Linked List

A linked list is an ordered collection of finite homogeneous data elements called nodes, where the linear order is maintained by means of links or pointers.

Key Terms.

Each node in a linked list has basically two fields.

1. Data Field
2. Link Field.

The data field contains the actual value to be stored and processed. And the link field contains the address of the next node in the linked list. The elements in a linked list are not ordered by their physical placement in memory, but by their logical links stored as part of the data within the node itself.

Null Pointer

The link field of the last node contains NULL rather than a valid address. It is a null pointer and indicates the end of the list.

⊛ External Pointer.

It is a pointer to the very first node in the linked list, it enables us to access the entire linked list.

⊛ Empty List

If the nodes are not present in a linked list, then it is called an empty linked list. A linked list can be made an empty list by assigning a NULL value to the external pointer.

Start = NULL

⊛ Advantages.

1. Linked lists are dynamic data structure. That is, they can grow or shrink during the execution of a program.
2. Efficient memory utilization. Here memory is not preallocated. Memory is allocated whenever it is required and it is deallocated when it is no longer needed.
3. Insertion and deletion are easier and efficient. Linked list provides flexibility in inserting a data item at a specified

position and deletion of a data item from a given position.

4. Many complex applications can be carried out with linked lists.

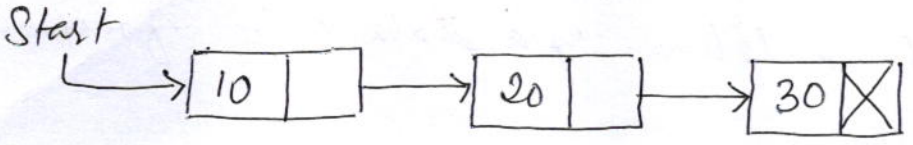
⊛ Disadvantages

1. More memory: If the number of fields are more, then more memory space is needed.
2. Access to an arbitrary data item is little cumbersome and also time consuming

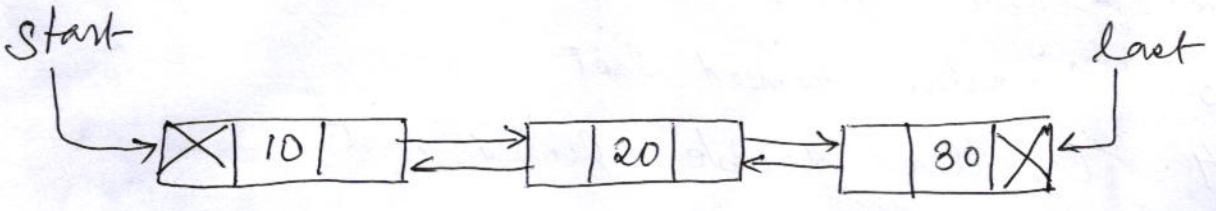
⊛ Types of Linked List

1. Single linked list
2. Double linked list
3. Circular linked list
4. Circular double linked list

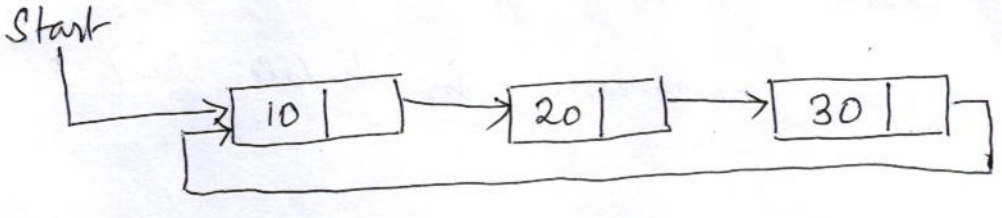
1. A <u>Single linked list</u> is one in which all nodes are linked together in some sequential manner. It has a beginning and an end. The problem with this list is that we cannot access the predecessor of node from the current node. This can be overcome in double linked list.



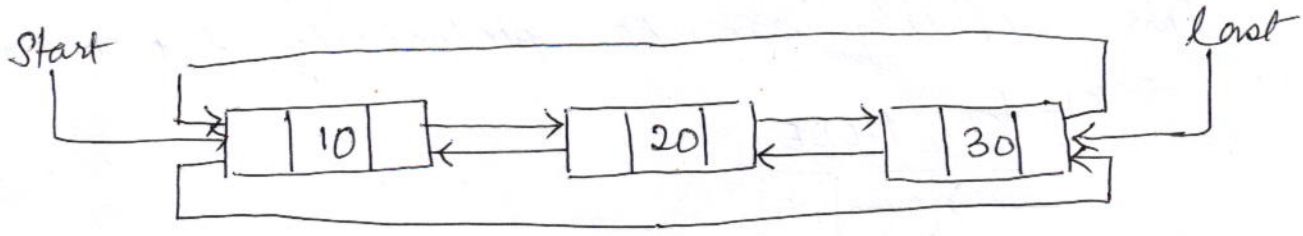
2. A <u>double linked list</u> is one in which all nodes are linked together by two links which help in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has 2 link fields, one pointing to the left node (previous) and other pointing to the right node (next). This helps to traverse the list both in the forward and backward direction.



3. A <u>circular linked list</u> is one which has no beginning and no end. It is a linked list where the last node points back to the first node



4. A circular double linked list is one which has both the successor pointer and predecessor pointer in circular manner.



⊛ Single Linked list

In C, a linked list is created using structures, pointers and dynamic memory allocation function malloc. We consider 'start' as an external pointer. This helps in creating and accessing other nodes in the linked list. Consider the following structure definition.

```
struct node
  {  int num;
     struct node *ptr;
  };
typedef struct node NODE;
  NODE  *start;
start = (NODE *) malloc (sizeof (NODE));
```

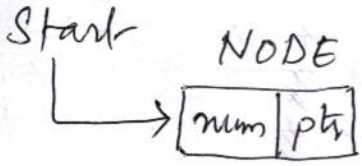
When the statement

```
start = (NODE *) malloc (sizeof (NODE));
```

is executed, a block of memory sufficient

to store the ~~NODE~~ NODE is allocated and assigns 'start' as the starting address of the NODE.

This activity can be pictorially shown as

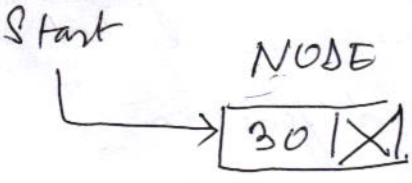


Now we can assign values to the respective fields of NODE.

member selection operator.

```
start → num = 30;
start → ptr = NULL;
```

Now the NODE would look like this:



⊛ Inserting Nodes

To insert an element in a linked list the following three things should be done

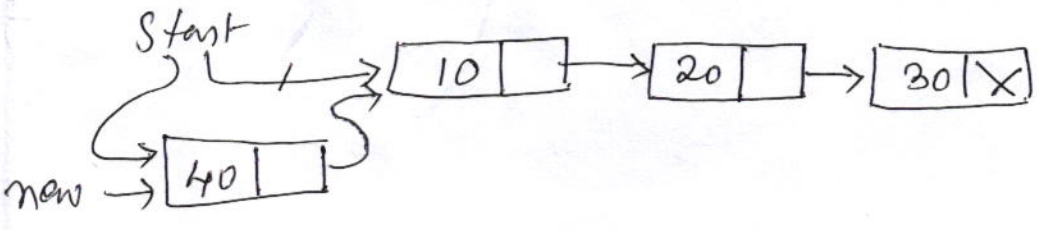
1. Allocating a node
2. Assigning the data.
3. Adjusting the pointers.

① TRAVERSE()

1. If (START = NULL) then
    1. Print "No elements in list"
    2. Return.
2. Endif.
3. PTR = START
4. Repeat while (PTR ≠ NULL)
    1. Print PTR → data
    2. PTR = PTR → link
5. Endwhile
6. Stop.

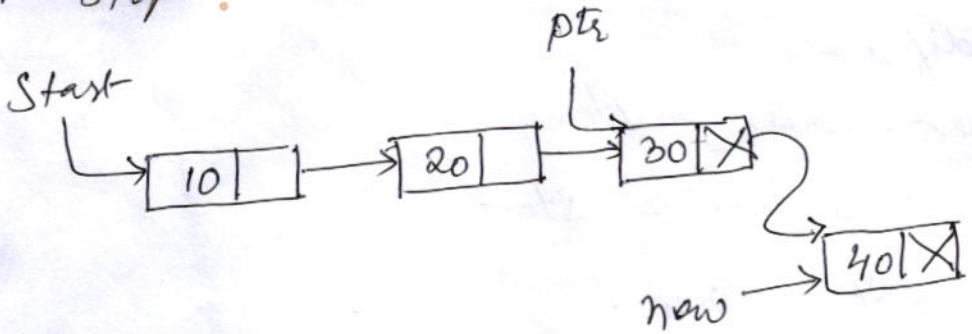
② INSERT_FRONT (start, item)

1. new = (NODE *) malloc (sizeof (NODE))
2. If (new = NULL) Then.
    1. Print "Memory Underflow: No Insertion"
    2. Return
3. Endif.
4. new → data = item.
5. new → link = start
6. start = new.
7. Stop.



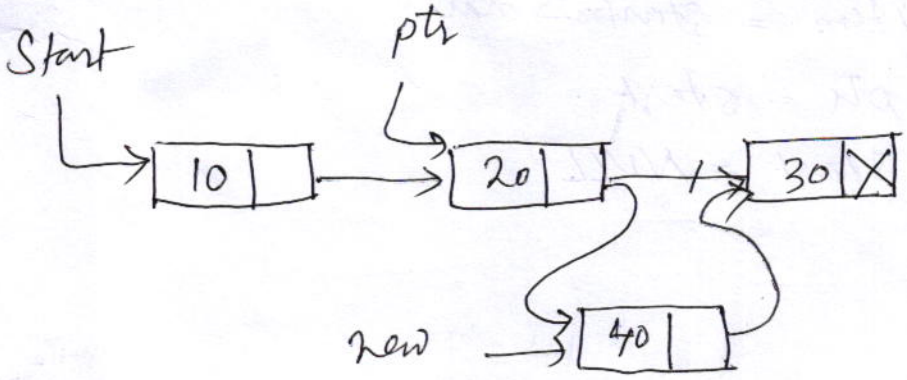
③ INSERT_LAST (start, item)

1. new = (NODE *) malloc (sizeof (NODE))
2. If (new = NULL) then.
    1. Print "Memory Underflow: No Insertion"
    2. Return.
3. Endif
4. new → data = item.
5. new → link = NULL.
6. If (start = NULL)
    1. Start = new.
7. Else
    1. ptr = start
    2. Repeat while (ptr → link ≠ NULL)
        1. ptr = ptr → link
    3. Endwhile
    4. ptr → link = new.
8. Endif.
9. Stop.



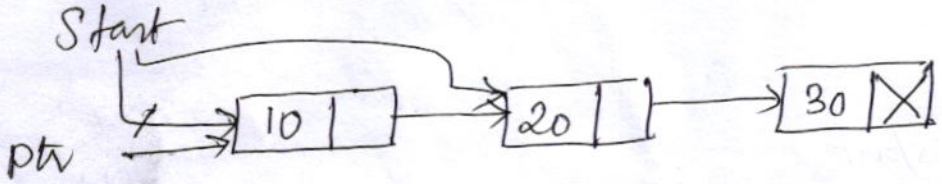
(4) INSERT_LOC (start, item, loc)

1. new = (NODE *) malloc (sizeof (NODE))
2. If (new = NULL) then:
    1. Print "Memory Underflow : No Insertion"
    2. Return
3. Endif
4. new → data = item
5. If (loc = 1) then
    1. new → link = start
    2. start = new
    3. return
6. Endif
7. ptr = start
8. Repeat for I = 1 to loc - 2, step 1.
    1. ptr = ptr → link
9. endfor
10. new → link = ptr → link
11. ptr → link = new
12. Stop



(5) DELETE_FRONT (start)

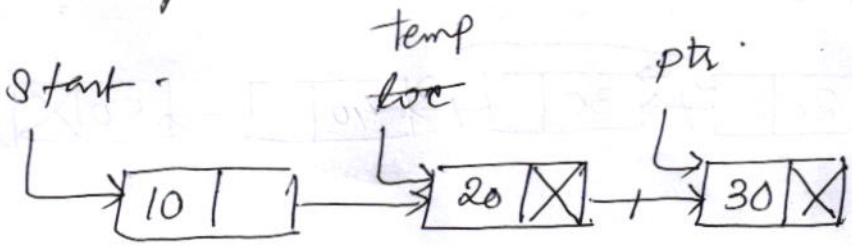
1. If (start = NULL) then.
    1. Print "Linked list is Empty"
    2. Return
2. Endif.
3. ptr = start.
4. item = start → data.
5. start = start → link
6. Free (ptr).
7. Stop.



(6) DELETE_LAST (start)

1. If (start = NULL) then.
    1. Print "Linked list is empty"
    2. Return.
2. Endif.
3. If (start → link = NULL) then.
    1. item = start → data.
    2. ptr = start.
    3. start = NULL
4. Endif.

5. ptr = start.
6. Repeat while (ptr → link ≠ NULL)
   1. ~~loc~~ temp = ptr.
   2. ptr = ptr → link.
7. Endwhile.
8. item = ptr → data.
9. ~~loc~~ temp → link = NULL. (temp → link = NULL)
10. Free (ptr).
11. Stop.



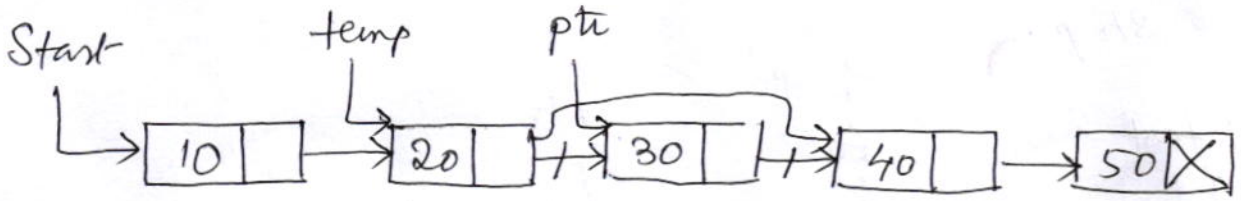
(7) DELETE - loc (start, loc)

1. If (start = NULL) then.
   1. Print "Linked list is empty"
   2. Return.
2. Endif.
3. ptr = start.
4. If (loc = 1) then.
   1. item = start → data.
   2. start = start → link
   3. Free (ptr).
5. Endif.

6. Repeat for I = 1 to loc - 1, step 1.

1. temp = ptr.
2. ptr = ptr → link.

7. Endfor.
8. item = ptr → data.
9. temp → link = ptr → link.
10. Free (ptr).



⊛ Double Linked list.

A single linked list termed as one-way list, because the beginning from the first node to any node by evaluating ptr → link, one can traverse the list in only one direction. Another problem of single linked list is that deleting a node from single linked list requries keeping track of previous node. Double linked list overcomes this problem.

Double linked list is a two-way list as we can traverse both in the foward and backward direction. This is accomplished by introducing two link fields instead of one as in single linked list. The type declaration for such a node containing integers is as follows.

```
typedef struct dlist
{   int data;
    struct dlist *llink;
    struct dlist *rlink;
} DNODE;
```

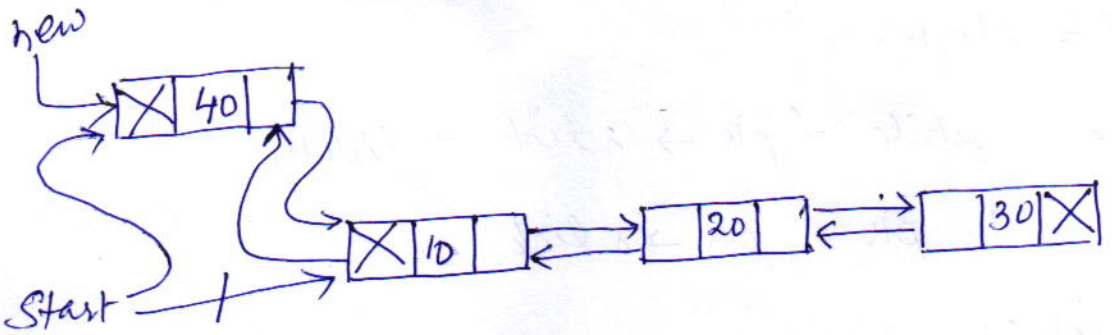
Node structure of Double linked list-

TRAVERSE()

1. If (start = NULL) then
    1. Print "No Elements"
    2. Return
2. Endif
3. ptr = start
4. Repeat while (ptr ≠ NULL)
    1. Print ptr → data
    2. ptr = ptr → rlink
5. Endwhile
6. Stop.

INSERT_FRONT()

1. new = (DNODE *) malloc (sizeof(DNODE))
2. new → data = item.
3. If (start = NULL) then.
    1. new → rlink = NULL
    2. new → llink = NULL
    3. start = new.
4. Endif.
5. new → rlink = start.
6. new → llink = NULL
7. start → llink = new
8. start = new.
9. Stop.



(*) Inserting a new node at the beginning of double linked list

1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Make the rlink of the new node to point to the first node of the list. And make the the llink of the first node to point to the new node.
4. Finaley reset the start pointer. That is, make it to point to the new node which has been inserted at the beginning.

```
INSERT_LAST()
1. new = (DNODE *) malloc (sizeof (DNODE));
2. new → data = item;
3. If (start = NULL) then
        1. new → rlink = NULL
        2. new → llink = NULL;
        3. start = new;
4. Endif;
```

5. ptr = start.

6. Repeat while (ptr → rlink ≠ NULL)

    1. ptr = ptr → rlink.

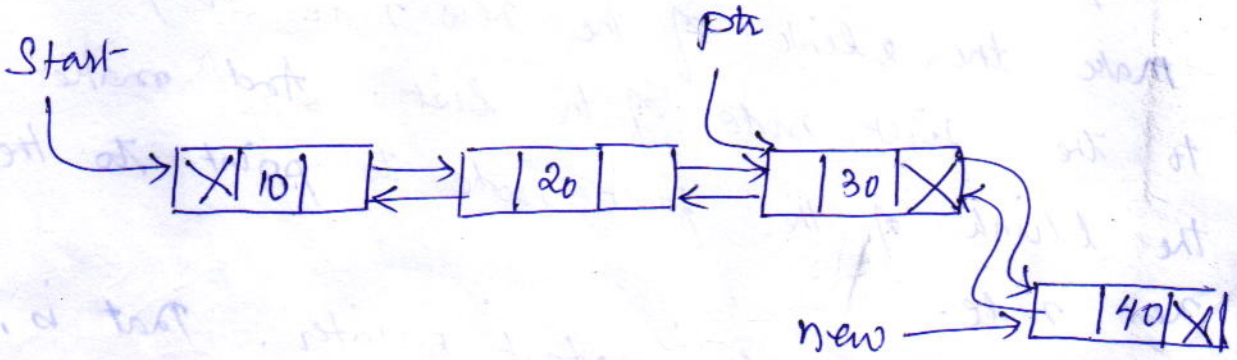
7. endwhile.

8. new → llink = ptr.

9. new → rlink = NULL.

10. ptr → rlink = new.

11. stop.



INSERT_LOC()

1. new = (DNODE *) malloc (sizeof (DNODE))

2. new → data = item.

3. If (loc = 1) then

    1. new → rlink = start.

    2. new → llink = NULL.

    3. start → llink = new.

    4. start = new.

4. endif.

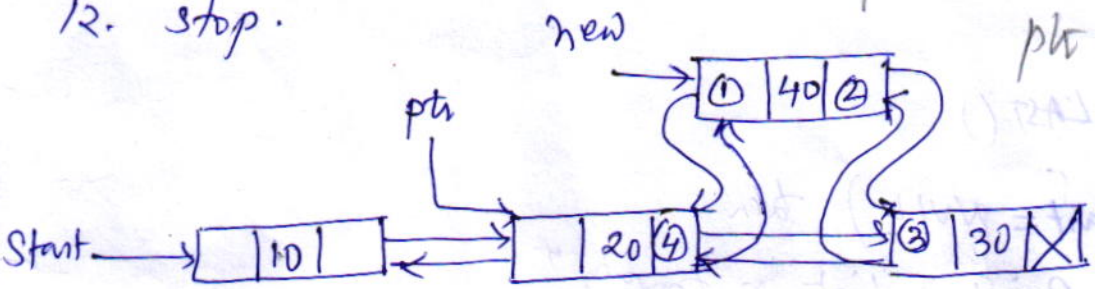
5. ptr = start.
6. Repeat for I = 1 to LOC-2 step1.
    1. ptr = ptr → rlink
7. Endfor.
8. new → llink = ptr.
9. new → rlink = ptr → rlink.
10. ptr → rlink → llink = new.
11. ptr → rlink = new.
12. stop.

new → rlink = ptr → rlink

new → llink = ptr

ptr → rlink → prev = new

ptr → next = new.



DELETE_FRONT()

1. If (start = NULL) then.
    1. Print "List is Empty"
    2. Return.
2. Endif.
3. item = start → data.
4. ptr = start.
5. If (start → rlink = NULL) then.
    1. start = NULL.
6. Else.

1. start = start → rlink.
2. start → llink = NULL.

[start → rlink → llink = NULL].

7. Endif.
8. Free(ptr).
9. Stop.

DELETE_LAST()

1. If (start = NULL) then.
   1. Print "List is Empty"
   2. Return.
2. Endif.
3. ptr = start.
4. If (start → rlink = NULL) then.
   1. item = start → data.
   2. start = NULL.
5. Endif.
6. Repeat while (ptr → rlink ≠ NULL)
   1. ptr = ptr → rlink.
7. Endwhile.

8. item = ptr → data.
9. ptr → llink → rlink = NULL
10. Free (ptr)
11. Stop.

⊛ Circular Linked List.

It is just a single linked list in which the linked field of the last node contains the address of the first node of the list. That is, the link field of the last node does not point to NULL rather it points to the beginning of the linked list.

INSERT_FRONT()

1. new = (NODE *) malloc (sizeof(NODE))
2. If new = NULL Then.
   1. Printf "Memory Underflow"
   2. Return.
3. Endif.
4. If start = NULL Then.
   1. new → data = item.
   2. new → link = new.
   3. start = new.
   4. last = new.
5. Endif.

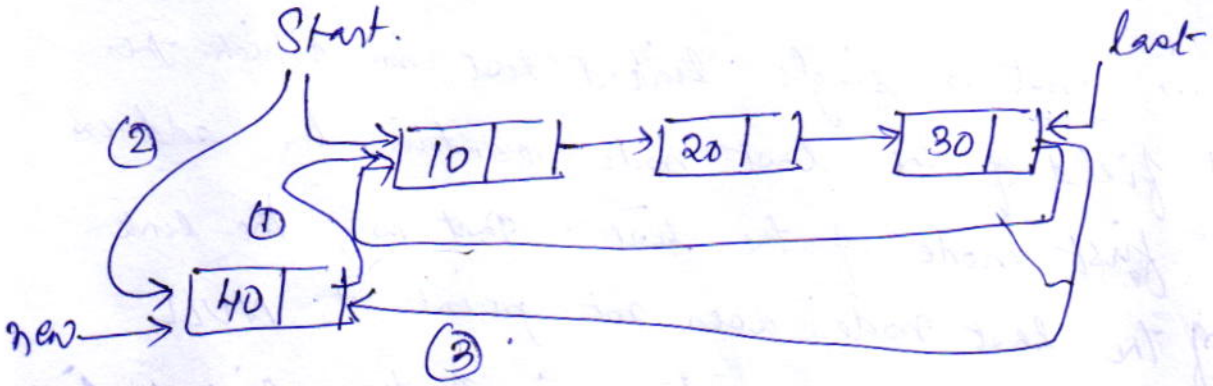
6. new → data = item.

7. new → link = start.

8. Start = new.

9. last → link = new.

10. Stop.



INSERT_LAST ()

1. new = (NODE *) malloc (sizeof (NODE))

2. If new = NULL then.

    1. Print "Memory Underflow"

    2. Return

3. Endif

4. If start = NULL then.

    1. new → data = item

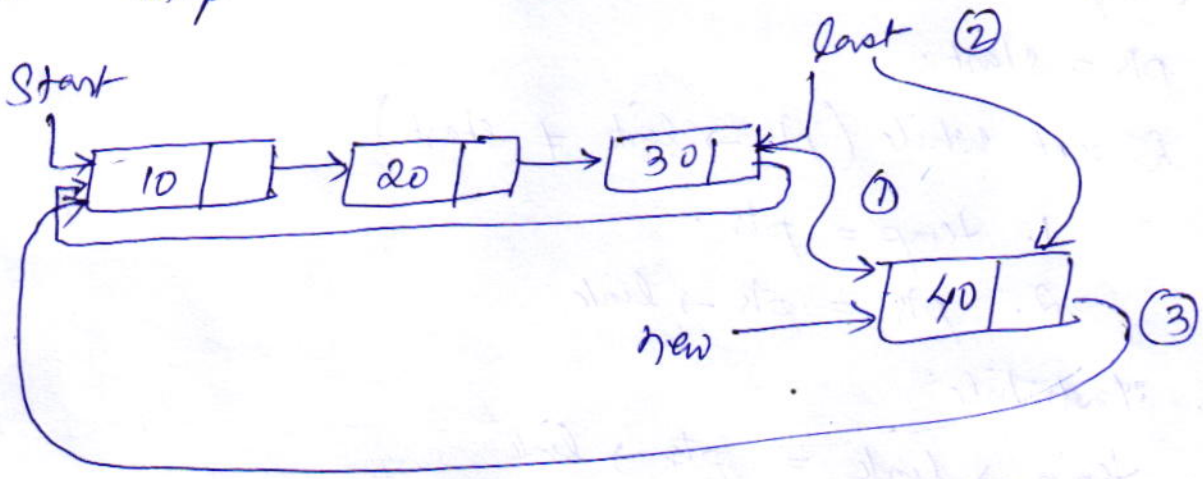
    2. new → link = new.

    3. start = new

    4. last = new.

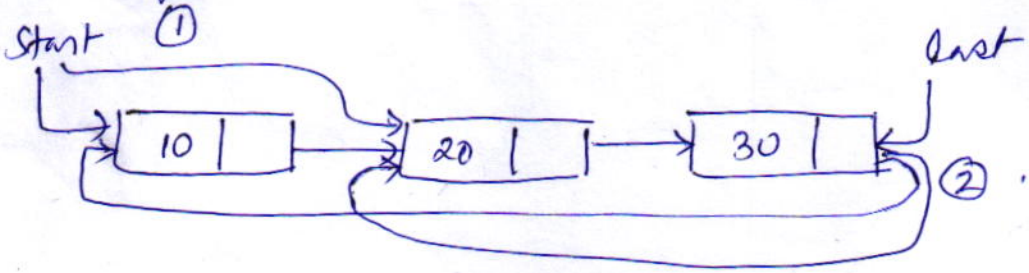
5. Endif

6. new → data = item.
7. last → link = new.
8. last = new.
9. last → link = start.
10. stop.



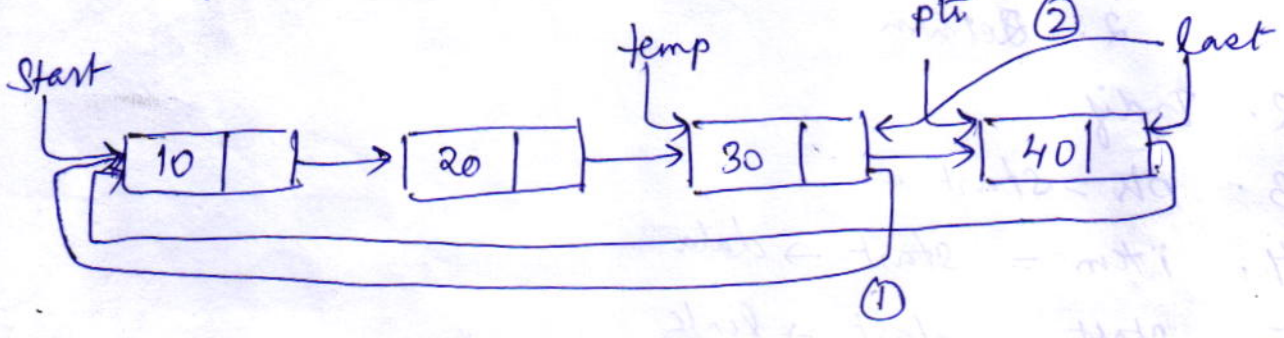
DELETE_FRONT ().

1. If start = NULL then.
   1. Print "List is Empty"
   2. Return.
2. Endif.
3. ptr = start.
4. item = start → data.
5. start = start → link
6. last → link = start.
7. Free (ptr)
8. Stop.



DELETE_LAST()

1. If start = NULL then.
    1. Print "List is Empty"
    2. Return.
2. Endif.
3. ptr = start.
4. Repeat while (ptr → link ≠ start)
    1. temp = ptr.
    2. ptr = ptr → link.
5. Endwhile.
6. temp → link = ptr → link.
7. last = temp.
8. Free (ptr).
9. Stop.



Linked list implementation of Stack and Queue.

Push_LL.

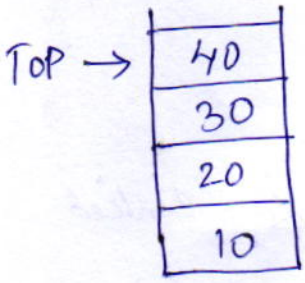
Item is the item to be inserted. A single linked list is used to represent the stack and TOP is a pointer to the first node.

1. new = (NODE *) malloc (sizeof (NODE)).
2. new → data = ITEM.
3. new → link = TOP
4. TOP = new.
5. Stop.

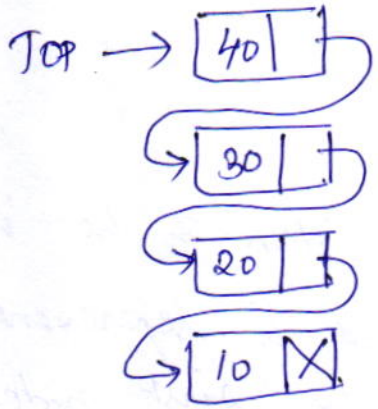
Pop_LL.

1. If TOP = NULL.
   1. Print "Stack is Empty"
   2. Return.
2. Else.
   1. ptr = TOP → link.
   2. ITEM = TOP → data.
   3. TOP = ptr.
3. Endif.
4. Stop.

ptr = TOP.
ITEM = TOP → data.
TOP = TOP → link
Free (ptr).



Array Implementation of Stack.



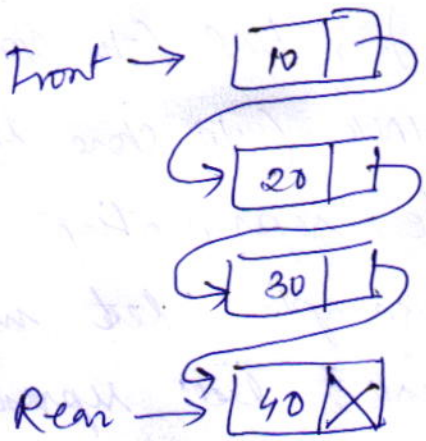
Linked list Implementation of Stack.

Q Insert_LL.

1. new = (NODE *) malloc (sizeof (NODE))
2. new → data = item.
3. new → link = NULL
4. If (front = NULL)
   1. rear = new.
   2. front = new.
5. Else
   1. rear → link = new.
   2. rear = new.
6. Endif.
7. Stop

Q Delete - LL.

1. If (front = NULL) then.
   1. Print "Queue is Empty"
   2. Return
2. Else ptr = front.
   1. item = front → data.
   2. front = front → link.
3. Endif Free (ptr).
4. Stop.



Linked list representation of Queue.

An important application of linked list is to represent and manipulate polynomials. Consider the general form of a polynomial having a single variable:

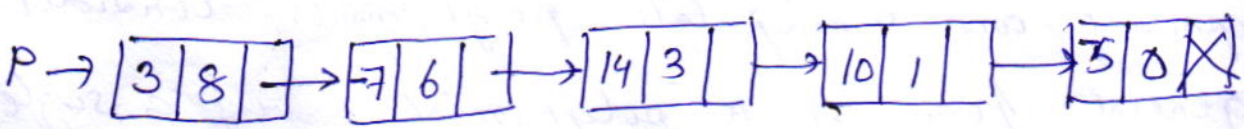
$$P(x) = a_n x^{e_n} + a_{n-1} x^{e_{n-1}} + \dots + a_1 x^{e_1}$$

where $a_i x^{e_i}$ is a term in the polynomial so that a_i is a non-zero coefficient and e_i

is the exponent. We will assume an ordering of the terms in the polynomial such that $e_n > e_{n-1} > \cdots > e_2 > e_1 \geq 0$. The structure of a node in order to represent a term is as shown below:

Considering the single linked list representation a node should have three fields: COEFF (to store the co-efficients a_i), EXP (to store the exponent e_i) and a LINK (to store the pointer to the next node representing the next term). As an example, let us consider the single linked list representation of the polynomial.

$P(x) = 3x^8 - 7x^6 + 14x^3 + 10x - 5$.



⊛ Addition of Two Polynomials.

⊛ Evaluation of Infix expression.

⊛ Insert After a particular item in a single Linked list

1. new = (NODE *) malloc (sizeof(NODE))
2. new → data = new_item
3. ptr = start.
4. while (ptr → data ≠ given_item && ptr ≠ NULL)
    1. ptr = ptr → link
5. Endwhile.
6. If (ptr ≠ NULL)
    1. new → link = ptr → link
    2. ptr → link = new
7. Else
    1. print "Item not found"
8. Endif
9. Stop

(*) Delete after a particular item in a single linked list

1. ptr = start
2. while (ptr → data ≠ item && ptr ≠ NULL)
   1. ptr = ptr → link
3. End while
4. If (ptr ≠ NULL)
   temp = ptr → link
   ptr = ptr → link
   free (temp)
5. Else
   1. Print "Item not found"
6. End if
7. Stop