LINKED LIST

It is a list which links some data elements to one another.

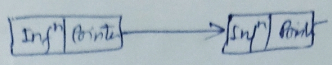
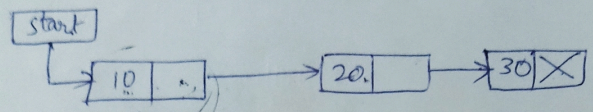


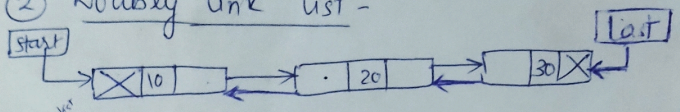
fig: Linked list

Types of Linked List -

① Singly link list -

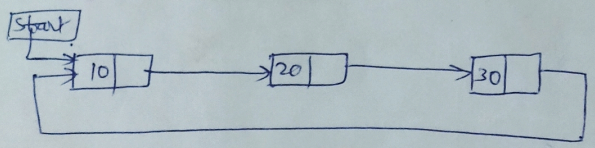


② Doubly link list -

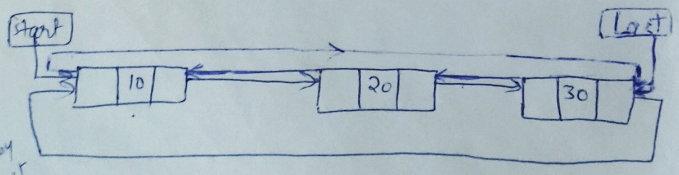


2 way Header list

③ Circular link list -



④ Circular doubly link list -



Two way Header list

to

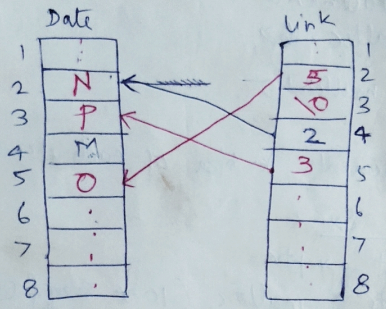
Representation — (Already Taught) & Implementation

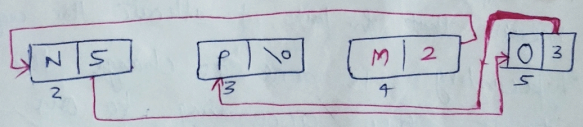
Ex Consider a linked list in memory where each node of the list contains single character.

Soln-

Let
Start = 4
Data[4] = M
Link[4] = 2.
Data[3] = N.
Data[5] = O
Data[3] = P

// next pointer
Link(2) = 5
Link[5] = 3
Link(3) = NULL





Declaration of Linear linked list (to store int nos in memory)

```
struct node
    {
      int a;          // info
      struct node *next   // pointer
    };
typedef struct node NODE;
NODE *start           // starting pointer
```

Operations on linked list (Algo's) —

0. Creation
1. Insertion
2. Deletion
3. Traversing
4. searching

Dynamic Allocation & Deallocation fn.

"malloc" function

Turbo C — <stdlib.h>
<alloc.h>

Unix — <malloc.h>

malloc (no. of elements * size of each element)

for Ex.

```
int *ptr
ptr = malloc (10 * sizeof(int))
```

malloc() returns a void pointer. If it returns a NULL it means that fn fails to allocate the required amount of memory.

Typecasting is done to change the returned pointer type according to our need.

```
ptr = (type_cast *) malloc (size)
```

"Free() Function

It deallocate the previous allocated memory

```
free (ptr);
```

"Realloc()" fn.

It resize the size of memory block, already allocated. → (~~minimize~~ / maximize)

```
ptr = malloc(size);
ptr = realloc(ptr, new_size);
```

"Calloc()" fn.

Same as malloc, except that it requires 2 arguments

```
ptr = (int*) calloc(10, 2)
```

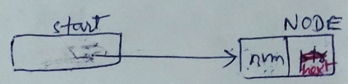
10 → no of elements

2 → size of datatype in byte (int)

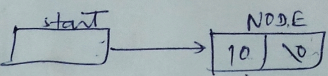
SINGLY LINKED LIST

Creation–

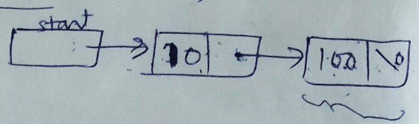
```
struct node
    {
      int num;
      struct node *next;
    };
typedef struct node NODE;
NODE *start;
start = (NODE*) malloc (sizeof (NODE));
```




```
start -> num = 10;
start -> next = '\0';
```




```
start -> next = (NODE *) malloc (sizeof (NODE))
start -> next · num = 100;
start -> next · next = '\0';
```



INSERTION

(A) Inserting a node at the begining:-

```
INSERT (START, ITEM)
① If ptr == NULL then
      Print Overflow
      exit
   else
      ptr = (NODE *) malloc (sizeof (NODE))
   end if
② set ptr → num = ITEM
③ set ptr → next = START
④ set START = ptr
End
```

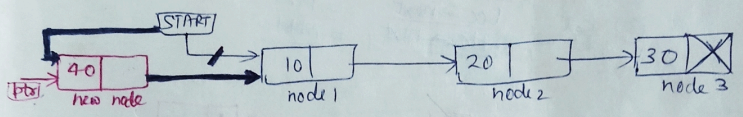


fig: Insertion of 'new node' at the begining

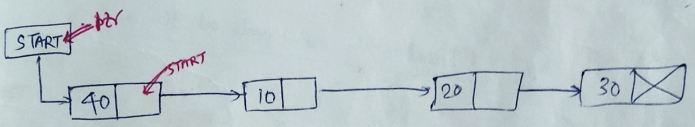


fig: After insertion

(B) Inserting a node at the end :-

INSERT (START, ITEM, LOC)

① If ptr == NULL then
Print Overflow
exit
else
ptr = (NODE *) malloc (sizeof (NODE))
end if

② Set ptr → num = ITEM

③ Set ptr → next = NULL

④ ~~Set Loc = Loc → next~~

⑤ Set Loc → next = ptr
Set ptr → next = NULL

End

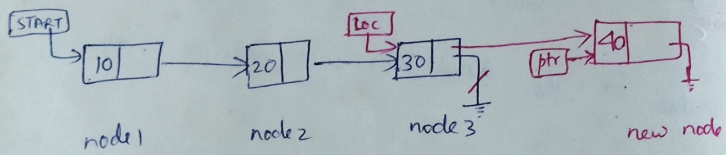


fig: Insertion of new node at the end

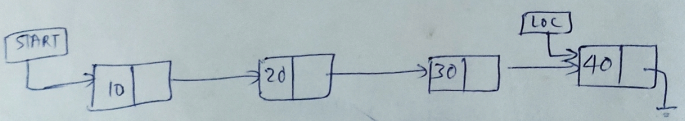


fig: After Insertion

(c) Inserting a node at the specified Position :-

```
INSERT ( START, ITEM, temp )
    ① If ptr == NULL then
            Print Overflow
            exit
       else
            ptr = (NODE *) malloc (sizeof (NODE))
       endif

    ② set ptr → num = ITEM
    ③ set ptr → next  =  temp → next
    ④ set temp → next  =  ptr
End
```

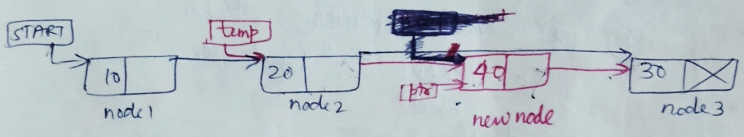


fig: Insertion of new node btw node 2 & 3

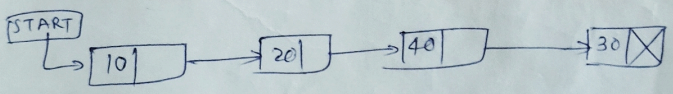
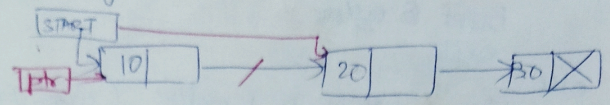


fig: After Insertion

Deletion

Deleting the 1st node :-



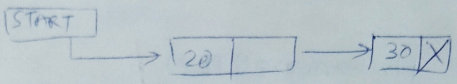


Fig: After Deletion

Delete (START)

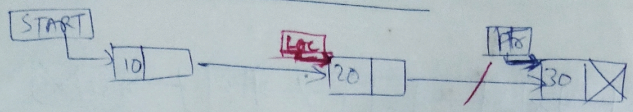
① If START = NULL Then
Print "Underflow"
Exit
end if

② START = ptr → next

③ free (ptr)

End

Deleting the last node :-



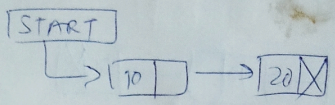


fig: After Deletion

Delete (START, LOC)

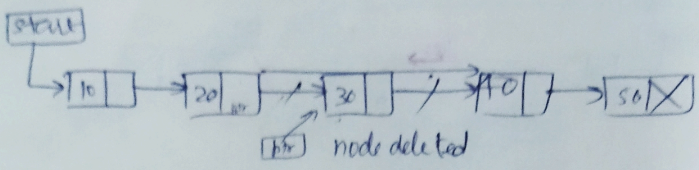
① if START = NULL then
Print "Underflow" // empty linked list
Exit
end if

② set LOC → next = NULL

③ free (ptr)

end

Deleting the node from the specified position.



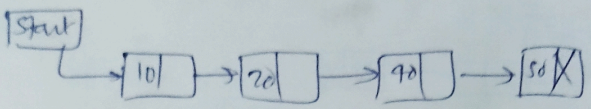


fig: Deletion

Delete (start, temp)

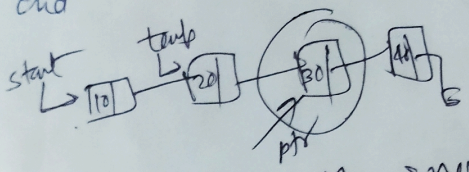
① If START = = NULL then
Print 'Underflow'
Exit

② temp = ptr

③ ptr = ptr → next

④ free (temp)

End



temp → next = ptr → next

free (ptr)

Application of Linked List :—
(Polynomial Representation & addition) —

Representation

Let $P(x)$ is a polynomial —

$$P(x) = a_n x^{m} + a_{n-1} x^{m-1} + \dots a_1 x^{1} + a$$

a_i = nonzero coeff

m_i = exponents

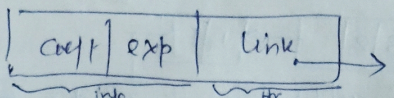
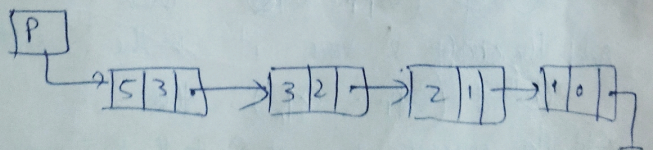


fig: A polynomial node

```
struct polynode
  {
    int coeft;
    int exp;
    struct polynode *ptr;
  };
typedef struct polynode PNODE;
```

Let $P = 5x^3 + 3x^2 + 2x + 1$

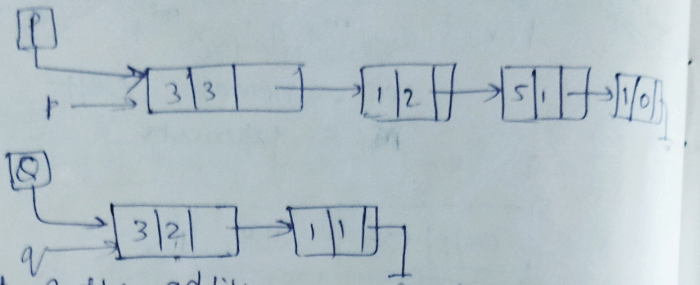


Addition:

Let 2 polyals P & Q are —

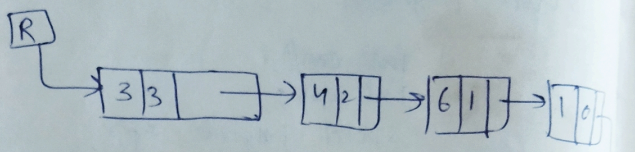
$P = 3x^3 + x^2 + 5x + 1$

$Q = 3x^2 + x$



Resultant of the addition R will be —

$R = 3x^3 + 4x^2 + 6x + 1$



Algo:

Add_poly (exp_p, exp_q, p, q, R) following steps would be followed for polyal addn

1. Read the no of terms, Coeff & exp of I polyal
2. ——————————— II polyal
3. Set the temp pointers p & q to traverse the polynomials
4. Compare the exp of both polyals starting from I node
   - (a) If ($exp_p = exp_q$) then
     Add the coeff & store it in the resultant linked list
   - (b) If ($exp_p < exp_q$) then
     Add the current term of Q to the R & move q to point to next node in Q
   - (c) If ($exp_p > exp_q$) then
     Add the current term of P to the R & move p to point to next node in P
5. Append the remaining nodes of either of the polyals to R

End

Garbage collection :—

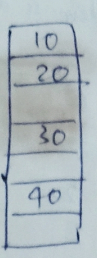
If the memory addresses in RAM are unnecessary engaged & there is no use of them, then it would produce a garbage collection.

To remove this problem, in case of pointer, free function works a lot.

free (ptr)

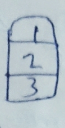
compaction :

(Mainly used at the time to program an Operating system to use the RAM efficiently)



Sahni : Page 179

Now Insert : (contiguous data)



It can't be inserted without compaction, so —

compaction

Already Taught :—

Generalised linked list

Link list in array

Traversing & searching