SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

**SCHOOL OF COMPUTING**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

# UNIT – II-Data Structures – SCSA1203

## UNIT –II

## LINKED LISTS

**Introduction - Singly linked list - Representation of a linked list in memory - Operations on a singly linked list - Merging two singly linked lists into one list - Reversing a singly linked list - Applications of singly linked list to represent polynomial - Advantages and disadvantages of singly linked list - Circular linked list - Doubly linked list - Circular Doubly Linked List.**

### INTRODUCTION

Linked lists and arrays are similar since they both store collections of data. Array is the most common data structure used to store collection of similar elements. Arrays are convenient to declare and provide the easy syntax to access any element by its index number. Once the array is set up, access to any element is convenient and fast. But arrays are suffer from the following limitations:

- Arrays have a fixed dimension. Once the size of an array is decided it cannot be increased or decreased during execution. For example, if we construct an array of 100 elements and then try to stuff more than 100 elements in it, our program may crash. On the other hand, if we use only 10 elements then the space for balance 90 elements goes waste.
- Array elements are always stored in contiguous memory locations. At times it might so happen that enough contiguous locations might not be available for the array that we are trying to create. Even though the total space requirement of the array can be met through a combination of non-contiguous blocks of memory, we would still not be allowed to create the array.
- Operations like insertion of a new element in an array or deletion of an existing element from the array are pretty tedious. This is because during insertion or deletion each element after the specified position has to be shifted one position to the right (in case of insertion) or one position to the left (in case of deletion).

Linked list overcomes all these disadvantages. A linked list can grow and shrink in size during its lifetime. In other words, there is no maximum size of a linked list. The second advantage of linked

lists is that, as node (elements) are stored at different memory locations it hardly happens that we fall short of memory when required. The third advantage is that, unlike arrays, while inserting or deleting the nodes of the linked list, shifting of nodes is not required.

What is a Linked list?

Linked list is a very common data structure often used to store similar data in memory. While the elements of an array occupy contiguous memory locations, those of a linked list are not constrained to be stored in adjacent locations. The individual elements are stored "somewhere" in memory, rather like a family dispersed, but still bound together. The order of the elements is maintained by explicit links between them. **-

- Linked List can be defined as collection of objects called nodes that are randomly stored in the memory.
- A node contains two fields i.e. data and pointer which contains the address of the next node in the memory.
- Linked list requires more memory compared to array because along with value it stores pointer to next node.
- Linked list 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…
- The last node of the list contains pointer to the null.
- Typically, a linked list, in its simplest form looks like the following

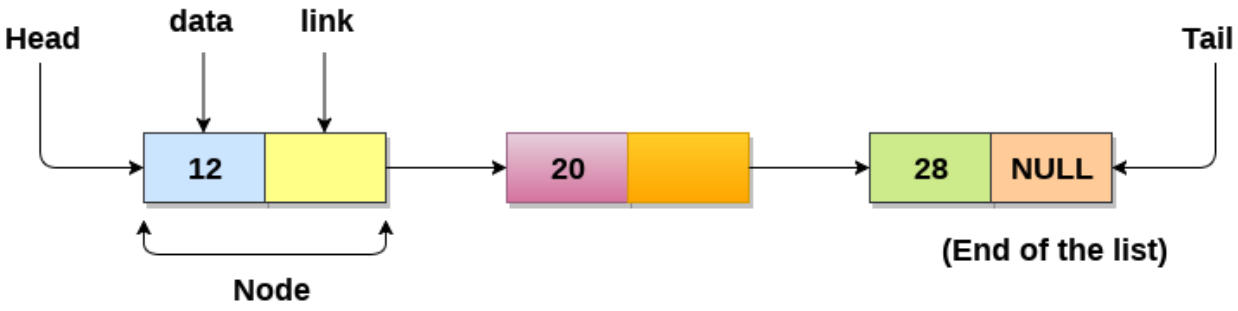



Fig:1 Linked List

Few salient features

- There is a pointer (called header) points the first element (also called node)
- Successive nodes are connected by pointers.
- Last element points to NULL.
- Linked lists have efficient memory utilization. Here, memory is not pre allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
- Linked lists are dynamic data structures. i.e., It can grow or shrink in size during execution of a program.
- Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
- Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node when compared to array.

2. Searching a particular element in list is difficult and also time consuming.

**Defining a Node of a Linked List**

Each structure of the list is called a node, and consists of two fields:

- Item (or) data
- Address or pointer to the next node in the list

How to define a node of a linked list?

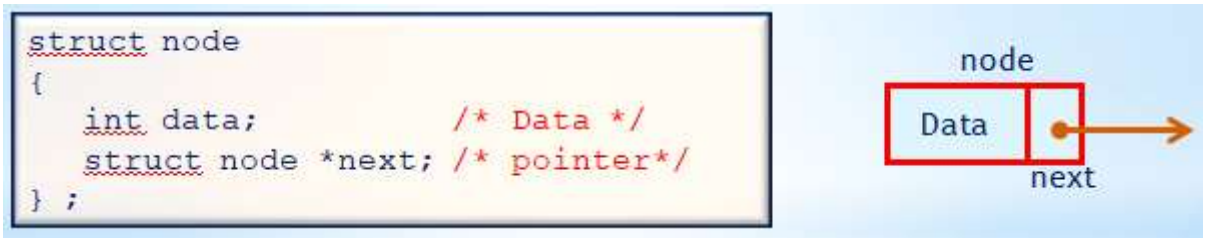



**Fig: 2 Defining a node**

**Note:**

Such structures which contain a member field pointing to the same structure type are called self-referential structures.

**Initialize the head node**

Struct node *head=NULL;

**Memory allocation for a node**

Struct node *newnode=(struct node *)malloc(sizeof(struct node));

**Creation of node**

Steps to create node:

1. Defining a structure of a node
2. Allocate memory dynamically.
3. Read the data into data field
4. Assign null value to its next

struct node

{

int data;

struct node *next; //defining node structure

};

struct node *head=null,*newnode; //initialize the head node

newnode=(struct node *)malloc(sizeof(struct node)) // memory allocation for a newnode

newnode->data =data // Read data

newnode-> next =null;

There are 3 different implementations of Linked List available, they are:

- Singly Linked List
- Doubly Linked List
- Circular Linked List

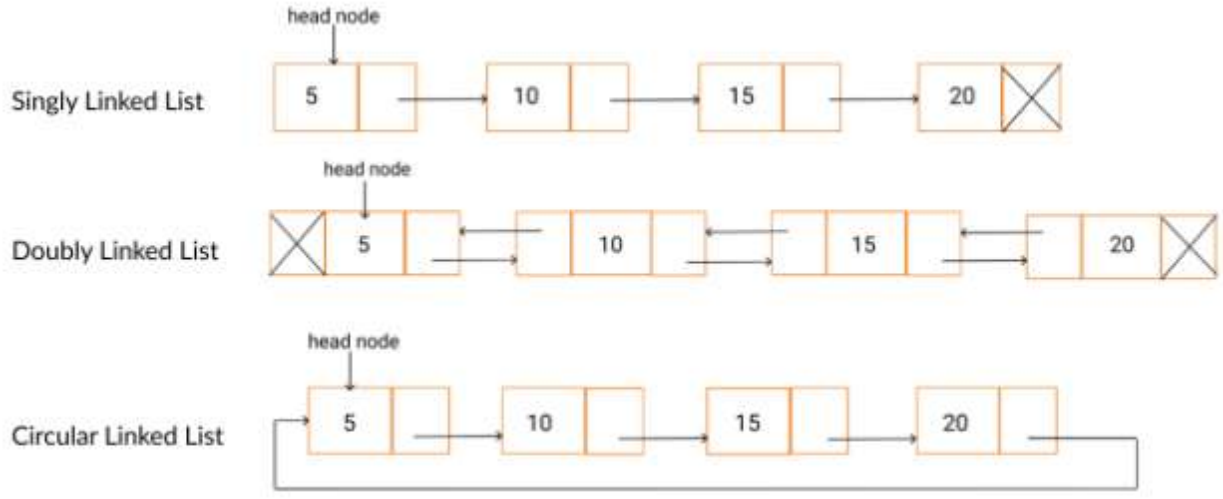



Fig: 3 Types of linked list

**Representation of Linked List in memory**

Linked lists can be represented in memory by using two arrays respectively known as data and next contains information of element and address of next node repectively.

The list also requires a variable name or start which contains address of first node. Pointer field of last node denoted by NULL which indicates the end of list. Eg. Consider a linked list given below. The linked list can be represented in memory as

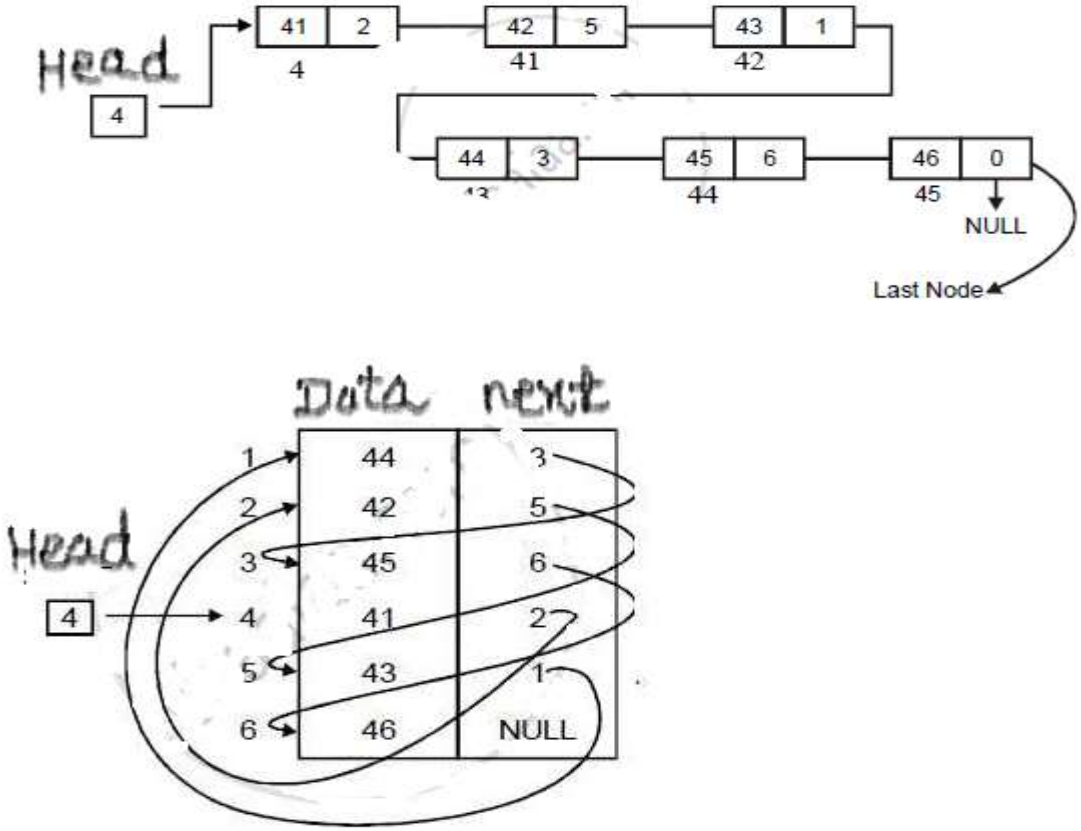



Fig 4: Representation of Linked List

## Singly linked list

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we cannot traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.

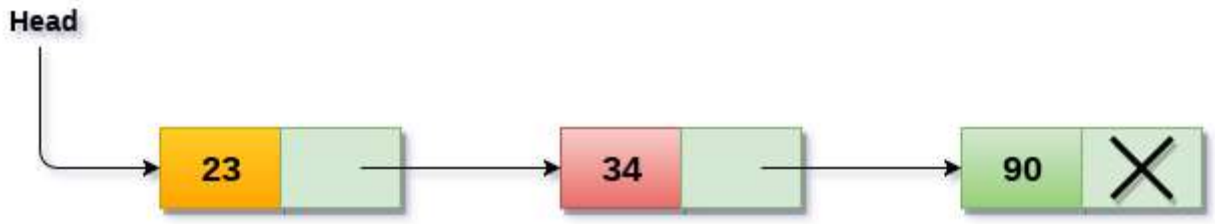



Fig:5 Singly Linked List

In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

## Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

- Traversing the list
- Inserting a node into the list
- Deleting a node from the list
- Merging the linked list with another one to make a larger list
- Searching for an element in the list.

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

**Insertion operation in singly linked list**

Inserting a node into a single linked list There are various positions where a node can be inserted:

| SN | Operation | Description |
|---|---|---|
| 1 | Insertion at beginning | It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list. |
| 2 | Insertion at end of the list | It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario. |
| 3 | Insertion after specified node | It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. . |

(i) Inserting at the front (as a first element)

(ii) Inserting at the end (as a last element)

(iii) Inserting at any other position.

**Inserting a node at the front of a single linked list**

Inserting a new element into a singly linked list at beginning is quite simple. We just need to make a few adjustments in the node links adjustments to make the new node as the first node of the list. There are the following steps which need to be followed in order to insert a new node in the list at beginning.

- Allocate the space for the new node and store data into the data part of the node.
- Make the link part of the new node pointing to the existing first node of the list.
- At the last, we need to make the new node as the first node of the list.

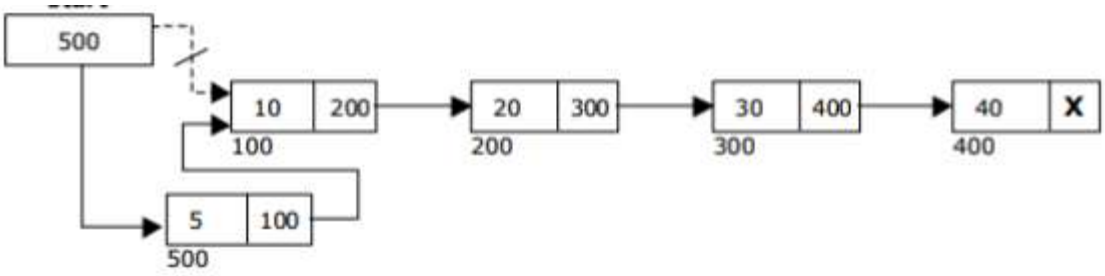



**Fig:5 Inserting a node at the front of a single linked list**

The algorithm lnsertatfront: is used to insert a node at the front of a single linked list.

**Algorithm INSERT_FRONT(head,X)**

**Input:** Head is the pointer to the first node and X is the data of the node to be inserted.

**Output:** A singly linked list with newly inserted node in the front of the list.

1. newnode = create newnode // Create a newnode and store its pointer in newnode
   newnode->data =data // Read data
   newnode-> next =null;
2. if(head!= NULL)
   newnode->next=head
   head=newnode
3. endif
4. stop

**Inserting a node at the end of a single linked list**

- We need to declare a temporary pointer temp in order to traverse through the list. temp is made to point the first node of the list. At the end of the loop, the temp will be pointing to the last node of the list. Now, allocate the space for the new node, and assign the item to its data part. Since, the new node is going to be the last node of the list hence, the next part of this node needs to be pointing to the null. We need to make the next part of the temp node (which is currently the last node of the list) to null .

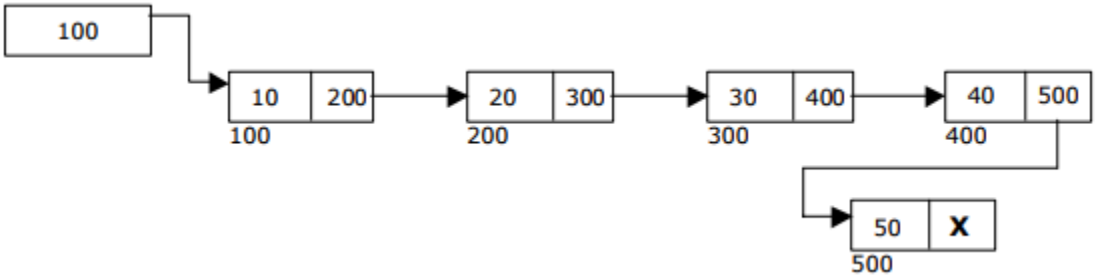



**Fig:6 Inserting a node at the end of a single linked list**

The algorithm Insert_End is used to insert a node at the end of a single linked list

**Algorithm INSERT_END(head,X)**

**Input:** Head is the pointer to the first node and X is the data of the node to be inserted.

**Output:** A singly linked list with newly inserted node at the end of a linked list

1. newnode = create newnode // Create a newnode and store its pointer in newnode
   newnode->data =data // Read data
   newnode-> next =null
2. temp = head

   while (temp -> next != NULL) do

   temp = temp -> next

   Endwhile

   temp->next = newnode

   newnode->next=null

   stop

**Inserting a node into a single linked list at any position in the list**

In order to insert an element after the specified number of nodes into the linked list, we need to skip the desired number of elements in the list to move the pointer at the position after which the node will be inserted.

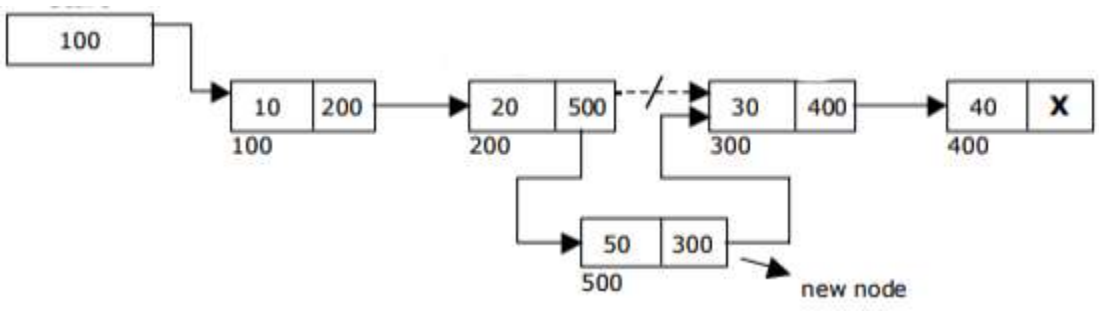



**Fig: 7 Inserting a node into a single linked list at any position in the list**

The algorithm InsertAnyPosition is used to insert a node into a single linked list at any position in the list.

Algorithm INSERT_ANYPOSITION(Head,X,pos)

1. temp=head;

   while(i<pos)

   temp=temp->next;

   i++

   Endwhile

2. newnode = create newnode // Create a newnode and store its pointer in newnode
   newnode->data =data // Read data
   newnode->next=temp->next

   temp->next=newnode

**Deletion Operation in Singly linked list**

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

| SN | Operation | Description |
|---|---|---|
| 1 | Deletion at beginning | It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers. |
| 2 | Deletion at the end of the list | It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios. |
| 3 | Deletion after specified node | It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list. |
| 4 | Traversing | In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list. |

**Deletion in singly linked list at beginning**

Deleting a node from the beginning of the list is the simplest operation of all. It just need a few adjustments in the node pointers. Since the first node of the list is to be deleted, therefore, we just need to make the head, point to the next of the head. This will be done by using the following statements.

```
temp=head;
```

head=head->next;

Now, free the pointer ptr which was pointing to the head node of the list. This will be done by using the following statement.

Free(temp)

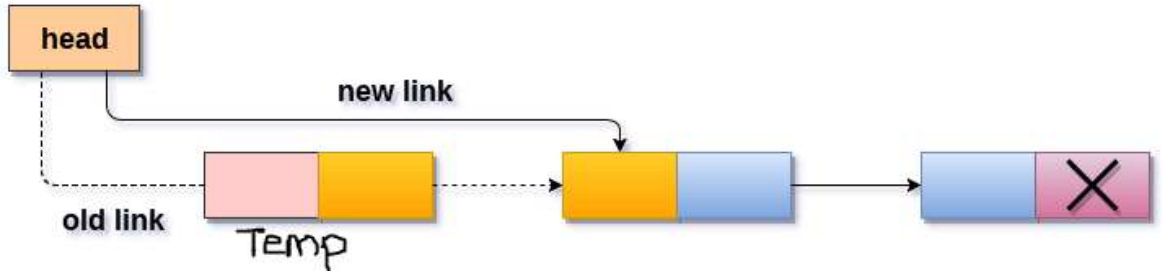



Fig 8: Deleting a node from the beginning

Algorithm DELETE_FRONT(Head,temp)

- **Step 1:** IF HEAD = NULL

  Write UNDERFLOW
  Go to Step 5
  [END OF IF]

- **Step 2:** SET TEMP = HEAD
- **Step 3:** SET HEAD = HEAD -> NEXT
- **Step 4:** FREE TEMP
- **Step 5:** EXIT

**Deletion in singly linked list at the end**

There are two scenarios in which, a node is deleted from the end of the linked list.

1. There is only one node in the list and that needs to be deleted.
2. There are more than one node in the list and the last node of the list will be deleted.

In the first scenario,

the condition head → next = NULL will survive and therefore, the only node head of the list will be assigned to null. This will be done by using the following statements.

temp = head

head = NULL

free(ptr)

In the second scenario,

The condition head → next = NULL would fail and therefore, we have to traverse the node in order to reach the last node of the list.

For this purpose, just declare a temporary pointer temp and assign it to head of the list. We also need to keep track of the second last node of the list. For this purpose, two pointers temp and ptr1 will be used where temp will point to the last node and ptr1 will point to the second last node of the list.

this all will be done by using the following statements.

```
temp = head;

while(temp->next != NULL)

{
```

```
    ptr1 = temp;

    temp = temp ->next;

}
```

Now, we just need to make the pointer ptr1 point to the NULL and the last node of the list that is pointed by temp will become free. It will be done by using the following statements.

```
temp->next = NULL;

free(ptr);
```

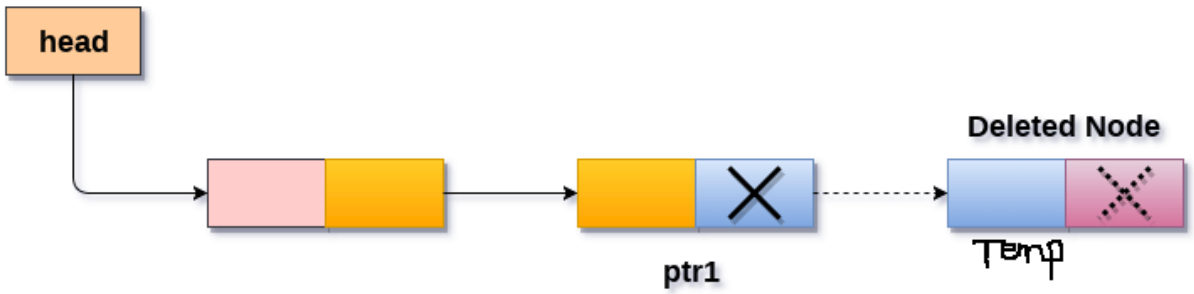



**Fig 9 Deletion in singly linked list at the end**

**Algorithm DELETE_END(Head,Temp,ptr1)**

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 8

[END OF IF]

Step 2: SET TEMP = HEAD

Step 3: Repeat Steps 4 and 5 while TEMP -> NEXT!= NULL

Step 4: SET PRETEMP = TEMP

Step 5: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 6: SET PRETEMP -> NEXT = NULL

Step 7: FREE TEMP

Step 8: EXIT

**Deletion in singly linked list after the specified node :**

In order to delete the node, which is present after the specified node, we need to skip the desired number of nodes to reach the node after which the node will be deleted. We need to keep track of the two nodes. The one which is to be deleted the other one if the node which is present before that node. For this purpose, two pointers are used: temp and ptr1.

Use the following statements to do so.

```
temp=head;
    for(i=0;i<loc;i++)
    {
       ptr1 = temp;
       temp = temp->next;
```

```
        if(temp == NULL)
        {
            printf("\nThere are less than %d elements in the list..",loc);
            return;
        }
    }
```

Now, our task is almost done, we just need to make a few pointer adjustments. Make the next of ptr1 (points to the specified node) point to the next of ptr (the node which is to be deleted).

This will be done by using the following statements.

```
ptr1 ->next = temp ->next;
    free(temp);
```

Algorithm DELETE_POS(Head,Temp,Ptr1)

STEP 1: IF HEAD = NULL

WRITE UNDERFLOW

GOTO STEP 10

END OF IF

STEP 2: SET TEMP = HEAD

STEP 3: SET I = 0

STEP 4: REPEAT STEP 5 TO 8 UNTIL I

STEP 5: PTR1 = TEMP

STEP 6: TEMP = TEMP → NEXT

STEP 7: IF TEMP = NULL

WRITE "DESIRED NODE NOT PRESENT"

GOTO STEP 12

END OF IF

STEP 8: I = I+1

END OF LOOP

STEP 9: PTR1 → NEXT = TEMP → NEXT

STEP 10: FREE TEMP

**Traversing in singly linked list**

Traversing is the most common operation that is performed in almost every scenario of singly linked list. Traversing means visiting each node of the list once in order to perform some operation on that. This will be done by using the following statements.

```
temp = head;
      while (temp!=NULL)
         {
            temp = temp -> next;
         }
```

Algorithm TRAVERSING()

STEP 1: SET TEMP = HEAD

STEP 2: IF TEMP = NULL

WRITE "EMPTY LIST"

GOTO STEP 7

END OF IF

STEP 4: REPEAT STEP 5 AND 6 UNTIL TEMP != NULL

STEP 5: PRINT TEMP→ DATA

STEP 6: TEMP = TEMP → NEXT

[END OF LOOP]

STEP 7: EXITMP

STEP 11: EXIT

Consider two singly linked list are sorted in increasing order, and merges the two together into one list which is in increasing order. SortedMerge() should return the new list. The new list should be made by splicing together the nodes of the first two lists.

For example: if L1 = 1 -> 3 -> 10 and L2 = 5 -> 6 -> 9 then your program should output the linked list

1 -> 3 -> 5 -> 6 -> 9 -> 10.

Algorithm

The algorithm for this question is quite simple since the two linked lists are already sorted. We create a new linked list and loop through both lists appending the smaller nodes. We'll be using some code that we used in a previous linked list interview question.

Algorithm:

(1) Create a new head pointer to an empty linked list.

(2) Check the first value of both linked lists.

(3) Whichever node from L1 or L2 is smaller, append it to the new list and move the pointer to the next node.

(4) Continue this process until you reach the end of a linked list.

Example

L1 = 1 -> 3 -> 10

L2 = 5 -> 6 -> 9

L3 = null

Compare the first two nodes in both linked lists: (1, 5), 1 is smaller so add it to the new linked list and move the pointer in L1.

L1 = 3 -> 10

L2 = 5 -> 6 -> 9

L3 = 1

Compare the first two nodes in both linked lists: (3, 5), 3 is smaller so add it to the new linked list and move the pointer in L1.

L1 = 10

L2 = 5 -> 6 -> 9

L3 = 1 -> 3

Compare the first two nodes in both linked lists: (10, 5), 5 is smaller so add it to the new linked list and move the pointer in L2.

L1 = 10

L2 = 6 -> 9

L3 = 1 -> 3 -> 5

Compare the first two nodes in both linked lists: (10, 6), 6 is smaller so add it to the new linked list and move the pointer in L2.

L1 = 10

L2 = 9

L3 = 1 -> 3 -> 5 -> 6

Compare the first two nodes in both linked lists: (10, 9), 9 is smaller so add it to the new linked list and move the pointer in L2.

L1 = 10

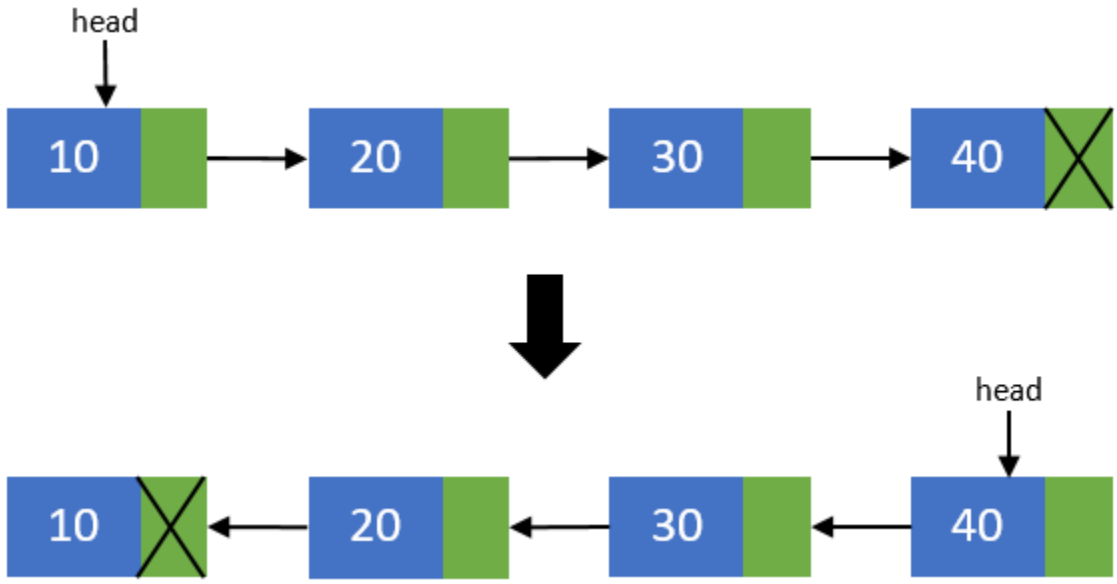
L2 = null

L3 = 1 -> 3 -> 5 -> 6 -> 9

Because L2 points to null, simply append the rest of the nodes from L1 and we have our merged linked list.

L3 = 1 -> 3 -> 5 -> 6 -> 9 -> 10

**Reverse a Singly Linked List**




Algorithm to reverse a Singly Linked List

Begin:

If (head != NULL) then

prevNode ← head

head ← head.next

curNode ← head

prevNode.next ← NULL

While (head != NULL) do

head ← head.next

curNode.next ← prevNode

```
        prevNode ← curNode

        curNode ← head

      End while

      head ← prevNode

   End if

End
```

**Application of singly linked list to represent polynomial**

**What is Polynomial?**

A polynomial p(x) is the expression in variable x which is in the form (axn + bxn-1 + …. + jx+ k), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

one is the coefficient

other is the exponent

Example

10x2 + 26x, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

1. The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
2. Additional terms having equal exponent is possible one

3. The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent

Polynomial can be represented in the various ways. These are:

1. By the use of arrays
2. By the use of Linked List

Polynomial can be represented in the various ways. These are:

- By the use of arrays
- By the use of Linked List

A polynomial can be thought of as an ordered list of non zero terms. Each non zero term is a two-tuple which holds two pieces of information:

The exponent part

The coefficient part

Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients who have same variable powers.

Example:

Input:

1st number = 5x2 + 4x1 + 2x0

2nd number = -5x1 - 5x0

Output:

5x2-1x1-3x0

Input:

1st number = 5x3 + 4x2 + 2x0

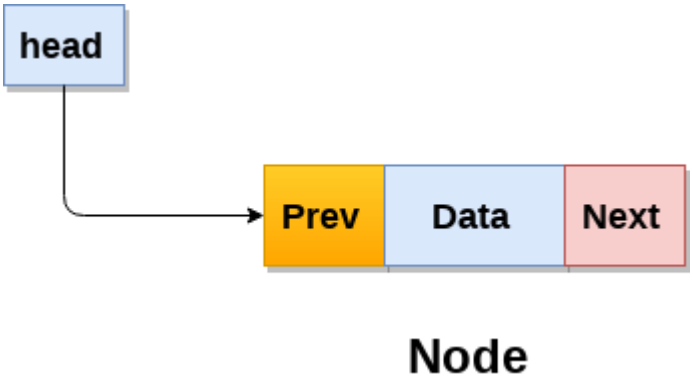
2nd number = 5x^1 - 5x^0

Output:

5x3 + 4x2 + 5x1 - 3x0

**Doubly linked list**

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.




A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.

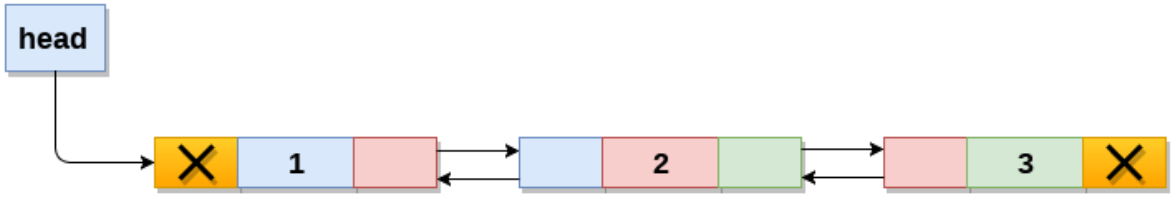



Fig: Doubly linked list

In C, structure of a node in doubly linked list can be given as :

```
struct node

{

    struct node *prev;
```

```
    int data;

    struct node *next;

}
```

The prev part of the first node and the next part of the last node will always contain null indicating end in each direction.

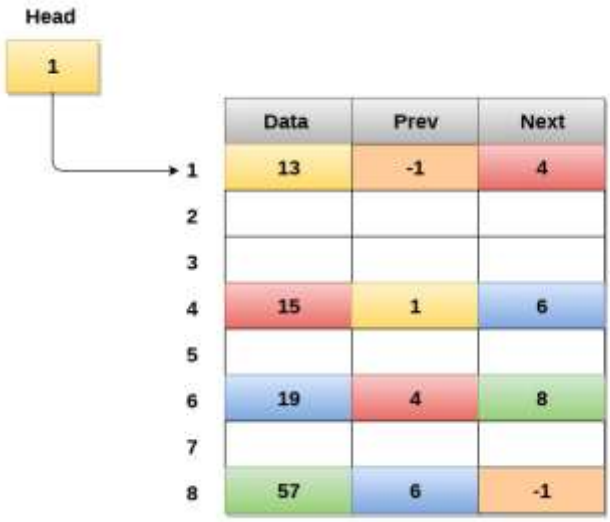
In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

**Memory Representation of a doubly linked list**

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the prev of the list contains null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.




**Operations on doubly linked list**

Node Creation

```
struct node
{
    struct node *prev;
    int data;
    struct node *next;
};
struct node *head;
```

All the remaining operations regarding doubly linked list are described in the following table.

| SN | Operation | Description |
|---|---|---|

| | | |
|---|---|---|
| 1 | Insertion at beginning | Adding the node into the linked list at beginning. |
| 2 | Insertion at end | Adding the node into the linked list to the end. |
| 3 | Insertion after specified node | Adding the node into the linked list after the specified node. |
| 4 | Deletion at beginning | Removing the node from beginning of the list |
| 5 | Deletion at the end | Removing the node from end of the list. |
| 6 | Deletion of the node having given data | Removing the node which is present just after the node containing the given data. |
| 7 | Traversing | Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc. |

**Insertion in doubly linked list at beginning**

As in doubly linked list, each node of the list contain double pointers therefore we have to maintain more number of pointers in doubly linked list as compare to singly linked list.

There are two scenarios of inserting any element into doubly linked list. Either the list is empty or it contains at least one element. Perform the following steps to insert a node in doubly linked list at beginning.

1. Allocate the space for the new node in the memory.
2. Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

newnode->next = NULL;

newnode>prev=NULL;

newnode->data=item;

head=newnode;

In the second scenario, the condition head == NULL become false and the node will be inserted in beginning. The next pointer of the node will point to the existing head pointer of the node. The prev pointer of the existing head will point to the new node being inserted.

This will be done by using the following statements.

newnode->next = head;

head→prev=newnode;

Since, the node being inserted is the first node of the list and therefore it must contain NULL in its prev pointer. Hence assign null to its previous part and make the head point to this node.

newnode→prev =NULL

head = newnode

Algorithm :

Step 1: IF NEWNODE = NULL

Write OVERFLOW

Go to Step 9

[END OF IF]

Step 2: SET NEW_NODE = TEMP

Step 3: SET TEMP = TEMP -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> PREV = NULL

Step 6: SET NEW_NODE -> NEXT = START

Step 7: SET head -> PREV = NEW_NODE

Step 8: SET head = NEW_NODE

Step 9: EXIT

**Insertion in doubly linked list at the end**

In order to insert a node in doubly linked list at the end, we must make sure whether the list is empty or it contains any element. Use the following steps in order to insert the node in doubly linked list at the end.

1. Allocate the memory for the new node. Make the pointer ptr point to the new node being inserted.
2. Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

```
newnode->next = NULL;

newnode ->prev=NULL;

newnode ->data=item;
```

head= newnode;

In the second scenario, the condition head == NULL become false. The new node will be inserted as the last node of the list. For this purpose, we have to traverse the whole list in order to reach the last node of the list. Initialize the pointer temp to head and traverse the list by using this pointer.

Temp = head;

while (temp != NULL)

{

temp = temp → next;

}

the pointer temp point to the last node at the end of this while loop. Now, we just need to make a few pointer adjustments to insert the new node ptr to the list. First, make the next pointer of temp point to the new node being inserted i.e. newnode.

temp→next =newnode;

make the previous pointer of the node ptr point to the existing last node of the list i.e. temp.

newnode → prev = temp;

make the next pointer of the node ptr point to the null as it will be the new last node of the list.

newnode → next = NULL

Algorithm INSERT_END(Head,Temp,Newnode)

Step 1: IF PTR = NULL

Write OVERFLOW

Go to Step 11

[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> NEXT = NULL

Step 6: SET TEMP = START

Step 7: Repeat Step 8 while TEMP -> NEXT != NULL

Step 8: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 9: SET TEMP -> NEXT = NEW_NODE

Step 10C: SET NEW_NODE -> PREV = TEMP

Step 11: EXIT

**Insertion in doubly linked list after Specified node**

In order to insert a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required.

Use the following steps for this purpose.

1. Allocate the memory for the new node.

2. Traverse the list by using the pointer temp to skip the required number of nodes in order to reach the specified node.

```
temp=head;
for(i=0;i<loc;i++)
{
   temp = temp->next;
   if(temp == NULL)  //up to mentioned location
   {
        return;
   }
}
```

The temp would point to the specified node at the end of the for loop. The new node needs to be inserted after this node therefore we need to make a few pointer adjustments here. Make the next pointer of ptr point to the next node of temp.

ptr → next = temp → next;

make the prev of the new node ptr point to temp.

ptr → prev = temp;

make the next pointer of temp point to the new node ptr.

temp → next = ptr;

make the previous pointer of the next node of temp point to the new node.

temp → next → prev = ptr;

**Algorithm INSERT_POS()**

Step 1: IF PTR = NULL

Write OVERFLOW

Go to Step 15

[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET TEMP = START

Step 6: SET I = 0

Step 7: REPEAT 8 to 10 until I

Step 8: SET TEMP = TEMP -> NEXT

STEP 9: IF TEMP = NULL

STEP 10: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"

GOTO STEP 15

[END OF IF]

[END OF LOOP]

Step 11: SET NEW_NODE -> NEXT = TEMP -> NEXT

Step 12: SET NEW_NODE -> PREV = TEMP

Step 13 : SET TEMP -> NEXT = NEW_NODE

Step 14: SET TEMP -> NEXT -> PREV = NEW_NODE

Step 15: EXIT

**Deletion at beginning**

Deletion in doubly linked list at the beginning is the simplest operation. We just need to copy the head pointer to pointer ptr and shift the head pointer to its next.

Ptr = head;

head = head → next;

now make the prev of this new head node point to NULL. This will be done by using the following statements.

head → prev = NULL

Now free the pointer ptr by using the free function.

free(ptr)

**Algorithm**

STEP 1: IF HEAD = NULL

WRITE UNDERFLOW

GOTO STEP 6

STEP 2: SET PTR = HEAD

STEP 3: SET HEAD = HEAD → NEXT

STEP 4: SET HEAD → PREV = NULL

STEP 5: FREE PTR

STEP 6: EXIT

**Deletion in doubly linked list at the end**

Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.

In order to delete the last node of the list, we need to follow the following steps.

If the list is already empty then the condition head == NULL will become true and therefore the operation can not be carried on.

If there is only one node in the list then the condition head → next == NULL become true. In this case, we just need to assign the head of the list to NULL and free head in order to completely delete the list.

Otherwise, just traverse the list to reach the last node of the list. This will be done by using the following statements.

```
ptr = head;
    if(ptr->next != NULL)
    {
       ptr = ptr -> next;
```

}

The ptr would point to the last node of the ist at the end of the for loop. Just make the next pointer of the previous node of ptr to NULL.

ptr → prev → next = NULL

free the pointer as this the node which is to be deleted.

free(ptr)

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 7

[END OF IF]

Step 2: SET TEMP = HEAD

Step 3: REPEAT STEP 4 WHILE TEMP->NEXT != NULL

Step 4: SET TEMP = TEMP->NEXT

[END OF LOOP]

Step 5: SET TEMP ->PREV-> NEXT = NULL

Step 6: FREE TEMP

Step 7: EXIT

**Deletion in doubly linked list after the specified node**

In order to delete the node after the specified data, we need to perform the following steps.

Copy the head pointer into a temporary pointer temp.

```
temp = head
```

Traverse the list until we find the desired data value.

```
while(temp -> data != val)

temp = temp -> next;
```

Check if this is the last node of the list. If it is so then we can't perform deletion.

```
if(temp -> next == NULL)
   {
return;
   }
```

Check if the node which is to be deleted, is the last node of the list, if it so then we have to make the next pointer of this node point to null so that it can be the new last node of the list.

```
if(temp -> next -> next == NULL)
   {
      temp ->next = NULL;
   }
```

Otherwise, make the pointer ptr point to the node which is to be deleted. Make the next of temp point to the next of ptr. Make the previous of next node of ptr point to temp. free the ptr.

```
ptr = temp -> next;
        temp -> next = ptr -> next;
        ptr -> next -> prev = temp;
        free(ptr);
```

Algorithm

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 9

[END OF IF]

Step 2: SET TEMP = HEAD

Step 3: Repeat Step 4 while TEMP -> DATA != ITEM

Step 4: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 5: SET PTR = TEMP -> NEXT

Step 6: SET TEMP -> NEXT = PTR -> NEXT

Step 7: SET PTR -> NEXT -> PREV = TEMP

Step 8: FREE PTR

Step 9: EXIT Traversing in doubly linked list

Traversing is the most common operation in case of each data structure. For this purpose, copy the head pointer in any of the temporary pointer ptr.

Ptr = head

then, traverse through the list by using while loop. Keep shifting value of pointer variable ptr until we find the last node. The last node contains null in its next part.

```
while(ptr != NULL)
    {
       printf("%d\n",ptr->data);
       ptr=ptr->next;
    }
```

Although, traversing means visiting each node of the list once to perform some specific operation. Here, we are printing the data associated with each node of the list.

Algorithm

Step 1: IF HEAD == NULL

WRITE "UNDERFLOW"

GOTO STEP 6

[END OF IF]

Step 2: Set PTR = HEAD

Step 3: Repeat step 4 and 5 while PTR != NULL

Step 4: Write PTR → data

Step 5: PTR = PTR → next

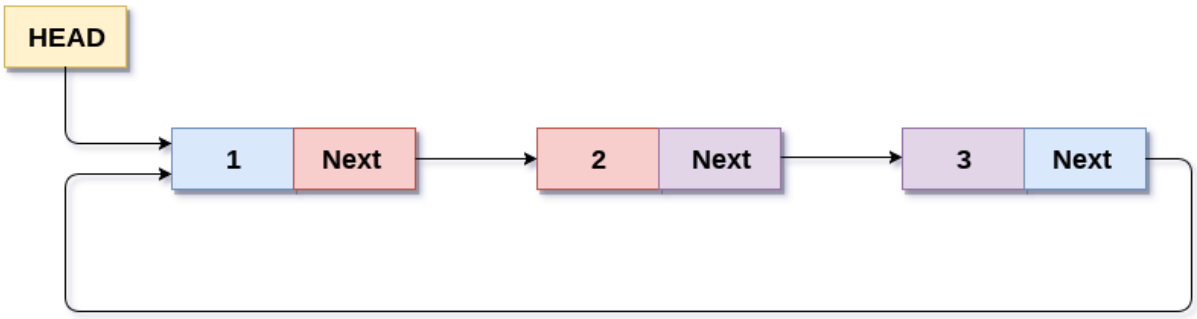
Step 6: Exit

Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.

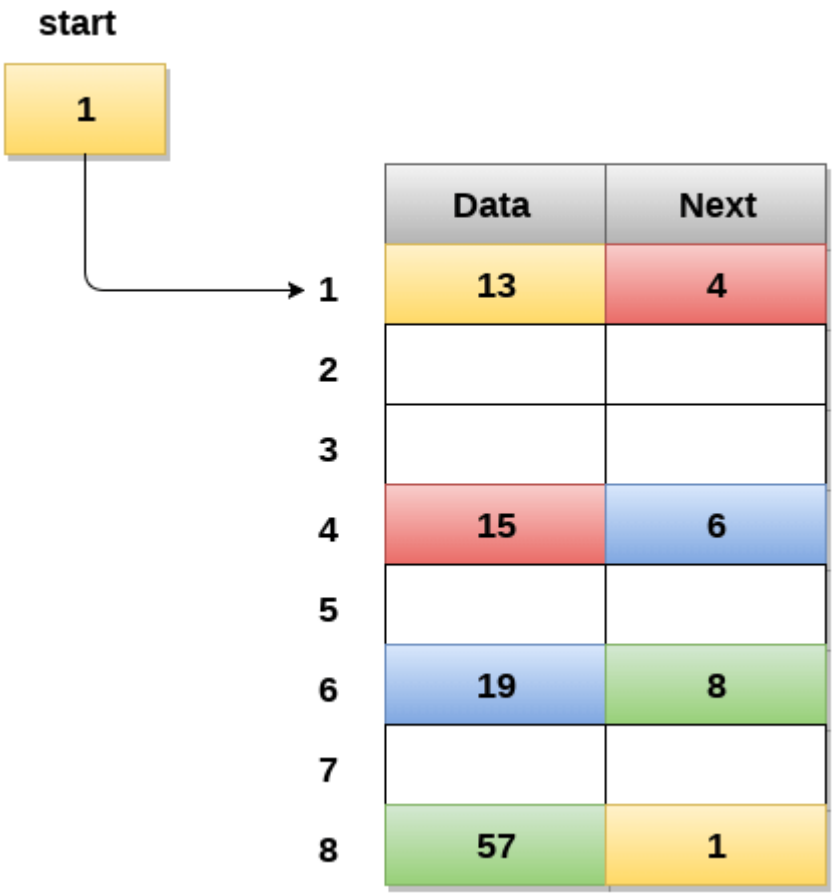



Circular Singly Linked List

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Memory Representation of circular linked list:

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.




However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.

Circular Singly Linked List

We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can find out the number of iterations which need to be performed while traversing the list.

**Operations on Circular Singly linked list:**

## Insertion

| SN | Operation | Description |
|---|---|---|
| 1 | Insertion at beginning | Adding a node into circular singly linked list at the beginning. |
| 2 | Insertion at the end | Adding a node into circular singly linked list at the end. |

Insertion into circular singly linked list at beginning

There are two scenario in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

Firstly, allocate the memory space for the new node

In the first scenario, the condition head == NULL will be true. Since, the list in which, we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only. We also need to make the head pointer point to this node. This will be done by using the following statements.

```
if(head == NULL)
    {
       head = ptr;
       ptr -> next = head;
    }
```

In the second scenario, the condition head == NULL will become false which means that the list contains at least one node. In this case, we need to traverse the list in order to reach the last node of the list. This will be done by using the following statement.

```
temp = head;
       while(temp->next != head)
          temp = temp->next;
```

At the end of the loop, the pointer temp would point to the last node of the list. Since, in a circular singly linked list, the last node of the list contains a pointer to the first node of the list. Therefore, we need to make the next pointer of the last node point to the head node of the list and the new node which is being inserted into the list will be the new head node of the list therefore the next pointer of temp will point to the new node ptr.

This will be done by using the following statements.

```
temp -> next = ptr;
```

the next pointer of temp will point to the existing head node of the list.

ptr->next = head;

Now, make the new node ptr, the new head node of the circular singly linked list.

head = ptr;

in this way, the node ptr has been inserted into the circular singly linked list at beginning.

Algorithm

Step 1: IF PTR = NULL

Write OVERFLOW

Go to Step 11

[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET TEMP = HEAD

Step 6: Repeat Step 8 while TEMP -> NEXT != HEAD

Step 7: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 8: SET NEW_NODE -> NEXT = HEAD

Step 9: SET TEMP → NEXT = NEW_NODE

Step 10: SET HEAD = NEW_NODE

Step 11: EXIT

Insertion into circular singly linked list at the end

There are two scenario in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

Algorithm

Step 1: IF PTR = NULL

Write OVERFLOW

Go to Step 1

[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> NEXT = HEAD

Step 6: SET TEMP = HEAD

Step 7: Repeat Step 8 while TEMP -> NEXT != HEAD

Step 8: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 9: SET TEMP -> NEXT = NEW_NODE

Step 10: EXIT

**Deletion & Traversing**

| N | Operation | Description |
|---|---|---|
| 1 | Deletion at beginning | Removing the node from circular singly linked list at the beginning. |
| 2 | Deletion at the end | Removing the node from circular singly linked list at the end. |
| 4 | Traversing | Visiting each element of the list at least once in order to perform some specific operation. |

Deletion in circular singly linked list at beginning

In order to delete a node in circular singly linked list, we need to make a few pointer adjustments.

There are three scenarios of deleting a node from circular singly linked list at beginning.

Scenario 1: (The list is Empty)

If the list is empty then the condition head == NULL will become true, in this case, we just need to print underflow on the screen and make exit.

```
if(head == NULL)
   {
      printf("\nUNDERFLOW");
      return;
   }
```

Scenario 2: (The list contains single node)

If the list contains single node then, the condition head → next == head will become true. In this case, we need to delete the entire list and make the head pointer free. This will be done by using the following statements.

```
if(head->next == head)
{
   head = NULL;
   free(head);
}
```

Scenario 3: (The list contains more than one node)

If the list contains more than one node then, in that case, we need to traverse the list by using the pointer ptr to reach the last node of the list. This will be done by using the following statements.

```
ptr = head;
 while(ptr -> next != head)
    ptr = ptr -> next;
```

At the end of the loop, the pointer ptr point to the last node of the list. Since, the last node of the list points to the head node of the list. Therefore this will be changed as now, the last node of the list will point to the next of the head node.

```
ptr->next = head->next;
```

Now, free the head pointer by using the free() method in C language.

```
free(head);
```

Make the node pointed by the next of the last node, the new head of the list.

```
head = ptr->next;
```

In this way, the node will be deleted from the circular singly linked list from the beginning.

Algorithm

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 8

[END OF IF]

Step 2: SET PTR = HEAD

Step 3: Repeat Step 4 while PTR → NEXT != HEAD

Step 4: SET PTR = PTR → next

[END OF LOOP]

Step 5: SET PTR → NEXT = HEAD → NEXT

Step 6: FREE HEAD

Step 7: SET HEAD = PTR → NEXT

Step 8: EXIT

Deletion in Circular singly linked list at the end

There are three scenarios of deleting a node in circular singly linked list at the end.

Scenario 1 (the list is empty)

If the list is empty then the condition head == NULL will become true, in this case, we just need to print underflow on the screen and make exit.

```
if(head == NULL)
   {
      printf("\nUNDERFLOW");
      return;
   }
```

Scenario 2(the list contains single element)

If the list contains single node then, the condition head → next == head will become true. In this case, we need to delete the entire list and make the head pointer free. This will be done by using the following statements.

```
if(head->next == head)
{
   head = NULL;
   free(head);
}
```

Scenario 3(the list contains more than one element)

If the list contains more than one element, then in order to delete the last element, we need to reach the last node. We also need to keep track of the second last node of the list. For this purpose, the two pointers ptr and preptr are defined. The following sequence of code is used for this purpose.

```
ptr = head;
      while(ptr ->next != head)
      {
         preptr=ptr;
         ptr = ptr->next;
      }
```

now, we need to make just one more pointer adjustment. We need to make the next pointer of preptr point to the next of ptr (i.e. head) and then make pointer ptr free.

```
preptr->next = ptr -> next;
```

```
    free(ptr);
```

Algorithm

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 8

[END OF IF]

Step 2: SET PTR = HEAD

Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != HEAD

Step 4: SET PREPTR = PTR

Step 5: SET PTR = PTR -> NEXT

[END OF LOOP]

Step 6: SET PREPTR -> NEXT = HEAD

Step 7: FREE PTR

Step 8: EXIT

Traversing in Circular Singly linked list

Traversing in circular singly linked list can be done through a loop. Initialize the temporary pointer variable temp to head pointer and run the while loop until the next pointer of temp becomes head. The algorithm and the c function implementing the algorithm is described as follows.

Algorithm

STEP 1: SET PTR = HEAD

STEP 2: IF PTR = NULL

WRITE "EMPTY LIST"

GOTO STEP 8

END OF IF

STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR → NEXT != HEAD

STEP 5: PRINT PTR → DATA

STEP 6: PTR = PTR → NEXT

[END OF LOOP]

STEP 7: PRINT PTR→ DATA

STEP 8: EXIT