Catalog

UNIT-III
NON LINERAR DATA STRUCTURES

Arrays and its representations – Stacks and Queues – Linked lists – Linked list-based implementation of Stacks and Queues – Evaluation of Expressions – Linked list based polynomial addition.

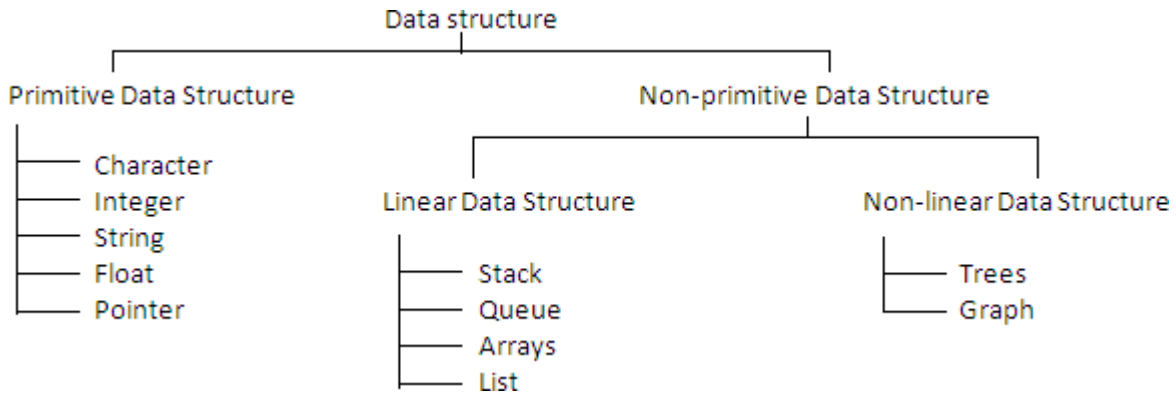
Definition:

- Data structure is a particular way of organizing, storing and retrieving data, so that it can be used efficiently. It is thestructural representation of logical relationships between elements ofdata.

Where data structures are used?

- Data structures are used in almost every program or software system. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks.
- **Applications** in which data structures are applied extensively
  - Compiler design (Hash tables),
  - Operating system,
  - Database management system (B+Trees),
  - Statistical analysis package,
  - Numerical analysis (Graphs),
  - Graphics,
  - Artificial intelligence,
  - Simulation

Classification of data structure



- **Primitive Data Structure -** Primitive data structures are predefined types of data, which are supported by the programming language. These are the basic data structures and are directly operated upon by the machine instructions, which is in a primitive level.

- **Non-Primitive Data Structure -** Non-primitive data structures are not defined by the programming language, but are instead created by the programmer. It is a more sophisticateddata structure emphasizing on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items**.**
- **Linear data structure**- only two elements are adjacent to each other. (Each node/elemen

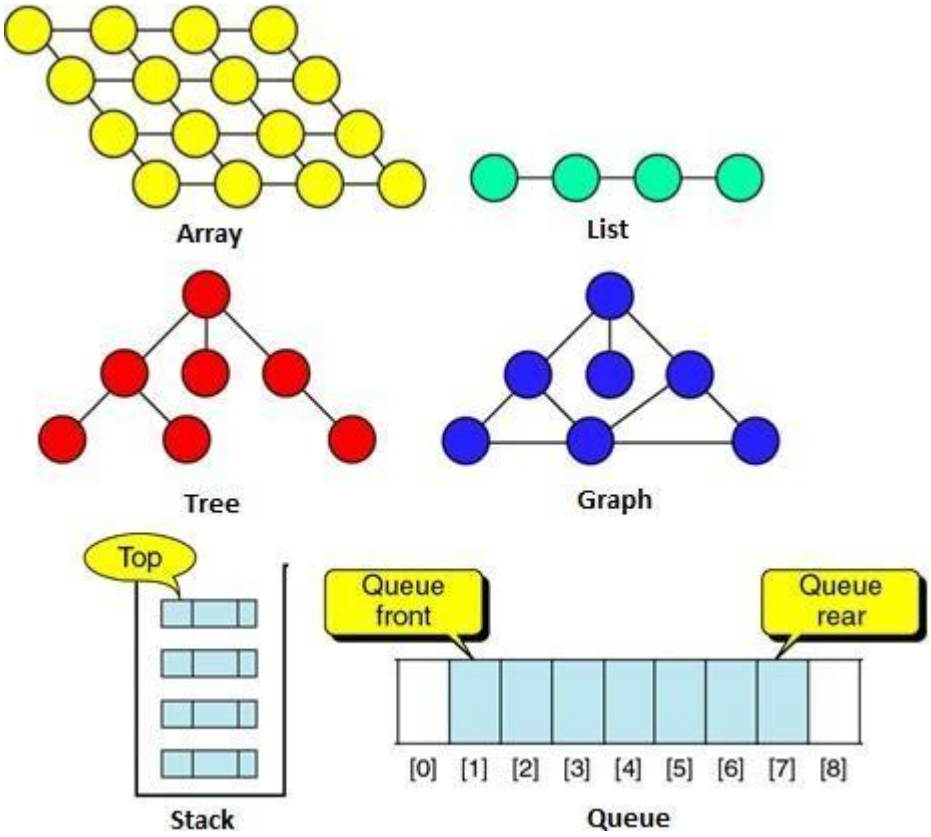
single successor)

- Restricted list (Addition and deletion of data are restricted to the ends of the list)
  - ✓ Stack (addition and deletion at **top** end)
  - ✓ Queue (addition at **rear** end and deletion from **front** end)
- General list (Data can be inserted or deleted anywhere in the list: at the beginning, in themiddle or at the end)

Non-linear data structure- One element can be connected to more than two adjacentelements.(Each node/elementcan have more than one successor)

- Tree (Each node could have multiple successors but just one predecessor)
- Graph (Each node may have multiple successors as well as multiple predecessors)

Note - Array and Linked list are the two basic structures for implementing all other ADTs.



MODULARITY

- **Module-** A module is a self-contained component of a larger software system.Each moduleis a logical unit and doesa specific job. Its size kept small by calling other modules.
- **Modularity** is the degree to which a system's components may be separated and recombined.Modularity refers tobreaking down software into different parts called modules.
- **Advantages** of modularity
  - It is easier to debug small routines than large routines.
  - Modules are easy to modify and to maintain.
  - Modules can be tested independently.
  - Modularity provides reusability.
  - It is easier for several people to work on a modular program simultaneously.

ABSTRACT DATA TYPE

What is Abstract Data Type (ADT)?

- ADT is a mathematical specification of the data, a list of operations that can be carried out on that data. It **includes** the specification of what it does, but **excludes** the specification of how it does. Operations on **set ADT**: Union, Intersection, Size and Complement.
- The **primary objective** is to separate the implementation of the abstract data types from their function. The program must know what the operations do, but it is actually better off not knowing how it is done. Three most common used abstract data types are Lists, Stacks, and Queues.
- ADT is an **extension of modular design**. The **basic idea** is that the implementation of these operations is **written once in the program**, and any other part of the program that needs to

perform an operation on the ADT can do soby calling the appropriate function. If for some reason implementation details need to change, it should be easy to do so by merely changing the routines that perform the ADT operations. This change, in a perfect world, would be completely transparent to the rest of the program.

- Examples of ADT: Stack, Queue, List, Trees, Heap, Graph, etc.

- **Benefits of using ADTs or Why ADTs**

- Code is easier to understand. Provides modularity and reusability.
- Implementations of ADTs can be changed without requiring changes to the program that usesthe ADTs.

LIST ADT

- List is a linear collection of ordered elements. The general form of the list of size N is: **A0, A1, …, AN-1**
  - Where A_1 - First element
    A_N - Last Element
    N - Size of the list
  - If the element at position 'i' is A_i then its successor is A_{i+1} and its predecessor is A_{i-1}.
- Various operations performed on a List ADT
  - Insert (X,5) - Insert the element X after the position 5.
  - Delete (X) - The element X is deleted.
  - Find (X) - Returns the position of X
  - Next (i) - Returns the position of its successor element i+1.
  - Previous (i) - Returns the position of its Predecessor element i-1.
  - PrintList - Displays the List contents.
  - MakeEmpty - Makes the List empty.

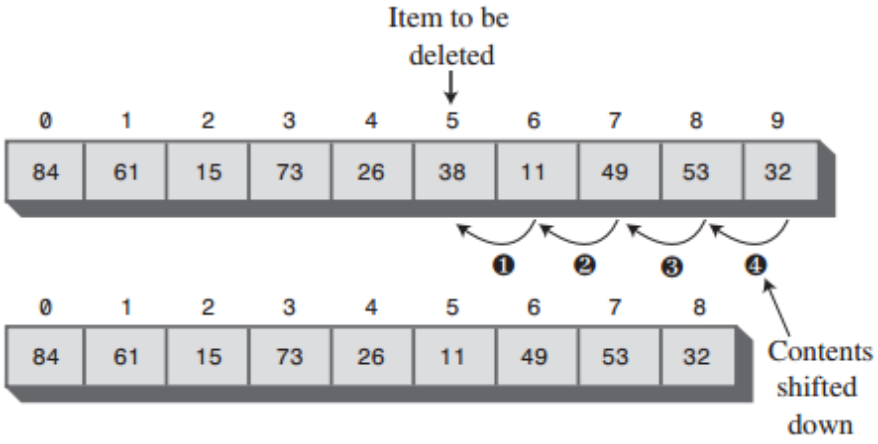
- **Implementation of List ADT**
  - Array implementation
  - Linked List implementation
  - Cursor implementation

ARRAY IMPLEMENTATION OF LIST ADT

- An array is a collection of homogeneous data elements described by a single name. Each element of an array is referenced by a subscripted variable or value, called subscript or indexenclosed in parenthesis. In array implementation, elements of list are stored in contiguous cellsof an array. Find K^{th} operation takes constant time. **Print List**, **Find** operations take linear time.
- Advantages - **Searching** an array for an **individual** element can be **very efficient -** Fast, randomaccess of elements.
- Limitations - Array implementation has some limitations such as
  1. Maximum size must be known in advance, even if it is dynamically allocated.
  2. The size of array can't be changed after its declaration (static data structure). i.e., the size isfixed.
  3. Data are stored in continuous memory blocks.
  4. The running time for Insertion and deletion of elements is so slow. Inserting and deletion requires shifting other data in the array. For example, inserting at position 0 requires first pushing the entire array down one spot to make room, whereas deleting the first element requires shifting all the elements in the list up one, so the worst case of these operations isO(n). On average, half the list needs to be moved

for either operation, so linear time is stillrequired.

5. Memory is wasted, as the memory remains allocated to the array throughout the program



execution even few nodes are stored.

Deleting an item

Type Declarations#define Max 10

```
int A[Max],N;
```

Routine to insert an Element in the specified position

```
void insert(int x, int p, int A[], int N)
{
int i; If(p==N)
printf("Array Overflow");
else
{
for(i=N-1;i>=p-1;i--)A[i+1]=A[i];
A[p-1]=x;N=N+1;
}
}
```

Routine to delete an Element in the specified

```
int deletion(int p, int A[],int N)
{
int Temp;
If(p==N)
Temp=A[p-
1];else
{
Temp=A[p-1];        For(i=p-1;i<=N-
1;i++)A[i]=A[i+1];
}
N=N-1;
return Temp;
}
```

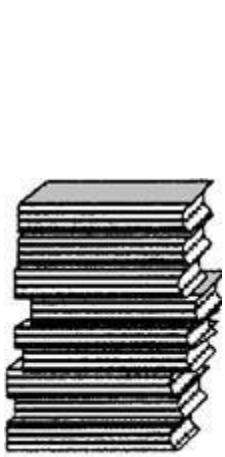
Find Routine

```
void Find (int X)
{
int i,f=0; for(i=0;i<N;i++)if(a[i]==x)
{ f=1;
break;
}
if (f==1)
printf(“Element
found”);else
printf(“Element not found”);
}
```

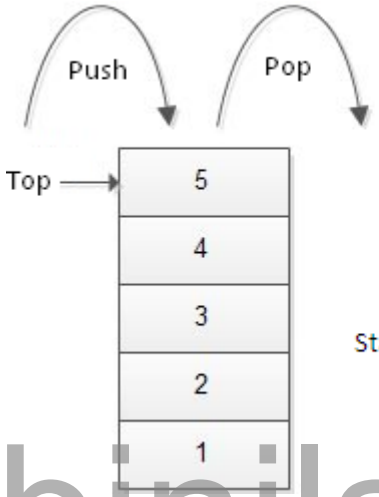
STACK

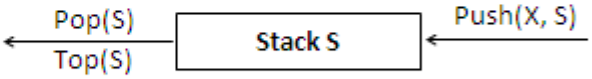
Definition

Stack is a linear list in which elements are added and removed from **only one end**, called the **top**. It is a "last in, first out" (**LIFO**) data structure. At the logical level, a stack is an **ordered**group of **homogeneous** items or elements. Because items are added and removed only from the top of the stack, the **last element** to be added is the first to be removed. Stacks are also referred as "piles" and "push-down lists".



Stack of Books

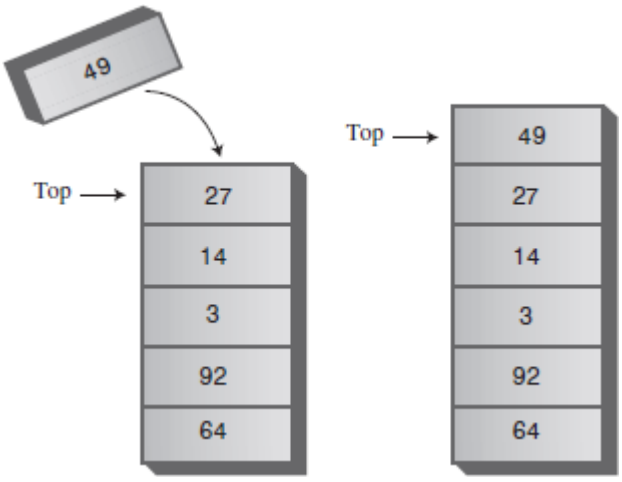




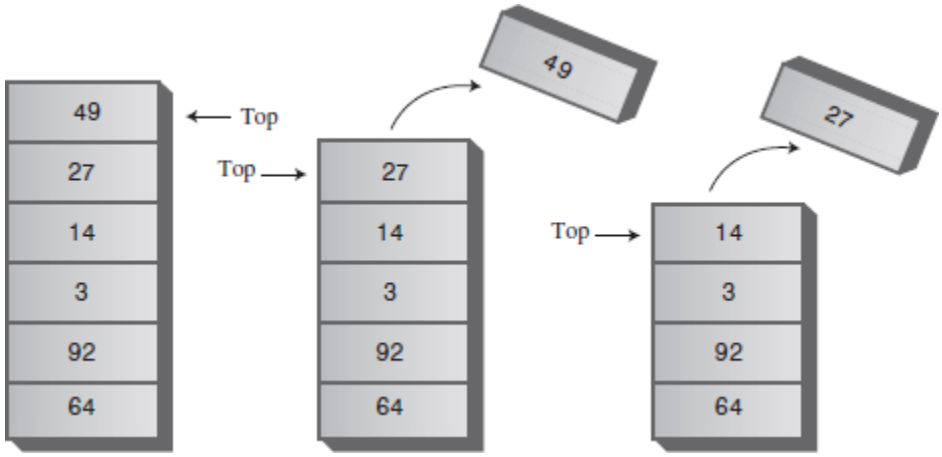
Stack model : Input bto a Stack is by push, output is by pop

- **Operations on stacks**
  - **Push** - Inserts new item to the top of the stack. After the push, the new item becomes thetop.
  - **Pop** - Deletes top item from the stack. The next older item in the stack becomes the top.
  - **Top** - Returns a copy of the top item on the stack, but does not delete it.
  - **MakeEmpty** - Sets stack to an empty state.
  - Boolean **IsEmpty** - Determines whether the stack is empty. IsEmpty should compare topwith **-1**.
  - Boolean **IsFull** - Determines whether the stack is full. IsFull should compare top with **MAX_ITEMS - 1**.

- **Conditions**
  - **Stack overflow -** The condition resulting from trying to push an element onto a full stack.
  - **Stack underflow -** The condition resulting from trying to pop an element from an empty system.



New item pushed on Stack



Two items popped from Stack

APPLICATIONS OF STACKS

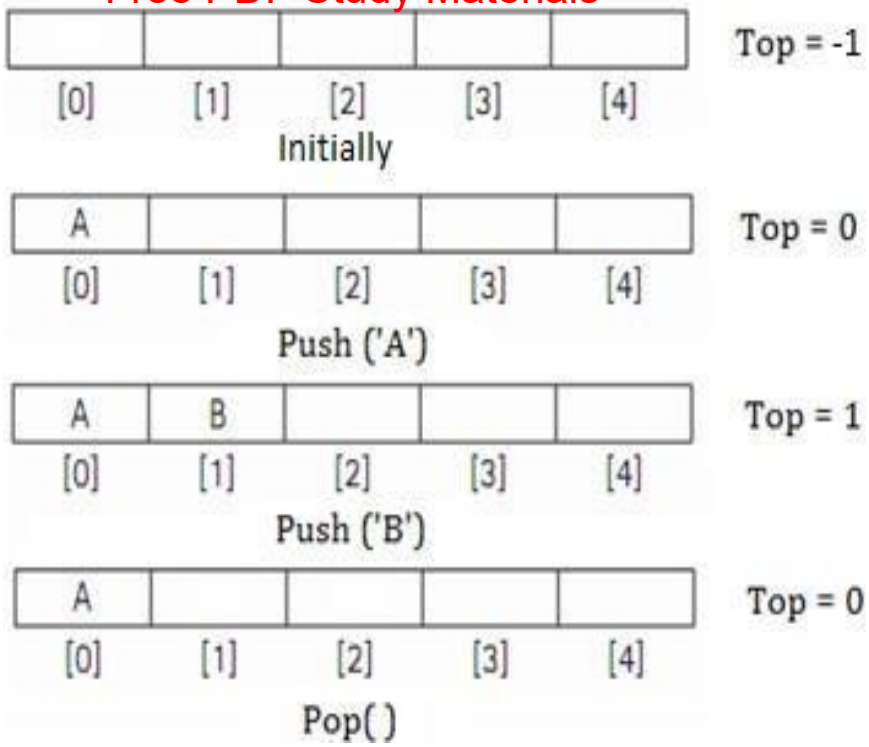
- **Recursion** - Example, Factorial, Tower of Hanoi.
- **Balancing Symbols**, i.e., finding the unmatched/missing parenthesis. For example, ((A+B)/C and (A+B)/C). Compilersoften use stacks **to perform syntax analysis oflanguage statements**.
- C**onversion** of infix expression to postfix expression and decimal number to binary
- number.
- **Evaluation** of postfix expression.
- **Backtracking**- For example, 8-Queens problem.

Function calls - When a call is made to a new function, all the variables local to the calling routine need to be saved by the system, since otherwise the new function will overwrite the calling routine's variables. Similarly the current location in the routine must be saved so that the new function knows where to go after it is done. For example, the main program calls operation A, which in turn calls operation B, which in turn calls operation C. When C finishes,control returns to B; when B finishes, control returns to A; and so on. The call-and-return sequence is essentially a LIFO sequence, so a stack is the perfect structure for tracking it.

- **Implementations of stack**
  1. Array implementation of stack
  2. Linked list implementation of stack

- **Array implementation of stack**

Stack can be represented using one dimensional array and it is probably the **more popular** solution.Here the stack is of fixed size. That is maximum limit for storing elements is specified. Once the maximum limit is reached, it is not possible to store the elements into it. So array implementation is not flexible and not an efficient method when resource optimization is concerned.



PUSH AND POP OPERATION

```
#include<stdio.h>
#include<conio.h
>#define MAX 5

void push();
void pop();
void
display();
int stack[MAX], top=-1,
item;void push()
{
        if(top == MAX-1)
                printf("Stack is full");
        els
        e
        {       printf("Enter item:
                ");
                scanf("%d",&item)
                ;top++;
                stack[top] = item;
                printf("Item pushed =
        }       %d",
}


void pop()
{
        if(top == -1)
        printf("Stack
        isempty");
        else
        {
                item =
                stack[top];top--;
                printf("Item popped = %d", item);
        }
}

void display()
{
        int i;
```

```
        if(top == -1)
                printf("Stack is empty");
        else
        {
                for(i=top; i>=0; i--)
                        printf("\n %d", stack[i]);
        }
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation.Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.

If the linked list is used to implement the stack, then in step 3, we need to allocatespace dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

Implementation of this algorithm in C, is very easy. See the following code −

Example

```
begin procedure push: stack, data

   if stack is fullreturn
null

      top ← top + 1
stack[top] ← data
```

```
void push(int data) {
  if(!isFull()) {
    top = top + 1;
    stack[top] = data;
  } else {
    printf("Could not insert data, Stack is full.\n");
  }
```

```
}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** ispointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.

Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

Implementation of this algorithm in C, is as follows −

Example

```
int pop(int data)
                    {
```

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]
   top ← top - 1
   return data
```

```
    data =
    stack[top];top =
    top - 1; return
    data;
  } else {
    printf("Could not retrieve data, Stack is empty.\n");
  }
```

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. We shall learn the same here in this chapter.

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.

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**.

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 / +
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 -

Parsing Expressions

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example −

As multiplication operation has precedence over addition, b * c will be evaluatedfirst. A table of operator precedence is provided later.

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determinedby associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as **(a + b) − c**.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) −

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (∗) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example −

In **a + b*c**, the expression part **b*c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a + b** to be evaluated first, like **(a + b)*c**.

Postfix Evaluation Algorithm

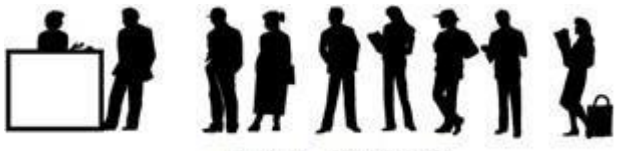
We shall now look at the algorithm on how to evaluate postfix

notation −Step 1 − scan the expression from left to right
Step 2 − if it is an operand push it to stack
Step 3 − if it is an operator pull operand from stack and perform operationStep 4 − store the output of step 3, back to stack
Step 5 − scan the expression until all operands are consumedStep 6 − pop the stack and perform operation

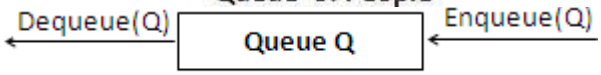
QUEUE

QUEUE

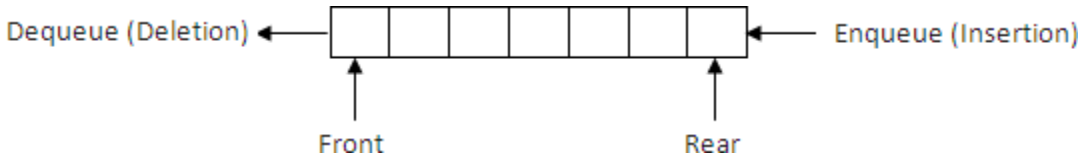
A queue is an **ordered** group of **homogeneous** items or elements, in which new elements are added at one end (the “rear”) and elements are removed from the other end (the “front”). It is a "First in, first out" (**FIFO**) linear data structure. Example, a line of students waiting to pay for their textbooks at a university bookstore.



Queue of People



Model of Queue



- **Types of Queues**

There are three major variations in a simple queue. They are

- Linear queue
- Circular queue
- **D**ouble **e**nded **que**ue (Deque)
  - Input restricted deque
  - Output restricted deque
- Priority queue

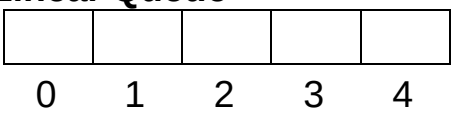
- **Operations on Queue**
  - **Enqueue -**Insertsan itemat the rear end of the queue.
  - **Dequeue-** Deletesan item at the front end of the queue and returns.

- **Conditions**
  - **Queue overflow -** The condition resulting from trying to enqueue an element onto a full Queue.
  - **Queue underflow -** The condition resulting from trying to dequeue an element from an empty Queue.

- **Implementation of Queue**

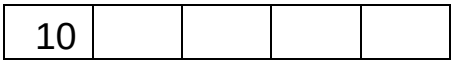
1. Array implementation
2. Linked list implementation
   - Array and linked list implementations give fast **O(1)** running times for every operation

- **Array implementation of Linear Queue**



0 1 2 3 4

Empty Queue F = R = -1

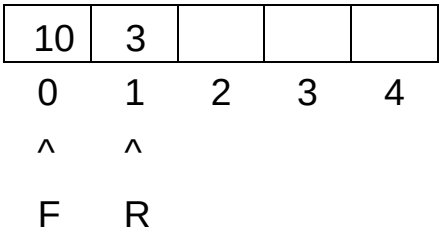


0 1 2 3 4

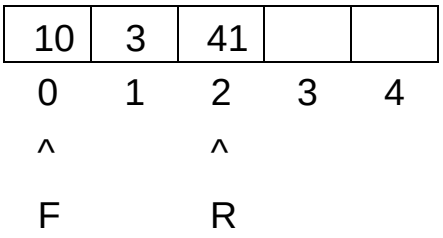
^^

FR

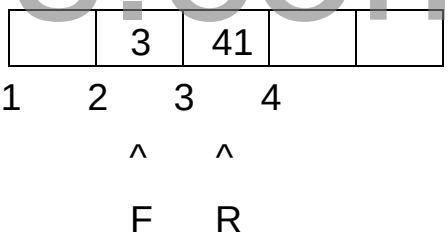
After Enqueue (10)



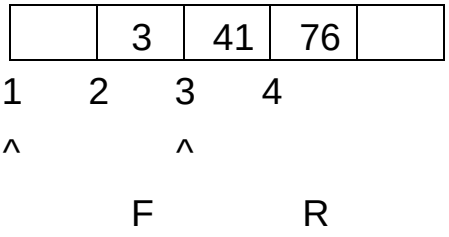
After Enqueue (3)



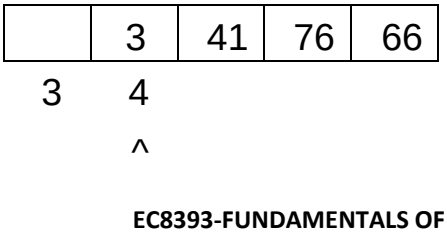
After Enqueue (41)

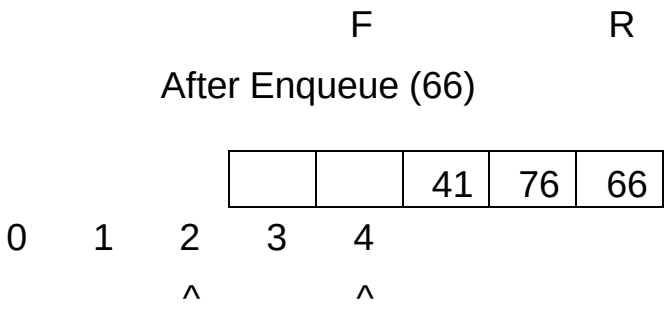


After Dequeue ()



After Enqueue (76)





F R

After Dequeue ()

There is one potential **problem** with array implementation. From the above queue, now if we attempt to add moreelements, even though 2 queue cells are free, the elements cannot be inserted. Because in a queue, elements are always inserted at the rear end and hence rear points to last location of the queue array Q[4]. That is queue is full(overflow condition) though it is empty.

The simple solution is that whenever front or rear gets to the end of the array, it is wrapped around to thebeginning. This is known as a **circular array** implementation.

Array implementation of Linear Queue

Array implementation of Linear Queue

```
#include
<stdio.h>
#include<co
nio.h>

#define MAX 3

void enqueue(); void
 dequeue(); void
 display();
int queue[MAX], rear=-1,
front=-1, item; void
enqueue()
{
if(rear == MAX-
1)
printf("Queue
is full"); else
{
printf("Enter
item : ");
scanf("%d",
&item);
if (rear == -1 && front == -1) rear
= front = 0; else
rear = rear + 1;
queue[rear] = item;
printf("Item enqueued :
%d", item);
}
}

void dequeue()
{
if(front == -1) printf("Queue is empty"); else
{
item = queue[front]; if
(front == rear) front = rear
= -1;
else
front = front + 1;
printf("Item dequeued : %d", item);
}
}

void display()
{
int i;
if(front == -1)
printf("Queue is
empty"); else
for(i=front;
i<=rear; i++)
printf("%d ",
queue[i]);
```

}

Circular Queue

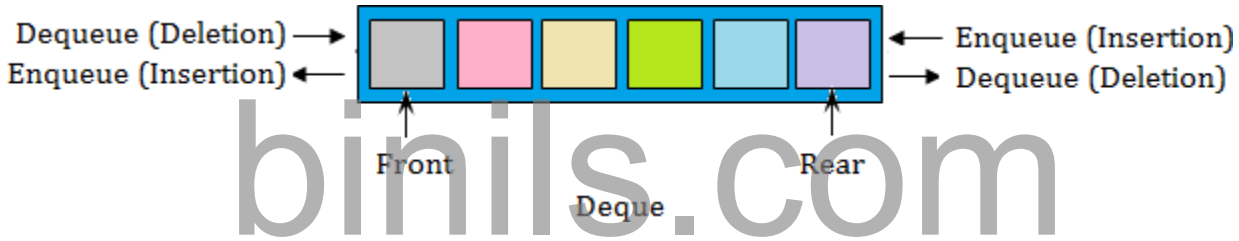
- In circular queues the elements Q[0],Q[1],Q[2] Q[n – 1] is represented in a circularfashion with Q[1]
  following
  Q[n]. A circularqueue is one in which the insertion of a newelementis done at the very first location of thequeue ifthe last location at the queue isfull.
- Initially **Front = Rear = -1**. That is, front and rear are at the same position.
- At any time the **position** of the element to be **inserted** will be calculated by therelation: **Rear = (Rear + 1) % SIZE**

- After deleting an element from circular queue the position of the front end is calculatedby the relation:
  Front=(Front + 1) % SIZE.
- After locating the position of the new element to be inserted, rear, compare it withfront. If R**ear = Front**, the queueis full and **cannot be inserted anymore**.
- No of elements in a queue = **(Rear – Front + 1) % N**

Deque - Double Ended QUEeue

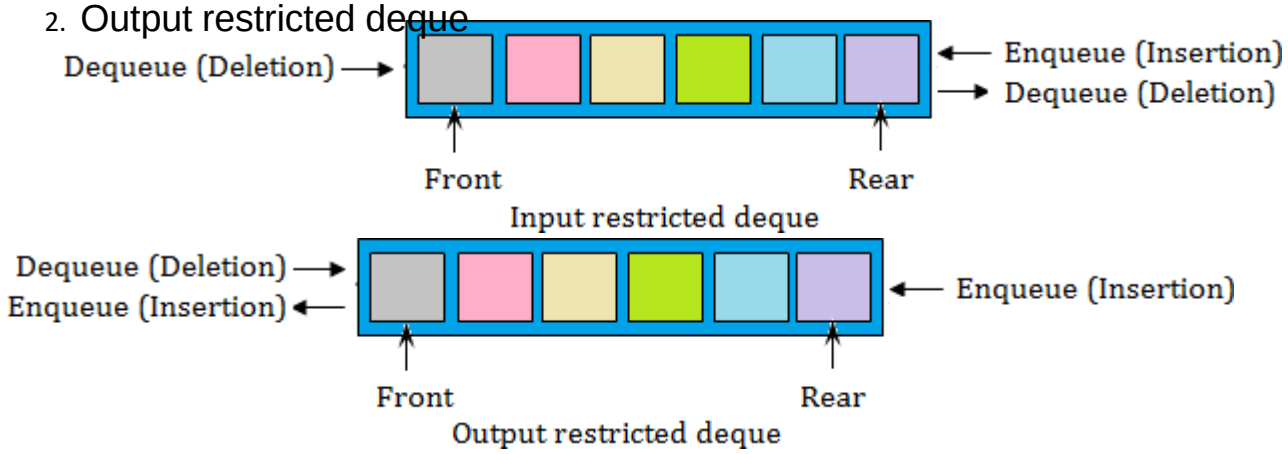
➢ **Definition**

A deque is a homogeneous list in which inserted and deleted operations are performed at either ends of the queue. That is, we can add a new element at the rear or front end and also we can remove an element from both front and rear end. Hence it is called double ended queue. The most common ways of **representing** deque are: **doubly linked list**, **circular list.**



➢ **Types of deques**

1. Input restricted deque
2. Output restricted deque



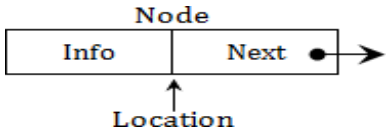
✓ An **input restricted deque** is a deque, which allows insertion at only 1 end, rear end, but allows deletion at bothends, rear and front end of the lists.

- ✓ An **output-restricted deque** is a deque, which allows deletion at only one end, front end, but allows insertion at both ends, rear and front ends, of the lists.

UNIT III LINEAR DATA STRUCTURES

LINKED LIST

Definition



Linked list is a **dynamic** data structure which is an ordered collection of homogeneous data elements called nodes, in which each element contains two parts: **data** or **Info** and one or more **links**. The data holds the application data to be processed.

UNIT-III EC8393-FUNDAMENTALS OF DATASTRUCTURES IN C

Why Linked List?

- Even though **searching** an array for an **individual** element can be **very efficient**, array has some limitations. So arrays are generally not used to implement Lists**.**

Advantages of Linked List

1. Linked list are dynamic data structures - The size is not fixed. They can grow or shrink during the execution of a program.
2. Efficient memory utilization - memory is not pre-allocated. Memory is allocated, whenever it is required and it is de-allocated whenever it is not needed. Data are stored in non-continuous memory blocks.
3. Insertion and deletion of elements are easier and efficient. Provides flexibility. No need to shift elements of a linked list to make room for a new element or to delete an element.

Disadvantages of Linked List

1. More memory - Needs space for pointer (link field).
2. Accessing arbitrary element is time consuming. Only sequential search is supported not binary search.

Operations on Linked List

The primitive operations performed on the linked list are as follows

1. **Creation**- This operation is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.
2. **Insertion**- This operation is used to insert a new node at any specified location in the linked list. A new node may be inserted,
   - ✓ At the beginning of the linked list,
   - ✓ At the end of the linked list,
   - ✓ At any specified position in between in a linked list.
3. **Deletion**- This operation is used to delete an item (or node) from the linked list. A node may be deleted from the,
   - ✓ Beginning of a linked list,
   - ✓ End of a linked list,
   - ✓ Specified location of the linked list.
4. **Traversing** - It is the process of going through all the nodes from one end to another end of a linked list. In a singly linked list we can visit the nodes only from left to right (forward traversing). But in doubly linked list forward and backward traversing is possible.
5. **Searching**- It is the process finding a specified node in a linked list.
6. **Concatenation**- It is the process of appending the second list to the end of the first list. Consider a list A having n nodes and B with m nodes. Then the operation concatenation will place the 1st node of B in the (n+1) the node in A. After concatenation A will contain (name) nodes.

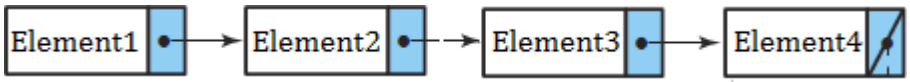
Types of linked list

1. Singlylinked list or Linear list or One-waylist
2. Doubly linked list or Two-way list
3. Circular linked list
4. Doubly circular linked list

SINGLY LINKED LIST

- **Definition**

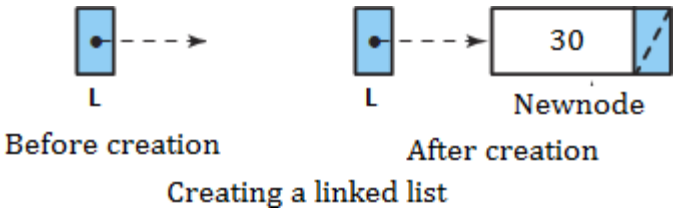
In singly linked list, each element (except the first one) has a unique predecessor, and each element (except the last one) has a unique successor. Each node contains two parts: **data** or **Info** and **link**. The data holds the application data to be processed. The link contains the address of the next node in the list. That is, each node has a single pointer to the next node. The last node contains a NULL pointer indicating the end of the list.



- **SentinelNode**
  - It is also called as **Header node**or **Dummy node.**
  - **Advantages**
    - Sentinel node is used to solve the following problems
      - ✓ First, there is **no really obvious way to insert at the front** of the list from the definitions given.
      - ✓ Second, **deleting from the front of the list** is a special case, because it changes the start of the list; careless coding will lose the list.
      - ✓ A third problem concerns deletion in general. Although the pointer moves above are simple, the deletion algorithm requires us to **keep track of the cell before the one that we want to delete.**
  - **Disadvantages**
    - It consumes extra space.

- **Insertion**

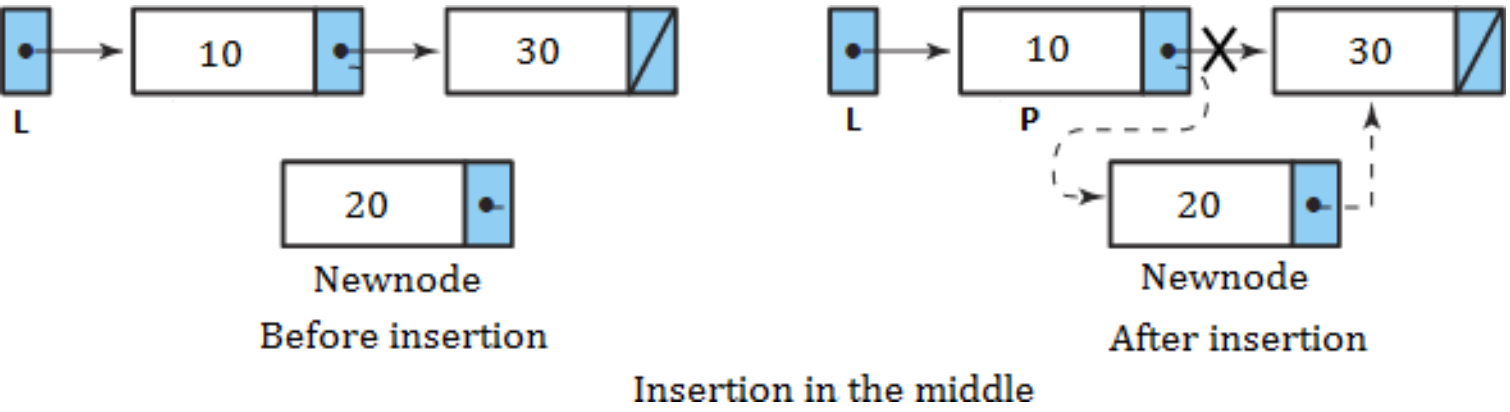
a. **Creating a newnode from empty List**



Creating a linked list

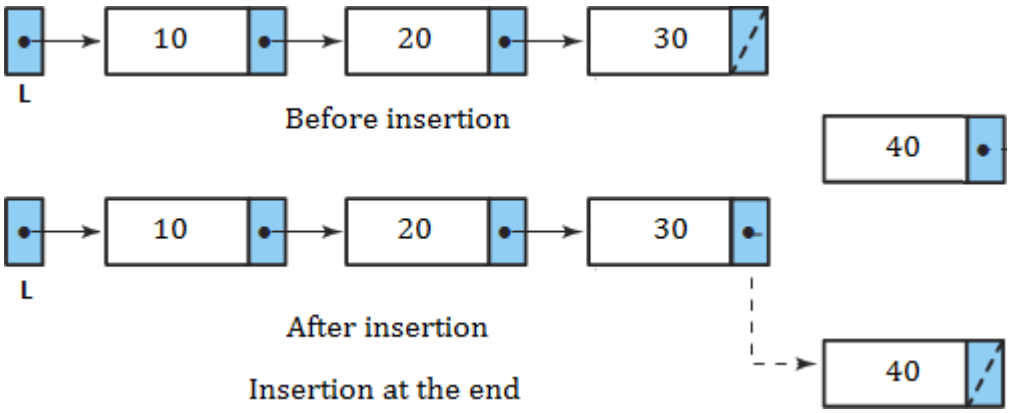
b. **Inserting a node to the front of list**

c. **Inserting a node in the middle**



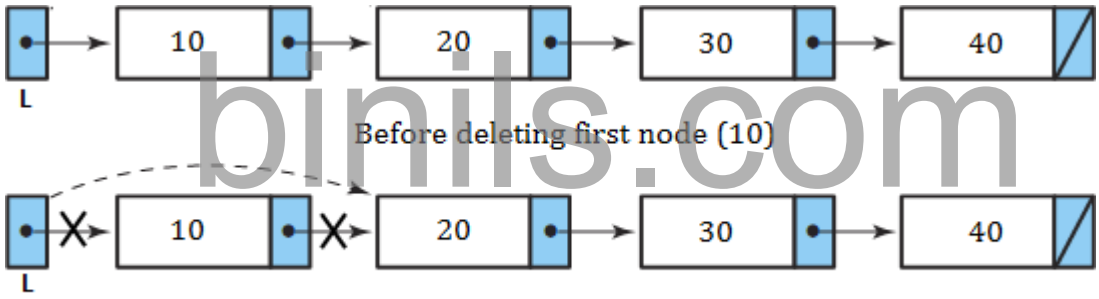
Insertion in the middle

d. Inserting a node to the end of list

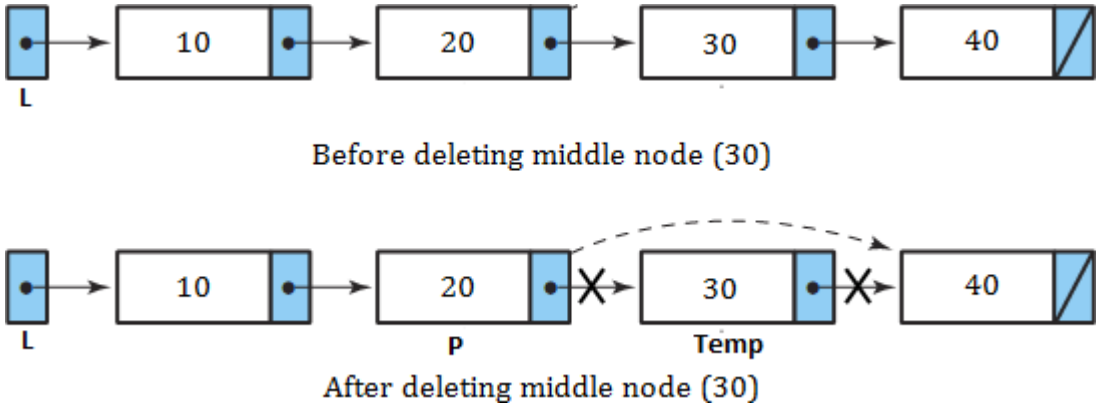


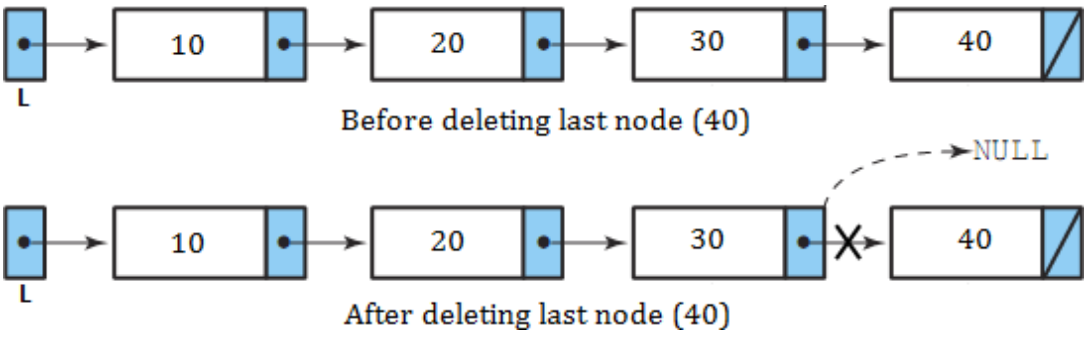
➢ **Deletion**

a. Inserting a node to the front of list



b. Deleting the middle node





c. **Deleting the last node**

Type Declarations

```
struct node
{
    int data;
    struct node *next;
}*head=NULL:
typedef struct Node *position;
```

Routine to check whether the List is empty

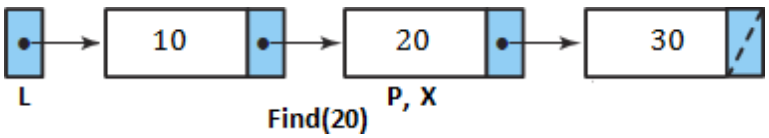
```
/* Returns 1 if List is empty */int IsEmpty
(position head)
{
        if (head->next == NULL)return(1);
}
```

Routine to check whether the current position is last

```
/* Returns 1 if P is the last position in L */int IsLast (position p)
{
        if (p->Next == NULL)return(1);
}
```


Find Routine

```
/* Returns the position of X in L; NULL if not found */Position Find (int
X)
{
        position p;
        P = head->next;
        while( (p!= NULL) && (p->data != X) )p
        = p->next;return P;
}
```

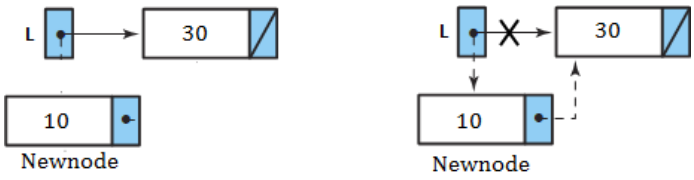


Singly linked list

FindPrevious Routine
/* Returns the previous position of X in L */

implementation

Inserting a node to the front of list



Before insertion

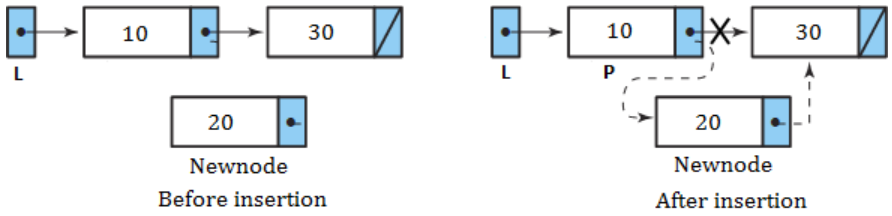
After insertion

Insertion at the beginning

Insert at Beginning

```
void Insert_beg (int X)
{
        position NewNode;
        NewNode = malloc (sizeof(struct Node))
```

```
    if(NewNode != NULL)
      {
            NewNode->data =
            X; NewNode->next
            = L->Next; head-
            >next = NewNode;
      }
}
```



Insertion in the middle

b.Inserting a node in the middle

Insertion at Middle

```
/* Insert element X after
position P */void
Insert_mid (int X,  position
P)
{
      position NewNode;
      NewNode = malloc
      (sizeof(struct Node));
      if(NeWNode != NULL)
      {
            NewNode->data =
            X; NewNode->next
            = P->next; P->next
            = NewNode;
      }
}
```

c.Inserting a node to the end of listInsert at Last

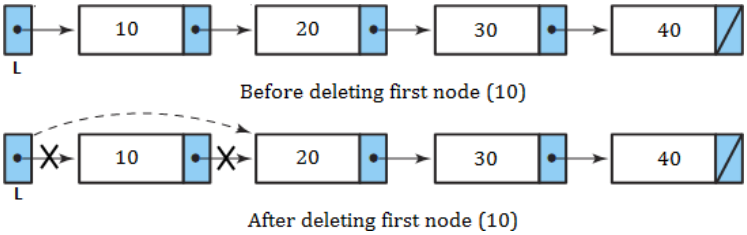
```
void Insert_last (int X)
{
      position NewNode,P;
      NewNode = malloc
      (sizeof(struct Node));
      if(NewNode != NULL)
      {
            while(P-
            >next!=NULL)
            P = P->next;
            NewNode-
            >data = X;
```

```
        NewNode->next
        = NULL; P->next
        = NewNode;
}
```

UNIT-IIIEC8393-FUNDAMENTALS OF DATASTRUCTURES IN C

}

- **Deletion**

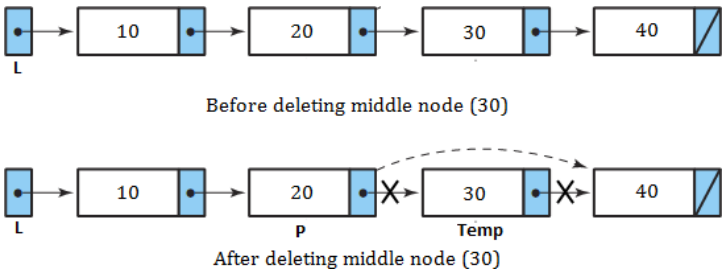


Deleting a node to the front of list

Delete at Beginning

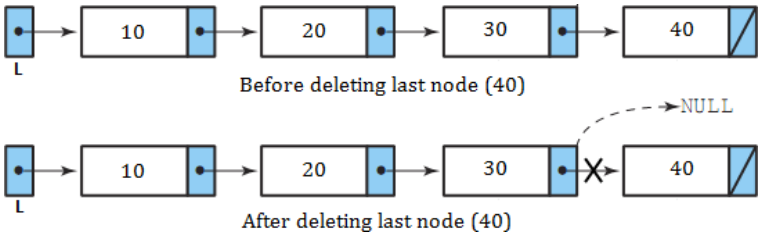
```
void Delete_beg ()
{
        position
        TempCell;
        if(head-
        >next!=NULL)
        {
                TempCell = head->next;
                head->next =
                TempCell ->next;
                free(TempCell);
        }
```

Deleting the middle node



Delete at Middle Routine

```
void Delete (int X)
{
        position P,
        TempCell; P =
        FindPrevious(
        X); TempCell =
        P->next;
        P->Next =
        TempCell ->Next;
        free(TempCell);
}
```



Deleting the last node

Delete at Last

```
void Delete_last ()
{
        position
---------------------------------------------------------------
        TempCell,P;
        while(P->next-
        >next!=NULL
```

```
    P = P->next;
TempCell = P->next; P-
>next = NULL;
free(TempCell);
      }
```

CIRCULAR LINKED LIST

➢ **Why circular linked list? Or advantages over singly linked list**

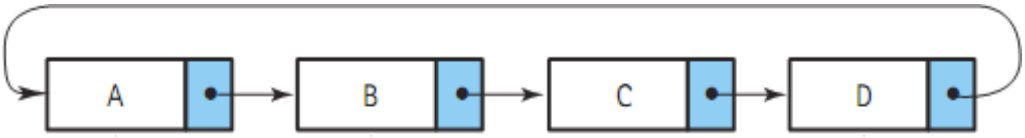
- With a singly linked list structure, given a pointer to a node anywhere in the list, we can access all the nodes that follow but none of the nodes that precede it. We must always have a pointer to the beginning of the list to be able to access all the nodes in the list.In a circular linked list, every node is accessible from a given node.
- In deletion of singly linked list, to find the predecessor requires that a search be carried out by chaining through the nodes from the first node of the list. But this requirement does not exist for a circular list, since the search for the predecessor of node X can be initiated from X itself.
- Concatenation and splitting becomes more efficient.

➢ **Disadvantages**

- The circular linked list requires extra care to detect the end of the list. It may be possible to get into an infinite loop. So it needs a header node to indicate the start or end of the list.

➢ **Definition**

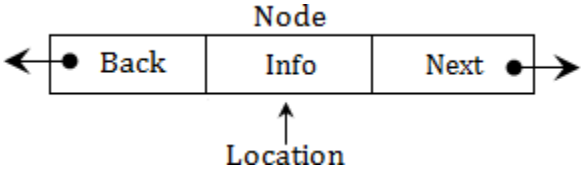
- A circular linked list is one, which has no beginning and no end. Circular linked list is a list in which every node has a successor; the "last" element is succeeded by the "first" element. We can start at any node in the list and traverse the entire list.



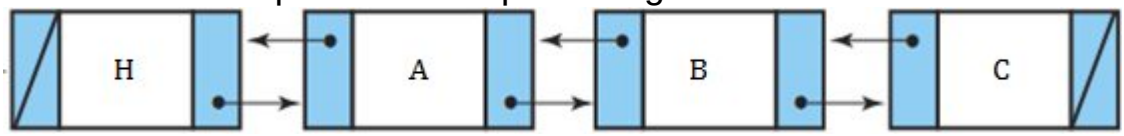
DOUBLY LINKED LIST

➢ **Definition**

- Doubly linked list is a linked list in which each node is linked to both its successor and its predecessor. In a doubly linked list, the nodes are linked in both directions. Each **node** of a doubly linked list contains three parts:



- **Info**: the data stored in the node
- **Next/FLink**: the pointer to the following node.
- **Back/BLink**: the pointer to the preceding node



- **Why doubly linked list?**
- In singly linked list, it is difficult to perform traversing the list in reverse.
- To delete a node, we need find its predecessor of that node.
- **Advantages**
- Traversing in reverse is possible.
- Deletion operation is easier, since it has pointers to its predecessor and successor.

Finding the predecessor and successor of a node is easier.

- **Disadvantages**

- A doubly linked list needs **more operations** while inserting or deleting and it needs **more space** (to store the extra pointer). There are more pointers to keep track of in a doubly linked list. For example, to insert a new node after a given node, in a singly linked list, we need to change two pointers. The same operation on a doubly linked list requires four pointer changes.

DOUBLY CIRCULAR LINKED LIST

- **Why doubly circular linked list**?

The aim of considering doubly circular linked list is to simplify the insertion and deletion operations performed on doubly linked list.

- **Definition**

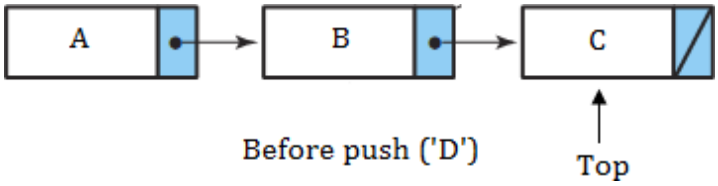
A circular linked list is one, which has no beginning and no end. A doubly circular linked list is a doubly linked list with circular structure in which the last node points to the first node and the first node points to the last node and there are two links between the nodes of the linked list. In doubly circular linked list, the left link of the leftmost node contains the address of the rightmost node and the right link of the rightmost node contains the address of the leftmost node.

All kinds of dynamic allocation related problems can be solved using linked lists. Some of the applications are given below:

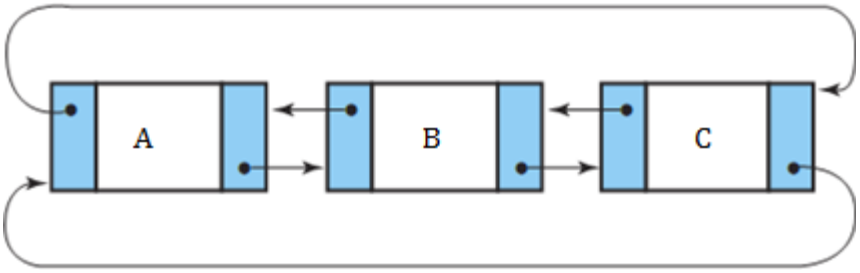
1. Polynomial ADT
2. Radix sort or Card sort
3. Multi-list
4. Stacks and Queues

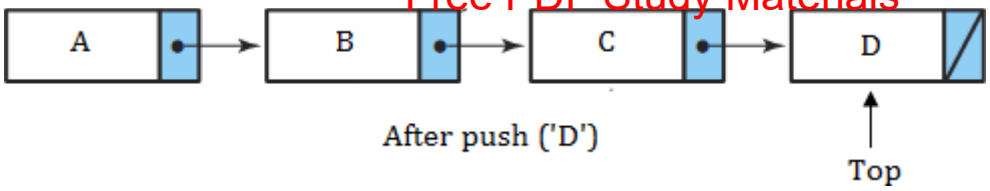
Linked list implementation of stack

The limitations of array implementation can be overcome by dynamically implementing (is also called linked list representation) the stack using pointers.In linked list implementation, the stack does not need to be of fixed size. Insertions and deletions are done more efficiently. Memory space also not wasted, because memory space is allocated only when it is necessary (when an element is pushed) and is de-allocated when the element is deleted.

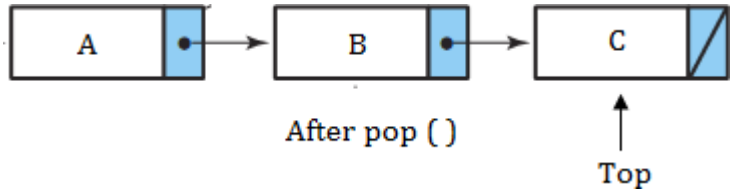


Before push ('D')





Push operation



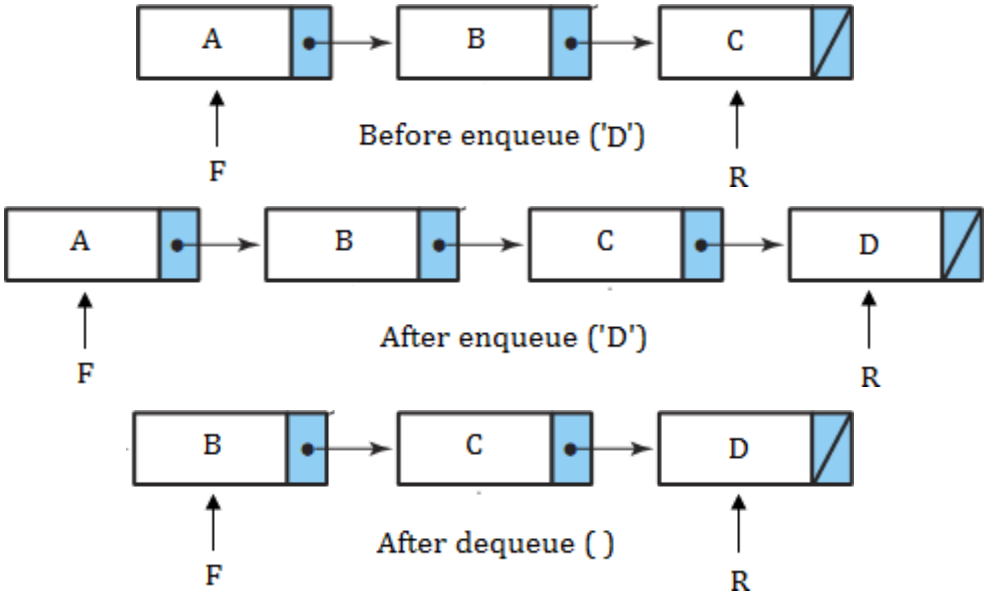
Pop operation

Linked list implementation of Stack

```
void
push(int x);
void pop();
void display();

struct node
{
        int data;
        struct node *next;
} *top = NULL;

typedef truct node *

position;void push(int

x)
{
   position p;
   p =(struct node
   *)malloc(sizeof(struct node));if (p
   == NULL)
     printf("Memory allocation
   error \n");else
   {
     if (top == NULL)
     {
        top =(struct node
        *)malloc(sizeof(struct node));p-
        >data=x;
        p->next =
        NULL;top-
```

```
        >next = p;
      }
     else
     {
        p->data=x;
        p->next = top-
        >next;top-
        >next=p;
      }

   }
}
```

Linked list implementation of Linear Queue



Linked list implementation of Linear Queue

```
struct node
{
        int data;
        struct node *next;
} *front = NULL,

*rear=NULL; typedef

truct node * position;

void
enqueue(int
x);void
dequeue();
void display();
int item;

struct node
{
        int data;
        struct node *next;
} *top = NULL;

typedef truct node * position;


void dequeue()
{

  position p;
  p = front->next;

  if (front == NULL)
   printf("Queue is
   empty\n");
 else
 {
       printf("\n Dequeued value :
       %d\n", p->data);front-
       >next=front->next->next;
       free(p);
 }

}
```

```
void display()
{
  position p;
  p = front->next;

  if (front == NULL)
   printf("Queue is
   empty\n");
 else
 {
      printf("Queue elements
      are : \n");while (p !=
      NULL)
      {
```

```
            printf("%d ",p->data); p = p->next;
        }
    }

}
```

APPLICATIONS OF STACKS

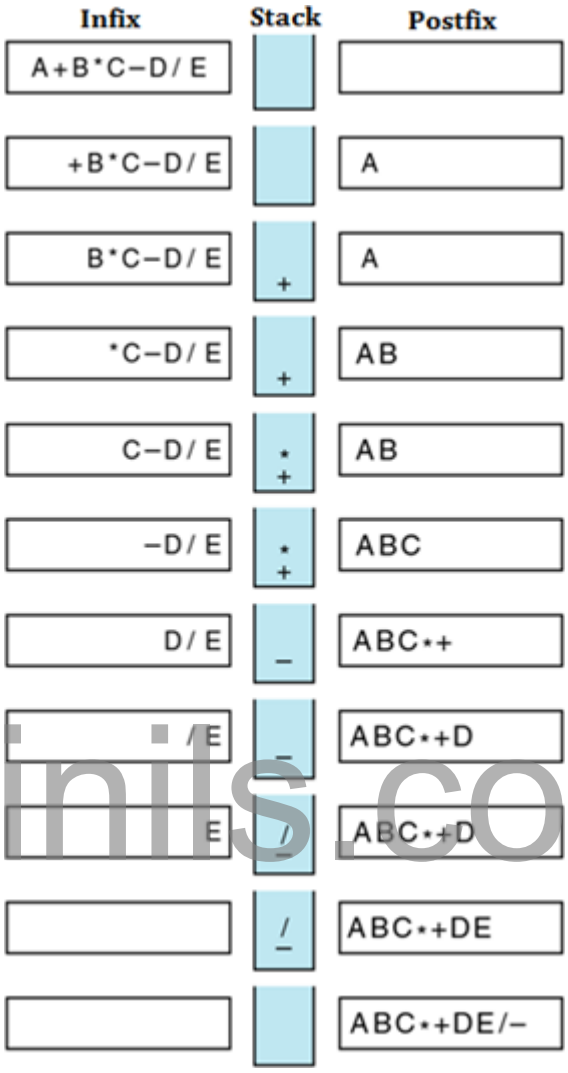
Recursion - Example, Factorial, Tower of Hanoi.

- **Balancing Symbols**, i.e., finding the unmatched/missing parenthesis. For example, ((A+B)/C and (A+B)/C). Compilers often use stacks **to perform syntax analysis of language statements**.
- C**onversion** of infix expression to postfix expression and decimal number to binary number.
- **Evaluation** of postfix expression.
- **Backtracking**- For example, 8-Queens problem.
- **Function calls -** When a call is made to a new function, all the variables local to the calling routine need to be saved by the system, since otherwise the new function will overwrite the calling routine's variables. Similarly the current location in the routine must be saved so that the new function knows where to go after it is done. For example, the main program calls operation A, which in turn calls operation B, which in turn calls operation C. When C finishes, control returns to B; when B finishes, control returns to A; and so on. The call-and-return sequence is essentially a LIFO sequence, so a stack is the perfect structure for tracking it.

Conversion of infix expression into postfix expression

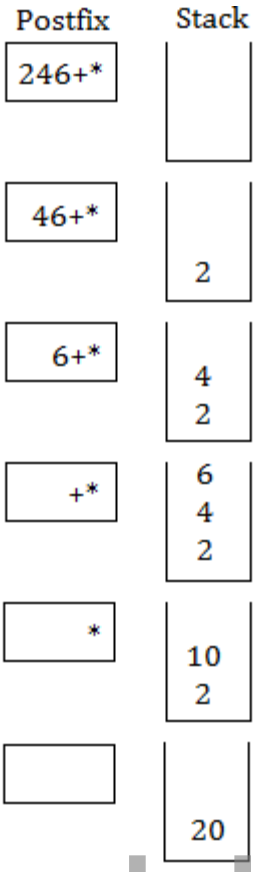
1. Scan the infix expression from left to right. Repeat Steps 3 to 6 for each element of expression until the stack is empty.
2. If an operand is encountered, add it to the postfix expression.
3. If an opening parenthesis is encountered, push it onto the stack and do not remove it until closing parenthesis is encountered.
4. If an operator 'op' is encountered, then
   a. Repeatedly pop from stack and add each operator (on the top of stack), which has the same precedence as, or higher precedence than 'op'.
   b. Add 'op' to stack.
5. If a closing parenthesis is encountered, then
   a. Repeatedly pop from stack and add to postfix expression (on the top of stack) until anopening parenthesis is encountered.
   b. Remove the opening parenthesis from the stack. [Do not add the opening parenthesis to postfix expression.]

Operator precedence	
(	High est
^	-
*, /	-
+, -	Least



- **Evaluation of postfix expression**

1. Scan the postfix expression from left to right and repeat steps 2 & 3 for each element of postfix expression.
2. If an operand is encountered, push it onto the stack.
3. If an operator 'op' is encountered,
   a. Pop two elements from the stack, where A is the top element and B is the next top element.
   b. Evaluate B 'op' A.
   c. Push the result onto stack.
4. The evaluated value is equal to the value at the top of the stack.

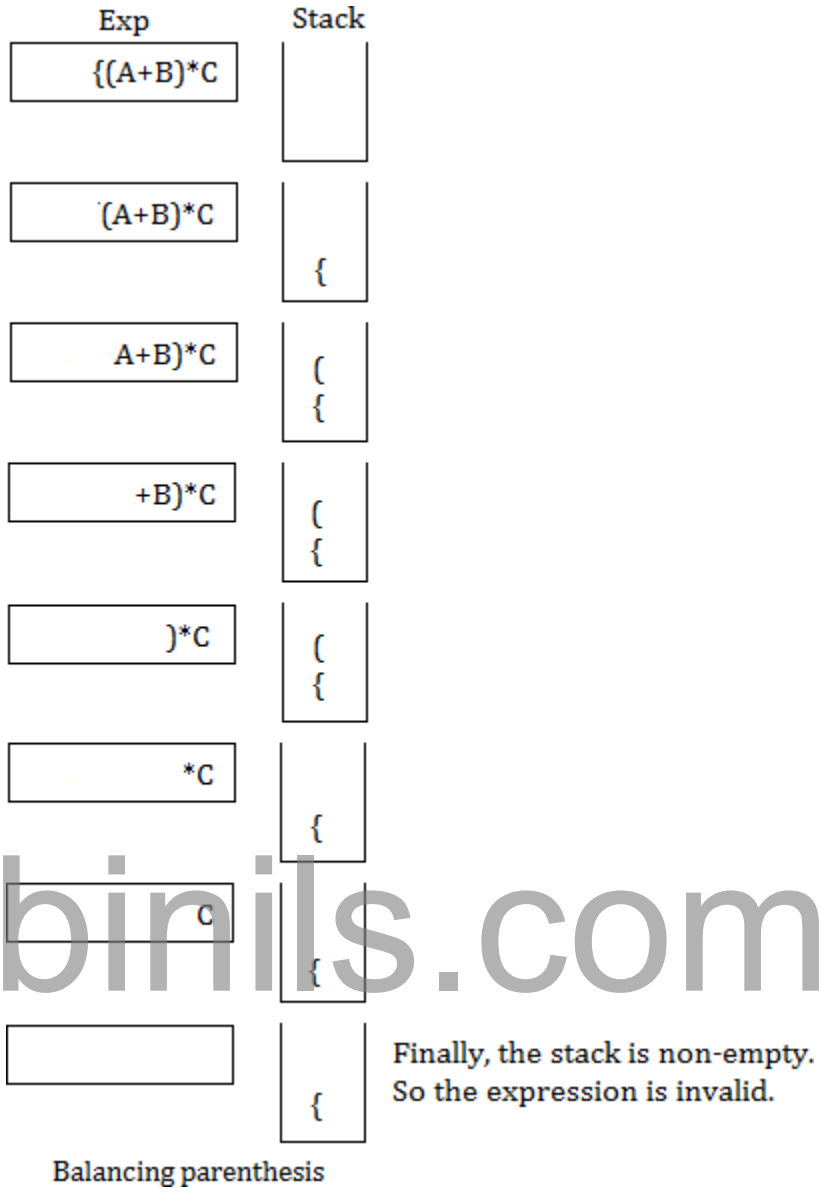


Evaluation of postfix expression

- **Balancing parenthesis**

- One common programming problem is unmatched parenthesis in an algebraic expression. When parentheses are unmatched, two types of errors can occur:
  - Opening parenthesis can be missing. For example, [A+B]/C}.
  - Closing parenthesis can be missing. For example, {(A+B)/C.
- The steps involved in checking the validity of an arithmetic expression

1. Scan the arithmetic expression from left to right.
2. If an opening parenthesis is encountered, push it onto the stack.
3. If a closing parenthesis is encountered, the stack is examined.
   a. If the stack is **empty**, the closing parenthesis does not have an opening parenthesis. So the expression is invalid.
   b. If the stack is **not empty**, pop from the stack and check whether the popped item corresponds to the closing parenthesis. If a match occurs, continue. Otherwise, the expression is invalid.
4. When the end of the expression is reached, the stack must be empty; otherwise one or more opening parenthesis does not have corresponding closing parenthesis. So the expression is invalid.



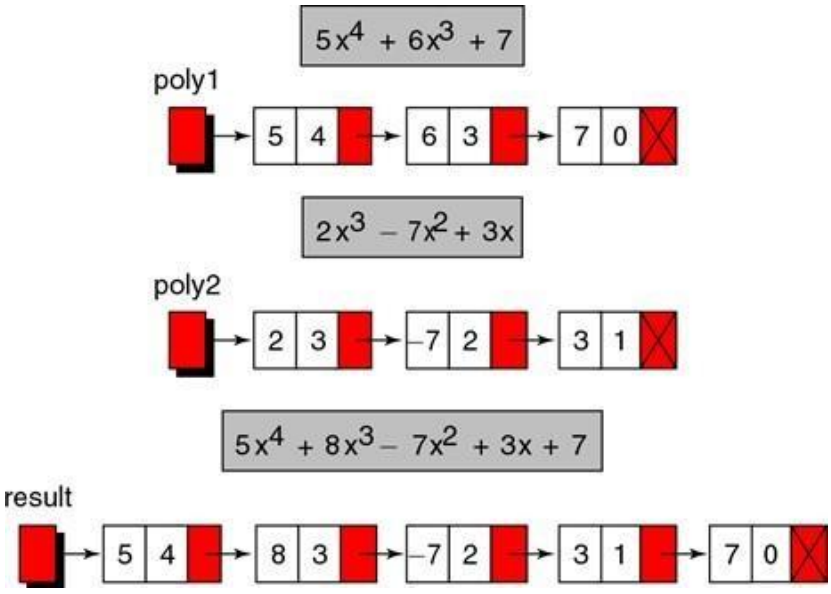
- **Polynomial ADT**

Polynomials are expressions containing terms with non-zero coefficients and exponents. Linked list is generally used to represent and manipulate single variable polynomials. Different operations, such as addition, subtraction, division and multiplication of polynomials can be performed using linked list. In this representation, each term/element is referred as a node. Each node contains three fields namely,

1. Coefficient - Holds value of the coefficient of a term
2. Exponent - Holds exponent value of a term
3. Link - Holds the address of the next term.

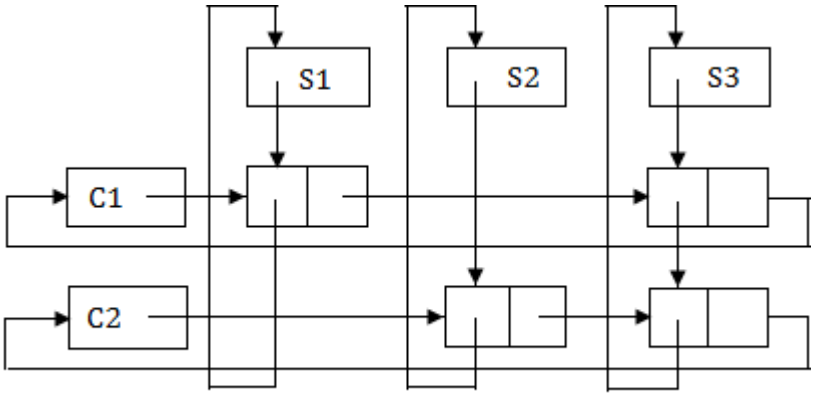
Coefficient	Exponent	Link

For example,



- **Multi-list**

Multi-list is the **most complicated applications** of linked list. It is useful to maintain student registration in a university, employee involvement in different projects etc. The student registration contains two reports. The first report lists the registration for each class (C) and the second report lists, by student, the classes that each student (S) is registered for. In this implementation, we have combined two lists into one. All lists use a header and are circular. Circular list **saves space** but does so at the **expense of time**.



Multi-list implementation for student registration problem

EVALUATION OF EXPRESSION IN C

Evaluation of Infix expressions

Infix notation is commonly used in arithmetic formula or statements; theoperators are written in-between their operands.

Let's assume the below

- Operands are real numbers.
- Permitted operators: +,-, *, /, ^(exponentiation)
- Blanks are permitted in expression.
- Parenthesis are permitted

Example:

Order of precedence of operations–

1. ^ (Exponential)
2. /
3. + –

Note: brackets ( ) are used to override these rules.

Let's define the ***Process***: (will be used for the main algorithm)

1. Pop-out two values from the operand stack, let's say it is A and B.
2. Pop-out operation from operator stack. let's say it is '+'.
3. Do A + B and push the result to the operand stack.

Algorithm:

Iterate through given expression, one character at a time

1. If the character is an operand, push it to the operand stack.

2. If the character is an operator,
    1. If the operator stack is empty then push it to the operator stack.
    2. Else If the operator stack is not empty,
        - If the character's precedence is greater than or equal to the precedence of the stack top of the operator stack, then push thecharacter to the operator stack.
        - If the character's precedence is less than the precedence of the stacktop of the operator stack then do Process (as explained above) until character's precedence is less or stack is not empty.
3. If the character is "(", then push it onto the operator stack.
4. If the character is ")", then do ***Process*** (as explained above) until the corresponding "(" is encountered in operator stack.

Infix Expression: 2 * (5 *(3+6)) / 15-2				
Token	**Action**	**Operand Stack**	**Operator Stack**	**Notes**
2	Push it to operand stack	2		
*	Push it to operator stack	2	*	
(	Push it to operator stack	2	(*	
5	Push it to operand stack	5 2	(*	
*	Push it to operator stack	5 2	* (*	
(	Push it to operator stack	5 2	(* (*	
3	Push it to operand stack	3 5 2	(* (*	
+	Push it to operator stack	3 5 2	+ (* (*	
6	Push it to operand stack	6 3 5 2	+ (* (*	
)	Pop 6 and 3 from operand stack	5 2	+ (* (*	Do process until (is popped from operator stack
	Pop + from operator stack	5 2	(* (*	
	Do 6+3 = 9	5 2	(* (*	
	Push 9 to operand stack	9 5 2	(* (*	
	Pop (from operator stack	9 5 2	* (*	
)	Pop 9 and 5 from operand stack	2	* (*	Do process until (is popped from operator stack
	Pop * from operator stack	2	(*	
	Do 9*5 = 45	2	(*	
	Push 45 into operand stack	45 2	(*	
	Pop (from operator stack	45 2	*	
/	Push / into operator stack	45 2	/ *	/ has equal precedence to *
15	Push 15 to operand stack	15 45 2	/ *	
-	Pop 15 and 45 from operand stack	2	/ *	- has lower precedence than / , do the process
	Pop / from operator stack	2	*	
	Do 45/15 = 3	2	*	
	Push 3 into operand stack	3 2	*	
	Pop 3 and 2 from operand stack		*	- has lower precedence than * , do the process
	Pop * from operator stack			
	Do 3*2 = 6			
	Push 6 into operand stack	6		
	Push - into operator stack	6	-	- has equal precedence to +
2	Push 2 into operand stack	2 6	-	
	Pop 2 and 6 from the operand stack			Given expression is iterated, do Process till operator stack is not empty, It will give the final result
	Pop - from operator stack			
	Do 6-2 =4			
	Push 4 to operand stack	4		

Evaluation of Postfix Expressions(Polish Postfix notation)

Postfix notation is a notation for writing arithmetic expressions in which the operands appear before their operators. There are no precedence rules to learn,and parentheses are never needed. Because of this simplicity. Let's assume the below

- Operands are real numbers in single digits. (Read: **Evaluation of PostfixExpressions for any Number**)
- Permitted operators: +,-, *, /, ^(exponentiation)
- Blanks are NOT permitted in expression.
- Parenthesis are permitted

Example:

Postfix: 54+

Output: 9

Explanation: Infix expression 5+ 4 which resolves to 9

Postfix: 2536+**5/2-

Output: 16

Explanation: Infix expression of above postfix is: 2 * (5 *(3+6))/5-2 which resolves to 16

Approach: Use Stack

Algorithm:

Iterate through given expression, one character at a time

1. If the character is an operand, push it to the operand stack.
2. If the character is an operator,
    1. pop an operand from the stack, say it's s1.
    2. pop an operand from the stack, say it's s2.
    3. perform **(s2 operator s1)** and push it to stack.

3. Once the expression iteration is completed, The stack will have the finalresult. Pop from the stack and return the result.

Please see the walkthrough of an example below for more understanding.

Postfix Expression : 2536+**5/2-		
Token	**Action**	**Stack**
2	Push **2** to stack	[2]
5	Push **5** to stack	[2, 5]
3	Push **3** to stack	[2, 5, 3]
6	Push **6** to stack	[2, 5, 3, 6]
+	Pop **6** from stack	[2, 5, 3]
	Pop **3** from stack	[2, 5]
	Push **3+6 =9** to stack	[2, 5, 9]
*	Pop **9** from stack	[2, 5]
	Pop **5** from stack	[2]
	Push **5*9=45** to stack	[2, 45]
*	Pop **45** from stack	[2]
	Pop **2** from stack	[]
	Push **2*45=90** to stack	[90]
5	Push **5** to stack	[90, 5]
/	Pop **5** from stack	[90]
	Pop **90** from stack	[]
	Push **90/5=18** to stack	[18]
2	Push **2** to stack	[18, 2]
-	Pop **2** from stack	[18]
	Pop **18** from stack	[]
	Push **18-2=16** to stack	[16]
Result : 16		

Evaluation of Postfix Expressions(Polish Postfix notation)

Earlier we had discussed how to evaluate postfix expressions where operands areof single-digit. In this article, we will discuss how to evaluate postfix expressions for any number (not necessarily single digit.)
Postfix notation is a notation for writing arithmetic expressions in which theoperands appear before their operators.
Let's assume the below

- Operands are real numbers (could be multiple digits).
- Permitted operators: +,-, *, /, ^(exponentiation)
- Blanks are used as a **separator** in expression.
- Parenthesis are permitted

Example:
Postfix: 500 40+

Output: 540

Explanation: Infix expression of above postfix is: 500 + 40 which resolves to540

Postfix: 20 50 3 6 + * * 300 / 2 -

Output: 28

Explanation: Infix expression of above postfix is: 20 * (50 *(3+6))/300-2 which resolves to 28

Approach: Use Stack
Algorithm:
Iterate through given expression, one character at a time

1. If the character is a digit, initialize number = 0
   - while the next character is digit

1. do number = number*10 + currentDigit
- push number to the stack.

2. If the character is an operator,
   - pop operand from the stack, say it's s1.
   - pop operand from the stack, say it's s2.
   - perform **(s2 operator s1)** and push it to stack.
3. Once the expression iteration is completed, The stack will have the finalresult. pop from the stack and return the result.

Please see the walkthrough of an example below for more understanding.

Evaluation of Prefix Expressions(Polish Notation)
Earlier we had discussed how to evaluate prefix expression where

Postfix Expression : 20 50 3 6 + * * 300 / 2 -		
Token	**Action**	**Stack**
2	Push **20** to stack	[20]
5	Push **50** to stack	[20, 50]
3	Push **3** to stack	[20, 50, 3]
6	Push **6** to stack	[20, 50, 3, 6]
+	Pop **6** from stack	[20, 50, 3]
	Pop **3** from stack	[20, 50]
	Push **3+6 =9** to stack	[20, 50, 9]
*	Pop **9** from stack	[20, 50]
	Pop **50** from stack	[20]
	Push **50*9=450** to stack	[20, 450]
*	Pop **450** from stack	[20]
	Pop **20** from stack	[]
	Push **20*450=9000** to stack	[9000]
300	Push **300** to stack	[9000, 300]
/	Pop **300** from stack	[9000]
	Pop **9000** from stack	[]
	Push **9000/300=30** to stack	[30]
2	Push **2** to stack	[30, 2]
-	Pop **2** from stack	[30]
	Pop **30** from stack	[]
	Push **30-2=28** to stack	[28]
Result : 28		

operands are ofsingle-digit. Here we will discuss how to evaluate prefix expression for any number (not necessarily single digit.)

Prefix notation is a notation for writing arithmetic expressions in which the operands appear after their operators. Let's assume the below

- Operands are real numbers (could be multiple digits).
- Permitted operators: +,-, *, /, ^(exponentiation)
- Blanks are used as a **separator** in expression.
- Parenthesis are permitted

Example:
Postfix: + 500 40

Output: 540

Explanation: Infix expression of the above prefix is: 500 + 40 which resolvesto 540

Postfix: - / * 20 * 50 + 3 6 300 2

Output: 28

Explanation: Infix expression of above prefix is: 20 * (50 *(3+6))/300-2 which resolves to 28

Approach: Use Stack
Algorithm:
Reverse the given expression and Iterate through it, one character at a time

1. If the character is a digit, initialize String temp;
   - while the next character is not a digit
     - do temp = temp + currentDigit
   - convert Reverse temp into Number.
   - push Number to the stack.
2. If the character is an operator,
   - pop the operand from the stack, say it's s1.
   - pop the operand from the stack, say it's s2.
   - perform **(s1 operator s2)** and push it to stack.
3. Once the expression iteration is completed, The stack will have the finalresult. Pop from the stack and return the result.

Please see the walkthrough of an example below for more understanding.

Prefix Expression : - / * 20 * 50 + 3 6 300 2		
Reversed Prefix Expression: 2 003 6 3 + 05 * 02 * / -		
Token	**Action**	**Stack**
2	Reverse **2**, Push **2** to stack	[2]
003	Reverse **003**, Push **300** to stack	[2, 300]
6	Reverse **6**, Push **6** to stack	[2, 300, 6]
3	Reverse **3**, Push **3** to stack	[2, 300, 6, 3]
+	Pop **3** from stack	[2, 300, 6]
	Pop **6** from stack	[2, 300]
	Push **3+6 =9** to stack	[2, 300, 9]
05	Reverse **05**, Push **50** to stack	[2, 300, 9, 50]
*	Pop **50** from stack	[2, 300, 9]
	Pop **9** from stack	[2, 300]
	Push **50*9=450** to stack	[2, 300, 450]
02	Reverse **02**, Push **20** to stack	[2, 300, 450, 20]
*	Pop **20** from stack	[2, 300, 450]
	Pop **450** from stack	[2, 300]
	Push **20*450=9000** to stack	[2, 300, 9000]
/	Pop **9000** from stack	[2, 300]
	Pop **300** from stack	[2]
	Push **9000/300=30** to stack	[2, 30]
-	Pop **30** from stack	[2]
	Pop **2** from stack	[]
	Push **30-2=28** to stack	[28]
Result : 28		

POLYNOMIAL ADDITION

Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients who have same variable powers.**Example:**

Input:

1st number = $5x^2 + 4x^1 + 2x^0$2nd number = $-5x^1 - 5x^0$

Output:

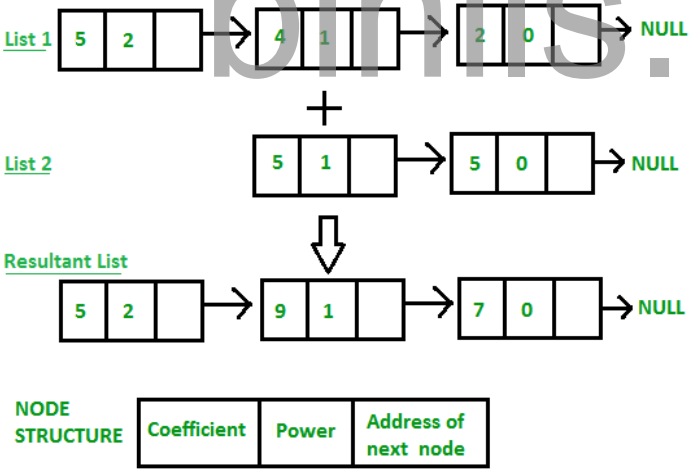
$5x^2-1x^1-3x^0$

Input:

1st number = $5x^3 + 4x^2 + 2x^0$2nd number = 5x^1 - 5x^0

Output:

$5x^3 + 4x^2 + 5x^1 - 3x^0$



Polynomial Addition

Type Declaration:struct node

```
{int coeff;int pow;
        struct node *next;
}*poly1=NULL,*poly2=NULL,*poly=NULL;
```

```
void polyadd(struct node *poly1,struct node *poly2,struct node *poly)
{
        while((poly1->next !=NULL)&& (poly2->next!=NULL))
        {
        if(poly1->pow > poly2->pow)
        {
pol->pow=poly1-
>pow;          pol-
>coeff=poly1-
>coff;
pol1=poly1-
>next;
        }
else if(poly1->pow<poly2->pow)
        {
poly-
>pow=poly2>po
w;pol-
>coeff=poly2-
>coeff;poly2=poly2-
>next;
        }
       else
       {
           poly-
           >pow=poly1-
           >pow;poly-
           >coeff=poly1-
           >coeff+poly2-
           >coeff;
           poly1=po
           ly1-
           >next;
           poly2=poly2->next;
       }
           poly->next=(struct node
           *)malloc(sizeof(struct
           node));poly=poly-
           >next;
           poly->next=NULL;
}
```

```
while(poly1->next !=NULL)
{
poly->pow=poly1-
>pow; poly-
>coeff=poly1-
>coeff;
poly1=poly1-
>next;
}
while(poly2->next!=NULL)
{poly->pow=poly2-
>pow;          poly-
>coeff=poly2-
>coeff;
poly2=poly2-
>next;
}
}
```

Addition of polynomials can be solved in two methods.

(i) By arranging the like terms together and then add.

For example:

1. Add: 5x + 3y, 4x – 4y + z and -3x + 5y + 2z

First we need to write in the addition

form.Thus, the required addition

= (5x + 3y) + (4x – 4y + z) + (-3x + 5y + 2z)

= 5x + 3y + 4x – 4y + z - 3x + 5y + 2z

Now we need to arrange all the like terms and then all the like terms are added.

= 5x + 4x - 3x + 3y – 4y + 5y + z + 2z

= 6x + 4y + 3z

2. Add: $3a^2 + ab – b^2$, $-a^2 + 2ab + 3b^2$ and $3a^2 – 10ab + 4b^2$

First we need to write in the addition form.

Thus, the required addition

= $(3a^2 + ab – b^2) + (-a^2 + 2ab + 3b^2) + (3a^2 – 10ab + 4b^2)$

= $3a^2 + ab – b^2 - a^2 + 2ab + 3b^2 + 3a^2 – 10ab +$

$4b^2$ Here, we need to arrange the like terms and then

add

= $3a^2 - a^2 + 3a^2 + ab + 2ab – 10ab – b^2 + 3b^2 + 4b^2$

$= 5a^2 - 7ab + 6b^2$

(ii) By arranging expressions in lines so that the like terms with their signs are one below the other i.e. like terms are in same vertical column and then add the differentgroups of like terms.

For example:

1. Add: 7a + 5b, 6a – 6b + 3c and -5a + 7b + 4c

First we will arrange the three expressions one below the other, placing the like terms in the same column.

Now the like terms are added by adding their coefficients with their signs.

Therefore, adding 7a + 5b, 6a – 6b + 3c and -5a + 7b + 4c is 8a + 6b + 7c.

2. Add: $3x^3 – 5x^2 + 8x + 10$, $15x^3 – 6x – 23$, $9x^2 – 4x + 15$ and $-8x^3 + 2x^2 – 7x$.

First we will arrange the like terms in the vertical column and then the like terms are added by adding their coefficients with their signs.

Therefore, adding $3x^3 – 5x^2 + 8x + 10$, $15x^3 – 6x – 23$, $9x^2 – 4x + 15$ and $-8x^3 + 2x^2 – 7x$ is $10x^3 + 6x^2 – 9x + 2$.

Thus, we have learnt how to solve addition of polynomials in both the methods.