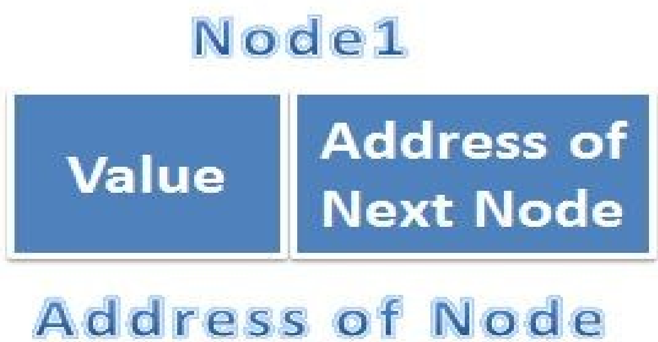
Linked List

Introduction to Linked List

- It is a data Structure which consists if group of nodes that forms a sequence.

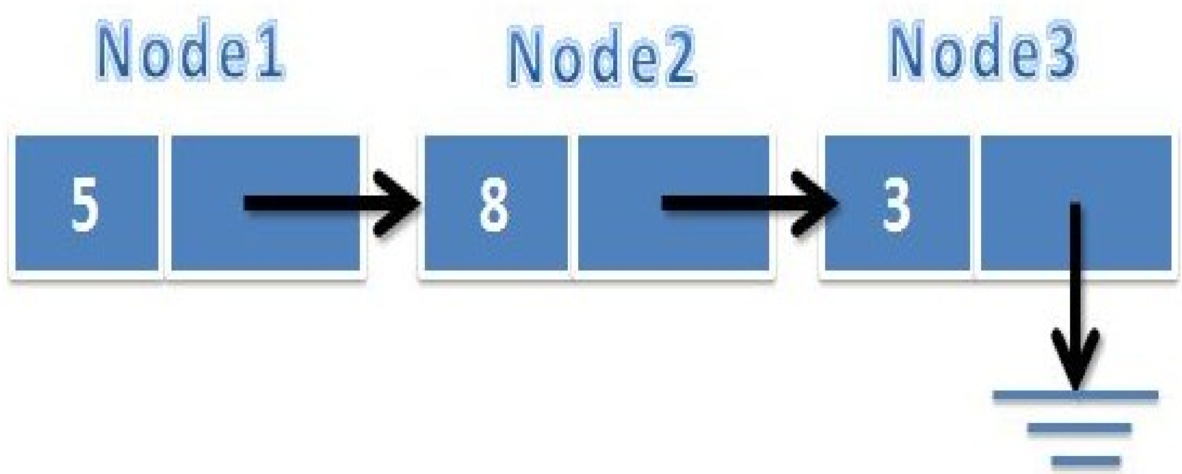
- It is very common data structure that is used to create tree, graph and other abstract data types.



- Linked list comprise of group or list of nodes in which each node have link to next node to form a chain

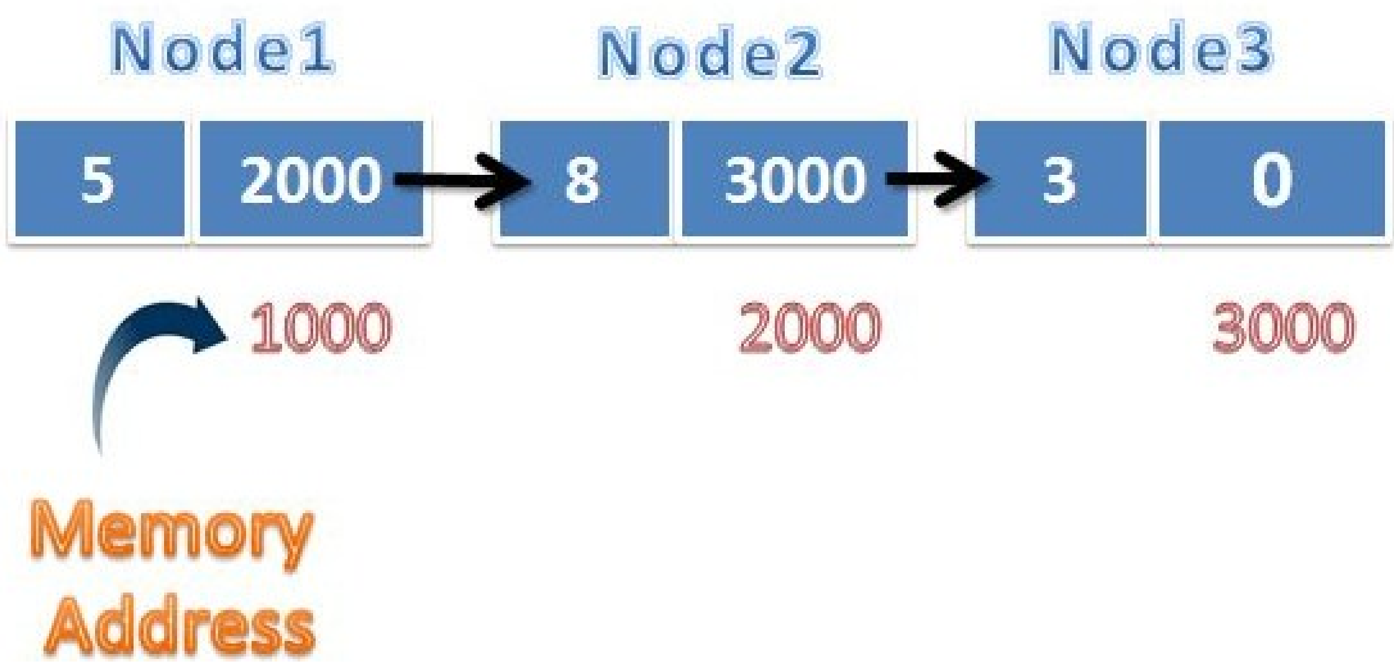
Linked List definition

- Linked List is **<u>Series of Nodes</u>**
- Each node Consist of two Parts **<u>Data Part & Pointer Part</u>**
- Pointer Part stores the **<u>address of the next node</u>**



What is linked list Node ?

Node A has two part one data part which consists of the 5 as data and the second part which contain the address of the next node (**i.e it contain the address of the next node**)

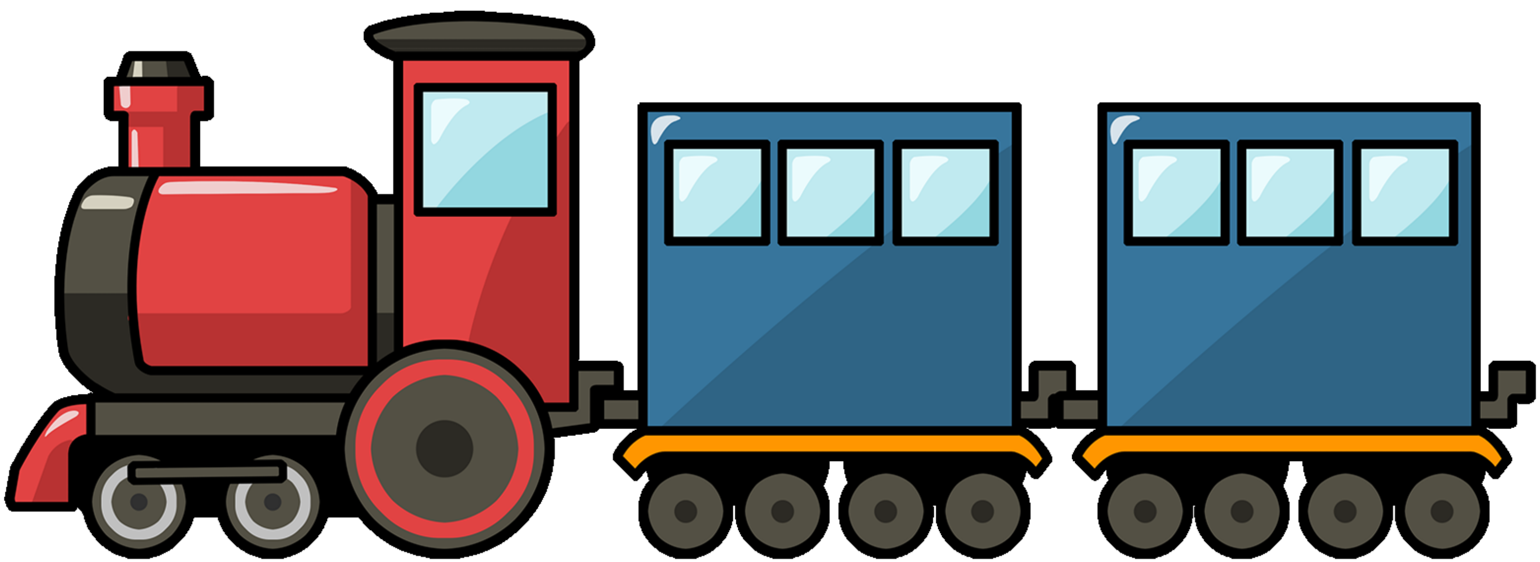


Linked list Blocks

No	Element	Explanation
1	Node	Linked list is collection of number of nodes
2	Address Field in Node	Address field in node is used to keep address of next node
3	Data Field in Node	Data field in node is used to hold data inside linked list.

Linked list Blocks

We can represent linked list in real life using train in which all the buggies are nodes and two coaches are connected using the connectors.

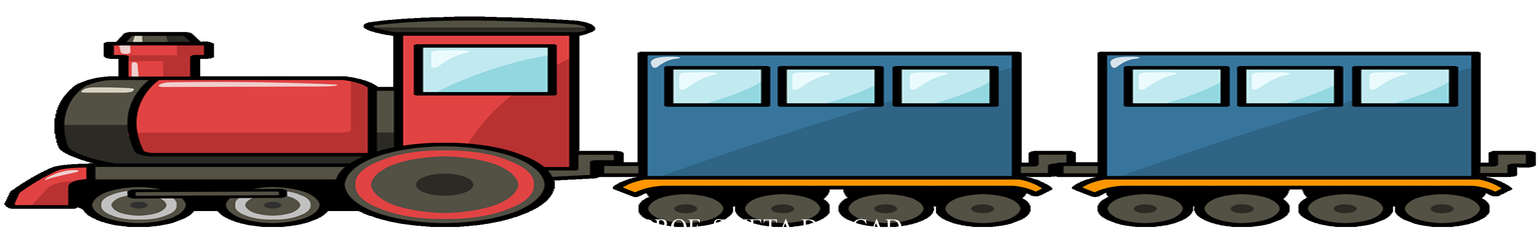


Linked list Blocks

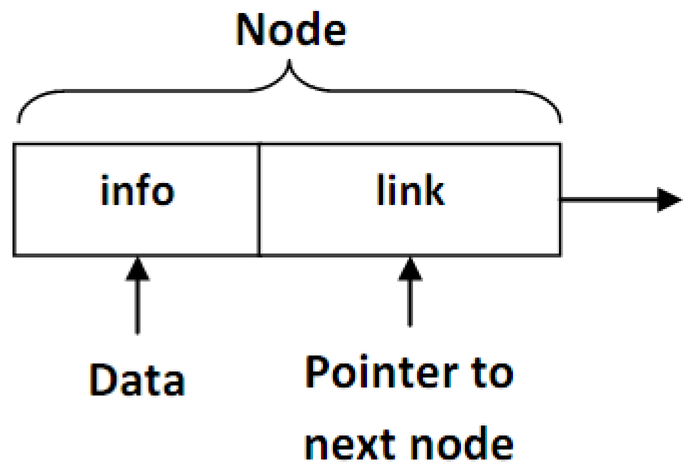
- In case of railway we have peoples seating arrangement inside the coaches is called as data part of lined list while connection between two buggies is address filed of linked list.

- Like linked list, trains also have last coach which is not further connected to any of the buggie.

- Engine can be called as first node of linked list



linked list



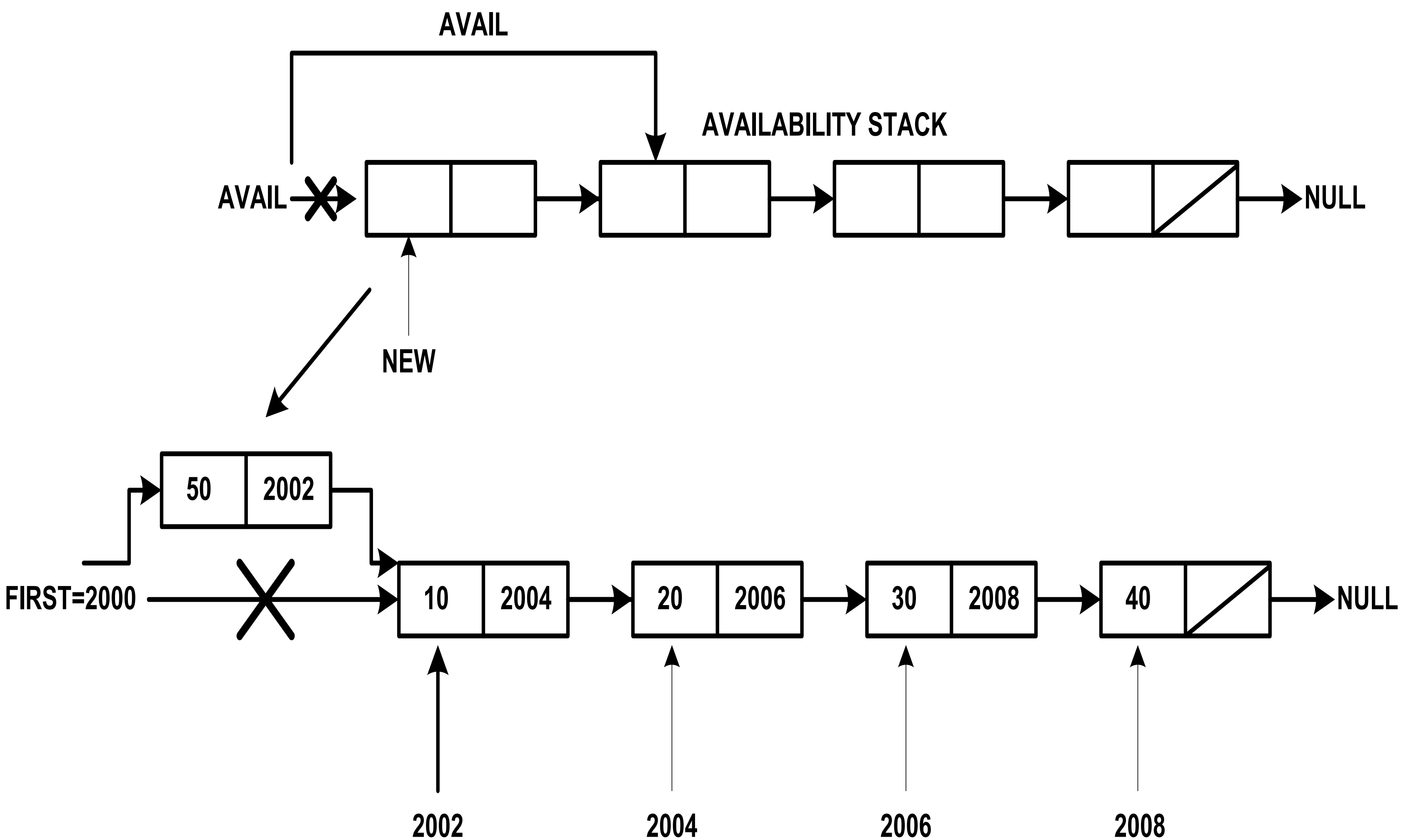

```
// C Structure to represent a node
struct node
{
int info;
struct node *link;
};
```

Operations on linked list

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

Algorithms for Singly linked lis

➢Algorithm to insert new node at beginning of the linked list



➢Algorithm to insert new node at beginning of the linked list

INSERTBEG (VAL,FIRST)

➢This function inserts a new element VAL at the beginning of the linked list.

➢FIRST is a pointer which contains address of first node in the list.

1[Check for availability stack underflow]
If AVAIL = NULL then

Write "Availability stack underflow"

Return

2[Obtain address of next free node]
NEW←AVAIL

3 [Remove free node from availability stack]
AVAIL←LINK (AVAIL)

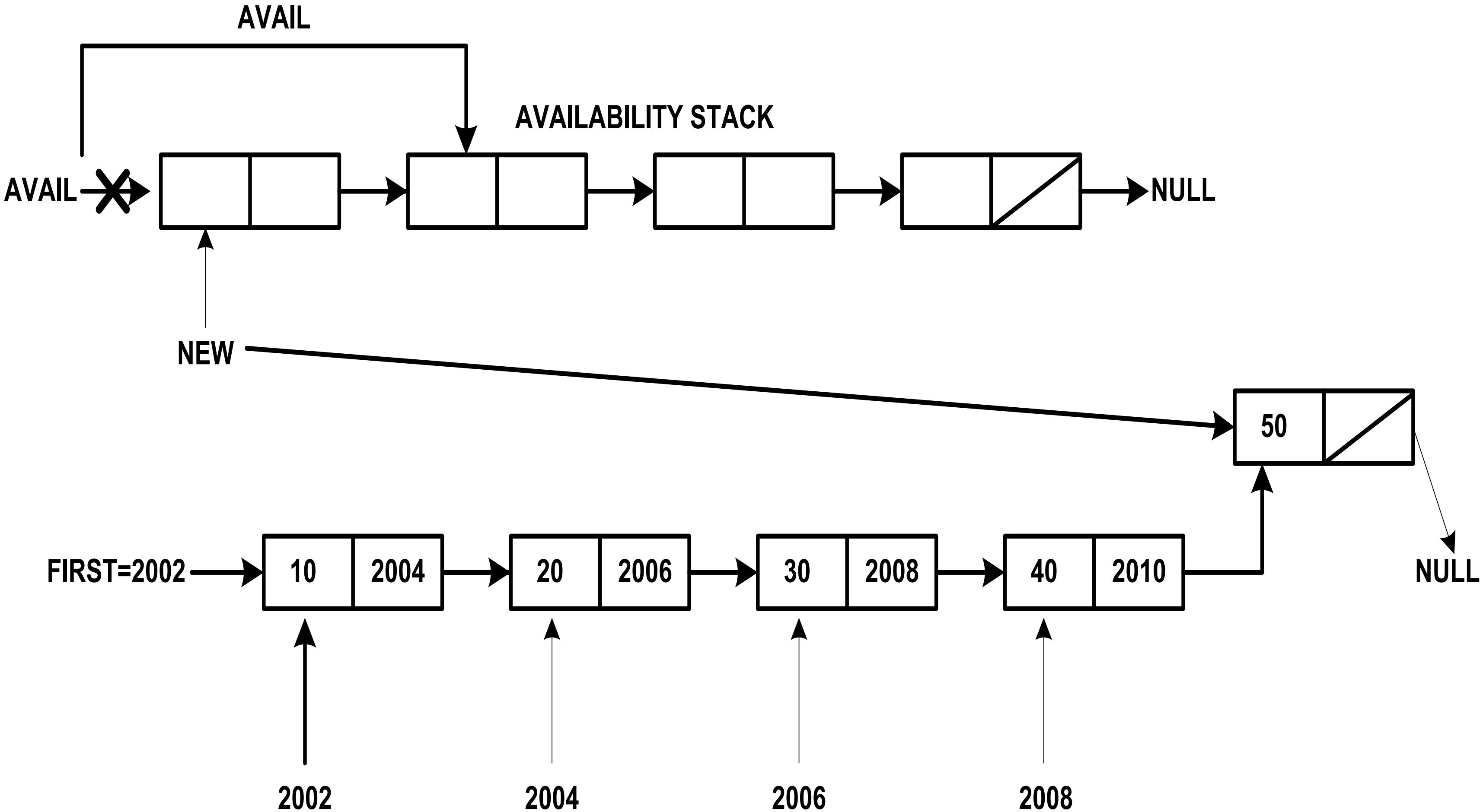
4[Initialize node to the linked list]
INFO (NEW) ←VAL

LINK (NEW) <- First

5 [Assign the address of the Temporary node to the First Node]
FIRST←NEW

6[Finished]
Return (FIRST)

➢Algorithm to insert new node at end of the linked list



➢Algorithm to insert new node at end of the linked list

INSERTEND (VAL,FIRST)

- ➢This function inserts a new element VAL at the end of the linked list.
- ➢FIRST is a pointer which contains address of first node in the list.

1[Check for availability stack underflow]
If AVAIL = NULL then

Write “Availability stack underflow”

Return

2[Obtain address of next free node]
NEW←AVAIL

3 [Remove free node from availability stack]
AVAIL←LINK (AVAIL)

4[initialize field of new node]
INFO (NEW) ←VAL

LINK (NEW) ←NULL

5[If list is empty?]
If FIRST = NULL then

FIRST←NEW

6[initialize search for last node]
SAVE←FIRST

7[Search end of the list]
Repeat while LINK (SAVE) ≠ NULL

SAVE←LINK (SAVE)

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

9 [Finished]
Return (FIRST)

➢Algorithm to insert new node at specific location

INSPOS (VAL, FIRST, X)

- This function inserts a new element VAL into the linked list before the node value X.
- FIRST is a pointer which contains address of first node in the list.

1[Check for availability stack underflow]
If AVAIL = NULL then

Write “Availability stack underflow”

Return

2[Obtain address of next free node]
NEW←AVAIL

3 [Remove free node from availability stack]
AVAIL←LINK (AVAIL)

4[initialize field of new node]
INFO (NEW) ←VAL

5[If list is empty?]
If FIRST = NULL then

LINK (NEW) ←NULL

FIRST←NEW, Return FIRST

6[If list contain only one node?]
If LINK(FIRST) = NULL then

LINK (NEW) ←FIRST

FIRST←NEW, Return FIRST

7[Search the list until desired address found]
SAVE←FIRST

Repeat while LINK (SAVE) ≠ NULL and INFO(SAVE) ≠ X

PRED←SAVE
SAVE←LINK (SAVE)

8[Node found?]

If(LINK(SAVE)==NULL) {
LINK(NEW)←NULL,LINK(SAVE)←NEW }

else

{ LINK (NEW) ←SAVE

LINK(PRED)←NEW}

9[Finished]
Return (FIRST)

➢Algorithm to delete first node of linked list

DELFIRST(FIRST)

➢This function deletes a first node from the list.

➢FIRST is a pointer which contains address of first node in the list.

1[Check for empty list]

If FIRST = NULL then

Write "List is empty"

Return

2[Check for the element in the list and delete it]

If LINK (FIRST) = NULL then

Y←INFO (FIRST)

FIRST←NULL

Else

TEMP←FIRST

Y←INFO (TEMP)

FIRST←LINK (TEMP)

3[Finished]

Return (FIRST)

Algorithm to delete last node of linked list

DELLAST(FIRST)

- This function deletes a last node from the list.
- FIRST is a pointer which contains address of first node in the list.

1[Check for empty list]

If FIRST = NULL then

Write "List is empty"

Return

2[Check for the element in the list and delete it]

If LINK (FIRST) = NULL then

Y ← INFO (FIRST)

FIRST ← NULL

Else

(Assign the address pointed by FIRST pointer to TEMP pointer)

TEMP ← FIRST

Repeat while LINK (TEMP) ≠ NULL

PRED ← TEMP

TEMP ← LINK (TEMP)

3 [Delete Last Node]

Y ← INFO (TEMP)

LINK (PRED) ← NULL

4[Finished]

Return (FIRST)

Algorithm to delete specific node from linked list

DELPOS(FIRST,N)

➢This function deletes a node with specific value from the list.

➢FIRST is a pointer which contains address of first node in the list.

1[Check for empty list]

If FIRST = NULL then

Write "List is empty"

Return

2[If there is only one node?]

If LINK (FIRST) = NULL then

Y←INFO (FIRST)

FIRST←NULL

3 [If list contains more than one node?]

TEMP←FIRST

Repeat while LINK (TEMP) ≠ NULL and INFO(TEMP) ≠ N

PRED←TEMP

TEMP←LINK (TEMP)

4[Node found?]

If INFO(TEMP) ≠ N then

Write "Node not found"

Else

Y←INFO (TEMP)

LINK (PRED) ←LINK (TEMP)

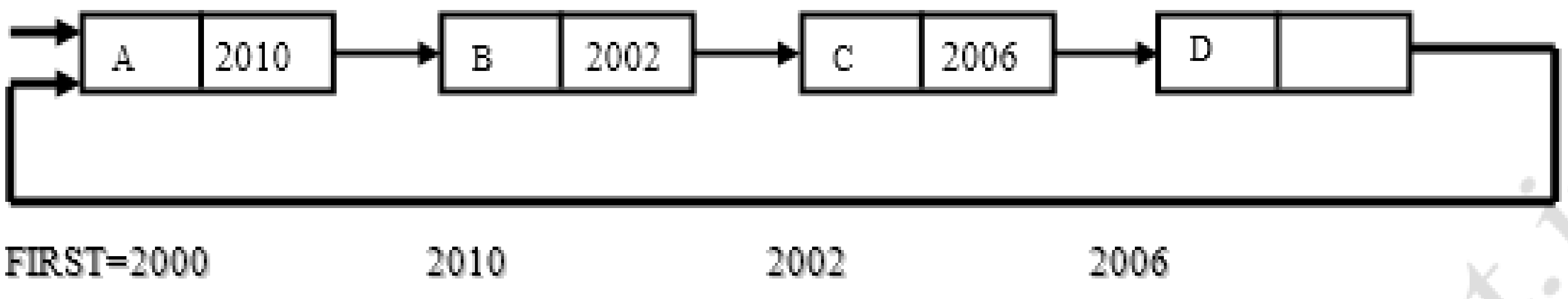
5[Finished]

Return (FIRST)

Circular Linked list

Circular Linked list

- A list in which last node contains a link or pointer to the first node in the list is known as circular linked list.
- Representation of circular linked list is shown below:



Advantage of circular linked list over singly linked linear list:

1. In singly linked list the last node contains NULL address. So if we are at middle of the list and want to access the first node we can not go backward in the list. Thus every node in the list is not accessible from given node. While in Circular linked list last node contains an address of the first node so every node is in the list is accessible from given node.
2. In singly linked list if we want to delete node at location X, first we have to find out the predecessor of the node. To find out the predecessor of the node we have to search entire list from starting from FIRST node in the list. So in order to delete the node at Location X we also have to give address of the FIRST node in the list. While in Circular linked list every node is accessible from given node so there is no need to give address of the first node in the list.
3. Concatenation and splitting operations are also efficient in circular linked list as compared to the singly linked list.

Disadvantage:

1. Without some care in processing it is possible to get into the infinite loop. So we must able to detect end of the list.
2. To detect the end of the list in circular linked list we use one special node called the HEAD node.

```
struct node
{
int data;
struct node *add;
};
struct node *first;
```

```
void insert_first(struct node **first,int val)
{
        struct node *new_node,*temp;
        new_node=(struct node  *) malloc (sizeof(struct node));
        new_node->data=val;
        if(*first==NULL)
        {
                *first=new_node;
                new_node->add=*first;
                return;
        }
        temp=*first;
        while(temp->add!=*first)
                temp=temp->add;
        new_node->add=*first;
        *first=new_node;
        temp->add=new_node;
}
```

```
void insert_last(struct node **first,int val)
{
        struct node *new_node,*temp;
        new_node=(struct node *)malloc(sizeof(struct node));
        new_node->data=val;
        if((*first)==NULL)
        {
                new_node->add=(*first);
                *first=new_node;
                return;
        }
        temp=(*first);
        while(temp->add!=(*first))
                temp=temp->add;
        new_node->add=(*first);
        temp->add=new_node;


}
```

```
void insert_after_any(struct node **first,int val,int key)
{
     struct node *temp,*new_node;
     if(first==NULL)
      {
          printf("link List is empty\n");
          return;
      }
    temp=first;
    do
    {
          if(temp->data==key)
          {
                    new_node=(struct node *) malloc(sizeof(struct node));
                    new_node->data=val;
```

```
            new_node->add=temp->add;
                    temp->add=new_node;
                    return;
            }
            temp=temp->add;
    }
    while(temp!=first);
    printf("key not found\n");
}
```

```c
int delete_first(struct node **first)
{
    struct node *temp,*temp2;
    int no;
    if(*first==NULL)
    {
        printf("Link List is Empty\n");
        return -1;
    }
    temp=*first;
    if((*first)==(*first)->add)
    {
        *first=NULL;
        no=temp->data;
        free(temp);
        return (no);
    }
```

```
while(temp->add!=*first)
        temp=temp->add;
    temp2=*first;
    no=(*first)->data;
    *first=(*first)->add;
    temp->add=*first;
    temp->add=*first;
    free(temp2);
    return no;
}
```

```
int delete_last(struct node **first)
{
   struct node *temp,*pred;
   int no;
   if(*first==NULL)
   {
                printf("\nLink List is Empty\n");
                return -1;
   }
   temp=*first;
   if((*first)->add==*first)
   {
                no=(*first)->data;
                *first=NULL;
                free(temp);
                return no;
   }
   while(temp->add!=*first)
   {
                pred=temp;
                temp=temp->add;
   }
    no=temp->data;
    pred->add=*first;
    free(temp);
    return no;
}
```

```
void delete_any(struct node **first,int key)
{
    struct node *temp,*pred;
    if(*first==NULL)
    {    printf("Link LIst is Empty\n"); return;
    }
    temp=*first;
    if(*first==(*first)->add && (*first)->data==key)
    {    *first=NULL;
                free(temp);
    }
    while(temp->data!=key&&temp->add!=*first)
    {
              pred=temp;
              temp=temp->add;
    }
    if(temp->data!=key)
    {
             printf("Key node not found\n");
             return;
    }
    else
    { pred->add=temp->add;
              free(temp);
    }
```

```c
void display(struct node *first)
{
        struct node *temp;
        if(first==NULL)
        {
                printf("\nlink list is empty\n");
                return;
        }
  printf("Link List Contains\n");
  temp=first;
  do{
        printf("%d ",temp->data);
        temp=temp->add;
  } while(temp!=first);
}
```

Doubly link list

Doubly link list

- In Singly linked list we are able to traverse the list in only one direction. However in some cases it is necessary to traverse the list in both directions.

- This property of linked list implies that each node must contain two link fields instead of one link field.

- One link is used to denote the predecessor of the node and another link is used to denote the successor of the node.

Doubly link list

Thus each node in the list consist of three fields:

1. Information
2. LPTR
3. RPTR

NODE

LPTR	INFO	RPTR

Doubly link list

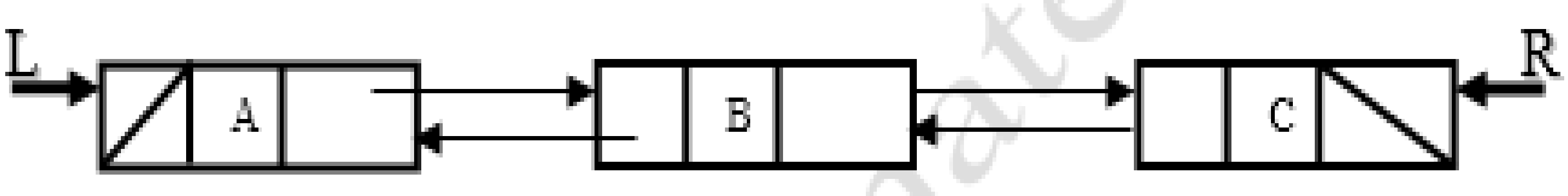
- In singly linked list we can traverse only in one direction while in doubly linked list we can traverse the list in both directions.

- Deletion operation is faster in doubly linked list as compared to the singly linked list

Advantage of Doubly link list

- A list in which each node contains two links one to the predecessor of the node and one to the successor of the node is known as doubly linked linear list or two way chain.

- Representation of doubly linked linear list is shown below



- L is a pointer denotes the left most node in the list.
- R is a pointer denotes the right most node in the list.
- The left link of the left most node and right link of right most node are set to NULL to indicate end of the list in each direction.

```
struct node
{
int data;
struct node *left,*right;
};

struct node *first;
```

```
void insert_last(struct node **first,struct node **last,int val)
{
struct node *new_node;
new_node=(struct node  *) malloc (sizeof(struct node));
new_node->data=val;
new_node->right=NULL;
if((*last) == NULL)
{
new_node->left=NULL;
*first=new_node;
*last=new_node;
return;
}
(*last)->right=new_node;
new_node->left=(*last);
*last=new_node;
}
```

```
void insert_first(struct node **first,struct node **last,int val)
{
struct node *new_node;
new_node=(struct node  *) malloc (sizeof(struct node));
new_node->data=val;
new_node->left=NULL;
if((*last) == NULL)
{
new_node->right=NULL;
*first=new_node;
*last=new_node;
return;
}
new_node->right=(*first);
(*first)->left=new_node;
(*first)=new_node;
}
```

```
void insert_after_any(struct node *first,int val,int key)
{
struct node *new_node,*temp;
if(first==NULL)
{
printf("\nLinked list is empty !");
return;
}
new_node=(struct node *)malloc(sizeof(struct node));
new_node->data=val;

if(first->right==NULL && first->data==key)
{
   new_node->right=NULL;
   new_node->left=first;
   first->right=new_node;
   return;
}
```

```
        while(first!=NULL)
{
if(first->data==key)
  {
new_node->right=first->right;
temp=first->right;
temp->left=new_node;
first->right=new_node;
new_node->left=first;
return;
    }
      first=first->right;
}
printf("\nKey  not found !");
free(new_node);
}
```

```
int delete_first(struct node **first,struct node **last)
{
struct node *temp;
int deleted_item;
if(*last==NULL)
{
printf("\nLinked list is empty !\n");
return -1;}
temp=*first;
Deleted_item=temp->data;
if(*first==*last)
{
*first=NULL;
*last=NULL;
free(temp);
return (deleted_item);
}
*first=temp->right;
free(temp);
return(deleted_item);
}
```

```
int delete_last(struct node **first, struct node **last)
{
struct node *temp,*temp2;
int deleted_item;
if(*last==NULL)
{
printf("\nLinked list is empty !\n");
return -1;
}
temp=*first;

if(*first==*last)
{
Deleted_item=temp->data;
*first=NULL;
*last=NULL;
free(temp);
return (deleted_item);
}
```

```
while(temp->right!=*last)
temp=temp->right;
temp2=*last;
no=temp2->data;
*last=temp;
temp->right=NULL;
free(temp2);
return(deleted_item);
}
```

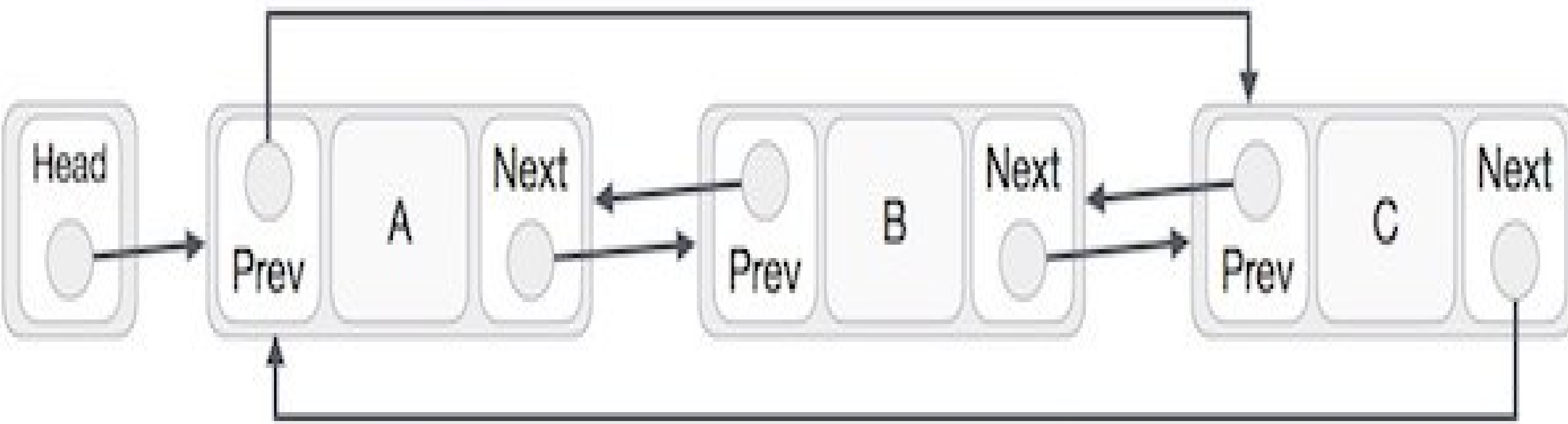
```
void delete_any(struct node **first,struct node **last,int key)
{
struct node *temp,*pred;
if(*last==NULL)
  {
printf("\nLinked List is empty !\n");
return ;
  }
temp=*first;
if(temp->data==key)
  {
if(temp->right==NULL)
{
  *first=NULL;
  *last=NULL;
  free(temp);
  return;
}
```

```
else
{
  temp=temp->right;
  free(*first);
  *first=temp;
  return;
}
  }
  while(temp->right!=NULL)
  {
if((temp)->data==key)
  {
(pred)->right=(temp)->right;
free(temp);
return ;
  }
pred=temp;
temp=temp->right;
```

```
    }
if(temp->data==key)
{
  *last=pred;
  pred->right=NULL;
  free(temp);
  return;
}
  printf("\nKey not found !\n");
}
```

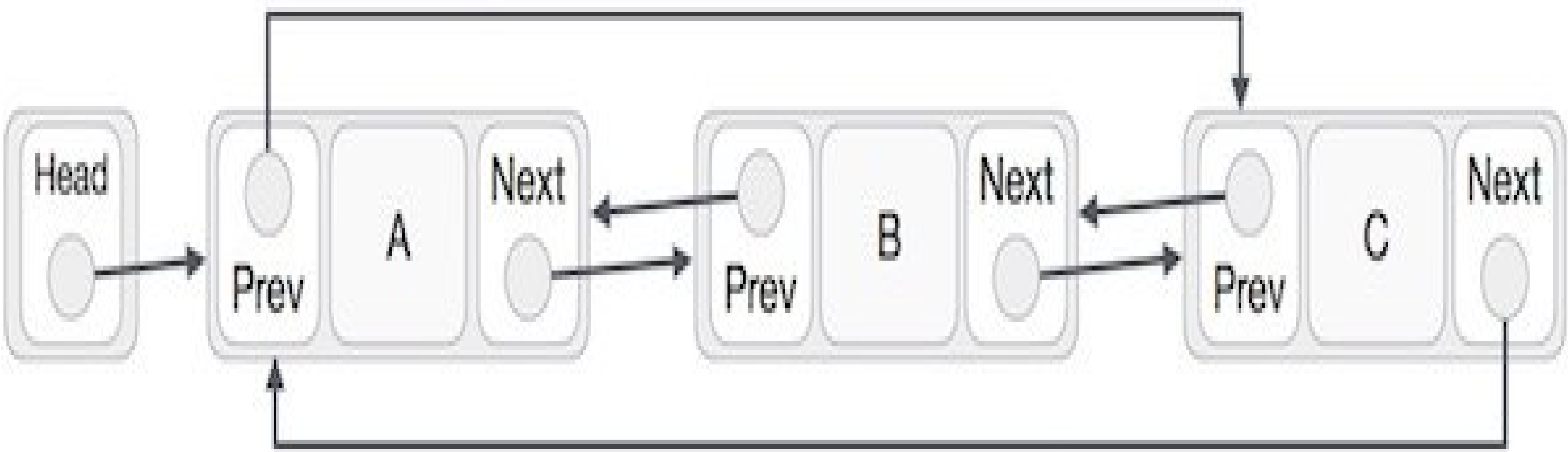
DOUBLY CIRCULAR LINKED LIST

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



→The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.

→The first link's previous points to the last of the list in case of doubly linked list.



- **Advantages**
- List can be traversed both ways from head to tail as well as tail to head
- Being a circular linked list tail can be reached with one operation from head node
- **Disadvantages**
- It takes slightly extra memory in each node to accommodate previous pointer
- **Practical Applications**
- Managing songs playlist in media player applications
- Managing shopping cart in online shopping

```
void insert_beg_dc ( )
{
struct node *ptr,*tpt;
ptr = (struct node *) malloc (sizeof (struct node));
if (ptr == NULL)
{printf ("OVERFLOW");
return;
}
printf ("Input new node");
scanf ("%d", & ptr -> info);
tpt = first->lpt;
Ptr->rpt=first;
First->lpt=ptr
Ptr->lpt=tpt;
Tpt->rpt=ptr;
printf ("New node is insert\n");
}
```

```
void insert_end_dc ( )
{
struct node *ptr,*tpt;
ptr = (struct node *) malloc (sizeof (struct node));
if (ptr == NULL)
{printf ("OVERFLOW");
return;
}
printf ("Input new node");
scanf ("%d", & ptr -> info);
tpt = first->lpt;
tpt->rpt=ptr;
ptr->lpt=tpt;
Ptr->rpt=first;
first->lpt=ptr;
printf ("New node is insert\n");
}
```

```
void delete_beg_dc ( )
{
struct node *ptr,*tpt,*cpt;
if(first == NULL)
{
printf ("Underflow\n");
return;
}
tpt=first;
ptr = first->rpt;
cpt=first->lpt;
Ptr->lpt=cpt;
cpt->rpt=ptr;
First=ptr;
free(tpt);
}
```

```
void delete_end_dc ( )
{
struct node *ptr,*tpt,*cpt;
if(first == NULL)
{
printf ("Underflow\n");
return;
}
Ptr=first->lpt;
Cpt=ptr->lpt;
First->lpt=cpt;
Cpt->rpt=first;
free(ptr);
}
```

DOUBINS (L, R, M, X)

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

DOUBINS (L, R, M, X)

1. [Create New Empty Node]

   NEW ⇦ NODE

2. [Copy information field]

   INFO (NEW) ← X

3. [Insert into an empty list]

   If R = NULL

   then LPTR (NEW) ← RPTR (NEW) ← NULL

   L ← R ← NEW

   Return

4. [Is left most insertion ?]

   If M = L

   then LPTR (NEW) ←NULL

   RPTR (NEW) ← M

   LPTR (M)← NEW

   L ← NEW

   Return

5. [Insert in middle]

   LPTR (NEW)← LPTR (M)

   RPTR (NEW) ← M

   LPTR (M) ← NEW

   RPTR (LPTR (NEW)) ← NEW

   Return

DOUBDEL (L, R, OLD)

- Given a doubly linked list with the addresses of left most and right most nodes are given by the pointer variables L and R respectively.

- It is required to delete the node whose address is contained in the variable OLD.

- Node contains left and right links with names LPTR and RPTR respectively.

DOUBDEL (L, R, OLD)

1. **[Is underflow ?]**

```
If      R=NULL
then    write (' UNDERFLOW')
        return
```

2. **[Delete node]**

```
If      L = R (single node in list)
then    L ← R ← NULL
else    If      OLD = L (left most node)
        then    L ← RPTR(L)
                LPTR (L) ← NULL
        else    if      OLD = R (right most)
                then    R ← LPTR (R)
                        RPTR (R) ← NULL
                else    RPTR  (LPTR (OLD)) ← RPTR (OLD)
                        LPTR  (RPTR (OLD)) ← LPTR (OLD)
```

3. **[FREE deleted node]**

```
FREE (OLD)
```

DOUBINS_ORD (L, R, M, X)

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

DOUBINS_ORD (L, R, M, X)

1. [Create New Empty Node]

NEW ⇐ NODE

2. [Copy information field]

INFO (NEW) ← X

3. [Insert into an empty list]

If R = NULL

then LPTR (NEW) ← RPTR (NULL) ← NULL

L ← R ← NEW

return

4. [Does the new node precedes all other nodes in List?]

If INFO(NEW) ≤ INFO(L)

then RPTR (NEW) ← L

LPTR(NEW)← NULL

LPTR (L) ← NEW

L ← NEW

Return

DOUBINS_ORD (L, R, M, X)

5. [Initialize temporary Pointer]

SAVE ← L

6. [Search for predecessor of New node]

Repeat while RPTR(SAVE) ≠ NULL and INFO(NEW) ≥ INFO(RPTR(SAVE))

SAVE ← RPTR (SAVE)

7. [Set link field of new node and its predecessor]

RPTR (NEW) ← RPTR(SAVE)

LPTR (RPTR(SAVE)) ← NEW

RPTR (SAVE) ← NEW

LPTR (NEW) ← SAVE

If SAVE = R

then RPTR(SAVE) ← NEW

Implementation procedure of basic primitive operations of the stack using:

(i) **Linear array**

(ii) **linked list.**

Implement PUSH and POP using Linear array

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

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

Implement PUSH and POP using Linear array

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

Implement PUSH and POP using Linked List

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

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

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

Implement PUSH and POP using Linked List

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

Implementation procedure of basic primitive operations of the Queue using:

(i) Linear array
(ii) linked list

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

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

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

```
int dequeue()
{
        int data;
        if(front == -1)
        {
                printf("Queue is underflow") ;
                return -1;
        }
        data = queue [front];
        if(front == rear)
        {
                front = rear = -1;
        }
        else
        {
                front++;
        }
        return data;
}
```

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

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

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


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

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

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