UNIT - 3: LINKED LIST

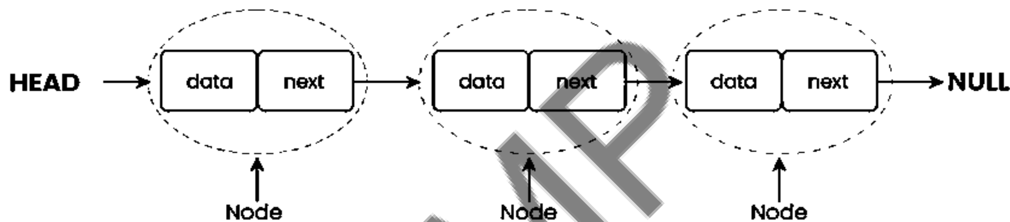
3.1 Introduction

Linked lists were developed in 1955-1956, by Allen Newell, Cliff Shaw and Herbert A. Simon at RAND Corporation and Carnegie Mellon University as the primary data structure for their Information Processing Language (IPL).

A linked list is a linear collection of data elements in which each element points to the next element.

A linked list is a linear data structure consisting of a collection of nodes which together represent a sequence.

A linked list is a sequence of nodes that contain two fields - **data** and **a link** (a **pointer to the next node**) to the next node. The last node is linked to a terminator used to indicate the end of the list more often denoted by NULL.



GMP

Figure 3.1: Structure of Linked list

Example 1: Linked list containing character data

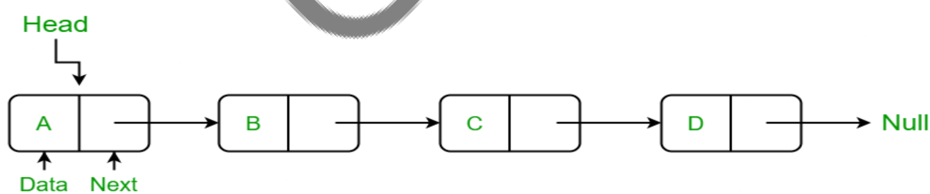


Figure 3.2: Linked list containing character data

Example 2: Linked list containing integer data

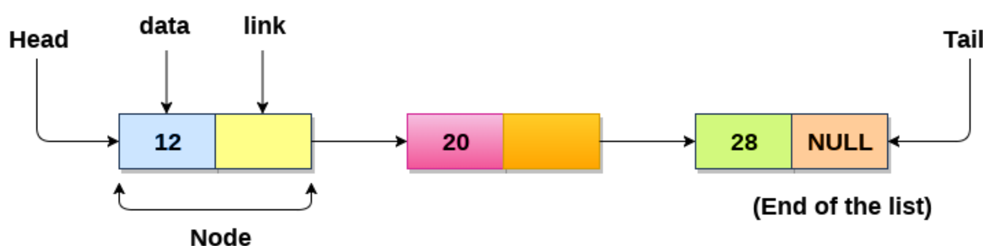


Figure 3.3: Linked list containing integer data

Some important points to be considered in linked list

- The entry point of a linked list is known as the head which is link element.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.
- The consecutive elements are connected by pointers.
- The size of a linked list is not fixed.
- The last node of the linked list points to null.
- Memory is not wasted but extra memory is consumed as it also uses pointers to keep track of the next successive node.
- The entry point of a linked list is known as the head.

Basically each node contains two fields as data and a reference/link to the next node in sequential manner. This structure allows insertion or removal of elements efficiently from any position in the sequence. As in linked lists the nodes are serially linked the data access time is linear in respect to the number of nodes present in the list. To access any node requires that the prior node be accessed beforehand.

Advantages of a linked list:

- Dynamic Data structure: The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.
- Ease of Insertion/Deletion: The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.
- Efficient Memory Utilization: As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.
- Implementation: Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

Disadvantages of Linked Lists:

- Memory usage - Linked lists require additional memory for storing the pointers, compared to arrays. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.
- Random Access: Unlike arrays, linked lists do not allow direct access to elements by index. Traversal is required to reach a specific node.
- Traversal - Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly, while in case of array we can randomly access it by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
- Reverse traversing - Backtracking or reverse traversing is difficult in a linked list. In a doubly-linked list, it is easier but requires more memory to store the back pointer.

GMP

Applications of Linked list:

The applications of the Linked list are given as follows -

- With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
- A linked list can be used to represent the sparse matrix.
- The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
- Using linked list, we can implement stack, queue, tree, and other various data structures.
- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.
- A linked list can be used in undo functionality of software's

Major differences between array and linked-list are listed below:

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

GMP

3.2 Representation of Linked List:

Each node of the linked list is represented as follows.

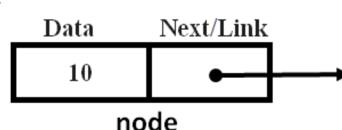


Figure 3.4: node representation

Each node consists:

- A data item
- An address of another node

Both the data item and the next node reference are wrapped in a structure as:

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

Each struct node has a data item and a pointer to another struct node.

Representation of linked list in memory:

1. Linked list can be represented in memory by using two array respectively known as INFO/DATA and LINK. INFO[K] contains information of element and LINK[K] contains next node address respectively.
2. The list also requires a variable "Name" or "Start" or "Head", which contains address of the first node.
3. Pointer field of last node denoted by NULL which indicates the end of list.

The linked list can be represented in memory as shown in following figure.

Figure 3.5: Representation of linked list in memory

GMP

The Figure shows a linked list in memory where each node of the list contains a single character. We can obtain the actual list of characters as follows:

START=9, so INFO [9] =N is the first character.

LINK [9] =3, so INFO [3] =O is the second character.

LINK [3] =6, so INFO [6] =\n (blank) is the third character.

LINK [6] =11 so INFO [11] =E is the fourth character.

LINK [11] =7, so INFO [7] =X is the fifth character.

LINK [7] =10, so INFO [10] =I is the sixth character.

LINK [10] -4, so INFO [4] = T is the seventh character.

LINK [4] =0, the NULL value, so the list is ended.

In other words NO EXIT is the character string.

3.3 Operations of Linked Lists

The basic operations in a linked list are:

1. **Creation:** Creating a singly linked list process starts with creating a new node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the dynamic memory allocation malloc() function.

2. **Insertion:** Adding a new node to a linked list involves adjusting the pointers of the existing nodes to maintain the proper sequence. Insertion can be performed at the beginning, end, or any position within the list

3. **Deletion:** Removing a node from a linked list requires adjusting the pointers of the neighboring nodes to bridge the gap left by the deleted node. Deletion can be performed at the beginning, end, or any position within the list.

4. **Searching:** Searching for a specific value in a linked list involves traversing the list from the head node until the value is found or the end of the list is reached.

GMP

3.4 Types of linked lists

There are various types of linked list:

1. **Singly Linked Lists**
2. **Doubly Linked Lists**
3. **Circular Linked Lists**
4. **Circular Doubly Linked List**

1. **Singly Linked Lists:**

The nodes only point to the address of the next node in the list.

Singly linked lists contain two "buckets" in one node; one bucket holds the data and the other bucket holds the address of the next node of the list. Traversals can be done in one direction only as there is only a single link between two nodes of the same list.

Figure 3.6: Singly linked lists

2. **Doubly Linked Lists:**

The nodes point to the addresses of both previous and next nodes.

Doubly Linked Lists contain three "buckets" in one node; one bucket holds the data and the other buckets hold the addresses of the previous and next nodes in the list. The list is traversed twice as the nodes in the list are connected to each other from both sides.

Figure 3.7: Doubly linked lists

GMP

3. **Circular Linked Lists**

The last node in the list will point to the first node in the list. It can either be singly linked or doubly linked.

Circular linked lists can exist in both singly linked list and doubly linked list.

Since the last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.

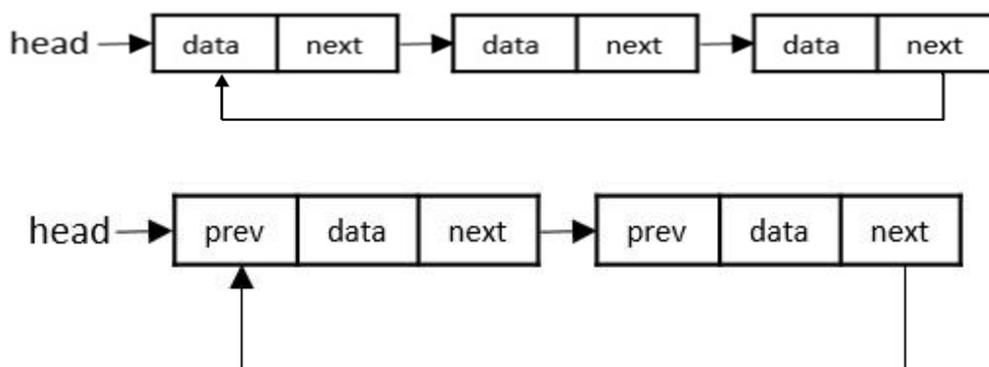


Figure 3.8: Circular linked lists

4. **Circular Doubly Linked List:**

The last node in the list will point to the first node in the list and the nodes point to the addresses of both previous and next nodes.

Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the nodes. The last node of the list contains the address of the first node of the list. The first node of the list also contains the address of the last node in its previous pointer.

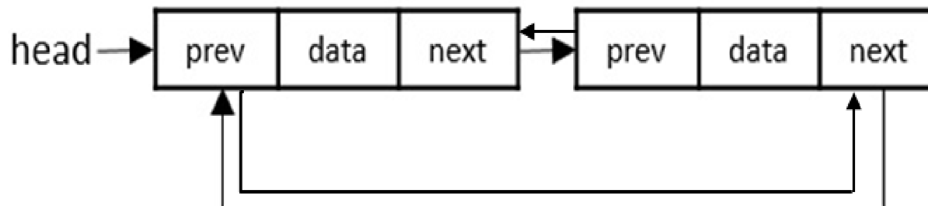


Figure 3.9: Circular Doubly linked lists

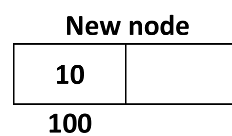
GMP

Singly Linked Lists

a. Creating a node for Single Linked List:

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (data) has to be read from the user, set next field to NULL and finally returns the address of the node. Figure 5 illustrates the creation of a node for single linked list.

```
node* get node( )
{
        node* new node;
        new node =  ( node  * )  malloc( sizeof( node) ;
        printf( "\ n Enter data:  " ) ;
        scanf( " % d " ,  & new node  - >  data) ;
        new node  - >  next =  NUL;
        return new nod e;
}
```



Creating a Singly Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

1. Get the new node using getnode().

   newnode = getnode();

2. If the list is empty, assign new node as start.

   start = newnode;

3. If the list is not empty, follow the steps given below:

   - The next field of the new node is made to point the first node (1.e. start node) in the list by assigning the address of the first node.
   - The start pointer is made to point new node by assigning address of new node times.

4. Repeat the above steps 'n' times.

Following Figure 3.10 shows 4 elements (data items) in a single linked list stored at different locations in memory.

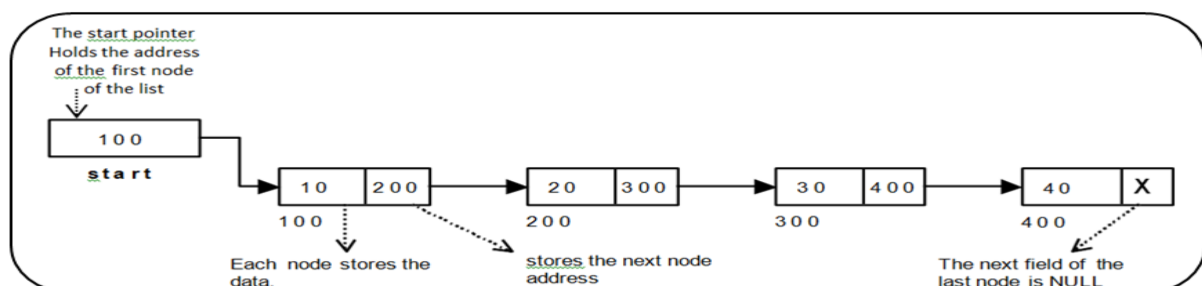


Figure 3.10: Single Linked List with four nodes

GMP

We can observe in the above Figure 3.10 that there are four different nodes having address 100, 200, 300 and 400 respectively. The first node contains the address of the next node, which is 200, the second node contains the address of the next node which is 300, the third node contains the address of the last node, which is 400, and the fourth node contains the NULL value in its address part as it does not point to any node. The pointer that holds the address of the initial node is known as a head pointer named as start and it contains address of first node which is 100.

b. Insertion of a node in Single Linked List:

The insertion of a node in a singly linked list is one of the most basic operations. Memory is to be allocated for the new node as allocated for creating a list before reading the data. The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL.

The new node can then be inserted at three different places as:

i) Inserting a node at the beginning.

ii) Inserting a node at the end.

iii) Inserting a node at intermediate position.

The process of inserting the node at particular place is discussed below.

i) Inserting a node at the beginning of the list:

Approach: To insert a node at the start/beginning/front of a Linked List, we need to:

1. Make the first node of Linked List linked to the new node
2. Remove the head from the original first node of Linked List
3. Make the new node as the Head of the Linked List.

The following are the **steps** to insert a new node at the beginning of the list.

1. Get the new node using getnode().

   newnode = getnode();

2. If the list is empty, assign new node as start.

   start = newnode;

3. If the list is not empty, follow the steps given below

   - newnode->next = start;
   - start=newnode;

GMP

The Figure 3.11 shows inserting a node into the single linked list at the beginning.

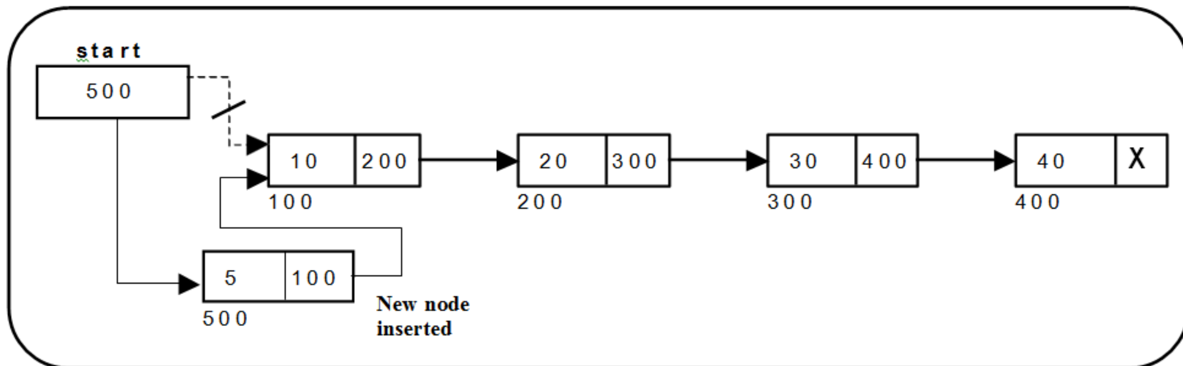


Figure 3.11**:** inserting a node at the beginning of the list

ii) Inserting a node at the end of the list:

Approach: To insert a node at the end of a Linked List, we need to:

1. Go to the last node of the Linked List
2. Change the next pointer of last node from NULL to the new node
3. Make the next pointer of new node as NULL to show the end of Linked List

The **steps** to insert a new node at the end of the list are given below.

1. Get the new node using getnode().

   newnode = getnode();

2. If the list is empty, assign new node as start.

   start = newnode;

3. If the list is not empty follow the steps given below:

   temp = start;

   while(temp->next != NULL)

   temp temp->next;

   temp->next = newnode;

GMP

Following Figure 3.12 shows how a node is inserted at the end into the single linked.

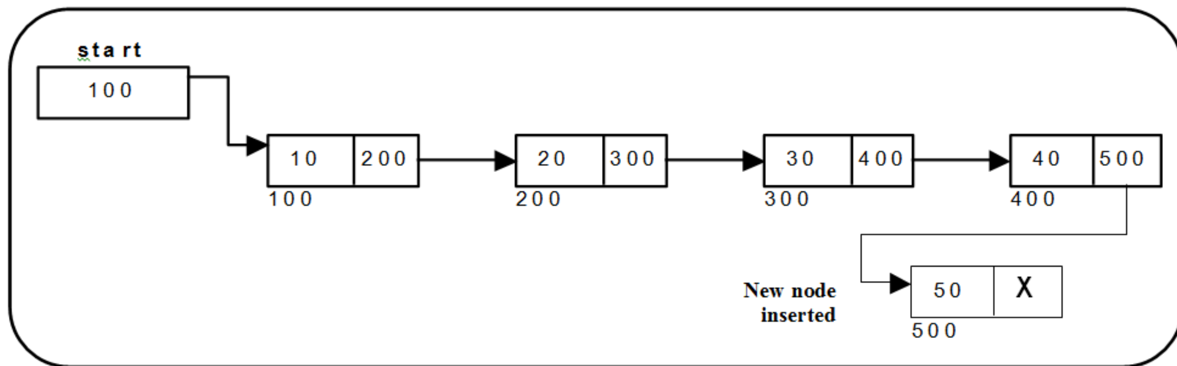


Figure 3.12: Inserting a node at the end of the list

iii) Inserting a node at intermediate position of list

Approach: To insert a node after a given node in a Linked List, we need to:

1. Check if the given node exists or not.
   a. If it do not exists,
      i. Terminate the process.
   b. If the given node exists,
      i. Make the element to be inserted as a new node
      ii. Change the next pointer of given node to the new node
      iii. Now shift the original next pointer of given node to the next pointer of new node

GMP

The following are the steps to insert a new node in an intermediate position in the list.

1. Get the new node using getnode().

   newnode = getnode();

2. Ensure that the specified position is in between first node and last node.

   If not, specified position is invalid.

   This can be done by other countnode() function.

3. Store the starting address (which is in start pointer) in temp and prev pointers.

   Then traverse the temp pointer upto specified position followed by prev pointer.

4. After reaching the specified position, follow the steps given below:

   prev -> next = newnode;

   newnode -> next = temp;

Figure 3.13 shows inserting a node at a specified intermediate position other than beginning and end into the single linked list. Let the intermediate position be 3.

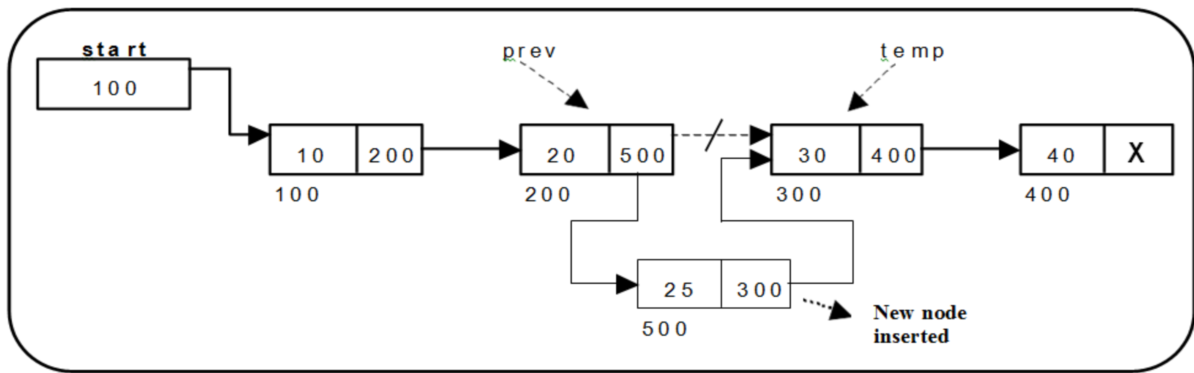


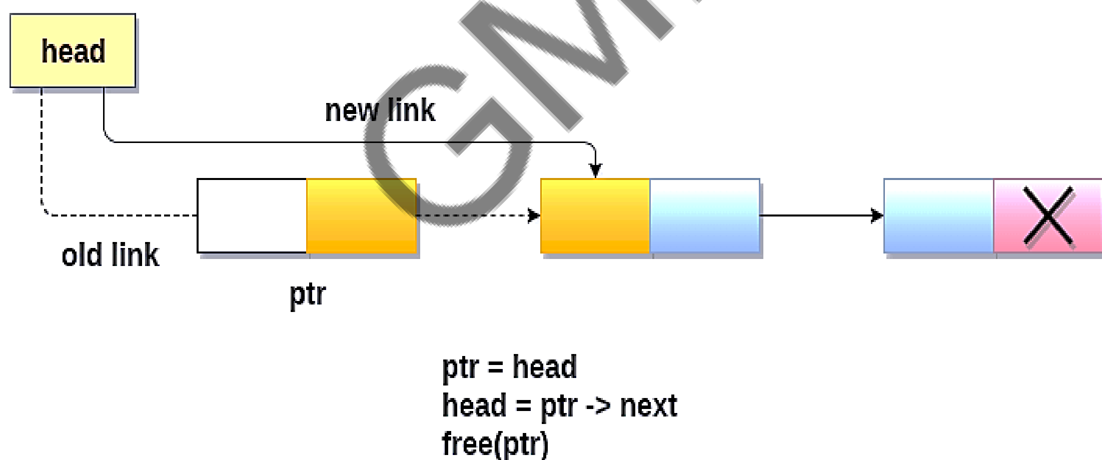
Figure 3.13: inserting a node at a specified intermediate position into single linked list

c. Deletion of a node from Single 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.

i) Delete node from Beginning of the list
ii) Delete node from End of the list
iii) Delete a Specific Node (intermediate) of the list

Memory is always release out after each delete operation on linked list at any position.



GMP

ptr = head
head = ptr -> next
free(ptr)

Figure 3.14: Deletion of a node from Single Linked List

i) Delete node from Beginning of the list

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 start/head, point to the next of the start/head. This will be done by using the following statements.

The following steps are followed, to delete a node from the beginning of the list:

1. Check If list is empty then
   Display 'Empty List' message.
2. If the list is not empty, follow the steps given below:
   temp = start;
   start start->next;
3. free(temp);

The following Figure 3.15 shows deleting a node at the beginning of a single linked list.

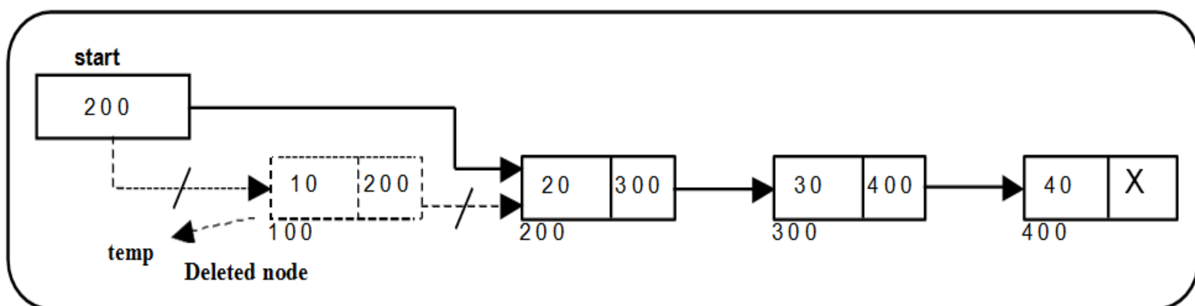


Figure 3.15: deleting a node at the beginning of a single linked list

Memory is then release out by free operation with the pointer temp which was pointing to the start/head node of the list. This will be done by using the following statement.

free(temp);

ii) Delete node from 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.

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 start -> next = NULL will survive and therefore, the only node start/head of the list will be assigned to null.

This will be done by using the following statements.

1. temp = start
2. start = NULL
3. free(temp)

GMP

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 **prev** will be used where **temp** will point to the last node and **prev** will point to the second last node of the list.

The following **steps** are followed to delete a node at the end of the list:

1. If list is empty then

    Display 'Empty List' message.

2. If the list is not empty, follow the steps given below:

```
temp= prev= start;
while(temp-> next != NULL)
{
prev = temp;
temp= temp-> next;
}
Prev->next=NULL,
free(temp);
```

The following Figure 3.16 shows deleting a node at the end of the list.

GMP

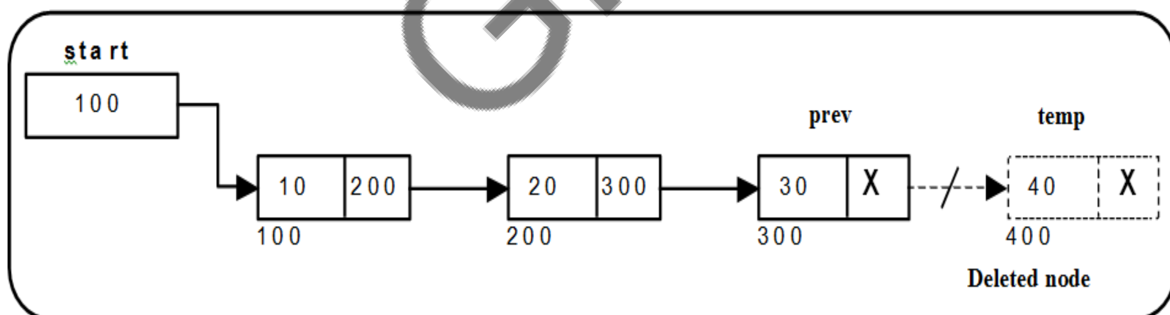


Figure 3.16: deleting a node at the end of the list.

iii) Delete a Specific Node (intermediate) of the list

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 and then just need to make a few pointer adjustments.

The **steps** to delete a node from an intermediate position from the list having more than two nodes:

1. If list is empty then

   Display 'Empty List' message.

2. If the list is not empty, follow the steps given below.

```
if(pos> 1 && pos<nodectr)
{
    temp= prev= start,
    ctr=1;
    while(ctr<pos)
    {
        prev=temp;
        temp=temp->next;
        ctr++;
    }
    prev-> next= temp -> next;
    free(temp);
    printf("\n node deleted..");
}
```

GMP

Figure 3.17 shows deleting a Specific node (intermediate) from a single linked list.

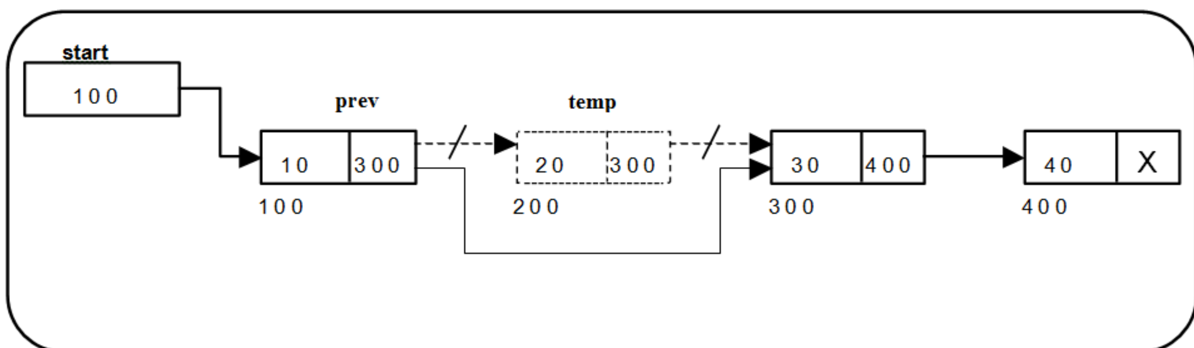


Figure 3.17: deleting a Specific node (intermediate) of the list

d. Traversing and Displaying Single Linked List:

The data or information contained in a list can be display by traversing a list. Traversing is the most common operation that is performed in almost every scenario of singly linked list. Traversing is the process of visiting each node of the list once in order to perform some

operation on that. Traverse a linked list is done node by node from the first node until the end of the list is reached.

Approach:

1. Assign the address of start pointer to a temp pointer.
2. Display the information from the data field of each node.

Traversing a list involves the following **steps.**

1. temp = start;
2. IF temp = NULL then

    Display 'Empty List';
3. If list not empty then do following steps

    While (temp != NULL)

    {

    Display temp->DATA;

    Temp=temp -> NEXT;

    }

e. Searching Single Linked List:

Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and makes the comparison of every element of the list with the specified element. If the element is matched with any of the list element then the location of the element is returned from the function.

Steps for Searching Single Linked List are given below.

1. Assign temp= start;
2. i = 0;
3. IF temp = NULL then

    Display 'Empty List';
4. If list not empty then do following steps

    while (temp!=NULL)

    {

        If temp->data=ITEM then

        Display i+1;

GMP

```
    i=i+1;
    temp=temp->next;
}
```

GMP

Doubly Linked Lists

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.

A doubly linked list is a two-way list in which all nodes will have two links. It is a complex type of linked list in which a node contains a pointer to the previous/left as well as the next/right node in the sequence.

In a doubly linked list, a node consists of three parts:

- Data
- Pointer to the next/right node in sequence (next/right pointer)
- Pointer to the previous/left node (previous/left pointer).

GMP

The structure definition of doubly Linked List:

```
struct  dlinklist
{
        struct dlinklist *left ;
        int data;
        struct  dlinklist *right ;
} ;
Typedef struct  dlinklist node;
node *start = NULL;
```

A sample node in a doubly linked list is shown in the Figure 3.18.

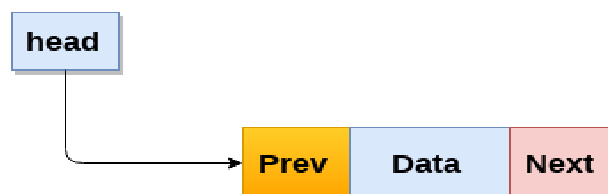


Figure 3.18: Node in a doubly linked list

Example: A doubly linked list containing three nodes having numbers from 1 to 3 in their data part is shown in the following Figure 3.19.

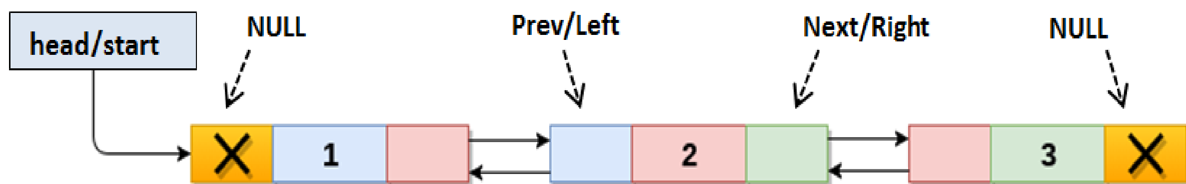


Figure 3.19: Example of a doubly linked list

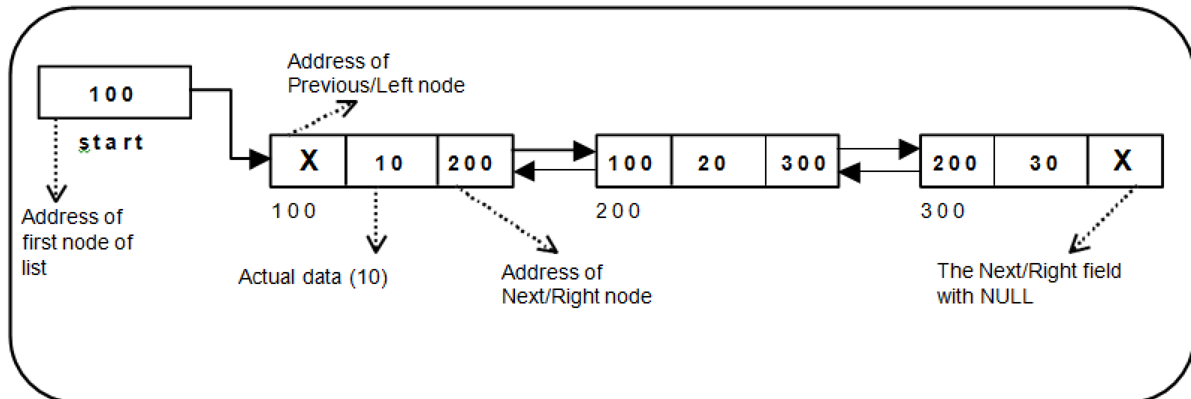


Figure 3.20: Example of a doubly linked list

The basic operations in a doubly linked list are:

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

a. Creating a node in doubly Linked List:

For creating a double linked list sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode() is used for creating a node. After allocating memory for the structure of type node the information for the item/data has to be read from the user and set both left field and right field to NULL.

The code to create a new node in double linked list:

```
node* getnode( )
{
        nod e* newnode;
        newnode = ( nod e * ) malloc( sizeo f ( node) ) ;
        printf ( " \n Enter data: " ) ;
        sca nf( " %d " , & newnode -> data) ;
        newnode - > left = NULL;
        newnode  - > right =   NULL;
         return newnode;
}
```

GMP

After creating a new node with data value 10 the node can be visualize as shown in Figure 3.21.

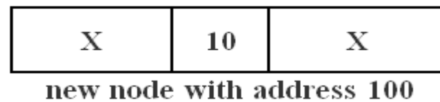


Figure 3.21: creation of new node in doubly link list

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.

a) Creating a Double Linked List with 'n' number of nodes:

The following **steps** are to be followed to create 'n' number of nodes:

1. Get the new node using getnode().
   newnode =getnode();
2. If the list is empty then
   start = newnode.
3. If the list is not empty, follow the steps given below:
   - The left field of the new node is made to point the previous node.
   - The previous nodes right field must be assigned with address of the new node.
4. Repeat the above steps 'n' times.

GMP

Figure 3.22 shows 3 items in a double linked list stored at different locations using createlist() function.

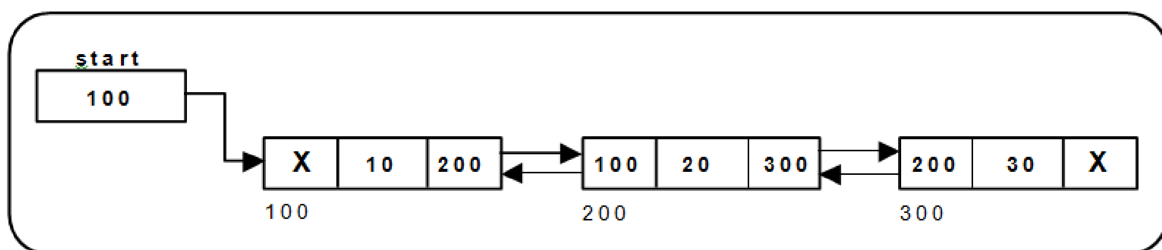


Figure 3.22: creation of new node with 3 nodes in doubly link list

To create 'n' number of nodes in Double Linked List the function createlist () is given below:

```
void createlist( int n)
{
```

```
  int i;
  node *newnode;
  node *temp;
  for(i=0; i<n; i++)
  {
    newnode = getnode();
    if(start = = NULL)
    {
        start = newnode;
    }
    else
    {
        temp = start;
        while(temp -> right)
        temp=temp-> right;
        temp-> right= newnode;
        newnode-> left = temp;
    }
  }
}
```

b) Inserting a node in doubly linked list

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.

Node can be inserted at various position in doubly linked list as:

i) Inserting a node at the beginning in doubly linked list

ii) Inserting a node at the end in doubly linked list

iii) Inserting a node at the intermediate position in doubly linked list

i) Inserting a node at the beginning in doubly 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.

The following **steps** are to be followed to insert a new node at the beginning of the list:

1. Get the new node using getnode().

   newnode=getnode();

2. If the list is empty then

   start = newnode.

3. If the list is not empty, follow the steps given below:

GMP

```
newnode-> right= start;
start->left= newnode;
start = newnode;
```

Figure 3.23 shows inserting a node into the double linked list at the beginning.

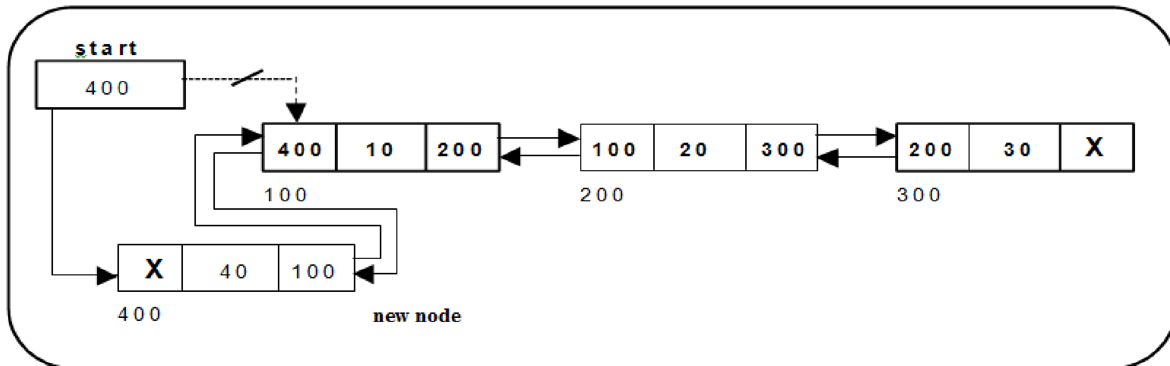


Figure 3.23: inserting a node at the beginning in doubly linked list

ii) Inserting a node at the end in doubly linked list

Figure 3.24 shows inserting a node into the double linked list at the end.

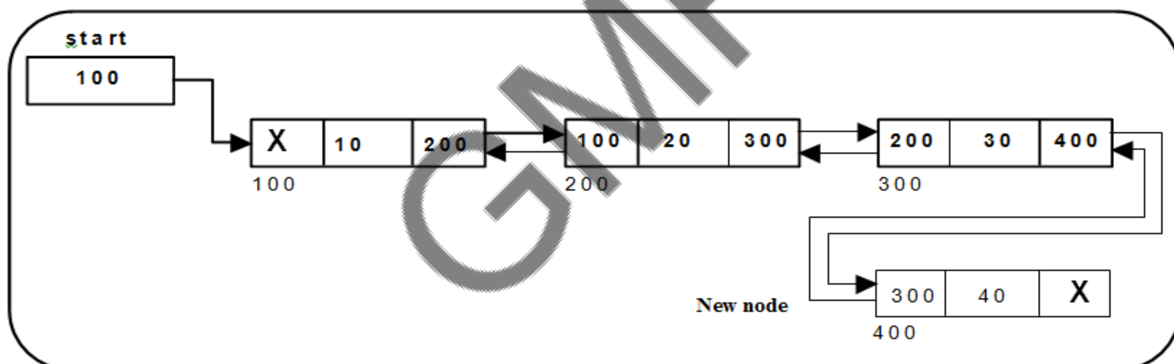


Figure 3.24: Inserting a node at the end in doubly linked list

GMP

The **steps** to insert a new node at the end of the list:

1. Get the new node using getnode()
2. If the list is empty then start = newnode.
3. If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> right!= NULL)
temp = temp -> right;
temp -> right = newnode;
newnode -> left = temp;
```

iii) Inserting a node at the intermediate position in doubly linked list

Figure 3.25 shows inserting a node into the doubly linked list at intermediate position.

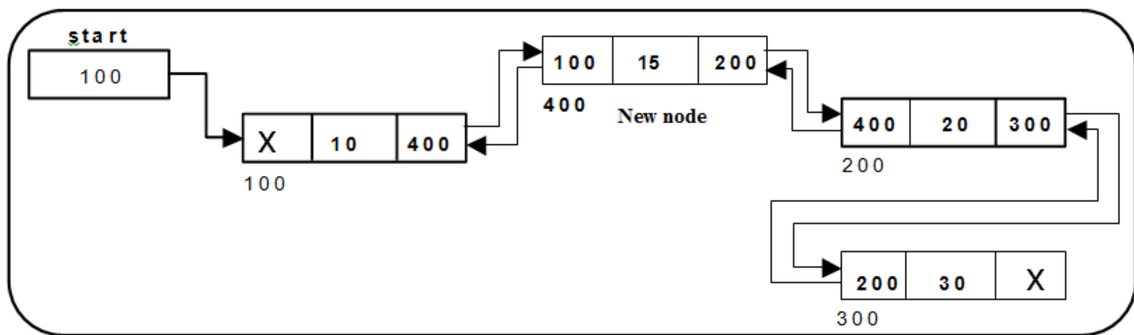


Figure 3.25: Inserting a node at intermediate position in doubly linked list

The **steps** to insert a new node in an intermediate position in the doubly linked list:

1. Get the new node using getnode().

   newnode=getnode();

2. Ensure that the specified position is in between first node and last node.

   If not, specified position is invalid. (done by countnode() function)

3. Store the starting address (which is in start pointer) in temp and prev pointers.
4. Then traverse the temp pointer upto the specified position followed by prev pointer.
5. After reaching the specified position, follow the steps given below:

   newnode->left = temp;

   newnode-> right = temp -> right;

   temp -> right -> left = newnode;

   temp -> right = newnode;

GMP

c) Deleting a node from doubly linked list

Node can be deleted from various position in doubly linked list as:

i) Deleting a node from the beginning in doubly linked list
ii) Deleting a node from the end in doubly linked list
iii) Deleting a node from the intermediate position in doubly linked list

i) Deleting a node from the beginning in doubly linked list

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

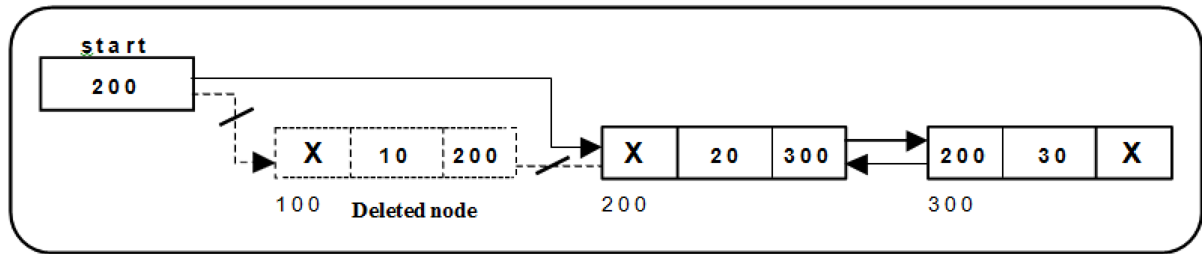


Figure 3.26: Deleting a node at beginning in doubly linked list

ii) Deleting a node from the end in doubly linked list

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, follow the given steps as below.

- If the list is already empty then the condition start == 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 start → next == NULL become true. In this case, we just need to assign the start of the list to NULL and free start in order to completely delete the list.
- Otherwise, just traverse the list to reach the last node of the list.

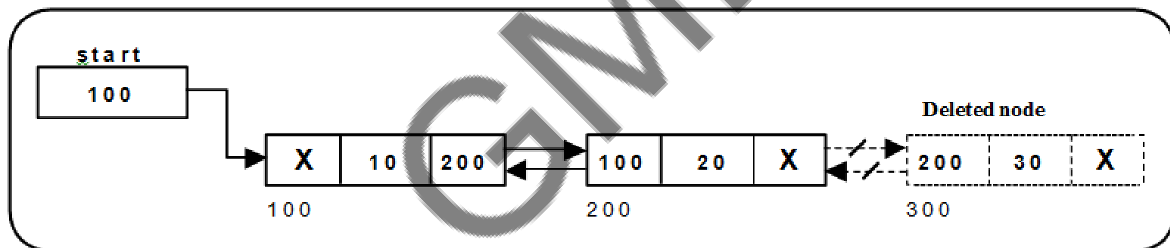


Figure 3.27: Deleting a node at end in doubly linked list

GMP

The following steps are followed to delete a node at the end of the list:

1. If list is empty then display 'Empty List' message

2. If the list is not empty, follow the steps given below:

```
temp start;
while(temp-> right != NULL)
{
        temp = temp-> right;
}
temp -> left -> right = NULL;
```

3. free(temp);

iii) Deleting a node at an Intermediate position in doubly linked list

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.
- Traverse the list until we find the desired data value.
- Check if this is the last node of the list. If it is so then we can't perform deletion.
- 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.
- 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.

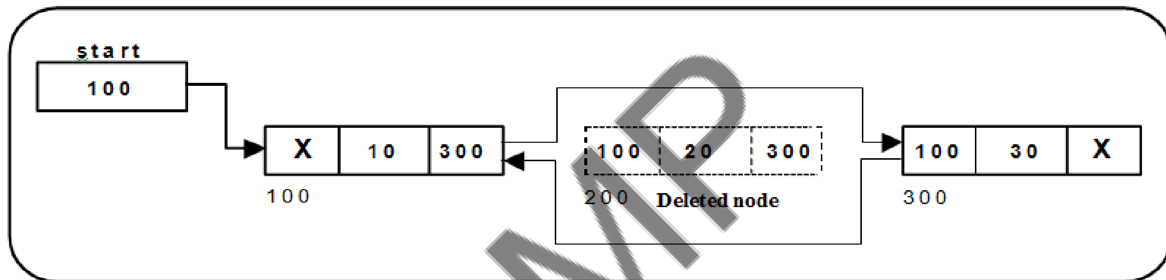


Figure 3.28: Deleting a node at Intermediate position in doubly linked list

The following **steps** are followed, to delete a node from an intermediate position in the list (List must have more than two nodes).

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:
   - Get the position of the node to delete.
   - Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
   - Then perform the following steps:

```
if(pos> 1 && pos<nodectr)
{ temp=start;
  i=1:
  while(i<pos)
  { temp=temp-> right;
    i++;
  }
  temp-> right-> left = temp-> left;
```

GMP

```
temp-> left-> right = temp -> right;
free(temp);
printf("\n node deleted..");
}
```

d) Traversal and displaying a doubly linked list (Left to Right):

Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc. To display the information, you have to traverse the list, node by node from the first node, until the end of the list is reached.

The following steps are followed, to traverse a list from left to right:

1. If list is empty then display 'Empty List'
2. If the list is not empty, follow the steps given below:

```
temp = start;
while(temp != NULL)
{
Display temp-> data;
temp = temp-> right;
}
```

GMP

Circular Linked List

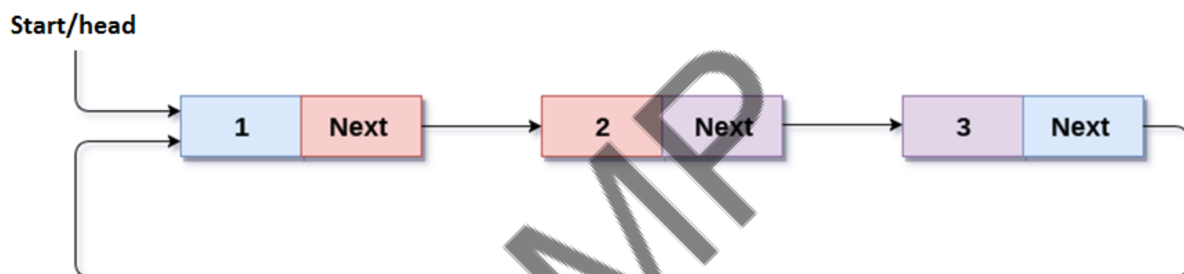
In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list. That means circular linked list is similar to the single linked list except that the last node points to the first node in 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 figure 3.29 shows a circular singly linked list.

Example: 1



GMP

Example: 2

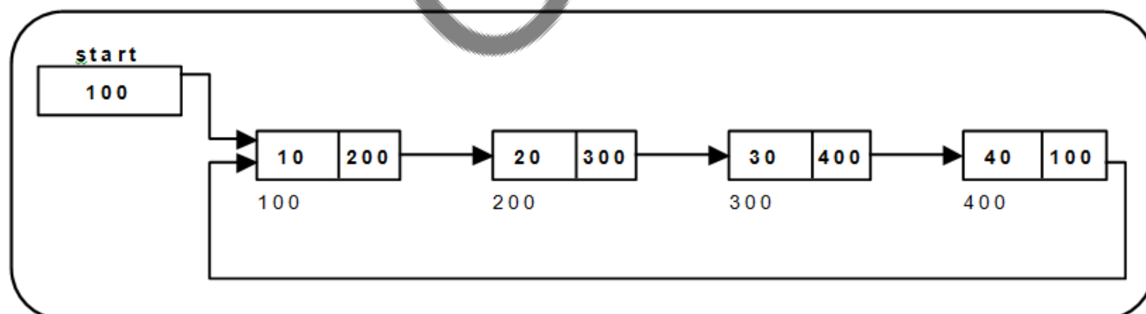


Figure 3.29: Circular singly linked list

The basic operations in a circular single linked list are:

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

a) Creating a circular single Linked List with 'n' number of nodes

The **steps** to create n number of nodes:

1. Get the new node using getnode().
   newnode = getnode();
2. If the list is empty, assign new node as start.
   start = newnode;
3. If the list is not empty then
   temp = start;
   while(temp->next != NULL)
   temp = temp-> next;
   temp->next = newnode;
4. Repeat the above steps n times.
5. newnode->next = start;

b) Inserting a node in circular linear linked list

A node can be inserted at various positions (start, end, intermediate) in the circular linear list as:

i) Inserting a node at the beginning in circular linear linked list
ii) Inserting a node at the end in circular linear linked list

i) Inserting a node at the beginning in circular linear linked list

There are two scenarios 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.

Figure 3.30 shows inserting a node into the circular single linked list at the beginning.

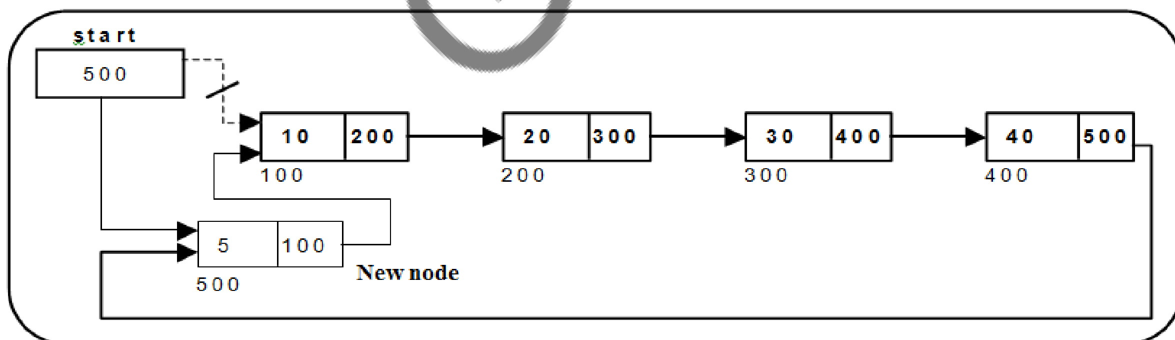


Figure 3.30: inserting a node at the beginning in circular single linked list

The following **steps** are to be followed to insert a new node at the beginning of the circular list:

1. Get the new node using getnode().
   newnode = getnode();
2. If the list is empty then assign new node as start.
   Start= newnode;
   newnode->next = start;

GMP

3. If the list is not empty then

```
last = start;
while (last-> next != start)
last=last->next;
newnode->next = start;
start = newnode;
last-> next= start;
```

ii) Inserting a node at the end in circular linear linked list

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

Figure 3.31 shows inserting a node into the circular single linked list at the end.

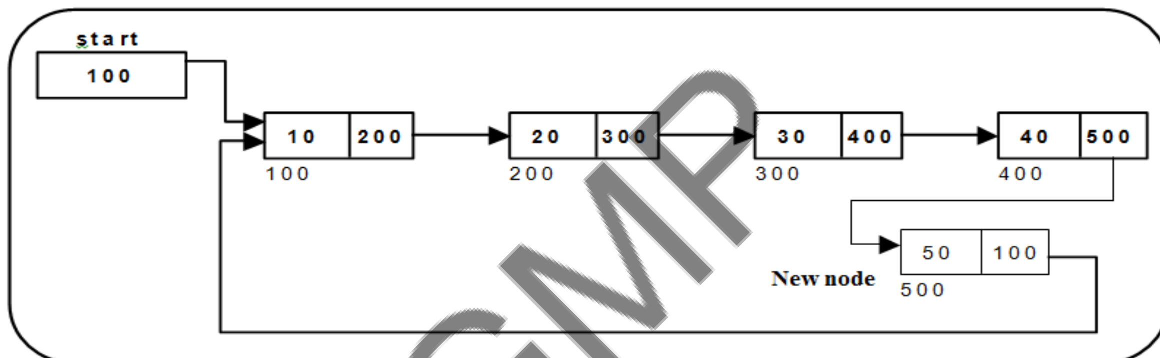


Figure 3.31: inserting a node at the end in circular single linked list

The following **steps** are to be followed to insert a new node at the end of the circular list:

1. Get the new node using getnode().

```
newnode = getnode();
```

2. If the list is empty, assign new node as start.

```
start = newnode;
newnode->next = start;
```

3. If the list is not empty then

```
temp = start;
while(temp-> next != start)
temp= temp-> next;
temp->next = newnode;
newnode->next = start;
```

c) Deleting a node from circular linear linked list

A node can be deleted from various positions (start, end, intermediate) in the circular linear list as:

i) Deleting a node at the beginning in circular linear linked list
ii) Deleting a node at the end in circular linear linked list

i) Deleting a node at the beginning in circular linear linked list

Removing the node from circular singly linked list at the beginning.

Figure 3.32 shows deleting a node from the circular single linked list at the beginning.

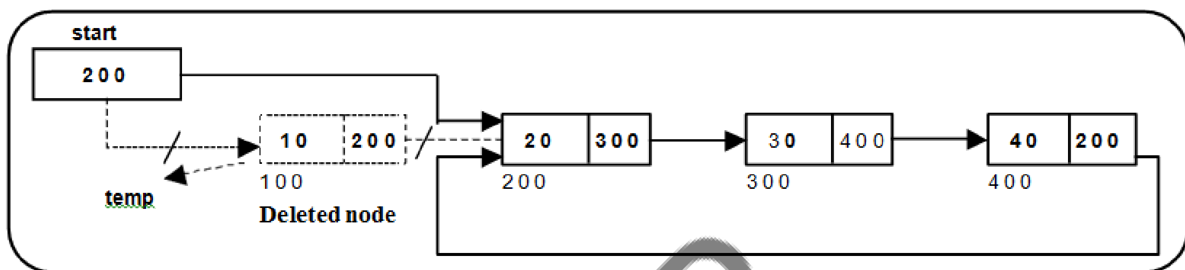


Figure 3.32: deleting a node at the beginning in circular single linked list

The **steps** to delete a node at the beginning of the circular single linked list:

1. If the list is empty

   Display 'Empty List"

2. If the list is not empty then

   Last= temp= start;

   while(last-> next != start)

   last= last-> next;

   start= start->next;

   last-> next= start,

3. After deleting the node, if the list is empty then

   start = NULL

ii) Deleting a node at the end in circular linear linked list

There are three scenarios of deleting a node in circular singly linked list at the end: the list is empty, the list contains single element and the list contains more than one element.

The following **steps** are followed to delete a node at the end of the circular linear linked list:

1. If the list is empty

GMP

Display a message 'Empty List"

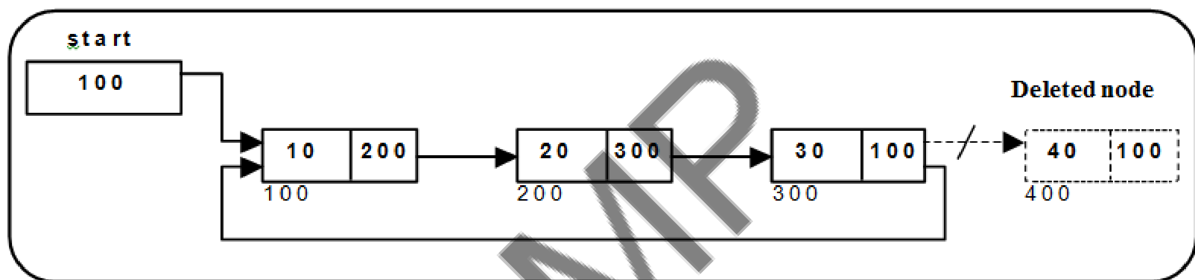
2. If the list is not empty then

```
temp= start;
prev= start;
while(temp-> next != start)
{prev=temp;
 temp = temp-> next;
}prev-> next= start;
```

3. After deleting the node, if the list is empty then

```
start = NULL.
```

Figure 3.33 shows deleting a node at the end of a circular single linked list.



GMP

Figure 3.33: deleting a node at the end in circular single linked list

d) Traversing circular linear linked list from Left to Right

Traversing circular linear linked list means visiting each element of the list at least once in order to perform some specific operation. 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 **start/head**.

Steps to traverse a circular single linked list from left to right:

1. If list is empty then

   Display 'Empty List' message

2. If the list is not empty then

```
temp = start;
do
{
        Display ("%d", temp- > data);
        temp = temp->next;
} while(temp!= start);
```

Circular Doubly Linked List

Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list. The first node of the list also contains address of the last node in its previous pointer.

A circular doubly linked list is shown in the following Figure 3.34.

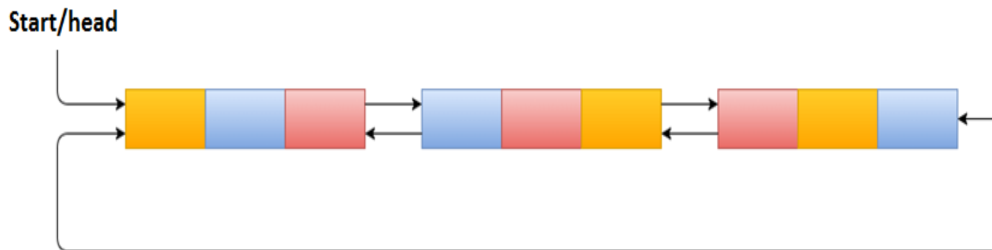


Figure 3.34: circular doubly linked list

Operations on circular doubly linked list:

There are various operations which can be performed on circular doubly linked list. The node structure of a circular doubly linked list is similar to doubly linked list.

The basic operations in a circular doubly linked list are:

1. Insertion
2. Deletion

a) Inserting a node in circular doubly Linked List

A node in circular doubly Linked List can be inserted in various ways as:

i) Inserting a node at beginning in circular doubly linked list
ii) Inserting a node at end in circular doubly linked list

i) Inserting a node at beginning in circular doubly linked list

The following are **steps** to insert a new node at the beginning of the circular doubly linked list:

1. Get the new node using getnode().

   ptr = getnode();

2. If the list is empty (head == NULL)then

   head = ptr;

   ptr -> next = head;

GMP

ptr -> prev = head;

3. If the list is not empty (head != NULL)then

   temp = head;

   while(temp -> next != head)

   {

   temp = temp -> next;

   }

4. All the pointer adjustments done

   temp -> next = ptr;

   ptr -> prev = temp;

   head -> prev = ptr;

   ptr -> next = head;

   head = ptr;

Figure 3.34 shows how the new node is inserted at beginning in circular doubly linked list

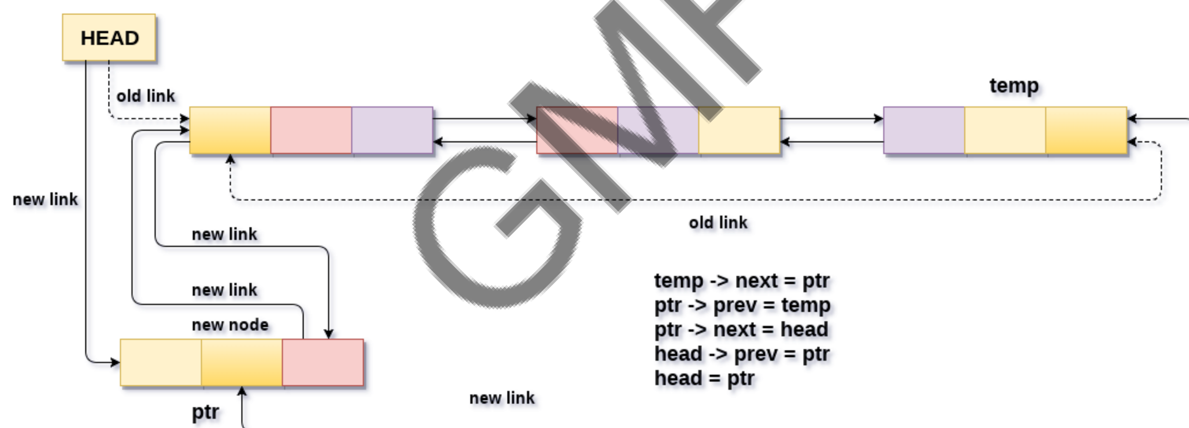


Figure 3.34: a node is inserted at beginning in circular doubly linked list

ii) Inserting a node at end in circular doubly linked list

The following are **steps** to insert a new node at the end of the circular doubly linked list:

1. Get the new node using getnode().

   ptr = getnode();

2. If the list is empty (head == NULL)then

   head = ptr;

   ptr -> next = head;

   ptr -> prev = head;

3. If the list is not empty (head != NULL) then

    head -> prev = ptr;

    ptr -> next = head;

4. All the pointer adjustments done

    temp->next = ptr;

    ptr ->prev=temp;

Figure 3.35 shows how the new node is inserted at end in circular doubly linked list

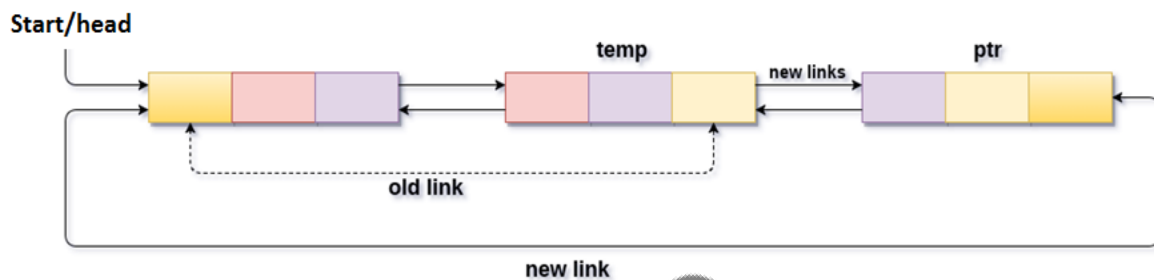


Figure 3.35: a node is inserted at end in circular doubly linked list

b) Deleting a node in circular doubly Linked List

A node in circular doubly Linked List can be deleted in various ways as:

i) Deleting a node at beginning in circular doubly linked list
ii) Deleting a node at end in circular doubly linked list

i) Deleting a node at beginning in circular doubly linked list

There can be two scenario of deleting the beginning node in a circular doubly linked list. The node which is to be deleted can be the only node present in the linked list and the list contains more than one element in the list

The following are **steps** to delete node at the beginning of the circular doubly linked list:

1. If only one node (head → next == head)

    head = NULL;

    free(head);

2. Otherwise (head → next == head is false)

    temp = head;

    while(temp -> next != head)

    {

        temp = temp -> next;

    }

GMP

3. All the pointer adjustments done

   temp -> next = head -> next;

   head -> next -> prev = temp;

4. free the head and make new head node of the list

   free(head);

   head = temp -> next;

Figure 3.36 shows how the beginning node is deleted in circular doubly linked list

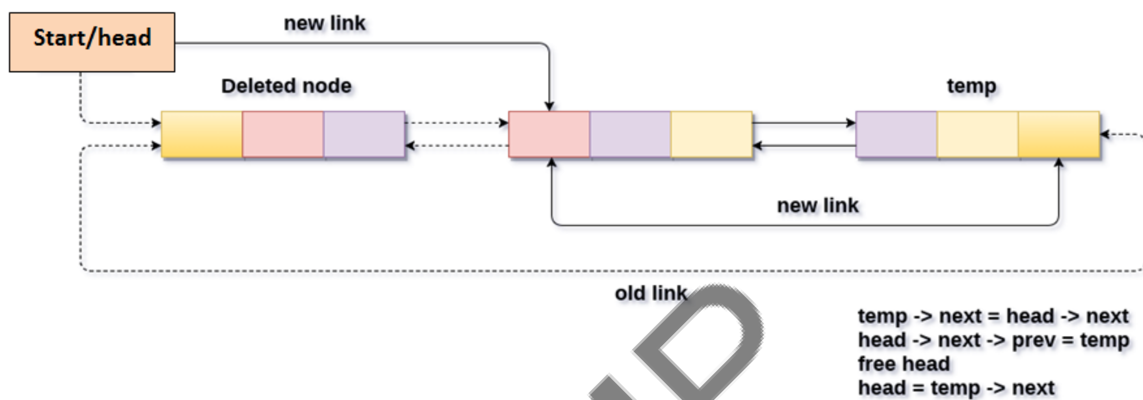


Figure 3.36: deleted beginning node in circular doubly linked list

ii) Deleting a node at end in circular doubly linked list

There can be two scenario of deleting the end node in a circular doubly linked list. The node which is to be deleted can be the only node present in the linked list and the list contains more than one element in the list

The following are **steps** to delete node at the end of the circular doubly linked list:

1. If only one node (head → next == head)

   head = NULL;

   free(head);

2. Otherwise (head → next == head is false)

   temp = head;

   while(temp -> next != head)

   {

   temp = temp -> next;

   }

3. All the pointer adjustments done

   temp -> prev -> next = head;

GMP

head -> prev = ptr -> prev;

4. free the head

free(head);

Figure 3.36 shows how the end node is deleted in circular doubly linked list

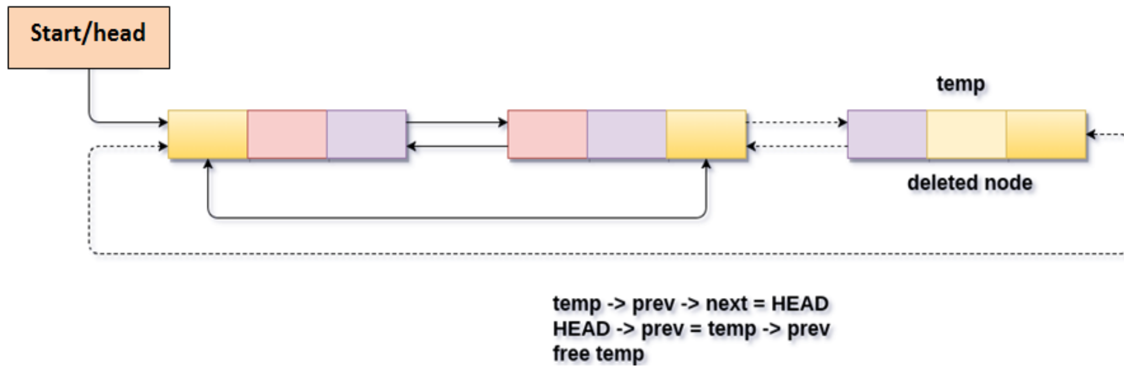


Figure 3.36: deleted end node in circular doubly linked list

GMP