VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

ADAVANCED DATA STRUCTURES

Submitted by

LIKITHA B (1BM19CS079)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Oct 2022-Feb 2023

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

<u>CERTIFICATE</u>

This is to certify that the Lab work entitled "**ADVANCED DATA STRUCTURES**" carried out by **LIKITHA B(1BM19CS079),** who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2022-23. The Lab report has been approved as it satisfies the academic requirements in respect of Advanced Data structures Lab **- (20CS5PEADS)** work prescribed for the said degree.

Dr. Latha N R	**Dr. Jyothi S Nayak**
Professor	Professor and Head
Department of CSE	Department of CSE
BMSCE, Bengaluru	BMSCE, Bengaluru

`

Index Sheet

Course Outcome

CO1	Ability to analyze the usage of appropriate data structure for a given application.
CO2	Ability to design an efficient algorithm for performing operations on various advanced data structure.
CO3	Ability to apply the knowledge of hashing techniques.
CO4	Ability to conduct practical experiments to solve problems using an appropriate data structure.

PROGRAM 1: Write a program to implement memory efficient Doubly Linked List is called XOR Linked List or Memory Efficient as the list uses bitwise XOR operation to save space for one address.

```
#include <inttypes.h>

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

   int data;

   struct Node* nxp;

}Node;

struct Node* XOR(struct Node* a,struct Node* b)

{

   return (struct Node*)((uintptr_t)(a) ^ (uintptr_t)(b));

}

struct Node* insert(struct Node** head,int value, int position)

{

   if (*head == NULL) {

      if (position == 1) {

         struct Node* node = (struct Node*)malloc(sizeof(struct Node));

         node->data = value;

         node->nxp = XOR(NULL, NULL);

         *head = node;

      }

      else {

         printf("Invalid Position\n");

      }

   }

   else {

      int Pos = 1;

      struct Node* curr = *head;
```

```
    struct Node* prev = NULL;
    struct Node* next = XOR(prev, curr->nxp);
    while (next != NULL && Pos < position - 1) {
        prev = curr;
        curr = next;
        next = XOR(prev, curr->nxp);
        Pos++;
    }
    if (Pos == position - 1) {
        struct Node* node = (struct Node*)malloc(sizeof(struct Node));
        struct Node* temp = XOR(curr->nxp, next);
        curr->nxp = XOR(temp, node);
        if (next != NULL) {
            next->nxp = XOR(node, XOR(next->nxp, curr));
        }
        node->nxp = XOR(curr, next);
        node->data = value;
    }
    else if (position == 1) {
        struct Node* node = (struct Node*)malloc(sizeof(struct Node));
        curr->nxp = XOR(node, XOR(NULL, curr->nxp));
        node->nxp = XOR(NULL, curr);
        *head = node;
        node->data = value;
    }
    else {

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

```
    return *head;
}
void printList(struct Node** head)
{
    struct Node* curr = *head;
    struct Node* prev = NULL;
    struct Node* next;
    while (curr != NULL) {
        printf("%d ",  curr->data);
        next = XOR(prev, curr->nxp);
        prev = curr;
        curr = next;
    }
}
int delete(Node** head, int key) {
    if((*head) == NULL)
        return -1;
    if((*head)->data==key) {
        Node* next = XOR(NULL, (*head)->nxp);
        if(next!=NULL) {
            next->nxp = XOR(NULL,XOR(next->nxp, (*head)));
        }
        free(*head);
        *head = next;
        return 0;
    }
    struct Node* curr = *head;
    struct Node* prev = NULL;
    while(curr!=NULL && curr->data!=key){
        Node* temp = curr;
```

```
        curr = XOR(curr->nxp, prev);
        prev = temp;
    }
    if(curr==NULL) // key not found in list
        return -1;
    Node* prevPrev = XOR(prev->nxp, curr);
    Node* next = XOR(curr->nxp, prev);
    prev->nxp = XOR(prevPrev, next);
    if(next!=NULL)
        next->nxp = XOR(XOR(next->nxp, curr), prev);
    free(curr);
    return 0; // successful
}
int main()
{
    struct Node* head = NULL;
    int data,pos,ch,ret;
    while(1){
    printf("\nEnter your choice: 1. Insert 2. Display 3. Delete 4. Exit\n");
    scanf("%d",&ch);
    switch(ch){
    case 1: printf("Enter the value:");
    scanf("%d",&data);
    printf("Enter the position:");
    scanf("%d",&pos);
    insert(&head, data, pos);
    break;
    case 2: printList(&head);
    break;
    case 3:  printf("\nEnter data to delete from the list : ");
```

```
    scanf("%d", &data);
    ret = delete(&head, data);
    if(ret==-1)
     printf("\n%d not found in list", data);
    else
     printf("\n%d deleted from the list", data);
    break;
    case 4: exit(0);
    }
    }
}
```

OUTPUT:

```
Enter your choice: 1. Insert 2. Display 3. Delete 4. Exit
1
Enter the value:10
Enter the position:1

Enter your choice: 1. Insert 2. Display 3. Delete 4. Exit
1
Enter the value:20
Enter the position:2

Enter your choice: 1. Insert 2. Display 3. Delete 4. Exit
1
Enter the value:30
Enter the position:1

Enter your choice: 1. Insert 2. Display 3. Delete 4. Exit
2
30 10 20
Enter your choice: 1. Insert 2. Display 3. Delete 4. Exit
3

Enter data to delete from the list : 40

40 not found in list
Enter your choice: 1. Insert 2. Display 3. Delete 4. Exit
3

Enter data to delete from the list : 10

10 deleted from the list
Enter your choice: 1. Insert 2. Display 3. Delete 4. Exit
2
30 20
Enter your choice: 1. Insert 2. Display 3. Delete 4. Exit
```

PROGRAM 2: Write a program to perform insertion, deletion and searching operations on a skip list.

```
#include <stdlib.h>

#include  <stdio.h>

#include <limits.h>

#define SKIPLIST_MAX_LEVEL 6

typedef struct snode {

   int key;

   int value;

   struct snode **forward;

} snode;

typedef struct skiplist {

   int level;

   int size;

   struct snode *header;

} skiplist;

skiplist *skiplist_init(skiplist *list) {

   int i;

   snode *header = (snode *) malloc(sizeof(struct snode));

   list->header = header;

   header->key = INT_MAX;

   header->forward = (snode **) malloc(

         sizeof(snode*) * (SKIPLIST_MAX_LEVEL + 1));

   for (i = 0; i <= SKIPLIST_MAX_LEVEL; i++) {

      header->forward[i] = list->header;

   }

   list->level = 1;

   list->size = 0;


   return list;
```

```
}
static int rand_level() {
    int level = 1;
    while (rand() < RAND_MAX / 2 && level < SKIPLIST_MAX_LEVEL)
        level++;
    return level;
}
int skiplist_insert(skiplist *list, int key, int value) {
    snode *update[SKIPLIST_MAX_LEVEL + 1];
    snode *x = list->header;
    int i, level;
    for (i = list->level; i >= 1; i--) {
        while (x->forward[i]->key < key)
            x = x->forward[i];
        update[i] = x;
    }
    x = x->forward[1];
    if (key == x->key) {
        x->value = value;
        return 0;
    } else {
        level = rand_level();
        if (level > list->level) {
            for (i = list->level + 1; i <= level; i++) {
                update[i] = list->header;
            }
            list->level = level;
        }
        x = (snode *) malloc(sizeof(snode));
        x->key = key;
```

```
        x->value = value;
        x->forward = (snode **) malloc(sizeof(snode*) * (level + 1));
        for (i = 1; i <= level; i++) {
            x->forward[i] = update[i]->forward[i];
            update[i]->forward[i] = x;
        }
    }
    return 0;
}
snode *skiplist_search(skiplist *list, int key) {
    snode *x = list->header;
    int i;
    for (i = list->level; i >= 1; i--) {
        while (x->forward[i]->key < key)
            x = x->forward[i];
    }
    if (x->forward[1]->key == key) {
        return x->forward[1];
    } else {
        return NULL;
    }
    return NULL;
}
static void skiplist_node_free(snode *x) {
    if (x) {
        free(x->forward);
        free(x);
    }
}
int skiplist_delete(skiplist *list, int key) {
```

```
    int i;
    snode *update[SKIPLIST_MAX_LEVEL + 1];
    snode *x = list->header;
    for (i = list->level; i >= 1; i--) {
        while (x->forward[i]->key < key)
            x = x->forward[i];
        update[i] = x;
    }
    x = x->forward[1];
    if (x->key == key) {
        for (i = 1; i <= list->level; i++) {
            if (update[i]->forward[i] != x)
                break;
            update[i]->forward[1] = x->forward[i];
        }
        skiplist_node_free(x);
        while (list->level > 1 && list->header->forward[list->level]
                == list->header)
            list->level--;
        printf("Item deleted!\n");
    }
    else{
    printf("Element not found to delete\n");
}
}
static void skiplist_dump(skiplist *list) {
    snode *x = list->header;
    while (x && x->forward[1] != list->header) {
        printf("%d[%d]->", x->forward[1]->key, x->forward[1]->value);
        x = x->forward[1];
```

```
    }
    printf("NIL\n");
}
int main() {
    skiplist list;
    skiplist_init(&list);
    while(1){
        int ch;
        printf("Choose an option:\n");
        printf("1.Insert 2.Delete 3.Search 4.Display 5.Exit\n");
        scanf("%d",&ch);
        int elem;
        switch(ch){
            case 1: printf("Enter the element to be inserted:");
                    scanf("%d",&elem);
                    skiplist_insert(&list, elem, elem);
            break;
            case 2: printf("Enter the element to be deleted:");
                    scanf("%d",&elem);
                    skiplist_delete(&list, elem);
            break;
            case 3: printf("Enter the element to search:");
                    scanf("%d",&elem);
                    snode *x = skiplist_search(&list, elem);
                    if (x){
                        printf("key = %d, value = %d\n", elem, x->value);
                    }
                    else{
                        printf("key = %d, not fuound\n", elem);
                    }
```

```
        break;
        case 4: skiplist_dump(&list);
        break;
        case 5: exit(0);
    }}}
```

OUTPUT:

```
C:\Users\yathri\Desktop\skipl
Choose an option:
1.Insert 2.Delete 3.Search 4.Display 5.Exit
1
Enter the element to be inserted:2
Choose an option:
1.Insert 2.Delete 3.Search 4.Display 5.Exit
1
Enter the element to be inserted:4
Choose an option:
1.Insert 2.Delete 3.Search 4.Display 5.Exit
1
Enter the element to be inserted:5
Choose an option:
1.Insert 2.Delete 3.Search 4.Display 5.Exit
4
2[2]->4[4]->5[5]->NIL
Choose an option:
1.Insert 2.Delete 3.Search 4.Display 5.Exit
3
Enter the element to search:1
key = 1, not fuound
Choose an option:
1.Insert 2.Delete 3.Search 4.Display 5.Exit
3
Enter the element to search:4
key = 4, value = 4
Choose an option:
1.Insert 2.Delete 3.Search 4.Display 5.Exit
2
Enter the element to be deleted:2
Item deleted!
Choose an option:
1.Insert 2.Delete 3.Search 4.Display 5.Exit
4
4[4]->5[5]->NIL
Choose an option:
1.Insert 2.Delete 3.Search 4.Display 5.Exit
2
Enter the element to be deleted:8
Element not found to delete
```

PROGRAM 3: Find the number of Island using disjoint set

```
#include <bits/stdc++.h>
using namespace std;
class DisjointUnionSets
{
    vector<int> rank, parent;
    int n;
    public:
    DisjointUnionSets(int n)
    {
        rank.resize(n);
        parent.resize(n);
        this->n = n;
        makeSet();
    }
    void makeSet()
    {
        for (int i = 0; i < n; i++)
            parent[i] = i;
    }
    int find(int x)
    {
        if (parent[x] != x)
        {
            return find(parent[x]);
        }
        return x;
    }
```

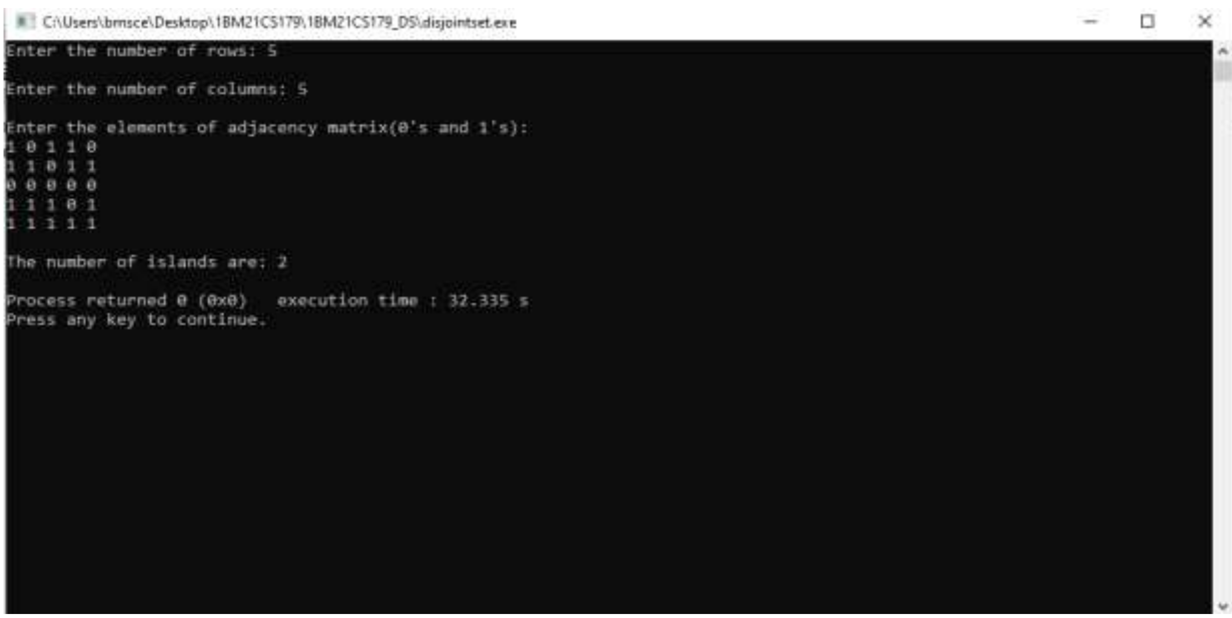
```
    void Union(int x, int y)
    {
        int xRoot = find(x);
        int yRoot = find(y);
        if (xRoot == yRoot)
            return;
        if (rank[xRoot] < rank[yRoot])
            parent[xRoot] = yRoot;
        else if (rank[yRoot] < rank[xRoot])
            parent[yRoot] = xRoot;
        else
        {
            parent[yRoot] = xRoot;
            rank[xRoot] = rank[xRoot] + 1;
        }
    }
};
int countIslands(vector<vector<int>>a)
{
    int n = a.size();
    int m = a[0].size();
    DisjointUnionSets *dus = new DisjointUnionSets(n * m);
    for (int j = 0; j < n; j++)
    {
        for (int k = 0; k < m; k++)
        {
            if (a[j][k] == 0)
                continue;
            if (j + 1 < n && a[j + 1][k] == 1)
```

```
            dus->Union(j * (m) + k,
                       (j + 1) * (m) + k);
        if (j - 1 >= 0 && a[j - 1][k] == 1)
            dus->Union(j * (m) + k,
                       (j - 1) * (m) + k);
        if (k + 1 < m && a[j][k + 1] == 1)
            dus->Union(j * (m) + k,
                       (j) * (m) + k + 1);
        if (k - 1 >= 0 && a[j][k - 1] == 1)
            dus->Union(j * (m) + k,
                       (j) * (m) + k - 1);
        if (j + 1 < n && k + 1 < m &&
                a[j + 1][k + 1] == 1)
            dus->Union(j * (m) + k,
                       (j + 1) * (m) + k + 1);
        if (j + 1 < n && k - 1 >= 0 &&
                a[j + 1][k - 1] == 1)
            dus->Union(j * m + k,
                       (j + 1) * (m) + k - 1);
        if (j - 1 >= 0 && k + 1 < m &&
                 a[j - 1][k + 1] == 1)
            dus->Union(j * m + k,
                       (j - 1) * m + k + 1);
        if (j - 1 >= 0 && k - 1 >= 0 &&
                 a[j - 1][k - 1] == 1)
            dus->Union(j * m + k,
                       (j - 1) * m + k - 1);
    }
}
```

```
    int *c = new int[n * m];
    int numberOfIslands = 0;
    for (int j = 0; j < n; j++)
    {
        for (int k = 0; k < m; k++)
        {
            if (a[j][k] == 1)
            {
                int x = dus->find(j * m + k);
                if (c[x] == 0)
                {
                    numberOfIslands++;
                    c[x]++;
                }
                else
                    c[x]++;
            }
        }
    }
    return numberOfIslands;
}
int main(void)
{
    vector<vector<int>>a = {{1, 1, 0, 0, 0},
                            {0, 1, 0, 0, 1},
                            {1, 0, 0, 1, 1},
                            {0, 0, 0, 0, 0},
                            {1, 0, 1, 0, 1}};
     cout<<"Given input"<<endl;
     for(int i=0;i<a.size();i++)
```

```
   {
    for(int j=0;j<a[0].size();j++)
    {
    cout<<a[i][j]<<" ";
     }
     cout<<endl;
    }
  cout << "Number of Islands is: "
       << countIslands(a) << endl;
}
```

OUTPUT:



PROGRAM 4: Write a program to perform insertion and deletion operations on AVL trees.

```
#include <stdio.h>
#include <stdlib.h>
#define count 10
struct Node {
  int key;
  struct Node *left;
  struct Node *right;
  int height;
};
int max(int a, int b);
int height(struct Node *N) {
  if (N == NULL)
    return 0;
  return N->height;
}
int max(int a, int b) {
  return (a > b) ? a : b;
}
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);
}
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;
}
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;
}
int getBalance(struct Node *N) {
  if (N == NULL)
    return 0;
  return height(N->left) - height(N->right);
}
struct Node *insertNode(struct Node *node, int key) {
  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
```

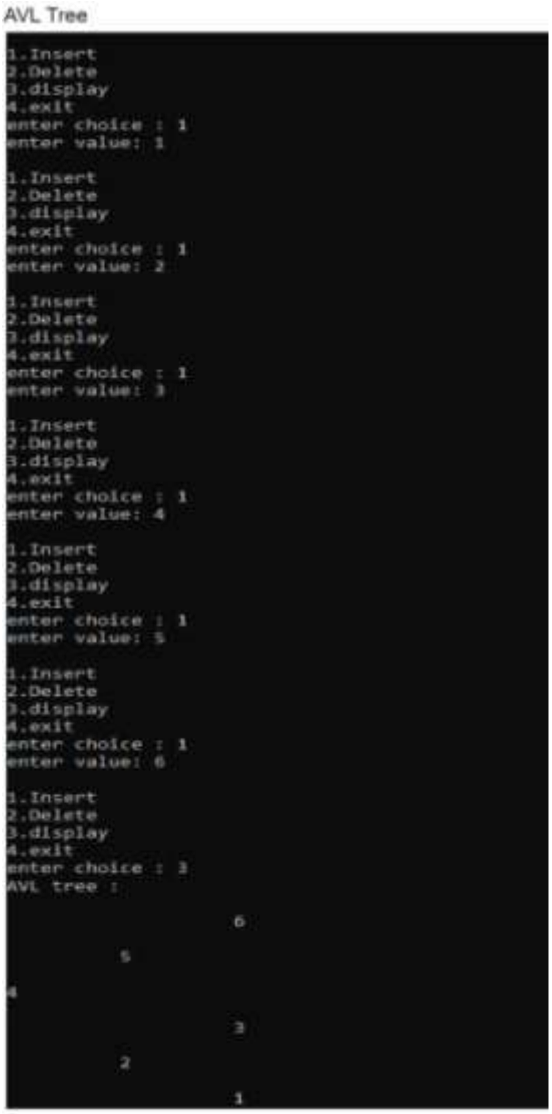
```
    return node;
  node->height = 1 + max(height(node->left),
              height(node->right));
  int balance = getBalance(node);
  if (balance > 1 && key < node->left->key)
    return rightRotate(node);
  if (balance < -1 && key > node->right->key)
    return leftRotate(node);
  if (balance > 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;
}
struct Node *deleteNode(struct Node *root, int key) {
  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;
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);
```

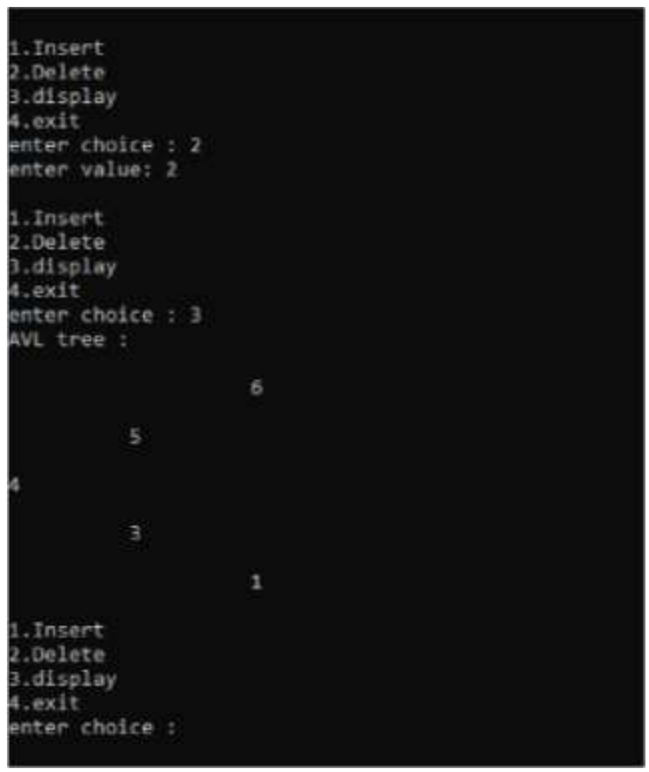
```
  if (balance < -1 && getBalance(root->right) > 0) {
    root->right = rightRotate(root->right);
    return leftRotate(root);
  }
  return root;
}
void print2DUtil(struct Node* root, int space)
{
    if (root == NULL)
        return;
    space += count;
    print2DUtil(root->right, space);
    printf("\n");
    for (int i = count; i < space; i++)
        printf(" ");
    printf("%d\n", root->key);
    print2DUtil(root->left, space);
}
void print2D(struct Node* root)
{
    print2DUtil(root, 0);
}
int main()
{
    struct Node *root = NULL;
    int choice;
    int key;
    while(1)
    {
```

```
    printf("\n1.Insert\n");
    printf("2.Delete\n");
    printf("3.display\n");
    printf("4.exit\n");
    printf("enter choice : ");
    scanf("%d",&choice);
    switch(choice)
    {
    case 1:
       printf("enter value: ");
        scanf("%d",&key);
        root=insertNode(root, key);
        break;
    case 2:
        printf("enter value: ");
        scanf("%d",&key);
        root=deleteNode(root, key);
        break;
    case 3:
        printf("AVL tree :\n");
        print2D(root);
        break;
    case 4:
        exit(1);
    default:
        printf("choice wrong\n");
        break;
    }
}
return 0; }
```

OUTPUT:



```
AVL Tree
1.Insert
2.Delete
3.display
4.exit
enter choice : 1
enter value: 1

1.Insert
2.Delete
3.display
4.exit
enter choice : 1
enter value: 2

1.Insert
2.Delete
3.display
4.exit
enter choice : 1
enter value: 3

1.Insert
2.Delete
3.display
4.exit
enter choice : 1
enter value: 4

1.Insert
2.Delete
3.display
4.exit
enter choice : 1
enter value: 5

1.Insert
2.Delete
3.display
4.exit
enter choice : 1
enter value: 6

1.Insert
2.Delete
3.display
4.exit
enter choice : 3
AVL tree :
                    6
          5
4
                    3
          2
                    1
```



```
1.Insert
2.Delete
3.display
4.exit
enter choice : 2
enter value: 2

1.Insert
2.Delete
3.display
4.exit
enter choice : 3
AVL tree :

                    6

          5

4

          3

                    1

1.Insert
2.Delete
3.display
4.exit
enter choice :
```

PROGRAM 5: Write a program to perform insertion and deletion operations on 2-3 trees.

```
#include  <stdio.h>
#include <stdlib.h>
#define M 3
struct node {
   int n;
   int       keys[M-1];
   struct node *p[M];
}*root=NULL;
enum KeyStatus { Duplicate,SearchFailure,Success,InsertIt,LessKeys };
void insert(int key);
void display(struct node *root,int);
void DelNode(int x);
enum KeyStatus ins(struct node *r, int x, int* y, struct node** u);
int searchPos(int x,int *key_arr, int n);
enum KeyStatus del(struct node *r, int x);
void eatline(void);
int main()
{
   int  key;
   int choice;
   while(1)
   {
      printf("1.Insert\n");
      printf("2.Delete\n");
      printf("3.display\n");
      printf("4.exit\n");
      printf("enter choice : ");
      scanf("%d",&choice); eatline();
```

```
        switch(choice)
        {
        case 1:
            printf("enter value: ");
            scanf("%d",&key); eatline();
            insert(key);
            break;
        case 2:
            printf("enter value: ");
            scanf("%d",&key); eatline();
            DelNode(key);
            break;
        case 3:
            printf("23 tree :\n");
            display(root,0);
            break;
        case 4:
            exit(1);
        default:
            printf("choice wrong\n");
            break;
        }
    }
    return 0;
}
void insert(int key)
{
    struct node *newnode;
    int upKey;
    enum KeyStatus value;
```

```
    value = ins(root, key, &upKey, &newnode);
    if (value == Duplicate)
        printf("number is already there\n");
    if (value == InsertIt)
    {
        struct node *uproot = root;
        root=malloc(sizeof(struct node));
        root->n = 1;
        root->keys[0] = upKey;
        root->p[0] = uproot;
        root->p[1] = newnode;
    }
}
enum KeyStatus ins(struct node *ptr, int key, int *upKey,struct node **newnode)
{
    struct node *newPtr, *lastPtr;
    int pos, i, n,splitPos;
    int newKey, lastKey;
    enum 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;
}
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)=malloc(sizeof(struct 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;
}
void display(struct 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);
```

```
    }
}
int searchPos(int key, int *key_arr, int n)
{
    int pos=0;
    while (pos < n && key > key_arr[pos])
        pos++;
    return pos;
}
void DelNode(int key)
{
    struct node *uproot;
    enum KeyStatus value;
    value = del(root,key);
    switch (value)
    {
    case SearchFailure:
        printf("number %d not found\n",key);
        break;
    case LessKeys:
        uproot = root;
        root = root->p[0];
        free(uproot);
        break;
    }
}
enum KeyStatus del(struct node *ptr, int key)
{
    int pos, i, pivot, n ,min;
    int *key_arr;
```

```
enum KeyStatus value;
struct node **p,*lptr,*rptr;
if (ptr == NULL)
    return SearchFailure;
n=ptr->n;
key_arr = ptr->keys;
p = ptr->p;
min = (M - 1)/2;
pos = searchPos(key, key_arr, n);
if (p[0] == NULL)
{
    if (pos == n || key < key_arr[pos])
        return SearchFailure;
    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;
}
if (pos < n && key == key_arr[pos])
{
    struct node *qp = p[pos], *qp1;
    int nkey;
    while(1)
    {
        nkey = qp->n;
        qp1 = qp->p[nkey];
        if (qp1 == NULL)
            break;
```

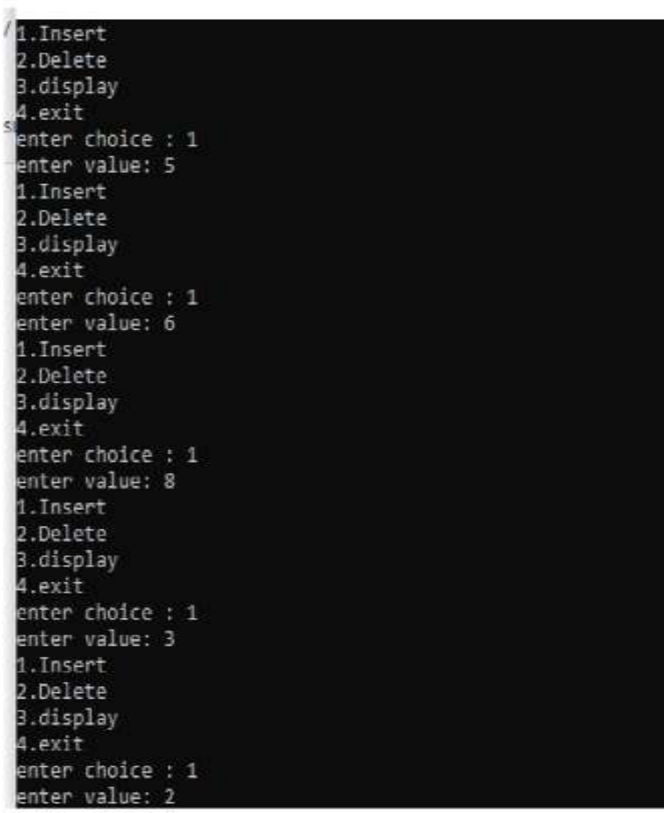
```
        qp = qp1;
    }
    key_arr[pos] = qp->keys[nkey-1];
    qp->keys[nkey - 1] = key;
}
value = del(p[pos], key);
if (value != LessKeys)
    return value;
if (pos > 0 && p[pos-1]->n > min)
{
    pivot = pos - 1;
    lptr = p[pivot];
    rptr = p[pos];

    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;
}
if (pos < n && p[pos + 1]->n > min)
{
    pivot = pos;
    lptr = p[pivot];
```

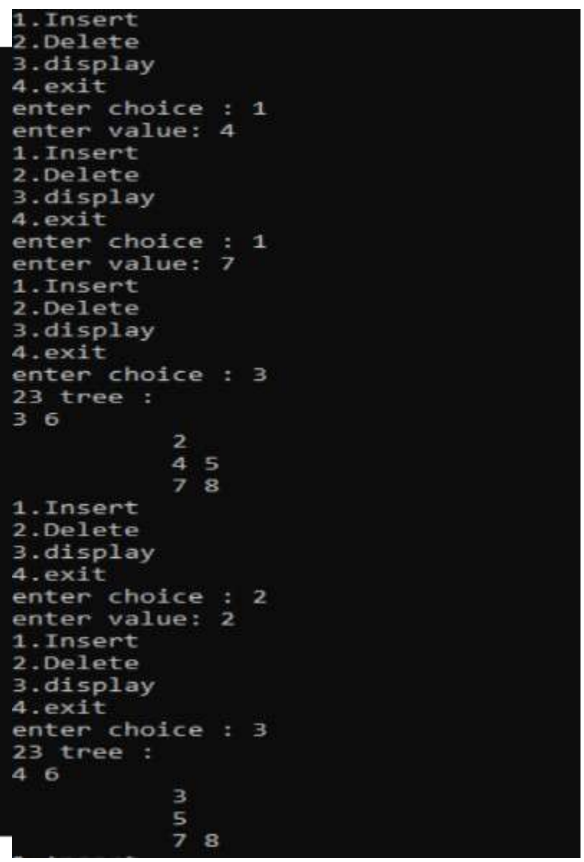
```
    rptr = p[pivot+1];
    lptr->keys[lptr->n] = key_arr[pivot];
    lptr->p[lptr->n + 1] = rptr->p[0];
    key_arr[pivot] = rptr->keys[0];
    lptr->n++;
    rptr->n--;
    for (i=0; i < rptr->n; i++)
    {
        rptr->keys[i] = rptr->keys[i+1];
        rptr->p[i] = rptr->p[i+1];
    }
    rptr->p[rptr->n] = rptr->p[rptr->n + 1];
    return Success;
}
if(pos == n)
    pivot = pos-1;
else
    pivot = pos;
lptr = p[pivot];
rptr = p[pivot+1];
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);
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;
}
void eatline(void) {
  char c;
  printf("");
  while (c=getchar()!='\n') ;
}
```

OUTPUT:

2-3 Tree





PROGRAM 6: Write a program to implement insertion operation on a red black tree. During insertion, appropriately show how recoloring or rotation operation is used.

```
#include <stdio.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 inorderTraversal(struct rbNode *node) {
  if (node) {
    inorderTraversal(node->link[0]);
    printf("%d-->%d ", node->data,node->color);
    inorderTraversal(node->link[1]);
  }
  return;
}
int main() {
  int ch, data;
  while (1) {
    printf("1. Insertion\t");
    printf("2. Traverse\t3. 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:
        inorderTraversal(root);
        printf("\n");
        break;
      case 3:
        exit(0);
      default:
        printf("Not available\n");
        break;
    }
    printf("\n");
  } return 0;}
```

OUTPUT:

```
1. Insertion    2. Traverse     3. Exit
Enter your choice:1
Enter the element to insert:12

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

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

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

1. Insertion    2. Traverse     3. Exit
Enter your choice:1
Enter the element to insert:67

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

1. Insertion    2. Traverse     3. Exit
Enter your choice:2
1-->0 3-->1 4-->1 6-->0 12-->1 67-->0
```

PROGRAM 7: Write a program to implement insertion operation on a B-tree.

```
#include<stdio.h>
#include<stdlib.h>
struct BTnode
{
int keyVal;
struct BTnode *leftNode;
struct BTnode *rightNode;
};
struct BTnode *getNode(int value)
{
struct BTnode *newNode = malloc(sizeof(struct BTnode));
newNode->keyVal = value;
newNode->leftNode = NULL;
newNode->rightNode = NULL;
return newNode;
}
struct BTnode *insert(struct BTnode *rootNode, int value)
{
if(rootNode == NULL)
return getNode(value);
if(rootNode->keyVal < value)
rootNode->rightNode = insert(rootNode->rightNode,value);
else if(rootNode->keyVal > value)
rootNode->leftNode = insert(rootNode->leftNode,value);
return rootNode;
}
void insertorder(struct BTnode *rootNode)
{
```

```
if(rootNode == NULL)
return;
insertorder(rootNode->leftNode);
printf("%d ",rootNode->keyVal);
insertorder(rootNode->rightNode);
}
int main()
{ struct BTnode *rootNode = NULL;
int n,ch;
while(1){
printf("\nEnter your choice 1.Insert 2.Exit\n");
scanf("%d",&ch);
switch(ch){
case 1:printf("\nEnter the element to insert:");
   scanf("%d",&n);
   rootNode = insert(rootNode,n);
   insertorder(rootNode);
   break;
case 2: exit(0); } } }
```

OUTPUT:

```
Enter your choice 1.Insert 2.Exit
1

Enter the element to insert:10
10
Enter your choice 1.Insert 2.Exit
1

Enter the element to insert:12
10 12
Enter your choice 1.Insert 2.Exit
1

Enter the element to insert:4
4 10 12
Enter your choice 1.Insert 2.Exit
1

Enter the element to insert:5
4 5 10 12
Enter your choice 1.Insert 2.Exit
1

Enter the element to insert:6
4 5 6 10 12
Enter your choice 1.Insert 2.Exit
1

Enter the element to insert:7
4 5 6 7 10 12
Enter your choice 1.Insert 2.Exit
1

Enter the element to insert:1
1 4 5 6 7 10 12
Enter your choice 1.Insert 2.Exit
```

PROGRAM 8: Write a program to implement functions of Dictionary using Hashing.

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <cstdio>
using namespace std;
const int T_S = 200;
class HashTableEntry
{
public:
   int k;
   int v;
   HashTableEntry(int k, int v)
   {
      this->k = k;
      this->v = v;    }
};
class HashMapTable
{
private:
   HashTableEntry **t;
public:
   HashMapTable()
   {
      t = new HashTableEntry *[T_S];
      for (int i = 0; i < T_S; i++)
      {
         t[i] = NULL;
      }    }
```

```
int HashFunc(int k)
{
   return k % T_S;
}
void Insert(int k, int v)
{
   int h = HashFunc(k);
   while (t[h] != NULL && t[h]->k != k)
   {
      h = HashFunc(h + 1);
   }
   if (t[h] != NULL)
      delete t[h];
   t[h] = new HashTableEntry(k, v);
}
int SearchKey(int k)
{
   int h = HashFunc(k);
   while (t[h] != NULL && t[h]->k != k)
   {
      h = HashFunc(h + 1);
   }
   if (t[h] == NULL)
      return -1;
   else
      return t[h]->v;
}
void Remove(int k)
{
   int h = HashFunc(k);
```

```
        while (t[h] != NULL)
        {
            if (t[h]->k == k)
                break;
            h = HashFunc(h + 1);
        }
        if (t[h] == NULL)
        {
            cout << "No Element found at key " << k << endl;
            return;
        }
        else
        {
            delete t[h];
        }
        cout << "Element Deleted" << endl;
    }
    ~HashMapTable()
    {
        for (int i = 0; i < T_S; i++)
        {
            if (t[i] != NULL)
                delete t[i];
            delete[] t;
        }  } };
int main()
{
    HashMapTable hash;
    int k, v;
    int c;
```

```
while (1)
{
    cout << "1.Insert element into the table";
    cout << "\t2.Search element from the key";
    cout << "\t3.Delete element at a key";
    cout << "4.Exit" << endl;
    cout << "Enter your choice: ";
    cin >> c;
    switch (c)
    {
    case 1:
        cout << "Enter element to be inserted: ";
        cin >> v;
        cout << "Enter key at which element to be inserted: ";
        cin >> k;
        hash.Insert(k, v);
        break;
    case 2:
        cout << "Enter key of the element to be searched: ";
        cin >> k;
        if (hash.SearchKey(k) == -1)
        {
            cout << "No element found at key " << k << endl;
            continue;
        }
        else
        {
            cout << "Element at key " << k << " : ";
            cout << hash.SearchKey(k) << endl;
        }
```

```
            break;
        case 3:
            cout << "Enter key of the element to be deleted: ";
            cin >> k;
            hash.Remove(k);
            break;
        case 4:
            exit(1);
        default:
            cout << "\nEnter correct option\n";
        }
    }  return 0; }
```

OUTPUT:

```
1.Insert element into the table 2.Search element from the key   3.Delete element at a key4.Exit
Enter your choice: 1
Enter element to be inserted: 12
Enter key at which element to be inserted: 12
1.Insert element into the table 2.Search element from the key   3.Delete element at a key4.Exit
Enter your choice: 1
Enter element to be inserted: 12
Enter key at which element to be inserted: 11
1.Insert element into the table 2.Search element from the key   3.Delete element at a key4.Exit
Enter your choice: 2
Enter key of the element to be searched: 11
Element at key 11 : 12
1.Insert element into the table 2.Search element from the key   3.Delete element at a key4.Exit
Enter your choice: 3
Enter key of the element to be deleted: 11
Element Deleted
1.Insert element into the table 2.Search element from the key   3.Delete element at a key4.Exit
Enter your choice: 2
Enter key of the element to be searched: 11
No element found at key 11
1.Insert element into the table 2.Search element from the key   3.Delete element at a key4.Exit
Enter your choice: 4
```

PROGRAM 9: Write a program to implement Binomial Heap insert(), extractmin(), getmin()

```
#include<bits/stdc++.h>
using namespace std;
struct Node
{
	int data, degree;
	Node *child, *sibling, *parent;
};
Node* newNode(int key)
{
	Node *temp = new Node;
	temp->data = key;
	temp->degree = 0;
	temp->child = temp->parent = temp->sibling = NULL;
	return temp;
}
// merge two Binomial Trees.
Node* mergeBinomialTrees(Node *b1, Node *b2)
{
	if (b1->data > b2->data)
		swap(b1, b2);
	b2->parent = b1;
	b2->sibling = b1->child;
	b1->child = b2;
	b1->degree++;
	return b1;
}
list<Node*> unionBionomialHeap(list<Node*> l1,
                                        list<Node*> l2)
```

```
{
	list<Node*> _new;
	list<Node*>::iterator it = l1.begin();
	list<Node*>::iterator ot = l2.begin();
	while (it!=l1.end() && ot!=l2.end())
	{
		if((*it)->degree <= (*ot)->degree)
		{
			_new.push_back(*it);
			it++;
		}
		else
		{
			_new.push_back(*ot);
			ot++;
		}
	}
	while (it != l1.end())
	{
		_new.push_back(*it);
		it++;
	}
	while (ot!=l2.end())
	{
		_new.push_back(*ot);
		ot++;
	}
	return _new;
}
```

```
list<Node*> adjust(list<Node*> _heap)
{
	if (_heap.size() <= 1)
		return _heap;
	list<Node*> new_heap;
	list<Node*>::iterator it1,it2,it3;
	it1 = it2 = it3 = _heap.begin();

	if (_heap.size() == 2)
	{
		it2 = it1;
		it2++;
		it3 = _heap.end();
	}
	else
	{
		it2++;
		it3=it2;
		it3++;
	}
	while (it1 != _heap.end())
	{
		if (it2 == _heap.end())
			it1++;
		else if ((*it1)->degree < (*it2)->degree)
		{
			it1++;
			it2++;
			if(it3!=_heap.end())
				it3++;
```

```
        }

        else if (it3!=_heap.end() &&
                (*it1)->degree == (*it2)->degree &&
                (*it1)->degree == (*it3)->degree)
        {
            it1++;
            it2++;
            it3++;
        }

        else if ((*it1)->degree == (*it2)->degree)
        {
            Node *temp;
            *it1 = mergeBinomialTrees(*it1,*it2);
            it2 = _heap.erase(it2);
            if(it3 != _heap.end())
                it3++;
        }
    }
    return _heap;
}
list<Node*> insertATreeInHeap(list<Node*> _heap,
                                        Node *tree)
{
    list<Node*> temp;
    temp.push_back(tree);
    temp = unionBionomialHeap(_heap,temp);
    return adjust(temp);
}
```

```
list<Node*> removeMinFromTreeReturnBHeap(Node *tree)
{
	list<Node*> heap;
	Node *temp = tree->child;
	Node *lo;
	while (temp)
	{
		lo = temp;
		temp = temp->sibling;
		lo->sibling = NULL;
		heap.push_front(lo);
	}
	return heap;
}
list<Node*> insert(list<Node*> _head, int key)
{
	Node *temp = newNode(key);
	return insertATreeInHeap(_head,temp);
}
Node* getMin(list<Node*> _heap)
{
	list<Node*>::iterator it = _heap.begin();
	Node *temp = *it;
	while (it != _heap.end())
	{
		if ((*it)->data < temp->data)
			temp = *it;
		it++;
	}
	return temp;
```

```
}
list<Node*> extractMin(list<Node*> _heap)
{
	list<Node*> new_heap,lo;
	Node *temp;
	temp = getMin(_heap);
	list<Node*>::iterator it;
	it =  _heap.begin();
	while (it != _heap.end())
	{
		if (*it != temp)
		{
			new_heap.push_back(*it);
		}
		it++;
	}
	lo = removeMinFromTreeReturnBHeap(temp);
	new_heap = unionBionomialHeap(new_heap,lo);
	new_heap = adjust(new_heap);
	return new_heap;
}
void printTree(Node *h)
{
	while (h)
	{
		cout << h->data << " ";
		printTree(h->child);
		h = h->sibling;
	}
}
```

```
void printHeap(list<Node*> _heap)
{
	list<Node*> ::iterator it;
	it =  _heap.begin();
	while (it != _heap.end())
	{
		printTree(*it);
		it++;
	}
}
int main()
{
	int ch,key,i,n,a[10];
	list<Node*> _heap;
  printf("Enter the number of elements you want inside the Binomial Heap: \t");
  scanf("%d",&n);
  printf("Enter %d elements :\n",n);
  for(i=1;i<=n;i++)
  {
    printf("%d :\t",i);
    scanf("%d",&a[i]);
    _heap = insert (_heap,a[i]);
  }
	cout << "Heap elements after insertion:\n";
	printHeap(_heap);
	Node *temp = getMin(_heap);
	cout << "\nMinimum element of heap "
		<< temp->data << "\n";
	_heap = extractMin(_heap);
	cout << "Heap after deletion of minimum element\n";
```

printHeap(_heap);

return 0;

}

OUTPUT:

```
Enter the number of elements you want inside the Binomial Heap:     4
Enter 4 elements :
1 : 6
2 : 78
3 : 45
4 : 12
Heap elements after insertion:
6 12 45 78
Minimum element of heap 6
Heap after deletion of minimum element
78 12 45
```

PROGRAM 10: Write a program to implement Binomial heap delete(), decreasekey()

```
#include <bits/stdc++.h>
using namespace std;
struct Node
{
    int val, degree;
    Node *parent, *child, *sibling;
};
Node *root = NULL;
int binomialLink(Node *h1, Node *h2)
{
    h1->parent = h2;
    h1->sibling = h2->child;
    h2->child = h1;
    h2->degree = h2->degree + 1;
}
Node *createNode(int n)
{
    Node *new_node = new Node;
    new_node->val = n;
    new_node->parent = NULL;
    new_node->sibling = NULL;
    new_node->child = NULL;
    new_node->degree = 0;
    return new_node;
}
Node *mergeBHeaps(Node *h1, Node *h2)
{
    if (h1 == NULL)
```

```
        return h2;
    if (h2 == NULL)
        return h1;
    Node *res = NULL;
    if (h1->degree <= h2->degree)
        res = h1;
    else if (h1->degree > h2->degree)
        res = h2;
    while (h1 != NULL && h2 != NULL)
    {
        if (h1->degree < h2->degree)
            h1 = h1->sibling;
        else if (h1->degree == h2->degree)
        {
            Node *sib = h1->sibling;
            h1->sibling = h2;
            h1 = sib;
        }
        else
        {
            Node *sib = h2->sibling;
            h2->sibling = h1;
            h2 = sib;
        }
    }
    return res;
}
Node *unionBHeaps(Node *h1, Node *h2)
{
    if (h1 == NULL && h2 == NULL)
```

```
   return NULL;
Node *res = mergeBHeaps(h1, h2);
Node *prev = NULL, *curr = res,
     *next = curr->sibling;
while (next != NULL)
{
    if ((curr->degree != next->degree) ||
            ((next->sibling != NULL) &&
            (next->sibling)->degree ==
            curr->degree))
    {
        prev = curr;
        curr = next;
    }
    else
    {
        if (curr->val <= next->val)
        {
            curr->sibling = next->sibling;
            binomialLink(next, curr);
        }
        else
        {
            if (prev == NULL)
                res = next;
            else
                prev->sibling = next;
            binomialLink(curr, next);
            curr = next;
        }
```

```
        }
        next = curr->sibling;
    }
    return res;
}
void binomialHeapInsert(int x)
{
    root = unionBHeaps(root, createNode(x));
}
void display(Node *h)
{
    while (h)
    {
        cout << h->val << " ";
        display(h->child);
        h = h->sibling;
    }
}
int revertList(Node *h)
{
    if (h->sibling != NULL)
    {
        revertList(h->sibling);
        (h->sibling)->sibling = h;
    }
    else
        root = h;
}
Node *extractMinBHeap(Node *h)
{
```

```
    if (h == NULL)
        return NULL;
    Node *min_node_prev = NULL;
    Node *min_node = h;
    int min = h->val;
    Node *curr = h;
    while (curr->sibling != NULL)
    {
        if ((curr->sibling)->val < min)
        {
            min = (curr->sibling)->val;
            min_node_prev = curr;
            min_node = curr->sibling;
        }
        curr = curr->sibling;
    }
    if (min_node_prev == NULL &&
        min_node->sibling == NULL)
        h = NULL;
    else if (min_node_prev == NULL)
        h = min_node->sibling;
    else
        min_node_prev->sibling = min_node->sibling;
    if (min_node->child != NULL)
    {
        revertList(min_node->child);
        (min_node->child)->sibling = NULL;
    }
    return unionBHeaps(h, root);
}
```

```
Node *findNode(Node *h, int val)
{
    if (h == NULL)
      return NULL;
    if (h->val == val)
        return h;
    Node *res = findNode(h->child, val);
    if (res != NULL)
       return res;
    return findNode(h->sibling, val);
}
void decreaseKeyBHeap(Node *H, int old_val,
                               int new_val)
{
    Node *node = findNode(H, old_val);

    if (node == NULL)
        return;
    node->val = new_val;
    Node *parent = node->parent;
    while (parent != NULL && node->val < parent->val)
    {
        swap(node->val, parent->val);
        node = parent;
        parent = parent->parent;
    }
}
Node *binomialHeapDelete(Node *h, int val)
{
    if (h == NULL)
```

```
        return  NULL;
    decreaseKeyBHeap(h, val, INT_MIN);
    return extractMinBHeap(h);
}
int main()
{
   int n,temp,del;
   cout<<"Enter no of elements in the heap"<<endl;
   cin>>n;
   cout<<"Enter elements"<<endl;
   for(int i=0;i<n;i++)
   {
    cin>>temp;
    binomialHeapInsert(temp);
    }
    cout << "The heap is:\n";
    display(root);
    cout<<endl;
    cout<<"Enter element to be deleted"<<endl;
    cin>>del;
    root = binomialHeapDelete(root, del);
    cout << "After deletion, the heap is:\n";
    display(root);
    cout<<endl;
    return 0;
}
```

OUTPUT:

```
Enter no of elements in the heap
5
Enter elements
23 45 12 3 67
The heap is:
67 3 23 45 12
Enter element to be deleted
23
After deletion, the heap is:
3 12 67 45
```