LINKED LISTS

In this chapter, the list data structure is presented. This structure can be used as the basis for the implementation of other data structures (stacks, queues etc.). The basic linked list can be used without modification in many programs. However, some applications require enhancements to the linked list design. These enhancements fall into three broad categories and yield variations on linked lists that can be used in any combination: circular linked lists, double linked lists and lists with header nodes.

Linked lists and arrays are similar since they both store collections of data. Array is the most common data structure used to store collections of 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. The disadvantages of arrays are:

The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.

Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.

Deleting an element from an array is not possible.

Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. Generally array's allocates the memory for all its elements in one block whereas linked lists use an entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.

Here is a quick review of the terminology and rules of pointers. The linked list code will depend on the following functions:

malloc() is a system function which allocates a block of memory in the "heap" and returns a pointer to the new block. The prototype of malloc() and other heap functions are in stdlib.h. malloc() returns NULL if it cannot fulfill the request. It is defined by:

*void *malloc (number_of_bytes)*

Since a void * is returned the C standard states that this pointer can be converted to any type. For example,

```
char *cp;
cp = (char *) malloc (100);
```

Attempts to get 100 bytes and assigns the starting address to cp. We can also use the sizeof() function to specify the number of bytes. For example,

```
int *ip;
ip = (int *) malloc (100*sizeof(int));
```

free() is the opposite of malloc(), which de-allocates memory. The argument to free() is a pointer to a block of memory in the heap — a pointer which was obtained by a malloc() function. The syntax is:

free (ptr);

The advantage of free() is simply memory management when we no longer need a block.

Linked List Concepts:

A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list.

The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:

Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
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.
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:

It consumes more space because every node requires a additional pointer to store address of the next node.
Searching a particular element in list is difficult and also time consuming.

Types of Linked Lists:

Basically we can put linked lists into the following four items:

Single Linked List.

Double Linked List.

Circular Linked List.

Circular Double Linked List.

A single linked list is one in which all nodes are linked together in some sequential manner. Hence, it is also called as linear linked list.

A double linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse in forward direction and backward direction.

A circular linked list is one, which has no beginning and no end. A single linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node.

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

Comparison between array and linked list:

ARRAY	LINKED LIST
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

Trade offs between linked lists and arrays:

FEATURE	ARRAYS	LINKED LISTS
Sequential access	efficient	efficient
Random access	efficient	inefficient
Resigning	inefficient	efficient
Element rearranging	inefficient	efficient
Overhead per elements	none	1 or 2 links

Applications of linked list:

Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non zero coefficient and exponents. For example:

$$P(x) = a_0 X^n + a_1 X^{n-1} + \ldots\ldots + a_{n-1} X + a_n$$

Represent very large numbers and operations of the large number such as addition, multiplication and division.

Linked lists are to implement stack, queue, trees and graphs.

Implement the symbol table in compiler construction

Single Linked List:

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node. Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). The front of the list is a pointer to the "start" node.

A single linked list is shown in figure 3.2.1.

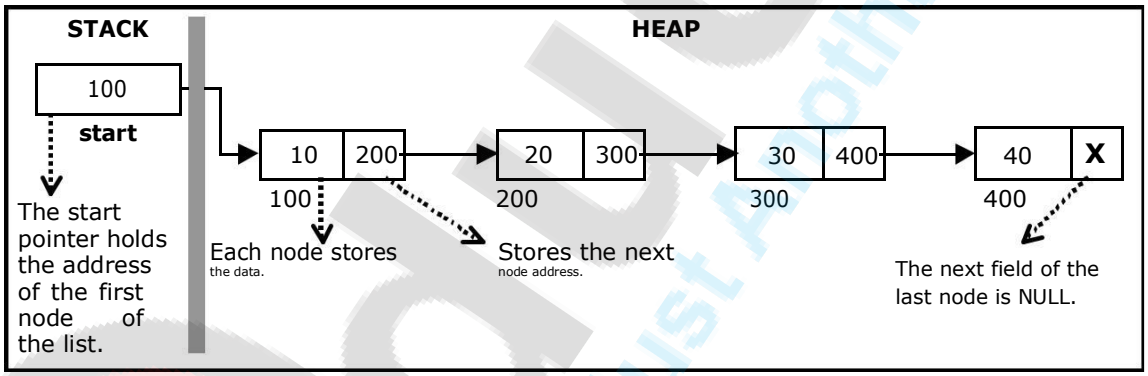


Figure 3.2.1. Single Linked List

The beginning of the linked list is stored in a "**start**" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the **start** and following the next pointers.

The **start** pointer is an ordinary local pointer variable, so it is drawn separately on the left top to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.

Implementation of Single Linked List:

Before writing the code to build the above list, we need to create a **start** node**,** used to create and access other nodes in the linked list. The following structure definition will do (see figure 3.2.2):

Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.

Initialise the start pointer to be NULL.

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

typedef struct slinklist node;

node *start = NULL;
```

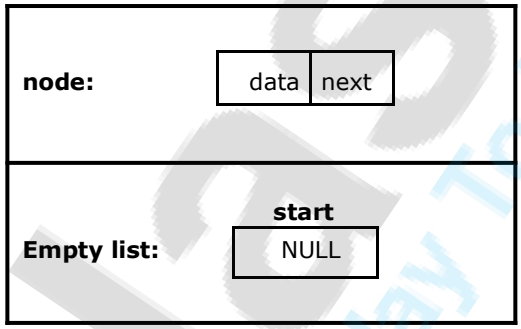


Figure 3.2.2. Structure definition, single link node and empty list

The basic operations in a single linked list are:

Creation.

Insertion.

Deletion.

Traversing.

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 (i.e., data) has to be read from the user, set next field to NULL and finally returns the address of the node. Figure 3.2.3 illustrates the creation of a node for single linked list.

```
node* getnode()
{
      node* newnode;
      newnode = (node *) malloc(sizeof(node));
      printf("\n Enter data: ");
      scanf("%d", &newnode -> data);
      newnode -> next = NULL;
      return newnode;
}
```

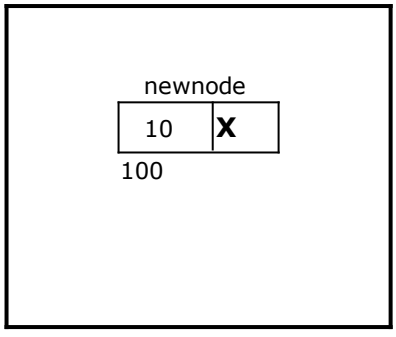


Figure 3.2.3. new node with a value of 10

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

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

Get the new node using getnode().
newnode = getnode();

If the list is empty, assign new node as start.
start = newnode;

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 (i.e. start node) in the list by assigning the address of the first node.

The start pointer is made to point the new node by assigning the address of the new node.

Repeat the above steps 'n' times.

Figure 3.2.4 shows 4 items in a single linked list stored at different locations in memory.

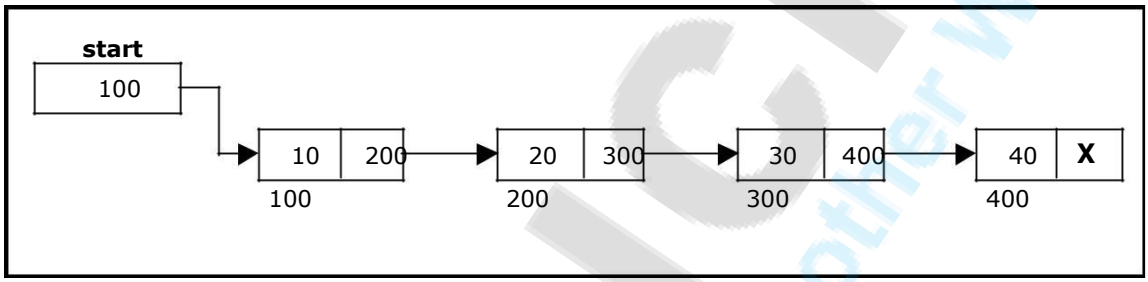


Figure 3.2.4. Single Linked List with 4 nodes

The function createlist(), is used to create 'n' number of nodes:

```
vo id createlist(int n)
{
        int i;
        no de * new no
        de; no de *tem p;
        for(i = 0; i < n ; i+ +)
        {
                new no de = getno de();
                if(start = = NULL)
                {
                        start = new no de;
                }
                else
                {
                        tem p = start;
                        w hile(tem p - > next != NULL)
                                tem p = tem p - > next;
                        tem p - > next = new no de;
                }
        }
}
```

Insertion of a Node:

One of the most primitive operations that can be done in a singly linked list is the insertion of a node. Memory is to be allocated for the new node (in a similar way that is done while 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 namely:

Inserting a node at the beginning.

Inserting a node at the end.

Inserting a node at intermediate position.

Inserting a node at the beginning:

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

Get the new node using getnode().
newnode = getnode();

If the list is empty then *start = newnode.*

If the list is not empty, follow the steps given below:
newnode -> next = start;
start = newnode;

Figure 3.2.5 shows inserting a node into the single linked list at the beginning.

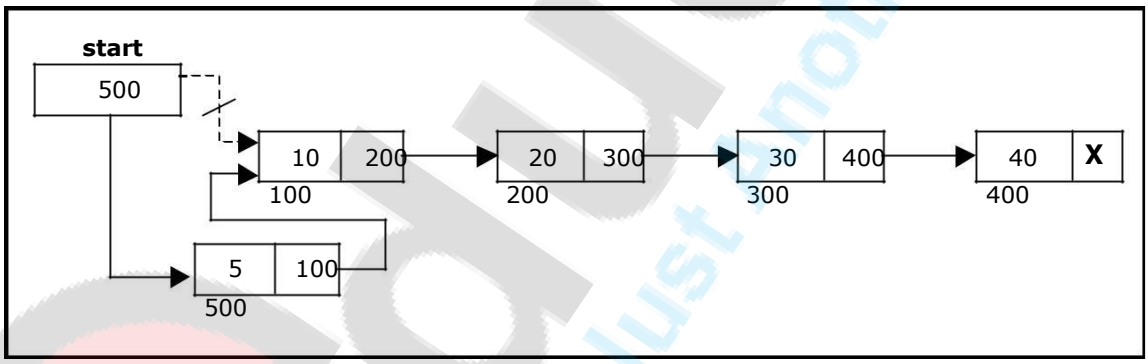


Figure 3.2.5. Inserting a node at the beginning

The function insert_at_beg(), is used for inserting a node at the beginning

```
void insert_at_beg()
{
        node *newnode;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
          newnode -> next = start;
          start = newnode;
        }
}
```

Inserting a node at the end:

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

```
Get the new node using getnode()
    newnode = getnode();

If the list is empty then start = newnode.

If the list is not empty follow the steps given below:
    temp = start;
    while(temp -> next != NULL)
            temp = temp -> next;
    temp -> next = newnode;
```

Figure 3.2.6 shows inserting a node into the single linked list at the end.

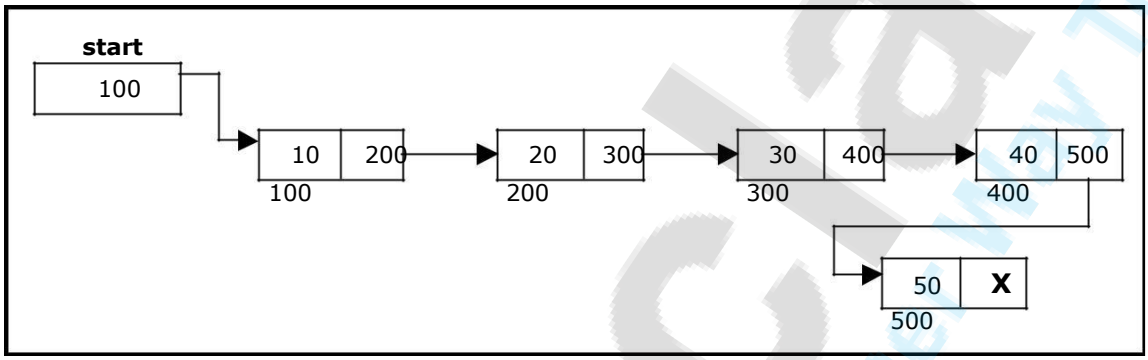


Figure 3.2.6. Inserting a node at the end.

The function insert_at_end(), is used for inserting a node at the end.

```
void insert_at_end()
{
        node *newnode, *temp;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                temp = start;
                while(temp -> next != NULL)
                        temp = temp -> next;
                temp -> next = newnode;
        }
}
```

Inserting a node at intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using getnode().

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

Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.

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

prev -> next = newnode;
newnode -> next = temp;

Let the intermediate position be 3.

Figure 3.2.7 shows inserting a node into the single linked list at a specified intermediate position other than beginning and end.

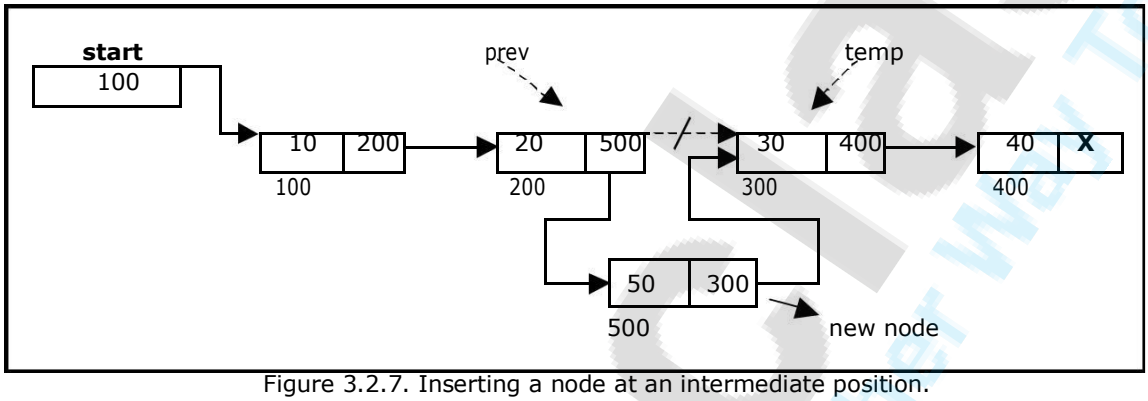


Figure 3.2.7. Inserting a node at an intermediate position.

The function insert_at_mid(), is used for inserting a node in the intermediate position.

```
void insert_at_mid()
{
        node *newnode, *temp, *prev;
        int pos, nodectr, ctr = 1;
        newnode = getnode();
        printf("\n Enter the position: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos > 1 && pos < nodectr)
        {
                temp = prev = start;
                while(ctr < pos)
                {
                        prev = temp;
                        temp = temp ->
                        next; ctr++;
                }
                prev -> next = newnode;
                newnode -> next = temp;
        }
        else
        {
                printf("position %d is not a middle position", pos);
        }
}
```

Deletion of a node:

Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.

Deleting a node at the beginning.

Deleting a node at the end.

Deleting a node at intermediate position.

Deleting a node at the beginning:

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

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

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

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

Figure 3.2.8 shows deleting a node at the beginning of a single linked list.

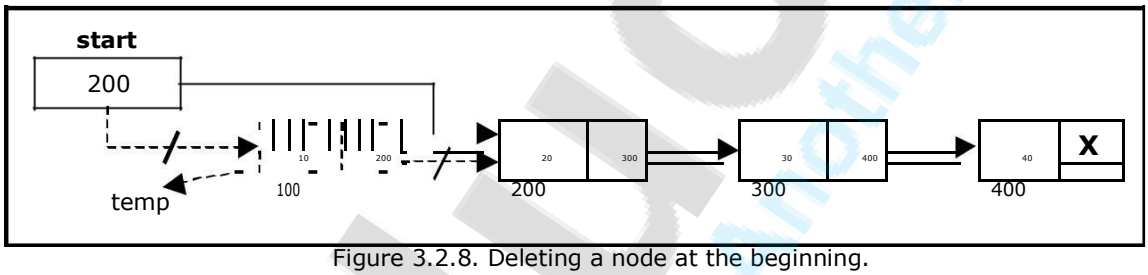


Figure 3.2.8. Deleting a node at the beginning.

The function delete_at_beg(), is used for deleting the first node in the list.

```
void delete_at_beg()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n No nodes are exist..");
                return ;
        }
        else
        {
                temp = start;
                start = temp -> next;
                free(temp);
                printf("\n Node deleted ");
        }
}
```

Deleting a node at the end:

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

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

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);
```

Figure 3.2.9 shows deleting a node at the end of a single linked list.

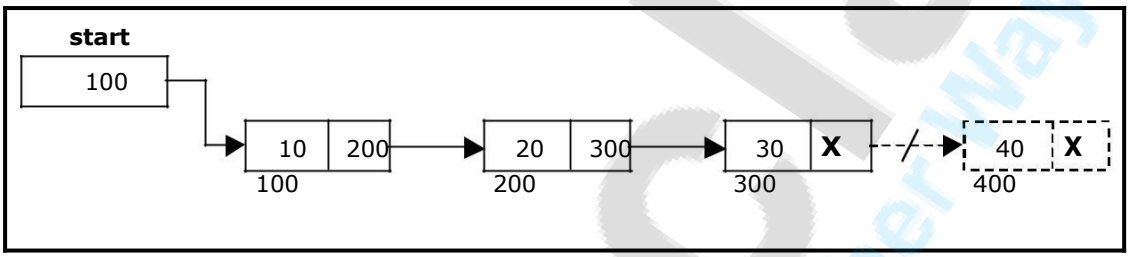


Figure 3.2.9. Deleting a node at the end.

The function delete_at_last(), is used for deleting the last node in the list.

```
void delete_at_last()
{
        node *temp, *prev;
        if(start == NULL)
        {
                printf("\n Empty ist..");
                return ;
        }
        else
        {
                temp = start;
                prev = start;
                while(temp -> next != NULL)
                {
                        prev = temp;
                        temp = temp -> next;
                }
                prev -> next = NULL;
                free(temp);
                printf("\n Node deleted ");
        }
}
```

Deleting a node at Intermediate position:

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

If list is empty then display 'Empty List' message

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..");
}
```

Figure 3.2.10 shows deleting a node at a specified intermediate position other than beginning and end from a single linked list.

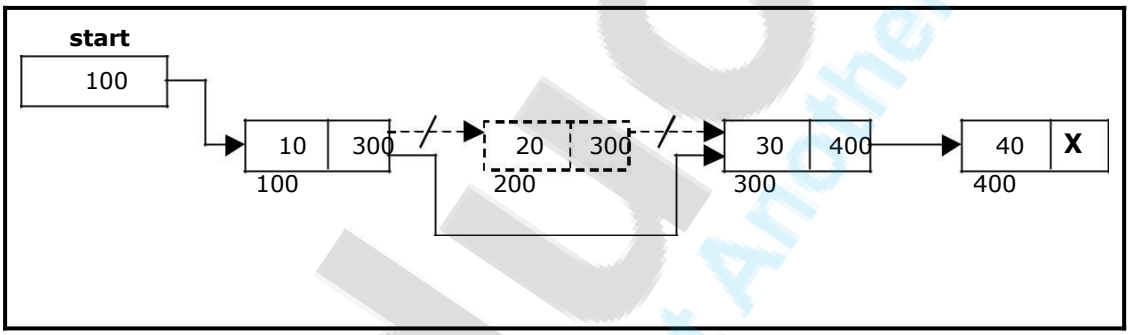


Figure 3.2.10. Deleting a node at an intermediate position.

The function delete_at_mid(), is used for deleting the intermediate node in the list.

```
void delete_at_mid()
{
        int ctr = 1, pos,
        nodectr; node *temp,
        *prev; if(start == NULL)
        {
                printf("\n Empty
                List.."); return ;
        }
        else
        {
                printf("\n Enter position of node to delete: ");
                scanf("%d", &pos);
                nodectr = countnode(start);
                if(pos > nodectr)
                {
                        printf("\nThis node doesnot exist");
                }
```

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

        }
}
```

Traversal and displaying a list (Left to Right):

To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached. Traversing a list involves the following steps:

Assign the address of start pointer to a temp pointer.

Display the information from the data field of each node.

The function *traverse*() is used for traversing and displaying the information stored in the list from left to right.

```
void traverse()
{
        node *temp;
        temp = start;
        printf("\n The contents of List (Left to Right):
        \n"); if(start == NULL )
                printf("\n Empty List");
        else
        {
                while (temp != NULL)
                {
                        printf("%d ->", temp ->
                        data); temp = temp -> next;
                }
        }
        printf("X");
}
```

Alternatively there is another way to traverse and display the information. That is in reverse order. The function rev_traverse(), is used for traversing and displaying the information stored in the list from right to left.

```
vo id rev_traverse(no de *st)
{
        if(st = = NULL)
        {
                return;
        }
        else
        {
                rev_traverse(st - > next);
                printf("%d - >", st - > data);
        }
}
```

Counting the Number of Nodes:

The following code will count the number of nodes exist in the list using *recursion*.

```
int co untno de(no de *st)
{
        if(st = = NULL)
                return 0;
        else
                return(1 + co untno de(st - > next));
}
```

Source Code for the Implementation of Single Linked List:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

struct slinklist
{
        int data;
        struct slinklist *next;
};

typedef struct slinklist node;

node *start = NULL;
int menu()
{
        int ch;
        clrscr();
        printf("\n 1.Create a list ");
        printf("\n-------------------------");
        printf("\n 2.Insert a node at beginning ");
        printf("\n 3.Insert a node at end");
        printf("\n 4.Insert a node at middle");
        printf("\n-------------------------");
        printf("\n 5.Delete a node from beginning");
        printf("\n 6.Delete a node from Last");
        printf("\n 7.Delete a node from Middle");
        printf("\n-------------------------");
        printf("\n 8.Traverse the list (Left to Right)");
        printf("\n 9.Traverse the list (Right to Left)");
```

```
        printf("\n--------------------------");
        printf("\n 10. Count nodes ");
        printf("\n 11. Exit ");
        printf("\n\n Enter your choice: ");
        scanf("%d",&ch);
        return ch;
}

node* getnode()
{
        node * newnode;
        newnode = (node *) malloc(sizeof(node));
        printf("\n Enter data: ");
        scanf("%d", &newnode -> data);
        newnode -> next = NULL; return
        newnode;
}

int countnode(node *ptr)
{
        int count=0;
        while(ptr != NULL)
        {
                count++;
                ptr = ptr -> next;
        }
        return (count);
}

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 -> next != NULL)
                                temp = temp -> next;
                        temp -> next = newnode;
                }
        }
}

void traverse()
{
        node *temp;
        temp = start;
        printf("\n The contents of List (Left to Right): \n");
        if(start == NULL)
        {
                printf("\n Empty List");
                return;
        }
        else
        {
```

```
                while(temp != NULL)
                {
                        printf("%d-->", temp ->
                        data); temp = temp -> next;
                }
        }
        printf(" X ");
}

void rev_traverse(node *start)
{
        if(start == NULL)
        {
                return;
        }
        else
        {
                rev_traverse(start -> next);
                printf("%d -->", start -> data);
        }
}

void insert_at_beg()
{
        node *newnode;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                newnode -> next =
                start; start = newnode;
        }
}

void insert_at_end()
{
        node *newnode, *temp;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                temp = start;
                while(temp -> next != NULL)
                        temp = temp -> next;
                temp -> next = newnode;
        }
}

void insert_at_mid()
{
        node *newnode, *temp, *prev;
        int pos, nodectr, ctr = 1;
        newnode = getnode();
        printf("\n Enter the position: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
```

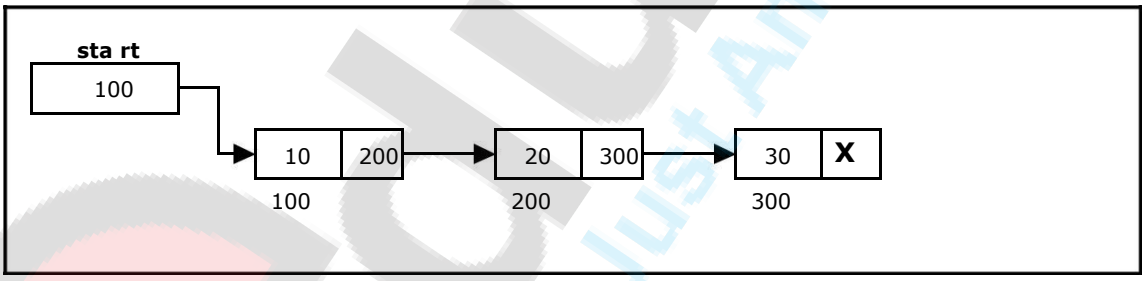
```
        if(pos > 1 && pos < nodectr)
        {
                temp = prev = start;
                while(ctr < pos)
                {
                        prev = temp;
                        temp = temp ->
                        next; ctr++;
                }
                prev -> next = newnode;
                newnode -> next = temp;
        }
        else
                printf("position %d is not a middle position", pos);
}

void delete_at_beg()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n No nodes are exist..");
                return ;
        }
        else
        {
                temp = start;
                start = temp -> next;
                free(temp);
                printf("\n Node deleted ");
        }
}

void delete_at_last()
{
        node *temp, *prev;
        if(start == NULL)
        {
                printf("\n Empty
                List.."); return ;
        }
        else
        {
                temp = start;
                prev = start;
                while(temp -> next != NULL)
                {
                        prev = temp;
                        temp = temp -> next;
                }
                prev -> next = NULL;
                free(temp);
                printf("\n Node deleted ");
        }
}

void delete_at_mid()
{
        int ctr = 1, pos,
        nodectr; node *temp,
        *prev; if(start == NULL)
        {
                printf("\n Empty List..");
```

```
                return ;
        }
        else
        {
                printf("\n Enter position of node to delete: ");
                scanf("%d", &pos);
                nodectr = countnode(start);
                if(pos > nodectr)
                {
                        printf("\nThis node doesnot exist");

                }
                if(pos > 1 && pos < nodectr)
                {
                        temp = prev = start;
                        while(ctr < pos)
                        {
                                prev = temp;
                                temp = temp ->
                                next; ctr ++;
                        }
                        prev -> next = temp -> next;
                        free(temp);
                        printf("\n Node deleted..");
                }
                else
                {
                        printf("\n Invalid position..");
                        getch();
                }
        }
}

void main(void)
{
        int ch, n;
        clrscr();
        while(1)
        {
                ch = menu();
                switch(ch)
                {
                case 1:
                        if(start == NULL)
                        {
                                printf("\n Number of nodes you want to create: ");
                                scanf("%d", &n);
                                createlist(n);
                                printf("\n List created..");
                        }
                        else
                                printf("\n List is already created..");
                                break;
                case 2:
                        insert_at_beg();
                        break;
                case 3:
                        insert_at_end();
                        break;
                case 4:
                        insert_at_mid();
                        break;
```

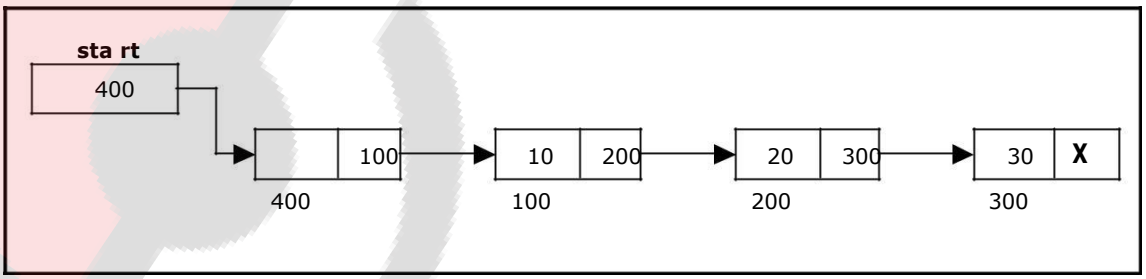
```
                case 5:
                        delete_at_beg();
                        break;
                case 6:
                        delete_at_last();
                        break;
                case 7:
                        delete_at_mid();
                        break;
                case 8:
                        traverse();
                        break;
                case 9:
                        printf("\n The contents of List (Right to Left): \n");
                        rev_traverse(start);
                        printf(" X ");
                        break;
                case 10:
                        printf("\n No of nodes : %d ", countnode(start));
                        break;
                case 11 :
                        exit(0);
                }
                getch();
        }
}
```

Using a header node:

A header node is a special dummy node found at the front of the list. The use of header node is an alternative to remove the first node in a list. For example, the picture below shows how the list with data 10, 20 and 30 would be represented using a linked list without and with a header node:



Single Linke d List w it ho ut a he a der no de



Single Linke d List w it h he a der no de

Note that if your linked lists do include a header node, there is no need for the special case code given above for the *remove* operation; node ***n*** can never be the first node in the list, so there is no need to check for that case. Similarly, having a header node can simplify the code that adds a node before a given node ***n***.

Note that if you do decide to use a header node, you must remember to initialize an empty list to contain one (dummy) node, you must remember not to include the header node in the count of "real" nodes in the list.

It is also useful when information other than that found in each node of the list is needed. For example, imagine an application in which the number of items in a list is often calculated. In a standard linked list, the list function to count the number of nodes has to traverse the entire list every time. However, if the current length is maintained in a header node, that information can be obtained very quickly.

Array based linked lists:

Another alternative is to allocate the nodes in blocks. In fact, if you know the maximum size of a list a head of time, you can pre-allocate the nodes in a single array. The result is a hybrid structure – an array based linked list. Figure 3.5.1 shows an example of null terminated single linked list where all the nodes are allocated contiguously in an array.

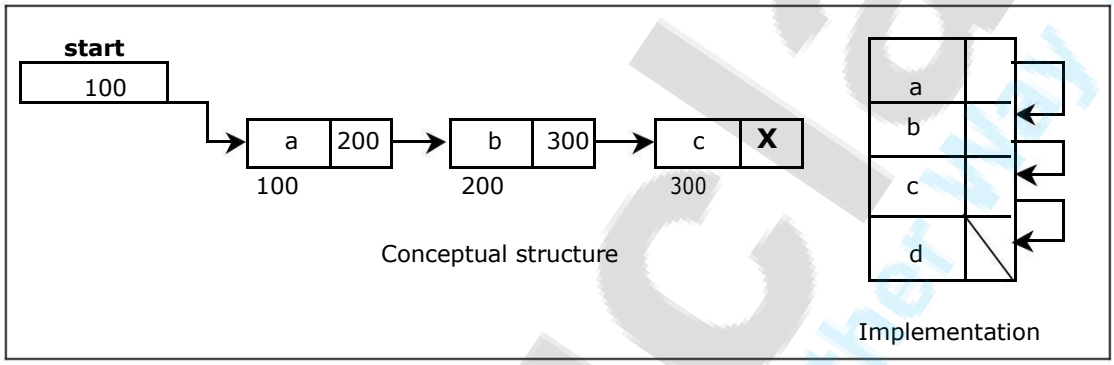


Figure 3.5.1. An array based linked list

Double Linked List:

A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi-directional traversing. Each node contains three fields:

Left link.
Data.
Right link.

The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data.

Many applications require searching forward and backward thru nodes of a list. For example searching for a name in a telephone directory would need forward and backward scanning thru a region of the whole list.

The basic operations in a double linked list are:

Creation.
Insertion.
Deletion.
Traversing.

A double linked list is shown in figure 3.3.1.

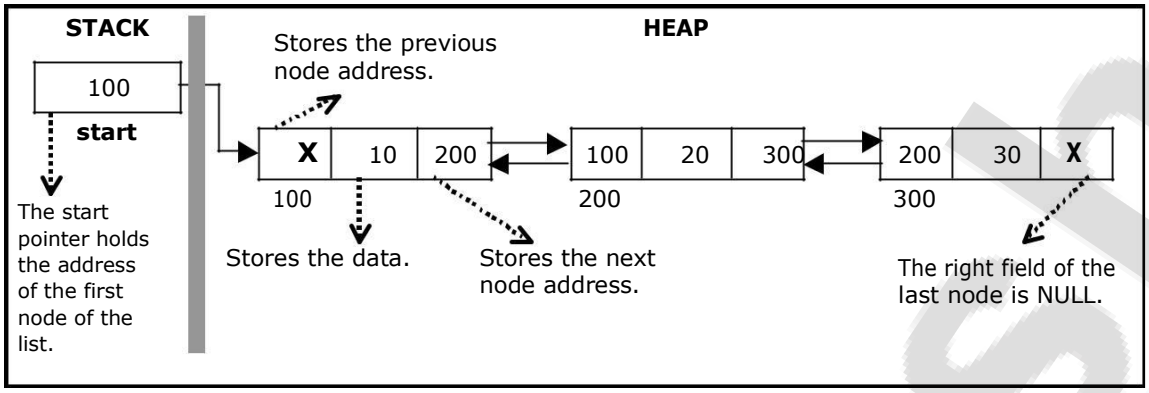


Figure 3.3.1. Double Linked List

The beginning of the double linked list is stored in a "**start**" pointer which points to the first node. The first node's left link and last node's right link is set to NULL.

The following code gives the structure definition:

```
struct dlinklist
{
        struct dlinklist *left;
        int data;
        struct dlinklist *right;

};
typedef struct dlinklist node;
node *start = NULL;
```

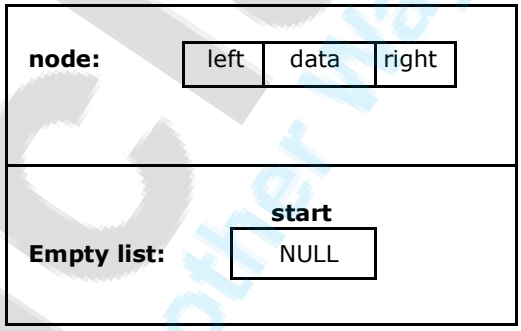


Figure 3.4.1. Structure definition, double link node and empty list

Creating a node for Double Linked List:

Creating a double 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 (i.e., data) has to be read from the user and set left field to NULL and right field also set to NULL (see figure 3.2.2).

```
node* getnode()
{
      node* newnode;
      newnode = (node *) malloc(sizeof(node));
      printf("\n Enter data: ");
      scanf("%d", &newnode -> data);
      newnode -> left = NULL;
      newnode -> right = NULL;
      return newnode;
}
```

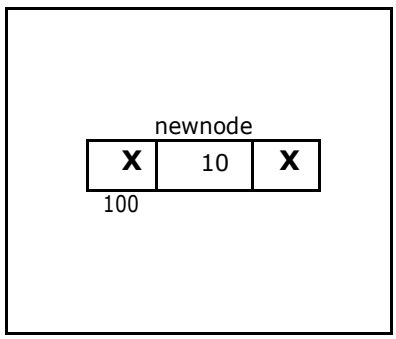


Figure 3.4.2. new node with a value of 10

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

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

Get the new node using getnode().

newnode =getnode();

If the list is empty then *start* = *newnode*.

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.

Repeat the above steps 'n' times.

The function createlist(), is used to create 'n' number of nodes:

```
void createlist(int n)
{
        int i;
        node * new node;
        node *temp;
        for(i = 0; i < n; i+ +)
        {
                new node = getnode();
                if(start = = NULL)
                {
                        start = new no de;
                }
                else
                {
                        tem p = start;
                        w hile(tem p - > right)
                                tem p = tem p - > right;
                        tem p - > right = new no de; new
                        no de - > left = tem p;
                }
        }
}
```

Figure 3.4.3 shows 3 items in a double linked list stored at different locations.

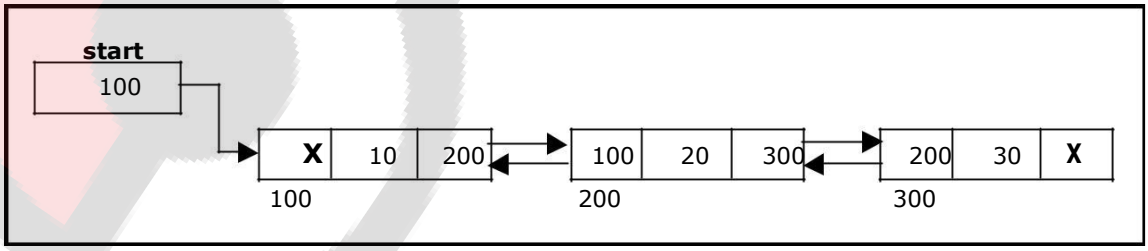


Figure 3.4.3. Double Linked List with 3 nodes

Inserting a node at the beginning:

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

Get the new node using getnode().

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

If the list is empty then *start = newnode*.

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

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

The function dbl_insert_beg(), is used for inserting a node at the beginning. Figure 3.4.4 shows inserting a node into the double linked list at the beginning.

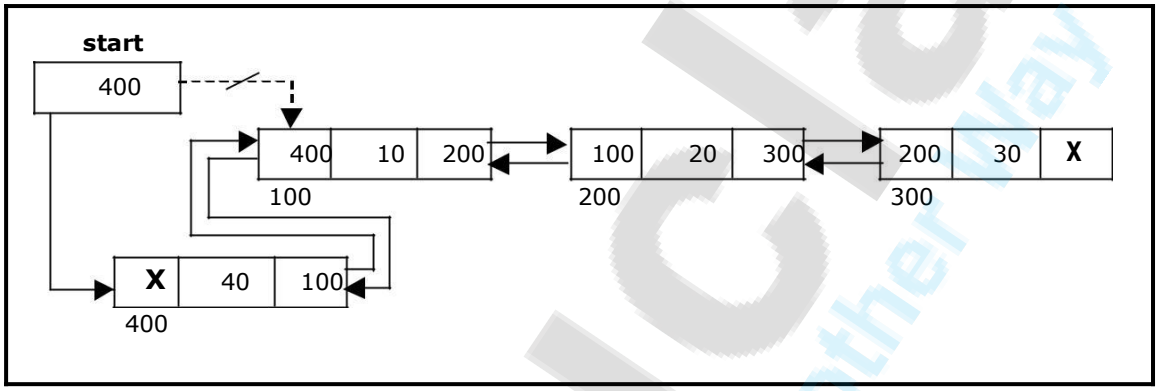


Figure 3.4.4. Inserting a node at the beginning

Inserting a node at the end:

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

Get the new node using getnode()

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

If the list is empty then *start = newnode*.

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;
```

The function dbl_insert_end(), is used for inserting a node at the end. Figure 3.4.5 shows inserting a node into the double linked list at the end.

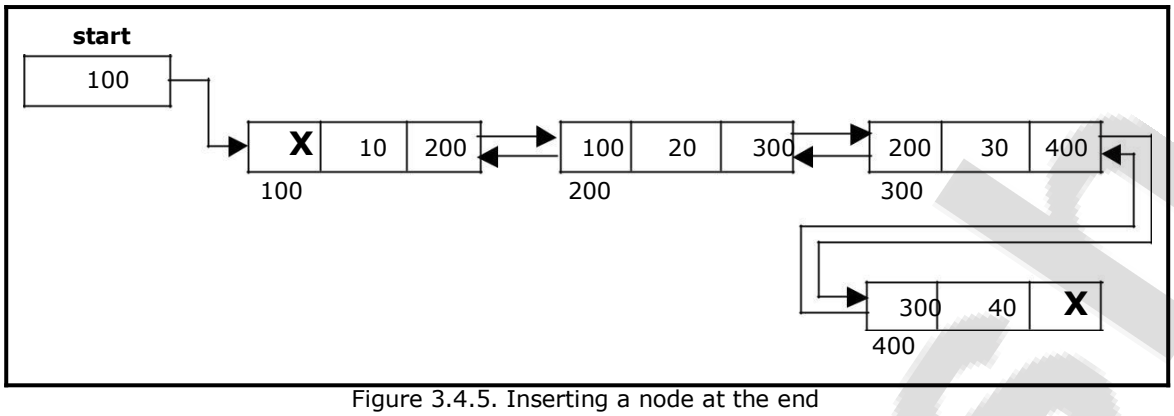


Figure 3.4.5. Inserting a node at the end

Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

Get the new node using getnode().

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

Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.

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

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

The function dbl_insert_mid(), is used for inserting a node in the intermediate position. Figure 3.4.6 shows inserting a node into the double linked list at a specified intermediate position other than beginning and end.

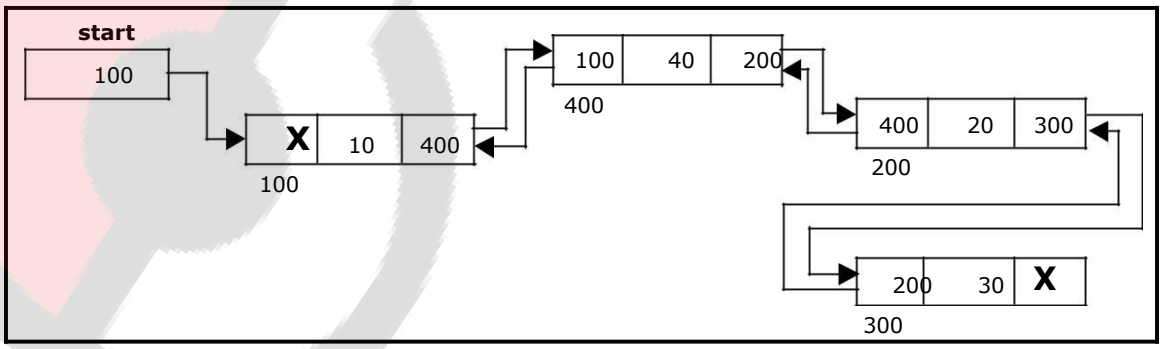


Figure 3.4.6. Inserting a node at an intermediate position

Deleting a node at the beginning:

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

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

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

```
temp = start;
start = start -> right;
start -> left = NULL;
free(temp);
```

The function dbl_delete_beg(), is used for deleting the first node in the list. Figure 3.4.6 shows deleting a node at the beginning of a double linked list.

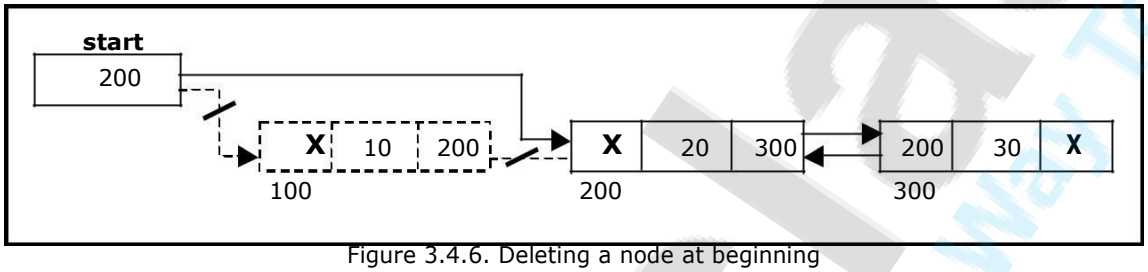


Figure 3.4.6. Deleting a node at beginning

Deleting a node at the end:

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

If list is empty then display 'Empty List' message

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

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

The function dbl_delete_last(), is used for deleting the last node in the list. Figure 3.4.7 shows deleting a node at the end of a double linked list.

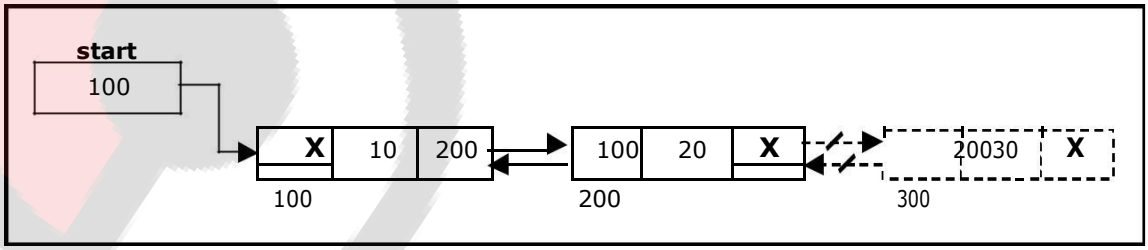


Figure 3.4.7. Deleting a node at the end

Deleting a node at Intermediate position:

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

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

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

The function delete_at_mid(), is used for deleting the intermediate node in the list. Figure 3.4.8 shows deleting a node at a specified intermediate position other than beginning and end from a double linked list.

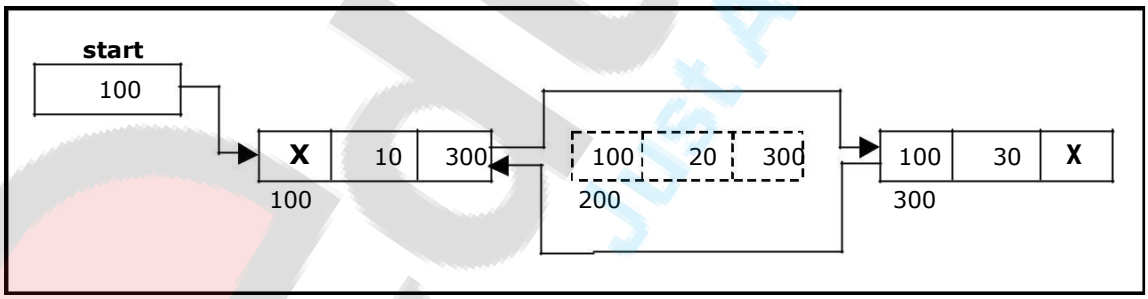


Figure 3.4.8 Deleting a node at an intermediate position

Traversal and displaying a list (Left to Right):

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 function *traverse_left_right*() is used for traversing and displaying the information stored in the list from left to right.

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

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

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

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

Traversal and displaying a list (Right to Left):

To display the information from right to left, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function *traverse_right_left*() is used for traversing and displaying the information stored in the list from right to left. The following steps are followed, to traverse a list from right to left:

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

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

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

Counting the Number of Nodes:

The following code will count the number of nodes exist in the list (using recursion).

```
int co untno de(no de *start)
{
        if(start = = NULL)
                return 0;
        else
                return(1 + co untno de(start - >right ));
}
```

A Complete Source Code for the Implementation of Double Linked List:

```
#include  <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct dlinklist
{
        struct dlinklist *left;
        int data;
        struct dlinklist *right;
};

typedef struct dlinklist node;
node *start = NULL;
```

```
node* getnode()
{
        node * newnode;
        newnode = (node *) malloc(sizeof(node));
        printf("\n Enter data: ");
        scanf("%d", &newnode -> data);
        newnode -> left = NULL;
        newnode -> right = NULL;
        return newnode;
}

int countnode(node *start)
{
        if(start == NULL)
                return 0;
        else
                return 1 + countnode(start -> right);
}

int menu()
{
        int ch;
        clrscr();
        printf("\n 1.Create");
        printf("\n-----------------------------");
        printf("\n 2. Insert a node at beginning ");
        printf("\n 3. Insert a node at end");
        printf("\n 4. Insert a node at middle");
        printf("\n-----------------------------");
        printf("\n 5. Delete a node from beginning");
        printf("\n 6. Delete a node from Last");
        printf("\n 7. Delete a node from Middle");
        printf("\n-----------------------------");
        printf("\n 8. Traverse the list from Left to Right
        "); printf("\n 9. Traverse the list from Right to
        Left "); printf("\n-----------------------------");
        printf("\n 10.Count the Number of nodes in the list");
        printf("\n 11.Exit ");
        printf("\n\n Enter your choice: ");
        scanf("%d", &ch);
        return ch;
}

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;
                }
        }
}
```

```
void traverse_left_to_right()
{
        node *temp;
        temp = start;
        printf("\n The contents of List:
        "); if(start == NULL )
                printf("\n Empty List");
        else
        {
                while(temp != NULL)
                {
                        printf("\t %d ", temp -> data);
                        temp = temp -> right;
                }
        }
}
void traverse_right_to_left()
{
        node *temp;
        temp = start;
        printf("\n The contents of List:
        "); if(start == NULL)
                printf("\n Empty List");
        else
        {
                while(temp -> right != NULL)
                        temp = temp -> right;
        }
        while(temp != NULL)
        {
                printf("\t%d", temp ->
                data); temp = temp -> left;
        }
}
void dll_insert_beg()
{
        node *newnode;
        newnode = getnode();
        if(start == NULL)
                start = newnode;
        else
        {
                newnode -> right = start;
                start -> left = newnode;
                start = newnode;
        }
}

void dll_insert_end()
{
        node *newnode, *temp;
        newnode = getnode();
        if(start == NULL)
                start = newnode;
        else
        {
                temp = start;
                while(temp -> right != NULL)
                        temp = temp -> right;
                temp -> right = newnode;
                newnode -> left = temp;
        }
}
```

```
void dll_insert_mid()
{
        node *newnode,*temp;
        int pos, nodectr, ctr = 1;
        newnode = getnode();
        printf("\n Enter the position: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos - nodectr >= 2)
        {
                printf("\n Position is out of range..");
                return;
        }
        if(pos > 1 && pos < nodectr)
        {
                temp = start;
                while(ctr < pos - 1)
                {
                        temp = temp -> right;
                        ctr++;
                }
                newnode -> left = temp; newnode
                -> right = temp -> right; temp ->
                right -> left = newnode; temp ->
                right = newnode;
        }
        else
                printf("position %d of list is not a middle position ", pos);
}


void dll_delete_beg()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n Empty
                list"); getch();
                return ;
        }
        else
        {
                temp = start;
                start = start -> right;
                start -> left = NULL;
                free(temp);
        }
}


void dll_delete_last()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n Empty
                list"); getch();
                return ;
        }
        else
        {
                temp = start;
                while(temp -> right != NULL)
```

```
                        temp = temp -> right;
                temp -> left -> right = NULL;
                free(temp);
                temp = NULL;
        }
}

void dll_delete_mid()
{
        int i = 0, pos, nodectr;
        node *temp;
        if(start == NULL)
        {
                printf("\n Empty List");
                getch();
                return;
        }
        else
        {
                printf("\n Enter the position of the node to delete: ");
                scanf("%d", &pos);
                nodectr = countnode(start);
                if(pos > nodectr)
                {
                        printf("\nthis node does not
                        exist"); getch();
                        return;
                }
                if(pos > 1 && pos < nodectr)
                {
                        temp =
                        start; i = 1;
                        while(i < pos)
                        {
                                temp = temp -> right;
                                i++;
                        }
                        temp -> right -> left = temp -> left;
                        temp -> left -> right = temp -> right;
                        free(temp);
                        printf("\n node deleted..");
                }
                else
                {
                        printf("\n It is not a middle position..");
                        getch();
                }
        }
}

void main(void)
{
        int ch, n;
        clrscr();
        while(1)
        {
                ch = menu();
                switch( ch)
                {
                        case 1 :
                                printf("\n Enter Number of nodes to create: ");
                                scanf("%d", &n);
                                createlist(n);
```

```
                        printf("\n List
                        created.."); break;
                case 2 :
                        dll_insert_beg();
                        break;
                case 3 :
                        dll_insert_end();
                        break;
                case 4 :
                        dll_insert_mid();
                        break;
                case 5 :
                        dll_delete_beg();
                        break;
                case 6 : dll_delete_last();
                        break;

                case 7 :
                        dll_delete_mid();
                        break;
                case 8 :
                        traverse_left_to_right();
                        break;
                case 9 :
                        traverse_right_to_left();
                        break;

                case 10 :
                        printf("\n Number of nodes: %d", countnode(start));
                        break;
                case 11:
                        exit(0);
                }
                getch();
        }
}
```

Circular Single Linked List:

It is just a single linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called *start* pointer always pointing to the first node of the list. Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.

A circular single linked list is shown in figure 3.6.1.

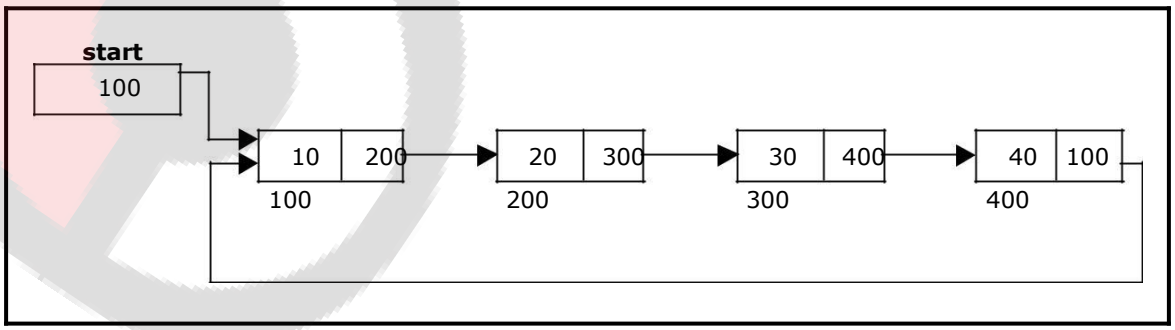


Figure 3.6.1. Circular Single Linked List

The basic operations in a circular single linked list are:

Creation.
Insertion.
Deletion.
Traversing.

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

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

Get the new node using getnode().

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

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

```
start = newnode;
```

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

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

Repeat the above steps 'n' times.

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

The function createlist(), is used to create 'n' number of nodes:

Inserting a node at the beginning:

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

Get the new node using getnode().

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

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

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

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

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

The function cll_insert_beg(), is used for inserting a node at the beginning. Figure 3.6.2 shows inserting a node into the circular single linked list at the beginning.

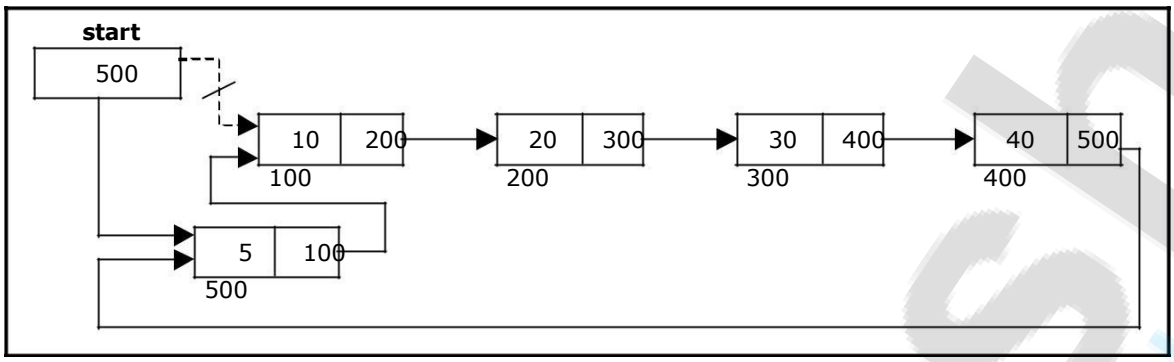


Figure 3.6.2. Inserting a node at the beginning

Inserting a node at the end:

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

Get the new node using getnode().

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

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

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

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

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

The function cll_insert_end(), is used for inserting a node at the end.

Figure 3.6.3 shows inserting a node into the circular single linked list at the end.

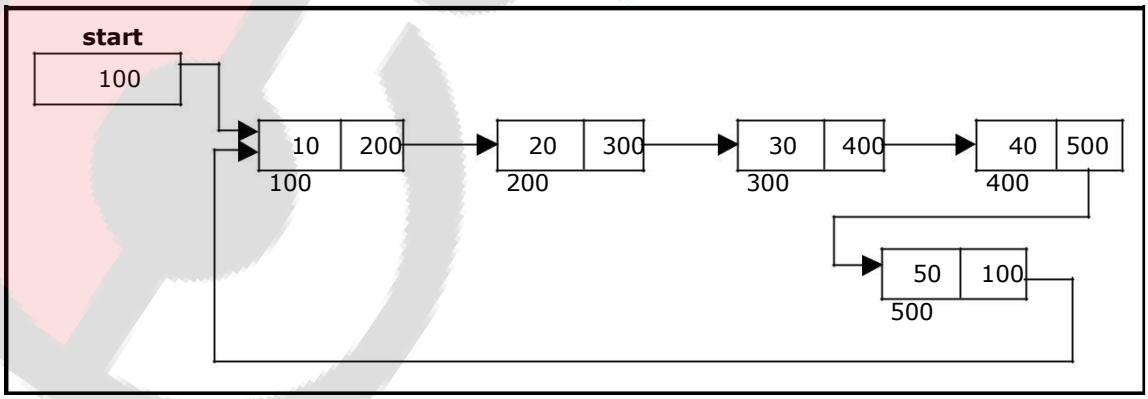


Figure 3.6.3 Inserting a node at the end.

Deleting a node at the beginning:

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

If the list is empty, display a message 'Empty List'.

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

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

After deleting the node, if the list is empty then *start = NULL.*

The function cll_delete_beg(), is used for deleting the first node in the list. Figure 3.6.4 shows deleting a node at the beginning of a circular single linked list.

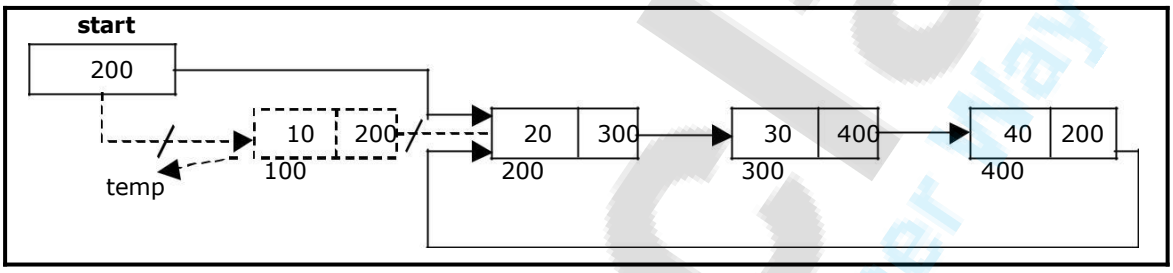


Figure 3.6.4. Deleting a node at beginning.

Deleting a node at the end:

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

If the list is empty, display a message 'Empty List'.

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

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

After deleting the node, if the list is empty then *start = NULL.*

The function cll_delete_last(), is used for deleting the last node in the list.

Figure 3.6.5 shows deleting a node at the end of a circular single linked list.

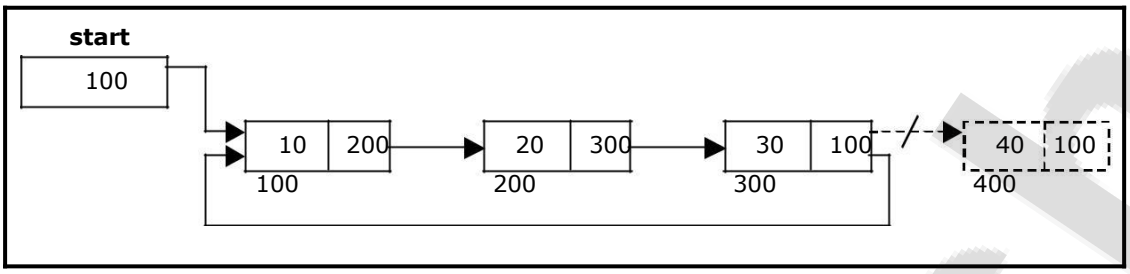


Figure 3.6.5. Deleting a node at the end.

Traversing a circular single linked list from left to right:

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

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

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

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

Source Code for Circular Single Linked List:

```
include <stdio.h>
include <conio.h>
include <stdlib.h>

struct cslinklist
{
        int data;
        struct cslinklist *next;
};

typedef struct cslinklist node;

node *start = NULL;

int nodectr;

node* getnode()
{
        node * newnode;
        newnode = (node *) malloc(sizeof(node));
        printf("\n Enter data: ");
        scanf("%d", &newnode -> data);
        newnode -> next = NULL; return
        newnode;
}
```

```
int menu()
{
        int ch;
        clrscr();
        printf("\n 1. Create a list ");
        printf("\n\n-------------------------");
        printf("\n 2. Insert a node at beginning ");
        printf("\n 3. Insert a node at end");
        printf("\n 4. Insert a node at middle");
        printf("\n\n-------------------------");
        printf("\n 5. Delete a node from beginning");
        printf("\n 6. Delete a node from Last");
        printf("\n 7. Delete a node from Middle");
        printf("\n\n-------------------------");
        printf("\n 8. Display the list");
        printf("\n 9. Exit");
        printf("\n\n-------------------------");
        printf("\n Enter your choice:
        "); scanf("%d", &ch);
        return ch;
}

void createlist(int n)
{
        int i;
        node *newnode;
        node *temp;
        nodectr = n;
        for(i = 0; i < n ; i++)
        {
                newnode = getnode();
                if(start == NULL)
                {
                        start = newnode;
                }
                else
                {
                        temp = start;
                        while(temp -> next != NULL)
                                temp = temp -> next;
                        temp -> next = newnode;
                }
        }
        newnode ->next = start;       /* last node is pointing to starting node */
}

void display()
{
        node *temp;
        temp = start;
        printf("\n The contents of List (Left to Right): ");
        if(start == NULL )
                printf("\n Empty List");
        else
        {
                do
                {
                        printf("\t %d ", temp -> data);
                        temp = temp -> next;
                } while(temp !=
                start); printf(" X ");
        }
}
```

```
void cll_insert_beg()
{
        node *newnode, *last;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode; newnode
                -> next = start;
        }
        else
        {
                last = start;
                while(last -> next != start)
                        last = last -> next;
                newnode -> next =
                start; start = newnode;
                last -> next = start;
        }
        printf("\n Node inserted at beginning..");
        nodectr++;
}

void cll_insert_end()
{
        node *newnode, *temp;
        newnode = getnode();
        if(start == NULL )
        {
                start = newnode; newnode
                -> next = start;
        }
        else
        {
                temp = start;
                while(temp -> next != start)
                        temp = temp -> next;
                temp -> next = newnode;
                newnode -> next = start;
        }
        printf("\n Node inserted at end..");
        nodectr++;
}

void cll_insert_mid()
{
        node *newnode, *temp, *prev;
        int i, pos ;
        newnode = getnode(); printf("\n
        Enter the position: ");
        scanf("%d", &pos);
        if(pos > 1 && pos < nodectr)
        {
                temp        =
                start; prev =
                temp; i = 1;
                while(i < pos)
                {
                        prev = temp;
                        temp = temp ->
                        next; i++;
                }
                prev -> next = newnode;
                newnode -> next = temp;
```

```
                nodectr++;
                printf("\n Node inserted at middle..");
        }
        else
        {
                printf("position %d of list is not a middle position ", pos);
        }
}

void cll_delete_beg()
{
        node *temp, *last;
        if(start == NULL)
        {
                printf("\n No nodes
                exist.."); getch();
                return ;
        }
        else
        {
                last = temp = start;
                while(last -> next != start)
                        last = last -> next;
                start = start -> next;
                last -> next = start;
                free(temp);
                nodectr--;
                printf("\n Node deleted..");
                if(nodectr == 0)
                        start = NULL;
        }
}

void cll_delete_last()
{
        node *temp,*prev;
        if(start == NULL)
        {
                printf("\n No nodes
                exist.."); getch();
                return ;
        }
        else
        {
                temp = start;
                prev = start;
                while(temp -> next != start)
                {
                        prev = temp;
                        temp = temp -> next;
                }
                prev -> next = start;
                free(temp); nodectr-
                -;
                if(nodectr == 0) start
                        = NULL;
                printf("\n Node deleted..");
        }
}
```

```
void cll_delete_mid()
{
        int i = 0, pos;
        node *temp, *prev;

        if(start == NULL)
        {
                printf("\n No nodes
                exist.."); getch();
                return ;
        }
        else
        {
                printf("\n Which node to delete: ");
                scanf("%d", &pos);
                if(pos > nodectr)
                {
                        printf("\nThis node does not
                        exist"); getch();
                        return;
                }
                if(pos > 1 && pos < nodectr)
                {
                        temp=start;
                        prev = start;
                        i = 0;
                        while(i < pos - 1)
                        {
                                prev = temp;
                                temp = temp -> next ;
                                i++;
                        }
                        prev -> next = temp -> next;
                        free(temp);
                        nodectr--;
                        printf("\n Node Deleted..");
                }
                else
                {
                        printf("\n It is not a middle position..");
                        getch();
                }
        }
}

void main(void)
{
        int result;
        int ch, n;
        clrscr();
        while(1)
        {
                ch = menu();
                switch(ch)
                {
                        case 1 :
                                if(start == NULL)
                                {
                                        printf("\n Enter Number of nodes to create: ");
                                        scanf("%d", &n);
                                        createlist(n);
                                        printf("\nList created..");
                                }
```

```
                        else
                                printf("\n List is already Exist..");
                        break;
                case 2 :
                        cll_insert_beg();
                        break;
                case 3 :
                        cll_insert_end();
                        break;
                case 4 :
                        cll_insert_mid();
                        break;
                case 5 :
                        cll_delete_beg();
                        break;
                case 6 : cll_delete_last();
                        break;

                case 7 :
                        cll_delete_mid();
                        break;
                case 8 :
                        display();
                        break;
                case 9 :
                        exit(0);
                }
                getch();
        }
}
```

Circular Double Linked List:

A circular double linked list has both successor pointer and predecessor pointer in circular manner. The objective behind considering circular double linked list is to simplify the insertion and deletion operations performed on double linked list. In circular double linked list the *right* link of the right most node points back to the *start* node and *left* link of the first node points to the last node. A circular double linked list is shown in figure 3.8.1.

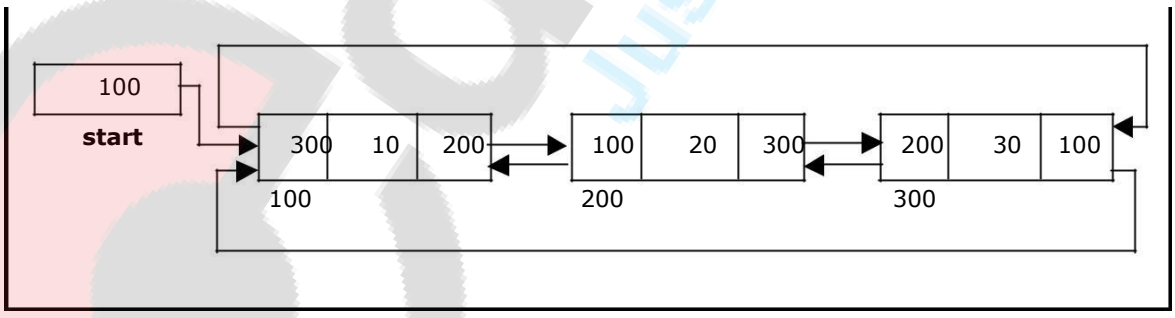


Figure 3.8.1. Circular Double Linked List

The basic operations in a circular double linked list are:

Creation.
Insertion.
Deletion.
Traversing.

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

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

Get the new node using getnode().
newnode = getnode();

If the list is empty, then do the following
start = newnode;
newnode -> left = start;
newnode ->right = start;

If the list is not empty, follow the steps given below:
newnode -> left = start -> left;
newnode -> right = start;
start -> left->right = newnode;
start -> left = newnode;

Repeat the above steps 'n' times.

The function cdll_createlist(), is used to create 'n' number of nodes:

Inserting a node at the beginning:

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

Get the new node using getnode().
newnode=getnode();

If the list is empty, then
start = newnode;
newnode -> left = start;
newnode -> right = start;

- If the list is not empty, follow the steps given below:
newnode -> left = start -> left;
newnode -> right = start;
start -> left -> right = newnode;
start -> left = newnode;
start = newnode;

The function cdll_insert_beg(), is used for inserting a node at the beginning. Figure 3.8.2 shows inserting a node into the circular double linked list at the beginning.

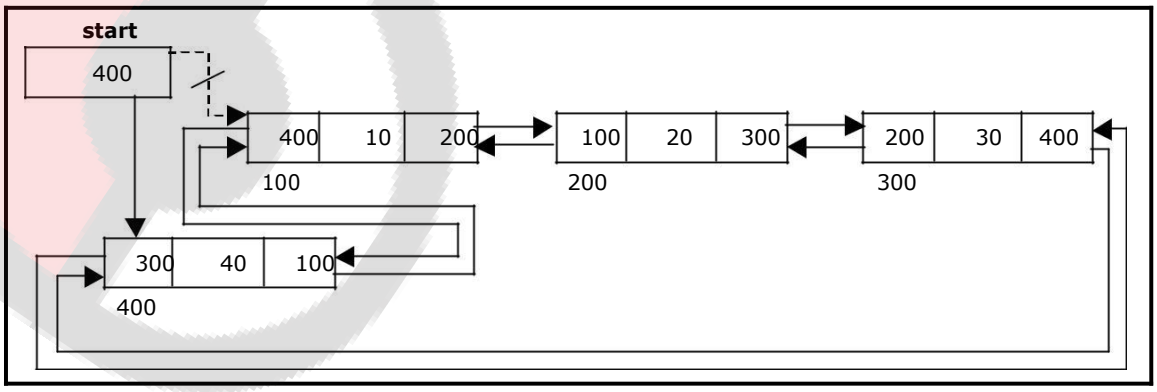


Figure 3.8.2. Inserting a node at the beginning

Inserting a node at the end:

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

Get the new node using getnode()
newnode=getnode();

If the list is empty, then
start = newnode;
newnode -> left = start;
newnode -> right = start;

- If the list is not empty follow the steps given below:
  newnode -> left = start -> left;
  newnode -> right = start;
  start -> left -> right = newnode;
  start -> left = newnode;

The function cdll_insert_end(), is used for inserting a node at the end. Figure 3.8.3 shows inserting a node into the circular linked list at the end.

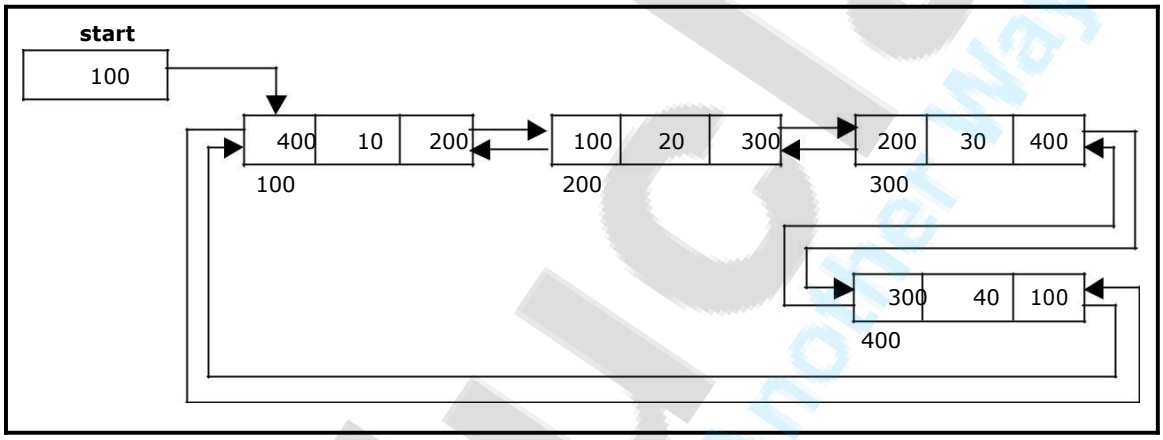


Figure 3.8.3. Inserting a node at the end

Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

Get the new node using getnode().
newnode=getnode();

Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

Store the starting address (which is in start pointer) in temp. Then traverse the temp pointer upto the specified position.

After reaching the specified position, follow the steps given below:
newnode -> left = temp; newnode
-> right = temp -> right; temp ->
right -> left = newnode; temp ->
right = newnode; nodectr++;

The function cdll_insert_mid(), is used for inserting a node in the intermediate position. Figure 3.8.4 shows inserting a node into the circular double linked list at a specified intermediate position other than beginning and end.

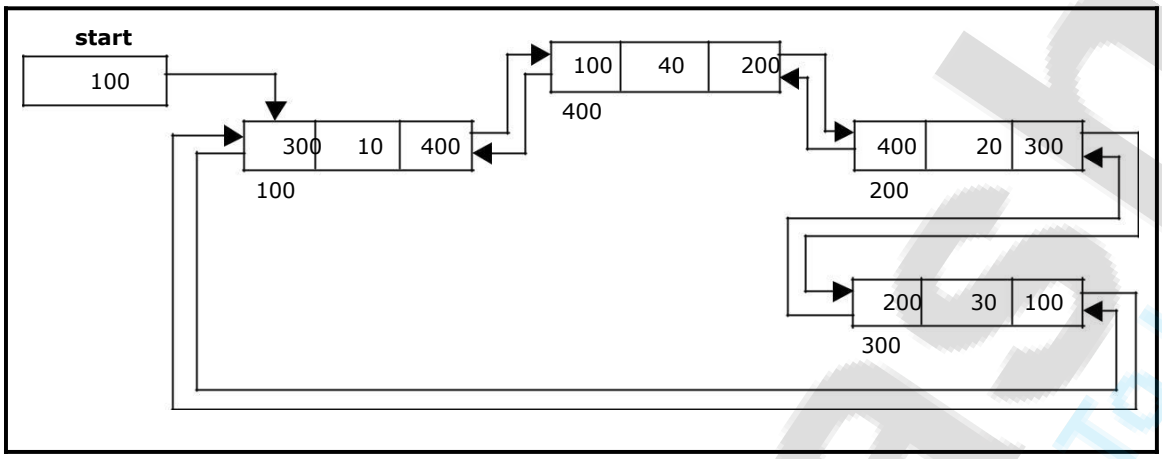


Figure 3.8.4. Inserting a node at an intermediate position

Deleting a node at the beginning:

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

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

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

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

The function cdll_delete_beg(), is used for deleting the first node in the list. Figure 3.8.5 shows deleting a node at the beginning of a circular double linked list.

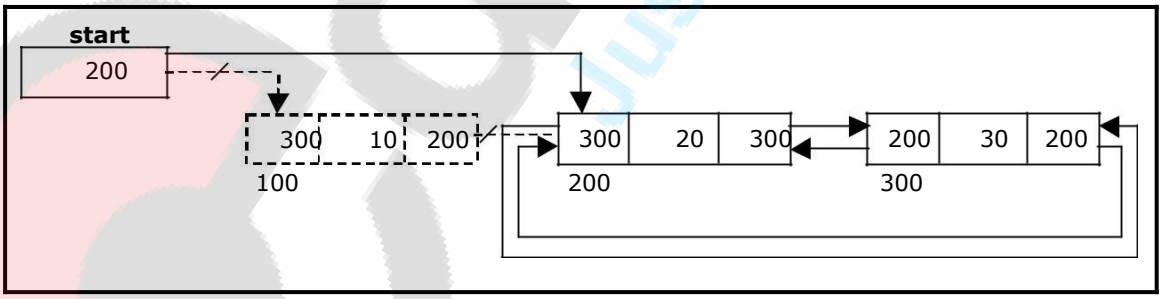


Figure 3.8.5. Deleting a node at beginning

Deleting a node at the end:

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

If list is empty then display 'Empty List' message

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

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

The function cdll_delete_last(), is used for deleting the last node in the list. Figure 3.8.6 shows deleting a node at the end of a circular double linked list.

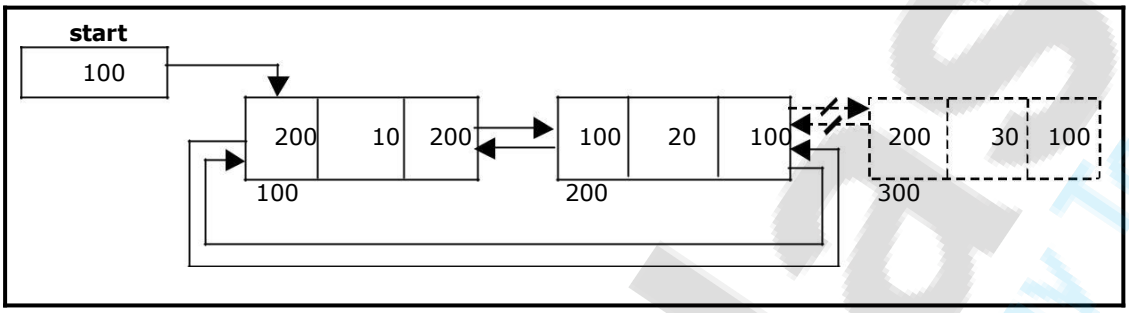


Figure 3.8.6. Deleting a node at the end

Deleting a node at Intermediate position:

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

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

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

The function cdll_delete_mid(), is used for deleting the intermediate node in the list.

Figure 3.8.7 shows deleting a node at a specified intermediate position other than beginning and end from a circular double linked list.

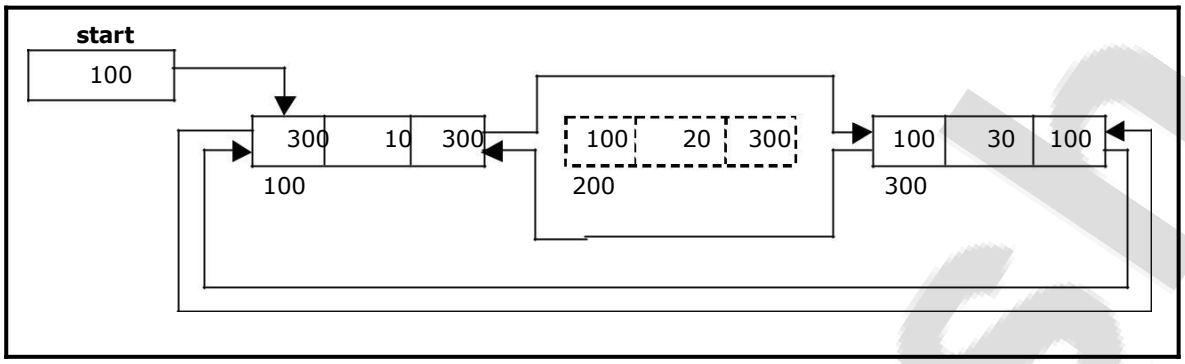


Figure 3.8.7. Deleting a node at an intermediate position

Traversing a circular double linked list from left to right:

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

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

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

```
temp = start;
Print temp -> data;
temp = temp -> right;
while(temp != start)
{
        print temp -> data;
        temp = temp -> right;
}
```

The function cdll_display_left _right(), is used for traversing from left to right.

Traversing a circular double linked list from right to left:

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

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

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

```
temp = start;
do
{
        temp = temp -> left;
        print temp -> data;
} while(temp != start);
```

The function cdll_display_right_left(), is used for traversing from right to left.

Source Code for Circular Double Linked List:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
```

```
struct cdlinklist
{
        struct cdlinklist
        *left; int data;
        struct cdlinklist *right;
};

typedef struct cdlinklist
node; node *start = NULL;
int nodectr;

node* getnode()
{
        node * newnode;
        newnode = (node *) malloc(sizeof(node));
        printf("\n Enter data: ");
        scanf("%d", &newnode -> data);
        newnode -> left = NULL;
        newnode -> right = NULL;
        return newnode;
}

int menu()
{
        int ch;
        clrscr();
        printf("\n 1. Create ");
        printf("\n\n-------------------------");
        printf("\n 2. Insert a node at Beginning");
        printf("\n 3. Insert a node at End");
        printf("\n 4. Insert a node at Middle");
        printf("\n\n-------------------------");
        printf("\n 5. Delete a node from Beginning");
        printf("\n 6. Delete a node from End");
        printf("\n 7. Delete a node from Middle");
        printf("\n\n-------------------------");
        printf("\n 8. Display the list from Left to Right");
        printf("\n 9. Display the list from Right to Left");
        printf("\n 10.Exit");
        printf("\n\n Enter your choice: ");
        scanf("%d", &ch);
        return ch;
}

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

```
                                        newnode -> right = start; start
                                        -> left->right = newnode;
                                        start -> left = newnode;
                                }
                        }
                }
                else
                        printf("\n List already exists..");
}

void cdll_display_left_right()
{
        node *temp;
        temp = start;
        if(start == NULL)
                printf("\n Empty List");
        else
        {
                printf("\n The contents of List:
                "); printf(" %d ", temp -> data);
                temp = temp -> right;
                while(temp != start)
                {
                        printf(" %d ", temp -> data);
                        temp = temp -> right;
                }
        }
}

void cdll_display_right_left()
{
        node *temp;
        temp = start;
        if(start == NULL)
                printf("\n Empty List");
        else
        {
                printf("\n The contents of List:
                "); do
                {
                        temp = temp -> left;
                        printf("\t%d", temp -> data);
                } while(temp != start);
        }
}

void cdll_insert_beg()
{
        node *newnode;
        newnode = getnode();
        nodectr++;
        if(start == NULL)
        {
                start = newnode;
                newnode -> left = start;
                newnode -> right = start;
        }
        else
        {
                newnode -> left = start -> left;
                newnode -> right = start;
                start -> left -> right = newnode;
                start -> left = newnode;
```

```
                start = newnode;
        }
}

void cdll_insert_end()
{
        node *newnode,*temp;
        newnode  =  getnode();
        nodectr++;
        if(start == NULL)
        {
                start = newnode;
                newnode -> left = start;
                newnode -> right = start;
        }
        else
        {
                newnode -> left = start -> left;
                newnode -> right = start;
                start -> left -> right = newnode;
                start -> left = newnode;
        }
        printf("\n Node Inserted at End");
}

void cdll_insert_mid()
{
        node *newnode, *temp, *prev;
        int pos, ctr = 1;
        newnode = getnode();
        printf("\n Enter the position: ");
        scanf("%d", &pos);
        if(pos - nodectr >= 2)
        {
                printf("\n Position is out of range..");
                return;
        }
        if(pos > 1 && pos <= nodectr)
        {
                temp = start;
                while(ctr < pos - 1)
                {
                        temp = temp -> right;
                        ctr++;
                }
                newnode -> left = temp; newnode
                -> right = temp -> right; temp ->
                right -> left = newnode; temp ->
                right = newnode; nodectr++;

                printf("\n Node Inserted at Middle.. ");
        }
        else
                printf("position %d of list is not a middle position", pos);
        }
}

void cdll_delete_beg()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n No nodes exist..");
```

```
                getch();
                return ;
        }
        else
        {
                nodectr--;
                if(nodectr == 0)
                {
                        free(start);
                        start = NULL;
                }
                else
                {
                        temp = start;
                        start = start -> right;
                        temp -> left -> right = start;
                        start -> left = temp -> left;
                        free(temp);
                }
                printf("\n Node deleted at Beginning..");
        }
}

void cdll_delete_last()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n No nodes
                exist.."); getch();
                return;
        }
        else
        {
                nodectr--;
                if(nodectr == 0)
                {
                        free(start);
                        start = NULL;
                }
                else
                {
                        temp = start;
                        while(temp -> right != start)
                                temp = temp -> right;
                        temp -> left -> right = temp -> right;
                        temp -> right -> left = temp -> left;
                        free(temp);
                }
                printf("\n Node deleted from end ");
        }
}

void cdll_delete_mid()
{
        int ctr = 1, pos;
        node *temp;
        if( start == NULL)
        {
                printf("\n No nodes
                exist.."); getch();
                return;
        }
```

```
        else
        {
                printf("\n Which node to delete: ");
                scanf("%d", &pos);
                if(pos > nodectr)
                {
                        printf("\nThis node does not
                        exist"); getch();
                        return;
                }
                if(pos > 1 && pos < nodectr)
                {
                        temp = start;
                        while(ctr < pos)
                        {
                                temp = temp -> right ;
                                ctr++;
                        }
                        temp -> right -> left = temp -> left;
                        temp -> left -> right = temp -> right;
                        free(temp);
                        printf("\n node deleted..");
                        nodectr--;
                }
                else
                {
                        printf("\n It is not a middle position..");
                        getch();
                }
        }
}

void main(void)
{
        int ch,n;
        clrscr();
        while(1)
        {
                ch = menu();
                switch( ch)
                {
                        case 1 :
                                printf("\n Enter Number of nodes to create: ");
                                scanf("%d", &n);
                                cdll_createlist(n);
                                printf("\n List
                                created.."); break;
                        case 2 : cdll_insert_beg();
                                break;

                        case 3 : cdll_insert_end();
                                break;

                        case 4 :
                                cdll_insert_mid();
                                break;
                        case 5 : cdll_delete_beg();
                                break;

                        case 6 :
                                cdll_delete_last();
                                break;
```

```
                        case 7 :
                                cdll_delete_mid();
                                break;
                        case 8 :
                                cdll_display_left_right();
                                break;
                        case 9 :
                                cdll_display_right_left();
                                break;
                        case 10:
                                exit(0);
                }
                getch();
        }
}
```

Comparison of Linked List Variations:

The major disadvantage of doubly linked lists (over singly linked lists) is that they require more space (every node has two pointer fields instead of one). Also, the code to manipulate doubly linked lists needs to maintain the *prev* fields as well as the *next* fields; the more fields that have to be maintained, the more chance there is for errors.

The major advantage of doubly linked lists is that they make some operations (like the removal of a given node, or a right-to-left traversal of the list) more efficient.

The major advantage of circular lists (over non-circular lists) is that they eliminate some extra-case code for some operations (like deleting last node). Also, some applications lead naturally to circular list representations. For example, a computer network might best be modeled using a circular list.

3.10. Polynomials:

A polynomial is of the form: $\sum_{i=0}^{n} c_i x^i$

Where, c_i is the coefficient of the i^{th} term and
n is the degree of the polynomial

Some examples are:

$5x^2 + 3x + 1$
$12x^3 - 4x$
$5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0$

It is not necessary to write terms of the polynomials in decreasing order of degree. In other words the two polynomials 1 + x and x + 1 are equivalent.

The computer implementation requires implementing polynomials as a list of pairs of coefficient and exponent. Each of these pairs will constitute a structure, so a polynomial will be represented as a list of structures. A linked list structure that represents polynomials $5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0$ illustrates in figure 3.10.1.

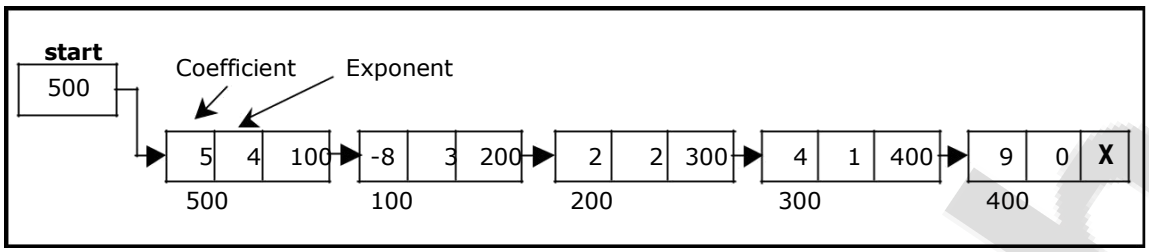


Figure 3.10.1. Single Linked List for the polynomial $F(x) = 5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0$

Source code for polynomial creation with help of linked list:

```
#include <conio.h>
#include <stdio.h>
#include <malloc.h>

struct link
{
        float coef;
        int expo;
        struct link *next;
};

typedef struct link node;
node * getnode()
{
                node *tmp;
                tmp =(node *) malloc( sizeof(node) );
                printf("\n Enter Coefficient : ");
                fflush(stdin); scanf("%f",&tmp-
                >coef);
                printf("\n Enter Exponent : ");
                fflush(stdin);
                scanf("%d",&tmp->expo);
                tmp->next = NULL;
                return tmp;
}
node * create_poly (node *p )
{
                char ch;
                node *temp,*newnode;
                while( 1 )
                {
                        printf ("\n Do U Want polynomial node (y/n):
                        "); ch = getche();
                        if(ch == 'n')
                                break;
                        newnode = getnode();
                        if( p == NULL )
                                p = newnode;
                        else
                        {
                                temp = p; while(temp-
                                >next != NULL )
                                        temp = temp->next;
                                temp->next = newnode;
                        }

                }
                return p;
}
```

```
void display (node *p)
{
		node *t = p;
		while (t != NULL)
		{
			printf("+ %.2f", t -> coef);
			printf("X^ %d", t -> expo);
			t =t -> next;
		}
}

void main()
{
		node *poly1 = NULL ,*poly2 = NULL,*poly3=NULL;
		clrscr();
		printf("\nEnter First Polynomial..(in ascending-order of exponent)");
		poly1 = create_poly (poly1);
		printf("\nEnter Second Polynomial..(in ascending-order of exponent)");
		poly2 = create_poly (poly2);
		clrscr();
		printf("\n Enter Polynomial 1:
		"); display (poly1);
		printf("\n Enter Polynomial 2:
		"); display (poly2);
		getch();
}
```

Addition of Polynomials:

To add two polynomials we need to scan them once. If we find terms with the same exponent in the two polynomials, then we add the coefficients; otherwise, we copy the term of larger exponent into the sum and go on. When we reach at the end of one of the polynomial, then remaining part of the other is copied into the sum.

To add two polynomials follow the following steps:

Read two polynomials.
Add them.
Display the resultant polynomial.

Source code for polynomial addition with help of linked list:

```
#include <conio.h>
#include <stdio.h>
#include <malloc.h>

struct link
{
	float coef;
	int expo;
	struct link *next;
};

typedef struct link node;

node * getnode()
{
		node *tmp;
```

```
                tmp =(node *) malloc( sizeof(node) );
                printf("\n Enter Coefficient : ");
                fflush(stdin); scanf("%f",&tmp-
                >coef);
                printf("\n Enter Exponent : ");
                fflush(stdin);
                scanf("%d",&tmp->expo);
                tmp->next = NULL;
                return tmp;
}

node * create_poly (node *p )
{
                char ch;
                node *temp,*newnode;
                while( 1 )
                {
                        printf ("\n Do U Want polynomial node (y/n):
                        "); ch = getche();
                        if(ch == 'n')
                                break;
                        newnode = getnode();
                        if( p == NULL )
                                p = newnode;
                        else
                        {
                                temp = p; while(temp-
                                >next != NULL )
                                        temp = temp->next;
                                temp->next = newnode;
                        }

                }
                return p;
}

void display (node *p)
{
                node *t = p;
                while (t != NULL)
                {
                        printf("+ %.2f", t -> coef);
                        printf("X^ %d", t -> expo);
                        t = t -> next;
                }
}

void add_poly(node *p1,node *p2)
{

        node *newnode;
        while(1)
        {
                if( p1 == NULL || p2 == NULL
                        ) break;
                if(p1->expo == p2->expo )
                {
                                printf("+ %.2f X ^%d",p1->coef+p2->coef,p1->expo);
                                p1 = p1->next; p2 = p2->next;
                }
                else
                {
                        if(p1->expo < p2->expo)
```

```
                        {
                                printf("+ %.2f X ^%d",p1->coef,p1->expo);
                                p1 = p1->next;
                        }
                        else
                        {
                                printf(" + %.2f X ^%d",p2->coef,p2-
                                >expo); p2 = p2->next;
                        }
                }
        }
        while(p1 != NULL )
        {
                printf("+ %.2f X ^%d",p1->coef,p1->expo);
                p1 = p1->next;
        }
        while(p2 != NULL )
        {

                printf("+ %.2f X ^%d",p2->coef,p2->expo);
                p2 = p2->next;
        }
}

void main()
{
                node *poly1 = NULL ,*poly2 = NULL,*poly3=NULL;
                clrscr();
                printf("\nEnter First Polynomial..(in ascending-order of exponent)");
                poly1 = create_poly (poly1);
                printf("\nEnter Second Polynomial..(in ascending-order of exponent)");
                poly2 = create_poly (poly2);
                clrscr();
                printf("\n Enter Polynomial 1:
                "); display (poly1);
                printf("\n Enter Polynomial 2:
                "); display (poly2);
                printf( "\n Resultant Polynomial :
                "); add_poly(poly1, poly2);
                display (poly3);
                getch();
}
```