

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT

on

ADVANCED DATA STRUCTURES

Submitted by

SRI DEVI K P NAIK (1BM20CS162)

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



CERTIFICATE

This is to certify that the Lab work entitled “**ADVANCED DATA STRUCTURES**” carried out by **SRI DEVI K P NAIK(1BM20CS162)**, 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.GR Prasad
Associate Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Sl. No.	Experiment Title	Page No.
1	Write a program to implement the following list: An ordinary Doubly Linked List requires space for two address fields to store the addresses of previous and next nodes. A memory efficient version of Doubly Linked List can be created using only one space for address field with every node. This 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. In the XOR linked list, instead of storing actual memory addresses, every node stores the XOR of addresses of previous and next nodes.	05-08
2	Write a program to perform insertion, deletion and searching operations on a skip list.	09-17
3	Given a boolean 2D matrix, find the number of islands. A group of connected 1s forms an island. For example, the below matrix contains 5 islands {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} A cell in the 2D matrix can be connected to 8 neighbours. Use disjoint sets to implement the above scenario.	18-24
4	Write a program to perform insertion and deletion operations on AVL trees.	25-36
5	Write a program to perform insertion and deletion operations on 2-3 trees.	37-50
6	Write a program to implement insertion operation on a red black tree. During insertion, appropriately show how recolouring or rotation operation is used.	51-60
7	Write a program to implement insertion operation on a B-tree.	61-70
8	Write a program to implement functions of Dictionary using Hashing.	71-76
9	Write a program to implement the following functions on a Binomial heap: 1. insert(H, k): Inserts a key 'k' to Binomial Heap 'H'. This operation first creates a Binomial Heap with single key 'k', then calls union on H and the new Binomial heap. 2. getMin(H): A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key. 3. extractMin(H): This operation also uses union(). We first call	77-91

	getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally we call union() on H and the newly created Binomial Heap.	
10	Write a program to implement the following functions on a Binomial heap: 1. delete(H): Like Binary Heap, delete operation first reduces the key to minus infinite, then calls extractMin(). 2. decreaseKey(H): decreaseKey() is also similar to Binary Heap. We compare the decreases key with it parent and if parent's key is more, we swap keys and recur for parent. We stop when we either reach a node whose parent has smaller key or we hit the root node.	92-106

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 structures.
CO3	Ability to apply the knowledge of hashing techniques.
CO4	Ability to conduct practical experiments to solve problems using an appropriate data structure.

LAB PROGRAM 1:

Write a program to implement the following list: An ordinary Doubly Linked List requires space for two address fields to store the addresses of previous and next nodes. A memory efficient version of Doubly Linked List can be created using only one space for address field with every node. This 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. In the XOR linked list, instead of storing actual memory addresses, every node stores the XOR of addresses of previous and next nodes.

```
#include <stdio.h>

#include <stdlib.h>

struct Node {
    int data;
    struct Node* npx; // XOR of next and previous node
};

// Helper function to get XOR of two nodes
struct Node* XOR(struct Node* a, struct Node* b) {
    return (struct Node*) ((unsigned int) (a) ^ ( unsigned int) (b));
}

// Function to add a node to the XOR linked list
void insert(struct Node** head, int data) {
    // Allocate memory for the new node
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->npx = XOR(*head, NULL);
```

```
// If the list is not empty, update the npx of the current head
```

```
if (*head != NULL) {
```

```
    struct Node* next = XOR((*head)->npx, NULL);
```

```
    (*head)->npx = XOR(new_node, next);
```

```
}
```

```
*head = new_node;
```

```
}
```

```
// Function to find a pair whose sum is equal to 'k'
```

```
void findPair(struct Node* head, int k) {
```

```
    struct Node* left = head;
```

```
    struct Node* right = head;
```

```
    struct Node* prev = NULL;
```

```
    int found = 0;
```

```
    while (left != NULL && right != NULL) {
```

```
        if (left->data + right->data == k) {
```

```
            printf("Pair found: (%d, %d)\n", left->data, right->data);
```

```
            found = 1;
```

```
            break;
```

```
        }
```

```
        else if (left->data + right->data < k) {
```

```
            prev = left;
```

```
            left = XOR(prev, right->npx);
```

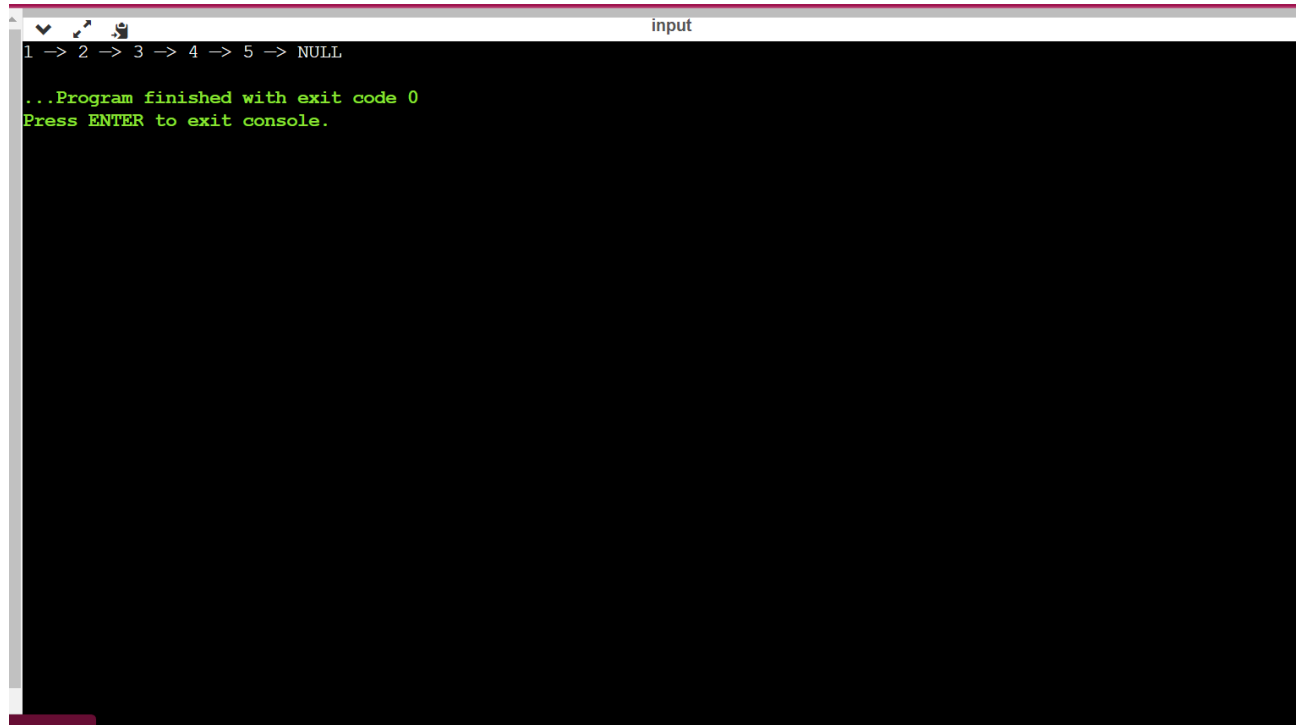
```
        }
```

```
    } else {
```

```
        prev = right;
```

```
right = XOR(prev, left->npx);  
}  
}  
if (!found) printf("No pair found whose sum is equal to %d\n", k);  
}  
int main() {  
    int k = 5;  
    struct Node* head = NULL;  
    insert(&head, 1);  
    insert(&head, 3);  
    insert(&head, 4);  
    insert(&head, 6);  
    insert(&head, 8);  
    findPair(head, k);  
    return 0;  
}
```

OUTPUT:



```
1 -> 2 -> 3 -> 4 -> 5 -> NULL
...Program finished with exit code 0
Press ENTER to exit console.
```

The image shows a terminal window with a title bar that includes the text 'input'. The terminal has a black background with green text. The first line of output is '1 -> 2 -> 3 -> 4 -> 5 -> NULL'. The second line is '...Program finished with exit code 0'. The third line is 'Press ENTER to exit console.'. The terminal window has standard OS window controls (minimize, maximize, close) in the top-left corner of the title bar.

LAB 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--;
```

```
        return 0;
    }
    return 1;
}
```

```
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() {
    int choice,arr[7],i,n,del;
    while(1){

        printf("1.insertion\n 2.searching\n 3.deleting\n 4.wrong choice\n");
        printf("Enter the choice\n");
        scanf("%d",&choice);

        switch(choice)
        {
```

case 1:

```
printf("ENter the number of elements in the skip list\n");
```

```
scanf("%d",&n);
```

```
printf("Enter the nodes of the array\n");
```

```
for(i=0;i<n;i++)
```

```
    scanf("%d",&arr[i]);
```

```
    skiplist list;
```

```
    skiplist_init(&list);
```

```
    printf("Insert:-----\n");
```

```
    for (i = 0; i < sizeof(arr) / sizeof(arr[0]); i++) {
```

```
        skiplist_insert(&list, arr[i], arr[i]);
```

```
    }
```

```
    skiplist_dump(&list);
```

```
break;
```

```
case 2:printf("Search:-----\n");
```

```
int keys[] = { 3, 4, 7, 10, 111 };
```

```
for (i = 0; i < sizeof(keys) / sizeof(keys[0]); i++)
```

```
{
```

```
    snode *x = skiplist_search(&list, keys[i]);
```

```
    if (x)
```

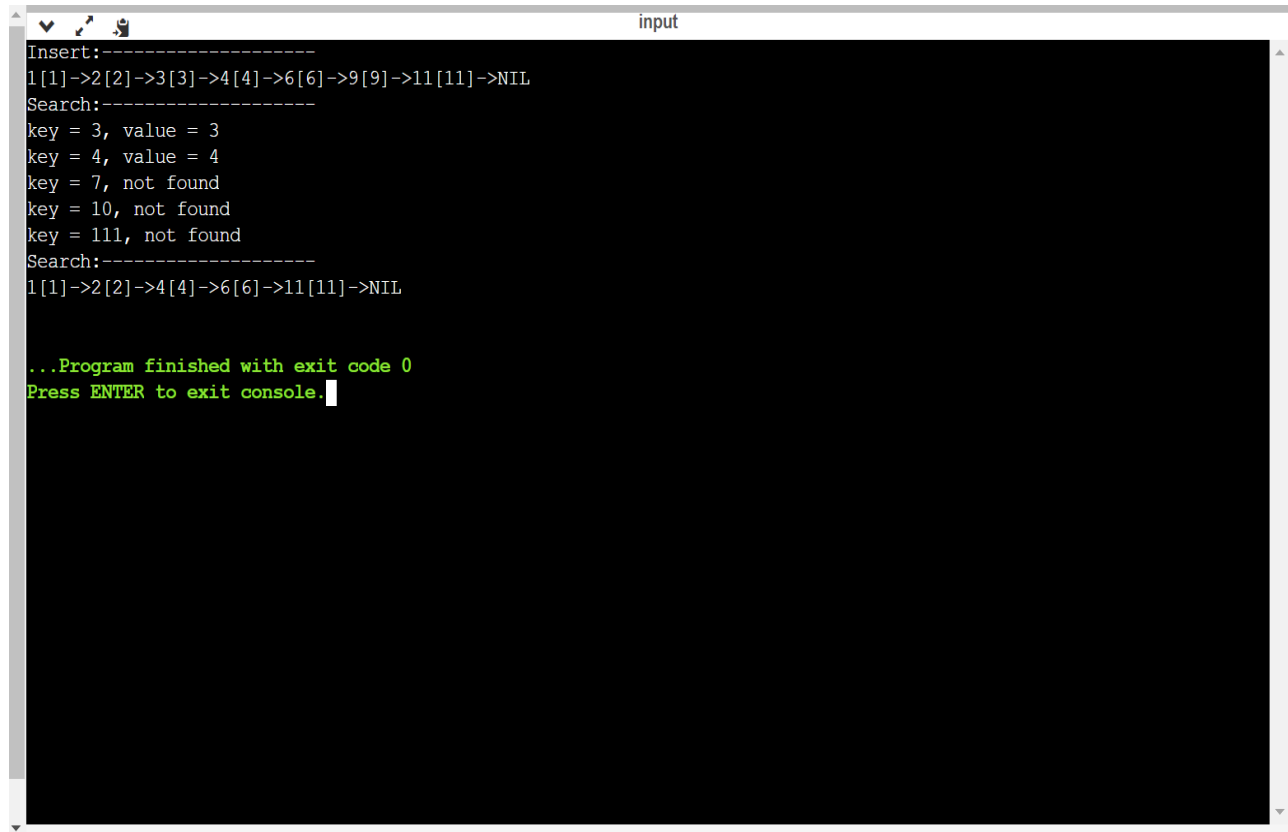
```
{
```

```
        printf("key = %d, value = %d\n", keys[i], x->value);
    } else
    {
        printf("key = %d, not fuound\n", keys[i]);
    }
}
break;
case 3:

    printf("Deletion:-----\n");
    printf("Enter the element to be deleted\n");
    scanf("%d",&del);
    skiplist_delete(&list, del);
    //skiplist_delete(&list, 9);
    skiplist_dump(&list);
    break;
default:printf("wrong choice \n");
    exit(0);
}
}

return 0;
}
```


OUTPUT:



```
input
Insert:-----
1[1]->2[2]->3[3]->4[4]->6[6]->9[9]->11[11]->NIL
Search:-----
key = 3, value = 3
key = 4, value = 4
key = 7, not found
key = 10, not found
key = 111, not found
Search:-----
1[1]->2[2]->4[4]->6[6]->11[11]->NIL

...Program finished with exit code 0
Press ENTER to exit console.
```

LAB PROGRAM 3:

Given a boolean 2D matrix, find the number of islands. A group of connected 1s forms an island. For example, the below matrix contains 5 islands

{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}

A cell in the 2D matrix can be connected to 8 neighbours. Use disjoint sets to implement the above scenario.

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

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

```
void makeset(int n,int parent[])
```

```
{
```

```
    int i;
```

```
    for(i = 0; i < n; i++)
```

```
    {
```

```
        parent[i] = i;
```

```
    }
```

```
}
```

```
int find(int x,int parent[])
```

```
{
```

```
    if(parent[x] != x)
```

```
{
    parent[x]=find(parent[x],parent);
}
return parent[x];
}
void Union(int x, int y, int parent[], int rank[])
{
    int x0 = find(x,parent);
    int y0 = find(y,parent);
    if(x0 == y0)
    {
        return;
    }
    if(rank[x0] < rank[y0])
    {
        parent[x0] = y0;
    }
    else if(rank[y0] < rank[x0])
    {
        parent[y0] = x0;
    }
    else
    {
        parent[y0] = x0;
    }
}
```

```

        rank[x0] = rank[x0] + 1;
    }
}
int count_of_Islands(int arr[20][20],int a,int b,int n,int parent[], int rank[])
{
    int j,k;

    for(j = 0; j < a; j++)
    {
        for(k = 0; k < b; k++)
        {
            if(arr[j][k] == 0)
            {
                continue;
            }
            if(j + 1 < a && arr[j + 1][k] == 1)
            {
                Union(j * (b) + k, (j + 1) * (b) + k, parent, rank);
            }
            if(j - 1 >= 0 && arr[j - 1][k] == 1)
            {
                Union(j * (b) + k, (j - 1) * (b) + k, parent, rank);
            }
            if(k + 1 < b && arr[j][k + 1] == 1)
            {

```

```

    Union(j * (b) + k, (j) * (b) + k + 1, parent, rank);
}
if(k - 1 >= 0 && arr[j][k - 1] == 1)
{
    Union(j * (b) + k, (j) * (b) + k - 1, parent, rank);
}
if(j + 1 < a && k + 1 < b && arr[j + 1][k + 1] == 1)
{
    Union(j * (b) + k, (j + 1) * (b) + k + 1, parent, rank);
}
if(j + 1 < a && k - 1 >= 0 && arr[j + 1][k - 1] == 1)
{
    Union(j * b + k, (j + 1) * (b) + k - 1, parent, rank);
}
if(j - 1 >= 0 && k + 1 < b && arr[j - 1][k + 1] == 1)
{
    Union(j * b + k, (j - 1) * b + k + 1, parent, rank);
}
if(j - 1 >= 0 && k - 1 >= 0 && arr[j - 1][k - 1] == 1)
{
    Union(j * b + k, (j - 1) * b + k - 1, parent, rank);
}
}
}

```

```
int freq[n],i;
for(i=0;i<n;i++)
{
    freq[i] = 0;
}
int num = 0;
for(j = 0; j < a; j++)
{
    for(k = 0; k < b; k++)
    {
        if(arr[j][k] == 1)
        {
            int x = find((j * b) + k, parent);

            if(freq[x] == 0)
            {
                num = num + 1;
                freq[x] = freq[x] + 1;
            }
            else
            {
                freq[x] = freq[x] + 1;
            }
        }
    }
}
```

```

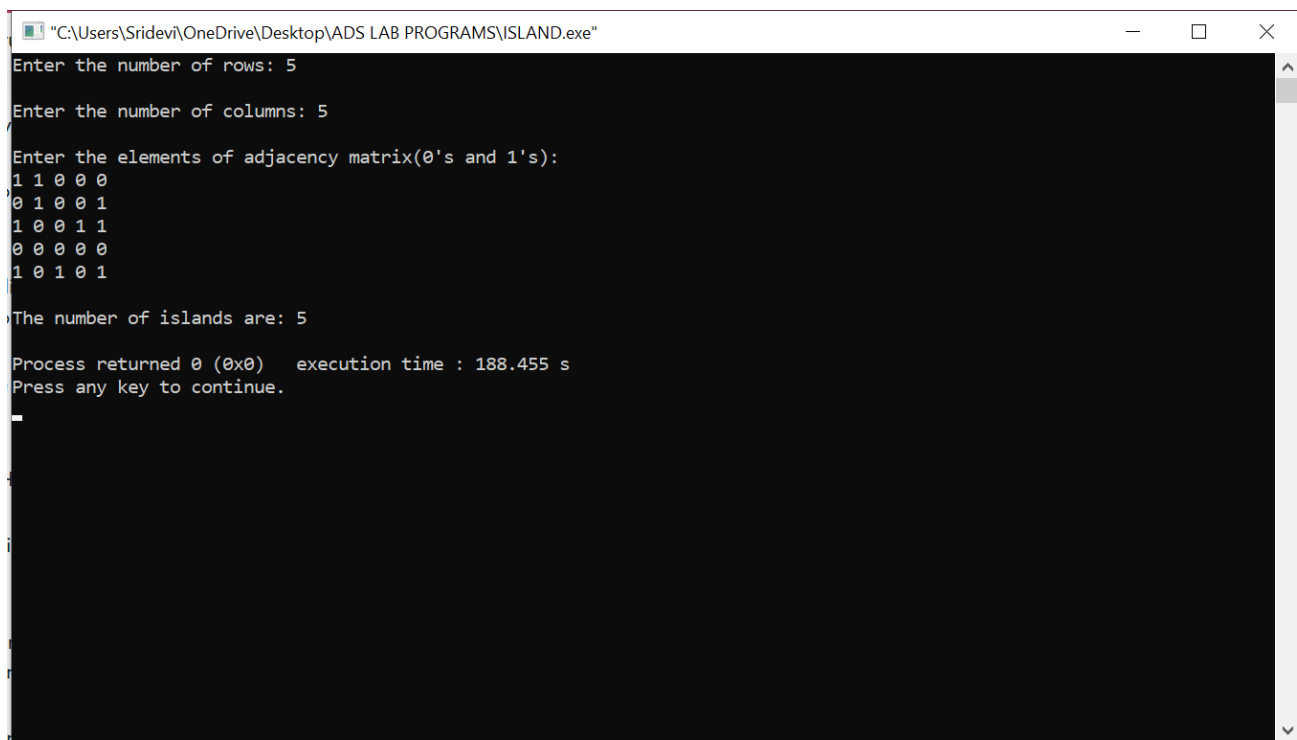
    }
}
return num;
}
int main()
{
    int r,c;
    int i,j,n,l;
    int total;
    printf("Enter the number of rows: ");
    scanf("%d",&r);
    printf("\nEnter the number of columns: ");
    scanf("%d",&c);
    n = r*c;
    int parent[n];
    int rank[n];
    for(l = 0; l < n; l++)
    {
        rank[l] = 0;
    }
    makeset(n,parent);
    int arr[20][20];
    printf("\nEnter the elements of adjacency matrix(0's and 1's):\n");
    for(i=0;i<r;i++)
    {

```

```
    for(j=0;j<c;j++)
    {
        scanf("%d",&arr[i][j]);
    }
}

total = count_of_Islands(arr,r,c,n,parent,rank);
printf("\nThe number of islands are: %d\n",total);
return 0;
}
```

OUTPUT:



```
"C:\Users\Sridevi\OneDrive\Desktop\ADS LAB PROGRAMS\ISLAND.exe"
Enter the number of rows: 5
Enter the number of columns: 5
Enter the elements of adjacency matrix(0's and 1's):
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
The number of islands are: 5
Process returned 0 (0x0)   execution time : 188.455 s
Press any key to continue.
```


LAB PROGRAM 4:

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

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

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

```
#define count 10
```

```
// Create Node
```

```
struct Node {
```

```
    int key;
```

```
    struct Node *left;
```

```
    struct Node *right;
```

```
    int height;
```

```
};
```

```
int max(int a, int b);
```

```
// Calculate height
```

```
int height(struct Node *N) {
```

```
    if (N == NULL)
```

```
        return 0;
```

```
    return N->height;
```

```
}
```

```
int max(int a, int b) {
```

```
    return (a > b) ? a : b;  
}
```

```
// Create a node
```

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

```
// Right rotate
```

```
struct Node *rightRotate(struct Node *y) {  
    struct Node *x = y->left;  
    struct Node *T2 = x->right;  
  
    x->right = y;  
    y->left = T2;  
  
    y->height = max(height(y->left), height(y->right)) + 1;  
    x->height = max(height(x->left), height(x->right)) + 1;
```

```
    return x;
}

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

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

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

    return y;
}

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

// Insert node
```

```

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

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

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

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

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

    if (balance > 1 && key > node->left->key) {

```

```
node->left = leftRotate(node->left);  
return rightRotate(node);  
}
```

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

```
return node;  
}
```

```
struct Node *minValueNode(struct Node *node) {  
    struct Node *current = node;  
  
    while (current->left != NULL)  
        current = current->left;  
  
    return current;  
}
```

// Delete a nodes

```
struct Node *deleteNode(struct Node *root, int key) {  
    // Find the node and delete it  
    if (root == NULL)
```

```
return root;
```

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

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

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

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

```
else {
```

```
    if ((root->left == NULL) || (root->right == NULL)) {
```

```
        struct Node *temp = root->left ? root->left : root->right;
```

```
        if (temp == NULL) {
```

```
            temp = root;
```

```
            root = NULL;
```

```
        } else
```

```
            *root = *temp;
```

```
        free(temp);
```

```
    } else {
```

```
        struct Node *temp = minValueNode(root->right);
```

```
        root->key = temp->key;
```

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

```
    }
```

```
}
```

```
if (root == NULL)
```

```
    return root;
```

```
// Update the balance factor of each node and
```

```
// balance the tree
```

```
root->height = 1 + max(height(root->left),  
                        height(root->right));
```

```
int balance = getBalance(root);
```

```
if (balance > 1 && getBalance(root->left) >= 0)
```

```
    return rightRotate(root);
```

```
if (balance > 1 && getBalance(root->left) < 0) {
```

```
    root->left = leftRotate(root->left);
```

```
    return rightRotate(root);
```

```
}
```

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

```
    return leftRotate(root);
```

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

```
    root->right = rightRotate(root->right);
```

```
    return leftRotate(root);
```

```
}
```

```
return root;
```

```
}
```

```
// Print the tree
```

```
/*void printPreOrder(struct Node *root) {
```

```
if (root != NULL) {
```

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

```
    printPreOrder(root->left);
```

```
    printPreOrder(root->right);
```

```
}
```

```
*/
```

```
void print2DUtil(struct Node* root, int space)
```

```
{
```

```
    // Base case
```

```
    if (root == NULL)
```

```
        return;
```

```
    // Increase distance between levels
```

```
    space += count;
```

```
    // Process right child first
```

```
    print2DUtil(root->right, space);
```



```

    // Print current node after space
    // count
    printf("\n");
    for (int i = count; i < space; i++)
        printf(" ");
    printf("%d\n", root->key);

    // Process left child
    print2DUtil(root->left, space);
}

// Wrapper over print2DUtil()
void print2D(struct Node* root)
{
    // Pass initial space count as 0
    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:

```

"C:\Users\Sridevi\OneDrive\Desktop\ADS LAB PROGRAMS\AVL.exe"
1.Insert
2.Delete
3.display
4.exit
enter choice : 1
enter value: 10

1.Insert
2.Delete
3.display
4.exit
enter choice : 1
enter value: 15

1.Insert
2.Delete
3.display
4.exit
enter choice : 1
enter value: 20

1.Insert
2.Delete
3.display
4.exit
enter choice : 1
enter value: 9

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

```

```

Select "C:\Users\Sridevi\OneDrive\Desktop\ADS LAB PROGRAMS\AVL.exe"
1.Insert
2.Delete
3.display
4.exit
enter choice : 1
enter value: 16

1.Insert
2.Delete
3.display
4.exit
enter choice : 1
enter value: 17

1.Insert
2.Delete
3.display
4.exit
enter choice : 1
enter value: 8

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 :

    20
   /
  17
 /
16

```

```
"C:\Users\Sridevi\OneDrive\Desktop\ADS LAB PROGRAMS\AVL.exe"
1.Insert
2.Delete
3.display
4.exit
enter choice : 2
enter value: 16

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

      20
     /  \
    17   15
   /  \  /  \
  10  9 8   5
 /  \
6   5

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

```
"C:\Users\Sridevi\OneDrive\Desktop\ADS LAB PROGRAMS\AVL.exe"
1.Insert
2.Delete
3.display
4.exit
enter choice : 2
enter value: 16

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

      20
     /  \
    17   15
   /  \  /  \
  10  9 8   5
 /  \
6   5

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

LAB 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,c=1;
    for(i=1; i<=blanks; i++)
        printf(" ");
    for (i=0; i < ptr->n; i++)
        printf("%d ",ptr->keys[i]);
    printf("\n");
    printf("-----level :%d",c);
    c++;
    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];

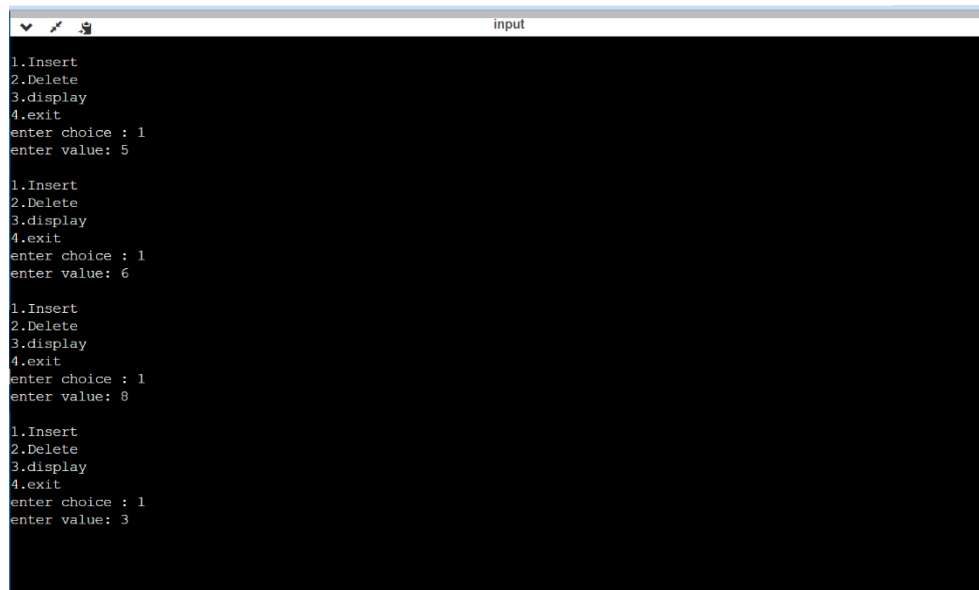
lptr->keys[lptr->n] = key_arr[pivot];
lptr->p[lptr->n + 1] = rptr->p[0];
for (i=0; i < rptr->n; i++)
{
    lptr->keys[lptr->n + 1 + i] = rptr->keys[i];
    lptr->p[lptr->n + 2 + i] = rptr->p[i+1];
}
lptr->n = lptr->n + rptr->n + 1;
free(rptr);
for (i=pos+1; i < n; i++)
{
    key_arr[i-1] = key_arr[i];
    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:



```
input  
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 : 1  
enter value: 8  
  
1.Insert  
2.Delete  
3.display  
4.exit  
enter choice : 1  
enter value: 3
```



```
input  
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: 2  
  
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: 7  
  
1.Insert  
2.Delete  
3.display  
4.exit  
enter choice :
```

```
input
2.Delete
3.display
4.exit
enter choice : 1
enter value: 7

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

      8
     /
    7
   /  \
  5    6
 /  \
3    4
   \
    2

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

```
input
1.Insert
2.Delete
3.display
4.exit
enter choice : 2
enter value: 8

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

      7
     /
    6
   /  \
  5    4
 /  \
3    2

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

LAB PROGRAM:6

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

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

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

```
struct node {  
    int d; // data  
    int c; // 1-red, 0-black  
    struct node* p; // parent  
    struct node* r; // right-child  
    struct node* l; // left child  
};
```

```
// global root for the entire tree
```

```
struct node* root = NULL;
```

```
// function to perform BST insertion of a node
```

```
struct node* bst(struct node* trav,  
                 struct node* temp)  
{  
    // If the tree is empty,  
    // return a new node  
    if (trav == NULL)
```

```

        return temp;

// Otherwise recur down the tree
if (temp->d < trav->d)
{
    trav->l = bst(trav->l, temp);
    trav->l->p = trav;
}
else if (temp->d > trav->d)
{
    trav->r = bst(trav->r, temp);
    trav->r->p = trav;
}

// Return the (unchanged) node pointer
return trav;
}

```

```

// Function performing right rotation
// of the passed node
void rightrotate(struct node* temp)
{
    struct node* left = temp->l;
    temp->l = left->r;
    if (temp->l)

```

```

        temp->l->p = temp;
    left->p = temp->p;
    if (!temp->p)
        root = left;
    else if (temp == temp->p->l)
        temp->p->l = left;
    else
        temp->p->r = left;
    left->r = temp;
    temp->p = left;
}

```

// Function performing left rotation

// of the passed node

```
void leftrotate(struct node* temp)
```

```

{
    struct node* right = temp->r;
    temp->r = right->l;
    if (temp->r)
        temp->r->p = temp;
    right->p = temp->p;
    if (!temp->p)
        root = right;
    else if (temp == temp->p->l)
        temp->p->l = right;
}

```

```

        else
            temp->p->r = right;
            right->l = temp;
            temp->p = right;
    }

// This function fixes violations
// caused by BST insertion
void fixup(struct node* root, struct node* pt)
{
    struct node* parent_pt = NULL;
    struct node* grand_parent_pt = NULL;

    while ((pt != root) && (pt->c != 0)
           && (pt->p->c == 1))
    {
        parent_pt = pt->p;
        grand_parent_pt = pt->p->p;

        /* Case : A
           Parent of pt is left child
           of Grand-parent of
           pt */
        if (parent_pt == grand_parent_pt->l)
        {

```

```
struct node* uncle_pt = grand_parent_pt->r;
```

```
/* Case : 1
```

```
    The uncle of pt is also red
```

```
    Only Recoloring required */
```

```
if (uncle_pt != NULL && uncle_pt->c == 1)
```

```
{
```

```
    grand_parent_pt->c = 1;
```

```
    parent_pt->c = 0;
```

```
    uncle_pt->c = 0;
```

```
    pt = grand_parent_pt;
```

```
}
```

```
else {
```

```
    /* Case : 2
```

```
        pt is right child of its parent
```

```
        Left-rotation required */
```

```
if (pt == parent_pt->r) {
```

```
    leftrotate(parent_pt);
```

```
    pt = parent_pt;
```

```
    parent_pt = pt->p;
```

```
}
```

```

        /* Case : 3
            pt is left child of its parent
            Right-rotation required */
        rightrotate(grand_parent_pt);
        int t = parent_pt->c;
        parent_pt->c = grand_parent_pt->c;
        grand_parent_pt->c = t;
        pt = parent_pt;
    }
}

/* Case : B
    Parent of pt is right
    child of Grand-parent of
    pt */
else {
    struct node* uncle_pt = grand_parent_pt->l;

    /* Case : 1
        The uncle of pt is also red
        Only Recoloring required */
    if ((uncle_pt != NULL) && (uncle_pt->c == 1))
    {
        grand_parent_pt->c = 1;
        parent_pt->c = 0;
    }
}

```



```

        uncle_pt->c = 0;
        pt = grand_parent_pt;
    }
else {
    /* Case : 2
    pt is left child of its parent
    Right-rotation required */
    if (pt == parent_pt->l) {
        rightrotate(parent_pt);
        pt = parent_pt;
        parent_pt = pt->p;
    }

    /* Case : 3
    pt is right child of its parent
    Left-rotation required */
    leftrotate(grand_parent_pt);
    int t = parent_pt->c;
    parent_pt->c = grand_parent_pt->c;
    grand_parent_pt->c = t;
    pt = parent_pt;
    }
}
}

```

```

        root->c = 0;
    }

// Function to print inorder traversal
// of the fixated tree
void inorder(struct node* trav)
{
    if (trav == NULL)
        return;
    inorder(trav->l);
    printf("%d ", trav->d);
    inorder(trav->r);
}

// driver code
int main()
{
    int n,a[20],i;
    printf("Enter the number of elements\n");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {

        printf("Enter the element %d :",i+1);
    }
}

```

```
scanf("%d",&a[i]);  
}  
  
for (int i = 0; i < n; i++) {  
    struct node* temp  
        = (struct node*)malloc(sizeof(struct node));  
    temp->r = NULL;  
    temp->l = NULL;  
    temp->p = NULL;  
    temp->d = a[i];  
    temp->c = 1;  
  
    root = bst(root, temp);  
  
    fixup(root, temp);  
}  
  
printf("Inorder Traversal of Created Tree\n");  
inorder(root);  
  
return 0;  
}
```

OUTPUT:

```
"C:\Users\Sridevi\OneDrive\Desktop\ADS LAB PROGRAMS\REDBLACK.exe"
Enter the number of elements
10
Enter the element 1 :10
Enter the element 2 :18
Enter the element 3 :7
Enter the element 4 :15
Enter the element 5 :16
Enter the element 6 :30
Enter the element 7 :25
Enter the element 8 :40
Enter the element 9 :60
Enter the element 10 :2
Inorder Traversal of Created Tree
2 7 10 15 16 18 25 30 40 60
Process returned 0 (0x0)   execution time : 42.677 s
Press any key to continue.
```

LAB PROGRAM 7:

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

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

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

```
struct node {
```

```
    int d; // data
```

```
    int c; // 1-red, 0-black
```

```
    struct node* p; // parent
```

```
    struct node* r; // right-child
```

```
    struct node* l; // left child
```

```
};
```

```
// global root for the entire tree
```

```
struct node* root = NULL;
```

```
// function to perform BST insertion of a node
```

```
struct node* bst(struct node* trav,
```

```
                struct node* temp)
```

```
{
```

```
    // If the tree is empty,
```

```
    // return a new node
```

```
    if (trav == NULL)
```

```
        return temp;
```

```

// Otherwise recur down the tree
if (temp->d < trav->d)
{
    trav->l = bst(trav->l, temp);
    trav->l->p = trav;
}
else if (temp->d > trav->d)
{
    trav->r = bst(trav->r, temp);
    trav->r->p = trav;
}

// Return the (unchanged) node pointer
return trav;
}

```

```

// Function performing right rotation
// of the passed node
void rightrotate(struct node* temp)
{
    struct node* left = temp->l;
    temp->l = left->r;
    if (temp->l)
        temp->l->p = temp;
}

```

```
    left->p = temp->p;
    if (!temp->p)
        root = left;
    else if (temp == temp->p->l)
        temp->p->l = left;
    else
        temp->p->r = left;
    left->r = temp;
    temp->p = left;
}
```

// Function performing left rotation

// of the passed node

void leftrotate(struct node* temp)

```
{
    struct node* right = temp->r;
    temp->r = right->l;
    if (temp->r)
        temp->r->p = temp;
    right->p = temp->p;
    if (!temp->p)
        root = right;
    else if (temp == temp->p->l)
        temp->p->l = right;
    else
```

```

        temp->p->r = right;
    right->l = temp;
    temp->p = right;
}

// This function fixes violations
// caused by BST insertion
void fixup(struct node* root, struct node* pt)
{
    struct node* parent_pt = NULL;
    struct node* grand_parent_pt = NULL;

    while ((pt != root) && (pt->c != 0)
           && (pt->p->c == 1))
    {
        parent_pt = pt->p;
        grand_parent_pt = pt->p->p;

        /* Case : A
           Parent of pt is left child
           of Grand-parent of
           pt */
        if (parent_pt == grand_parent_pt->l)
        {

```



```
struct node* uncle_pt = grand_parent_pt->r;
```

```
/* Case : 1
```

```
    The uncle of pt is also red
```

```
    Only Recoloring required */
```

```
if (uncle_pt != NULL && uncle_pt->c == 1)
```

```
{
```

```
    grand_parent_pt->c = 1;
```

```
    parent_pt->c = 0;
```

```
    uncle_pt->c = 0;
```

```
    pt = grand_parent_pt;
```

```
}
```

```
else {
```

```
    /* Case : 2
```

```
        pt is right child of its parent
```

```
        Left-rotation required */
```

```
if (pt == parent_pt->r) {
```

```
    leftrotate(parent_pt);
```

```
    pt = parent_pt;
```

```
    parent_pt = pt->p;
```

```
}
```

```
/* Case : 3
```

```

        pt is left child of its parent
        Right-rotation required */
    rightrotate(grand_parent_pt);
    int t = parent_pt->c;
    parent_pt->c = grand_parent_pt->c;
    grand_parent_pt->c = t;
    pt = parent_pt;
    }
}

/* Case : B
    Parent of pt is right
    child of Grand-parent of
    pt */
else {
    struct node* uncle_pt = grand_parent_pt->l;

    /* Case : 1
        The uncle of pt is also red
        Only Recoloring required */
    if ((uncle_pt != NULL) && (uncle_pt->c == 1))
    {
        grand_parent_pt->c = 1;
        parent_pt->c = 0;
        uncle_pt->c = 0;
    }
}

```

```

        pt = grand_parent_pt;
    }
    else {
        /* Case : 2
        pt is left child of its parent
        Right-rotation required */
        if (pt == parent_pt->l) {
            rightrotate(parent_pt);
            pt = parent_pt;
            parent_pt = pt->p;
        }

        /* Case : 3
        pt is right child of its parent
        Left-rotation required */
        leftrotate(grand_parent_pt);
        int t = parent_pt->c;
        parent_pt->c = grand_parent_pt->c;
        grand_parent_pt->c = t;
        pt = parent_pt;
    }
}

root->c = 0;

```

```
}
```

```
// Function to print inorder traversal
```

```
// of the fixated tree
```

```
void inorder(struct node* trav)
```

```
{
```

```
    if (trav == NULL)
```

```
        return;
```

```
    inorder(trav->l);
```

```
    printf("%d ", trav->d);
```

```
    inorder(trav->r);
```

```
}
```

```
// driver code
```

```
int main()
```

```
{
```

```
    int n,a[20],i;
```

```
    printf("Enter the number of elements\n");
```

```
    scanf("%d",&n);
```

```
    for(i=0;i<n;i++)
```

```
{
```

```
    printf("Enter the element %d :",i+1);
```

```
    scanf("%d",&a[i]);
```

```
}
```

```
for (int i = 0; i < n; i++) {
```

```
    struct node* temp
```

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

```
    temp->r = NULL;
```

```
    temp->l = NULL;
```

```
    temp->p = NULL;
```

```
    temp->d = a[i];
```

```
    temp->c = 1;
```

```
    root = bst(root, temp);
```

```
    fixup(root, temp);
```

```
}
```

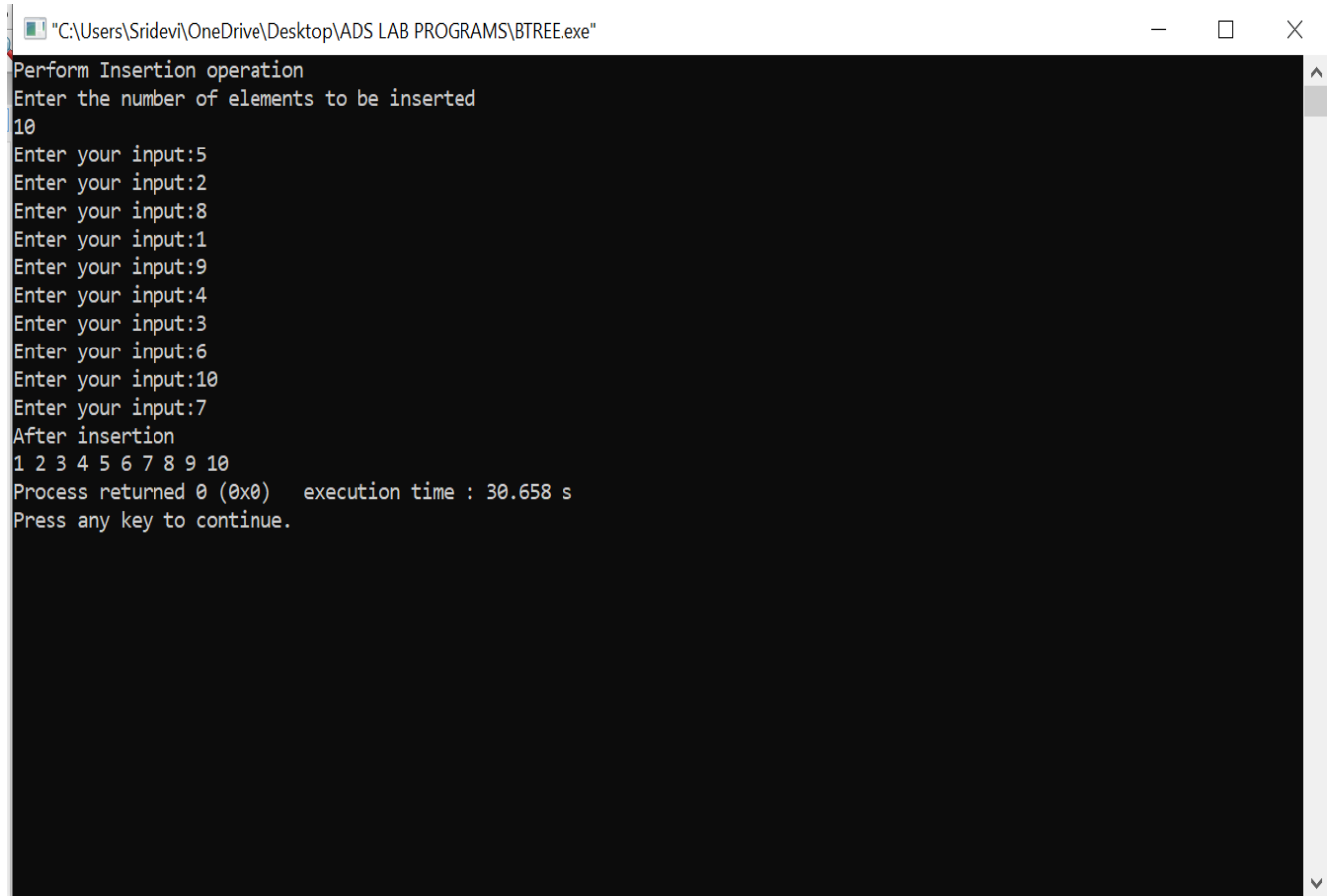
```
printf("Inorder Traversal of Created Tree\n");
```

```
inorder(root);
```

```
return 0;
```

```
}
```

OUTPUT:



```
"C:\Users\Sridevi\OneDrive\Desktop\ADS LAB PROGRAMS\BTREE.exe"
Perform Insertion operation
Enter the number of elements to be inserted
10
Enter your input:5
Enter your input:2
Enter your input:8
Enter your input:1
Enter your input:9
Enter your input:4
Enter your input:3
Enter your input:6
Enter your input:10
Enter your input:7
After insertion
1 2 3 4 5 6 7 8 9 10
Process returned 0 (0x0)   execution time : 30.658 s
Press any key to continue.
```

LAB PROGRAM 8:

Write a program to implement functions of Dictionary using Hashing.

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

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

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

```
#include <stdbool.h>
```

```
#define SIZE 20
```

```
struct DataItem {
```

```
    int data;
```

```
    int key;
```

```
};
```

```
struct DataItem* hashArray[SIZE];
```

```
struct DataItem* dummyItem;
```

```
struct DataItem* item;
```

```
int hashCode(int key) {
```

```
    return key % SIZE;
```

```
}
```

```
struct DataItem *search(int key) {
```

```
//get the hash
int hashIndex = hashCode(key);

//move in array until an empty
while(hashArray[hashIndex] != NULL) {

    if(hashArray[hashIndex]->key == key)
        return hashArray[hashIndex];

    //go to next cell
    ++hashIndex;

    //wrap around the table
    hashIndex %= SIZE;
}

return NULL;
}

void insert(int key,int data) {

    struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
    item->data = data;
    item->key = key;
```



```
//get the hash
```

```
int hashIndex = hashCode(key);
```

```
//move in array until an empty or deleted cell
```

```
while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {
```

```
    //go to next cell
```

```
    ++hashIndex;
```

```
    //wrap around the table
```

```
    hashIndex %= SIZE;
```

```
}
```

```
hashArray[hashIndex] = item;
```

```
}
```

```
struct Dataltem* delete(struct Dataltem* item) {
```

```
    int key = item->key;
```

```
    //get the hash
```

```
    int hashIndex = hashCode(key);
```

```
    //move in array until an empty
```

```
    while(hashArray[hashIndex] != NULL) {
```

```
        if(hashArray[hashIndex]->key == key) {
```

```
    struct DataItem* temp = hashArray[hashIndex];

    //assign a dummy item at deleted position
    hashArray[hashIndex] = dummyItem;
    return temp;
}

//go to next cell
++hashIndex;

//wrap around the table
hashIndex %= SIZE;
}

return NULL;
}

void display() {
    int i = 0;

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

        if(hashArray[i] != NULL)
            printf(" (%d,%d)",hashArray[i]->key,hashArray[i]->data);
        else
```

```
    printf(" ~~ ");  
}
```

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

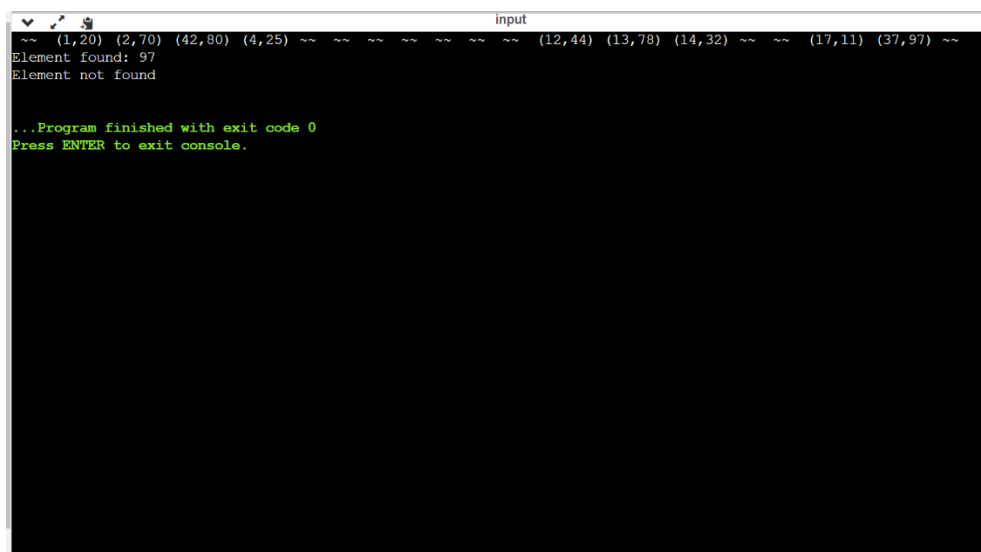
```
int main() {  
    dummyItem = (struct DataItem*) malloc(sizeof(struct DataItem));  
    dummyItem->data = -1;  
    dummyItem->key = -1;  
  
    insert(1, 20);  
    insert(2, 70);  
    insert(42, 80);  
    insert(4, 25);  
    insert(12, 44);  
    insert(14, 32);  
    insert(17, 11);  
    insert(13, 78);  
    insert(37, 97);  
  
    display();  
    item = search(37);  
  
    if(item != NULL) {
```

```
    printf("Element found: %d\n", item->data);
} else {
    printf("Element not found\n");
}

delete(item);
item = search(37);

if(item != NULL) {
    printf("Element found: %d\n", item->data);
} else {
    printf("Element not found\n");
}
}
```

OUTPUT:

A screenshot of a Windows command prompt window titled "input". The window shows the output of a C++ program. The first line of output is a list of coordinates: "(1,20) (2,70) (42,80) (4,25) ~ ~ ~ ~ ~ ~ ~ ~ (12,44) (13,78) (14,32) ~ ~ ~ (17,11) (37,97) ~ ~". Below this, the program prints "Element found: 97" and "Element not found". At the bottom, it says "...Program finished with exit code 0" and "Press ENTER to exit console." in green text.

```
input
~~ (1,20) (2,70) (42,80) (4,25) ~ ~ ~ ~ ~ ~ ~ ~ (12,44) (13,78) (14,32) ~ ~ ~ (17,11) (37,97) ~~
Element found: 97
Element not found

...Program finished with exit code 0
Press ENTER to exit console.
```

LAB PROGRAM 9:

Write a program to implement the following functions on a Binomial heap:

1. insert(H, k): Inserts a key 'k' to Binomial Heap 'H'. This operation first creates a Binomial Heap with single key 'k', then calls union on H and the new Binomial heap.

2. getMin(H): A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key.

3. extractMin(H): This operation also uses union(). We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally we call union() on H and the newly created Binomial Heap.

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

```
#include<malloc.h>
```

```
struct node {
```

```
    int n;
```

```
    int degree;
```

```
    struct node* parent;
```

```
    struct node* child;
```

```
    struct node* sibling;
```

```
};
```

```
struct node* MAKE_bin_HEAP();
```

```
int bin_LINK(struct node*, struct node*);  
struct node* CREATE_NODE(int);  
struct node* bin_HEAP_UNION(struct node*, struct node*);  
struct node* bin_HEAP_INSERT(struct node*, struct node*);  
struct node* bin_HEAP_MERGE(struct node*, struct node*);  
struct node* bin_HEAP_EXTRACT_MIN(struct node*);  
int REVERT_LIST(struct node*);  
int DISPLAY(struct node*);  
struct node* FIND_NODE(struct node*, int);  
int bin_HEAP_DECREASE_KEY(struct node*, int, int);  
int bin_HEAP_DELETE(struct node*, int);
```

```
int count = 1;
```

```
struct node* MAKE_bin_HEAP() {  
    struct node* np;  
    np = NULL;  
    return np;  
}
```

```
struct node * H = NULL;  
struct node *Hr = NULL;
```

```
int bin_LINK(struct node* y, struct node* z) {  
    y->parent = z;
```

```
y->sibling = z->child;
z->child = y;
z->degree = z->degree + 1;
}
```

```
struct node* CREATE_NODE(int k) {
    struct node* p;//new node;
    p = (struct node*) malloc(sizeof(struct node));
    p->n = k;
    return p;
}
```

```
struct node* bin_HEAP_UNION(struct node* H1, struct node* H2) {
    struct node* prev_x;
    struct node* next_x;
    struct node* x;
    struct node* H = MAKE_bin_HEAP();
    H = bin_HEAP_MERGE(H1, H2);
    if (H == NULL)
        return H;
    prev_x = NULL;
    x = H;
    next_x = x->sibling;
    while (next_x != NULL) {
        if ((x->degree != next_x->degree) || ((next_x->sibling != NULL)
```

```

        && (next_x->sibling->degree == x->degree)) {
    prev_x = x;
    x = next_x;
} else {
    if (x->n <= next_x->n) {
        x->sibling = next_x->sibling;
        bin_LINK(next_x, x);
    } else {
        if (prev_x == NULL)
            H = next_x;
        else
            prev_x->sibling = next_x;
        bin_LINK(x, next_x);
        x = next_x;
    }
}
next_x = x->sibling;
}
return H;
}

```

```

struct node* bin_HEAP_INSERT(struct node* H, struct node* x) {
    struct node* H1 = MAKE_bin_HEAP();
    x->parent = NULL;
    x->child = NULL;
}

```



```

x->sibling = NULL;
x->degree = 0;
H1 = x;
H = bin_HEAP_UNION(H, H1);
return H;
}

```

```

struct node* bin_HEAP_MERGE(struct node* H1, struct node* H2) {
    struct node* H = MAKE_bin_HEAP();
    struct node* y;
    struct node* z;
    struct node* a;
    struct node* b;
    y = H1;
    z = H2;
    if (y != NULL) {
        if (z != NULL && y->degree <= z->degree)
            H = y;
        else if (z != NULL && y->degree > z->degree)
            /* need some modifications here;the first and the else conditions can be
            merged together!!!! */
            H = z;
        else
            H = y;
    } else

```

```

    H = z;
while (y != NULL && z != NULL) {
    if (y->degree < z->degree) {
        y = y->sibling;
    } else if (y->degree == z->degree) {
        a = y->sibling;
        y->sibling = z;
        y = a;
    } else {
        b = z->sibling;
        z->sibling = y;
        z = b;
    }
}
return H;
}

```

```

int DISPLAY(struct node* H) {
    struct node* p;
    if (H == NULL) {
        printf("\nHEAP EMPTY");
        return 0;
    }
    printf("\nTHE ROOT NODES ARE:-\n");
    p = H;

```

```

while (p != NULL) {
    printf("%d", p->n);
    if (p->sibling != NULL)
        printf("-->");
    p = p->sibling;
}
printf("\n");
}

```

```

struct node* bin_HEAP_EXTRACT_MIN(struct node* H1) {
    int min;
    struct node* t = NULL;
    struct node* x = H1;
    struct node *Hr;
    struct node* p;
    Hr = NULL;
    if (x == NULL) {
        printf("\nNOTHING TO EXTRACT");
        return x;
    }
    // int min=x->n;
    p = x;
    while (p->sibling != NULL) {
        if ((p->sibling)->n < min) {
            min = (p->sibling)->n;

```

```

        t = p;
        x = p->sibling;
    }
    p = p->sibling;
}
if (t == NULL && x->sibling == NULL)
    H1 = NULL;
else if (t == NULL)
    H1 = x->sibling;
else if (t->sibling == NULL)
    t = NULL;
else
    t->sibling = x->sibling;
if (x->child != NULL) {
    REVERT_LIST(x->child);
    (x->child)->sibling = NULL;
}
H = bin_HEAP_UNION(H1, Hr);
return x;
}

```

```

int REVERT_LIST(struct node* y) {
    if (y->sibling != NULL) {
        REVERT_LIST(y->sibling);
        (y->sibling)->sibling = y;
    }
}

```

```
    } else {  
        Hr = y;  
    }  
}
```

```
struct node* FIND_NODE(struct node* H, int k) {  
    struct node* x = H;  
    struct node* p = NULL;  
    if (x->n == k) {  
        p = x;  
        return p;  
    }  
    if (x->child != NULL && p == NULL) {  
        p = FIND_NODE(x->child, k);  
    }  
  
    if (x->sibling != NULL && p == NULL) {  
        p = FIND_NODE(x->sibling, k);  
    }  
    return p;  
}
```

```
int bin_HEAP_DECREASE_KEY(struct node* H, int i, int k) {  
    int temp;  
    struct node* p;
```

```

struct node* y;
struct node* z;
p = FIND_NODE(H, i);
if (p == NULL) {
    printf("\nINVALID CHOICE OF KEY TO BE REDUCED");
    return 0;
}
if (k > p->n) {
    printf("\nSORRY!THE NEW KEY IS GREATER THAN CURRENT ONE");
    return 0;
}
p->n = k;
y = p;
z = p->parent;
while (z != NULL && y->n < z->n) {
    temp = y->n;
    y->n = z->n;
    z->n = temp;
    y = z;
    z = z->parent;
}
printf("\nKEY REDUCED SUCCESSFULLY!");
}

int bin_HEAP_DELETE(struct node* H, int k) {

```

```
struct node* np;  
if (H == NULL) {  
    printf("\nHEAP EMPTY");  
    return 0;  
}
```

```
bin_HEAP_DECREASE_KEY(H, k, -1000);  
np = bin_HEAP_EXTRACT_MIN(H);  
if (np != NULL)  
    printf("\nNODE DELETED SUCCESSFULLY");  
}
```

```
int main() {  
    int i, n, m, l;  
    struct node* p;  
    struct node* np;  
    char ch;  
    printf("\nENTER THE NUMBER OF ELEMENTS:");  
    scanf("%d", &n);  
    printf("\nENTER THE ELEMENTS:\n");  
    for (i = 1; i <= n; i++) {  
        scanf("%d", &m);  
        np = CREATE_NODE(m);  
        H = bin_HEAP_INSERT(H, np);  
    }
```

```

DISPLAY(H);
do {
    printf("\nMENU:-\n");
    printf(
        "\n1)INSERT AN ELEMENT\n2)EXTRACT THE MINIMUM KEY
NODE\n3)QUIT\n");
    scanf("%d", &l);
    switch (l) {
    case 1:
        do {
            printf("\nENTER THE ELEMENT TO BE INSERTED:");
            scanf("%d", &m);
            p = CREATE_NODE(m);
            H = bin_HEAP_INSERT(H, p);
            printf("\nNOW THE HEAP IS:\n");
            DISPLAY(H);
            printf("\nINSERT MORE(y/Y)= \n");
            fflush(stdin);
            scanf("%c", &ch);
        } while (ch == 'Y' || ch == 'y');
        break;
    case 2:
        do {
            printf("\nEXTRACTING THE MINIMUM KEY NODE");
            p = bin_HEAP_EXTRACT_MIN(H);

```



```

        if (p != NULL)
            printf("\nTHE EXTRACTED NODE IS %d", p->n);
        printf("\nNOW THE HEAP IS:\n");
        DISPLAY(H);
        printf("\nEXTRACT MORE(y/Y)\n");
        fflush(stdin);
        scanf("%c", &ch);
    } while (ch == 'Y' || ch == 'y');
    break;
case 3:
    do {
        printf("\nENTER THE KEY OF THE NODE TO BE DECREASED:");
        scanf("%d", &m);
        printf("\nENTER THE NEW KEY : ");
        scanf("%d", &l);
        bin_HEAP_DECREASE_KEY(H, m, l);
        printf("\nNOW THE HEAP IS:\n");
        DISPLAY(H);
        printf("\nDECREASE MORE(y/Y)\n");
        fflush(stdin);
        scanf("%c", &ch);
    } while (ch == 'Y' || ch == 'y');
    break;
case 4:
    do {

```

```
    printf("\nEnter the key to be deleted: ");
    scanf("%d", &m);
    bin_HEAP_DELETE(H, m);
    printf("\nDelete more (y/Y)\n");
    fflush(stdin);
    scanf("%c", &ch);
} while (ch == 'y' || ch == 'Y');
break;
case 5:
    printf("\nWrong choice\n");
    break;
default:
    printf("\nInvalid entry...try again....\n");
}
} while (l != 5);
}
```

OUTPUT:

```
"C:\Users\Sridevi\OneDrive\Desktop\ADS LAB PROGRAMS\BINOMIALHEAP.exe"
ENTER THE NUMBER OF ELEMENTS:10
ENTER THE ELEMENTS:
12 66 83 98 24 88 35 87 54 78 87
THE ROOT NODES ARE:-
54-->8
MENU:-
1)INSERT AN ELEMENT
2)EXTRACT THE MINIMUM KEY NODE
3)DECREASE A NODE KEY
4)DELETE A NODE
5)QUIT
INVALID ENTRY...TRY AGAIN...
MENU:-
1)INSERT AN ELEMENT
2)EXTRACT THE MINIMUM KEY NODE
3)DECREASE A NODE KEY
4)DELETE A NODE
5)QUIT
1
ENTER THE ELEMENT TO BE INSERTED:8
```

```
"C:\Users\Sridevi\OneDrive\Desktop\ADS LAB PROGRAMS\BINOMIALHEAP.exe"
2)EXTRACT THE MINIMUM KEY NODE
3)QUIT
1
ENTER THE ELEMENT TO BE INSERTED:8
NOW THE HEAP IS:
THE ROOT NODES ARE:-
8-->67-->33
INSERT MORE(y/Y)=
N
MENU:-
1)INSERT AN ELEMENT
2)EXTRACT THE MINIMUM KEY NODE
3)QUIT
2
EXTRACTING THE MINIMUM KEY NODE
THE EXTRACTED NODE IS 8
NOW THE HEAP IS:
THE ROOT NODES ARE:-
67-->33
EXTRACT MORE(y/Y)
```

```
"C:\Users\Sridevi\OneDrive\Desktop\ADS LAB PROGRAMS\BINOMIALHEAP.exe"
ENTER THE ELEMENTS:
46 55 46 86 33 89 99 100 346 67
THE ROOT NODES ARE:-
67-->33
MENU:-
1)INSERT AN ELEMENT
2)EXTRACT THE MINIMUM KEY NODE
3)QUIT
1
ENTER THE ELEMENT TO BE INSERTED:8
NOW THE HEAP IS:
THE ROOT NODES ARE:-
8-->67-->33
INSERT MORE(y/Y)=
N
MENU:-
1)INSERT AN ELEMENT
2)EXTRACT THE MINIMUM KEY NODE
3)QUIT
```

LAB PROGRAM 10:

Write a program to implement the following functions on a Binomial heap:

1. delete(H): Like Binary Heap, delete operation first reduces the key to minus infinite, then calls extractMin().

2. decreaseKey(H): decreaseKey() is also similar to Binary Heap. We compare the decreases key with it parent and if parent's key is more, we swap keys and recur for parent. We stop when we either reach a node whose parent has smaller key or we hit the root node.

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

```
#include<malloc.h>
```

```
struct node {  
    int n;  
    int degree;  
    struct node* parent;  
    struct node* child;  
    struct node* sibling;  
};
```

```
struct node* MAKE_bin_HEAP();  
int bin_LINK(struct node*, struct node*);  
struct node* CREATE_NODE(int);  
struct node* bin_HEAP_UNION(struct node*, struct node*);  
struct node* bin_HEAP_INSERT(struct node*, struct node*);
```

```
struct node* bin_HEAP_MERGE(struct node*, struct node*);  
struct node* bin_HEAP_EXTRACT_MIN(struct node*);  
int REVERT_LIST(struct node*);  
int DISPLAY(struct node*);  
struct node* FIND_NODE(struct node*, int);  
int bin_HEAP_DECREASE_KEY(struct node*, int, int);  
int bin_HEAP_DELETE(struct node*, int);
```

```
int count = 1;
```

```
struct node* MAKE_bin_HEAP() {  
    struct node* np;  
    np = NULL;  
    return np;  
}
```

```
struct node * H = NULL;  
struct node *Hr = NULL;
```

```
int bin_LINK(struct node* y, struct node* z) {  
    y->parent = z;  
    y->sibling = z->child;  
    z->child = y;  
    z->degree = z->degree + 1;  
}
```

```

struct node* CREATE_NODE(int k) {
    struct node* p;//new node;
    p = (struct node*) malloc(sizeof(struct node));
    p->n = k;
    return p;
}

```

```

struct node* bin_HEAP_UNION(struct node* H1, struct node* H2) {
    struct node* prev_x;
    struct node* next_x;
    struct node* x;
    struct node* H = MAKE_bin_HEAP();
    H = bin_HEAP_MERGE(H1, H2);
    if (H == NULL)
        return H;
    prev_x = NULL;
    x = H;
    next_x = x->sibling;
    while (next_x != NULL) {
        if ((x->degree != next_x->degree) || ((next_x->sibling != NULL)
            && (next_x->sibling->degree == x->degree)) {
            prev_x = x;
            x = next_x;
            next_x = x->sibling;
        } else {

```

```

    if (x->n <= next_x->n) {
        x->sibling = next_x->sibling;
        bin_LINK(next_x, x);
    } else {
        if (prev_x == NULL)
            H = next_x;
        else
            prev_x->sibling = next_x;
        bin_LINK(x, next_x);
        x = next_x;
    }
}
next_x = x->sibling;
}
return H;
}

```

```

struct node* bin_HEAP_INSERT(struct node* H, struct node* x) {
    struct node* H1 = MAKE_bin_HEAP();
    x->parent = NULL;
    x->child = NULL;
    x->sibling = NULL;
    x->degree = 0;
    H1 = x;
    H = bin_HEAP_UNION(H, H1);
}

```

```
    return H;
}
```

```
struct node* bin_HEAP_MERGE(struct node* H1, struct node* H2) {
    struct node* H = MAKE_bin_HEAP();
    struct node* y;
    struct node* z;
    struct node* a;
    struct node* b;
    y = H1;
    z = H2;
    if (y != NULL) {
        if (z != NULL && y->degree <= z->degree)
            H = y;
        else if (z != NULL && y->degree > z->degree)
            /* need some modifications here;the first and the else conditions can be
            merged together!!!! */
            H = z;
        else
            H = y;
    } else
        H = z;
    while (y != NULL && z != NULL) {
        if (y->degree < z->degree) {
            y = y->sibling;
```



```

    } else if (y->degree == z->degree) {
        a = y->sibling;
        y->sibling = z;
        y = a;
    } else {
        b = z->sibling;
        z->sibling = y;
        z = b;
    }
}
return H;
}

```

```

int DISPLAY(struct node* H) {
    struct node* p;
    if (H == NULL) {
        printf("\nHEAP EMPTY");
        return 0;
    }
    printf("\nTHE ROOT NODES ARE:-\n");
    p = H;
    while (p != NULL) {
        printf("%d", p->n);
        if (p->sibling != NULL)
            printf("-->");
    }
}

```

```
    p = p->sibling;
}
printf("\n");
}
```

```
struct node* bin_HEAP_EXTRACT_MIN(struct node* H1) {
    int min;
    struct node* t = NULL;
    struct node* x = H1;
    struct node *Hr;
    struct node* p;
    Hr = NULL;
    if (x == NULL) {
        printf("\nNOTHING TO EXTRACT");
        return x;
    }
    // int min=x->n;
    p = x;
    while (p->sibling != NULL) {
        if ((p->sibling)->n < min) {
            min = (p->sibling)->n;
            t = p;
            x = p->sibling;
        }
        p = p->sibling;
    }
```

```

    }
    if (t == NULL && x->sibling == NULL)
        H1 = NULL;
    else if (t == NULL)
        H1 = x->sibling;
    else if (t->sibling == NULL)
        t = NULL;
    else
        t->sibling = x->sibling;
    if (x->child != NULL) {
        REVERT_LIST(x->child);
        (x->child)->sibling = NULL;
    }
    H = bin_HEAP_UNION(H1, Hr);
    return x;
}

```

```

int REVERT_LIST(struct node* y) {
    if (y->sibling != NULL) {
        REVERT_LIST(y->sibling);
        (y->sibling)->sibling = y;
    } else {
        Hr = y;
    }
}

```

```
struct node* FIND_NODE(struct node* H, int k) {
```

```
    struct node* x = H;
```

```
    struct node* p = NULL;
```

```
    if (x->n == k) {
```

```
        p = x;
```

```
        return p;
```

```
    }
```

```
    if (x->child != NULL && p == NULL) {
```

```
        p = FIND_NODE(x->child, k);
```

```
    }
```

```
    if (x->sibling != NULL && p == NULL) {
```

```
        p = FIND_NODE(x->sibling, k);
```

```
    }
```

```
    return p;
```

```
}
```

```
int bin_HEAP_DECREASE_KEY(struct node* H, int i, int k) {
```

```
    int temp;
```

```
    struct node* p;
```

```
    struct node* y;
```

```
    struct node* z;
```

```
    p = FIND_NODE(H, i);
```

```
    if (p == NULL) {
```

```

    printf("\nINVALID CHOICE OF KEY TO BE REDUCED");
    return 0;
}
if (k > p->n) {
    printf("\nSORRY!THE NEW KEY IS GREATER THAN CURRENT ONE");
    return 0;
}
p->n = k;
y = p;
z = p->parent;
while (z != NULL && y->n < z->n) {
    temp = y->n;
    y->n = z->n;
    z->n = temp;
    y = z;
    z = z->parent;
}
printf("\nKEY REDUCED SUCCESSFULLY!");
}

```

```

int bin_HEAP_DELETE(struct node* H, int k) {
    struct node* np;
    if (H == NULL) {
        printf("\nHEAP EMPTY");
        return 0;
    }
}

```

```
}
```

```
bin_HEAP_DECREASE_KEY(H, k, -1000);
```

```
np = bin_HEAP_EXTRACT_MIN(H);
```

```
if (np != NULL)
```

```
    printf("\nNODE DELETED SUCCESSFULLY");
```

```
}
```

```
int main() {
```

```
    int i, n, m, l;
```

```
    struct node* p;
```

```
    struct node* np;
```

```
    char ch;
```

```
    printf("\nENTER THE NUMBER OF ELEMENTS:");
```

```
    scanf("%d", &n);
```

```
    printf("\nENTER THE ELEMENTS:\n");
```

```
    for (i = 1; i <= n; i++) {
```

```
        scanf("%d", &m);
```

```
        np = CREATE_NODE(m);
```

```
        H = bin_HEAP_INSERT(H, np);
```

```
    }
```

```
    DISPLAY(H);
```

```
    do {
```

```
        printf("\nMENU:-\n");
```

```
        printf(
```

```
"\n1)INSERT AN ELEMENT\n2)EXTRACT THE MINIMUM KEY  
NODE\n3)QUIT\n");
```

```
scanf("%d", &l);
```

```
switch (l) {
```

```
case 1:
```

```
do {
```

```
    printf("\nENTER THE ELEMENT TO BE INSERTED:");
```

```
    scanf("%d", &m);
```

```
    p = CREATE_NODE(m);
```

```
    H = bin_HEAP_INSERT(H, p);
```

```
    printf("\nNOW THE HEAP IS:\n");
```

```
    DISPLAY(H);
```

```
    printf("\nINSERT MORE(y/Y)= \n");
```

```
    fflush(stdin);
```

```
    scanf("%c", &ch);
```

```
} while (ch == 'Y' || ch == 'y');
```

```
break;
```

```
case 2:
```

```
do {
```

```
    printf("\nEXTRACTING THE MINIMUM KEY NODE");
```

```
    p = bin_HEAP_EXTRACT_MIN(H);
```

```
    if (p != NULL)
```

```
        printf("\nTHE EXTRACTED NODE IS %d", p->n);
```

```
    printf("\nNOW THE HEAP IS:\n");
```

```
    DISPLAY(H);
```

```

        printf("\nEXTRACT MORE(y/Y)\n");
        fflush(stdin);
        scanf("%c", &ch);
    } while (ch == 'Y' || ch == 'y');
    break;
case 3:
    do {
        printf("\nENTER THE KEY OF THE NODE TO BE DECREASED:");
        scanf("%d", &m);
        printf("\nENTER THE NEW KEY : ");
        scanf("%d", &l);
        bin_HEAP_DECREASE_KEY(H, m, l);
        printf("\nNOW THE HEAP IS:\n");
        DISPLAY(H);
        printf("\nDECREASE MORE(y/Y)\n");
        fflush(stdin);
        scanf("%c", &ch);
    } while (ch == 'Y' || ch == 'y');
    break;
case 4:
    do {
        printf("\nENTER THE KEY TO BE DELETED: ");
        scanf("%d", &m);
        bin_HEAP_DELETE(H, m);
        printf("\nDELETE MORE(y/Y)\n");

```



```
        fflush(stdin);
        scanf("%c", &ch);
    } while (ch == 'y' || ch == 'Y');
    break;
case 5:
    printf("\n Wrong Choice\n");
    break;
default:
    printf("\nINVALID ENTRY...TRY AGAIN....\n");
}
} while (l != 5);
}
```

OUTPUT:

```
"C:\Users\Sridevi\OneDrive\Desktop\ADS LAB PROGRAMS\BINOMIALHEAP.exe"
MENU:-
1)INSERT AN ELEMENT
2)EXTRACT THE MINIMUM KEY NODE
3)DECREASE A NODE KEY
  4)DELETE A NODE
5)QUIT
3
ENTER THE KEY OF THE NODE TO BE DECREASED:74
ENTER THE NEW KEY : 8
INVALID CHOICE OF KEY TO BE REDUCED
NOW THE HEAP IS:
THE ROOT NODES ARE:-
345
DECREASE MORE(y/Y)
N
MENU:-
1)INSERT AN ELEMENT
2)EXTRACT THE MINIMUM KEY NODE
3)DECREASE A NODE KEY
  4)DELETE A NODE
5)QUIT
```

```
"C:\Users\Sridevi\OneDrive\Desktop\ADS LAB PROGRAMS\BINOMIALHEAP.exe"
ENTER THE ELEMENTS:
12 64 55 74 345
THE ROOT NODES ARE:-
345-->12
MENU:-
1)INSERT AN ELEMENT
2)EXTRACT THE MINIMUM KEY NODE
3)DECREASE A NODE KEY
  4)DELETE A NODE
5)QUIT
4
ENTER THE KEY TO BE DELETED: 55
KEY REDUCED SUCCESSFULLY!
NODE DELETED SUCCESSFULLY
DELETE MORE(y/Y)
N
MENU:-
1)INSERT AN ELEMENT
2)EXTRACT THE MINIMUM KEY NODE
3)DECREASE A NODE KEY
  4)DELETE A NODE
5)QUIT
```

