**Name: Riya Jain**

**Section: A (CSE)**

**Roll No: 17**

**EXPERIMENT: 6**

**Aim:** To study doubly-linked linear list and implement various operations on it – Insert, Delete, Reverse, Sorting, Locate, Linked Stack, and Linked Queue”.

**Problem Definition:** Create a self-referential structure, Node to represent a node of a doubly linked linear list. Implement the routines to (1) find the length of the list, (2) create a list, (3) insert an element – at the beginning, at the end, and at a specified position in the list, insertion in an ordered way (4) delete an element from the beginning, end or a specified position at the list, (5) reverse the list, (6) search the list, (7) Sort the list. Create a menu-driven program to test these routines. Use the singly-linked linear list routines to implement a linked stack and a linked queue.

**Theory:**

**Linear Data Structure:** A Linear data structure has data elements arranged in a sequential manner and each member element is connected to its previous and next element. This connection helps to traverse a linear data structure in a single level and in a single run. Such data structures are easy to implement as computer memory is also sequential. Examples of linear data structures are List, Queue, Stack, Array etc.

Types of Linked Lists:

Following are the various types of linked lists.

● Simple Linked List − Item navigation is forward only.

● Doubly Linked List − Items can be navigated forward and backward.

● Circular Linked List − The last item contains a link of the first element as next and the first element has a link to the last element as previous.

**Doubly Linked Lists:**

A doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly-linked list, a node consists of three parts: node data, a pointer to the next node in sequence (next

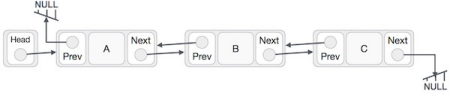
pointer), a pointer to the previous node (previous pointer). Following are the important terms to understand the concept of a doubly linked list.

● Link − Each link of a linked list can store data called an element.

● Next − Each link of a linked list contains a link to the next link called Next.

● Prev − Each link of a linked list contains a link to the previous link called Prev.

● LinkedList − A Linked List contains the connection link to the first link called First and to the last link called Last.



**Basic Operations on Linked Lists:**

Following are the basic operations supported by a list.

● Insertion − Adds an element at the beginning of the list.

● Deletion − Deletes an element at the beginning of the list.

● Display − Displays the complete list.

● Search − Searches an element using the given key.

● Delete − Deletes an element using the given key.

**Algorithm:**

**1. Insertion at the beginning:**

1. Create a NewNode

2. NewNode → DATA = DATA

3. If (START equal to NULL)

(a) NewNode → RPoint = NULL

(b) NewNode → LPoint = NULL

(c)START = NewNode

4. Else

(a) NewNode → LPoint = NULL

(b) NewNode → RPoint = START

(c) START → LPoint = NewNode

(d) START = NewNode

5. Exit

**2. Insertion at the end:**

1. Create a NewNode

2. NewNode → DATA = DATA

3. NewNode → RPoint = NULL

4. If (START equal to NULL)

(a) NewNode → RPoint = NULL (b) NewNode → LPoint = NULL

(c) START = NewNode

5. Else (a) TEMP = START

(b) While ((TEMP → RPoint )<>NULL)

(i) TEMP =TEMP → RPoint

6. TEMP → RPoint = NewNode

7. NewNode → LPoint = TEMP

8. Exit

**3. Insertion at a specific position:**

1. Input the position as, POS

2. TEMP = START

3. i = 0

4. while((i < POS-1) and (TEMP <>NULL)) (a) TEMP = TEMP → RPoint;

(b) i = i +1

5. If((TEMP <> NULL) and (i = POS-1)) (a) Create a New Node

(b) NewNode → DATA = DATA

(c) NewNode → RPoint = TEMP → RPoint (d) NewNode → LPoint = TEMP

(e) (TEMP → RPoint) → LPoint = NewNode (f ) TEMP → RPoint = New Node

6. Else

(a) Display “Position NOT found”

7. Exit

**4. Deletion at the beginning:**

1. Is the List Empty?

If START=NULL

Print “ Empty List, Deletion not Possible” Return START

2. Is Only one node in List?

If START → RPoint = NULL

Key = START → DATA

FREE( START)

Exit

3. TEMP = START

4. Key = TEMP → DATA

5. START = START → Rpoint

6. START → Lpoint = NULL

7. FREE(TEMP)

8. Print “Key deleted is: Key”

9. Exit

**5. Deletion at the end:**

1. Is the List Empty?

If START=NULL

Print “ Empty List, Deletion not Possible” Return START

2. Is Only one node in List?

If START → Next = NULL

Key = START → DATA

FREE( START)

Return NULL

3. TEMP = START

4. Traverse till end of list

While ((TEMP → RPoint )<>NULL) TEMP = TEMP → RPoint

5. Delete node & Adjust pointers (i) TEMP → LPoint → RPoint = NULL (ii) FREE(TEMP)

6. Exit

**6. Deletion at specific position:**

1. Initialize TEMP = START; and k = 0

2. Is the List Empty?

If TEMP=NULL

Return NULL

3. while(k < POS-1)

(a) TEMP = TEMP →RPoint

(b) If (TEMP = NULL)

(i) Display “Node in the list less than

the position”

(ii) Exit

(c) k = k + 1

4. HOLD = TEMP → RPoint → RPoint

5. Key = Temp → RPoint → DATA

6. FREE(TEMP → RPoint)

7. TEMP → RPoint = HOLD

8. HOLD → LPoint = TEMP

9. Print “Deleted Key is “,Key

10. EXIT

**7. Traversal:**

1. If (START is equal to NULL)

(a) Display “The list is Empty”

(b) Exit

2. Initialize TEMP = START

3. Repeat the step 4 and 5 until (TEMP not equal to NULL)

4. Display TEMP → DATA

5. TEMP = TEMP → Next

6. Exit

**8. Search:**

1. IF HEAD == NULL

2. WRITE "UNDERFLOW"

GOTO STEP 8

[END OF IF]Step 2: Set PTR = HEAD 3. Set i = 0

4. Repeat step 5 to 7 while PTR != NULL 5. IF PTR → data = item

6. return i

[END OF IF]

7. i = i + 1

8. PTR = PTR → next

9. Exit

**10. Sorting:**

1.HOLD = START

2. Initialize CNT=0, i=0

3. while(HOLD <> NULL)

(a)HOLD=HOLD → next

(b)CNT=CNT+1

4. while(i less than CNT-1)

5. TEMP = START

6. while ((TEMP→ NEXT) <> NULL)

(a) if ((TEMP →DATA) >(TEMP → NEXT →DATA)) (i) k=TEMP →DATA

(ii)TEMP →DATA = TEMP → NEXT →DATA (iii)TEMP → NEXT →DATA =k

(b) TEMP = TEMP →NEXT

7. i=i+1

8. EXIT

**11. Copy:**

list\* Copy(list\* s)

{

list \*temp;

list \*s1=NULL;

temp=s;

while (temp!=NULL)

{

s1 = InsertAtEnd(s1,temp->data);

temp=temp->next;

}

return s1;

}

**12. Reverse:**

1. current ← head;

2. While (current != NULL) do

temp ← current.next;

current.next ← current.prev;

current.prev ← temp;

current ← temp;

End while

3. temp ← head;

head ← last;

last ← temp;

4. End

**CODE**:

#include<stdio.h>

#include<stdlib.h>

struct Node

{

    int data;

    struct Node\*LPoint;

    struct Node\*RPoint;

};

typedef struct Node List;

List\*create\_Node(List\*head);

List\*create\_space();

List\*Insertatbegin(struct Node\*head,int data);

List\*Insertatend(struct Node\*head,int data);

List\*Insertatindex(struct Node\*head,int data,int index);

List\*Deleteatbegin(struct Node\*head);

int length(struct Node\*ptr);

void traversallinkedlist(struct Node\*ptr);

void displaylinkedlist(struct Node\*head);

void search(struct Node\*head,int ele);

List\*DeleteatEnd(struct Node\*head);

List\*Deleteatindex(List\*head,int index);

void concatenation();

void Copy(List\*head);

List\*Reverse(List\*head);

void sorting(List\*head);

int main()

{

List\*head=NULL;

int ch,n,element,element1,index=0,indexx=0;

while(1)

{

printf("\nDear user please enter operation you want to perform on the array of your choice!!!");

printf("\nPRESS: 1)Insertion Operations\n 2)Deletion Operations\n 3)Concatenation\n 4)length of linked list\n 5) Searching\n 6) Copy\n 7)Reverse\n8)Sorting\n9)Traversing\n10)exit:");

scanf("%d",&ch);

switch(ch)

{

case 1:printf("You opted for Insertion Operations!!");

       int ch1,l=1;

 while(l)

 {

 printf("\nPRESS: 1)Insertatbegin 2)Insertatend 3)Insertatindex 4)display:");

 scanf("%d",&ch1);

 switch(ch1)

 {

case 1: printf("You opted for Insertion at Beginning!\n");

        printf("Enter data do you want to insert:");

        scanf("%d",&element);

         head=Insertatbegin(head,element);

         break;

case 2: printf("You opted for Insertion at End!!\n");

        printf("Enter the data do you want to insert:");

        scanf("%d",&element);

        head=Insertatend(head,element);

        break;

case 3: printf("You opted for Insertion at Index!!\n");

        printf("Enter the data do you want to insert:");

        scanf("%d",&element);

        printf("Enter the index position do you want to insert:");

        scanf("%d",&index);

        head=Insertatindex(head,element,index);

        break;

case 4: printf("You opted for Displaying Linkedlist!!");

        printf("\nYour Linked list is:\n");

        traversallinkedlist(head);

        break;

case 5: l=0;

        break;

default: printf("You entered invalid choice");

}

}

break;

case 2: printf("You opted for Deletion Operations!!");

         int ch2,l1=1;

 while(l1)

 {

 printf("\nPRESS: 1)Deletionatbegin 2) Deletionatend 3)Deletionatindex 4)display:");

 scanf("%d",&ch2);

 switch(ch2)

 {

case 1: printf("You opted for Deletion at beginning!\n");

        head=Deleteatbegin(head);

        break;

case 2: printf("You opted for Deletion at End!!\n");

        head=DeleteatEnd(head);

        break;

case 3: printf("You opted for Delete a value from a particular index in your linked list!!\n");

        printf("Enter the index position do you want to delete:");

        scanf("%d",&indexx);

        head=Deleteatindex(head,indexx);

        break;

case 4: printf("You opted for displaying Linked list!!");

        printf("\nYour Linked list is:\n");

        displaylinkedlist(head);

        break;

case 5: l1=0;

        break;

default: printf("You entered invalid choice");

}

}

break;

case 3: printf("You opted for Concatenation Operation!!\n");

        concatenation();

        break;

case 4: printf("You opted for Calculation of length of linked List Operation!!");

        int res=length(head);

        printf("\nYour length of linked list is:%d",res);

        break;

case 5: printf("You opted for searching a given node in Linked list!!");

        printf("Enter data do you want to search:");

        scanf("%d",&element1);

        search(head,element1);

break;

case 6:printf("You opted for Copying Operation in Linked list!!");

       Copy(head);

break;

case 7:printf("You opted for Reversing a Linked list!!");

       head=Reverse(head);

break;

case 8:printf("You opted for Sorting a Linked list!!");

       sorting(head);

       break;

case 9: printf("\nYou opted for Traversing a Linked list!!\n");

        traversallinkedlist(head);

        break;

/\*case 10:printf("You opted for Stack Operations!!");

       int ch3,l2=1;

 while(l2)

 {

 printf("\nPRESS: 1)PUSH 2)POP 3)TOP VALUE 4)display:");

 scanf("%d",&ch3);

 switch(ch3)

 {

case 1: printf("You opted for PUSH OPERATION!\n");

        printf("Enter data do you want to insert:");

        scanf("%d",&element);

        head=Insertatbegin(head,element);

        break;

case 2: printf("You opted for POP OPERATION!!\n");

        head=Deleteatbegin(head);

        break;

case 3: printf("Your top value is:%d",head->data);

        break;

case 4: printf("You opted for Displaying Linkedlist!!");

        printf("\nYour Linked list is:\n");

        traversallinkedlist(head);

        break;

case 5: l2=0;

        break;

default: printf("You entered invalid choice");

}

}

break;

case 11:printf("You opted for Queue Operations!!");

        int ch4,l3=1;

 while(l3)

{

 printf("\nPRESS: 1)Insert in queue 2) Delete from queue 3)display:");

 scanf("%d",&ch4);

 switch(ch4)

{

case 1: printf("You opted for Insertion in Queue!\n");

        printf("Enter data do you want to insert:");

        scanf("%d",&element);

        head=Insertatend(head,element);

         break;

case 2: printf("You opted for Deletion in Queue!!\n");

        head=Deleteatbegin(head);

        break;

case 3: printf("You opted for Displaying Linkedlist!!");

        printf("\nYour Linked list is:\n");

        traversallinkedlist(head);

        break;

case 4: l3=0;

        break;

default: printf("You entered invalid choice");

}

}

 break;\*/

case 10: exit(0);

default : printf("You entered invalid choice!!");

}

}

return 0;

}

List\*create\_Node(List\*head)

{

   int n,d;

   printf("Enter the number of nodes to be created:");

   scanf("%d",&n);

   for(int i=0;i<n;i++)

   {       printf("Enter data:");

           scanf("%d",&d);

           head=Insertatbegin(head,d);

   }

 return head;

}

List\*Insertatbegin(List\*head,int data)

{

    List\*ptr=create\_space();

    ptr->data=data;

    ptr->LPoint=NULL;

    ptr->RPoint=NULL;

    if(head==NULL)

    {

      head=ptr;

      return head;

    }

    ptr->RPoint=head;

    head->LPoint=ptr;

    head=ptr;

    return head;

}

List\*Insertatend(List\*head,int data)

{

   List\*ptr=create\_space();

   ptr->data=data;

   ptr->LPoint=NULL;

   ptr->RPoint=NULL;

   if(head==NULL)

   {

   head=ptr;

   return head;

   }

   List\*p=head;

   while(p->RPoint!=NULL)

   {

      p=p->RPoint;

   }

   p->RPoint=ptr;

   ptr->LPoint=p;

   return head;

}

List\*Insertatindex(List\*head,int data,int index)

{

   if(head==NULL)  // if list is empty

   {

   head=Insertatbegin(head,data);

   return head;

   }

    if(index==0)

    {

    head=Insertatbegin(head,data);

    return head;

    }

     int l=length(head);

     if(index==l)

     {

     head=Insertatend(head,data);

     return head;

     }

   int i=0;

   List\*p=head;

   List\*ptr=create\_space();

   while(i!=index-1)

   {

    p=p->RPoint;

     if (p==NULL)

     {

     printf("Nodes in the list are less than the index provided"); //agar nodes kam hai aur index jyaada

     return head;

     }

    i++;

   }

  // printf("%d",p->data);

   ptr->data=data;

   ptr->RPoint=p->RPoint;

   ptr->LPoint=p;

   (p->RPoint)->LPoint=ptr;

   p->RPoint=ptr;

return head;

}

void traversallinkedlist(List\*ptr)

{

    if(ptr==NULL)

    printf("Node is Empty\n");

    while(ptr!=NULL)

   {

       printf("Address:%lu Element:%d Address:%lu->",(long unsigned)ptr->LPoint,ptr->data,(long unsigned)ptr->RPoint);

       ptr=ptr->RPoint;

   }

}

List\*Deleteatbegin(List\*head)

{

 int temp;

 if(head==NULL)

 {

 printf("Deletion not possible as your linked list is empty");

 return head;

 }

 if(head->RPoint==NULL)

 {  temp=head->data;

    free(head);

    return NULL;

 }

 List\*ptr=head;

 temp=ptr->data;

 head=head->RPoint;

 head->LPoint=NULL;

 free(ptr);

 printf("Your deleted node was data with data value of %d",temp);

 return head;

}

int length(List\*ptr)

{

   if(ptr==NULL)

   {

    return 0;

   }

   int count=1;

   while(ptr->RPoint!=NULL)

   {

           count++;

           ptr=ptr->RPoint;

   }

   return count;

}

List\*create\_space()

{

   List\*head=(struct Node\*) malloc(sizeof(struct Node));

   if(head==NULL) //agar memory allocate hi nhi huyi

   {

       printf("Node not created\n");

       return NULL; //return head

   }

   else

   {

       printf("Node created\n");

       return head;

   }

}

List\*DeleteatEnd(struct Node\*head)

{

 int temp;

 if(head==NULL)

 {

 printf("Deletion not possible as your linked list is empty");

 return head;

 }

 if(head->RPoint==NULL)

 {  temp=head->data;

    free(head);

    return NULL;

 }

 List\*p=head;

 List\*q=head->RPoint;

 while(q->RPoint!=NULL)

 {

         p=p->RPoint;

         q=q->RPoint;

 }

 p->RPoint=NULL;

 temp=q->data;

 free(q);

 printf("Your deleted node was data with data value of %d",temp);

 return head;

}

void displaylinkedlist(struct Node\*head)

{

if(head==NULL)

 {

 printf("Deletion not possible as your linked list is empty");

 }

 struct Node\*temp=head;

while(temp!=NULL)

   {

       printf("Address:%lu Element:%d Address:%lu->",(long unsigned)temp->LPoint,temp->data,(long unsigned)temp->RPoint);

       temp=temp->RPoint;

   }

}

void search(struct Node\*head,int ele)

{

  int i=0;

 List\*p=head;

 while(p!=NULL)

 {

  if(p->data==ele)

  {

  printf("The element is found at index %d",i);

  break;

  }

  else

  {

  p=p->RPoint;

  i++;

  }

}

if(p==NULL)

printf("\nThe data is not found in the list");

}

List\*Deleteatindex(List\*head,int index)

{    int temp;

     List\*p=head;

     if(head==NULL)

     {

      printf("The list is empty");

      return head;

     }

     int l=length(head);

     /\*if(index>l)

     {

        printf("Deletion not possible\n");

        return head;

     } \*/

     if(index==0)

     {

             head=p->RPoint;

             free(p);

             return head;

     }

     if(index==l-1)

     {

        head=DeleteatEnd(head);

        return head;

     }

     int i=0;

     List \*q=head->RPoint;

     while(i!=index-1)

     {

       p=p->RPoint;

       q=q->RPoint;

       if(q==NULL)

       {

       printf("Nodes in the list are less than the index provided"); //agar nodes kam hai aur index jyaada

       return head;

       }

       i++;

     }

     temp=q->data;

     p->RPoint=q->RPoint;

     q->RPoint->LPoint=p;

     free(q);

     printf("Your deleted node was data with data value of %d",temp);

     return head;

 }

 void concatenation()

 {      List \*head1=NULL;

        List \*head2=NULL;

        head1=create\_Node(head1);

        head2=create\_Node(head2);

        List\*p=head1;

         while(p->RPoint!=NULL)

         {

             p=p->RPoint;

         }

         p->RPoint=head2;

         traversallinkedlist(head1);

}

void Copy(List\*head)

{

List \*q=NULL;

List \*p=head;

while (p!=NULL)

{

q=Insertatend(q,p->data);

p=p->RPoint;

}

traversallinkedlist(q);

}

List\*Reverse(List\*head)

{

/\*List\* prev=NULL;

List\*p=head;

List\*next;

while(p!= NULL) {

// Store next

next=p->RPoint;

// Reverse current node's pointer

p->RPoint = prev;

// Move pointers one position ahead.

prev =p;

p=next;

}

head= prev;\*/

List\*p=head;

List\*q=head->RPoint;

p->RPoint=NULL;

p->LPoint=q;

while(q!=NULL)

{

   q->LPoint=q->RPoint;

   q->RPoint=p;

   p=q;

   q=q->LPoint;

} head=p;

return head;

}

void sorting(List\*head)

{

int i=0;

int n=length(head);

List\*p;

while(i<n-1)

{       p=head;

        while(p->RPoint!=NULL)

        {

                if(p->data>p->RPoint->data)

                {

                int temp=p->data;

                p->data=p->RPoint->data;

                p->RPoint->data=temp;

                }p=p->RPoint;

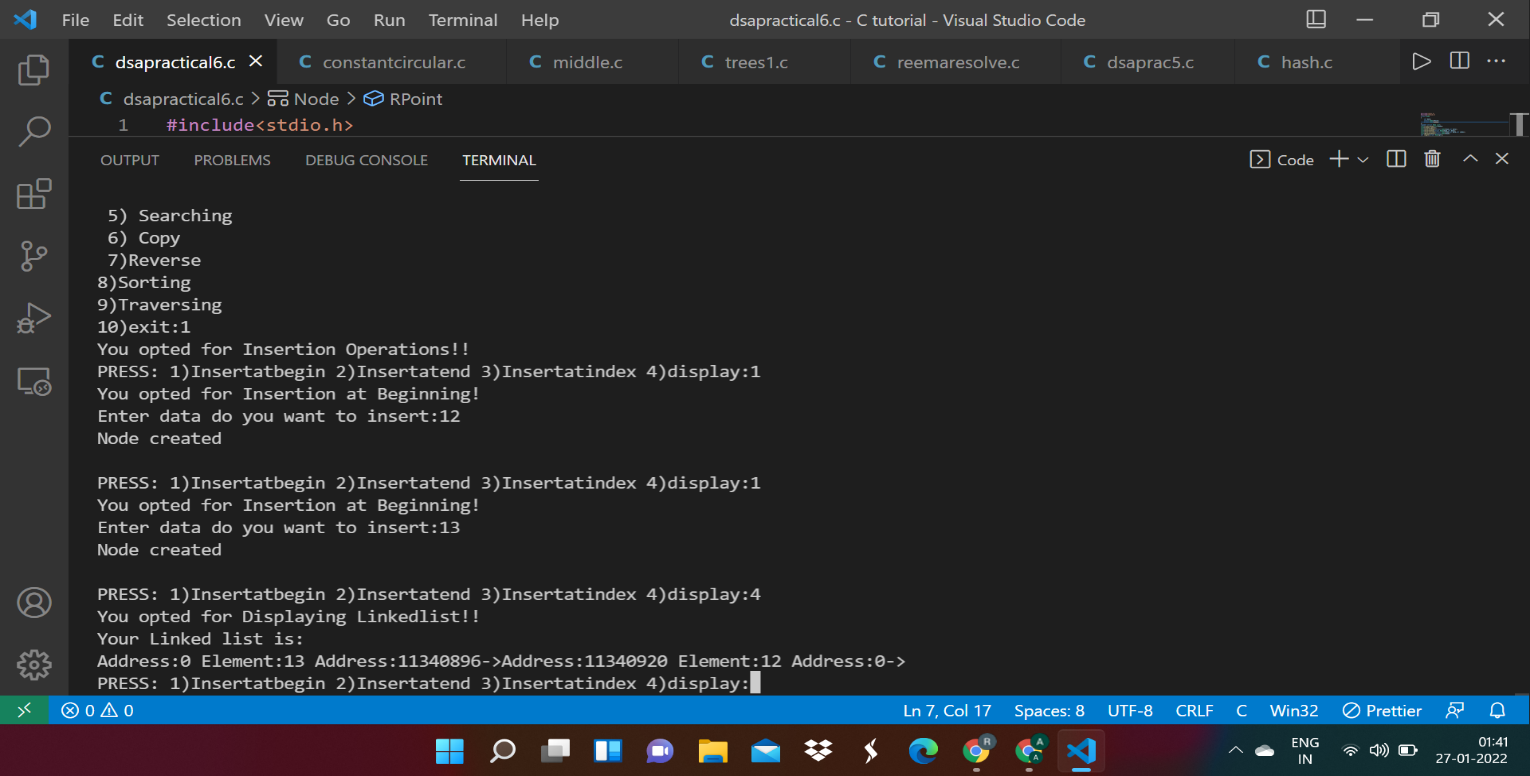
        }

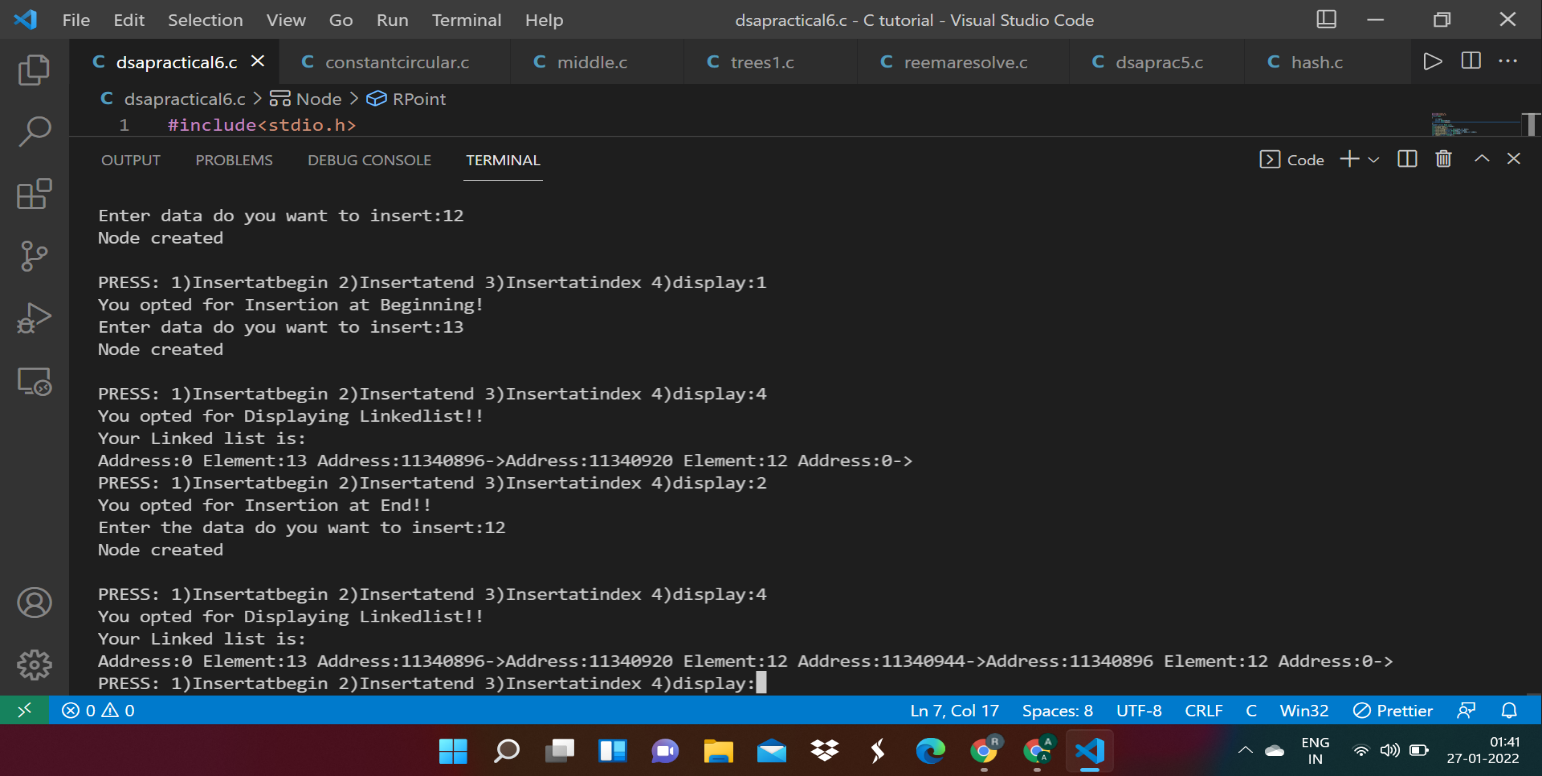
        i++;

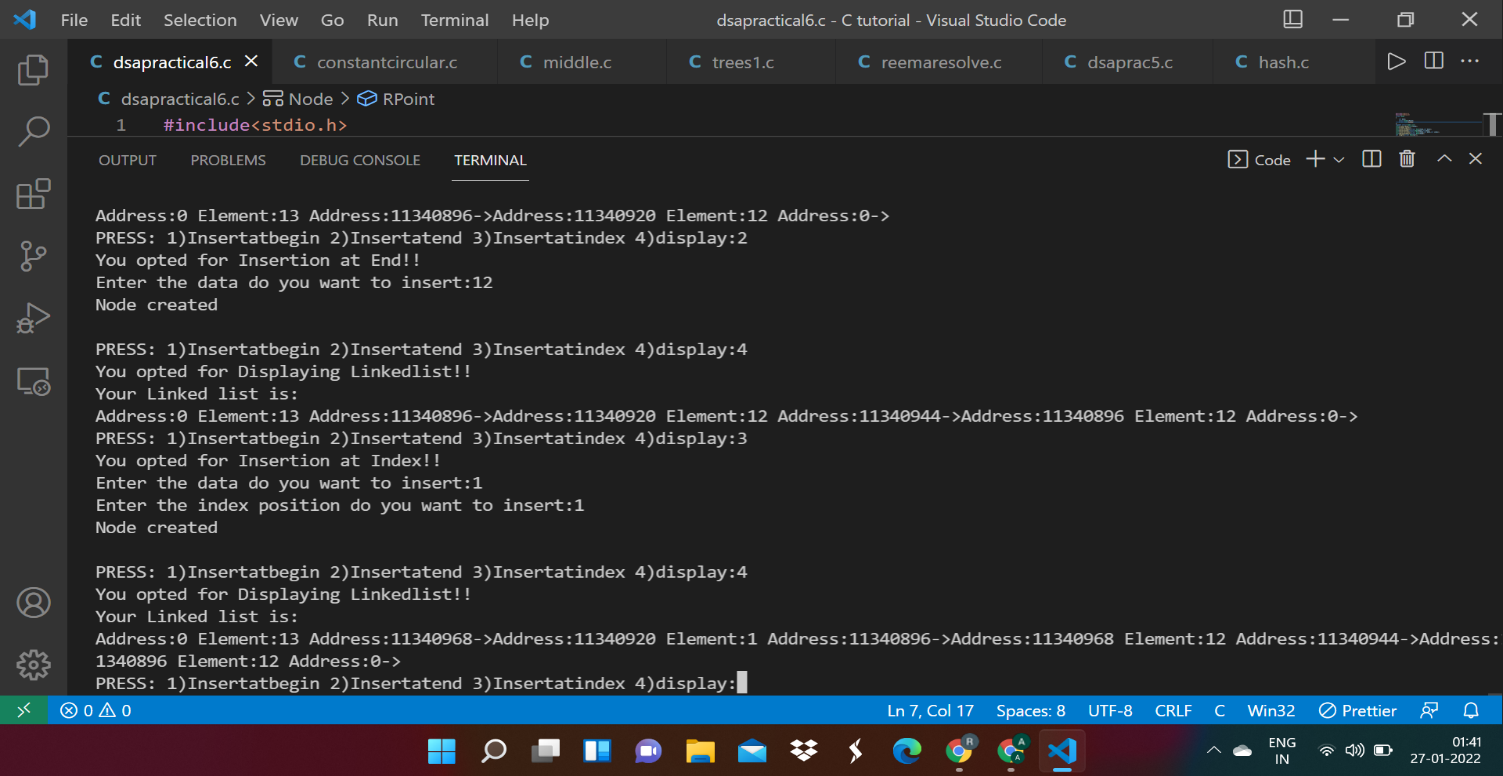
}

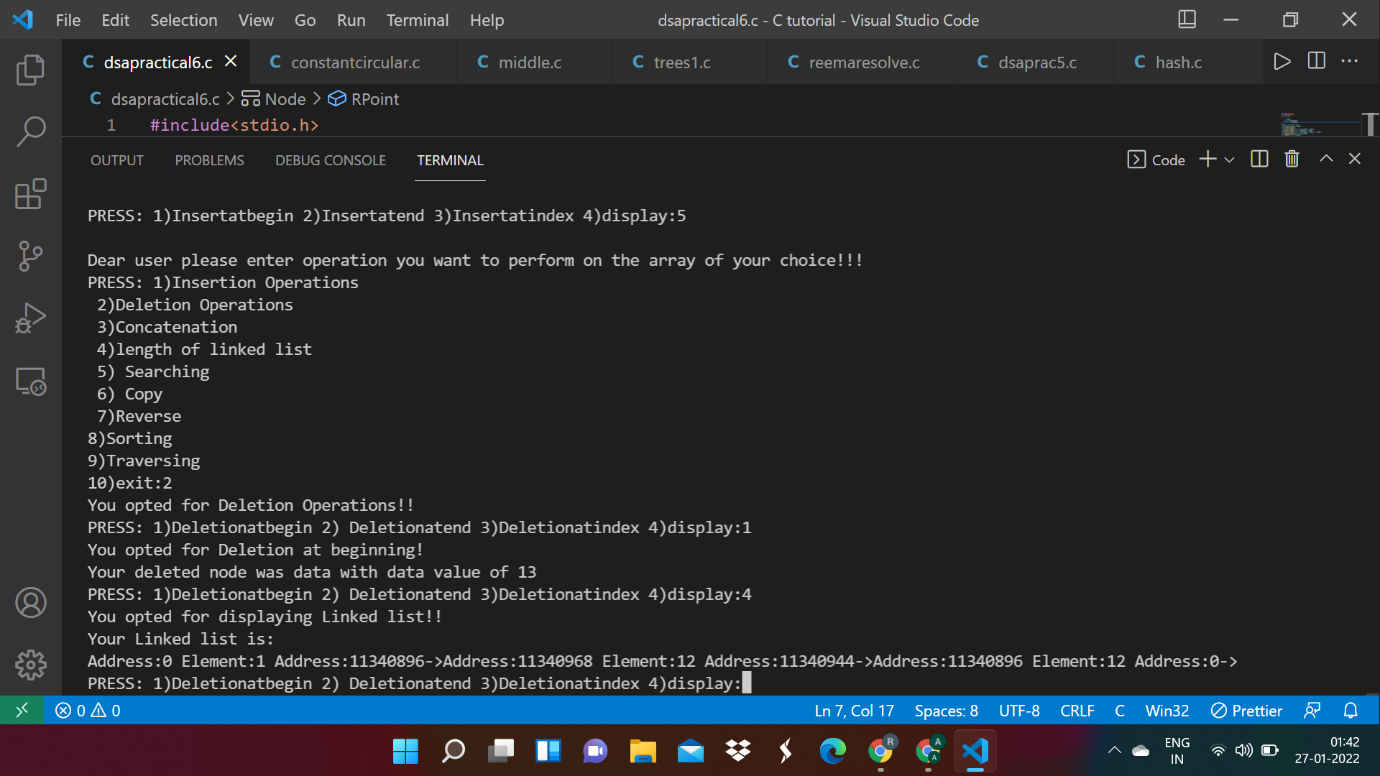
}

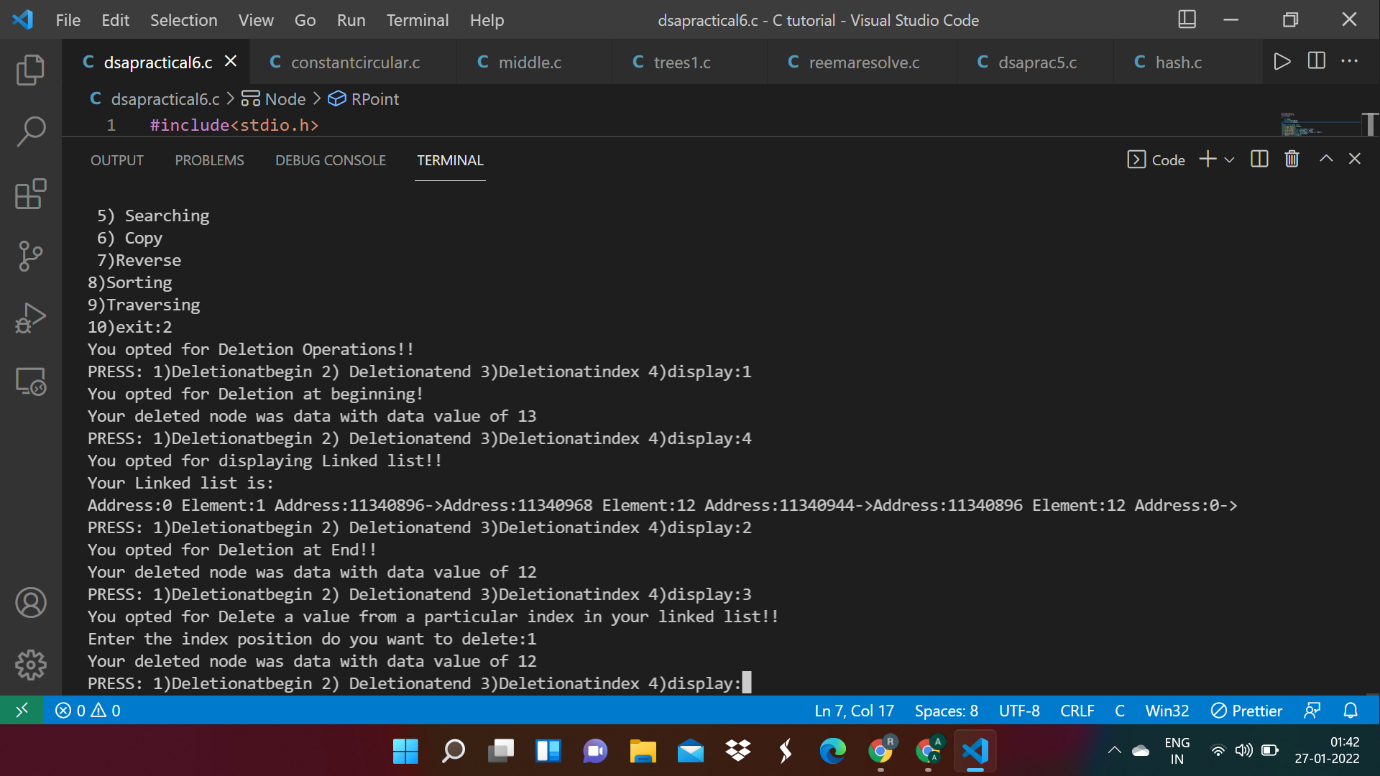
**OUTPUTS:**



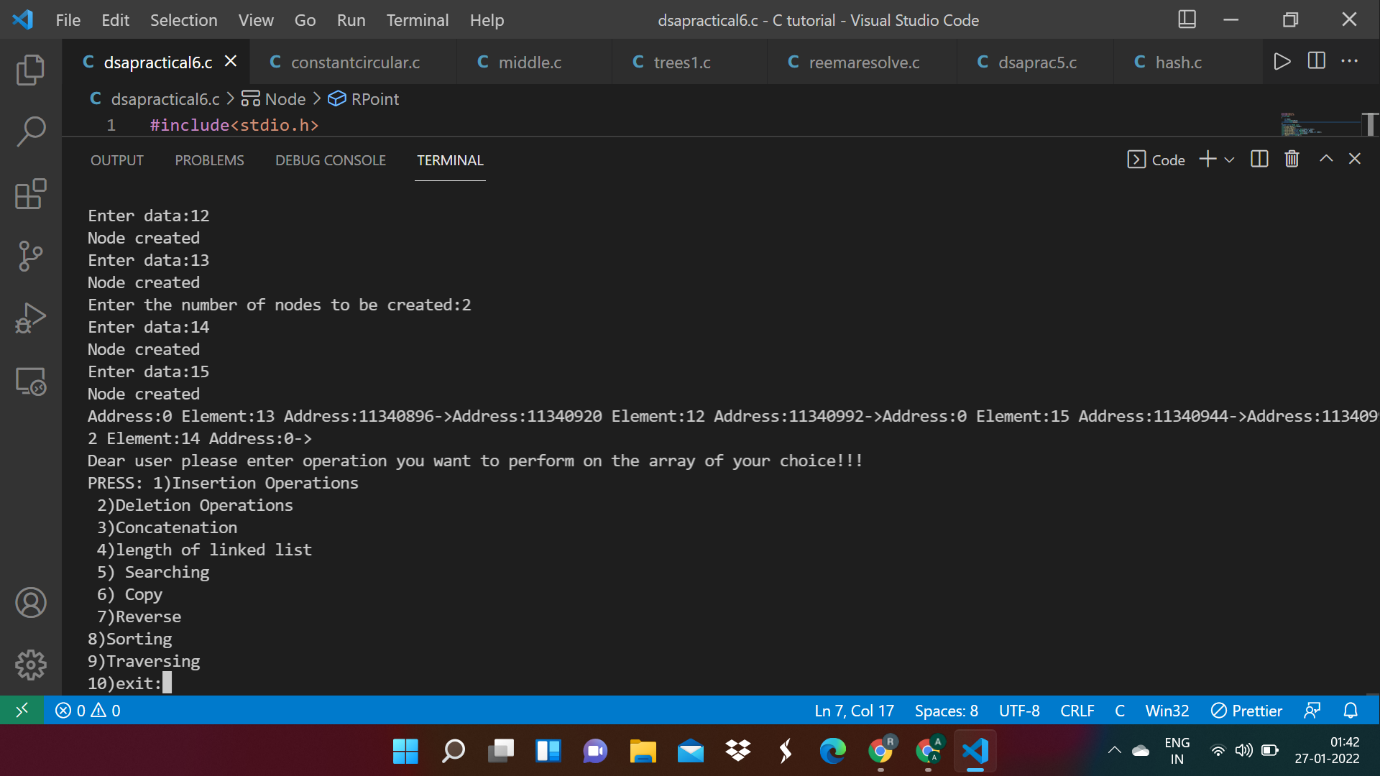


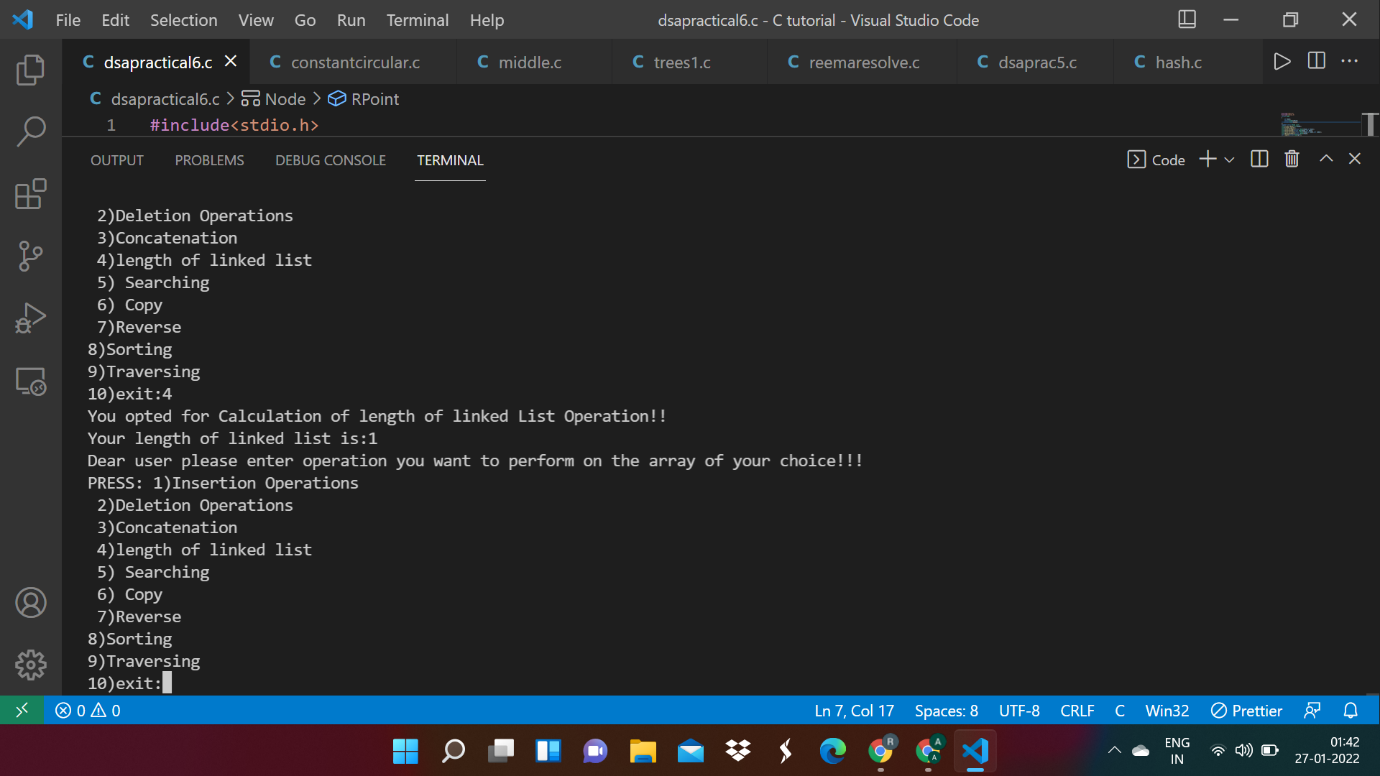


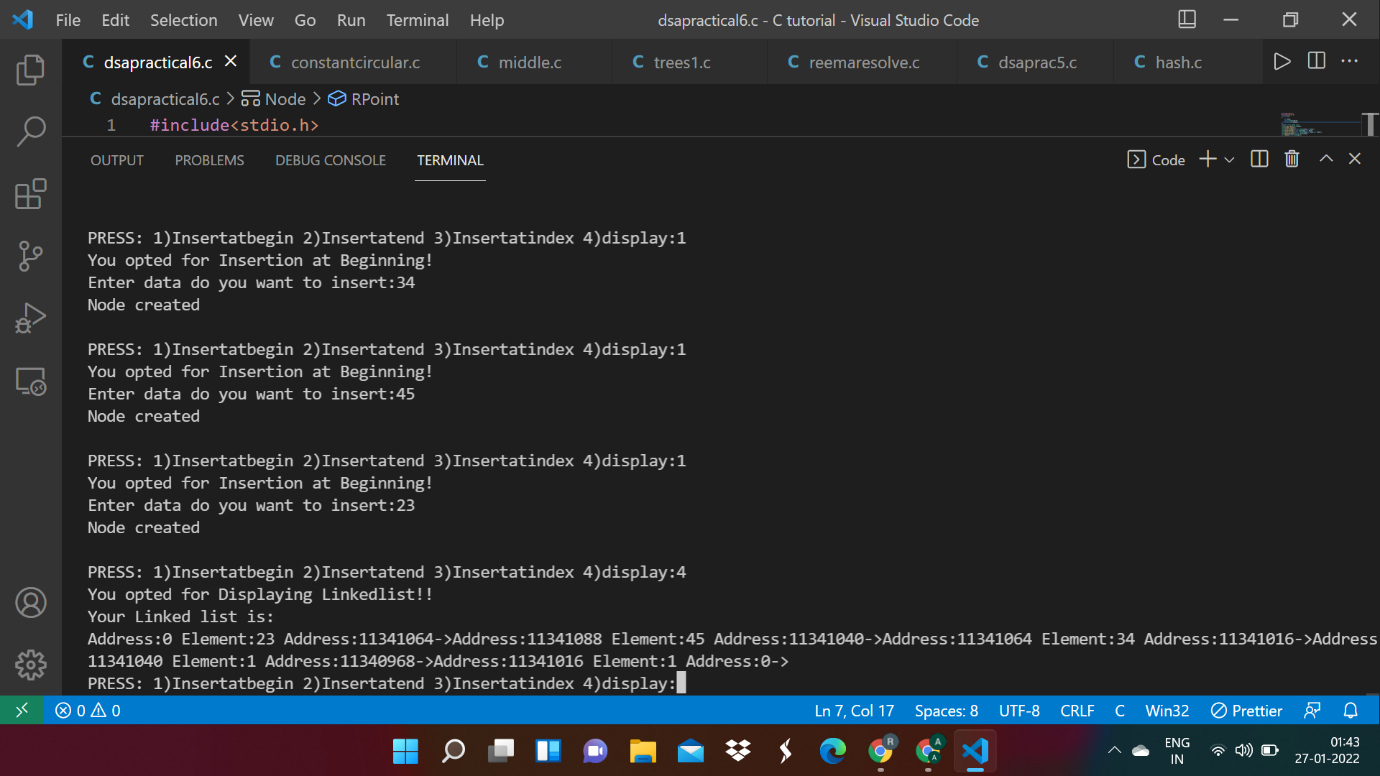


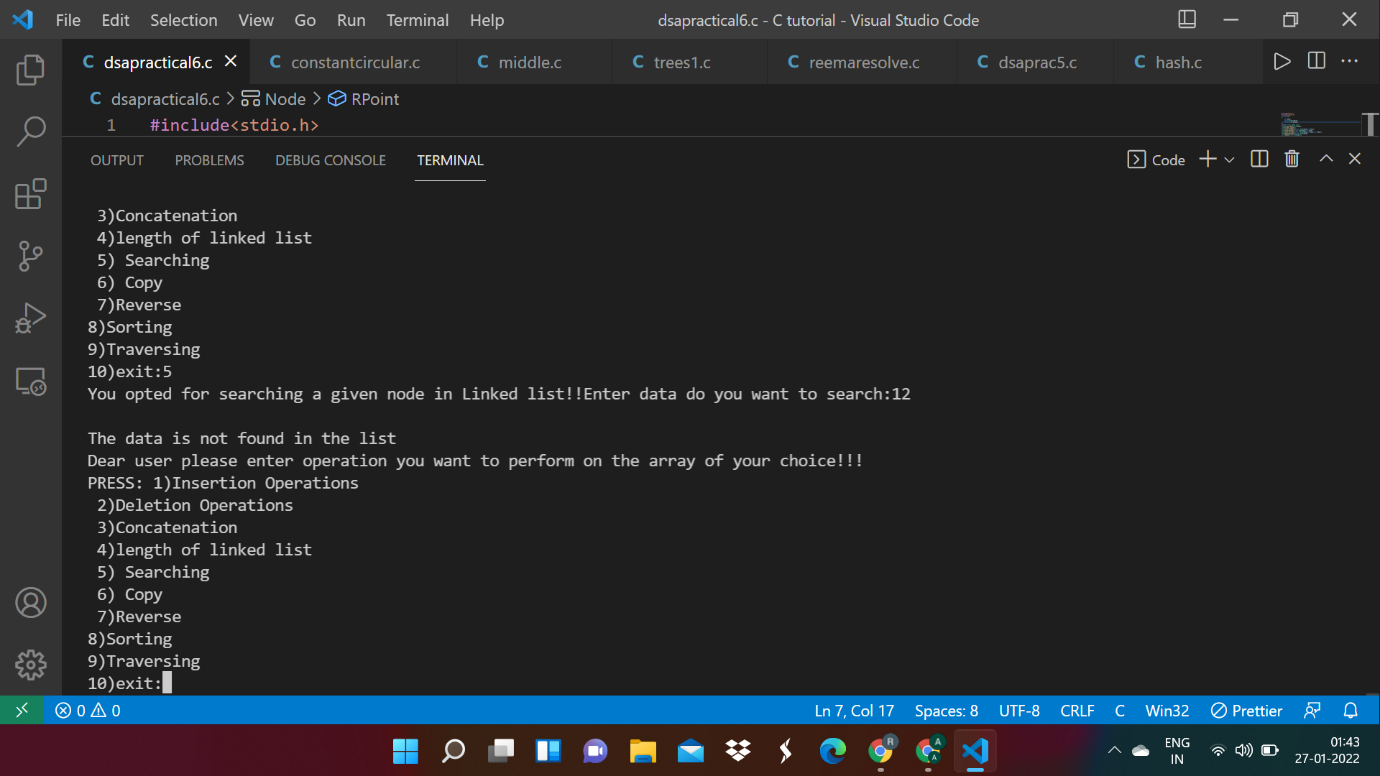


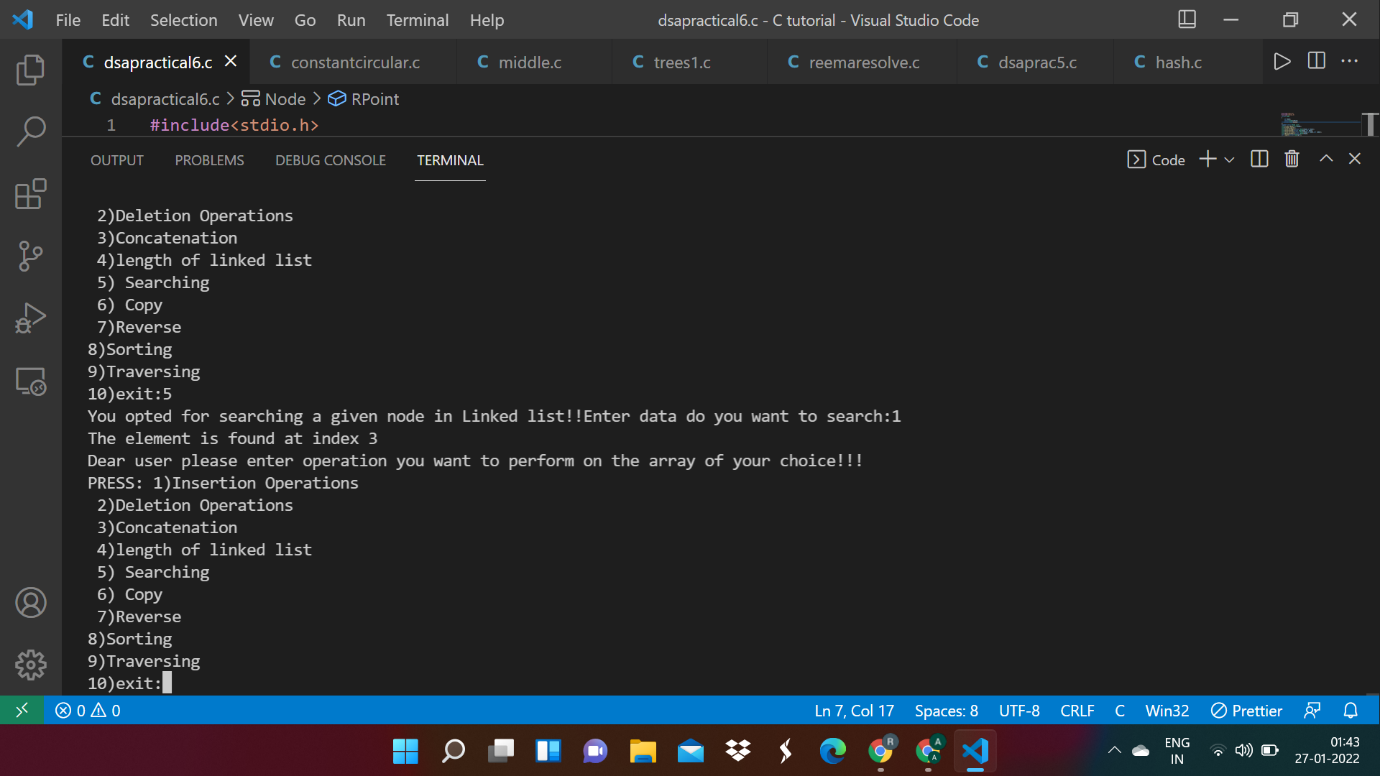


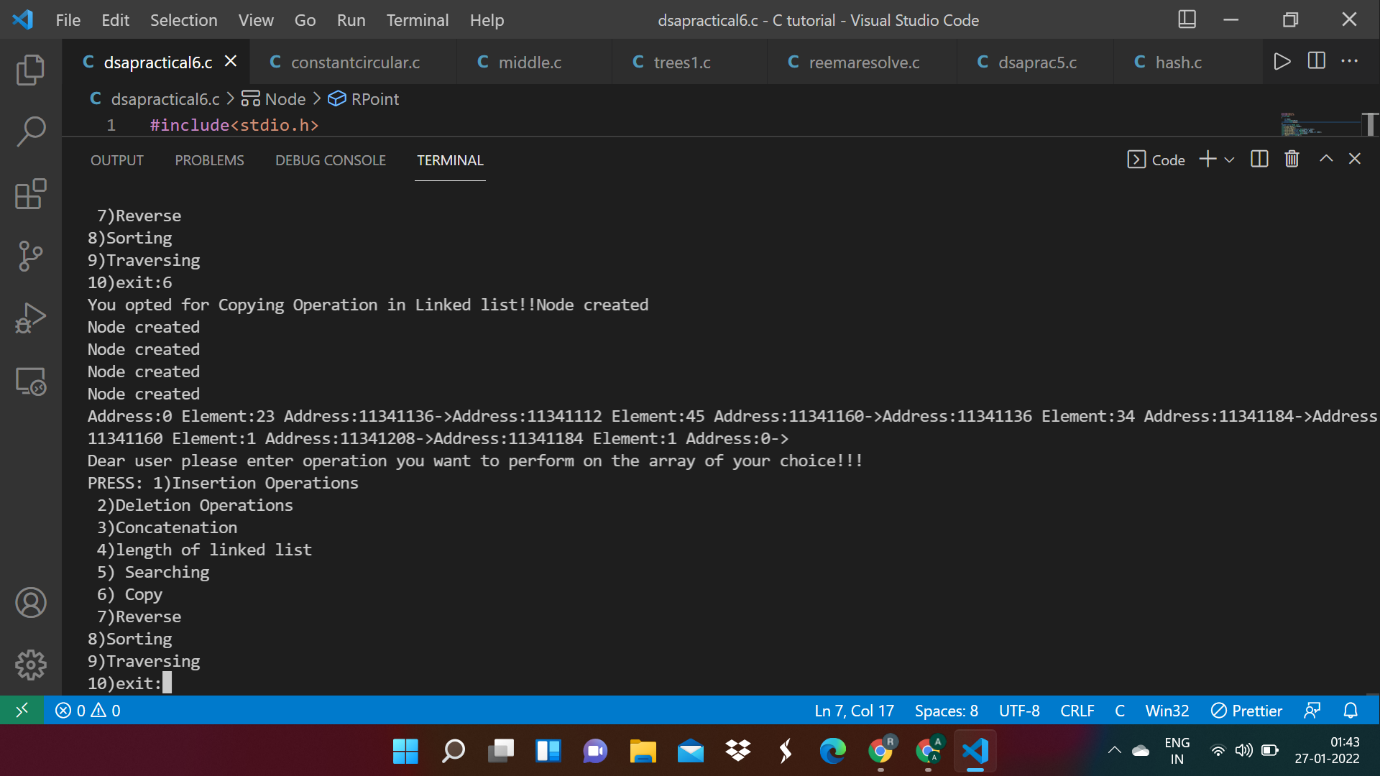


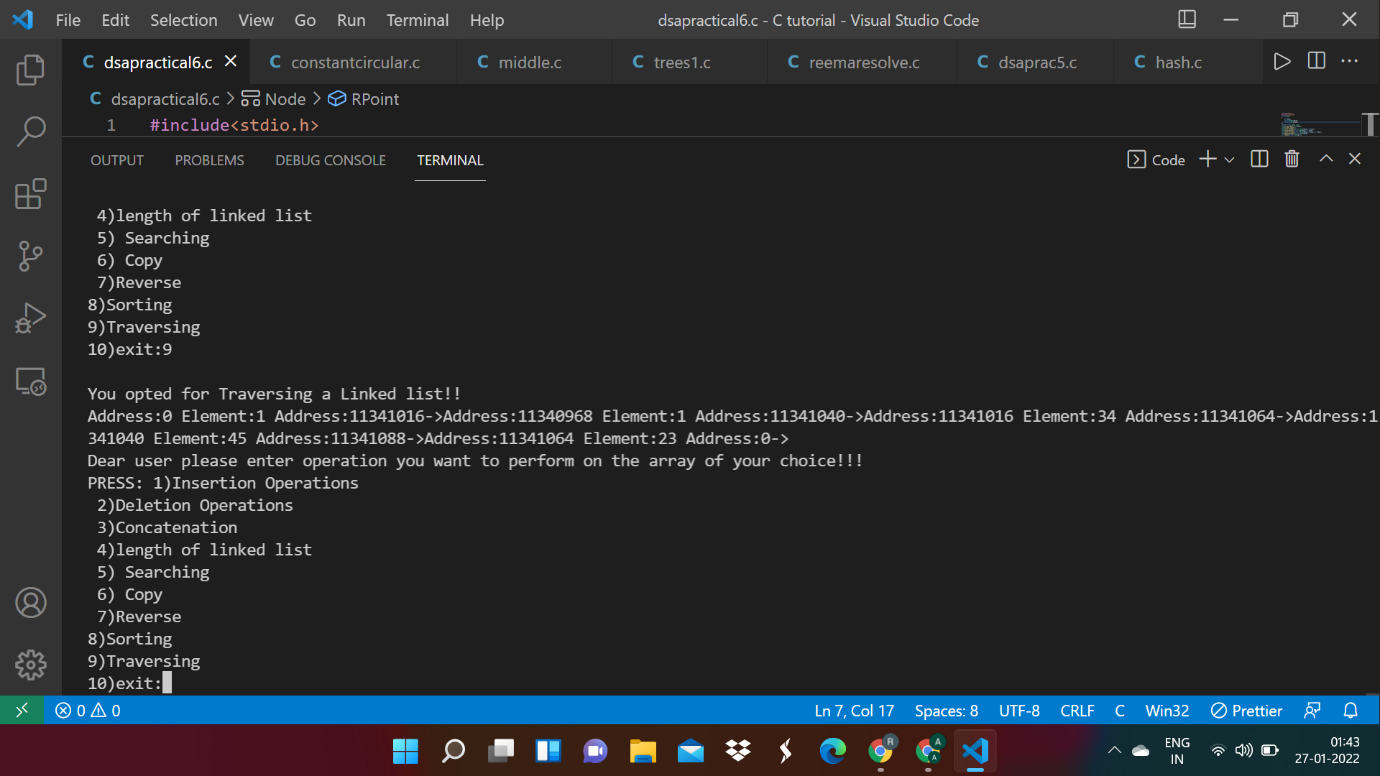


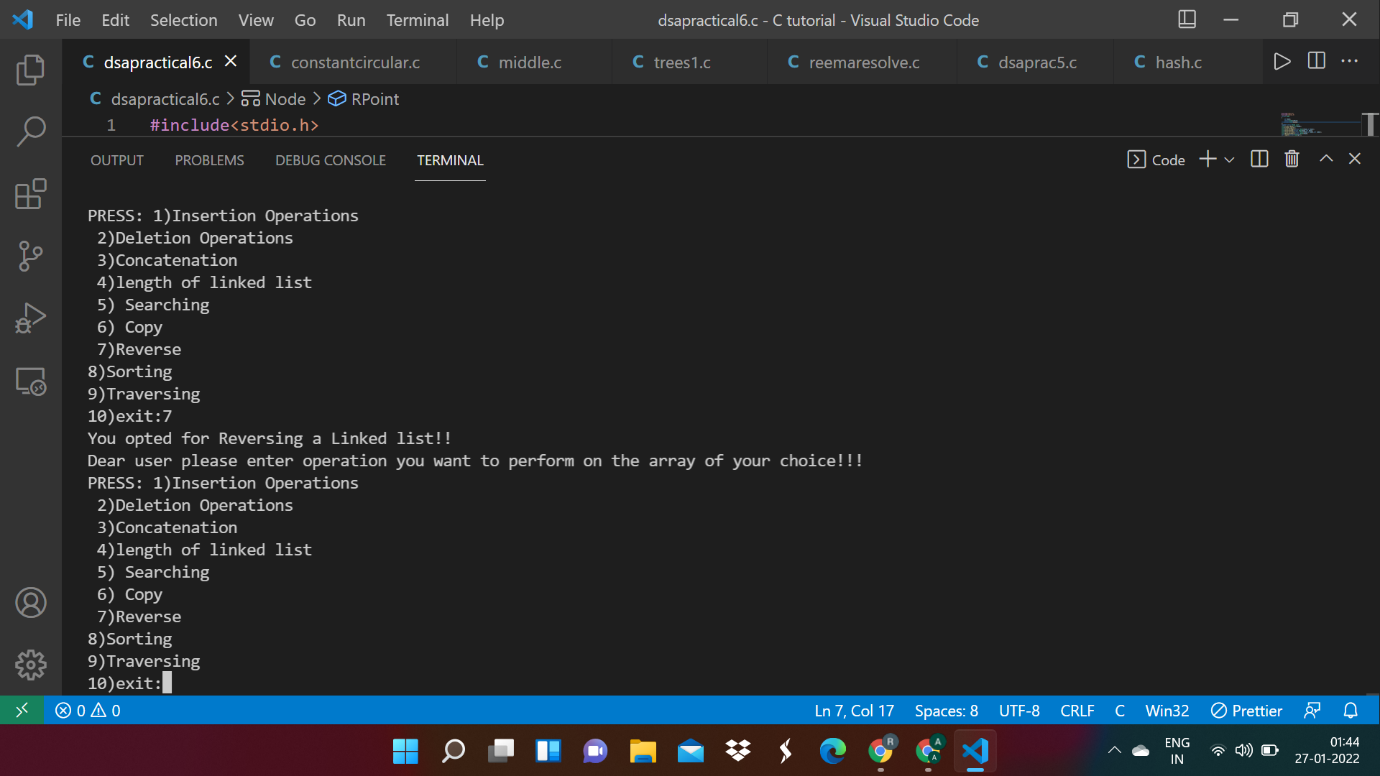


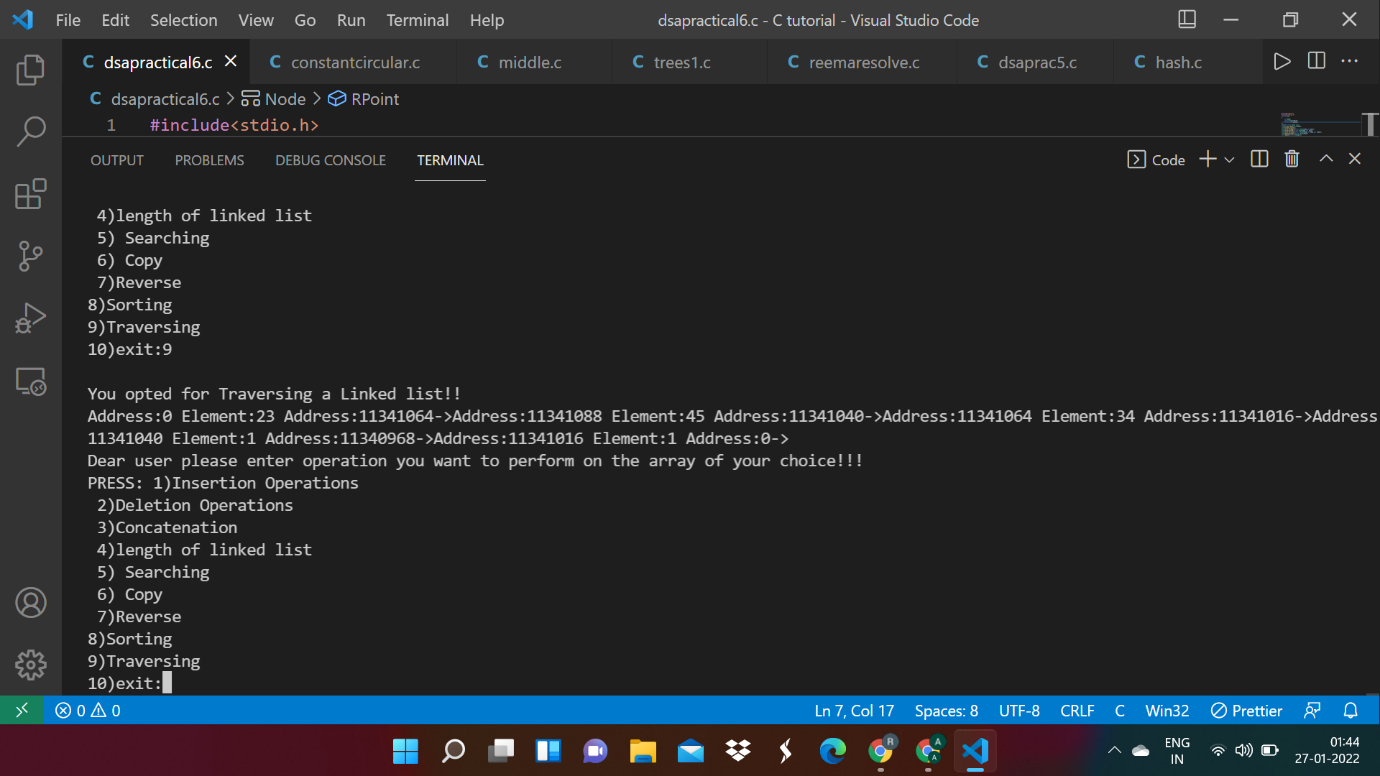


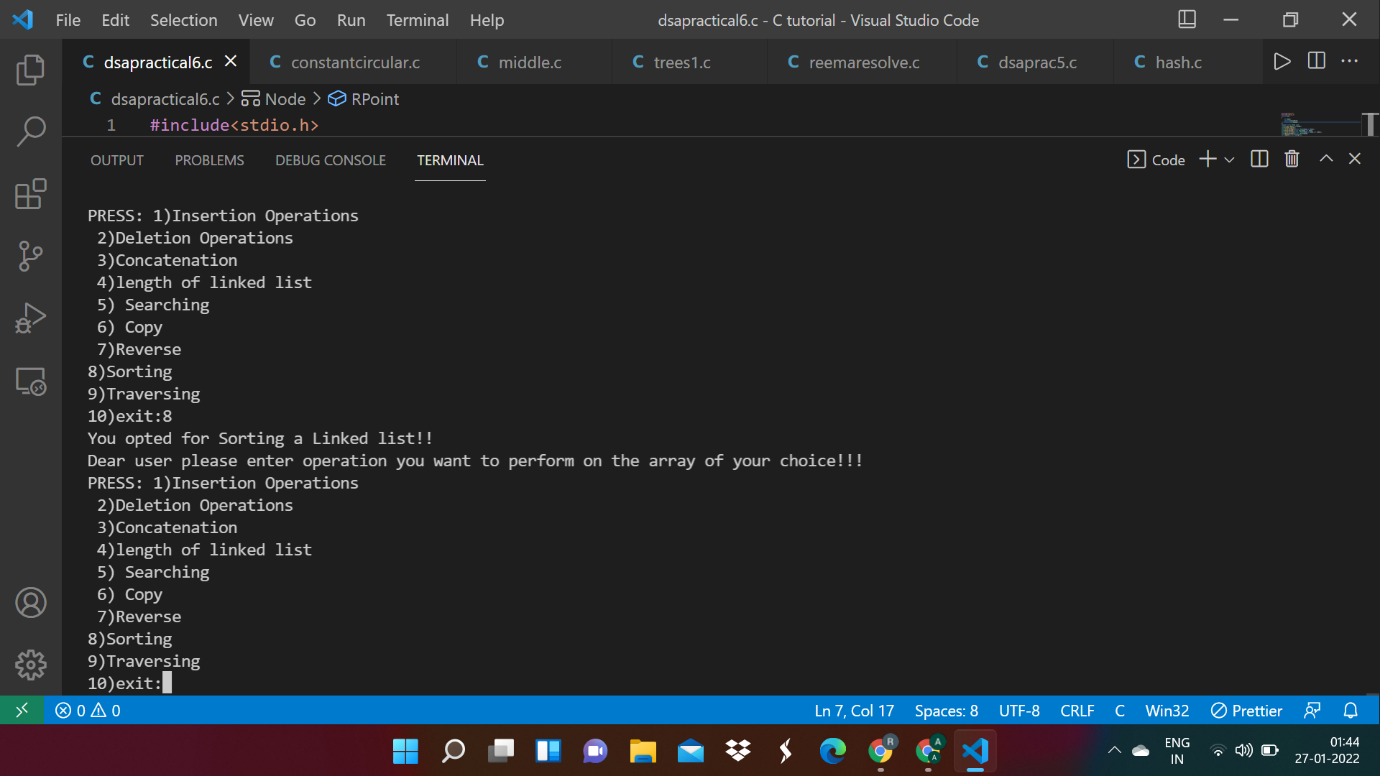


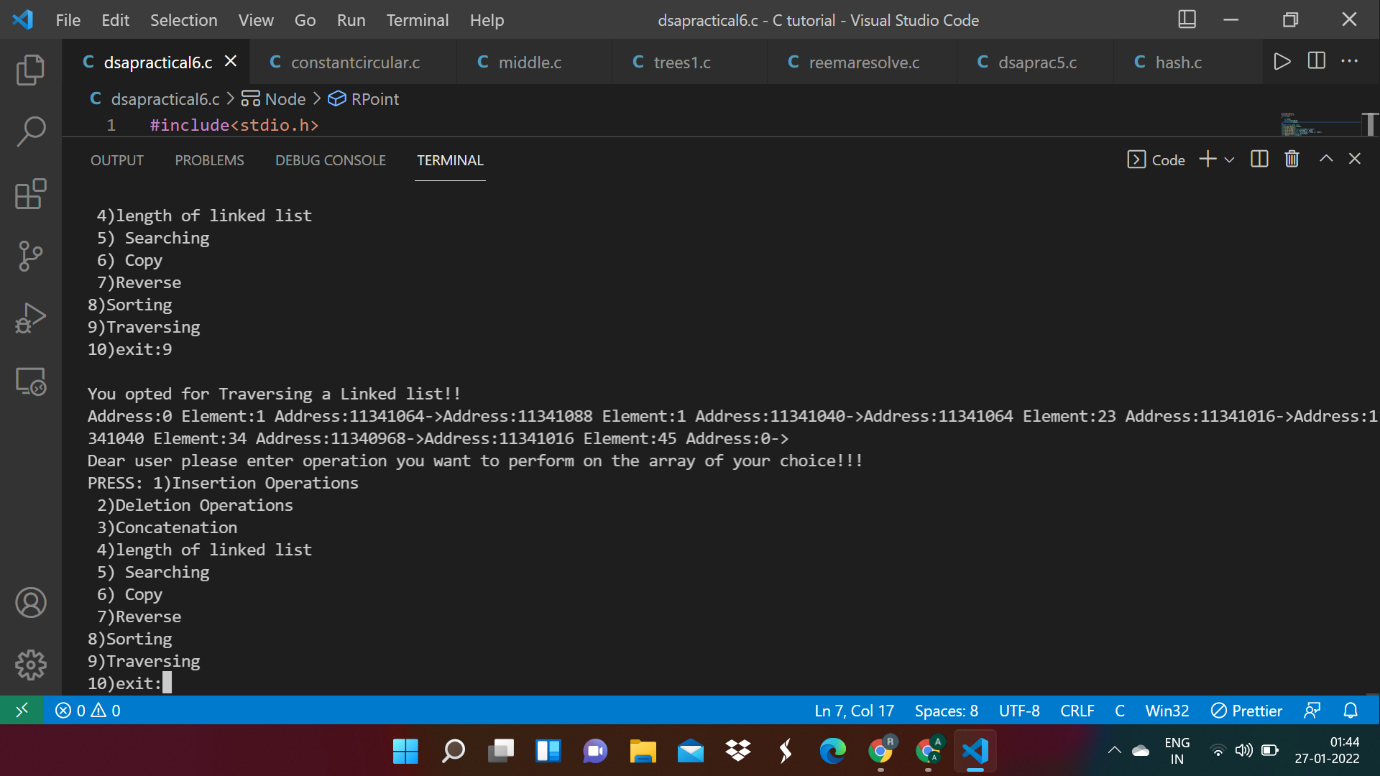


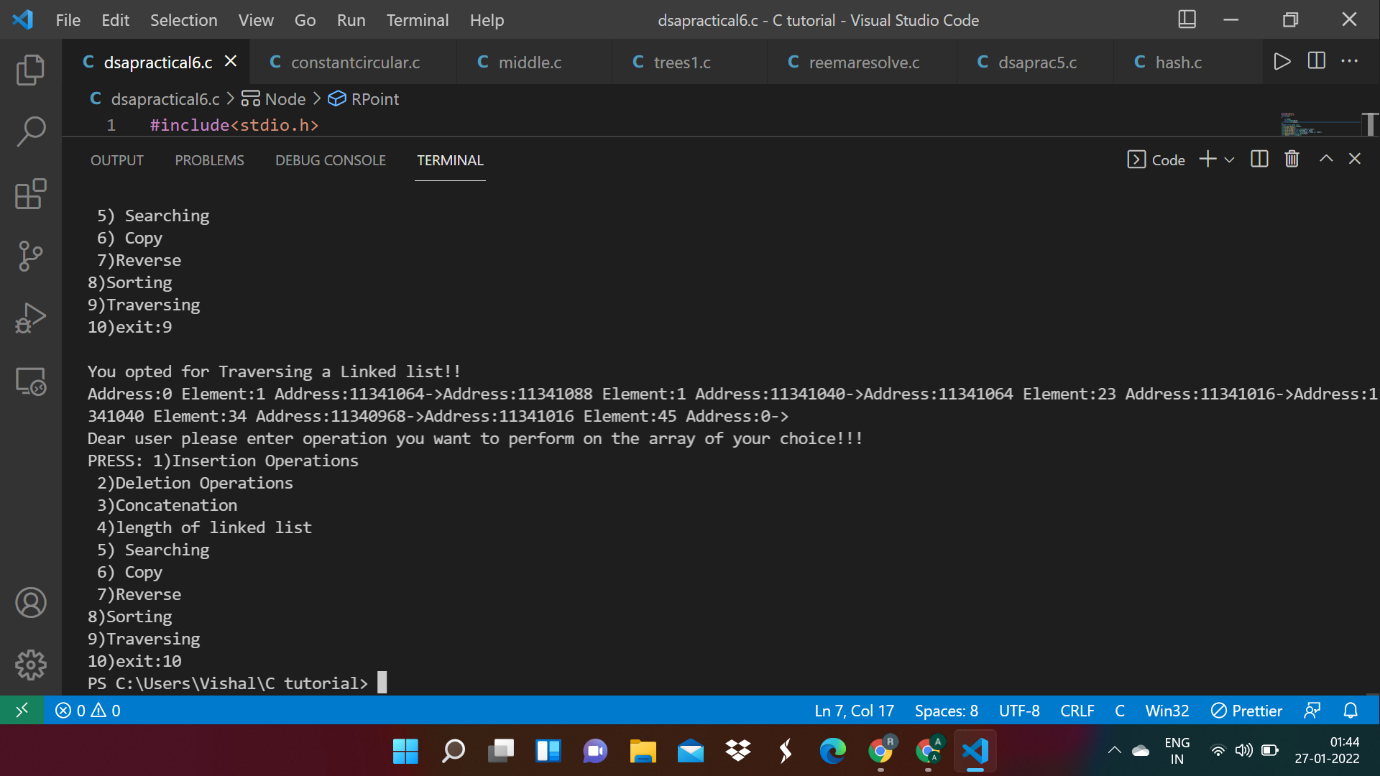












**Conclusion:**

In this experiment, we learned about Doubly Linked Lists and how it is a linear data structure. 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. In a singly linked list, as we need a previous node to delete the given node, sometimes we need to traverse the list to find the previous node. This hits the performance. Insertion operation can be done easily in a doubly linked list when compared to the singly linked list. As the doubly linked list contains one more extra pointer i.e. previous, the memory space taken up by the doubly linked list is larger when compared to the singly linked list. Since two pointers are present i.e. previous and next, all the operations performed on the doubly linked list have to take care of these pointers and maintain them thereby resulting in a performance bottleneck.