**ASSIGNMENT – 23**

**1.PROBLEM STATEMENT**

Write a program in C to implement doubly linked list.

**2.ALGORITHMS**

Algorithm **Create\_Node**

**Input:** The item to be stored in a node

**Output:** The address to a node constructed in memory with item stored in its data part

**Remarks:** The node allocated has three sections: a data section to store integer data, a prev section to reference the previous node and a next section to reference the next node in a linked list.

**Steps:**

1. A node is allocated in memory and it’s address is stored in a pointer ptr
2. Set the next part to a null value
3. Set the prev part to a null value
4. Set the data part to the input item
5. **Return** ptr
6. **Stop**

Algorithm **Insert\_At\_End**

**Input:** The address of the first node ‘start’ of the list and the item to be inserted.

**Output:** The item stored in a node and added to the linked list at end.

**Remarks:** Node is created only when sufficient memory is available.

**Steps:**

1. ptr=**Create\_Node**(item)
2. **If**(ptr=null) **then** //failed allocation of ptr
3. **Print** “Insufficient memory”
4. **Exit**
5. **EndIf**
6. **If**(start=null)
7. start=ptr //update the address in start
8. end=ptr //set end to ptr
9. **Else**
10. end.NEXT=ptr//point end link to ptr
11. ptr.PREV=end
12. end=ptr //set end to ptr
13. **EndIf**
14. **Return** start
15. **Stop**

Algorithm **Insert\_At\_Front**

**Input:** The address of the first node ‘start’ of the list and the item to be inserted.

**Output:** The item stored in a node and added to the linked list at front.

**Remarks:** Node is created only when sufficient memory is available.

**Steps:**

1. ptr=**Create\_Node**(item)
2. **If**(ptr=null) **then** //failed allocation of ptr
3. **Print** “Insufficient memory”
4. **Exit**
5. **EndIf**
6. **If**(start=null)
7. start=ptr //update start
8. end=ptr //update end
9. **Else**
10. ptr.NEXT=start //point ptr link to start
11. start=ptr //reset start to ptr
12. **EndIf**
13. **Return** start
14. **Stop**

Algorithm **Insert\_At\_Any\_Position**

**Input:** The address of the first node ‘start’ of the list, the item to be inserted and the position ‘pos’ at which it is to be inserted in the list.

**Output:** The item stored in a node and added to the linked list at the position ‘pos’

**Remarks:** Node is created only when sufficient memory is available.

**Steps:**

1. **count=1**
2. **If**(pos=1) **then** //if insertion is to be done at front
3. start=**Insert\_At\_Front**(start,item)
4. **Else If**(pos=max) **then** //if insertion needs to be done at end
5. start=**Insert\_At\_End**(start,item)
6. **Else**
7. ptr=**Create\_Node**(item)
8. **If**(ptr=null) **then** //failed allocation of ptr
9. **Print** “insufficient memory”
10. **Exit**
11. **EndIf**
12. temp=start //set a temp pointer to start
13. **While**(count≠pos) **do** //count till pos is reached
14. count=count+1
15. temp=temp.NEXT //point temp to temp link
16. **EndWhile**
17. temp2=temp.NEXT
18. ptr.NEXT=temp2 //point ptr link to temp link
19. ptr.PREV=temp
20. temp.NEXT=ptr
21. temp2.PREV=ptr
22. **EndIf**
23. **Return** start
24. **Stop**

Algorithm **Delete\_From\_Front**

**Input:** The address of the first node ‘start’ of the linked list

**Output:** The first node of the list deleted and freed from memory.

**Remarks:** The list should not be empty

**Steps:**

1. ptr=start //set ptr to beginning of the list
2. **If**(start.NEXT=null) **then**
3. start=null
4. **Else**
5. start=start.NEXT
6. Start->PREV=null
7. **EndIf**
8. free the allocated pointer ptr from memory
9. **Return** start
10. **Stop**

Algorithm **Delete\_From\_End**

**Input:** The address of the first node ‘start’ of the linked list

**Output:** The last node of the list deleted and freed from memory.

**Remarks:** The list should not be empty

**Steps:**

1. ptr=start
2. **While**(ptr.NEXT≠end) **do**
3. ptr=ptr.NEXT
4. **EndWhile**
5. temp=end
6. end=ptr
7. end.NEXT=null //set end pointer to null
8. The allocated memory at address temp is freed
9. **Stop**

Algorithm **Delete\_From\_Any\_Position**

**Input:** The address of the first node ‘start’ of the list and the position ‘pos’ of the node which is to be deleted.

**Output:** The node at position ‘pos’ deleted from the list and freed from memory.

**Remarks:** The list should not be empty.

**Steps:**

1. max = **Get\_Length**(start) //get number of nodes in list
2. count=1 //loop counter
3. **If**(pos=max) **then** //if end element is to be deleted
4. **Delete\_From\_End**(start)
5. **Else If**(pos=1) **then //**if front element is to be deleted
6. start=**Delete\_From\_Front**(start)
7. **Else**
8. temp=start
9. **While**(count≠pos) **do** //while pos is not reached
10. count=count+1
11. temp=temp.NEXT //traverse the list
12. **EndWhile**
13. temp2=temp.NEXT //point temp2 to temp link
14. temp3=temp2.NEXT
15. temp.NEXT=temp3 //point temp link to temp2 link
16. temp3.PREV=temp
17. The allocated memory at address temp2 is freed
18. **EndIf**
19. **Return** start
20. **Stop**

Algorithm **Get\_Length**

**Input:** The address of the first node ‘start’ of the list.

**Output:** The total number of nodes in the list.

**Remarks:** The list should not be empty.

**Steps:**

1. temp=start
2. **While**(temp≠null) **do** //traverse till the end of the list
3. count=count+1
4. temp=temp.LINK
5. **Return** start
6. **Stop**

Algorithm **Display\_List**

**Input:** The address of the first node ‘start’ of the list.

**Output:** The data of the list displayed to the user

**Remarks:** The list should not be empty.

**Steps:**

1. temp=start
2. **If**(start=null)
3. **Print** “the list is empty”
4. **Return**
5. **EndIf**
6. **While**(temp≠null) **do** //traverse till the end of the list
7. Print the data section of each node
8. temp=temp.LINK
9. **Return**
10. **Stop**

Algorithm **Display\_List\_In\_Reverse**

**Input:** The address of the first node ‘start’ of the list.

**Output:** The data of the list displayed in reverse order

**Remarks:** The list should not be empty.

**Steps:**

1. ptr=end
2. **If**(start=null) **then**
3. **Print** “list is empty”
4. **Exit**
5. **EndIf**
6. **While**(ptr≠null) **do**
7. display the data part of each node
8. ptr=ptr.PREV
9. **EndWhile**
10. **Stop**

**3.SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

typedef struct node

{

    struct node\* prev;

    int data;

    struct node\* next;

}node;

node\* createnode(int item)

{

    node\* ptr;

    ptr=(node\*)malloc(sizeof(node));

    ptr->data=item;

    ptr->next=NULL;

    ptr->prev=NULL;

    return ptr;

}

node\* end;

void displist(node \*start)

{

    node \*temp;

    temp=start;

    if(start==NULL)

    {

        printf("LIST EMPTY");

        return;

    }

    printf("LIST: ");

    while(temp!=NULL)

    {

        printf("%d ",temp->data);

        temp=temp->next;

    }

}

node\* endinsert(node \*start,int item)

{

    node \*ptr,\*temp;

    ptr=createnode(item);

    if(start==NULL)

    {

        start=ptr;

        end=ptr;

        return start;

    }

    else

    {

        end->next=ptr;

        ptr->prev=end;

        end=ptr;

        return start;

    }

}

node\* frontinsert(node \*start,int item)

{

    node \*ptr;

    ptr=createnode(item);

    if(start==NULL)

    {

        start=ptr;

        end=ptr;

        return start;

    }

    else

    {

        ptr->next=start;

        start=ptr;//caution

        return start;

    }

}

int getmax(node\* start)

{

    int count=0;

    node\* temp=start;

    while(temp!=NULL)

    {

        temp=temp->next;

        count++;

    }

    return count;

}

node\* insertatany(node\* start,int item,int pos)

{

    node\* ptr,\*temp,\*temp2;

    int count=1,max=getmax(start);

    if(pos==1)

        start=frontinsert(start,item);

    else if(pos==max)

        start=endinsert(start,item);

    else

    {

        ptr = createnode(item);

        temp = start;

        while(count!=pos-1)

        {

            count++;

            temp=temp->next;

        }

        temp2=temp->next;

        ptr->next=temp2;

        ptr->prev=temp;

        temp->next=ptr;

        temp2->prev=ptr;

    }

        return start;

}

node\* frontdelete(node \*start)

{

    node \*ptr;

    ptr=start;

    if(start->next==NULL)

        start=NULL;

    else

    {

        start=start->next;

        start->prev=NULL;

    }

    free(ptr);

    return start;

}

void enddelete(node \*start)

{

    node \*temp,\*temp2;

    temp=start;

    while((temp->next!=end))

        temp=temp->next;

    temp2=end;

    end=temp;

    end->next=NULL;

    free(temp2);

}

node\* deletefromany(node\* start,int pos)

{

    node\* temp,\*temp2,\*temp3;

    int max=getmax(start);

    int count=0;

    if(pos==max)

        enddelete(start);

    else if(pos==1)

    {

        start=frontdelete(start);

        return start;

    }

    else

    {

        temp=start;

        while(count!=pos-2)

        {   count++;

            temp=temp->next;

        }

        temp2=temp->next;

        temp3=temp2->next;

        temp->next=temp3;

        temp3->prev=temp;

        free(temp2);

    }

    return start;

}

void disprev(node\* start)

{

    node\* ptr;

    ptr=end;

    if(start==NULL)

    {

        printf("LIST EMPTY");

        return;

    }

    printf("REVERSE LIST: ");

    while(ptr!=NULL)

    {

        printf("%d ",ptr->data);

        ptr=ptr->prev;

    }

}

int main(void)

{

    node \*start=NULL;

    int ch,item,pos,max;

    while(1)

    {

        printf("\nMENU:");

        printf("\n1.INSERT AT END\n2.INSERT AT FRONT\n3.INSERT AT ANY POSITION\n4.DELETE FROM FRONT\n5.DELETE FROM END\n6.DELETE FROM ANY POSITION\n7.DISPLAY LIST\n8.DISPLAY LIST IN REVERSE\n9.EXIT");

        printf("\nEnter Your Choice:");

        scanf("%d",&ch);

        switch(ch)

        {

            case 1:

            printf("Enter the element to be inserted: ");

            scanf("%d",&item);

            start=endinsert(start,item);

            displist(start);

            break;

            case 2:

            printf("Enter the element to be inserted:");

            scanf("%d",&item);

            start=frontinsert(start,item);

            displist(start);

            break;

            case 3:

            if(start==NULL)

                start=endinsert(start,item);

            else

            {

                printf("Enter the insertion position: ");

                scanf("%d",&pos);

                max=getmax(start);

                //Checking validity of given position

                if(pos<1||pos>max+1)

                    printf("\nInvalid Position\n");

                else

                {

                    printf("Enter the element to be inserted:");

                    scanf("%d",&item);

                    start=insertatany(start,item,pos);

                }

            }

            displist(start);

            break;

            case 4:

            if(start==NULL)

                printf("LIST EMPTY");

            else

            {

                start=frontdelete(start);

                displist(start);

            }

            break;

            case 5:

            if(start==NULL)

                printf("LIST EMPTY");

            else

            {

                if(start->next==NULL)

                    start=frontdelete(start);

                else

                    enddelete(start);

                displist(start);

            }

            break;

            case 6:

            if(start==NULL)

                printf("LIST EMPTY");

            else

            {

                printf("Enter the deletion position: ");

                scanf("%d",&pos);

                max=getmax(start);

                if(pos<1 || pos>max)

                    printf("Invalid position");

                else

                {

                    start=deletefromany(start,pos);

                    displist(start);

                }

            }

            break;

            case 7:

                displist(start);

            break;

            case 8:

                disprev(start);

            break;

            case 9:

                exit(0);

            break;

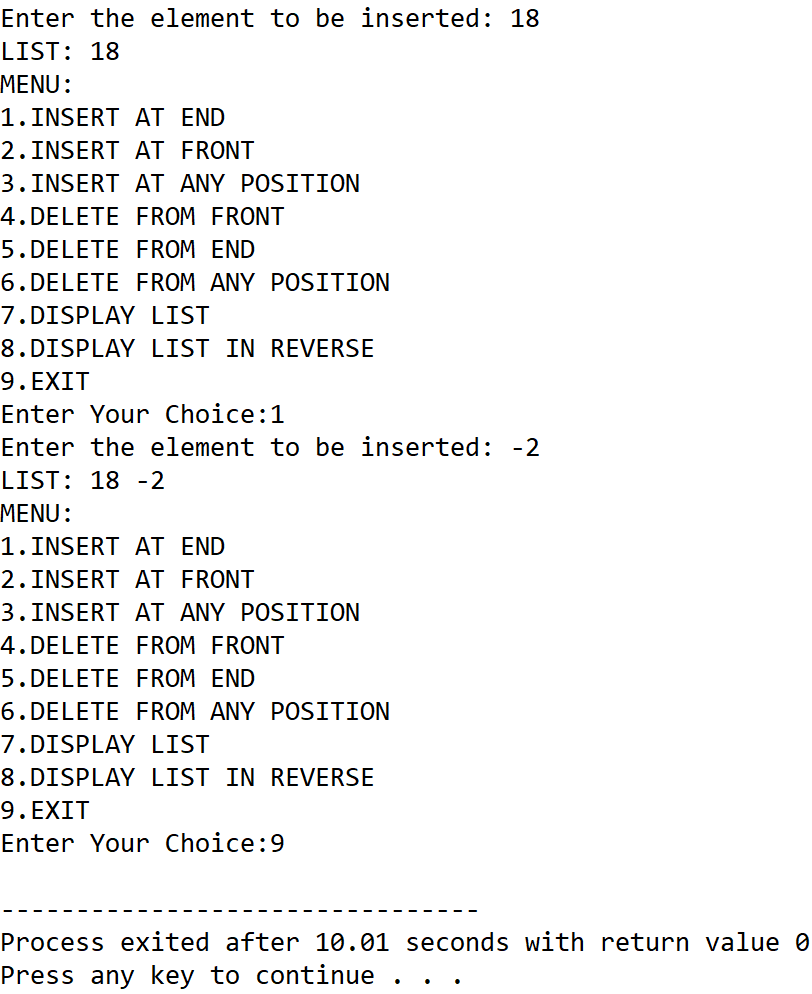
        }

    }

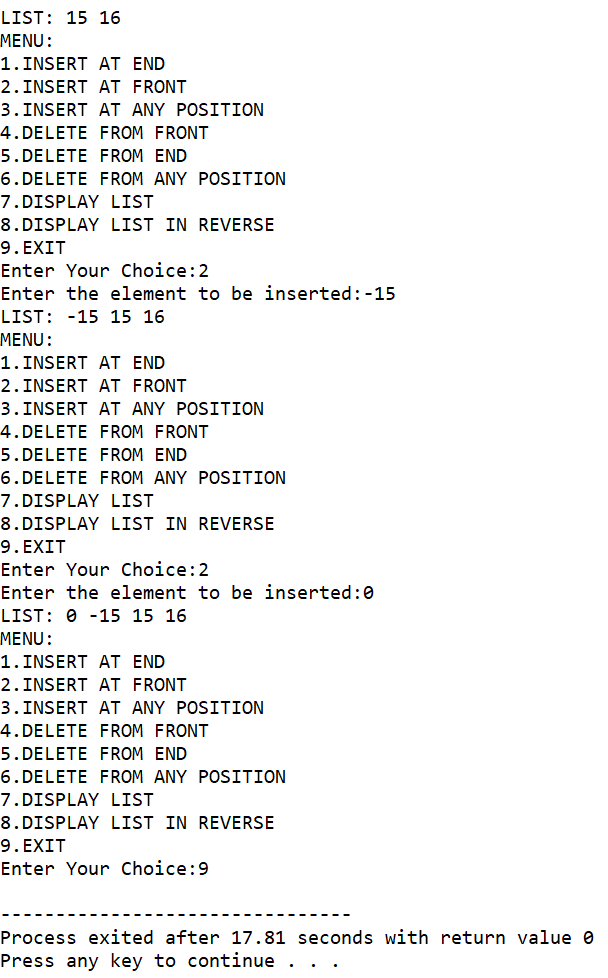
}

**4.OUTPUT**

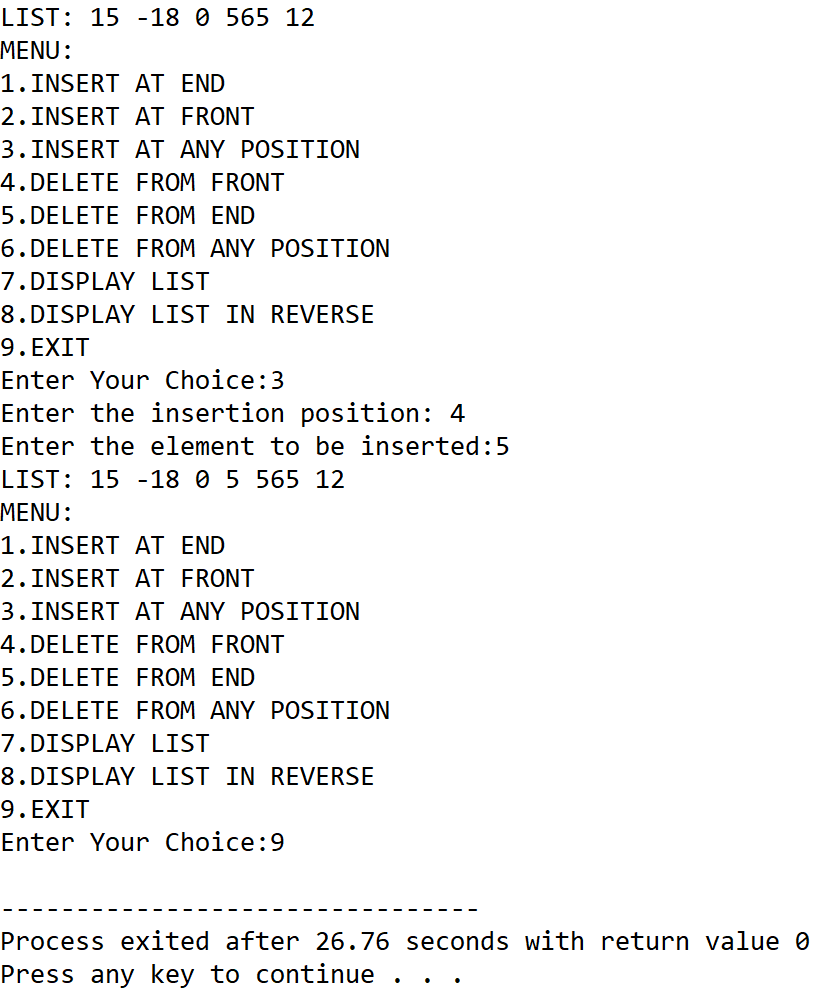
**SET 1:** Insert at end



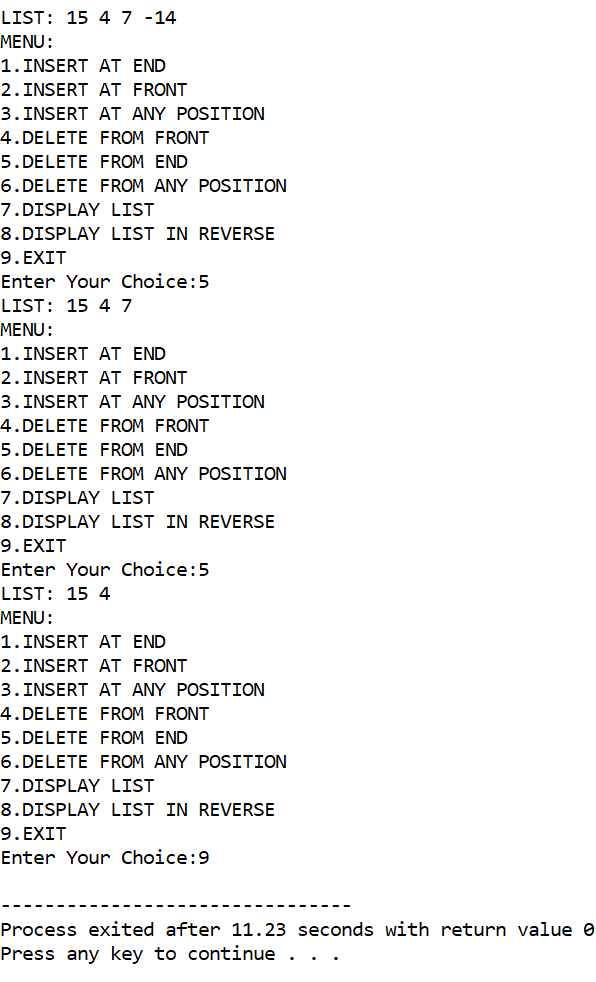
**SET 2:** Insert at front



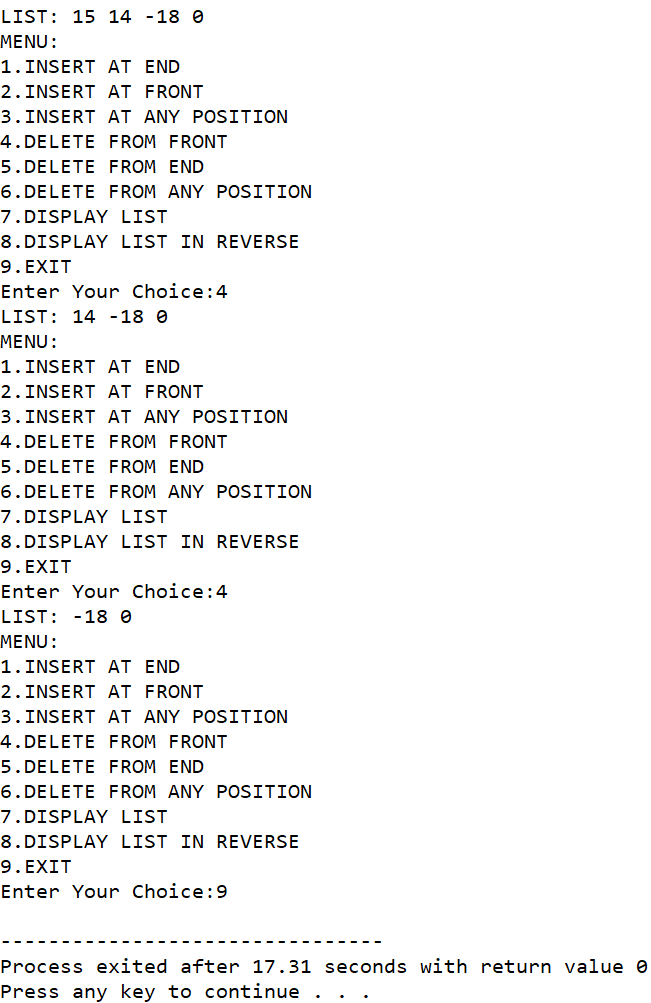
**SET 3:** Insert at any position



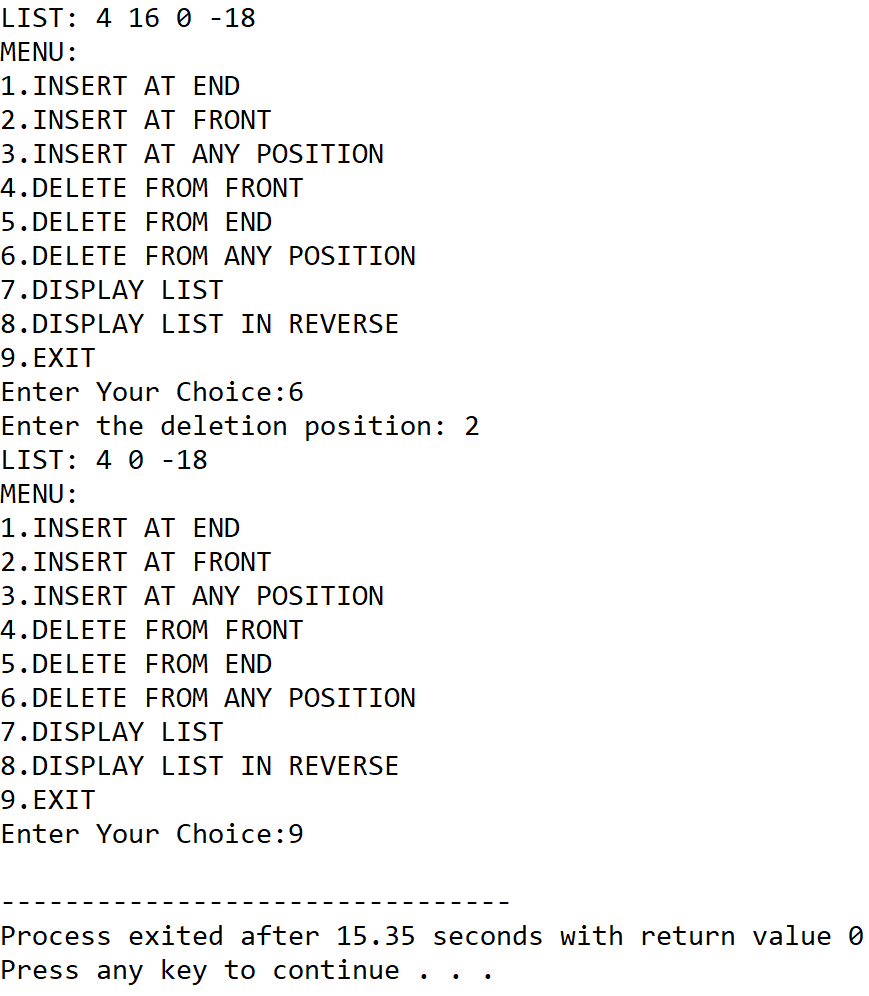
**SET 4:** Delete from end



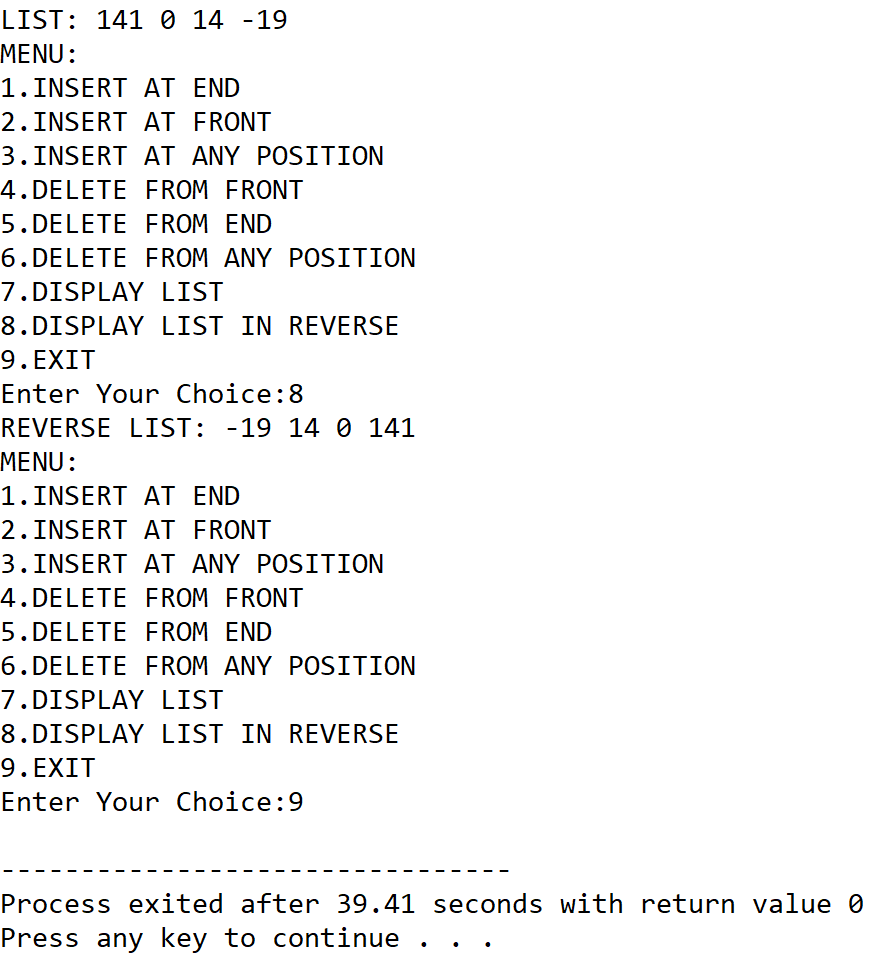
**SET 5:** Delete from front



**SET 6:** Delete from any position



**SET 7:** Display list in reverse order



**5.DISCUSSIONS**

**Variable Description**

* **start:** pointer to the first node of the linked list.
* **item:** The item to be inserted into a node.
* **pos:** to hold user input position of a node.
* **max:** The total number of nodes in the list.
* **end:** pointer to the last element of the list.
* **ptr,temp,temp2,temp3**: pointers to the nodes of the list
* **count:** loop counter

**Limitations**

* Unlike arrays, extra variables need to be stored as pointers to refer to other nodes in the list, thus more memory is required to create the data structure.
* The nodes are not indexed intrinsically and thus, loops need to be used to find the location of a node.

**Uses**

* The program can be used to perfrom various operations including insertion and deletion of nodes. For example, it can find implementation in a employee database where a list of employee data needs to be stored and the list needs to be updated with time by adding and deleting nodes at various positions throughout the list.

**Future Scope:**

* A separate field can be added to instrinsically store the index of a node for easier maintainance of very large lists.