Practical : -1

AIM : - Write a program which creates Binary Search Tree. And also implement recursive and non-recursive tree traversing methods inorder, preorder and post-order for the BST.

Code: -

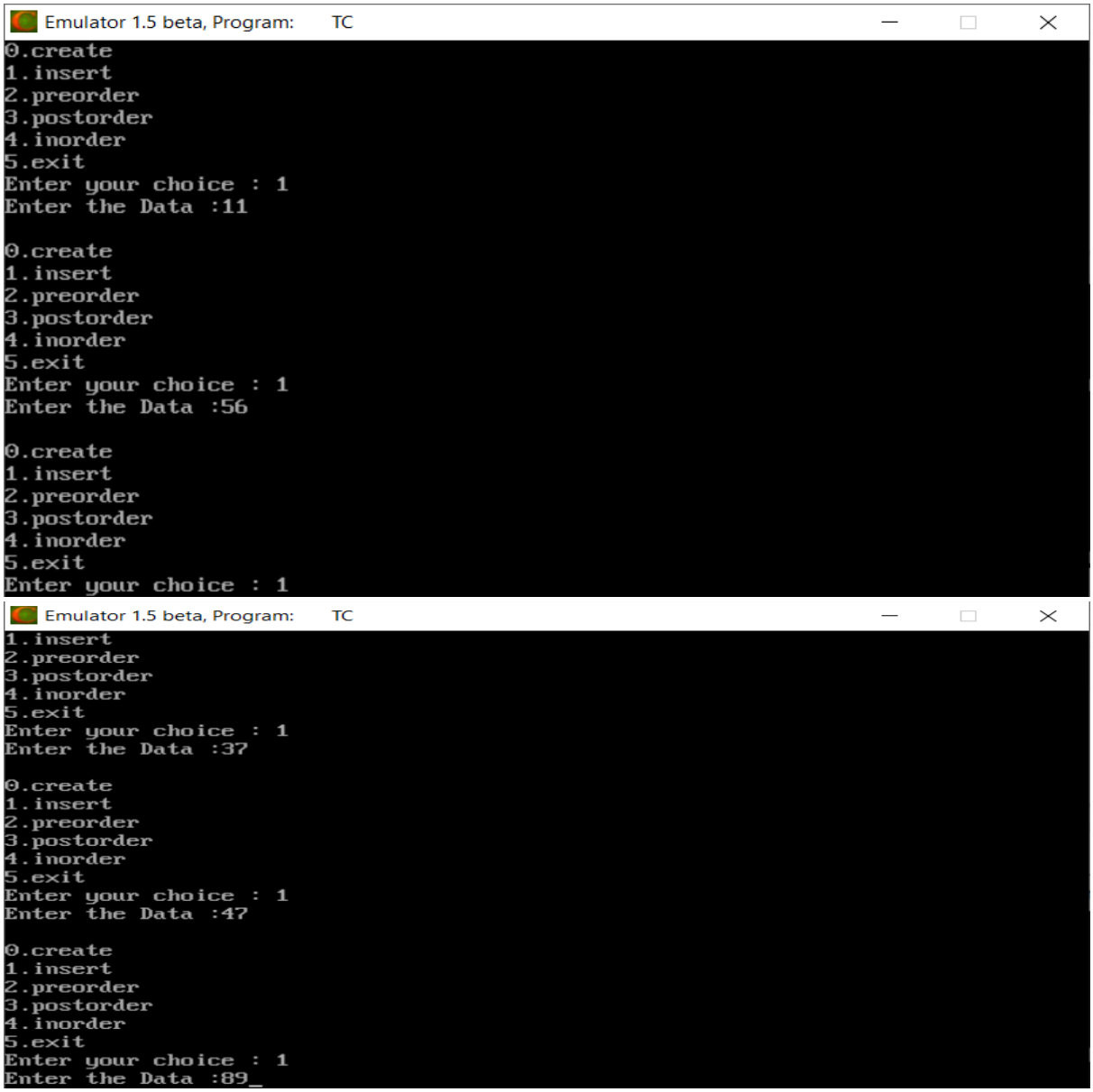
```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node{
        int data;
        struct node *rlink;
        struct node *llink;
}*tmp=NULL;
typedef struct node NODE;
NODE *create();
void preorder(NODE *);
void inorder(NODE *);
void postorder(NODE *);
void insert(NODE *);
void main(){
int n,i,m;
clrscr();
do{
        printf("\n0.create\n1.insert\n2.preorder\n3.postorder\n4.inorder\n5.exit");
        printf("\nEnter your choice : ");
        scanf("%d",&m);
        switch(m){
                case 0:
                tmp=create();
                break;
                case 1:
                insert(tmp);
                break;
                case 2:
                printf("\nDisplay tree in Preorder : ");
                preorder(tmp);
                break;
                case 3:
                printf("\nDisplay Tree in Postorder : ");
                postorder(tmp);
                break;
                case 4:
                printf("\nDisplay Tree in Inorder : ");
```

```
                inorder(tmp);
                break;
                case 5:
                exit(0);
                }
        }
        while(n!=5);
        getch();
}
void insert(NODE *root){
        NODE *newnode;
        if(root==NULL){
                newnode=create();
                root=newnode;
        }
        else{
                newnode=create();
                  while(1){
                        if(newnode->data<root->data){
                                if(root->llink==NULL){
                                        root->llink=newnode;
                                        break;
                                }
                                root=root->llink;
                        }
                        if(newnode->data>root->data){
                                if(root->rlink==NULL){
                                        root->rlink=newnode;
                                        break;
                                }
                        root=root->rlink;
                        }
                   }
                }
        }
NODE *create(){
        NODE *newnode;
        int n;
        newnode=(NODE *)malloc(sizeof(NODE));
        printf("Enter the Data :");
        scanf("%d",&n);
        newnode->data=n;
        newnode->llink=NULL;
        newnode->rlink=NULL;
```

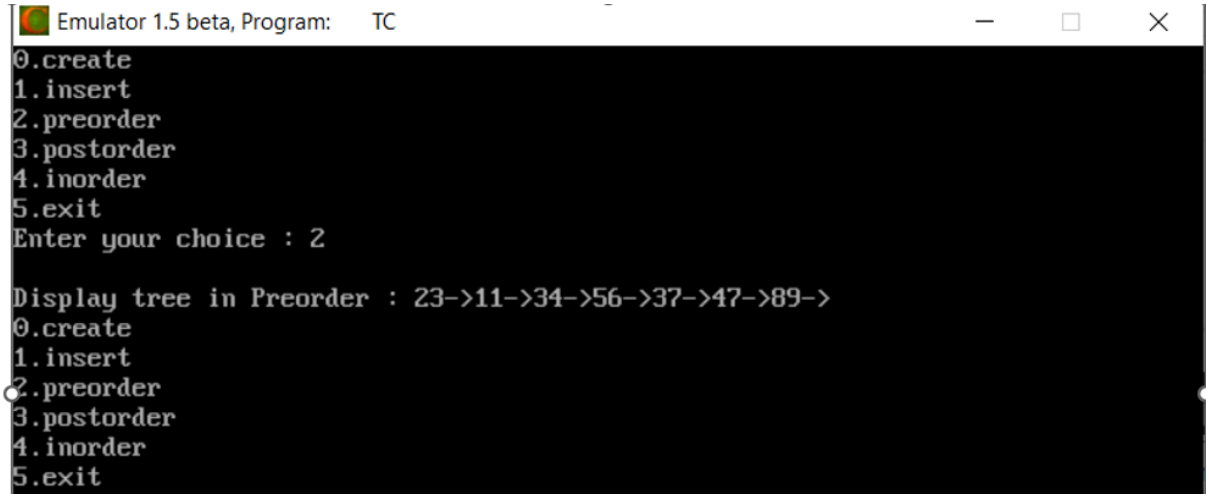
```
        return(newnode);
}
void postorder(NODE *tmp){
        if(tmp!=NULL){
                postorder(tmp->llink);
                postorder(tmp->rlink);
                printf("%d->",tmp->data);
        }
}
void inorder(NODE *tmp){
        if(tmp!=NULL){
                inorder(tmp->llink);
                printf("%d->",tmp->data);
                inorder(tmp->rlink);
        }
}
void preorder(NODE *tmp){
        if(tmp!=NULL){
                printf("%d->",tmp->data);
                preorder(tmp->llink);
                preorder(tmp->rlink);
        }
}
```

OUTPUT:-

1) Create tree



2) Pre order Traversal



3) Post order Traversal

```
0.create
1.insert
2.preorder
3.postorder
4.inorder
5.exit
Enter your choice : 3

Display Tree in Postorder : 11->47->37->89->56->34->23->
0.create
1.insert
2.preorder
3.postorder
4.inorder
5.exit
```

4) In order Traversal

```
0.create
1.insert
2.preorder
3.postorder
4.inorder
5.exit
Enter your choice : 4

Display Tree in Inorder : 11->23->34->37->47->56->89->
0.create
1.insert
2.preorder
3.postorder
4.inorder
5.exit
```

Practical :-2

AIM :- Write a program to implement any two hashing methods. Use any one of the hashing method to implement Insert, Delete and Search operations for Hash Table Management.

Code :-

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int tableSize = 0, totEle = 0 ,tsize;
struct node *hashTable = NULL;
struct node {
        int value, key;
        int marker;
};
int hasht(int key)
{
 int i ;
 i = key%tsize ;
 return i;
}
int rehashq(int key, int j)
{
 int i ;
 i = (key+(j*j))%tsize ;
 return i ;
}
void insertInHash(int key,  int value) {
        int hashIndex = key % tableSize;
        if (tableSize == totEle) {
                printf("Can't perform Insertion..Hash Table is full!!");
                return;
        }
        while (hashTable[hashIndex].marker == 1) {
                hashIndex = (hashIndex + 1)%tableSize;
        }
        hashTable[hashIndex].key = key;
        hashTable[hashIndex].value = value;
        hashTable[hashIndex].marker = 1;
        totEle++;
        return;
}
void deleteFromHash(int key) {
        int hashIndex = key % tableSize, count = 0, flag = 0;
```

```
        if (totEle == 0) {
                printf("Hash Table is Empty!!\n");
                return;
        }
        while (hashTable[hashIndex].marker != 0 && count <= tableSize) {
                if (hashTable[hashIndex].key == key) {
                        hashTable[hashIndex].key = 0;
                        /* set marker to -1 during deletion operation*/
                        hashTable[hashIndex].marker = -1;
                        hashTable[hashIndex].value = 0;
                        totEle--;
                        flag = 1;
                        break;
                }
                hashIndex = (hashIndex + 1)%tableSize;
                count++;
        }
        if (flag)
                printf("Given data deleted from Hash Table\n");
        else
                printf("Given data is not available in Hash Table\n");
        return;
}
void searchElement(int key) {
        int hashIndex = key % tableSize, flag = 0, count = 0;
        if (totEle == 0) {
                printf("Hash Table is Empty!!");
                return;
        }
        while (hashTable[hashIndex].marker != 0 && count <= tableSize) {
                if (hashTable[hashIndex].key == key) {
                        printf("Key : %d\n", hashTable[hashIndex].key);
                        printf("Value : %d\n", hashTable[hashIndex].value);
                        flag = 1;
                        break;
                }
                hashIndex = (hashIndex + 1)%tableSize;
        }

        if (!flag)
                printf("Given data is not present in hash table\n");
        return;
}
void display() {
```

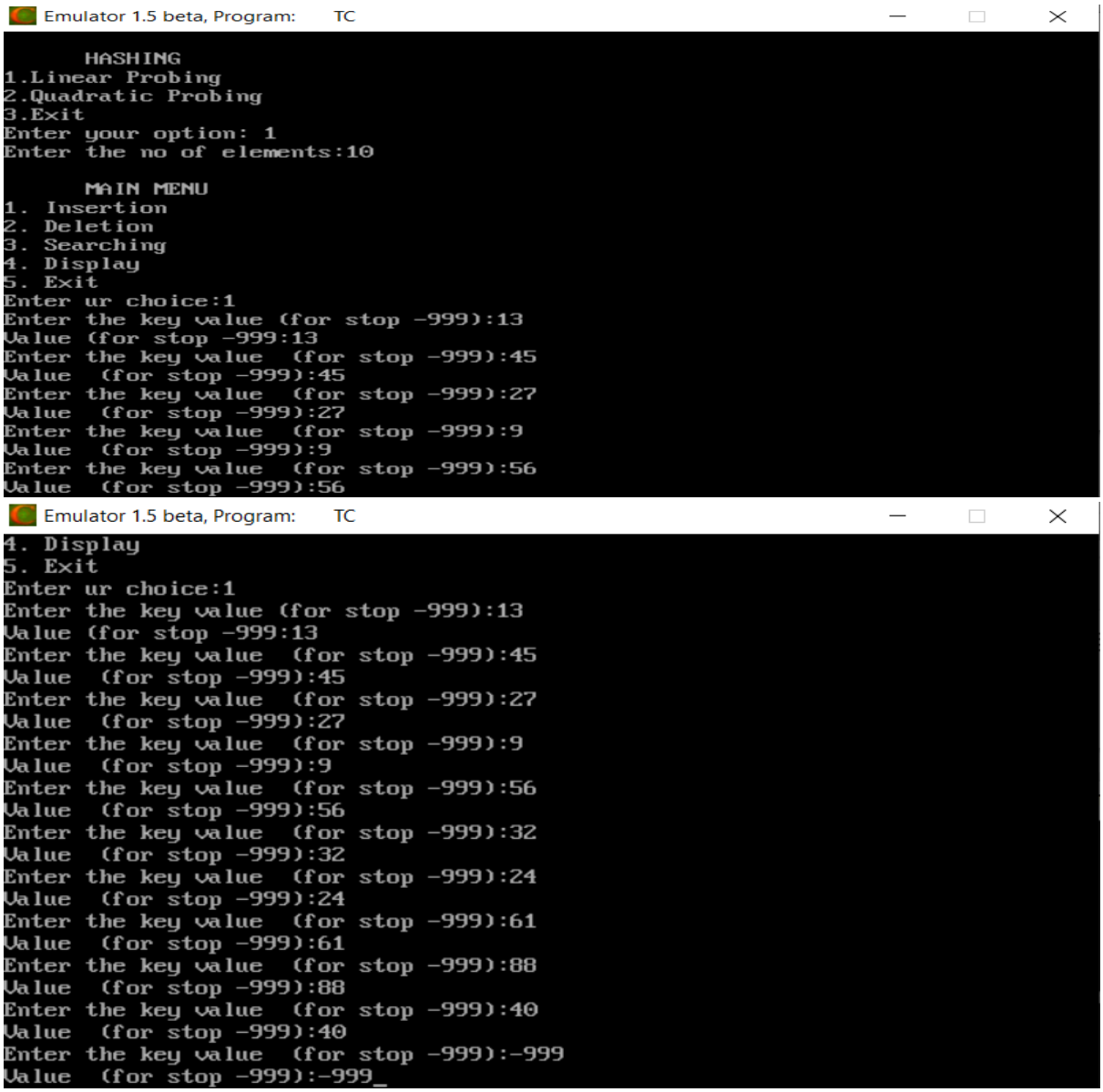
```
	int i;
	if (totEle == 0) {
		printf("Hash Table is Empty!!\n");
		return;
	}
	printf("Key    Value    Index \n");
	printf("----------------------\n");
	for (i = 0; i < tableSize; i++) {
		if (hashTable[i].marker == 1) {
			printf("%-13d", hashTable[i].key);
			printf("%-7d", hashTable[i].value);
			printf("%d\n", i);
		}
	}
	printf("\n");
	return;
}
void linear_probing() {
	int key, value, ch;
	printf("Enter the no of elements:");
	scanf("%d", &tableSize);
	hashTable = (struct node *)calloc(tableSize, sizeof(struct node));
	while (1) {
		printf("\n      MAIN MENU      \n");
		printf("1. Insertion\n2. Deletion\n");
		printf("3. Searching\n4. Display\n");
		printf("5. Exit\nEnter ur choice:");
		scanf("%d", &ch);
		switch (ch) {
			case 1:
				printf("Enter the key value (for stop -999):");
				scanf("%d", &key);
				printf("Value (for stop -999:");
				scanf("%d", &value);
				insertInHash(key, value);
				while(key!=-999 && value != -999){
					printf("Enter the key value  (for stop -999):");
					scanf("%d", &key);
					printf("Value  (for stop -999):");
					scanf("%d", &value);
					insertInHash(key, value);
				}
				break;
			case 2:
```

```
                        printf("Enter the key value:");
                        scanf("%d", &key);
                        deleteFromHash(key);
                        break;
                case 3:
                        printf("Enter the key value:");
                        scanf("%d", &key);
                        searchElement(key);
                        break;
                case 4:
                        display();
                        break;
                case 5:
                        exit(0);
                default:
                        printf("U have entered wrong Option!!\n");
                        break;
            }
        }
}
void quar_probing()
{
        int key,arr[20],hash[20],i,n,s,op,j,k ;
        printf ("Enter the size of the hash table:  ");
   scanf ("%d",&tsize);
   printf ("\nEnter the number of elements: ");
   scanf ("%d",&n);
   for (i=0;i<tsize;i++)
 hash[i]=-1 ;
   printf ("Enter Elements: ");
   for (i=0;i<n;i++)
   {
 scanf("%d",&arr[i]);
   }
    for (i=0;i<tsize;i++)
 hash[i]=-1 ;
    for(k=0;k<n;k++)
    {
 j=1;
 key=arr[k] ;
 i = hasht(key);
 while (hash[i]!=-1)
 {
    i = rehashq(i,j);
```

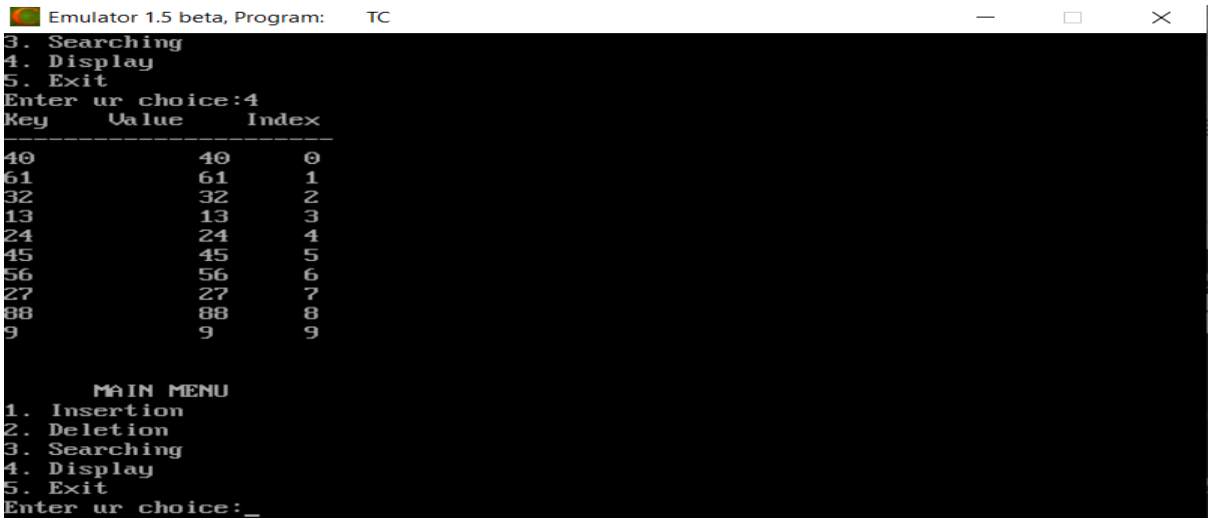
```
      j++ ;
  }
  hash[i]=key ;
     }
     printf("\nThe elements in the array are: ");
     for (i=0;i<tsize;i++)
     {
  printf("\n Element at position %d: %d",i,hash[i]);
     }
}
void main()
{
    int key,arr[20],hash[20],i,n,s,op,j,k ;
    clrscr() ;
    do
    {
    printf("\n      HASHING   ");
 printf("\n1.Linear Probing\n2.Quadratic Probing \n3.Exit \nEnter your option: ");
 scanf("%d",&op);
 switch(op)
 {
 case 1:
     linear_probing();
            break;
 case 2:quar_probing();
                    break;
 }
    }while(op!=3);
    getch() ;
}
```

OUTPUT:

1)Insertion in Linear Probing



2)Display in Linear Probing



3) Delete in Linear Probing

```
      MAIN MENU
1. Insertion
2. Deletion
3. Searching
4. Display
5. Exit
Enter ur choice:2
Enter the key value:24
Given data deleted from Hash Table
```

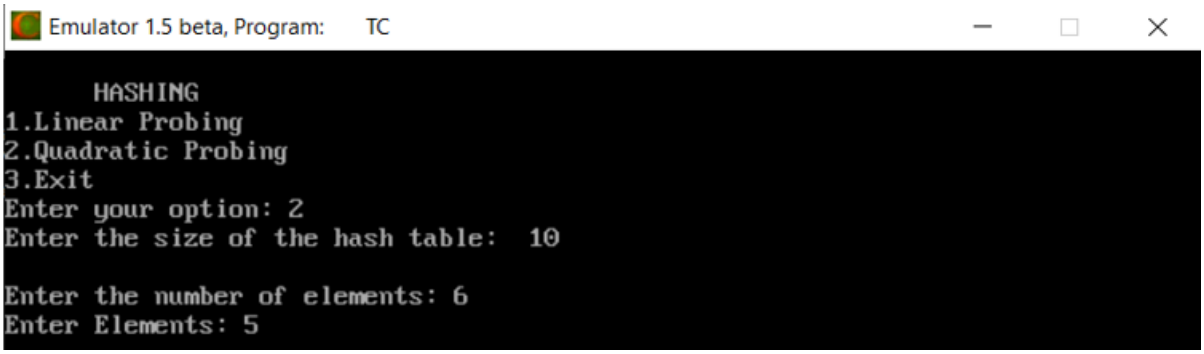
4) Display after Delete operation in Linear Probing

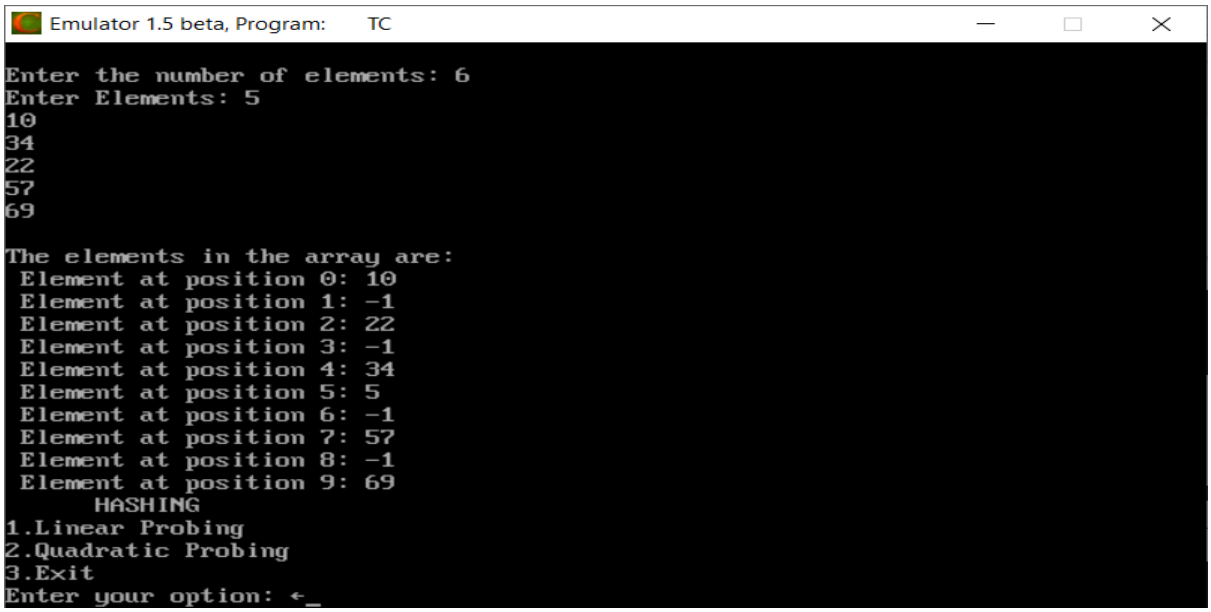
```
2. Deletion
3. Searching
4. Display
5. Exit
Enter ur choice:4
Key     Value     Index
----------------------
40           40      0
61           61      1
32           32      2
13           13      3
45           45      5
56           56      6
27           27      7
88           88      8
9            9       9
```

5) Searching in Linear Probing

```
      MAIN MENU
1. Insertion
2. Deletion
3. Searching
4. Display
5. Exit
Enter ur choice:3
Enter the key value:88
Key : 88
Value : 88

      MAIN MENU
1. Insertion
2. Deletion
3. Searching
4. Display
5. Exit
Enter ur choice:_
```

- **Quadratic Probing**





Practical :- 3

AIM :- Explain Dictionary as an Abstract Data Type. Implement Dictionary using suitable Data Structure.

Code :-

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<conio.h>
struct hash *hashTable = NULL;
int eleCount = 0;
struct node {
        int roll_no, mark,std;
        char name[100];
        struct node *next;
};
struct hash {
        struct node *head;
        int count;
};
struct node * createNode(int roll_no, char *name, int mark,int std) {
        struct node *newnode;
        newnode = (struct node *)malloc(sizeof(struct node));
        newnode->roll_no = roll_no;
        newnode->mark = mark;
        newnode->std = std;
        strcpy(newnode->name, name);
        newnode->next = NULL;
        return newnode;
}
void insertToHash(int roll_no, char *name, int mark,int std) {
        int hashIndex = roll_no % eleCount;
        struct node *newnode =  createNode(roll_no, name, mark,std);
        if (!hashTable[hashIndex].head) {
                hashTable[hashIndex].head = newnode;
                hashTable[hashIndex].count = 1;
                return;
        }
        newnode->next = (hashTable[hashIndex].head);
        hashTable[hashIndex].head = newnode;
        hashTable[hashIndex].count++;
        return;
}
void deleteFromHash(int roll_no) {
```

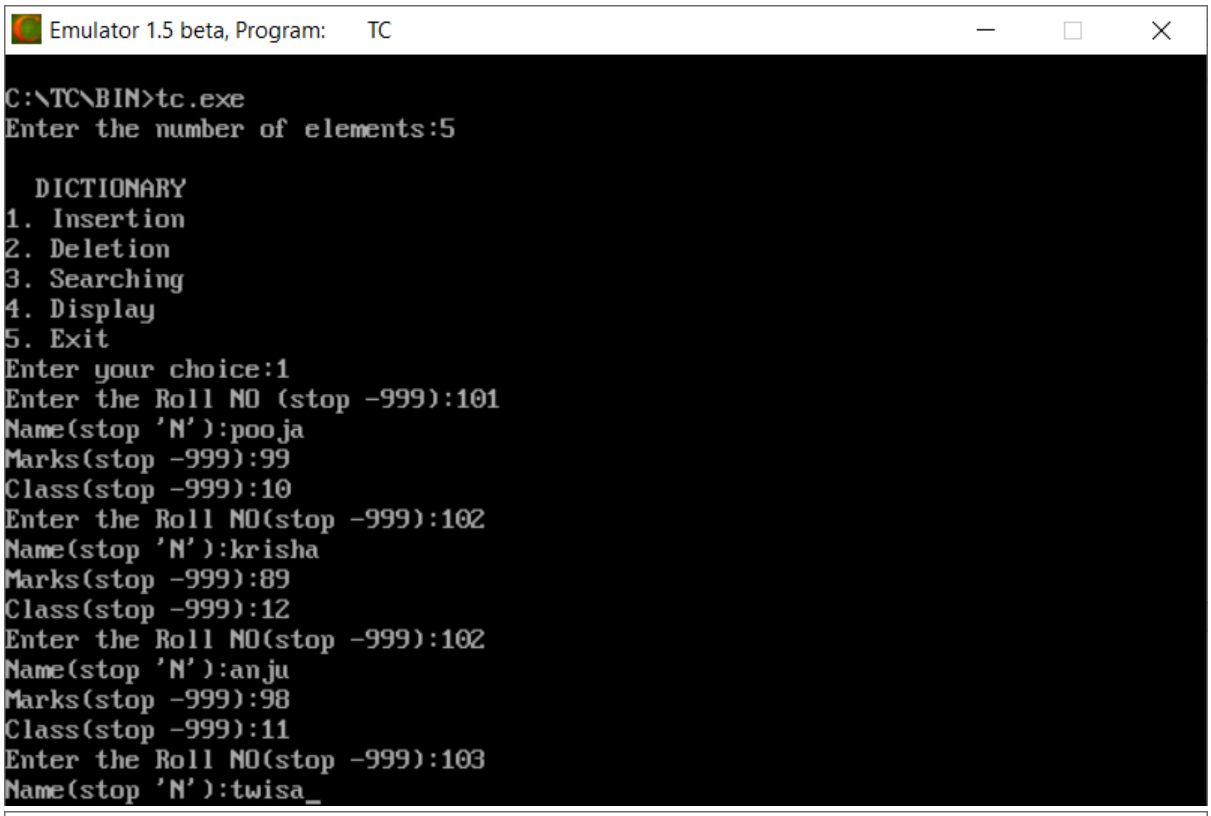
```
        int hashIndex = roll_no % eleCount, flag = 0;
        struct node *temp, *myNode;
        myNode = hashTable[hashIndex].head;
        if (!myNode) {
                printf("Given data is not present in hash Table!!\n");
                return;
        }
        temp = myNode;
        while (myNode != NULL) {
                if (myNode->roll_no == roll_no) {
                        flag = 1;
                        if (myNode == hashTable[hashIndex].head)
                                hashTable[hashIndex].head = myNode->next;
                        else
                                temp->next = myNode->next;
                        hashTable[hashIndex].count--;
                        free(myNode);
                        break;
                }
                temp = myNode;
                myNode = myNode->next;
        }
        if (flag)
                printf("Data deleted successfully from Hash Table\n");
        else
                printf("Given data is not present in hash Table!!!!\n");
        return;
}
void searchInHash(int roll_no) {
        int hashIndex = roll_no % eleCount, flag = 0;
        struct node *myNode;
        myNode = hashTable[hashIndex].head;
        if (!myNode) {
                printf("Search element unavailable in hash table\n");
                return;
        }
        while (myNode != NULL) {
                if (myNode->roll_no == roll_no) {
                        printf("RollNo  : %d\n", myNode->roll_no);
                        printf("Name    : %s\n", myNode->name);
                        printf("Mark    : %d\n", myNode->mark);
                        printf("Class   : %d\n",myNode->std);
                        flag = 1;
                        break;
```

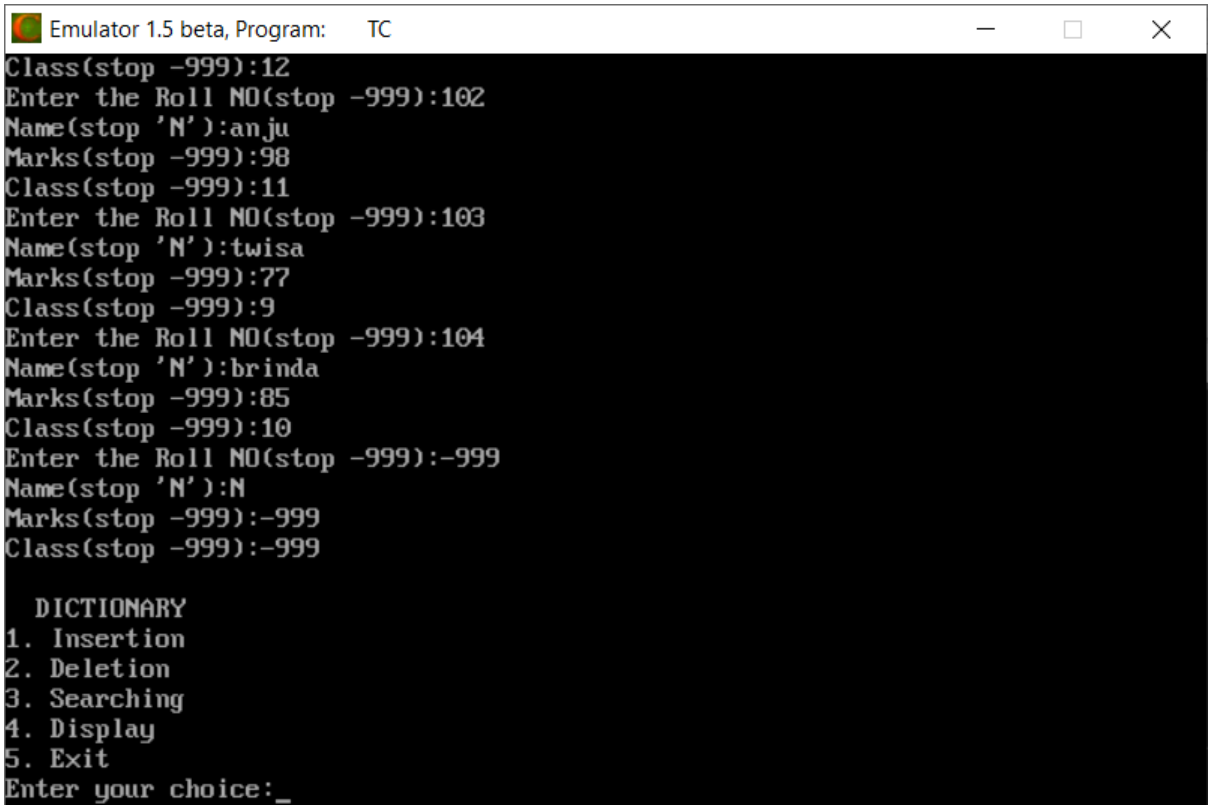
```
                }
                myNode = myNode->next;
        }
        if (!flag)
                printf("Search element unavailable in hash table\n");
        return;
}
void display() {
        struct node *myNode;
        int i;
        for (i = 0; i < eleCount; i++) {
                if (hashTable[i].count == 0)
                        continue;
                myNode = hashTable[i].head;
                if (!myNode)
                        continue;
                printf("\nData at index %d in Hash Table:\n", i);
                printf("Rollno     Name          Class      Mark   \n");
                printf("-----------------------------------------\n");
                while (myNode != NULL) {
                        printf("%-12d", myNode->roll_no);
                        printf("%-15s", myNode->name);
                        printf("%-12d",myNode->std);
                        printf("%d\n", myNode->mark);
                        myNode = myNode->next;
                }
        }
        return;
}
int main() {
        int n, ch, roll_no, mark,std;
        char name[100];
        // clrscr();
        printf("Enter the number of elements:");
        scanf("%d", &n);
        eleCount = n;
        hashTable = (struct hash *)calloc(n, sizeof (struct hash));
        while (1) {
                printf("\n  DICTIONARY    ");
                printf("\n1. Insertion\n2. Deletion\n");
                printf("3. Searching\n4. Display\n5. Exit\n");
                printf("Enter your choice:");
                scanf("%d", &ch);
                switch (ch) {
```

```
                        case 1:
                                printf("Enter the Roll NO (stop -999):");
                                scanf("%d", &roll_no);
                                printf("Name(stop 'N'):");
                                scanf("%s",&name);
                                printf("Marks(stop -999):");
                                scanf("%d", &mark);
                                printf("Class(stop -999):");
                                scanf("%d",&std);
                                insertToHash(roll_no, name, mark,std);
                        while(roll_no!=-999&& name!="n" && mark!=-999 && std!=-999){
                                printf("Enter the Roll NO(stop -999):");
                                scanf("%d", &roll_no);
                                printf("Name(stop 'N'):");
                                scanf("%s",&name);
                                printf("Marks(stop -999):");
                                scanf("%d", &mark);
                                printf("Class(stop -999):");
                                scanf("%d",&std);
                                insertToHash(roll_no, name, mark,std);
                                }
                                break;
                        case 2:
                                printf("Enter the roll_no to perform deletion:");
                                scanf("%d", &roll_no);
                                deleteFromHash(roll_no);
                                break;
                        case 3:
                                printf("Enter the roll_no to search:");
                                scanf("%d", &roll_no);
                                searchInHash(roll_no);
                                break;
                        case 4:
                                display();
                                break;
                        case 5:
                                exit(0);
                        default:
                                printf("U have entered wrong option!!\n");
                                break;
                }
        }
        return 0;
}
```

OUTPUT :

1) **Insertion in dictionary**





2) Display of dictionary

```
101         pooja          10          99

Data at index 2 in Hash Table:
Rollno      Name           Class       Mark
-----------------------------------------
102         anju           11          98
102         krisha         12          89

Data at index 3 in Hash Table:
Rollno      Name           Class       Mark
-----------------------------------------
103         twisa          9           77

Data at index 4 in Hash Table:
Rollno      Name           Class       Mark
-----------------------------------------
104         brinda         10          85

  DICTIONARY
1. Insertion
2. Deletion
3. Searching
4. Display
5. Exit
Enter your choice:
```

3) Searching in dictionary

```
  DICTIONARY
1. Insertion
2. Deletion
3. Searching
4. Display
5. Exit
Enter your choice:3
Enter the roll_no to search:104
RollNo  : 104
Name    : brinda
Mark    : 85
```

4) Deletion in dictionary

```
  DICTIONARY
1. Insertion
2. Deletion
3. Searching
4. Display
5. Exit
Enter your choice:2
Enter the roll_no to perform deletion:103
Data deleted successfully from Hash Table
```

Practical :- 4

AIM :- Write a program which creates AVLTree. Implement Insert and Delete Operations in AVL Tree. (Note that each time the tree must be balanced.)

Code :-

```
#include<stdio.h>
#include<conio.h>
typedef struct node {
        int data;
        struct node *left,*right;
        int ht;
}node;
node * rotateright(node *x) {
        node *y;
        y=x->left;
        x->left=y->right;
        y->right=x;
        x->ht=height(x);
        y->ht=height(y);
        return(y);
}
node * rotateleft(node *x) {
        node *y;
        y=x->right;
        x->right=y->left;
        y->left=x;
        x->ht=height(x);
        y->ht=height(y);
        return(y);
}
  node * LL(node *T) {
        T=rotateright(T);
        return(T);
}
node * LR(node *T) {
        T->left=rotateleft(T->left);
        T=rotateright(T);
        return(T);
}
node * RL(node *T) {
        T->right=rotateright(T->right);
        T=rotateleft(T);
        return(T);
}
int BF(node *T) {
        int lh,rh;
        if(T==NULL)
                return(0);
        if(T->left==NULL)
                lh=0;
```

```
        else
                lh=1+T->left->ht;
        if(T->right==NULL)
                rh=0;
        else
                rh=1+T->right->ht;
        return(lh-rh);
}
void postorder(node *T) {
        if(T!=NULL) {
                postorder(T->left);
                printf("%d(Bf=%d)",T->data,BF(T));
                postorder(T->right);
        }
}
node * RR(node *T) {
        T=rotateleft(T);
        return(T);
}
node * insert(node *T,int x) {
        if(T==NULL) {
                T=(node*)malloc(sizeof(node));
                T->data=x;
                T->left=NULL;
                T->right=NULL;
        }
        else
                if(x > T->data) {               // insert in right subtree
                        T->right=insert(T->right,x);
                        if(BF(T)==-2)
                                if(x>T->right->data)
                                        T=RR(T);
                                else
                                        T=RL(T);
                }
                else
                        if(x<T->data) {
                                T->left=insert(T->left,x);
                                if(BF(T)==2)
                                        if(x < T->left->data)
                                                T=LL(T);
                                        else
                                                T=LR(T);
                        }
                T->ht=height(T);
                return(T);
}
node * Delete(node *T,int x) {
        node *p;
        if(T==NULL) {
```

```
                return NULL;
        }
        else
                if(x > T->data) {               // insert in right subtree
                        T->right=Delete(T->right,x);
                        if(BF(T)==2)
                                if(BF(T->left)>=0)
                                        T=LL(T);
                                else
                                        T=LR(T);
                }
                else
                        if(x<T->data) {
                                T->left=Delete(T->left,x);
                                if(BF(T)==-2)
                                        if(BF(T->right)<=0)
                                                T=RR(T);
                                        else
                                                T=RL(T);
                        }
                        Else {
                                if(T->right!=NULL) {
                                        p=T->right;
                                        while(p->left!= NULL)
                                                p=p->left;
                                        T->data=p->data;
                                        T->right=Delete(T->right,p->data);
                                        if(BF(T)==2)//Rebalance during windup
                                                if(BF(T->left)>=0)
                                                        T=LL(T);
                                                else
                                                        T=LR(T);\
                                }
                                else
                                        return(T->left);
                        }
        T->ht=height(T);
        return(T);
}
int height(node *T) {
        int lh,rh;
        if(T==NULL)
                return(0);
        if(T->left==NULL)
                lh=0;
        else
                lh=1+T->left->ht;
        if(T->right==NULL)
                rh=0;
        else
```

```
                rh=1+T->right->ht;
        if(lh>rh)
                return(lh);

        return(rh);
}
int main() {
        node *root=NULL;
        int x,n,i,op,ch;
        do{
                printf("\n1.Create");
                printf("\n2.Insert");
                printf("\n3.Delete");
                printf("\n4.Print");
                printf("\n5.Quit");
                printf("\nEnter Your Choice:");
                scanf("%d",&ch);
                switch(ch) {
                        case 1: printf("\nEnter no. of elements:");
                                scanf("%d",&n);
                                printf("\nEnter tree data:");
                                root=NULL;
                                for(i=0;i<n;i++) {
                                        scanf("%d",&x);
                                        root=insert(root,x);
                                }
                                break;
                        case 2: printf("\nEnter a data:");
                                scanf("%d",&x);
                                root=insert(root,x);
                                break;
                        case 3: printf("\nEnter a data:");
                                scanf("%d",&x);
                                root=Delete(root,x);
                                break;
                        case 4: printf("\nPostorder sequence: ");
                                postorder(root);
                                break;
                }
        }while(ch!=5);
        return 0;
}
```

OUTPUT :

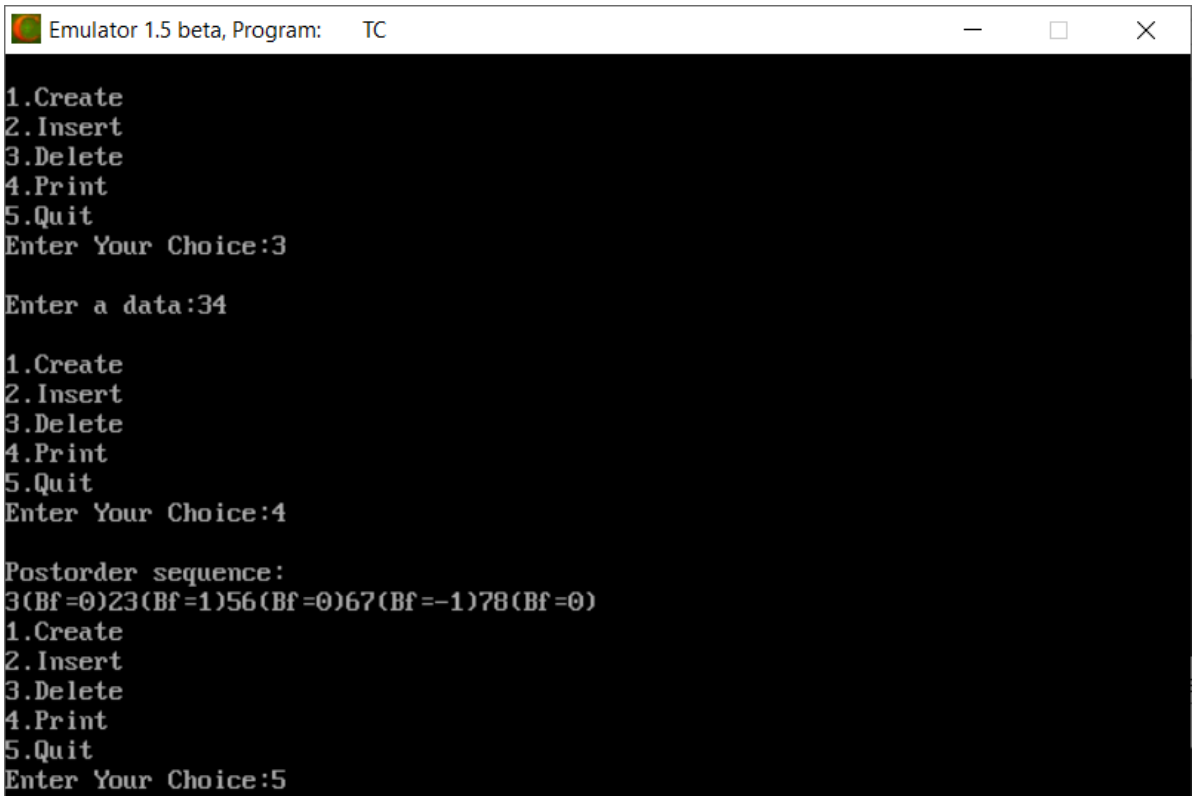
1) **Create & Insertion in AVL tree**

```
C:\TC\BIN>tc.exe

1.Create
2.Insert
3.Delete
4.Print
5.Quit
Enter Your Choice:1

Enter no. of elements:5

Enter tree data:23
67
34
3
78_
```

2) **Display AVL tree**

```
Emulator 1.5 beta, Program:     TC
Enter Your Choice:1

Enter no. of elements:5

Enter tree data:23
67
34
3
78

1.Create
2.Insert
3.Delete
4.Print
5.Quit
Enter Your Choice:4

Postorder sequence:
3(Bf=0)23(Bf=1)34(Bf=0)67(Bf=-1)78(Bf=0)
1.Create
2.Insert
3.Delete
4.Print
5.Quit
Enter Your Choice:_
```

3) Deletion in AVL tree and display it.



Practical :- 5

AIM :- Implement Red-Black Tree.

Code :-

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
//enum nodeColor{RED,BLACK};
struct rbNode{
  int data, color;
  struct rbNode *link[2];
};
struct rbNode *root = NULL;
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;
}
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;
}
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;
```

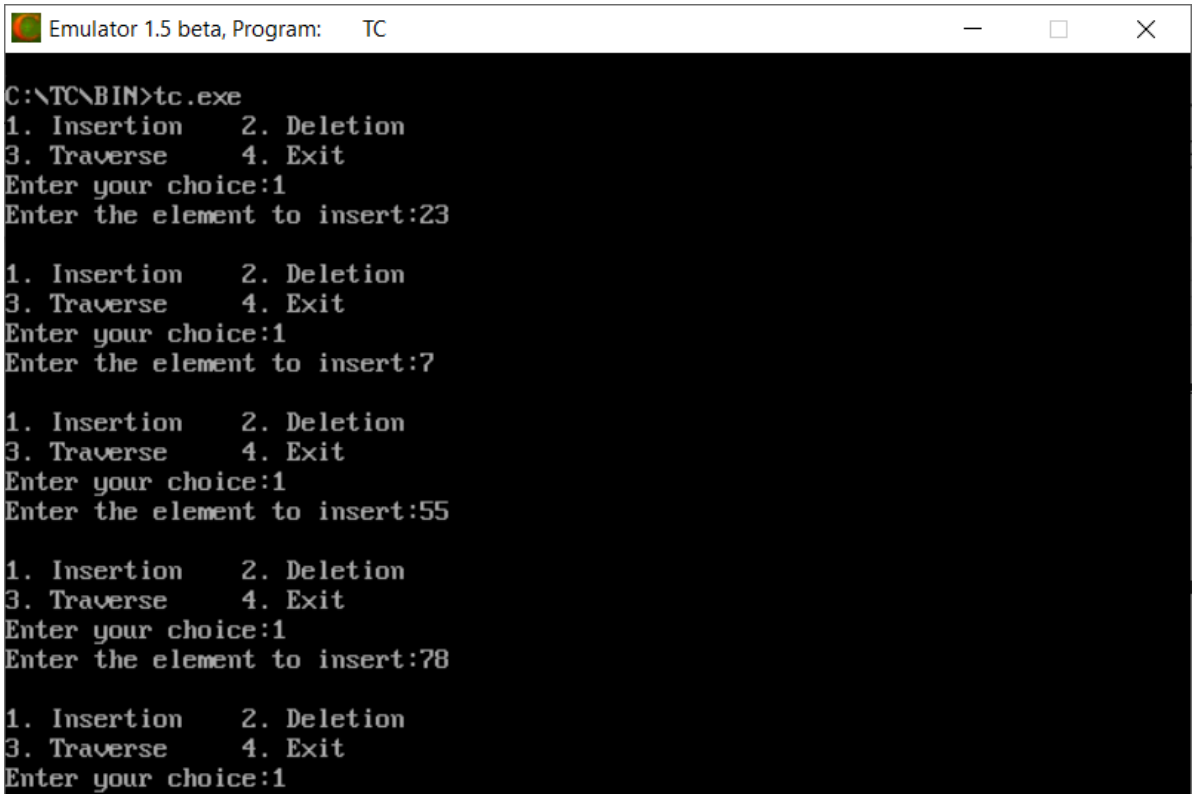
```
        }
        color = yPtr->color;
        yPtr->color = ptr->color;
        ptr->color = color;
      }
  }
  if (ht < 1)
      return;
  if (ptr->color == BLACK){
      while (1){
        pPtr = stack[ht - 1]->link[dir[ht - 1]];
        if (pPtr && pPtr->color == RED){
          pPtr->color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

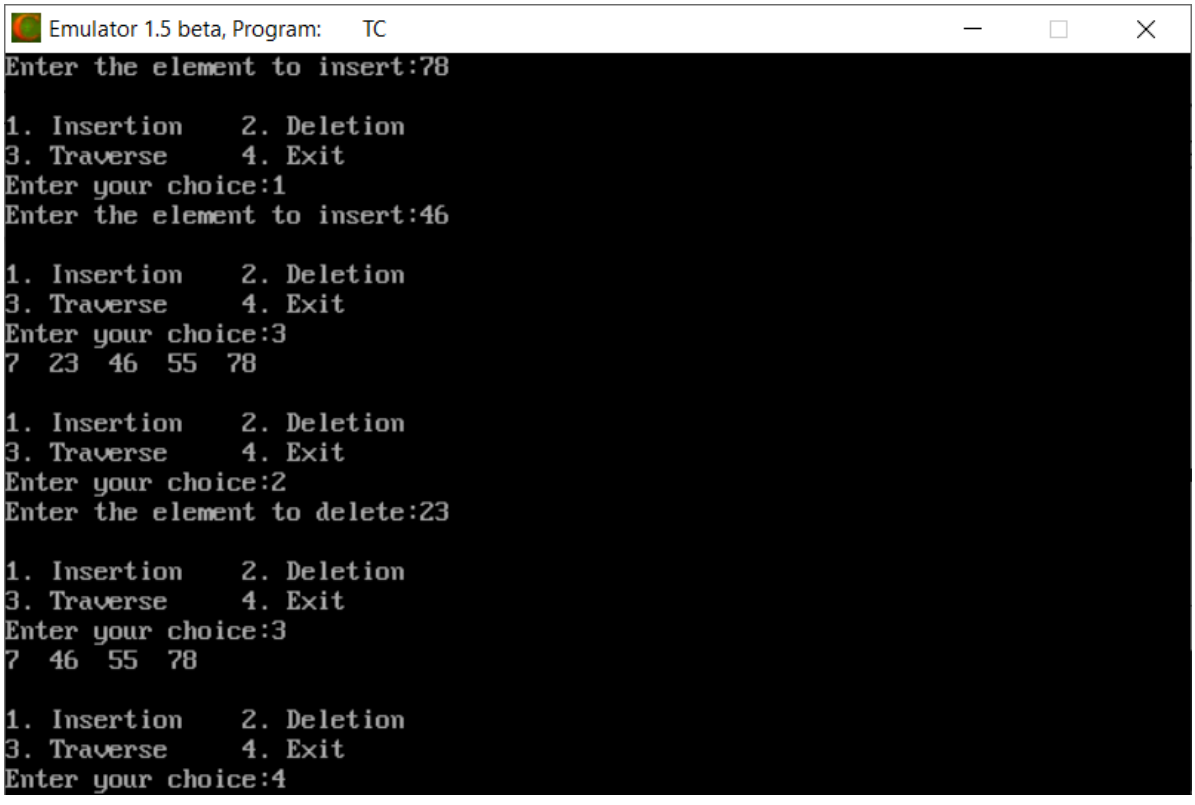
```
            qPtr = rPtr->link[0];
            rPtr->color = RED;
            qPtr->color = BLACK;
            rPtr->link[0] = qPtr->link[1];
            qPtr->link[1] = rPtr;
            rPtr = stack[ht - 1]->link[1] = qPtr;
        }
        rPtr->color = stack[ht - 1]->color;
        stack[ht - 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--;
      }
  }
}
void inorderTraversal(struct rbNode *node) {
  if (node) {
      inorderTraversal(node->link[0]);
      printf("%d  ", node->data);
      inorderTraversal(node->link[1]);
  }
  return;
}
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:


Emulator 1.5 beta, Program: TC
C:\TC\BIN>tc.exe
1. Insertion 2. Deletion
3. Traverse 4. Exit
Enter your choice:1
Enter the element to insert:23
1. Insertion 2. Deletion
3. Traverse 4. Exit
Enter your choice:1
Enter the element to insert:7
1. Insertion 2. Deletion
3. Traverse 4. Exit
Enter your choice:1
Enter the element to insert:55
1. Insertion 2. Deletion
3. Traverse 4. Exit
Enter your choice:1
Enter the element to insert:78
1. Insertion 2. Deletion
3. Traverse 4. Exit
Enter your choice:1


Emulator 1.5 beta, Program: TC
Enter the element to insert:78
1. Insertion 2. Deletion
3. Traverse 4. Exit
Enter your choice:1
Enter the element to insert:46
1. Insertion 2. Deletion
3. Traverse 4. Exit
Enter your choice:3
7 23 46 55 78
1. Insertion 2. Deletion
3. Traverse 4. Exit
Enter your choice:2
Enter the element to delete:23
1. Insertion 2. Deletion
3. Traverse 4. Exit
Enter your choice:3
7 46 55 78
1. Insertion 2. Deletion
3. Traverse 4. Exit
Enter your choice:4

Practical :- 6

AIM :- Implement 2-3 Tree.

Code :-

```
#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;
/* creating new 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;
}
/* Places the value in appropriate position */
void addValToNode(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 the 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) {
                addValToNode(val, pos, node, child);
        } else {
                addValToNode(val, pos - median, *newNode, child);
        }
        *pval = node->val[node->count];
        (*newNode)->link[0] = node->link[node->count];
        node->count--;
}
/* sets the value val in the node */
int setValueInNode(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 not allowed\n");
                                return 0;
                        }
                }
        if (setValueInNode(val, pval, node->link[pos], child)) {
                if (node->count < MAX) {
                        addValToNode(*pval, pos, node, *child);
                        } else {
                                splitNode(*pval, pval, pos, node, *child, child);
                                return 1;
                }
```

```
        }
        return 0;
}
/* insert val in B-Tree */
void insertion(int val) {
        int flag, i;
        struct btreeNode *child;
        flag = setValueInNode(val, &i, root, &child);
                if (flag)
        root = createNode(i, child);
}
/* copy successor for the value to be deleted */
void copySuccessor(struct btreeNode *myNode, int pos) {
        struct btreeNode *dummy;
        dummy = myNode->link[pos];
        for (;dummy->link[0] != NULL;)
        dummy = dummy->link[0];
        myNode->val[pos] = dummy->val[1];
}
/* removes the value from the given node and rearrange values */
void removeVal(struct btreeNode *myNode, int pos) {
        int i = pos + 1;
        while (i <= myNode->count) {
                myNode->val[i - 1] = myNode->val[i];
                myNode->link[i - 1] = myNode->link[i];
                i++;
        }
        myNode->count--;
}
/* shifts value from parent to right child */
void doRightShift(struct btreeNode *myNode, int pos) {
        struct btreeNode *x = myNode->link[pos];
        int j = x->count;
        while (j > 0) {
                x->val[j + 1] = x->val[j];
                x->link[j + 1] = x->link[j];
        }
        x->val[1] = myNode->val[pos];
        x->link[1] = x->link[0];
        x->count++;
        x = myNode->link[pos - 1];
        myNode->val[pos] = x->val[x->count];
        myNode->link[pos] = x->link[x->count];
        x->count--;
```

```
        return;
}
/* shifts value from parent to left child */
void doLeftShift(struct btreeNode *myNode, int pos) {
        int j = 1;
        struct btreeNode *x = myNode->link[pos - 1];
        x->count++;
        x->val[x->count] = myNode->val[pos];
        x->link[x->count] = myNode->link[pos]->link[0];
        x = myNode->link[pos];
        myNode->val[pos] = x->val[1];
        x->link[0] = x->link[1];
        x->count--;
        while (j <= x->count) {
                x->val[j] = x->val[j + 1];
                x->link[j] = x->link[j + 1];
                j++;
        }
        return;
}
/* merge nodes */
void mergeNodes(struct btreeNode *myNode, int pos) {
        int j = 1;
        struct btreeNode *x1 = myNode->link[pos], *x2 = myNode->link[pos - 1];
        x2->count++;
        x2->val[x2->count] = myNode->val[pos];
        x2->link[x2->count] = myNode->link[0];
        while (j <= x1->count) {
                x2->count++;
                x2->val[x2->count] = x1->val[j];
                x2->link[x2->count] = x1->link[j];
                j++;
        }
        j = pos;
        while (j < myNode->count) {
                myNode->val[j] = myNode->val[j + 1];
                myNode->link[j] = myNode->link[j + 1];
                j++;
        }
        myNode->count--;
        free(x1);
}
/* adjusts the given node */
void adjustNode(struct btreeNode *myNode, int pos) {
```

```
        if (!pos) {
                if (myNode->link[1]->count > MIN) {
                        doLeftShift(myNode, 1);
                } else {
                        mergeNodes(myNode, 1);
                }
        } else {
                if (myNode->count != pos) {
                        if(myNode->link[pos - 1]->count > MIN) {
                                doRightShift(myNode, pos);
                        } else {
                                if (myNode->link[pos + 1]->count > MIN) {
                                        doLeftShift(myNode, pos + 1);
                                } else {
                                                        mergeNodes(myNode, pos);
                                }
                        }
                } else {
                        if (myNode->link[pos - 1]->count > MIN)
                        doRightShift(myNode, pos);
                        else
                        mergeNodes(myNode, pos);
                }
        }
}
/* delete val from the node */
int delValFromNode(int val, struct btreeNode *myNode) {
        int pos, flag = 0;
        if (myNode) {
                if (val < myNode->val[1]) {
                        pos = 0;
                        flag = 0;
                } else {
                        for (pos = myNode->count;
                        (val < myNode->val[pos] && pos > 1); pos--);
                        if (val == myNode->val[pos]) {
                                flag = 1;
                                } else {
                                        flag = 0;
                        }
                }
                if (flag) {
                        if (myNode->link[pos - 1]) {
                                copySuccessor(myNode, pos);
```

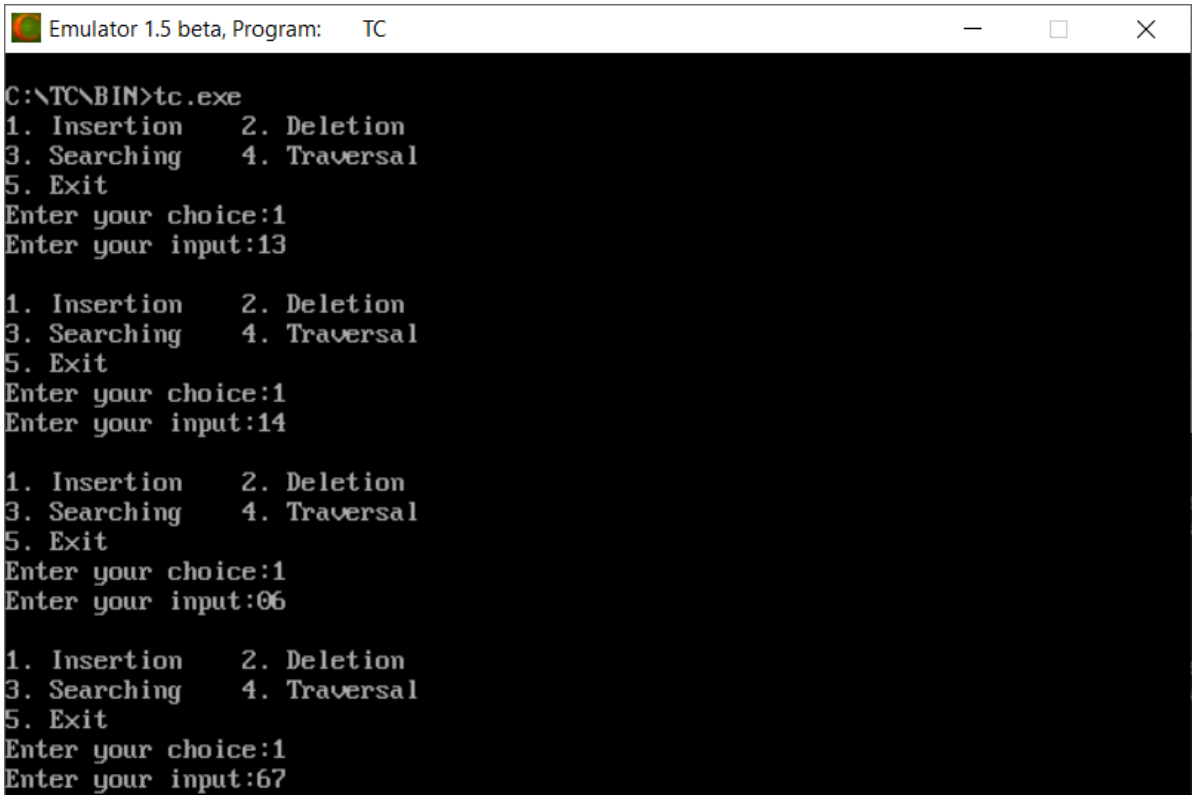
```
                                flag = delValFromNode(myNode->val[pos], myNode->link[pos]);
                                if (flag == 0) {
                                printf("Given data is not present in B-Tree\n");
                                }
                        } else {
                                removeVal(myNode, pos);
                        }
                } else {
                        flag = delValFromNode(val, myNode->link[pos]);
                }
                if (myNode->link[pos]) {
                        if (myNode->link[pos]->count < MIN)
                        adjustNode(myNode, pos);
                }
        }
        return flag;
}
/* delete val from B-tree */
void deletion(int val, struct btreeNode *myNode) {
        struct btreeNode *tmp;
                if (!delValFromNode(val, myNode)) {
                        printf("Given value is not present in B-Tree\n");
                        return;
                } else {
                        if (myNode->count == 0) {
                        tmp = myNode;
                        myNode = myNode->link[0];
                        free(tmp);
                }
        }
        root = myNode;
        return;
}
/* search val in B-Tree */
void searching(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("Given data %d is present in B-Tree", val);
                        return;
                }
        }
        searching(val, pos, myNode->link[*pos]);
        return;
}
/* B-Tree Traversal */
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;
        while (1){
                printf("1. Insertion\t2. Deletion\n");
                printf("3. Searching\t4. Traversal\n");
                printf("5. Exit\nEnter your choice:");
                scanf("%d", &ch);
                switch (ch) {
                        case 1:
                        printf("Enter your input:");
                        scanf("%d", &val);
                        insertion(val);
                        break;
                        case 2:
                        printf("Enter the element to delete:");
                        scanf("%d", &val);
                        deletion(val, root);
                        break;
                        case 3:
                        printf("Enter the element to search:");
                        scanf("%d", &val);
                        searching(val, &ch, root);
                        break;
```

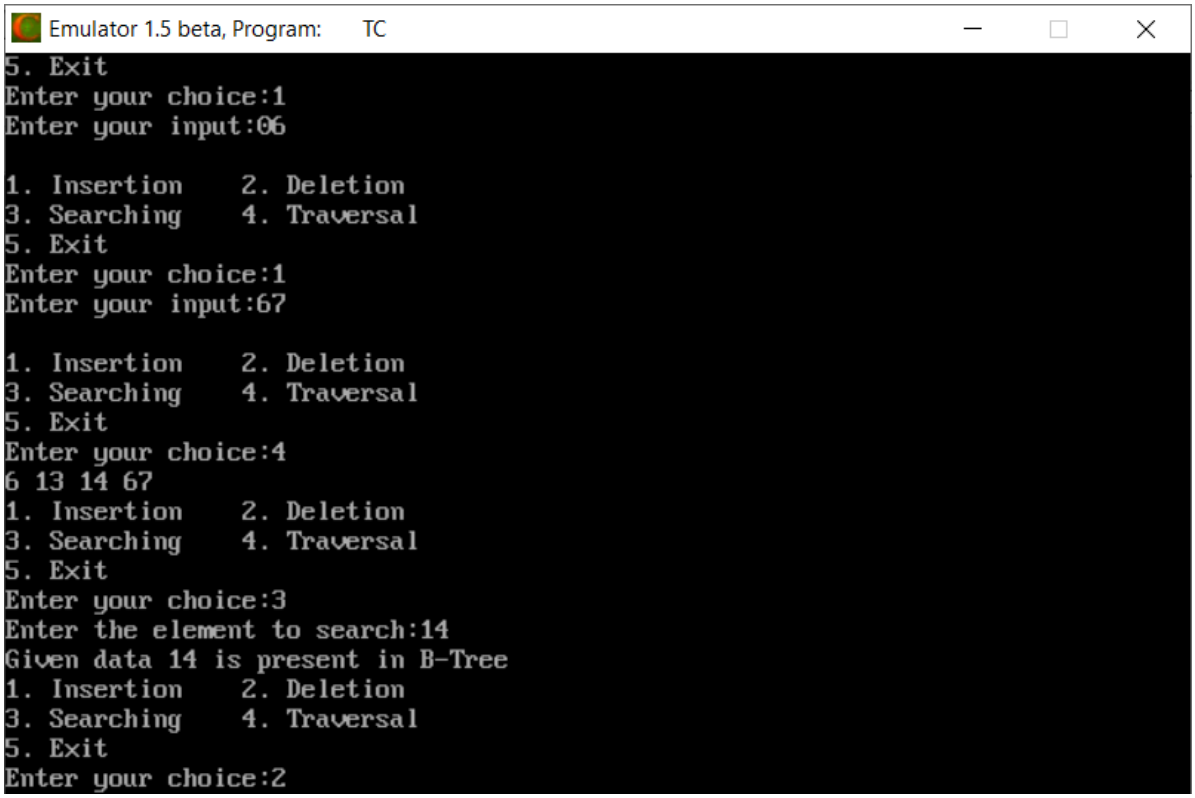
```
                    case 4:
                    traversal(root);
                    break;
                    case 5:
                    exit(0);
                    default:
                    printf("U have entered wrong option!!\n");
                    break;
            }
            printf("\n");
        }
}
```

OUTPUT:

1) **INSERT DATA**

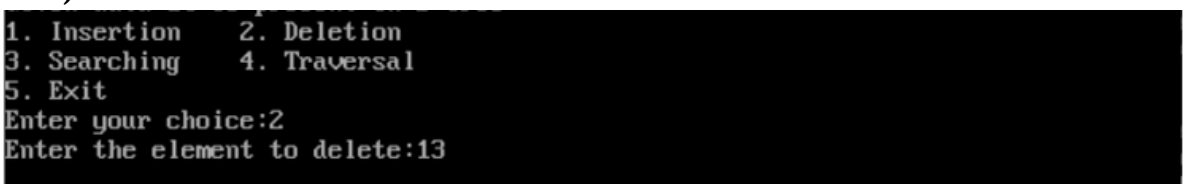


2) **TRAVERSAL**



3) SEARCHING DATA

4) DELETE DATA



Practical :- 7

AIM :- Implement B Tree.

Code :-

```
#include <stdio.h>
#include <stdlib.h>
//#include<conio.h>
#define M 3
typedef struct _node {
	int    n;
	int            keys[M - 1];
	struct _node *p[M];
} node;
node *root = NULL;

typedef enum KeyStatus {
	Duplicate,
	SearchFailure,
	Success,
	InsertIt,
	LessKeys,
} KeyStatus;
void insert(int key);
void display(node *root, int);
void DelNode(int x);
void search(int x);
KeyStatus ins(node *r, int x, int* y, node** u);
int searchPos(int x, int *key_arr, int n);
KeyStatus del(node *r, int x);
void eatline(void);
void inorder(node *ptr);
int totalKeys(node *ptr);
void printTotal(node *ptr);
int getMin(node *ptr);
int getMax(node *ptr);
void getMinMax(node *ptr);
int max(int first, int second, int third);
int maxLevel(node *ptr);
void printMaxLevel(node *ptr);
int main() {
	int key;
	int choice;
	printf("Creation of B tree for M=%d\n", M);
	while (1) {
```

```
printf("1.Insert\n");
printf("2.Delete\n");
printf("3.Search\n");
printf("4.Display\n");
printf("5.Quit\n");
printf("6.Enumerate\n");
printf("7.Total Keys\n");
printf("8.Min and Max Keys\n");
printf("9.Max Level\n");
printf("Enter your choice : ");
scanf("%d", &choice); eatline();
switch (choice) {
case 1:
        printf("Enter the key : ");
        scanf("%d", &key); eatline();
        insert(key);
        break;
case 2:
        printf("Enter the key : ");
        scanf("%d", &key); eatline();
        DelNode(key);
        break;
case 3:
        printf("Enter the key : ");
        scanf("%d", &key); eatline();
        search(key);
        break;
case 4:
        printf("Btree is :\n");
        display(root, 0);
        break;
case 5:
        exit(1);
case 6:
        printf("Btree in sorted order is:\n");
        inorder(root); putchar('\n');
        break;
case 7:
        printf("The total number of keys in this tree is:\n");
        printTotal(root);
        break;
case 8:
        getMinMax(root);
        break;
```

```
            case 9:
                printf("The maximum level in this tree is:\n");
                printMaxLevel(root);
                break;
            default:
                printf("Wrong choice\n");
                break;
            }
        }
        return 0;
}
void insert(int key) {
        node *newnode;
        int upKey;
        KeyStatus value;
        value = ins(root, key, &upKey, &newnode);
        if (value == Duplicate)
            printf("Key already available\n");
        if (value == InsertIt) {
            node *uproot = root;
            root = (node*)malloc(sizeof(node));
            root->n = 1;
            root->keys[0] = upKey;
            root->p[0] = uproot;
            root->p[1] = newnode;
        }
}
KeyStatus ins(node *ptr, int key, int *upKey, node **newnode) {
        node *newPtr, *lastPtr;
        int pos, i, n, splitPos;
        int newKey, lastKey;
        KeyStatus value;
        if (ptr == NULL) {
            *newnode = NULL;
            *upKey = key;
            return InsertIt;
        }
        n = ptr->n;
        pos = searchPos(key, ptr->keys, n);
        if (pos < n && key == ptr->keys[pos])
            return Duplicate;
        value = ins(ptr->p[pos], key, &newKey, &newPtr);
        if (value != InsertIt)
            return value;
```

```
        if (n < M - 1) {
                pos = searchPos(newKey, ptr->keys, n);
                        for (i = n; i>pos; i--) {
                        ptr->keys[i] = ptr->keys[i - 1];
                        ptr->p[i + 1] = ptr->p[i];
                }
                      ptr->keys[pos] = newKey;
                ptr->p[pos + 1] = newPtr;
                ++ptr->n;
                return Success;
        }/*End of if */
              if (pos == M - 1) {
                lastKey = newKey;
                lastPtr = newPtr;
        }
        else {
                lastKey = ptr->keys[M - 2];
                lastPtr = ptr->p[M - 1];
                for (i = M - 2; i>pos; i--) {
                        ptr->keys[i] = ptr->keys[i - 1];
                        ptr->p[i + 1] = ptr->p[i];
                }
                ptr->keys[pos] = newKey;
                ptr->p[pos + 1] = newPtr;
        }
        splitPos = (M - 1) / 2;
        (*upKey) = ptr->keys[splitPos];
        (*newnode) = (node*)malloc(sizeof(node));
        ptr->n = splitPos;
        (*newnode)->n = M - 1 - splitPos;
        for (i = 0; i < (*newnode)->n; i++) {
                (*newnode)->p[i] = ptr->p[i + splitPos + 1];
                if (i < (*newnode)->n - 1)
                        (*newnode)->keys[i] = ptr->keys[i + splitPos + 1];
                else
                        (*newnode)->keys[i] = lastKey;
        }
        (*newnode)->p[(*newnode)->n] = lastPtr;
        return InsertIt;
}/*End of ins()*/
void display(node *ptr, int blanks) {
        if (ptr) {
                int i;
                for (i = 1; i <= blanks; i++)
```

```
            printf(" ");
        for (i = 0; i < ptr->n; i++)
            printf("%d ", ptr->keys[i]);
        printf("\n");
        for (i = 0; i <= ptr->n; i++)
            display(ptr->p[i], blanks + 10);
    }/*End of if*/
}/*End of display()*/
void search(int key) {
    int pos, i, n;
    node *ptr = root;
    printf("Search path:\n");
    while (ptr) {
        n = ptr->n;
        for (i = 0; i < ptr->n; i++)
            printf(" %d", ptr->keys[i]);
        printf("\n");
        pos = searchPos(key, ptr->keys, n);
        if (pos < n && key == ptr->keys[pos]) {
            printf("Key %d found in position %d of last dispalyed node\n", key, i);
            return;
        }
        ptr = ptr->p[pos];
    }
    printf("Key %d is not available\n", key);
}/*End of search()*/
int searchPos(int key, int *key_arr, int n) {
    int pos = 0;
    while (pos < n && key > key_arr[pos])
        pos++;
    return pos;
}/*End of searchPos()*/
void DelNode(int key) {
    node *uproot;
    KeyStatus value;
    value = del(root, key);
    switch (value) {
    case SearchFailure:
        printf("Key %d is not available\n", key);
        break;
    case LessKeys:
        uproot = root;
        root = root->p[0];
        free(uproot);
```

```
            break;
        default:
            return;
        }/*End of switch*/
    }/*End of delnode()*/
    KeyStatus del(node *ptr, int key) {
        int pos, i, pivot, n, min;
        int *key_arr;
        KeyStatus value;
        node **p, *lptr, *rptr;

        if (ptr == NULL)
            return SearchFailure;
        /*Assigns values of node*/
        n = ptr->n;
        key_arr = ptr->keys;
        p = ptr->p;
        min = (M - 1) / 2;/*Minimum number of keys*/
        pos = searchPos(key, key_arr, n);
        // p is a leaf
        if (p[0] == NULL) {
            if (pos == n || key < key_arr[pos])
                return SearchFailure;
            /*Shift keys and pointers left*/
            for (i = pos + 1; i < n; i++)
            {
                key_arr[i - 1] = key_arr[i];
                p[i] = p[i + 1];
            }
            return --ptr->n >= (ptr == root ? 1 : min) ? Success : LessKeys;
        }/*End of if */
        if (pos < n && key == key_arr[pos]) {
            node *qp = p[pos], *qp1;
            int nkey;
            while (1) {
                nkey = qp->n;
                qp1 = qp->p[nkey];
                if (qp1 == NULL)
                    break;
                qp = qp1;
            }/*End of while*/
            key_arr[pos] = qp->keys[nkey - 1];
            qp->keys[nkey - 1] = key;
        }/*End of if */
```

```
value = del(p[pos], key);
if (value != LessKeys)
        return value;
if (pos > 0 && p[pos - 1]->n > min) {
        pivot = pos - 1; /*pivot for left and right node*/
        lptr = p[pivot];
        rptr = p[pos];
                rptr->p[rptr->n + 1] = rptr->p[rptr->n];
        for (i = rptr->n; i>0; i--) {
                rptr->keys[i] = rptr->keys[i - 1];
                rptr->p[i] = rptr->p[i - 1];
        }
        rptr->n++;
        rptr->keys[0] = key_arr[pivot];
        rptr->p[0] = lptr->p[lptr->n];
        key_arr[pivot] = lptr->keys[--lptr->n];
        return Success;
}/*End of if */
 //if (posn > min)
if (pos < n && p[pos + 1]->n > min) {
        pivot = pos; /*pivot for left and right node*/
        lptr = p[pivot];
        rptr = p[pivot + 1];
        /*Assigns values for left node*/
        lptr->keys[lptr->n] = key_arr[pivot];
        lptr->p[lptr->n + 1] = rptr->p[0];
        key_arr[pivot] = rptr->keys[0];
        lptr->n++;
        rptr->n--;
        for (i = 0; i < rptr->n; i++) {
                rptr->keys[i] = rptr->keys[i + 1];
                rptr->p[i] = rptr->p[i + 1];
        }/*End of for*/
        rptr->p[rptr->n] = rptr->p[rptr->n + 1];
        return Success;
}/*End of if */

if (pos == n)
        pivot = pos - 1;
else
        pivot = pos;
lptr = p[pivot];
rptr = p[pivot + 1];
/*merge right node with left node*/
```

```
        lptr->keys[lptr->n] = key_arr[pivot];
        lptr->p[lptr->n + 1] = rptr->p[0];
        for (i = 0; i < rptr->n; i++) {
            lptr->keys[lptr->n + 1 + i] = rptr->keys[i];
            lptr->p[lptr->n + 2 + i] = rptr->p[i + 1];
        }
        lptr->n = lptr->n + rptr->n + 1;
        free(rptr); /*Remove right node*/
        for (i = pos + 1; i < n; i++) {
            key_arr[i - 1] = key_arr[i];
            p[i] = p[i + 1];
        }
        return --ptr->n >= (ptr == root ? 1 : min) ? Success : LessKeys;
    }/*End of del()*/
    void eatline(void) {
        char c;
        while ((c = getchar()) != '\n');
    }
    void inorder(node *ptr) {
        if (ptr) {
            if (ptr->n >= 1) {
                inorder(ptr->p[0]);
                printf("%d ", ptr->keys[0]);
                inorder(ptr->p[1]);
                if (ptr->n == 2) {
                    printf("%d ", ptr->keys[1]);
                    inorder(ptr->p[2]);
                }
            }
        }
    }
    int totalKeys(node *ptr) {
        if (ptr) {
            int count = 1;
            if (ptr->n >= 1) {
                count += totalKeys(ptr->p[0]);
                count += totalKeys(ptr->p[1]);
                if (ptr->n == 2) count += totalKeys(ptr->p[2]) + 1;
            }
            return count;
        }
        return 0;
    }
    void printTotal(node *ptr) {
```

```
	printf("%d\n", totalKeys(ptr));
}
int getMin(node *ptr) {
	if (ptr) {
		int min;
		if (ptr->p[0] != NULL) min = getMin(ptr->p[0]);
		else min = ptr->keys[0];
		return min;
	}
	return 0;
}
int getMax(node *ptr) {
	if (ptr) {
		int max;
		if (ptr->n == 1) {
			if (ptr->p[1] != NULL) max = getMax(ptr->p[1]);
			else max = ptr->keys[0];
		}
		if (ptr->n == 2) {
			if (ptr->p[2] != NULL) max = getMax(ptr->p[2]);
			else max = ptr->keys[1];
		}
		return max;
	}
	return 0;
}
void getMinMax(node *ptr) {
	printf("%d %d\n", getMin(ptr), getMax(ptr));
}
int max(int first, int second, int third) {
	int max = first;
	if (second > max) max = second;
	if (third > max) max = third;
	return max;
}
int maxLevel(node *ptr) {
	if (ptr) {
		int l = 0, mr = 0, r = 0, max_depth;
		if (ptr->p[0] != NULL) l = maxLevel(ptr->p[0]);
		if (ptr->p[1] != NULL) mr = maxLevel(ptr->p[1]);
		if (ptr->n == 2) {
			if (ptr->p[2] != NULL) r = maxLevel(ptr->p[2]);
		}
		max_depth = max(l, mr, r) + 1;
```

```
            return max_depth;
        }
        return 0;
}
void printMaxLevel(node *ptr) {
        int max = maxLevel(ptr) - 1;
        if (max == -1) printf("tree is empty\n");
        else printf("%d\n", max);
}
```

OUTPUT :

1) **INSERT DATA**

```
Creation of B tree for M=3
1.Insert
2.Delete
3.Search
4.Display
5.Quit
6.Enumerate
7.Total Keys
8.Min and Max Keys
9.Max Level
Enter your choice : 1
Enter the key : 34
1.Insert
2.Delete
3.Search
4.Display
5.Quit
6.Enumerate
7.Total Keys
8.Min and Max Keys
9.Max Level
Enter your choice : 1
Enter the key : 78
1.Insert
2.Delete
3.Search
4.Display
5.Quit
6.Enumerate
7.Total Keys
8.Min and Max Keys
9.Max Level
Enter your choice : 1
Enter the key : 3
```

```
Enter your choice : 1
Enter the key : 33
1.Insert
2.Delete
3.Search
4.Display
5.Quit
6.Enumerate
7.Total Keys
8.Min and Max Keys
9.Max Level
Enter your choice : 1
Enter the key : 67
1.Insert
2.Delete
3.Search
4.Display
5.Quit
6.Enumerate
7.Total Keys
8.Min and Max Keys
9.Max Level
Enter your choice : 1
Enter the key : 99
1.Insert
2.Delete
3.Search
4.Display
5.Quit
6.Enumerate
7.Total Keys
8.Min and Max Keys
9.Max Level
Enter your choice : 1
Enter the key : 79
```

2) DISPLAY DATA

```
1.Insert
2.Delete
3.Search
4.Display
5.Quit
6.Enumerate
7.Total Keys
8.Min and Max Keys
9.Max Level
Enter your choice : 4
Btree is :
34 78
          3 33
          67
          79 99
```

3) ENUMERATE DATA

```
1.Insert
2.Delete
3.Search
4.Display
5.Quit
6.Enumerate
7.Total Keys
8.Min and Max Keys
9.Max Level
Enter your choice : 6
Btree in sorted order is:
3 33 34 67 78 79 99
```

4) TOTAL KEYS

```
1.Insert
2.Delete
3.Search
4.Display
5.Quit
6.Enumerate
7.Total Keys
8.Min and Max Keys
9.Max Level
Enter your choice : 7
The total number of keys in this tree is:
7
```

5) MIN AND MAX KEYS

```
1.Insert
2.Delete
3.Search
4.Display
5.Quit
6.Enumerate
7.Total Keys
8.Min and Max Keys
9.Max Level
Enter your choice : 8
3 99
```

6) MAX LEVEL

```
1.Insert
2.Delete
3.Search
4.Display
5.Quit
6.Enumerate
7.Total Keys
8.Min and Max Keys
9.Max Level
Enter your choice : 9
The maximum level in this tree is:
2
```

7) SEARCH KEY

```
1.Insert
2.Delete
3.Search
4.Display
5.Quit
6.Enumerate
7.Total Keys
8.Min and Max Keys
9.Max Level
Enter your choice : 3
Enter the key : 33
Search path:
 67
 33
Key 33 found in position 1 of last dispalyed node
```

8) DELETE KEY

```
1.Insert
2.Delete
3.Search
4.Display
5.Quit
6.Enumerate
7.Total Keys
8.Min and Max Keys
9.Max Level
Enter your choice : 2
Enter the key : 3
1.Insert
2.Delete
3.Search
4.Display
5.Quit
6.Enumerate
7.Total Keys
8.Min and Max Keys
9.Max Level
Enter your choice : 4
Btree is :
67 79
          33 34
          78
          99
```

Practical :- 8

AIM :- Implement a program for String Matching using Boyer Moore Algorithm on a text file content.

Code :-

```
# include <limits.h>
# include <string.h>
# include <stdio.h>
# define NO_OF_CHARS 256
int max (int a, int b) { return (a > b)? a: b; }
void badCharHeuristic( char *str, int size, int badchar[NO_OF_CHARS]){
   int i;
         for (i = 0; i < NO_OF_CHARS; i++)
          badchar[i] = -1;
   for (i = 0; i < size; i++)
          badchar[(int) str[i]] = i;
}
void search( char *txt,  char *pat){
   int m = strlen(pat);
   int n = strlen(txt);
   int s=0;
   int badchar[NO_OF_CHARS];
      badCharHeuristic(pat, m, badchar);
   while(s <= (n - m)){
         int j = m-1;
         while(j >= 0 && pat[j] == txt[s+j])
            j--;
         if (j < 0){
            printf("\n pattern occurs at shift = %d", s);
            s += (s+m < n)? m-badchar[txt[s+m]] : 1;
         }
         else
            s += max(1, j - badchar[txt[s+j]]);
   }
}
void main(){
   char txt[] = "Hello How Are You!!";
   char pat[] = "Are";
   clrscr();
   search(txt, pat);
   getch();
}
```

OUTPUT:

Practical :- 9

AIM :- Implement a program for String Matching using Knuth-Morris-Pratt Algorithm on a text file content.

Code :-

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void computeLPSArray(char *pat, int M, int *lps);
void KMPSearch(char *pat, char *txt) {
  int M = strlen(pat);
  int N = strlen(txt);
 //create lps[] that will hold the longest prefix suffix values for pattern
  int *lps = (int *)malloc(sizeof(int) * (M + 1));
  int i=0;  //index for txt[]
  int j= 0;  //index for pat[]
 //Preprocess the pattern (calculate lps[] array)
  computeLPSArray(pat, M, lps);
//at this point i may be incremented while i < N && txt[i] != pat[0] - for performance
  while (i < N){
    if (pat[j] == txt[i]) {
      i++;
      j++;
      if (j == M)  {
        printf("Found pattern at index %d \n", i-j);
        j = lps[j];
      }
    }
    else {//if (pat[j] != txt[i]) //mismatch after j matches
        //Pattern shift
      j = lps[j];
      if (j < 0) {
        i++;
        j++;
      //at this point i may be incremented while i < N && txt[i] != pat[0] - for performance
      }
    }
  }
  free(lps); //to avoid memory leak
}
 void computeLPSArray(char *pat, int M, int *lps) {
  int i=1;
  int j=0;
  lps[0] = -1;
```

```
  while (i < M) {
    if (pat[j] == pat[i]) {
      lps[i] = lps[j];
      i++;
      j++;
    }
    else   { // (pat[j] != pat[i])
      lps[i] = j;
      j = lps[j];
      while (j >= 0 && pat[j] != pat[i]) {
            j = lps[j];
      }
      i++;
      j++;
    }
  }
  lps[i] = j;
}
// Driver program to test above function
int main() {
//          clrscr();
  char *txt = "hello how are you";
  char *pat = "you";
  KMPSearch(pat, txt);
  return 0;
}
```

OUTPUT:

```
Found pattern at index 14
```

Practical :- 10

AIM :- Implement Huffman-Coding Method. Show the result with suitable example.

Code :-

```
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
typedef struct node {
	char ch;
	int freq;
	struct node *left;
	struct node *right;
}node;
node * heap[100];
int heapSize=0;
void Insert(node * element) {
	heapSize++;
	heap[heapSize] = element;
	int now = heapSize;
	while(heap[now/2] -> freq > element -> freq) {
		heap[now] = heap[now/2];
		now /= 2;
	}
	heap[now] = element;
}
node * DeleteMin() {
	node * minElement,*lastElement;
	int child,now;
	minElement = heap[1];
	lastElement = heap[heapSize--];
	for(now = 1; now*2 <= heapSize ;now = child)  {
	    child = now*2;
		 if(child != heapSize && heap[child+1]->freq < heap[child] -> freq ) {
			child++;
		}
		if(lastElement -> freq > heap[child] -> freq) {
			heap[now] = heap[child];
		}
		else{
			break;
		}
	}
	heap[now] = lastElement;
	return minElement;
```

```
}
void print(node *temp,char *code) {
	if(temp->left==NULL && temp->right==NULL {
		printf("char %c code %s\n",temp->ch,code);
		return;
	}
	int length = strlen(code);
	char leftcode[10],rightcode[10];
	strcpy(leftcode,code);
	strcpy(rightcode,code);
	leftcode[length] = '0';
	leftcode[length+1] = '\0';
	rightcode[length] = '1';
	rightcode[length+1] = '\0';
	print(temp->left,leftcode);
	print(temp->right,rightcode);
}
int main(){
	heap[0] = (node *)malloc(sizeof(node));
	heap[0]->freq = 0;
	int n ;
	printf("Enter the no of characters: ");
	scanf("%d",&n);
	printf("Enter the characters and their frequencies: ");
	char ch;
	int freq,i;
	for(i=0;i<n;i++) {
		scanf(" %c",&ch);
		scanf("%d",&freq);
		node * temp = (node *) malloc(sizeof(node));
		temp -> ch = ch;
		temp -> freq = freq;
		temp -> left = temp -> right = NULL;
		Insert(temp);
	}
	if(n==1) {
		printf("char %c code 0\n",ch);
		return 0;
	}
	for(i=0;i<n-1 ;i++) {
		node * left = DeleteMin();
		node * right = DeleteMin();
		node * temp = (node *) malloc(sizeof(node));
		temp -> ch = 0;
```

```
            temp -> left = left;
            temp -> right = right;
            temp -> freq = left->freq + right -> freq;
            Insert(temp);
        }
        node *tree = DeleteMin();
        char code[10];
        code[0] = '\0';
        print(tree,code);
}
```

OUTPUT:

```
Enter the no of characters: 5
Enter the characters and their frequencies: z 1 y 2 x 4 w 3 v 5
char w code 00
char z code 010
char y code 011
char x code 10
char v code 11
```

Practical :- 11

AIM :- Write a program which creates Skip Lists. Implement Insert, Search and Update Operations in Skip-Lists.

Code :-

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
typedef struct node np;
        struct node{
        int data;
        np *up;
        np *down;
        np *left;
        np *right;
};
np *list=NULL;
int height=1,width=1;
int toss_coin();
np *createNode();
void print_sl();
void toss_it(np *x);
int search_sl(int item);
int delete_sl(int item);
void main(){
        srand(time(NULL));
        //option for choice
        int choice=1;
        while(choice!=0){
                printf("\nOption for operation:\n 0)Exit 1)insert 2)search 3)print List 4)delete\nOption: ");
                scanf("%d",&choice);
                while(choice<0 || choice>4){
                        printf("Select correct option: ");
                        scanf("%d",&choice);
                }
                int data;
                switch(choice){
                        case 1:
                        printf("\nEnter a nonnegative value: ");
                        scanf("%d",&data);
                        if(insert_sl(data)==1){

                        printf("\n%d was inserted successfully\n",data);
```

```
                        }
                        else{
                        printf("\n%d was not inserted!!\n",data);
                        }
                        break;
                        case 2:
                        printf("\nEnter a value to search: ");
                        scanf("%d",&data);
                        if(search_sl(data))
                        printf("\n%d has been found.\n",data);
                        else
                        printf("\n%d is not available.\n",data);
                        break;
                        case 3:
                        print_sl();
                        break;
                        case 4:
                        printf("\nEnter a value to delete: ");
                        scanf("%d",&data);
                        if(delete_sl(data))
                        printf("\n%d has been deleted.\n",data);
                        else
                        printf("\n%d is not available.\n",data);
                        break;
                }
        }
}
int delete_sl(int item){
np *curr=list,*temp;
int down_count=0;
while(curr!=NULL){
                temp=curr;
                if(curr->right && curr->right->data<item){
                        curr=curr->right;
                        printf("Right-");
                }
        else if(curr->right && curr->right->data==item){
                np *node=curr->right;
                while(node) {//go down one by one
                        if(node->left->data==-1 && node->right==NULL) {
                                down_count++;//count the level from uppermost level
                        }
                        if(node->right){
                                node->right->left=node->left;
```

```
                        node->left->right=node->right;
                }
                else{
                        node->left->right=NULL;
                }
                np *nd=node->down;
                free(node);
                node=nd;
        }
        //update the width and height
        width--;
        height=height-down_count;
        return 1;
    }
        else {
                curr=curr->down;
                printf("Down-");
        }
    }
    return 0;
    printf("\n");
}
int search_sl(int item){
    np *curr=list,*temp;
    while(curr!=NULL){
        temp=curr;
        if(curr->right && curr->right->data<item){
                curr=curr->right;
                printf("Right-");
        }
        else if(curr->right && curr->right->data==item)return 1;
        else {
                curr=curr->down;
                printf("Down-");
        }
    }
    return 0;
    printf("\n");
}
int insert_sl(int item){
    if(item<0)return 0;
    if(!list){//for the first item
                        list=createNode();
                        np *newnode=createNode();
```

```
                        list->right=newnode;
                        newnode->left=list;
                        newnode->data=item;
                        //toss to go upper level
                        toss_it(list->right);
                        width++;
                return 1;
        }
        //if list is not empty, find the right position
        np *curr=list,*temp;
        while(curr!=NULL){
                temp=curr;
                if(curr->right && curr->right->data<item){
                        curr=curr->right;
                        printf("Right-");
                }
                else if(curr->right && curr->right->data==item)return 0;
                else {
                        curr=curr->down;
                        printf("Down-");
                }
        }
        printf("\n");
        np *newnode=createNode();
        newnode->data=item;
        if(temp->right==NULL){//when added at the right most
                temp->right=newnode;
                newnode->left=temp;
        }
        else{//when added between two nodes
                newnode->left=temp;
                newnode->right=temp->right;
                temp->right->left=newnode;
                temp->right=newnode;
        }
        toss_it(newnode);
        width++;
        return 1;
}
void toss_it(np *x){
        int h=1;
        while(toss_coin()){
                printf("\nToss Win");
                h++;
```

```
                        if(h>height){//create a new level
                                height=h;
                                np *ln=createNode();
                                ln->down=list;
                                list->up=ln;
                                list=ln;
                                //add the node to the new level
                                np *newnode=createNode();
                                ln->right=newnode;
                                newnode->data=x->data;
                                newnode->down=x;
                                newnode->left=ln;
                                x->up=newnode;
                                x=newnode;
                        }
                        else{//add the node to an existing level
                                np *temp=x->left;
                                while(temp->up==NULL){
                                        temp=temp->left;
                                }
                                temp=temp->up;
                                np *newnode=createNode();
                                newnode->data=x->data;
                                newnode->left=temp;
                                newnode->down=x;
                                temp->right=newnode;
                                x->up=newnode;
                                x=newnode;
                        }
                }
        }
np *createNode(){
np *newnode=(np *)malloc(sizeof(np));
newnode->data=-1;
newnode->left=NULL;
newnode->down=NULL;
newnode->up=NULL;
newnode->right=NULL;
return newnode;
}
void print_sl(){
        if(!list)return;
        int v[height][width],i,j;
        for(i=0;i<height;i++){
```

```
                for(j=0;j<width;j++){
                        v[i][j]=-1;
                }
        }
        np *base=list;
        for(base=list;base->down;base=base->down);
        j=0;
        for(;base;base=base->right) {
                i=height-1;
                np *goup=base;
                for(;goup;goup=goup->up){
                        v[i][j]=goup->data;
                        i--;
                }
                j++;
        }
        printf("\n");
        for(i=0;i<height;i++){
                for(j=1;j<width;j++){
                        if(v[i][j]==-1){
                                printf(" - ");
                        }
                        else{
                                printf("%3d",v[i][j]);
                        }
                }
                printf("\n");
        }
        printf("\n");
}
int toss_coin(){
        float t=(float)(rand()%100)/100;
        return t>0.5?1:0;
}
```

OUTPUT:

1) **INSERT DATA**

```
Option for operation:
 0)Exit 1)insert 2)search 3)print List 4)delete
Option: 1

Enter a nonnegative value: 34

34 was inserted successfully

Option for operation:
 0)Exit 1)insert 2)search 3)print List 4)delete
Option: 1

Enter a nonnegative value: 56
Right-Down-

56 was inserted successfully

Option for operation:
 0)Exit 1)insert 2)search 3)print List 4)delete
Option: 1

Enter a nonnegative value: 89
Right-Right-Down-

Toss Win
89 was inserted successfully
```

2) **DISPLAY DATA**

```
Option for operation:
 0)Exit 1)insert 2)search 3)print List 4)delete
Option: 1

Enter a nonnegative value: 47
Down-Right-Down-

47 was inserted successfully

Option for operation:
 0)Exit 1)insert 2)search 3)print List 4)delete
Option: 3

 -  -  -  89
 34 47 56 89
```

3) SEARCH DATA

```
Option for operation:
 0)Exit 1)insert 2)search 3)print List 4)delete
Option: 2

Enter a value to search: 47
Down-Right-
47 has been found.
```

4) DELETE DATA

```
Option for operation:
 0)Exit 1)insert 2)search 3)print List 4)delete
Option: 4

Enter a value to delete: 56
Down-Right-Right-
56 has been deleted.

Option for operation:
 0)Exit 1)insert 2)search 3)print List 4)delete
Option: 3

 -  -  89
 34 47 89
```