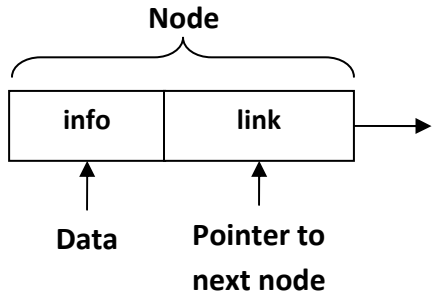
1. What is linked list? What are different types of linked list? OR Write a short note on singly, circular and doubly linked list. OR Advantages and disadvantages of singly, circular and doubly linked list.

- A linked list is a collection of objects stored in a list form.
- A linked list is a sequence of items (objects) where every item is linked to the next.
- A linked list is a non primitive type of data structure in which each element is dynamically allocated and in which elements point to each other to define a linear relationship.
- Elements of linked list are called nodes where each node contains two things, data and pointer to next node.
- Linked list require more memory compared to array because along with value it stores pointer to next node.
- Linked lists are among the simplest and most common data structures. They can be used to implement other data structures like stacks, queues, and symbolic expressions, etc…




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

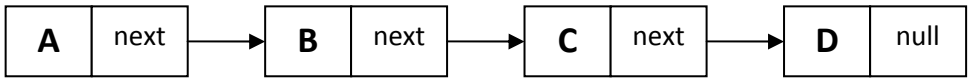
Operations on linked list

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

Types of linked list

Singly Linked List

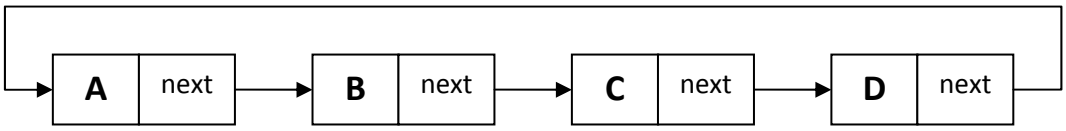
- It is basic type of linked list.
- Each node contains data and pointer to next node.
- Last node's pointer is null.
- Limitation of singly linked list is we can traverse only in one direction, forward direction.



Singly Linked List

Circular Linked List

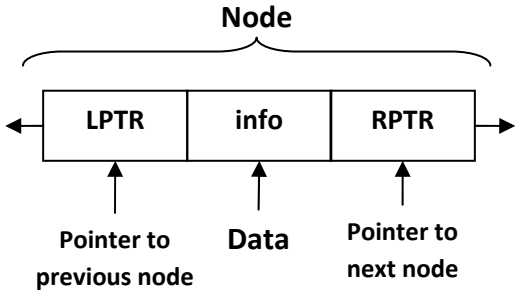
- Circular linked list is a singly linked list where last node points to first node in the list.
- It does not contain null pointers like singly linked list.
- We can traverse only in one direction that is forward direction.
- It has the biggest advantage of time saving when we want to go from last node to first node, it directly points to first node.
- A good example of an application where circular linked list should be used is a timesharing problem solved by the operating system.



Circular Linked List

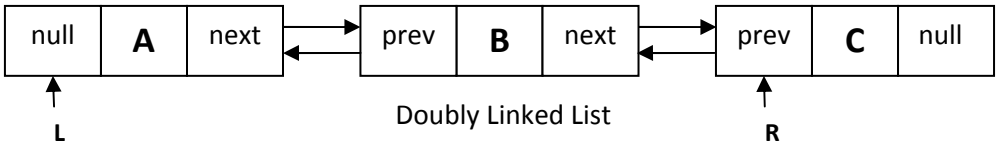
Doubly Linked list

- Each node of doubly linked list contains data and two pointers to point previous (LPTR) and next (RPTR) node.




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

- Main advantage of doubly linked list is we can traverse in any direction, forward or reverse.
- Other advantage of doubly linked list is we can delete a node with little trouble, since we have pointers to the previous and next nodes. A node on a singly linked list cannot be removed unless we have the pointer to its predecessor.
- Drawback of doubly linked list is it requires more memory compared to singly linked list because we need an extra pointer to point previous node.
- L and R in image denotes left most and right most nodes in the list.
- Left link of L node and right link of R node is NULL, indicating the end of list for each direction.



Doubly Linked List

2. Discuss advantages and disadvantages of linked list over array.

Advantages of an array

1. We can access any element of an array directly means random access is easy
2. It can be used to create other useful data structures (queues, stacks)
3. It is light on memory usage compared to other structures

Disadvantages of an array

1. Its size is fixed
2. It cannot be dynamically resized in most languages
3. It is hard to add/remove elements
4. Size of all elements must be same.
5. Rigid structure (Rigid = Inflexible or not changeable)

Advantages of Linked List

1. Dynamic size
2. It is easy to add/remove/change elements
3. Elements of linked list are flexible, it can be primary data type or user defined data types

Disadvantages of Linked List

1. Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
2. It cannot be easily sorted
3. We must traverse 1/2 the list on average to access any element
4. More complex to create than an array
5. Extra memory space for a pointer is required with each element of the list

3. What are the advantages and disadvantages of stack and queue implemented using linked list over array?

Advantages and disadvantages of stack & queue implemented using linked list over array is described below,

Insertion & Deletion Operation

- Insertion and deletion operations are known as push and pop operation in stack and as insert and delete operation in queue.
- In the case of an array, if we have n-elements list and it is required to insert a new element between the first and second element then n-1 elements of the list must be moved so as to make room for the new element.
- In case of linked-list, this can be accomplished by only interchanging pointers.
- Thus, insertion and deletions are more efficient when performed in linked list then array.

Searching a node

- If a particular node in a linked list is required, it is necessary to follow links from the first node onwards until the desired node is found.
- Where as in the case of an array, directly we can access any node

Join & Split

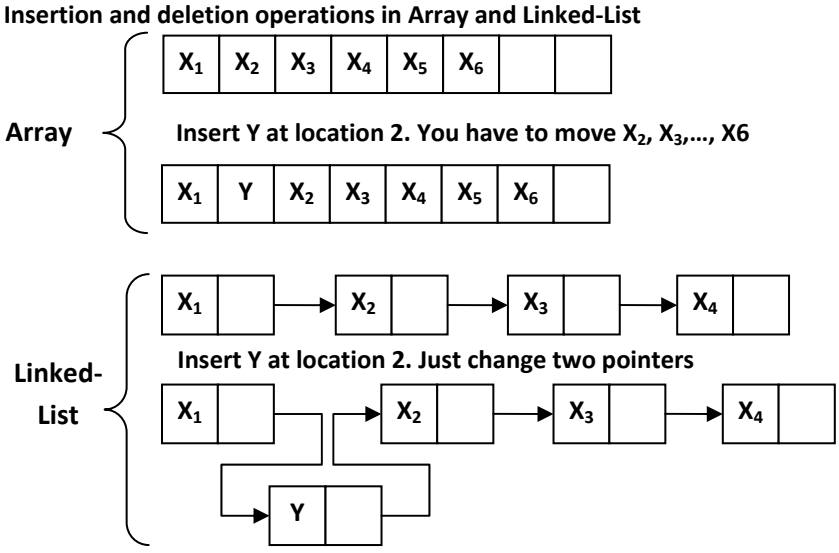
- We can join two linked list by assigning pointer of second linked list in the last node of first linked list.
- Just assign null address in the node from where we want to split one linked list in two parts.
- Joining and splitting of two arrays is much more difficult compared to linked list.

Memory

- The pointers in linked list consume additional memory compared to an array

Size

- Array is fixed sized so number of elements will be limited in stack and queue.
- Size of linked list is dynamic and can be changed easily so it is flexible in number of elements

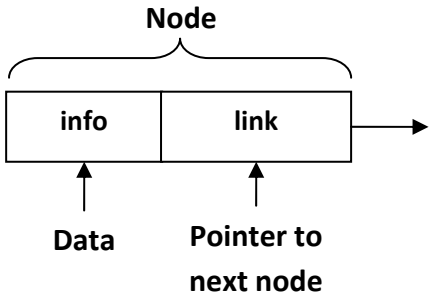


4. Write following algorithms for singly linked list.

1) Insert at first position
2) Insert at last position
3) Insert in Ordered Linked list
4) Delete Element

First few assumptions,

- Unless otherwise stated, we assume that a typical element or node consists of two fields namely; an information field called INFO and pointer field denoted by LINK. The name of a typical element is denoted by NODE.




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

Function : INSERT(X, First)

X is new element and FIRST is a pointer to the first element of a linked linear list then this function inserts X. Avail is a pointer to the top element of the availability stack; NEW is a temporary pointer variable. It is required that X precedes the node whose address is given by FIRST.

1 [Create New Empty Node]
NEW ⇦ NODE

1. [Initialize fields of new node and its link to the list]
INFO (NEW) ← X
LINK (NEW) ← FIRST

2. [Return address of new node]
return (NEW)

When INSERT is invoked it returns a pointer value to the variable FIRST

FIRST ← INSERT (X, FIRST)

Function: INSEND(X, First) (Insert at end)

A new element is X and FIRST is a pointer to the first element of a linked linear list then this function inserts X. AVAIL is a pointer to the top element of the availability stack; NEW and SAVE are temporary pointer variables. It is required that X be inserted at the end of the list.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Initialize field of NEW node]**
   INFO (NEW) ← X
   LINK (NEW) ← NULL
3. **[Is the list empty?]**
   If FIRST = NULL
   then return (NEW)
4. **[Initialize search for a last node]**
   SAVE ← FIRST
5. **[Search for end of list]**
   Repeat while LINK (SAVE) ≠ NULL
   SAVE ← LINK (SAVE)
6. **[Set link field of last node to NEW)**
   LINK (SAVE) ← NEW
7. **[Return first node pointer]**
   return (FIRST)

Function : INSORD(X, FIRST)

- There are many applications where it is desirable to maintain an ordered linear list. The ordering is in increasing or decreasing order on INFO field. Such ordering results in more efficient processing.
- The general algorithm for inserting a node into an ordered linear list is as below.
  1. Remove a node from availability stack.
  2. Set the field of new node.
  3. If the linked list is empty then return the address of new node.
  4. If node precedes all other nodes in the list then inserts a node at the front of the list and returns its address.
  5. Repeat step 6 while information contain of the node in the list is less than the information content of the new node.
  6. Obtain the next node in the linked list.
  7. Insert the new node in the list and return address of its first node.
- A new element is X and FIRST is a pointer to the first element of a linked linear list then this function inserts X. AVAIL is a pointer to the top element of the availability stack; NEW and SAVE are temporary points variables. It is required that X be inserted so that it preserves the ordering of the terms in increasing order of their INFO field.

1. **[Create New Empty Node]**
   NEW ⇦ NODE

2. **[Is the list is empty]**
   If FIRST = NULL
   then LINK (NEW) ← NULL
   return (NEW)

3. **[Does the new node precede all other node in the list?]**
   If INFO(NEW) ≤ INFO (FIRST)
   then LINK (NEW) ← FIRST
   return (NEW)

4. **[Initialize temporary pointer]**
   SAVE ← FIRST

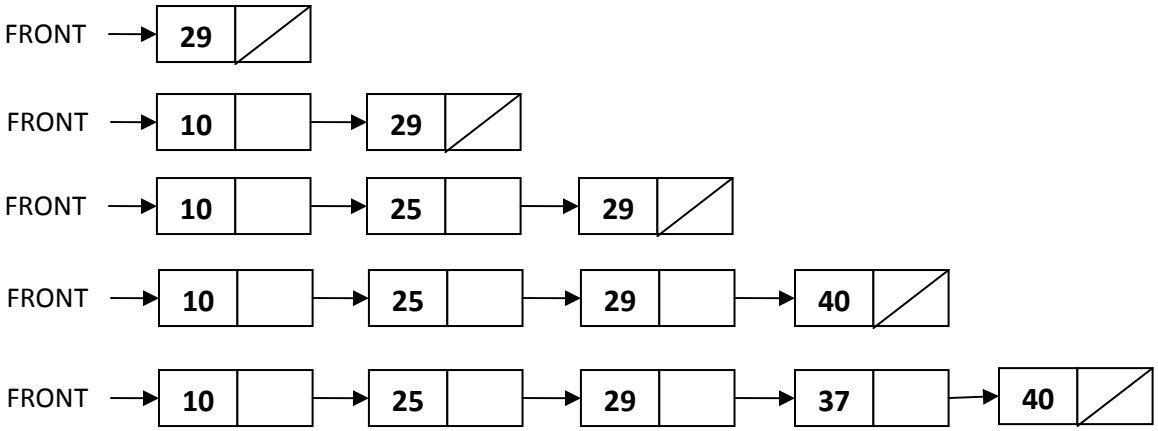
5. **[Search for predecessor of new node]**
   Repeat while LINK (SAVE) ≠ NULL and INFO (NEW) ≥ INFO (LINK (SAVE))
   SAVE ← LINK (SAVE)

6. **[Set link field of NEW node and its predecessor]**
   LINK (NEW) ← LINK (SAVE)
   LINK (SAVE) ← NEW

7. **[Return first node pointer]**
   return (FIRST)

By repeatedly involving function INSORD, we can easily obtains an ordered liner list for example the sequence of statements.

FRONT ← NULL
FRONT ← INSORD (29, FRONT)
FRONT ← INSORD (10, FRONT)
FRONT ← INSORD (25, FRONT)
FRONT ← INSORD (40, FRONT)
FRONT ← INSORD (37, FRONT)



Trace of construction of an ordered linked linear list using function INSORD

Procedure : DELETE(X, FIRST)

- Algorithm that deletes node from a linked linear list:-
  1. If a linked list is empty, then write under flow and return.
  2. Repeat step 3 while end of the list has not been reached and the node has not been found.
  3. Obtain the next node in list and record its predecessor node.
  4. If the end of the list has been reached then write node not found and return.
  5. Delete the node from list.
  6. Return the node into availability area.
- A new element is X and FIRST is a pointer to the first element of a linked linear list then this procedure deletes the node whose address is given by X. SAVE is used to find the desired node, and PRED keeps track of the predecessor of TEMP. Note that FIRST is changed only when X is the first element of the list.

1. **[Is Empty list?]**
   If FIRST = NULL
   then write ('Underflow')
   return

2. **[Initialize search for X]**
   SAVE ← FIRST

3. **[Find X]**
   Repeat thru step-5 while SAVE ≠ X and LINK (SAVE) ≠ NULL

4. **[Update predecessor marker]**
   PRED ← SAVE

5. **[Move to next node]**
   SAVE ← LINK (SAVE)

6. **[End of the list]**
   If SAVE ≠ X
   then write ('Node not found')
   return

7. **[Delete X]**
   If X = FIRST (if X is first node?)
   then FIRST ← LINK (FIRST)
   else LINK (PRED) ← LINK (X)

8. **[Free Deleted Node]**
   Free (X)

Function COPY (FIRST)

- FIRST is a pointer to the first node in the linked list, this function makes a copy of the list.
- The new list is to contain nodes whose information and pointer fields are denoted by FIELD and PTR, respectively. The address of the first node in the newly created list is to be placed in BEGIN. NEW, SAVE and PRED are points variables.
- A general algorithm to copy a linked list
  1. If the list is empty then return null
  2. If the availability stack is empty then write availability stack underflow and return else copy the first node.
  3. Report thru step 5 while the old list has not been reached.
  4. Obtain next node in old list and record its predecessor node.
  5. If availability stack is empty then write availability stack underflow and return else copy the node and add it to the rear of new list.

6. Set link of the last node in the new list to null and return.

1. **[Is Empty List?]**
   If FIRST = NULL
   then return (NULL)

2. **[Copy first node]**
   NEW ⇦ NODE
   New ← AVAIL
   AVAIL ← LINK (AVAIL)
   FIELD (NEW) ← INFO (FIRST)
   BEGIN ← NEW

3. **[Initialize traversal]**
   SAVE ← FIRST

4. **[Move the next node if not at the end if list]**
   Repeat thru step 6 while (SAVE) ≠ NULL

5. **[Update predecessor and save pointer]**
   PRED ← NEW
   SAVE ← LINK (SAVE)

6. **[Copy node]**
   If AVAIL = NULL
   then write ('Availability stack underflow')
   return (0)
   else NEW ← AVAIL
   AVAIL ← LINK (AVAIL)
   FIELD (NEW) ← INFO (SAVE)
   PTR (PRED) ← NEW

7. **[Set link of last node and return]**
   PTR (NEW) ← NULL
   return (BEGIN)

5. Write following algorithms for circular link list

1) Insert at First Position
2) Insert at Last Position
3) Insert in Ordered Linked List
4) Delete Element

FUNCTION: CIRCULAR_LINK_INSERT_FIRST (X, FIRST, LAST)

- A new element is X; and FIRST and LAST a pointer to the first and last element of a linked linear list respectively whose typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack; NEW is a temporary points variable. This function inserts X. It is required that X precedes the node whose address is given by FIRST.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Initialize fields of new node and its link to the list]**
   INFO (NEW) ← X
   If FIRST = NULL
   then LINK (NEW) ← NEW
   FIRST ← LAST ← NEW
   return(FISRT)
   else LINK (NEW) ← FIRST
   LINK (LAST) ← NEW
   FIRST ← NEW
   return(FIRST)

When invoked, INSERT returns a pointer value to the variable FIRST.

FIRST← INSERT (X, FIRST, LAST)

FUNCTION: CIR_LINK_INSERT_END (X, FIRST, LAST)

- A new element is X; and FIRST and LAST a pointer to the first and last element of a linked linear list respectively whose typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack; NEW is a temporary points variable. This function inserts X. It is required that X be inserted at the end of the list.

1. **[Create New Empty Node]**
   NEW ⇦ NODE

2. **[Initialize fields of new node and its link to the list]**
   If FIRST = NULL
   then LINK (NEW) ← NEW
   FIRST ← LAST ← NEW
   return(FIRST)
   else LINK(NEW) ← FIRST
   LINK(LAST) ← NEW
   LAST ← NEW
   return(FIRST)

FUNCTION: CIR_LINK_INSERT_ORDER (X, FIRST, LAST)

- A new element is X; and FIRST and LAST a pointer to the first and last element of a linked linear list respectively whose typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack; NEW and SAVE are temporary points variables. It is required that X be inserted so that it preserves the ordering of the terms in increasing order of their INFO field.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Copy information content into new node]**
   INFO (NEW) ← X
3. **[Is Linked List is empty?]**
   If FIRST = NULL
   then LINK (NEW) ← NEW
   FIRST ← LAST ← NEW
   return(FIRST)
4. **[Does new node precedes all other nodes in List?]**
   If INFO (NEW) ≤ INFO (FIRST)
   then LINK (NEW) ← FIRST
   LINK (LAST) ← NEW
   FIRST ← NEW
   return(FIRST)
5. **[Initialize Temporary Pointer]**
   SAVE ← FIRST
6. **[Search for Predecessor of new node]**
   Repeat while SAVE ≠ LAST and INFO(NEW) ≥ INFO(LINK(SAVE))
   SAVE ← LINK(SAVE)
7. **[Set link field of NEW node and its Predecessor]**
   LINK(NEW) ← LINK(SAVE)
   LINK(SAVE) ← NEW
   If SAVE = LAST
   then LAST ← NEW
8. **[Return first node address]**
   return(FIRST)

PROCEDURE: CIR_LINK_DELETE (X, FIRST, LAST)

- A new element is X; and FIRST and LAST a pointer to the first and last element of a linked linear list respectively whose typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack;, this procedure deletes the node whose address is given by X. TEMP is used to find the desired node, and PRED keeps track of the predecessor of TEMP. Note that FIRST is changed only when X is the first element of the list.

1. **[Is Empty List?]**
   If FIRST = NULL
   then write ('Linked List is Empty')
   return
2. **[Initialize Search for X]**
   TEMP ← FIRST
3. **[Find X]**
   Repeat thru step 5 while SAVE ≠ X and SAVE ≠ LAST
4. **[Update predecessor marker]**
   PRED ← SAVE
5. **[Move to next node]**
   SAVE ← LINK (SAVE)
6. **[End of Linked List]**
   If SAVE ≠ X
   then write('Node not found')
   return
7. **[Delete X]**
   If X = FIRST
   then FIRST ← LINK (FIRST)
   LINK (LAST) ← FIRST
   else LINK (PRED) ← LINK(X)
   If X = LAST
   then LAST ← PRED
8. **[Free Deleted Node]**
   Free (X)

6. Write an algorithm to perform each of the following operations on Circular singly linked list using header node

1) add node at beginning

2) add node at the end

3) insert a node containing x after node having address P

4) delete a node which contain element x

FUNCTION: CIR_LINK_HEAD_INSERT_FIRST (X, FIRST, LAST)

- A new element is X; and FIRST and LAST a pointer to the first and last element of a linked linear list respectively whose typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack; NEW is a temporary points variable. HEAD is the address of HEAD node. This function inserts X. It is required that X precedes the node whose address is given by FIRST.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Initialize fields of new node and its link to the list]**
   INFO (NEW) ← X
   LINK (NEW) ← LINK (HEAD)
   LINK (HEAD) ← NEW

FUNCTION: CIR_LINK_HEAD_INSERT_LAST (X, FIRST, LAST)

- A new element is X; and FIRST and LAST a pointer to the first and last element of a linked linear list respectively whose typical node contains INFO and LINK fields. Avail is a pointer to the top element of the availability stack; NEW is a temporary points variable. HEAD is the address of HEAD node. This function inserts X. It is required that X be inserted at the end of the list.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Initialize fields of new node and its link to the list]**
   INFO (NEW) ← X
   LINK (NEW) ← HEAD
   LINK (LAST) ← NEW
   LAST ← NEW

FUNCTION: CIR_LINK_HEAD_INSERT_AFTER_Node-P (X, FIRST, LAST)

- A new element is X; and FIRST and LAST a pointer to the first and last element of a linked linear list respectively whose typical node contains INFO and LINK fields. Avail is a pointer to the top element of the availability stack; NEW is a temporary points variable. HEAD is the address of HEAD node. This function inserts X. It is required to insert a node after a node having address P.

1. **[Create New Empty Node]**
   NEW ⇦ NODE

2. **[Initialize fields of new node and its link to the list]**
   INFO (NEW) ← X
   LINK (NEW) ← LINK (P)
   LINK (P) ← NEW
   If P = LAST
   then LAST ← NEW

PROCEDURE: CIR_LINK_HEAD_DELETE (X, FIRST, LAST)

- FIRST and LAST a pointer to the first and last element of a linked linear list respectively whose typical node contains INFO and LINK fields. Avail is a pointer to the top element of the availability stack; SAVE is a temporary pointer variable. HEAD is the address of HEAD node. This function inserts X. It is required to delete element having value X.

1. **[Is Empty List?]**
   If FIRST = NULL
   then write ('Underflow)
   return
2. **[Initialize Search for X]**
   SAVE ← FIRST
3. **[Find X]**
   Repeat thru step 5 while INFO(SAVE) ≠ X and SAVE ≠ LAST
4. **[Update Predecessor]**
   PRED ← SAVE
5. **[Move to next node]**
   SAVE ← LINK(SAVE)
6. **[End of the List]**
   If INFO (SAVE) ≠ X
   then write('Node not Found')
   return
7. **[Delete node X]**
   If INFO (FIRST) = X
   then LINK (HEAD) ← LINK(FIRST)
   else LINK (PRED) ← LINK(SAVE)
   If SAVE = LAST
   then LAST ← PRED
8. **[Free Deleted Node]**
   Free (X)

7. Write following algorithms for doubly link list

1) Insert

2) Insert in Ordered Linked List

3) Delete Element

PRDCEDURE 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.

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. **[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

PROCEDURE 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.

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
5. **[Initialize top 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

PROCEDURE 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 id contained in the variable OLD. Node contains left and right links with names LPTR and RPTR respectively.

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. **[Return deleted node]**

```
restore (OLD)
return
```

8. Write the 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;
}

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;
}

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

9. Write the 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

```
# 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++;
      }
}
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;
     }
}

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);
}
```

10. Write an algorithm to implement ascending priority queue using singular linear linked list which has insert() function such that queue remains ordered list. Also implement remove() function

```
struct node
{
      int priority;
      int info;
      struct node *link;
}*front = NULL;

insert()
{
      struct node *tmp,*q;
      int added_item,item_priority;
      tmp = (struct node *)malloc(sizeof(struct node));
      printf("Input the item value to be added in the queue : ");
      scanf("%d",&added_item);
      printf("Enter its priority : ");
      scanf("%d",&item_priority);
      tmp->info = added_item;
      tmp->priority = item_priority;
      /*Queue is empty or item to be added has priority more than
first item*/
      if( front == NULL || item_priority < front->priority )
      {
            tmp->link = front;
            front = tmp;
      }
      else
      {
            q = front;
            while( q->link != NULL &&
                        q->link->priority <= item_priority )
            {
                  q=q->link;
            }
            tmp->link = q->link;
            q->link = tmp;
      }/*End of else*/
}/*End of insert()*/
```

```
remove()
{
      struct node *tmp;
      if(front == NULL)
            printf("Queue Underflow\n");
      else
      {
            tmp = front;
            printf("Deleted item is %d\n",tmp->info);
            front = front->link;
            free(tmp);
      }
}/*End of remove()*/

display()
{
      struct node *ptr;
      ptr = front;
      if(front == NULL)
            printf("Queue is empty\n");
      else
      {
            printf("Queue is :\n");
            printf("Priority Item\n");
            while(ptr != NULL)
            {
                  printf("%5d %5d\n",ptr->priority,ptr->info);
                  ptr = ptr->link;
            }
      }/*End of else */
}/*End of display() */
```