UNIT-II

**LINKED LISTS:** Introduction, Types of Linked List - Singly Linked List, Doubly Linked List, Circular Linked List; Operations - Insertion, Deletion, Traverse forward/reverse order; Multi lists, Applications of Linked Lists.

**Objectives:**

Make them to implement the various operations on different types of linked lists and application of linked lists.

**Outcomes:**

1. To get expose on linked data representation.
2. To know how to manage memory address.
3. To be able to differentiate array and linked list data structure.

**Prerequisites:**

1. Knowledge on pointers in C language.
2. Knowledge on Structure Data type and its usage.

**Topics:**

1. Linked List.
2. Types of Linked Lists.
3. Operations on Linked Lists.
4. Application of Linked Lists.

Linked Lists

**2.1 Introduction**

We have studied that an array is a linear collection of data elements in which the elements arestored in consecutive memory locations. While declaring arrays, we have to specify the size of the array, which will restrict the number of elements that the array can store. For example, if we declare an array as int marks[10], then the array can store a maximum of 10 data elements but not more than that. But what if we are not sure of the number of elements in advance? Moreover, to make efficient use of memory, the elements must be stored randomly at any location rather than in consecutive locations. So, there must be a data structure that removes the restrictions on the maximum number of elements and the storage condition to write efficient programs.

Linked list is a data structure that is free from the aforementioned restrictions. A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it. However, unlike an array, a linked list does not allow random access of data. Elements in a linked list can be accessed only in a sequential manner. But like an array, insertions and deletions can be done at any point in the list in a constant time.

**2.1.1 Basic terminologies**

A linked list, in simple terms, is a linear collection of data elements. These data elements are called *nodes*. Linked list is a data structure which in turn can be used to implement other data structures. Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations. A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.



Fig 2.1 Simple linked list

From the above figure we can see a linked list in which

Every node contains two parts,

* an integer(Data field)
* a pointer to the next node.(address field)

The left part of the node which contains data may include a simple data type, an array, or a structure.

The right part of the node contains a pointer to the next node (or address of the next node in sequence). The last node will have no next node connected to it, so it will store a special value called NULL. In the above figure NULL pointer is represented by X. While programming, we usually define NULL as –1. Hence, a NULL pointer denotes the end of the list.

Since in a linked list, every node contains a pointer to another node which is of the same type, it is also called a *self-referential data type.*

Linked lists contain a pointer variable START that stores the address of the first node in the list.

We can traverse the entire list using START which contains the address of the first node; the next part of the first node in turn stores the address of its succeeding node. Using this technique, the individual nodes of the list will form a chain of nodes. If START = NULL, then the linked list is empty and contains no nodes.

In C, we can implement a linked list using the following code:

struct node

{

int data;

struct node \*next;

};

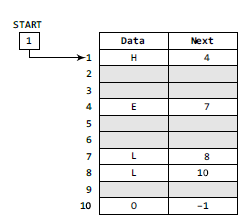


Fig 2.2 Start pointing to the first element of the linked list in memory

In theabove figure, we can see that the variable START is used to store the address of the first node. Here, in this example, START= 1, so the first data is stored at address 1, which is H. The corresponding NEXT stores the address of the next node, which is 4. So, we will look at address 4 to fetch the next data item. The second data element obtained from address 4 is E. Again, we see the corresponding NEXT to go to the next node. From the entry in the NEXT, we get the next address, that is 7, and fetch L as the data. We repeat this procedure until we reach a position where the NEXT entry contains –1 or NULL, as this would denote the end of the linked list.

In our example, the nodes for the linked list are stored at addresses 1, 4, 7, 8, and 10.

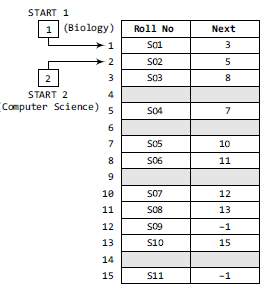
****

Fig 2.3 Two linked lists which are simultaneously sharing memory

Now, look at Fig. 2.3, two different linked lists are simultaneously maintained in the memory.There is no ambiguity in traversing through the list because each list maintains a separate Start pointer, which gives the address of the first node of their respective linked lists. The rest of the nodes are reached by looking at the value stored in the NEXT.

By looking at the figure, we can conclude that roll numbers of the students who have opted for Biology are S01, S03, S06, S08, S10, and S11. Similarly, roll numbers of the students who chose Computer Science are S02, S04, S05, S07, and S09

Types in linked list

1. Singly linked list
2. Circular linked list
3. Double linked list

**2.2 Singly linked list**

A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. By saying that the node contains a pointer to the next node, we mean that the node stores the address of the next node in sequence. A singly linked list allows traversal of data only in one way. Figure 2.4 shows a singly linked list.



Fig 2.4 singly linked list.

**2.2.1 Traversing a Linked List**

Traversing a linked list means accessing the nodes of the list in order to perform some processing on them. Remember a linked list always contains a pointer variable START which stores the address of the first node of the list. End of the list is marked by storing NULL or –1 in the NEXT field of the last node. For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed.

Algorithm

Step 1: SET PTR = START

Step 2: Repeat Steps 3 and 4 while PTR != NULL

Step 3: Apply Process to PTR🡪DATA

Step 4: SET PTR = PTR🡪NEXT

[END OF LOOP

Step 5: EXIT

Counting the number of nodes in the list

Traverse each and every node of the list and while traversing every individual node, we will increment the counter by 1. Once we reach NULL, that is, when all the nodes of the linked list have been traversed, the final value of the counter will be displayed.

Step 1: [INITIALIZE] SET Count=1

Step 2: [INITIALIZE] SET PTR = START

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

Step 4: SET count=count+1

Step 5: SET PTR = PTR NEXT

[END OF LOOP]

Step 6: Write COUNT

Step 7: EXIT

**2.2.2 Searching for a Value in a Linked List**

Searching a linked list means to find a particular element in the linked list. As already discussed, a linked list consists of nodes which are divided into two parts, the information part and the next part. So searching means finding whether a given value is present in the information part of the node or not. If it is present, the algorithm returns the address of the node that contains the value.

Step 1: [INITIALIZE] SET PTR = START

Step 2: Repeat Step 3 while PTR != NULL

Step 3: IF VAL = PTR🡪DATA

SET POS = PTR

Go To Step 5

ELSE

SET PTR = PTR🡪 NEXT

[END OF IF]

[END OF LOOP]

Step 4: SET POS = NULL

Step 5: EXIT

If we have VAL = 4, then the flow of the algorithm can be explained as shown in the figure.

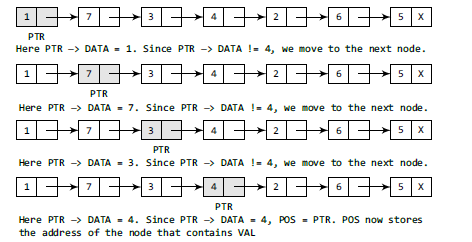


Fig 2.5 Searching linked list

**2.2.3 Inserting a New Node in a Linked List**

In this, Let us see how a new node is added into an already existing linked list. We willtake four cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning.

Case 2: The new node is inserted at the end.

Case 3: The new node is inserted after a given node.

Case 4: The new node is inserted before a given node.

Before we describe the algorithms to perform insertions in all these four cases, let us first discuss

an important term called OVERFLOW. Overflow is a condition that occurs when AVAIL = NULL or no free memory cell is present in the system. When this condition occurs, the program must give an appropriate message.

**Case 1: Inserting a Node at the Beginning of a Linked List**

Consider the linked list shown in Fig.2.6 Suppose we want to add a new node with data 9 and add it as the first node of the list. Then the following changes will be done in the linked list.

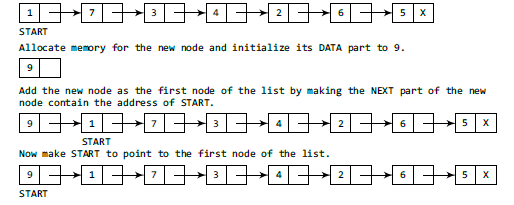


Fig 2.6 inserting a node at the beginning of linked list

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 7

[END OF IF]

Step 2: SET NEW\_NODE = AVAIL

Step 3: SET AVAIL = AVAIL🡪NEXT

Step 4: SET NEW\_NODE🡪DATA = VAL

Step 5: SET NEW\_NODE🡪NEXT = START

Step 6: SET START = NEW\_NODE

Step 7: EXIT

**Case 2: Inserting a Node at the End of a Linked List**

Take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. By the while loop,traverse through the linked list to reach thelast node. Once we reach the last node, change the NEXT pointer of the last node tostore the address of the new node. Remember that the NEXT field of the new node contains NULL, which signifies the end of the linked list.

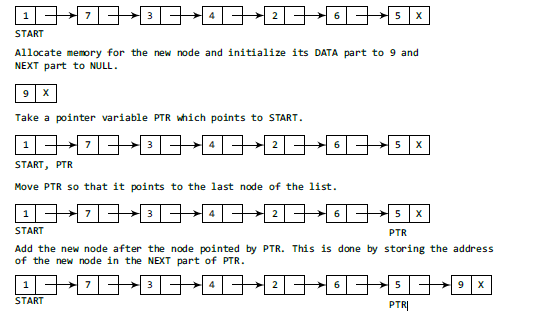
******

Fig 2.7 inserting node at the last

Algorithm

SET PTR = START

while PTR🡪NEXT != NULL

PTR = PTR🡪NEXT

PTR🡪NEXT=NEW\_NODE

**Case 3: Inserting a Node after a Given Node in a Linked List**

Suppose we want to add a new node with value 9 after the node containing data 3. we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. Then we take another pointer variable PREPTR which will be used to store the address of the node preceding PTR. Initially, PREPTR is initialized to PTR. So now, PTR, PREPTR, and START are all pointing to the first node of the linked list.

In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted after this node. Once we reach this node, we change the NEXT pointers in such a way that new node is inserted after the desired node.

SET NEW\_NODE🡪DATA = VAL

SET PTR = START

SET PREPTR = PTR

Steps 8 and 9 while PREPTR🡪DATA!=!= NUM

SET PREPTR = PTR

SET PTR = PTR🡪NEXT

[END OF LOOP]

PREPTR🡪NEXT =NEW\_NODE

SET NEW\_NODE🡪NEXT = PTR

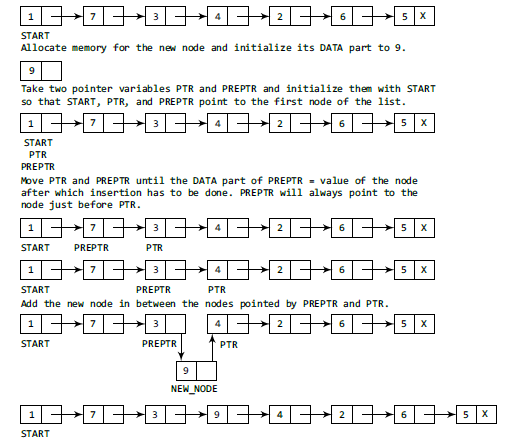


Fig 2,8 inserting an element after a given node in linked list

**Case 4: Inserting a Node Before a Given Node in a Linked List**

Suppose we want to add a new node with value 9 before the node containing 3.

We take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. Then, we take another pointer variable PREPTR and initialize it with PTR. So now, PTR, PREPTR, and START are all pointing to the first node of the linked list.

In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted before this node. Once we reach this node change the NEXT pointers in such a way that the new node is inserted before the desired node.

Algorithm

NEW\_NODE🡪DATA = VAL

SET PTR = START

SET PREPTR = PTR

while PTR🡪DATA != NUM

SET PREPTR = PTR

SET PTR = PTR🡪NEXT

[END OF LOOP]

PREPTR🡪NEXT =NEW\_NODE

NEW\_NODE🡪NEXT = PTR

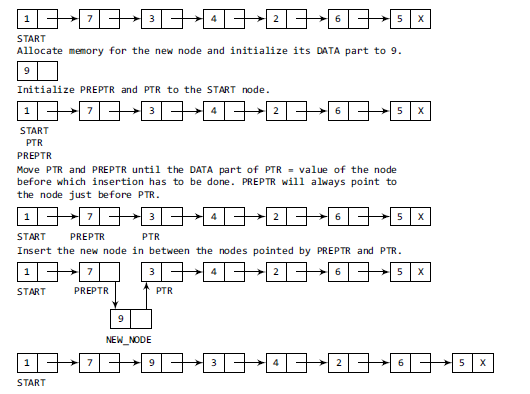


Fig 2.9 Inserting an element before a given node in linked list

**2.2.4 Deleting a Node from a Linked List**

We will consider three cases and then see how deletion is done in each case.

Case 1: The first node is deleted.

Case 2: The last node is deleted.

Case 3: The node after a given node is deleted.

Let us first discuss an important termcalled UNDERFLOW. Underflow is a condition that occurs when we try to delete a node from a linked list that is empty. This happens when START = NULL or when there are no more nodes to delete. Note that when we delete a node from a linked list, we actually have to free the memory occupied by that node.

**Case 1: Deleting the First Node from a Linked List**

When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

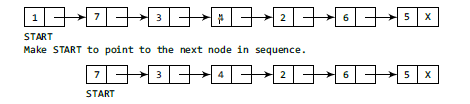


Fig 2.10Deleting the first node of a linked list

Algorithm

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 5

[END OF IF]

Step 2: SET PTR = START

Step 3: SET START = START🡪NEXT

Step 4: FREE PTR

Step 5: EXIT

**Case 2: Deleting the Last Node from a Linked List**

Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.

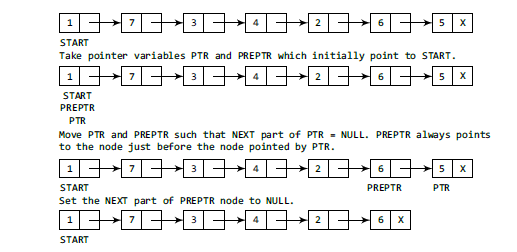


Fig 2.11 Deleting the last node of a linked list

**Algorithm**

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 8

[END OF IF]

Step 2: SET PTR = START

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

Step 4: SET PREPTR = PTR

Step 5: SET PTR = PTR NEXT

[END OF LOOP]

Step 6: SET PREPTR NEXT = NULL

Step 7: FREE PTR

Step 8: EXIT

**Case 2: Deleting the Node After a Given Node in a Linked List**

Suppose we want to delete the node that succeeds the node which contains data value 4. Then the following changes will be done in the linked list.

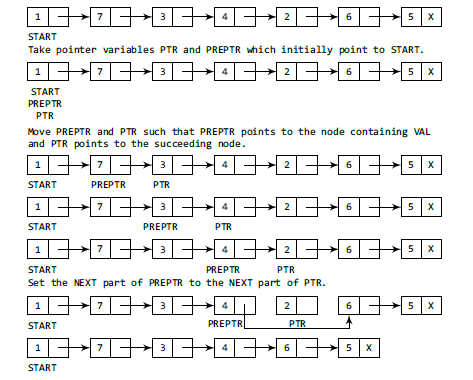


Fig 2.12 Deleting node after a given node of a linked list

In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to thefirst node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR. Once we reach the node containing VAL and the node succeeding it, we set the next pointer of the node containing VAL to the address contained in next field of the node succeeding it. The memory of the node succeeding the given node is freed and returned back to the free pool.

Algorithm

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 1

[END OF IF]

Step 2: SET PTR = START

Step 3: SET PREPTR = PTR

Step 4: Repeat Steps 5 and 6 while PREPTR DATA != NUM

Step 5: SET PREPTR = PTR

Step 6: SET PTR = PTR NEXT

[END OF LOOP]

Step 7: SET TEMP = PTR

Step 8: SET PREPTR NEXT = PTR NEXT

Step 9: FREE TEMP

Step 1 : EXIT

**Implementation**

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#include <malloc.h>

struct node

{

int data;

struct node \*next;

};

struct node \*start = NULL;

struct node \*create\_ll(struct node \*);

struct node \*display(struct node \*);

struct node \*insert\_beg(struct node \*);

struct node \*insert\_end(struct node \*);

struct node \*insert\_before(struct node \*);

struct node \*insert\_after(struct node \*);

struct node \*delete\_beg(struct node \*);

struct node \*delete\_end(struct node \*);

struct node \*delete\_node(struct node \*);

struct node \*delete\_after(struct node \*);

struct node \*delete\_list(struct node \*);

struct node \*sort\_list(struct node \*);

int main(intargc, char \*argv[]) {

int option;

do

{

printf(“\n\n \*\*\*\*\*MAIN MENU \*\*\*\*\*”);

printf(“\n 1: Create a list”);

printf(“\n 2: Display the list”);

printf(“\n 3: Add a node at the beginning”);

printf(“\n 4: Add a node at the end”);

printf(“\n 5: Add a node before a given node”);

printf(“\n 6: Add a node after a given node”);

printf(“\n 7: Delete a node from the beginning”);

printf(“\n 8: Delete a node from the end”);

printf(“\n 9: Delete a given node”);

printf(“\n 10: Delete a node after a given node”);

printf(“\n 11: Delete the entire list”);

printf(“\n 12: Sort the list”);

printf(“\n 13: EXIT”);

printf(“\n\n Enter your option : “);

scanf(“%d”, &option);

switch(option)

{

case 1: start = create\_ll(start);

printf(“\n LINKED LIST CREATED”);

break;

case 2: start = display(start);

break;

case 3: start = insert\_beg(start);

break;

case 4: start = insert\_end(start);

break;

case 5: start = insert\_before(start);

break;

case 6: start = insert\_after(start);

break;

case 7: start = delete\_beg(start);

break;

case 8: start = delete\_end(start);

break;

case 9: start = delete\_node(start);

break;

case 10: start = delete\_after(start);

break;

case 11: start = delete\_list(start);

printf(“\n LINKED LIST DELETED”);

break;

case 12: start = sort\_list(start);

break;

}

}while(option !=13);

getch();

return 0;

}

struct node \*create\_ll(struct node \*start)

{

struct node \*new\_node, \*ptr;

intnum;

printf(“\n Enter -1 to end”);

printf(“\n Enter the data : “);

scanf(“%d”, &num);

while(num!=-1)

{

new\_node = (struct node\*)malloc(sizeof(struct node));

new\_node -> data=num;

if(start==NULL)

{

new\_node -> next = NULL;

start = new\_node;

}

else

{

ptr=start;

while(ptr->next!=NULL)

ptr=ptr->next;

ptr->next = new\_node;

new\_node->next=NULL;

}

printf(“\n Enter the data : “);

scanf(“%d”, &num);

}

return start;

}

struct node \*display(struct node \*start)

{

struct node \*ptr;

ptr = start;

while(ptr != NULL)

{

printf(“\t %d”, ptr -> data);

ptr = ptr -> next;

}

return start;

}

struct node \*insert\_beg(struct node \*start)

{

struct node \*new\_node;

intnum;

printf(“\n Enter the data : “);

scanf(“%d”, &num);

new\_node = (struct node \*)malloc(sizeof(struct node));

new\_node -> data = num;

new\_node -> next = start;

start = new\_node;

return start;

}

struct node \*insert\_end(struct node \*start)

{

struct node \*ptr, \*new\_node;

intnum;

printf(“\n Enter the data : “);

scanf(“%d”, &num);

new\_node = (struct node \*)malloc(sizeof(struct node));

new\_node -> data = num;

new\_node -> next = NULL;

ptr = start;

while(ptr -> next != NULL)

ptr = ptr -> next;

ptr -> next = new\_node;

return start;

}

struct node \*insert\_before(struct node \*start)

{

struct node \*new\_node, \*ptr, \*preptr;

intnum, val;

printf(“\n Enter the data : “);

scanf(“%d”, &num);

printf(“\n Enter the value before which the data has to be inserted : “);

scanf(“%d”, &val);

new\_node = (struct node \*)malloc(sizeof(struct node));

new\_node -> data = num;

ptr = start;

while(ptr -> data != val)

{

preptr = ptr;

ptr = ptr -> next;

}

preptr -> next = new\_node;

new\_node -> next = ptr;

return start;

}

struct node \*insert\_after(struct node \*start)

{

struct node \*new\_node, \*ptr, \*preptr;

intnum, val;

printf(“\n Enter the data : “);

scanf(“%d”, &num);

printf(“\n Enter the value after which the data has to be inserted : “);

scanf(“%d”, &val);

new\_node = (struct node \*)malloc(sizeof(struct node));

new\_node -> data = num;

ptr = start;

preptr = ptr;

while(preptr -> data != val)

{

preptr = ptr;

ptr = ptr -> next;

}

preptr -> next=new\_node;

new\_node -> next = ptr;

return start;

}

struct node \*delete\_beg(struct node \*start)

{

struct node \*ptr;

ptr = start;

start = start -> next;

free(ptr);

return start;

}

struct node \*delete\_end(struct node \*start)

{

struct node \*ptr, \*preptr;

ptr = start;

while(ptr -> next != NULL)

{

preptr = ptr;

ptr = ptr -> next;

}

preptr -> next = NULL;

free(ptr);

return start;

}

struct node \*delete\_node(struct node \*start)

{

struct node \*ptr, \*preptr;

intval;

printf(“\n Enter the value of the node which has to be deleted : “);

scanf(“%d”, &val);

ptr = start;

if(ptr -> data == val)

{

start = delete\_beg(start);

return start;

}

else

{

while(ptr -> data != val)

{

preptr = ptr;

ptr = ptr -> next;

}

preptr -> next = ptr -> next;

free(ptr);

return start;

}

}

struct node \*delete\_after(struct node \*start)

{

struct node \*ptr, \*preptr;

intval;

printf(“\n Enter the value after which the node has to deleted : “);

scanf(“%d”, &val);

ptr = start;

preptr = ptr;

while(preptr -> data != val)

{

preptr = ptr;

ptr = ptr -> next;

}

preptr -> next=ptr -> next;

free(ptr);

return start;

}

struct node \*delete\_list(struct node \*start)

{

struct node \*ptr; // Lines 252-254 were modified from original code to fix

unresposiveness in output window

if(start!=NULL){

ptr=start;

while(ptr != NULL)

{

printf(“\n %d is to be deleted next”, ptr -> data);

start = delete\_beg(ptr);

ptr = start;

}

}

return start;

}

struct node \*sort\_list(struct node \*start)

{

struct node \*ptr1, \*ptr2;

int temp;

ptr1 = start;

while(ptr1 -> next != NULL)

{

ptr2 = ptr1 -> next;

while(ptr2 != NULL)

{

if(ptr1 -> data > ptr2 -> data)

{

temp = ptr1 -> data;

ptr1 -> data = ptr2 -> data;

ptr2 -> data = temp;

}

ptr2 = ptr2 -> next;

}

ptr1 = ptr1 -> next;

}

returnstart;

}

**2.3 Circular Linked list**

In a circular linked list, the last node contains a pointer to the first node of the list. We can havea circular singly linked list as well as a circular doubly linked list. While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started. Thus, a circular linked list has no beginning and

no ending. Figure 2.13 shows a circular linked list.

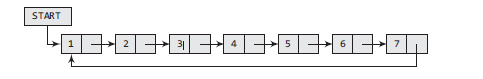


Fig 2.14 Circular linked list.

Memory representation of circular linked list

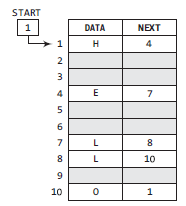


Fig 2.15 Memory representation of circular linked list

**2.3.1 Inserting a New Node in a Circular Linked List**

How a new node is added into an already existing linked list. We willtake two cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning of the circular linked list.

Case 2: The new node is inserted at the end of the circular linked list.

**Case 1: Inserting a Node at the Beginning of a Circular Linked List**

Consider the linked list shown in Fig. 6.29. Suppose we want to add a new node with data 9 as the first node of the list. Then the following changes will be done in the linked list.

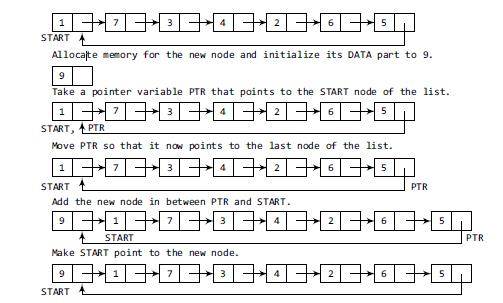


Fig 2.16inserting the new node at the beginning of circular linked list

Set the NEW\_NODEDATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START. Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of the NEW\_NODE.

While inserting a node in a circular linked list, we have to use a while loop to traverse to the last node of the list. Because the last node contains a pointer to START, its NEXT field is updated sothat after insertion it points to the new node which will be now known as START.

Step 4: SET NEW\_NODE🡪DATA = VAL

Step 5: SET PTR = START

Step 6: Repeat Step 7 while PTR🡪NEXT != START

Step 7: PTR = PTR🡪NEXT

[END OF LOOP]

Step 8: SET NEW\_NODE🡪NEXT = START

Step 9: SET PTR🡪NEXT =NEW\_NODE

Step 10 : SET START =NEW\_NODE

**Case 2: Inserting a Node at the End of a Circular Linked List**

Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.

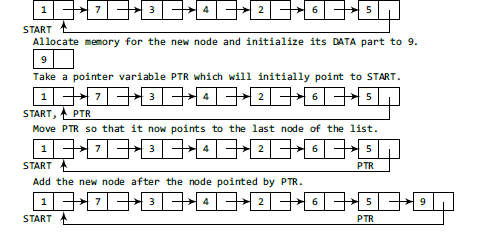


Fig 2.17 inserting the new node at the end of circular linked list

Take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 6, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains the address of the first node which is denoted by START.

Step 1: SET NEW\_NODE🡪 DATA = VAL

Step 2: SET NEW\_NODE🡪 NEXT = START

Step 3: SET PTR = START

Step 4: Repeat Step 5while PTR🡪NEXT!=START

Step 5: SET PTR = PTR🡪NEXT

[END OF LOOP]

Step 6: SET PTR🡪NEXT =NEW\_NODES

**2.3.2 Deleting a Node from a Circular Linked List**

In this section, we will discuss how a node is deleted from an already existing circular linked list. We will take two cases and then see how deletion is done in each case. Rest of the cases of deletion is same as that given for singly linked lists.

Case 1: The first node is deleted.

Case 2: The last node is deleted.

**Case 1:Deleting the First Node from a Circular Linked List**

Consider the circular linked list shown in Fig. 6.33. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

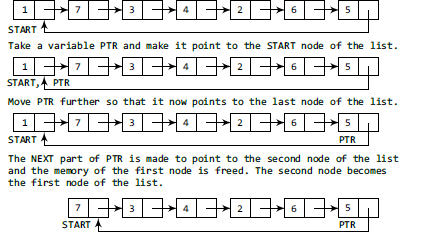


Fig 2.18deleting the first node from circular linked list

In Step 1 ofthe algorithm, we check if the linked list exists or not. If START = NULL, then it signifies that thereare no nodes in the list and the control is transferred to the last statement of the algorithm.

However, if there are nodes in the linked list, then we use a pointer variable PTR which will be used to traverse the list to ultimately reach the last node. In Step 5, we change the next pointer of the last node to point to the second node of the circular linked list. In Step 6, the memory occupied by the first node is freed. Finally, in Step 7, the second node now becomes the first node of the list and its address is stored in the pointer variable START.

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 8

[END OF IF]

Step 2: SET PTR = START

Step 3: Repeat Step 4 while PTR🡪NEXT != START

Step 4: SET PTR = PTR🡪NEXT

[END OF LOOP]

Step 5: SET PTR🡪NEXT = START🡪NEXT

Step 6: FREE START

Step 7: SET START = PTR🡪NEXT

Step 8: EXIT

**Case 2 : Deleting the Last Node from a Circular Linked List**

Consider the circular linked list shown in Fig. 2.19. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.

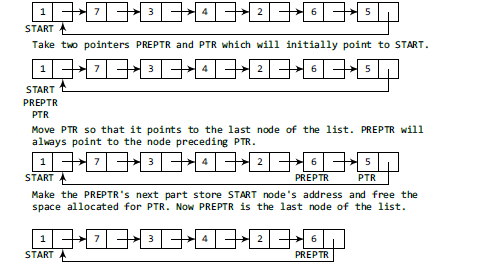


Fig 2.18deleting the last node from circular linked list

Take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that PREPTR always points to one node before PTR. Once we reach the last node and the second last node, we set the next pointer of the second last node to START, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned to the free pool.

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 8

[END OF IF]

Step 2: SET PTR = START

Step 3: Repeat Steps 4 and 5 while PTR NEXT != START

Step 4: SET PREPTR = PTR

Step 5: SET PTR = PTR NEXT

[END OF LOOP]

Step 6: SET PREPTR NEXT = START

Step 7: FREE PTR

Step 8: EXIT

**Implementation**

Write a program to create a circular linked list. Perform insertion and deletion at the beginning and end of the list.

#include <stdio.h>

#include <conio.h>

#include <malloc.h>

struct node

{

int data;

struct node \*next;

};

struct node \*start = NULL;

struct node \*create\_cll(struct node \*);

struct node \*display(struct node \*);

struct node \*insert\_beg(struct node \*);

struct node \*insert\_end(struct node \*);

struct node \*delete\_beg(struct node \*);

struct node \*delete\_end(struct node \*);

struct node \*delete\_after(struct node \*);

struct node \*delete\_list(struct node \*);

int main()

{

int option;

clrscr();

do

{

printf("\n\n \*\*\*\*\*MAIN MENU \*\*\*\*\*");

printf("\n 1: Create a list");

printf("\n 2: Display the list");

printf("\n 3: Add a node at the beginning");

printf("\n 4: Add a node at the end");

printf("\n 5: Delete a node from the beginning");

printf("\n 6: Delete a node from the end");

printf("\n 7: Delete a node after a given node");

printf("\n 8: Delete the entire list");

printf("\n 9: EXIT");

printf("\n\n Enter your option : ");

scanf("%d", &option);

switch(option)

{

case 1: start = create\_cll(start);

printf("\n CIRCULAR LINKED LIST CREATED");

break;

case 2: start = display(start);

break;

case 3: start = insert\_beg(start);

break;

case 4: start = insert\_end(start);

break;

case 5: start = delete\_beg(start);

break;

case 6: start = delete\_end(start);

break;

case 7: start = delete\_after(start);

break;

case 8: start = delete\_list(start);

printf("\n CIRCULAR LINKED LIST DELETED");

break;

}

}while(option !=9);

getch();

return 0;

}

struct node \*create\_cll(struct node \*start)

{

struct node \*new\_node, \*ptr;

intnum;

printf("\n Enter –1 to end");

printf("\n Enter the data : ");

scanf("%d", &num);

while(num!=–1)

{

new\_node = (struct node\*)malloc(sizeof(struct node));

new\_node->data = num;

if(start == NULL)

{

new\_node->next = new\_node;

start = new\_node;

}

else

{

ptr = start;

while(ptr->next != start)

ptr = ptr->next;

ptr->next = new\_node;

new\_node->next = start;

}

printf("\n Enter the data : ");

scanf("%d", &num);

}

return start;

}

struct node \*display(struct node \*start)

{

struct node \*ptr;

ptr=start;

while(ptr->next != start)

{

printf("\t %d", ptr->data);

ptr = ptr->next;

}

printf("\t %d", ptr->data);

return start;

}

struct node \*insert\_beg(struct node \*start)

{

struct node \*new\_node, \*ptr;

intnum;

printf("\n Enter the data : ");

scanf("%d", &num);

new\_node = (struct node \*)malloc(sizeof(struct node));

new\_node->data = num;

ptr = start;

while(ptr->next != start)

ptr = ptr->next;

ptr->next = new\_node;

new\_node->next = start;

start = new\_node;

return start;

}

struct node \*insert\_end(struct node \*start)

{

struct node \*ptr, \*new\_node;

intnum;

printf("\n Enter the data : ");

scanf("%d", &num);

new\_node = (struct node \*)malloc(sizeof(struct node));

new\_node->data = num;

ptr = start;

while(ptr->next != start)

ptr = ptr->next;

ptr->next = new\_node;

new\_node->next = start;

return start;

}

struct node \*delete\_beg(struct node \*start)

{

struct node \*ptr;

ptr = start;

while(ptr->next != start)

ptr = ptr->next;

ptr->next = start ->next;

free(start);

start = ptr->next;

return start;

}

struct node \*delete\_end(struct node \*start)

{

struct node \*ptr, \*preptr;

ptr = start;

while(ptr->next != start)

{

preptr = ptr;

ptr = ptr->next;

}

preptr->next = ptr->next;

free(ptr);

return start;

}

struct node \*delete\_after(struct node \*start)

{

struct node \*ptr, \*preptr;

intval;

printf("\n Enter the value after which the node has to deleted : ");

scanf("%d", &val);

ptr = start;

preptr = ptr;

while(preptr->data != val)

{

preptr = ptr;

ptr = ptr->next;

}

preptr->next = ptr->next;

if(ptr == start)

start = preptr->next;

free(ptr);

return start;

}

struct node \*delete\_list(struct node \*start)

{

struct node \*ptr;

ptr = start;

while(ptr->next != start)

start = delete\_end(start);

free(start);

return start;

}

**Output**

\*\*\*\*\*MAIN MENU \*\*\*\*\*

1: Create a list

2: Display the list

3: Add a node at the beginning

––––––––––––––––––––––––

8: Delete the entire list

9: EXIT

Enter your option : 1

Enter –1 to end

Enter the data: 1

Enter the data: 2

Enter the data: 4

Enter the data: –1

CIRCULAR LINKED LIST CREATED

Enter your option : 3

Enter your option : 5

Enter your option : 2

5 1 2 4

Enter your option : 9

**2.4 Doubly Linked Lists**

A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence as shown in Fig 2.19.

Therefore, it consists of three parts

* data,
* a pointer to the next node,
* and a pointer to the previous node.

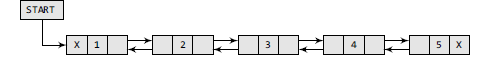


Fig 2.19 Doubly Linked list

In C, the structure of a doubly linked list can be given as,

struct node

{

struct node \*prev;

int data;

struct node \*next;

};

The PREV field of the first node and the NEXT field of the last node will contain NULL. The PREV Field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction. Thus, we see that a doubly linked list calls for more space per node and more expensive basic operations. However, a doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward). The main advantage of using a doubly linked list is that it makes searching twice as efficient. Let us view how a doubly linked list is maintained in the memory. Consider Fig. 2.20.

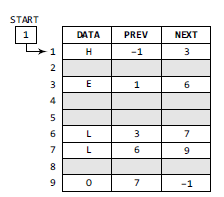


Fig 2.20 Memory representation of a doubly linked list

**2.4.1 Inserting a New Node in a Doubly Linked List**

In this section, we will discuss how a new node is added into an already existing doubly linked list. We will take four cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning.

Case 2: The new node is inserted at the end.

Case 3: The new node is inserted after a given node.

Case 4: The new node is inserted before a given node.

**Case 1: Inserting a Node at the Beginning of a Doubly Linked List**

Consider the doubly linked list shown in Fig. 2.21 Suppose we want to add a new node with data 9 as the first node of the list. Then the following changes will be done in the linked list.

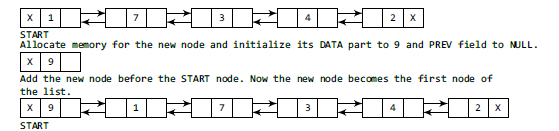


Fig 2.21 Inserting a Node at the Beginning of a Doubly Linked List

In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise, if free memory cell is available, then we allocate space for the new node. Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START. Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of NEW\_NODE.

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 9

[END OF IF]

Step 2: SET = AVAIL

Step 3: SET AVAIL = AVAIL NEXT

Step 4: SET NEW\_NODE🡪DATA = VAL

Step 5: SET NEW\_NODE🡪PREV = NULL

Step 6: SET NEW\_NODE🡪NEXT= START

Step 7: SET START🡪PREV =NEW\_NODE

Step 7: SET START =NEW\_NODE

Step 9: EXIT

**Case 2: Inserting a Node at the end of a Doubly Linked List**

Consider the doubly linked list shown in Fig. 2.22. Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.

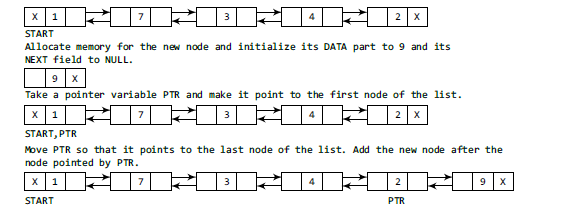


Fig 2.22 Inserting a Node at the end of a Doubly Linked List

Take a pointer variable PTR and initialize it with START. In the while loop,traverse through the linked list to reach the last node. Once we reach the last node, in Step 4, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains NULL which signifies the end of the linked list. The PREV field of the NEW\_NODE will be set so that it points to the node pointed by PTR (now the second last node of the list).

Step 1: SET PTR = START

Step 2: Repeat Step 8 while PTR🡪NEXT != NULL

Step 3: SET PTR = PTR🡪NEXT

[END OF LOOP]

Step 4: SET PTR🡪NEXT =NEW\_NODE

Step 5 : SET NEW\_NODE🡪PREV = PTR

Step 6: EXIT

**Case 3: Inserting a Node after a Given Node in a Doubly Linked List**

Consider the doubly linked list shown in Fig. 2.23. Suppose we want to add a new node with value 9 after the node containing 3.

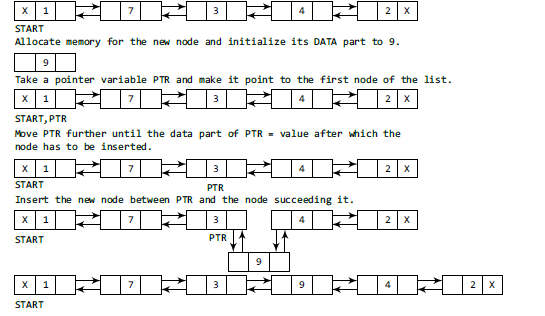
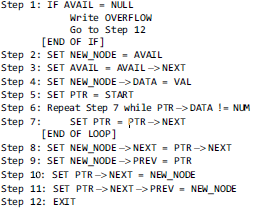


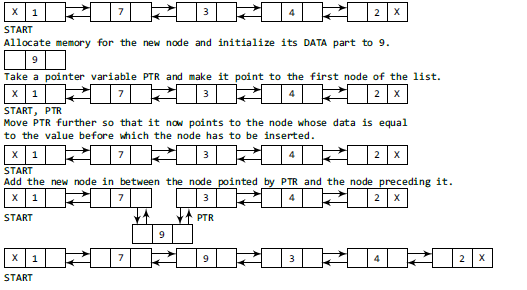
Fig 2.23Inserting a Node after a given nodein a Doubly Linked List



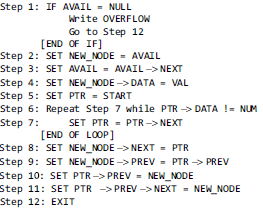
**Case 4: Inserting a node before a given node in a Doubly Linked List**

Consider the doubly linked list shown in Fig. 6.46. Suppose we want to add a new node with value 9 before the node containing 3. Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown in Fig. 6.45.

In Step 1, we first check whether memory is available for the new node. In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted before this node. Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted before the desired node.



**Fig 2.24 inserting a node before a given node in a Doubly Linked List**

****

**2.4.2 Deleting a Node from a Doubly Linked List**

In this section, we will see how a node is deleted from an already existing doubly linked list. We will take four cases and then see how deletion is done in each case.

Case 1: The first node is deleted.

Case 2: The last node is deleted.

Case 3: The node after a given node is deleted.

Case 4: The node before a given node is deleted.

**Case 1: Deleting the First Node from a Doubly Linked List**

Consider the doubly linked list shown in Fig.2.25 When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

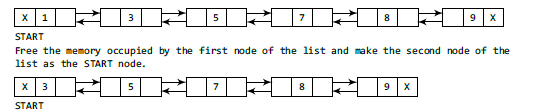


Fig 2.25 deleting the first node from the doubly linked list

To delete the node, check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm. However, if there are nodes in the linked list, then we use a temporary pointer variable PTR that is set to point to the first node of the list. For this, we initialize PTR with START that stores the address of the first node of the list. In Step 3, START is made to point to the next node in sequence and finally the memory occupied by PTR (initially the first node of the list) is freed and returned to the free pool.

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 6

[END OF IF]

Step 2: SET PTR = START

Step 3: SET START = START🡪NEXT

Step 4: SET START🡪PREV = NULL

Step 5: FREE PTR

Step 6: EXIT

**Case 2: Deleting the Last Node from a Doubly Linked List**

Consider the doubly linked list shown in Fig. 2.26. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.

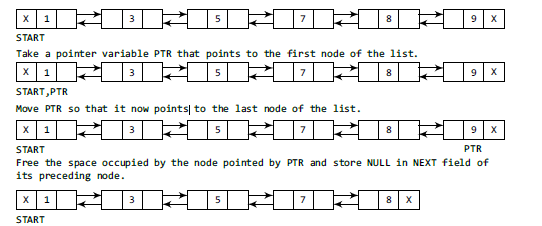


Fig 2.26 Deleting the Last Node from a Doubly Linked List

Takea pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. The while loop traverses through the list to reach the last node. Once we reach the last node, we can also access the second last node by taking its address from the PREV field of the last node. To delete the last node, we simply have to set the next field of second last node to NULL, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned to the free pool.

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 7

[END OF IF]

Step 2: SET PTR = START

Step 3: Repeat Step 4 while PTR🡪NEXT != NULL

Step 4: SET PTR = PTR🡪NEXT

[END OF LOOP]

Step 5: SET PTR🡪PREV🡪NEXT = NULL

Step 6: FREE PTR

Step 7: EXIT

**Case 3: Deleting the Node after a Given Node in a Doubly Linked List**

Consider the doubly linked list shown in Fig. 2.27. Suppose we want to delete the node that succeeds the node which contains data value 4. Then the following changes will be done in the linked list.

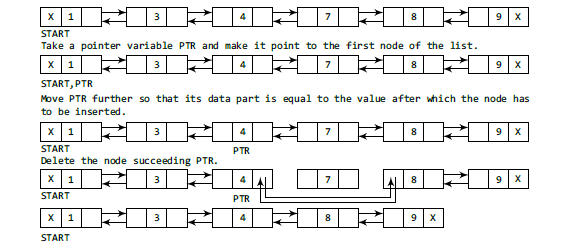
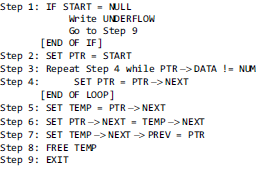


Fig 2.27 Deleting the Node after a Given Node in a Doubly Linked List

Take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the doubly linked list. The while loop traverses through the linked list to reach the given node.

Once we reach the node containing VAL, the node succeeding it can be easily accessed by using the address stored in its NEXT field. The NEXT field of the given node is set to contain the contents in the NEXT field of the succeeding node. Finally, the memory of the node succeeding the given node is freed and returned to the free pool.



**Case 4: *Deleting the Node before a Given Node in a Doubly Linked List***

Consider the doubly linked list shown in Fig. 2.28. Suppose we want to delete the node preceding the node with value 4. Before discussing the changes that will be done in the linked list, let us first look at the algorithm.

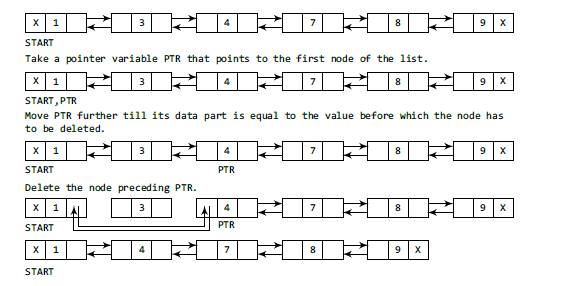
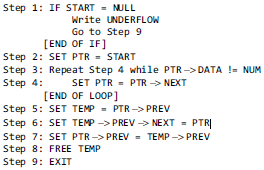


Fig 2.28 Deleting the Node before a Given Node in a Doubly Linked List

In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. The while loop traverses through the linked list to reach the desired node. Once we reach the node containing VAL, the PREV field of PTR is set to contain the address of the node precedingthe node which comes before PTR. The memory of the node preceding PTR is freed and returned to the free pool.

Hence, we see that we can insert or delete a node in a constant number of operations given only that node’s address. Note that this is not possible in the case of a singly linked list which requires the previous node’s address also to perform the same operation.

**Algorithm**



**Implementation**

struct node

{

struct node \*next;

int data;

struct node \*prev;

};

struct node \*start = NULL;

struct node \*create\_ll(struct node \*);

struct node \*display(struct node \*);

struct node \*insert\_beg(struct node \*);

struct node \*insert\_end(struct node \*);

struct node \*insert\_before(struct node \*);

struct node \*insert\_after(struct node \*);

struct node \*delete\_beg(struct node \*);

struct node \*delete\_end(struct node \*);

struct node \*delete\_before(struct node \*);

struct node \*delete\_after(struct node \*);

struct node \*delete\_list(struct node \*);

int main()

{

int option;

clrscr();

do

{

printf("\n\n \*\*\*\*\*MAIN MENU \*\*\*\*\*");

printf("\n 1: Create a list");

printf("\n 2: Display the list");

printf("\n 3: Add a node at the beginning");

printf("\n 4: Add a node at the end");

printf("\n 5: Add a node before a given node");

printf("\n 6: Add a node after a given node");

printf("\n 7: Delete a node from the beginning");

printf("\n 8: Delete a node from the end");

printf("\n 9: Delete a node before a given node");

printf("\n 10: Delete a node after a given node");

printf("\n 11: Delete the entire list");

printf("\n 12: EXIT");

printf("\n\n Enter your option : ");

scanf("%d", &option);

switch(option)

{

case 1: start = create\_ll(start);

printf("\n DOUBLY LINKED LIST CREATED");

break;

case 2: start = display(start);

break;

case 3: start = insert\_beg(start);

break;

case 4: start = insert\_end(start);

break;

case 5: start = insert\_before(start);

break;

case 6: start = insert\_after(start);

break;

case 7: start = delete\_beg(start);

break;

case 8: start = delete\_end(start);

break;

case 9: start = delete\_before(start);

break;

case 10: start = delete\_after(start);

break;

case 11: start = delete\_list(start);

printf("\n DOUBLY LINKED LIST DELETED");

break;

}

}while(option != 12);

getch();

return 0;

}

struct node \*create\_ll(struct node \*start)

{

struct node \*new\_node, \*ptr;

intnum;

printf("\n Enter –1 to end");

printf("\n Enter the data : ");

scanf("%d", &num);

while(num != –1)

{

if(start == NULL)

{

new\_node = (struct node\*)malloc(sizeof(struct node));

new\_node ->prev = NULL;

new\_node -> data = num;

new\_node -> next = NULL;

start = new\_node;

}

else

{

ptr=start;

new\_node = (struct node\*)malloc(sizeof(struct node));

new\_node–>data=num;

while(ptr–>next!=NULL)

ptr = ptr–>next;

ptr–>next = new\_node;

new\_node–>prev=ptr;

new\_node–>next=NULL;

}

printf("\n Enter the data : ");

scanf("%d", &num);

}

return start;

}

struct node \*display(struct node \*start)

{

struct node \*ptr;

ptr=start;

while(ptr!=NULL)

{

printf("\t %d", ptr -> data);

ptr = ptr -> next;

}

return start;

}

struct node \*insert\_beg(struct node \*start)

{

struct node \*new\_node;

intnum;

printf("\n Enter the data : ");

scanf("%d", &num);

new\_node = (struct node \*)malloc(sizeof(struct node));

new\_node -> data = num;

start ->prev = new\_node;

new\_node -> next = start;

new\_node ->prev = NULL;

start = new\_node;

return start;

}

struct node \*insert\_end(struct node \*start)

{

struct node \*ptr, \*new\_node;

intnum;

printf("\n Enter the data : ");

scanf("%d", &num);

new\_node = (struct node \*)malloc(sizeof(struct node));

new\_node -> data = num;

ptr=start;

while(ptr -> next != NULL)

ptr = ptr -> next;

ptr -> next = new\_node;

new\_node ->prev = ptr;

new\_node -> next = NULL;

return start;

}

struct node \*insert\_before(struct node \*start)

{

struct node \*new\_node, \*ptr;

intnum, val;

printf("\n Enter the data : ");

scanf("%d", &num);

printf("\n Enter the value before which the data has to be inserted : ");

scanf("%d", &val);

new\_node = (struct node \*)malloc(sizeof(struct node));

new\_node -> data = num;

ptr = start;

while(ptr -> data != val)

ptr = ptr -> next;

new\_node -> next = ptr;

new\_node ->prev = ptr->prev;

ptr ->prev -> next = new\_node;

ptr ->prev = new\_node;

return start;

}

struct node \*insert\_after(struct node \*start)

{

struct node \*new\_node, \*ptr;

intnum, val;

printf("\n Enter the data : ");

scanf("%d", &num);

printf("\n Enter the value after which the data has to be inserted : ");

scanf("%d", &val);

new\_node = (struct node \*)malloc(sizeof(struct node));

new\_node -> data = num;

ptr = start;

while(ptr -> data != val)

ptr = ptr -> next;

new\_node ->prev = ptr;

new\_node -> next = ptr -> next;

ptr -> next ->prev = new\_node;

ptr -> next = new\_node;

return start;

}

struct node \*delete\_beg(struct node \*start)

{

struct node \*ptr;

ptr = start;

start = start -> next;

start ->prev = NULL;

free(ptr);

return start;

}

struct node \*delete\_end(struct node \*start)

{

struct node \*ptr;

ptr = start;

while(ptr -> next != NULL)

ptr = ptr -> next;

ptr ->prev -> next = NULL;

free(ptr);

return start;

}

struct node \*delete\_after(struct node \*start)

{

struct node \*ptr, \*temp;

intval;

printf("\n Enter the value after which the node has to deleted : ");

scanf("%d", &val);

ptr = start;

while(ptr -> data != val)

ptr = ptr -> next;

temp = ptr -> next;

ptr -> next = temp -> next;

temp -> next ->prev = ptr;

free(temp);

return start;

}

struct node \*delete\_before(struct node \*start)

{

struct node \*ptr, \*temp;

intval;

printf("\n Enter the value before which the node has to deleted : ");

scanf("%d", &val);

ptr = start;

while(ptr -> data != val)

ptr = ptr -> next;

temp = ptr ->prev;

if(temp == start)

start = delete\_beg(start);

else

{

ptr ->prev = temp ->prev;

temp ->prev -> next = ptr;

}

free(temp);

return start;

}

struct node \*delete\_list(struct node \*start)

{

while(start != NULL)

start = delete\_beg(start);

return start;

}

**2.5 Multi-Linked Lists**

In a multi-linked list, each node can have n number of pointers to other nodes. A doubly linkedlist is a special case of multi-linked lists. However, unlike doubly linked lists, nodes in a multilinked list may or may not have inverses for each pointer. We can differentiate a doubly linked list from a multi-linked list in two ways:

(a) A doubly linked list has exactly two pointers. One pointer points to the previous node and the other points to the next node. But a node in the multi-linked list can have any number of pointers.

(b) In a doubly linked list, pointers are exact inverses of each other, i.e., for every pointer which points to a previous node there is a pointer which points to the next node. This is not true for a multi-linked list.

Multi-linked lists are generally used to organize multiple orders of one set of elements. For example, if we have a linked list that stores name and marks obtained by students in a class, then we can organize the nodes of the list in two ways:

(i) Organize the nodes alphabetically (according to the name)

(ii) Organize the nodes according to decreasing order of marks so that the information of student who got highest marks comes before other students.

Figure 2.29 shows a multi-linked list in which students’ nodes are organized by both the

aforementioned ways.

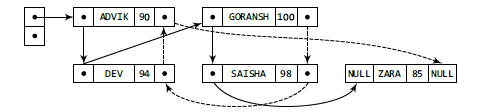


Fig 2.29 Multi-linked list that stores names alphabetically as well as according to decreasing order of marks

**2.6 Applications of Linked Lists**

Linked lists can be used to represent polynomials and the different operations that can be performed on them. In this section, we will see how polynomials are represented in the memory using linked lists.



Fig 2.30 Linked representation of polynomial

Let us see how a polynomial is represented in the memory using a linked list. Consider a polynomial 6x3 + 9x2 + 7x + 1. Every individual term in a polynomial consists of two parts, a coefficient and a power. Here, 6, 9, 7, and 1 are the coefficients of the terms that have 3, 2, 1, and 0 as their powers respectively.

Every term of a polynomial can be represented as a node of the linked list. Figure 6.74 shows

the linked representation of the terms of the above polynomial.

**Review Questions**

**1.** Make a comparison between a linked list and a linear array. Which one will you prefer to use and when?

**2.** Why is a doubly linked list more useful than a singly linked list?

**3.** Give the advantages and uses of a circular linked list.

**4.** Specify the use of a header node in a header linked list.

**5.** Give the linked representation of the following polynomial:

7*x3y2* – 8x2y+ 3*xy* + 11*x* – 4

**6.** Explain the difference between a circular linked list and a singly linked list.

**7.** Form a linked list to store students’ details.

**8.** Use the linked list of the above question to insert the record of a new student in the list.

**9.** Delete the record of a student with a specified roll number from the list maintained in Question 7.

**10.** Given a linked list that contains English alphabet. The characters may be in upper case or in lower case. Create two linked lists—one which stores upper case characters and the other that stores lower case characters.

**11.** Create a linked list which stores names of the employees. Then sort these names and re-display the contents of the linked list.

**Programming Exercises**

**1.** Write a program that removes all nodes that have duplicate information.

**2.** Write a program to print the total number of occurrences of a given item in the linked list.

**3.** Write a program to multiply every element of the linked list with 10.

**4.** Write a program to print the number of non-zero elements in the list.

**5.** Write a program that prints whether the given linked list is sorted (in ascending order) or not.

**6.** Write a program that copies a circular linked list.

**7.** Write a program to merge two linked lists.

**8.** Write a program to sort the values stored in a doubly circular linked list.

**9.** Write a program to merge two sorted linked lists. The resultant list must also be sorted.

**10.** Write a program to delete the first, last, and middle node of a header linked list.

**11.** Write a program to create a linked list from an already given list. The new linked list must contain every alternate element of the existing linked list.

**12.** Write a program to concatenate two doubly linked lists.

**13.** Write a program to delete the first element of a doubly linked list. Add this node as the last node of the list.

**14.** Write a program to

(a) Delete the first occurrence of a given character in a linked list

(b) Delete the last occurrence of a given character

(c) Delete all the occurrences of a given character

**15.** Write a program to reverse a linked list using recursion.

**16.** Write a program to input an n digit number. Now, break this number into its individual digits and then store every single digit in a separate node thereby forming a linked list. For example, if you enter 12345, then there will 5 nodes in the list containing nodes with values 1, 2, 3, 4, 5.

**17.** Write a program to add the values of the nodes of a linked list and then calculate the mean.

**18.** Write a program that prints minimum and maximum values in a linked list that stores integer values.

**19.** Write a program to interchange the value of the first element with the last element, second element with second last element, so on and so forth of a doubly linked list.

**20.** Write a program to make the first element of singly linked list as the last element of the list.

**21.** Write a program to count the number of occurrences of a given value in a linked list.

**22.** Write a program that adds 10 to the values stored in the nodes of a doubly linked list.

**23.** Write a program to form a linked list of floating point numbers. Display the sum and mean of thesenumbers.

**24.** Write a program to delete the kthnode from a linked list.

**25.** Write a program to perform deletions in all the cases of a circular header linked list.

**26.** Write a program to multiply a polynomial with a given number.

**27.** Write a program to count the number of non-zero values in a circular linked list.

**28.** Write a program to create a linked list which stores the details of employees in a department. Read and print the information stored in the list.

**29.** Use the linked list of Question 28 so that it displays the record of a given employee only.

**30.** Use the linked list of Question 28 and insert information about a new employee.

**31.** Use the linked list of Question 28 and delete information about an existing employee.

**32.** Write a program to move a middle node of a doubly link list to the top of the list.

**33.** Write a program to create a singly linked list and reverse the list by interchanging the links and not the data.

**34.** Write a program that prints the nth element from the end of a linked list in a single pass.

**35.** Write a program that creates a singly linked list. Use a function issorted that returns 1 if the list is sorted and 0 otherwise.

**36.** Write a program to interchange the kth and the (k+1)th node of a circular doubly linked list.

**37.** Write a program to create a header linked list.

**38.** Write a program to delete a node from a circular header linked list.

**39.** Write a program to delete all nodes from a header linked list that has negative values in its data part.

**Multiple-choice Questions**

**1.** A linked list is a

(a) Random access structure

(b) Sequential access structure

(c) Both

(d) None of these

**2.** An array is a

(a) Random access structure

(b) Sequential access structure

(c) Both

(d) None of these

**3.** Linked list is used to implement data structures like

(a) Stacks (b) Queues

(c) Trees (d) All of these

**4.** Which type of linked list contains a pointer to the next as well as the previous node in the sequence?

(a) Singly linked list (b) Circular linked list

(c) Doubly linked list (d) All of these

**5.** Which type of linked list does not store NULL in next field?

(a) Singly linked list (b) Circular linked list

(c) Doubly linked list (d) All of these

**6.** Which type of linked list stores the address of the header node in the next field of the last node?

(a) Singly linked list

(b) Circular linked list

(c) Doubly linked list

(d) Circular header linked list

**7.** Which type of linked list can have four pointers per node?

(a) Circular doubly linked list

(b) Multi-linked list

(c) Header linked list

(d) Doubly linked list

**True/False**

1. A node in a linked list can point to only one node at a time.
2. A node in a singly linked list can reference the previous node.
3. A linked list can store only integer values.
4. Linked list is a random access structure.
5. Deleting a node from a doubly linked list is easier than deleting it from a singly linked list.
6. Every node in a linked list contains an integer part and a pointer.
7. START stores the address of the first node in the list.
8. Underflow is a condition that occurs when we try to delete a node from a linked list that is empty.

**Fill in the Blanks**

**1.** \_\_\_\_\_\_ is used to store the address of the first free memory location.

**2.** The complexity to insert a node at the beginning of the linked list is \_\_\_\_\_\_.

**3.** The complexity to delete a node from the end of the linked list is \_\_\_\_\_\_.

**4.** Inserting a node at the beginning of the doublylinked list needs to modify \_\_\_\_\_\_ pointers.

**5.** Inserting a node in the middle of the singly linked list needs to modify \_\_\_\_\_\_ pointers.

**6.** Inserting a node at the end of the circular linked list needs to modify \_\_\_\_\_\_ pointers.

**7.** Inserting a node at the beginning of the circular doubly linked list needs to modify \_\_\_\_\_\_ pointers.

**8.** Deleting a node from the beginning of the singly linked list needs to modify \_\_\_\_\_\_ pointers

**9.** Deleting a node from the middle of the doubly linked list needs to modify \_\_\_\_\_\_ pointers.

**10.** Deleting a node from the end of a circular linked list needs to modify \_\_\_\_\_\_ pointers.

**11.** Each element in a linked list is known as a \_\_\_\_\_\_.

**12.** First node in the linked list is called the \_\_\_\_\_\_.

**13.** Data elements in a linked list are known as \_\_\_\_\_\_.

**14.** Overflow occurs when \_\_\_\_\_\_.

**15.** In a circular linked list, the last node contains a pointer to the \_\_\_\_\_\_ node of the list.