# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

**"JnanaSangama", Belgaum -590014, Karnataka.**

**LAB REPORT**
**On**

## ADVANCED DATA STRUCTURES (22CS5PEADS)

**Submitted by**

**ADVITHI D (1BM21CS009)**

**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**
**March -June 2024**

**B. M. S. College of Engineering,**
**Bull Temple Road, Bangalore 560019**
**(Affiliated To Visvesvaraya Technological University, Belgaum)**
**Department of Computer Science and Engineering**

This is to certify that the Lab work entitled **"ADVANCED DATA STRUCTURES"** carried out by ADVITHI D **(1BM21CS009)**, 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 2023-24. The Lab report has been approved as it satisfies the academic requirements in respect of Advanced Data structures Lab - **(22CS5PEADS)** work prescribed for the said degree.


**Prof. Namratha M**
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

## Index Sheet

**Course outcomes:**

| | |
|---|---|
| CO1 | Apply the concepts of advanced data structures for the given scenario. |
| CO2 | Analyze the usage of appropriate data structure for a given application. |
| CO3 | Design algorithms for performing operations on various advanced data structures. |
| CO4 | 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>
#include <stdint.h>
struct Node {
    int data;
    struct Node* both;
};
struct Node* XOR(struct Node* a, struct Node* b) {
    return (struct Node*)((uintptr_t)(a) ^ (uintptr_t)(b));
}
void insertNode(struct Node** head, int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->both = XOR(*head, NULL);
    if (*head != NULL) {
        struct Node* next = XOR((*head)->both, NULL);
        (*head)->both = XOR(new_node, next);
    }
    *head = new_node;
}
void displayList(struct Node* head) {
    struct Node* current = head;
    struct Node* prev = NULL;
```

```
    struct Node* next;
    while (current != NULL) {
        printf("%d ", current->data);
        next = XOR(prev, current->both);
        prev = current;
        current = next;
    }
    printf("\n");
}
int main() {
    struct Node* head = NULL;
    int n, data;
    printf("Enter the number of elements to insert: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        printf("Enter element %d: ", i + 1);
        scanf("%d", &data);
        insertNode(&head, data);
    }
    printf("XOR Linked List: ");
    displayList(head);
    return 0;
}
```

```
Enter the number of elements to insert: 5
Enter element 1: 3
Enter element 2: 4
Enter element 3: 7
Enter element 4: 2
Enter element 5: 8
XOR Linked List: 8 2 7 4 3
```

```
Enter the number of elements to insert: 4
Enter element 1: 8
Enter element 2: 4
Enter element 3: 2
Enter element 4: 5
XOR Linked List: 5 2 4 8
```

**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 = (skiplist *)malloc(sizeof(skiplist));
    if (list == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    snode *header = (snode *)malloc(sizeof(snode));
    if (header == NULL) {
```

```
        printf("Memory allocation failed!\n");
        exit(1);
    header->key = INT_MAX;
    header->forward = (snode **)malloc(sizeof(snode *) * (SKIPLIST_MAX_LEVEL + 1));
    if (header->forward == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    for (int i = 0; i <= SKIPLIST_MAX_LEVEL; i++) {
        header->forward[i] = NULL;
    }
    list->header = 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;
}
void skiplist_insert(skiplist *list, int key, int value) {
    snode *update[SKIPLIST_MAX_LEVEL + 1];
    snode *x = list->header;
    for (int i = list->level; i >= 1; i--) {
        while (x->forward[i] != NULL && x->forward[i]->key < key)
            x = x->forward[i];
        update[i] = x;
```

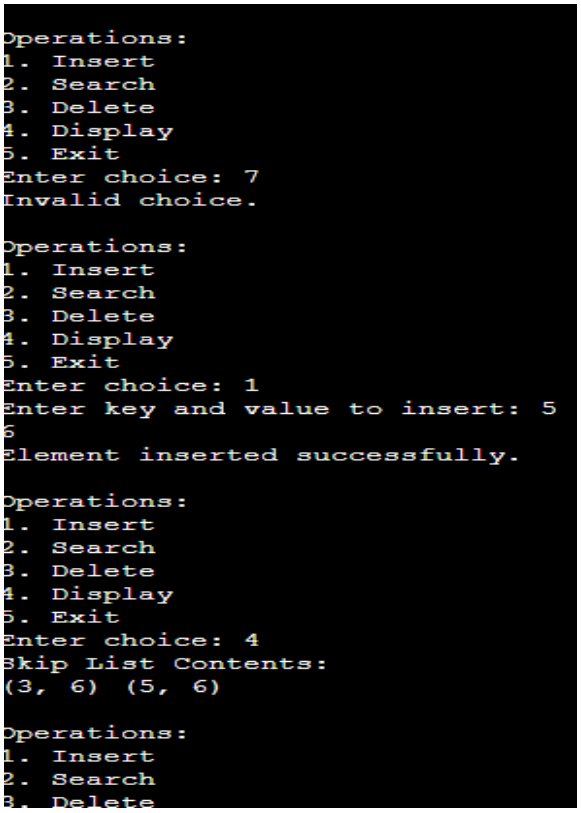
```
    }
    x = x->forward[1];
    if (x != NULL && x->key == key) {
        x->value = value;
    } else {
        int level = rand_level();
        if (level > list->level) {
            for (int i = list->level + 1; i <= level; i++) {
                update[i] = list->header;
            }
            list->level = level;
        }
        x = (snode *)malloc(sizeof(snode));
        if (x == NULL) {
            printf("Memory allocation failed\n");
            exit(1);
        }
        x->key = key;
        x->value = value;
        x->forward = (snode **)malloc(sizeof(snode *) * (level + 1));
        if (x->forward == NULL) {
            printf("Memory allocation failed\n");
            exit(1);
        }
        for (int i = 1; i <= level; i++) {
            x->forward[i] = update[i]->forward[i];
            update[i]->forward[i] = x;
        }
        list->size++;
```

```
    }
}
snode *skiplist_search(skiplist *list, int key) {
    snode *x = list->header;
    for (int i = list->level; i >= 1; i--) {
        while (x->forward[i] != NULL && x->forward[i]->key < key)
            x = x->forward[i];
    }
    if (x->forward[1] != NULL && x->forward[1]->key == key) {
        return x->forward[1];
    } else {
        return NULL;
    }
}
void skiplist_delete(skiplist *list, int key) {
    snode *update[SKIPLIST_MAX_LEVEL + 1];
    snode *x = list->header;
    for (int i = list->level; i >= 1; i--) {
        while (x->forward[i] != NULL && x->forward[i]->key < key)
            x = x->forward[i];
        update[i] = x;
    }
    x = x->forward[1];
    if (x != NULL && x->key == key) {
        for (int i = 1; i <= list->level; i++) {
            if (update[i]->forward[i] != x)
                break;
            update[i]->forward[i] = x->forward[i];
        }
```

```
        free(x->forward);
        free(x);
        while (list->level > 1 && list->header->forward[list->level] == NULL)
            list->level--;
        list->size--;
    }
}
void skiplist_dump(skiplist *list) {
    printf("Skip List Contents:\n");
    snode *x = list->header->forward[1];
    while (x != NULL) {
        printf("(%d, %d) ", x->key, x->value);
        x = x->forward[1];
    }
    printf("\n");
}
int main() {
    skiplist *list = skiplist_init();
    int choice, key, value;
    snode *result;
    do {
        printf("\nOperations:\n");
        printf("1. Insert\n");
        printf("2. Search\n");
        printf("3. Delete\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);
```

```
switch (choice) {
    case 1:
        printf("Enter key and value to insert: ");
        scanf("%d %d", &key, &value);
        skiplist_insert(list, key, value);
        printf("Element inserted successfully.\n");
        break;
    case 2:
        printf("Enter key to search: ");
        scanf("%d", &key);
        result = skiplist_search(list, key);
        if (result != NULL)
            printf("Key %d found with value %d\n", key, result->value);
        else
            printf("Key %d not found\n", key);
        break;
    case 3:
        printf("Enter key to delete: ");
        scanf("%d", &key);
        skiplist_delete(list, key);
        printf("Element deleted.\n");
        break;
    case 4:
        skiplist_dump(list);
        break;
   case 5:
        printf("Exiting.\n");
        break;
    default:
```

```
            printf("Invalid choice.\n");
        }
    } while (choice != 5);
    return 0;
}
```



```
4. Display
5. Exit
Enter choice: 4
Skip List Contents:
(1, 2) (6, 4) (7, 4)

Operations:
1. Insert
2. Search
3. Delete
4. Display
5. Exit
Enter choice: 3
Enter key to delete: 6
Element deleted.

Operations:
1. Insert
2. Search
3. Delete
4. Display
5. Exit
Enter choice: 4
Skip List Contents:
(1, 2) (7, 4)

Operations:
1. Insert
2. Search
3. Delete
4. Display
5. Exit
Enter choice: 2
Enter key to search: 7
Key 7 found with value 4
```

**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 find(int **M, int i, int j, int ROW, int COL) {

   if (i < 0 || j < 0 || i >= ROW || j >= COL || M[i][j] != 1) {

      return;

   }
```

```
    M[i][j] = 0;
    find(M, i + 1, j, ROW, COL);
    find(M, i - 1, j, ROW, COL);
    find(M, i, j + 1, ROW, COL);
    find(M, i, j - 1, ROW, COL);
    find(M, i + 1, j + 1, ROW, COL);
    find(M, i - 1, j - 1, ROW, COL);
    find(M, i + 1, j - 1, ROW, COL);
    find(M, i - 1, j + 1, ROW, COL);
}
int countIslands(int **M, int ROW, int COL) {
    int count = 0;
    for (int i = 0; i < ROW; i++) {
        for (int j = 0; j < COL; j++) {
            if (M[i][j] == 1) {
                count++;
                find(M, i, j, ROW, COL);
            }
        }
    }
    return count;
}
int main() {
    int ROW, COL;
    printf("Enter number of rows: ");
    scanf("%d", &ROW);
    printf("Enter number of columns: ");
    scanf("%d", &COL);
    int **M = (int **)malloc(ROW * sizeof(int *));
```

```
    for (int i = 0; i < ROW; i++) {
        M[i] = (int *)malloc(COL * sizeof(int));
    }
    printf("Enter the matrix elements (0 or 1):\n");
    for (int i = 0; i < ROW; i++) {
        for (int j = 0; j < COL; j++) {
            scanf("%d", &M[i][j]);
        }
    }
    printf("Original Matrix:\n");
    for (int i = 0; i < ROW; i++) {
        for (int j = 0; j < COL; j++) {
            printf("%d ", M[i][j]);
        }
        printf("\n");
    }
    printf("Number of islands is: %d\n", countIslands(M, ROW, COL));
    for (int i = 0; i < ROW; i++) {
        free(M[i]);
    }
    free(M);
    return 0;
}
```

```
Enter the number of rows: 3
Enter the number of columns: 3
Enter the matrix elements (0 or 1):
1
0
1
1
1
1
10
0
0
Number of islands: 1
```

```
Enter number of rows: 3
Enter number of columns: 3
Enter the matrix elements (0 or 1):
1
1
1
1
1
1
10
0
0
Original Matrix:
1 1 1
1 1 1
10 0 0
Number of islands is: 1
```

**Lab program 4:**

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

```
#include <stdio.h>
#include <stdlib.h>
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* insert(struct Node* node, int key) {
        if (node == NULL)
                return(newNode(key));
        if (key < node->key)
                node->left = insert(node->left, key);
        else if (key > node->key)
                node->right = insert(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);
```

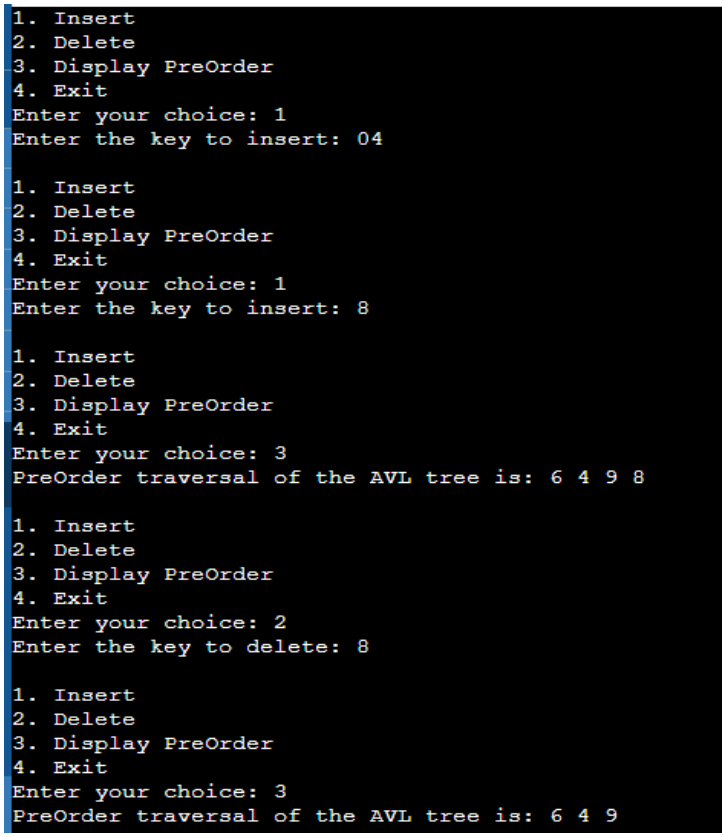
```
        }
        if (balance < -1 && key < node->right->key) {
                node->right = rightRotate(node->right);
                return leftRotate(node);
        }
        return node;
}
struct Node *minValueNode(struct Node* node) {
        struct Node* current = node;
        while (current->left != 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 preOrder(struct Node *root) {
        if (root != NULL) {
            printf("%d ", root->key);
            preOrder(root->left);
            preOrder(root->right);
```

```
        }
}
int main() {
        struct Node *root = NULL;
        int choice, key;
        while (1) {
                printf("\n1. Insert\n2. Delete\n3. Display PreOrder\n4. Exit\nEnter your choice: ");
                scanf("%d", &choice);
                switch (choice) {
                        case 1:
                                printf("Enter the key to insert: ");
                                scanf("%d", &key);
                                root = insert(root, key);
                                break;
                        case 2:
                                printf("Enter the key to delete: ");
                                scanf("%d", &key);
                                root = deleteNode(root, key);
                                break;
                        case 3:
                                printf("PreOrder traversal of the AVL tree is: ");
                                preOrder(root);
                                printf("\n");
                                break;
                        case 4:
                                exit(0);
                        default:
                                printf("Invalid choice! Please enter again.\n");
                }
```

```
    }
    return 0;
}
```

```
1. Insert
2. Delete
3. Display PreOrder
4. Exit
Enter your choice: 1
Enter the key to insert: 9

1. Insert
2. Delete
3. Display PreOrder
4. Exit
Enter your choice: 1
Enter the key to insert: 0

1. Insert
2. Delete
3. Display PreOrder
4. Exit
Enter your choice: 3
PreOrder traversal of the AVL tree is: 4 2 0 9

1. Insert
2. Delete
3. Display PreOrder
4. Exit
Enter your choice: 2
Enter the key to delete: 0

1. Insert
2. Delete
3. Display PreOrder
4. Exit
Enter your choice: 3
PreOrder traversal of the AVL tree is: 4 2 9
```




**Lab program 5:**

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

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
typedef struct TreeNode {
    int *keys;
    struct TreeNode **child;
    int n;
    bool leaf;
} TreeNode;

typedef struct Tree {
    TreeNode *root;
```

```
} Tree;

TreeNode* createTreeNode(bool leaf);
void traverse(TreeNode* node);
int findKey(TreeNode* node, int k);
void insertNonFull(TreeNode* node, int k);
void splitChild(TreeNode* parent, int i, TreeNode* y);
void removeKey(TreeNode* node, int k);
void removeFromLeaf(TreeNode* node, int idx);
void removeFromNonLeaf(TreeNode* node, int idx);
int getPred(TreeNode* node, int idx);
int getSucc(TreeNode* node, int idx);
void fill(TreeNode* node, int idx);
void borrowFromNext(TreeNode* node, int idx);
void borrowFromPrev(TreeNode* node, int idx);
void merge(TreeNode* node, int idx);
void traverseTree(Tree* tree);
void insert(Tree* tree, int k);
void removeTree(Tree* tree, int k);
TreeNode* createTreeNode(bool leaf) {
    TreeNode* node = (TreeNode*)malloc(sizeof(TreeNode));
    node->leaf = leaf;
    node->keys = (int*)malloc(3 * sizeof(int));
    node->child = (TreeNode**)malloc(4 * sizeof(TreeNode*));
    node->n = 0;
    return node;
}
void traverse(TreeNode* node) {
    if (node == NULL) return;
```

```
    int i;
    for (i = 0; i < node->n; i++) {
        if (node->leaf == false)
            traverse(node->child[i]);
        printf(" %d", node->keys[i]);
    }
    if (node->leaf == false)
        traverse(node->child[i]);
}
int findKey(TreeNode* node, int k) {
    int idx = 0;
    while (idx < node->n && node->keys[idx] < k)
        ++idx;
    return idx;
}
void insert(Tree* tree, int k) {
    if (tree->root == NULL) {
        tree->root = createTreeNode(true);
        tree->root->keys[0] = k;
        tree->root->n = 1;
    } else {
        if (tree->root->n == 3) {
            TreeNode* s = createTreeNode(false);
            s->child[0] = tree->root;
            splitChild(s, 0, tree->root);
            int i = 0;
            if (s->keys[0] < k)
                i++;
            insertNonFull(s->child[i], k);
```

```
            tree->root = s;
        } else {
            insertNonFull(tree->root, k);
        }
    }
}
void insertNonFull(TreeNode* node, int k) {
    int i = node->n - 1;
    if (node->leaf == true) {
        while (i >= 0 && node->keys[i] > k) {
            node->keys[i + 1] = node->keys[i];
            i--;
        }
        node->keys[i + 1] = k;
        node->n = node->n + 1;
    } else {
        while (i >= 0 && node->keys[i] > k)
            i--;
        if (node->child[i + 1]->n == 3) {
            splitChild(node, i + 1, node->child[i + 1]);
            if (node->keys[i + 1] < k)
                i++;
        }
        insertNonFull(node->child[i + 1], k);
    }
}
void splitChild(TreeNode* parent, int i, TreeNode* y) {
    TreeNode* z = createTreeNode(y->leaf);
    z->n = 1;
```

```
    z->keys[0] = y->keys[2];
    if (y->leaf == false) {
        for (int j = 0; j < 2; j++)
            z->child[j] = y->child[j + 2];
    }
    y->n = 1;
    for (int j = parent->n; j >= i + 1; j--)
        parent->child[j + 1] = parent->child[j];
    parent->child[i + 1] = z;
    for (int j = parent->n - 1; j >= i; j--)
        parent->keys[j + 1] = parent->keys[j];
    parent->keys[i] = y->keys[1];
    parent->n = parent->n + 1;
}
void traverseTree(Tree* tree) {
    if (tree->root != NULL)
        traverse(tree->root);
    printf("\n");
}
void removeTree(Tree* tree, int k) {
    if (!tree->root) {
        printf("The tree is empty\n");
        return;
    }
    removeKey(tree->root, k);
    if (tree->root->n == 0) {
        TreeNode* tmp = tree->root;
        if (tree->root->leaf)
            tree->root = NULL;
```

```
        else
            tree->root = tree->root->child[0];
        free(tmp);
    }
}
void removeKey(TreeNode* node, int k) {
    int idx = findKey(node, k);
    if (idx < node->n && node->keys[idx] == k) {
        if (node->leaf)
            removeFromLeaf(node, idx);
        else
            removeFromNonLeaf(node, idx);
    } else {
        if (node->leaf) {
            printf("The key %d doesn't exist\n", k);
            return;
        }
        bool flag = ((idx == node->n) ? true : false);
        if (node->child[idx]->n < 2)
            fill(node, idx);
        if (flag && idx > node->n)
            removeKey(node->child[idx - 1], k);
        else
            removeKey(node->child[idx], k);
    }
}
void removeFromLeaf(TreeNode* node, int idx) {
    for (int i = idx + 1; i < node->n; ++i)
        node->keys[i - 1] = node->keys[i];
```

```
    node->n--;
}
void removeFromNonLeaf(TreeNode* node, int idx) {
    int k = node->keys[idx];
    if (node->child[idx]->n >= 2) {
        int pred = getPred(node, idx);
        node->keys[idx] = pred;
        removeKey(node->child[idx], pred);
    } else if (node->child[idx + 1]->n >= 2) {
        int succ = getSucc(node, idx);
        node->keys[idx] = succ;
        removeKey(node->child[idx + 1], succ);
    } else {
        merge(node, idx);
        removeKey(node->child[idx], k);
    }
}
int getPred(TreeNode* node, int idx) {
    TreeNode* curr = node->child[idx];
    while (!curr->leaf)
        curr = curr->child[curr->n];
    return curr->keys[curr->n - 1];
}
int getSucc(TreeNode* node, int idx) {
    TreeNode* curr = node->child[idx + 1];
    while (!curr->leaf)
        curr = curr->child[0];
    return curr->keys[0];
}
```

```
void fill(TreeNode* node, int idx) {
    if (idx != 0 && node->child[idx - 1]->n >= 2)
        borrowFromPrev(node, idx);
    else if (idx != node->n && node->child[idx + 1]->n >= 2)
        borrowFromNext(node, idx);
    else {
        if (idx != node->n)
            merge(node, idx);
        else
            merge(node, idx - 1);
    }
}

void borrowFromPrev(TreeNode* node, int idx) {
    TreeNode* c = node->child[idx];
    TreeNode* sibling = node->child[idx - 1];
    for (int i = c->n - 1; i >= 0; --i)
        c->keys[i + 1] = c->keys[i];
    if (!c->leaf) {
        for (int i = c->n; i >= 0; --i)
            c->child[i + 1] = c->child[i];
    }
    c->keys[0] = node->keys[idx - 1];
    if (!c->leaf)
        c->child[0] = sibling->child[sibling->n];
    node->keys[idx - 1] = sibling->keys[sibling->n - 1];
    c->n += 1;
    sibling->n -= 1;
}
```

```
void borrowFromNext(TreeNode* node, int idx) {
    TreeNode* c = node->child[idx];
    TreeNode* sibling = node->child[idx + 1];
    c->keys[(c->n)] = node->keys[idx];
    if (!(c->leaf))
        c->child[(c->n) + 1] = sibling->child[0];
    node->keys[idx] = sibling->keys[0];
    for (int i = 1; i < sibling->n; ++i)
        sibling->keys[i - 1] = sibling->keys[i];
    if (!sibling->leaf) {
        for (int i = 1; i <= sibling->n; ++i)
            sibling->child[i - 1] = sibling->child[i];
    }
    c->n += 1;
    sibling->n -= 1;
}

void merge(TreeNode* node, int idx) {
    TreeNode* c = node->child[idx];
    TreeNode* sibling = node->child[idx + 1];
    c->keys[1] = node->keys[idx];
    for (int i = 0; i < sibling->n; ++i)
        c->keys[i + 2] = sibling->keys[i];
    if (!c->leaf) {
        for (int i = 0; i <= sibling->n; ++i)
            c->child[i + 2] = sibling->child[i];
    }
    for (int i = idx + 1; i < node->n; ++i)
```

```
        node->keys[i - 1] = node->keys[i];
    for (int i = idx + 2; i <= node->n; ++i)
        node->child[i - 1] = node->child[i];
    c->n += sibling->n + 1;
    node->n--;
    free(sibling);
}
int main() {
    Tree t;
    t.root = NULL;
    while (1) {
        printf("Choose:\t1.Insert\t2.Display\t3.Delete\t4.Exit\t");
        int key;
        scanf("%d", &key);
        switch (key) {
            case 1: {
                int ele;
                printf("Insert: ");
                scanf("%d", &ele);
                insert(&t, ele);
                printf("Traversal of tree constructed is");
                traverseTree(&t);
                break;
            }
            case 2:
                printf("Traversal of tree constructed is");
                traverseTree(&t);
                break;
            case 3: {
```

```
            printf("Enter deletion element: ");
            int k;
            scanf("%d", &k);
            removeTree(&t, k);
            printf("Traversal of tree constructed is");
            traverseTree(&t);
            break;
        }
        case 4:
            exit(0);
            break;
      }
   }
   return 0;
}
```

```
Choose: 1.Insert        2.Display       3.Delete        4.Exit  1
Insert: 35
Traversal of tree constructed is 35
Choose: 1.Insert        2.Display       3.Delete        4.Exit  1
Insert: 6
Traversal of tree constructed is 6 35
Choose: 1.Insert        2.Display       3.Delete        4.Exit  1
Insert: 9
Traversal of tree constructed is 6 9 35
Choose: 1.Insert        2.Display       3.Delete        4.Exit  1
Choose: 1.Insert        2.Display       3.Delete        4.Exit  1
Insert: 42
Traversal of tree constructed is 6 9 35 42
Choose: 1.Insert        2.Display       3.Delete        4.Exit  2
Traversal of tree constructed is 6 9 35 42
Choose: 1.Insert        2.Display       3.Delete        4.Exit  3
Enter deletion element: 9
Traversal of tree constructed is 6 35 42
```

```
Choose: 1.Insert        2.Display       3.Delete        4.Exit  1
Insert: 34
Traversal of tree constructed is 34
Choose: 1.Insert        2.Display       3.Delete        4.Exit  1
Insert: 87
Traversal of tree constructed is 34 87
Choose: 1.Insert        2.Display       3.Delete        4.Exit  1
Insert: 68
Traversal of tree constructed is 34 68 87
Choose: 1.Insert        2.Display       3.Delete        4.Exit  2
Traversal of tree constructed is 34 68 87
Choose: 1.Insert        2.Display       3.Delete        4.Exit  3
Enter deletion element: 68
Traversal of tree constructed is 34 87
```

**Lab 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>
typedef struct Node {
    int data;
    struct Node *parent;
    struct Node *left;
    struct Node *right;
    int color;
} Node;
typedef Node *NodePtr;
typedef struct RedBlackTree {
    NodePtr root;
    NodePtr TNULL;
} RedBlackTree;
NodePtr createNode(int data, NodePtr TNULL) {
    NodePtr node = (NodePtr)malloc(sizeof(Node));
    node->data = data;
```

```
    node->parent = NULL;
    node->left = TNULL;
    node->right = TNULL;
    node->color = 1;
    return node;
}
RedBlackTree* createTree() {
    RedBlackTree* tree = (RedBlackTree*)malloc(sizeof(RedBlackTree));
    tree->TNULL = createNode(0, NULL);
    tree->TNULL->color = 0;
    tree->TNULL->left = NULL;
    tree->TNULL->right = NULL;
    tree->root = tree->TNULL;
    return tree;
}
void preOrderHelper(NodePtr node, NodePtr TNULL) {
    if (node != TNULL) {
        printf("%d ", node->data);
        preOrderHelper(node->left, TNULL);
        preOrderHelper(node->right, TNULL);
    }
}
void leftRotate(RedBlackTree* tree, NodePtr x) {
    NodePtr y = x->right;
    x->right = y->left;
    if (y->left != tree->TNULL) {
        y->left->parent = x;
    }
    y->parent = x->parent;
```

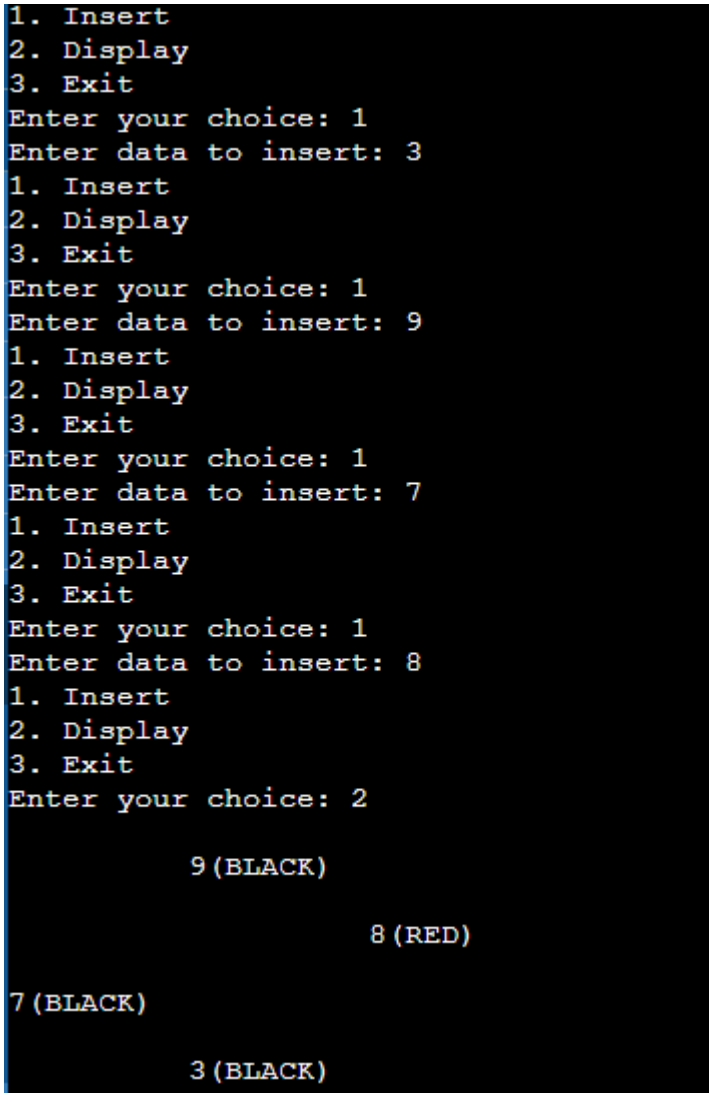
```
    if (x->parent == NULL) {
        tree->root = y;
    } else if (x == x->parent->left) {
        x->parent->left = y;
    } else {
        x->parent->right = y;
    }
    y->left = x;
    x->parent = y;
}
void rightRotate(RedBlackTree* tree, NodePtr x) {
    NodePtr y = x->left;
    x->left = y->right;
    if (y->right != tree->TNULL) {
        y->right->parent = x;
    }
    y->parent = x->parent;
    if (x->parent == NULL) {
        tree->root = y;
    } else if (x == x->parent->right) {
        x->parent->right = y;
    } else {
        x->parent->left = y;
    }
    y->right = x;
    x->parent = y;
}
void insertFix(RedBlackTree* tree, NodePtr k) {
    NodePtr u;
```

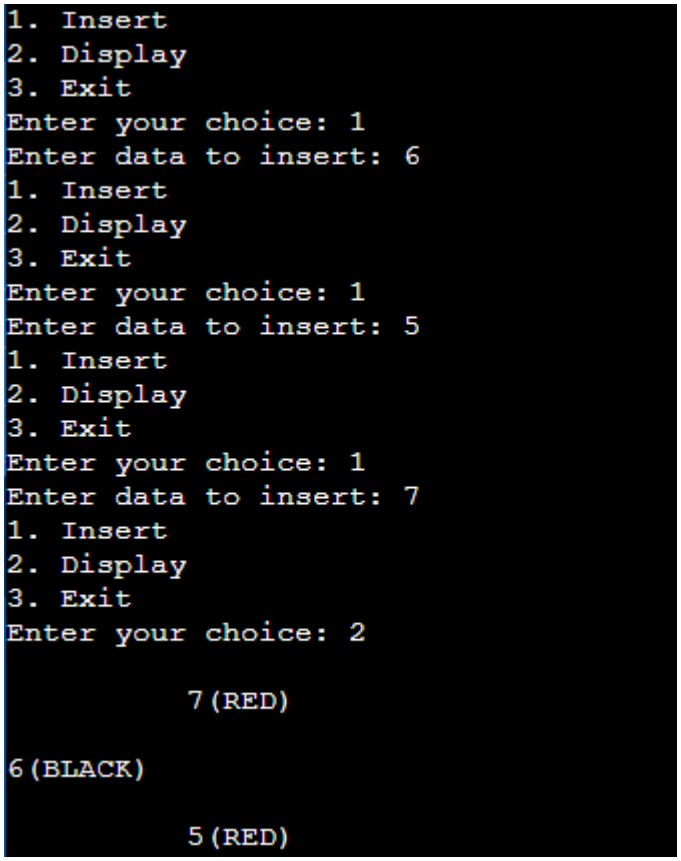
```
while (k->parent->color == 1) {
    if (k->parent == k->parent->parent->right) {
        u = k->parent->parent->left;
        if (u->color == 1) {
            u->color = 0;
            k->parent->color = 0;
            k->parent->parent->color = 1;
            k = k->parent->parent;
        } else {
            if (k == k->parent->left) {
                k = k->parent;
                rightRotate(tree, k);
            }
            k->parent->color = 0;
            k->parent->parent->color = 1;
            leftRotate(tree, k->parent->parent);
        }
    } else {
        u = k->parent->parent->right;
        if (u->color == 1) {
            u->color = 0;
            k->parent->color = 0;
            k->parent->parent->color = 1;
            k = k->parent->parent;
        } else {
            if (k == k->parent->right) {
                k = k->parent;
                leftRotate(tree, k);
            }
```

```
            k->parent->color = 0;
            k->parent->parent->color = 1;
            rightRotate(tree, k->parent->parent);
        }
    }
    if (k == tree->root) {
        break;
    }
  }
  tree->root->color = 0;
}
void insert(RedBlackTree* tree, int key) {
  NodePtr node = createNode(key, tree->TNULL);
  NodePtr y = NULL;
  NodePtr x = tree->root;

  while (x != tree->TNULL) {
    y = x;
    if (node->data < x->data) {
      x = x->left;
    } else {
      x = x->right;
    }
  }
  node->parent = y;
  if (y == NULL) {
    tree->root = node;
  } else if (node->data < y->data) {
    y->left = node;
```

```
    } else {
        y->right = node;
    }
    if (node->parent == NULL) {
        node->color = 0;
        return;
    }
    if (node->parent->parent == NULL) {
        return;
    }

    insertFix(tree, node);
}
void printHelper(NodePtr root, NodePtr TNULL, int space) {
    if (root == TNULL)
        return;
    space += 10;
    printHelper(root->right, TNULL, space);
    printf("\n");
    for (int i = 10; i < space; i++)
        printf(" ");
    printf("%d(%s)\n", root->data, root->color ? "RED" : "BLACK");

    printHelper(root->left, TNULL, space);
}
int main() {
    RedBlackTree* bst = createTree();
    int choice, data;
```

```
    while (1) {
        printf("1. Insert\n2. Display\n3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                insert(bst, data);
                break;
            case 2:
                printHelper(bst->root, bst->TNULL, 0);
                break;
            case 3:
                exit(0);
                break;
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
    return 0;
}
```


1. Insert
2. Display
3. Exit
Enter your choice: 1
Enter data to insert: 3
1. Insert
2. Display
3. Exit
Enter your choice: 1
Enter data to insert: 9
1. Insert
2. Display
3. Exit
Enter your choice: 1
Enter data to insert: 7
1. Insert
2. Display
3. Exit
Enter your choice: 1
Enter data to insert: 8
1. Insert
2. Display
3. Exit
Enter your choice: 2
9(BLACK)
8(RED)
7(BLACK)
3(BLACK)




**Lab program 7:**

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

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
typedef struct BTreeNode {
    int *keys;
    int t;
    struct BTreeNode **C;
    int n;
    bool leaf;
} BTreeNode;
typedef struct BTree {
    BTreeNode *root;
```

```
    int t;
} BTree;
BTreeNode* createBTreeNode(int t, bool leaf) {
    BTreeNode* newNode = (BTreeNode*)malloc(sizeof(BTreeNode));
    newNode->t = t;
    newNode->leaf = leaf;
    newNode->keys = (int*)malloc(sizeof(int) * (2 * t - 1));
    newNode->C = (BTreeNode**)malloc(sizeof(BTreeNode*) * (2 * t));
    newNode->n = 0;
    return newNode;
}
void traverse(BTreeNode* node) {
    int i;
    for (i = 0; i < node->n; i++) {
        if (!node->leaf) {
            traverse(node->C[i]);
        }
        printf(" %d", node->keys[i]);
    }
    if (!node->leaf) {
        traverse(node->C[i]);
    }
}
void insertNonFull(BTreeNode* node, int k) {
    int i = node->n - 1;
    if (node->leaf) {
        while (i >= 0 && node->keys[i] > k) {
            node->keys[i + 1] = node->keys[i];
            i--;
```

```
        }
        node->keys[i + 1] = k;
        node->n++;
    } else {
        while (i >= 0 && node->keys[i] > k) {
            i--;
        }
        if (node->C[i + 1]->n == 2 * node->t - 1) {
            splitChild(node, i + 1, node->C[i + 1]);
            if (node->keys[i + 1] < k) {
                i++;
            }
        }
        insertNonFull(node->C[i + 1], k);
    }
}
void splitChild(BTreeNode* node, int i, BTreeNode* y) {
    BTreeNode* z = createBTreeNode(y->t, y->leaf);
    z->n = y->t - 1;

    for (int j = 0; j < y->t - 1; j++) {
        z->keys[j] = y->keys[j + y->t];
    }
    if (!y->leaf) {
        for (int j = 0; j < y->t; j++) {
            z->C[j] = y->C[j + y->t];
        }
    }
    y->n = y->t - 1;
```

```
    for (int j = node->n; j >= i + 1; j--) {
        node->C[j + 1] = node->C[j];
    }
    node->C[i + 1] = z;
    for (int j = node->n - 1; j >= i; j--) {
        node->keys[j + 1] = node->keys[j];
    }
    node->keys[i] = y->keys[y->t - 1];
    node->n++;
}
void insert(BTree* tree, int k) {
    BTreeNode* root = tree->root;
    if (root->n == 2 * tree->t - 1) {
        BTreeNode* s = createBTreeNode(tree->t, false);
        s->C[0] = root;
        tree->root = s;
        splitChild(s, 0, root);
        insertNonFull(s, k);
    } else {
        insertNonFull(root, k);
    }
}
BTree* createBTree(int t) {
    BTree* tree = (BTree*)malloc(sizeof(BTree));
    tree->t = t;
    tree->root = createBTreeNode(t, true);
    return tree;
}
int main() {
```

```
int degree, key;
BTree* t;
printf("Enter degree: ");
scanf("%d", &degree);
t = createBTree(degree);
while (1) {
    printf("\nChoose:\n1.Insert\n2.Display\n3.Exit\n");
    scanf("%d", &key);
    switch (key) {
        case 1: {
            int num;
            printf("Insert key: ");
            scanf("%d", &num);
            insert(t, num);
            break;
        }
        case 2: {
            printf("Traversal of tree constructed is:");
            traverse(t->root);
            printf("\n");
            break;
        }
        case 3: {
            exit(0);
        }
        default: {
            printf("Invalid choice\n");
            break;
        }
```

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

```
Insert key: 15

Choose:
1.Insert
2.Display
3.Exit
1
Insert key: 12

Choose:
1.Insert
2.Display
3.Exit
1
Insert key: 6

Choose:
1.Insert
2.Display
3.Exit
2
Traversal of tree constructed is: 6 9 12 15
```

```
Choose:
1.Insert
2.Display
3.Exit
1
Insert key: 4

Choose:
1.Insert
2.Display
3.Exit
1
Insert key: 8

Choose:
1.Insert
2.Display
3.Exit
1
Insert key: 2

Choose:
1.Insert
2.Display
3.Exit
2
Traversal of tree constructed is: 2 4 5 6 7 8 9 9
```

**Lab program 8:**

Write a program to implement functions of Dictionary using Hashing.

```
#include <stdio.h>
#define SIZE 7
#define EMPTY -9999
struct Hashtable {
    int keys[SIZE];
    int values[SIZE];
    int count;
};

void initialArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = EMPTY;
    }
}

int hashFunction(int k, int size) {
    return k % size;
}

void add(struct Hashtable *ht, int k, int v) {
    int x = hashFunction(k, SIZE);

    if (ht->keys[x] == EMPTY) {
        ht->keys[x] = k;
        ht->values[x] = v;
        ht->count++;
```

```
    } else {
        int i = x + 1;
        while (ht->keys[i] != EMPTY) {
            i += 1;
        }
        ht->keys[i] = k;
        ht->values[i] = v;
        ht->count++;
    }
}

void search(struct Hashtable ht, int k) {
    int x = hashFunction(k, SIZE);
    if (ht.keys[x] != EMPTY) {
        printf("The value for key %d is: %d\n", k, ht.values[x]);
    } else {
        printf("No value Found for key %d\n", k);
    }
}

void print(struct Hashtable ht) {
    printf("Key Value\n");
    for (int i = 0; i < SIZE; i++) {
        if (ht.keys[i] != EMPTY) {
            printf("%d : %d\n", ht.keys[i], ht.values[i]);
        }
    }
}
```

```
void deleteElement(struct Hashtable *ht, int k) {
    int x = hashFunction(k, SIZE);
    if (ht->keys[x] == EMPTY) {
        printf("No element found for key %d\n", k);
    } else {
        ht->keys[x] = EMPTY;
        ht->values[x] = EMPTY;
        ht->count--;
        printf("Element with key %d deleted successfully\n", k);
    }
}

int main() {
    struct Hashtable ht;
    ht.count = 0;
    initialArray(ht.keys, SIZE);
    initialArray(ht.values, SIZE);

    int key, value;

    while (1) {
        printf("\nChoose:\n1. Add key-value pair\n2. Search for a key\n3. Delete an element\n4. Print the hashtable\n5. Exit\n");
        scanf("%d", &key);

        switch (key) {
            case 1: {
                printf("Enter key and value to add: ");
                scanf("%d %d", &key, &value);
                add(&ht, key, value);
```

```
            break;
        }
        case 2: {
            printf("Enter the key to search: ");
            scanf("%d", &key);
            search(ht, key);
            break;
        }
        case 3: {
            printf("Enter the key to delete: ");
            scanf("%d", &key);
            deleteElement(&ht, key);
            break;
        }
        case 4: {
            print(ht);
            break;
        }
        case 5: {
            printf("Exiting program.\n");
            return 0;
        }
        default: {
            printf("Invalid choice\n");
            break;
        }
    }
}
return 0;
```

}

```
Choose:
1. Add key-value pair
2. Search for a key
3. Delete an element
4. Print the hashtable
5. Exit
1
Enter key and value to add: 2
4

Choose:
1. Add key-value pair
2. Search for a key
3. Delete an element
4. Print the hashtable
5. Exit
1
Enter key and value to add: 6
7

Choose:
1. Add key-value pair
2. Search for a key
3. Delete an element
4. Print the hashtable
5. Exit
2
Enter the key to search: 6
The value for key 6 is: 7
```

```
3. Delete an element
4. Print the hashtable
5. Exit
1
Enter key and value to add: 3
4

Choose:
1. Add key-value pair
2. Search for a key
3. Delete an element
4. Print the hashtable
5. Exit
1
Enter key and value to add: 5
6

Choose:
1. Add key-value pair
2. Search for a key
3. Delete an element
4. Print the hashtable
5. Exit
4
Key Value
3 : 4
5 : 6
```

**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 <stdlib.h>

struct Node {

    int data, degree;

    struct Node *child, *sibling, *parent;
```

```
};

struct Node* newNode(int key) {
    struct Node *temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = key;
    temp->degree = 0;
    temp->child = temp->parent = temp->sibling = NULL;
    return temp;
}

struct Node* mergeBinomialTrees(struct Node *b1, struct Node *b2) {
    if (b1->data > b2->data) {
        struct Node *temp = b1;
        b1 = b2;
        b2 = temp;
    }
    b2->parent = b1;
    b2->sibling = b1->child;
    b1->child = b2;
    b1->degree++;
    return b1;
}

struct Node* unionBinomialHeap(struct Node *h1, struct Node *h2) {
    struct Node *head = NULL;
    struct Node *tail = NULL;
    while (h1 != NULL || h2 != NULL) {
        if (h1 == NULL) {
            if (tail == NULL) {
```

```
            head = h2;
        } else {
            tail->sibling = h2;
        }
        break;
    } else if (h2 == NULL) {
        if (tail == NULL) {
            head = h1;
        } else {
            tail->sibling = h1;
        }
        break;
    }
    if (h1->degree <= h2->degree) {
        if (tail == NULL) {
            head = h1;
        } else {
            tail->sibling = h1;
        }
        tail = h1;
        h1 = h1->sibling;
    } else {
        if (tail == NULL) {
            head = h2;
        } else {
            tail->sibling = h2;
        }
        tail = h2;
        h2 = h2->sibling;
```

```
        }
    }
    return head;
}

struct Node* adjustHeap(struct Node *h) {
    if (h == NULL || h->sibling == NULL) {
        return h;
    }
    struct Node *prev = NULL;
    struct Node *curr = h;
    struct Node *next = h->sibling;
    while (next != NULL) {
        if ((curr->degree != next->degree) || (next->sibling != NULL && next->sibling->degree
== curr->degree)) {
            prev = curr;
            curr = next;
        } else {
            if (curr->data <= next->data) {
                curr->sibling = next->sibling;
                mergeBinomialTrees(curr, next);
            } else {
                if (prev == NULL) {
                    h = next;
                } else {
                    prev->sibling = next;
                }
                mergeBinomialTrees(next, curr);
                curr = next;
            }
```

```
        }
        next = curr->sibling;
    }
    return h;
}

struct Node* insertTreeInHeap(struct Node *h, struct Node *tree) {
    h = unionBinomialHeap(h, tree);
    return adjustHeap(h);
}

struct Node* removeMinFromTreeReturnHeap(struct Node *tree) {
    struct Node *temp = tree->child;
    struct Node *lo = NULL;
    while (temp != NULL) {
        lo = temp;
        temp = temp->sibling;
        lo->sibling = NULL;
    }
    return lo;
}

struct Node* extractMin(struct Node *h) {
    if (h == NULL) {
        return NULL;
    }
    struct Node *minNode = h;
    struct Node *prevMin = NULL;
    struct Node *curr = h;
```

```
    struct Node *prev = NULL;
    while (curr->sibling != NULL) {
        if (curr->sibling->data < minNode->data) {
            minNode = curr->sibling;
            prevMin = prev;
        }
        prev = curr;
        curr = curr->sibling;
    }
    if (prevMin == NULL) {
        h = minNode->sibling;
    } else {
        prevMin->sibling = minNode->sibling;
    }
    struct Node *newHeap = removeMinFromTreeReturnHeap(minNode);
    return unionBinomialHeap(h, newHeap);
}

void printTree(struct Node *h) {
    while (h != NULL) {
        printf("%d ", h->data);
        printTree(h->child);
        h = h->sibling;
    }
}

void printHeap(struct Node *h) {
    while (h != NULL) {
        printTree(h);
```

```
        h = h->sibling;
    }
}

int main() {
    int n, key;
    struct Node *heap = NULL;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        printf("Enter element %d: ", i + 1);
        scanf("%d", &key);
        struct Node *temp = newNode(key);
        heap = insertTreeInHeap(heap, temp);
    }
    printf("Heap elements after insertion:\n");
    printHeap(heap);
    struct Node *minNode = heap;
    while (minNode != NULL) {
        if (minNode->data < heap->data) {
            heap = minNode;
        }
        minNode = minNode->sibling;
    }
    printf("\nMinimum element of heap: %d\n", heap->data);
    heap = extractMin(heap);
    printf("Heap after deletion of minimum element:\n");
```

```
    printHeap(heap);
    return 0;
}
```

```
Enter the number of elements: 3
Enter element 1: 2
Enter element 2: 6
Enter element 3: 1
Heap elements after insertion:
1 2 6 2 6
Minimum element of heap: 1
Heap after deletion of minimum element:
2 6
```

```
Enter the number of elements: 6
Enter element 1: 12
Enter element 2: 14
Enter element 3: 23
Enter element 4: 5
Enter element 5: 623
Enter element 6: 12
Heap elements after insertion:
12 623 5 12 14 23 5 12 14 23
Minimum element of heap: 5
Heap after deletion of minimum element:
23
```

**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 <stdlib.h>
#include <limits.h>

struct Node {
    int val, degree;
    struct Node *parent, *child, *sibling;
```

```
};

struct Node *root = NULL;

struct Node *createNode(int n) {
    struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
    new_node->val = n;
    new_node->parent = NULL;
    new_node->sibling = NULL;
    new_node->child = NULL;
    new_node->degree = 0;
    return new_node;
}

void binomialLink(struct Node *h1, struct Node *h2) {
    h1->parent = h2;
    h1->sibling = h2->child;
    h2->child = h1;
    h2->degree = h2->degree + 1;
}

struct Node *mergeBHeaps(struct Node *h1, struct Node *h2) {
    if (h1 == NULL) return h2;
    if (h2 == NULL) return h1;

    struct 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) {
            struct Node *sib = h1->sibling;
            h1->sibling = h2;
            h1 = sib;
        } else {
            struct Node *sib = h2->sibling;
            h2->sibling = h1;
            h2 = sib;
        }
    }

    return res;
}

struct Node *unionBHeaps(struct Node *h1, struct Node *h2) {
    if (h1 == NULL && h2 == NULL) return NULL;
    struct Node *res = mergeBHeaps(h1, h2);

    struct 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(struct Node *h) {
    while (h) {
        printf("%d ", h->val);
        display(h->child);
        h = h->sibling;
    }
}

void revertList(struct Node *h) {
    if (h->sibling != NULL) {
        revertList(h->sibling);
        (h->sibling)->sibling = h;
```

```
    } else root = h;
}

struct Node *extractMinBHeap(struct Node *h) {
    if (h == NULL) return NULL;

    struct Node *min_node_prev = NULL;
    struct Node *min_node = h;
    int min = h->val;
    struct 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);
}

struct Node *findNode(struct Node *h, int val) {
    if (h == NULL) return NULL;
    if (h->val == val) return h;

    struct Node *res = findNode(h->child, val);
    if (res != NULL) return res;

    return findNode(h->sibling, val);
}

void decreaseKeyBHeap(struct Node *H, int old_val, int new_val) {
    struct Node *node = findNode(H, old_val);
    if (node == NULL) return;
    node->val = new_val;
    struct Node *parent = node->parent;
    while (parent != NULL && node->val < parent->val) {
        int temp = node->val;
        node->val = parent->val;
        parent->val = temp;
        node = parent;
        parent = parent->parent;
    }
}
struct Node *binomialHeapDelete(struct Node *h, int val) {
    if (h == NULL) return NULL;
    decreaseKeyBHeap(h, val, INT_MIN);
```

```
    struct Node *minNode = extractMinBHeap(h);
    if (minNode->val != INT_MIN) {
        decreaseKeyBHeap(h, INT_MIN, val);
    }
    return h;
}
int main() {
    int n;
    printf("Enter the number of elements to insert: ");
    scanf("%d", &n);
    for (int i = 0; i < n; ++i) {
        int k;
        printf("Enter element %d: ", i + 1);
        scanf("%d", &k);
        binomialHeapInsert(k);
    }
    printf("The heap is: ");
    display(root);
    printf("\n");
    int m;
    printf("Enter the element to delete: ");
    scanf("%d", &m);
    root = binomialHeapDelete(root, m);
    printf("After deleting %d, the heap is: ", m);
    display(root);
    return 0;
}
```

```
Enter the number of elements to insert: 2
Enter element 1: 3
Enter element 2: 4
The heap is: 3 4
Enter the element to delete: 1
After deleting 1, the heap is: 3 4
```

```
Enter the number of elements to insert: 4
Enter element 1: 12
Enter element 2: 767
Enter element 3: 123
Enter element 4: 788
The heap is: 12 123 788 767
Enter the element to delete: 123
After deleting 123, the heap is: 123 12 788
```