24CQ - Final Test

Data Structures and Algorithms (CSC10004)

Duration: 60 minutes

GENERAL INSTRUCTIONS:

- This exam consists of **2 problems** with a total of **10 points**.
- You may use any IDE. All code must be written in C++.
- Prohibited libraries: `<set>`, `<unordered_set>`, `<map>`, `<unordered_map>`, `<algorithm>`.
- Submit a compressed file named `StudentID.zip` containing:
  - `Exercise_01.cpp`
  - `Exercise_02.cpp`
- Time management is crucial. Allocate time proportionally to the points.

1 P1 - Perfect Balance (5 points)

Given a set of sorted distinct keys, implement a function that constructs a **Binary Search Tree (BST)** such that it is perfectly balanced and exhibits the same comparison sequence as binary search on the sorted array of these keys. The tree must be built by placing the median of each sub-array at the root of each sub-tree, recursively dividing the problem.

Recall, the median of a set of n values is the middle value when arranged in ascending order:

- If n is odd, the median is the $\left(\frac{n+1}{2}\right)$-th smallest element
- If n is even, the median is the average of the $\left(\frac{n}{2}\right)$-th and $\left(\frac{n}{2}+1\right)$-th smallest elements

Constraints

- $1 \leq n \leq 10^4$
- All input keys are distinct integers in sorted order
- The resulting tree must satisfy all BST properties
- You can implement the solution by yourself or base it on the given template code

Input/Output Format

Input:

- A sorted array `A` of n distinct keys: $A[0] < A[1] < \cdots < A[n-1]$

Output: A Binary Search Tree where:

- The median element of A is the root **(1 points)**
- The left sub-tree is constructed from elements before the median **(1 points)**
- The right sub-tree is constructed from elements after the median **(1 points)**
- This pattern applies recursively to all sub-trees
- For any key k, the search path in the BST matches the comparison sequence in binary search **(2 points)**

Examples

Example:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Expected Output:

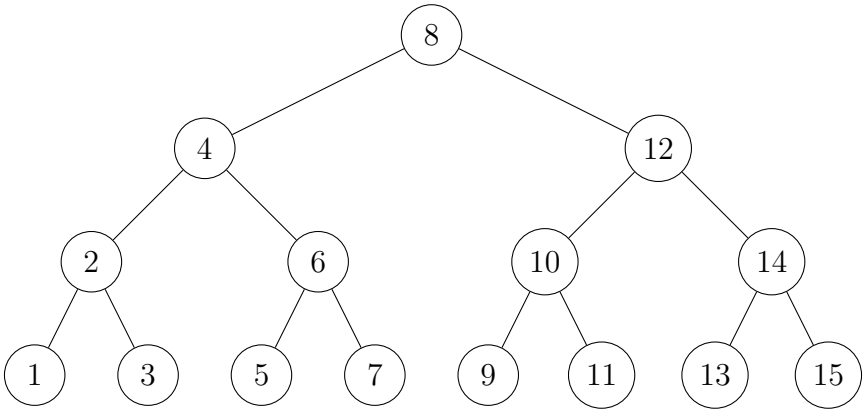
```
Inorder Traversal: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Postoder Traversal: 1 3 2 5 7 6 4 9 11 10 13 15 14 12 8
Preoder Traversal: 8 4 2 1 3 6 5 7 12 10 9 11 14 13 15
Level Order Traversal:
8
4 12
2 6 10 14
```

```
1 3 5 7 9 11 13 15

Search for key 7:
BST search comparisons: 8 4 6 7
Binary search comparisons: 8 4 6 7
```

Visual Explanation

For the example array `[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]`, the resulting BST should look like:



Code Template

```cpp
// File: Exercise_01.cpp
#include <iostream>
#include <vector>
#include <queue> // For level order traversal

// Tree Node
struct Node {
    int key;
    Node* left;
    Node* right;

    // Constructor
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

// TODO: Building tree function
```

```cpp
// Hint: Use recursive approach with start and end indices
Node* createPerfectBST(const std::vector<int>& sortedKeys, int start, int end);

// Wrapper function
Node* createPerfectBST(const std::vector<int>& sortedKeys) {
    return createPerfectBST(sortedKeys, 0, sortedKeys.size() - 1);
}

// TODO: Traversal tree functions
void preorderTraversal(Node* root);
void inorderTraversal(Node* root);
void postorderTraversal(Node* root);
void levelOrderTraversal(Node* root);

// TODO: Search function
bool search(Node* root, int key, std::vector<int>& comparisons);

// TODO: Free tree function
void deleteTree(Node* root);

// TODO: Binary search function for comparison
bool binarySearch(const std::vector<int>& arr, int key, std::vector<int>&
    comparisons);

int main() {
    std::vector<int> sortedKeys = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
    14, 15};
    Node* root = createPerfectBST(sortedKeys);

    std::cout << "Inorder Traversal: ";
    inorderTraversal(root);
    std::cout << std::endl;

    std::cout << "Postoder Traversal: ";
    postorderTraversal(root);
    std::cout << std::endl;

    std::cout << "Preoder Traversal: ";
    preorderTraversal(root);
    std::cout << std::endl;

    std::cout << "Level Order Traversal:" << std::endl;
```

```
    levelOrderTraversal(root);

    int testKey = 7;

    std::vector<int> bstComparisons;
    bool foundInBST = search(root, testKey, bstComparisons);

    std::vector<int> binarySearchComparisons;
    bool foundInArray = binarySearch(sortedKeys, testKey,
   binarySearchComparisons);

    std::cout << "\nSearch for key " << testKey << ":" << std::endl;

    std::cout << "BST search comparisons: ";
    for (int comp : bstComparisons) {
        std::cout << comp << " ";
    }
    std::cout << std::endl;

    std::cout << "Binary search comparisons: ";
    for (int comp : binarySearchComparisons) {
        std::cout << comp << " ";
    }
    std::cout << std::endl;

    deleteTree(root);

    return 0;
}
```

2 P2 - Lowest Common Ancestor (5 points)

(i) Construct an **AVL tree** from a list of n distinct integers **(2.5 points)**

(ii) Find the **Lowest Common Ancestor (LCA)** of two given nodes in the constructed AVL tree **(2.5 points)**

The **Lowest Common Ancestor (LCA)** is defined as the lowest node in the tree that has both p and q as descendants (a node can be a descendant of itself).

Constraints

- $1 \leq n \leq 10^4$
- $-10^9 \leq a_i, p, q \leq 10^9$, for all $i \in [1, \ldots, n]$
- p and q are guaranteed to exist in the input list
- You can implement the solution by yourself or base it on the given template code

Input/Output Format

Input:

- The first line contains the integer n (number of elements)
- The second line contains n distinct integers $a_1, a_2, \ldots, a_n$
- The third line contains two integers p and q

Output:

- A single integer representing the value of the Lowest Common Ancestor (LCA) of nodes p and q

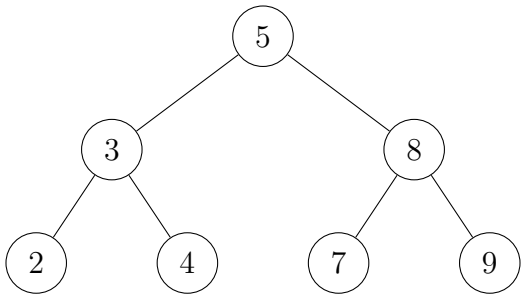
Examples

Input	Output
7 5 3 8 2 4 7 9 2 4	3

Input	Output
7 5 3 8 2 4 7 9 4 9	5

Visual Explanation

For the input `[5, 3, 8, 2, 4, 7, 9]`, the AVL tree constructed would be:



- For $p = 2$ and $q = 4$, their LCA is node 3
- For $p = 4$ and $q = 9$, their LCA is node 5 (the root)

Code Template

```cpp
// File: Exercise_02.cpp
#include <iostream>
#include <vector>

// AVL Tree Node
struct Node {
    int val;
    Node* left;
    Node* right;
    int height;

    // Constructor
    Node(int v) : val(v), left(nullptr), right(nullptr), height(1) {}
};

// TODO: Function to get the height of a node
int getHeight(Node* node);
```

```
18
19 // TODO: Function to calculate the balance factor of a node
20 int getBalanceFactor(Node* node);
21
22 // TODO: Right rotation
23 Node* rightRotate(Node* y);
24
25 // TODO: Left rotation
26 Node* leftRotate(Node* x);
27
28 // TODO: Insert a node into the AVL tree
29 Node* insert(Node* node, int val);
30
31 // TODO: Find the path from root to a node with given value
32 bool findPath(Node* root, int val, std::vector<int>& path);
33
34 // TODO: Find the Lowest Common Ancestor (LCA) of two nodes
35 int findLCA(Node* root, int p, int q);
36
37 // TODO: Utility function to free the memory
38 void deleteTree(Node* root);
39
40 int main() {
41     // Read input
42     int n;
43     std::cin >> n;
44
45     std::vector<int> values(n);
46     for (int i = 0; i < n; i++) {
47         std::cin >> values[i];
48     }
49
50     int p, q;
51     std::cin >> p >> q;
52
53     // Build AVL tree
54     Node* root = nullptr;
55     for (int val : values) {
56         root = insert(root, val);
57     }
58
59     // Find and output the LCA
```

```
60      int lca = findLCA(root, p, q);
61      std::cout << lca << std::endl;
62
63      // Clean up
64      deleteTree(root);
65
66      return 0;
67  }
```

Regulations

Please follow these regulations:

- During the test, internet access and mobile phones are strictly prohibited. Students are responsible for their personal belongings.
- After completing the test, check your submission before and after uploading to Moodle.
- Before leaving, delete your code, turn off the computer, and arrange the chairs properly.
- Prohibited libraries: `<set>`, `<unordered_set>`, `<map>`, `<unordered_map>`, `<algorithm>`.

Your source code must be submitted as a compressed file according to the format `StudentID.zip` with the following structure:

```
StudentID.zip
├── Exercise_01.cpp
└── Exercise_02.cpp
```

Good luck!