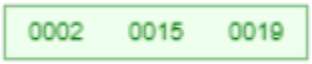
Solutions for HW4 - CS 6033 Fall 2021

Q1 → 2-3-4 Tree into an R-B Tree

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.

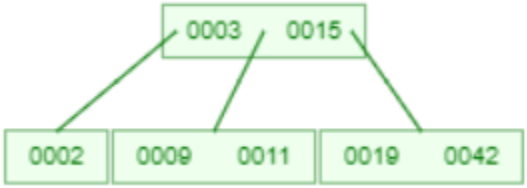
We are building the tree by doing preemptive splits as discussed regarding B-trees.



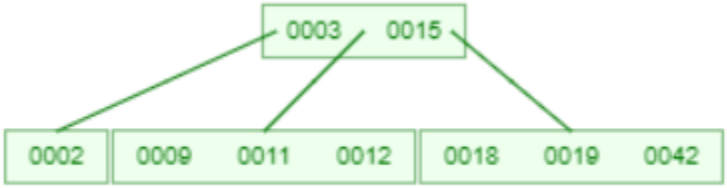
→ After inserting 2, 19 and 15

→ After inserting 3

→ After inserting 42 and 11



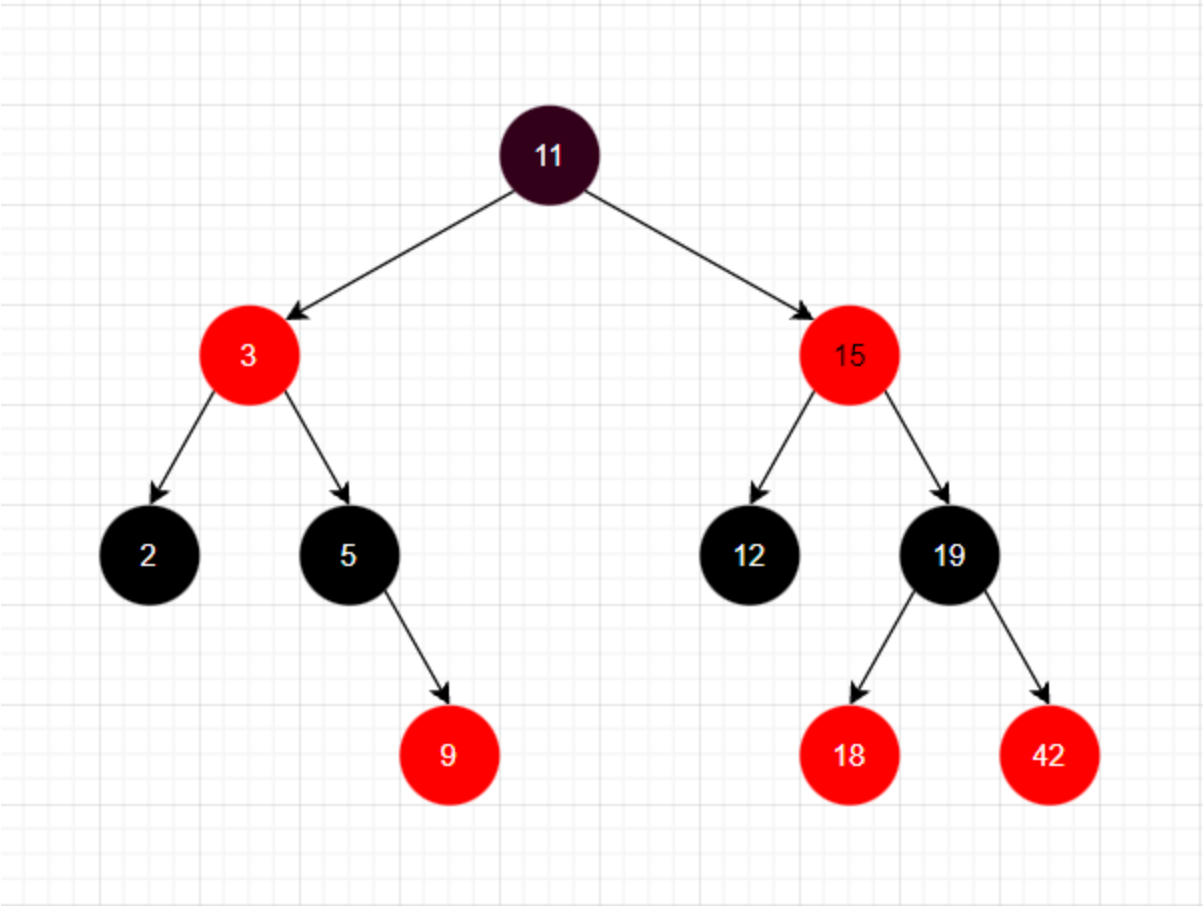
→ After inserting 9

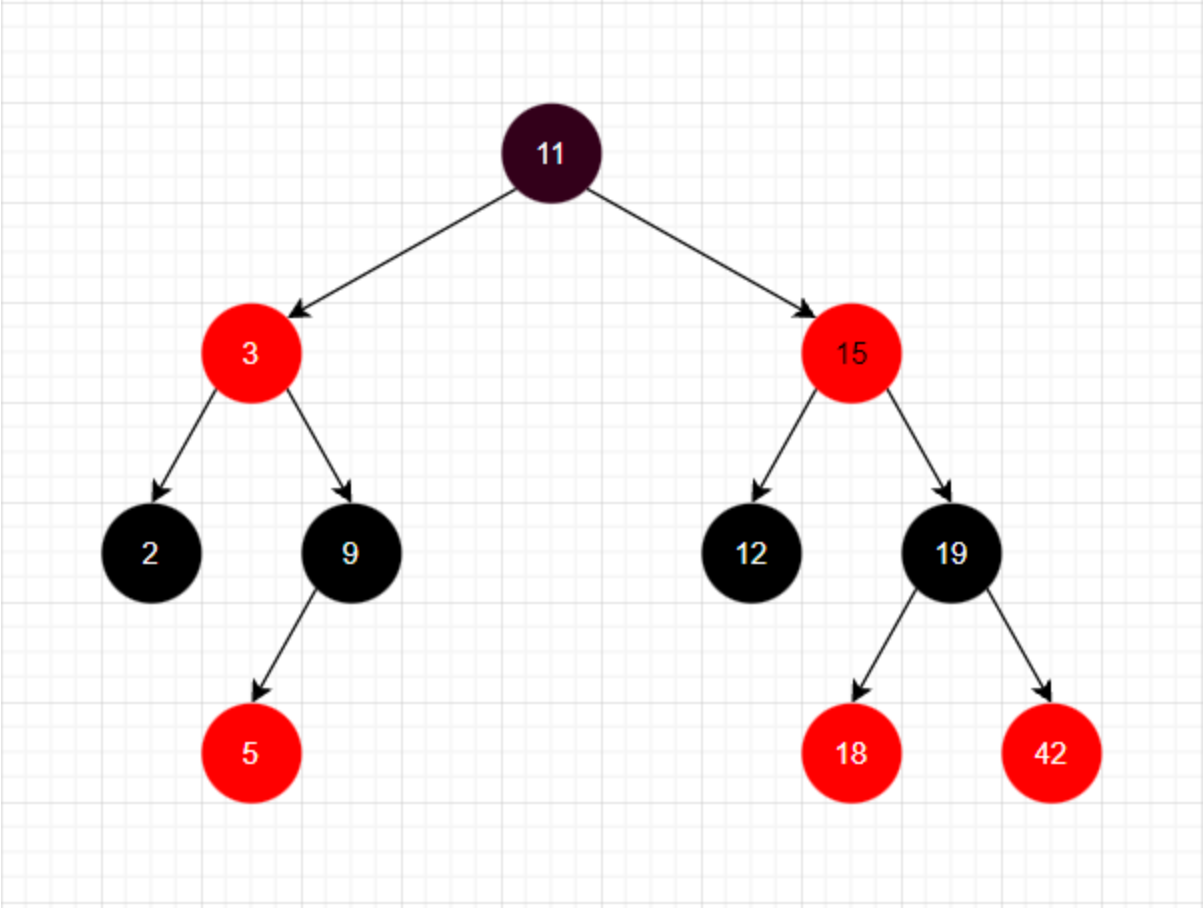


→ After inserting 18 and 12

→ After inserting 5 (the final tree)

There are two possible constructions of the R-B tree. Both are shown below.


11
3
15
2
5
12
19
9
18
42



Q2 → R-B Tree into a 2-3-4 Tree

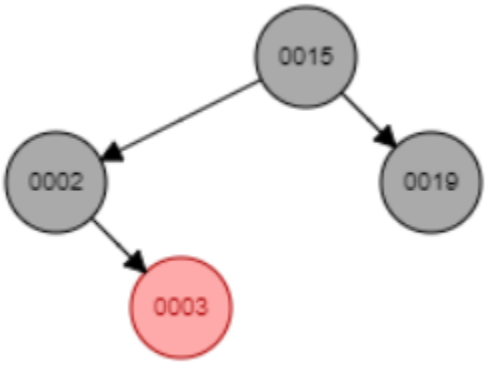
2. Insert in order the keys $2, 19, 15, 3, 42, 11, 9, 18, 12, 5$ into a red-black BST tree. Show your tree after every insertion. Indicate the color of each node by letters B (black) and R (red). You may omit representing the `NIL` nodes.

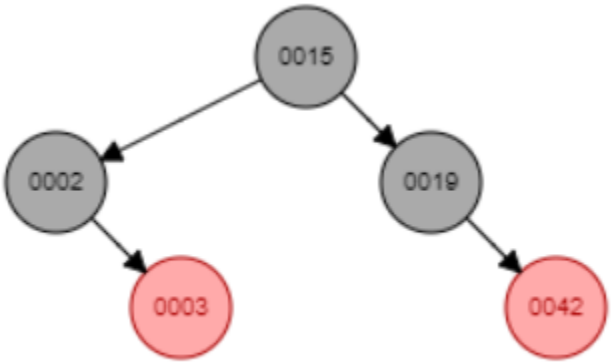
   After that, convert the red-black BST you have constructed into a $2-3-4$ tree, using the steps we discussed in class.

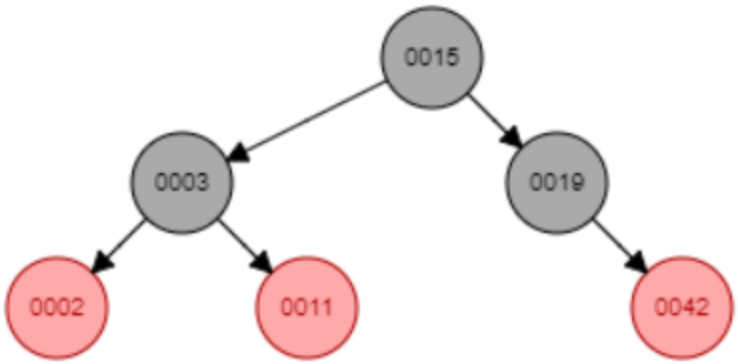
The insertion progression is as follows:

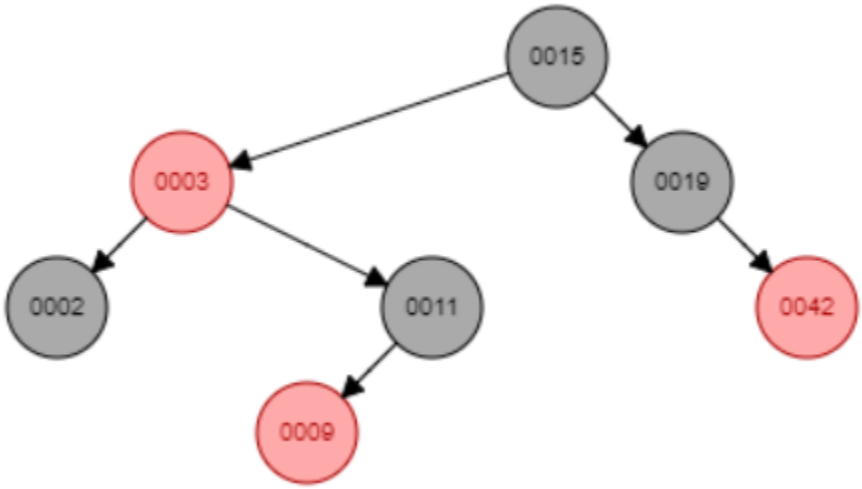

0002
0019
0002

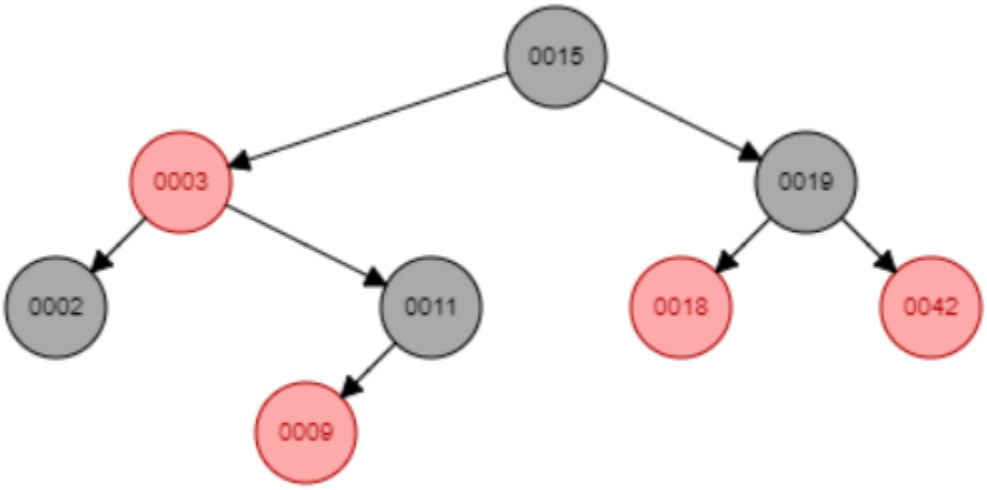

0015
0002
0019

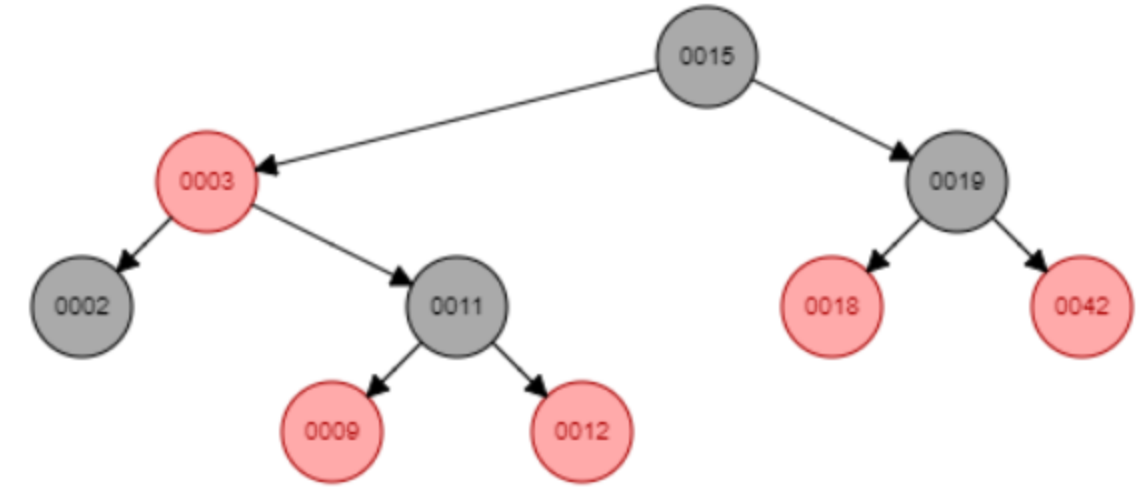

0015
0002
0019
0003

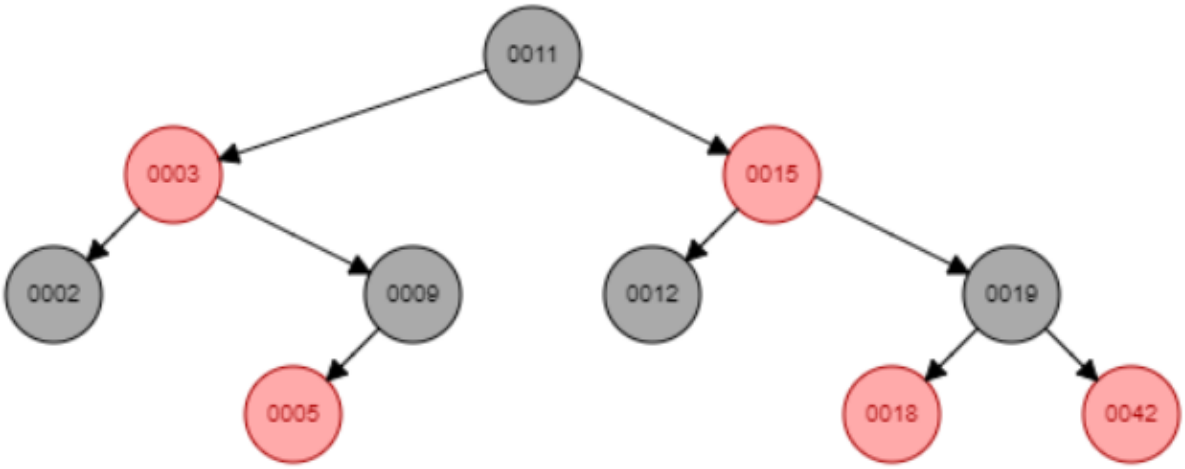

0015
0002
0019
0003
0042


0015
0003
0019
0002
0011
0042

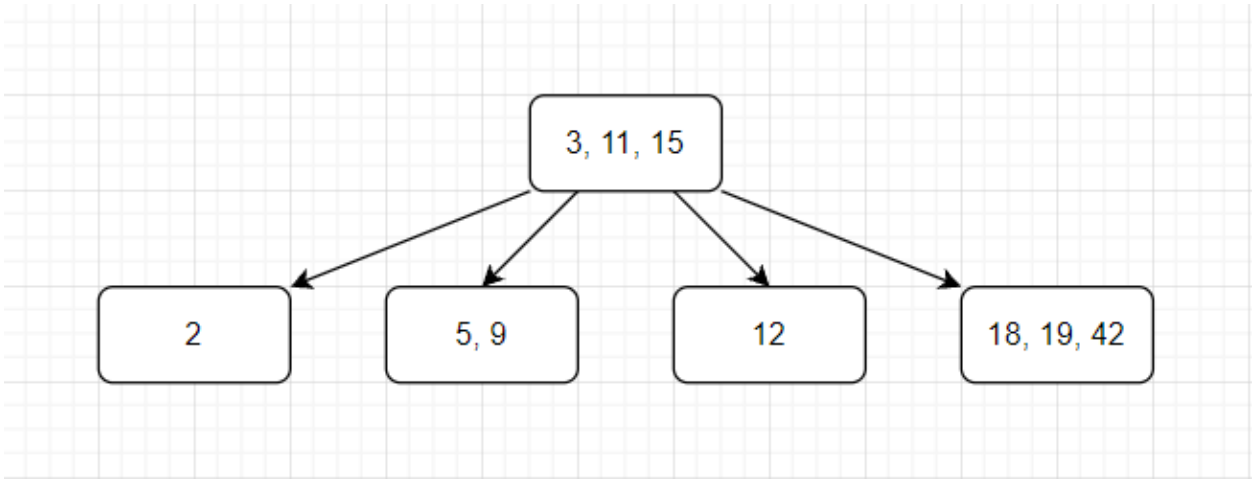

0015
0003
0019
0002
0011
0042
0009


0015
0003
0019
0002
0011
0018
0042
0009





We get the following 2-3-4 tree by converting this R-B tree:



Q3 → Height of 2-3-4 Tree

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

(a) Maximum height → Only 1 key per node and therefore, it behaves like a perfectly balanced binary search tree.

Nodes at level 0 → 2^0 = 1

Nodes at level 1 → 2^1 = 2

Nodes at level 2 → 2^2 = 4

…

Nodes at level h → 2^h (where h is the height of the tree)

Total nodes = n = 1 + 2 + 4 + … + $2^h = \frac{1-2^{h+1}}{1-2} \Rightarrow n = 2^{h+1} - 1 \Rightarrow h = log_2(n+1) - 1$

The height of the tree is $log_2(n+1) - 1$

(b) Minimum height → 3 keys per node

Nodes at level 0 → 4^0 = 1

Nodes at level 1 → 4^1 = 4

Nodes at level 2 → 4^2 = 16

…

Nodes at level h → 4^h (where h is the height of the tree)

Total nodes = n = 1 + 4 + 16 + … + $4^h = \frac{1-4^{h+1}}{1-4} \Rightarrow 3 * n = 4^{h+1} - 1 \Rightarrow$

$h = log_4(3 * n + 1) - 1$

The height of the tree is $log_4(3 * n + 1) - 1$

Q4 → Attribute of R-B Tree

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.

From the red-black properties, we have that every simple path from node x to a descendant leaf has the same number of black nodes and that red nodes do not occur immediately next to each other on such paths. Then the shortest possible simple path from node x to a descendant leaf will have all black nodes,

and the longest possible simple path will have alternating black and red nodes. Since the leaf must be black, there are at most the same number of red nodes as black nodes on the path.

Q5 → RB-Insert-Fixup

5. CLRS page 322, question 13.3-4

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 z is the root. Show that the professor's concern is unfounded by arguing that RB-INSERT-FIXUP never sets $T.nil.color$ to RED.

First observe that RB-INSERT-FIXUP only modifies the child of a node if it is already RED, so we will never modify a child which is set to T.nil. We just need to check that the parent of the root is never set to RED.

Since the root and the parent of the root are automatically black, if z is at a depth less than 2, the while loop will be broken. We only modify colors of nodes at most two levels above z, so the only case we need to worry about is when z is at depth 2. In this case, we risk modifying the root to be RED, but this is handled in line 16. When z is updated, it will be either the root or the child of the root. Either way, the root and the parent of the root are still BLACK, so the while condition is violated, making it impossibly to modify T.nil to be RED.

Q6 → Relation between R-B tree and 2-3-4 tree

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)

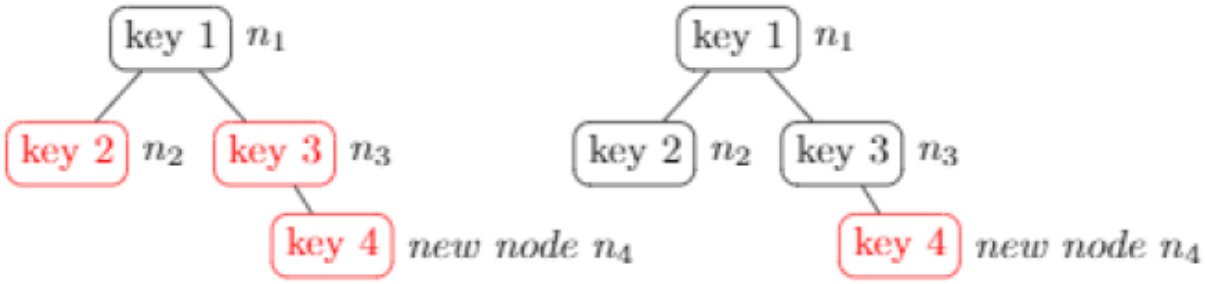
P.S: "key 1" in the node doesn't mean integer "1", just indicate some reasonable number for that place.

case 1: double red problem

red-black tree:

situation: both parent n_2 and sibling of parent n_1 are red nodes.

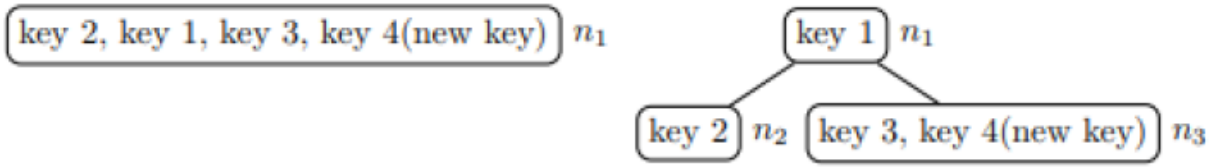
fix: change both parent n_3 and parent's sibling n_2 into black, and change parent's parent n_3 into red, finally set root to be black.



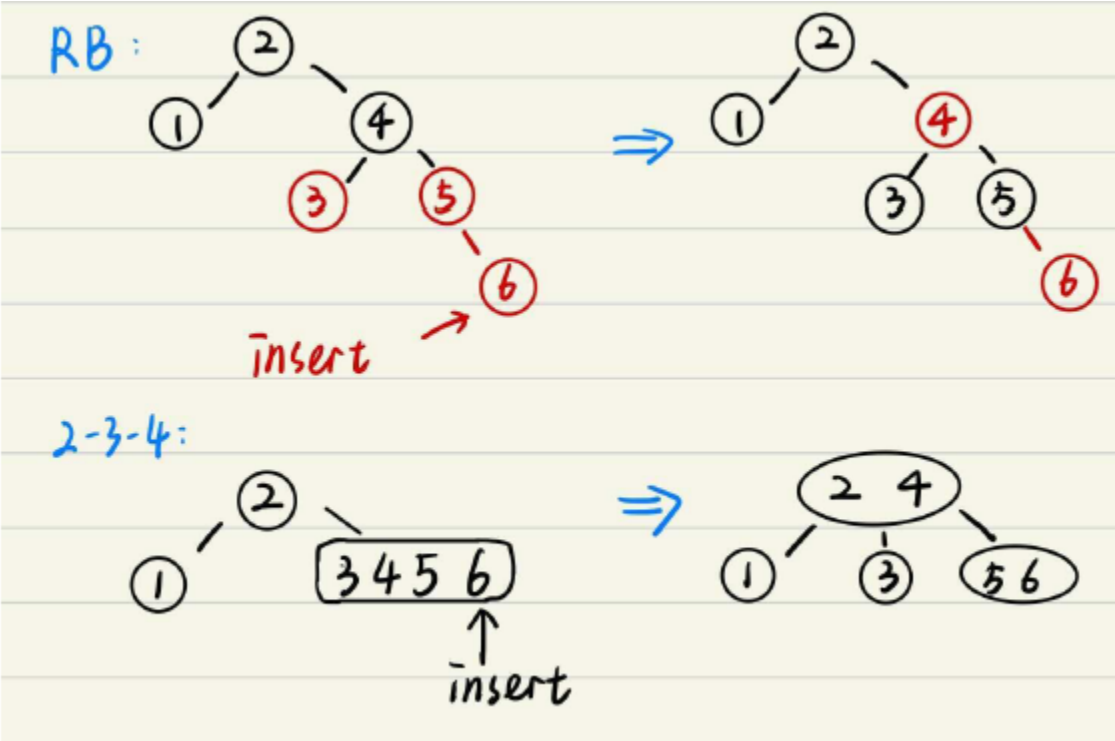
2-3-4 tree:

situation: situation: the node to be inserted has already three keys

fix: since the origin node has 3 keys, it splits into two children, chooses the middle one key_1 to be the parent.



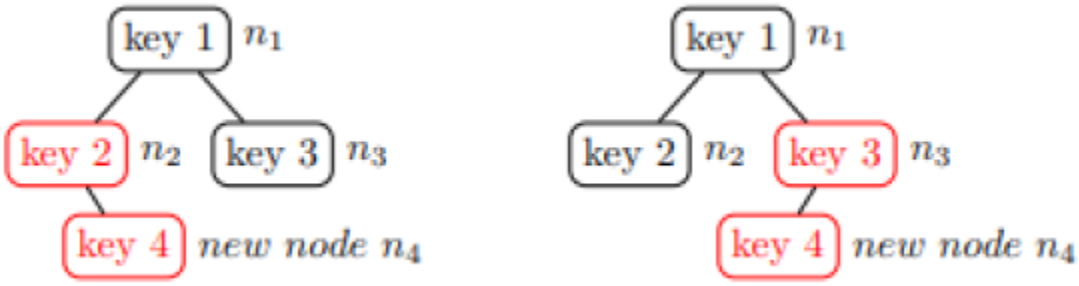
if n_1 is not a root, then:



case 2: one black and one red problem

red-black tree:

situation: uncle node is black, the new node and new nodes' parent are on the different side



fix: change rotate new node and new node's parent, to make sure they are on the same side, then it would be case 3.

2-3-4 tree:

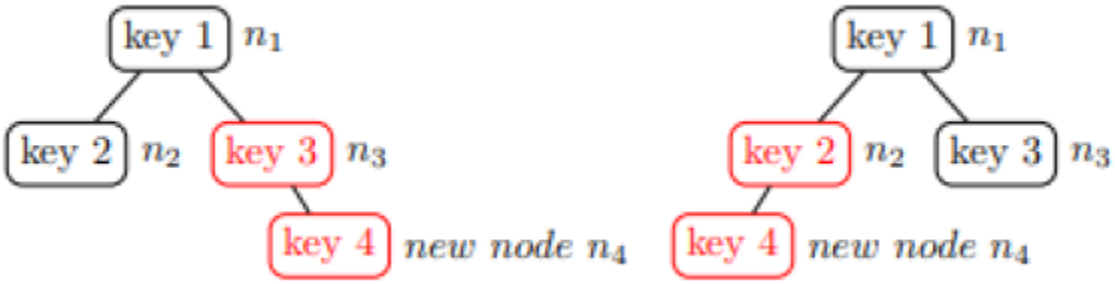
situation: the node to be inserted into has 2 keys

fix: the node to be inserted only has two keys, so we can directly insert a new key into the corresponding node.

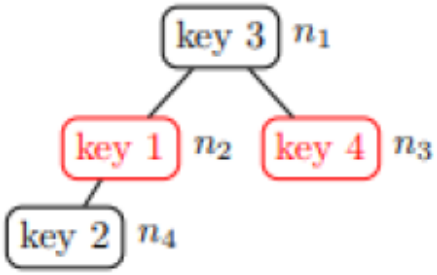
case 3: one black and one red problem

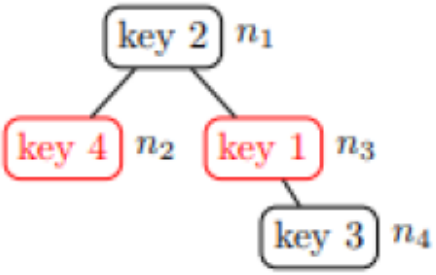
red-black tree:

situation: sibling of the parent is black, the new node and new nodes' parent are on the same side



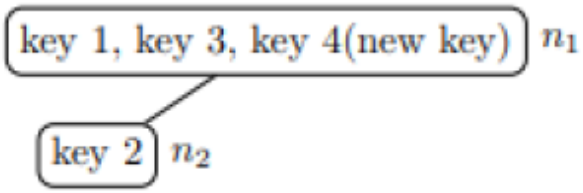
fix: rotate the new node' parent and new node's parent's parent, make new node and new node's parent step into a higher level, let new node's parent's parent be another child of new node's parent, and then change new node's parent into black, and change new node's parent's parent into red.

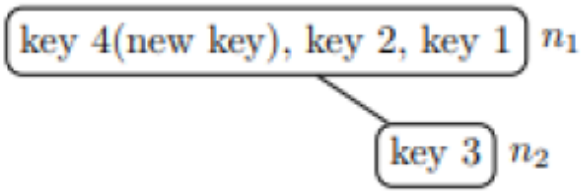




2-3-4 tree:

situation: the node to be inserted into has 2 keys





fix: since the origin node has only two keys, just insert a new key into the node is fine.

Q7 → Application of Trees - Subscribers

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.

Answer: One answer would be to use a balanced tree such as a Red-Black Tree. As for the indicator which we use to compare nodes in the Tree, we can use the email (as it is going to be unique for each subscriber). So, we insert nodes based on the lexical order of the string ‘email_id’ for every subscriber. The pseudocodes will be:

insert_subscriber(subscriber_tree, new_subcriber): //INSERTION

```
        if not TREE-SEARCH(subscriber_tree.root, new_subscriber.email_id):
                RB-INSERT(subscriber_tree, new_subscriber)

delete_subscriber(subscriber_tree, subscriber_to_delete):                    //DELETION
        if TREE-SEARCH(subscriber_tree.root, subscriber_to_delete.email_id):
                RB-DELETE(subscriber_tree, subscriber_to_delete)
        else:
                print "Subscriber not found"
```

where subscriber_tree is the red-black where we store the subscribers. Both the above subroutines have a search operation (taken directly from the book) which takes O(log n) in a balanced tree, and an RB-INSERT and RB-DELETE operation respectively, which take O(log n) each. So, the total running time of both these subroutines is O(log n) + O(log n) = O(log n).

Q8 → Application of Trees - Public Speaking

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.

Answer: A Red-Black (or any balanced) tree is a suitable choice of a data structure for this implementation.

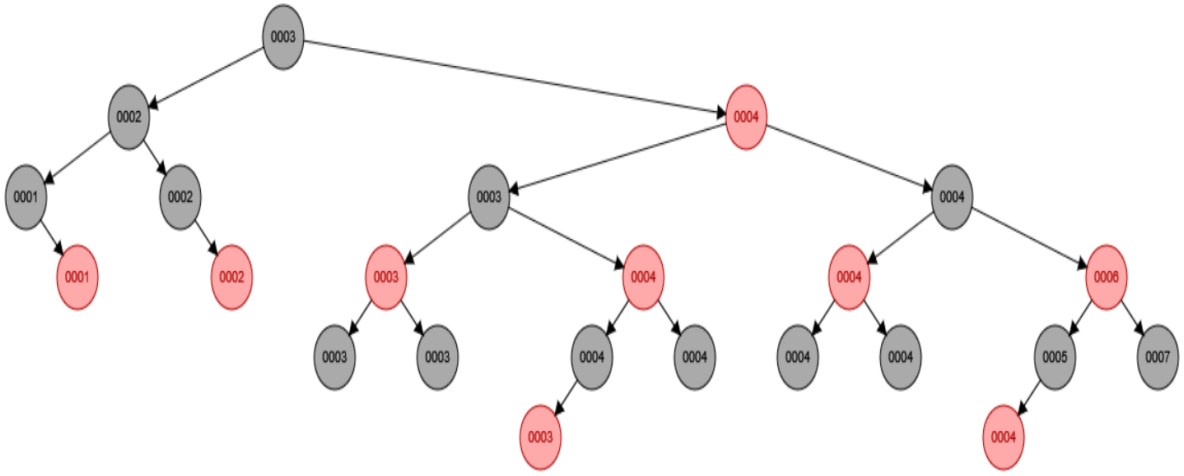
We will order the nodes by their zip code. The pseudocodes are:

```
BUILD(subscriber_tree, subcriber_list):
```

```
for i = 1 to subscriber_list.length:
        INSERT(subscriber_tree, subcriber_list[i])          //INSERTION defined below
```

This operation takes O(log n) in the worst case for each element in subscriber_list, which has length n, so the total runtime is O(n log n).

Finding the emails of all the subscribers within a zip code is somewhat tricky with a Red-Black tree. Recall that during insertion in any BST, if we come across an existing node that has the same key, we go towards the right. Thus, in an unbalanced BST, search is simpler as we always go toward the right. But in a Red-Black tree, rotations made to balance the tree might result in a scenario where it looks like nodes with the same key could be on either side of the parent. I've included an example below for clarity.



We see how the nodes with key: 4 are randomly placed even though the structure of the tree is maintained. We see that once we reach one node with the specified key, we need to search both the left and right subtree all over again to find all the required nodes, which is exactly what the function CHECK_NODE( ) below does.

CHECK_NODE(subscriber_node, output_list, zipcode):

```
        current_node = subscriber_node                          // Let us assume
        while (current_node.exists( ) and current_node.zip != zipcode):     // that all these
                if current_node.zip < zipcode:                  //statements together
                        current_node = current_node.right       // take cost
                else:                                           //orange_cost_j when
                        current_node = current_node.left        //CHECK_NODE is
        if current_node.exists( ):                              //called for the j^th time.
```

```
        output_list.append(current_node.email)                          //These statements are
        if current_node.left:                                           //executed once for the
            CHECK_NODE(current_node.left, output_list, zipcode)         //required S nodes, so
        if current_node.right:                                          //they have total cost
            CHECK_NODE(current_node.right, output_list, zipcode)        // O(S)
```

```
LOCAL_SUBSCRIBER_LIST(subscriber_tree, zipcode):
    output_list = [ ]
    CHECK_NODE(subscriber_tree.root, output_list, zipcode)
    return output_list
```

The runtime for LOCAL_SUBSCRIBER_LIST is the same as the runtime for CHECK_NODE. To find the running time of CHECK_NODE, I've split the code into two blocks. The top (orange) part is dependent on the position of subscriber_node in the tree. So, I'm denoting it as *orange_cost_j* for the j^{th} instance CHECK_NODE is called. The bottom (blue?) part is much easier. These statements are executed once for each required node so their runtime is going to be *O(S)*. Here we see that CHECK_NODE is going to be called twice for each required node, so the total run-time is *O(S)* + $\sum_{j=1}^{2S} orange\_cost\_j$.

The best case is when $S=1$, that is we find the one required node in just one path traversing from the root to the leaf. So $\sum_{j=1}^{2S} orange\_cost\_j$ will be $O(\log n)$. In the worst-case $S=n$, every node we hit will be a required node, and $\sum_{j=1}^{2S} orange\_cost\_j$ will be $O(S)$. We can conclude that $\sum_{j=1}^{2S} orange\_cost\_j$ is dominated by either S or $\log n$ depending on the value of S. If S goes down $O(\log n)$ would be dominant, and if S is high, $O(S)$ would be dominant. So, for the average case we can write the total runtime of LOCAL_SUBSCRIBER_LIST as $O(S) + \sum_{j=1}^{2S} orange\_cost\_j = O(S) + O(S + \log n)$ which is equal to $O(S + \log n)$.

```
INSERT(subscriber_tree, new_subcriber):
        RB-INSERT(subscriber_tree, new_subscriber)
```

We reuse the same routine we defined in the previous question, with a small change. Instead of ordering the tree by the email address of the subscribers, we will use their zip codes. This will help us with the

subroutine LOCAL_SUBSCRIBER_LIST. I have also not opted to check if the same zip code exists in the tree, because zip codes are not unique. The runtime is the same at O(log n).

Q9 → Application of Trees - Public Speaking with 2 people

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.[3]

Firstly we can create a list of all the items (items = my subscribers) in the red-black tree in O(m1) time - by simply traversing the tree.
Then we can merge the two lists - as is done in merge sort (where I also remove duplicates at this time) into an array.
Then we can create a nearly complete tree (but with pointers) and then color all the nodes black except the nodes on the last level which we can color red. (i.e. for every item in the new merged list (list stored in an array) we can create a tree node, and then we can add the pointers by using the array positions to find the locations of the nodes)

At last, we can add the nil node (s)

Q10 → Application of Trees - Public speaking with n people

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 ith sorted array, A_i for $1 \leq i \leq M$.
    - 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$(T, A_1, A_2, \ldots, A_M)$ 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$(T, A_1, A_2, \ldots, A_M)$ 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.

As we learned from the Q9, it takes O(m1+m2) to combine two lists, now we can use the merge-sort approach(divide and conquer) to tackle this problem.

Suppose, we have an M list and have elements (n1, n2, n3, n4, n5, n6, n7….. nm)

What we will do is first we will merge 2-2 list … so it would be like [n1 + n2, n3 + n4, n5 + n6 ……. nm-1 + nm]

Now we will merge again 2 lists but this time we have the already merged list from the above step. So now [(n1+n2, n3 + n4), (n5 + n6 + n7 + n8) ……. (nm-3 + nm-2 + nm-1 + nm)]

So we will do this step till we will get the whole list like [n1 + n2 + n3 …. nm]

Now, let's calculate running time for this. Step 1: [(n1+n2), (n3 + n4), (n5 + n6) ……. (nm-1 + nm)] → total time would n

Step 2: [(n1+n2, n3 + n4), ……. (nm-3 + nm-2 + nm-1 + nm)] → total time would n . . .

Last step: [n1 + n2 + n3 …. nm-1 + nm] - > Also total time would be n

Now lest calculate how much steps we will take: In every step we are merging 2 lists (it’s kind of concurring 2 list)

So total step would be: log (total number of list)

So total time = Total step * n = (log M)*n

(The solution is credited to Vikas Parmar and Sanket Mendapara)