UNIT III
LINEAR 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.

INTRODUCTION

Definition

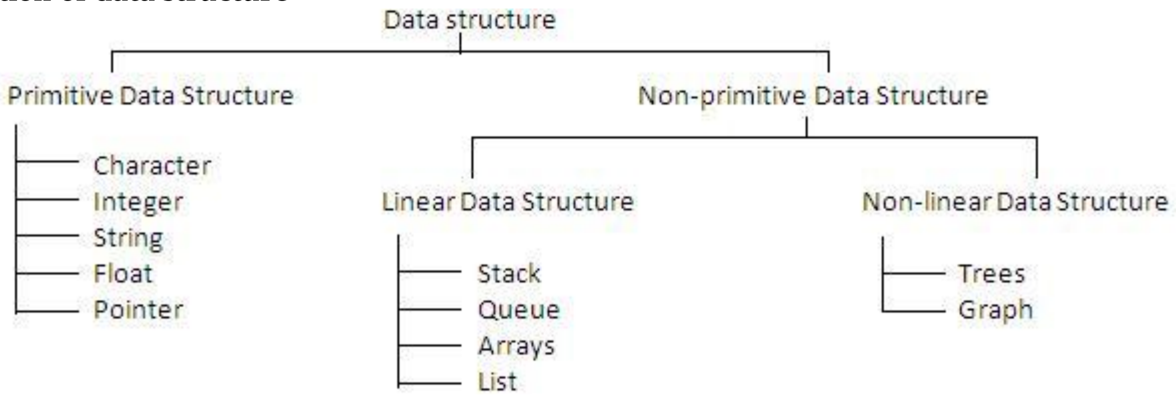
Data structure is a particular way of organizing, storing and retrieving data, so that it can be used efficiently. It is the structural representation of logical relationships between elements of data.

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

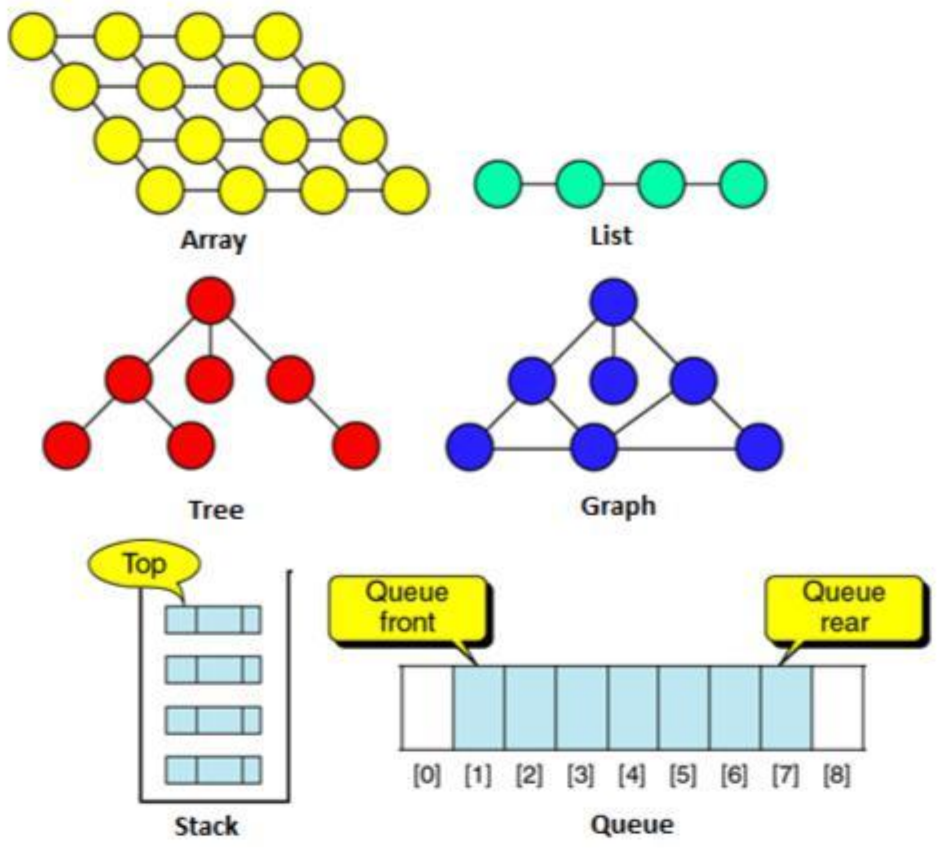
o Compiler design (Hash tables)	o Operating system
o Database management system (B+Trees)	o Statistical analysis package
o Numerical analysis (Graphs)	o Graphics
o Artificial intelligence	o Simulation

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 sophisticated data 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/element has a single successor) o 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)
  - o General list (Data can be inserted or deleted anywhere in the list: at the beginning, in the middle or at the end)
- **Non-linear data structure**- One element can be connected to more than two adjacent elements.(Each node/element can have more than one successor)
  - o Tree (Each node could have multiple successors but just one predecessor)
  - o 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 module is a logical unit and does a 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 to breaking down software into different parts called modules.
- **Advantages** of modularity

  o It is easier to debug small routines than large routines.
  o Modules are easy to modify and to maintain.
  o Modules can be tested independently.
  o Modularity provides reusability.
  o 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 so by 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 uses the ADTs.

LIST ADT

- List is a linear collection of ordered elements. General form of the list of size N is: $\mathbf{A_0, A_1, \ldots, A_{N-1}}$
  - Where A_1 - First element
  - If the element at position 'i' is Ai then its successor is Ai+1 and its predecessor is Ai-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

o Array implementation o Linked List implementation o 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 index enclosed in parenthesis. In array implementation, elements of list are stored in contiguous cells of an array. Find Kth operation takes constant time. **PrintList**, **Find** operations take linear time.
- **Advantages**
  - **Searching** an array for an **individual** element can be **very efficient**
  - Fast, random access 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 is fixed.
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 is O(n). On average, half the list needs to be moved for either operation, so linear time is still required.
5. Memory is wasted, as the memory remains allocated to the array throughout the program execution even few nodes are stored.

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

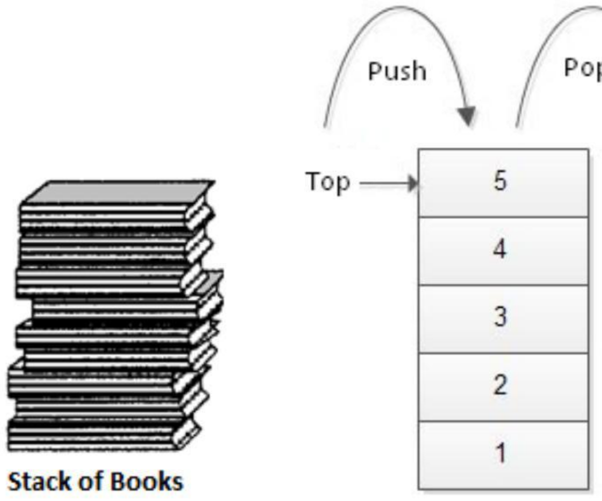
```
void Find (int X)
{
        int i,f=0;
        for(i=0;i<N;i++)
                if(a[i]==x)
                {
                        f=1;
                        break;
                }
                if (f==1)
```

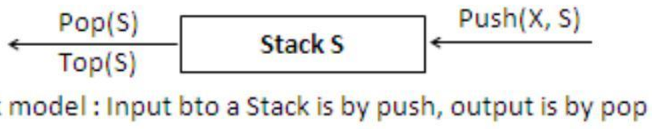
```
                    printf("Element found");
            else
                    printf("Element not found");
}
```

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".



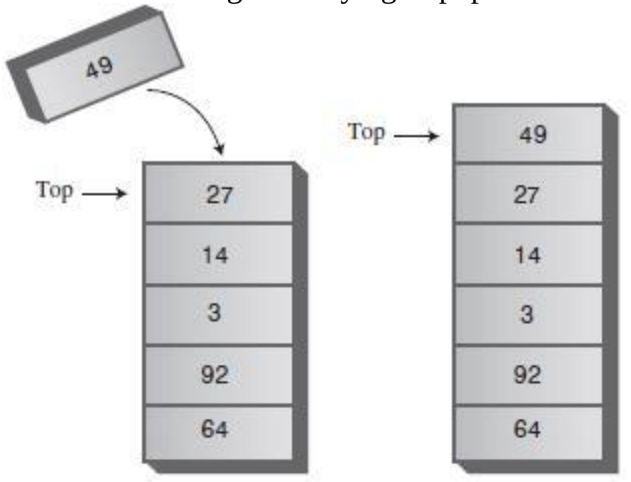

Pop(S)
Top(S)
Stack S
Push(X, S)
Stack model : Input bto a Stack is by push, output is by pop

Operations on stacks

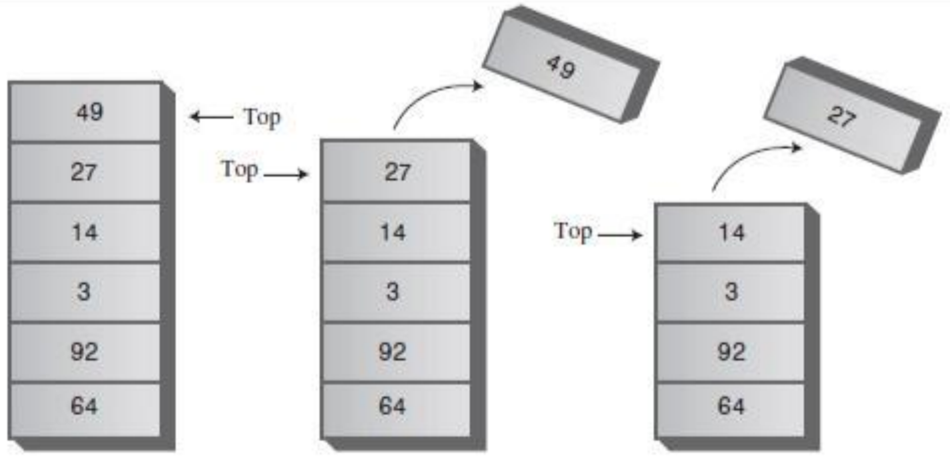
- **Push** - Inserts new item to the top of the stack. After the push, the new item becomes the top.
- **Pop** - Deletes top item from the stack. The next older item in the stack becomes the top.
- **Top** - Returns a copy of the top item on the stack, but does not delete it.
- **MakeEmpty** - Sets stack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



New item pushed on Stack



Two items popped from Stack

APPLICATIONS OF STACKS

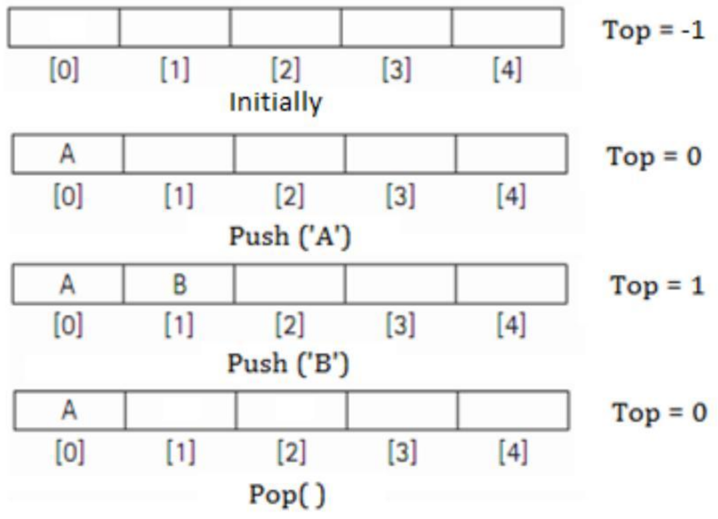
- **Recursion** - Example, Factorial, Tower of Hanoi.
- **Balancing Symbols**, i.e., finding the unmatched/missing parenthesis. For example, ((A+B)/C and (A+B)/C). Compilers often use stacks **to perform syntax analysis of language statements**.
- C**onversion** of infix expression to postfix expression 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.

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

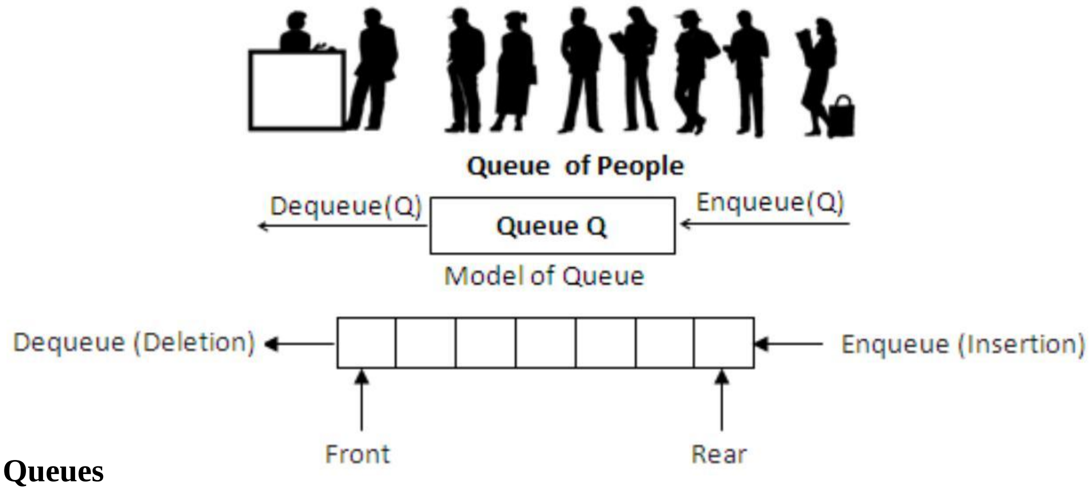
Array implementation of Stack

```
#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");
        else
        {
                printf("Enter item: ");
                scanf("%d",&item);
                top++;
                stack[top] = item;
                printf("Item pushed = %d", item);
        }
}

void pop()
{
        if(top == -1)
        printf("Stack is empty");
        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]);
        }
}
```

QUEUE (LINEAR QUEUE)

➢ **Definition**

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.



➢ **Types of Queues**

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

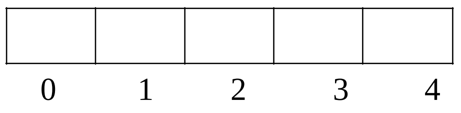
➢ **Conditions**

- **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.

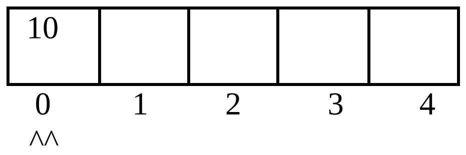
➢ **Implementation of Queue**

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

➢ **Array implementation of Linear Queue**



Empty Queue F = R = -
1



F R

After Enqueue (10)

10	3			
0	1	2	3	4
^	^			
F	R			

After Enqueue (3)

10	3	41		
0	1	2	3	4
^		^		
F		R		

After Enqueue (41)

	3	41		
0	1	2	3	4
	^	^		
	F	R		

After Dequeue ()

	3	41	76	
0	1	2	3	4
	^		^	
	F		R	

After Enqueue (76)

	3	41	76	66
0	1	2	3	4
	^			^
	F			R

After Enqueue (66)

		41	76	66
0	1	2	3	4
		^		^
		F		R

After Dequeue ()

- There is one potential **problem** with array implementation. From the above queue, now if we attempt to add more elements, 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 the beginning. This is known as a **circular array** implementation.

Array implementation of Linear Queue

```
#include <stdio.h>
#include<conio.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]);
}
```

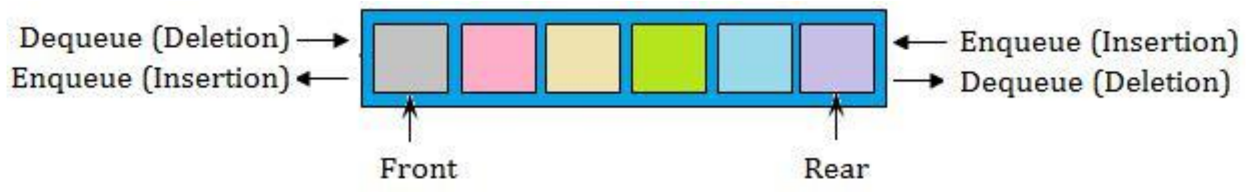
Circular Queue

- In circular queues the elements Q[0],Q[1],Q[2] .... Q[n – 1] is represented in a circular fashion with Q[1] following Q[n]. A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full.
- 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 the relation: **Rear = (Rear + 1) % SIZE**
- After deleting an element from circular queue the position of the front end is calculated by the relation: **Front= (Front + 1) % SIZE.**
- After locating the position of the new element to be inserted, rear, compare it with front. If R**ear = Front**, the queue is full and **cannot be inserted anymore**.
- No of elements in a queue = **(Rear – Front + 1) % N**

Deque - Double Ended QUEeue

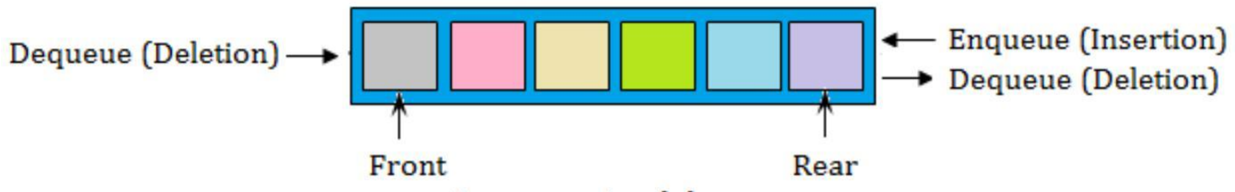
Definition

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.**

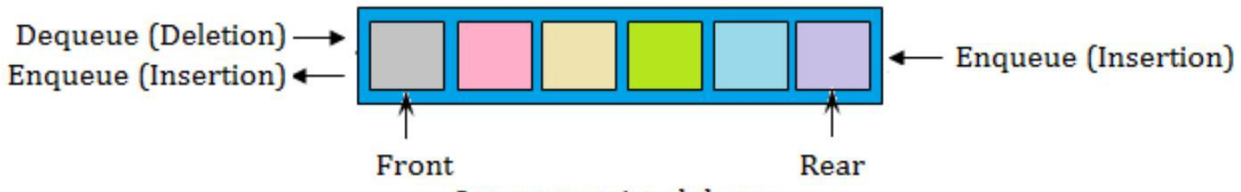


Deque

➢ **Types of deques**
1. Input restricted deque
2. Output restricted deque



Input restricted deque



Output restricted deque

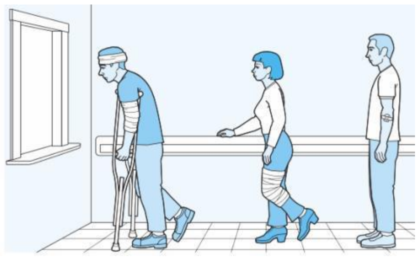
✓ An **input restricted deque** is a deque, which allows insertion at only 1 end, rear end, but allows deletion at both ends, rear and front end of the lists.

✓ 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.

Priority Queue

- **Definition**

- Priority Queue is a queue where each element is assigned a priority. The priority may **implicit** (decided by its value) or **explicit** (assigned). In priority queue, the elements are deleted and processed by following rules.
  - An element of higher priority is processed before any element of lower priority.
  - Two elements with the same priority are processed according to the order in which they were inserted to the queue.
- Example for priority queue:
  - In a telephone answering system, calls are answered in the order in which they are received;
  - Hospital emergency rooms see patients in priority queue order; the patient with the most severe injuries sees the doctor first.



Queue of people with priority

- A node in the priority queue will contain Data, Priority and Next field. Data field will store the actual information; Priority field will store its corresponding priority of the data and Next will store the address of the next node.
- When an element is inserted into the priority queue, it will check the priority of the element with the element(s) present in the linked list to find the suitable position to insert. The node will be inserted in such a way that the data in the priority field(s) is in ascending order. We do not use rear pointer when it is implemented using linked list, because the new nodes are not always inserted at the rear end.

- **Types of priority queues**

1. **Ascending priority queue** - It is a queue in which items can be inserted arbitrarily (in any order) and from which only the **smallest** item can be deleted first.
2. **Descending priority queue** - It is a queue in which items can be inserted arbitrarily (in any order) and from which only the **largest** item can be deleted first.

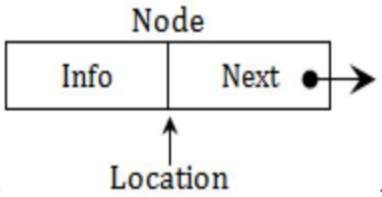
Applications of queue

- When jobs are submitted to a printer, they are arranged in order of arrival. Thus, essentially, jobs sent to a line printer are placed on a queue.
- Virtually every real-life line is (supposed to be) a queue. For instance, lines at ticket counters are queues, because service is first-come first-served.
- Another example concerns computer networks. There are many network setups of personal computers in which the disk is attached to one machine, known as the file server. Users on other machines are given access to files on a first-come first-served basis, so the data structure is a queue.
- Calls to large companies are generally placed on a queue when all operators are busy.
- There are several algorithms that use queues to solve problems easily. For example, BFS, Binary tree traversal etc.
- Round robin techniques for processor scheduling is implemented using queue.

LINKED LIST

Definition

Linked list is a**dynamic** data structure which is an ordered collection of homogeneous dataelements 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. The link contains (the pointer) the address of the next element in the list.



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.

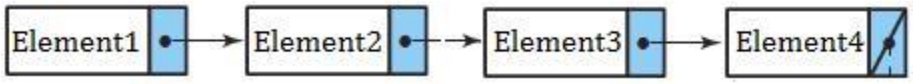
Types of linked list

1. Singly linked 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**

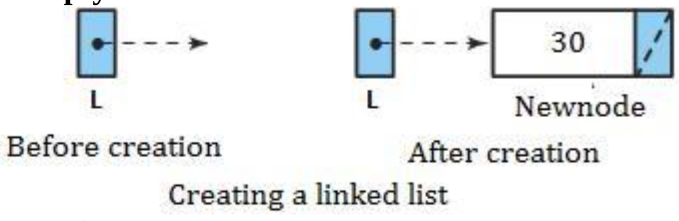
In singly linked list, each element (except the first one) has a unique predecessor, and each element (except the last one) has a unique successor. Each node contains two parts: **data** or **Info** and **link**. The data holds the application data to be processed. The link contains the address of the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



- **Sentinel Node**
- 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.

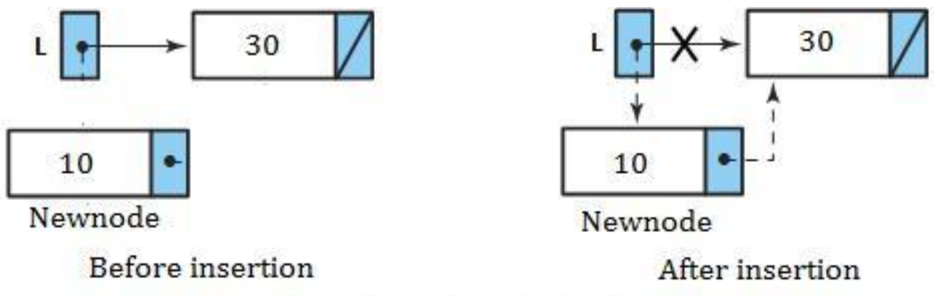
➢ **Insertion**

a. Creating a newnode from empty List



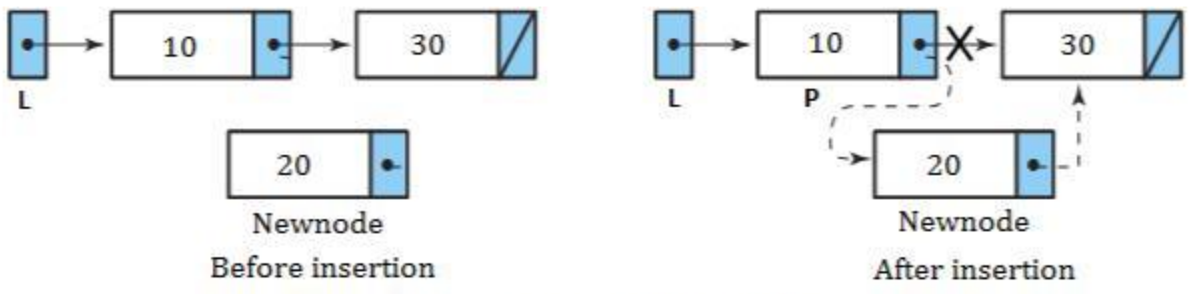
Creating a linked list

b. Inserting a node to the front of list

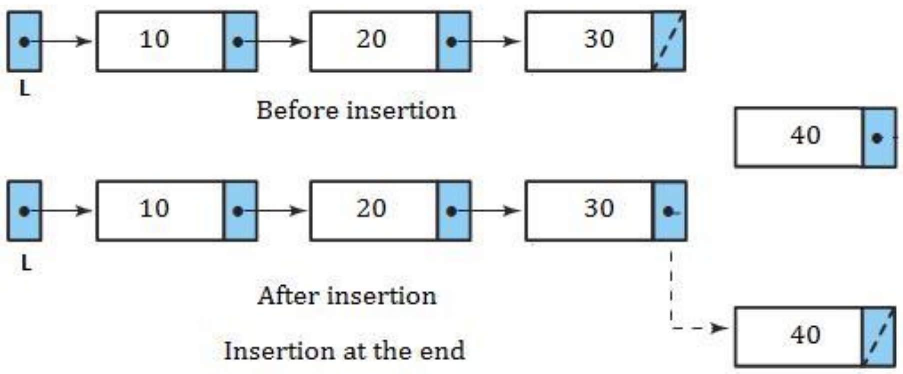


Insertion at the beginning

c. Inserting a node in the middle



Insertion in the middle

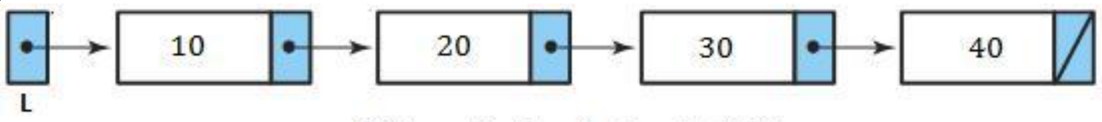


Insertion at the end

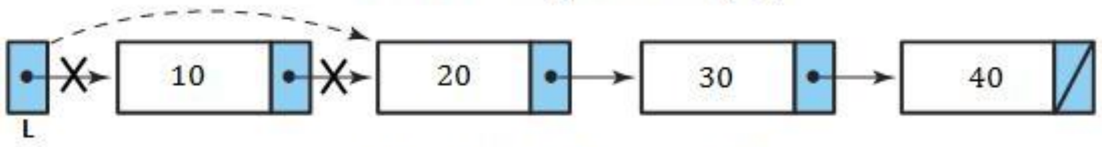
d. Inserting a node to the end of list

➢ **Deletion**

a. Inserting a node to the front of list

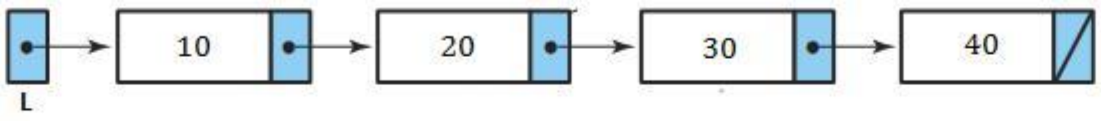


Before deleting first node (10)

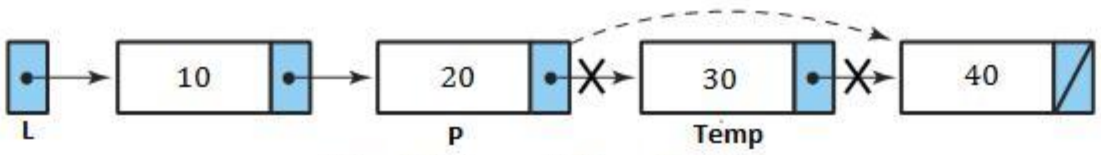


After deleting first node (10)

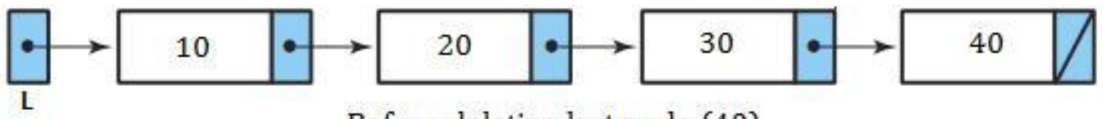
b. Deleting the middle node



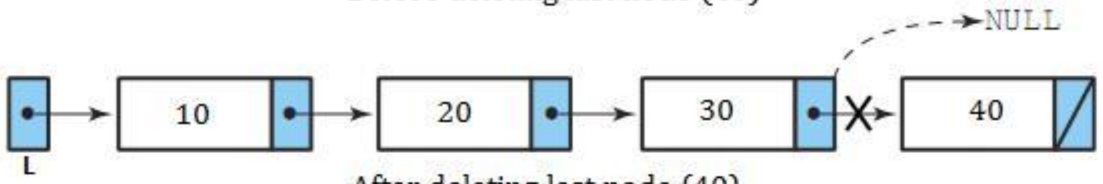
Before deleting middle node (30)



After deleting middle node (30)



Before deleting last node (40)



After deleting last node (40)

Singly linked list implementation

Type Declarations

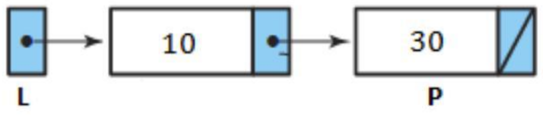
```
struct node
{
    int data;
    struct node *next;
}*head=NULL:
typedef struct Node *position;
```

Routine to check whether the List is empty

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

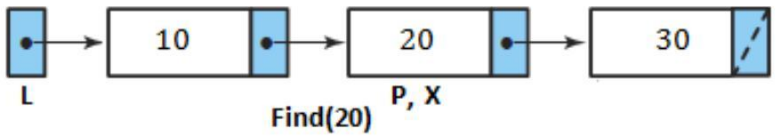
Routine to check whether the current position is last

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



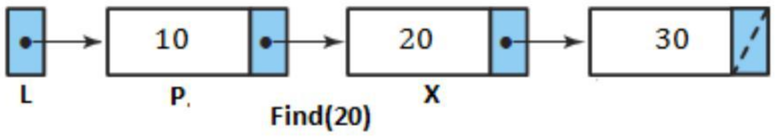
Find Routine

```
/* 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;
}
```



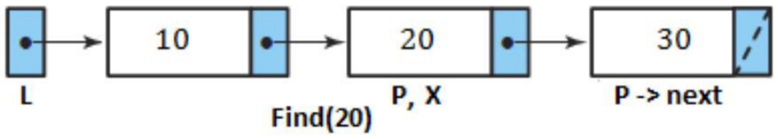
FindPrevious Routine

```
/* Returns the previous position of X in L */
position FindPrevious (int X)
{
        position P;
        P = head;
        while( (P->Next!= NULL) && (P->Next->data != X) )
                P = P->Next;
        return P;
}
```



FindNext Routine

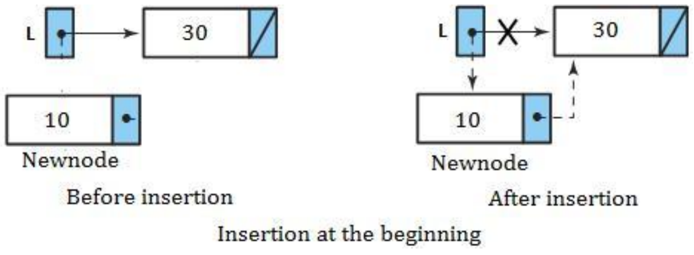
```
/* Returns the position of X in L; NULL if not found */
Position Find (int X)
{
        position p;
        P = head->next;
        while( (p!= NULL) && (p->data != X) )
                p = p->next;
        return P->next;
}
```



```
void traversal ()
{
        position p;
        P = head->next;
        while( p!= NULL)
        {
                printf(p->data);
                p = p->next;
        }
}
```

Insertion

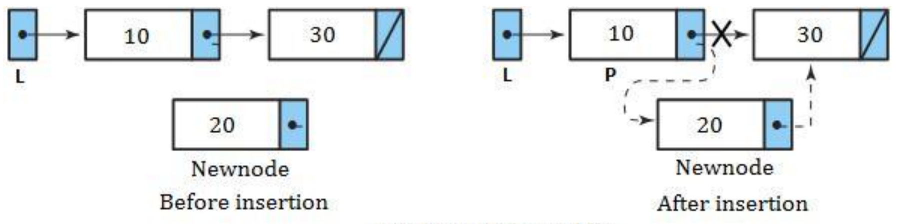
Inserting a node to the front of list



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;
        }
}
```

b.Inserting a node in the middle

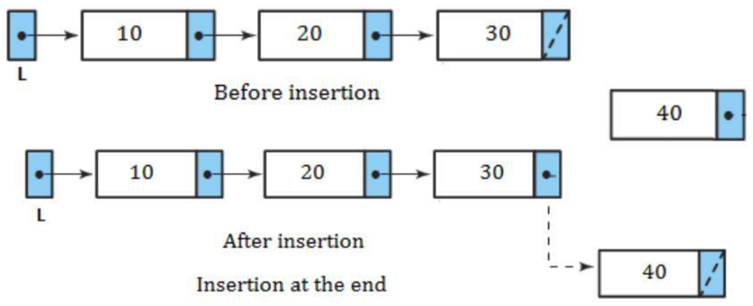


Insertion in the middle

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 list



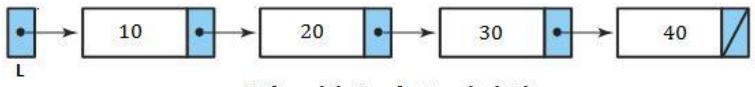
Insertion at the end

Insert at Last

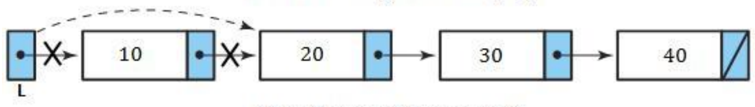
```
void Insert_last (int X)
{
        position NewNode,P;
        NewNode = malloc (sizeof(struct Node));
        if(NewNode != NULL)
        {
                while(P->next!=NULL)
                 P= P->next;
                 NewNode->data = X;
                 NewNode->next = NULL; P-
           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

➢ **Deletion**

Deleting a node to the front of list



Before deleting first node (10)

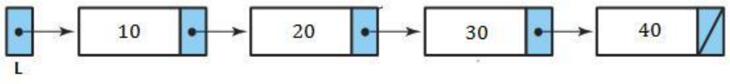


After deleting first node (10)

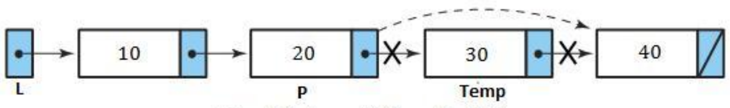
Delete at Beginning

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

Deleting the middle node



Before deleting middle node (30)

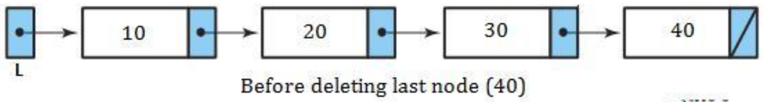


After deleting middle node (30)

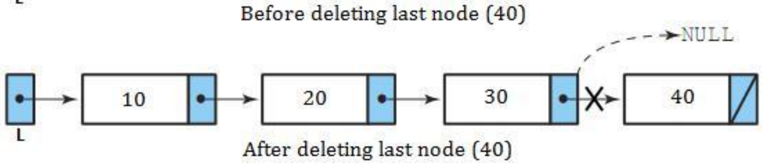
Delete at Middle Routine

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

Deleting the last node



Before deleting last node (40)



After deleting last node (40)

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

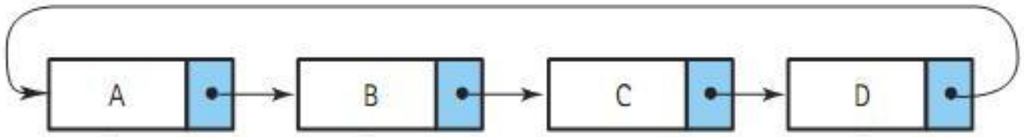
- With a singly linked list structure, given a pointer to a node anywhere in the list, we can access all the nodes that follow but none of the nodes that precede it. We must always have a pointer to the beginning of the list to be able to access all the nodes in the list.In a circular linked list, every 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

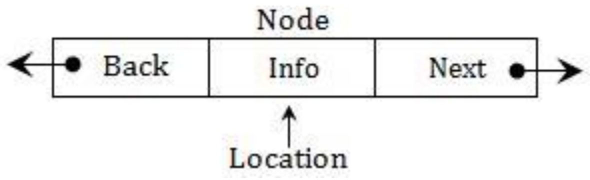
- A circular linked list is one, which has no beginning and no end. Circular linked list is a list in which every node has a successor; the "last" element is succeeded by the "first" element. We can start at any node in the list and traverse the entire list.

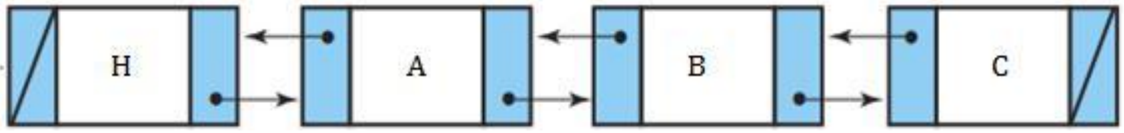


DOUBLY LINKED LIST

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

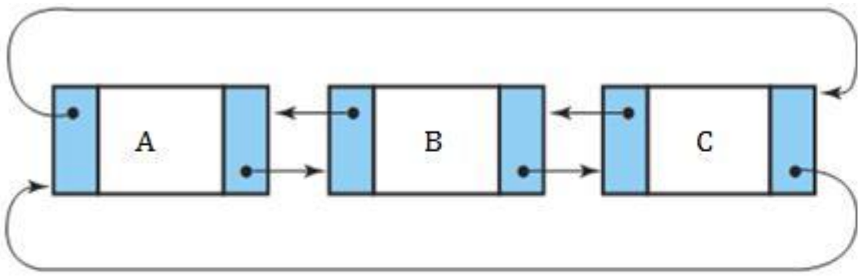
DOUBLY CIRCULAR LINKED LIST

➢ **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.



A Doubly Linked List

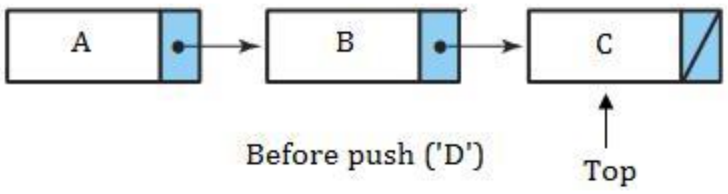
APPLICATIONS OF LINKED LIST

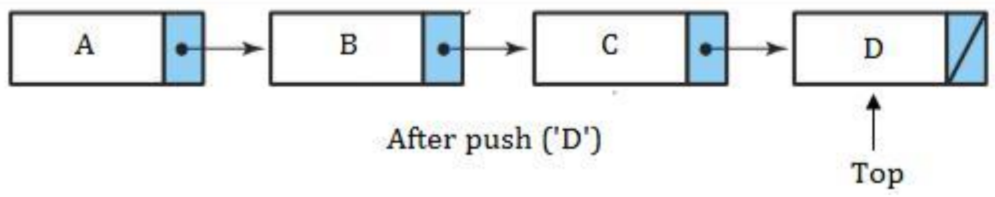
All kinds of dynamic allocation related problems can be solved using linked lists. Some of the applications are given below:

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

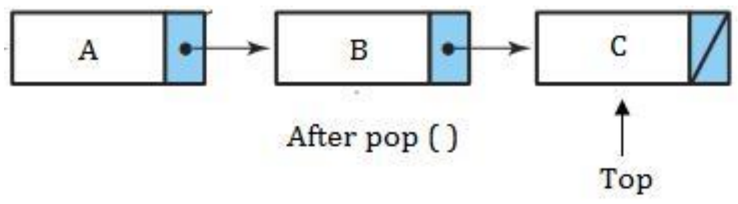
Linked list implementation of stack

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.





Push operation



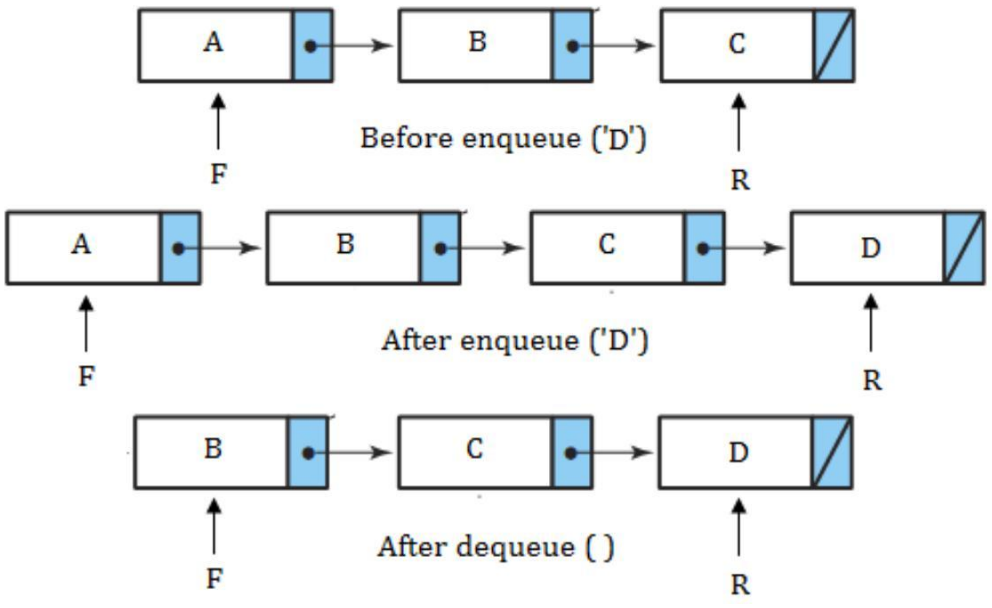
Pop operation

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;
     }
   }
}
```

```
void pop()
{
    position p;
    p = top->next;

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

void display()
{
        struct node *p;
        if (top == NULL)
                printf("Stack is empty");
        else
        {
                p = top;
                while(p != NULL)
                {
                        printf("\n%d", p->data);
                        p = p->next;
                }
        }
}
```

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;
        }
    }
}
```

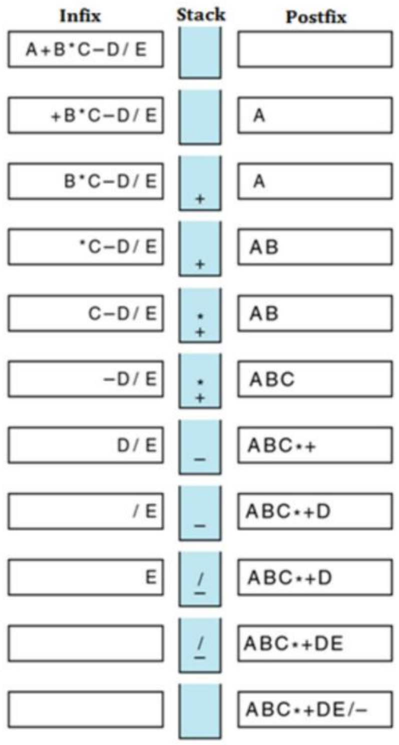
APPLICATIONS OF STACKS

- **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

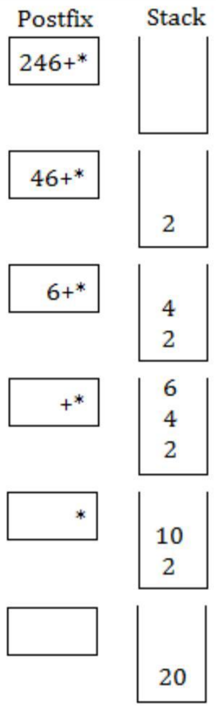
1. Scan the infix expression from left to right. Repeat Steps 3 to 6 for each element of expression until the stack is empty.
2. If an operand is encountered, add it to the postfix expression.
3. If an opening parenthesis is encountered, push it onto the stack and do not remove it until closing parenthesis 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	
(	Highest
^	-
*, /	-
+, -	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.

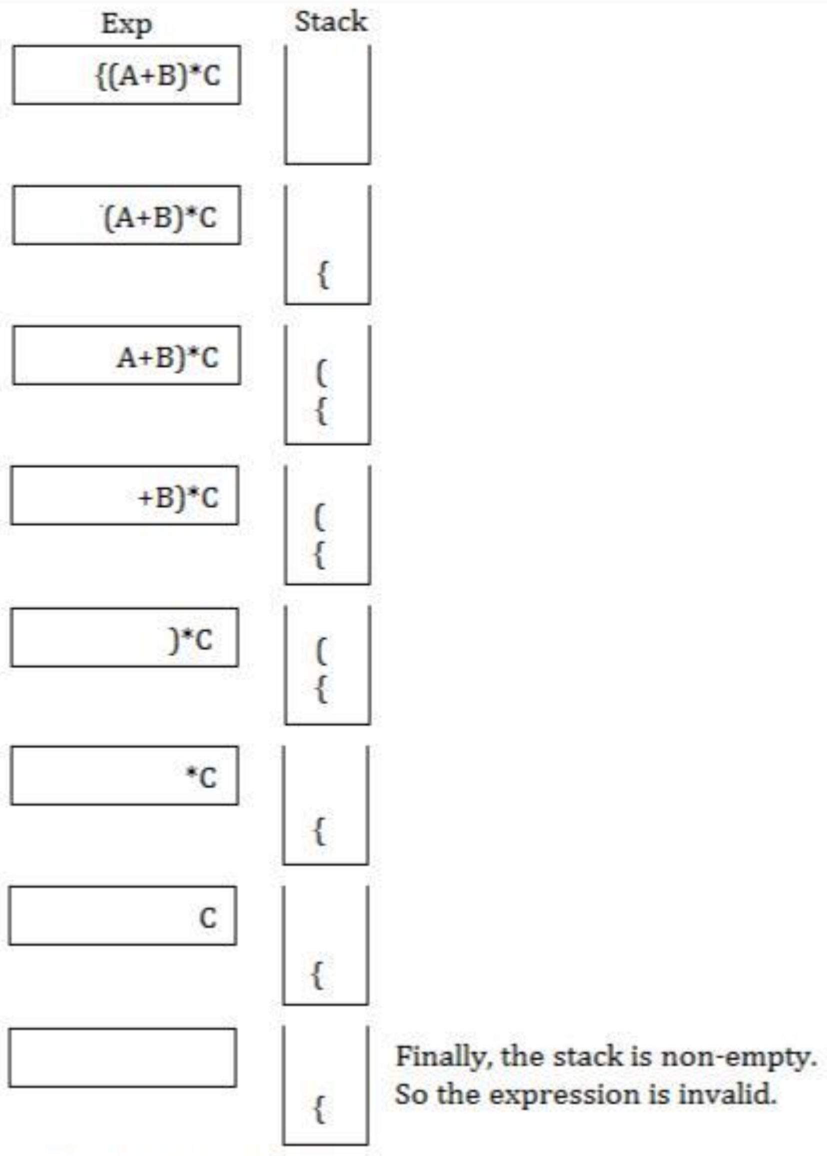


Evaluation of postfix expression

Balancing parenthesis

- One common programming problem is unmatched parenthesis in an algebraic expression. When parentheses are unmatched, two types of errors can occur:
  - o Opening parenthesis can be missing. For example, [A+B]/C}.
  - o 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.

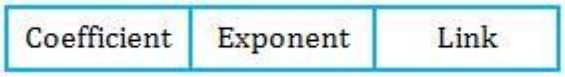

Exp
Stack
{(A+B)*C
(A+B)*C
{
A+B)*C
(
{
+B)*C
(
{
)*C
(
{
*C
{
C
{
{
Finally, the stack is non-empty.
So the expression is invalid.

Balancing parenthesis

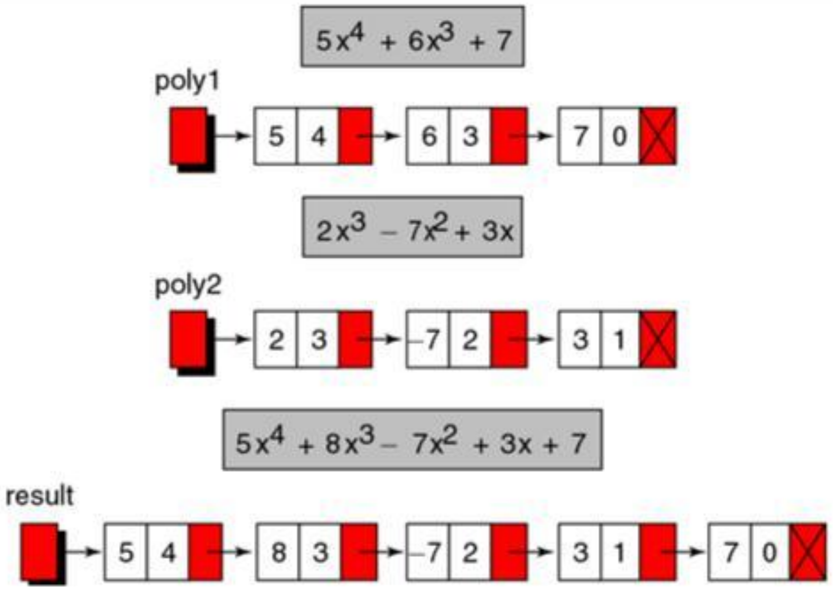
Polynomial ADT

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.

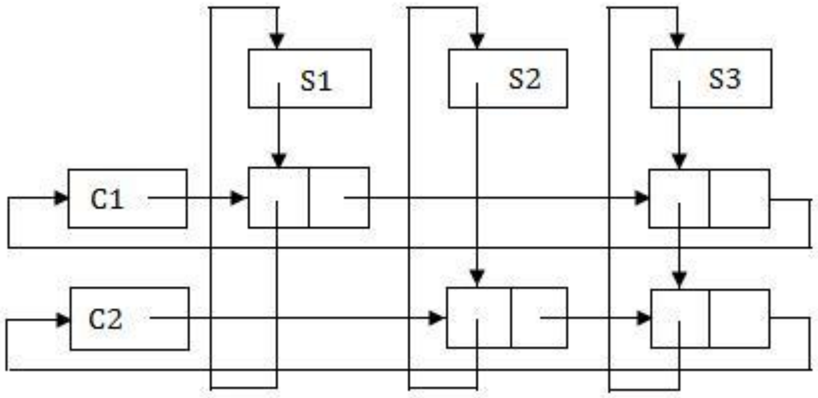


For example,



➢ **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

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)
.           {
                        poly->pow=poly1->pow;
                        poly->coeff=poly1->coeff;
                        poly1=poly1->next;
            }
          else if(poly1->pow<poly2->pow)
          {
                        poly->pow=poly2->pow;
                        poly->coeff=poly2->coeff;
                        poly2=poly2->next;
        }
        else
        {
            poly->pow=poly1->pow;
            poly->coeff=poly1->coeff+poly2->coeff;
            poly1=poly1->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;
}
}
```