Government College of Engineering, Karad Programming for Problem Solving Lab

Nanekar Saurabh Rajesh

20141212

I1

**Experiment No. 11**

**Title**: Implement a program to perform following operations on doubly linked list: Create, Insert – start, end, in between, search and delete, display etc.

**Outcome:** Students can perform doubly linked list operations and its related applications.

**Theory:**

**Doubly Linked List**

A doubly linked list is a type of [linked list](https://www.programiz.com/dsa/linked-list) in which each node consists of 3 components:

* \*prev - address of the previous node
* data - data item
* \*next - address of next node



A doubly linked list node

**Representation of Doubly Linked List**

Let's see how we can represent a doubly linked list on an algorithm/code. Suppose we have a doubly linked list:



Newly created doubly linked list

Here, the single node is represented as

struct node {

int data;

struct node \*next;

struct node \*prev;

}

Each struct node has a data item, a pointer to the previous struct node, and a pointer to the next struct node.

Now we will create a simple doubly linked list with three items to understand how this works.

/\* Initialize nodes \*/

struct node \*head;

struct node \*one = NULL;

struct node \*two = NULL;

struct node \*three = NULL;

/\* Allocate memory \*/

one = malloc(sizeof(struct node));

two = malloc(sizeof(struct node));

three = malloc(sizeof(struct node));

/\* Assign data values \*/

one->data = 1;

two->data = 2;

three->data = 3;

/\* Connect nodes \*/

one->next = two;

one->prev = NULL;

two->next = three;

two->prev = one;

three->next = NULL;

three->prev = two;

/\* Save address of first node in head \*/

head = one;

In the above code, one, two, and three are the nodes with data items **1**, **2**, and **3** respectively.

* **For node one**: next stores the address of two and prev stores null (there is no node before it)
* **For node two**: next stores the address of three and prev stores the address of one
* **For node three**: next stores null (there is no node after it) and prev stores the address of two.

**Note**: In the case of the head node, prev points to null, and in the case of the tail pointer, next points to null. Here, one is a head node and three is a tail node.

**Insertion on a Doubly Linked List**

Pushing a node to a doubly-linked list is similar to pushing a node to a linked list, but extra work is required to handle the pointer to the previous node.

We can insert elements at 3 different positions of a doubly-linked list:

1. [Insertion at the beginning](https://www.programiz.com/dsa/doubly-linked-list" \l "insertion-at-beginning)
2. [Insertion in-between nodes](https://www.programiz.com/dsa/doubly-linked-list" \l "insertion-in-between)
3. [Insertion at the End](https://www.programiz.com/dsa/doubly-linked-list" \l "insertion-at-end)

Suppose we have a double-linked list with elements **1**, **2**, and **3**.

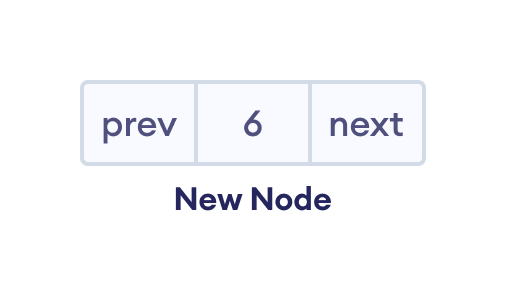
Original doubly linked list

**1. Insertion at the Beginning**

Let's add a node with value **6** at the beginning of the doubly linked list we made above.

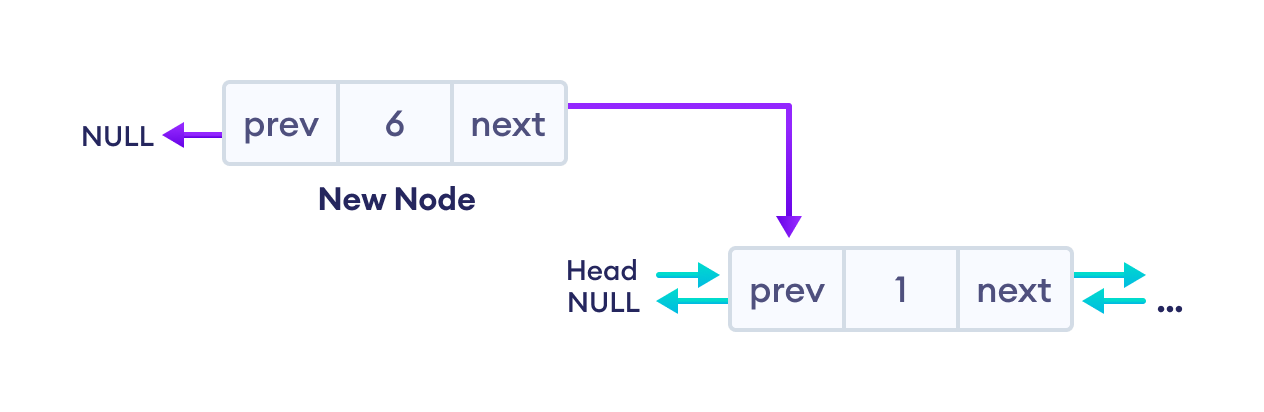
**1. Create a new node**

* allocate memory for newNode
* assign the data to newNode.

New node

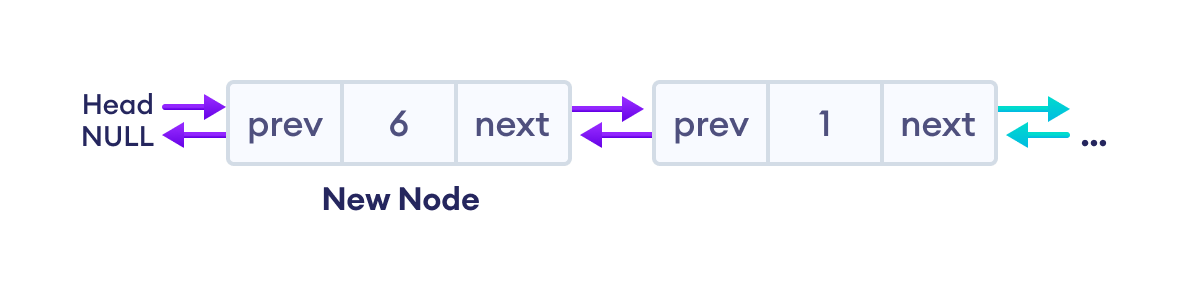
**2. Set prev and next pointers of new node**

* point next of newNode to the first node of the doubly linked list
* point prev to null

Reorganize the pointers (changes are denoted by purple arrows)

**3. Make new node as head node**

* Point prev of the first node to newNode (now the previous head is the second node)
* Point head to newNode

Reorganize the pointers

**Code for Insertion at the Beginning**

// insert node at the front

void insertFront(struct Node\*\* head, int data) {

// allocate memory for newNode

struct Node\* newNode = new Node;

// assign data to newNode

newNode->data = data;

// point next of newNode to the first node of the doubly linked list

newNode->next = (\*head);

// point prev to NULL

newNode->prev = NULL;

// point previous of the first node (now first node is the second node) to newNode

if ((\*head) != NULL)

(\*head)->prev = newNode;

// head points to newNode

(\*head) = newNode;

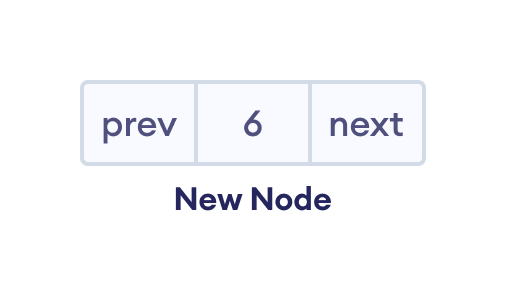
}

**2. Insertion in between two nodes**

Let's add a node with value 6 after node with value 1 in the doubly linked list.

**1. Create a new node**

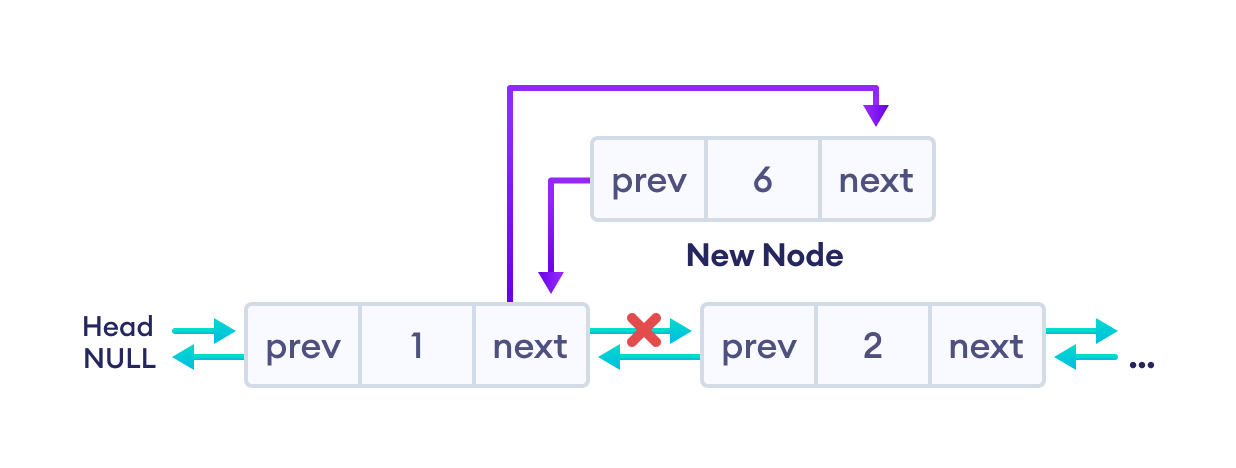
* allocate memory for newNode
* assign the data to newNode.



New node

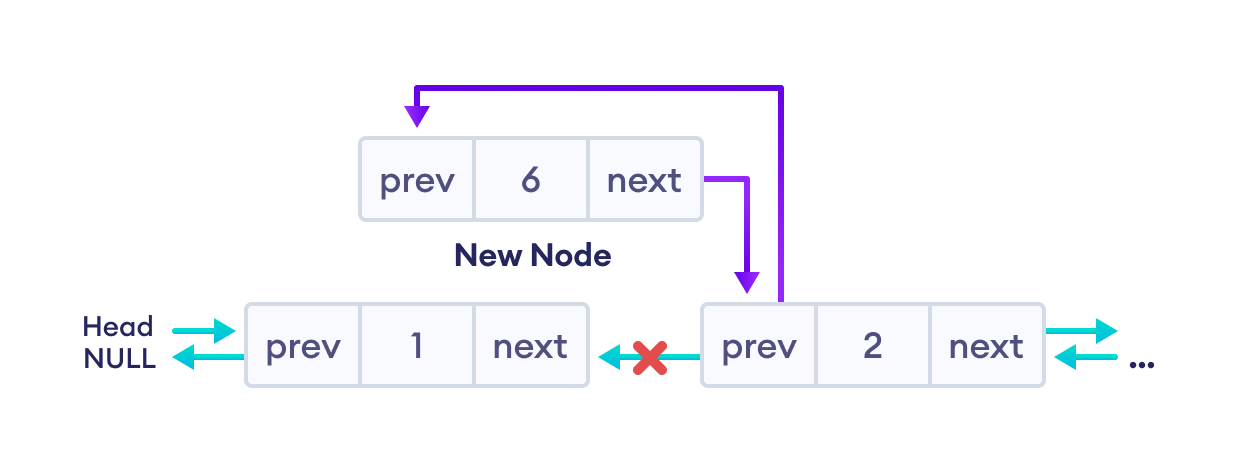
**2. Set the next pointer of new node and previous node**

* assign the value of next from previous node to the next of newNode
* assign the address of newNode to the next of previous node

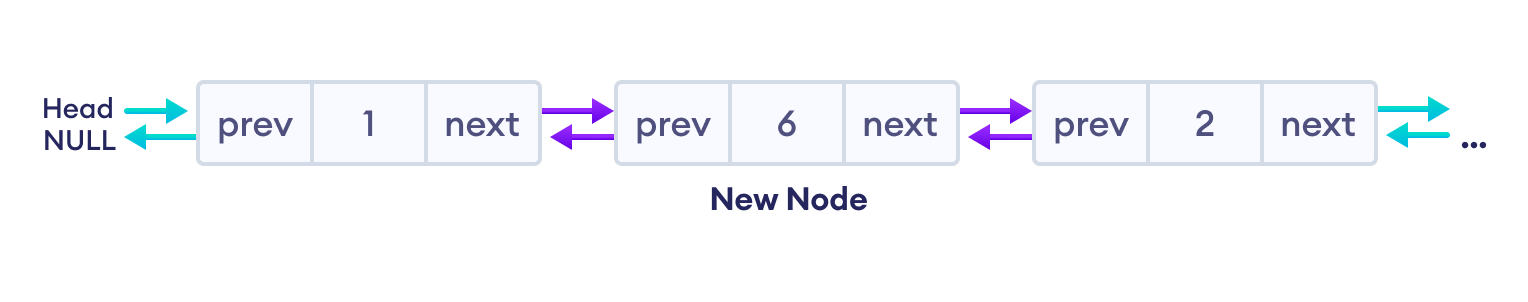
Reorganize the pointers

**3. Set the prev pointer of new node and the next node**

* assign the value of prev of next node to the prev of newNode
* assign the address of newNode to the prev of next node

Reorganize the pointers

The final doubly linked list is after this insertion is:

Final list

**Code for Insertion in between two Nodes**

// insert a node after a specific node

void insertAfter(struct Node\* prev\_node, int data) {

// check if previous node is NULL

if (prev\_node == NULL) {

cout << "previous node cannot be NULL";

return;

}

// allocate memory for newNode

struct Node\* newNode = new Node;

// assign data to newNode

newNode->data = data;

// set next of newNode to next of prev node

newNode->next = prev\_node->next;

// set next of prev node to newNode

prev\_node->next = newNode;

// set prev of newNode to the previous node

newNode->prev = prev\_node;

// set prev of newNode's next to newNode

if (newNode->next != NULL)

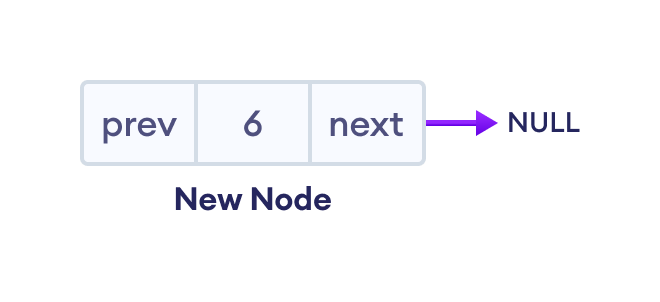
newNode->next->prev = newNode;

}

**3. Insertion at the End**

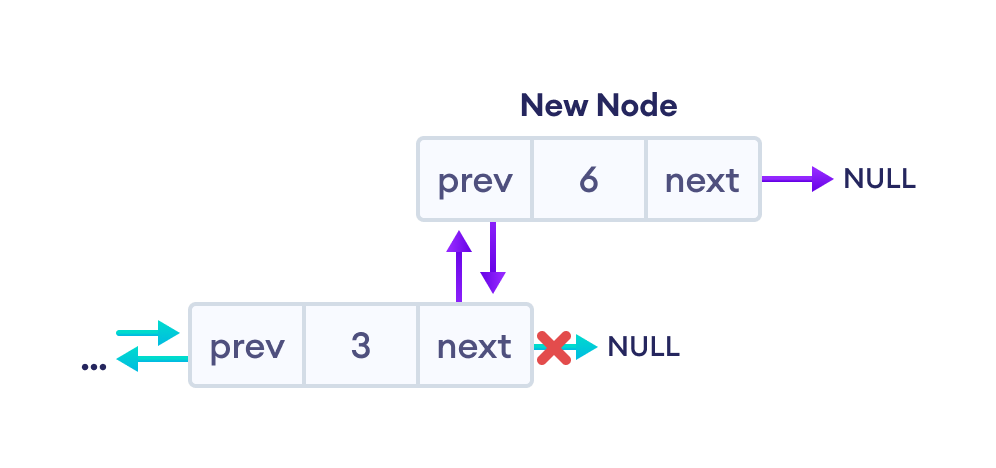
Let's add a node with value 6 at the end of the doubly linked list.

**1. Create a new node**

New node

**2. Set prev and next pointers of new node and the previous node**

If the linked list is empty, make the newNode as the head node. Otherwise, traverse to the end of the doubly linked list and

Reorganize the pointers

The final doubly linked list looks like this.

The final list

**Code for Insertion at the End**

// insert a newNode at the end of the list

void insertEnd(struct Node\*\* head, int data) {

// allocate memory for node

struct Node\* newNode = new Node;

// assign data to newNode

newNode->data = data;

// assign NULL to next of newNode

newNode->next = NULL;

// store the head node temporarily (for later use)

struct Node\* temp = \*head;

// if the linked list is empty, make the newNode as head node

if (\*head == NULL) {

newNode->prev = NULL;

\*head = newNode;

return;

}

// if the linked list is not empty, traverse to the end of the linked list

while (temp->next != NULL)

temp = temp->next;

// now, the last node of the linked list is temp

// point the next of the last node (temp) to newNode.

temp->next = newNode;

// assign prev of newNode to temp

newNode->prev = temp;

}

**Deletion from a Doubly Linked List**

Similar to insertion, we can also delete a node from **3** different positions of a doubly linked list.

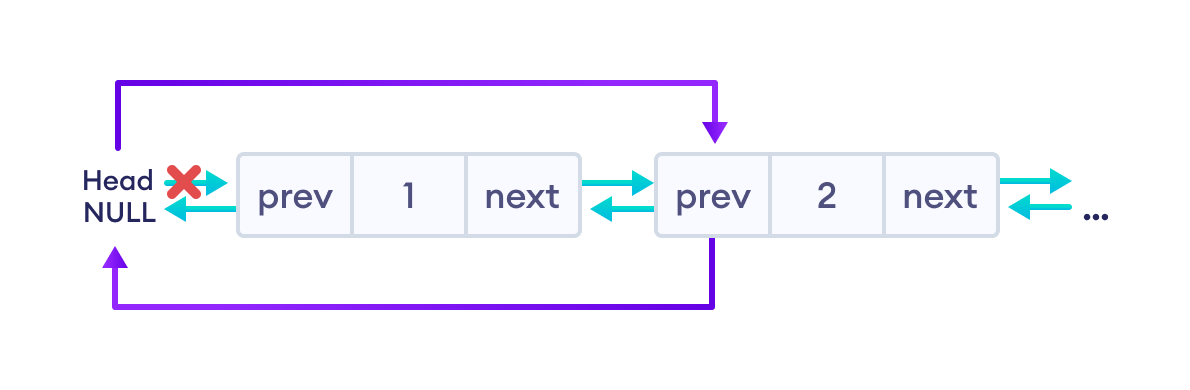
Suppose we have a double-linked list with elements **1**, **2**, and **3**.

Original doubly linked list

**1. Delete the First Node of Doubly Linked List**

If the node to be deleted (i.e. del\_node) is at the beginning

**Reset value node after the del\_node (i.e. node two)**

Reorganize the pointers

Finally, free the memory of del\_node. And, the linked will look like this

Final list

**Code for Deletion of the First Node**

if (\*head == del\_node)

\*head = del\_node->next;

if (del\_node->prev != NULL)

del\_node->prev->next = del\_node->next;

free(del);

**2. Deletion of the Inner Node**

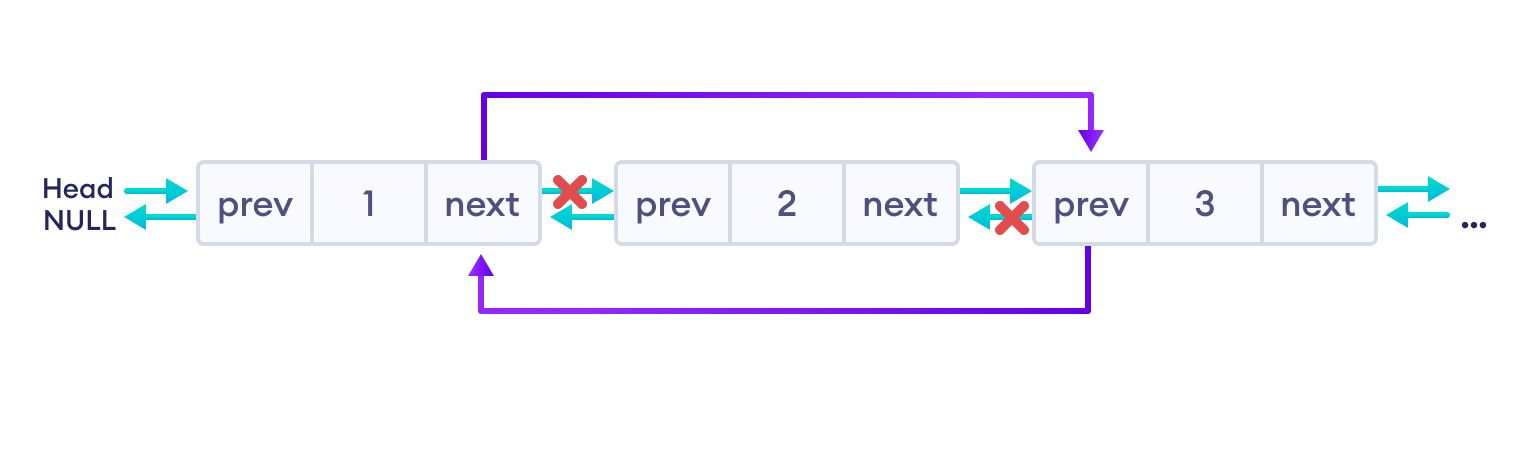
If del\_node is an inner node (second node), we must have to reset the value of next and prev of the nodes before and after the del\_node.

**For the node before the del\_node (i.e. first node)**

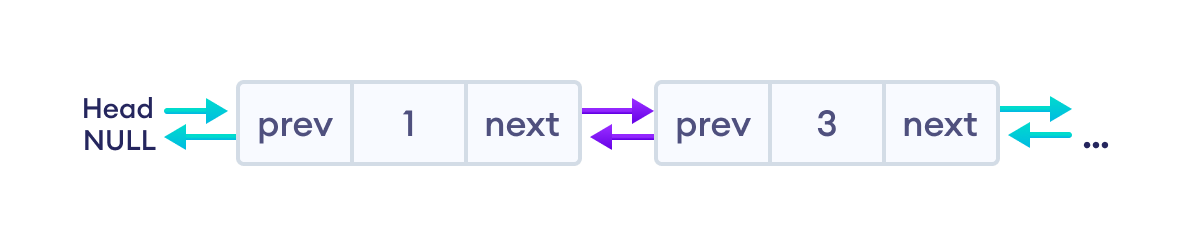
Assign the value of next of del\_node to the next of the first node.

**For the node after the del\_node (i.e. third node)**

Assign the value of prev of del\_node to the prev of the third node.

Reorganize the pointers

Finally, we will free the memory of del\_node. And, the final doubly linked list looks like this.

Final list

**Code for Deletion of the Inner Node**

if (del\_node->next != NULL)

del\_node->next->prev = del\_node->prev;

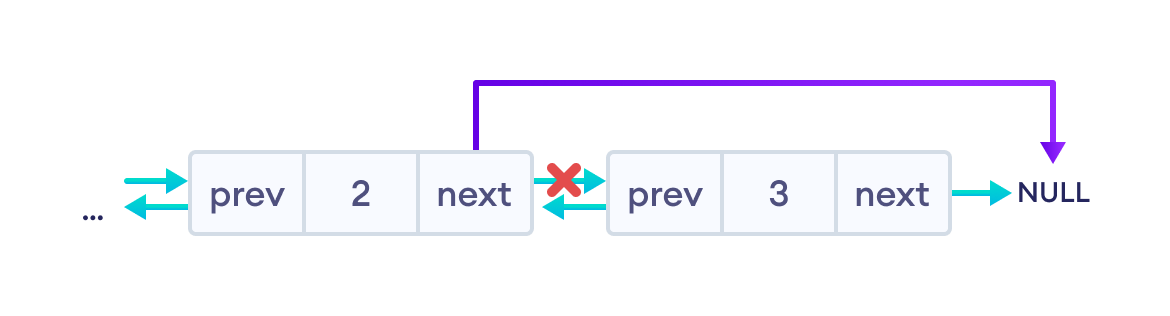
if (del\_node->prev != NULL)

del\_node->prev->next = del\_node->next;

**3. Delete the Last Node of Doubly Linked List**

In this case, we are deleting the last node with value **3** of the doubly linked list.

Here, we can simply delete the del\_node and make the next of node before del\_node point to NULL.

Reorganize the pointers

The final doubly linked list looks like this.

Final list

**Code for Deletion of the Last Node**

if (del\_node->prev != NULL)

del\_node->prev->next = del\_node->next;

Here, del\_node ->next is NULL so del\_node->prev->next = NULL.

**Note**: We can also solve this using the first condition (for the node before del\_node) of the second case (Delete the inner node).

**Doubly Linked List Complexity**

|  |  |  |
| --- | --- | --- |
| Doubly Linked List Complexity | **Time Complexity** | **Space Complexity** |
| **Insertion Operation** | O(1) or O(n) | O(1) |
| **Deletion Operation** | O(1) | O(1) |

1. **Complexity of Insertion Operation**

* The insertion operations that do not require traversal have the time complexity of O(1).
* And, insertion that requires traversal has time complexity of O(n).
* The space complexity is O(1).

**2. Complexity of Deletion Operation**

* All deletion operations run with time complexity of O(1).
* And, the space complexity is O(1).

**Doubly Linked List Applications**

1. Redo and undo functionality in software.
2. Forward and backward navigation in browsers.
3. For navigation systems where forward and backward navigation is required.

**Analysis:**



**List of similar programs: Solve any one.**

1. Write a C program to [Reverse a Doubly Linked List](https://www.geeksforgeeks.org/reverse-a-doubly-linked-list/)
2. Write a C program to [Find pairs with given sum in doubly linked list](https://www.geeksforgeeks.org/find-pairs-given-sum-doubly-linked-list/)
3. Write a C program to [Rotate Doubly linked list by N nodes](https://www.geeksforgeeks.org/rotate-doubly-linked-list-n-nodes/)

Refer <https://www.geeksforgeeks.org/data-structures/linked-list/doubly-linked-list/> for more practice.

**Title Program:**

Implement a program to perform following operations on doubly linked list: Create, Insert – start, end, in between, search and delete, display etc.

**Source code of Implemented Programs:**

//Nanekar Saurabh Rajesh

#include<stdio.h>

#include<stdlib.h>

struct node

{

    struct node \*prev;

    struct node \*next;

    int data;

};

struct node \*head;

void insertion\_beginning();

void insertion\_last();

void insertion\_specified();

void deletion\_beginning();

void deletion\_last();

void deletion\_specified();

void display();

void search();

void main ()

{

int choice =0;

    while(choice != 9)

    {

        printf("\n\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\n");

        printf("\nChoose one option from the following list ...\n");

        printf("\n===============================================\n");

        printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n 5.Delete from last\n6.Delete the node after the given data\n7.Search\n8.Show\n9.Exit\n");

        printf("\nEnter your choice?\n");

        scanf("\n%d",&choice);

        switch(choice)

        {

            case 1:

            insertion\_beginning();

            break;

            case 2:

                    insertion\_last();

            break;

            case 3:

            insertion\_specified();

            break;

            case 4:

            deletion\_beginning();

            break;

            case 5:

            deletion\_last();

            break;

            case 6:

            deletion\_specified();

            break;

            case 7:

            search();

            break;

            case 8:

            display();

            break;

            case 9:

            exit(0);

            break;

            default:

            printf("Please enter valid choice..");

        }

    }

}

void insertion\_beginning()

{

   struct node \*ptr;

   int item;

   ptr = (struct node \*)malloc(sizeof(struct node));

   if(ptr == NULL)

   {

       printf("\nOVERFLOW");

   }

   else

   {

    printf("\nEnter Item value");

    scanf("%d",&item);

   if(head==NULL)

   {

       ptr->next = NULL;

       ptr->prev=NULL;

       ptr->data=item;

       head=ptr;

   }

   else

   {

       ptr->data=item;

       ptr->prev=NULL;

       ptr->next = head;

       head->prev=ptr;

       head=ptr;

   }

   printf("\nNode inserted\n");

}

}

void insertion\_last()

{

   struct node \*ptr,\*temp;

   int item;

   ptr = (struct node \*) malloc(sizeof(struct node));

   if(ptr == NULL)

   {

       printf("\nOVERFLOW");

   }

   else

   {

       printf("\nEnter value");

       scanf("%d",&item);

        ptr->data=item;

       if(head == NULL)

       {

           ptr->next = NULL;

           ptr->prev = NULL;

           head = ptr;

       }

       else

       {

          temp = head;

          while(temp->next!=NULL)

          {

              temp = temp->next;

          }

          temp->next = ptr;

          ptr ->prev=temp;

          ptr->next = NULL;

          }

       }

     printf("\nnode inserted\n");

    }

void insertion\_specified()

{

   struct node \*ptr,\*temp;

   int item,loc,i;

   ptr = (struct node \*)malloc(sizeof(struct node));

   if(ptr == NULL)

   {

       printf("\n OVERFLOW");

   }

   else

   {

       temp=head;

       printf("Enter the location");

       scanf("%d",&loc);

       for(i=0;i<loc;i++)

       {

           temp = temp->next;

           if(temp == NULL)

           {

               printf("\n There are less than %d elements", loc);

               return;

           }

       }

       printf("Enter value");

       scanf("%d",&item);

       ptr->data = item;

       ptr->next = temp->next;

       ptr -> prev = temp;

       temp->next = ptr;

       temp->next->prev=ptr;

       printf("\nnode inserted\n");

   }

}

void deletion\_beginning()

{

    struct node \*ptr;

    if(head == NULL)

    {

        printf("\n UNDERFLOW");

    }

    else if(head->next == NULL)

    {

        head = NULL;

        free(head);

        printf("\nnode deleted\n");

    }

    else

    {

        ptr = head;

        head = head -> next;

        head -> prev = NULL;

        free(ptr);

        printf("\nnode deleted\n");

    }

}

void deletion\_last()

{

    struct node \*ptr;

    if(head == NULL)

    {

        printf("\n UNDERFLOW");

    }

    else if(head->next == NULL)

    {

        head = NULL;

        free(head);

        printf("\nnode deleted\n");

    }

    else

    {

        ptr = head;

        if(ptr->next != NULL)

        {

            ptr = ptr -> next;

        }

        ptr -> prev -> next = NULL;

        free(ptr);

        printf("\nnode deleted\n");

    }

}

void deletion\_specified()

{

    struct node \*ptr, \*temp;

    int val;

    printf("\n Enter the data after which the node is to be deleted : ");

    scanf("%d", &val);

    ptr = head;

    while(ptr -> data != val)

    ptr = ptr -> next;

    if(ptr -> next == NULL)

    {

        printf("\nCan't delete\n");

    }

    else if(ptr -> next -> next == NULL)

    {

        ptr ->next = NULL;

    }

    else

    {

        temp = ptr -> next;

        ptr -> next = temp -> next;

        temp -> next -> prev = ptr;

        free(temp);

        printf("\nnode deleted\n");

    }

}

void display()

{

    struct node \*ptr;

    printf("\n printing values...\n");

    ptr = head;

    while(ptr != NULL)

    {

        printf("%d\n",ptr->data);

        ptr=ptr->next;

    }

}

void search()

{

    struct node \*ptr;

    int item,i=0,flag;

    ptr = head;

    if(ptr == NULL)

    {

        printf("\nEmpty List\n");

    }

    else

    {

        printf("\nEnter item which you want to search?\n");

        scanf("%d",&item);

        while (ptr!=NULL)

        {

            if(ptr->data == item)

            {

                printf("\nitem found at location %d ",i+1);

                flag=0;

                break;

            }

            else

            {

                flag=1;

            }

            i++;

            ptr = ptr -> next;

        }

        if(flag==1)

        {

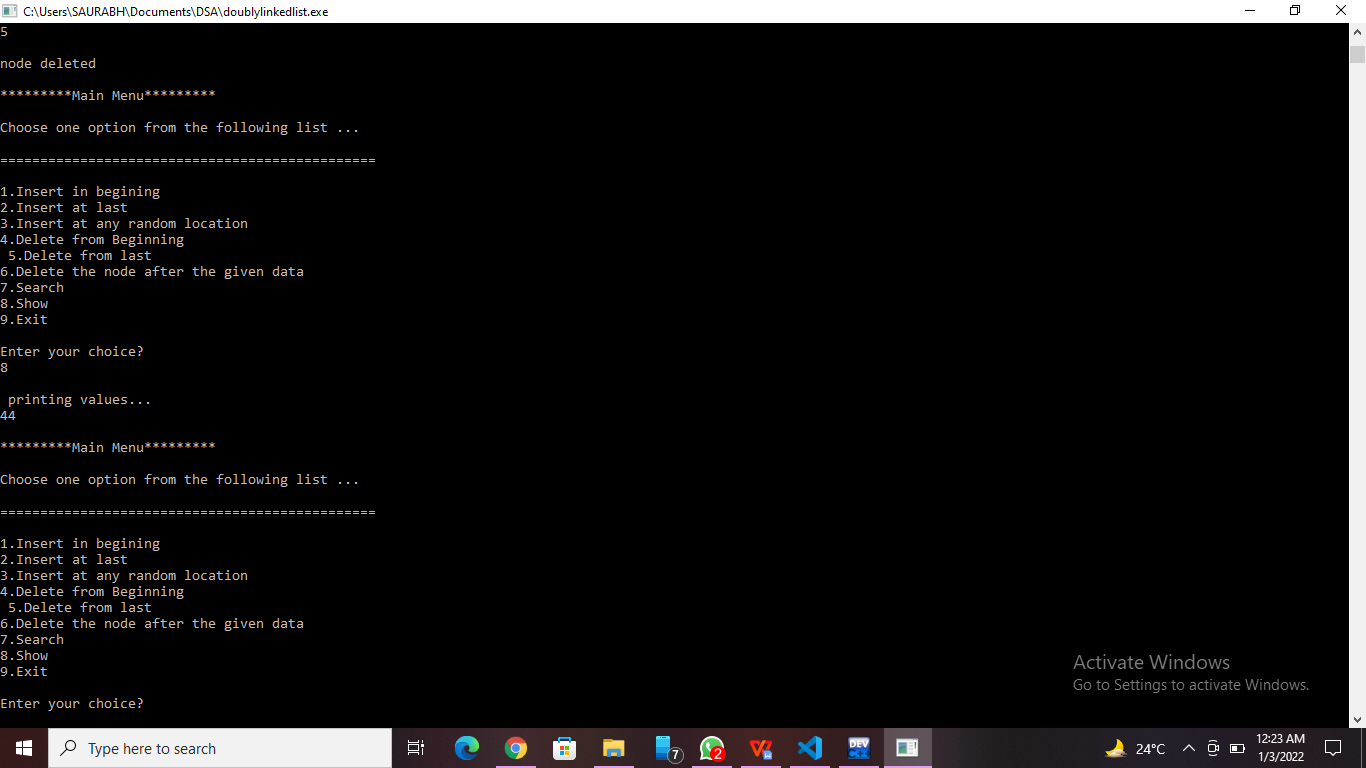
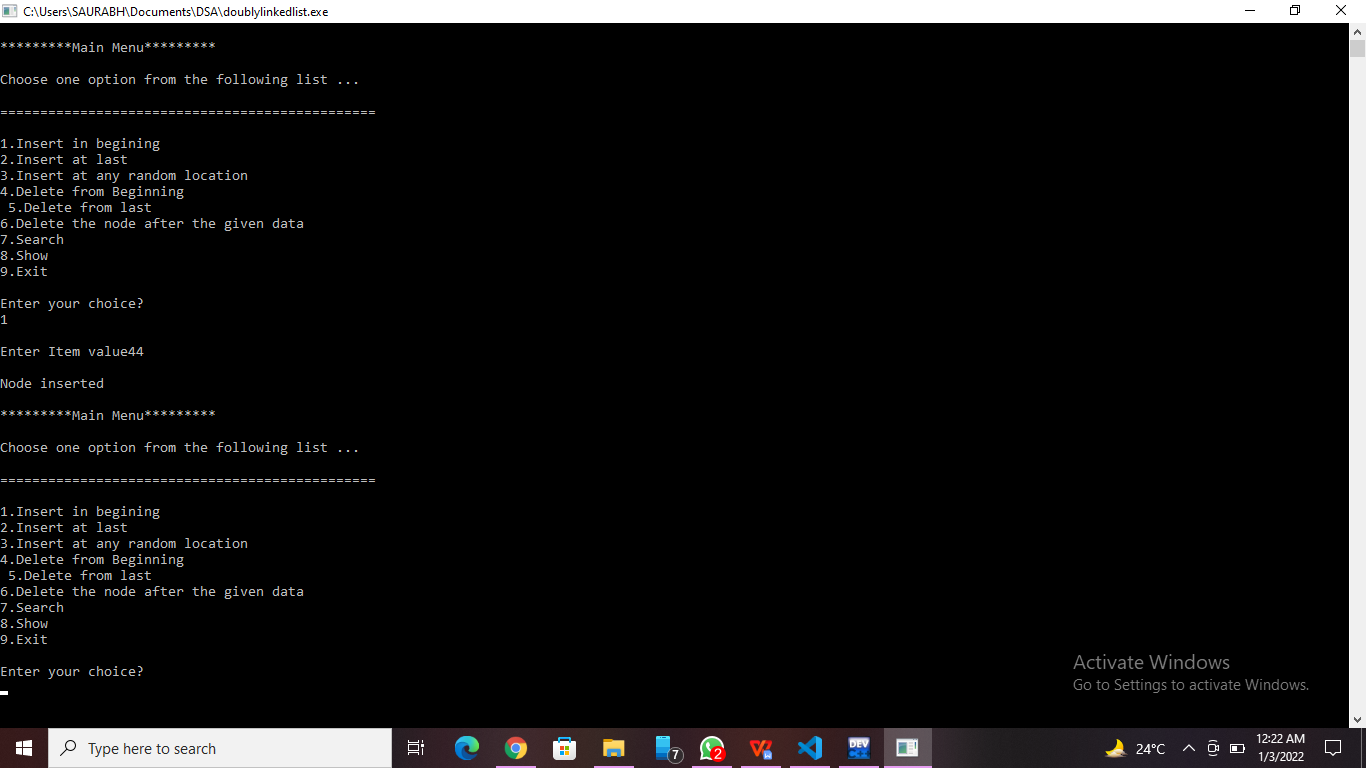
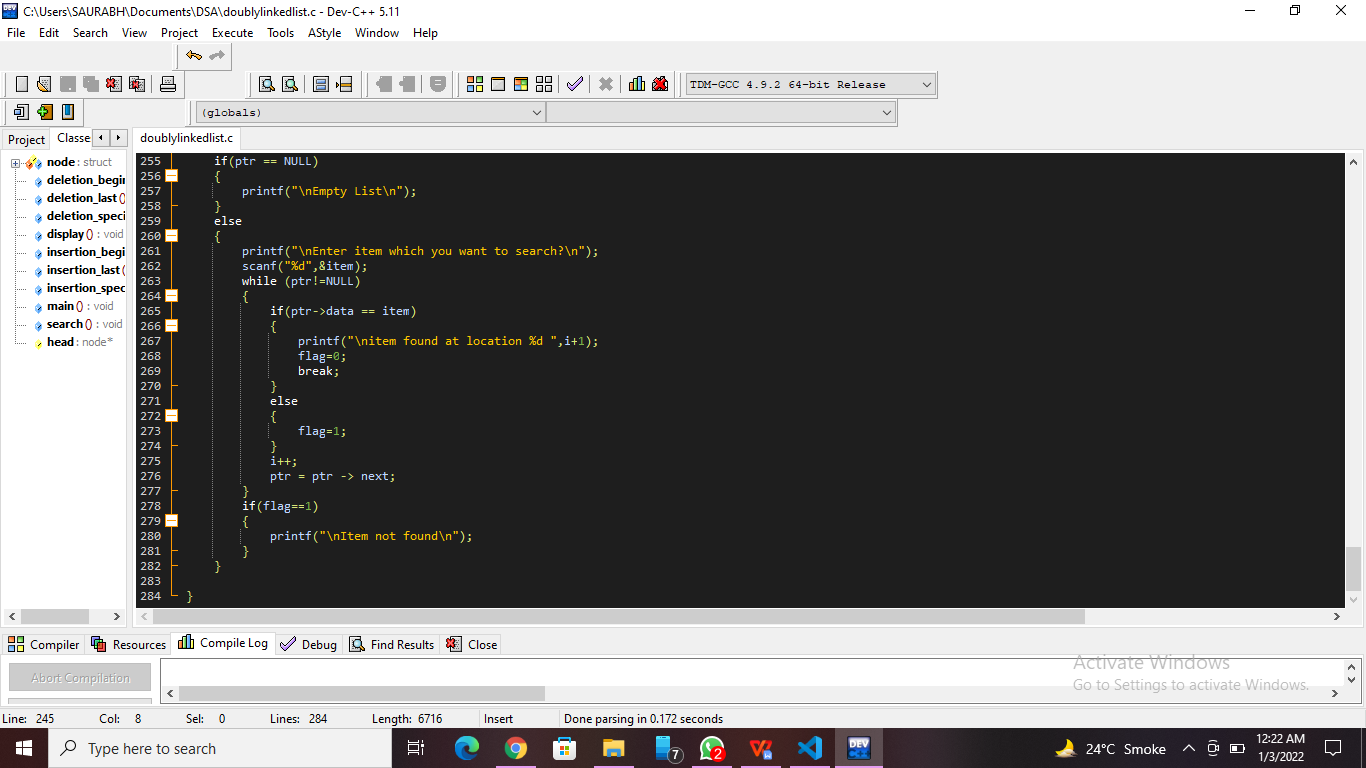
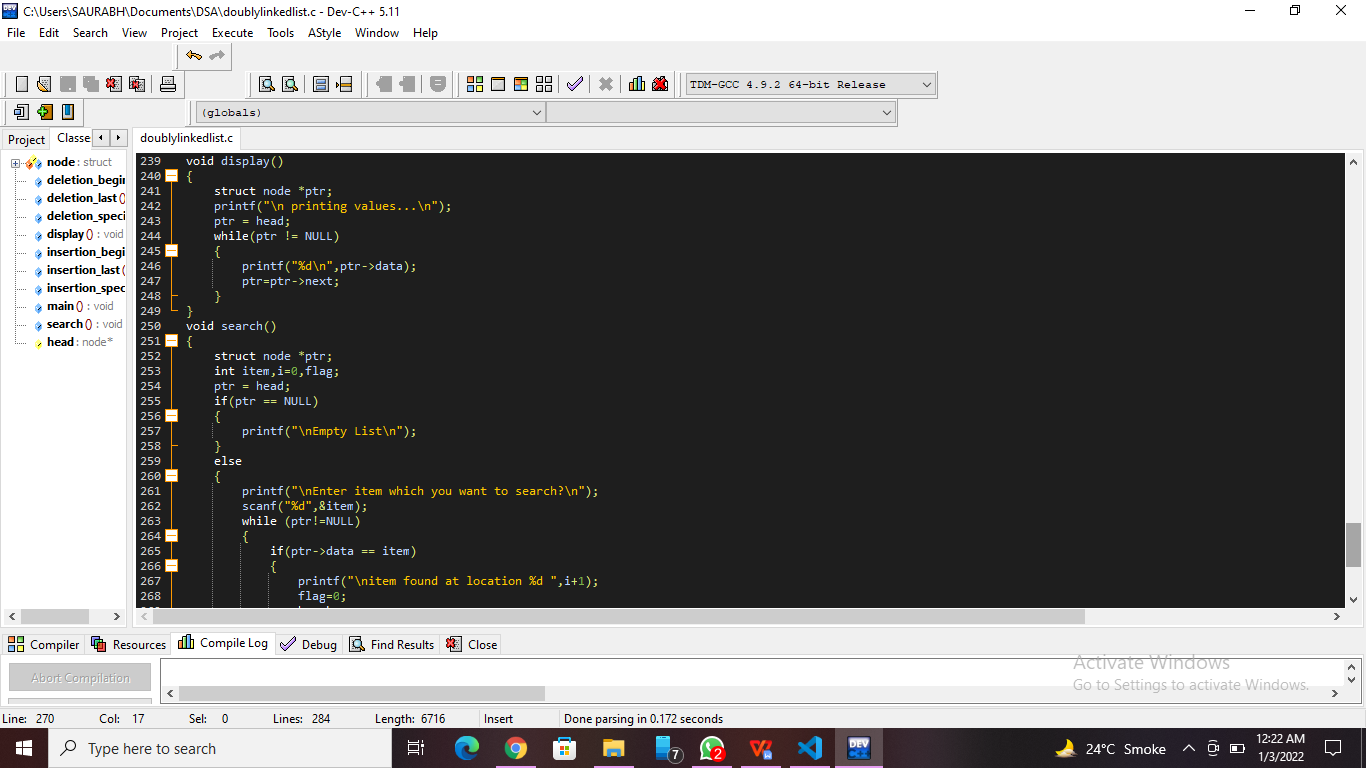
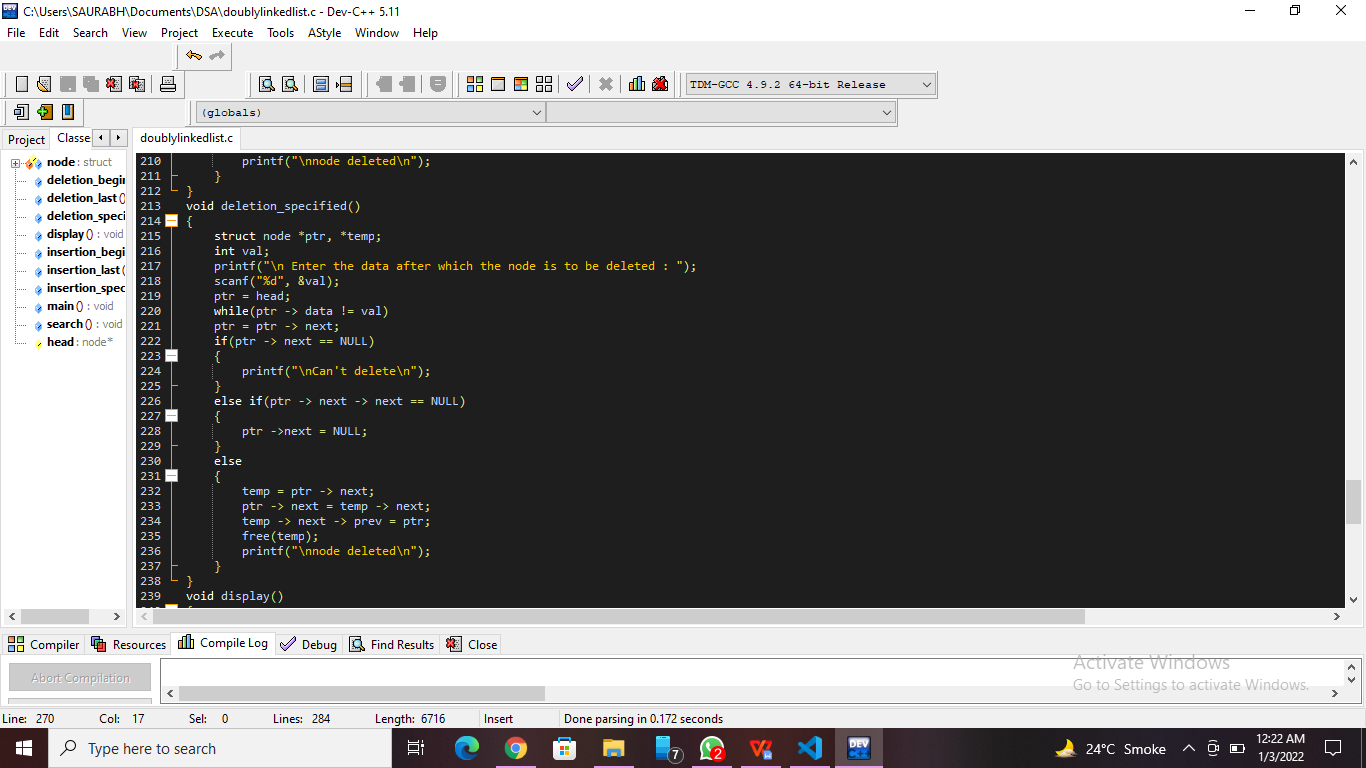
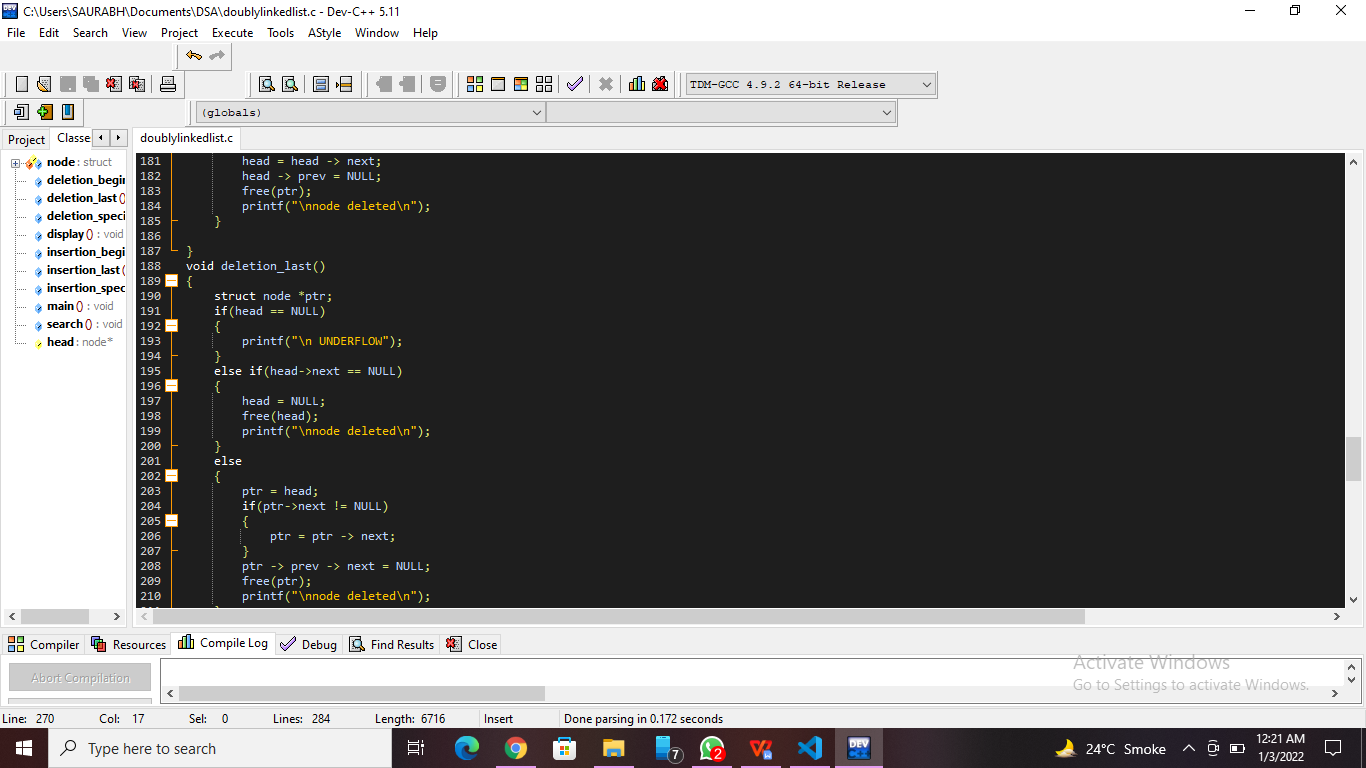
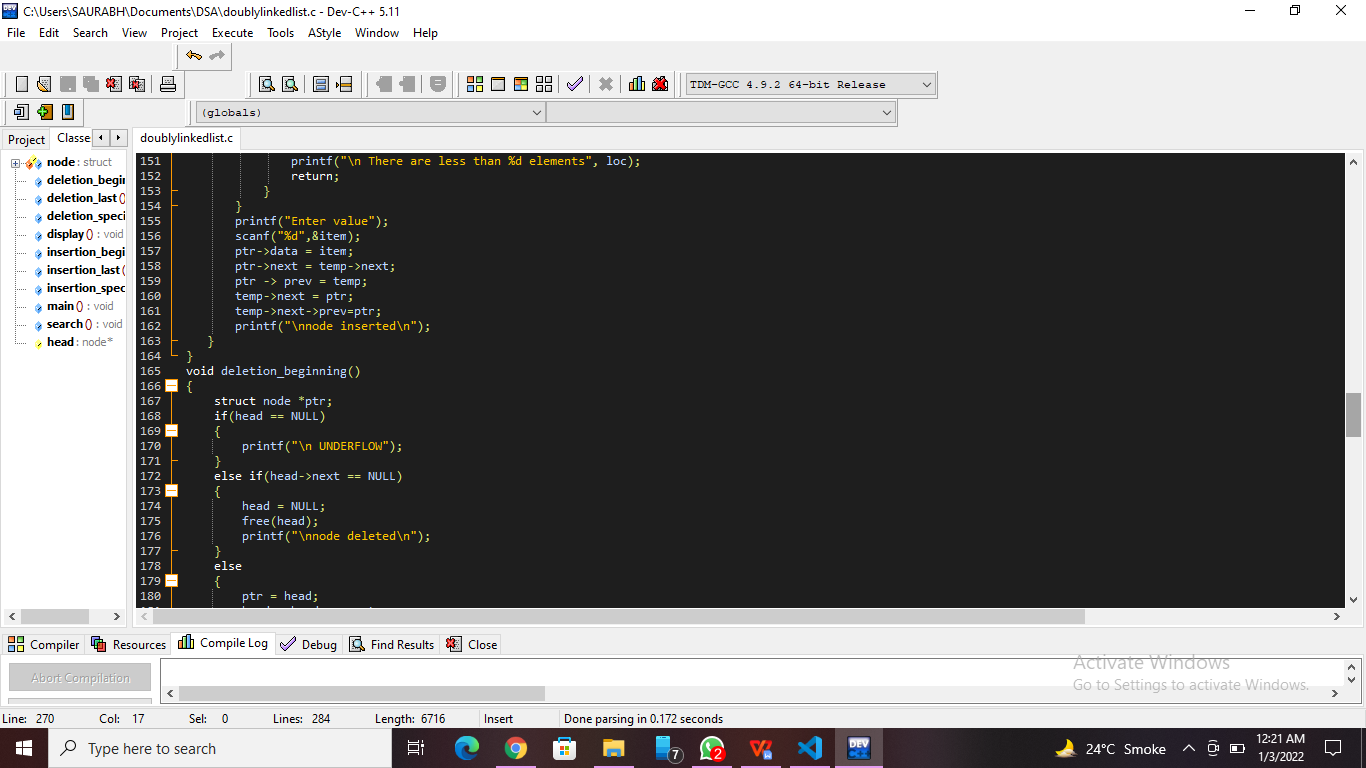
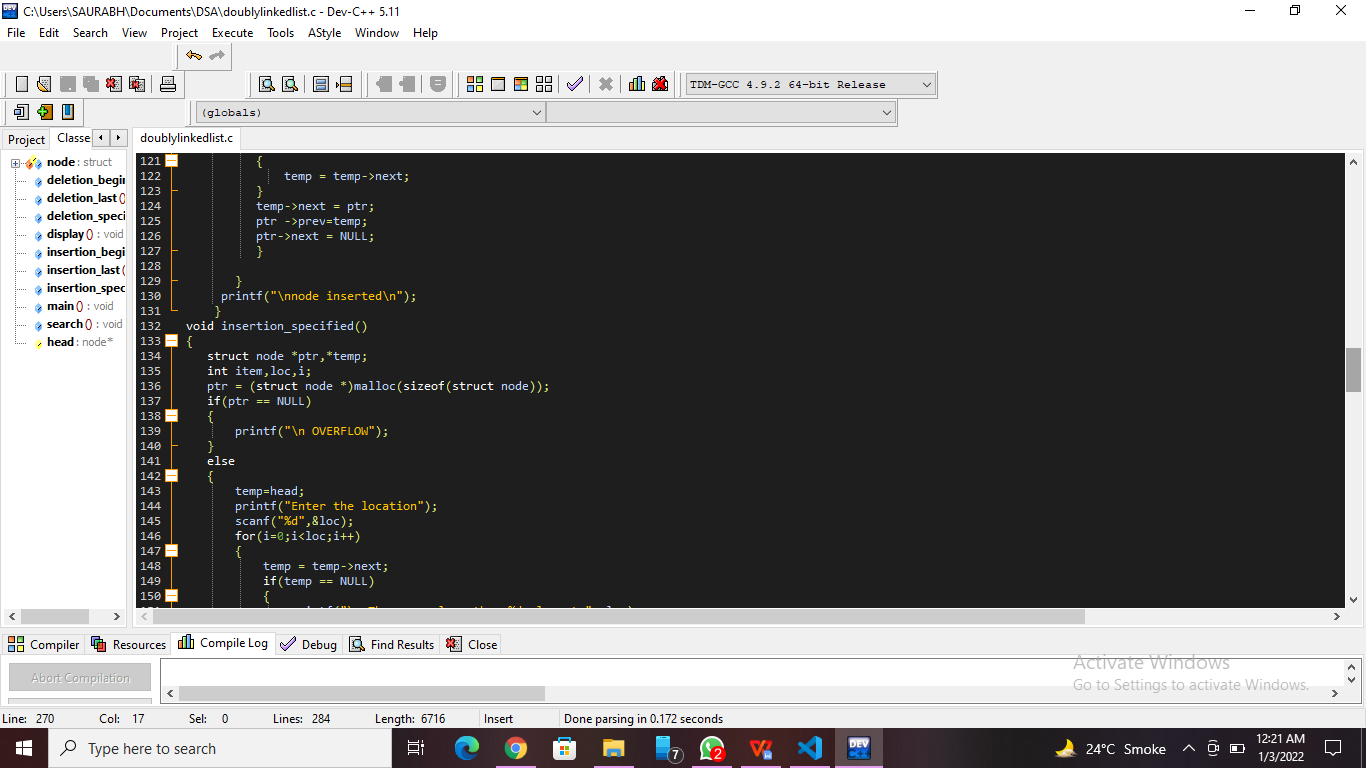
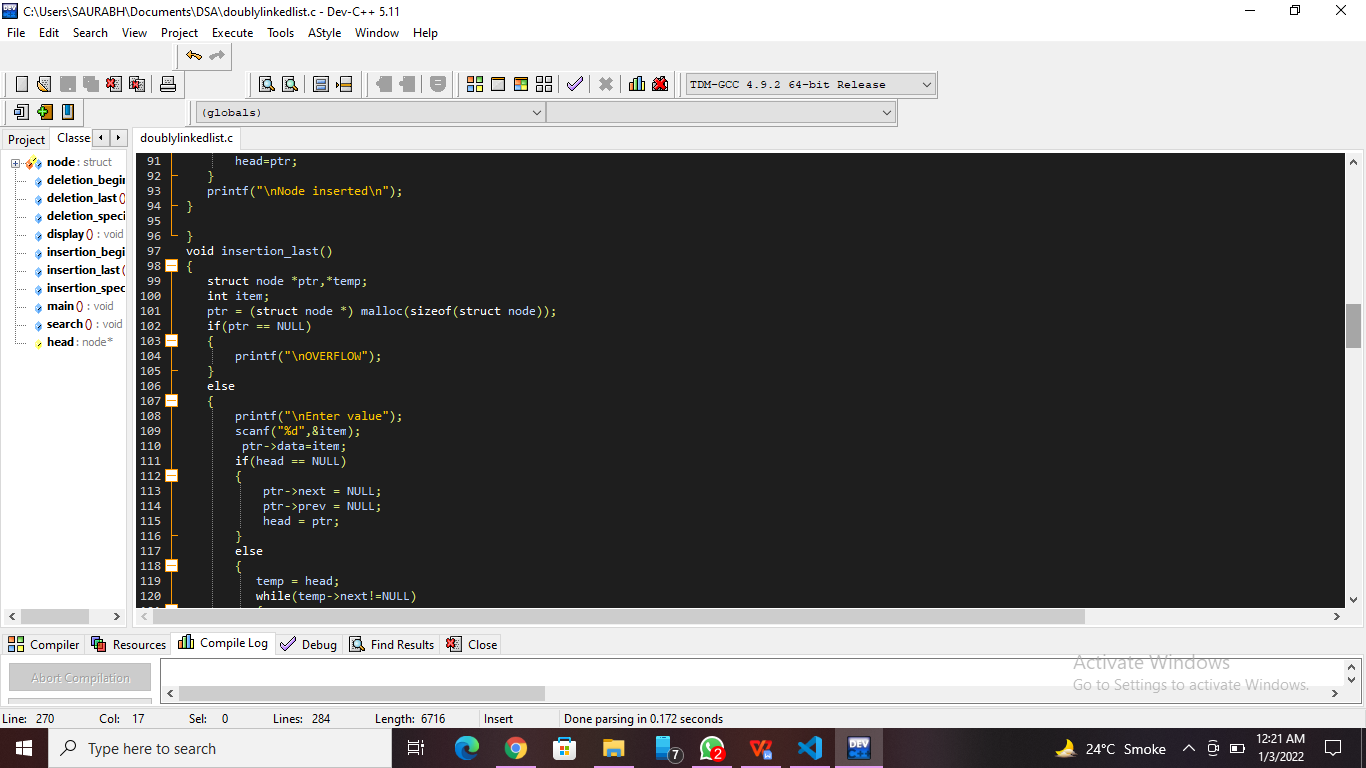
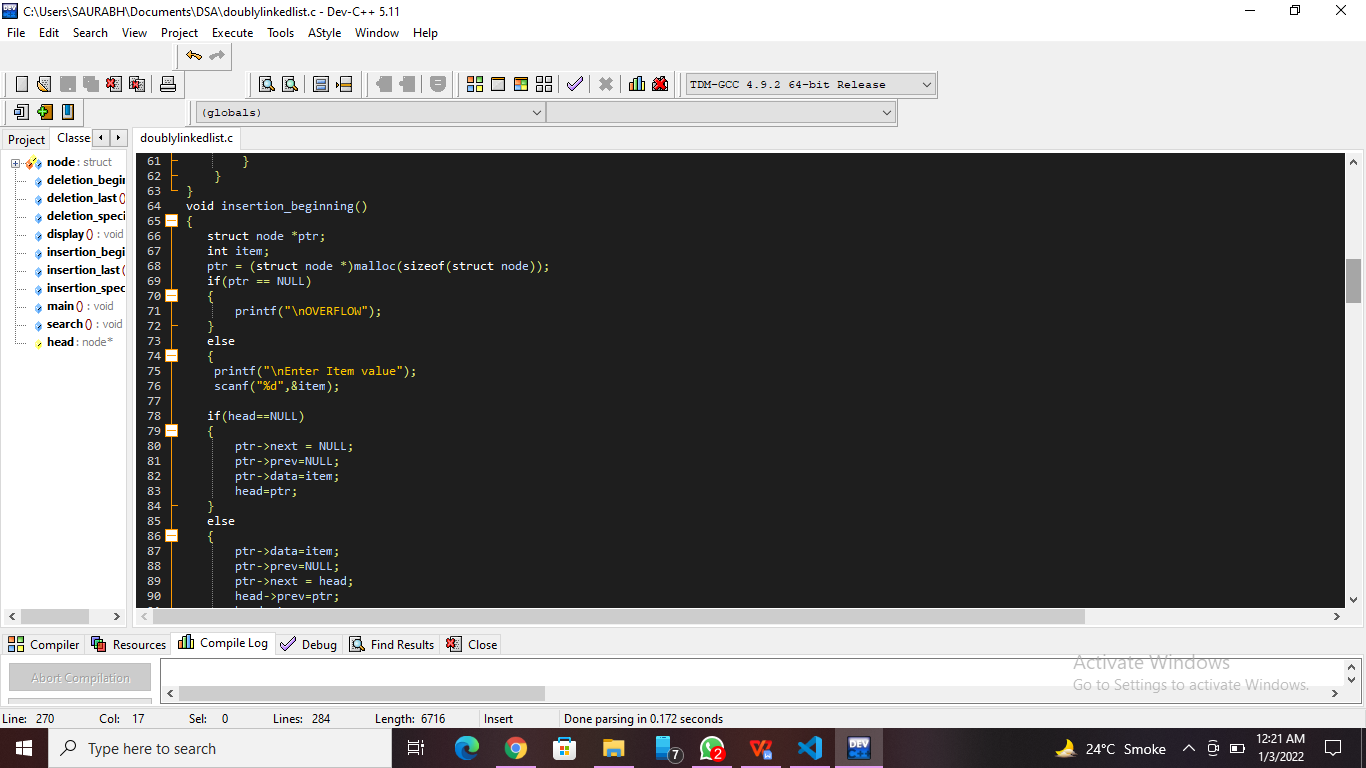
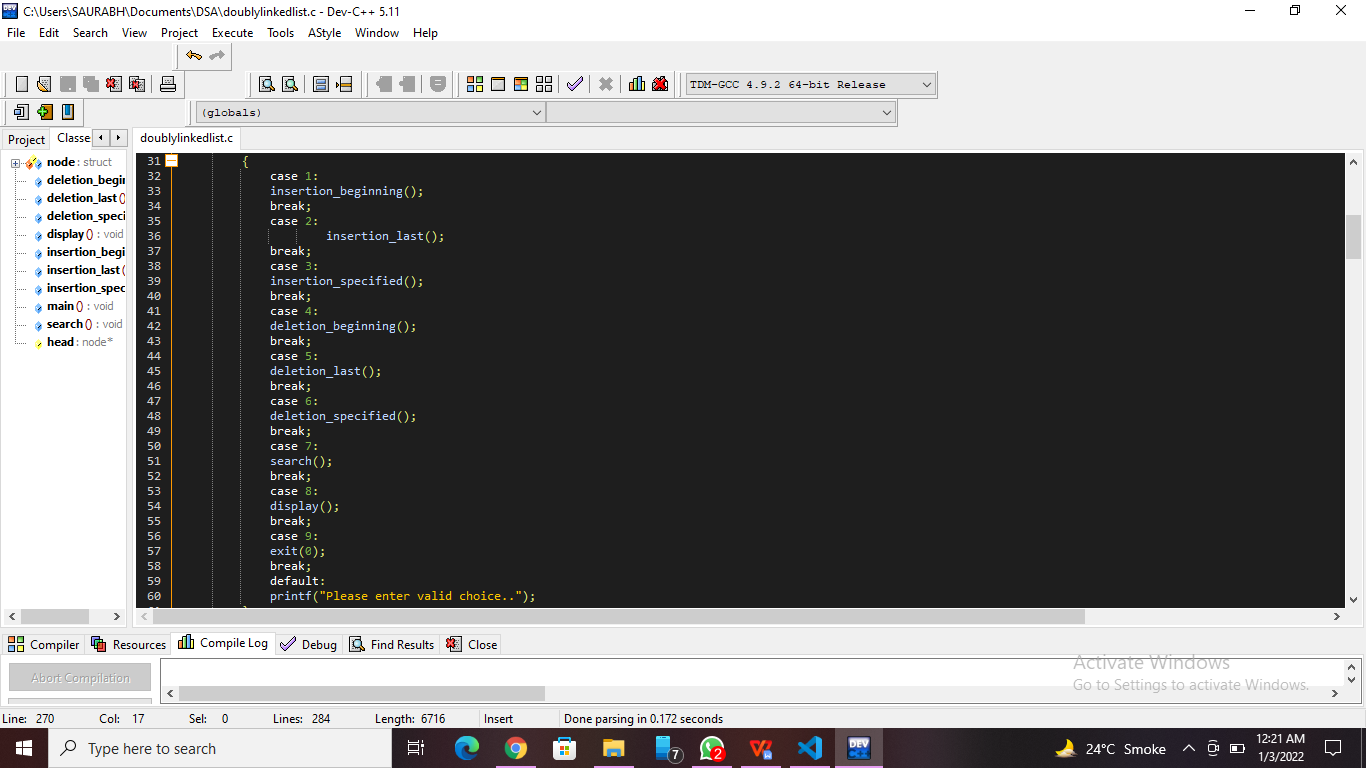
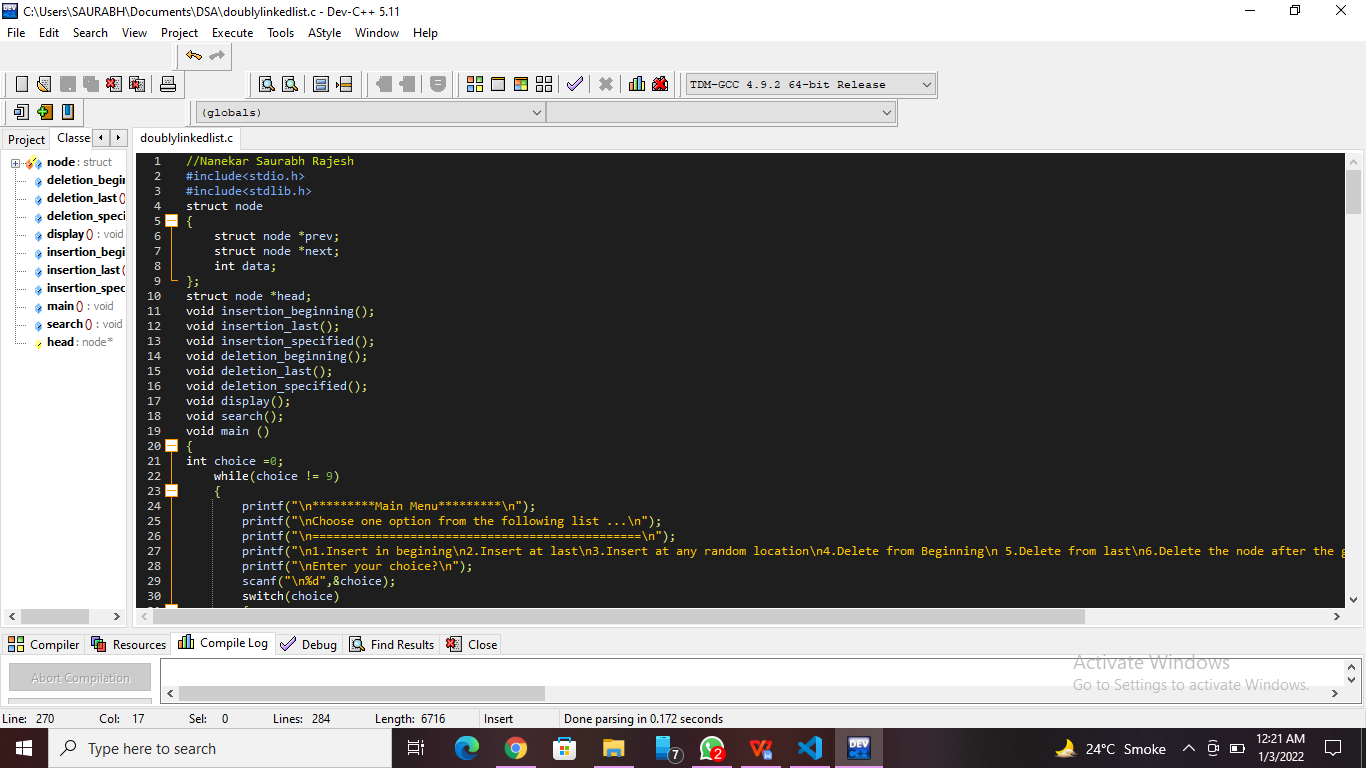
            printf("\nItem not found\n");

        }

    }

}

**Screenshots of Output:**

****

**Practice Program:** Write a C program to [Rotate Doubly linked list by N nodes](https://www.geeksforgeeks.org/rotate-doubly-linked-list-n-nodes/).

//Nanekar Saurabh Rajesh\_20141212\_I1

#include <stdio.h>

#include<stdlib.h>

struct node

{

    int data;

    struct node \*previous;

    struct node \*next;

};

int size = 0;

struct node \*head, \*tail = NULL;

void addNode(int data)

{

    struct node \*newNode = (struct node\*)malloc(sizeof(struct node));

    newNode->data = data;

    if(head == NULL)

    {

        head = tail = newNode;

        head->previous = NULL;

        tail->next = NULL;

    }

    else

   {

        tail->next = newNode;

        newNode->previous = tail;

        tail = newNode;

        tail->next = NULL;

    }

    size++;

}

void rotateList(int n)

{

    struct node \*current = head;

    if(n == 0 || n >= size)

        return;

    else

     {

        int i;

        for( i = 1; i < n; i++)

        current = current->next;

        tail->next = head;

        head = current->next;

        head->previous = NULL;

        tail = current;

        tail->next = NULL;

    }

}

void display()

 {

    struct node \*current = head;

    if(head == NULL) {

        printf("List is empty\n");

        return;

    }

    while(current != NULL)

    {

        printf("%d ", current->data);

        current = current->next;

    }

    printf("\n");

}

int main()

{

    addNode(1);

    addNode(2);

    addNode(3);

    addNode(4);

    addNode(5);

    printf("Original List: \n");

    display();

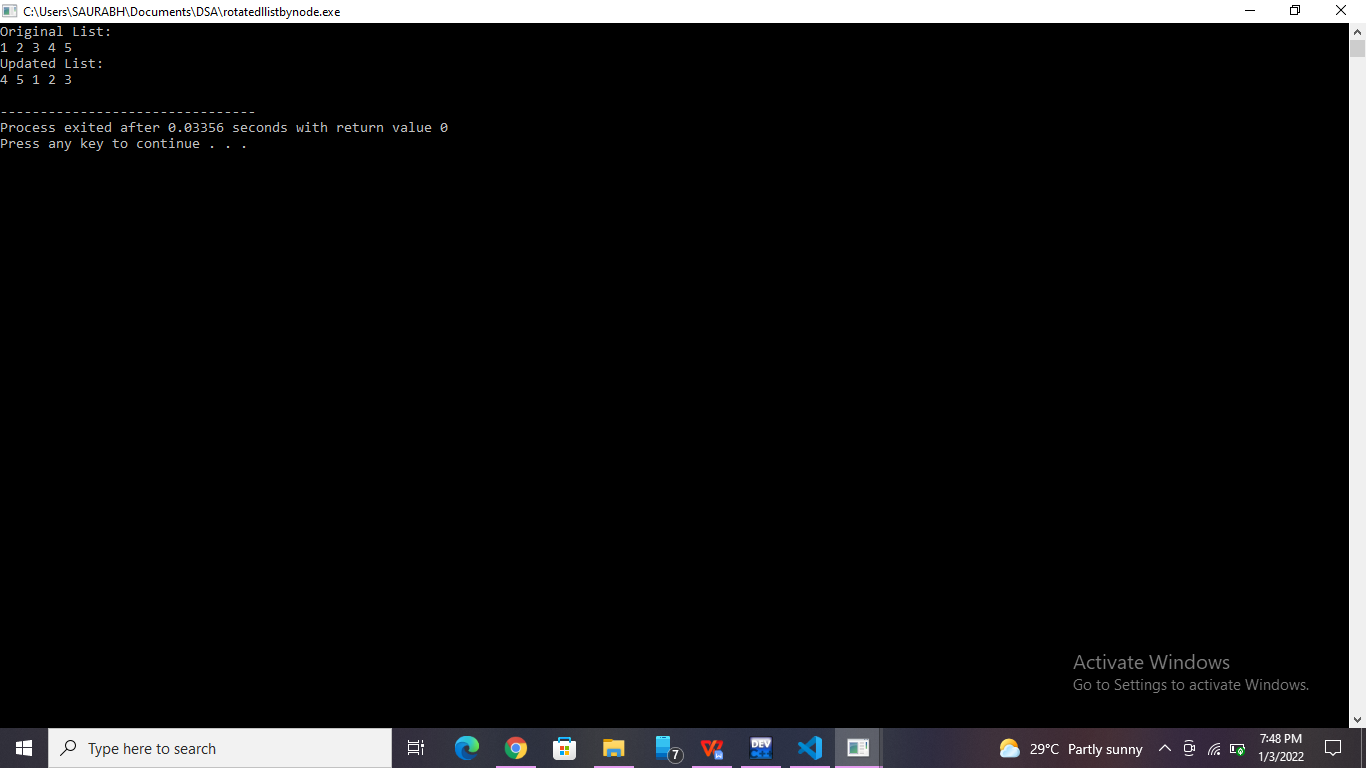
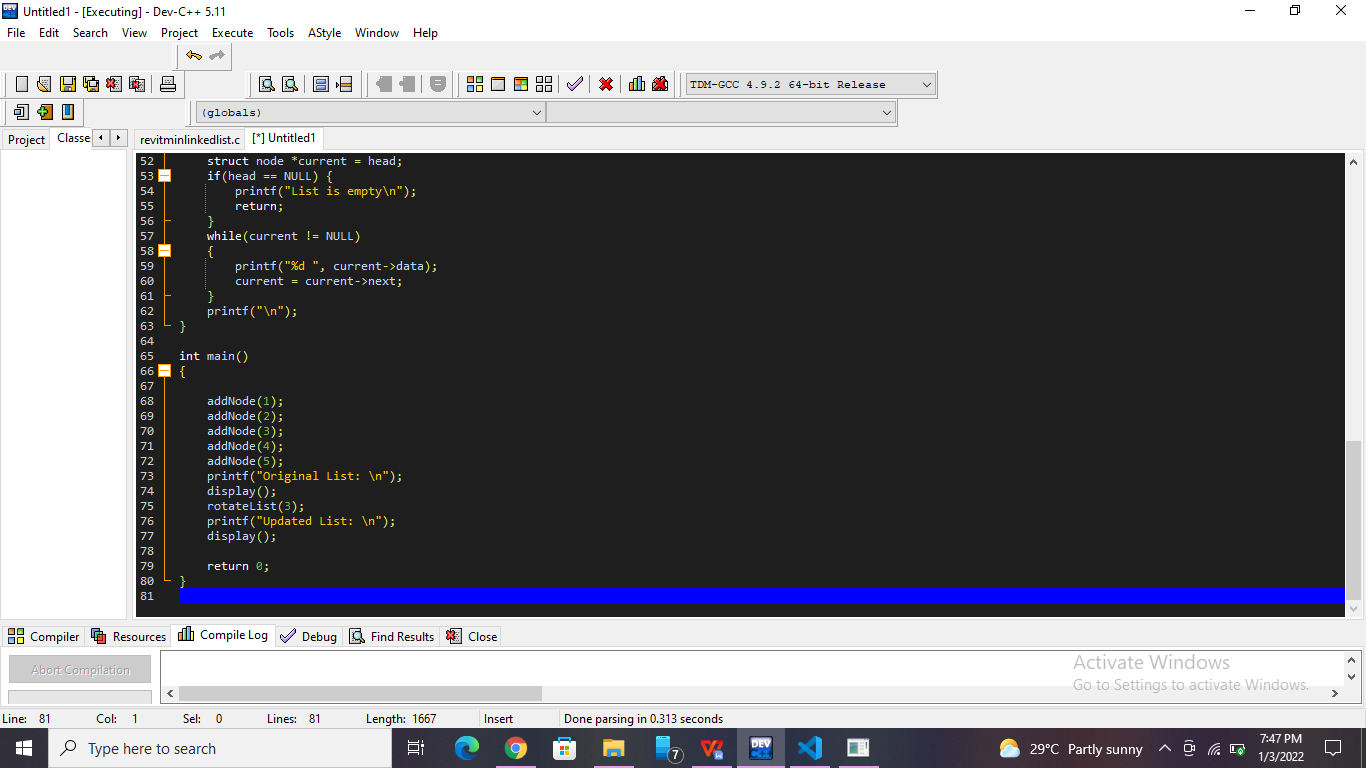
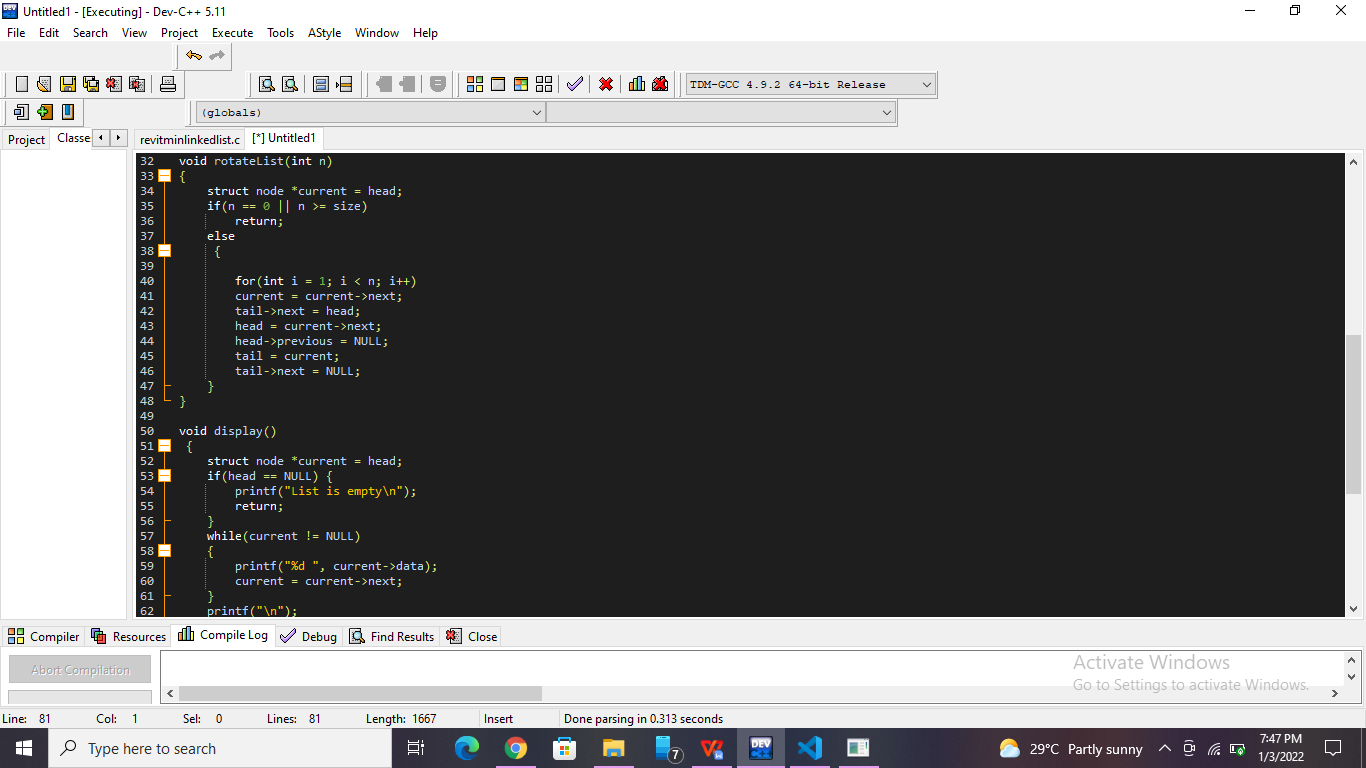
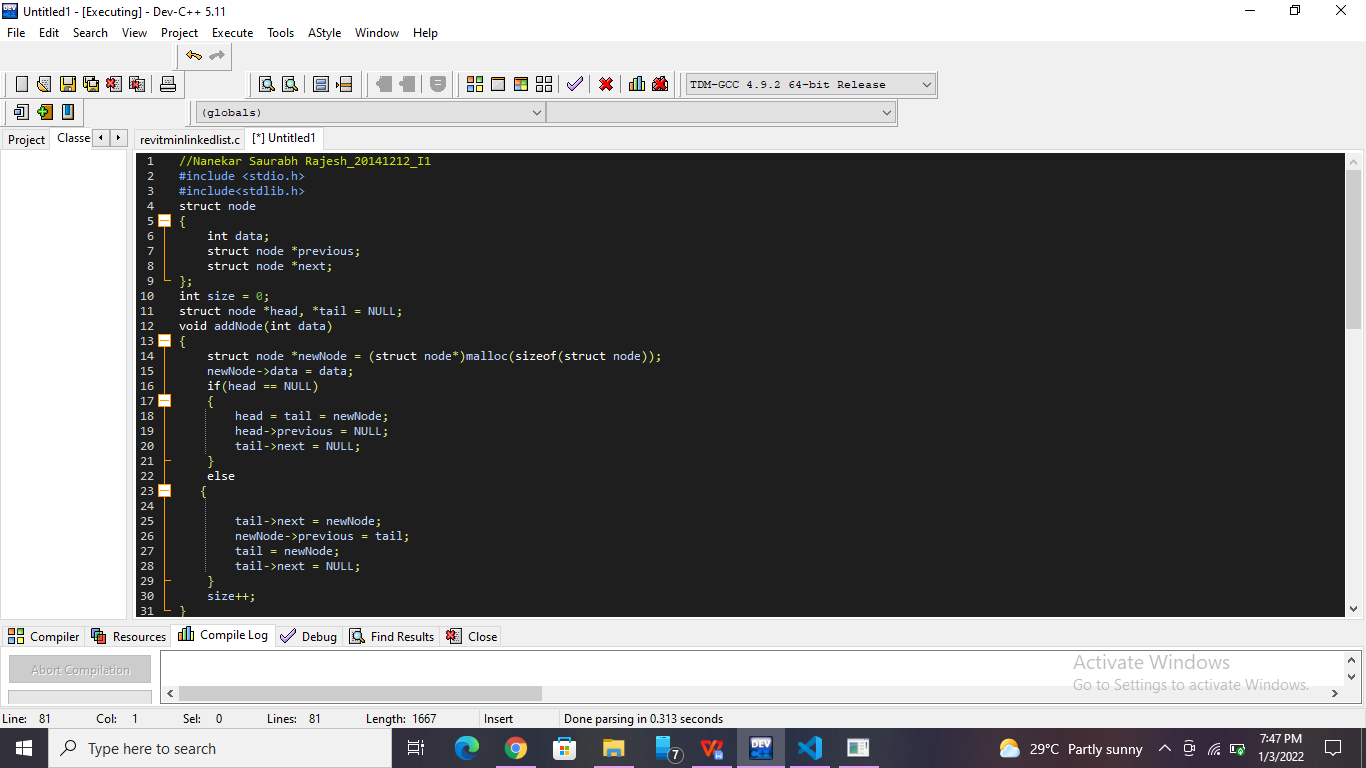
    rotateList(3);

    printf("Updated List: \n");

    display();

    return 0;

}

**Screenshot of Practice Program:-**

**List of sample questions for oral examination:**

1. What do mean by a double linked list?
2. What are the types of doubly linked list?
3. What is doubly linked list advantages?
4. Why is doubly linked list more useful than singly linked list?
5. Can a doubly linked list be circular?

**Conclusion:**

The doubly linked list can be traversed in forward as well as backward directions, unlike singly linked list which can be traversed in the forward direction only. Delete operation in a doubly-linked list is more efficient when compared to singly list when a given node is given.