Exam Data Structures And Algorithms (B)

For all questions marked with B, please provide your answers using either pseudo-code or simplified Java code. It is important to note that an English text description alone will not be sufficient and will result in zero points. Therefore, your algorithm must be expressed using pseudo-code or simplified Java code.
Although the syntax of your code is not a primary concern, it is important that the idea behind your algorithm is clearly explained.
Any data structure covered in the course can be used without describing it. For example, you may use terms like "list" or "tree", as long as it is clear what data structure you are referring to. However, if there are any specific implementation details that are important to the algorithm, please include them in your pseudo-code. Additionally, please make sure that your code is written in a consistent style, and that variables and functions are named clearly and descriptively.

For example, the following code snippet is acceptable as a description of an algorithm that adds all elements in X to a linked list:

```
LinkedList myList <- LinkedList()
myList.addAll(X)
```

Similarly, the following code snippet is an acceptable description of using a binary search tree to add two elements and search for an element, even though it is not valid Java or pseudo-code:

```
BST bst = new BST()
bst.add(1)
bst.add(5)
if bst.search(4)
    print “found”
else
    print “not found”
```

Please make sure that your algorithm is clearly explained and that you use either pseudo-code or simplified Java code to express it.

B1

You are given an array of integers, and you want to find the two numbers that add up to a given target. Write a Java function that takes the array and target as inputs, and returns the indices of the two numbers that add up to the target. If no such pair exists, the function should return null. Your solution should use a hash table and should have a time complexity of O(n).

Your method should be defined as follows:

```
/**
 * Given an array of integers and a target, returns the indices of the two
numbers that add up to the target. If no such pair exists, returns null.
 *
 * @param nums the array of integers
 * @param target the target sum
 * @return the indices of the two numbers that add up to the target, or
null if no such pair exists
 */
public static int[] twoSum(int[] nums, int target) {
    // your code here
}
```

Examples of what the algorithm should return:

```
twoSum([2, 7, 11, 15], 9)        // expected output: [0, 1]
twoSum([3, 2, 4], 6)             // expected output: [1, 2]
twoSum([3, 3], 6)                // expected output: [0, 1]
twoSum([1, 2, 3, 4, 5], 10)      // expected output: [3, 4]
twoSum([1, 2, 3, 4, 5], 2)       // expected output: [0, 1]
```

To earn the maximum of 20 points, your solution must:

- Correctness (10 points): Code correctly implements the algorithm and produces the correct output for the given inputs.
- Use of HashTable (5 points): Code utilizes a HashTable data structure in an appropriate and effective manner.
- Complexity analysis (5 points): The student correctly identifies and explains the time and space complexity of their code, including the impact of the HashTable data structure on the overall complexity.

Although we do not grade your code based on its quality of formatting and readability, if we cannot understand your code this may have an effect on your final grade for the answer.

Answer:

```
public static int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
        int complement = target - nums[i];
        if (map.containsKey(complement)) {
            return new int[] { map.get(complement), i };
        }
        map.put(nums[i], i);
    }
    return null;
}
```

The solution uses a hash table to keep track of the values in the array and their indices. For each element ***nums[i]*** in the array, we compute its complement ***target - nums[i]*** and check if the complement is already in the hash table. If it is, we return the indices of the two numbers. If it is not, we add ***nums[i]*** and its index to the hash table so that we can look it up later. If we reach the end of the loop without finding a pair that adds up to the target, we return ***null***.

The time complexity of this solution is ***O(n)***, where n is the number of elements in the array. The loop iterates over each element once, and the hash table lookups take constant time on average. We can prove this time complexity by considering the worst case scenario where all elements need to be checked, and noting that the hash table lookups take ***O(1)*** time on average (assuming a good hash function). Therefore, the overall time complexity is ***O(n)***.

B2

You are given a binary tree and a node in the tree, and you want to find the inorder successor of the given node in the tree.

The inorder successor of a node in a binary tree is defined as the next node in the inorder traversal of the tree. In other words, it is the node that you would visit immediately after the given node when doing an inorder traversal.

To find the inorder successor of a node, you can follow these steps:

1. If the given node has a right child, then the inorder successor is the leftmost node in the right subtree of the given node.
2. If the given node does not have a right child, then the inorder successor is the first ancestor that is a left child of its parent. In other words, you need to follow the path up the tree to the root until you encounter a node that is a left child of its parent. The parent of this node is the inorder successor.

If there is no inorder successor (i.e., the given node is the rightmost node in the tree), then the function should return null.

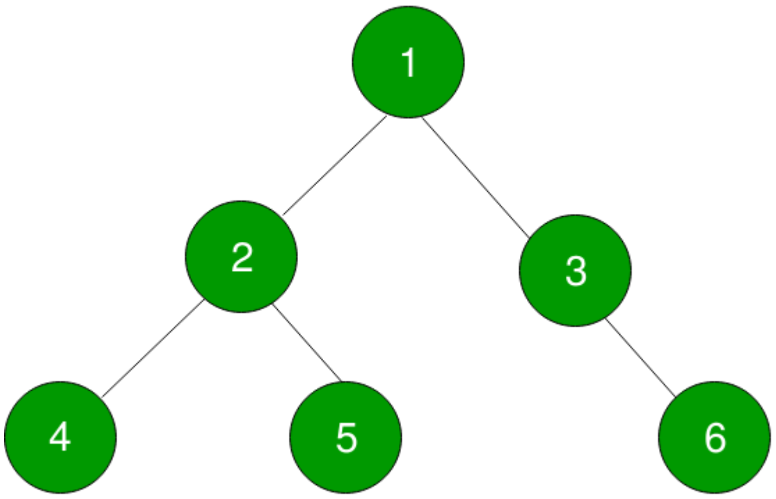
Write a Java function or pseudocode that takes the root of the binary tree and the node as inputs, and returns the inorder successor of the given node. If there is no inorder successor, the function should return null.

After writing the code, give a short explanation of your approach, and explain the complexity of your solution in O notation.

Your method should be defined as follows:

```java
/**
* Given the root of a binary tree and a node in the tree, returns the inorder
successor of the node.
* @param root the root of the binary tree
* @param node the node whose inorder successor is to be found
* @return the inorder successor of the node, or null if there is no inorder
successor
*/
public static TreeNode inorderSuccessor(TreeNode root, TreeNode node) {
       // your code here
}
```

For example, given the following binary tree, where we reference each node with a number:



Example calls to the algorithm will be:

```
inorderSuccessor(root, root.right);          // Returns node  number 6
inorderSuccessor(root, root.left.left);      // Returns node number 2
inorderSuccessor(root, root.right.right);    // Return null
```

To earn the maximum of 20 points, your solution must:

- Correctness (10 points): Code correctly implements the algorithm and produces the correct output for the given inputs.
- Efficiency (5 points): Code utilizes an efficient algorithm and avoids unnecessary computations.
- Complexity analysis (5 points): The student correctly identifies and explains the time and space complexity of their code.

Although we do not grade your code based on its quality of formatting and readability, if we cannot understand your code this may have an effect on your final grade for the answer.

Answer:

```
/**
 * Given the root of a binary tree and a node in the tree, returns the inorder successor of
the node.
 *
 * @param root the root of the binary tree
 * @param node the node whose inorder successor is to be found
 * @return the inorder successor of the node, or null if there is no inorder successor
 */
public static TreeNode inorderSuccessor(TreeNode root, TreeNode node) {
    if (node.right != null) {
        // If the node has a right subtree, the inorder successor is the leftmost node in
the right subtree
        TreeNode current = node.right;
        while (current.left != null) {
            current = current.left;
        }
        return current;
    } else {
        // If the node doesn't have a right subtree, the inorder successor is the nearest
ancestor whose left child is also an ancestor of the node
        TreeNode current = root;
        TreeNode successor = null;
        while (current != null) {
            if (node.val < current.val) {
                successor = current;
                current = current.left;
            } else if (node.val > current.val) {
                current = current.right;
            } else {
                break;
            }
        }
        return successor;
    }
}
```

Explanation:
The algorithm first checks if the given node has a right subtree. If it does, the inorder successor of the node is the leftmost node in the right subtree. If the given node does not have a right subtree, the algorithm searches for the nearest ancestor whose left child is also an ancestor of the node. This can be done by starting at the root of the tree and traversing the tree while keeping track of the nearest ancestor that has a left child on the path to the given node.

Time complexity:
The time complexity of the algorithm is O(h) (can also be explained as ***O(n)*** but then we should explain that ***n*** is the height of the tree), where h is the height of the tree. This is because in the worst case, the algorithm may need to traverse the entire height of the tree to find the inorder successor.