Unit II

Linear Data Structure

Unit 2 Linear Data Structure

- Array
- Stack
- Queue
- Linked List and its types
  - Various Representations
  - Operations & Application of Linear Data Structures.

Introduction to data structure

- A data structure is a particular way of organizing data in a computer so that it can be used effectively.
- The idea is to reduce the space and time complexities of different tasks.
- some popular linear data structures.
  - Array
  - Linked List
  - Stack
  - Queue

Data Type

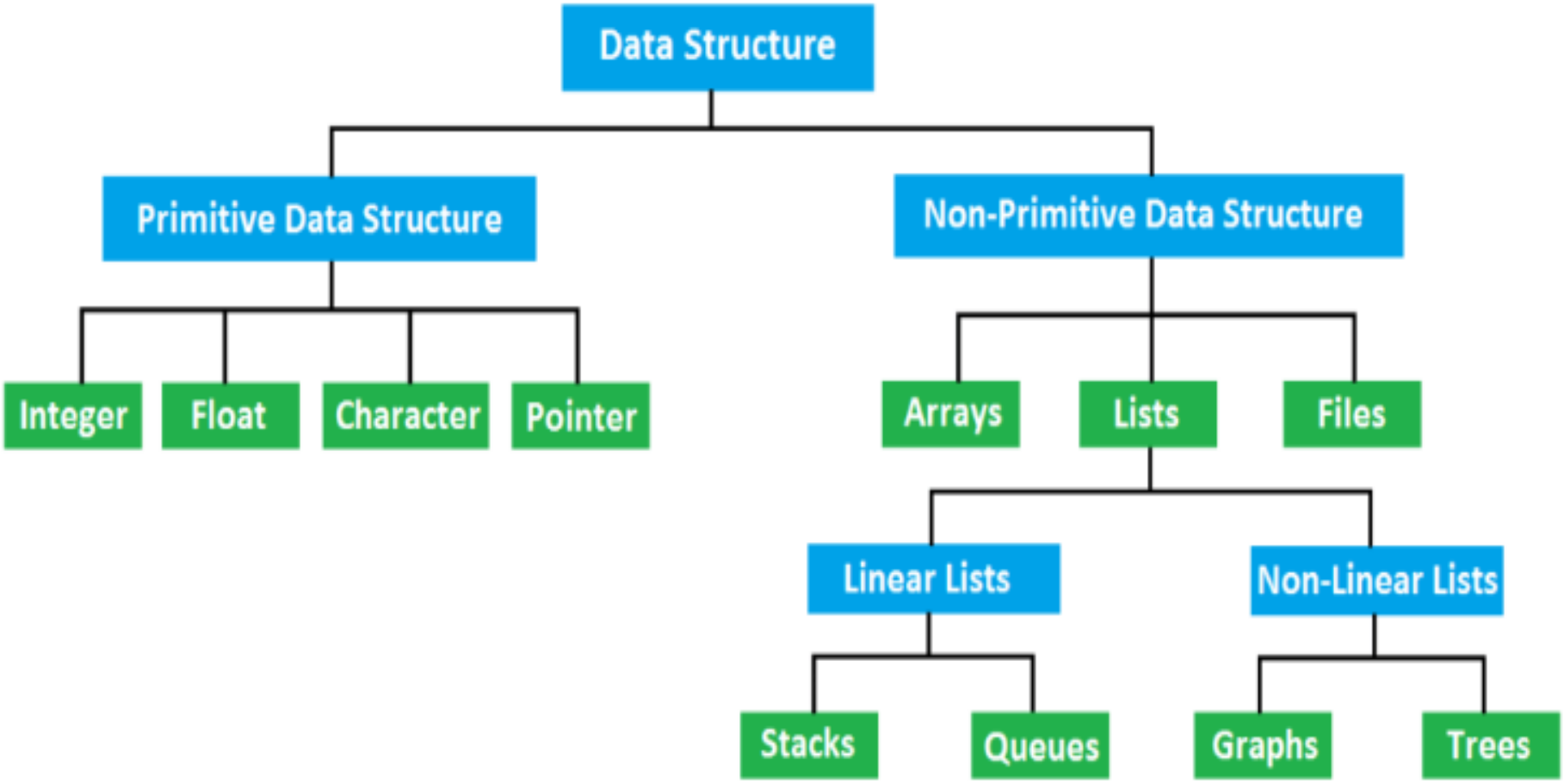
- Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types –
- Built-in Data Type
- Derived Data Type

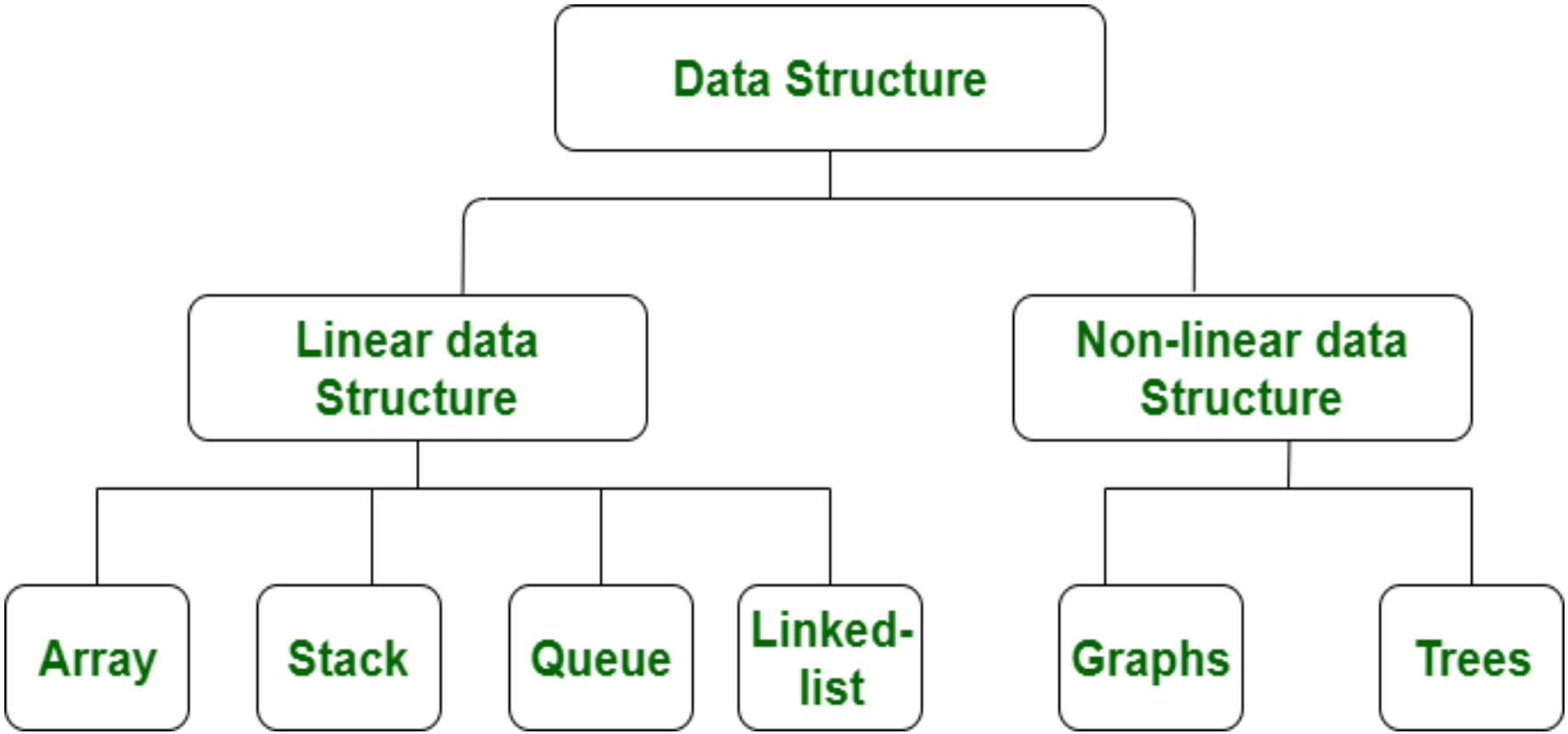
- Built-in Data Type
- Those data types for which a language has built-in support are known as Built-in Data types.
- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

- Derived Data Type
- Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types.
- These data types are normally built by the combination of primary or built-in data types and associated operations on them.
  - List
  - Array
  - Stack
  - Queue

Basic Operations

- The data in the data structures are processed by certain operations.
- The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.
  - Traversing
  - Searching
  - Insertion
  - Deletion
  - Sorting
  - Merging


Data Structure
Primitive Data Structure
Non-Primitive Data Structure
Integer
Float
Character
Pointer
Arrays
Lists
Files
Linear Lists
Non-Linear Lists
Stacks
Queues
Graphs
Trees


Data Structure
Linear data Structure
Non-linear data Structure
Array
Stack
Queue
Linked-list
Graphs
Trees

Linear Data Structure

- Data structure where data elements are arranged sequentially or linearly where the elements are attached to its previous and next adjacent in what is called a linear data structure.
- In linear data structure, single level is involved. Therefore, we can traverse all the elements in single run only.
- Linear data structures are easy to implement because computer memory is arranged in a linear way.
- Its examples are array, stack, queue, linked list, etc.

Non-linear Data Structure

- Data structures where data elements are not arranged sequentially or linearly are called non-linear data structures.
- In a non-linear data structure, single level is not involved. Therefore, we can’t traverse all the elements in single run only.
- Non-linear data structures are not easy to implement in comparison to linear data structure.
- It utilizes computer memory efficiently in comparison to a linear data structure. Its examples are trees and graphs.

S.NO	Linear Data Structure	Non-linear Data Structure
1.	In a linear data structure, data elements are arranged in a linear order where each and every elements are attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchically manner.
2.	In linear data structure, single level is involved.	Whereas in non-linear data structure, multiple levels are involved.
3.	Its implementation is easy in comparison to non-linear data structure.	While its implementation is complex in comparison to linear data structure.
4.	In linear data structure, data elements can be traversed in a single run only.	While in non-linear data structure, data elements can’t be traversed in a single run only.
5.	In a linear data structure, memory is not utilized in an efficient way.	While in a non-linear data structure, memory is utilized in an efficient way.
6.	Its examples are: array, stack, queue, linked list, etc.	While its examples are: trees and graphs.
7.	Applications of linear data structures are mainly in application software development.	Applications of non-linear data structures are in Artificial Intelligence and image processing.

Description of various Data Structures : Arrays

- An array is defined as a set of finite number of homogeneous elements or same data items.
- It means an array can contain one type of data only, either all integer, all float-point number or all character.

One dimensional array:

- ***An array with only one row or column is called one-dimensional array.***
- It is finite collection of n number of elements of same type such that:
  - can be referred by indexing.
  - The syntax Elements are stored in continuous locations.
  - Elements x to define one-dimensional array is:
- **Syntax: Datatype Array_Name [Size];**
- Where,

  Datatype : Type of value it can store (Example: int, char, float)

  Array_Name: To identify the array.
- Size : The maximum number of elements that the array can hold.

Arrays

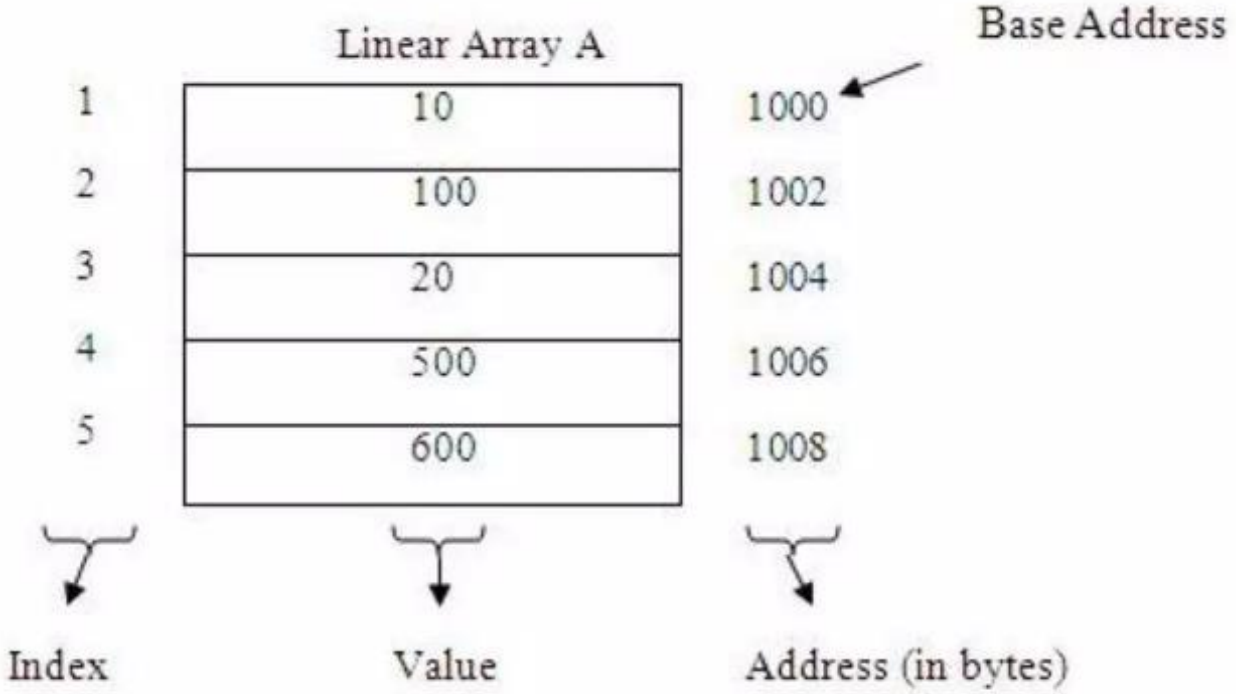
- Simply, declaration of array is as follows:

  int arr[10]

- Where int specifies the data type or type of elements arrays stores.
- “arr” is the name of array & the number specified inside the square brackets is the number of elements an array can store, this is also called sized or length of array.

Represent a Linear Array in memory

- The elements of linear array are stored in consecutive memory locations. It is shown below:



Arrays

- The elements of array will always be stored in the consecutive (continues) memory location.
- The number of elements that can be stored in an array, that is the size of array or its length is given by the following equation:

  (Upperbound-lowerbound)+1
- For the above array it would be (9-0)+1=10,where 0 is the lower bound of array and 9 is the upper bound of array.
- Array can always be read or written through loop.

```
For(i=0;i<=9;i++)
{   scanf("%d",&arr[i]);
   printf("%d",arr[i]); }
```

Arrays types

- Single Dimension Array
  - Array with one subscript
- Two Dimension Array
  - Array with two subscripts (Rows and Column)
- Multi Dimension Array
  - Array with Multiple subscripts

Basic operations of Arrays

- Some common operation performed on array are:
  - Traversing
  - Searching
  - Insertion
  - Deletion
  - Sorting
  - Merging

Traversing Arrays

- **Traversing:** It is used to access each data item exactly once so that it can be processed.
  E.g.
  We have linear array A as below:
- 1 2 3 4 5
- 10 20 30 40 50

Here we will start from beginning and will go till last element and during this process we will access value of each element exactly once as below:

A [1] = 10
A [2] = 20
A [3] = 30
A [4] = 40
A [5] = 50

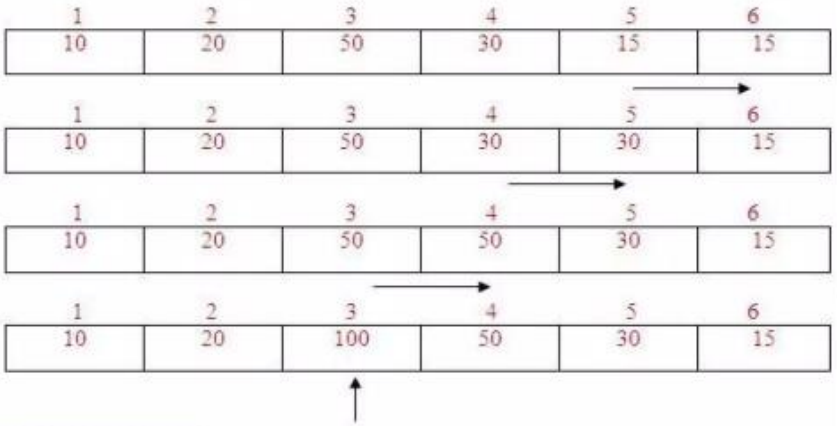
```
ALGORITHM: Traversal (A, LB, UB) A is an array with Lower Bound LB and
Upper Bound UB.
        Step 1:     for LOC = LB to UB do
        Step 2:         PROCESS A [LOC]
                    [End of for loop]
        Step 3:     Exit
```

Insertion into Array

- **Insertion:** It is used to add a new data item in the given collection of data items.
  E.g. We have linear array A as below:

1	2	3	4	5
10	20	50	30	15

New element to be inserted is 100 and location for insertion is 3. So shift the elements from 5th location to 3rd location downwards by 1 place. And then insert 100 at 3rd location. It is shown below:



ALGORITHM: Insert (A, N, ITEM, Pos) A is an array with N elements. ITEM is the element to be inserted in the position Pos.

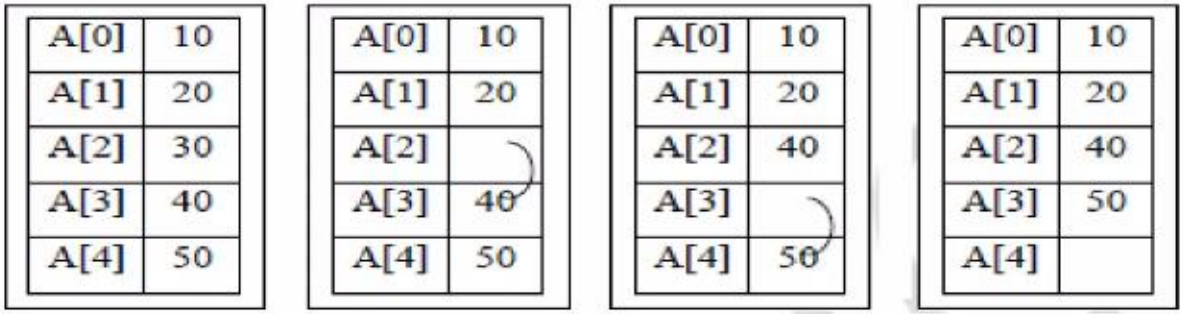
```
Step 1:   for I = N-1 down to Pos
              A[ I + 1] = A[I]
          [End of for loop]
Step 2:   A [Pos] = ITEM
Step 3:   N = N+1
Step 4:   Exit
```

Deletion from Array

- **Deletion:** It is used to delete an existing data item from the given collection of data items.

For example: Let A[4] be an array with items 10, 20, 30, 40, 50 stored at consecutive locations. Suppose item 30 has to be deleted at position 2. The following procedure is applied.

- Copy 30 to ITEM, i.e. Item = 30.
- Move Number 40 to the position 2.
- Move Number 50 to the position 3.



ALGORITHM: Delete (A, N, ITEM, Pos) A is an array with N elements. ITEM is the element to be deleted in the position Pos and it is stored into variable Item.

```
Step 1:   ITEM = A [Pos]
Step 2:   for I = Pos down to N-1
              A[ I ] = A[I+1]
          [End of for loop]
Step 3:   N = N-1
Step 4:   Exit
```

Searching in Arrays

- **Searching:** It is used to find out the location of the data item if it exists in the given collection of data items.

E.g. We have linear array A as below:

1	2	3	4	5
15	50	35	20	25

Suppose item to be searched is 20. We will start from beginning and will compare 20 with each element. This process will continue until element is found or array is finished. Here:

1) Compare 20 with 15
   20 # 15, go to next element.
2) Compare 20 with 50
   20 # 50, go to next element.

3) Compare 20 with 35
   20 #35, go to next element.

4) Compare 20 with 20
   20 = 20, so 20 is found and its location is 4.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Merging from Array

- **Merging:** It is used to combine the data items of two sorted files into single file in the sorted form
  We have sorted linear array A as below:

1	2	3	4	5	6
10	40	50	80	95	100

  And sorted linear array B as below:

1	2	3	4
20	35	45	90

  After merging merged array C is as below:

1	2	3	4	5	6	7	8	9	10
10	20	35	40	45	50	80	90	95	100

Two dimensional array

- ***A two dimensional array is a collection of elements and each element is identified by a pair of subscripts. (A[3] [3])***
- The elements are stored in continuous memory locations.
- The elements of two-dimensional array as rows and columns.
- The number of rows and columns in a matrix is called as the order of the matrix and denoted as mxn.
- The number of elements can be obtained by multiplying number of rows and number of columns.

	A[0]	A[1]	A[2]
A[0]	10	20	30
A[1]	40	50	60
A[2]	70	80	90

Representation of Two Dimensional Array:

- A is the array of order m x n. To store m*n number of elements, we need m*n memory locations.
- The elements should be in contiguous memory locations.
- There are two methods:
  - Row-major method
  - Column-major method

Two Dimensional Array:

- Row-Major Method: All the first-row elements are stored in sequential memory locations and then all the second-row elements are stored and so on. Ex: A[Row][Col]
- Column-Major Method: All the first column elements are stored in sequential memory locations and then all the second-column elements are stored and so on. Ex: A [Col][Row]

1000	10	A[0][0]
1002	20	A[0][1]
1004	30	A[0][2]
1006	40	A[1][0]
1008	50	A[1][1]
1010	60	A[1][2]
1012	70	A[2][0]
1014	80	A[2][1]
1016	90	A[2][2]

Row-Major Method

1000	10	A[0][0]
1002	40	A[1][0]
1004	70	A[2][0]
1006	20	A[0][1]
1008	50	A[1][1]
1010	80	A[2][1]
1012	30	A[0][2]
1014	60	A[1][2]
1016	90	A[2][2]

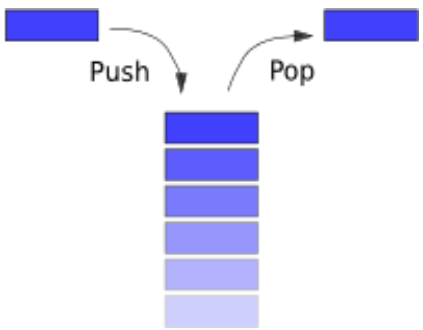
Col-Major Method

Applications on Array

- Array stores data elements of the same data type.
- Arrays can be used for CPU scheduling.
- Used to Implement other data structures like Stacks, Queues, Heaps, Hash tables, etc.

STACK

- "A *stack* is an ordered list in which all insertions and deletions are made at one end, called the *top".*
- Stacks are sometimes referred to as ***Last In First Out*** (LIFO) lists



- **<u>SIMPLE REPRESENTATION OF A STACK USING ARRAY</u>**
-
- Given a stack S = (a[1],a[2],.....a[n]) then we say that a1 is the bottom most element and element a[i]) is on top of element a[i-1], 1<i<=n.

- **The operations of stack is**

  - PUSH operations
  - POP operations
  - PEEK operations

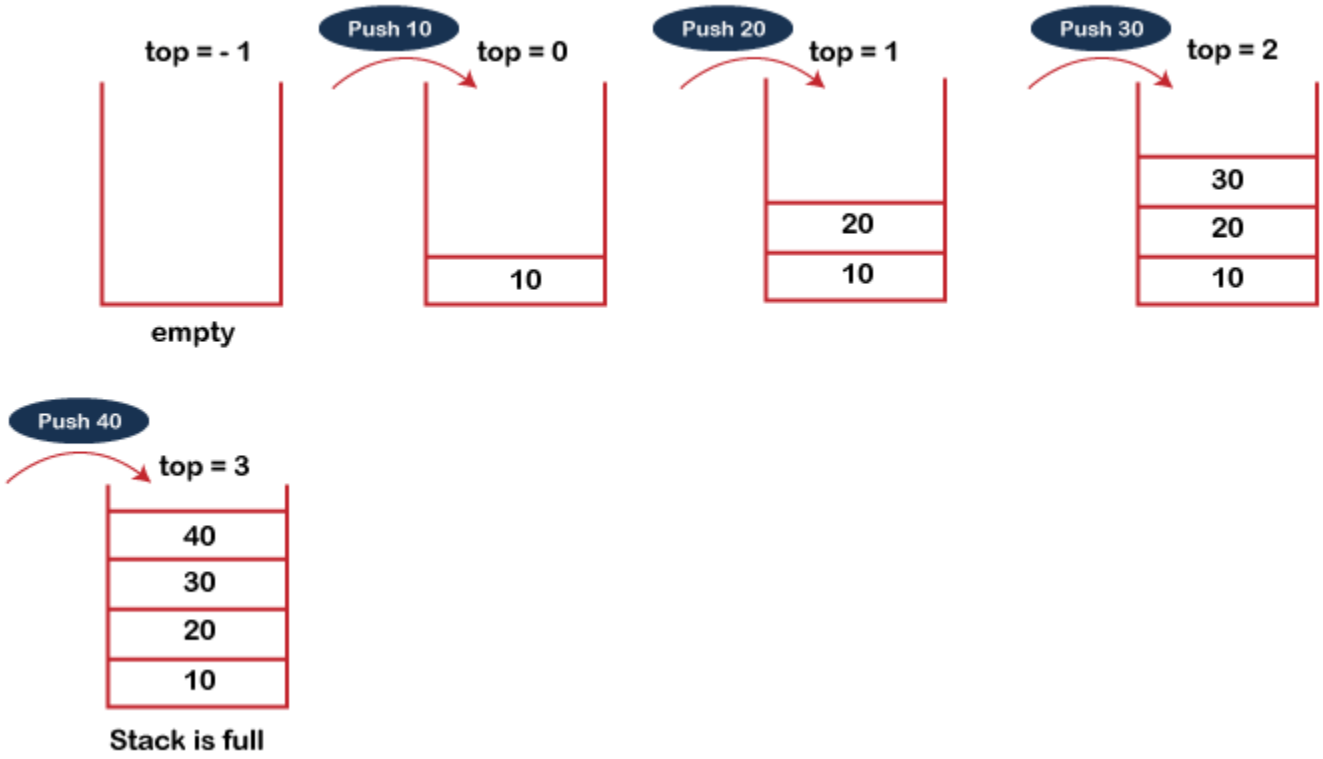
- **ABSTRACT DATA TYPE STACK**
-
- A stack S is an abstract data type (ADT) supporting the following three methods:
- **push(n) :** Inserts the item ***n*** at the top of stack
- **pop() :** Removes the top element from the stack and returns that top element. An error occurs if the stack is empty.
- **peek():** Returns the top element and an error occurs if the stack is empty.

- ***PUSH OPERATION***
- Push operation inserts an element onto the stack.
- Now we represented the stack by means of array.
- An attempt to push an element onto the stack, when the array is full, causes an *overflow*.
- Push operation involves:
- Check whether the array is full before attempting to push another element. If so halt execution.
- Increment the top pointer.
- Push the element onto the top of the stack.

- /* here, the variables stack, top and size are global variables */

```
void push (int item)
{
        if (top == size-1)
                printf("Stack is Overflow");
        else
        {
                top = top + 1;
                stack[top] = item;
        }
}
```

- **The steps involved in the PUSH operation is given below:**
- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the ***max*** size of the stack.

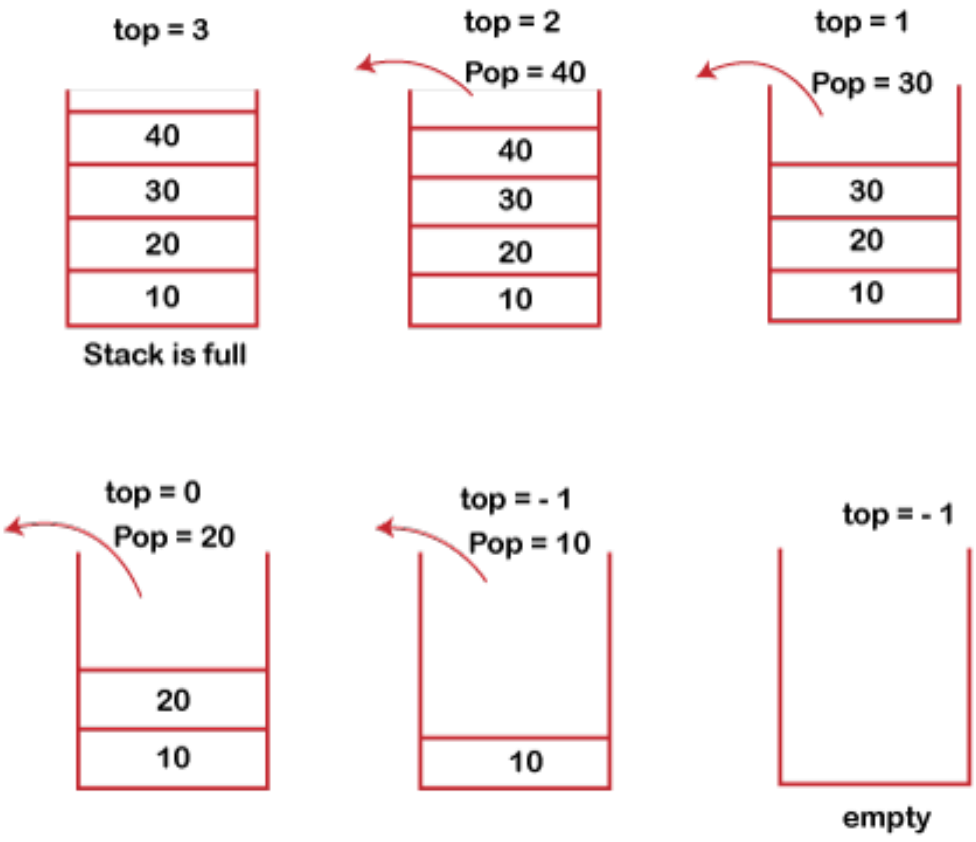

top = - 1
Push 10
top = 0
Push 20
top = 1
Push 30
top = 2
10
20
10
30
20
10
empty
Push 40
top = 3
40
30
20
10
Stack is full

- ***POP OPERATION***
- POP operation removes an element from the stack.
- An attempt to pop an element from the stack, when the array is empty, causes an *underflow*.
- Pop operation involves:
- Check whether the array is empty before attempting to pop another element. If so halt execution.
- Decrement the top pointer.
- Pop the element from the top of the stack.

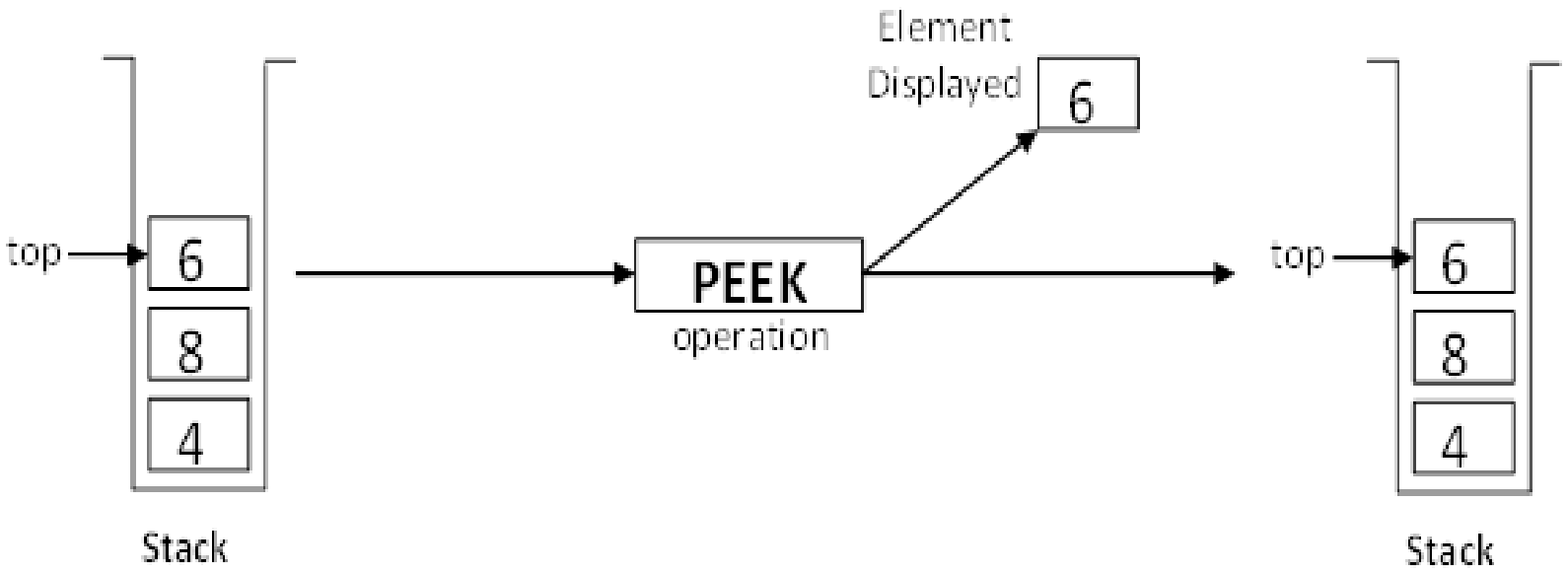
- /* here, the variables stack, and top are global variables */

```
 int pop ( )
{
        if (top == -1)
        {
                printf("Stack is Underflow");
                return (0);
        }
        else
        {
                return (stack[top--]);
        }
}
```

- **The steps involved in the POP operation is given below:**
- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the ***top***
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.


top = 3
40
30
20
10
Stack is full
top = 2
Pop = 40
40
30
20
10
top = 1
Pop = 30
30
20
10
top = 0
Pop = 20
20
10
top = - 1
Pop = 10
10
top = - 1
empty

- ***PEEK OPERATION***
- Returns the item at the top of the stack but does not delete it.
- If the stack is empty, which result in ***underflow***.


Element
Displayed
6
top
6
8
4
Stack
PEEK
operation
top
6
8
4
Stack

- /* here, the variables stack, and top are global variables */

```
 int pop ( )
{
        if (top == -1)
        {
                printf("Stack is Underflow");
                return (0);
        }
        else
        {
                return (stack[top]);
        }
}
```

- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.'
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

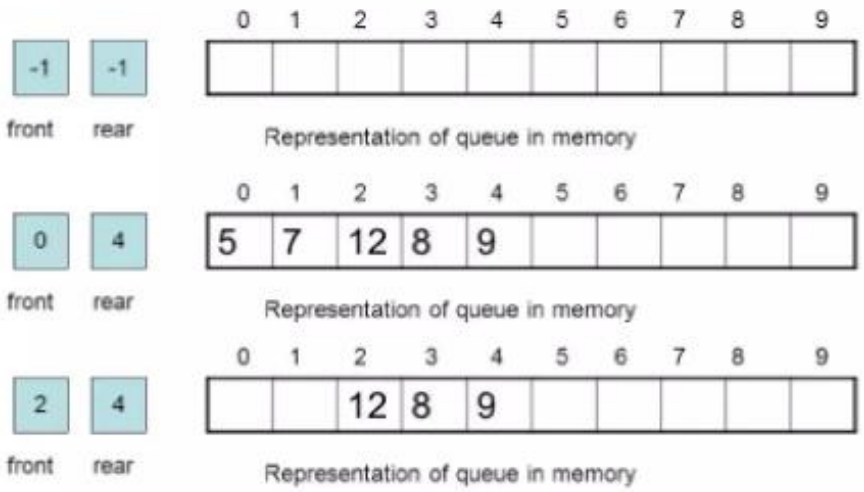
Queue

- *A queue is an ordered collection of items where an item is inserted at one end called the "rear" and an existing item is removed at the other end, called the "front".*
- Queue is also called as FIFO list i.e. First-In First-Out.
- In the queue only two operations are allowed enqueue and dequeue.
- Enqueue means to insert an item into back of the queue.
- Dequeue means removing the front item.The people standing in a railway reservation row are an example of queue.

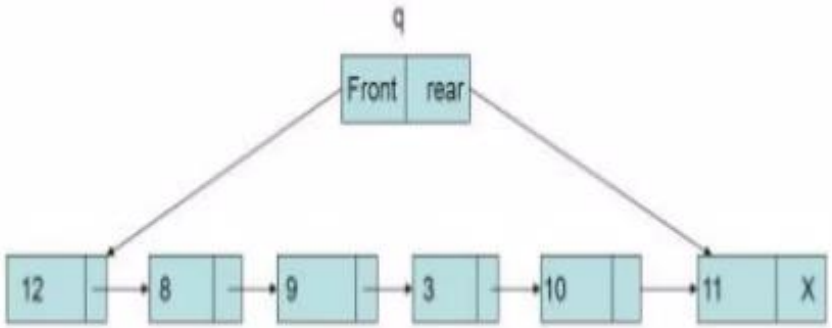
Queue

- The queue can be implemented into two ways:
  - Using arrays (Static implementation)
  - Using pointer (Dynamic implementation)

Array representation of linear queue



Representation of a queue in memory



Types of Queues

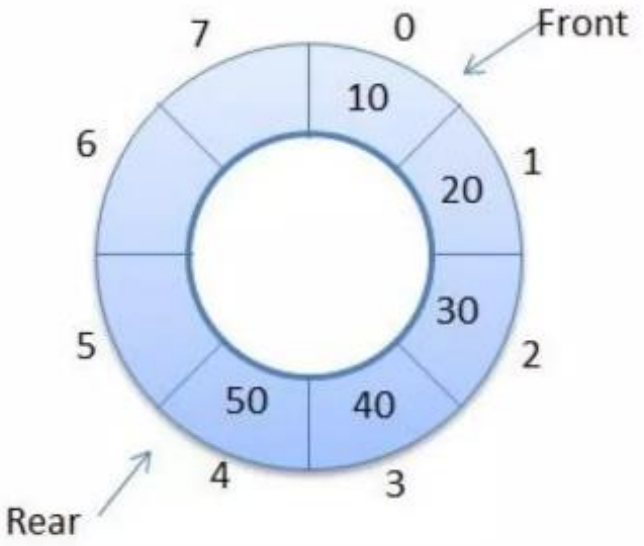
- Queue can be of four types:
  - Simple Queue
  - Circular Queue
  - Priority Queue
  - De-queue (Double Ended Queue)

Simple Queue

- Simple Queue: In simple queue insertion occurs at the rear end of the list and deletion occurs at the front end of the list.

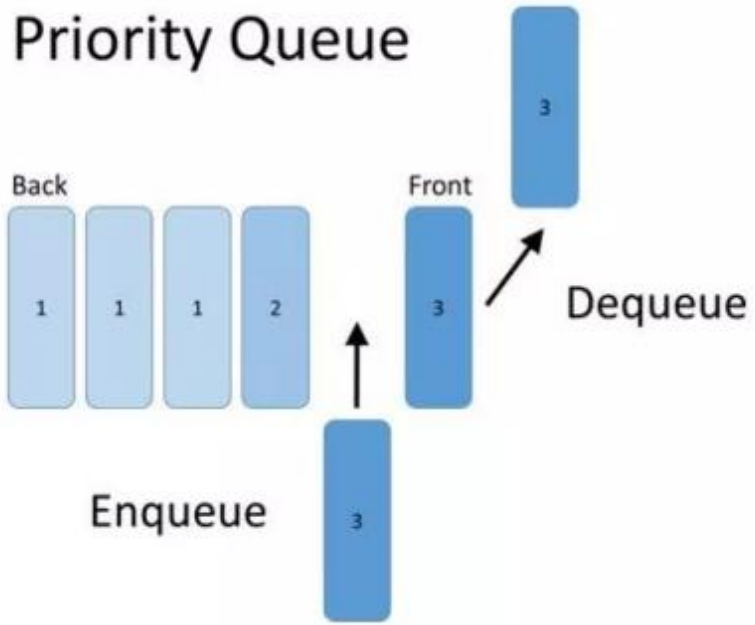
Circular Queue

- Circular Queue: A circular queue is a queue in which all nodes are treated as circular such that the last node follows the first node.



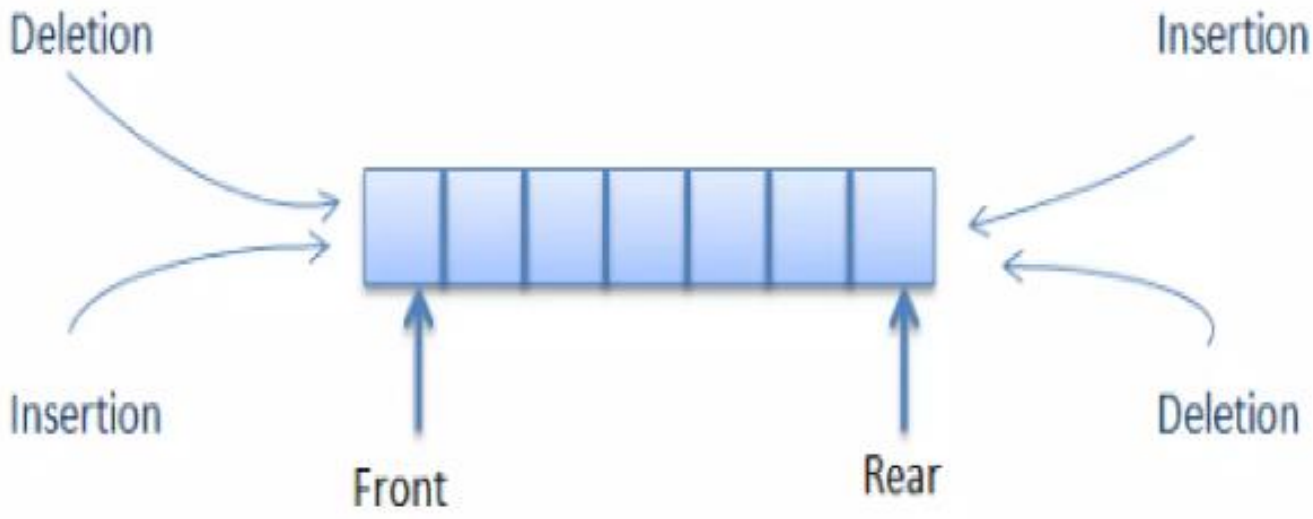
Priority Queue

- **A priority queue is a** queue that contains items that have some present priority. An element can be inserted or removed from any position depending upon some priority.



Dequeue Queue

- Dequeue: It is a queue in which insertion and deletion takes place at the both ends.



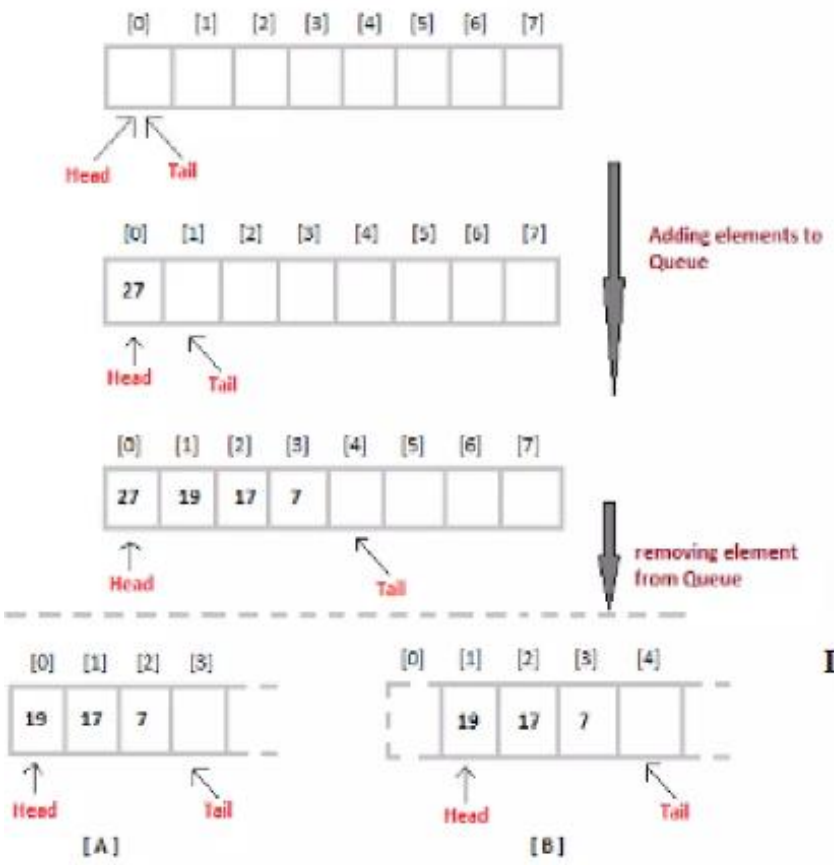
Operation on Queues

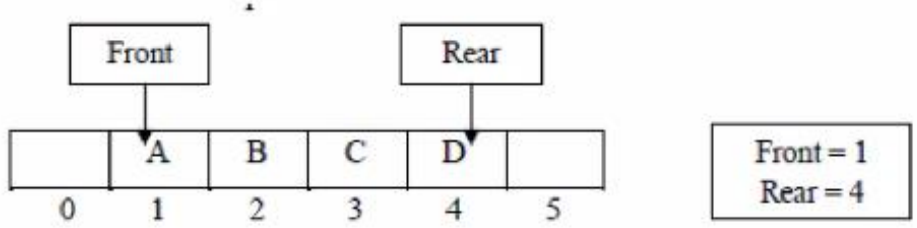
- **Queue( ):** It creates a new queue that is empty.
- **enqueue(item):** It adds a new item to the rear of the queue.
- **dequeue( ):** It removes the front item from the queue.
- **isEmpty( ):** It tests to see whether the queue is empty.
- **size( ):** It returns the number of items in the queue.

Memory Representation of a queue using array

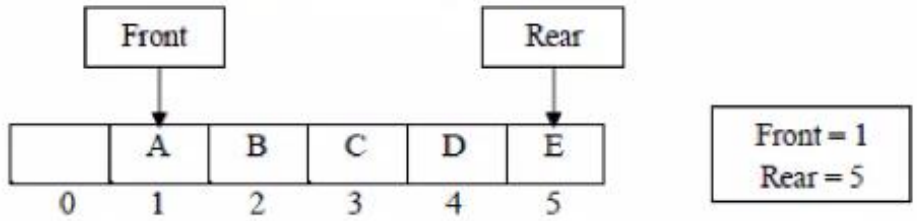
- Queue is represented in memory using linear array.
- Let QUEUE is a array, two pointer variables called FRONT and REAR are maintained.
- The pointer variable FRONT contains the location of the element to be removed or deleted.
- The pointer variable REAR contains location of the last element inserted.
- The condition FRONT = NULL indicates that queue is empty.
- The condition REAR = N-1 indicates that queue is full.

Memory Representation of a queue using array

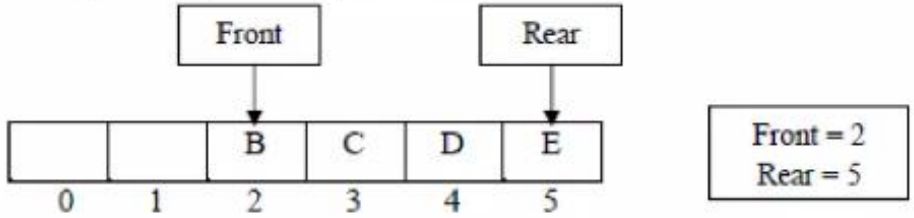




REAR = REAR + 1, QUEUE [REAR] = 'E'



ITEM = QUEUE [FRONT], FRONT = FRONT + 1



Queue Insertion Operation (ENQUEUE):

- **ALGORITHM: ENQUEUE (QUEUE, REAR, FRONT, ITEM)**

  QUEUE is the array with N elements. FRONT is the pointer that contains the location of the element to be deleted and REAR contains the location of the inserted element. ITEM is the element to be inserted.

Step 1: if REAR = N-1 then [Check Overflow]

PRINT "QUEUE is Full or Overflow"

Exit

[End if]

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
8	12	4	18	34

Step 2: if FRONT = NULL then [Check Whether Queue is empty]

FRONT = -1

REAR = -1

else

REAR = REAR + 1 [Increment REAR Pointer]

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
8	12			

Step 3: QUEUE[REAR] = ITEM [Copy ITEM to REAR position]

Step 4: Return

Queue Deletion Operation (DEQUEUE)

ALGORITHM: DEQUEUE (QUEUE, REAR, FRONT, ITEM)

QUEUE is the array with N elements. FRONT is the pointer that contains the location of the element to be deleted and REAR contains the location of the inserted element. ITEM is the element to be inserted.

Step 1: if FRONT = NULL then [Check Whether Queue is empty]

PRINT "QUEUE is Empty or Underflow"

Exit

[End if]

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]

Step 2: ITEM = QUEUE[FRONT]

Step 3: if FRONT = REAR then [if QUEUE has only one element]

FRONT = NULL

REAR = NULL

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
8				

else

FRONT = FRONT + 1 [Increment FRONT pointer]

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
8	12			

Step 4: Return

Application of Queue

- Simulation
- Various features of Operating system
- Multi-programming platform systems.
- Different types of scheduling algorithms
- Round robin technique algorithms
- Printer server routines
- Various application software's is also based on queue data structure.

ABSTRACT DATA TYPES(ADT)

In programming each program is breakdown into modules, so that no routine should ever exceed a page.

Each module is a logical unit and does specific job modules which in turn will call another module.

Modularity has several advantages

1. Modules can be compiled separately which makes debugging process easier.
2. Several modules can be implemented and executed simultaneously.
3. Modules can be easily enhanced.

ABSTRACT DATA TYPES(ADT)

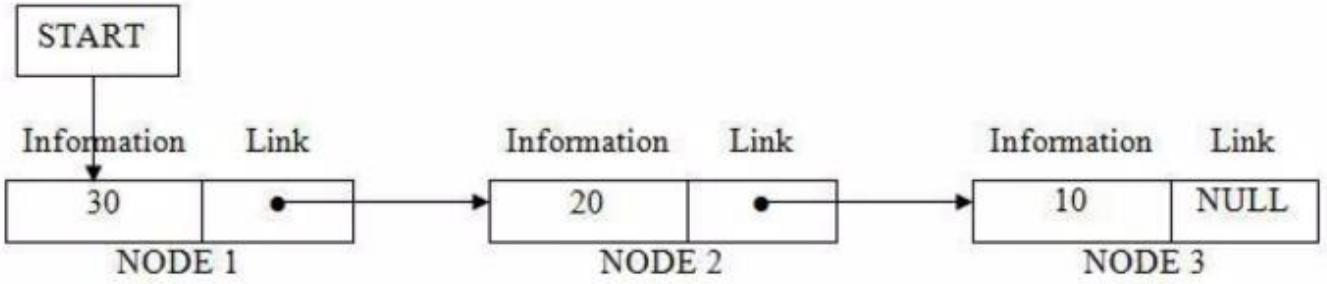
Abstract Data type is an extension of modular design.

An abstract data type is a set of operations such as Union, Intersection, Complement, Find etc.,

The basic idea of implementing ADT is that the operations are written once in program and can be called by any part of the program.

Lists

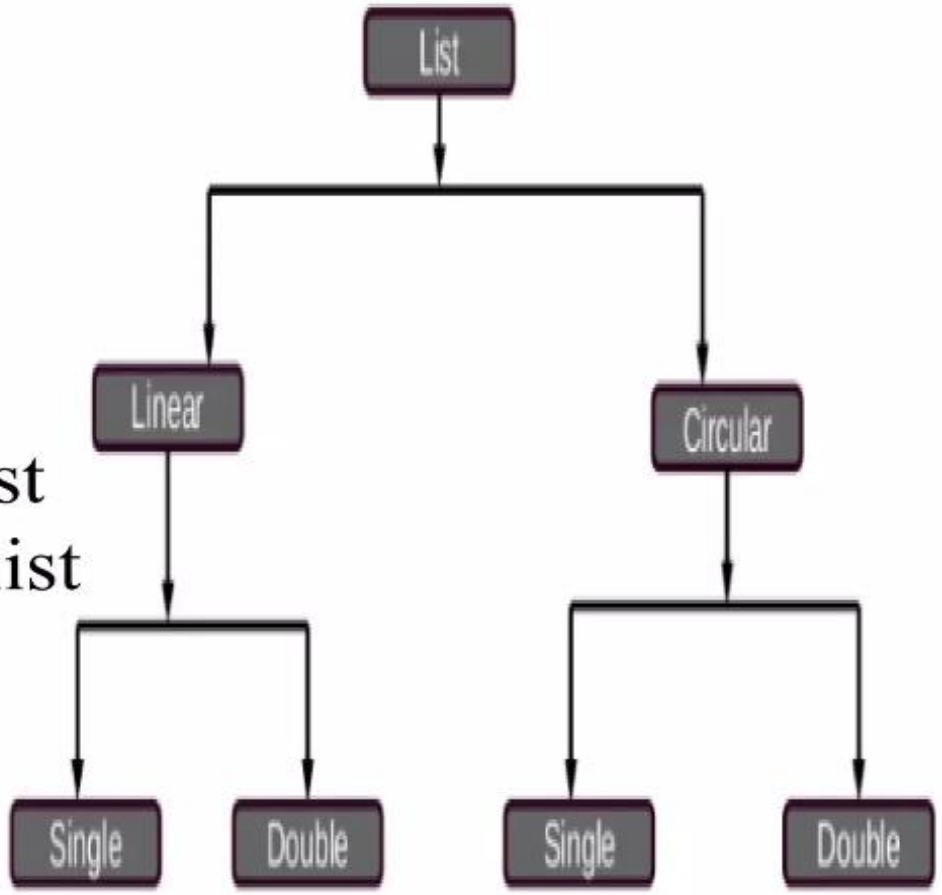
- A lists (Linear linked list) can be defined as a collection of variable number of data items called ***nodes***.
- Lists are the most commonly used non-primitive data structures.
- Each nodes is divided into two parts:
  - The first part contains the information of the element.
  - o The second part contains the memory address of the next node in the list. Also called Link part.



Lists

Types of linked lists:

- Single linked list
- Doubly linked list
- Single circular linked list
- Doubly circular linked list



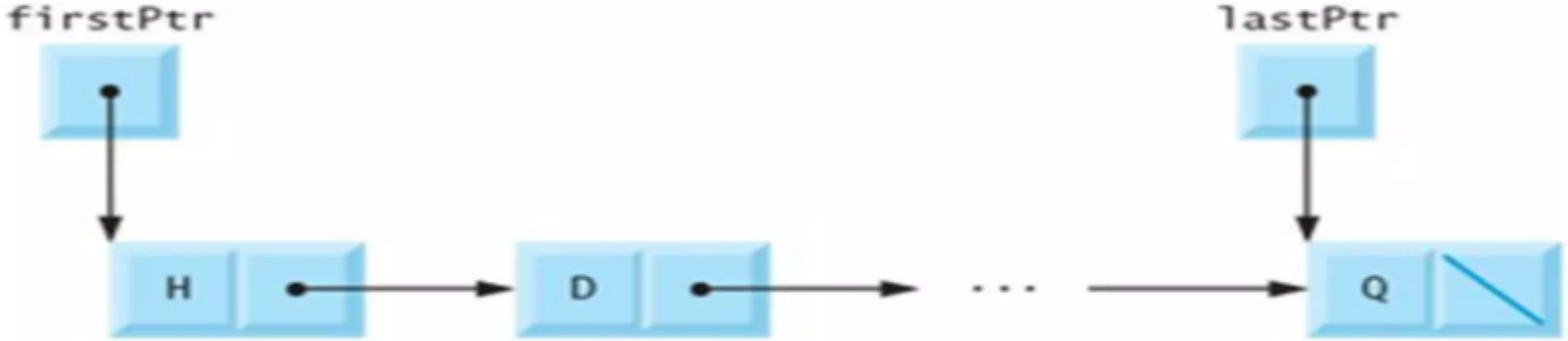
Single linked list

A singly linked list contains two fields in each node - an information field and the linked field.

•The ***information*** field contains the data of that node.

•The ***link*** field contains the memory address of the next node.

There is only one link field in each node, the linked list is called singly linked list.

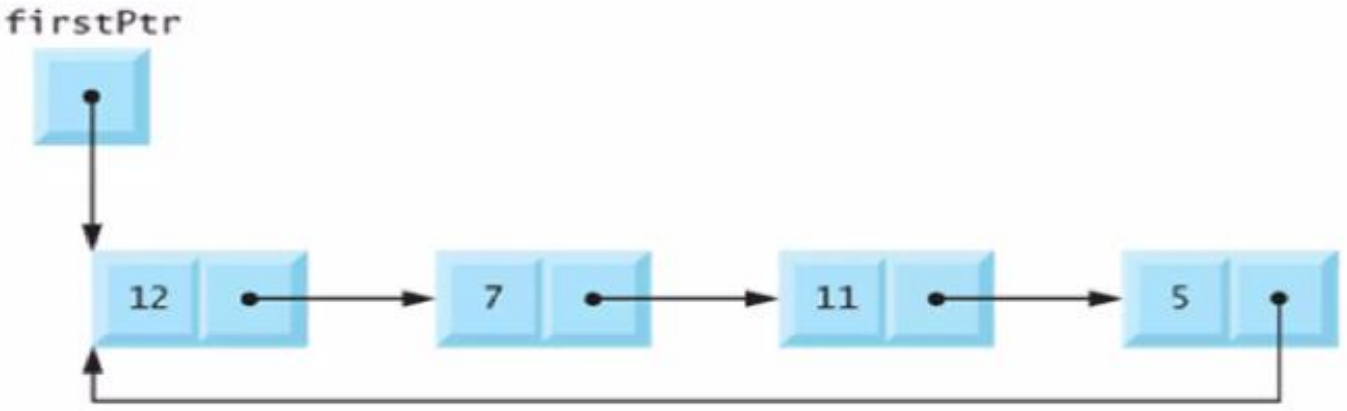


A graphical representation of a list.

Single circular linked list

The link field of the last node contains the memory address of the first node, such a linked list is called circular linked list.

· In a circular linked list every node is accessible from a given node.

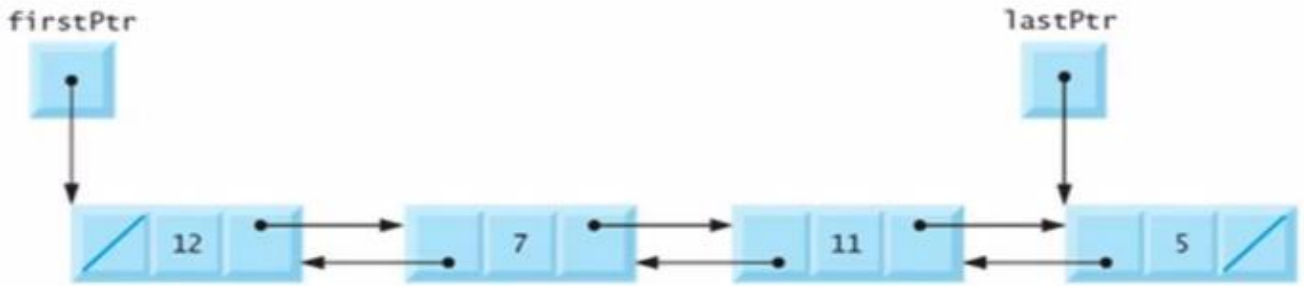


Circular, singly linked list.

Doubly linked list

It is a linked list in which each node is points both to the next node and also to the previous node.

- In doubly linked list each node contains three parts:
  - FORW : It is a pointer field that contains the address of the next node
  - BACK: It is a pointer field that contains the address of the previous node.
  - INFO: It contains the actual data.
- In the first node, if BACK contains NULL, it indicated that it is the first node in the list.
- The node in which FORW contains, NULL indicates that the node is the last node.



Doubly linked list.

Doubly circular linked list

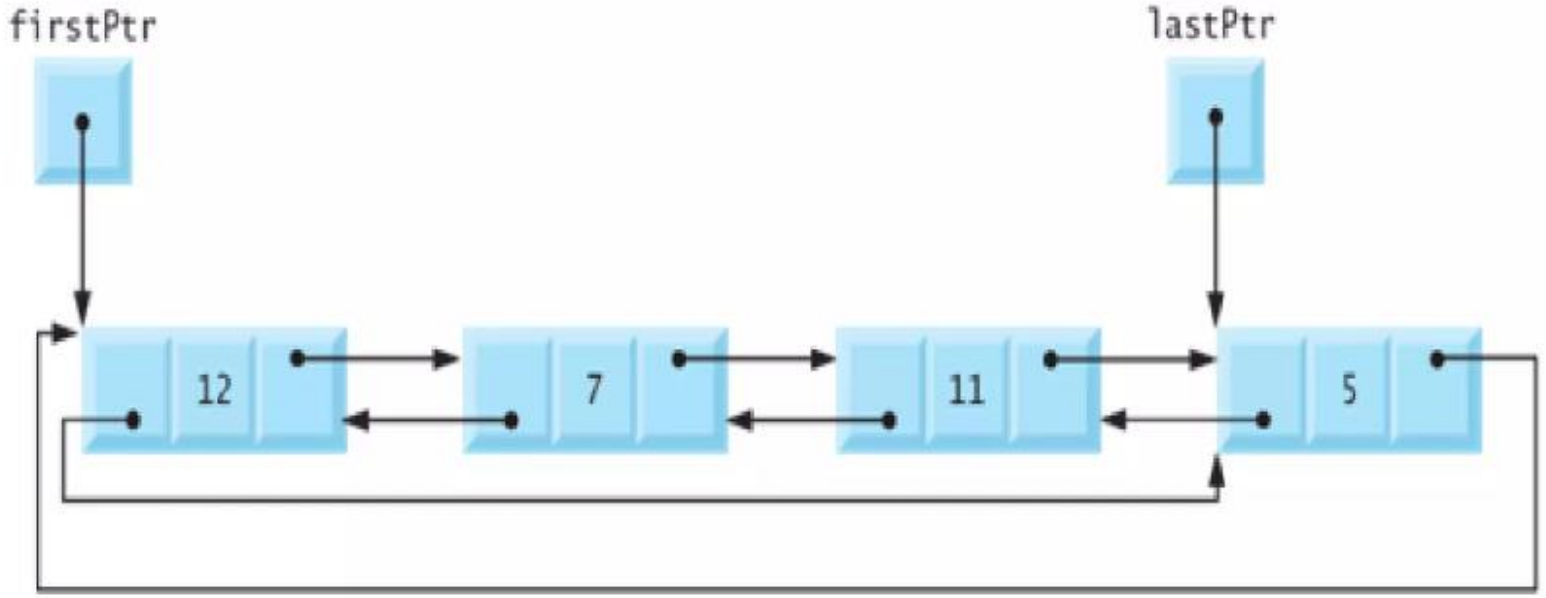


Fig. 19.12 | Circular, doubly linked list.

Operation on Linked List

- The operation that are performed on linked lists are:
  - Creating a linked list
  - Traversing a linked list
  - Inserting an item into a linked list.
  - Deleting an item from the linked list.
  - Searching an item in the linked list
  - Merging two or more linked lists.

Linked list node creation

```
struct Node {
    int data;
    struct Node* next;
} *head;
```

Explanation:

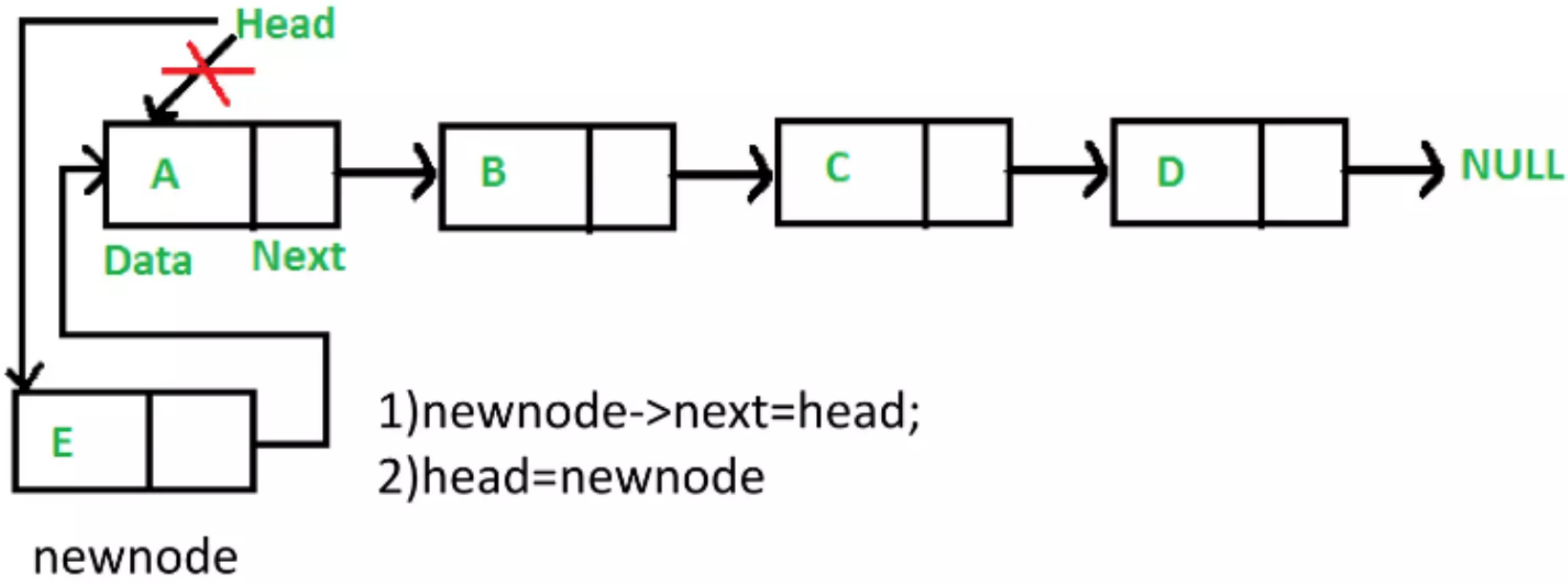
declared structure of type **“NODE”**,

First field stores **actual data and another field stores address**

Insertion

- insertion operation can be performed in three ways
  1. Inserting At Beginning of the list
  2. Inserting At End of the list
  3. Inserting At Specific position in the list

Insert at Beginning of the list



Inserting At Beginning of the list

steps to insert a new node at beginning of the single linked list

Step 1 - Create a **newnode** with given value.

Step 2 - Check whether list is **Empty** (**head** == **NULL**)

Step 3 - If it is **Empty** then, set **newnode→next** = **NULL** and **head** = **new Node**.

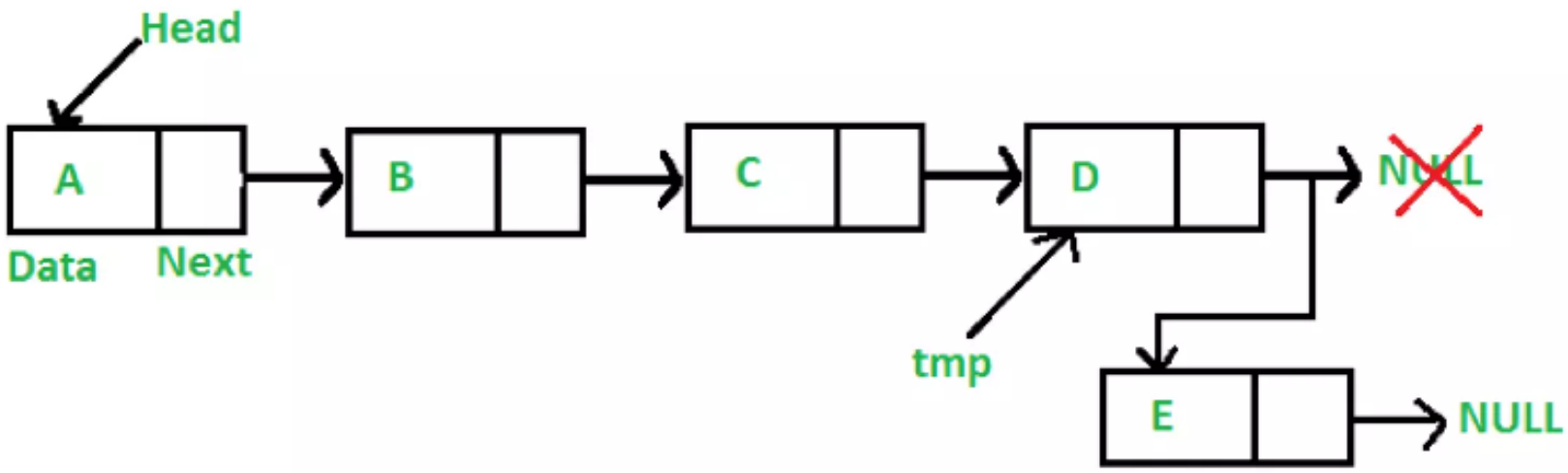
Step 4 - If it is **Not Empty** then, set **newnode→next** = **head** and **head** = **newnode**

```
void insertAtBeginning(int value)
{
struct Node *newnode;
newnode = (struct Node*)malloc(sizeof(struct Node));
newnode->data = value;
 if(head == NULL)
 {
 newnode->next = NULL;
 head = newnode;
 }
else
 {
newnode->next = head;
head = newnode;
}
```

Inserting At Beginning of the list

```
void insertAtBeginning(int value)
{
struct Node *newnode;
newnode = (struct Node*)malloc(sizeof(struct Node));
newnode->data = value;
newnode->next = head;
head = newnode;
}
```

Insert at End of the list



1) Traverse –Now temp points to last node
2) temp ->next=newnode;

Insert at End of the list

steps to insert a new node at end of the single linked list

Step 1 - Create a **newnode** with given value and **newnode → next** as **NULL**.

Step 2 - Check whether list is **Empty** (**head** == **NULL**).

Step 3 - If it is **Empty** then, set **head** = **newnode**.

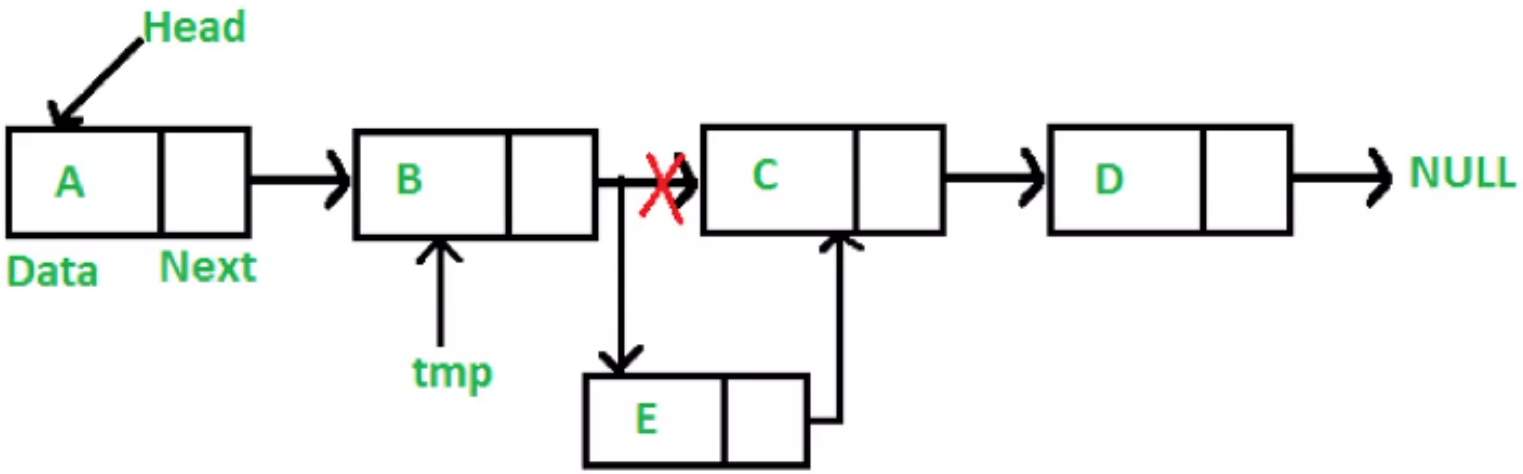
Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5 - Keep moving the **temp** to its next node until it to the last node in the list (until **temp → next** is equal to **NULL**).

Step 6 - Set **temp → next** = **newnode**.

```
void insertAtEnd(int value)
{
struct Node *newnode;
newnode = (struct Node*)malloc(sizeof(struct Node));
newnode->data = value;
newnode->next = NULL;
if(head == NULL)
        head = newnode;
else
{
        struct Node *temp = head;
        while(temp->next != NULL)
                temp = temp->next;
        temp->next = newnode;
 }
 }
```

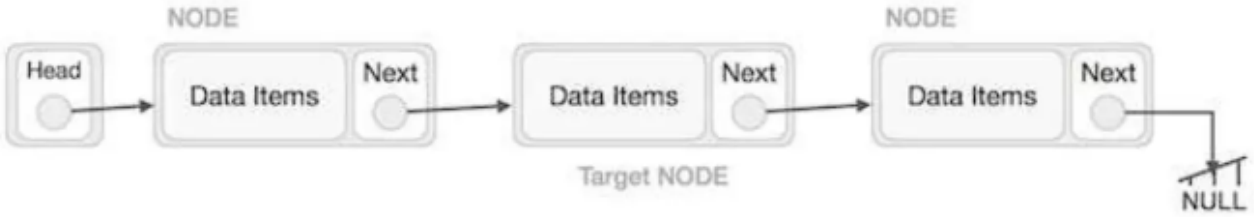
Insert at a given position



```
void insertatmiddle(int value, int pos)
{
struct Node *newNode,*temp;
Int i,pos;
newnode = (struct Node*)malloc(sizeof(struct Node));
newnode->data = value;
temp=head;
for(i=1;i<pos-1;i++)
{
     temp=temp->next;
}
If(temp==head)
{
    newnode->next=head;
    head=newnode;
}
else
{
   newnode->next=temp->next;
   temp->next=newnode;
}
```

Deletion Operation

- locate the target node to be removed



- left (previous) node of the target node now should point to the next node of the target node
- LeftNode.next –> TargetNode.next;

Delete at begin

```
Void deleteatbegin()
{
   struct node *temp;
   temp=head;
   printf("%d is deleted", temp->data);
   head=temp->next;
   free(temp);
}
```

Delete at last

```
void deleteatlast()
 {
  struct node *last, *secondlast;
  last=head;
  secondlast=head;
  while(last->next!=NULL)
  {
          secondlast= last;
          last=last->next;
  }
  If(last==head)
  {
  head=NULL;
  }
  else
  {
   secondlast->next=NULL;
   free(last);
    }
```

Displaying the linked list

```
Void display()
{
    struct node *temp;
   temp=head;
   while(temp!=NULL)
   {
        printf("%d", temp->data);
        temp=temp->next;
   }
}
```

Searching in a LL

```
void search()
{
   int key;
   printf("enter the element to search");
   scanf("%d",&key);
     temp = head;
   // Iterate till last element until key is not found
   while (temp != NULL && temp->data != key)
   {
            temp = temp->next;
   }
   if(temp->data==key)
   printf("element  found");
   else
   printf("element not found");
}
```

DOUBLY LINKED LIST

Insertion at beginning of the LL

```
Void insertatbegin()
{
struct node *newnode;
newnode = (struct node*)malloc(sizeof(struct node));
printf("Enter the value");
scanf("%d", &newnode->data);
newnode->prev= NULL;
newnode->next = head;
head ->prev= newnode;
head=newnode;
 }
```

Insertion at end of the LL

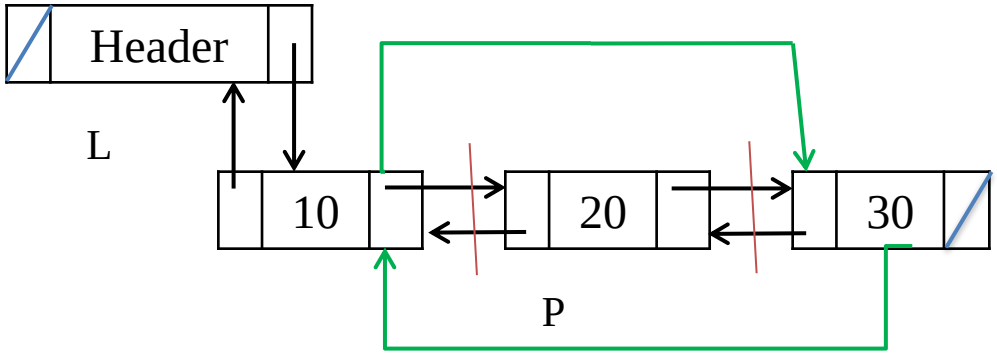
```
Void insertatend()
{
struct node *newnode;
If(head==NULL)
{
newnode = (struct node*)malloc(sizeof(struct node));
printf("Enter the value");
scanf("%d", &newnode->data);
newnode->prev= NULL;
newnode->next = NULL;
head=newnode;
current=head;
 }
```

Insertion at end of the LL(CONTD…)

```
else
{
newnode = (struct node*)malloc(sizeof(struct node));
printf(“Enter the value”);
scanf(“%d”, &newnode->data);
newnode->prev= current;
newnode->next = NULL;
Current->next=newnode;
current=newnode;
 }
```

ROUTINE TO DELETE AN ELEMENT

```
void Delete (int X, List L)
{
        position P;
        P = Find (X, L);
        If ( IsLast (P, L))
        {
                Temp = P;
                P →Blink →Flink = NULL;
                free (Temp);
        }
        else
        {
                Temp = P;
                P →Blink→ Flink = P→Flink;
                P →Flink →Blink = P→Blink;
                free (Temp);
        }
}
```



DOUBLY LINKED LIST

Advantage

* Deletion operation is easier.

* Finding the predecessor & Successor of a node is easier.

Disadvantage

* More Memory Space is required since it has two pointers.

Difference between Singly Linked List & Doubly Linked List

S.No	Singly Linked List	Doubly Linked List
1.	It is a collection of nodes and each node is having one data field and one next field.	It is a collection of nodes and each node is having one data field, one previous field and one next field.
2.	The element can be accessed using next link.	The element can be accessed using both previous and next link.
3.	No extra field is required. Hence node takes less memory space.	One extra field is required to store previous link hence node takes more memory space.

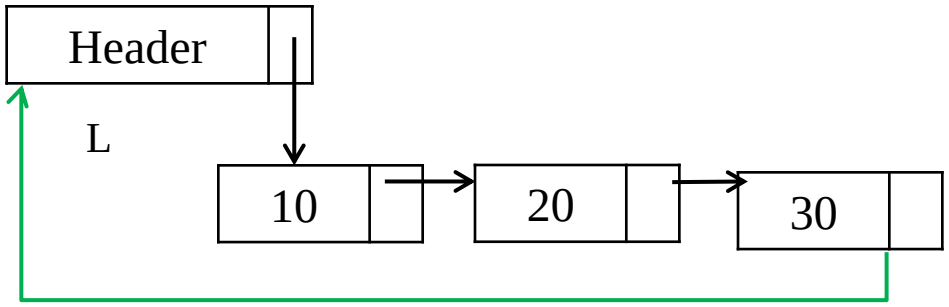
CIRCULAR LINKED LIST

In circular linked list the pointer of the last node points to the first node.

Circular linked list can be implemented as Singly linked list and Doubly linked list with or without headers.

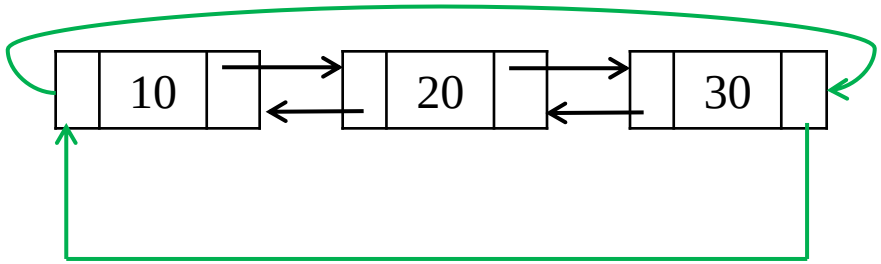
Singly Linked Circular List

A singly linked circular list is a linked list in which the last node of the list points to the first node.



DOUBLY LINKED CIRCULAR LIST

A doubly linked circular list is a Doubly linked list in which the forward link of the last node points to the first node and backward link of the first node points to the last node of the list.



DOUBLY LINKED CIRCULAR LIST

Advantages of Circular Linked List

- It allows traversing the list starting at any point.
- It allows quick access to the first and last records
- Circularly doubly linked list allows to traverse the list in wither direction

APPLICATIONS OF LINKED LIST

1. Polynomial ADT
2. Radix Sort
3. Multilist

Applications of linked list

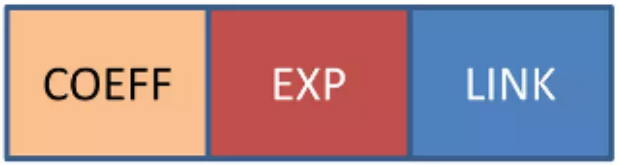
Polynomial

Dynamic memory allocation

Polynomial manipulations

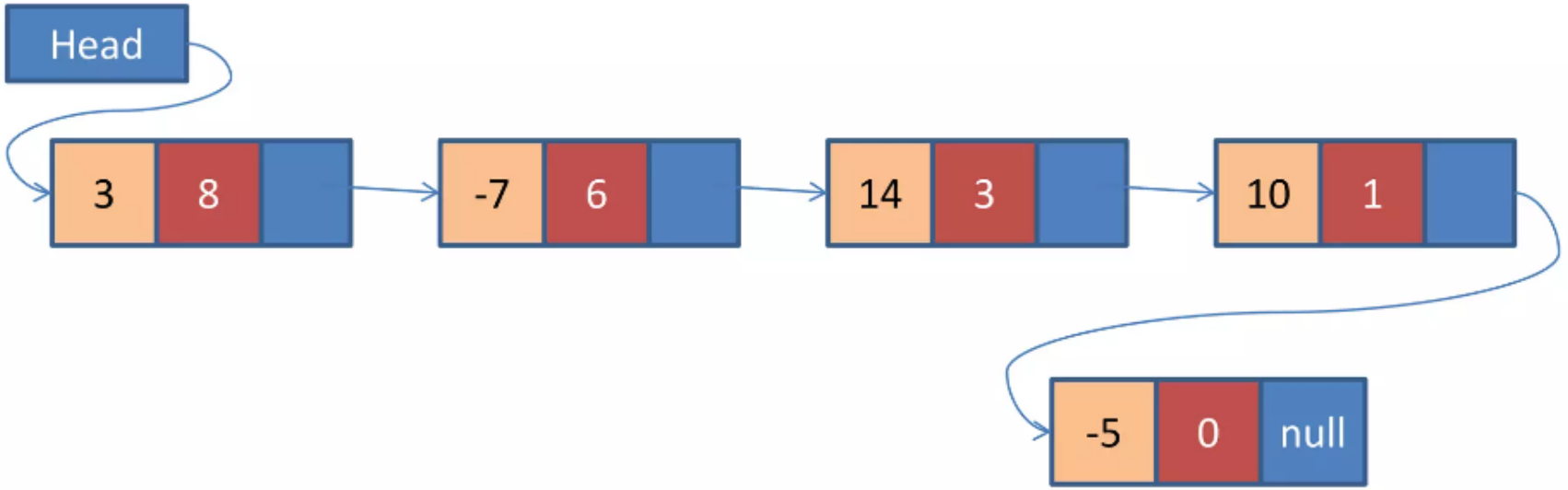
- An important application of linked list is to represent polynomials and their manipulations
- The main advantage is that it can accommodate no. of polynomials of growing sizes, so that their combined size does not exceed the total memory available.
- Polynomial representation
  - Consider the general form of a polynomial with single variable

    $P(x)= a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^n$
  - The structure of the node to represent a term in polynomial
  - 3 fields



Polynomial manipulations

- No. of nodes to represent a polynomial is the same as the no. of terms in the polynomial.
- An additional header node is also used.
- Eg: $P(x)= 3x^8 - 7x^6 + 14x^3 + 10x - 5$



Different operations

- Addition & Multiplication of 2 polynomials

1. Polynomial addition

- Consider two polynomials P&Q
- R=P+Q
- Use 2 pointers Pptr & Qptr
- 3 cases

Case 1: Exponents of two terms are equal

- Here, coefficients in the two nodes are added to form the new term.
- Rptr. COEFF = Pptr. COEFF + Qptr. COEFF
- Rptr. EXP= Pptr.EXP

Polynomial addition

Case 2: Pptr. EXP > Qptr. EXP

- i.e. exponent of the current term in P is greater than the exponent of the current term in Q
- Here a duplicate of the current term in P is created and inserted in polynomial R

Case 3: Pptr. EXP < Qptr. EXP

- i.e. exponent of the current term in P is less than the exponent of the current term in Q
- Here a duplicate of the current term in Q is created and inserted in polynomial R

Polynomial addition

Algorithm: POLYNOMIAL_ADD(Phead, Qhead, Rhead)

1. Set Pptr= Phead, Qptr= Qhead
2. Rhead= null
3. Rptr= Rhead
4. Repeat while (Pptr!=NULL) and (Qptr!= NULL)
5. if(Pptr->EXP=Qptr.EXP) **//case1**
6. temp=GetNode(Node) //create a new node
7. Rptr=temp Rptr->link= temp
8. Rptr->COEFF=Pptr->COEFF + Qptr->COEFF
9. Rptr->EXP= Pptr->EXP
10. Rptr->link=null
11. Pptr=Pptr->link, Qptr=Qptr->link

Polynomial addition

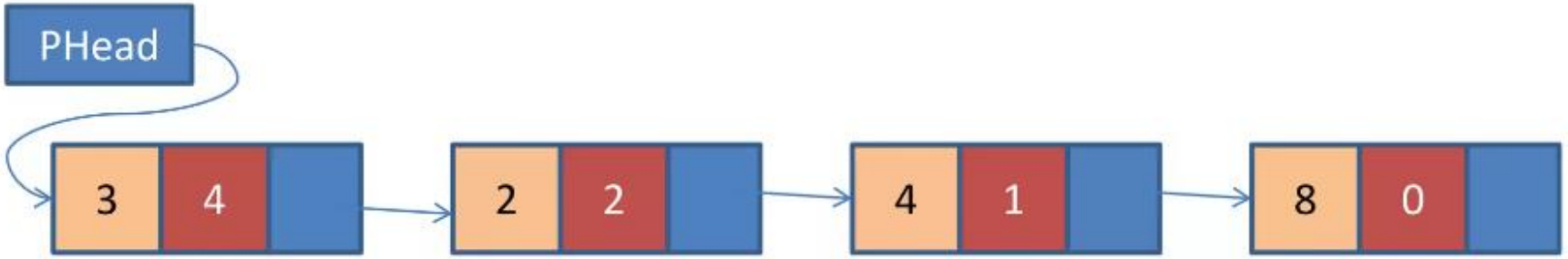
```
12.      if( Pptr->EXP > Qptr->EXP )                //case2
13               temp=GetNode(Node)          //create a new node
14.              Rptr->link= temp, Rptr=temp
15.              Rptr->COEFF=Pptr->COEFF
16.              Rptr-> EXP= Pptr->EXP
17.              Rptr->linkl=null
18.              Pptr=Pptr->link

19.      if( Pptr->EXP < Qptr->EXP )                //case3
20               temp=GetNode(Node) //create a new node
21.              Rptr->link= temp, Rptr=temp
22.              Rptr->COEFF=Qptr->COEFF
23               Rptr.->EXP= Qptr->EXP
24.              Rptr->link=null
25.              Qptr=Qptr.->link
     end while
```

Polynomial addition

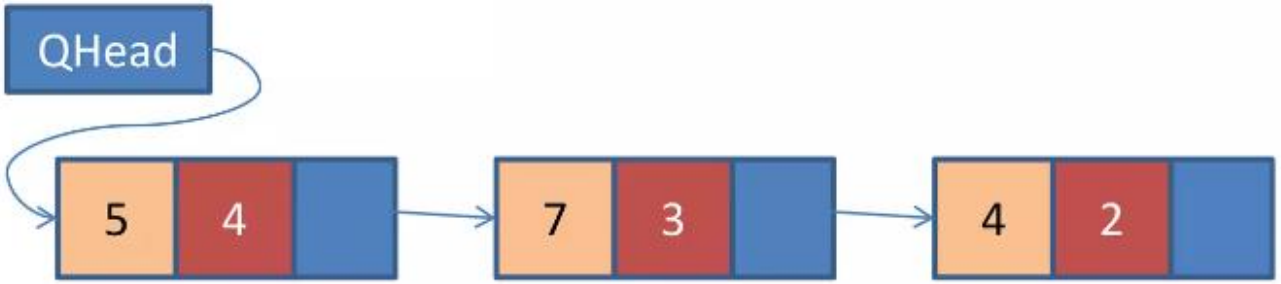
```
26.  while( Pptr != null)                  // To add remaining terms of P in to R
27                temp=GetNode(Node)       //create a new node
28.               Rptr->link= temp, Rptr=temp
29.               Rptr->COEFF=Pptr->COEFF
30.               Rptr.->EXP= Pptr->EXP
31.               Rptr->link=null
32.               Pptr=Pptr->link

33.  while(Qptr!= null)                    // To add remaining terms of Q in to R
34                temp=GetNode(Node)       //create a new node
35.               Rptr->link= temp, Rptr=temp
36.               Rptr->COEFF=Qptr->COEFF
37                Rptr-> EXP= Qptr->EXP
38.               Rptr->link=null
39.                Qptr=Qptr->link
40.    end while
```

Example

$P=3x^4 + 2x^2+ 4x+8$ Pptr=Phead Pptr=Pptr->link



$Q=5x^4 + 7x^3+4x^2$ Qptr=Qhead QPtr=Qptr->link



R=P+Q

$=8x^4+7x^3+6x^2+4x+8$