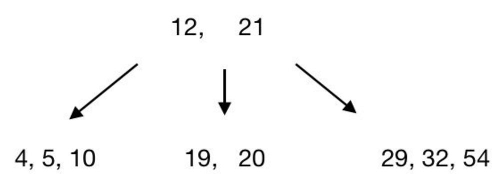
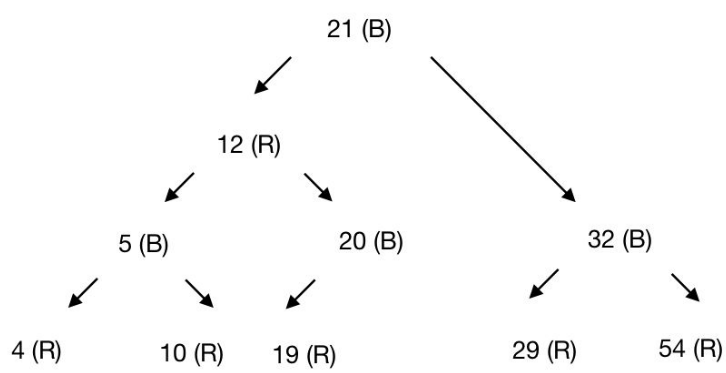
Solution - 4

Revision 2

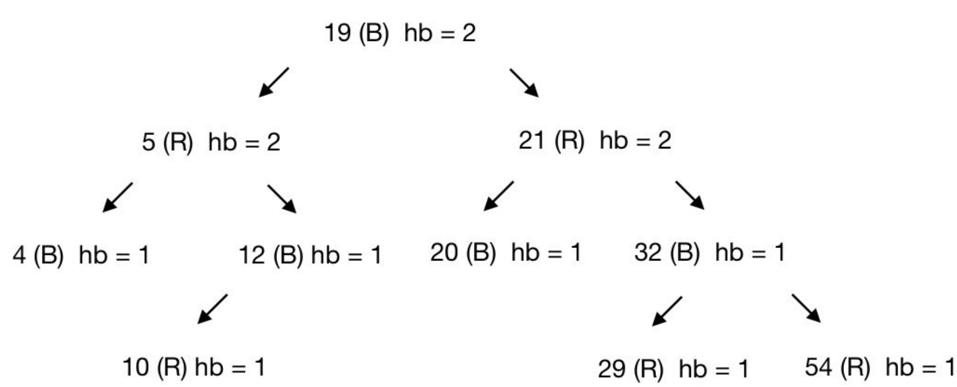
1. The solutions might be different when using different split rules.

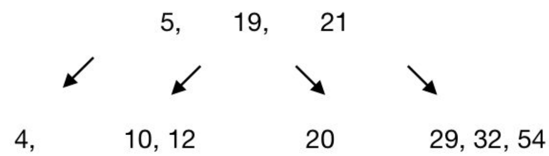
All reasonable solution will get points.





2.





3.

For maximal height of a 2-4 tree, we will be having one key per node, hence it will behave like a Binary Search Tree.

keys at level 0 = 1

keys at level 1 = 2

keys at level 2=4 and so on. . .

Adding total number of keys at each level we get a GP on solving which we will get the maximal height of the tree.

Hence, height = log2(n + 1) − 1

For minimal height of a 2-4 tree, we will be having three keys(maximum possible number) per node. keys at level 0 = 3

keys at level 1 = 3*(4)

keys at level 2=3*(42)and so on. . .

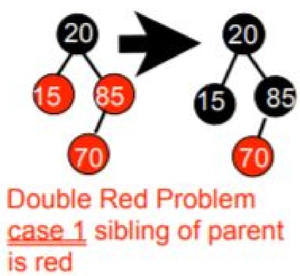
Hence, height = log4(n + 1) − 1

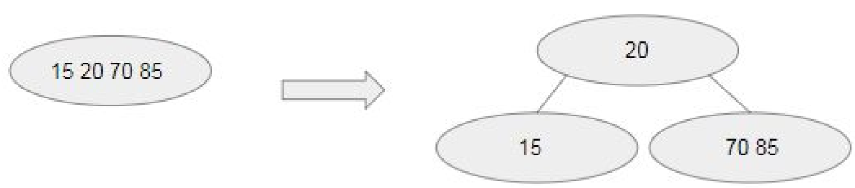
4.

The shortest simple path from any node x will be the black height of the tree with x as root(i.e., bh(x)). There could be many branches in the tree; each branch is a combination of red and black nodes. The longest simple path in any tree will be that path which has the total number of nodes = (Property 4) bh(x) + max possible number of red nodes. The maximum possible number of red nodes will be equal to the bh(x), as to satisfy the red-black property., for each red node, its children has to be clack (no two consecutive red nodes in a path). Hence the max height of the tree could be 2 * bh(x), twice the shortest simple path.

5.

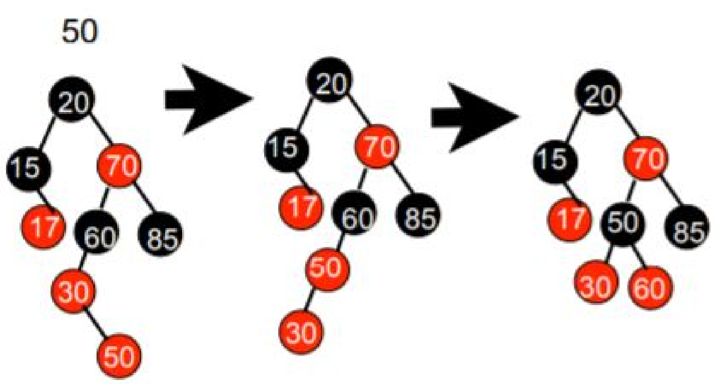
Case 1: Sibling of parent is red

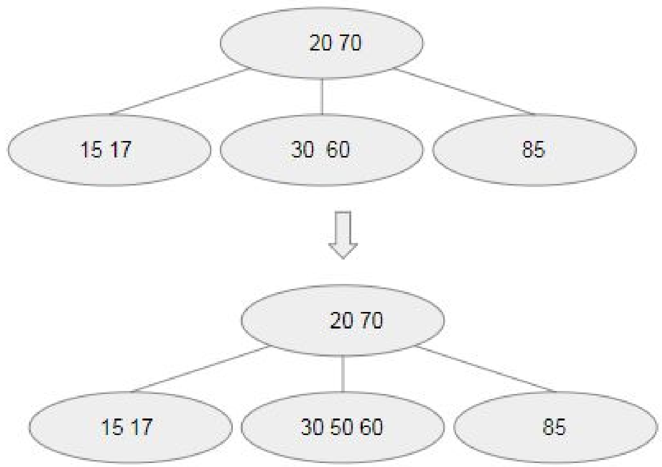




Case 2 and 3:

These two cases should be considered together, because case 2 leads to case 3.





6.

In this problem, we want to use the first fit heuristic to find the smallest drive that would fit.

We can take our USB drives and convert them into red-black tree. Since there are n drives, we can build this tree in O(n) time. We then loop through our images and for each image we perform the following steps:

1. We find the first drive that has enough space to hold the image, which takes O(log n) time.

2. We delete this node from our tree, which takes O(log n) time.

3. We then insert this node back into our tree in O(log n) time.

For each image we spend O(logn + logn + logn) = O(logn) time. For processing all m images, the overall time complexity is O(n + m log n) = O(mlogn).

Pseudocode:

```
StoreImage(drives, images):

tree = BUILD_RED_BLACK_TREE(drives)             // O(n)

for each image in images:

    drive = SEARCH_FIRST_FIT(tree, image.size)     // log(n)

    DELETE_FROM_TREE(tree,drive)                   // log(n)

    drive.size = drive.size - image.size
```

```
    ADD_TO_TREE(tree, drive)                    // log(n)
}
```

7.

based on the properties of BTS, we can pick the middle element(nums[(0+nums.length)/2]) from the sorted array and it is the root of tree. Based on this, we can use recursion to do the same thing on elements form 0 to mid,and (mid+1) to nums.length

```
public TreeNode sortedArrayToBST(int[] nums) {
        return recursion(nums,0,nums.length);
    }
    public TreeNode recursion(int[] nums,int i,int j){
        if(i>=j) return null;
        TreeNode root=new TreeNode(nums[(i+j)/2]);
        root.left=recursion(nums,i,(i+j)/2);
        root.right=recursion(nums,(i+j)/2+1,j);
        return root;
    }
```

8.

11. (a) Assume a binary tree has height of h, and for each node $max(size(t.left), size(t.right)) \leq (\frac{2}{3})size(t)$. Then we have $(\frac{2}{3})^h n \geq 1$. Hence, $h \leq log_{\frac{2}{3}} \frac{1}{n} = O(logn)$. This satisfies the definition of a balanced binary tree.

(b)

```
procedure LEFT-ROTATE(T, x)
    y = x.right
    temp = x.size
    if y.left = NIL then
        x.size = x.size - y.size
    else
        x.size = x.size - y.size + y.left.size
    y.size = temp
    LEFT-ROTATE(T, x) // From lecture
```

```
procedure RIGHT-ROTATE(T, x)
    y = x.left
    temp = x.size
    if y.right = NIL then
        x.size = x.size - y.size
    else
        x.size = x.size - y.size + y.right.size
    y.size = temp
    RIGHT-ROTATE(T, x) // From lecture
```

(c) Find the median node and apply rotation so that this node is the root of the tree. Then apply the same process recursively to its children.

```
procedure PERFECT-BALANCE(T)
    Balance-Tree(T.root)
```

```
procedure BALANCE-TREE(T)
    n = T.root
    medianValue = n. size/2
    medianNode = FindMedian(n, medianValue)
    parent = n.p
    while T.root ≠ medianNode do
        if T.left ≠ null and T.left.size ≥ medianValue then
```

```
            right-rotate(T,medianNode.p)
        else if T.right ≠ null and T.right.size ≥ medianValue then
            left-rotate(T,medianNode.p)
    if parent ≠ null then
        if parent.left == n then
            parent.left = medianNode
        else
            parent.right = medianNode
    medianNode.p = parent
    if medianNode.left ≠ null then
        Balance-Tree(medianNode.left)
    if medianNode.right ≠ null then
        Balance-Tree(medianNode.right)
    return
```

```
procedure Find-Median(Node root, int value)
    if root.left.size + 1 == value then
        return root
    else if root.left.size geq value then
        return FindMedian(n.left,value)
    else
        return FindMedian(root.right, value-1-root.left-size)
```

9.The following solution has been adapted from an anonymous student's solution.

Let the system be implemented using a two level data-structure of nested red black trees. The first level is a single red-black tree where the keys of the nodes are all the possible first characters of the names that are to be stored into the tree and each node, contains a pointer, say lvl2, to another red-black tree which is used to actually store all names (and other data) with identical starting letters.

Let the structure of a level 1 node and a level 2 node be as follows:

```
class level_1_node
Initialization(character)
    key = character
    level2 = NULL
    left = NULL
    right = NULL
    p = NULL
    color = RED
```

```
class level_2_node
Initialization(name, data)
    key = name
    left = NULL
    right = NULL
    p = NULL
    Color = RED
    data = data
```

Let *T* be the root node of the whole structure and is of a root of the level 1 red black tree.

1) Insert: when we are to insert a new name into the system, we first check if the first character of the new name already exists in the tree rooted at *T*.
    a) If it does exists, we get a pointer to that node, say level1_ptr by RB-SEARCH. Using level1_ptr we get the root of the level2 red black tree (which contains the all names with the same first character the name we wish to insert) and use RB-INSERT to insert the new name into the level2 red black tree.
    b) If it does not exists, we create first create a new red black tree with root, say level2_root and use RB-INSERT to insert the new name into the tree. Subsequently we create a new level1 tree node with the first character as the key and set the pointer lvl2 to level2_root. We then use RB-INSERT to insert the level1 node into the first level red-black tree

    Time complexity O(log n)

Insert Pseudo code:

```
Insert(T, name, data)
lvl2node = new level_2_node(name, data)
first_character = name[1]
level1_ptr = RB-SEARCH(T, first_character)
if level1_ptr == NULL
    lvl1node = new level_1_node(first_character)
    level1_ptr = lvl1node
    RB-INSERT(T, lvl1node)
if level1_ptr.level2 == NULL
    level1_ptr.level2 = lvl2node
else
    RB-INSERT(level1_ptr.level2, lvl2node)
```

2) Search: we first search for the node in the level 1 tree, lvl1node, corresponding to the first character of the name we are looking for, i.e. RB-SEARCH(T, first_character). Once we find the lvl1node, we search for the the name and get a pointer to the node containing the name and its corresponding data. RB-SEARCH(lvl1node.level2, name). Time complexity: O(log n)

Search Pseudo code:

```
Search(T, name)
first_character = name[1]
lvl1node = RB-SEARCH(T, first_character)
return RB-SEARCH(lvl1node.level2, name)
```

3) Delete: we first search for the node in the level1 tree, lvl1node, corresponding to the first character of the name we want to delete, i.e. RB-SEARCH(T, first_character). Once we find the lvl1node, we Delete the name from the tree rooted at lvl1node.level2 i.e. RB-DELETE(lvl1node.level2, name). Time complexity: O(log n)

Delete Pseudo code:

```
Delete(T, name)
    first_character = name[1]
    lvl1node = RB-SEARCH(T, first_character)
    RB-DELETE(lvl1node.level2, name)
```

4) Print names starting character: we first search for the node in the level1 tree, lvl1node, where the key is the equal to the given character, i.e. RB-SEARCH(T, first_character). Once we have found the lvl1node, we perform an inorder traversal on the tree rooted at lvl1node.level2. Time complexity: O(g + log n)

Character Pseudo code:

```
GET-FIRST-NAME-SEARCH (T, character)
    first_character = name[1]
    lvl1node = RB-SEARCH(T, first_character)
    name_list = init list
    INORDER-TRAVERSAL (lvl1node.level2, name_list )
    return name_list
```

```
INORDER-TRAVERSAL (node, name_list)
    if node == NULL
        return NULL
    INORDER-TRAVERSAL (node.left, name_list)
    name_list.insert_head(node.key)
    INORDER-TRAVERSAL (node.right, name_list)
```

10. To solve this question, let us use a Red-Black tree where each node is augmented with 2 pieces of extra information.

1) every node stores the size of its subtree in an attribute called **size** with default of 1

2) every node stores the sum of all rates of its subtree, i.e. the sum of the rates of all nodes present in the subtree including the node itself in an attribute called **rate_sum** default of the node's own rate

a) Insert(d, r): This is very similar to a regular RB tree insert which takes O(log n), where the key is the distance d. We create a new node, with attributes distance, rate, size and rate_sum; where distance = d, rate = r, size = 1 and rate_sum = r. Slight modifications need to be made to RB-INSERT to maintain our augmentations. As we search for the correct position to insert our new node

1) we increment the **size** attribute of every node we encounter along the path.

2) we add the new node's rate attribute to the **rate_sum** attribute of every node we encounter along the path

```
RB-INSERT(T, z)
    y = T.nil
    x = T.root
    while x ≠ T.nil
        y = x
        x.size = x.size + 1
        x.rate_sum = x.rate_sum + z.rate
        if z.key < x.key
            x = x.left
        else
            x = x.right
    z.p = y
    if y == T.nil
        T.root = z
    elseif z.key < y.key
        y.left = z
    else
        y.right = z
    z.left = T.nil
    z.right = T.nil
    z.color = RED
    RB-INSERT-FIXUP(T, z)
```

```
class RB-NODE
    Initialization(d, r)
        key = d
        distance = d
        rate = r
        size = 1
        rate_sum = rate

        p = null
        left = null
        right = null
        color = null
```

```
INSERT(d, r) //assuming T is accessible
    node = new RB-NODE(d, r)
    RB-INSERT(T, node)
```

We also need to modify the LEFT-ROTATE and RIGHT-ROTATE operations to maintain the augmentations

LEFT-ROTATE (T, x)

```
y = x.right
x.right = y.left
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p
if x.p == T.nil
    T.root = y
else if x == x.p.left
    x.p.left = y
else
    x.p.right = y
y.left = x
x.p = y
y.size = x.size
y.rate_sum = x.rate_sum
x.size = x.left.size + x.right.size + 1
x.rate_sum = x.left.rate_sum +
             x.right_rate_sum +
             x.rate
```

RIGHT-ROTATE (T, x)

```
y = x.left
x.left = y.right
if y.right ≠ T.nil
    y.right.p = x
y.p = x.p
if x.p == T.nil
    T.root = y
else if x == x.p.right
    x.p.right = y
else
    x.p.left = y
y.right = x
x.p = y
y.size = x.size
y.rate_sum = x.rate_sum
x.size = x.left.size + x.right.size + 1
x.rate_sum = x.left.rate_sum +
             x.right_rate_sum +
             x.rate
```

b) delete(k): we first, need to find the node that is at k^{th} distance away from the headquarters. We can do this in O(log n) using OS-SELECT(T, k) where T is the root. We can use OS-SELECT because the nodes of our tree contain the size augmentation that is needed by OS-SELECT. Let the k^{th} node be knode. After finding knode, we can find it's key and toll rate. We then search for the knode using it's key (similar to searching for a node in a binary tree) and decrement the size by 1 and the rate_sum by knode.rate for all nodes we encounter along the path as we search for the knode in O(log n). Then we invoke the subroutine RB-DELETE(T, knode) where T is the root node and knode is the node that we want to delete which takes O(log n).

c) The core logic behind this method is that; since every node maintains the size of it's subtree and also the sum of all rates in the subtree, we find the root of the subtree of interest (this subtree must contain both the lower bound and the upper bound nodes) and remove the branches of this subtree that we don't require (branches that contain keys not in the lower and upper bounds), we would obtain the correct number of number of nodes and the sum of rates of all nodes within the lower and upper bounds. Thus the average is the sum of all rates / number of nodes.

The method FIND-LEFT-INCORRECT and FIND-RIGHT-INCORRECT are used to find the sum of rates and sum of sizes of those nodes that are not within the Lower bound and Upper bound respectively.

The method GET-LOWEST-COMMON-ANCESTOR is used to get the root of the subtree that contains both the lower bound node and the upper bound node.

```
FIND-LEFT-INCORRECT (node, Lbound, minus_rate_sum, minus_size_sum)
    if node.key == Lbound
        if node.left != NULL
            minus_rate_sum = minus_rate_sum + node.left.rate_sum
            minus_size_sum = minus_size_sum + node.left.size

    else if node.key < Lbound
        minus_rate_sum = minus_rate_sum + node.rate
        minus_size_sum = minus_size_sum + 1

        if node.left != NULL
            minus_rate_sum = minus_rate_sum + node.left.rate_sum
            minus_size_sum = minus_size_sum + node.left.size

        FIND-LEFT-INCORRECT(node.right, Lbound, minus_rate_sum, minus_size_sum)

    else
        FIND-LEFT-INCORRECT(node.left, Lbound, minus_rate_sum, minus_size_sum)
```

```
FIND-RIGHT-INCORRECT (node, Ubound, minus_rate_sum, minus_size_sum)
    if node.key == Ubound
        if node.right != NULL
            minus_rate_sum = minus_rate_sum + node.right.rate_sum
            minus_size_sum = minus_size_sum + node.right.size

    else if node.key > Ubound
        minus_rate_sum = minus_rate_sum + node.rate
        minus_size_sum = minus_size_sum + 1

        if node.right != NULL
            minus_rate_sum = minus_rate_sum + node.right.rate_sum
            minus_size_sum = minus_size_sum + node.right.size

        FIND-RIGHT-INCORRECT(node.left, Ubound, minus_rate_sum, minus_size_sum)

    else
        FIND-RIGHT-INCORRECT(node.right, Ubound, minus_rate_sum, minus_size_sum)
```

```
GET-LOWEST-COMMON-ANCESTOR (T, Lbound, Ubound)
    pointer_node = T.root
    while True
        if pointer_node.key < Lbound and pointer_node.key < Ubound
            pointer_node = pointer_node.right
        else if pointer_node.key > Lbound and pointer_node.key > Ubound
            pointer_node = pointer_node.left
        else
            break
    return pointer_node
```

```
TOLL(i, j)                              //assuming the RB tree 'T' is accessible to this method
    inode = OS-SELECT(T, i)             // finding the ith node.                    //O(log n)
    jnode = OS-SELECT(T, j)             // finding the jth node.                    //O(log n)
    idist = inode.distance
    jdist = jnode.distance
    lca = GET-LOWEST-COMMON-ANCESTOR(T, idist, jdist)                               //O(log n)

    minus_rate_sum = 0
    minus_size_sum = 0

    FIND-LEFT-INCORRECT(lca, idist, minus_rate_sum, minus_size_sum)      //O(log n)
    FIND-RIGHT-INCORRECT(lca, jdist, minus_rate_sum, minus_size_sum)     //O(log n)

    total_rate_sum = lca.rate_sum – minus_rate_sum
    total_size_sum = lca.size – minus_size_sum

    return total_rate_sum / total_size_sum
```