Bachelor of Technology (B.Tech)

Department of Computer Science and Engineering

II year I sem- Data Structures
Laboratory Manual

SIDDHARTHA INSTITUTE OF TECHNOLOGY & SCIENCES

(Approved by AICTE, New Delhi & Affiliated to JNTUH,
Hyderabad)Accredited by NBA and NAAC with 'A+' Grade
Narapally, Korremula Road, Ghatkesar, Medchal- Malkajgiri (Dist)-501 301

Vision of the Institute

To be a reputed institute in technical education towards research, industrial and societal needs.

Mission of the Institute

Mission	Statement
IM_1	Provide state-of-the-art infrastructure, review, innovative and experiment teaching –learning methodologies.
IM_2	Promote training, research and consultancy through an integrated institute industry symbiosis
IM_3	Involve in activities to groom professional, ethical values and social responsibility

SIDDHARTHA INSTITUTE OF TECHNOLOGY AND SCIENCES

(*Approved by AICTE, New Delhi &Affiliated to JNTUH, Hyderabad*)
Narapally, Telangana – 500 088.

Department of Computer Science and Engineering

Vision of the Department

To be a recognized center of Computer Science education with values, and quality research

Mission of the Department

Mission	Statement
DM_1	Impart high quality professional training with an emphasis on basic principles of Computer Science and allied Engineering
DM_2	Imbibe social awareness and responsibility to serve the society
DM_3	Provide academic facilities, organize collaborated activities to enable overall development of stakeholders

SIDDHARTHA INSTITUTE OF TECHNOLOGY AND SCIENCES

(*Approved by AICTE, New Delhi &Affiliated to JNTUH, Hyderabad*)
Narapally, Telangana – 500 088.

Department of Computer Science and Engineering

Program Educational Objectives (PEOs)

PEO's	Statement
PEO1	Graduates will be able to solve Computer Science and allied Engineering problems, develop proficiency in computational tools.
PEO2	Graduates will be able to communicate and work efficiently in Multidisciplinary teams with a sense of professional and social responsibility.
PEO3	Graduates will be able to exhibit lifelong learning ability andpursue career as architects, software developers and entrepreneurs.

Programme Outcomes

PO1	**Engineering Knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	**Problem Analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	**Design/development of Solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	**Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
PO6	**The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	**Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental context, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	**Individual and team network:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	**Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	**Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	**Life-Long learning:** Recognize the need for, and have the preparation and able to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes:

PSO1	**Program Applications:** Able to develop programs modules for cloud based applications.
PSO2	**Development Tools:** Able to use tools such as Weka, Rational Rose Raspberry-Pi, Sql and advanced tools

SIDDHARTHA INSTITUTE OF TECHNOLOGY & SCIENCES
(UGC - AUTONOMOUS)

2230578: DATA STRUCTURES LAB
(Common to CSE, AIML, DS, SE, CS, IOT)

B.Tech. II Year I Sem	**L T P C**
	0 0 3 1.5

Prerequisites: A Course on "Programming for problem solving".

Course Objectives:

- It covers various concepts of C programming language
- It introduces searching and sorting algorithms
- It provides an understanding of data structures such as stacks and queues.
- To know how linear data structures work.
- To develop programs for performing operations on Trees and Graphs.

Course Outcomes:

- Ability to develop C programs for computing and real-life applications using basic elements likecontrol statements, arrays
- Ability to develop C programs using functions, pointers and strings.
- Ability to implement data structures like stacks, queues and linked lists.
- Able to write program to implement the trees.
- Ability to Implement searching and sorting algorithms

List of Experiments:

1. Write a program that implements the following sorting methods to sort a given list of integers in ascending order
   i) Quick sort ii) Heap sort iii) Merge sort
2. Write a program that implement stack (its operations) using
   i) Arrays ii) Pointers
3. Write a program that implement Queue (its operations) using
   i) Arrays ii) Pointers
4. Write a program that uses functions to perform the following operations on singly linked list.
   i) Creation ii) Insertion iii) Deletion iv) Traversal
5. Write a program that uses functions to perform the following operations on doubly linked list.
   i) Creation ii) Insertion iii) Deletion iv) Traversal
6. Write a program that uses functions to perform the following operations on circular linked list.
   i) Creation ii) Insertion iii) Deletion iv) Traversal
7. Write a program to implement the tree traversal methods (Recursive and Non Recursive).

8. Write a program to implement
   i) Binary Search tree ii) B Trees iii) B+ Trees iv) AVLtrees
   v) Red - Black trees

9. Write a program to implement the graph traversal methods.
10. Implement a Pattern matching algorithms using Boyer- Moore, Knuth-Morris-Pratt

TEXT BOOKS:

1. Fundamentals of Data Structures in C, 2nd Edition, E. Horowitz, S. Sahni and Susan Anderson Freed, Universities Press.
2. Data Structures using C – A. S. Tanenbaum, Y. Langsam, and M. J. Augenstein, PHI/PearsonEducation.

REFERENCE BOOK:

1. Data Structures: A Pseudocode Approach with C, 2nd Edition, R. F. Gilberg and B. A. Forouzan,Cengage Learning.

List of Experiments

1.Write a program that implements the following sorting methods to sort a given list of integers in ascending order

i)Quick sort ii)Heap sort iii)Merge sort

2.Write a program that implement stack(its operations) using

i)Arrays ii)Pointers

3.Write a program that implement Queue(its operations)using

I)Arrays ii)Pointers

4.Write a program that uses functions to perform the following operations on singly linked list.

i)Creation ii)Insertion iii)Deletion iv)Traversal

5.Write a program that uses functions to perform the following operation on doubly linked list.

i)Creation ii)Insertion iii)Deletion iv)Traversal

6.Write a program that uses functions to perform the following operations on circular linked list.

i)Creation ii)Insertion iii)Deletion iv)Traversal

7. Write a program to implement the tree traversal methods.

8.Write a program to implement

i)Binary Search tree ii)B Trees iii)B+ Trees iv)AVL trees v)Red-Black trees

9.Write a program to implement the graph traversal methods.

10.Implement a Pattern matching algorithms using Boyer-Moore, Knuth-Morris-Pratt

1.Write a program that implements the following sorting methods to sort a given list of integers in ascending order

i)Quick sort ii)Heap sort iii)Merge sort

```
#include <stdio.h>
// Function to swap two elements
void swap(int* a, int* b) {
   int t = *a;
   *a = *b;
   *b = t;
}
int partition(int arr[], int low, int high) {
   int pivot = arr[high];
   int i = (low - 1);

   for (int j = low; j <= high - 1; j++) {
      if (arr[j] < pivot) {
         i++;
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}
void quickSort(int arr[], int low, int high) {
   if (low < high) {
      int pi = partition(arr, low, high);
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
   }
}

// Function to print the array
void printArray(int arr[], int size) {
   int i;
   for (i = 0; i < size; i++)
      printf("%d ", arr[i]);
   printf("\n");
}
```

```c
int main() {
    int arr[] = { 12, 17, 6, 25, 1, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```
Output:
Sorted array:
1 5 6 12 17 25

ii)heap sort

```c
// Heap Sort in C

#include <stdio.h>

// Function to swap the position of two elements

void swap(int* a, int* b)
{

        int temp = *a;
        *a = *b;
        *b = temp;
}

// To heapify a subtree rooted with node i
// which is an index in arr[].
// n is size of heap
void heapify(int arr[], int N, int i)
{
        // Find largest among root,
        // left child and right child

        // Initialize largest as root
        int largest = i;

        // left = 2*i + 1
        int left = 2 * i + 1;

        // right = 2*i + 2
        int right = 2 * i + 2;
```

```
	// If left child is larger than root
	if (left < N && arr[left] > arr[largest])

		largest = left;

	// If right child is larger than largest
	// so far
	if (right < N && arr[right] > arr[largest])

		largest = right;

	// Swap and continue heapifying
	// if root is not largest
	// If largest is not root
	if (largest != i) {

		swap(&arr[i], &arr[largest]);

		// Recursively heapify the affected
		// sub-tree
		heapify(arr, N, largest);
	}
}

// Main function to do heap sort
void heapSort(int arr[], int N)
{

	// Build max heap
	for (int i = N / 2 - 1; i >= 0; i--)

		heapify(arr, N, i);

	// Heap sort
	for (int i = N - 1; i >= 0; i--) {

		swap(&arr[0], &arr[i]);

		// Heapify root element
		// to get highest element at
		// root again
		heapify(arr, i, 0);
	}
}

// A utility function to print array of size n
```

```
void printArray(int arr[], int N)
{
        for (int i = 0; i < N; i++)
                printf("%d ", arr[i]);
        printf("\n");
}

// Driver's code
int main()
{
        int arr[] = { 12, 11, 13, 5, 6, 7 };
        int N = sizeof(arr) / sizeof(arr[0]);

        // Function call
        heapSort(arr, N);
        printf("Sorted array is\n");
        printArray(arr, N);
}
```

Output:
Sorted array is
5 6 7 11 12 13

iii)Merge sort

```
#include <stdio.h>

void merge(int arr[], int left, int mid, int right) {
   int i, j, k;
   int n1 = mid - left + 1;
   int n2 = right - mid;

   int L[n1], R[n2];

   for (i = 0; i < n1; i++)
      L[i] = arr[left + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[mid + 1 + j];

   i = 0;
   j = 0;
   k = left;

   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
         arr[k] = L[i];
         i++;
      }
      else {
```

```
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = { 64, 34, 25, 12, 22, 11, 90 };
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Output:
Original array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90

Q2)i)Write a program that implement Stack (its operations) using Arrays

```
#include<stdio.h>
#define n 5
int stack[n];
int top=-1;
void push()
{
int val;
printf("\nenter value");
scanf("%d",&val);
if(top==n-1)
{
printf("\nstack over flow");
}
else
{
top++;
stack[top]=val;
}
}
void peek()
{

   printf("\nthe top item i.e,peek=%d",stack[top]);
}
void pop()
{
if(top==-1)
{
printf("\nstack is underflow");
}
else
{
top--;

}
}
void display()
{int i;
   i=top;
```

```
    while(i>=0)
    {
      printf("%d\t",stack[i]);
      i--;
    }
}
void main()
{int choice;
while(1)
{
    printf("\nenter \n1.push\n2.pop\n3.peek\n4.display\n5.exit");
    scanf("%d",&choice);
switch(choice)
{
case 1:push();
             break;
case 2:pop();
            break;
case 3:peek();
            break;
case 4:display();
            break;
case 5:exit(1);
           break;
}
}
}
```

Output:

```
enter
1.push
2.pop
3.peek
4.display
5.exit1

enter value22

enter
1.push
2.pop
3.peek
4.display
```

```
5.exit1

enter value25

enter
1.push
2.pop
3.peek
4.display
5.exit1

enter value30

enter
1.push
2.pop
3.peek
4.display
5.exit4
30      25      22
enter
1.push
2.pop
3.peek
4.display
5.exit3

the top item i.e,peek=30
enter
1.push
2.pop
3.peek
4.display
5.exit2

enter
1.push
2.pop
3.peek
4.display
```

```
5.exit4
25      22
enter
1.push
2.pop
3.peek
4.display
5.exit
```

Q2)ii)Write a program that implement Stack (its operations) using Pointers

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

#define MAX 50
int size;
// Defining the stack structure
struct stack {
int arr[MAX];
int top;
};
// Initializing the stack(i.e., top=-1)
void init_stk(struct stack *st) {
st->top = -1;
}
// Entering the elements into stack
void push(struct stack *st, int num) {
if (st->top == size - 1) {
printf("\nStack overflow(i.e., stack full).");
return;
}
st->top++;
st->arr[st->top] = num;
}
//Deleting an element from the stack.
int pop(struct stack *st) {
int num;
if (st->top == -1) {
```

```
printf("\nStack underflow(i.e., stack empty).");
return NULL;
}
num = st->arr[st->top];
st->top--;
return num;
}
void display(struct stack *st) {
int i;
for (i = st->top; i >= 0; i--)
printf("\n%d", st->arr[i]);
}
int main() {
int element, opt, val;
struct stack ptr;
init_stk(&ptr);
printf("\nEnter Stack Size :");
scanf("%d", &size);
while (1) {
printf("\n\nSTACK PRIMITIVE OPERATIONS");
printf("\n1.PUSH");
printf("\n2.POP");
printf("\n3.DISPLAY");
printf("\n4.QUIT");
printf("\n");
printf("\nEnter your option : ");
scanf("%d", &opt);
switch (opt) {
case 1:
printf("\nEnter the element into stack:");
scanf("%d", &val);
push(&ptr, val);
break;
case 2:      element = pop(&ptr);
printf("\nThe element popped from stack is : %d", element);
break;
case 3:
printf("\nThe current stack elements are:");
display(&ptr);
break;
case 4:
exit(0);
default:
printf("\nEnter correct option!Try again.");
}
}
```

```
return (0);
}
```

Output:

```
Enter Stack Size :3


STACK PRIMITIVE OPERATIONS
1.PUSH
2.POP
3.DISPLAY
4.QUIT

Enter your option : 1

Enter the element into stack:11


STACK PRIMITIVE OPERATIONS
1.PUSH
2.POP
3.DISPLAY
4.QUIT

Enter your option : 1

Enter the element into stack:22


STACK PRIMITIVE OPERATIONS
1.PUSH
2.POP
3.DISPLAY
4.QUIT

Enter your option : 1

Enter the element into stack:33


STACK PRIMITIVE OPERATIONS
1.PUSH
2.POP
3.DISPLAY
4.QUIT

Enter your option : 3
```

```
The current stack elements are:
33
22
11

STACK PRIMITIVE OPERATIONS
1.PUSH
2.POP
3.DISPLAY
4.QUIT

Enter your option : 2

The element popped from stack is : 33

STACK PRIMITIVE OPERATIONS
1.PUSH
2.POP
3.DISPLAY
4.QUIT

Enter your option : 3

The current stack elements are:
22
11

STACK PRIMITIVE OPERATIONS
1.PUSH
2.POP
3.DISPLAY
4.QUIT

Enter your option : 4
```

Q3)i) Write a program that implement Queue (its operations) using Arrays

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5
int queue[SIZE], front = -1, rear = -1;
void enQueue()
{
   int value;
```

```
    printf("enter value");
    scanf("%d",&value);
if(rear == SIZE-1)
{
printf("\nQueue is Full!!! Insertion is not possible!!!");
}
else if(rear == -1)
{
front = 0;
rear++;
queue[rear] = value;
printf("\nInsertion success!!!");
}
else{
    rear++;
    queue[rear]=value;

}
}
void deQueue(){
if(front == rear)
front = rear = -1;
else if(front == -1)
printf("\nQueue is Empty!!! Deletion is not possible!!!");
else{
front++;
}
}
void display(){
if(rear == -1)
printf("\nQueue is Empty!!!");
else{
int i;
printf("\nQueue elements are:\n");
for(i=front; i<=rear; i++)
printf("%d\t",queue[i]);
}
}

void main()
{
int value, choice;
while(1){
printf("\n\n***** MENU *****\n");
printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");
printf("\nEnter your choice: ");
```

```
scanf("%d",&choice);
switch(choice){
case 1: enQueue();
break;
case 2: deQueue();
break;
case 3: display();
break;
case 4: exit(0);
default: printf("\nWrong selection!!! Try again!!!");
}
}
}
```

Output:

```
***** MENU *****

1. Insertion

2. Deletion

3. Display

4. Exit

Enter your choice: 1

enter value11


Insertion success!!!


***** MENU *****

1. Insertion

2. Deletion

3. Display

4. Exit

Enter your choice: 1

enter value22




***** MENU *****

1. Insertion
```

```
2. Deletion
3. Display
4. Exit
Enter your choice: 1
enter value33


***** MENU *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 3

Queue elements are:
11      22      33

***** MENU *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 2


***** MENU *****
1. Insertion
2. Deletion
3. Display
4. Exit
```

```
Enter your choice: 3


Queue elements are:
22      33


***** MENU *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 4
```

Q3ii) Aim: Write a program that implement Queue (its operations) using Pointers

```
#include<stdlib.h>
struct q
{
int no;
struct q *next;
}
*start=NULL;
void add();
int del();
void display();
void main()
{
int ch;
char choice;
while(1)
{
printf(" \n MENU \n");
printf("\n1.Insert an element in Queue\n");
printf("\n2.Delete an element from Queue\n");
printf("\n3.Display the Queue\n");
printf("\n4.Exit!\n");
printf("\nEnter your choice:");
scanf("%d",&ch);
```

```
switch(ch)
{
case 1:add();
break;
case 2:
printf("\nThe deleted element is=%d",del());
break;
case 3:display();
getch();
break;
case 4:exit(0);
break;
default:printf("\nYou entered wrong choice");
getch();
break;
}
}
getch();
}
void add()
{
struct q *p,*temp;
temp=start;
p=(struct q*)malloc(sizeof(struct q));
printf("\nEnter the element:");
scanf("%d",&p->no);
p->next=NULL;
if(start==NULL)
{
start=p;
}
else
{
while(temp->next!=NULL)
{
temp=temp->next;
}
temp->next=p;
}
}
int del()
{
struct q *temp;
int value;
if(start==NULL)
{
```

```
printf("\nQueue is Empty");
getch();
return(0);
}
else
{
temp=start;
value=temp->no;
start=start->next;
free(temp);
}
return(value);
}
void display()
{
struct q *temp;
temp=start;
if(temp==NULL)
printf("queue is empty");
else
{
while(temp->next!=NULL)
{
printf("\nno=%d",temp->no);
temp=temp->next;
}
printf("\nno=%d",temp->no);
}
getch();
}
```

Output:

```
MENU

1.Insert an element in Queue

2.Delete an element from Queue

3.Display the Queue

4.Exit!

Enter your choice:1
```

```
Enter the element:11

 MENU

1.Insert an element in Queue

2.Delete an element from Queue

3.Display the Queue

4.Exit!

Enter your choice:1

Enter the element:22

 MENU

1.Insert an element in Queue

2.Delete an element from Queue

3.Display the Queue

4.Exit!

Enter your choice:1

Enter the element:33

 MENU

1.Insert an element in Queue

2.Delete an element from Queue

3.Display the Queue

4.Exit!

Enter your choice:3

no=11
no=22
no=33
```

```
 MENU

1.Insert an element in Queue

2.Delete an element from Queue

3.Display the Queue

4.Exit!

Enter your choice:3

no=11
no=22
no=33
 MENU

1.Insert an element in Queue

2.Delete an element from Queue

3.Display the Queue

4.Exit!

Enter your choice:2

The deleted element is=11
 MENU

1.Insert an element in Queue

2.Delete an element from Queue

3.Display the Queue

4.Exit!

Enter your choice:3

no=22
no=33
```

Q4) Write a program that uses functions to perform the following operations on singly linked list.

I)Creation ii)Insertion iii)Deletion iv)Traversal

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int count=0;
struct node
{
int data;
struct node *next;
}*head,*newn,*trav;
//-------------------------------------------------------
void create_list()
{
int value;
struct node *temp;
temp=head;
newn=(struct node *)malloc(sizeof (struct node));
printf("\nenter the value to be inserted");
scanf("%d",&value);
newn->data=value;
if(head==NULL)
{
head=newn;
head->next=NULL;
count++;
}
else
{
while(temp->next!=NULL)
{
temp=temp->next;
}
temp->next=newn;
newn->next=NULL;
count++;
}
}
//--------------------------------------------------
void insert_at_begning(int value)
{
newn=(struct node *)malloc(sizeof (struct node));
```

```
newn->data=value;
if(head==NULL)
{
head=newn;
head->next=NULL;
count++;
}
else
{
newn->next=head;
head=newn;
count++;
}
}
//------------------------------------------------------------
void insert_at_end(int value)
{
struct node *temp;
temp=head;
newn=(struct node *)malloc(sizeof (struct node));
newn->data=value;
if(head==NULL)
{
head=newn;
head->next=NULL;
count++;
}
else
{
while(temp->next!=NULL)
{
temp=temp->next;
}
temp->next=newn;
newn->next=NULL;
count++;
}
}
//--------------------------------------------------------
int insert_at_middle()
{
if(count>=2)
{
struct node *var1,*temp;
int loc,value;
printf("\n after which value you want to insert : ");
```

```
scanf("%d",&loc);
printf("\nenter the value to be inserted");
scanf("%d",&value);
newn=(struct node *)malloc(sizeof (struct node));
newn->data=value;
temp=head;

/* if(head==NULL)
{
head=newn;
head->next=NULL;
count++;
return 0;
}
else
{*/
while(temp->data!=loc)
{
temp=temp->next;
if(temp==NULL)
{
printf("\nSORRY...there is no %d element",loc);
return 0;
}
}
//var1=temp->next;
newn->next=temp->next;//var1;
temp->next=newn;

count++;
//}
}
else
{
printf("\nthe no of nodes must be >=2");
}
}
//---------------------------------------------------------------
int delete_from_middle()
{
if(count==0)
printf("\n List is Empty!!!! you can't delete elements\n");
else if(count>2)
{
struct node *temp,*var;
int value;
```

```
temp=head;
printf("\nenter the data that you want to delete from the list shown above");
scanf("%d",&value);
while(temp->data!=value)
{
var=temp;
temp=temp->next;
if(temp==NULL)
{
printf("\nSORRY...there is no %d element",value);
return 0;
}
}
if(temp==head)
{
head=temp->next;

}
else{
var->next=temp->next;
temp->next=NULL;

}
count--;
if(temp==NULL)
printf("Element is not avilable in the list \n**enter only middle elements..**");
else
printf("\ndata deleted from list is %d",value);
free(temp);
}
else
{
printf("\nthere no middle elemts..only %d elemts is avilable\n",count);
}
}
//----------------------------------------------------------------
int delete_from_front()
{
struct node *temp;
temp=head;
if(head==NULL)
{
printf("\nno elements for deletion in the list\n");
return 0;
}
else
```

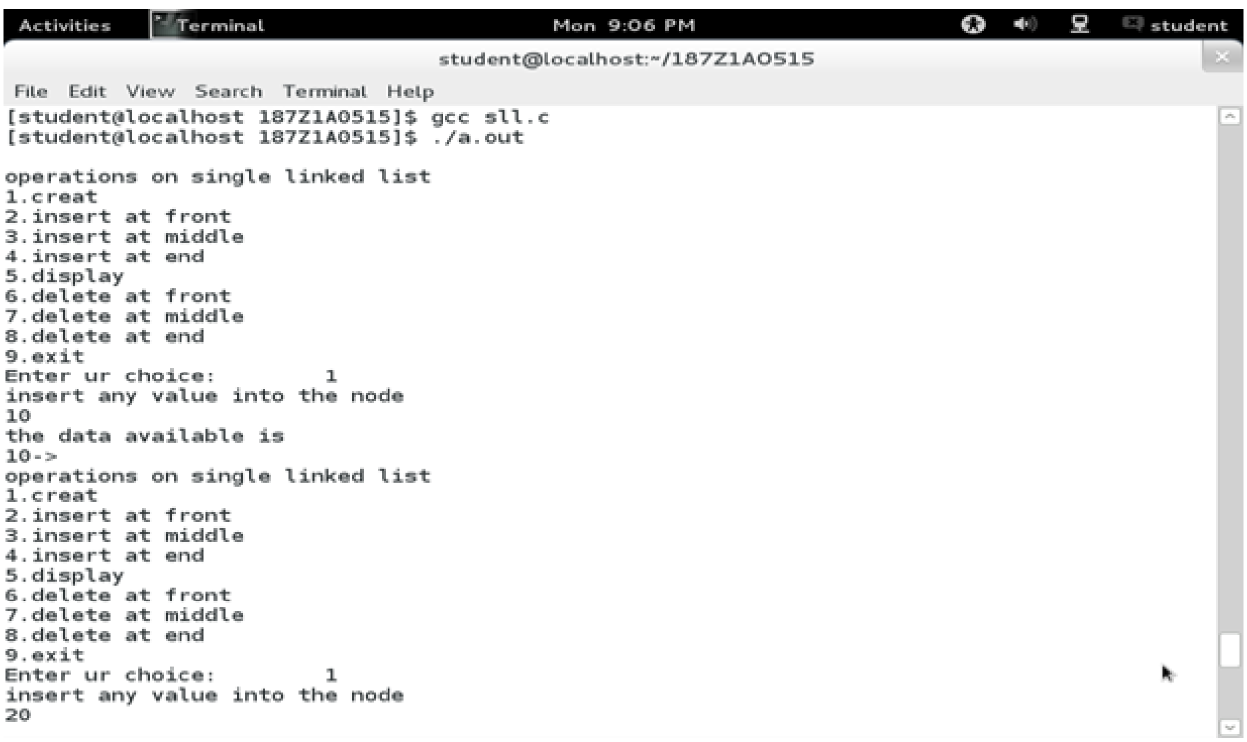
```
{
printf("\ndeleted element is :%d",head->data);
if(temp->next==NULL)
{
head=NULL;
}
else{
head=temp->next;
temp->next=NULL;

}
count--;
free(temp);

}
}
//---------------------------------------------------------
int delete_from_end()
{
struct node *temp,*var;
temp=head;
if(head==NULL)
{
printf("\nno elemts in the list");
return 0;
}
else{
if(temp->next==NULL )
{
head=NULL;//temp->next;
}
else{
while(temp->next != NULL)
{
var=temp;
temp=temp->next;
}
var->next=NULL;
}
printf("\ndata deleted from list is %d",temp->data);
free(temp);
count--;
}
return 0;
}
//-------------------------------------------------------
```

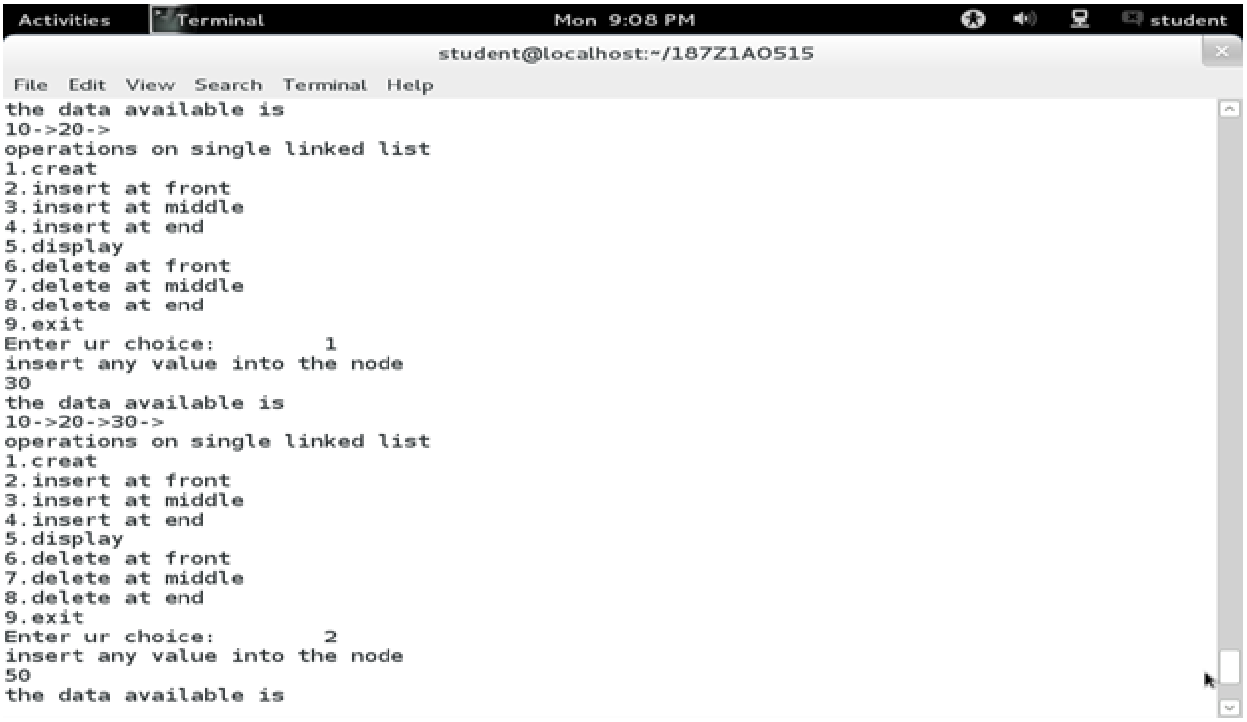
```
int display()
{
trav=head;
if(trav==NULL)
{
printf("\nList is Empty\n");
return 0;
}
else
{
printf("\n\nElements in the Single Linked List is %d:\n",count);
while(trav!=NULL)
{
printf(" -> %d ",trav->data);
trav=trav->next;
}
printf("\n");
}
}
//-------------------------------------------------------------
int main()
{
int ch=0;
char ch1;

head=NULL;
while(1)
{
printf("\n1.create linked list");
printf("\n2.insertion at begning of linked list");
printf("\n3.insertion at the end of linked list");
printf("\n4.insertion at the middle where you want");
printf("\n5.deletion from the front of linked list");
printf("\n6.deletion from the end of linked list ");
printf("\n7.deletion of the middle data that you want");
printf("\n8.display the linked list");
printf("\n9.exit\n");
printf("\nenter the choice of operation to perform on linked list");
scanf("%d",&ch);
switch(ch)
{
case 1:
{
do{
create_list();
display();
```

```
printf("do you want to create list ,y / n");
getchar();
scanf("%c",&ch1);
}while(ch1=='y');

break;
}
case 2:
{
int value;
printf("\nenter the value to be inserted");
scanf("%d",&value);
insert_at_begning(value);
display();
break;
}
case 3:
{
int value;
printf("\nenter value to be inserted");
scanf("%d",&value);
insert_at_end(value);
display();
break;
}
case 4:
{
insert_at_middle();
display();
break;
}
case 5:
{
delete_from_front();
display();
}break;
case 6:
{
delete_from_end();
display();
break;
}
case 7:
{
display();
delete_from_middle();
```

```
display();
break;
}
case 8:
{
display();
break;
}
case 9:
{
exit(1);
}
default:printf("\n****Please enter correct choice****\n");
}
}
getch();
}
```

OUTPUT:






```
the data available is
10->20->
operations on single linked list
1.creat
2.insert at front
3.insert at middle
4.insert at end
5.display
6.delete at front
7.delete at middle
8.delete at end
9.exit
Enter ur choice:        1
insert any value into the node
30
the data available is
10->20->30->
operations on single linked list
1.creat
2.insert at front
3.insert at middle
4.insert at end
5.display
6.delete at front
7.delete at middle
8.delete at end
9.exit
Enter ur choice:        2
insert any value into the node
50
the data available is
```

Q5)Write a program that uses functions to perform the following operations on Doubly Linked List.:

i) Creation ii) Insertion iii) Deletion iv) Traversal

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int count=0;
struct node
{
int data;
struct node *next,*prev;
}*head,*last,*newn,*trav;
//---------------------------------------------------
void create_list()
{
struct node *temp;
int value;
temp=last;
newn=(struct node *)malloc(sizeof (struct node));
printf("\n enter value");
scanf("%d",&value);
newn->data=value;
if(last==NULL)
{
head=last=newn;
```

```
head->prev=NULL;
head->next=NULL;
count++;
}
else
{
newn->next=NULL;
newn->prev=last;
last->next=newn;
last=newn;
count++;
}
}
//---------------------------------------------------
void insert_at_begning(int value)
{
newn=(struct node *)malloc(sizeof (struct node));
newn->data=value;
if(head==NULL)
{
head=last=newn;
head->prev=NULL;
head->next=NULL;
count++;
}
else
{
newn->next=head;
head->prev=newn;
newn->prev=NULL;
head=newn;
count++;
}
}
//--------------------------------------------------------
void insert_at_end(int value)
{
struct node *temp;
temp=last;
newn=(struct node *)malloc(sizeof (struct node));
newn->data=value;
if(last==NULL)
{
head=last=newn;
head->prev=NULL;
head->next=NULL;
count++;
}
else
{
newn->next=NULL;
```

```
newn->prev=last;
last->next=newn;
last=newn;
count++;
}
}
//----------------------------------------------------
int insert_at_middle()
{
if(count>=2)
{
struct node *var2,*temp;
int loc,value;
printf("\nselect location where you want to insert the data");
scanf("%d",&loc);
printf("\nenter which value do u want to inserted");
scanf("%d",&value);
newn=(struct node *)malloc(sizeof (struct node));
newn->data=value;
temp=head;
while(temp->data!=loc)
{
temp=temp->next;
if(temp==NULL)
{
printf("\nSORRY...there is no %d element",loc);
return 0;
}
}
if(temp->next==NULL)
{
printf("\n%d is last node..please enter midddle node values ",loc) ;
return;
}
var2=temp->next;
temp->next=newn;
newn->next=var2;
newn->prev=temp;
var2->prev=newn;
count++;
}
else
{
printf("\nthe no of nodes must be >=2");
}
}
//-------------------------------------------------------------
int delete_from_middle()
{
if(count>2)
{
```

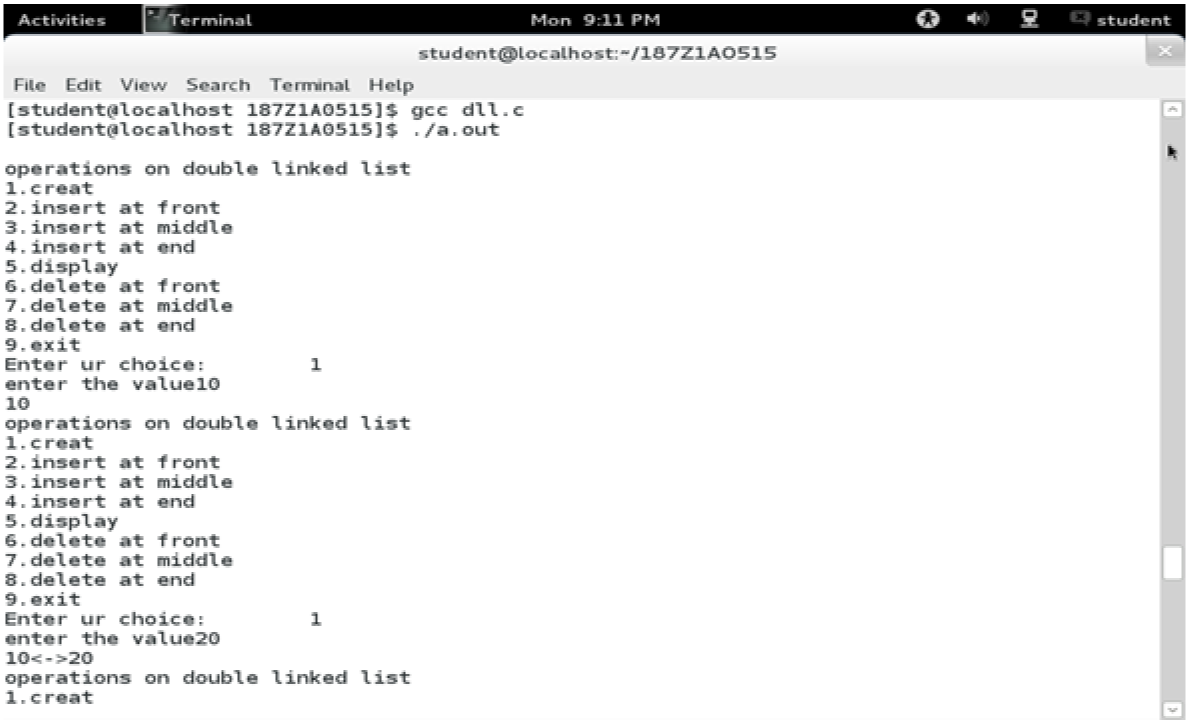
```
struct node *temp,*var;
int value;
temp=head;
printf("\nenter the data that you want to delete from the list shown above");
scanf("%d",&value);
while(temp!=NULL)
{
if(temp->data == value)
{
if(temp->next==NULL)//if(temp==head)
{
printf("\n\n sorry %d is last node ..please enter middle nodes only",value);
return 0;
}
else
{
if(temp==head)
{
printf("\n\n %d is first node..plz enter middle nodes",value);
return;

}
var->next=temp->next;
temp->next->prev=var;
free(temp);
count--;
return 0;
}
}
else
{
var=temp;
temp=temp->next;
}
}
if(temp==NULL)
printf("\n%d is not avilable.. enter only middle elements..",value);
else
printf("\ndata deleted from list is %d",value);
}
else
{
printf("\nthere no middle elemts..only %d elemts is avilable",count);
}
}
//----------------------------------------------------------------
int delete_from_front()
{
struct node *temp;
if(head==NULL)
{
```

```
printf("no elements for deletion in the list");
return 0;
}
else if(head->next==NULL)
{
printf("deleted element is :%d",head->data);
head=last=NULL;

}
else
{
temp=head->next;
printf("deleted element is :%d",head->data);
head->next=NULL;
temp->prev=NULL;
head=temp;
count--;
return 0;
}
}
//----------------------------------------------------------
int delete_from_end()
{
struct node *temp,*var;
temp=last;
if(last==NULL)
{
printf("no elemts in the list");
return 0;
}
else if(last->prev==NULL)
{
printf("data deleted from list is %d",last->data);
head=last=NULL;
//free(last);
count--;
return 0;
}
else{
printf("data deleted from list is %d",last->data);
var=last->prev;
last->prev->next=NULL;
last=var;
count--;
return 0;
}
}
//--------------------------------------------------------
int display()
{
trav=last;//head;
```

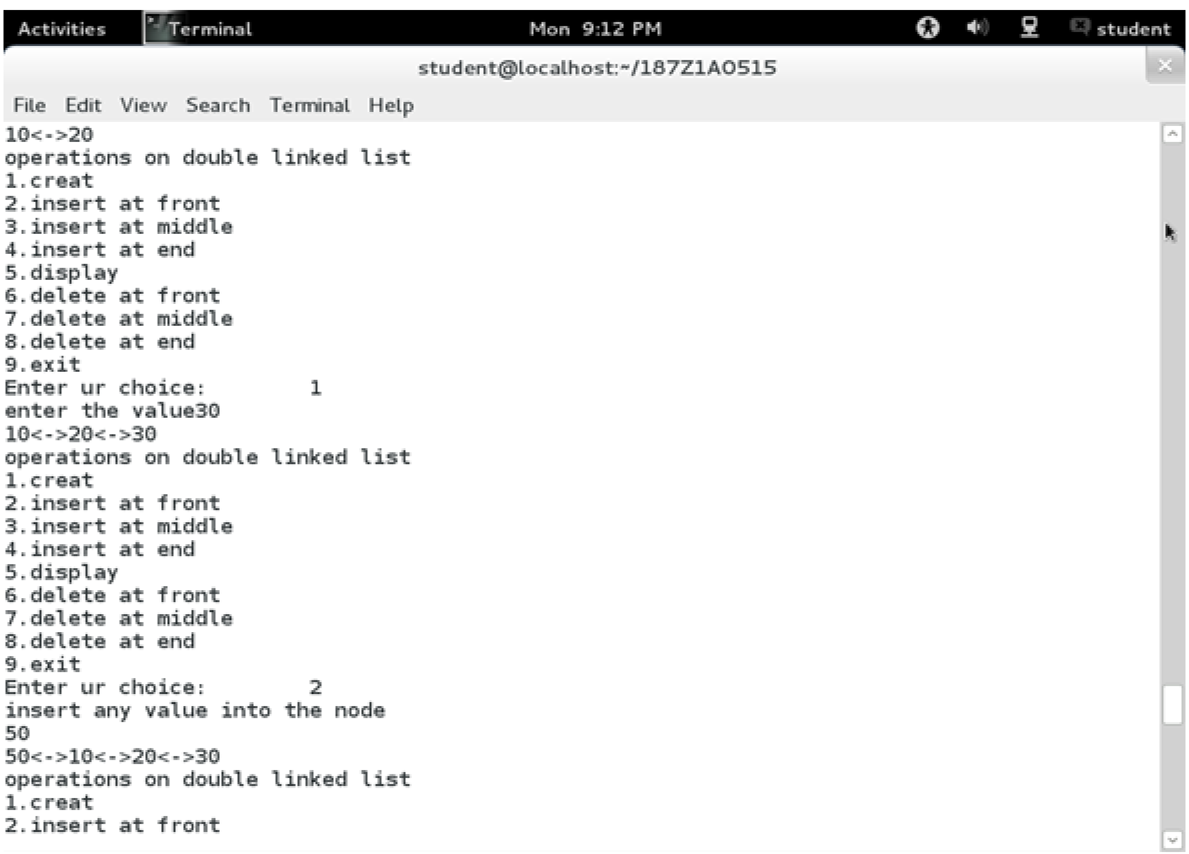
```
if(trav==NULL)
{
printf("\nList is Empty");
return 0;
}
else
{
printf("\n\nElements in the List is %d:\n",count);
while(trav!=NULL)
{
printf("%d<--> ",trav->data);
trav=trav->prev;//next;
}
printf("\n");
}
}
//------------------------------------------------------------
int main()
{
int ch=0;
char ch1;
// clrscr();
head=NULL;
last=NULL;
while(1)
{
printf("\n Double Linked List Operations");
printf("\n1.Create Double Linked List");
printf("\n2.insertion at begning of linked list");
printf("\n3.insertion at the end of linked list");
printf("\n4.insertion at the middle where you want");
printf("\n5.deletion from the front of linked list");
printf("\n6.deletion from the end of linked list ");
printf("\n7.deletion of the middle data that you want");
printf("\n8.display");
printf("\n9.exit\n");
printf("\nenter the choice of operation to perform on linked list");
scanf("%d",&ch);
switch(ch)
{
case 1:
{
do{
create_list();
display();
printf("do you want to create list ,y / n");
getchar();
scanf("%c",&ch1);
}while(ch1=='y');
break;
}
```

```
case 2:
{
int value;
printf("\nenter the value to be inserted");
scanf("%d",&value);
insert_at_begning(value);
display();
break;
}
case 3:
{
int value;
printf("\nenter value to be inserted");
scanf("%d",&value);
insert_at_end(value);
display();
break;
}
case 4:
{
insert_at_middle();
display();
break;
}
case 5:
{
delete_from_front();
display();
}break;
case 6:
{
delete_from_end();
display();
break;
}
case 7:
{
display();
delete_from_middle();
display();
break;
}
case 8:display();break;
case 9:
{
exit(0);
}
}
}
getch();
}
```

Output:



```
Activities   Terminal   Mon 9:11 PM   student
student@localhost:~/187Z1A0515
File Edit View Search Terminal Help
[student@localhost 187Z1A0515]$ gcc dll.c
[student@localhost 187Z1A0515]$ ./a.out

operations on double linked list
1.creat
2.insert at front
3.insert at middle
4.insert at end
5.display
6.delete at front
7.delete at middle
8.delete at end
9.exit
Enter ur choice:        1
enter the value10
10
operations on double linked list
1.creat
2.insert at front
3.insert at middle
4.insert at end
5.display
6.delete at front
7.delete at middle
8.delete at end
9.exit
Enter ur choice:        1
enter the value20
10<->20
operations on double linked list
1.creat
```



```
Activities   Terminal   Mon 9:12 PM   student
student@localhost:~/187Z1A0515
File Edit View Search Terminal Help
10<->20
operations on double linked list
1.creat
2.insert at front
3.insert at middle
4.insert at end
5.display
6.delete at front
7.delete at middle
8.delete at end
9.exit
Enter ur choice:        1
enter the value30
10<->20<->30
operations on double linked list
1.creat
2.insert at front
3.insert at middle
4.insert at end
5.display
6.delete at front
7.delete at middle
8.delete at end
9.exit
Enter ur choice:        2
insert any value into the node
50
50<->10<->20<->30
operations on double linked list
1.creat
2.insert at front
```

Q6) Write a program that uses functions to perform the following operations on circular linked list.

i)Creation   ii)Insertion   iii)Deletion   iv)Traversal

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *head;
void beginsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();
void main ()
{
int choice =0;
while(choice != 7)
{
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.Delete from Beginning\n4.Delete from
last\n5.Search for an element\n6.Show\n7.Exit\n");
printf("\nEnter your choice?\n");
scanf("\n%d",&choice);
switch(choice)
{
case 1:
beginsert();
break;
case 2:
lastinsert();
break;
case 3:
begin_delete();
break;
case 4:
last_delete();
break;
case 5:
search();
break;
case 6:
```

```
display();
break;
case 7:
exit(0);
break;
default:
printf("Please enter valid choice..");
}
}
}
void beginsert()
{
struct node *ptr,*temp;
int item;
ptr = (struct node *)malloc(sizeof(struct node));
if(ptr == NULL)
{
printf("\nOVERFLOW");
}
else
{
printf("\nEnter the node data?");
scanf("%d",&item);
ptr -> data = item;
if(head == NULL)
{
head = ptr;
ptr -> next = head;
}
else
{
temp = head;
while(temp->next != head)
temp = temp->next;
ptr->next = head;
temp -> next = ptr;
head = ptr;
}
printf("\nnode inserted\n");
}
}
void lastinsert()
{
struct node *ptr,*temp;
int item;
ptr = (struct node *)malloc(sizeof(struct node));
if(ptr == NULL)
{
printf("\nOVERFLOW\n");
}
else
```

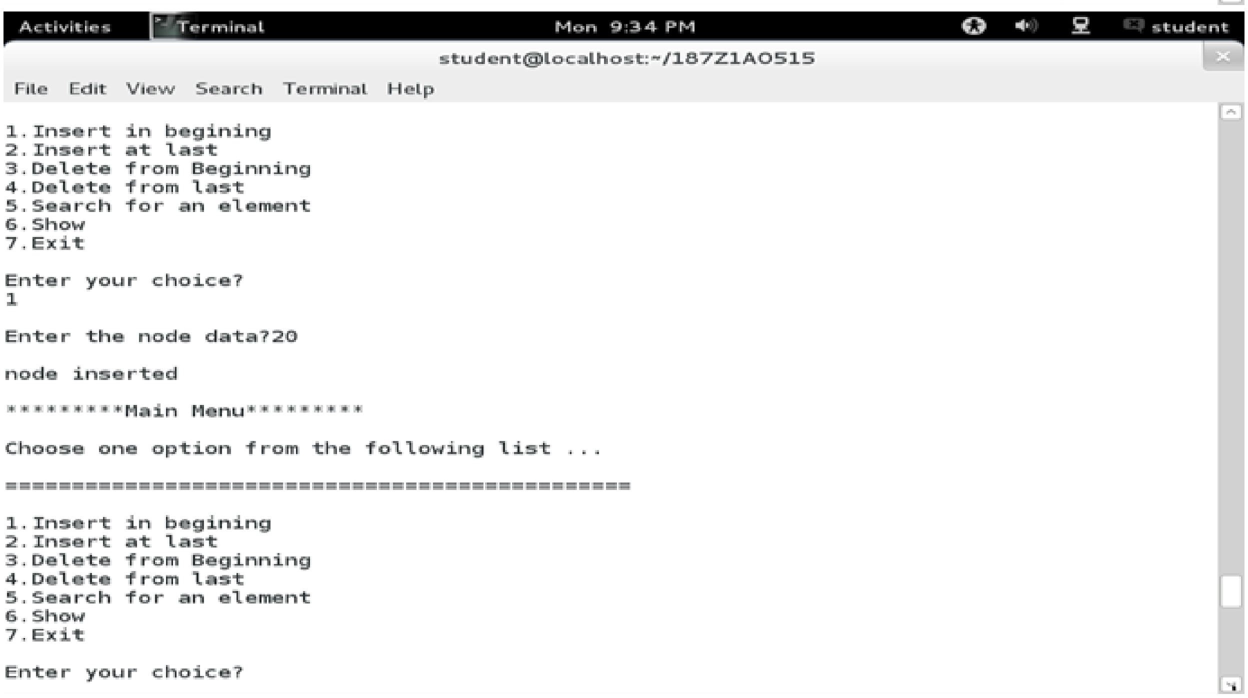
```
{
printf("\nEnter Data?");
scanf("%d",&item);
ptr->data = item;
if(head == NULL)
{
head = ptr;
ptr -> next = head;
}
else
{
temp = head;
while(temp -> next != head)
{
temp = temp -> next;
}
temp -> next = ptr;
ptr -> next = head;
}

printf("\nnode inserted\n");
}
}

void begin_delete()
{
struct node *ptr;
if(head == NULL)
{
printf("\nUNDERFLOW");
}
else if(head->next == head)
{
head = NULL;
free(head);
printf("\nnode deleted\n");
}
else
{   ptr = head;
while(ptr -> next != head)
ptr = ptr -> next;
ptr->next = head->next;
free(head);
head = ptr->next;
printf("\nnode deleted\n");
}
}
void last_delete()
{
struct node *ptr, *preptr;
if(head==NULL)
```

```
{
printf("\nUNDERFLOW");
}
else if (head ->next == head)
{
head = NULL;
free(head);
printf("\nnode deleted\n");
}
else
{
ptr = head;
while(ptr ->next != head)
{
preptr=ptr;
ptr = ptr->next;
}
preptr->next = ptr -> next;
free(ptr);
printf("\nnode deleted\n");
}
}
void search()
{
struct node *ptr;
int item,i=0,flag=1;
ptr = head;
if(ptr == NULL)
{
printf("\nEmpty List\n");
}
else
{
printf("\nEnter item which you want to search?\n");
scanf("%d",&item);
if(head ->data == item)
{
printf("item found at location %d",i+1);
flag=0;
}
else
{
while (ptr->next != head)
{
if(ptr->data == item)
{
printf("item found at location %d ",i+1);
flag=0;
break;
}
else
```

```
{
flag=1;
}
i++;
ptr = ptr -> next;
}
}
if(flag != 0)
{
printf("Item not found\n");
} } }

void display()
{
struct node *ptr;
ptr=head;
if(head == NULL)
{
printf("\nnothing to print");
}
else
{
printf("\n printing values ... \n");
while(ptr -> next != head)
{
printf("%d\n", ptr -> data);
ptr = ptr -> next;
}
printf("%d\n", ptr -> data);
}
}
```

Output:

```
Activities   Terminal   Mon 9:33 PM   student
student@localhost:~/187Z1A0515
File  Edit  View  Search  Terminal  Help
[student@localhost 187Z1A0515]$ gcc CLL.c
[student@localhost 187Z1A0515]$ ./a.out

*********Main Menu*********

Choose one option from the following list ...

===============================================

1.Insert in begining
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search for an element
6.Show
7.Exit

Enter your choice?
1

Enter the node data?10

node inserted

*********Main Menu*********

Choose one option from the following list ...

===============================================

1.Insert in begining
```



```
Activities   Terminal   Mon 9:34 PM   student
student@localhost:~/187Z1A0515
File  Edit  View  Search  Terminal  Help

1.Insert in begining
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search for an element
6.Show
7.Exit

Enter your choice?
1

Enter the node data?20

node inserted

*********Main Menu*********

Choose one option from the following list ...

===============================================

1.Insert in begining
2.Insert at last
3.Delete from Beginning
4.Delete from last
5.Search for an element
6.Show
7.Exit

Enter your choice?
```

Q7)Write a program to implement the tree traversal methods.

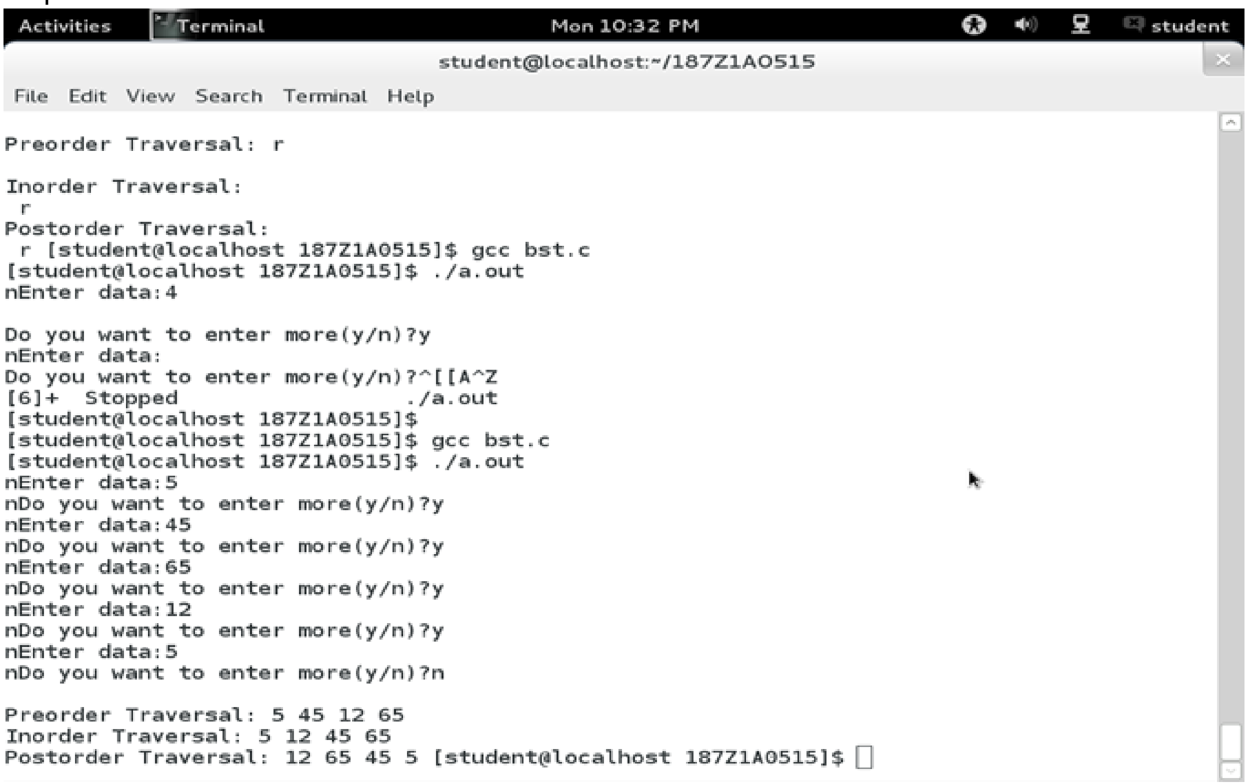
```
#include<stdio.h>
#include<stdlib.h>
typedef struct BST
{
int data;
struct BST *left;
```

```
struct BST *right;
}node;

node *create();
void insert(node *,node *);
void preorder(node *);
void inorder(node *);
void postorder(node *);

int main()
{
char ch;
node *root=NULL,*temp;

do
{
temp=create();
if(root==NULL)
root=temp;
else
insert(root,temp);
printf("nDo you want to enter more(y/n)?");
getchar();
scanf("%c",&ch);
}while(ch=='y'|ch=='Y');

printf("\nPreorder Traversal: ");
preorder(root);

printf("\nInorder Traversal: ");
inorder(root);

printf("\nPostorder Traversal: ");
postorder(root);
return 0;
}

node *create()
{
node *temp;
printf("nEnter data:");
temp=(node*)malloc(sizeof(node));
scanf("%d",&temp->data);
temp->left=temp->right=NULL;
return temp;
}
```

```
void insert(node *root,node *temp)
{
if(temp->data<root->data)
{
if(root->left!=NULL)
insert(root->left,temp);
else
root->left=temp;
}

if(temp->data>root->data)
{
if(root->right!=NULL)
insert(root->right,temp);
else
root->right=temp;
}
}

void preorder(node *root)
{
if(root!=NULL)
{
printf("%d ",root->data);
preorder(root->left);
preorder(root->right);
}
}

void inorder(node *root)
{
if(root!=NULL)
{
inorder(root->left);
printf("%d ",root->data);
inorder(root->right);
}
}

void postorder(node *root)
{
if(root!=NULL)
{
postorder(root->left);
postorder(root->right);
```

```
printf("%d ",root->data);

}
}
```

Output:



8.Write a program to implement

i)Binary Search tree

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

```
struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insert(root->left, data);
    } else if (data > root->data) {
        root->right = insert(root->right, data);
    }
    return root;
}

void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}
void preorderTraversal(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}

void postorderTraversal(struct Node* root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->data);
    }
}
//find min value
struct Node* findMin(struct Node* root) {
    while (root->left != NULL) {
        root = root->left;
    }
    return root;
}

struct Node* deleteNode(struct Node* root, int data) {
    if (root == NULL) {
        return root;
```

```
    } else if (data < root->data) {
        root->left = deleteNode(root->left, data);
    } else if (data > root->data) {
        root->right = deleteNode(root->right, data);
    } else {
        // Case 1: No child or only one child
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        // Case 2: Two children
        struct Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
int main() {
    struct Node* root = NULL;
    root = insert(root, 100);
    insert(root, 20);
    insert(root, 200);
    insert(root, 10);
    insert(root, 30);
    insert(root, 150);
    insert(root, 300);
     printf("Pre-order traversal of the binary search tree: ");
    preorderTraversal(root);
    printf("\n");

    printf("in-order traversal of the binary search tree: ");
    inorderTraversal(root);
    printf("\n");

    printf("Post-order traversal of the binary search tree: ");
    postorderTraversal(root);
    printf("\n");

     int key = 150; // Key to be deleted
    root = deleteNode(root, key);
```

```
  printf("\nBinary Search Tree after deletion of %d:\n", key);
  printf("in-order traversal of the binary search tree: ");
  inorderTraversal(root);
  return 0;
}
```

Output:

```
Pre-order traversal of the binary search tree: 100 20 10 30 200 150 300
in-order traversal of the binary search tree: 10 20 30 100 150 200 300
Post-order traversal of the binary search tree: 10 30 20 150 300 200 100

Binary Search Tree after deletion of 150:
in-order traversal of the binary search tree: 10 20 30 100 200 300
```

ii)B Trees

```
// Searching a key on a B-tree in C

#include <stdio.h>
#include <stdlib.h>

#define MAX 3
#define MIN 2

struct BTreeNode {
  int val[MAX + 1], count;
  struct BTreeNode *link[MAX + 1];
};

struct BTreeNode *root;

// Create a node
struct BTreeNode *createNode(int val, struct BTreeNode *child) {
  struct BTreeNode *newNode;
  newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
  newNode->val[1] = val;
  newNode->count = 1;
  newNode->link[0] = root;
  newNode->link[1] = child;
  return newNode;
}
```

```
// Insert node
void insertNode(int val, int pos, struct BTreeNode *node,
        struct BTreeNode *child) {
  int j = node->count;
  while (j > pos) {
    node->val[j + 1] = node->val[j];
    node->link[j + 1] = node->link[j];
    j--;
  }
  node->val[j + 1] = val;
  node->link[j + 1] = child;
  node->count++;
}

// Split node
void splitNode(int val, int *pval, int pos, struct BTreeNode *node,
         struct BTreeNode *child, struct BTreeNode **newNode) {
  int median, j;

  if (pos > MIN)
    median = MIN + 1;
  else
    median = MIN;

  *newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
  j = median + 1;
  while (j <= MAX) {
    (*newNode)->val[j - median] = node->val[j];
    (*newNode)->link[j - median] = node->link[j];
    j++;
  }
  node->count = median;
  (*newNode)->count = MAX - median;

  if (pos <= MIN) {
    insertNode(val, pos, node, child);
  } else {
    insertNode(val, pos - median, *newNode, child);
  }
  *pval = node->val[node->count];
  (*newNode)->link[0] = node->link[node->count];
  node->count--;
}

// Set the value
int setValue(int val, int *pval,
```

```
        struct BTreeNode *node, struct BTreeNode **child) {
  int pos;
  if (!node) {
    *pval = val;
    *child = NULL;
    return 1;
  }

  if (val < node->val[1]) {
    pos = 0;
  } else {
    for (pos = node->count;
       (val < node->val[pos] && pos > 1); pos--)
      ;
    if (val == node->val[pos]) {
      printf("Duplicates are not permitted\n");
      return 0;
    }
  }
  if (setValue(val, pval, node->link[pos], child)) {
    if (node->count < MAX) {
      insertNode(*pval, pos, node, *child);
    } else {
      splitNode(*pval, pval, pos, node, *child, child);
      return 1;
    }
  }
  return 0;
}

// Insert the value
void insert(int val) {
  int flag, i;
  struct BTreeNode *child;

  flag = setValue(val, &i, root, &child);
  if (flag)
    root = createNode(i, child);
}

// Search node
void search(int val, int *pos, struct BTreeNode *myNode) {
  if (!myNode) {
    return;
  }
```

```
  if (val < myNode->val[1]) {
    *pos = 0;
  } else {
    for (*pos = myNode->count;
       (val < myNode->val[*pos] && *pos > 1); (*pos)--)
      ;
    if (val == myNode->val[*pos]) {
      printf("%d is found", val);
      return;
    }
  }
  search(val, pos, myNode->link[*pos]);

  return;
}

// Traverse then nodes
void traversal(struct BTreeNode *myNode) {
  int i;
  if (myNode) {
    for (i = 0; i < myNode->count; i++) {
      traversal(myNode->link[i]);
      printf("%d ", myNode->val[i + 1]);
    }
    traversal(myNode->link[i]);
  }
}

int main() {
  int val, ch;

  insert(8);
  insert(9);
  insert(10);
  insert(11);
  insert(15);
  insert(16);
  insert(17);
  insert(18);
  insert(20);
  insert(23);

  traversal(root);

  printf("\n");
  search(11, &ch, root);
```

```
}
```

Output:

```
8 9 10 11 15 16 17 18 20 23
11 is found
```

iii)B+ Trees

```
// Searching on a B+ Tree in C

#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Default order
#define ORDER 3

typedef struct record {
  int value;
} record;

// Node
typedef struct node {
  void **pointers;
  int *keys;
  struct node *parent;
  bool is_leaf;
  int num_keys;
  struct node *next;
} node;

int order = ORDER;
node *queue = NULL;
bool verbose_output = false;

// Enqueue
void enqueue(node *new_node);

// Dequeue
node *dequeue(void);
int height(node *const root);
int pathToLeaves(node *const root, node *child);
void printLeaves(node *const root);
```

```
void printTree(node *const root);
void findAndPrint(node *const root, int key, bool verbose);
void findAndPrintRange(node *const root, int range1, int range2, bool verbose);
int findRange(node *const root, int key_start, int key_end, bool verbose,
        int returned_keys[], void *returned_pointers[]);
node *findLeaf(node *const root, int key, bool verbose);
record *find(node *root, int key, bool verbose, node **leaf_out);
int cut(int length);

record *makeRecord(int value);
node *makeNode(void);
node *makeLeaf(void);
int getLeftIndex(node *parent, node *left);
node *insertIntoLeaf(node *leaf, int key, record *pointer);
node *insertIntoLeafAfterSplitting(node *root, node *leaf, int key,
                   record *pointer);
node *insertIntoNode(node *root, node *parent,
          int left_index, int key, node *right);
node *insertIntoNodeAfterSplitting(node *root, node *parent,
                   int left_index,
                   int key, node *right);
node *insertIntoParent(node *root, node *left, int key, node *right);
node *insertIntoNewRoot(node *left, int key, node *right);
node *startNewTree(int key, record *pointer);
node *insert(node *root, int key, int value);

// Enqueue
void enqueue(node *new_node) {
  node *c;
  if (queue == NULL) {
    queue = new_node;
    queue->next = NULL;
  } else {
    c = queue;
    while (c->next != NULL) {
      c = c->next;
    }
    c->next = new_node;
    new_node->next = NULL;
  }
}

// Dequeue
node *dequeue(void) {
  node *n = queue;
  queue = queue->next;
```

```
  n->next = NULL;
  return n;
}

// Print the leaves
void printLeaves(node *const root) {
  if (root == NULL) {
    printf("Empty tree.\n");
    return;
  }
  int i;
  node *c = root;
  while (!c->is_leaf)
    c = c->pointers[0];
  while (true) {
    for (i = 0; i < c->num_keys; i++) {
      if (verbose_output)
        printf("%p ", c->pointers[i]);
      printf("%d ", c->keys[i]);
    }
    if (verbose_output)
      printf("%p ", c->pointers[order - 1]);
    if (c->pointers[order - 1] != NULL) {
      printf(" | ");
      c = c->pointers[order - 1];
    } else
      break;
  }
  printf("\n");
}

// Calculate height
int height(node *const root) {
  int h = 0;
  node *c = root;
  while (!c->is_leaf) {
    c = c->pointers[0];
    h++;
  }
  return h;
}

// Get path to root
int pathToLeaves(node *const root, node *child) {
  int length = 0;
  node *c = child;
```

```
  while (c != root) {
    c = c->parent;
    length++;
  }
  return length;
}

// Print the tree
void printTree(node *const root) {
  node *n = NULL;
  int i = 0;
  int rank = 0;
  int new_rank = 0;

  if (root == NULL) {
    printf("Empty tree.\n");
    return;
  }
  queue = NULL;
  enqueue(root);
  while (queue != NULL) {
    n = dequeue();
    if (n->parent != NULL && n == n->parent->pointers[0]) {
      new_rank = pathToLeaves(root, n);
      if (new_rank != rank) {
        rank = new_rank;
        printf("\n");
      }
    }
    if (verbose_output)
      printf("(%p)", n);
    for (i = 0; i < n->num_keys; i++) {
      if (verbose_output)
        printf("%p ", n->pointers[i]);
      printf("%d ", n->keys[i]);
    }
    if (!n->is_leaf)
      for (i = 0; i <= n->num_keys; i++)
        enqueue(n->pointers[i]);
    if (verbose_output) {
      if (n->is_leaf)
        printf("%p ", n->pointers[order - 1]);
      else
        printf("%p ", n->pointers[n->num_keys]);
    }
    printf("| ");
```

```
  }
  printf("\n");
}

// Find the node and print it
void findAndPrint(node *const root, int key, bool verbose) {
  node *leaf = NULL;
  record *r = find(root, key, verbose, NULL);
  if (r == NULL)
    printf("Record not found under key %d.\n", key);
  else
    printf("Record at %p -- key %d, value %d.\n",
         r, key, r->value);
}

// Find and print the range
void findAndPrintRange(node *const root, int key_start, int key_end,
             bool verbose) {
  int i;
  int array_size = key_end - key_start + 1;
  int returned_keys[array_size];
  void *returned_pointers[array_size];
  int num_found = findRange(root, key_start, key_end, verbose,
              returned_keys, returned_pointers);
  if (!num_found)
    printf("None found.\n");
  else {
    for (i = 0; i < num_found; i++)
      printf("Key: %d   Location: %p  Value: %d\n",
           returned_keys[i],
           returned_pointers[i],
           ((record *)
            returned_pointers[i])
             ->value);
  }
}

// Find the range
int findRange(node *const root, int key_start, int key_end, bool verbose,
        int returned_keys[], void *returned_pointers[]) {
  int i, num_found;
  num_found = 0;
  node *n = findLeaf(root, key_start, verbose);
  if (n == NULL)
    return 0;
  for (i = 0; i < n->num_keys && n->keys[i] < key_start; i++)
```

```
    ;
  if (i == n->num_keys)
    return 0;
  while (n != NULL) {
    for (; i < n->num_keys && n->keys[i] <= key_end; i++) {
      returned_keys[num_found] = n->keys[i];
      returned_pointers[num_found] = n->pointers[i];
      num_found++;
    }
    n = n->pointers[order - 1];
    i = 0;
  }
  return num_found;
}

// Find the leaf
node *findLeaf(node *const root, int key, bool verbose) {
  if (root == NULL) {
    if (verbose)
      printf("Empty tree.\n");
    return root;
  }
  int i = 0;
  node *c = root;
  while (!c->is_leaf) {
    if (verbose) {
      printf("[");
      for (i = 0; i < c->num_keys - 1; i++)
        printf("%d ", c->keys[i]);
      printf("%d] ", c->keys[i]);
    }
    i = 0;
    while (i < c->num_keys) {
      if (key >= c->keys[i])
        i++;
      else
        break;
    }
    if (verbose)
      printf("%d ->\n", i);
    c = (node *)c->pointers[i];
  }
  if (verbose) {
    printf("Leaf [");
    for (i = 0; i < c->num_keys - 1; i++)
      printf("%d ", c->keys[i]);
```

```
    printf("%d] ->\n", c->keys[i]);
  }
  return c;
}

record *find(node *root, int key, bool verbose, node **leaf_out) {
  if (root == NULL) {
    if (leaf_out != NULL) {
      *leaf_out = NULL;
    }
    return NULL;
  }

  int i = 0;
  node *leaf = NULL;

  leaf = findLeaf(root, key, verbose);

  for (i = 0; i < leaf->num_keys; i++)
    if (leaf->keys[i] == key)
      break;
  if (leaf_out != NULL) {
    *leaf_out = leaf;
  }
  if (i == leaf->num_keys)
    return NULL;
  else
    return (record *)leaf->pointers[i];
}

int cut(int length) {
  if (length % 2 == 0)
    return length / 2;
  else
    return length / 2 + 1;
}

record *makeRecord(int value) {
  record *new_record = (record *)malloc(sizeof(record));
  if (new_record == NULL) {
    perror("Record creation.");
    exit(EXIT_FAILURE);
  } else {
    new_record->value = value;
  }
  return new_record;
```

```
}

node *makeNode(void) {
  node *new_node;
  new_node = malloc(sizeof(node));
  if (new_node == NULL) {
    perror("Node creation.");
    exit(EXIT_FAILURE);
  }
  new_node->keys = malloc((order - 1) * sizeof(int));
  if (new_node->keys == NULL) {
    perror("New node keys array.");
    exit(EXIT_FAILURE);
  }
  new_node->pointers = malloc(order * sizeof(void *));
  if (new_node->pointers == NULL) {
    perror("New node pointers array.");
    exit(EXIT_FAILURE);
  }
  new_node->is_leaf = false;
  new_node->num_keys = 0;
  new_node->parent = NULL;
  new_node->next = NULL;
  return new_node;
}

node *makeLeaf(void) {
  node *leaf = makeNode();
  leaf->is_leaf = true;
  return leaf;
}

int getLeftIndex(node *parent, node *left) {
  int left_index = 0;
  while (left_index <= parent->num_keys &&
      parent->pointers[left_index] != left)
    left_index++;
  return left_index;
}

node *insertIntoLeaf(node *leaf, int key, record *pointer) {
  int i, insertion_point;

  insertion_point = 0;
  while (insertion_point < leaf->num_keys && leaf->keys[insertion_point] < key)
    insertion_point++;
```

```
  for (i = leaf->num_keys; i > insertion_point; i--) {
    leaf->keys[i] = leaf->keys[i - 1];
    leaf->pointers[i] = leaf->pointers[i - 1];
  }
  leaf->keys[insertion_point] = key;
  leaf->pointers[insertion_point] = pointer;
  leaf->num_keys++;
  return leaf;
}

node *insertIntoLeafAfterSplitting(node *root, node *leaf, int key, record *pointer)
{
  node *new_leaf;
  int *temp_keys;
  void **temp_pointers;
  int insertion_index, split, new_key, i, j;

  new_leaf = makeLeaf();

  temp_keys = malloc(order * sizeof(int));
  if (temp_keys == NULL) {
    perror("Temporary keys array.");
    exit(EXIT_FAILURE);
  }

  temp_pointers = malloc(order * sizeof(void *));
  if (temp_pointers == NULL) {
    perror("Temporary pointers array.");
    exit(EXIT_FAILURE);
  }

  insertion_index = 0;
  while (insertion_index < order - 1 && leaf->keys[insertion_index] < key)
    insertion_index++;

  for (i = 0, j = 0; i < leaf->num_keys; i++, j++) {
    if (j == insertion_index)
      j++;
    temp_keys[j] = leaf->keys[i];
    temp_pointers[j] = leaf->pointers[i];
  }

  temp_keys[insertion_index] = key;
  temp_pointers[insertion_index] = pointer;
```

```
  leaf->num_keys = 0;

  split = cut(order - 1);

  for (i = 0; i < split; i++) {
    leaf->pointers[i] = temp_pointers[i];
    leaf->keys[i] = temp_keys[i];
    leaf->num_keys++;
  }

  for (i = split, j = 0; i < order; i++, j++) {
    new_leaf->pointers[j] = temp_pointers[i];
    new_leaf->keys[j] = temp_keys[i];
    new_leaf->num_keys++;
  }

  free(temp_pointers);
  free(temp_keys);

  new_leaf->pointers[order - 1] = leaf->pointers[order - 1];
  leaf->pointers[order - 1] = new_leaf;

  for (i = leaf->num_keys; i < order - 1; i++)
    leaf->pointers[i] = NULL;
  for (i = new_leaf->num_keys; i < order - 1; i++)
    new_leaf->pointers[i] = NULL;

  new_leaf->parent = leaf->parent;
  new_key = new_leaf->keys[0];

  return insertIntoParent(root, leaf, new_key, new_leaf);
}

node *insertIntoNode(node *root, node *n,
         int left_index, int key, node *right) {
  int i;

  for (i = n->num_keys; i > left_index; i--) {
    n->pointers[i + 1] = n->pointers[i];
    n->keys[i] = n->keys[i - 1];
  }
  n->pointers[left_index + 1] = right;
  n->keys[left_index] = key;
  n->num_keys++;
  return root;
}
```

```
node *insertIntoNodeAfterSplitting(node *root, node *old_node, int left_index,
                 int key, node *right) {
  int i, j, split, k_prime;
  node *new_node, *child;
  int *temp_keys;
  node **temp_pointers;

  temp_pointers = malloc((order + 1) * sizeof(node *));
  if (temp_pointers == NULL) {
    exit(EXIT_FAILURE);
  }
  temp_keys = malloc(order * sizeof(int));
  if (temp_keys == NULL) {
    exit(EXIT_FAILURE);
  }

  for (i = 0, j = 0; i < old_node->num_keys + 1; i++, j++) {
    if (j == left_index + 1)
      j++;
    temp_pointers[j] = old_node->pointers[i];
  }

  for (i = 0, j = 0; i < old_node->num_keys; i++, j++) {
    if (j == left_index)
      j++;
    temp_keys[j] = old_node->keys[i];
  }

  temp_pointers[left_index + 1] = right;
  temp_keys[left_index] = key;

  split = cut(order);
  new_node = makeNode();
  old_node->num_keys = 0;
  for (i = 0; i < split - 1; i++) {
    old_node->pointers[i] = temp_pointers[i];
    old_node->keys[i] = temp_keys[i];
    old_node->num_keys++;
  }
  old_node->pointers[i] = temp_pointers[i];
  k_prime = temp_keys[split - 1];
  for (++i, j = 0; i < order; i++, j++) {
    new_node->pointers[j] = temp_pointers[i];
    new_node->keys[j] = temp_keys[i];
    new_node->num_keys++;
```

```
  }
  new_node->pointers[j] = temp_pointers[i];
  free(temp_pointers);
  free(temp_keys);
  new_node->parent = old_node->parent;
  for (i = 0; i <= new_node->num_keys; i++) {
    child = new_node->pointers[i];
    child->parent = new_node;
  }

  return insertIntoParent(root, old_node, k_prime, new_node);
}

node *insertIntoParent(node *root, node *left, int key, node *right) {
  int left_index;
  node *parent;

  parent = left->parent;

  if (parent == NULL)
    return insertIntoNewRoot(left, key, right);

  left_index = getLeftIndex(parent, left);

  if (parent->num_keys < order - 1)
    return insertIntoNode(root, parent, left_index, key, right);

  return insertIntoNodeAfterSplitting(root, parent, left_index, key, right);
}

node *insertIntoNewRoot(node *left, int key, node *right) {
  node *root = makeNode();
  root->keys[0] = key;
  root->pointers[0] = left;
  root->pointers[1] = right;
  root->num_keys++;
  root->parent = NULL;
  left->parent = root;
  right->parent = root;
  return root;
}

node *startNewTree(int key, record *pointer) {
  node *root = makeLeaf();
  root->keys[0] = key;
  root->pointers[0] = pointer;
```

```
  root->pointers[order - 1] = NULL;
  root->parent = NULL;
  root->num_keys++;
  return root;
}

node *insert(node *root, int key, int value) {
  record *record_pointer = NULL;
  node *leaf = NULL;

  record_pointer = find(root, key, false, NULL);
  if (record_pointer != NULL) {
    record_pointer->value = value;
    return root;
  }

  record_pointer = makeRecord(value);

  if (root == NULL)
    return startNewTree(key, record_pointer);

  leaf = findLeaf(root, key, false);

  if (leaf->num_keys < order - 1) {
    leaf = insertIntoLeaf(leaf, key, record_pointer);
    return root;
  }

  return insertIntoLeafAfterSplitting(root, leaf, key, record_pointer);
}

int main() {
  node *root;
  char instruction;

  root = NULL;

  root = insert(root, 5, 33);
  root = insert(root, 15, 21);
  root = insert(root, 25, 31);
  root = insert(root, 35, 41);
  root = insert(root, 45, 10);

  printTree(root);

  findAndPrint(root, 15, instruction = 'a');
```

```
}
```

Output:

```
5 | 15 | 25 | 35 45 |
[25] 0 ->
[15] 1 ->
Leaf [15] ->
Record at 00000000007014C0 -- key 15, value 21.
```

iv)AVL trees

```
// AVL tree implementation in C

#include <stdio.h>
#include <stdlib.h>

// Create Node
struct Node {
  int key;
  struct Node *left;
  struct Node *right;
  int height;
};

int max(int a, int b);

// Calculate height
int height(struct Node *N) {
  if (N == NULL)
    return 0;
  return N->height;
}

int max(int a, int b) {
  return (a > b) ? a : b;
}

// Create a node
struct Node *newNode(int key) {
  struct Node *node = (struct Node *)
    malloc(sizeof(struct Node));
  node->key = key;
  node->left = NULL;
  node->right = NULL;
  node->height = 1;
```

```
  return (node);
}

// Right rotate
struct Node *rightRotate(struct Node *y) {
  struct Node *x = y->left;
  struct Node *T2 = x->right;

  x->right = y;
  y->left = T2;

  y->height = max(height(y->left), height(y->right)) + 1;
  x->height = max(height(x->left), height(x->right)) + 1;

  return x;
}

// Left rotate
struct Node *leftRotate(struct Node *x) {
  struct Node *y = x->right;
  struct Node *T2 = y->left;

  y->left = x;
  x->right = T2;

  x->height = max(height(x->left), height(x->right)) + 1;
  y->height = max(height(y->left), height(y->right)) + 1;

  return y;
}

// Get the balance factor
int getBalance(struct Node *N) {
  if (N == NULL)
    return 0;
  return height(N->left) - height(N->right);
}

// Insert node
struct Node *insertNode(struct Node *node, int key) {
  // Find the correct position to insertNode the node and insertNode it
  if (node == NULL)
    return (newNode(key));

  if (key < node->key)
    node->left = insertNode(node->left, key);
```

```
  else if (key > node->key)
    node->right = insertNode(node->right, key);
  else
    return node;

  // Update the balance factor of each node and
  // Balance the tree
  node->height = 1 + max(height(node->left),
             height(node->right));

  int balance = getBalance(node);
  if (balance > 1 && key < node->left->key)
    return rightRotate(node);

  if (balance < -1 && key > node->right->key)
    return leftRotate(node);

  if (balance > 1 && key > node->left->key) {
    node->left = leftRotate(node->left);
    return rightRotate(node);
  }

  if (balance < -1 && key < node->right->key) {
    node->right = rightRotate(node->right);
    return leftRotate(node);
  }

  return node;
}

struct Node *minValueNode(struct Node *node) {
  struct Node *current = node;

  while (current->left != NULL)
    current = current->left;

  return current;
}

// Delete a nodes
struct Node *deleteNode(struct Node *root, int key) {
  // Find the node and delete it
  if (root == NULL)
    return root;

  if (key < root->key)
```

```
  root->left = deleteNode(root->left, key);

else if (key > root->key)
  root->right = deleteNode(root->right, key);

else {
  if ((root->left == NULL) || (root->right == NULL)) {
    struct Node *temp = root->left ? root->left : root->right;

    if (temp == NULL) {
      temp = root;
      root = NULL;
    } else
      *root = *temp;
    free(temp);
  } else {
    struct Node *temp = minValueNode(root->right);

    root->key = temp->key;

    root->right = deleteNode(root->right, temp->key);
  }
}

if (root == NULL)
  return root;

// Update the balance factor of each node and
// balance the tree
root->height = 1 + max(height(root->left),
             height(root->right));

int balance = getBalance(root);
if (balance > 1 && getBalance(root->left) >= 0)
  return rightRotate(root);

if (balance > 1 && getBalance(root->left) < 0) {
  root->left = leftRotate(root->left);
  return rightRotate(root);
}

if (balance < -1 && getBalance(root->right) <= 0)
  return leftRotate(root);

if (balance < -1 && getBalance(root->right) > 0) {
  root->right = rightRotate(root->right);
```

```
    return leftRotate(root);
  }

  return root;
}

// Print the tree
void printPreOrder(struct Node *root) {
  if (root != NULL) {
    printf("%d ", root->key);
    printPreOrder(root->left);
    printPreOrder(root->right);
  }
}

int main() {
  struct Node *root = NULL;

  root = insertNode(root, 2);
  root = insertNode(root, 1);
  root = insertNode(root, 7);
  root = insertNode(root, 4);
  root = insertNode(root, 5);
  root = insertNode(root, 3);
  root = insertNode(root, 8);

  printPreOrder(root);

  root = deleteNode(root, 3);

  printf("\nAfter deletion: ");
  printPreOrder(root);

  return 0;
}
```

Output:

```
4 2 1 3 7 5 8
After deletion: 4 2 1 7 5 8
```

v)Red-Black trees

```
// Implementing Red-Black Tree in C
```

```
#include <stdio.h>
#include <stdlib.h>

enum nodeColor {
  RED,
  BLACK
};

struct rbNode {
  int data, color;
  struct rbNode *link[2];
};

struct rbNode *root = NULL;

// Create a red-black tree
struct rbNode *createNode(int data) {
  struct rbNode *newnode;
  newnode = (struct rbNode *)malloc(sizeof(struct rbNode));
  newnode->data = data;
  newnode->color = RED;
  newnode->link[0] = newnode->link[1] = NULL;
  return newnode;
}

// Insert an node
void insertion(int data) {
  struct rbNode *stack[98], *ptr, *newnode, *xPtr, *yPtr;
  int dir[98], ht = 0, index;
  ptr = root;
  if (!root) {
    root = createNode(data);
    return;
  }

  stack[ht] = root;
  dir[ht++] = 0;
  while (ptr != NULL) {
    if (ptr->data == data) {
      printf("Duplicates Not Allowed!!\n");
      return;
    }
    index = (data - ptr->data) > 0 ? 1 : 0;
    stack[ht] = ptr;
    ptr = ptr->link[index];
    dir[ht++] = index;
```

```
}
stack[ht - 1]->link[index] = newnode = createNode(data);
while ((ht >= 3) && (stack[ht - 1]->color == RED)) {
  if (dir[ht - 2] == 0) {
    yPtr = stack[ht - 2]->link[1];
    if (yPtr != NULL && yPtr->color == RED) {
      stack[ht - 2]->color = RED;
      stack[ht - 1]->color = yPtr->color = BLACK;
      ht = ht - 2;
    } else {
      if (dir[ht - 1] == 0) {
        yPtr = stack[ht - 1];
      } else {
        xPtr = stack[ht - 1];
        yPtr = xPtr->link[1];
        xPtr->link[1] = yPtr->link[0];
        yPtr->link[0] = xPtr;
        stack[ht - 2]->link[0] = yPtr;
      }
      xPtr = stack[ht - 2];
      xPtr->color = RED;
      yPtr->color = BLACK;
      xPtr->link[0] = yPtr->link[1];
      yPtr->link[1] = xPtr;
      if (xPtr == root) {
        root = yPtr;
      } else {
        stack[ht - 3]->link[dir[ht - 3]] = yPtr;
      }
      break;
    }
  } else {
    yPtr = stack[ht - 2]->link[0];
    if ((yPtr != NULL) && (yPtr->color == RED)) {
      stack[ht - 2]->color = RED;
      stack[ht - 1]->color = yPtr->color = BLACK;
      ht = ht - 2;
    } else {
      if (dir[ht - 1] == 1) {
        yPtr = stack[ht - 1];
      } else {
        xPtr = stack[ht - 1];
        yPtr = xPtr->link[0];
        xPtr->link[0] = yPtr->link[1];
        yPtr->link[1] = xPtr;
        stack[ht - 2]->link[1] = yPtr;
```

```
      }
      xPtr = stack[ht - 2];
      yPtr->color = BLACK;
      xPtr->color = RED;
      xPtr->link[1] = yPtr->link[0];
      yPtr->link[0] = xPtr;
      if (xPtr == root) {
        root = yPtr;
      } else {
        stack[ht - 3]->link[dir[ht - 3]] = yPtr;
      }
      break;
    }
   }
  }
  root->color = BLACK;
}

// Delete a node
void deletion(int data) {
  struct rbNode *stack[98], *ptr, *xPtr, *yPtr;
  struct rbNode *pPtr, *qPtr, *rPtr;
  int dir[98], ht = 0, diff, i;
  enum nodeColor color;

  if (!root) {
    printf("Tree not available\n");
    return;
  }

  ptr = root;
  while (ptr != NULL) {
    if ((data - ptr->data) == 0)
      break;
    diff = (data - ptr->data) > 0 ? 1 : 0;
    stack[ht] = ptr;
    dir[ht++] = diff;
    ptr = ptr->link[diff];
  }

  if (ptr->link[1] == NULL) {
    if ((ptr == root) && (ptr->link[0] == NULL)) {
      free(ptr);
      root = NULL;
    } else if (ptr == root) {
      root = ptr->link[0];
```

```
    free(ptr);
  } else {
    stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];
  }
} else {
  xPtr = ptr->link[1];
  if (xPtr->link[0] == NULL) {
    xPtr->link[0] = ptr->link[0];
    color = xPtr->color;
    xPtr->color = ptr->color;
    ptr->color = color;

    if (ptr == root) {
      root = xPtr;
    } else {
      stack[ht - 1]->link[dir[ht - 1]] = xPtr;
    }

    dir[ht] = 1;
    stack[ht++] = xPtr;
  } else {
    i = ht++;
    while (1) {
      dir[ht] = 0;
      stack[ht++] = xPtr;
      yPtr = xPtr->link[0];
      if (!yPtr->link[0])
        break;
      xPtr = yPtr;
    }

    dir[i] = 1;
    stack[i] = yPtr;
    if (i > 0)
      stack[i - 1]->link[dir[i - 1]] = yPtr;

    yPtr->link[0] = ptr->link[0];

    xPtr->link[0] = yPtr->link[1];
    yPtr->link[1] = ptr->link[1];

    if (ptr == root) {
      root = yPtr;
    }

    color = yPtr->color;
```

```
    yPtr->color = ptr->color;
    ptr->color = color;
  }
}

if (ht < 1)
  return;

if (ptr->color == BLACK) {
  while (1) {
    pPtr = stack[ht - 1]->link[dir[ht - 1]];
    if (pPtr && pPtr->color == RED) {
      pPtr->color = BLACK;
      break;
    }

    if (ht < 2)
      break;

    if (dir[ht - 2] == 0) {
      rPtr = stack[ht - 1]->link[1];

      if (!rPtr)
        break;

      if (rPtr->color == RED) {
        stack[ht - 1]->color = RED;
        rPtr->color = BLACK;
        stack[ht - 1]->link[1] = rPtr->link[0];
        rPtr->link[0] = stack[ht - 1];

        if (stack[ht - 1] == root) {
          root = rPtr;
        } else {
          stack[ht - 2]->link[dir[ht - 2]] = rPtr;
        }
        dir[ht] = 0;
        stack[ht] = stack[ht - 1];
        stack[ht - 1] = rPtr;
        ht++;

        rPtr = stack[ht - 1]->link[1];
      }

      if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
        (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
```

```
      rPtr->color = RED;
    } else {
      if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) {
        qPtr = rPtr->link[0];
        rPtr->color = RED;
        qPtr->color = BLACK;
        rPtr->link[0] = qPtr->link[1];
        qPtr->link[1] = rPtr;
        rPtr = stack[ht - 1]->link[1] = qPtr;
      }
      rPtr->color = stack[ht - 1]->color;
      stack[ht - 1]->color = BLACK;
      rPtr->link[1]->color = BLACK;
      stack[ht - 1]->link[1] = rPtr->link[0];
      rPtr->link[0] = stack[ht - 1];
      if (stack[ht - 1] == root) {
        root = rPtr;
      } else {
        stack[ht - 2]->link[dir[ht - 2]] = rPtr;
      }
      break;
    }
  } else {
    rPtr = stack[ht - 1]->link[0];
    if (!rPtr)
      break;

    if (rPtr->color == RED) {
      stack[ht - 1]->color = RED;
      rPtr->color = BLACK;
      stack[ht - 1]->link[0] = rPtr->link[1];
      rPtr->link[1] = stack[ht - 1];

      if (stack[ht - 1] == root) {
        root = rPtr;
      } else {
        stack[ht - 2]->link[dir[ht - 2]] = rPtr;
      }
      dir[ht] = 1;
      stack[ht] = stack[ht - 1];
      stack[ht - 1] = rPtr;
      ht++;

      rPtr = stack[ht - 1]->link[0];
    }
    if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
```

```
        (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
        rPtr->color = RED;
      } else {
        if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) {
          qPtr = rPtr->link[1];
          rPtr->color = RED;
          qPtr->color = BLACK;
          rPtr->link[1] = qPtr->link[0];
          qPtr->link[0] = rPtr;
          rPtr = stack[ht - 1]->link[0] = qPtr;
        }
        rPtr->color = stack[ht - 1]->color;
        stack[ht - 1]->color = BLACK;
        rPtr->link[0]->color = BLACK;
        stack[ht - 1]->link[0] = rPtr->link[1];
        rPtr->link[1] = stack[ht - 1];
        if (stack[ht - 1] == root) {
          root = rPtr;
        } else {
          stack[ht - 2]->link[dir[ht - 2]] = rPtr;
        }
        break;
      }
    }
    ht--;
  }
 }
}

// Print the inorder traversal of the tree
void inorderTraversal(struct rbNode *node) {
  if (node) {
    inorderTraversal(node->link[0]);
    printf("%d  ", node->data);
    inorderTraversal(node->link[1]);
  }
  return;
}

// Driver code
int main() {
  int ch, data;
  while (1) {
    printf("1. Insertion\t2. Deletion\n");
    printf("3. Traverse\t4. Exit");
    printf("\nEnter your choice:");
```

```
    scanf("%d", &ch);
    switch (ch) {
      case 1:
        printf("Enter the element to insert:");
        scanf("%d", &data);
        insertion(data);
        break;
      case 2:
        printf("Enter the element to delete:");
        scanf("%d", &data);
        deletion(data);
        break;
      case 3:
        inorderTraversal(root);
        printf("\n");
        break;
      case 4:
        exit(0);
      default:
        printf("Not available\n");
        break;
    }
    printf("\n");
  }
  return 0;
}
```

Output:

```
1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:10
Not available

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:1
Enter the element to insert:30

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:1
Enter the element to insert:4

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:1
Enter the element to insert:6
```

```
1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:1
Enter the element to insert:90

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:1
Enter the element to insert:45

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:1
Enter the element to insert:20

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:3
4  6  20  30  45  90

1. Insertion    2. Deletion
3. Traverse     4. Exit
Enter your choice:
```

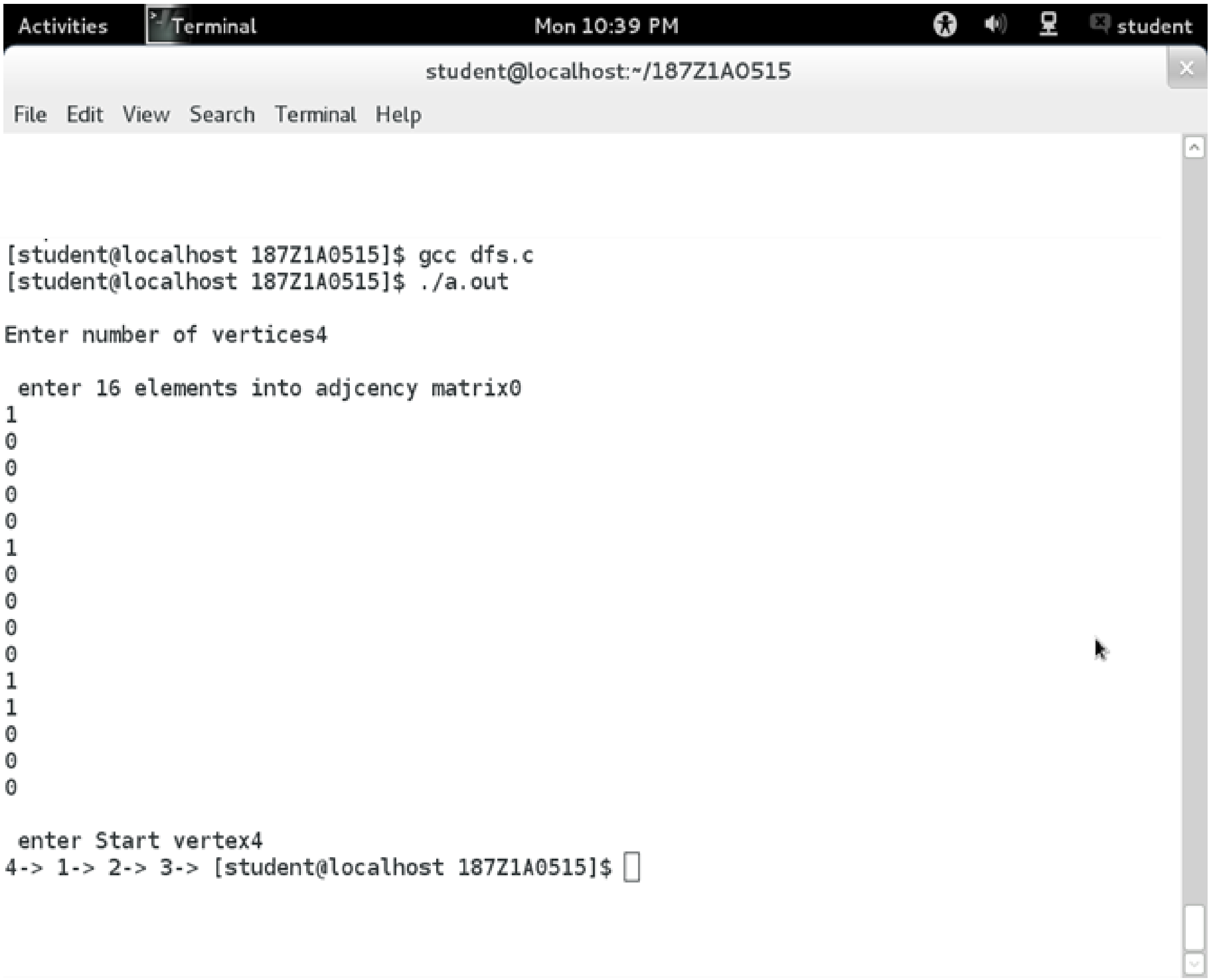
9.Write a program to implement the graph traversal methods.

Program1:Depth First Search (DFS) graph traversal method

```
#include <stdio.h>
int stack[5],top=-1,known[10],n,a[10][10];
void push(int val){
stack[++top]=val;
}
int pop(){
int ver=stack[top--];
printf("%d-> ",ver);
return ver;
}
int stackempty(){
if(top==-1)
return 1;
else
return 0;
}
void dfs(int sv){
```

```
push(sv);
known[sv]=1;
while(!stackempty()){
int i;
int v=pop();
for(i=n;i>0;i--){
if(!(!(a[v][i]))&&!(known[i])){
push(i);
known[i]=1;
}
}
}
}
int main(void) {
// your code goes here
int i,j,sv;
printf("\nEnter number of vertices");
scanf("%d",&n);
printf("\n enter %d elements into adjcency matrix",n*n);
for(i=1;i<=n;i++){
for(j=1;j<=n;j++){
scanf("%d",&a[i][j]);
}
}
printf("\n enter Start vertex");
scanf("%d",&sv);

dfs(sv);
return 0;
}
```

Output:



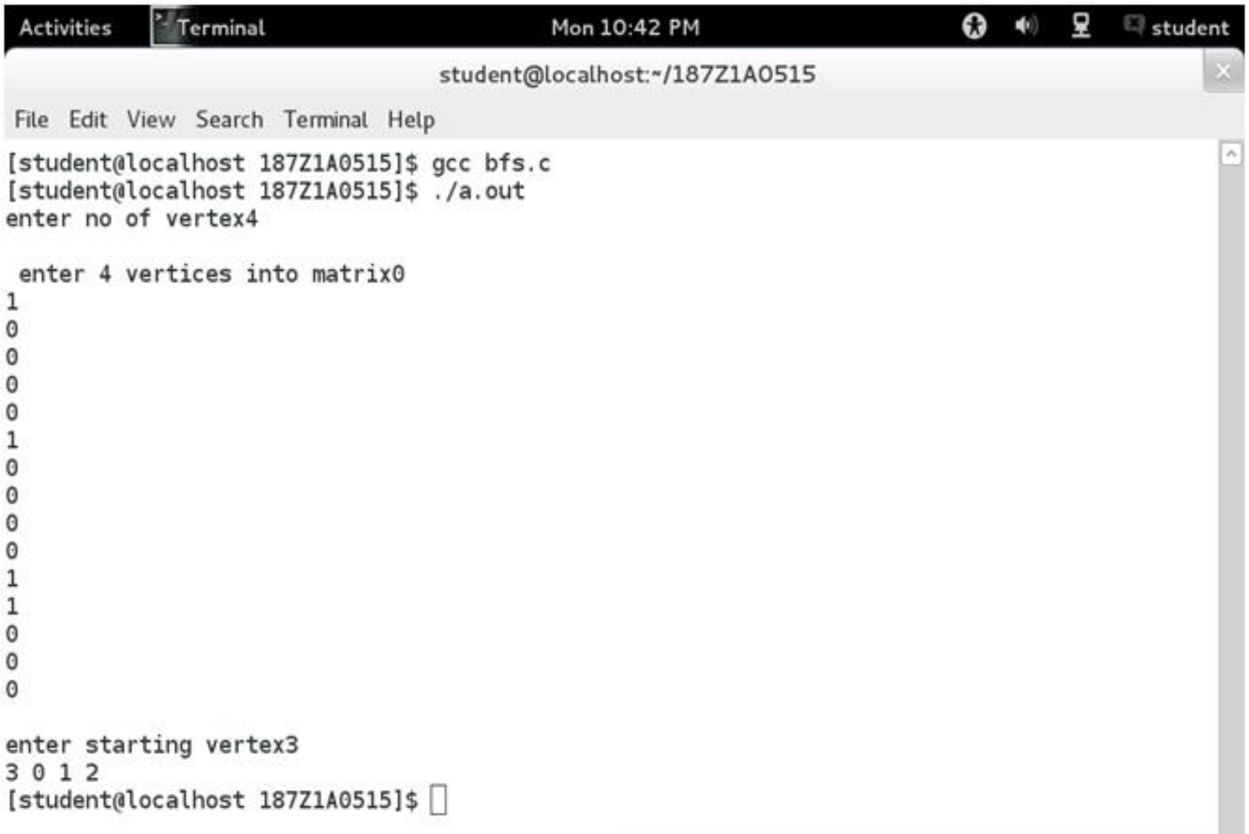
Program 2: Breadth First Search (BFS) graph traversal method

```
#include <stdio.h>

#define QUEUE_SIZE 20
#define MAX 20

//queue
int queue[QUEUE_SIZE];
int queue_front, queue_end;
void enqueue(int v);
int dequeue();

void bfs(int Adj[MAX][MAX], int n, int source);

int main(void) {
```

```
//Adj matrix
int Adj[MAX][MAX] ;//= {{0,1,0,0},{0,0,0,1},{1,0,0,0},{1,0,1,0}};//    {0,1,0,0},
{0,1,1,1},    {1,0,0,1},    {0,0,1,0}  };

int i,j,n;// = 4;  //no. of vertex
int starting_vertex ;//= 2;
printf("enter no of vertex");
scanf("%d",&n);
printf("\n enter %d vertices into matrix",n);
for(i=0;i<n;i++)
for(j=0;j<n;j++)
scanf("%d",&Adj[i][j]);
printf("\nenter starting vertex");
scanf("%d",&starting_vertex);

bfs(Adj, n, starting_vertex);

return 0;
}

void bfs(int Adj[MAX][MAX], int n, int source) {
//variables
int i, j;

//visited array to flag the vertex that
//were visited
int visited[MAX];

//queue
queue_front = 0;
queue_end = 0;

//set visited for all vertex to 0 (means unvisited)
for(i = 0; i <= MAX; i++) {
visited[i] = 0;
}

//mark the visited source
visited[source] = 1;

//enqueue visited vertex
enqueue(source);

//print the vertex as result
printf("%d ", source);
```

```
//continue till queue is not empty
while(queue_front <= queue_end) {
//dequeue first element from the queue
i = dequeue();

for(j = 0; j <n; j++) {
if(visited[j] == 0 && Adj[i][j] == 1) {
//mark vertex as visited
visited[j] = 1;

//push vertex into stack
enqueue(j);

//print the vertex as result
printf("%d ", j);
}
}
}
printf("\n");
}

void enqueue(int v) {
queue[queue_end] = v;
queue_end++;
}

int dequeue() {
int index = queue_front;
queue_front++;
return queue[index];
}
```

Output:



10.Implement a Pattern matching algorithms using Boyer-Moore, Knuth-Morris-Pratt

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

void inputLine(char *line, int maxLen)
{
   fflush(stdin);
   int i = 0;
   char c;
   while ((c = getchar()) != '\n' && i < maxLen)
   {
      line[i++] = tolower(c);
   }
   line[i] = '\0';
}

int main(void)
{
   char word[100];
```

```
char mainString[100];
int i, j, k;
int wordLen, mainStringLen;
int skipTable[100];
int wordIndex, mainStringIndex;
printf("Enter the word: ");
inputLine(word, 100);
printf("Enter the main string: ");
inputLine(mainString, 100);
wordLen = strlen(word);
mainStringLen = strlen(mainString);
for (i = 0; i < wordLen; i++)
{
   skipTable[i] = 1;
}
for (i = 1; i < wordLen; i++)
{
   j = i - 1;
   k = i;
   while (j >= 0 && word[j] == word[k])
   {
      skipTable[k] = j + 1;
      j--;
      k--;
   }
}
wordIndex = 0;
mainStringIndex = 0;
while (mainStringIndex < mainStringLen)
{
   if (word[wordIndex] == mainString[mainStringIndex])
   {
      wordIndex++;
      mainStringIndex++;
   }
   else
   {
      mainStringIndex += skipTable[wordIndex];
      wordIndex = 0;
   }
   if (wordIndex == wordLen)
   {
      printf("The word is found at index %d\n", mainStringIndex - wordLen);
      wordIndex = 0;
   }
}
```

```
    if (wordIndex != 0)
    {
        printf("The word is not found\n");
    }
    return 0;
}
```

Output:

```
Enter the word: aba
Enter the main string: abc ana dhg aana aba

The word is found at index 17
```