Assignment 9

**R-3.11** Consider the following sequence of keys:

(5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 13, 33)

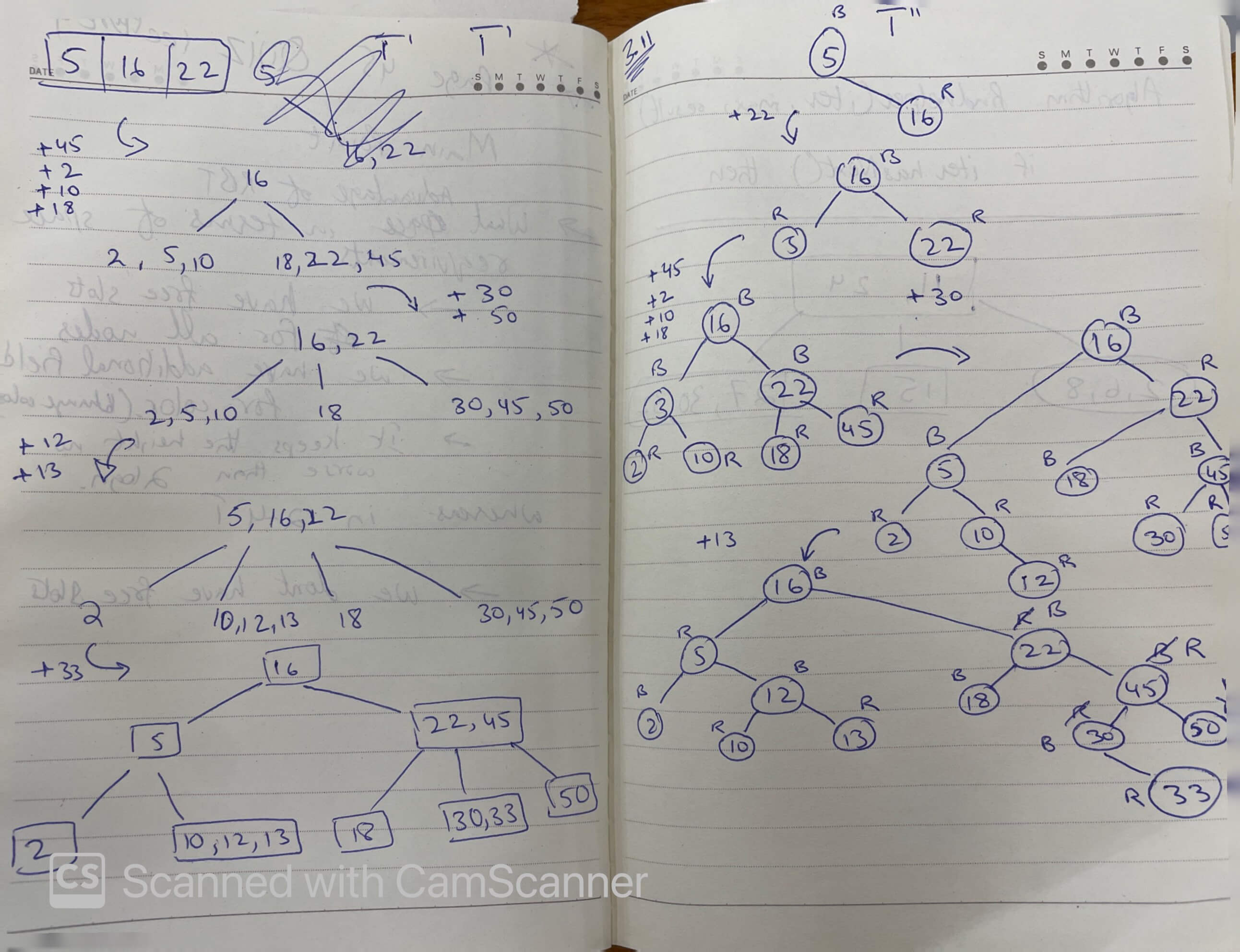
Consider the insertion of items with this set of keys, in the order given, into:

a. an initially empty (2,4) tree T’.

b. an initially empty red-black tree T’’.

Draw T’ and T’’ after each insertion.

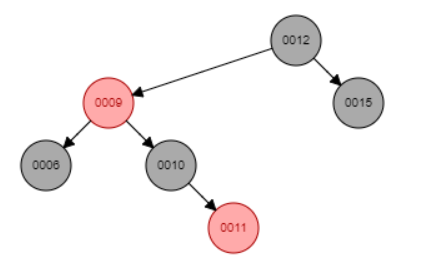
At each step you can check your diagram using the following simulators: http://cs.armstrong.edu/liang/animation/web/24Tree.html https://www.cs.usfca.edu/~galles/visualization/RedBlack.html



**R-3.14** For each of the following statements about red-black trees, determine whether it is true or false. If you think it is true, provide a justification. If you think it is false, give a counterexample.

1. a subtree of a red-black tree is itself a red-black tree.

**Answer:** False. For example:



The subtree 0009 is not a red-black tree when the node 0009 is red

1. the sibling of an external node is either external or it is red.

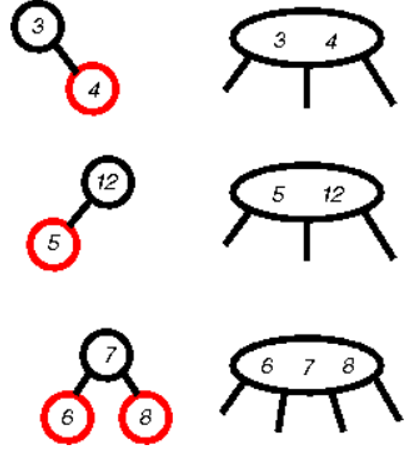
**Answer: True:** the sibling of an external node is either external or it is red.

False. As the example above, 0010 is sibling node of an external node of the node 0015, but the node 0010 is black.

1. given a red-black tree T, there is a unique (2,4) tree T’ associated with T.

**Answer:** False, given a red-black tree T, there is an unique (2,4) tree T’ associated with T.

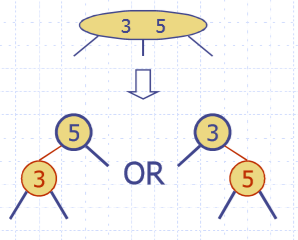
True because a red node always belongs to a black parent node



d. given a (2,4) tree T, there is a unique red-black tree T’ associated with T.

**Answer:**  given a (2,4) tree T, there is an unique red-black tree T’ associated with T.

False. For example: the node (3,5) as below can perform to 2 types of red-black tree



Design a pseudo code algorithm isValidAVL(T) that decides whether or not a binary tree is a valid AVL tree. For this problem, we define valid to mean that the height of the left and right sub-trees of every node do not differ by more than one.

What is the time complexity of your algorithm?

|  |  |
| --- | --- |
| Algorithm isValidAVL(T, p):  **if** T.isEmpty(p) then  **return** **True**  left\_height := findHeight(T.leftChild(p))  right\_height := findHeight(T.rightChild(p))  **if** **abs**(left\_height - right\_height) > 1:  **return** **False**  **return** isValidAVL(T.leftChild(p)) **and**  isValidAVL(T.rightChild(p))  Algorithm findHeight(T, p):  **if** T.isEmpty(p) then  **return** -1  **return** 1 + **max**(  findHeight(T.leftChild(p)),  findHeight (T.rightChild(p)  )) | O(1)  O(1)  O(logn)  O(logn)  O(1)  O(1)  O(nlogn)  O(logn)  O(logn)  O(logn)  Total running time is O(nlogn) |

Design an algorithm, **isPermutation**(A,B) that takes two sequences A and B and determines whether or not they are permutations of each other, i.e., they contain same elements but possibly occurring in a different order. Assume the elements in A and B cannot be sorted. **Hint**: A and B may contain duplicates. Same problem as in previous homework, but this time use a dictionary to solve the problem.

What is the worst case time complexity of your algorithm? Justify your answer.

|  |  |
| --- | --- |
| **Algorithm** **isPermutation**(A, B)  **if** A**.size**() != B**.size**() **or** (A.isEmpty() **or** B.isEmpty()) then  **return** **False**  **Algorithm** **makeDict**(dictTemp, S, i)  **if** i == S.**size**():  **return** dictTemp  e := S.atElement(i)  **if** dictTemp.hasKey(e) then  dictVal := dictTemp.findValue(e)  dictTemp.insertItem(e, dicVal + 1)  **else**  dictTemp.insertItem(id, 1)  **return** **makeDict**(dictTemp, S, i+1)  dictA := makeDict(new Dictionary(HT), A, 0)  dictB := makeDict(new Dictionary(HT), B, 0)  **def** **compareABHelper**(keys, p)  **if** p < keys.**len()** then  **if** dictB.hasKey(keys.atElement(p)) then  **return** **False**  val1 := dictTemp.findValue(keys.atElement(p))  val2 := dictB.findValue(keys.atElement(p))  **if** val1 != val2 then  **return** **False**  **return** compareABHelper(keys, p + 1)  **else**:  **return** **True**  keys := **list**(dictA.keys())  **return** compareABHelper(keys, 0) | O(1)  O(1)  O(n)  O(n)  O(n)  O(n)  O(n)  O(n)  O(n)  O(n)  O(n)  O(1)  O(n)  O(n)  O(n)  O(n)  O(n)  O(n)  O(n)  O(n)  O(n)  O(n)  O(n)  O(n)  O(1)  O(1)  Total running time O(n)  And The worst case time O(n) |

**C-3.10** Let D be an ordered dictionary with n items implemented by means of an AVL tree (or a Red-Black tree). Show how to implement the following operation on D in time O(log n + s), where s is the size of the iterator returned:

FindAllInRange(k1, k2): Return an iterator of all the elements in D with key k such that k1 < k < k2.

|  |  |
| --- | --- |
| **Algorithm** **findAllinRange**(D, k1, k2)  **if** D.isEmpty() **or** not D.hashKey(k1) **or** not D.hashKey(k2) then  **return** **False**  root := D.root()  rangeIndex := new List  rangeHelper(rangeIndex , root, k1, k2)  return rangeIndex  **Algorithm** **rangeHelper**(rangeIndex, p, k1, k2)  **if** p is None then  return rangeIndex  **if** k1< p.key /\ p.key < k2 then  rangeIndex.insertLast(p.key)  **if** p.key > k2 then  **rangeHelper**(rangeIndex, p.leftChild(), k1, k2)  **elif** p.key > k1 then  **rangeHelper**(rangeIndex, p.rightChild(), k1, k2)  else:  **rangeHelper**(rangeIndex, p.leftChild(), k1, k2)  **rangeHelper**(rangeIndex, p.rightChild(), k1, k2) | O(1)  O(1)  O(1)  O(1)  O(logn+s)  O(1)  O(logn+s)  O(logn+s)  O(1)  O(logn+s)  O(logn+s)  O(logn+s)  O(logn+s)  O(logn+s)  O(logn+s)  O(logn+s)  O(logn+s)  Total running time O(logn+s) |