Advanced Algorithms and Data Structures

1DV516

Assignment 1

Pranav Patel, Henrique Hemerly

Exercise 1: MyIntegerBST

```
public Integer mostSimilarValue(Integer value) {
		if (containsNode(value)) {
			return value;
		} else {
			return searchDifference(root, value);
		}
	}

public boolean containsNode(Integer key) {
		if (searchForNode(root, key) == null) {
			return false;
		}
		else if (searchForNode(root, key).key == key) {
			return true;
		}
		else {
			return false;
		}
	}

public Integer searchDifference(Node root, Integer key) {
		Integer min = Integer.MAX_VALUE;
		Integer difference;
		Node closestNode = null;
		Node temp = root;
		while (root != null) {
			if (key > root.key) {
				difference = Math.abs(root.key - key);
				if (difference < min) {
					min = difference;
					closestNode = root;
				}
				if(root.right == null) {
					return closestNode.key;
				} else {
				root = root.right;
				}
			} else if (key < root.key) {
				difference = Math.abs(root.key - key);
				if(difference < min) {
					min = difference;
					closestNode = root;
				}
				if (root.left == null) {
					return closestNode.key;
```

```
                        } else {
                        root = root.left;
                        }

                } else {
                        return root.key;
                }
        }
        return null;
    }

public void printByLevels() {
        int height = heightOfBST(root);

        for(int i = 1; i <= height; i++) {
                System.out.print("Depth " + (i-1) + ": ");
                printOneLevel(root, i);
                System.out.println("");
        }
    }

public void printOneLevel(Node n, int depth) {
    if (n == null ) {
        return;
    }
    if(depth == 1) {
        System.out.print(n.key);
        System.out.print(" ");
    }
    else if(depth > 1) {
        printOneLevel(n.left, depth - 1);
        printOneLevel(n.right, depth - 1);
    }
}
```

In the most similar method, we see that the first section is the contains node method which is called. This calls upon another method which searches for the value itself in the nodes. Taking all the methods into account, the worst time complexity for this algorithm is O(n) since the tree can be fully unbalanced and behave like a linked list. But considering that a Binary Search Tree has at least O(log n) levels, it would take at least O(log n) attempts to find a node. The print method must traverse every element in every depth of the Binary Search Tree meaning that using the method shown above, the worst-case scenario for printing the Binary Search Tree is $O(n^2)$. In a skewed BST, the time complexity would be reduced to O(n). We tried to implement a method using a queue rather than using recursive calls to improve the time complexity for the print levels, however we could not make it function fully. Using a queue would allow the data to be stored instead of having to repeatedly and recursively check every node. If the queue implementation was successful, our calculations indicate that the time complexity would have been O(n) in the worst case scenario.

Exercise 2.3: MySequenceADT

Our proposed idea was to create a linked list without the use of java libraries. This linked list was created with the use of three inner nodes, one root, and two on the side being left and right respectively. Each node points to the ones adjacent and the root node is always in the middle to facilitate the growth of the structure on both sides without misplacing the middle value. Every time there is an insert, we reassign the respective node of the insert method to different pointers and add another node beside it. This implementation of a linked list made it easier and reduced the time complexity.

```
public void insertRight(Integer value) {
        if (root == null) {
                root = new Node(value);
                left = root;
                right = root;
        } else {
                Node inserted = new Node(value);
                inserted.left = right;
                right.right = inserted;
                right = inserted;
        }
}

public void insertLeft(Integer value) {
        1) if (root == null) {
                2)root = new Node(value);
                3)left = root;
                4)right = root;
        } else {
                5)Node inserted = new Node(value);
                6)inserted.right = left;
                7)left.left = inserted;
                8)left = inserted;
        }
}
```

C1. *Time complexity calculation for insert methods:*
Method run 1 time per call with no loops or recursions, therefore:
O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) = O(1)

```
public Integer removeRight() {
		if (right == null) {
			System.err.println("There is nothing in the list.");
			return null;
		} else {
			Integer rightmost = right.value;
			right = right.left;
			right.right = null;
			return rightmost;
		}
	}

public Integer removeLeft() {
		1)if (left == null) {
			2)System.err.println("There is nothing in the list.");
			3)return null;
		} else {
			4)Integer leftmost = left.value;
			5)left = left.right;
			6)left.left = null;
			7)return leftmost;
		}
	}
```

C2. *Time complexity calculation for remove methods:*
Method run 1 time per call with no loops or recursions, therefore:
O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) = O(1)

In the insert methods, each side node pointed to the furthest node in their respective side in real time which meant that inserting values into the sequence is done immediately. Considering there are no loops or recursions, the calculation of the time complexity for the functions was deduced to be O(1). This is shown in the above calculation *C1*.

The remove methods were similar as well due to each side node pointing to their furthest node in the sequence. Using the same reasoning as with the insert methods, the time complexity was deduced to O(1). This calculation is shown in *C2*.

Find Minimum:

```
public Integer findMinimum() {
        1)Integer min = null;
        2)Node now = right;
        3)while (now.left != null) {
                4)if (min == null) {
                        5)min = now.value;
                } 6)else if (min > now.value) {
                        7)min = now.value;
                }
                8)now = now.left;
        }
        9)if (min > now.value) {
                10)min = now.value;
                11)return min;
        }
        12)return min;
    }
```

In the find minimum method, our solution traversed the sequence from one end to the other in an iterative loop. Since every call outside and inside the loop had a constant time complexity of O(1), the only increasing factor to the time complexity was the loop itself and the time complexity increased proportional to the number of elements in the sequence (n). Thus, the calculation for the find minimum time complexity algorithm is as follows:

Order of operations	**Time complexity**
1	O(1)
2	O(1)
3	O(n)
4	O(1)
5	O(1)
6	O(1)
7	O(1)
8	O(1)
9	O(1)
10	O(1)
11	O(1)
12	O(1)

Time complexity for algorithm = O(n)