**Assignment 14**

**R-3.11 Consider the following sequence of keys:  
(5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 1, 7, 55)  
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.**

16

22 45

5 10

50 55

30

12

18

1 2

7

**2-4 Tree**

**(5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 1, 7, 55)**

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

False. The subtree may have a red colored root which is not allowed in a red-black tree.

**b. the sibling of an external node is either external or it is red.**True. All new insertion are in external nodes and are in red color only.

**c. given a red-black tree *T*, there is an unique (2,4) tree *T’* associated with *T*.**True. A 2-4 tree accommodates three keys in one node so it becomes a unique tree for a 2-4 tree.

**d. given a (2,4) tree *T*, there is an unique red-black tree *T’* associated with *T*.**False. Since a 2-4 tree accommodates 3 keys in one node, different red-black tree structures can make same 2-4 tree.

**1. Design a pseudo-code 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. Hint: A  
and B may contain duplicates. Same problem as in previous homework, but this  
time use a dictionary to solve the problem.  
2. What is the worst case time complexity of your algorithm? Justify your answer.  
3. Design and solve this problem in four ways in JavaScript:**

**a. By sorting A and B**

Algorthim isPermutation (A,B)  
 If(A.length!==B.length)  
 return false;

|  |  |
| --- | --- |
| SA:=QuikSort(A).   SB:= QuikSort(B). |  |
| i:=0.   while(i<SA.length). do  if(SA[i]!==SB[i]. |  |
| return false;   i ++;   return true |  |

|  |
| --- |
| **b. Using a Priority Queue** Algorthim isPermutatin(A,B)  If(A.length !==B.length). |
| return false; |

PA:=new PriorityQueue().   
 PB:=new PriorityQueue().

|  |  |
| --- | --- |
| insertArrToPQ(A,PA).   insertArrToPQ(B,PB). |  |
|  |  |
|  |  |
| while !PA.isEmpty() do   a:=PQA.removeMin().   b:=PQB.removeMin(). |  |
| if(a!==b)   return false; |  |
| return true |  |

Algorthim insertArrToPQ(arr,PQ)  
 for i=0 to arr.length - 1 do

e = arr[i]  
 PQ.insertItem(e, e)

return PQ

**c. Using a Hash Table based Dictionary**

Algorthim isPermutatin(A,B)  
 If(A.length!==B.length). 1

|  |  |
| --- | --- |
| return false; |  |
| HA:=new HashTable() |  |
| HB:=new HashTable()   insertArrToHT(A,HA)   insertArrToHT(B,HB)   iterA:=HA.items();  iterB:=HB.items(); |  |

while iterA.hasNext() do   
 itemA:=iterA.nextObject();

itemB:=iterB.nextObject();

if itemA.key() != itemB.key() then

return false

if itemA.value() != itemB.value() then

return false

return true

Algorthim insertArrToHT(arr,HT)  
 for i=0 to arr.length-1 do   
 e= arr[i]

HT.insertItem(e, e) // Handle Collisions

return HT

**d. Using a BST based Dictionary**

Algorthim isPermutatin(A,B)  
 If(A.length!==B.length). 1

|  |  |
| --- | --- |
| return false; |  |
| BstA:=new BST-Dictionary() |  |
| BstB:=new BST-Dictionary ()   insertArrToBST(A,BstA)   insertArrToBST(B,BstB)   iterA:=BstA.items();  iterB:=BstB.items(); |  |

while iterA.hasNext() do   
 itemA:=iterA.nextObject();

itemB:=iterB.nextObject();

if itemA.key() != itemB.key() then

return false

if itemA.value() != itemB.value() then

return false

return true

Algorthim insertArrToBstT(arr,BST)  
 for i=0 to arr.length-1 do   
 e= arr[i]

BST.insertItem(e, e)

return HT

**4. Assume the elements in A and B cannot be sorted, i.e., there is no comparator.  
How would this restrict the way you would have to implement a solution to  
isPermutation(A,B), i.e., which of the above strategies could you use and which  
couldn’t you use?**

Hash Table would be the best option which does not require sorting.

**5. Which of the above strategies leaves the inputs A and B unchanged?**

Depending on the algorithm we write, all the strategies can leave inputs unchanged.

**6. Are any of the approaches considered in-place?**

Depending on the algorithm we write, the sorting strategy would be in-place

**7. Calculate the height of a Binary Tree. Implement your solution in the JavaScript  
file RBTree-HW.js that is provided. You are to do this both as a recursive  
function that traverses the tree and secondly using the Euler Tour template class  
(i.e., implement two different functions in JavaScript).**

Algorthim height (T)  
 return heightHelper(T,T.root())-1

Algorthim heightHelper(T,p)  
 If T.isExternal(p)

return 1;   
 leftHeight=1+ heightHelper(T,T.leftChiled(p)).   
 rightHeight=1+ heightHelper(T,T.rightChiled(p).   
 if leftHeight>rightHeight.   
 return leftHeight  
 else

return rightHeight

**8. Calculate the black height of each node of a Red-Black Tree. Implement your  
solution in the JavaScript file RBTree-HW.js that is provided. You are to do this  
both as a recursive function that traverses the tree and secondly using the Euler  
Tour template class (again two different functions). There are two methods on a  
Red-Black tree to determine the color of a node, i.e., T.isRed(p) and T.isBlack(p).  
The black height for each node corresponds to the height of that key in a 2-4  
Tree. The definition of the black-height of a node p, denoted bh(p), is the  
number of black nodes from p to every external node in the subtree rooted at p,  
but not including node p. See the lecture notes for more details and examples.**

Algorthim height (T)  
 return heightHelper(T,T.root())-1

Algorthim heightHelper(T,p)  
 If T.isExternal(p)

return 1;

if T.isRed(p)

return 0  
 leftHeight=1+ heightHelper(T,T.leftChiled(p)).   
 rightHeight=1+ heightHelper(T,T.rightChiled(p).   
 if leftHeight>rightHeight   
 return leftHeight  
 else

return rightHeight