

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT On

ADVANCED DATA STRUCTURES (22CS5PEADS)

Submitted by

AMULYA S A (1BM21CS020)

**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 **AMULYA S A(1BM21CS020)**, 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

Sl. No.	Experiment Title	Page No.
1	<p>Write a program to implement the following list:</p> <p>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.</p>	5
2	Write a program to perform insertion, deletion and searching operations on a skip list.	10
3	<p>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</p> <p>{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}</p> <p>A cell in the 2D matrix can be connected to 8 neighbours. Use disjoint sets to implement the above scenario.</p>	13
4	Write a program to perform insertion and deletion operations on AVL trees.	23
5	Write a program to perform insertion and deletion operations on 2-3 trees.	35
6	Write a program to implement insertion operation on a red black tree. During insertion, appropriately show how recoloring or rotation operation is used.	39
7	Write a program to implement insertion operation on a B-tree.	42
8	Write a program to implement functions of Dictionary using Hashing.	46
9	<p>Write a program to implement the following functions on a Binomial heap:</p> <p>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.</p> <p>2. getMin(H): A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key.</p>	51

	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.	
10	<p>Write a program to implement the following functions on a Binomial heap:</p> <ol style="list-style-type: none"> 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 decreased key with its 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 a smaller key or we hit the root node. 	54

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:

1. 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.

CODE:

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

```
typedef struct DisjointUnionSets {
```

```
    int rank[25];
```

```
    int parent[25];
```

```
} DisjointUnionSets;
```

```
void makeSet(DisjointUnionSets *dus, int n) {
```

```

for (int i = 0; i < n; i++) {
    dus->parent[i] = i;
    dus->rank[i] = 0;
}
}

int find(DisjointUnionSets *dus, int x) {
    if (dus->parent[x] != x) {
        dus->parent[x] = find(dus, dus->parent[x]);
    }
    return dus->parent[x];
}

void Union(DisjointUnionSets *dus, int x, int y) {
    int xRoot = find(dus, x);
    int yRoot = find(dus, y);

    if (xRoot == yRoot) {
        return;
    }

    if (dus->rank[xRoot] < dus->rank[yRoot]) {
        dus->parent[xRoot] = yRoot;
    } else if (dus->rank[yRoot] < dus->rank[xRoot]) {
        dus->parent[yRoot] = xRoot;
    } else {
        dus->parent[yRoot] = xRoot;
    }
}

```

```

        dus->rank[xRoot]++;
    }
}

int countIslands(int a[][5], int n, int m) {
    DisjointUnionSets dus;
    makeSet(&dus, n * m);

    for (int j = 0; j < n; j++) {
        for (int k = 0; k < m; k++) {
            if (a[j][k] == 0) {
                continue;
            }

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

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

            if (k + 1 < m && a[j][k + 1] == 1) {
                Union(&dus, j * (m) + k, (j) * (m) + k + 1);
            }

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

            if (j + 1 < n && k + 1 < m && a[j + 1][k + 1] == 1) {

```

```

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

```

```

int c[25] = {0};
int numberOfIslands = 0;
for (int j = 0; j < n; j++) {
    for (int k = 0; k < m; k++) {
        if (a[j][k] == 1) {
            int x = find(&dus, j * m + k);
            if (c[x] == 0) {
                numberOfIslands++;
                c[x]++;
            } else {
                c[x]++;
            }
        }
    }
}

```



```

        }
    }

    return numberOfIslands;
}

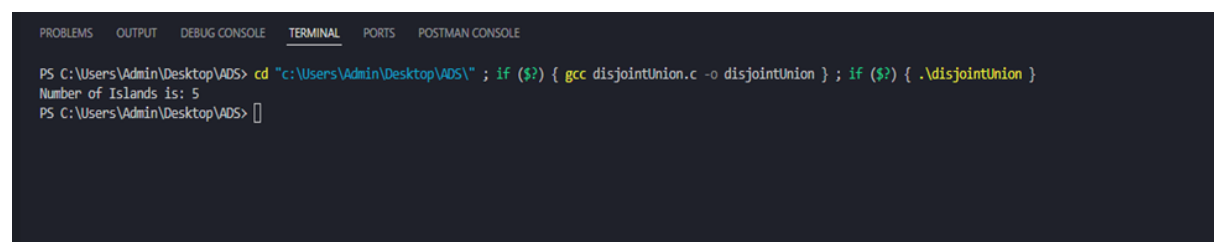
int main(void) {
    int a[5][5] = {
        {1, 1, 0, 0, 0},
        {0, 1, 0, 0, 1},
        {1, 0, 0, 1, 1},
        {0, 0, 0, 0, 0},
        {1, 0, 1, 0, 1}
    };

    printf("Number of Islands is: %d\n", countIslands(a, 5, 5));

    return 0;
}

```

OUTPUT:



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  POSTMAN CONSOLE
PS C:\Users\Admin\Desktop\ADS> cd "c:\Users\Admin\Desktop\ADS\" ; if ($?) { gcc disjointUnion.c -o disjointUnion } ; if ($?) { .\disjointUnion }
Number of Islands is: 5
PS C:\Users\Admin\Desktop\ADS>

```

- . 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.

CODE:

```
#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>


// Node structure
typedef struct Node {
    int data;

    struct Node* xor_ptr; // XOR of previous and next node addresses
} Node;


// Function to perform XOR operation on pointers
Node* xor(Node* a, Node* b) {
    return (Node*)((uintptr_t)(a) ^ (uintptr_t)(b));
}


// Function to insert a node at the beginning of the list
```

```

void insert(Node** head_ref, int data) {

    // Create a new node

    Node* new_node = (Node*)malloc(sizeof(Node));

    new_node->data = data;


    // Set XOR of new node's address and next node's address

    new_node->xor_ptr = *head_ref;


    // If list is not empty, then update the previous node's XOR

    if (*head_ref != NULL) {

        // XOR of new node's address and next node's XOR

        (*head_ref)->xor_ptr = xor(new_node, (*head_ref)->xor_ptr);

    }


    // Change head

    *head_ref = new_node;

}


// Function to print the XOR linked list

void printList(Node* head) {

    Node *prev = NULL, *curr = head, *next;


    printf("XOR Linked List: ");

    while (curr != NULL) {

        printf("%d ", curr->data);

        // Get next node

        next = xor(prev, curr->xor_ptr);
    }
}

```

```

        prev = curr;

        curr = next;

    }

    printf("\n");
}

int main() {

    Node* head = NULL;

    int data, count;

    printf("Enter the number of elements: ");

    scanf("%d", &count);

    printf("Enter %d elements:\n", count);

    for (int i = 0; i < count; i++) {

        scanf("%d", &data);

        insert(&head, data);

    }

    // Print the XOR linked list

    printList(head);

    return 0;

}

```

OUTPUT:

```

PS C:\Users\Admin\Desktop\ADS> cd "c:\Users\Admin\Desktop\ADS\" ; if ($?) { gcc xOrLinkedList.c -o xOrLinkedList } ; if ($?) { .\xOrLinkedList }
Enter the number of elements: 4
Enter 4 elements:
18
9
19
22
XOR Linked List: 22 19 9 18
PS C:\Users\Admin\Desktop\ADS> 

```

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

CODE:

```
#include <iostream>
```

```
#include <cstdlib>
```

```
#include <ctime>
```

```
#include <cstring>
```

```
const int MAX_LEVEL = 3;
```

```
// Node structure for the skip list
```

```
struct Node {
```

```
    int value;
```

```
    Node** forward;
```

```
    Node(int level, int &value) {
```

```
        forward = new Node*[level + 1];
```

```
        std::memset(forward, 0, sizeof(Node*) * (level + 1));
```

```
        this->value = value;
```

```
    }
```

```

        ~Node() {
            delete[] forward;
        }
};

// SkipList class
class SkipList {
private:
    Node* header;
    int level;

public:
    SkipList() {
        header = new Node(MAX_LEVEL, *(new int(-1)));
        level = 0;
    }

    ~SkipList() {
        delete header;
    }

    // Generate random level for a node
    int randomLevel() {
        int lvl = 0;

```

```

while (lvl < MAX_LEVEL && rand() % 2 == 0) {
    lvl++;
}
return lvl;
}

// Insert value into the skip list
void insertElement(int &value) {
    Node* current = header;

    Node* update[MAX_LEVEL + 1];
    std::memset(update, 0, sizeof(Node*) * (MAX_LEVEL + 1));

    for (int i = level; i >= 0; i--) {
        while (current->forward[i] != nullptr && current->forward[i]->value <
value) {
            current = current->forward[i];
        }
        update[i] = current;
    }

    current = current->forward[0];

    if (current == nullptr || current->value != value) {
        int newLevel = randomLevel();

```

```

if (newLevel > level) {
    for (int i = level + 1; i <= newLevel; i++) {
        update[i] = header;
    }
    level = newLevel;
}

Node* newNode = new Node(newLevel, value);

for (int i = 0; i <= newLevel; i++) {
    newNode->forward[i] = update[i]->forward[i];
    update[i]->forward[i] = newNode;
}

std::cout << "Inserted element: " << value << std::endl;
}
}

// Delete element from the skip list
void deleteElement(int &value) {
    Node* current = header;
    Node* update[MAX_LEVEL + 1];
    std::memset(update, 0, sizeof(Node*) * (MAX_LEVEL + 1));

    for (int i = level; i >= 0; i--) {

```



```

        while (current->forward[i] != nullptr && current->forward[i]->value <
value) {

            current = current->forward[i];

        }
        update[i] = current;
    }

    current = current->forward[0];

    if (current != nullptr && current->value == value) {
        for (int i = 0; i <= level; i++) {
            if (update[i]->forward[i] != current)
                break;
            update[i]->forward[i] = current->forward[i];
        }

        delete current;

        while (level > 0 && header->forward[level] == nullptr) {
            level--;
        }

        std::cout << "Deleted element: " << value << std::endl;
    }
}

```

```

// Search for an element in the skip list

bool searchElement(int &value) {

Node* current = header;

for (int i = level; i >= 0; i--) {

while (current->forward[i] != nullptr && current->forward[i]->value <
value) {

current = current->forward[i];

}

}

current = current->forward[0];

return (current != nullptr && current->value == value);

}

// Display the skip list

void displayList() {

std::cout << "Skip List:" << std::endl;

for (int i = 0; i <= level; i++) {

Node* node = header->forward[i];

std::cout << "Level " << i << ": ";

while (node != nullptr) {

std::cout << node->value << " ";

}

}

}

```

```

        node = node->forward[i];
    }
    std::cout << std::endl;
}
}
};

int main() {
    // Initialize random seed
    srand(time(nullptr));

    SkipList skipList;

    int choice;
    int element;
    while (true) {
        std::cout << "\nSkip List Menu:\n";
        std::cout << "1. Insert element\n";
        std::cout << "2. Delete element\n";
        std::cout << "3. Search element\n";
        std::cout << "4. Display list\n";
        std::cout << "5. Exit\n";
        std::cout << "Enter your choice: ";
        std::cin >> choice;
    }
}

```

```

switch (choice) {
case 1:
    std::cout << "Enter element to insert: ";
    std::cin >> element;
    skipList.insertElement(element);
    break;
case 2:
    std::cout << "Enter element to delete: ";
    std::cin >> element;
    skipList.deleteElement(element);
    break;
case 3:
    std::cout << "Enter element to search: ";
    std::cin >> element;
    if (skipList.searchElement(element)) {
        std::cout << "Element " << element << " found in the skip list." <<
std::endl;
    } else {
        std::cout << "Element " << element << " not found in the skip list."
<< std::endl;
    }
    break;
case 4:
    skipList.displayList();
    break;
case 5:

```

```
        std::cout << "Exiting program.\n";  
        return 0;  
    default:  
        std::cout << "Invalid choice. Please try again.\n";  
    }  
}  
  
    return 0;  
}
```

OUTPUT:

```
PS C:\Users\Admin\Desktop\ADS> cd "c:\Users\Admin\Desktop\ADS\" ; if ($?) { g++ skipList.cpp -o skipList } ; if ($?) { .\skipList }

Skip List Menu:
1. Insert element
2. Delete element
3. Search element
4. Display list
5. Exit
Enter your choice: 1
Enter element to insert: 11
Inserted element: 11

Skip List Menu:
1. Insert element
2. Delete element
3. Search element
4. Display list
5. Exit
Enter your choice: 1
Enter element to insert: 22
Inserted element: 22

Skip List Menu:
1. Insert element
2. Delete element
3. Search element
4. Display list
5. Exit
Enter your choice: 1
Enter element to insert: 33
Inserted element: 33

Skip List Menu:
1. Insert element
2. Delete element
3. Search element
4. Display list
5. Exit
Enter your choice: 1
Enter element to insert: 44
Inserted element: 44

Skip List Menu:
1. Insert element
2. Delete element
3. Search element
4. Display list
5. Exit
Enter your choice: 1
Enter element to insert: 55
Inserted element: 55

Skip List Menu:
1. Insert element
2. Delete element
3. Search element
4. Display list
```

```
5. Exit
Enter your choice: 1
Enter element to insert: 44
Inserted element: 44

Skip List Menu:
1. Insert element
2. Delete element
3. Search element
4. Display list
5. Exit
Enter your choice: 1
Enter element to insert: 55
Inserted element: 55

Skip List Menu:
1. Insert element
2. Delete element
3. Search element
4. Display list
5. Exit
Enter your choice: 2
Enter element to delete: 33
Deleted element: 33

Skip List Menu:
1. Insert element
2. Delete element
3. Search element
4. Display list
5. Exit
Enter your choice: 3
Enter element to search: 22
Element 22 found in the skip list.

Skip List Menu:
1. Insert element
2. Delete element
3. Search element
4. Display list
5. Exit
Enter your choice: 4
Skip List:
Level 0: 11 22 44 55
Level 1: 22 44 55
Level 2: 22 55
Level 3: 55

Skip List Menu:
1. Insert element
2. Delete element
3. Search element
4. Display list
5. Exit
Enter your choice: 5
Exiting program.
PS C:\Users\Admin\Desktop\ADS> 
```

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

CODE:

```
#include <iostream>

#include <algorithm>

using namespace std;

// Node structure for AVL tree

struct Node {

    int key;

    Node* left;

    Node* right;

    int height;

};

// Function to get height of a node

int height(Node* node) {

    if (node == nullptr)

        return 0;

    return node->height;

}

// Function to get the balance factor of a node

int getBalance(Node* node) {

    if (node == nullptr)

        return 0;
```



```

        return height(node->left) - height(node->right);
    }

// Function to create a new node
Node* newNode(int key) {
    Node* node = new Node();
    node->key = key;
    node->left = nullptr;
    node->right = nullptr;
    node->height = 1;
    return node;
}

// Function to right rotate subtree rooted with y
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
}

```

```

        // Return new root

        return x;
    }

// Function to left rotate subtree rooted with x
Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    // Perform rotation

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

    // Update heights

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

    // Return new root

    return y;
}

// Function to insert a node into AVL tree
Node* insert(Node* node, int key) {
    if (node == nullptr)

```

```

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 // Equal keys not allowed in AVL
return node;

// Update height of this ancestor node
node->height = 1 + max(height(node->left), height(node->right));

// Get the balance factor to check if this node became unbalanced
int balance = getBalance(node);

// If the node becomes unbalanced, there are 4 cases

// Left Left Case
if (balance > 1 && key < node->left->key)
return rightRotate(node);

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

```

```

// Left Right Case

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

    node->left = leftRotate(node->left);

    return rightRotate(node);

}


// Right Left Case

if (balance < -1 && key < node->right->key) {

    node->right = rightRotate(node->right);

    return leftRotate(node);

}


// If the node is balanced, return the unchanged node pointer

return node;

}


// Function to find the node with minimum key value in a subtree

Node* minValueNode(Node* node) {

    Node* current = node;

    while (current->left != nullptr)

        current = current->left;

    return current;

}


// Function to delete a node with given key from AVL tree

Node* deleteNode(Node* root, int key) {

    // Perform standard BST delete

    if (root == nullptr)

```

```

return root;

if (key < root->key)
    root->left = deleteNode(root->left, key);
else if (key > root->key)
    root->right = deleteNode(root->right, key);
else {
    // Node to be deleted found

    // Node with only one child or no child
    if ((root->left == nullptr) || (root->right == nullptr)) {
        Node* temp = root->left ? root->left : root->right;

        // No child case
        if (temp == nullptr) {
            temp = root;
            root = nullptr;
        } else // One child case
            *root = *temp; // Copy the contents of the non-empty child

        delete temp;
    } else {
        // Node with two children: Get the inorder successor (smallest
        // in the right subtree)
        Node* temp = minValueNode(root->right);

```

```

// Copy the inorder successor's data to this node
root->key = temp->key;

// Delete the inorder successor
root->right = deleteNode(root->right, temp->key);
}
}

// If the tree had only one node then return
if (root == nullptr)
return root;

// Update height of the current node
root->height = 1 + max(height(root->left), height(root->right));

// Get the balance factor to check if this node became unbalanced
int balance = getBalance(root);

// If this node becomes unbalanced, there are 4 cases

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

```

```

// Left Right Case

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

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

    return rightRotate(root);

}


// Right Right Case

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

    return leftRotate(root);


// Right Left Case

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

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

    return leftRotate(root);

}


return root;

}


// Function to print inorder traversal of AVL tree

void inorder(Node* root) {

    if (root != nullptr) {

        inorder(root->left);

        cout << root->key << " ";

        inorder(root->right);

    }

}

```

```

    }

}

// Function to display menu and handle user choice
void menu(Node* &root) {
    int choice;

    do {
        cout << "\nAVL Tree Menu:\n";

        cout << "1. Insert\n";
        cout << "2. Delete\n";
        cout << "3. Display\n";
        cout << "4. Exit\n";

        cout << "Enter your choice: ";

        cin >> choice;

        switch(choice) {
            case 1: {
                int key;

                cout << "Enter key to insert: ";

                cin >> key;

                root = insert(root, key);

                break;
            }

            case 2: {
                int key;

```



```

        cout << "Enter key to delete: ";

        cin >> key;

        root = deleteNode(root, key);

        break;
    }

    case 3: {

        cout << "Inorder traversal of AVL tree: ";

        inorder(root);

        cout << endl;

        break;
    }

    case 4: {

        cout << "Exiting program...\n";

        exit(0);

    }

    default:

        cout << "Invalid choice. Please try again.\n";

    }

    while(choice != 4);
}

// Driver program

int main() {

    Node* root = nullptr;

    menu(root);

```

```
        return 0;
    }
}
```

OUTPUT:

```
PS C:\Users\Admin\Desktop\ADS> cd "c:\Users\Admin\Desktop\ADS\" ; if ($?) { g++ avl.cpp -o avl } ; if ($?) { .\avl }
```

AVL Tree Menu:

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1

Enter key to insert: 12

AVL Tree Menu:

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1

Enter key to insert: 5

AVL Tree Menu:

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1

Enter key to insert: 66

AVL Tree Menu:

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1

Enter key to insert: 88

AVL Tree Menu:

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 3

Inorder traversal of AVL tree: 5 12 66 88

AVL Tree Menu:

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 2

Enter key to delete: 66

AVL Tree Menu:

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 3

Inorder traversal of AVL tree: 5 12 88

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

```
#include <iostream>
using namespace std;

class Node {
public:
    int data1;
    int data2;
    Node* left;
    Node* mid;
    Node* right;
    Node* parent;

    Node(int data) {
        data1 = data;
        data2 = -1; // -1 indicates empty
        left = nullptr;
        mid = nullptr;
        right = nullptr;
        parent = nullptr;
    }
};

class TwoThreeTree {
private:
    Node* root;

    Node* insert(Node* node, int data) {
        if (node == nullptr) {
            return new Node(data);
        }

        if (node->data2 == -1) {
            if (data < node->data1) {
                node->data2 = node->data1;
                node->data1 = data;
            } else {
                node->data2 = data;
            }
            return node;
        }

        if (data < node->data1) {
            node->left = insert(node->left, data);
        } else if (data > node->data2) {
            node->right = insert(node->right, data);
        } else {
            node->mid = insert(node->mid, data);
        }
    }
};
```

```

        return node;
    }

Node* remove(Node* node, int data) {
    if (node == nullptr) {
        return nullptr;
    }

    if (node->data1 == data && node->data2 == -1) {
        delete node;
        return nullptr;
    }

    if (node->data1 == data && node->data2 != -1) {
        node->data1 = node->data2;
        node->data2 = -1;
        return node;
    }

    if (node->data2 == data && node->mid == nullptr) {
        node->data2 = -1;
        return node;
    }

    if (data < node->data1) {
        node->left = remove(node->left, data);
    } else if ((data < node->data2 && data > node->data1) || (node->data1 == data &&
node->mid != nullptr)) {
        node->mid = remove(node->mid, data);
    } else {
        node->right = remove(node->right, data);
    }

    return node;
}

void traverse(Node* node) {
    if (node != nullptr) {
        traverse(node->left);
        cout << node->data1 << " ";
        if (node->data2 != -1) {
            cout << node->data2 << " ";
        }
        traverse(node->mid);
        traverse(node->right);
    }
}

```

public:

```

TwoThreeTree() {
    root = nullptr;
}

void insert(int data) {
    root = insert(root, data);
}

void remove(int data) {
    root = remove(root, data);
}

void display() {
    traverse(root);
}
};

int main() {
    TwoThreeTree tree;
    tree.insert(10);
    tree.insert(20);
    tree.insert(5);
    tree.insert(15);
    tree.insert(30);
    tree.insert(25);

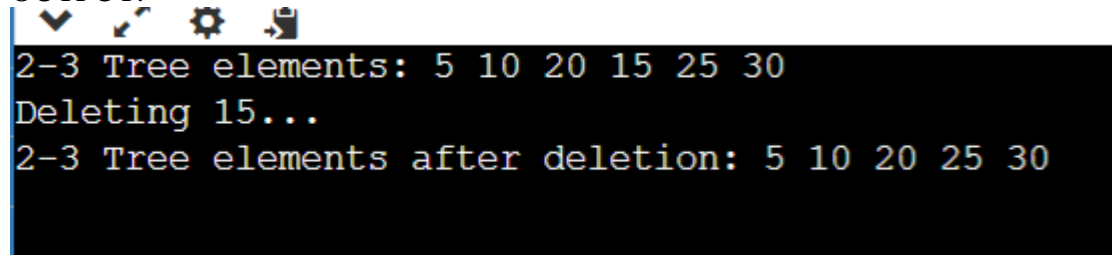
    cout << "2-3 Tree elements: ";
    tree.display();
    cout << endl;

    cout << "Deleting 15...\n";
    tree.remove(15);
    cout << "2-3 Tree elements after deletion: ";
    tree.display();
    cout << endl;

    return 0;
}

```

OUTPUT:



```

2-3 Tree elements: 5 10 20 15 25 30
Deleting 15...
2-3 Tree elements after deletion: 5 10 20 25 30

```

```
2-3 Tree elements: 5 10 20 15 25 30
Deleting 15...
2-3 Tree elements after deletion: 5 10 20 25 30
```

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

```
#include <iostream>

enum Color { RED, BLACK };

struct Node {
    int data;
    Color color;
    Node *parent, *left, *right;
};

class RedBlackTree {
private:
    Node *root;

    // Helper function to rotate left
    void rotateLeft(Node *x) {
        Node *y = x->right;
        x->right = y->left;
        if (y->left != nullptr)
            y->left->parent = x;
        y->parent = x->parent;
        if (x->parent == nullptr)
            root = y;
        else if (x == x->parent->left)
            x->parent->left = y;
        else
            x->parent->right = y;
        y->left = x;
        x->parent = y;
    }

    // Helper function to rotate right
    void rotateRight(Node *x) {
        Node *y = x->left;
        x->left = y->right;
        if (y->right != nullptr)
```

```

        y->right->parent = x;
    y->parent = x->parent;
    if (x->parent == nullptr)
        root = y;
    else if (x == x->parent->right)
        x->parent->right = y;
    else
        x->parent->left = y;
    y->right = x;
    x->parent = y;
}

// Helper function to fix the violation after BST insert
void fixViolation(Node *&ptr) {
    Node *parentPtr = nullptr;
    Node *grandParentPtr = nullptr;
    while ((ptr != root) && (ptr->color != BLACK) && (ptr->parent->color
== RED)) {
        parentPtr = ptr->parent;
        grandParentPtr = ptr->parent->parent;

        /* Case : A
        Parent of ptr is left child of Grand-parent of ptr */
        if (parentPtr == grandParentPtr->left) {
            Node *unclePtr = grandParentPtr->right;

            /* Case : 1
            The uncle of ptr is also red
            Only Recoloring required */
            if (unclePtr != nullptr && unclePtr->color == RED) {
                grandParentPtr->color = RED;
                parentPtr->color = BLACK;
                unclePtr->color = BLACK;
                ptr = grandParentPtr;
            } else {
                /* Case : 2
                ptr is right child of its parent
                Left-rotation required */
                if (ptr == parentPtr->right) {
                    rotateLeft(parentPtr);
                    ptr = parentPtr;
                    parentPtr = ptr->parent;
                }
            }
        }
    }
}

```

```

        /* Case : 3
           ptr is left child of its parent
           Right-rotation required */
        rotateRight(grandParentPtr);
        std::swap(parentPtr->color, grandParentPtr->color);
        ptr = parentPtr;
    }
} else {
    /* Case : B
       Parent of ptr is right child of Grand-parent of ptr */
    Node *unclePtr = grandParentPtr->left;

    /* Case : 1
       The uncle of ptr is also red
       Only Recoloring required */
    if ((unclePtr != nullptr) && (unclePtr->color == RED)) {
        grandParentPtr->color = RED;
        parentPtr->color = BLACK;
        unclePtr->color = BLACK;
        ptr = grandParentPtr;
    } else {
        /* Case : 2
           ptr is left child of its parent
           Right-rotation required */
        if (ptr == parentPtr->left) {
            rotateRight(parentPtr);
            ptr = parentPtr;
            parentPtr = ptr->parent;
        }

        /* Case : 3
           ptr is right child of its parent
           Left-rotation required */
        rotateLeft(grandParentPtr);
        std::swap(parentPtr->color, grandParentPtr->color);
        ptr = parentPtr;
    }
}
}
root->color = BLACK;
}

```



```

// Helper function to insert a new node with given data
void insertBST(int data) {
    Node *newNode = new Node;
    newNode->data = data;
    newNode->left = nullptr;
    newNode->right = nullptr;
    newNode->color = RED;

    Node *parent = nullptr;
    Node *current = root;

    while (current != nullptr) {
        parent = current;
        if (newNode->data < current->data)
            current = current->left;
        else
            current = current->right;
    }

    newNode->parent = parent;
    if (parent == nullptr)
        root = newNode;
    else if (newNode->data < parent->data)
        parent->left = newNode;
    else
        parent->right = newNode;

    fixViolation(newNode);
}

// Helper function to print the tree (inorder traversal)
void inorder(Node *ptr) {
    if (ptr == nullptr)
        return;
    inorder(ptr->left);
    std::cout << ptr->data << "(" << (ptr->color == RED ? "RED" :
"BLACK") << ") ";
    inorder(ptr->right);
}

public:
    RedBlackTree() : root(nullptr) {}

```

```

// Public method to insert data into the tree
void insert(int data) {
    insertBST(data);
    std::cout << "After inserting " << data << ": ";
    inorder(root);
    std::cout << std::endl;
}
};

int main() {
    RedBlackTree tree;
    char choice;

    do {
        int value;
        std::cout << "Enter value to insert: ";
        std::cin >> value;

        tree.insert(value);

        std::cout << "Do you want to insert another value? (y/n): ";
        std::cin >> choice;
    } while (choice == 'y' || choice == 'Y');

    return 0;
}

```

OUTPUT:

Write a program to implement functions of Dictionary using Hashing.

```
#include <iostream>
#include <vector>
using namespace std;

// Define the structure for dictionary entry
struct KeyValuePair {
    int key;
    string value;
    KeyValuePair(int k, const string& v) : key(k), value(v) {}
};

// Define the Dictionary class
class Dictionary {
private:
    vector<KeyValuePair*> table;
    int capacity;

    // Hash function to calculate index
    int hashFunction(int key) {
        return key % capacity;
    }

public:
    // Constructor
    Dictionary(int cap) : capacity(cap) {
        table.resize(capacity, nullptr);
    }
};
```

```
PS C:\Users\Admin\Desktop\ADS> cd "c:\Users\Admin\Desktop\ADS\" ; if ($?) { g++ redBlack.cpp -o redBlack } ; if ($?) { .\redBlack }
Enter value to insert: 5
After inserting 5: 5(BLACK)
Do you want to insert another value? (y/n): y
Enter value to insert: 7
After inserting 7: 5(BLACK) 7(RED)
Do you want to insert another value? (y/n): y
Enter value to insert: 2
After inserting 2: 2(RED) 5(BLACK) 7(RED)
Do you want to insert another value? (y/n): y
Enter value to insert: 12
After inserting 12: 2(BLACK) 5(BLACK) 7(BLACK) 12(RED)
Do you want to insert another value? (y/n): y
Enter value to insert: 44
After inserting 44: 2(BLACK) 5(BLACK) 7(RED) 12(BLACK) 44(RED)
Do you want to insert another value? (y/n): n
PS C:\Users\Admin\Desktop\ADS> 
```

```

// Destructor
~Dictionary() {
    for (auto entry : table) {
        delete entry;
    }
}

// Function to insert a key-value pair into the dictionary
void insert(int key, const string& value) {
    int index = hashFunction(key);
    KeyValuePair* newEntry = new KeyValuePair(key, value);
    if (table[index] == nullptr) {
        table[index] = newEntry;
    } else {
        // Collision handling (Linear Probing)
        int newIndex = (index + 1) % capacity;
        while (table[newIndex] != nullptr && newIndex != index) {
            newIndex = (newIndex + 1) % capacity;
        }
        if (table[newIndex] == nullptr) {
            table[newIndex] = newEntry;
        } else {
            cout << "Dictionary is full, unable to insert." << endl;
            delete newEntry;
        }
    }
}

// Function to retrieve the value associated with a given key
string get(int key) {
    int index = hashFunction(key);
    int originalIndex = index;
    while (table[index] != nullptr) {
        if (table[index]->key == key) {
            return table[index]->value;
        }
        index = (index + 1) % capacity;
        if (index == originalIndex) {
            break; // Key not found
        }
    }
    return "Key not found";
}

// Function to delete a key-value pair from the dictionary
void remove(int key) {
    int index = hashFunction(key);
    int originalIndex = index;
    while (table[index] != nullptr) {
        if (table[index]->key == key) {

```

```

        delete table[index];
        table[index] = nullptr;
        cout << "Key-value pair deleted." << endl;
        return;
    }
    index = (index + 1) % capacity;
    if (index == originalIndex) {
        break; // Key not found
    }
}
cout << "Key not found, unable to delete." << endl;
}
};

// Main function
int main() {
    int capacity;
    cout << "Enter the capacity of the dictionary: ";
    cin >> capacity;

    // Create a dictionary with user-defined capacity
    Dictionary dict(capacity);

    // Insert key-value pairs
    int key;
    string value;
    char choice;
    do {
        cout << "Enter key: ";
        cin >> key;
        cout << "Enter value: ";
        cin >> value;
        dict.insert(key, value);
        cout << "Do you want to insert another key-value pair? (y/n): ";
        cin >> choice;
    } while (choice == 'y' || choice == 'Y');

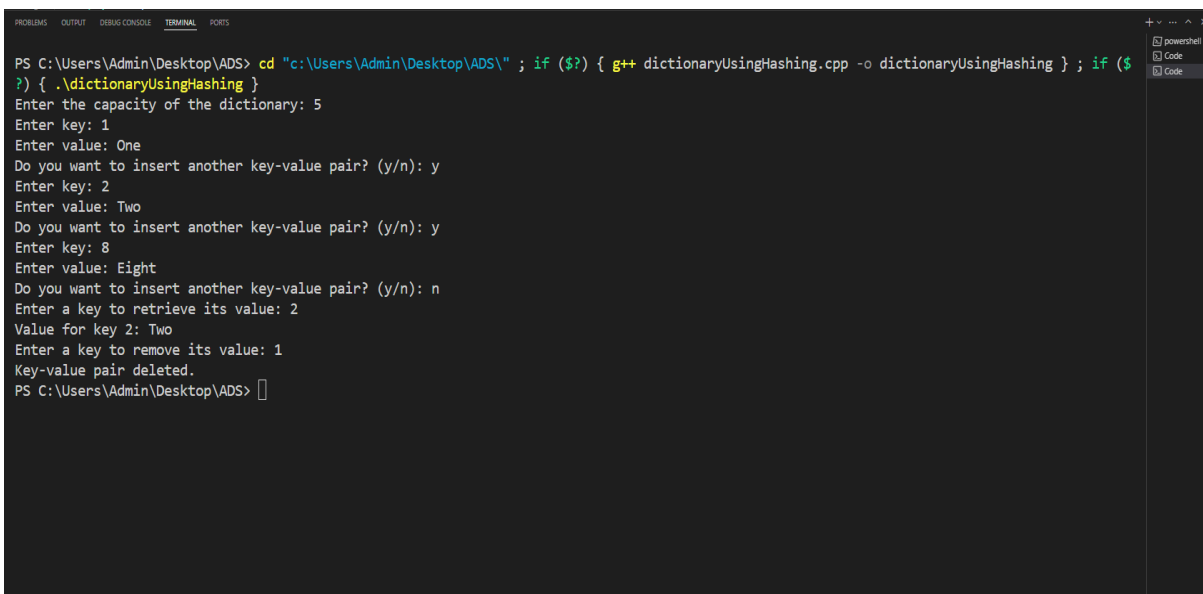
    // Retrieve values for keys
    cout << "Enter a key to retrieve its value: ";
    cin >> key;
    cout << "Value for key " << key << ": " << dict.get(key) << endl;

    // Remove a key-value pair
    cout << "Enter a key to remove its value: ";
    cin >> key;
    dict.remove(key);

    return 0;
}

```

OUTPUT:



```
PS C:\Users\Admin\Desktop\ADS> cd "C:\Users\Admin\Desktop\ADS\" ; if ($?) { g++ dictionaryUsingHashing.cpp -o dictionaryUsingHashing } ; if ($?) { .\dictionaryUsingHashing }
Enter the capacity of the dictionary: 5
Enter key: 1
Enter value: One
Do you want to insert another key-value pair? (y/n): y
Enter key: 2
Enter value: Two
Do you want to insert another key-value pair? (y/n): y
Enter key: 8
Enter value: Eight
Do you want to insert another key-value pair? (y/n): n
Enter a key to retrieve its value: 2
Value for key 2: Two
Enter a key to remove its value: 1
Key-value pair deleted.
PS C:\Users\Admin\Desktop\ADS>
```

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 a 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 <vector>

#include <climits>

using namespace std; struct Node {
int key;
int degree; Node* parent; Node* child; Node* sibling;
};

Node* createNode(int key) { Node* newNode = new Node; newNode->key = key;
newNode->degree = 0; newNode->parent = nullptr; newNode->child = nullptr; newNode->sibling = nullptr; return newNode;
}

Node* mergeTrees(Node* tree1, Node* tree2) { if (tree1->key > tree2->key)
swap(tree1, tree2);

tree2->parent = tree1;
tree2->sibling = tree1->child; tree1->child = tree2;
tree1->degree++;

return tree1;
}
```

```

Node* mergeHeaps(Node* heap1, Node* heap2) { Node* dummy = createNode(INT_MIN);
Node* tail = dummy;

while (heap1 && heap2) {
if (heap1->degree <= heap2->degree) { tail->sibling = heap1;
heap1 = heap1->sibling;
} else {
tail->sibling = heap2; heap2 = heap2->sibling;
}
tail = tail->sibling;
}

tail->sibling = (heap1) ? heap1 : heap2; return dummy->sibling;
}

Node* unionHeaps(Node* heap1, Node* heap2) { Node* mergedHeap = mergeHeaps(heap1,
heap2);

if (!mergedHeap) return nullptr;

Node* prev_x = nullptr; Node* x = mergedHeap; Node* next_x = x->sibling;

while (next_x) {
if (x->degree != next_x->degree || (next_x->sibling && next_x->
sibling->degree == x->degree)) {
prev_x = x; x = next_x;
} else {
if (x->key <= next_x->key) {

```



```
x->sibling = next_x->sibling; mergeTrees(x, next_x);
```

```
} else {
```

```
if (!prev_x)
```

```
mergedHeap = next_x;
```

```
else
```

```
prev_x->sibling = next_x;
```

```
mergeTrees(next_x, x); x = next_x;
```

```
}
```

```
}
```

```
next_x = x->sibling;
```

```
}
```

```
return mergedHeap;
```

```
}
```

```
Node* insert(Node* heap, int key) { Node* newNode = createNode(key); return  
unionHeaps(heap, newNode);
```

```
}
```

```
int getMin(Node* heap) { int minKey = INT_MAX; Node* curr = heap; while (curr) {
```

```
if (curr->key < minKey) minKey = curr->key;
```

```
curr = curr->sibling;
```

```
}
```

```
return minKey;
```

```
}
```

```
Node* extractMin(Node* heap) { if (!heap)
```

```
return nullptr;
```

```
int minKey = INT_MAX; Node* minNode = nullptr; Node* prev = nullptr; Node* curr = heap; Node* prevMin = nullptr;
```

```
while (curr) {  
    if (curr->key < minKey) { minKey = curr->key; minNode = curr; prevMin = prev;  
    }  
    prev = curr;  
    curr = curr->sibling;  
}
```

```
if (prevMin)  
    prevMin->sibling = minNode->sibling; else  
    heap = minNode->sibling;
```

```
Node* child = minNode->child; Node* prevChild = nullptr; while (child) {  
    child->parent = nullptr;  
    Node* nextChild = child->sibling; child->sibling = prevChild; prevChild = child;  
    child = nextChild;  
}
```

```
return unionHeaps(heap, prevChild);  
}
```

```
void printHeap(Node* heap) { cout << "Binomial Heap: "; while (heap) {  
    cout << heap->key << "(" << heap->degree << ") "; heap = heap->sibling;  
    }  
    cout << endl;  
}
```

```

int main() {
Node* heap = nullptr;

heap = insert(heap, 100); heap = insert(heap, 10); heap = insert(heap, 55); heap = insert(heap,
34); heap = insert(heap, 50); heap = insert(heap, 39);

printHeap(heap);

cout << "Minimum key: " << getMin(heap) << endl;

heap = extractMin(heap);
cout << "After extracting minimum key: "; printHeap(heap);

return 0;
}

```

OUTPUT:

```

Binomial Heap: 3(2)
Minimum key: 3
After extracting minimum key: Binomial Heap: 5(0) 10(1)

Binomial Heap: 39(1) 10(2)
Minimum key: 10
After extracting minimum key: Binomial Heap: 100(0) 34(2)

```

Lab Program 10: Write a program to implement the following functions on a Binomial heap:

1. **delete(H):** Like Binary Heap, the 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 decreased key with its parent and if the parent's key is more, we swap keys and recur for a parent. We stop when we either reach a node whose parent has a smaller key or we hit the root node.

```

include <iostream>

#include <vector>

```

```

#include <climits>

using namespace std; struct Node {
    int key;
    int degree; Node* parent; Node* child;
    Node* sibling;
};

Node* createNode(int key)
{
    Node* newNode = new Node;
    newNode->key = key;
    newNode->degree = 0;
    newNode->parent = nullptr; newNode->child = nullptr; newNode->sibling = nullptr; return
    newNode;
}

Node* mergeTrees(Node* tree1, Node* tree2) { if (tree1->key > tree2->key)
swap(tree1, tree2);

    tree2->parent = tree1;
    tree2->sibling = tree1->child; tree1->child = tree2;
    tree1->degree++;

    return tree1;
}

Node* mergeHeaps(Node* heap1, Node* heap2) { Node* dummy = createNode(INT_MIN);

```

```
Node* tail = dummy;
```

```
while (heap1 && heap2) {
```

```
if (heap1->degree <= heap2->degree) { tail->sibling = heap1;
```

```
heap1 = heap1->sibling;
```

```
} else {
```

```
tail->sibling = heap2; heap2 = heap2->sibling;
```

```
}
```

```
tail = tail->sibling;
```

```
}
```

```
tail->sibling = (heap1) ? heap1 : heap2; return dummy->sibling;
```

```
}
```

```
Node* unionHeaps(Node* heap1, Node* heap2) { Node* mergedHeap = mergeHeaps(heap1,  
heap2);
```

```
if (!mergedHeap) return nullptr;
```

```
Node* prev_x = nullptr; Node* x = mergedHeap; Node* next_x = x->sibling;
```

```
while (next_x) {
```

```
if (x->degree != next_x->degree || (next_x->sibling && next_x->  
>sibling->degree == x->degree)) {
```

```
prev_x = x; x = next_x;
```

```
} else {
```

```
if (x->key <= next_x->key) {
```

```
x->sibling = next_x->sibling; mergeTrees(x, next_x);
```

```
} else {
```

```

if (!prev_x)
mergedHeap = next_x;
else
prev_x->sibling = next_x;

mergeTrees(next_x, x); x = next_x;
}
}
next_x = x->sibling;
}

return mergedHeap;
}

Node* insert(Node* heap, int key) { Node* newNode = createNode(key); return
unionHeaps(heap, newNode);
}

Node* extractMin(Node* heap) { if (!heap)
return nullptr;

int minKey = INT_MAX; Node* minNode = nullptr; Node* prev = nullptr; Node* curr =
heap; Node* prevMin = nullptr;

while (curr) {
if (curr->key < minKey) { minKey = curr->key; minNode = curr; prevMin = prev;
}
}

```

```

prev = curr;
curr = curr->sibling;
}

if (prevMin)
prevMin->sibling = minNode->sibling; else
heap = minNode->sibling;

Node* child = minNode->child; Node* prevChild = nullptr; while (child) {
child->parent = nullptr;
Node* nextChild = child->sibling; child->sibling = prevChild; prevChild = child;
child = nextChild;
}

return unionHeaps(heap, prevChild);
}Node* findNode(Node* heap, int key) { if (!heap)
return nullptr; if (heap->key == key)
return heap;

Node* res = findNode(heap->child, key); if (res)
return res;

return findNode(heap->sibling, key);
}

void decreaseKey(Node* heap, int oldKey, int newKey) { Node* targetNode =
findNode(heap, oldKey);
if (!targetNode) return;

targetNode->key = newKey;

```

```
while (targetNode->parent && targetNode->key < targetNode->parent->key) {  
    swap(targetNode->key, targetNode->parent->key);
```

```
    targetNode = targetNode->parent;
```

```
}
```

```
}
```

```
Node* deleteNode(Node* heap, int keyToDelete) { decreaseKey(heap, keyToDelete,  
INT_MIN); return extractMin(heap);
```

```
}
```

```
void printHeap(Node* heap) { cout << "Binomial Heap: "; while (heap) {
```

```
    cout << heap->key << "(" << heap->degree << ")" "; heap = heap->sibling;
```

```
}
```

```
    cout << endl;
```

```
}
```

```
int main() {
```

```
    Node* heap = nullptr;
```

```
    heap = insert(heap, 55); heap = insert(heap, 34); heap = insert(heap, 50); heap = insert(heap,  
39);
```

```
// Decrease key example
```



```
cout << "Decreasing key 55 to 5:" << endl; decreaseKey(heap, 55, 5);  
printHeap(heap);
```

```
// Delete node example
```

```
cout << "Deleting key 34:" << endl; heap = deleteNode(heap, 34); printHeap(heap);
```

```
return 0;
```

```
}
```

OUTPUT:

```
Binomial Heap: 39(1) 10(2)  
Minimum key: 10  
After extracting minimum key: Binomial Heap: 100(0) 34(2)  
Decreasing key 55 to 5:  
Binomial Heap: 100(0) 5(2)  
Deleting key 34:  
Binomial Heap: 5(2)
```

```
Binomial Heap: 34(2)  
Minimum key: 34  
After extracting minimum key: Binomial Heap: 55(0) 39(1)  
Decreasing key 55 to 5:  
Binomial Heap: 5(0) 39(1)  
Deleting key 34:  
Binomial Heap: 39(1)
```