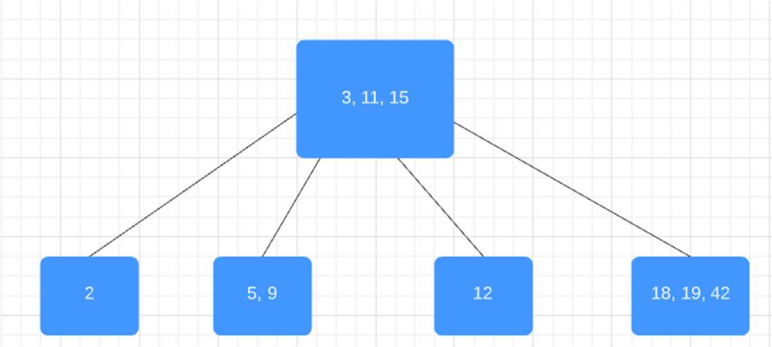
Brandon Vo

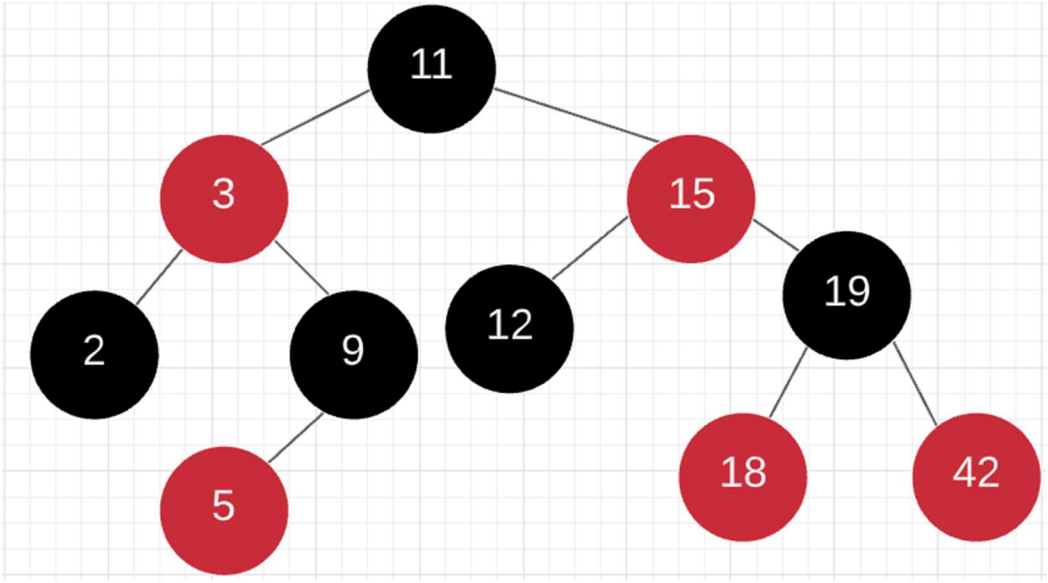
1. Insert, in order, the elements 2, 19, 15, 3, 42, 11, 9, 18, 12, 5 into a 2 − 3 − 4 tree.

Convert the 2 − 3 − 4 tree you constructed into a red-black binary search tree. Indicate the color of each node by using letters B (black) and R (red). You may omit representing the NIL nodes.



Converting to a Red-Black Tree:

- (3, 11, 15) is a 4-link node that splits. 11 becomes the black parent. 3 and 15 become red children.
- (2) is a 2-node that becomes a black child of 3.
- (5, 9) – is a 3-node that splits into one red and one black node. 5 will be the red child while 9 becomes the black parent.
- (12)- 2-key node that becomes a black child of 15
- (18, 19, 42) – 4-node that splits. 19 becomes the black parent. 18 and 42 becomes the red children of 19. 19 is also the child node of red 15.



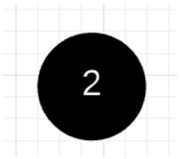
Brandon Vo

2. Insert in order the keys 2, 19, 15, 3, 42, 11, 9, 18, 12, 5 into a red-black BST tree. Show your tree after every insertion. Indicate the color of each node by letters B (black) and R (red). You may omit representing the NIL nodes.

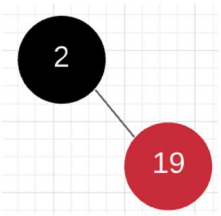
After that, convert the red-black BST you have constructed into a 2 − 3 − 4 tree, using the steps we discussed in class.

Red-Black tree:

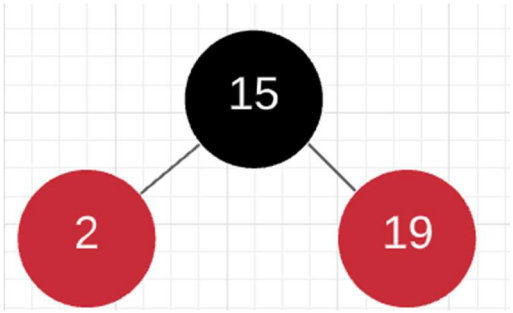
Inserting 2: Root will be black



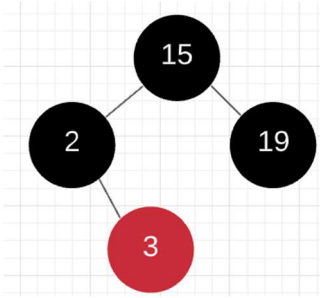
Inserting 19: To maintain equal black height, the node will be red.



Inserting 15: Insert 15 as the left child of 2. This creates a case 2 violation which must be solved by left rotation which leads to a case 3 violation. This case violation is resolved by left rotating and making 15 the black parent. 2 will be recolored as red.

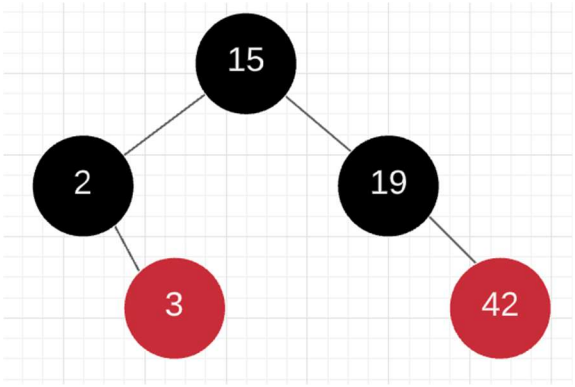


Inserting 3: Inserting 3 as the right child of 2. This creates a case 1 violation which is resolved by recoloring 2 and its sibling, 19, to be black.



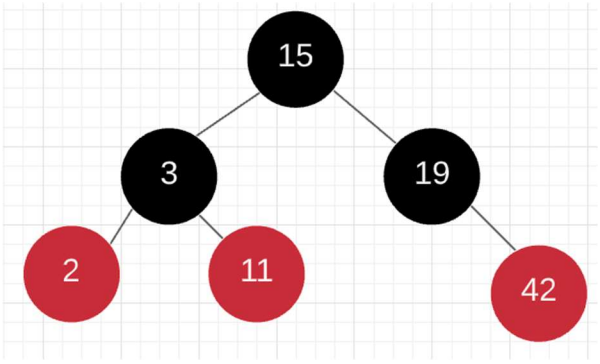
Brandon Vo

Inserting 42: Inserting 42 as the right child of 19. Red-Black property maintained.

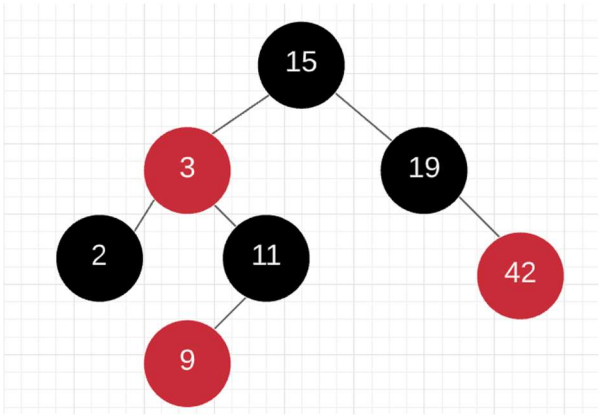


Inserting 11: Inserting 11 as the right child of 3. This becomes a case 3 Red-Red violation: Will rotate the sub-tree left

- Case 3 violation: Sibling of the 19 parent is also black. Must right rotate and recolor tomaake 19 the black parent of red children 3 and 19.

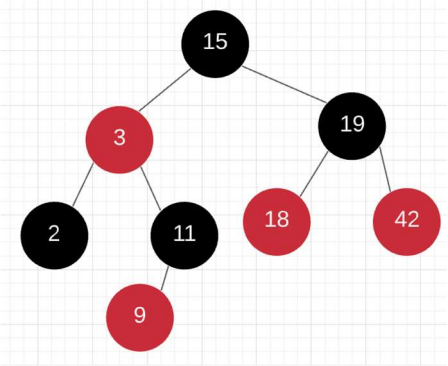


Inserting 9: Must insert 9 to be the left child of 11. This will be a case 1 violation which will be resolved by recoloring 2 and 11 to be red and 3 will be recolored black.

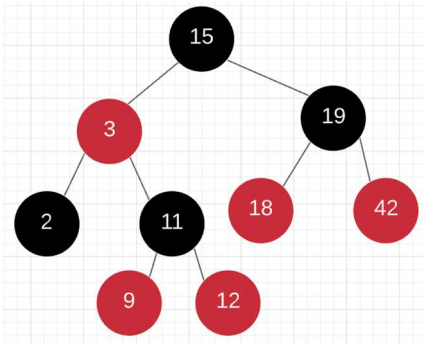


Inserting 18: Insert 18 as the left child of 19. Heap property maintained.

Brandon Vo



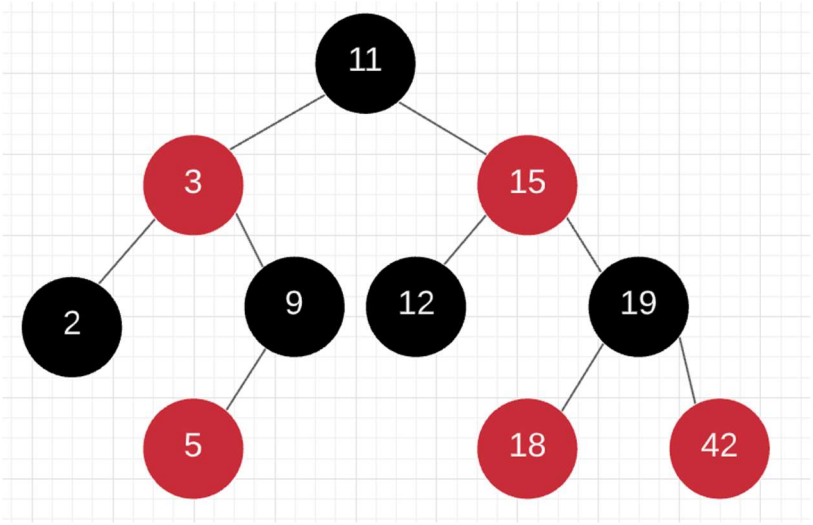
Inserting 12: Inserting 12 to be the right child of 11. No Red-Red violation found.



Inserting 5: Inserting 5 to be the left child of 9. This leads to a case 1 violation resolved by recoloring 9 and 12 to be black while 11 becomes red.

This leads to a case 2 violation between 3 and 11. This is resolved by left rotating and changing it to a case 3 violation.

The case 3 violation will mean that we will have to right rotate 11 to become the new parent of 3 and 15. 11 becomes the black node while 15 is recolored to become the red node.



Brandon Vo

3. (5 points) What is the minimum and maximum height of a 2 − 3 − 4 tree of n nodes. Justify your answer.

- Since 2-3-4 tree can have up to n nodes that hold 3 keys and 4 links, the minimum height would be a tree where all nodes have 4 links and 3 keys.
    - 4 links is the same as holding 4 nodes so the minimum height would be $\log_4 n$
- The maximum height would be a tree that holds only nodes with 2 links and 1 key which would be a standard binary tree
    - The maximum height would be $\log_2 n$

$$\lfloor \log_4 n \rfloor \leq height \leq \lfloor \log_2 n \rfloor$$

Brandon Vo

4. Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from x to a descendant leaf.

In a Red-Black tree, the black-height must be the same number for all paths which means the longest path will be determined by how many red nodes we can add for each black node in one path. This also means that the black-height is the minimum number of nodes we must travel through on a single path.

- Shortest path: All black nodes. Can be at minimum the height of the tree as bh.

The longest path that's allowed would be a path that alternates between red and black nodes for every step. For the longest path, the number of red nodes and black nodes would be nearly equal, essentially making the height twice that of the black height of that path which gives us 2*bh.

- Maximum height: Alternating between red nodes and black nodes. This would allow the maximum height to be twice the size of the black-height: 2*bh.

The shortest path can go as low as bh. The longest path can go as high as 2*bh which is at most twice that of the shortest path.

Brandon Vo

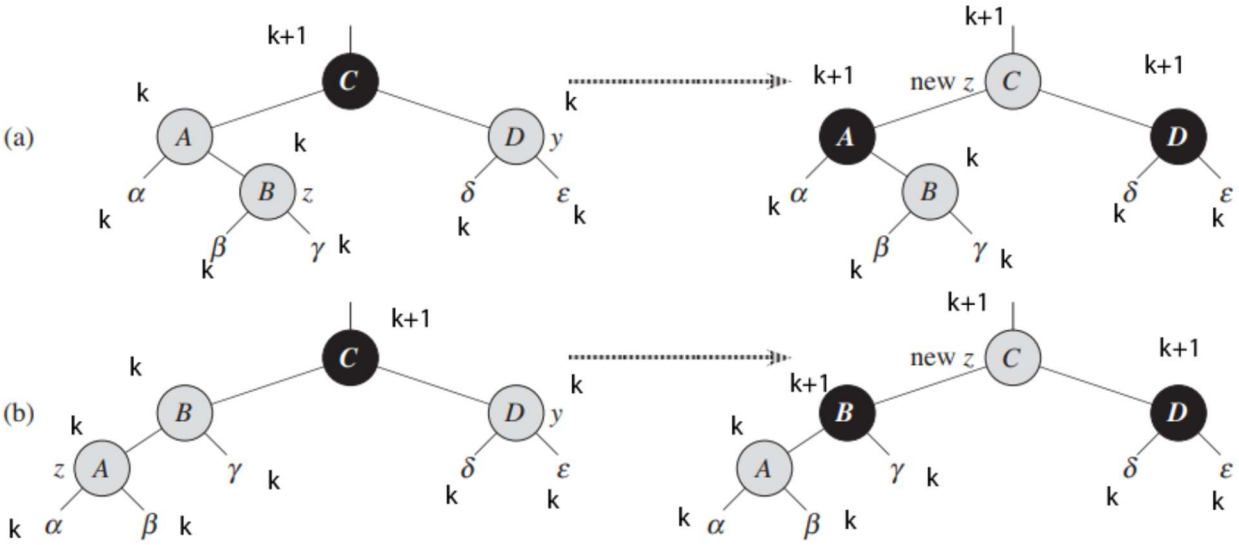
5. CLRS page 322, question 13.3-4

13.3-3

Suppose that the black-height of each of the subtrees $\alpha, \beta, \gamma, \delta, \varepsilon$ in Figures 13.5 and 13.6 is k. Label each node in each figure with its black-height to verify that the indicated transformation preserves property

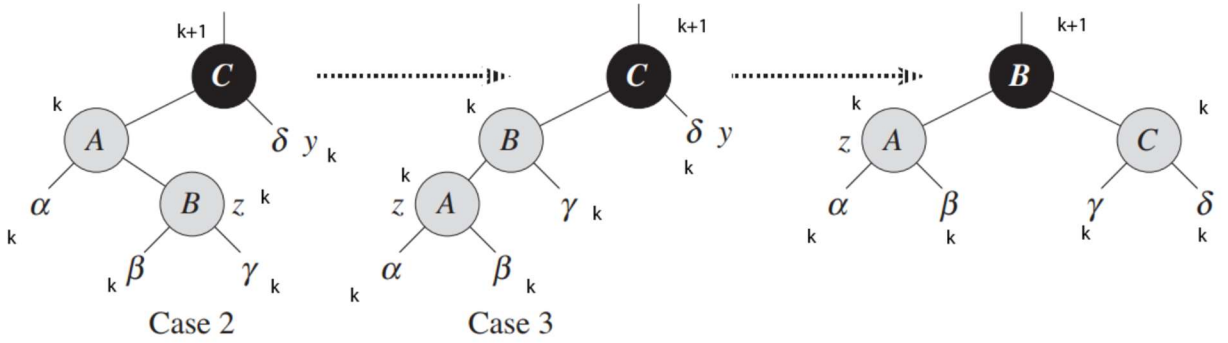
Figure 13.5

5.



- $\alpha, \beta, \gamma, \delta, \varepsilon$ all have a black height of k before and after the transformations occur. This shows that the black-height property is preserved during each transformation.

Figure 13.6



- Same as before, $\alpha, \beta, \gamma, \delta, \varepsilon$will retain the black height of k for both case 2 and case 3 rotations, proving that property 5 of the Red-Black trees is being maintained.

13.3-4

Professor Teach is concerned that RB-INSERT-FIXUP might set T.nil.color to RED, in which case the test in line 1 would not cause the loop to terminate when ζ is the root. Show that the professor's concern is unfounded by arguing that RBINSERT-FIXUP never sets T.nil.color to RED.

Brandon Vo

```
RB-INSERT-FIXUP(T,z)
1 while z.p.color == RED
2       if z.p = = z.p.p.left
3               y = z.p.p.right
4               if y.color == RED
5                       z.p.color = BLACK // case 1
6                       y.color = BLACK // case 1
7                       z.p.p.color = RED // case 1
8                       z = z.p.p // case 1
9               else if z == z.p.right
10                              z = z.p // case 2
11                              LEFT-ROTATE(T,z) // case 2
12                      z.p.color = BLACK // case 3
13                      z.p.p.color = RED // case 3
14                      RIGHT-ROTATE(T, z.p.p) // case 3
15              else (same as then clause with “right” and “left” exchanged)
16      T.root.color = BLACK
```

- When a node is initially inserted into the tree, it starts off red. Then, RB-INSERT-FIXUP is called which inspects the parent node’s color.
- After seeing a Red-Red violation, the function will have to check to see which case violation is occurring.
- When iterating over a red-black tree, we only percolate upwards. The only possibility of recoloring T.nil would be if we went past the root node.
- Case 1 and Case 3 are the only violations that recolor a node to red. In both cases, only z.p.p will be recolored red.
    - In a case 3 violation, the grandparent gets recolored. To recolor T.nil, we would have to be one of the root’s child nodes during a case 1 or case 3 violation.
    - However, this can’t happen because at the end of the loop, we will always recolor the root node black as shown on line 16.
    - When we perchlorate up, we would enter the root’s child nodes and see that the parent node, also the root node, is black and end the function.
- We will travel at most 2 levels upward during a case 1 violation or 1 level during a case 2 violation.

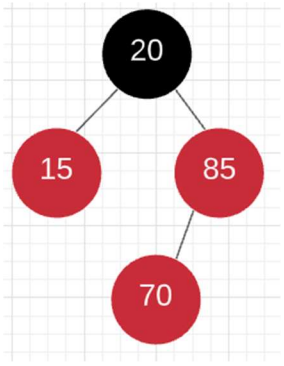
    - So in the event that the root node gets recolored red would be during a case 1 or case 3 violation two levels below the root node.
    - But the root node will still get recolored black after fixing the case violation, so the while loop will not see the root node as red.
- As a result, the while loop will not recolor T.nil to be red because it will never see the root node as red and will not use the root node for a case 1 or case 3 violation.

Brandon Vo

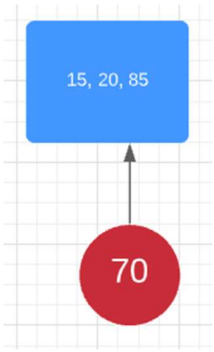
6. When inserting a node into a red-black tree there might occur a red-red violation. Show how each of the cases (i.e. case 1, case 2, and case 3) to solve the red-red violation can be understood by showing how it would be solved in a 2 − 3 − 4 tree (aka 2 − 4 tree)

- The correspondence between a 2-3-4 tree and a red-black tree is that black nodes can be seen as independent nodes that also retain the keys of any red children under that parent.
- A red-red violation is the equivalent of adding a dependent key to another dependent key that is already part of a node.

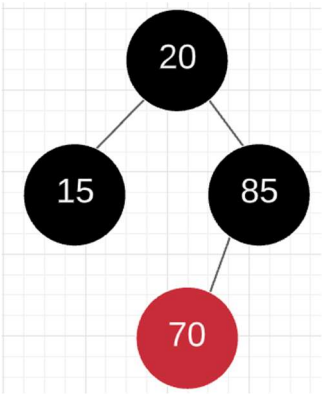
Case 1:

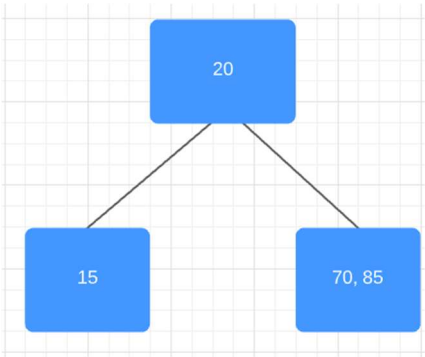


is equivalent to



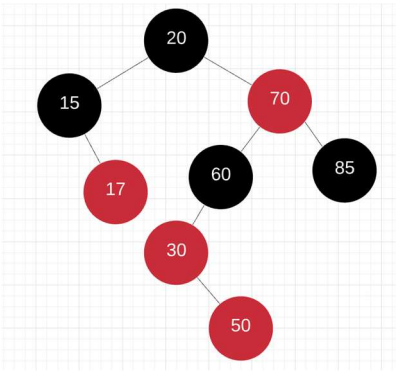
- **Case 1**: A case 1 violation is the equivalent of adding an additional key to a node that already has 3 keys.
    - This case violation is solved by marking 15 and 85 as black nodes.
    - This is the same as marking 15 and 85 as independent nodes in a 2-3-4 tree, effectively splitting them from being keys of 20.
    - This solves the red-red violation of the red-black tree and the 4-key violation of the 2-3-4 tree by splitting one node with 2 dependent keys into 3 different independent nodes.



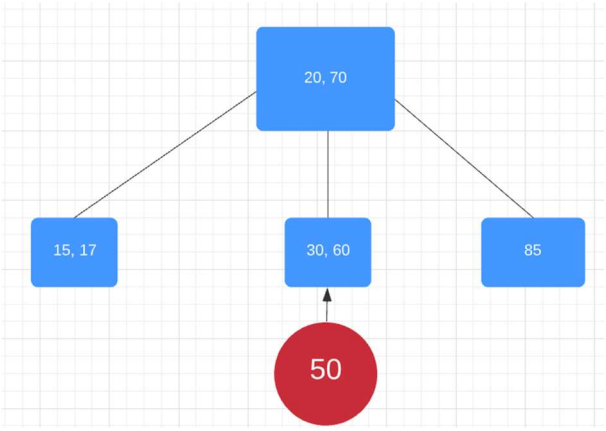


Brandon Vo

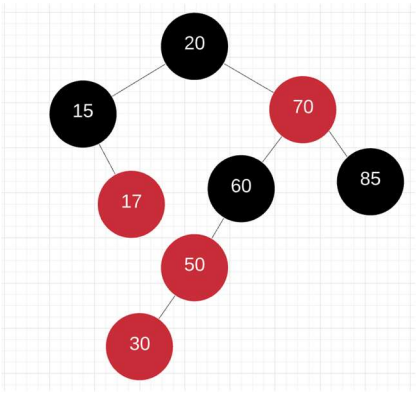
Case 2:



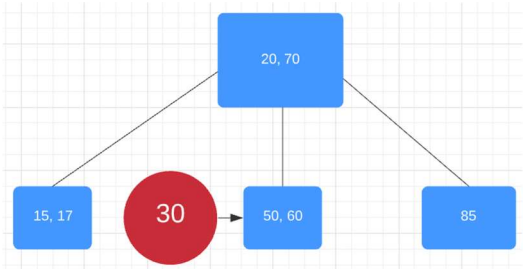
is equivalent to



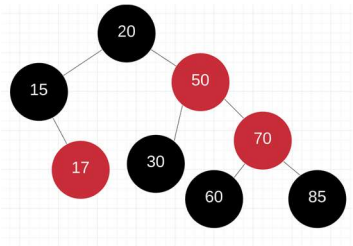
- A case 2 violation is equivalent to trying to insert an additional key to a 2-3-4 node.
    - A case 2 violation is solved by right rotating in the red-black tree to set up for a case 3 violation.
    - In a red-black tree, this is equivalent to swapping out the larger of the two links and acting as though we're re-inserting the larger key.
    - This is done to make it easier to identify where a node should be placed as it is easier to determine where to place a key that is larger than every other key in a node than it is to determine where to place a node that goes in-between the keys of an element.



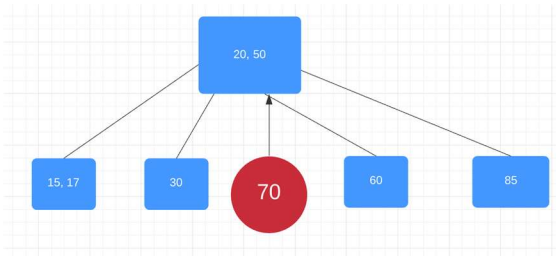
is equivalent to



Case 3:

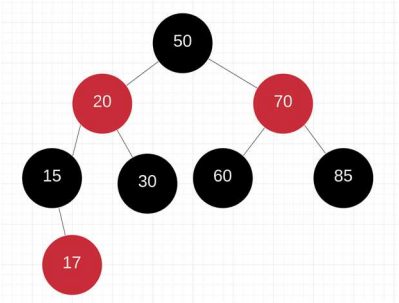


is equivalent to

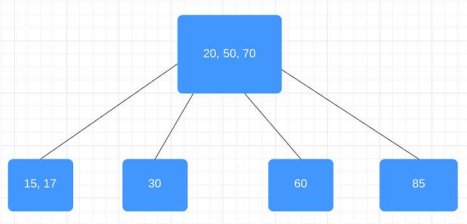


Brandon Vo

- A case 3 violation is similar to case 1 where a dependent key is trying to insert itself into another dependent key or a dependent red node is trying to insert itself to another dependent red node.
    - Solving the case 3 violation involves right rotating in a red-black tree and recoloring the new parent black with the previous parent turning red after being rotated down.
    - In a 2-3-4 tree, the rotation is equivalent to inserting a new key as a child of a node then merging that child with its parent before checking for any more 2-3-4 violations. If there's nothing wrong, then the new key will just be inserted into the node. Otherwise, it may incur an additional case violation and require splitting a node.



is equivalent to



Brandon Vo

7. Thank goodness you are in CS6033. You suddenly realize that there was a problem with your previous solution to knowing if someone is a subscriber.... what if someone canceled their subscription? You had no way of removing them from your hash table S without potentially also removing someone else whose email hashed to the same location.

You decide to invest in a cloud service and you now would like to determine if someone is a subscriber in O(log n) worst case time, and enter and remove a person in O(log n) worst case time.

Design an algorithm and justify your algorithm runs in O(log n) time for all operations.

- We can use a red-black tree to hold the list of subscribers instead
- In a R-B tree, it would cost O(log n) time just to insert, search , and delete a subscriber because the time complexity depends on the height of the red-black tree.
- Will only use the book's pseudocode for RB-deletion as deletion was not covered in class.

Insert-SUBSCRIBER(T, k)

//Will be using the code for RB-Insert but changing it so that it uses email_addresses as the key

Y = T.nil

X = T.root

While (X.value ≠ Nil)

    Y = X

    If Z.email_address < X.email_address

        X = X.left

    Else X = X.right

z.p = y

if y == T.nil

    T.root = z

Else if z.email_address < y.email_address

    y.left = z

else y.right = z

z.left = T.nil

z.right = T.nil

z.color = Red

Brandon Vo

RB-Insert-Fixup(T, z) //Will be using the function from the book

Find-SUBSCRIBER(T, k)

X = T.root

While(X.value ≠ Nil)

//Costs O(log n) time to go down the tree to find the subscriber's email address

    If X.email_address < k.email_address

        X = X.left

    Else if X.email_address > k.email_address

        X = X.right

    Else

        Print "Subscriber found" //Subscriber found

EndWhile

Print "This person is not a subscriber"

Brandon Vo

8. You discover you have a talent for public speaking! In fact, you love to be in front of a crowd. Now that covid restrictions are lifting, you decide to hold in-person seminars on how to pick stocks.

To save money on hotel fees, you will find a location a friend or family member lives, and when you go visit them you will hold a seminar in that location. As part of your marketing strategy, you decide to only invite established subscribers within the same geographic region to attend your seminar.

To quickly determine the names of all your subscribers who live in a certain area, design a data structure and implement the following operations:

Specifically:

- Build(T, A) Given an unsorted array of your subscribers,[1] create your data structure, T to hold the items in A, in O(n log n).
- Local_Subscriber_List(T, zipcode) The time it takes to determine the email addresses of all the your subscribers with a specific zip code is O(S + log n) where n is the number of your subscribers. (You can put the names in to an array A.)
- Insert(T, s) Add new subscribers, s, in O(log n) time.

- Using T as the data structure to hold our items and A to hold the list of subscribers and their information.
- We can construct T as two sets of Red-Black trees.
- T will initially hold the set of all unique zip codes as its nodes. This list will be a Red-Black tree
    - For each zip code, we will add another list
    - This list will hold another red-black tree which contains the list of all subscribers in that zip code
- When inserting a new subscriber to the list, we will check based on zip code first and see if it's a unique zip code or belongs to a zip code we already have
    - If it's a new zip code, we will insert the zip code into the first tree and do red-black tree balancing on the first tree
    - If it belongs to a zip code in the tree, we will search for that zip code then insert the subscriber into the tree and do red-black tree balancing in the second tree associated with that zip code. The time to rebalance the second tree would be O(log S) where S is the number of subscribers in the second tree. Since S < N, O(log n) would be asymptotically higher.
    - The worst case time would be having to insert a subscriber that belongs to a zip code in one of the leaves since we would have to travel to the bottom layer for both trees. The time needed to travel to the bottom of both trees would be the combined height of the list of subscribers and the list of zip codes which would be <= O(log n).
    - This would cost us O(log n) to insert a new subscriber.
- When searching for a list of subscribers in a zip code, we will go down the first tree until we either find the matching zip code or reach the end of the first tree. The time needed to go through the first tree and find the zip code would be O(log n) time.
    - If we find a node with a matching zip code, then we go to the second tree linked to that node

- In the second tree, we will traverse through the entire list and print out every subscriber found in the second tree. The time it would take to print every subscriber in a zip code would be O(S) time.

- The time to find the list of subscribers would be the combined time needed to find the node and print out every subscriber in the tree associated with that list: O(log n + S)

```
//T -> First list containing the list of zip codes

//T.list -> The second red-black tree that holds the list of subscribers for a particular node of a unique zip code

Build(T, A)

For i = 1 to A.length

//It takes O(log n) time to insert 1 node into the red-black tree and rebalance it.

//We have to insert n nodes each, making it cost O(n log n) time to insert every element in an array into a tree

        INSERT(T, A[i])             //Insert new subscriber

EndFor


Local_Subscriber_List(T, zipcode)

//Find a list of all subscribers in the area who share the same zipcode

While (T.node ≠ nil)                //Would require O(log n) time to travel down a tree to find the

                                    //subtree where subscribers with the same zip code would be found

        If (T.zipcode == zipcode)

                While(j = 0 to T.list.length)

                        //Will cost O(S) time

                        Print(T.list[j].subscriber_name)  //Print out every subscriber in the second list

                        Return

        Else If (zipcode > T.zipcode)       T = T.right

        Else      T = T.left

//Combined time would be O(S + log n)

EndWhile

//We reached the end of the tree

Print "There are no subscribers with that zip code"
```

Brandon Vo

Return

```
Insert(T, s)        //Insert a new subscriber into the tree
Y = T.nil
X = T.root
While (s ≠ X.Nil) //Search until we either reach the end or we find a match
        Y = X
        If (s.zipcode < X.zipcode)
                X = X.left
        Else if (s.zipcode > X.zipcode)
                X = X.right
        Else (s.zipcode == X.zipcode)
                Subscriber-Insert(X.list, s)        //Do a red-black tree insert into the second list
                Return                              //If we're inserting an extra element into the second tree,
                                                    //then we do not need to rebalance the first tree
EndLoop
//If it reaches this step, then that means this subscriber has a unique zip code
//Do a red-black tree insert into the first tree and rebalance the first tree
s.p = y
If y == T.nil
        T.root = s
Else if s.zip_code < y.zip_code
        y.left = s
else y.right = s
s.left = T.nil
s.right = T.nil
s.color = Red
RB-INSERT-FIXUP(T, s)
```

Brandon Vo

```
//Same as a red-black insert but we're doing it in the second tree contained within the zip code's node and we're using subscriber names instead.
Subscriber-Insert(T.list, s)
Y = T.list.nil
X = T.list.root
//Would take O(log s) time to insert a node here
//But there is additional overhead from having to search based on the zip codes as well
//making this O(log n)
While x ≠ T.list.nil
        Y = x
        If s.subscriber_name < x.subscriber_name
                X = x.left
        Else x = x.right
s.p = y
if y == T.list.nil
        T.root = s
Else if s.subscriber_name < y.subscriber_name
        y.left = s
else y.right = s
s.left = T.nil
S.right = T.nil
S.color = RED
RB-INSERT-FIXUP(T.list, s)
```

Brandon Vo

9. Unfortunately your first seminar did not go well. Very few people showed up. You discovered that another market guru held a seminar two days before you held your seminar. Instead of cursing your luck and giving up on your dream, you decide to collaborate. You contact the market guru and the two of you decide to join forces; you will hold joint seminars.

To make this work, you need to combine mailing lists. The market guru will send you a sorted array, A, of their subscribers where it is sorted based on zip codes and then within each zip code it is sorted by name.

Design a data structure that supports the following operations: Combine to insert a list of items into your data structure in $O(m_1 + m_2)$, Insert(T, s) to insert a new subscriber into your data structure in time $O(\log(m_1 + m_2))$, and Local Subscriber List to return a list of subscribers within a certain zip code in

$O(S + \log(m_1 + m_2))$ where S is the number of names with that zip code.

You only need to implement the following method:

- Insert_List(T, A) that takes both your mailing lists and creates a new mailing list. The time to combine the market guru's list with your list 2 is $O(m_1 + m_2)$ worst case time where m_1 is the size of your mailing list and m_2 is the size of market guru's mailing list. Remove duplicates. (You may assume all names are unique in a zip code - i.e. there are not two "John Smiths" in zip code 10003.) Justify the running time of Insert List.

[2]Your list, T, is stored in the data structure you used in question 8.

[3]If a TA does not believe it is possible for your choice of data structure to support Insert(T, s) and Local _Subscriber_List in the running times listed, you will not receive any credit for this question.

- Assuming m_2 are sorted arrays, we can construct a red-black tree from both lists by using a form of merge sort where we traverse through both m_1 and m_2 from the lowest element to the highest element.
    - We would need to convert our list T back into a sorted array by doing a pre-order traversal. This would cost $O(m_1)$ which would lose to $O(m_1 + m_2)$ time.
    - We check for $\min(m_1[i], m_2[j])$ and pick one element from the pair and increment the index for the chosen array.
- A list M will be created as a Red-Black tree so that it would take $O(\log (m_1 + m_2))$ time to insert a new subscriber into the list since we only have to travel down the height of the tree of M at worst. The height of M would be $O(\log (m_1 + m_2))$ because the size of M will be $m_1 + m_2$.
    - It would also take $O(\log (m_1 + m_2)$ time to call Local_Subscriber_List since it requires only to percolate down the tree to find the given element of a zipcode.
- The function Insert_List will cost us $O(m_1 + m_2)$ time
    - $O(m_1)$ to travel through the list of size m_1
    - $O(m_2)$ to travel through the list of size m_2
    - We have to go through both lists and read every element from both lists to construct our merged red-black tree, taking us $O(m_1 + m_2)$ time.

Brandon Vo

```
Convert_Tree(T, U, i)
//Convert list m1 from a Red-Black tree into a sorted array
//Perform a pre-order traversal, costing O(m1) time
If T.value != nil
        Convert_Tree(T.left, U, i)
        //Put the list of subscribers into a sorted array
        While(for j = 0 to T.list.length)
                U[i] = Tree.list[j]
        i++
        Convert_Tree(T.right, U, i)
```

```
Insert_List(T, A)
//Assuming our list of subscribers would be contained in Array N while the other guru's subscriber list is contained in A
Create array M to hold all subscribers from both lists
Create array N to hold the sorted array for m1
Create array DUP = FALSE as a hash table to check for duplicates
Create x = 0 and y = 0 as indexes for A and N
N = Convert_Tree(T, N, 1) //Convert T from a Red-Black tree into a sorted array and store it in N
//O(m1) time to convert from Red-Black to sorted array.  O(m1) < O(m1 + m2), so the time won't be an issue
MAX = A.length + N.length        //In case of duplicates, the combined size might be smaller than perceived
For i=0 up to MAX
//This merge will iterate over the list of A and the list of N.  It will take the lower value one of the two and add it to a third array
//It takes O(m1) to iterate over the list of A
//It also takes O(m2) to iterate over the list of N
//To go through both arrays, it would take O(m1 + m2)
```

Brandon Vo

```
//Do the equivalent of the merge step in merge sort
//Assuming nil effectively counts as infinity and that we'll never insert a nil node.
    While (DUP[H(A[X])] == TRUE)    //Duplicate found
        MAX--
        X++
    While (DUP[H(N[Y])] == TRUE)    //Any collision found means a duplicate was found
        MAX--
        Y++

    If (A[X].zipcode < N[Y].zipcode)  //Take the smaller one of the two
        INSERT(M, A[X])           //Using the Insert function from the previous question
        X++
        DUP[H(A[X])] = TRUE      //Take the hash of the subscriber and store it in the hash table
    Else if (A[x].zipcode > N[Y].zipcode)
        INSERT(M, N[Y])           //Insert N[Y] into the R-B tree of M
        Y++
        DUP[H(N[Y])] = TRUE      //Store the hash of the subscriber and mark it
    Else if (A[x].zipcode == N[Y].zipcode)      //Matching zip codes, check for names then
        If (A[x].subscriber_name < N[Y].subscriber_name)
            INSERT(M, A[X])
            X++
            DUP[H(A[X])] = TRUE
        Else (A[x].subscriber_name > N[Y].subscriber_name)
            INSERT(M, N[Y])
            Y++
            DUP[H(N[Y])] = TRUE
```

Brandon Vo

10. Wow. You didn't know how many other people were also holding seminars. If you had known this, you would have never started holding seminars, instead you would have gone into investment banking.

You and the market guru have now teamed up with the M other speakers. You will create a master list of all the subscribers names:

- Each of the M speakers has provided you their subscriber lists as a sorted array: sorted by zip code and within each zip code ordered alphabetically by the subscriber's name. Let n_i be the number of names in the *i*th sorted array, A_i for $1 \le i \le M$.
- You and the marketing guru's mailing list is in the data structure you used in question 9. The number of names in your list is n_0.

Design a data structure that supports the following operations:

- Insert_Lists(T, A_1, A_2, . . . , A_M) to insert the lists from the other speakers into your data structure in O(n log M) time where n = $\sum_{i=0}^{M} n_i$
- Insert(T, s) to insert a new subscriber into your data structure in time O(log(n))
- Local Subscriber List to return a list of subscribers within a certain zip code in O(S + log(n)) where S is the number of names with that zip code

You only need to implement the following method:

- Insert_Lists(T, A_1, A_2, . . . , A_M) This method add the mailing lists A_i to your data structure in O(n log M) worst time where n = $\sum_{i=0}^{M} n_i$ and $n_i = |A_i|$ is the number of items in the i'th subscriber list.[4] Remove duplicates(again, you may assume the names are unique within a zip code).

In your method, clearly write the data structures you will use and what you will store in each data structure. Justify the running time of Insert_Lists.[5]

Hint: this question heavily uses some of the data structures we covered in previous lectures along with the data structure discussed in lecture 4.

- Our data structure T will be designed in 2 parts: One part will hold the nodes of zip codes while the second part will hold the list of subscribers in that zip code.
    - Because the zip codes have higher priority, they will be the nodes represented in T first
    - These nodes will serve as the red-black tree.
    - However, for the 2nd part, we can use a red-black tree to represent the list of subscribers
- For insert_lists, what we can do is create a min-heap of size M
    - We will insert the smallest node from each array into the min-heap. This will be trivial because every array is already sorted.
    - Using extract-min-heap, we will always get the smallest item in O(1) time
        - We will then insert the next item from the A_i list into the min-heap.
            - However, the time to insert an item into our list T would be O(n log n)
        - It will cost O(log M) time to extract min and replace that node with a new node from list A_i.

        - We will repeat this process until all lists are exhausted and the min-heap is empty.
        - We would end up traversing every element from every array, which would take us O(n) time to repeat for every element in every array.
        - At that point, our merged list will be complete
- It would cost O(n log M) to go through every element in a min-heap, maintain that min-heap for every node extraction by replacing it with the total of n elements.
    - However, there is O(log n) time used to insert an item from the min-heap to our data structure T which might make Insert_lists O(n log n)
- Using our data structure T, it would be a 2-part tree with the first tree containing a list of unique zip codes. Each zip code would hold the list of subscribers in that zip code
    - Insert(T, s) would still take O(log n) time due to the maximum height to traverse both lists being log n
    - Local_Subscriber_List would still take O(S + log N) time since the time needed to traverse the list of zip codes would be O(log n) with the additional time needed to traverse the list of subscribers being O(S)

Insert_Lists(T, A_1, A_2, …, A_M)

//Add the mailing lists A_i to the data structure in O(n log M) time.

Create array B as the min-heap

Create array C[M] = 0 to mark the index for every array on A_m. This is to keep track of what nodes we have inserted into the list and where the next smallest nodes in the list are located at.

Create DUP as a hash table to mark duplicates

For i = 1 to M

//Set up the min-heap

//Assuming each subscriber also has an identifier describing from which list A_i that subscriber came from

    MIN-HEAP-INSERT(B, A[i].subscriber[0]) //Insert the smallest element from every node

    DUP(H(A[i].subscriber[0])) = TRUE

    C[i]++

EndFor

J = 0

//Repeat until we no longer have any elements in the min-heap

While(B is not empty)

    MIN = EXTRACT-MIN-HEAP(B) //Costs O(log m) time to extract and rebalance

Brandon Vo

```
        J = MIN.list                    //Record what list MIN came from
        If (DUP(H(MIN)) == FALSE)       //Not a duplicate.  Insert into list.
                Insert(T, MIN)                  //Insert the min node into our Red-Black tree.  O(n log n)
                DUP(H(MIN)) = TRUE
//Replace the extracted element in the min-heap with the next smallest element from the array we extracted MIN from
//Because we're constantly replacing an element with a new one, we will eventually go over every element, taking O(n) time
//Combined with the min-heap extraction process, we will complete the merged tree in O(n log M) time
        If(C[j] < A[j].length)      //That array still has elements leftover.  Insert the next one to the heap
                INSERT-MIN-HEAP(B, A[j].subscriber(C[j]) //Replace it with the next smallest node from
                                                        //that same list
                C[j]++
//Replacing the node costs O(log m) time for one element
//Will also cost O(log m) time to insert a new node into the min-heap
```

Brandon Vo

11. (3 bonus points) Think of a good 6 exam question for the material covered in Lecture

What would be the maximum potential height of a red-black tree if each node is able to hold up to 4 links similarly to a 2-3-4 tree. What would its maximum potential black-height be.