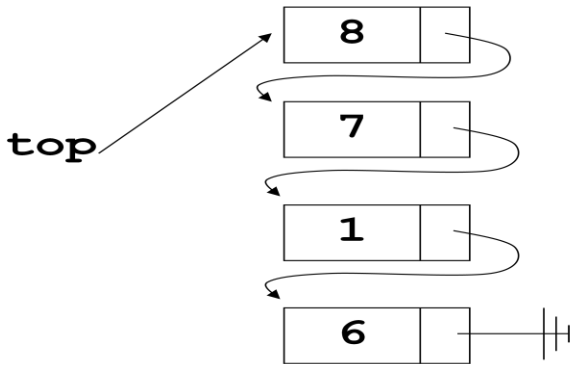
Pushing second node. Make top point to latest inserted node. And top->next pointes to previous top.



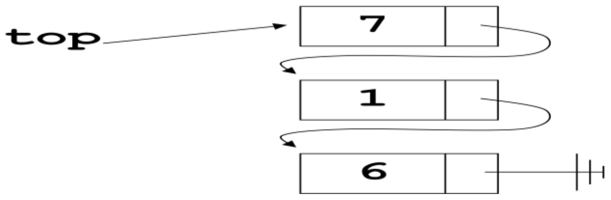
Similarly pushing third node



Similarly pushing fourth node



3. Popping element from stack
   Pop the node data pointed by top and set top to top->next



Basic operations for a stack implemented using linked list:

- createStack()
    - initialize top
- push(ele)
    - insert item onto stack
- pop()
    - delete item from stack
- isEmpty()
    - check whether a stack is empty.
- peek()
    - get item at top

isFull() operation is not needed since elements can be inserted into stack without limitation to the stack size.
In Circular implementation of stack, top points to first inserted node and top->next gives the address of last inserted node.

Expt No: 6.2

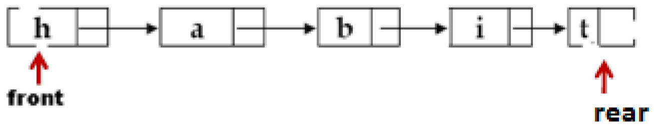
Aim: Dynamic implementation of Queue

Theory:

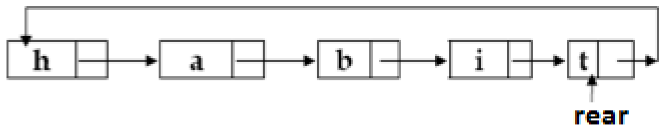
Pointer Based Implementation

• Can be implemented using linear linked list or circular linked list.

• Linear linked list Need two external pointers (front and rear)



- Circular linked list needs only one pointer, pointing to rear end.



Basic operations for a queue implemented using linked list:

- createQueue()
  - Initialize front and rear pointer
- destroyQueue()
  - delete all nodes of queue
- isEmpty()
  - return true if Queue is empty
- enQueue()
  - insert element in Queue at rear
- deQueue()
  - delete element from Queue from front end

Algorithm:

1) Algorithm for enqueue:

```
i) Create a new node using malloc(). pointer p is holding address of new node
ii) If REAR==NULL
        a) REAR=p
        b) FRONT=p
  else
      a) Traverse list till last node is reached and add new node as a last node of linked list
```

```
   b) Make rear point to this new node by writing REAR=p
```

Algorithm for dequeue:

```
1) If FRONT==NULL
      print "list is empty"
else
   a) Update FRONT pointer to go to next node by writing
      FRONT=FRONT->NEXT
   b) free first node
```

Post lab questions:

1) Write a function to print minimum and maximum element of a linked list.

Expt No: 7.1

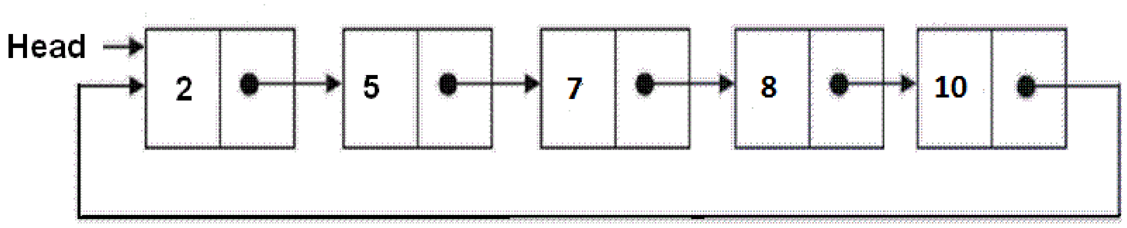
Aim: Implementation of Circular Linked List (CLL)

Theory :

Why Circular linked list is needed?

In a singly linked list, for accessing any node of linked list, we start traversing from the first node. If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node. This problem can be solved by slightly altering the structure of singly linked list. In a singly linked list, next part (pointer to next node) is NULL, if we utilize this link to point to the first node then we can reach preceding nodes.

The structure thus formed is circular singly linked list look like this:



Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.

Advantages of Circular Linked List:

1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

2) Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

3) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

Three basic operations can be performed on CLL

1) Insertion- adds a new node to CLL:

A node can be added in four ways:

- Insertion at the beginning of the list

- Insertion at the end of the list
- Insertion after a specific nodes
- insertion before a specific node

2) Deletion- removes a node from CLL

A node can be deleted in four ways:

- Deletion from the beginning of the list
- Deletion from the end of the list
- Deletion after a specific nodes
- Deletion before a specific node

3) Traversal- display contents of each node in CLL

Algorithm:

1) Insertion of new node in the beginning:

```
1. Create a new node using malloc(). pointer p is holding address of new node.
2. SET p->info=val
3. If START==NULL
    a) START=p
    b) START->next=p
4. Otherwise,
    4.1 SET ptr=START
    4.2 update ptr to point to next node by writing ptr=ptr->next till ptr->next != START
        [End of Loop]
    4.3 SET p->next=START
    4.4 SET ptr->next=START
    4.5 SET START=p
5. STOP
```

2) Insertion of new node at the end of CLL:

```
1. Create a new node using malloc(). pointer p is holding address of new node.
2. SET p->info=val
3. If START==NULL
    a) START=p
    b) START->next=p
4. Otherwise,
    4.1 SET ptr=START
    4.2 update ptr to point to next node by writing ptr=ptr->next till ptr->next != START
        [End of Loop]
    4.3 SET p->next=START
    4.4 SET ptr->next=START
```

```
5. STOP
```

3) Insertion of new node after a specific node of CLL: (key is info part of the node after which new node is to be inserted)

```
1. Create a new node using malloc(). pointer p is holding address of new node.
2. SET p->info=val
3. SET ptr=START
4. Repeat steps 5 while ptr->next!=START AND ptr->info!=key
 5. SET ptr=ptr->next
      [End of Loop]
  6. SET p->next=ptr->next
  7. SET ptr->next=p
  8. STOP
```

4) Insertion of new node before a specific node of CLL: (key is info part of the node before which new node is to be inserted)

```
1. Create a new node using malloc(). pointer p is holding address of new node.
2. SET p->info=val
3. SET ptr=START
4. SET preptr=START
 5. Repeat steps 6  and 7 while ptr->next!=START AND ptr->info!=key
 6. SET preptr=ptr
  7. SET ptr=ptr->next
      [End of Loop]
  8. SET p->next=ptr
  9. SET preptr->next=p
 10. STOP
```

5) Deleting a node from the beginning

```
1. If START==NULL
        display "underflow"
        return
2. otherwise
   2.1 SET ptr=START
   2.2 repeat step 2.3 while ptr->next!=START
   2.3 SET ptr= ptr->next
        [END of Loop]
   2.4 SET ptr->next= START->next
   2.5 free START
   2.6 SET START=ptr->next
3. STOP
```

6) Deleting a node from the end of CLL

```
1. If START==NULL
        display "underflow"
        return
2. Otherwise
  2.1 SET ptr=START
  2.2 repeat step 2.3 and 2.4 while ptr->next!=START
  2.3 SET preptr=ptr
  2.4 SET ptr= ptr->next
      [END of Loop]
  2.5 SET preptr->next= START
  2.5 free ptr
3. STOP
```

7) Deleting a node after a given node from CLL (node to be deleted is the one after a node having key as info)

```
1. If START==NULL
        display "underflow"
        return
2. Otherwise
  2.1 SET ptr=START
  2.2 SET preptr=START
  2.3 repeat step 2.4 and 2.5 while preptr->info!=key
  2.4 SET preptr=ptr
  2.5 SET ptr= ptr->next
      [END of Loop]
  2.6 SET preptr->next= ptr->next
  2.7 if(ptr=start)
      2.7.1  START=preptr->next
  2.8 free ptr
3. STOP
```

8) Traversing a circular linked list

```
1. If START==NULL
        display "underflow"
        return
2. Otherwise
  2.1 SET ptr=START
  2.2 repeat step 2.3 and 2.4 while ptr!=START
  2.3 PRINT ptr->info
  2.4 SET ptr= ptr->next
      [END of Loop]
  3. STOP
```

Expt No: 7.2

Aim: Implementation of stack using Circular Linked List (CLL)

Theory: Stack is a LIFO data structure in which element inserted last is denoted by top. In CLL implementation of stack external pointer top will point to a node inserted last in a linked list.

1) push() operation on a stack is implemented by adding a new node before a current top most node.

2) pop() operation on a stack is implemented by removing a node currently pointed by top pointer.

Algorithm:

1) push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack

```
1) Create a new node using malloc(). pointer p is holding address of new node.
2) SET p->info=val
3) if top=NULL    (if stack is empty)
    3.1 top=p
    3.2 top->next=p
4) otherwise      (if stack is not empty)
    4.1 SET ptr=top
    4.2 repeat step 4.3 while ptr->next!=top (traverse till last node)
    4.3 ptr=ptr->next
    4.4 ptr->next=p        (Make last node point to first node by updating next field of last node)
    4.5 p->next=top        (insert new node before current top node)
    4.6 top=p              (update pointer top to point to newly inserted node)
```

2) pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack.

1) Check whether **stack** is **Empty** (**top == NULL**).

2) If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and return

3) If it is **Not Empty**, then define a **Node** pointer '**ptr**' and set it to '**top**'.

```
    3.1  Repeat step 3.2 while ptr->next!=top (traverse till last node)
    3.2 SET ptr =ptr->next
    3.3 SET ptr->next=top->next (Make last node point to second node by updating next field of
                                 last node)
    3.4 top=top->next       (Make top point to second node as we are deleting first node)
    3.5 free(ptr->next)     (delete first node)
4) STOP
```

3) display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

1) Check whether stack is **Empty** (**top** == **NULL**).
2) If it is **Empty**, then display **'Stack is Empty!!!'** and return to caller.
3) If it is **Not Empty**, then define a Node pointer **'ptr'** and initialize with **top**.
4) Display '**ptr → info**--->' and move it to the next node. Repeat the same until **ptr** reaches to the first node in the stack (**ptr → next** != **top**).
5) Finally! Display '**ptr → info**. (to print info of last node)

Expt No: 7.3

Aim: Implementation of queue using Circular Linked List (CLL)

Theory: Queue is a FIFO data structure in which elements are inserted at the rear end and removed from the front end. In dynamic implementation of Queue we need two pointers one pointing to first node and the other pointing to last node. Hence we use **front** and **rear** pointer.
1) insert () operation on a queue is implemented by adding a new node at the end (i.e. after a node pointed by rear pointer)
2) delete() operation on a queue is implemented by deleting first node of a linked list (the one which is pointed by front pointer)

Algorithm:

1) Insert operation on a queue implemented using CLL:

```
1. Create a new node using malloc(). pointer p is holding address of new node.
2. SET p->info=val
3. If front==NULL
     a) front=p
     b) rear=p
     b) rear->next=p
4. Otherwise,
   4.1 SET p->next= front
   4.2 SET rear->next=p
   4.3  SET rear=p
5. STOP
```

2) Delete operation on a queue implemented using CLL

We can use the following steps to delete a node from the stack.

```
1) Check whether queue is Empty (front == NULL).
2) If it is Empty, then display "Queue overflow!!! Deletion is not possible!!!" and return
3) If it is Not Empty, then
   3.1 check whether front is equal to rear( i.e. only 1 element in a queue)
       3.1.1  SET front =NULL
       3.1.2  SET rear= NULL
   3.2 otherwise
       3.2.1 SET ptr=front
       3.2.2 SET front= front->next
       3.2.3 SET rear->next=front
       3.2.4 free(ptr)
```

3) display() - Displaying elements of queue

We can use the following steps to display the elements (nodes) of a stack...

1) Check whether queue is **Empty** (**front** == **NULL**).
2) If it is **Empty**, then display **'Queue underflow!!!'** and return to caller.
3) If it is **Not Empty**, then define a Node pointer **'ptr'** and initialize with **front**.
4) Display '**front → info**--->' and move it to the next node. Repeat the same until **ptr** reaches to the last node in the queue (**ptr** != **rear**).
5) Finally! Display '**ptr → info**. (to print info of last node)

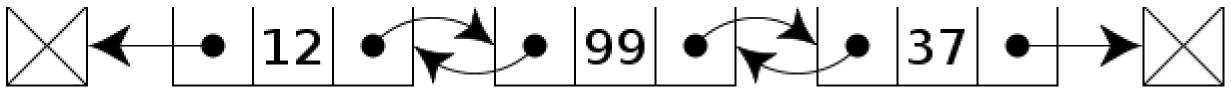
Post Lab Questions:
1) Write a function to count number of occurrences of a given value in circular linked list
2) Write advantages of circular linked list over singly linked list

Expt No: 8

Aim: Implementation of doubly linked list

Theory:

A **doubly-linked list** is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called *links*, which are references to the previous and to the next node in the sequence of nodes. The beginning and ending nodes' **previous** and **next** links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list. If there is only one sentinel node, then the list is circularly linked via the sentinel node. It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.



A doubly-linked list whose nodes contain three fields: an integer value, the link to the next node, and the link to the previous node.

The two node links allow traversal of the list in either direction. While adding or removing a node in a doubly-linked list requires changing more links than the same operations on a singly linked list, the operations are simpler and potentially more efficient (for nodes other than first nodes) because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified.

```
typedef struct node_type {
        ValueT value ;  /* data*/
        struct node_type *next ; /* pointer to next node */ struct
        node_type *prev; /* pointer to previous node */

} NodeD;
```

The operations possible on doubly-linked list are (same as singly linked list): •
insert a new node in the beg
- insert a new node in the end
- insert a new node after a specific node
- insert a new node before a specific node •
Delete a node from beg
- Delete a node from end
- Delete a node after a specific node

- Delete a node before a specific node •
  Display all nodes

Post Lab Questions:
1) What are the advantages and disadvantages of doubly linked list over singly linked list

Expt No: 9

Aim: Implementation of Priority Queue

Theory:

A priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed

Lower the number, Higher the priority!

Priority

0 -Highest

1 -High

2 -Medium

The general rules of processing are:

- An element with higher priority is processed before an element with a lower priority.
- Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

A priority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest priority is retrieved first. The priority of the element can be set based on various factors. Priority queues are widely used in operating systems to execute the highest priority process first.

When a Priority Queue is implemented using a linked list, then every node of the list will have three parts:

(a) the information or data part

(b) the priority number of the element, and

(c) the address of the next element.

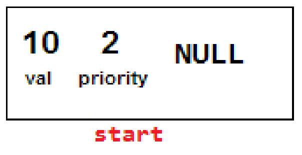
If we are using a sorted linked list, then the element with the higher priority will come before the element with the lower priority. (for processing it before)

Algorithm:

1) Inserting an element in a priority queue

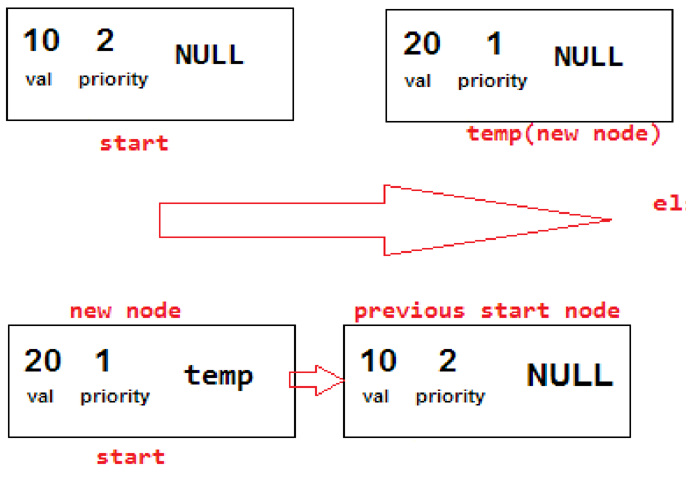
When a new element has to be inserted in a priority queue, we have to traverse the entire list until we find a node that has a priority lower than that of the new element. The new node is inserted before the node with the lower priority. However, if there exist an element that has the same priority as the new element, the new element is inserted after that element based on on **FCFS Rule**.

1st Node: val-10 pr-2



```
if(start==NULL)
        start = temp;
```

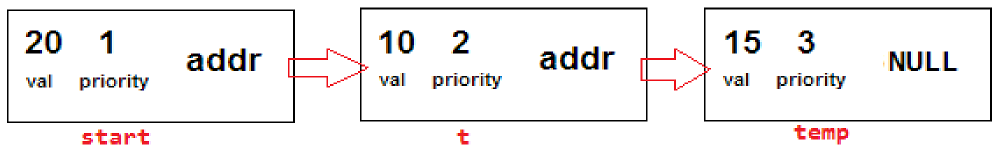
2nd Node: val-20 pr-1



```
else if(start->pr>pri)
     {
          temp->next=start;
          start=temp;
     }
```

3rd Node: val-15 pr-3

```
//here (t->next becomes NULL)
```



```
          t=start;
          while(t->next!=NULL && (t->next)->pr<=pri )
                t=t->next;
          temp->next=t->next;
          t->next=temp;
      }
```

2) Deleting an element from priority queue

Deletion is a very simple process in this case. The first node of the list needs to be deleted so that data of that node gets processed first.

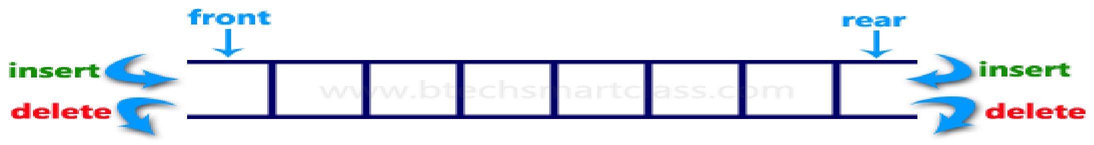
Post lab Questions: 1) Give examples of problems where priority queue is useful.

Expt No. 10.1 & Expt No 10.2

Aim: Static Implementation of Deque and dynamic implementation of Deque

Theory :

Double Ended Queue is a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.



Double Ended Queue can be represented in TWO ways, those are as follows:

1. Input Restricted Double Ended Queue : the insertion operation is performed at only one end and deletion operation is performed at both the ends.

2. Output Restricted Double Ended Queue: the deletion operation is performed at only one end and insertion operation is performed at both the ends.

Operations on Deque:

Mainly the following four basic operations are performed on queue:

insetFront(): Adds an item at the front of Deque.

insertRear(): Adds an item at the rear of Deque.

deleteFront(): Deletes an item from front of Deque.

deleteRear(): Deletes an item from rear of Deque.

Circular array implementation deque

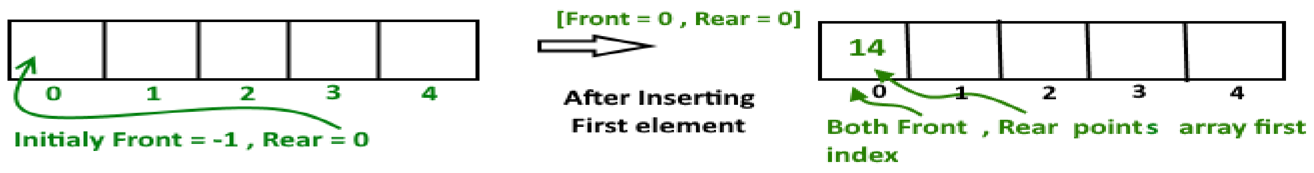
For implementing deque, we need to keep track of two indices, front and rear. We enqueue(push) an item at the rear or the front end of qedue and dequeue(pop) an item from both rear and front end.

Working :

1. Create an empty array 'arr' of size 'n'

initialize **front = -1** , **rear = -1**

Inserting First element in deque either front end or read end they both lead to the same result.


0
1
2
3
4
Initialy Front = -1 , Rear = 0
[Front = 0 , Rear = 0]
After Inserting
First element
14
Both Front , Rear points array first index

After insert **Front** Points to 0 and **Rear** points to 0

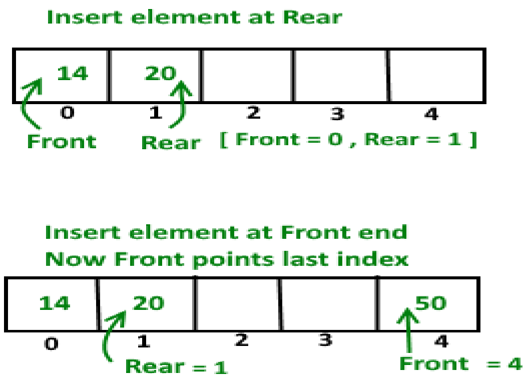
Algorithm:

1) Insert Elements at Rear end

```
a) First we check deque if Full or Not . if it is not full go to step b otherwise return
b) IF Rear == Size-1
        then reinitialize Rear = 0 ;
    Else
        increment Rear by '1'
c) Store value into Arr[ Rear ] = val
```

2) Insert Elements at Front end

```
a)  First we check deque if Full or Not. if it is not full go to step b otherwise return
b). IF Front = 0 || initial position
         move Front to point to last index of array
    Else decremented front by '1'
c) Store value into Arr[ Front ] = val
```

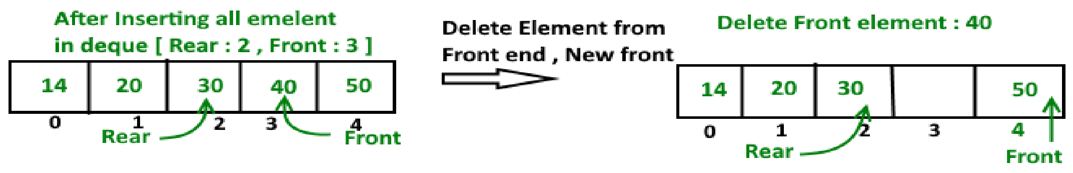


3) Delete Element From Rear end

```
a) Check whether deque is Empty or Not.  If empty return otherwise go to step b
b) Print the element at index Rear
c) If deque has only one element i.e.(front=rear)
        SET front = -1 ; rear =-1 ;
   Else IF Rear points to the first index of array then
        move Rear to last index of the array
   Else
        Decrement Rear by '1'
```

4) Delete Element From Front end

```
a) Check whether deque is Empty or Not.  If empty return otherwise go to step b
b) Print the element at index Front
c) If deque has only one element i.e.(front=rear)
        SET front = -1 ; rear =-1 ;
   Else IF front points to the last index of the array then move front to point to first index of
        array
   Else increment Front by '1'
```



Post Lab Questions:

1) Discuss real world application of Deque.

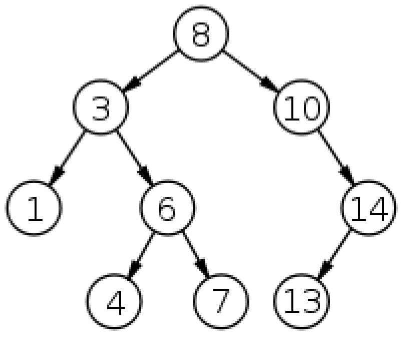
Expt No. 11.1

Aim: Implementation of Binary Search Tree (insertion, deletion, traversal)

Theory :

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys less 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. There must be no duplicate nodes.



The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

Operations on Binary search Tree:

1) Insertion of new node
2) Deletion of existing node
3) Traversal

Algorithm:

1) Insertion of a key

A new key is always inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

```
      100                         100
    /  \       Insert 40          /   \
  20    500    --------->        20    500
 /  \                          /  \
10   30                       10   30
                                     \
                                     40
```

2) Deletion of node from BST

When we delete a node, there possibilities arise.
1) *Node to be deleted is leaf:* Simply remove from the tree.

```
          50                  50
       /     \    delete(20)    /  \
      30      70   --------->   30     70
     /  \    /  \              \    /  \
    20   40  60   80             40  60   80
```

2) *Node to be deleted has only one child:* Copy the child to the node and delete the child

```
          50                  50
       /     \    delete(30)    /  \
      30      70   --------->   40     70
        \    /  \                  /  \
         40  60   80               60   80
```

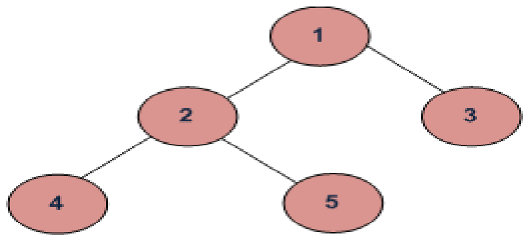
3) *Node to be deleted has two children:* Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

```
          50                      60
       /     \    delete(50)    /  \
      40      70   --------->   40    70
             /  \                  \
            60   80                 80
```

The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

3) Traversal

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Depth First Traversals:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2 3 4 5

a) Inorder Traversal:

```
Algorithm Inorder(root)
  1. Traverse the left subtree, i.e., call Inorder(left-subtree)
  2. Visit the root.
  3. Traverse the right subtree, i.e., call Inorder(right-subtree)
```

b) Postorder Traversal

```
Algorithm Postorder(root)
  1. Traverse the left subtree, i.e., call Postorder(left-subtree)
  2. Traverse the right subtree, i.e., call Postorder(right-subtree)
  3. Visit the root.
```

b) Preorder Traversal

```
Algorithm Preorder(root)
  1. Visit the root.
  2. Traverse the left subtree, i.e., call Preorder(left-subtree)
  3. Traverse the right subtree, i.e., call Preorder(right-subtree)
```

Post Lab questions:

1) How can we find minimum element from BST?

Expt No. 11.2

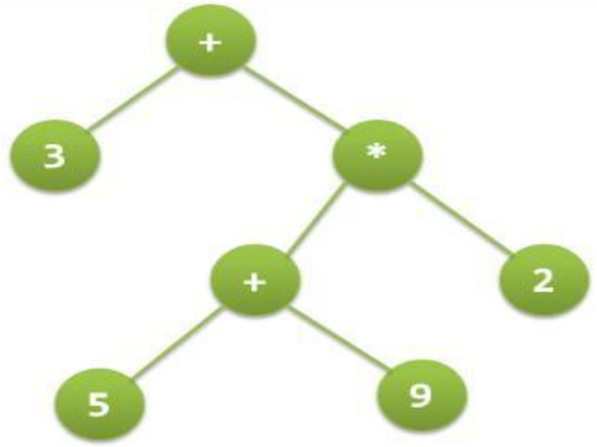
Aim: Implementation of Expression Tree

Theory:

A binary expression tree is a specific kind of a binary tree used to represent expressions. Two common types of expressions that a binary expression tree can represent are algebraic and Boolean. These trees can represent expressions that contain both unary and binary operators.

Each node of a binary tree, and hence of a binary expression tree, has zero, one, or two children. This restricted structure simplifies the processing of expression trees.

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for 3 + ((5+9)*2) would be:



Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)

Algorithm:

1) Evaluating the expression represented by expression tree:

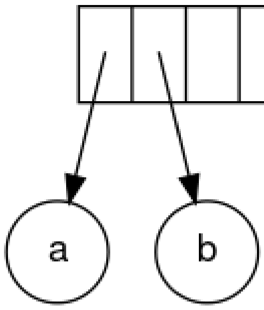
```
Let t be the expression tree
If  t is not null then
    If t.value is operand then
            Return  t.value
    A = solve(t.left)
    B = solve(t.right)
    // calculate applies operator 't.value'
    // on A and B, and returns value
    Return calculate(A, B, t.value)
```

2) Construction of Expression Tree:

The evaluation of the tree takes place by reading the postfix expression one symbol at a time. If the symbol is an operand, one-node tree is created and a pointer is pushed onto a stack. If the symbol is an operator, the pointers are popped to two trees *T1* and *T2* from the stack and a new tree whose root is the operator and whose left and right children point to *T2* and *T1* respectively is formed . A pointer to this new tree is then pushed to the Stack.

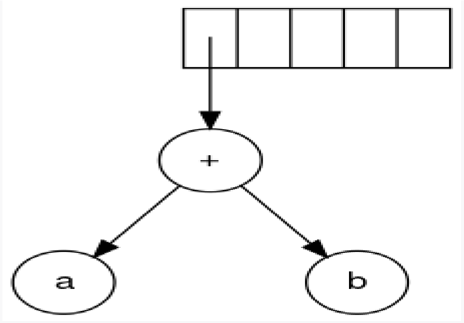
Example

The input is: a b + c d e + * * Since the first two symbols are operands, one-node trees are created and pointers are pushed to them onto a stack. For convenience the stack will grow from left to right.



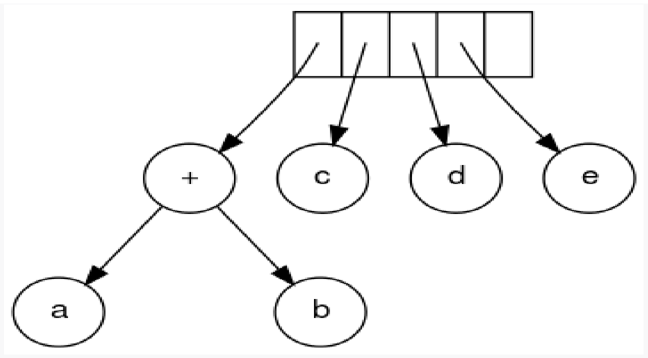
Stack growing from left to right

The next symbol is a '+'. It pops the two pointers to the trees, a new tree is formed, and a pointer to it is pushed onto to the stack.



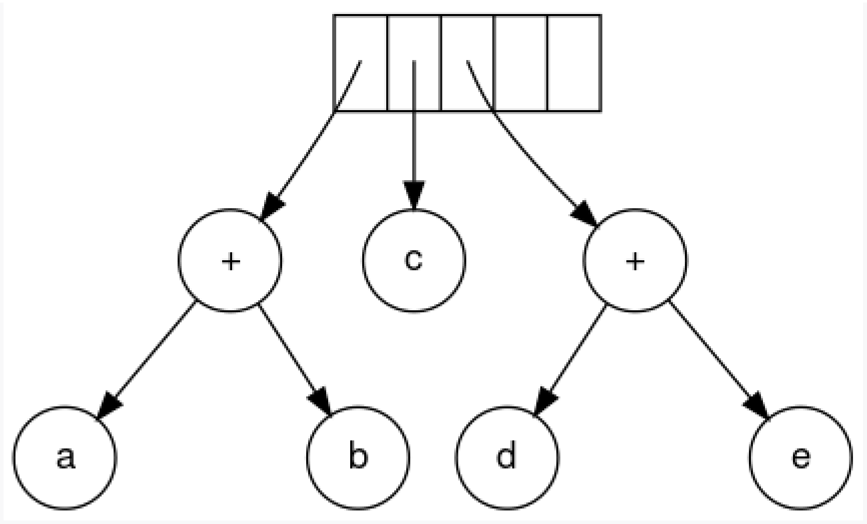
Formation of a new tree

Next, c, d, and e are read. A one-node tree is created for each and a pointer to the corresponding tree is pushed onto the stack.



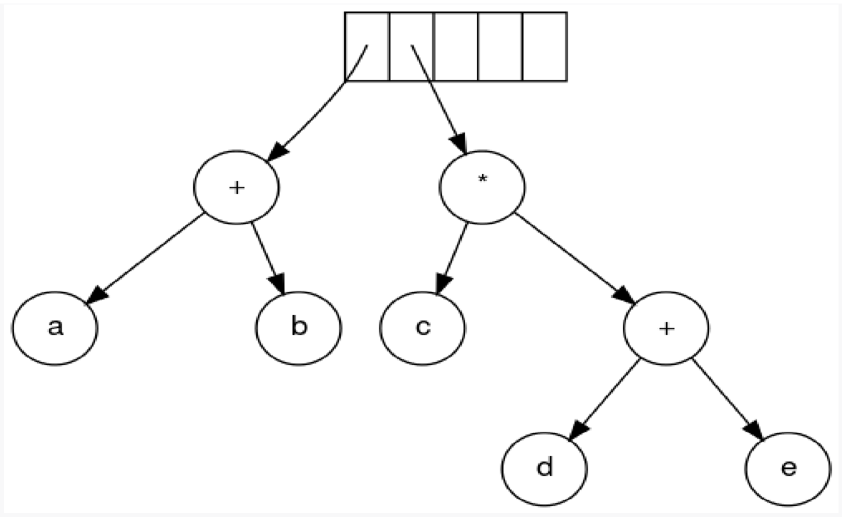
Creating a one-node tree

Continuing, a '+' is read, and it merges the last two trees.



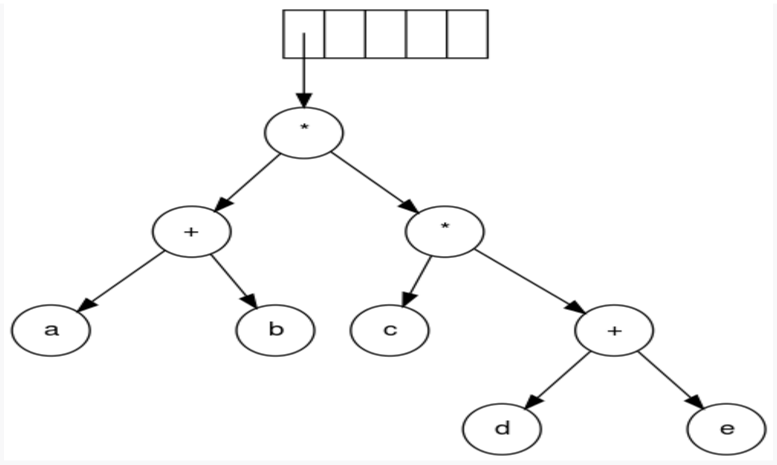
Merging two trees

Now, a '*' is read. The last two tree pointers are popped and a new tree is formed with a '*' as the root.



Forming a new tree with a root

Finally, the last symbol is read. The two trees are merged and a pointer to the final tree remains on the stack.



Steps to construct an expression tree a b + c d e + * *

Post Lab Question:

1) Discuss real world applications of expression tree

Exp No: 12

Aim: Implementation of graph and its traversals

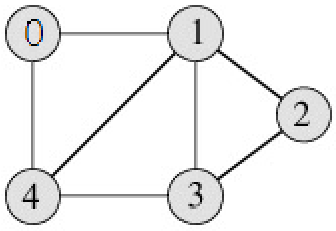
Theory:

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.

2. A finite set of ordered pair of the form (u, v) called as edge.

The pair is ordered because (u, v) is not same as (v, u) in case of directed graph(di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.



Representations of graph:

1. Adjacency Matrix

2. Adjacency List

1)Adjacency Matrix:

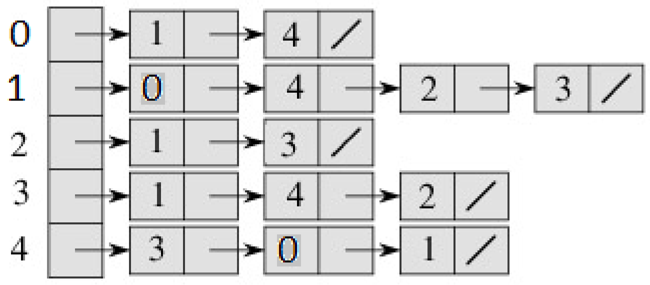
Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The adjacency matrix for the above example graph is:

$$\begin{array}{c|ccccc|} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 2 & 0 & 1 & 0 & 1 & 0 \\ 3 & 0 & 1 & 1 & 0 & 1 \\ 4 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

Adjacency Matrix Representation of the above graph

2) Adjacency List:
An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



Adjacency List Representation of the above Graph

Graph Traversal
Graph traversal is technique used for visiting each vertex of the graph exactly once. There are two graph traversal techniques:
1. DFS (Depth First Search)
2. BFS (Breadth First Search)

1) DFS (Depth First Search):

Algorithm:

1) Define a Stack whose size is number of vertices in the graph.
2) Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.
3) Visit any one of the adjacent vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.
4) Repeat step 3 until there are no new vertex to be visited from the vertex on top of the stack.
5) When there is no new vertex to be visited then use back tracking and pop one vertex from the stack.
6**)** Repeat steps 3, 4 and 5 until stack becomes Empty.

2) BFS (Breath First Search)
1) Define a Queue of size equal to total number of vertices in the graph.
2) Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
3) Visit all the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.

4) When there is no new vertex to be visited from the vertex which is at the front of the Queue then delete that vertex from the Queue.
5) Repeat step 3 and 4 until queue becomes empty.
6) When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

Post Lab Questions:

1) Discuss real world applications of graph.

Exp No 13.1

Aim: Implementation of Bubble sort, selection and Insertion sort

Theory:

Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their positions if they exist in the wrong order.

Assume that A[] is an unsorted array of n elements. This array needs to be sorted in ascending order.

The pseudo code is as follows:

We assume **list** is an array of **n** elements. We further assume that **swap**function swaps the values of the given array elements.

```
begin BubbleSort(list)

  for all elements of list
    if list[i] > list[i+1]
      swap(list[i], list[i+1])
    end if
  end for

  return list

end BubbleSort
```

Selection sort:

The Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array.

Assume that the array A=[7,5,4,2]A=[7,5,4,2] needs to be sorted in ascending order.

The minimum element in the array i.e. 22 is searched for and then swapped with the element that is currently located at the first position, i.e. 77. Now the minimum element in the remaining unsorted array is searched for and put in the second position, and so on.

```
procedure selection sort
  list  : array of items
  n     : size of list

  for i = 1 to n - 1
  /* set current element as minimum*/
```

```
    min = i

    /* check the element to be minimum */

    for j = i+1 to n
      if list[j] < list[min]
        then min = j;
     end if
       end
         for

    /* swap the minimum element with the current
    element*/ if indexMin != i  then
      swap list[min] and
    list[i] end if
 end for
end procedure
```

Insertion Sort:
Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array.

It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array. If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position. This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead

```
procedure insertionSort( A : array of items )
   int holePosition
   int valueToInsert

   for i = 1 to length(A) inclusive do:

      /* select value to be inserted */
      valueToInsert = A[i]
      holePosition = i

      /*locate hole position for the element to be inserted */

      while holePosition > 0 and A[holePosition-1] > valueToInsert do:
         A[holePosition] = A[holePosition-1]
```

```
         holePosition = holePosition -1
      end while

      /* insert the number at hole position */
      A[holePosition] = valueToInsert

   end for

end procedure
```

Exp No 13.2

Aim: Implementation of Quick sort and Merge sort

Theory:

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

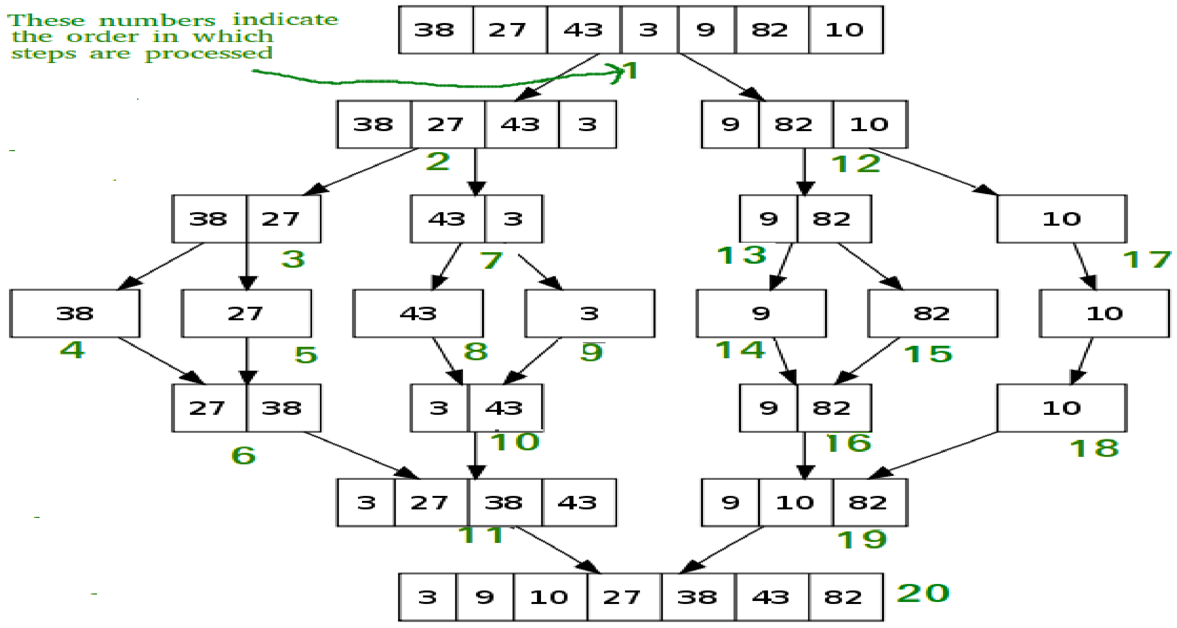
1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Merge Sort:

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

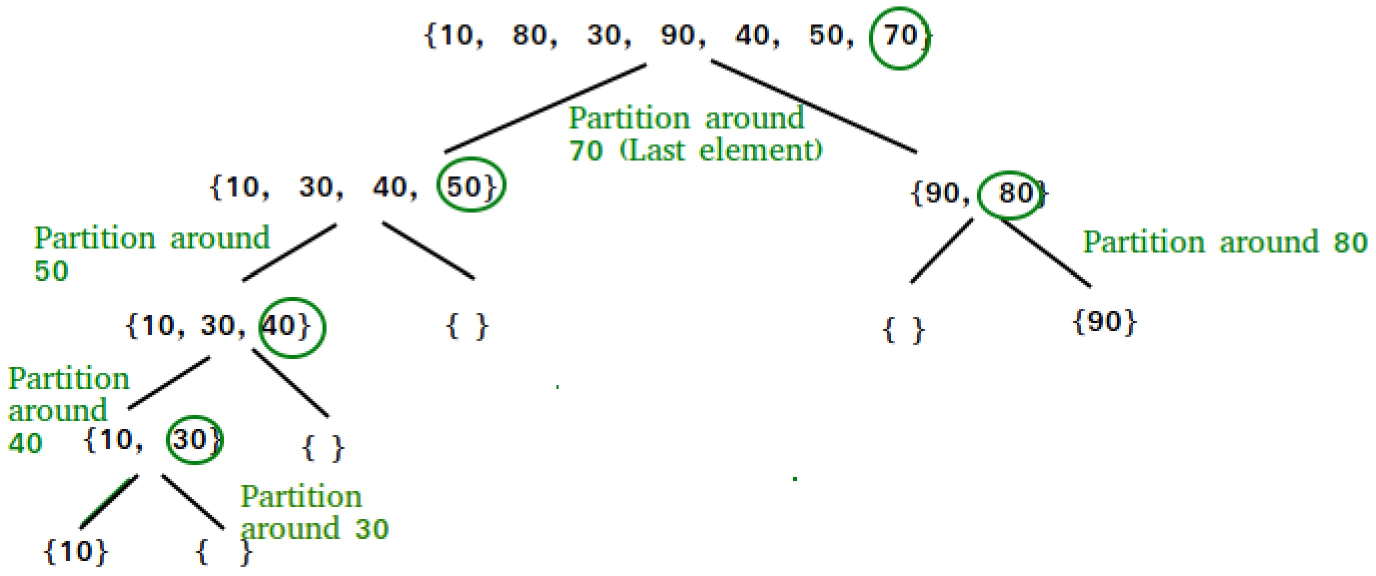
The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. The array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



Algorithm:

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```



Partition Algorithm

There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

```
/* low  --> Starting index,  high  --> Ending index */
  quickSort(arr[], low, high)
 {
    if (low < high) {/* pi is partitioning index, arr[p] is
now at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
```

Algorithm of Merge sort:

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

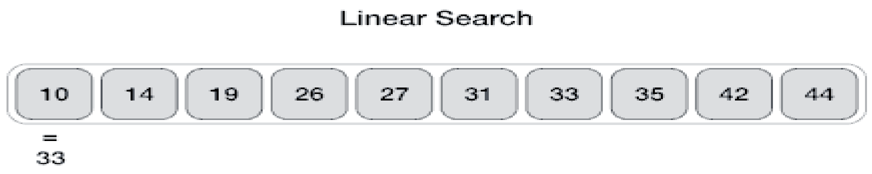
Post Lab question: Among the three sorting method which sorting algorithm in its typical implementation gives best performance when applied on an array which is sorted or almost sorted (maximum 1 or two elements are misplaced)?

Exp No: 14.1

Aim: Implementation of linear search method

Theory:

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



Algorithm

```
Linear Search ( Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7 Step
3: if A[i] = x then go to step 6 Step 4:
Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

Pseudocode

```
procedure linear_search (list, value)

   for each item in the list
         if match item == value return
            the item's location
        end if
  end for
end procedure
```

Exp No: 14.2

Aim: Implementation of binary search method

Theory:

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data elements should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the sub array reduces to zero.

Pseudocode

The pseudocode of binary search algorithms should look like this −

```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 1
  Set upperBound = n

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exists.
        set midPoint = lowerBound + ( upperBound - lowerBound ) / 2
            if A[midPoint] < x
                set lowerBound = midPoint + 1
           if A[midPoint] > x
                set upperBound = midPoint - 1
           if A[midPoint] = x
           EXIT: x found at location midpoint

end while

end procedure
```

Post lab question:

1) What is worst case, best case and average case time complexity of linear search and Binary search?