12 : Binary Search Trees

12.1 : Binary Search tree:

* Property : → If y is in left subtree of x, then $key[y] \leq key[x]$
→ If y is in right subtree of x, then $key[y] \geq key[x]$

```
INORDER-TREE-WALK(x)
    if x ≠ NIL
        INORDER-TREE-WALK(x.left)
        print x.key
        INORDER-TREE-WALK(x.right).
```

Theorem : If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$ time.

→ Since INORDER-TREE-WALK visits all n nodes of the subtree, we have $T(n) = \Omega(n)$. It remains to show that $T(n) = O(n)$.

→ For $n > 0$, suppose that INORDER-TREE-WALK is called a node x whose left subtree has k nodes & whose right subtree has $n-k-1$ nodes.

→ The time to perform of INORDER-TREE-WALK(x), ~~exclusive of the time~~ is bounded by $T(n) \leq T(k) + T(n-k-1) + d$ for some constant $d > 0$ that reflects an upper case bound on the time to execute the body of INORDER-TREE-WALK(x), exclusive of the time spent in recursive calls.

→ We use the substitution method to show that $T(n) = O(n)$ by proving that $\boxed{T(n) \leq (c+d)n + c}$ For $n = 0$, we have $(c+d) \cdot 0 + c = c = T(0)$
$\forall\ n > 0$, we have

$T(n) \le T(k) + T(n-k-1) + d$

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

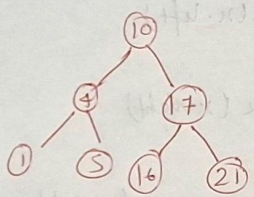
$= (c+d)n + c - (c+d) + c + d$

$\boxed{T(n) = (c+d)n + c}$ Hence proved.

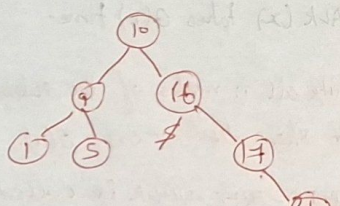
$T(k) = (c+d)k+c$

Exercise 12.1-1 Given keys < 1, 4, 5, 10, 16, 17, 21 > draw binary search trees of heights 2, 3, 4, 5 & 6.

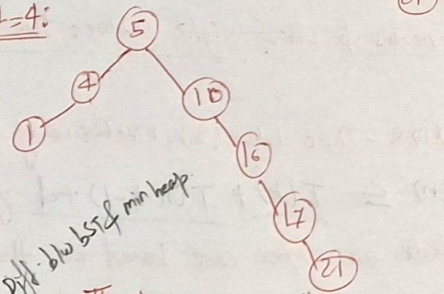
* height = 2:



* height = 3:

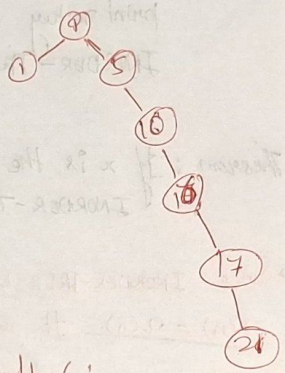


* height = 4:



Not a BST of min heap

* height = 5:



* height = 6:

12.1-2: → The binary-search-tree property guarantees that all nodes in the left subtree are smaller if all nodes in the right subtree are larger.

→ The min-heap property only guarantees the general child-larger-than-parent relation, but doesn't distinguish b/w left & right children. For this reason, the min-heap property can't be used to print out the keys in sorted order in linear time because we have no way to knowing which subtree contains the next smallest element.

12.1-3: Non-Recursive Inorder

```
INORDER-TREE-WALK(T)
    let S be an empty stack
    current = T.root
    done = 0
    while ! done
        if current != NIL
            PUSH(S, current)
            current = current.left
        else
            if !S.EMPTY()
                current = POP(S)
                print current
                current = current.right
            else done = 1
```

non-recursive INORDER.

12.1-4:

```
PREORDER-TREE-WALK(x)
    if x != NIL
        print x.key
        PREORDER-TREE-WALK(x.left)
        PREORDER-TREE-WALK(x.right)
```

```
POSTORDER-TREE-WALK(x)
    if x != NIL
        POSTORDER-TREE-WALK(x.left)
        POSTORDER-TREE-WALK(x.right)
        print x.key
```

12.1-5: Assume, for the sake of contradiction, that we can construct the binary search tree by comparison-based algorithm using less than $\Omega(n \lg n)$ time, since the inorder tree walk is $\Theta(n)$, then we can get the sorted elements in less than $\Omega(n \lg n)$ time, which contradicts the fact that sorting n elements takes $\Omega(n \lg n)$ time in the worst case.

12.2: Querying a binary search tree

```
* TREE-SEARCH (x,k)
① if x == NIL or k == x.key
②     return x
③ if k < x.key
④     return TREE-SEARCH(x.left, k)
⑤ else return TREE-SEARCH(x.right, k).
```

O(h), where h is the height of the tree.

```
* ITERATIVE-TREES-SEARCH(x,k)
① While x ≠ NIL && k ≠ x.key
②     if k < x.key
③         x = x.left
④     else x = x.right
⑤ return x
```

Minimum & Maximum

```
* TREE-MINIMUM(x)
① while x.left ≠ NIL
②     x = x.left
③ return x.
```

```
* TREE-MAXIMUM(x)
① while x.right ≠ NIL
②     x = x.right
③ return x
```

O(h)

Successor & predecessor

```
TREE-SUCCESSOR(x)
  if x.right ≠ NIL
      return TREE-MINIMUM(x.right)
  y = x.p
  while y ≠ NIL and x == y.right
      x = y
      y = y.p
  return y
```

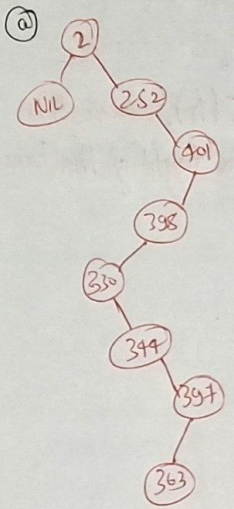
12.2-3

```
TREE-PREDECESSOR(x)
  if x.left ≠ NIL
      return TREE-MAXIMUM(x.left)
  y = x.p
  while y ≠ NIL and x == y.left
      x = y
      y = y.p
  return y
```

Exercise:

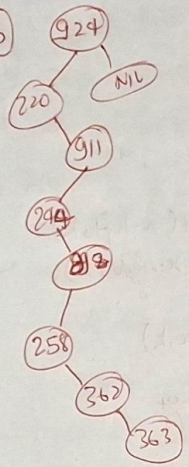
12.2-1 Soln

(a)



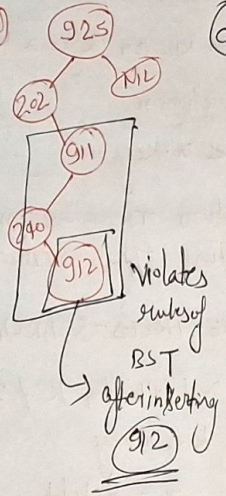
Correct

(b)



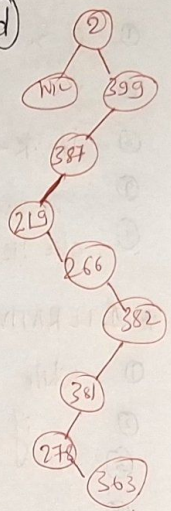
Correct

(c)



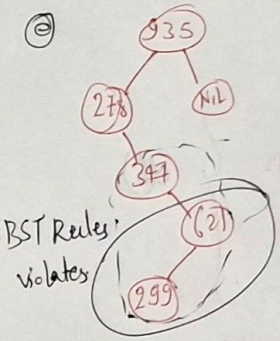
Not valid

(d)



correct

(e)



BST Rules violates

So, correct answer is option (d) & (e)

12.2-2

```
TREE-MINIMUM(x)
    if x.left ≠ NIL
        return TREE-MINIMUM(x.left)
    else return x
```

```
TREE-MAXIMUM(x)
    if x.right ≠ NIL
        return TREE-MAXIMUM(x.right)
    else return x.
```

12.2-4: Search for 9 in this tree. Then A={7}, B={5,8,9} & C={}. So, since 7>5 it breaks professor's claim.

12.2-5: Show that if a node in a binary search tree has 2 children, then its successor has no left child & its predecessor has no right child.

Suppose the node x has 2 children. Then it's successor is the minimum element of the BST rooted at x.right. If it had a left child then it would n't be the minimum element. So, it must not have a left child. Similarly, the predecessor must be the maximum element of the left subtree, so cannot have a right child.

12.2-6: First we establish that y must be an ancestor of x. If y weren't an ancestor of x, then let z denote the first common ancestor of x & y. By the binary-search-tree property, x<z<y, so y cannot be the successor of x. Next observe that y.left must be an ancestor of x because if it weren't, then y.right would be ancestor of x, implying that x≥y. Finally, suppose that y is not the lowest ancestor of x whose left child is also an ancestor of x. Let z denote this lowest ancestor. Then z must be in the left subtree of y, which implies z<y, contradicting the fact that y is successor of x.

12.2-7: To show this bound on the runtime, we will show that using this procedure, we traverse each edge twice. This will suffice because the number of edge in a tree is one less than the number of vertices.

Consider a vertex of a BST, say x. Then, we have that the edge b/w x.p & x gets used when successor is called on x.p & gets used again when it is called on the largest element in the subtree rooted at x. Since these are the only 2 times that edge can be used, apart from the initial finding of tree minimum. We have that the runtime is $O(n)$. We trivially get the runtime is $\Omega(n)$ because that is the size of the o/p.

12.2-8: Suppose x is the starting node & y is the ending node. The distance b/w x & y is at most 2h. & all the edges connecting the k nodes are visited twice, therefore it takes $O(k+h)$ time.

12.2-9: → If $x = y.left$, then calling successor on x will result in on iterations of the while loop. & so will return y.

→ If $x = y.right$, the while loop for calling predecessor will be run no times, & so y will be returned.

12.3: Insertion & deletion

* Insertion:

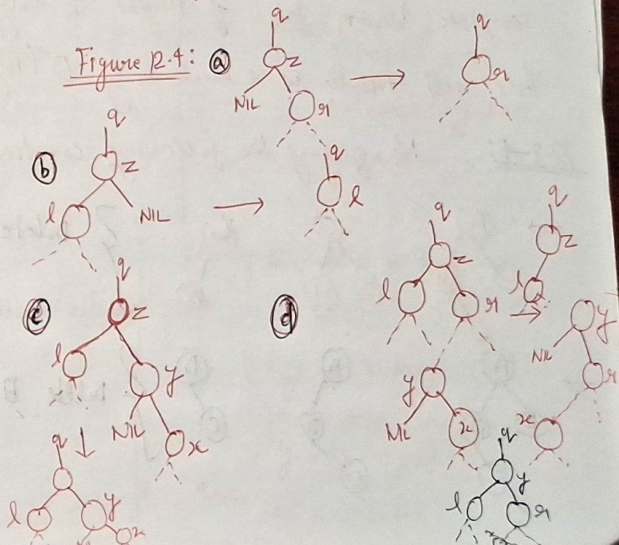
```
TREE-INSERT(T,z)
1  y = NIL
2  x = T.root
3  while x ≠ NIL
4      y = x
5      if z.key < x.key
6          x = x.left
7      else x = x.right
8  z.p = y
9  if y == NIL
10     T.root = z
11 elseif z.key < y.key
12     y.left = z
13 else y.right = z
```

* TRANSPLANT(T,u,v)

```
1  if u.p == NIL
2      T.root = v
3  elseif u = u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

* Deletion:

```
TREE-DELETE(T,z)
1  if z.left == NIL
2      TRANSPLANT(T, z, z.right)
3  elseif z.right == NIL
4      TRANSPLANT(T, z, z.left)
5  else y = TREE-MINIMUM(z.right)
6      if y.p ≠ z
7          TRANSPLANT(T, y, y.right)
8          y.right = z.right
9          y.right.p = y
10     TRANSPLANT(T, z, y)
11     y.left = z.left
12     y.left.p = y
```

Figure 12.4: 

Exercise: 12.3-1:

```
RECURSIVE-TREE-INSERT (T, z)
    if T.root == NIL
        T.root = z
    else INSERT (NIL, T.root, z)
```

```
INSERT(p, x, z)
    if x == NIL
        z.p = p
        if z.key < p.key
            p.left = z
        else p.right = z
    else if z.key ≥ x.key
        INSERT (x, x.left, z)
    Else INSERT (x, x.right, z)
```

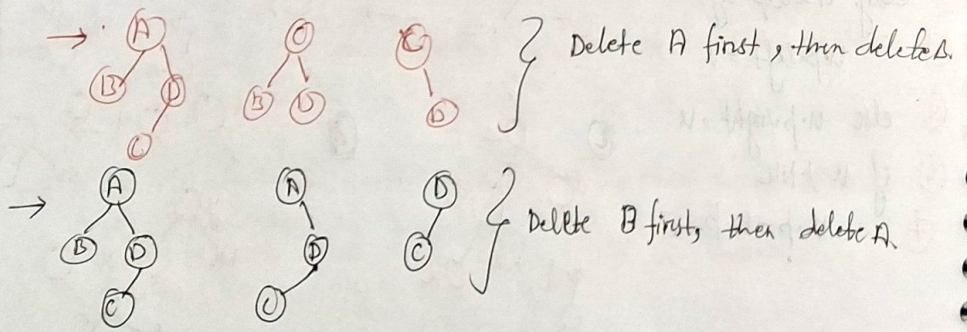
12.3-2: Number of nodes examined while searching also includes the node which is searched for, which isn't case when we inserted it.

12.3-3: → The worst-case is that the tree formed has height n because we were inserting them in already sorted order. This will result in a runtime of $\Theta(n^2)$.

→ The best case is that the tree formed is approximately balanced. This will mean that the height doesn't exceed $O(\lg n)$.

→ Note that it can't have an smaller height, because a complete binary tree of height h only has $\Theta(2^h)$ elements. This will result in a runtime of $O(n \lg n)$.

12.3-4: No, giving the following counterexample



12.3-5

```
* TREE-SEARCH (x, k)
    if x == NIL or k == x.key
        return x
    if k < x.key
        return TREE-SEARCH (x.left, k)
    else return TREE-SEARCH (x.right, k)

* PARENT (T, x)
    if x == T.root
        return NIL
    y = TREE-MAXIMUM (x).succ
    if y == NIL
        y = T.root
    else
        if y.left == x
            return y
        y = y.left
        while y.right ≠ x
            y = y.right
        return y

* TREE-PREDECESSOR (T, x)
    if x.left ≠ NIL
        return TREE-MAXIMUM (x.left)
    y = T.root
    pred = NIL
    while y ≠ NIL
        if y.key == x.key
            break
        if y.key < x.key
            pred = y
            y = y.right
        else
            y = y.left
    return pred
```

```
* INSERT (T, z)
    y = NIL
    x = T.root
    pred = NIL
    while x ≠ NIL
        y = x
        if z.key < x.key
            x = x.left
        else
            pred = x
            x = x.right
    if y == NIL
        T.root = z
        z.succ = NIL
    else if z.key < y.key
        y.left = z
        z.succ = y
        if pred ≠ NIL
            pred.succ = z
    else
        y.right = z
        z.succ = y.succ
        y.succ = z
```

* DELETE(T, z)

```
① ỹ = TREE-PREDECESSOR(T, z)
② ỹ.succ = z.succ
③ if z.left == NIL
④     TRANSPLANT(T, z, z.right)
⑤ else if z.right == NIL
⑥     TRANSPLANT(T, z, z.left)
⑦ else
⑧     y = TREE-MINIMUM(z.right)
⑨     if PARENT(T, y) ≠ z
⑩         TRANSPLANT(T, y, y.right)
⑪         y.right = z.right
⑫     TRANSPLANT(T, z, y)
⑬     y.left = z.left.
```

Therefore, all these five algorithms are still O(h) despite the increase in the hidden constant factor.

12.3-6: Update line 5 so that y is set equal to TREE-MAXIMUM(z.left) & line 6-12 so that every y.left & z.left is replace with y.right & z.right and vice-versa.

To implement the fair strategy, we could randomly decide each time TREE-DELETE is called whether or not to use the predecessor or successor.

13: Red-Black Tree

O(lg n) times in the worst case

13.1: Properties of red-black trees

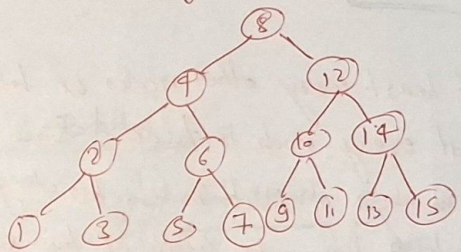
Exercise:

① Every node is either red or black
② The root is black
③ Every leaf (NIL) is black
④ If a node is red, then both its children are black
⑤ For each node, all simple paths from the node to descendant leaves contain the same number of black nodes. "Black-height"

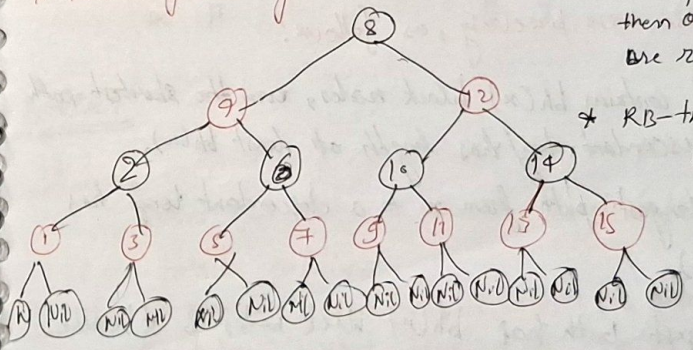
All downward simple paths from a node to a leaf have the same number of blacks.

Exercise:

13.1-1: * complete binary tree of height = 3:



* RB-tree of black-height = 2



* RB-tree of black-height = 3 then on Node ④ & ⑫ are red others are black

* RB-tree of black-height = 4 All nodes are blacks.

13.1-5: Suppose we have the longest simple path $(a_1, a_2 \dots a_s)$ & the shortest simple path $(b_1, b_2 \dots b_t)$. Then, by property 5 we know they have equal numbers of black nodes. By property 4, we know that neither contains a repeated red node. This tells us that at most $\lfloor \frac{s-1}{2} \rfloor$ of the nodes in the longest path are red. This means that at least $\lceil \frac{s+1}{2} \rceil$ are black. So, $t \geq \lceil \frac{s+1}{2} \rceil$.

Therefore, if by way of contradiction, we had that $s > t \cdot 2$, then $t \geq \lceil \frac{s+1}{2} \rceil \geq \lceil \frac{2t+2}{2} \rceil = t+1$

results are contradiction.

OR

In the longest path, at least every other nodes is black. In shortest path, at most every node is black. Since the two paths contain equal numbers of black nodes, the length of the longest path is at most 2x twice the length of the shortest path.

We can say this more precisely, as follows:

→ Since every path contains $bh(x)$ black nodes, even the shortest path from x to a descendant leaf has length at least $bh(x)$.

→ By definition, the longest path from x to a descendant leaf has length $height(x)$.

→ Since the longest path has $bh(x)$ black nodes & at least half the nodes on the longest path are black (by property 4),

$bh(x) \geq height(x)/2$,

So,

length of the longest path = $height(x) \leq 2 \cdot bh(x) \leq$ twice length of the shortest path

13.1-6:

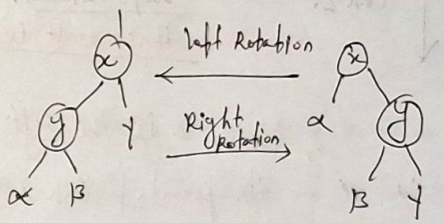
* The largest is a path with half black nodes & half red nodes, which has $2^{2k} - 1$ internal nodes.

* The smallest is a path with all black nodes, which has $2^k - 1$ internal nodes.

13.1-7:

* The largest ratio is 2, each black node has 2 red children
* The smallest ratio is 0.

13.2 : Rotation



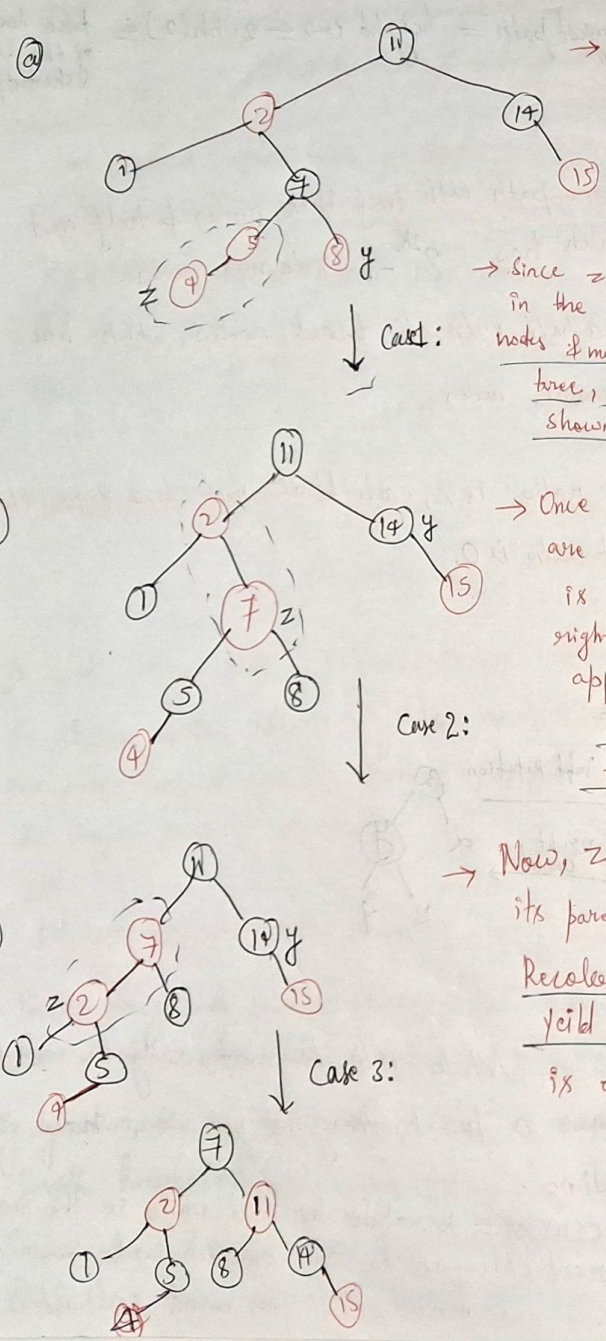
Exercise:

13.2-2: Every node can rotate with its parent, only the root does not have a parent, therefore there are $n-1$ possible rotations.

13.2-5: We can use $O(n)$ calls to rotate the node which is the root in T_2 to T_1's root, then use the same operation in the two subtrees. There are n nodes, ∴ upper bound is $O(n^2)$

13.3: INSERTION

$O(\lg n)$ = height of RB tree on n nodes.



→ A z node ④ after insertion. Because both z & its parent z.p are red, a violation of property 4 occurs.

→ Since z's uncle y is red, case 1 in the code applies. We recolor nodes & move the pointer z up the tree, resulting in the tree shown in ⓑ

→ Once again, z & its parent are both red, but z's uncle y is black. Since z is the right child of z.p, case 2 applies. We perform a left rotation, & the tree that result is shown in ⓒ

→ Now, z is the left child of its parent, & case 3 applies. Recoloring & right rotation yield the tree in ⓓ, which is a legal-red-black tree.

Case 1: z's uncle y is red

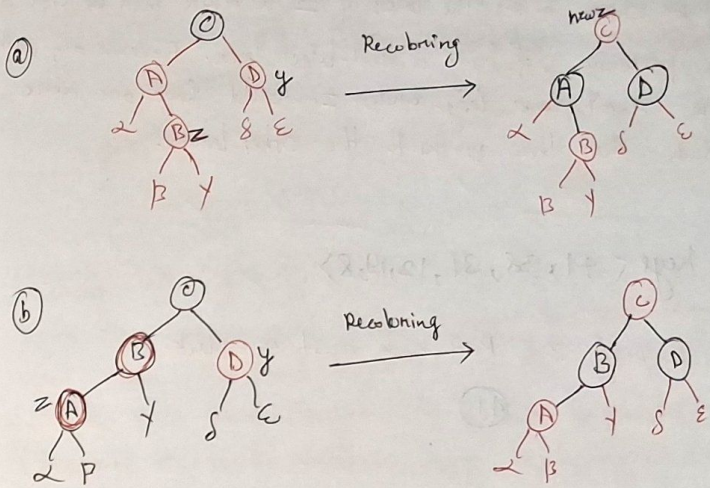
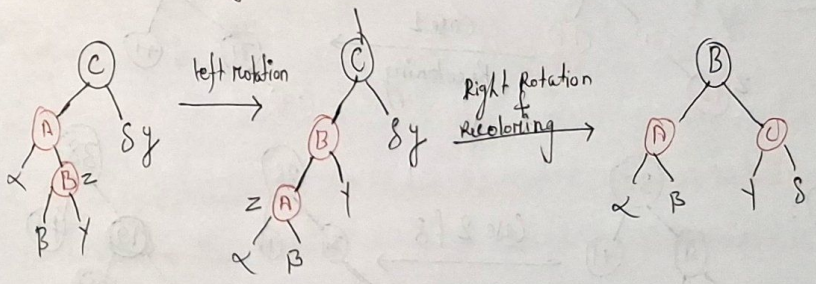


Figure 13.5: Case 1 of the procedure RB-INSERT-FIXUP. Property 4 is violated, since z & its parent z.p are both red. We take the same action whether ⓐ z is a right child or ⓑ z is a left child. Each of the subtrees α, β, γ, δ & ε has a black root & each has the same black-height.

Case 2: z's uncle y is black & z is a right child

Case 3: z's uncle y is black & z is a left child.

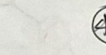


Exercise:

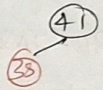
13.3-1: If we chage to set the color of z to black then we would be violating property 5 of being a red-black tree. Because any path from the root to a leaf under z would have one more black node than the paths to the other leaves.

13.3-2: Keys $\langle 41, 38, 31, 12, 19, 8 \rangle$

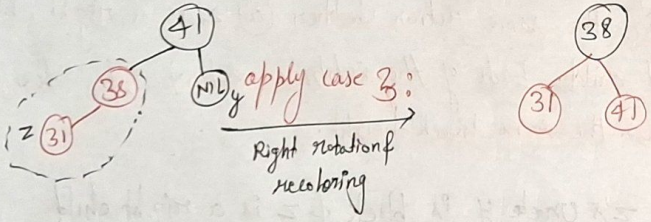
Insert 41: Property 2: Root node must be black



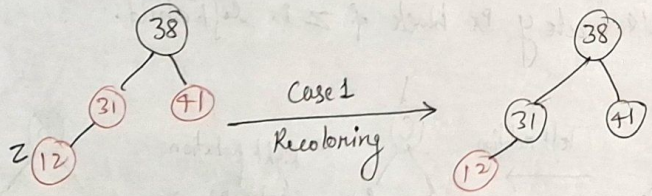
Insert 38:



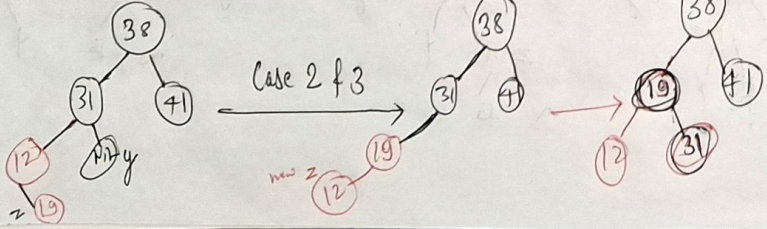
Insert 31:



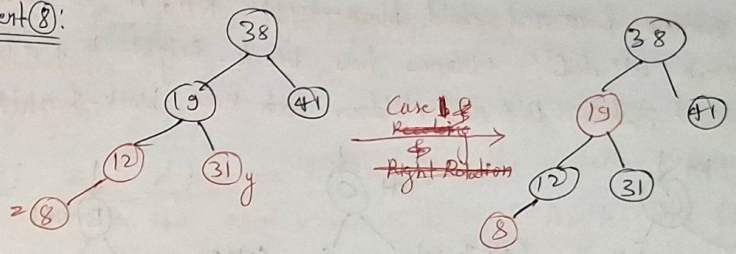
Insert 12:



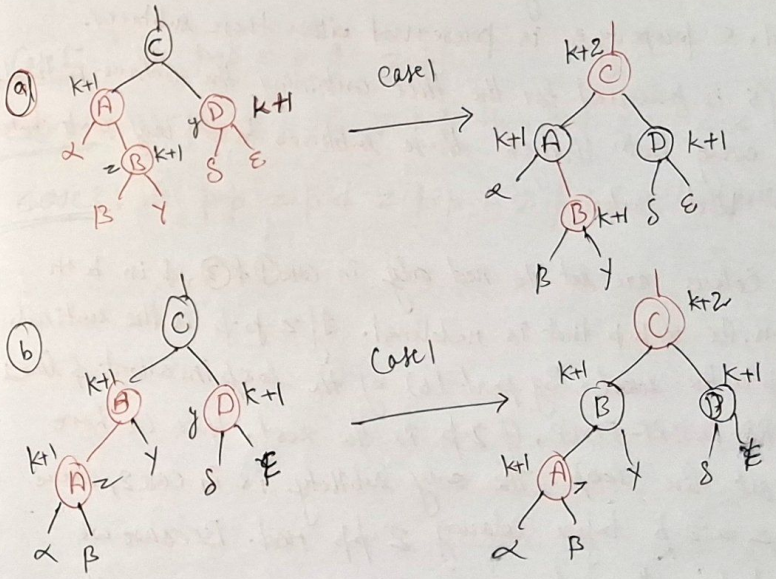
Insert 19:



Insert 8:

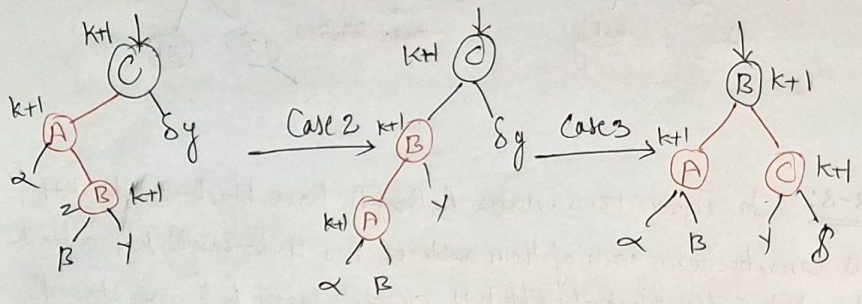


13.3-3: In Figure 13.5, nodes A, B, & D have black-height k+1 in all cases, because each of their subtrees has black-height k of a black root. Node C has black-height k+1 on the left (because its red children have black-height k+1) & black-height k+2 on the right (because its black children have black-height k+1).



In Figure 13.6, nodes A, B & C have black-height k+1 in all cases. At left & in the middle, each of A's & B's subtrees has black-height k of a black root, while c has one such

subtree of a red child black-height k+1, At the right, each A's & C's subtrees has black-height k & a black root, while B's red children each have black-height k+1.



Property 5 is preserved by the transformations. We have shown above that the black-height is well-defined within the subtrees pictured, so property 3 is preserved within those subtrees.
Property 5 is preserved for the three containing the subtree pictured, because every path through these subtrees to a leaf contributes k+2 black nodes.

B.3-4: Colors are set to red only in case ① & ③, & in both situations, it's z.p.p that is recolored. If z.p.p is the sentinel, then z.p is the root. By part (b) of the loop invariant & line 1 of the RB-INSERT-FIXUP, if z.p is the root, then we have dropped out the loop. The only subtlety is in case 2, where we set z = z.p before coloring z.p.p red. Because we rotate before the coloring, the identity of z.p.p is the same before & after case 2, so there's no problem.

B.3-5: Case 1: z & z.p.p are RED, if the loop terminates, then z could not be the root, thus z is RED after the fix-up.

Case 2: z & z.p.p are RED, & after the rotation z.p could not be the root, thus z.p is RED after the fixup.

Case 3: z is RED & z could not be the root, thus z is RED after the fix up.

Therefore, there is always at least one red node.

B.3-6: Use stack to record the path to the inserted node, then parent is the top element in the stack.

Case 1: We pop z.p & z.pp.

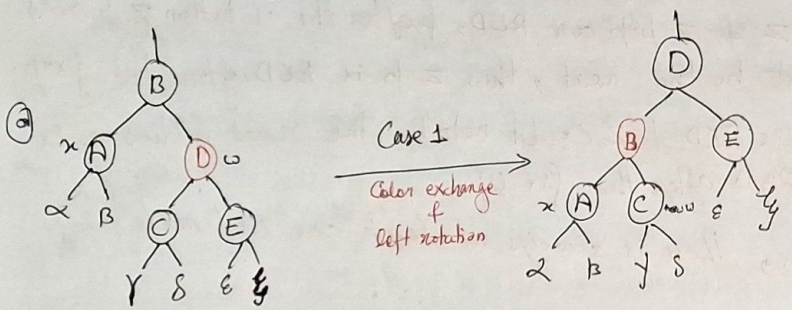
Case 2: : We pop z.p & z.p.p then push z.p.p & z

Case 3: We pop z.p & z.p.p & z.p.p.p & z.p.p.p.p, then push z.p.

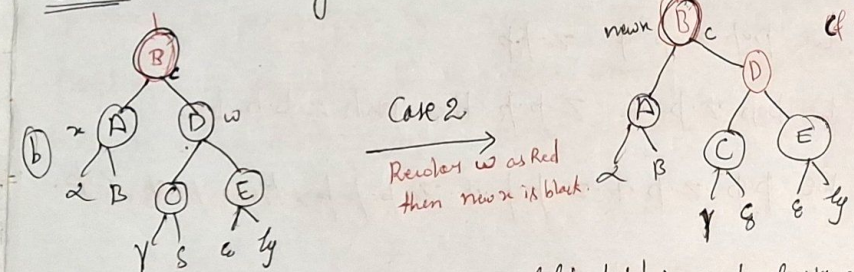
13.4 : Deletion $O(\lg n)$

Figure 13.7

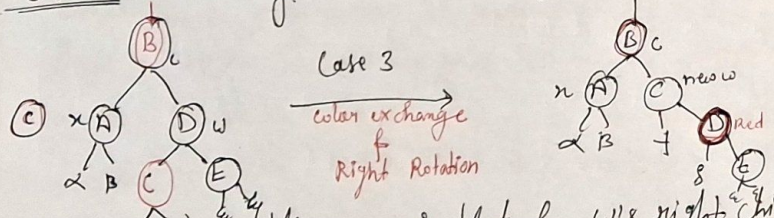
Case 1: x's sibling w is red



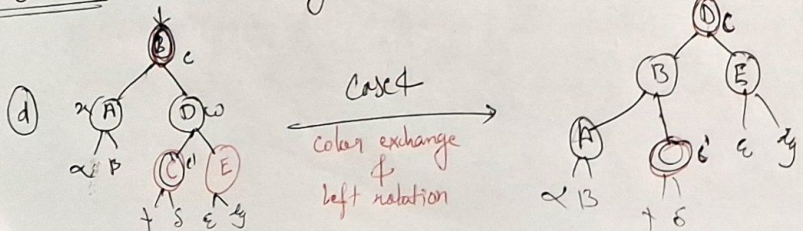
Case 2: x's sibling w is black, & both of w's children are black



Case 3: x's sibling w is black, w's left child is red, & w's right child is Black



Case 4: x's sibling w is black, & w's right child is red.



Exercise:

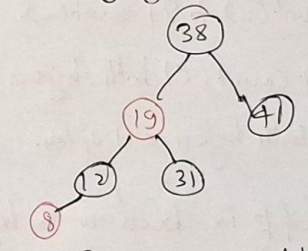
13.4-1:
- **Case 1:** transform to 2, 3, 4
- **Case 2:** If terminates, the root of the subtree (the new x) is set to black.
- **Case 3:** transform to 4
- **Case 4:** the root (the new x) is set to black.

13.4-2: Suppose that both x & x.p are red in RB-DELETE. This can only happen in the else-case of line 9. Since we are deleting from a red-black tree, the other child of y.p which becomes x's sibling in the call to RB-TRANSPLANT on line 14 must be black, so x is the only child of x.p which is red. The while-loop condition of RB-DELETE-FIXUP(T, x) is immediately violated & we simple set x.color = black, restoring property 4.

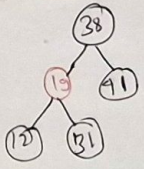
13.4-3: Deletion sequence $\langle 8, 12, 19, 31, 38, 41 \rangle$

Insertion keys $\langle 41, 38, 31, 12, 19, 8 \rangle$

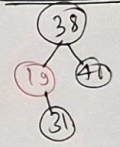
- Initially Inserted key of RB-tree will be



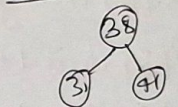
Delete (8):



Delete (12):



Delete (19):



Delete (31): 38, 41

Delete (38): 41

Delete (41): NIL

13.4-4 → When the node y in RB-DELETE has no children, the node $x = T.nil$, so we'll examine the line 2 of RB-DELETE-FIXUP. [if $x == x.p.left$]

→ When the root node is deleted, $x = T.nil$ & the root at this time is x, & the line 23 of RB-DELETE-FIXUP will draw x to black. [$x.color = BLACK$].

13.4-5: Our count will include the root (if it is black)

Case 1: For each subtree, it is 2 both before & after.

Case 2:
- For α & β, it is $1 + count(c)$ in both cases
- For the rest of the subtrees, it is from $2 + count(c)$ to $1 + count(c)$.

This decrease in the count for the other subtrees is handled by them having x represent an additional black.

Case 3:
- For ε & ζ, it is $2 + count(c)$ both before & after
- For all other subtrees, it is $1 + count(c)$ both before & after

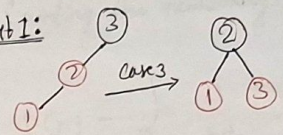
Case 4:
- For α & β, it is from $1 + count(c)$ to $2 + count(c)$.
- For γ & δ, it is $1 + count(c) + count(c')$ both before & after
- For ε & ζ, it is $1 + count(c)$ both before and after.

The increase in the count for α & β is because x before indicated an extra black.

13.4-7: No, the RB tree will not necessarily be the same.

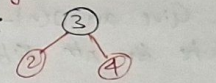
Ex:- Initial:

Insert 1:

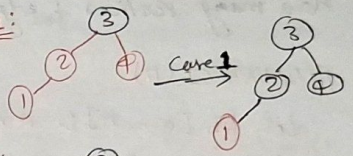


Delete 1:

Ex:- Initial:



Insert 1:



Delete 1: