==============================Aadesh Gulumbe==============================
============================DSA PRACTICAL EXAM===========================

**1. Consider a student database of SEIT class. Database contains different fields of every student like Roll No, Name and SGPA.**

**a. Design a roll call list, arrange list of students according to roll numbers in ascending order (Use Bubble Sort)**

**Program:**

```
// Online C++ compiler to run C++ program online
#include <iostream>
#include <string>

using namespace std;

// Structure to represent a student
struct Student {
   int rollNo;
   string name;
   float sgpa;
};

// Function to perform bubble sort on the student array based on roll numbers
void bubbleSort(Student arr[], int n) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - i - 1; j++) {
         // Compare roll numbers and swap if necessary
         if (arr[j].rollNo > arr[j + 1].rollNo) {
            // Swap
            swap(arr[j], arr[j + 1]);
         }
      }
   }
}

// Function to print the roll call list
void printRollCallList(Student arr[], int n) {
   cout << "Roll Call List (Sorted by Roll Numbers):\n";
   cout << "-------------------------------------\n";
   cout << "Roll No\tName\tSGPA\n";
   cout << "-------------------------------------\n";

   for (int i = 0; i < n; i++) {
      cout << arr[i].rollNo << "\t" << arr[i].name << "\t" << arr[i].sgpa << "\n";
   }
}

int main() {
   // Number of students in the class
   const int n = 5; // You can change this value based on the actual number of students
```

```
    // Creating an array of students
    Student students[n] = {
        {201, "John", 3.8},
        {204, "Diana", 3.5},
        {202, "Mike", 3.2},
        {205, "Sara", 3.9},
        {203, "Tom", 3.6}
    };

    // Applying bubble sort to arrange students based on roll numbers
    bubbleSort(students, n);

    // Printing the roll call list
    printRollCallList(students, n);

    return 0;
}
```

**Output :-**

```
Roll Call List (Sorted by Roll Numbers):
-------------------------------------
Roll No Name    SGPA
-------------------------------------
201 John    3.8
202 Mike    3.2
203 Tom 3.6
204 Diana   3.5
205 Sara    3.9
```

**2. Consider a student database of SEIT class. Database contains different fields of every student like Roll No, Name and SGPA.**
**Arrange list of students according to name. (Use Insertion sort)**
**Porgram :-**

```
#include <iostream>
#include <string>

using namespace std;

// Structure to represent a student
struct Student {
    int rollNo;
```

```
    string name;
    float sgpa;
};

// Function to perform insertion sort on the student array based on names
void insertionSort(Student arr[], int n) {
    int i, j;
    Student key;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        // Move elements of arr[0..i-1] that are greater than key.name
        // to one position ahead of their current position
        while (j >= 0 && arr[j].name > key.name) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

// Function to print the student list
void printStudentList(Student arr[], int n) {
    cout << "Student List (Sorted by Name):\n";
    cout << "------------------------------------\n";
    cout << "Roll No\tName\tSGPA\n";
    cout << "------------------------------------\n";

    for (int i = 0; i < n; i++) {
        cout << arr[i].rollNo << "\t" << arr[i].name << "\t" << arr[i].sgpa << "\n";
    }
}

int main() {
    // Number of students in the class
    const int n = 5; // You can change this value based on the actual number of students

    // Creating an array of students
    Student students[n] = {
        {101, "Alice", 3.8},
        {103, "Bob", 3.5},
        {102, "Charlie", 3.2},
        {105, "David", 3.9},
        {104, "Eve", 3.6}
    };

    // Applying insertion sort to arrange students based on names
    insertionSort(students, n);
```

```
    // Printing the sorted student list
    printStudentList(students, n);

    return 0;
}
```

**Output :-**

```
Student List (Sorted by Name):
-------------------------------------
Roll No Name    SGPA
-------------------------------------
101 Alice   3.8
103 Bob 3.5
102 Charlie 3.2
105 David   3.9
104 Eve 3.6
```

**3. Consider a student database of SEIT class. Database contains different fields of every student like Roll No, Name and SGPA.**
**Arrange list of students to find out first ten toppers from a class. (Use Quick sort)**
**Program:-**

```
#include <iostream>
#include <string>

using namespace std;

// Structure to represent a student
struct Student {
    int rollNo;
    string name;
    float sgpa;
};

// Function to partition the array and return the pivot index
int partition(Student arr[], int low, int high) {
    float pivot = arr[high].sgpa;
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j].sgpa >= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
```

```
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

// Function to perform quick sort on the student array based on SGPA
void quickSort(Student arr[], int low, int high) {
    if (low < high) {
        int pivot = partition(arr, low, high);

        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
    }
}

// Function to print the top N students
void printTopStudents(Student arr[], int n) {
    cout << "Top " << n << " Students:\n";
    cout << "-------------------------------------\n";
    cout << "Rank\tRoll No\tName\tSGPA\n";
    cout << "-------------------------------------\n";

    for (int i = 0; i < n; i++) {
        cout << i + 1 << "\t" << arr[i].rollNo << "\t" << arr[i].name << "\t" << arr[i].sgpa << "\n";
    }
}

int main() {
    // Number of students in the class
    const int totalStudents = 15; // You can change this value based on the actual number of
students
    const int topN = 5; // Number of top students to display

    // Creating an array of students
    Student students[totalStudents] = {
        {101, "Alice", 3.8},
        {103, "Bob", 3.5},
        {102, "Charlie", 3.2},
        {105, "David", 3.9},
        {104, "Eve", 3.6},
        // Add more students as needed
    };

    // Applying quick sort to arrange students based on SGPA in descending order
    quickSort(students, 0, totalStudents - 1);

    // Printing the top N students
    printTopStudents(students, topN);
```

```
    return 0;
}
```

**Output:-**

```
Top 5 Students:
-------------------------------------
Rank    Roll No Name    SGPA
-------------------------------------
1   105 David   3.9
2   101 Alice   3.8
3   104 Eve 3.6
4   103 Bob 3.5
5   102 Charlie 3.2
```

**4. Consider a student database of SEIT class. Database contains different fields of every student like Roll No, Name and SGPA.**
**Search students according to SGPA. If more than one student having same SGPA, then print list of all students having same SGPA.**
**Program:-**

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

// Structure to represent a student
struct Student {
    int rollNo;
    string name;
    float sgpa;
};

// Function to search students based on SGPA
void searchStudentsBySGPA(Student arr[], int n, float targetSGPA) {
    vector<Student> matchingStudents;

    // Iterate through the array to find students with the target SGPA
    for (int i = 0; i < n; i++) {
        if (arr[i].sgpa == targetSGPA) {
            matchingStudents.push_back(arr[i]);
        }
    }

    // Print the list of students with the target SGPA
    if (matchingStudents.empty()) {
```

```
        cout << "No students found with SGPA " << targetSGPA << endl;
    } else {
        cout << "Students with SGPA " << targetSGPA << ":\n";
        cout << "------------------------------------\n";
        cout << "Roll No\tName\tSGPA\n";
        cout << "------------------------------------\n";

        for (const auto& student : matchingStudents) {
            cout << student.rollNo << "\t" << student.name << "\t" << student.sgpa << "\n";
        }
    }
}

int main() {
    // Number of students in the class
    const int totalStudents = 10; // You can change this value based on the actual number of
students

    // Creating an array of students
    Student students[totalStudents] = {
        {101, "Alice", 3.8},
        {103, "Bob", 3.5},
        {102, "Charlie", 3.2},
        {105, "David", 3.9},
        {104, "Eve", 3.6},
        {106, "Frank", 3.5},
        {107, "Grace", 3.2},
        {108, "Harry", 3.9},
        {109, "Ivy", 3.6},
        {110, "Jack", 3.5}
        // Add more students as needed
    };

    // SGPA to search for
    float targetSGPA = 3.5; // You can change this value based on your search criteria

    // Search and print students with the target SGPA
    searchStudentsBySGPA(students, totalStudents, targetSGPA);

    return 0;
}
```

**Output:-**

```
Students with SGPA 3.5:
------------------------------------
Roll No Name    SGPA
------------------------------------
103 Bob 3.5
106 Frank   3.5
110 Jack    3.5
```

**5. Consider a student database of SEIT class. Database contains different fields of every student like Roll No, Name and SGPA.**
**Search a particular student according to name using binary search without recursion.**
**Program:-**

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

// Structure to represent a student
struct Student {
   int rollNo;
   string name;
   float sgpa;
};

// Function to perform binary search on the student array based on names
int binarySearchByName(Student arr[], int n, string targetName) {
   int low = 0;
   int high = n - 1;

   while (low <= high) {
      int mid = low + (high - low) / 2;

      // Check if the name is present at the middle
      if (arr[mid].name == targetName) {
         return mid;
      }

      // If the name is smaller, ignore the right half
      else if (arr[mid].name > targetName) {
         high = mid - 1;
      }

      // If the name is larger, ignore the left half
      else {
         low = mid + 1;
```

```
        }
    }

    // If the name is not present in the array
    return -1;
}

int main() {
    // Number of students in the class
    const int totalStudents = 10; // You can change this value based on the actual number of
students

    // Creating an array of students (Assuming the array is sorted based on names)
    Student students[totalStudents] = {
        {101, "Alice", 3.8},
        {103, "Bob", 3.5},
        {104, "Charlie", 3.2},
        {106, "David", 3.9},
        {108, "Eve", 3.6},
        {110, "Frank", 3.5},
        {112, "Grace", 3.2},
        {114, "Harry", 3.9},
        {116, "Ivy", 3.6},
        {118, "Jack", 3.7} // Assuming sorted based on names
        // Add more students as needed
    };

    // Name to search for
    string targetName = "Eve"; // You can change this value based on your search criteria

    // Perform binary search and print the result
    int result = binarySearchByName(students, totalStudents, targetName);

    if (result != -1) {
        cout << "Student found:\n";
        cout << "Roll No: " << students[result].rollNo << "\n";
        cout << "Name: " << students[result].name << "\n";
        cout << "SGPA: " << students[result].sgpa << "\n";
    } else {
        cout << "Student not found.\n";
    }

    return 0;
}
```

**Output:-**

```
Student found:
Roll No: 108
Name: Eve
SGPA: 3.6
```

**6. Implement stack as an abstract data type using singly linked list and use this ADT for conversion of infix expression to postfix, prefix and evaluation of postfix and prefix expression.**
**Program:-**

**7. Implement Circular Queue using Circular Linked List. Perform following operations on it.**
**a) Insertion (Enqueue)**
**b) Deletion (Dequeue)**
**c) Display**
**Program:-**

```
#include <iostream>

using namespace std;

// Node structure for the circular linked list
struct Node {
    int data;
    Node* next;
    Node(int value) : data(value), next(nullptr) {}
};

// Circular Queue class based on circular linked list
class CircularQueue {
private:
    Node* front;
    Node* rear;

public:
    CircularQueue() : front(nullptr), rear(nullptr) {}

    // Function to check if the queue is empty
    bool isEmpty() {
        return front == nullptr;
    }

    // Function to enqueue (insert) a value into the queue
    void enqueue(int value) {
        Node* newNode = new Node(value);
        if (isEmpty()) {
            front = rear = newNode;
            rear->next = front; // Make it circular
        } else {
            rear->next = newNode;
```

```
        rear = newNode;
        rear->next = front; // Make it circular
      }
      cout << "Enqueued: " << value << endl;
    }

    // Function to dequeue (delete) a value from the queue
    void dequeue() {
      if (isEmpty()) {
        cout << "Error: Queue is empty. Cannot dequeue.\n";
        return;
      }

      int value = front->data;
      Node* temp = front;

      if (front == rear) {
        front = rear = nullptr;
      } else {
        front = front->next;
        rear->next = front; // Make it circular
      }

      delete temp;
      cout << "Dequeued: " << value << endl;
    }

    // Function to display the elements of the queue
    void display() {
      if (isEmpty()) {
        cout << "Queue is empty.\n";
        return;
      }

      cout << "Circular Queue Elements: ";
      Node* current = front;

      do {
        cout << current->data << " ";
        current = current->next;
      } while (current != front);

      cout << endl;
    }
};

int main() {
    CircularQueue cq;
```

```
    // Enqueue elements
    cq.enqueue(1);
    cq.enqueue(2);
    cq.enqueue(3);

    // Display initial queue
    cq.display();

    // Dequeue elements
    cq.dequeue();
    cq.dequeue();

    // Display after dequeue
    cq.display();

    // Enqueue more elements
    cq.enqueue(4);
    cq.enqueue(5);

    // Display final queue
    cq.display();

    return 0;
}
```

**Output:-**

```
Enqueued: 1
Enqueued: 2
Enqueued: 3
Circular Queue Elements: 1 2 3
Dequeued: 1
Dequeued: 2
Circular Queue Elements: 3
Enqueued: 4
Enqueued: 5
Circular Queue Elements: 3 4 5
```

8. Construct an Expression Tree from postfix and prefix expression. Perform recursive and non-recursive In-order traversals.

Program:-

```
#include <iostream>
#include <stack>
#include <cctype>

using namespace std;

// Node structure for the expression tree
```

```
struct Node {
    char data;
    Node* left;
    Node* right;
    Node(char value) : data(value), left(nullptr), right(nullptr) {}
};

// Function to check if a character is an operator
bool isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/');
}

// Function to construct an expression tree from a postfix expression
Node* constructExpressionTreeFromPostfix(const string& postfix) {
    stack<Node*> st;

    for (char ch : postfix) {
        if (isalnum(ch)) {
            Node* operandNode = new Node(ch);
            st.push(operandNode);
        } else if (isOperator(ch)) {
            Node* operatorNode = new Node(ch);
            operatorNode->right = st.top();
            st.pop();
            operatorNode->left = st.top();
            st.pop();
            st.push(operatorNode);
        }
    }

    return st.top();
}

// Function to construct an expression tree from a prefix expression
Node* constructExpressionTreeFromPrefix(const string& prefix) {
    stack<Node*> st;

    for (int i = prefix.length() - 1; i >= 0; i--) {
        char ch = prefix[i];
        if (isalnum(ch)) {
            Node* operandNode = new Node(ch);
            st.push(operandNode);
        } else if (isOperator(ch)) {
            Node* operatorNode = new Node(ch);
            operatorNode->left = st.top();
            st.pop();
            operatorNode->right = st.top();
            st.pop();
            st.push(operatorNode);
```

```
        }
    }

    return st.top();
}

// Recursive In-order traversal of the expression tree
void recursiveInOrderTraversal(Node* root) {
    if (root) {
        recursiveInOrderTraversal(root->left);
        cout << root->data << " ";
        recursiveInOrderTraversal(root->right);
    }
}

// Non-recursive In-order traversal of the expression tree
void nonRecursiveInOrderTraversal(Node* root) {
    stack<Node*> st;
    Node* current = root;

    while (current || !st.empty()) {
        while (current) {
            st.push(current);
            current = current->left;
        }

        current = st.top();
        st.pop();

        cout << current->data << " ";

        current = current->right;
    }
}

int main() {
    // Example usage

    // Postfix expression: AB+C*
    string postfixExpression = "AB+C*";
    Node* postfixRoot = constructExpressionTreeFromPostfix(postfixExpression);

    // Prefix expression: *+ABC
    string prefixExpression = "*+ABC";
    Node* prefixRoot = constructExpressionTreeFromPrefix(prefixExpression);

    // Recursive In-order traversal of the postfix expression tree
    cout << "Recursive In-order traversal of Postfix Expression Tree: ";
    recursiveInOrderTraversal(postfixRoot);
```

```
    cout << endl;

    // Non-recursive In-order traversal of the prefix expression tree
    cout << "Non-recursive In-order traversal of Prefix Expression Tree: ";
    nonRecursiveInOrderTraversal(prefixRoot);
    cout << endl;

    return 0;
}
```

Output:-

```
Recursive In-order traversal of Postfix Expression Tree: A + B * C
Non-recursive In-order traversal of Prefix Expression Tree: A + B * C
```

**9. Construct an Expression Tree from postfix and prefix expression. Perform recursive and non- recursive pre-order traversals.**

**Program:-**

```
#include <iostream>
#include <stack>
#include <cctype>

using namespace std;

// Node structure for the expression tree
struct Node {
    char data;
    Node* left;
    Node* right;
    Node(char value) : data(value), left(nullptr), right(nullptr) {}
};

// Function to check if a character is an operator
bool isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/');
}

// Function to construct an expression tree from a postfix expression
Node* constructExpressionTreeFromPostfix(const string& postfix) {
    stack<Node*> st;

    for (char ch : postfix) {
        if (isalnum(ch)) {
            Node* operandNode = new Node(ch);
            st.push(operandNode);
        } else if (isOperator(ch)) {
            Node* operatorNode = new Node(ch);
            operatorNode->right = st.top();
            st.pop();
            operatorNode->left = st.top();
```

```
            st.pop();
            st.push(operatorNode);
        }
    }

    return st.top();
}

// Function to construct an expression tree from a prefix expression
Node* constructExpressionTreeFromPrefix(const string& prefix) {
    stack<Node*> st;

    for (int i = prefix.length() - 1; i >= 0; i--) {
        char ch = prefix[i];
        if (isalnum(ch)) {
            Node* operandNode = new Node(ch);
            st.push(operandNode);
        } else if (isOperator(ch)) {
            Node* operatorNode = new Node(ch);
            operatorNode->left = st.top();
            st.pop();
            operatorNode->right = st.top();
            st.pop();
            st.push(operatorNode);
        }
    }

    return st.top();
}

// Recursive Pre-order traversal of the expression tree
void recursivePreOrderTraversal(Node* root) {
    if (root) {
        cout << root->data << " ";
        recursivePreOrderTraversal(root->left);
        recursivePreOrderTraversal(root->right);
    }
}

// Non-recursive Pre-order traversal of the expression tree
void nonRecursivePreOrderTraversal(Node* root) {
    stack<Node*> st;
    st.push(root);

    while (!st.empty()) {
        Node* current = st.top();
        st.pop();

        cout << current->data << " ";
```

```
        if (current->right) {
            st.push(current->right);
        }

        if (current->left) {
            st.push(current->left);
        }
    }
}

int main() {
    // Example usage

    // Postfix expression: AB+C*
    string postfixExpression = "AB+C*";
    Node* postfixRoot = constructExpressionTreeFromPostfix(postfixExpression);

    // Prefix expression: *+ABC
    string prefixExpression = "*+ABC";
    Node* prefixRoot = constructExpressionTreeFromPrefix(prefixExpression);

    // Recursive Pre-order traversal of the postfix expression tree
    cout << "Recursive Pre-order traversal of Postfix Expression Tree: ";
    recursivePreOrderTraversal(postfixRoot);
    cout << endl;

    // Non-recursive Pre-order traversal of the prefix expression tree
    cout << "Non-recursive Pre-order traversal of Prefix Expression Tree: ";
    nonRecursivePreOrderTraversal(prefixRoot);
    cout << endl;

    return 0;
}
```

**Output:-**

```
Recursive Pre-order traversal of Postfix Expression Tree: * + A B C
Non-recursive Pre-order traversal of Prefix Expression Tree: * + A B C
```

**10. Construct an Expression Tree from postfix and prefix expression. Perform recursive and non- recursive post-order traversals.**

**Program:-**

```
#include <iostream>
#include <stack>
#include <cctype>

using namespace std;
```

```
// Node structure for the expression tree
struct Node {
    char data;
    Node* left;
    Node* right;
    Node(char value) : data(value), left(nullptr), right(nullptr) {}
};

// Function to check if a character is an operator
bool isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/');
}

// Function to construct an expression tree from a postfix expression
Node* constructExpressionTreeFromPostfix(const string& postfix) {
    stack<Node*> st;

    for (char ch : postfix) {
        if (isalnum(ch)) {
            Node* operandNode = new Node(ch);
            st.push(operandNode);
        } else if (isOperator(ch)) {
            Node* operatorNode = new Node(ch);
            operatorNode->right = st.top();
            st.pop();
            operatorNode->left = st.top();
            st.pop();
            st.push(operatorNode);
        }
    }

    return st.top();
}

// Function to construct an expression tree from a prefix expression
Node* constructExpressionTreeFromPrefix(const string& prefix) {
    stack<Node*> st;

    for (int i = prefix.length() - 1; i >= 0; i--) {
        char ch = prefix[i];
        if (isalnum(ch)) {
            Node* operandNode = new Node(ch);
            st.push(operandNode);
        } else if (isOperator(ch)) {
            Node* operatorNode = new Node(ch);
            operatorNode->left = st.top();
            st.pop();
            operatorNode->right = st.top();
            st.pop();
```

```
            st.push(operatorNode);
        }
    }

    return st.top();
}

// Recursive Post-order traversal of the expression tree
void recursivePostOrderTraversal(Node* root) {
    if (root) {
        recursivePostOrderTraversal(root->left);
        recursivePostOrderTraversal(root->right);
        cout << root->data << " ";
    }
}

// Non-recursive Post-order traversal of the expression tree
void nonRecursivePostOrderTraversal(Node* root) {
    stack<Node*> st;
    stack<char> result;

    st.push(root);

    while (!st.empty()) {
        Node* current = st.top();
        st.pop();

        result.push(current->data);

        if (current->left) {
            st.push(current->left);
        }

        if (current->right) {
            st.push(current->right);
        }
    }

    while (!result.empty()) {
        cout << result.top() << " ";
        result.pop();
    }
}

int main() {
    // Example usage

    // Postfix expression: AB+C*
    string postfixExpression = "AB+C*";
```

```
    Node* postfixRoot = constructExpressionTreeFromPostfix(postfixExpression);

    // Prefix expression: *+ABC
    string prefixExpression = "*+ABC";
    Node* prefixRoot = constructExpressionTreeFromPrefix(prefixExpression);

    // Recursive Post-order traversal of the postfix expression tree
    cout << "Recursive Post-order traversal of Postfix Expression Tree: ";
    recursivePostOrderTraversal(postfixRoot);
    cout << endl;

    // Non-recursive Post-order traversal of the prefix expression tree
    cout << "Non-recursive Post-order traversal of Prefix Expression Tree: ";
    nonRecursivePostOrderTraversal(prefixRoot);
    cout << endl;

    return 0;
}
```

**Output:-**

```
Recursive Post-order traversal of Postfix Expression Tree: A B + C *
Non-recursive Post-order traversal of Prefix Expression Tree: A B + C *
```

11**. Implement binary search tree and perform following operations:**
**a) Insert (Handle insertion of duplicate entry)**
**b) Delete**
**Program:-**

```
#include <iostream>

using namespace std;

// Node structure for the Binary Search Tree
struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int value) : key(value), left(nullptr), right(nullptr) {}
};

// Binary Search Tree class
class BinarySearchTree {
private:
    Node* root;

    // Private helper functions
    Node* insertRecursive(Node* current, int key);
    Node* deleteRecursive(Node* current, int key);
```

```
    Node* findMin(Node* node);
    void inorderTraversalRecursive(Node* node);

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

    // Public functions
    void insert(int key);
    void remove(int key);
    void displayInorder();
};

// Insert a key into the BST (handles duplicate entries)
Node* BinarySearchTree::insertRecursive(Node* current, int key) {
    if (current == nullptr) {
        return new Node(key);
    }

    if (key < current->key) {
        current->left = insertRecursive(current->left, key);
    } else if (key > current->key) {
        current->right = insertRecursive(current->right, key);
    } else {
        // Duplicate key, do nothing (or handle as needed)
    }

    return current;
}

// Delete a key from the BST
Node* BinarySearchTree::deleteRecursive(Node* current, int key) {
    if (current == nullptr) {
        return current;
    }

    if (key < current->key) {
        current->left = deleteRecursive(current->left, key);
    } else if (key > current->key) {
        current->right = deleteRecursive(current->right, key);
    } else {
        // Node with only one child or no child
        if (current->left == nullptr) {
            Node* temp = current->right;
            delete current;
            return temp;
        } else if (current->right == nullptr) {
            Node* temp = current->left;
            delete current;
            return temp;
```

```
        }

        // Node with two children, get the inorder successor (smallest in the right subtree)
        Node* temp = findMin(current->right);

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

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

    return current;
}

// Find the node with the minimum key value in the BST
Node* BinarySearchTree::findMin(Node* node) {
    while (node->left != nullptr) {
        node = node->left;
    }
    return node;
}

// Display the BST using inorder traversal
void BinarySearchTree::inorderTraversalRecursive(Node* node) {
    if (node != nullptr) {
        inorderTraversalRecursive(node->left);
        cout << node->key << " ";
        inorderTraversalRecursive(node->right);
    }
}

// Public function to insert a key into the BST
void BinarySearchTree::insert(int key) {
    root = insertRecursive(root, key);
}

// Public function to remove a key from the BST
void BinarySearchTree::remove(int key) {
    root = deleteRecursive(root, key);
}

// Public function to display the BST using inorder traversal
void BinarySearchTree::displayInorder() {
    cout << "Inorder Traversal: ";
    inorderTraversalRecursive(root);
    cout << endl;
}
```

```
int main() {
   BinarySearchTree bst;

   // Insert keys into the BST
   bst.insert(50);
   bst.insert(30);
   bst.insert(70);
   bst.insert(20);
   bst.insert(40);
   bst.insert(60);
   bst.insert(80);

   // Display the BST
   bst.displayInorder();

   // Remove a key from the BST
   bst.remove(30);

   // Display the BST after removal
   bst.displayInorder();

   return 0;
}
```

**Output:-**

```
Inorder Traversal: 20 30 40 50 60 70 80
Inorder Traversal: 20 40 50 60 70 80
```

**12. Implement binary search tree and perform following operations:**
**a) Insert (Handle insertion of duplicate entry)**
**b) Search**
**Program:-**

```
#include <iostream>

using namespace std;

// Node structure for the Binary Search Tree
struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int value) : key(value), left(nullptr), right(nullptr) {}
};

// Binary Search Tree class
class BinarySearchTree {
private:
    Node* root;
```

```cpp
    // Private helper functions
    Node* insertRecursive(Node* current, int key);
    bool searchRecursive(Node* current, int key);

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

    // Public functions
    void insert(int key);
    bool search(int key);
};

// Insert a key into the BST (handles duplicate entries)
Node* BinarySearchTree::insertRecursive(Node* current, int key) {
    if (current == nullptr) {
        return new Node(key);
    }

    if (key < current->key) {
        current->left = insertRecursive(current->left, key);
    } else if (key > current->key) {
        current->right = insertRecursive(current->right, key);
    } else {
        // Duplicate key, do nothing (or handle as needed)
    }

    return current;
}

// Search for a key in the BST
bool BinarySearchTree::searchRecursive(Node* current, int key) {
    if (current == nullptr) {
        return false;
    }

    if (key == current->key) {
        return true;
    } else if (key < current->key) {
        return searchRecursive(current->left, key);
    } else {
        return searchRecursive(current->right, key);
    }
}

// Public function to insert a key into the BST
void BinarySearchTree::insert(int key) {
    root = insertRecursive(root, key);
}

// Public function to search for a key in the BST
```

```
bool BinarySearchTree::search(int key) {
    return searchRecursive(root, key);
}

int main() {
    BinarySearchTree bst;

    // Insert keys into the BST
    bst.insert(50);
    bst.insert(30);
    bst.insert(70);
    bst.insert(20);
    bst.insert(40);
    bst.insert(60);
    bst.insert(80);

    // Search for keys in the BST
    cout << "Search for key 40: " << (bst.search(40) ? "Found" : "Not Found") << endl;
    cout << "Search for key 90: " << (bst.search(90) ? "Found" : "Not Found") << endl;

    return 0;
}
```

**Output:-**

```
Search for key 40: Found
Search for key 90: Not Found
```

**13. Implement binary search tree and perform following operations:**
**a) Insert (Handle insertion of duplicate entry)**
**b) Display - Depth of tree**

**Program:-**

```
#include <iostream>

using namespace std;

// Node structure for the Binary Search Tree
struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int value) : key(value), left(nullptr), right(nullptr) {}
};

// Binary Search Tree class
class BinarySearchTree {
private:
    Node* root;

    // Private helper functions
```

```
    Node* insertRecursive(Node* current, int key);
    int calculateDepth(Node* node);

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

    // Public functions
    void insert(int key);
    void displayDepth();
};

// Insert a key into the BST (handles duplicate entries)
Node* BinarySearchTree::insertRecursive(Node* current, int key) {
    if (current == nullptr) {
        return new Node(key);
    }

    if (key < current->key) {
        current->left = insertRecursive(current->left, key);
    } else if (key > current->key) {
        current->right = insertRecursive(current->right, key);
    } else {
        // Duplicate key, do nothing (or handle as needed)
    }

    return current;
}

// Calculate the depth of the BST
int BinarySearchTree::calculateDepth(Node* node) {
    if (node == nullptr) {
        return 0;
    }

    int leftDepth = calculateDepth(node->left);
    int rightDepth = calculateDepth(node->right);

    return 1 + max(leftDepth, rightDepth);
}

// Public function to insert a key into the BST
void BinarySearchTree::insert(int key) {
    root = insertRecursive(root, key);
}

// Public function to display the depth of the BST
void BinarySearchTree::displayDepth() {
    int depth = calculateDepth(root);
    cout << "Depth of the Binary Search Tree: " << depth << endl;
}
```

```
int main() {
    BinarySearchTree bst;

    // Insert keys into the BST
    bst.insert(50);
    bst.insert(30);
    bst.insert(70);
    bst.insert(20);
    bst.insert(40);
    bst.insert(60);
    bst.insert(80);

    // Display the depth of the BST
    bst.displayDepth();

    return 0;
}
```

**Output:-**

```
Depth of the Binary Search Tree: 3
```

**14. Implement binary search tree and perform following operations:**
**a) Insert (Handle insertion of duplicate entry)**
**b) Display - Mirror image**

**Program:-**

```
#include <iostream>

using namespace std;

// Node structure for the Binary Search Tree
struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int value) : key(value), left(nullptr), right(nullptr) {}
};

// Binary Search Tree class
class BinarySearchTree {
private:
    Node* root;

    // Private helper functions
    Node* insertRecursive(Node* current, int key);
    Node* mirrorImageRecursive(Node* node);
    void inorderTraversalRecursive(Node* node);

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

    // Public functions
```

```
    void insert(int key);
    void displayMirrorImage();
    void displayInorder();
};

// Insert a key into the BST (handles duplicate entries)
Node* BinarySearchTree::insertRecursive(Node* current, int key) {
    if (current == nullptr) {
        return new Node(key);
    }

    if (key < current->key) {
        current->left = insertRecursive(current->left, key);
    } else if (key > current->key) {
        current->right = insertRecursive(current->right, key);
    } else {
        // Duplicate key, do nothing (or handle as needed)
    }

    return current;
}

// Generate the mirror image of the BST
Node* BinarySearchTree::mirrorImageRecursive(Node* node) {
    if (node == nullptr) {
        return nullptr;
    }

    // Swap left and right subtrees
    Node* temp = node->left;
    node->left = mirrorImageRecursive(node->right);
    node->right = mirrorImageRecursive(temp);

    return node;
}

// Display the mirror image of the BST using inorder traversal
void BinarySearchTree::displayMirrorImage() {
    root = mirrorImageRecursive(root);
    cout << "Mirror Image (Inorder Traversal): ";
    inorderTraversalRecursive(root);
    cout << endl;
}

// Inorder traversal of the BST
void BinarySearchTree::inorderTraversalRecursive(Node* node) {
    if (node != nullptr) {
        inorderTraversalRecursive(node->left);
        cout << node->key << " ";
        inorderTraversalRecursive(node->right);
    }
}

// Public function to insert a key into the BST
void BinarySearchTree::insert(int key) {
```

```
    root = insertRecursive(root, key);
}

// Public function to display the inorder traversal of the BST
void BinarySearchTree::displayInorder() {
    cout << "Inorder Traversal: ";
    inorderTraversalRecursive(root);
    cout << endl;
}

int main() {
    BinarySearchTree bst;

    // Insert keys into the BST
    bst.insert(50);
    bst.insert(30);
    bst.insert(70);
    bst.insert(20);
    bst.insert(40);
    bst.insert(60);
    bst.insert(80);

    // Display the original BST
    bst.displayInorder();

    // Display the mirror image of the BST
    bst.displayMirrorImage();

    return 0;
}
```

**Output:-**

```
Inorder Traversal: 20 30 40 50 60 70 80
Mirror Image (Inorder Traversal): 80 70 60 50 40 30 20
```

**15. Implement binary search tree and perform following operations:**
**a) Insert (Handle insertion of duplicate entry)**
**b) Create a copy**
**Program:-**

```
#include <iostream>

using namespace std;

// Node structure for the Binary Search Tree
struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int value) : key(value), left(nullptr), right(nullptr) {}
};

// Binary Search Tree class
```

```
class BinarySearchTree {
private:
   Node* root;

   // Private helper functions
   Node* insertRecursive(Node* current, int key);
   Node* copyRecursive(Node* node);
   void inorderTraversalRecursive(Node* node);

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

   // Public functions
   void insert(int key);
   BinarySearchTree createCopy();
   void displayInorder();
};

// Insert a key into the BST (handles duplicate entries)
Node* BinarySearchTree::insertRecursive(Node* current, int key) {
   if (current == nullptr) {
      return new Node(key);
   }

   if (key < current->key) {
      current->left = insertRecursive(current->left, key);
   } else if (key > current->key) {
      current->right = insertRecursive(current->right, key);
   } else {
      // Duplicate key, do nothing (or handle as needed)
   }

   return current;
}

// Create a copy of the BST
Node* BinarySearchTree::copyRecursive(Node* node) {
   if (node == nullptr) {
      return nullptr;
   }

   Node* newNode = new Node(node->key);
   newNode->left = copyRecursive(node->left);
   newNode->right = copyRecursive(node->right);

   return newNode;
}

// Inorder traversal of the BST
```

```
void BinarySearchTree::inorderTraversalRecursive(Node* node) {
    if (node != nullptr) {
        inorderTraversalRecursive(node->left);
        cout << node->key << " ";
        inorderTraversalRecursive(node->right);
    }
}

// Public function to insert a key into the BST
void BinarySearchTree::insert(int key) {
    root = insertRecursive(root, key);
}

// Public function to create a copy of the BST
BinarySearchTree BinarySearchTree::createCopy() {
    BinarySearchTree copyTree;
    copyTree.root = copyRecursive(root);
    return copyTree;
}

// Public function to display the inorder traversal of the BST
void BinarySearchTree::displayInorder() {
    cout << "Inorder Traversal: ";
    inorderTraversalRecursive(root);
    cout << endl;
}

int main() {
    BinarySearchTree bst;

    // Insert keys into the BST
    bst.insert(50);
    bst.insert(30);
    bst.insert(70);
    bst.insert(20);
    bst.insert(40);
    bst.insert(60);
    bst.insert(80);

    // Display the original BST
    bst.displayInorder();

    // Create a copy of the BST
    BinarySearchTree copyTree = bst.createCopy();

    // Display the copied BST
    cout << "Inorder Traversal of Copied BST: ";
    copyTree.displayInorder();
```

```
    return 0;
}
```

**Output:-**

```
Inorder Traversal: 20 30 40 50 60 70 80
Inorder Traversal of Copied BST: Inorder Traversal: 20 30 40 50 60 70 80
```

**16. Implement Heap sort to sort given set of values using max heap.**

**Program:-**

```
#include <iostream>
#include <vector>

using namespace std;

// Function to heapify a subtree rooted with node i, assuming max heap property
void heapify(vector<int>& arr, int n, int i) {
    int largest = i;    // Initialize largest as root
    int left = 2 * i + 1;   // Left child
    int right = 2 * i + 2;  // Right child

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// Main function to perform Heap Sort
void heapSort(vector<int>& arr) {
    int n = arr.size();

    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from the heap
    for (int i = n - 1; i > 0; i--) {
        // Move the current root to the end
```

```
        swap(arr[0], arr[i]);

        // Call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

int main() {
    vector<int> values = {12, 11, 13, 5, 6, 7};

    cout << "Original array: ";
    for (int val : values) {
        cout << val << " ";
    }
    cout << endl;

    // Perform Heap Sort
    heapSort(values);

    cout << "Sorted array: ";
    for (int val : values) {
        cout << val << " ";
    }
    cout << endl;

    return 0;
}
```

**Output:-**

```
Original array: 12 11 13 5 6 7
Sorted array: 5 6 7 11 12 13
```

17. Implement Heap sort to sort given set of values using min heap.

Program:-

```
#include <iostream>
#include <vector>

using namespace std;

// Function to heapify a subtree rooted with node i, assuming min heap property
void heapify(vector<int>& arr, int n, int i) {
    int smallest = i;   // Initialize smallest as root
    int left = 2 * i + 1;   // Left child
    int right = 2 * i + 2;  // Right child

    // If left child is smaller than root
    if (left < n && arr[left] < arr[smallest])
        smallest = left;
```

```
    // If right child is smaller than smallest so far
    if (right < n && arr[right] < arr[smallest])
        smallest = right;

    // If smallest is not root
    if (smallest != i) {
        swap(arr[i], arr[smallest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, smallest);
    }
}

// Main function to perform Heap Sort
void heapSort(vector<int>& arr) {
    int n = arr.size();

    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from the heap
    for (int i = n - 1; i > 0; i--) {
        // Move the current root to the end
        swap(arr[0], arr[i]);

        // Call min heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

int main() {
    vector<int> values = {12, 11, 13, 5, 6, 7};

    cout << "Original array: ";
    for (int val : values) {
        cout << val << " ";
    }
    cout << endl;

    // Perform Heap Sort
    heapSort(values);

    cout << "Sorted array: ";
    for (int val : values) {
        cout << val << " ";
    }
    cout << endl;
```

```
    return 0;
}
```

Output:-

```
Original array: 12 11 13 5 6 7
Sorted array: 13 12 11 7 6 5
```