UNIT 3 LISTS

Structure **Page Nos.**

3.0 INTRODUCTION

In the previous unit, we have discussed arrays. Arrays are data structures of fixed size. Insertion and deletion involves reshuffling of array elements. Thus, array manipulation is time-consuming and inefficient. In this unit, we will see abstract data type-lists, array implementation of lists and linked list implementation, Doubly and Circular linked lists and their applications. In linked lists, items can be added or removed easily to the end or beginning or even in the middle.

3.1 OBJECTIVES

After going through this unit, you will be able to:

- define and declare Lists;
- understand the terminology of Singly linked lists;
- understand the terminology of Doubly linked lists;
- understand the terminology of Circularly linked lists, and
- use the most appropriate list structure in real life situations.

3.2 ABSTRACT DATA TYPE-LIST

Abstract Data Type (**ADT**) is a useful tool for specifying the logical properties of data type. An ADT is a collection of values and a set of operations on those values. Mathematically speaking, "a **TYPE** is a set, and elements of set are **Values** of that type".

ADT List

A list of elements of type **T** is a finite **sequence** of elements of type **T** together with the operations of create, update, delete, testing for empty, testing for full, finding the size, traversing the elements.

In defining Abstract Data Type, we are not concerned with space or time efficiency as well as about implementation details. The elements of a list may be integers, characters, real numbers and combination of multiple data types.

Consider a real world problem, where we have a company and we want to store the details of employees. To store this, we need a data type which can store the type details containing names of employee, date of joining, etc. The list of employees may

increase depending on the recruitment and may decrease on retirements or termination of employees. To make it very simple and for understanding purposes, we are taking the name of employee field and ignoring the date of joining etc. The operations we have to perform on this list of employees are creation, insertion, deletion, visiting, etc. We define employee_list as

```
typedef struct
{
        char    name[20];
        ……….
        ……..
}  emp_list;
```

Operations on emp_list can be defined as

```
Create_emplist (emp_list    * emp_list )
{
/* Here, we will be writing create function by taking help of 'C' programming
language. */
 }
```

The list has been created and **name** is a valid entry in **emplist**, and position **p** specifies the position in the list where name has to inserted

```
insert_emplist (emp_list    * emp_list ,    char      *name, int position    )
{
/* Here,  we will be writing insert function by taking help of 'C' programming
language. */
}
delete_emplist (emp_list    * emp_list,    char      *name)
{
/* Here, we will be writing delete function by taking help of 'C' programming
language. */
}
visit_emplist (emp_list    * emp_list )
{
/* Here, we will be writing visit function by taking help of 'C' programming
language. */
}
```

The list can be implemented in two ways: the contiguous (Array) implementation and the linked (pointer) implementation. In contiguous implementation, the entries in the list are stored next to each other within an array. The linked list implementation uses pointers and dynamic memory allocation. We will be discussing array and linked list implementation in our next section.

3.3 ARRAY IMPLEMENTATION OF LISTS

In the array implementation of lists, we will use array to hold the entries and a separate counter to keep track of the number of positions are occupied. A structure will be declared which consists of Array and counter.

```
typedef struct
{
        int count;
        int entry[100];
}list;
```

For simplicity, we have taken list entry as integer. Of course, we can also take list entry as structure of employee record or student record, etc.

Count	1	2	3	4	5	6	7	8
	11	22	33	44	55	66	77	

Insertion

In the array implementation of lists, elements are stored in continuous locations. To add an element to the list at the end, we can add it without any problem. But, suppose if we want to insert the element at the beginning or middle of the list, then we have to rewrite all the elements after the position where the element has to be inserted. We have to shift $(n)^{th}$ element to $(n+1)^{th}$ position, where 'n' is number of elements in the list. The $(n-1)^{th}$ element to $(n)^{th}$ position and this will continue until the (r) th element to $(r + 1)^{th}$ position, where 'r' is the position of insertion. For doing this, the **count** will be incremented.

From the above example, if we want to add element '35' after element '33'. We have to shift 77 to 8^{th} position, 66 to 7^{th} position, so on, 44 to 5^{th} position.

Before Insertion

Count	1	2	3	4	5	6	7	
	11	22	33	44	55	66	77	

Step 1

Count	1	2	3	4	5	6	7	8
	11	22	33	44	55	66	77	77

Step 2

Count	1	2	3	4	5	6	7	8
	11	22	33	44	55	66	66	77

Step 3

Count	1	2	3	4	5	6	7	8
	11	22	33	44	55	55	66	77

Step 4

Count	1	2	3	4	5	6	7	8
	11	22	33	44	44	55	66	77

Step 5

Count	1	2	3	4	5	6	7	8
	11	22	33	35	44	55	66	77

Program 3.1 will demonstrate the insertion of an element at desired position

```
/* Inserting an element into contiguous list (Linear Array) at specified position */
/* contiguous_list.C */
# include<stdio.h>
/* definition of linear list */
typedef struct
{
        int data[10];
        int count;
}list;
/*prototypes of functions */
void insert(list *, int, int);
void create(list *);
```

```
void traverse(list *);

/* Definition of the insert funtion */

void insert(list *start, int position, int element)
{
        int temp = start->count;
        while( temp >= position)
        {
                start->data[temp+1] = start->data[temp];
                temp --;
        }

        start->data[position] = element;
        start->count++ ;
}

/* definition of create function to  READ data values into the list */

void create(list *start)
{
        int i=0, test=1;
        while(test)
        {
                fflush(stdin);
                printf("\n input value value for: %d:(zero to come out) ", i);
                scanf("%d", &start->data[i]);

                if(start->data[i] == 0)
                        test=0;
                else
                        i++;
        }
        start->count=i;
}

/* OUTPUT FUNCTION TO PRINT ON THE CONSOLE */

void traverse(list *start)
{
        int i;
        for(i = 0; i< start->count; i++)
        {
                printf("\n Value at the position: %d: %d ", i, start->data[i]);
        }
}

/* main function */

void main( )
{
        int position, element;
        list l;
        create(&l);
        printf("\n Entered list as follows:\n");
        fflush(stdin);
        traverse(&l);
```

```
        fflush(stdin);
        printf("\n input the position where you want to add a new data item:");
        scanf("%d", &position);
        fflush(stdin);
        printf("\n input the value for the position:");
        scanf("%d", &element);
        insert(&l, position, element);
        traverse(&l);
}
```

Program 3.1: Insertion of an element into a linear array.

Deletion

To delete an element in the list at the end, we can delete it without any problem. But, suppose if we want to delete the element at the beginning or middle of the list, then, we have to rewrite all the elements after the position where the element that has to be deleted exists. We have to shift $(r+1)^{th}$ element to r^{th} position , where 'r' is position of deleted element in the list, the $(r + 2)^{th}$ element to $(r + 1)^{th}$ position, and this will continue until the $(n)^{th}$ element to $(n–1)^{th}$ position, where n is the number of elements in the list. And then the count is decremented.

From the above example, if we want to delete an element '44' from list. We have to shift 55 to 4^{th} position, 66 to 5^{th} position, 77 to 6^{th} position.

Before deletion

Count	1	2	3	4	5	6	7	
	11	22	33	44	55	66	77	

Step 1

Count	1	2	3	4	5	6	7	
	11	22	33	55	55	66	77	

Step 2

Count	1	2	3	4	5	6	7	
	11	22	33	55	66	66	77	

Step 3

Count	1	2	3	4	5	6		
	11	22	33	55	66	77		

Program 3.2 will demonstrate deletion of an element from the linear array

```
/* declaration  of delete_list function */
void delete_list(list *, int);

/* definition of delete_list  function*/
/* the position of the element is given by the user and the element is deleted from the
list*/
void delete_list(list *start, int position)
{
        int temp = position;
        printf("\n information which we have to delete: %d",l->data[position]);

        while( temp <= start->count-1)
```

```
        {
                start->data[temp] = start->data[temp+1];
                temp ++;
        }
        start->count = start->count - 1 ;
}

/* main  function */
void main()
{
        ……..
        …….

        printf("\n input the position of element you want to delete:");
        scanf("%d", &position);
        fflush(stdin);
        delete_list(&l, position);
        traverse(&l);
}
```

Program 3.2: Deletion of an element from the linear array

3.4 LINKED LISTS - IMPLEMENTATION

The Linked list is a chain of structures in which each structure consists of data as well as pointer, which stores the address (link) of the next logical structure in the list.

A linked list is a data structure used to maintain a dynamic series of data. Think of a linked list as a line of bogies of train where each bogie is connected on to the next bogie. If you know where the first bogie is, you can follow its link to the next one. By following links, you can find any bogie of the train. When you get to a bogie that isn't holding (linked) on to another bogie, you know you are at the end.

Linked lists work in the same way, except programmers usually refer to nodes instead of bogies. A single node is defined in the same way as any other user defined type or object, except that it also contains a pointer to a variable of the same type as itself.

We will be seeing how the linked list is stored in the memory of the computer. In the following *Figure 3.1*, we can see that **start** is a pointer which is pointing to the node which contains data as *madan* and the node *madan* is pointing to the node *mohan* and the last node *babu* is not pointing to any node. 1000,1050,1200 are memory addresses.

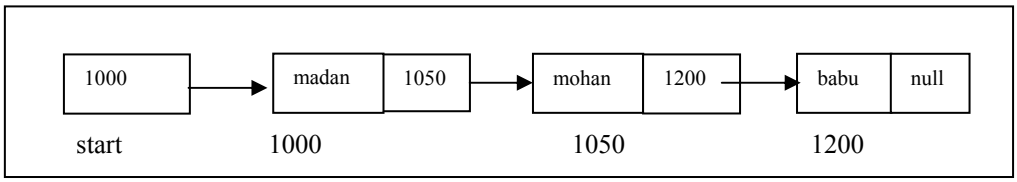


Figure 3.1: A Singly linked list

Consider the following definition:

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

Once you have a definition for a list node, you can create a list simply by declaring a pointer to the first element, called the "head". A pointer is generally used instead of a regular variable. List can be defined as

list *head;

It is as simple as that! You now have a linked list data structure. It isn't altogether useful at the moment. You can see if the list is empty. We will be seeing how to declare and define list-using pointers in the following program 3.3.

```
#include <stdio.h>

typedef struct node
{
   int data;
   struct node *next;
} list;

int main()
{
   list *head = NULL; /* initialize list head to NULL */
   if (head == NULL)
   {
      printf("The list is empty!\n");
   }
}
```

Program 3.3: Creation of a linked list

In the next example (Program 3.4), we shall look to the process of addition of new nodes to the list with the function create_list().

```
#include<stdio.h>
#include<stdlib.h>
#define NULL 0

struct linked_list
{
        int data;
        struct linked_list *next;
};
typedef struct linked_list list;

void main()
{
        list  *head;
        void create(list *);
        int count(list *);
        void traverse(list *);
        head=(list *)malloc(sizeof(list));
        create(head);
        printf("  \n  traversing the list \n");
        traverse(head);
        printf("\n number of elements in the list   %d \n", count(head));
}

void create(list *start)
{
        printf("inputthe element  -1111 for coming oout of the loop\n");
        scanf("%d", &start->data);
```

```
        if(start->data == -1111)
                start->next=NULL;
        else
        {
                start->next=(list*)malloc(sizeof(list));
                create(start->next);
        }
}

void traverse(list *start)
{
        if(start->next!=NULL)
        {
                printf("%d --> ", start->data);
                traverse(start->next);
        }
}

int count(list *start)

{
        if(start->next == NULL)
                return 0;
        else
                return (1+count(start->next));
}
```

Program 3.4: Insertion of elements into a Linked list

ALGORITHM (Insertion of element into a linked list)

Step 1	Begin
Step 2	if the list is empty or a new element comes before the start (head) element, then insert the new element as start element.
Step 3	else, if the new element comes after the last element, then insert the new element as the end element.
Step 4	else, insert the new element in the list by using the find function, find function returns the address of the found element to the insert_list function.
Step 5	End.

Figure 3.2 depicts the scenario of a linked list of two elements and a new element which has to be inserted between them. *Figure 3.3* depicts the scenario of a linked list after insertion of a new element into the linked list of *Figure 3.2*.

Before insertion

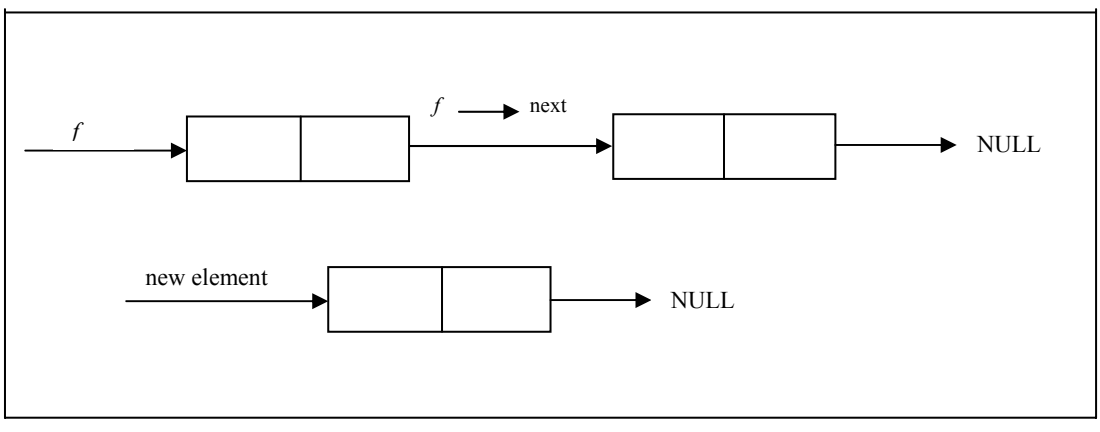


Figure 3.2: A linked list of two elements and an element that is to be inserted

After insertion

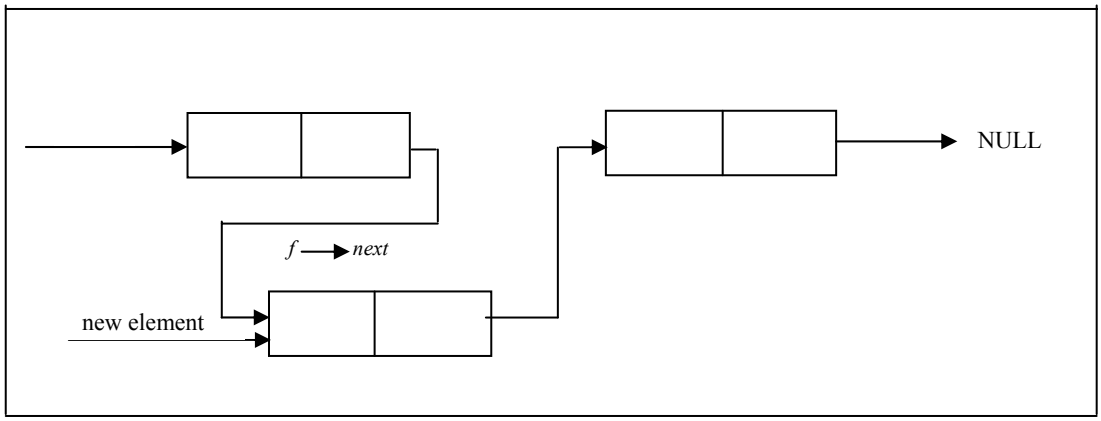


Figure 3.3: Insertion of a new element into linked list

Program 3.5 depicts the code for the insertion of an element into a linked list by searching for the position of insertion with the help of a **find** function.

INSERT FUNCTION

```
/*prototypes of insert and find functions */
list * insert_list(list *);
list * find(list *, int);
/*definition of insert function */
list * insert_list(list *start)
{
        list *n, *f;
        int key, element;
        printf("enter value of new element");
        scanf("%d", &element);
        printf("eneter value of key element");
        scanf("%d",&key);
        if(start->data ==key)
        {
                n=(list *)mallo(sizeof(list));
                n->data=element;
                n->next = start;
                start=n;
        }
        else
        {
                f = find(start, key);
```

```
                if(f == NULL)
                        printf("\n key is not found \n");
                else
                {
                        n=(list*)malloc(sizeof(list));
                        n->data=element;
                        n->next=f->next;
                        f->next=n;
                }
        }
        return(start);
}
/*definition of find function */
list * find(list *start, int key)
{
        if(start->next->data == key)
                return(start);
  if(start->next->next == NULL)
                return(NULL);
        else
                find(start->next, key);
}

void main()
{
        list  *head;
        void create(list *);
        int count(list *);
        void traverse(list *);
        head=(list *)malloc(sizeof(list));
        create(head);
        printf("  \n  traversing the created list \n");
        traverse(head);
        printf("\n number of elements in the list   %d \n", count(head));
        head=insert_list(head);
        printf("  \n  traversing the list after insert \n");
        traverse(head);
}
```

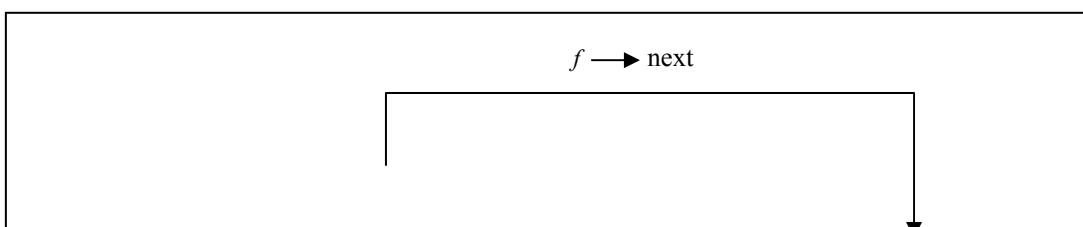
Program 3.5: Insertion of an element into a linked list at a specific position

ALGORITHM (Deletion of an element from the linked list)

Step 1 Begin
Step 2 if the list is empty, then element cannot be deleted
Step 3 else, if element to be deleted is first node, then make the start (head) to point to the second element.
Step 4 else, delete the element from the list by calling find function and returning the found address of the element.
Step 5 End

Figure 3.4 depicts the process of deletion of an element from a linked list.

After Deletion



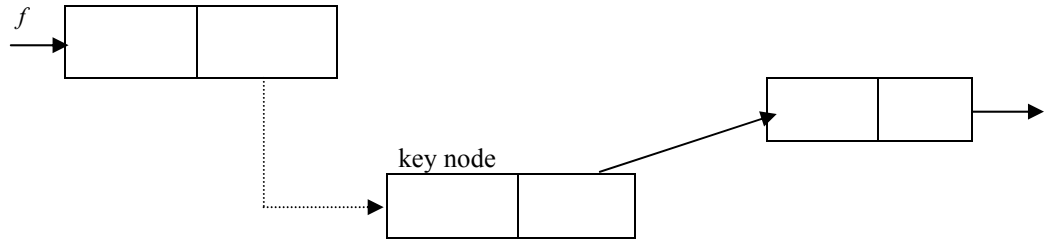


Figure 3.4: Deletion of an element from the linked list (Dotted line depicts the link prior to deletion)

Program 3.6 depicts the deletion of an element from the linked list. It includes a function which specifically searches for the element to be deleted.

DELETE LIST FUNCTION

```
/*  prototype of delete_function */
list *delete_list(list *);
list *find(list *, int);

/*definition of delete_list */
list *delete_list(list *start)
{
        int key; list * f, * temp;
        printf("\n enter the value of element to be deleted \n");
        scanf("%d", &key);
        if(start->data == key)
        {
                temp=start->next;
                free(start);
                start=temp;
        }
        else
        {
                f = find(start,key);
                if(f==NULL)
                        printf("\n key not fund");
                else
                {
                        temp = f->next->next;
                        free(f->next);
                        f->next=temp;
                }
        }
        return(start);
}
void main()
{

        list  *head;
        void create(list *);
        int count(list *);
        void traverse(list *);
        head=(list *)malloc(sizeof(list));
        create(head);
```

```
        printf("  \n  traversing the created list \n");
        traverse(head);
        printf("\n number of elements in the list   %d \n", count(head));
        head=insert(head);
        printf("  \n  traversing the list after insert \n");
        traverse(head);
        head=delete_list(head);
        printf("  \n  traversing the list after delete_list \n");
        traverse(head);
}
```

Program 3.6: Deletion of an element from the linked list by searching for element that is to be deleted

3.5 DOUBLY LINKED LISTS-IMPLEMENTATION

In a singly linked list, each element contains a pointer to the next element. We have seen this before. In single linked list, traversing is possible only in one direction. Sometimes, we have to traverse the list in both directions to improve performance of algorithms. To enable this, we require links in both the directions, that is, the element should have pointers to the right element as well as to its left element. This type of list is called **doubly linked list**.

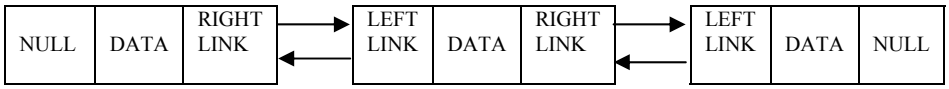


Figure 3.5: A Doubly Linked List

Doubly linked list **(***Figure 3.5***)** is defined as a collection of elements, each element consisting of three fields:

- pointer to left element,
- data field, and
- pointer to right element.

Left link of the leftmost element is set to NULL which means that there is no left element to that. And, right link of the rightmost element is set to NULL which means that there is no right element to that.

ALGORITHM (Creation)

Step 1 begin

Step 2 define a structure ELEMENT with fields
Data
Left pointer
Right pointer

Step 3 declare a pointer by name head and by using (malloc()) memory allocation function allocate space for one element and store the address in head pointer
Head = (ELEMENT *) malloc(sizeof(ELEMENT))

Step 4 read the value for head->data
head->left = NULL
head->right = (ELEMENT *) malloc(size of (ELEMENT))

Step 5 repeat step3 to create required number of elements

Step 6 end

Program 3.7 depicts the creation of a Doubly linked list.

```
/* CREATION OF A DOUBLY LINKED LIST */
/* DBLINK.C */

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

struct dl_list
{
        int data;
        struct dl_list *right;
        struct dl_list *left;
};
typedef struct dl_list dlist;

void dl_create (dlist *);
void traverse (dlist *);

/* Function creates a simple doubly linked list */

void dl_create(dlist *start)
{
        printf("\n Input the values of the element -1111 to come out :  ");
        scanf("%d", &start->data);
        if(start->data != -1111)
        {
                start->right = (dlist *) malloc(sizeof(dlist));
                start->right->left = start;
                start->right->right = NULL;
                dl_create(start->right);
        }
        else
        start->right = NULL;
}

/* Display the list */

void traverse (dlist *start)
{
        printf("\n traversing the list using right pointer\n");
        do {
                printf(" %d = ", start->data);
                start = start->right;
        } while (start->right);  /* Show value of last start only one time */


        printf("\n traversing the list using left pointer\n");
        start=start->left;
        do
        {
                printf(" %d =", start->data);
                start = start->left;
        }while(start->right);
}
```

```
void main()
{
        dlist *head;
        head = (dlist *) malloc(sizeof(dlist));
        head->left=NULL;
        head->right=NULL;
        dl_create(head);
        printf("\n Created doubly linked list is as follows");
        traverse(head);
}
```

Program 3.7: Creation of a Doubly Linked List

OUTPUT

```
Input the values of the element -1111 to come out :
1
Input the values of the element -1111 to come out :
2
Input the values of the element -1111 to come out :
3
Input the values of the element -1111 to come out :
-1111
Created doubly linked list is as follows
traversing the list using right pointer
1 = 2 = 3 =
traversing the list using left pointer
3 = 2 = 1 =
```

3.6 CIRCULARLY LINKED LISTS IMPLEMENTATION

A linked list in which the last element points to the first element is called CIRCULAR linked list. The chains do not indicate first or last element; last element does not contain the NULL pointer. The external pointer provides a reference to starting element.

The possible operations on a circular linked list are:

- Insertion,
- Deletion, and
- Traversing

Figure 3.6 depicts a Circular linked list.

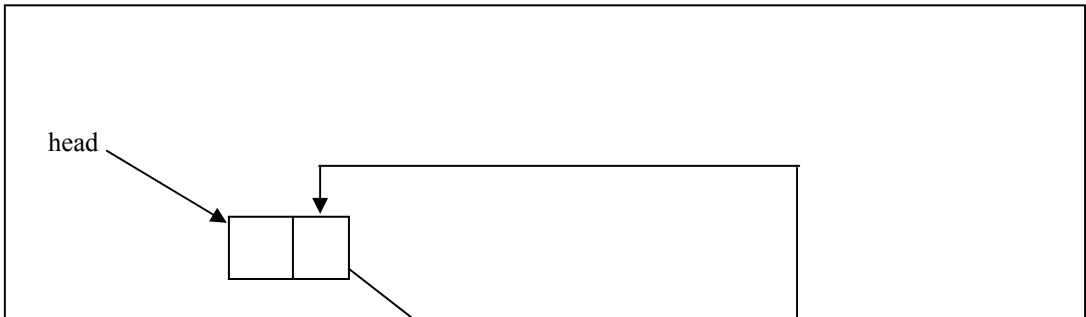


Figure 3.6 : A Circular Linked List

Program 3.8 depicts the creation of a Circular linked list.

```
#include<stdio.h>
#include<stdlib.h>
#define NULL 0

struct linked_list
{
        int data;
        struct linked_list *next;
};
typedef struct linked_list clist;

clist *head, *s;

void main()
{
        void create_clist(clist *);
        int count(clist *);
        void traverse(clist *);
        head=(clist *)malloc(sizeof(clist));
        s=head;
        create_clist(head);
        printf("  \n  traversing the created clist and the starting address is %u \n",
        head);
        traverse(head);
        printf("\n number of elements in the clist   %d \n", count(head));
}

void create_clist(clist *start)
{
        printf("input the element  -1111 for coming out of the loop\n");
        scanf("%d", &start->data);
        if(start->data == -1111)
                start->next=s;
        else
        {
                start->next=(clist*)malloc(sizeof(clist));
                create_clist(start->next);
        }
}

void traverse(clist *start)
{
```

```
        if(start->next!=s)
        {
                printf("data is %d \t next element address is %u\n", start->data, start-
>next);
                traverse(start->next);
        }
        if(start->next == s)
                printf("data is %d \t next element address is %u\n",start->data, start-
>next);
}

int count(clist *start)
{
        if(start->next == s)
                return 0;
        else
                return(1+count(start->next));
}
```

Program 3.8: Creation of a Circular linked list

ALGORITHM (Insertion of an element into a Circular Linked List)

Step 1 Begin

Step 2 if the list is empty or new element comes before the start (head) element, then insert the new element as start element.

Step 3 else, if the new element comes after the last element, then insert the new element at the end element and adjust the pointer of last element to the start element.

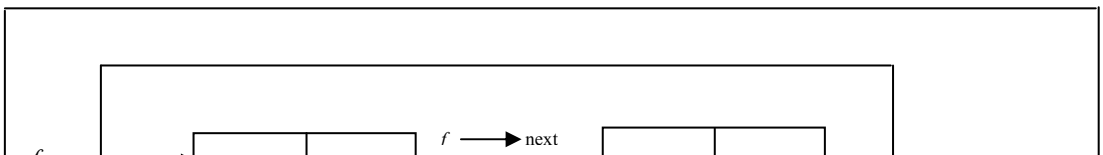
Step 4 else, insert the new element in the list by using the find function. find function returns the address of the found element to the insert_list function.

Step 5 End.

If new item is to be inserted after an existing element, then, call the find function recursively to trace the 'key' element. The new element is inserted before the 'key' element by using above algorithm.

Figure 3.7 depicts the Circular linked list with a new element that is to be inserted.

Figure 3.8 depicts a Circular linked list with the new element inserted between first and second nodes of *Figure 3.7.*



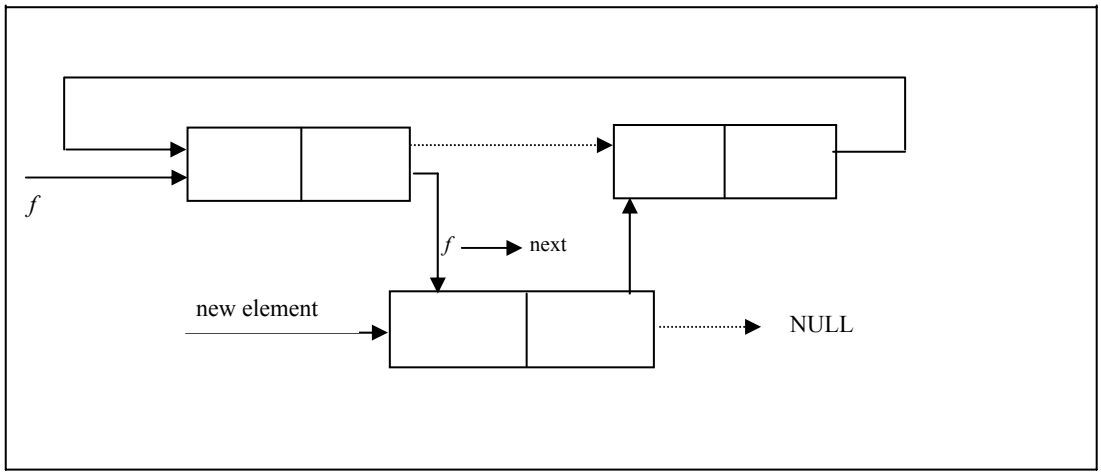


Figure 3.8: A Circular Linked List after insertion of the new element between first and second nodes (Dotted lines depict the links prior to insertion)

Program 3.9 depicts the code for insertion of a node into a Circular linked list.

```
#include<stdio.h>
#include<stdlib.h>
#define NULL 0
struct linked_list
{
        int data;
        struct linked_list *next;
};
typedef struct linked_list clist;
clist *head, *s;
/*  prototype of find and insert functions */
clist * find(clist *, int);
clist * insert_clist(clist *);
/*definition of insert_clist function */
clist * insert_clist(clist *start)    {
        clist *n, *n1;
        int key, x;
        printf("enter value of new element");
        scanf("%d", &x);
        printf("eneter value of key element");
        scanf("%d",&key);
        if(start->data ==key)
        {
```

```
                n=(clist *)malloc(sizeof(clist));
                n->data=x;
                n->next = start;
                start=n;
        }
        else
        {
                n1 = find(start, key);
                if(n1 == NULL)
                        printf("\n key is not found\n");
                else
                {
                        n=(clist*)malloc(sizeof(clist));
                        n->data=x;
                        n->next=n1->next;
                        n1->next=n;
                }
        }
        return(start);
}
/*definition of find function */
clist * find(clist *start, int key)
{
        if(start->next->data == key)
                return(start);
  if(start->next->next == NULL)
                return(NULL);
        else
                find(start->next, key);
}
void main()
{
        void create_clist(clist *);
        int count(clist *);
        void traverse(clist *);
        head=(clist *)malloc(sizeof(clist));
        s=head;
        create_clist(head);
        printf("  \n  traversing the created clist and the starting address is %u \n",
head);
        traverse(head);
        printf("\n number of elements in the clist   %d \n", count(head));
        head=insert_clist(head);
        printf("\n  traversing the clist after insert_clist and starting address is %u
\n",head);
        traverse(head);
}
void create_clist(clist *start)
{
        printf("inputthe element  -1111 for coming oout of the loop\n");
        scanf("%d", &start->data);
        if(start->data == -1111)
                start->next=s;
        else
        {
                start->next=(clist*)malloc(sizeof(clist));
                create_clist(start->next);
```

```
        }
}

void traverse(clist *start)
{
        if(start->next!=s)
        {
                printf("data is %d \t next element address is %u\n", start->data, start-
>next);
                traverse(start->next);
        }
        if(start->next == s)
                printf("data is %d \t next element address is %u\n",start->data, start-
>next);
}
int count(clist *start)
{
        if(start->next == s)
                return 0;
        else
                return(1+count(start->next));
}
```

Program 3.9 Insertion of a node into a Circular Linked List

Figure 3.9 depicts a Circular linked list from which an element was deleted.

ALGORITHM (Deletion of an element from a Circular Linked List)

Step 1 Begin
Step 2 if the list is empty, then element cannot be deleted.
Step 3 else, if element to be deleted is first node, then make the start (head) to point to the second element.
Step 4 else, delete the element from the list by calling find function and returning the found address of the element.
Step 5 End.

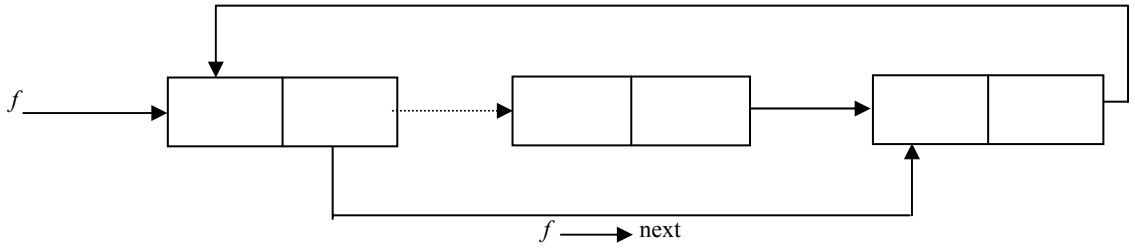


Figure 3.9 A Circular Linked List from which an element was deleted (Dotted line shows the linked that existed prior to deletion)

Program 3.10 depicts the code for the deletion of an element from the Circular linked list.

```
#include<stdio.h>
#include<stdlib.h>
#define NULL 0

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

```
typedef struct linked_list clist;
clist *head, *s;

/*  prototype of find and delete_function*/
clist * delete_clist(clist *);
clist * find(clist *, int);

/*definition of delete_clist       */
clist *delete_clist(clist *start)
{
        int key; clist * f, * temp;
        printf("\n enter the value of element to be deleted \n");
        scanf("%d", &key);
        if(start->data == key)
        {
                temp=start->next;
                free(start);
                start=temp;
        }
        else
        {
                f = find(start,key);
                if(f==NULL)
                        printf("\n key not fund");
                else
                {
                        temp = f->next->next;
                        free(f->next);
                        f->next=temp;
                }
        }
        return(start);
}
/*definition of find function */
clist * find(clist *start, int key)
{
        if(start->next->data == key)
                return(start);
  if(start->next->next == NULL)
                return(NULL);
        else
                find(start->next, key);
}

void main()
{
        void create_clist(clist *);
        int count(clist *);
        void traverse(clist *);
        head=(clist *)malloc(sizeof(clist));
        s=head;
        create_clist(head);
        printf("  \n  traversing the created clist and the starting address is %u \n",
        head);
        traverse(head);
        printf("\n number of elements in the clist   %d \n", count(head));
        head=delete_clist(head);
```

```
        printf("  \n  traversing the clist after delete_clistand starting address is %u
        \n",head);
        traverse(head);
}
void create_clist(clist *start)
{
        printf("inputthe element  -1111 for coming oout of the loop\n");
        scanf("%d", &start->data);
        if(start->data == -1111)
                start->next=s;
        else
        {
                start->next=(clist*)malloc(sizeof(clist));
                create_clist(start->next);
        }
}

void traverse(clist *start)
{
        if(start->next!=s)
        {
                printf("data is %d \t next element address is %u\n", start->data,   start-
>next);
                traverse(start->next);
        }
        if(start->next == s)
                printf("data is %d \t next element address is %u\n",start->data,    start-
>next);
}

int count(clist *start)
{
        if(start->next == s)
                return 0;
        else
                return(1+count(start->next));
}
```

Program 3.10: Deletion of an element from the circular linked list

3.7 APPLICATIONS

Lists are used to maintain POLYNOMIALS in the memory. For example, we have a function $f(x)= 7x^5 + 9x^4 - 6x^3 + 3x^2$. Figure 3.10 depicts the representation of a Polynomial using a singly linked list. 1000,1050,1200,1300 are memory addresses.

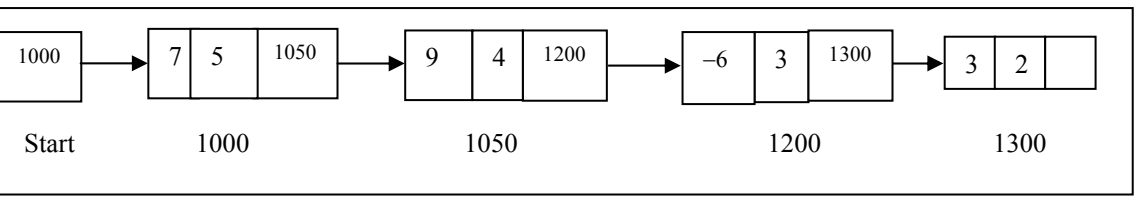


Figure 3.10: Representation of a Polynomial using a singly linked list

Polynomial contains two components, coefficient and an exponent, and 'x' is a formal parameter. The polynomial is a sum of terms, each of which consists of coefficient and an exponent. In computer, we implement the polynomial as list of structures consisting of coefficients and an exponents.

Program 3.11 accepts a Polynomial as input. It uses linked list to represent the Polynomial. It also prints the input polynomial along with the number of nodes in it.

```
/* Representation of Polynomial using Linked List */
# include <stdio.h>
# include <malloc.h>
struct link
{
        char sign;
        int coef;
        int expo;
        struct link *next;
};
typedef struct link poly;
void insertion(poly *);
void create_poly(poly *);
void display(poly *);
/* Function create a ploynomial list */
void create_poly(poly *start)
{
        char ch;
        static int i;
        printf("\n Input choice n for break: ");
        ch = getchar();
        if(ch != 'n')
        {
                printf("\n Input the sign: %d: ", i+1);
                scanf("%c", &start->sign);
                printf("\n Input the coefficient value: %d: ", i+1);
                scanf("%d", &start->coef);
                printf("\n Input the exponent value: %d: ", i+1);
                scanf("%d", &start->expo);
                fflush(stdin);
                i++;
                start->next = (poly *) malloc(sizeof(poly));
                create_poly(start->next);
        }
        else
        start->next=NULL;
}
/* Display the polynomial */
void display(poly *start)
{
        if(start->next != NULL)
        {
                printf(" %c", start->sign);
                printf(" %d", start->coef);
                printf("X^%d", start->expo);
                display(start->next);
        }
}
/* counting the number of nodes */
```

```
int count_poly(poly *start)
{
        if(start->next == NULL)
                return 0;
        else
                return(1+count_poly(start->next));
}
/* Function main */
void main()
{       poly *head = (poly *) malloc(sizeof(poly));
        create_poly(head);
        printf("\n Total nodes = %d \n", count_poly(head));
        display(head); }
```

Program 3.11: Representation of Polynomial using Linked list

☞ Check Your Progress

1) Write a function to print the memory location(s) which are used to store the data in a single linked list ?

……

……..……

.

2) Can we use doubly linked list as a circular linked list? If yes, Explain.

…….

……..……

3) Write the differences between Doubly linked list and Circular linked list.

..……

……..……...

4) Write a program to count the number of items stored in a single linked list.

..……

…….

5) Write a function to check the overflow condition of a list represented by an array.

……

……

3.8 SUMMARY

The advantage of Lists over Arrays is flexibility. Over flow is not a problem until the computer memory is exhausted. When the individual records are quite large, it may be difficult to determine the amount of contiguous storage that might be in need for the required arrays. With dynamic allocation, there is no need to attempt to allocate in advance. Changes in list, insertion and deletion can be made in the middle of the list, more quickly than in the contiguous lists.

The drawback of lists is that the links themselves take space which is in addition to the space that may be needed for data. One more drawback of lists is that they are not suited for random access. With lists, we need to traverse a long path to reach a desired node.

3.9 SOLUTIONS/ANSWERS

1)
```
void print_location(struct node *head)
{
        temp=head;
        while(temp->next !=NULL)
        {
           printf("%u", temp);
           temp=temp->next;
        }
        printf("%u", temp);
}
```

4)
```
void count_items(struct node *head)
{
        int count=0;
        temp=head;
        while(temp->next !=NULL)
        {
                count++;
        }
        count++;
        pintf("total items = %d", count);
}
```

5)
```
void Is_Overflow(int max_size, int last_element_position)
{
        if(last_element_position == max_size)
                printf("List Overflow");
        else
                printf("not Overflow");
}
```

3.10 FURTHER READINGS

1. *Fundamentals of Data Structures in C++* by E.Horowitz, Sahni and D.Mehta; Galgotia Publications

2. *Data Structures and Program Design in C* by Kruse, C.L.Tonodo and B.Leung; Pearson Education

Reference Websites

http://www.webopedia.com
http://www.ieee.org

.