Unit 10 Linked Lists

Structure:

10.1 Introduction

In computer science, a **linked list** is one of the fundamental data structures, and can be used to implement other data structures. It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes. The principal benefit of a linked list over a conventional array is that the order of the linked items may be different from the order that the data items are stored in memory or on disk, allowing the list of items to be traversed in a different order. A linked list is a self-referential data type because it contains a pointer or link to another datum of the same type. Linked lists permit insertion and removal of nodes at any point in the list in constant time, but do not allow random access. Several different types of linked lists exist: singly linked list, doubly linked list, and circularly linked list.

Operations on a list:

- **insert :** insert a new item into a list
- **delete :** delete an item from list
- **List:** list the items from the list
- **Retrieval :** search for an element in the list

Objectives:

At the end of this unit, you will be able to explain:

- Different types of Linked List and their Operations.
- Implementation of singly linked list.
- Implementation of circular singly linked list.
- Implementation of doubly linked list.

10.2 Types of Linked List

There are several types of linked list with different operations. Only the shaded blocks in the following figure are discussed here.

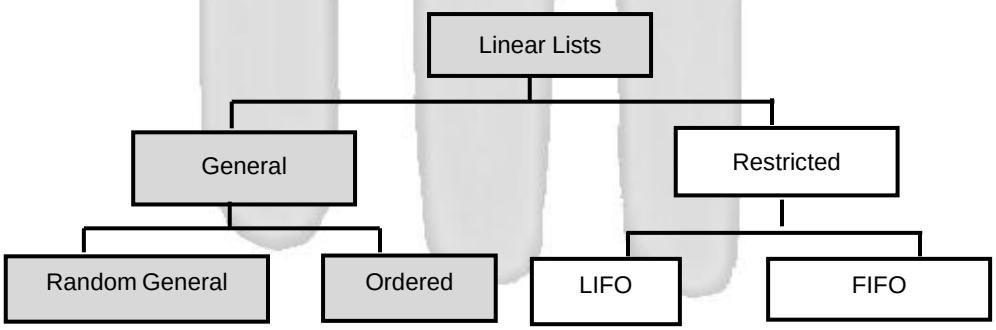


Fig. 10.1 Types of Linked List

There are four basic operations performed on linear lists:

- **Insertion of Nodes :**
  An insertion can be made at any position of the lists.

- **Deletion of Nodes :**

  Deletion from general list requires that the list be searched for the data to be deleted.

- **Retrieval of Values:**

  List retrieval requires that data be located in a list and presented to the calling module without changing the contents of the lists.

- **Traversal in the List:**

  List traversal is a special case of retrieval in which all the elements are retrieved in a sequence.

10.3 Singly Linked List

The simplest kind of linked list is a **singly linked list** (or **slist** for short), which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node.

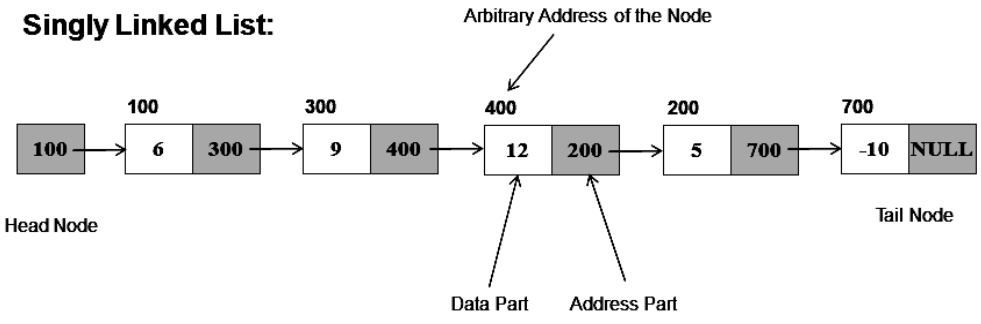
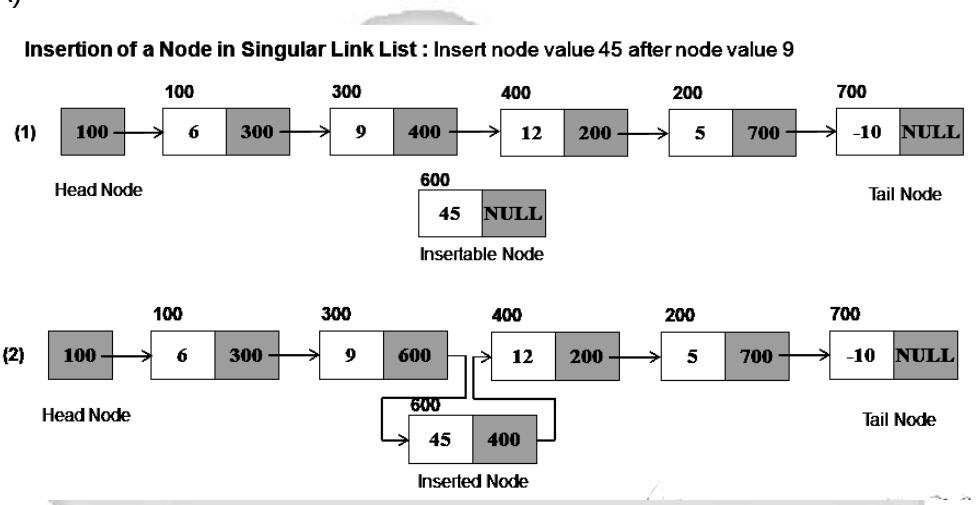


Fig. 10.2: Singly Linked List

The singly linked list is the most basic of all the linked data structures. A singly linked list is simply a sequence of dynamically allocated objects, each of which refers to its successor in the list. Despite this obvious simplicity there are many implementation variations.

The following figures show the most common singly linked lists.

(A)



(B)

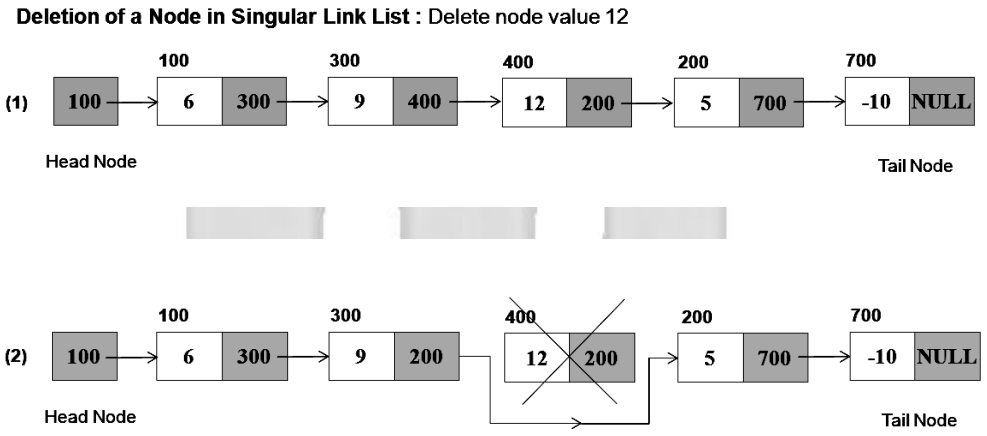


Fig. 10.3 Linked List Operations

Example 1: Q1) Write a C Program to demonstrate the singly linked list operations. [File Name u10q1.c]

```
/*************************************************************************/
/*    Program to demo the operation of singular  LINK-LIST       */
/*************************************************************************/
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define NULL 0
typedef struct record
{
char name[20];
int roll;
struct record *next;
}rec;
// -------  Node Declaration ----------------------------------------
int flag=0;
main()
{
rec *new,*app,*append(rec *pt),*ins(rec *pt),*del(rec *pt);
char nm[20],choice;
int rl;
void list(rec *pt, int);
new = (rec *)malloc(sizeof(rec));
app=new;
do
 {
   choice=menu();
   switch(choice)
```

```
  {
  case '1':
        app=append(app);      /* Append records  */
        break;
  case '2':
        heading();           /* Heading        */
        list(new,1);         /* List records   */
        getch();
        break;
  case '3':
         new=ins(new);       /* Insert records  */
        break;
  case '4':
        new=del(new);        /* Delete records  */
        break;
  case '5':
        exit(0);
  default :
        printf(" Your choice is invalid - Try again " ); getch();
  }
 }while(1);
}
/************************** end of main() **********************/
/************************** append() ******************************/
 rec* append(rec *record)
 {
 printf("\nName please [ X to exit ] : ");
   fflush(stdin); gets(record->name);
 if(strcmpi(record->name,"x")==0)
```

```
  {  record->next=NULL; return(record);   }
 else
  {
   printf("\nRoll no. :  "); fflush(stdin); scanf("%d", &record->roll);
   record->next = (rec *)malloc(sizeof(rec));
   append(record->next);
  }
 }
/************************* list() ******************************/
void list(rec *rd, int i)
 {
  if(rd->next==NULL) return;
   printf("%3d   %-20s  %5d\n", i++, strupr(rd->name), rd->roll);
   list(rd->next,i);
 }
/********************* ins() ************************************/
rec* ins(rec *record)
 {
 char word[20];
 rec *find(rec *,char *),*new1,*local;
 printf(" \nType NAME, before the new record to be inserted : ");
 fflush(stdin); gets(word);
 local=find(record,word);
 if(local==NULL)
  { printf("Record not found "); getch(); return(record); }
    new1=(rec *)malloc(sizeof(rec));
   printf("Enter students name to be inserted .. ");
     fflush(stdin); gets(new1->name);
   printf("\nEnter roll no....   ");
```

```
    fflush(stdin); scanf("%d", &new1->roll);
     if(flag==0)
    {
    new1->next=local->next;
    local->next=new1;
    return(record);
     }
    else
    {
     new1->next=record;
     record=new1;
     return(record);
   }
}
/********************** del() ******************************/
 rec* del(rec *record)
{
   rec *find(rec *, char *), *local,*temp;
   char word[20];
    printf("\n Enter students name to be deleted...   ");
    fflush(stdin); gets(word);
    local=find(record, word);
    if(local==NULL)
    { printf("Record not found "); getch(); return(record); }
          if(flag==0)
          {
           temp=local->next;
          local->next=local->next->next;
          free(temp); return(record);
```

```
        }
        else
        {
        temp=record;
        record=local->next;
        free(temp); return(record);
        }
}
/********************** find() ******************************/
 rec *find(rec *rd, char *nm)
{
  flag=0;
  if(strcmpi(rd->name, nm)==0)
   { flag=1; return(rd); }  /* flag=1 , found first record */
  if(strcmpi(rd->next->name, nm)==0)
    return(rd);
  else
   {
    if(rd->next->next == NULL)
        return(NULL);
    else
        find(rd->next, nm);
   }
 }
/***************************** menu() ******************************/
  menu()
  {
  char choice;
  clrscr();
```

```
  printf("\n\n                    Main Menu\n");
  printf("                   =========");
  printf("\n\n\n                    1  -  APPEND Records\n"
         "                   2  -  LIST   Records\n"
         "                   3  -  INSERT Records\n"
         "                   4  -  DELETE Records\n"
         "                   5  -  EXIT .\n\n");
   printf(" Your choice please.....");
   fflush(stdin); choice=getche();
   return(choice);
   }
/************************ heading() *****************************/
 heading()
  { clrscr(); printf("\n  SL..NAME.....ROLL\n\n");  }
```

10.4 Circular Linked List

In the linked lists discussed so far in earlier sections, the link field of the last node contained a NULL pointer. In this section, we discuss linear lists again with slight modification by storing address of the first node in the link field of the last node. The resulting list is called a circular singly linked list or a circular linked list. The pictorial representation of a circular list is shown in figure 10.4.

Note: Whatever operations are possible using singly linked lists, all those operations can be performed using circular lists also. A circular list can be used as a stack, queue or a deque. The basic operations required to implement these data structures are insert_front, insert_rear, delete_ front, delete_rear and display. Let us implement these functions one by one.

Circular Linked List:

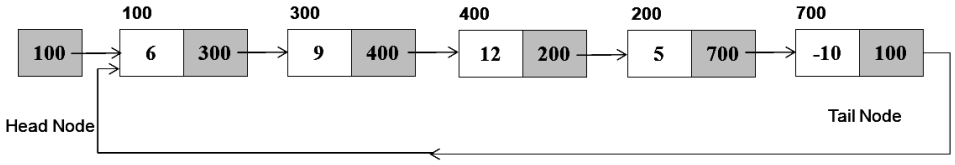


Fig. 10.4

10.4.1 Insert a node into the Circular Linked List

Consider the list shown in following fig. (a). The list contains 4 nodes and a pointer variable last contains address of the last node.

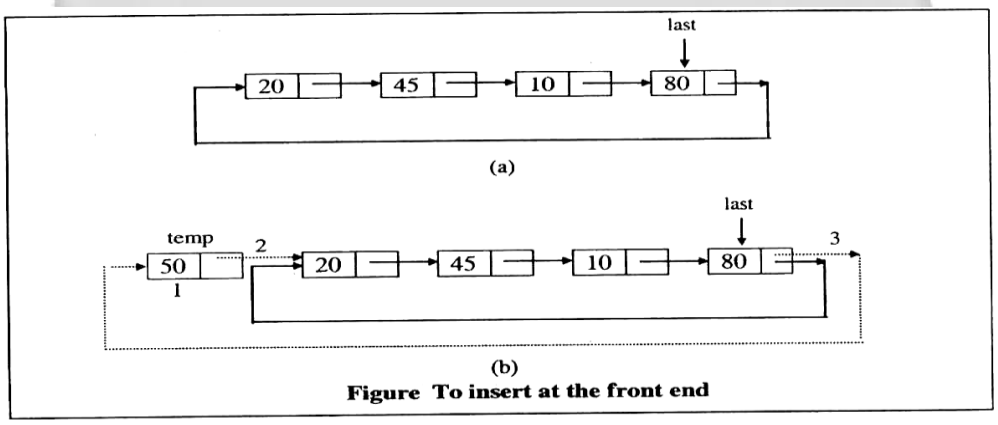


Fig. 10.5: To Insert at the Front End

Step 1: To insert an item 50 at the front of the list, obtain a free node temp from the availability list and store the item in info field as shown in dotted lines in above fig. (b). This can be accomplished using the following statements

```
temp = getnode( );
temp->info = item;
```

Step 2: Copy the address of the first node(i.e., last->link) into link field of newly obtained node temp and the statement to accomplish this task is

temp->link = last->link;

Step 3: Establish a link between the newly created node temp and the last node. This is achieved by copying the address of the node temp into link field of last node. The corresponding code for this is

last->link = temp;

Now, an item is successfully inserted at the front of the list. All these steps have been designed by assuming that the list already exists. If the list is empty, make temp itself as the last node and establish a link between the first node and the last node. Repeatedly insert the items using the above procedure to create a list.

10.4.2 Delete a Node from the Circular Linked List

The list in figure 10.6 contains 5 nodes and the last is a pointer variable that contains the address of the last node.

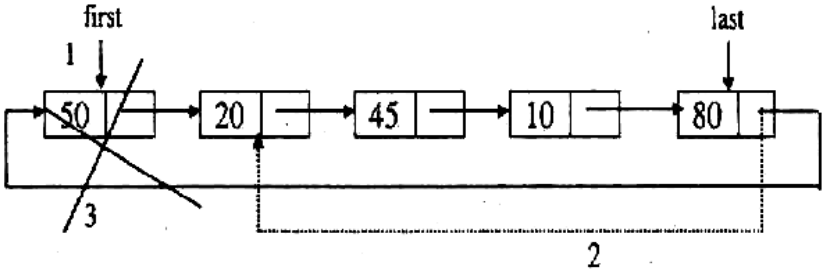


Fig 10.6: Figure to delete an item from the front end

The following steps are used to delete the front node in a linked list (see the sequence of numbers 1,2,3 shown in above figure 10.6).

Step 1: first = last->link;

/* obtain address of the first node */

Step 2: last->link = first->link;

/* link the last node and new first node */

Step 3: printf ("The item deleted is %d\n", first->info);

freenode (first); /*delete the old first node */

If there is only one node in the list, the deletion of a node makes the list as a an empty list.

Example 2: Q2) Write a program to implement the circular linked list. [File Name: u10q2.c]

```
#include <stdio.h>
#include <alloc.h>
#include <process.h>
struct node
{
int info;
struct node *link;
};

typedef struct node* NODE;
/********** prototype declaration */
NODE insert_front (int, NODE);
NODE insert_rear (int, NODE);
NODE delete_front(NODE);
NODE delete_rear(NODE);
NODE getnode(void);
void freenode(NODE);
void display(NODE);
void main( )
```

```
{
  NODE last;
  int choice, item;
  last = NULL;
  while(1)
  {
        clrscr();
        printf("\t\t Circular Linked List\n");
        printf("\t 1: Insert  Front\n");
        printf("\t 2: Insert  Rear\n");
        printf("\t 3: Delete  Front\n");
        printf("\t 4: Delete  Rear\n");
        printf("\t 5: Display\n");
        printf("\t 6: Exit\n");
        printf("\t Enter the choice : "); scanf("%d", &choice);
   switch(choice)
   {
       case 1:   // Insert at the front position
                 printf("Enter the item to be inserted : "); scanf("%d", &item);
                 last = insert_front(item, last);
                 continue;
       case  2:  // Insert at the rear position
                 printf("Enter the item to be inserted : "); scanf("%d", &item);
                 last = insert_rear(item, last);
                 continue;
       case 3:   //delete from front position.
                 last = delete_front(last);
                 break;
```

```
        case 4:  // delete from rear position
                 last = delete_rear(last);
                 break;
        case 5:  // display list
                 display(last); getch();
                 break;
        case 6:
                exit(0);
        default:
                printf("Invalid Input - Try Again");
       }  // end of switch
      getch();
   }     // end of loop
 }     // end of main()

/*****************************/
NODE insert_front(int item, NODE last)
{
        NODE temp;
        temp = getnode();          /* Create a new node to be inserted */
        temp->info = item;
        if (last == NULL)         /* Make temp as the first node */
                last = temp;
        else                       /* Insert at the front end */
                temp->link = last->link;
        last->link = temp;         /* link last node to first node */
        return last;              /* Return the last node */
}
/*********************************/
```

```
NODE insert_rear(int item, NODE last)
{
        NODE temp;
        temp = getnode( );      /* Create a new node to be inserted */
        temp->info = item;
        if ( last == NULL)    /* Make temp as the first node */
                last = temp;
        else                    /*Insert at the rear end */
                temp->link = last->link;
        last->link = temp;       /* link last node to first node */
        return temp;              /* Make the new node as the last node */
}
/*********************************/

NODE delete_front(NODE last)
{
        NODE temp, first;
        if ( last == NULL )
        {
          printf("List is empty  ");
          return NULL;
        }
        if ( last->link == last)   /* Only one node is present */
        {
          printf("The item deleted is %d\n", last->info);
          freenode (last);
          return NULL;
        }
                                /* List contains more than one node */
```

```
        first = last->link;      /* obtain node to be deleted */
        last->link = first->link; /* Store new first node in link of last */
        printf ("The item deleted is %d\n", first->info);
        freenode (first);        /* delete the old first node */
        return last;
}
/*****************************/
NODE delete_rear (NODE last)
{
                NODE prev;
                 if ( last == NULL )
                {
                   printf("List is empty\n");
                 return NULL; }
                 if ( last->link == last) /* Only one node is present */
                 {
                    printf("The item deleted is %d\n", last->info);
                    freenode(last);
                     return NULL;
                  }
                 /* Obtain address of previous node */
                 prev = last->link;
                 while( prev->link != last )
                 {
                          prev = prev->link;
                 }
                 prev->link = last->link;  /* prev node is made the last node */
                 printf("The item deleted is %d\n", last->info);
                 freenode(last);             /* delete the old last node */
```

```
                return prev;            /* return the new last node */
}
/*****************************/
void display(NODE last)
{
NODE temp;
if ( last == NULL)
{
printf("List is empty\n");
return;
}
printf("Contents of the list is\n  <Front> ");
for (temp = last->link; temp != last; temp = temp->link)
  printf("%d ", temp->info);
printf("%d <Rear> \n", temp->info);
}
/*****************************************************/
NODE getnode(void)
{
NODE *p;
p=(NODE *)malloc(sizeof(NODE));
return(*p);
}
/*************************************/
void freenode(NODE p)
{
  free(p);
}
/***************************/
```

Disadvantages of singly linked list

1. Using singly linked lists and circular lists it is not possible to traverse the list backwards.
2. To find the predecessor, it is required to traverse the list from the first node in case of singly linked list. In case of circular list, the predecessor can be obtained by traversing the whole list from the node specified. For example, if the position of the current node is 15, to find the position of node 14, the whole list has to be traversed.

All these disadvantages can be overcome by using doubly linked lists.

10.5 Doubly Linked Lists

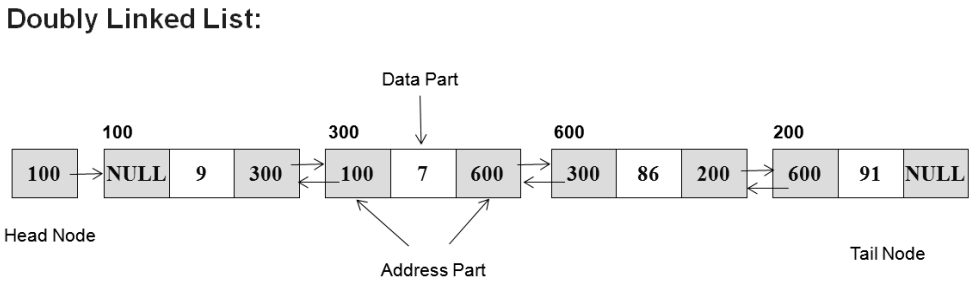


Fig. 10.7: Doubly Linked List

In singly linked lists (including circular lists), each node contains the address of the next node. If there is one more field which contains the address of the previous node, it is possible to obtain the address of the predecessor of a node specified. Using such lists, it is possible to traverse the list in forward and backward directions. A list where both forward and backward traversal is possible should have two link fields. The link field, which contains address of the left node, is called left link denoted by **prev** and the link field which contains the address of the right node is called right link and is denoted by

next. Such a list where each node has two links is called a doubly linked list or a two-way list.

The list can be traversed from the first node whose address is stored in a pointer variable, to the last node in the forward direction. It can also be traversed from the last node whose address is stored in a pointer variable last, to the first node in backward direction.

All those problems that cannot be solved using singly linked lists can be solved using doubly linked lists. It is left to the reader to implement all the problems solved so far, using doubly linked lists and doubly linked circular list. Given any problem, let us implement them using doubly linked lists and with a header node. Using a header node, problems can be solved very easily and effectively.

Pictorial representation of Insertion of node(s) in doubly linked list

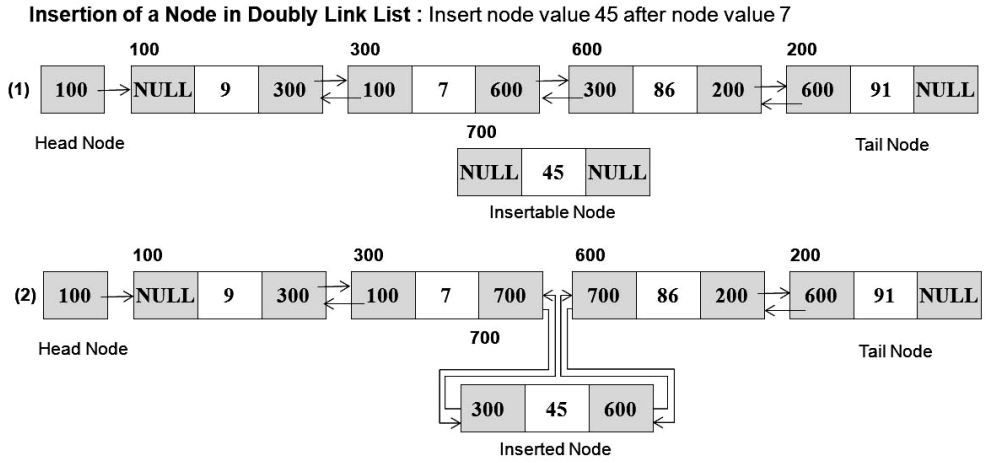


Fig. 10.8: Insert Operation

Pictorial representation of Deletion of node(s) in doubly linked list

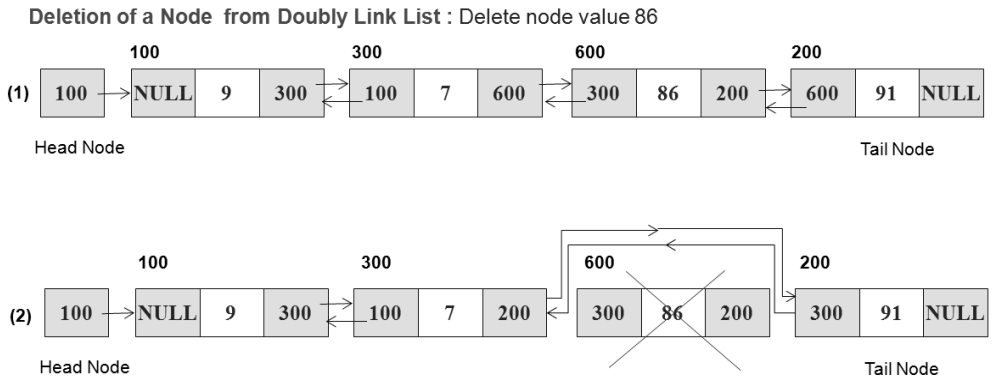


Fig. 10.9: Delete Operation

Example: Q3) Write a program to perform all the operations over doubly linked List. [File Name: u10q3.c]

```
#include<stdio.h>
/* Program for Doubly Link List for the operatoion
   of Append, Insert , List, Delete of Nodes.     */
#include<stdlib.h>
#include<conio.h>
typedef struct node
{
int val;              /* node defination */
struct node *prev;
struct node *next;
}rec;
int flag=0;
/*************************************/
main()
{
rec *head,* delete(rec *);
```

```
void insert(rec *), dbox(int, int, int, int), append(rec *), list(rec *);
char choice;
 head=(rec *)malloc(sizeof(rec));
 head->next=NULL;
 head->prev=NULL;
 do{
   clrscr();
   textcolor(WHITE);
   dbox(25, 8, 56, 22);
   dbox(20, 4, 62, 6);
   textcolor(YELLOW+BLINK);
   gotoxy(24,05); cprintf("OPERATION THROUGH DOUBLE LINK LIST");
   textcolor(GREEN);
   gotoxy(30,10); cprintf("1......Append Records");
   gotoxy(30, 12); cprintf("2......List   Records");
   gotoxy(30, 14); cprintf("3......Insert Records");
   gotoxy(30, 16); cprintf("4......Delete Records");
   gotoxy(30,18); cprintf("5......Exit   Program");
   gotoxy(30, 20); printf("choice [1/2/3/4/5 ]  ");
   choice=getche();
   textcolor(WHITE);
/****************************** Accept the choice****/
   switch(choice)
   {
   case '1':          /* Append the records */
        clrscr();  dbox(28, 4, 45, 6); textcolor(YELLOW);
        gotoxy(30, 5); cprintf("Append Records\n\n");
        textcolor(WHITE); append(head);  break;
```

```
   case '2':           /* List the records   */
        clrscr();  dbox(28, 3, 50, 5);
        gotoxy(30, 4); printf("  List of Records\n\n\n");
        list(head);    break;
    case '3':           /* Insert the records */
       clrscr();
       dbox(28, 4, 50, 6);
       gotoxy(30, 5); printf("Insertion of Node");
       insert(head); break;
    case '4':           /* Delete the records */
      clrscr(); dbox(28, 4, 50, 6);
      gotoxy(30, 5); printf("Deletion of Node");
      head=delete(head); break;
    case '5':              /* EXIT FROM PROGRAM */
     exit(0);
   }
  }while(1);
}
/*************  end of main() *******************/
void append(rec *p)
{
 while(p->next !=NULL)  p=p->next;
 if(flag==0)     /* for first record */
  { gotoxy(30, 10);
    printf("Enter 1st value "); scanf("%d", &p->val);
    flag=1;  return;
  }
 else                    /* other than first recors */
  {
```

```
   p->next=(rec *)malloc(sizeof(rec));
   p->next->prev=p; p->next->next=NULL;
   gotoxy(30,10);
   printf("Enter value "); scanf("%d", &p->next->val);
   return;
  } /*************** end of append() ***************/
}
void list(rec *p)
{
    textcolor(YELLOW); gotoxy(1,8);
    cprintf("================          GIVEN                ORDER          LIST
====================");
    textcolor(WHITE);
    gotoxy(1,10);
  do{
     printf("%5d", p->val);
     if(p->next==NULL) break;
     p=p->next;
    }while(1);
    textcolor(YELLOW); gotoxy(1,15);
    cprintf("================REVERSE              ORDER              LIST
====================");
    textcolor(WHITE);
    gotoxy(1, 17);
  do{
     printf("%5d", p->val);
     if(p->prev==NULL) break;
     p=p->prev;
    }while(1);
```

```
  getch(); return;
}
/************************* end of list() ********/
void insert(rec *p)
{
 rec *temp=p;
 int value, found=0;
 gotoxy(20,10);
 printf("Enter the node Value after insertion : ");  scanf("%d", &value);
 while(p->next!=NULL)
 {
  if(p->val==value)
   {
       found=1;
         temp=(rec *)malloc(sizeof(rec));
       temp->next=p->next; temp->prev=p;
       p->next=temp;       temp->next->prev=temp;
       printf("Enter the new number");scanf("%d", &temp->val);
   }  /* end of if */
  p=p->next;
 }  /* End of while */
 if(p->val==value)           /* for last Node */
   {
       found=1;
       p->next=(rec *)malloc(sizeof(rec));
       p->next->next=NULL; p->next->prev=p;
       printf("Enter the new number"); scanf("%d",&p->next->val);
   }  /* end of if */
```

```
 if(found==0)
  { printf("Node not Found"); getch(); }
}   /***************** end of insert() ************/
rec *delete(rec *p)
{
  rec *temp, *hd=p;
 int value, found=0;
 gotoxy(25, 10);
 printf("Enter the number to be deleted "); scanf("%d", &value);
 while(p->next!=NULL)
 {
  if(p->val==value)
   {
       if(p==hd)      /* deletion of first node */
        {
       found=1;
       temp=p;  p->next->prev=NULL;
       hd=p->next; free(temp);
         printf("Node is deleted Successfully"); getch();
       return(hd);
        }
    found=1;  temp=p;
    p->next->prev=p->prev;
    p->prev->next=p->next;  free(temp);
    printf("Node is deleted Successfully"); getch();
    return(hd);
   }
  p=p->next;
 } /* end of while */
```

```
  if(p->val==value)  /* Deletion of Last Node */
  { temp=p; p->prev->next=NULL; free(temp); return(hd); }
   if(found==0) printf("Record not found"); getch(); return hd;
 }
 /*************** end of delete()  ********************/
void dbox(int a, int b, int c, int d)
{
        int i;
        if(a<1||a>80||b<1||b>24||c<1||c>80 || d<1||d>24)
                { printf("Position off the Screen"); getch(); exit(0);}
/* Four corners */
        gotoxy(a, b); printf("%c", 201);
        gotoxy(c, b); printf("%c", 187);
        gotoxy(a, d); printf("%c", 200);
        gotoxy(c, d); printf("%c", 188);
/* End of four corners*/
        for(i=a+1; i<c; i++)
        {
                gotoxy(i, b); printf("%c", 205);
                gotoxy(i, d); printf("%c", 205);
        }
        for(i=b+1; i<d; i++)
        {
                gotoxy(a, i); printf("%c", 186);
                gotoxy(c, i); printf("%c", 186);
        }
} /* End of Function */
```

10.6 Application of Linked List

- The main Applications of Linked Lists are
  - For representing Polynomials
- It means in addition/subtraction /multiplication of two polynomials.
  - Eg:p1=2x^2+3x+7 and p2=3x^3+5x+2
  - p1+p2=3x^3+2x^2+8x+9
- In Dynamic Memory Management
  - In allocation and releasing memory at runtime.
- In Symbol Tables
  - in Balancing parentheses
- Representing Sparse Matrix

10.8 Summary

A list that displays the relationship of adjacency between elements is said to be linear list. It also can be defined to consist of an ordered set of elements. Linear linked list is the most commonly used data structure of linked lists. A simple way to represent a linear list is to expand each node to contain a link or pointer to the next node. Different types of linked list and their operations are discussed in this unit.

Self-Assessment Questions

1. In the linked list we can access any node randomly. (True / False)
2. Node of a singular linked list have only one reference field. (True / False)
3. Circular Linked List has two way references. (True / False)
4. Optimum Utilization of Memory Space is the application of Linked List. (True / False)
5. By Doubly Linked List, we can move in both the direction. (True / False)

10.8 Terminal Questions

1. What is List? Discuss the functions defined to operate on List.
2. Explain the typical basic operations on linked list.
3. Explain the singly linked list with a neat diagram.
4. Illustrate the ‘C’ program for singly link list operators using points.
5. Write a note on circular singly linked list.
6. Explain the procedures for insertion and deletion of a node in a circular singly linked list.
7. Write C program to implement dqueue.
8. Explain the features of a doubly linked list.
9. Explain doubly linked list operations.
10. Write a C program to split a singular linked list into two lists where one will contain odd numbers and the other will contain even numbers.
11. Write a program to add two numbers which is sufficiently large beyond the language data range.
12. Write a program to add two polynomials.
13. Write a program to print the values in reverse order using a singular linked list.

10.9 Answer to Self Assessment and Terminal Questions

Self Assessment Questions

1. False
2. True
3. False
4. True
5. True

Terminal Questions

1. Section 10.2
2. Section 10.3
3. Section 10.3
4. Section 10.3
5. Section 10.4
6. Section 10.4.2
7. Section 10.4
8. Section 10.5
9. Section 10.5