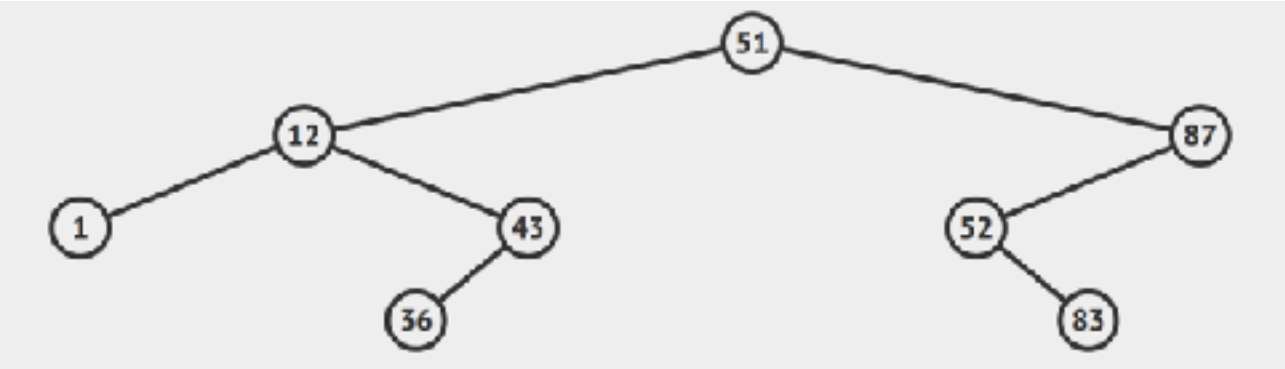
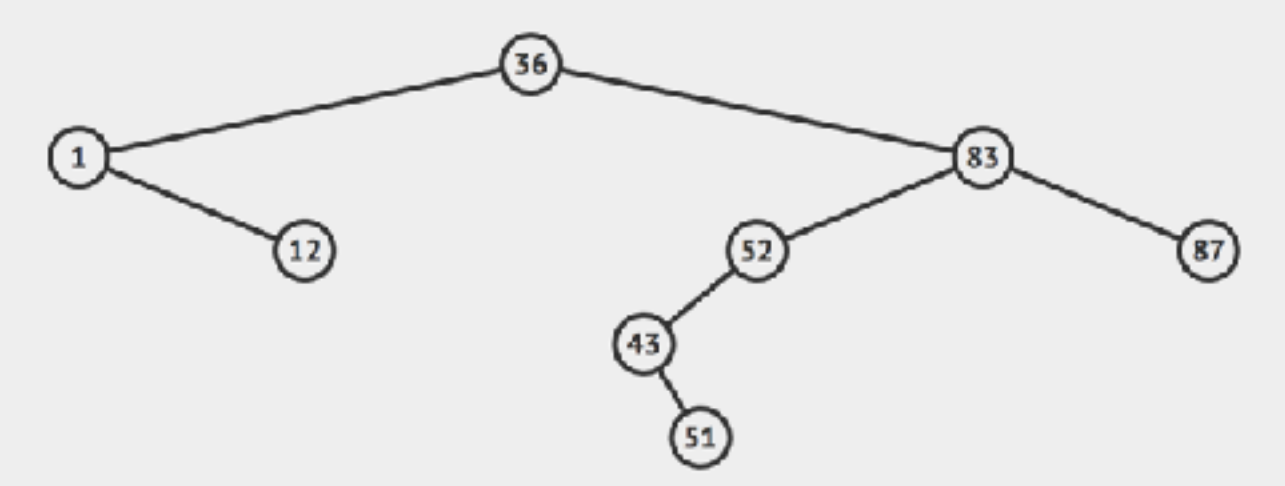
Exercise 6

(a)





(b)

Order 1: 51,12,1,43,36,87,52,83
Order 2: 51,87,52,83,12,1,43,36

Exercise 7

Pseudo Code:

Input: a BST T
Output: a BST T which stores the sum of odd keys in the subtree rooted at x

```
StoreSumOddKeys(T)
0: x ← root[T]
1: ModifiedPostOrderTreeWalk(x)
2: return T

ModifiedPostOrderTreeWalk(x)
1: if x != nil then
2:       PostOrderTreeWalk(x.left)
3:       PostOrderTreeWalk(x.right)
4:       sum = 0
5:       if x.key%2==1 then
6:                sum += x.key
7:       if x.left != nil then
8:                sum += x.left.s
9:       if x.right != nil then
10:               sum += x.right.s
11:      x.s ← sum
```

Proof of Correctness:
In order to proof correctness of the StoreSumOddKeys, we use a structural induction proof.

Proof: we must prove that (i) StoreSumOddKeys is correct for an empty tree $T(\varepsilon)$ and (ii) if StoreSumOddKeys is correct for all subtrees of T, then StoreSumOddKeys is correct for the tree T.

(i)Base: Suppose, the tree has nil nodes, it is empty => $P(\varepsilon)$ which is trivial. Nothing is done, and the tree returns correctly as how it was. Therefore, for this property StoreSumOddKeys is correct for T.

(ii)Inductive step: {IH} assume, T containing some arbitrary amount of nodes in its sub-structures L and R that correctly store a field s that stores the sum of all odd keys in the sub-tree rooted at x. Then T contains a root N. By the induction hypothesis, the left sub-structure(s) L, is traversed in StoreSumOddKeys in the order left, right and the root is reported, if some key at the root x of some substructure is odd or already contains a field s, then this value is correctly stored in the sum for all levels of the tree. Furthermore, by the induction hypothesis, sub-structure R is also recursively traversed and evaluating the subtrees rooted at x correctly in the same way. Thus, for the entire tree T, the total traversal will be, root N to the left subtree L and right subtree R on all levels of the tree. Since it traverses over the whole structure of tree T, we conclude by structural induction that the pseudocode is correct.

Running-Time Analysis:
Claim: The running time of the algorithm for StoreSumOddKeys is O(n).

Proof: Let T(n) denote the total running time of the StoreSumOddKeys algorithm with n nodes when it is called on the root. Let ModifiedPostOrderTreeWalk get invoked on the root and the root is not empty after a test T(0) = c for some constant c > 0. Then, suppose ModifiedPostOrderTreeWalk is getting called on some node n > 0 and its left subtree that has k nodes and the right subtree that contains n-k-1 nodes. Then the total time to run ModifiedPostOrderTreeWalk takes,

$$T(n) = T(k) + T(n-k-1) + d$$

For some positive constant d > 0 that indicates the upper bound of the body and extra operations of the ModifiedPostOrderTreeWalk algorithm. We can show that T(n) = O(n) by using the substitution method with a positive constant c such that $T(n) \leq c\,n$ for all sufficiently large n.

IH = assume $T(n) \leq (c+d)n+c$
Base: $T(0) \leq (c+d)*0 + c = c$

For some n > 0 we have,

$$\begin{aligned} T(n) &= T(k) + T(n-k-1) + d \quad \{ \text{IH} \} \\ &\leq ((c+d)k+c)+((c+d)(n-k-1)+c)+d \\ &= ((c+d)n+c)-(c+d)+c+d \\ &= (c+d)n + c \end{aligned}$$

Where, c is some constant > 0 and for d some constant > 0. Then ModifiedPostOrderTreeWalk is O(n). Since linear time runs slower than constant time the return statement of StoreSumOddKeys doesn't affect the asymptotic running time. Therefore, the total running time of StoreSumOddKeys will result in T(n) = O(n)

Exercise 8

(a)

Claim: prove that after a right-rotation on x(rotating the edge between left[x] and x) the height of T is still at most h+1.

Proof: let there be subtrees Tl rooted x.left and Tr rooted at x.right. We assume that the height of Tl is at least one larger than Tr. We will prove this claim by the use case distinction and the definition of height which states that the height of tree to subtree is defined by the longest path from a node x to a leaf. Since the height of the root node x is defined to be h+1, we know that of x rooted at x.left or x.right must have height h. Further, we know that the subtree Tl at x.left is said to be at least one larger height than Tr root at x.right. We therefore conclude by definition of height that Tr must be height h.

Now if we rotate T, then the new root becomes x.left with Tl and x as children. Let the root be k and its right child n in T. The height of T is then h(Tr)+1, where h() is the annotation to express height. To evaluate the height of Tl we must distinguish three cases:

Case 1 (Tl1 is of height h and h(Tl1) > h(Tl2)) => the height of T is at most h+1 because of Tl1 being larger and determining the height of the tree from the leave up to the root which is now x.left.

or

Case 2 (Tl1 is of height h and h(Tl1) < h(Tl2)) => the height of T is at most h+1 because of Tl2 being larger and determining the height of the tree from the leave up to the root which is now x.left.

or

Case 3 (Tl1 and Tl2 are of height h) => the height of T is at most h+1 because both Tl1 and Tl2 are of height h, Tr is now one larger and will be h+1.

Then by case distinction on case 1,2 and 3 we conclude that the claim, height of T is still at most h+1 in either case, holds.

(b)

The height of the BST T becomes less than h+1 when the height of Tl1 is larger than Tr by 2 and larger Tl1 > tl2. Then after a right rotation, Tl1 is the largest subtree T' from the root with a height h-1. If the root is included the height can at most be h. Therefore, the height of T' is h which is less than h+1.

Exercise 9

(a)

Field f cannot be maintained without affecting the asymptotic running time. An insertion of a node at the end of the tree will affect the values of its ancestors throughout the tree which takes $\Omega(n)$ time to update because we need to visit every node repeatedly, and thus affecting the asymptotic running time of O(log n).

(b)

Leaves = f[x] = 0 if x = null.
Otherwise, f[x] = max{f[left[x]], f[right[x]]}

Therefore, field f only depend on the contents of its children. This is why it can be maintained without affecting the asymptotic running time of insertion and deletion by the RB-tree augmentation theorem.

(c)

Assume, every node x has already an additional field c that stores the number of nodes in the subtree rooted at x. Then c[x] = 0 if x = null and otherwise c[x] = c[x.left] + c[x.right] + 1. Therefore, c[x] depends only on the contents of x and its children. According to the RB-tree augmentation theorem the field c can be maintained without the affecting of the asymptotic running time of insertion and deletion. Now, consider the additional field f which dependent on c. f[x] = 0 iff c[x] = 0. Otherwise $f[x] = \frac{key[x] + key[left] + key[right]}{c[x.left] + c[x.right] + 1}$ which is the average of the nodes in the subtree rooted at x. Furthermore, this shows that f[x] only depends on the contents of x and its children. And again, according to the RB-tree augmentation theorem the field c can be maintained without the affecting of the asymptotic running time of insertion and deletion.

Exercise 10

(a)

In order to search for the cheapest price, we can use a RB-tree and store the values pi =(xi,xi) in order to look up the lowest price in the tree. Since price=quality, the asymptotic complexity is maintained because the operation takes O(log n) time to look up the lowest key. Insert a new element in a red black tree also takes O(log n) time.

(b)

To find the cheapest quality, we augmenting a RB tree such that any key contains the price, x.key = p.price. Furthermore, an additional field *f* is stored and contains the highest quality in a subtree rooted at x and can be accessed in O(log n) time e.g., with a BST TreeSearch, because RB-tree is a balanced data structure the running time can be minimised to $\Theta(\log n)$ if we guarantee a small height.

Then, *f[x] = 0* when *x = nil* and *max{f[x], max{f[x.left], f[x.right]}* when *x!= nil.* Because *f[x]* only depends on the content of its children, *f* can be maintained without affecting the asymptotic running time of insertion and deletions by the RB-tree augmentation theorem.

(c)

Pseudo code:

Input: a RB tree S, where keys are price and an additional file *f* with the max quality in a subtree rooted at x and q ∈ N.

Output: an item with the lowest price that has at least a quality q ∈ N (or Nil if no such item exists)

CheapestQuality(S,q)

```
0: chQuality ← nil
1: x ← root[S]
2: if x = nil then
3:       return nil
4: if f[x] >= q then
5:       chQuality ← x
6: while x != nil do
7:       if f [x.left] >= q then
8:             chQuality ← x.left
9:                   x ← x.left
10:      else if f[x.right]>= q then
11:            chQuality ← x.right
12:                  x ← x.right
13:      else
14:            x ← nil
15: return chQuality
```

Proof of Correctness

LI: at the start of iteration i, the tree is returned in a correct order and stores an item with the lowest price that has at least a quality $q \in N$ (or Nil if no such item exists).

Init: prior to the first iteration of the loop, the root of the tree is examined. It is trivial that this hold since this is the only element and hence, the cheapest quality.

Main: in some iteration, the loop correctly recursively iterate over the nodes of the RB-tree without affecting the tree in any way. While the field f for the left subtrees of the tree is larger or equal to q, then the position of the x node is updated, the operation for the right subtrees of the tree is identical.

Termination: till the last ith iteration, the code will terminate when there are no elements of the tree anymore to evaluate and returns a nil value when there is no such thing as a cheapestQuality for some $q \in N$, or the algorithm terminates when the algorithm found and returns the correct cheapest quality value in the tree.

Running Time Analysis

Normally just like with BST, the running time of a recursive call, querying the tree takes O(n) time. Since the data structure used in this assignment, a RB-tree the data structure is balanced and therefore it could be O(log n) just like the same argument given in 10.b. Since the while loop goes recursively over the n elements the algorithm runs in O(log n) + O(1) for all the constant if, else, assignment and return operations. Since O(log n) runs slower than constant time the total running time will be O(log n).