VIETNAM GENERAL CONFEDERATION OF LABOUR
TON DUC THANG UNIVERSITY
INFORMATION TECHNOLOGY FACULTY

REPORT ASSIGNMENT OF

DATA STRUCTURES AND ALGORITHMS 2 SUBJECT

TREE DATA STRUCTURES

Instructing Lecturer: Mr. NGUYEN QUOC BINH

Student's name: **HUYNH LE THIEN Y – 518H0320**

Class : **18H50204**

Course : **22**

HO CHI MINH CITY, YEAR 2020

VIETNAM GENERAL CONFEDERATION OF LABOUR
TON DUC THANG UNIVERSITY
INFORMATION TECHNOLOGY FACULTY

REPORT ASSIGNMENT OF

DATA STRUCTURES AND ALGORITHMS 2 SUBJECT

TREE DATA STRUCTURES

Instructing Lecturer: Mr. NGUYEN QUOC BINH

Student's name: **HUYNH LE THIEN Y – 518H0320**

Class : **18H50204**

Course : **22**

HO CHI MINH CITY, YEAR 2020

ACKNOWLEDGEMENT

With deep affection, sincerity, we express deep gratitude to all individuals and agencies who have helped us in our study and research. First of all, I would like to express a special appreciation to Ton Duc Thang University's teachers for their conscientious guidance and advices throughout the last semester by gave me their modern outlook and meticulous supervision to carry out the job perfectly. Especially I would like to send my sincere thanks to Mr. Nguyen Quoc Binh has paid attention, help me to complete the report well over the past time. I would like to express my sincere gratitude to the leadership of Ton Duc Thang University for supporting, helping and facilitating me to complete the report well during the study period. With limited time and experience, this report cannot avoid mistakes. I'm looking forward to receiving advice and comments from teachers so that I can improve my awareness, better serve the practical work later. I sincerely thank you!

THE PROJECT WAS COMPLETED AT TON DUC THANG UNIVERSITY

I pledge that this is a product of our own project and is under the guidance of Mr. Nguyen Quoc Binh. The content of research, results in this subject is honest and not published in any form before. The data in the tables used for the analysis, comment, and evaluation were collected by the authors themselves from various sources indicated in the reference section. In addition, many comments and assessments as well as data from other authors and organizations have been used in the project, with references and annotations. If any fraud is found, I am fully responsible for the content of my project. Ton Duc Thang University is not involved in any copyright infringement or copyright infringement in the course of implementation (if any).

Ho Chi Minh city, May 20th 2020

Author

(Signed)

Huynh Le Thien Y

EVALUATION OF INSTRUCTING LECTURER

Confirmation of the instructor

Ho Chi Minh city , month day year

(Sign and write full name)

The assessment of the teacher marked

Ho Chi Minh city , month day year

(Sign and write full name)

TABLE OF CONTENTS

LIST OF TABLES, DRAWINGS, GRAPHICS

CHAPTER 1: INTRODUCTION

1.1 Concept

In graph theory, a tree is a connected graph and has no cycles. The tree in which one vertex is selected as the root and the edges are oriented as the direction of the paths from the root to the other vertices. In this case, any two vertices are connected, implying that they are parent-child.

Tree in tree data structure simulates tree in graph theory. However, trees in data structures are effectively applied in many algorithms.

A tree data structure is a data structure where a collection of nodes contains a certain value and is linked together in a parent-child relationship. This data structure tree starts at the root node and connects to the other nodes through edges with the rule that no edge is duplicated and no edge is pointing back to the root node.

a. Node

Each node in a tree may not have or have some children, the children have a higher level than it is (different from the natural tree, the tree in the data structure grows from the top down). A node with children is called the parent node of child nodes. A node has at most one parent node.

Root node

The root node has no parent node. From the root node will connect to all other nodes with edges or links.

Leaf node

Any node without a child will be called a leaf node.

Branch node

A node with at least one child will be called a branch node.

b. *Subtree*

A subtree is a part of a tree data structure that itself is a tree. Any node in a tree named X, together with the nodes below it, forms a subtree of X.

c. *Example*

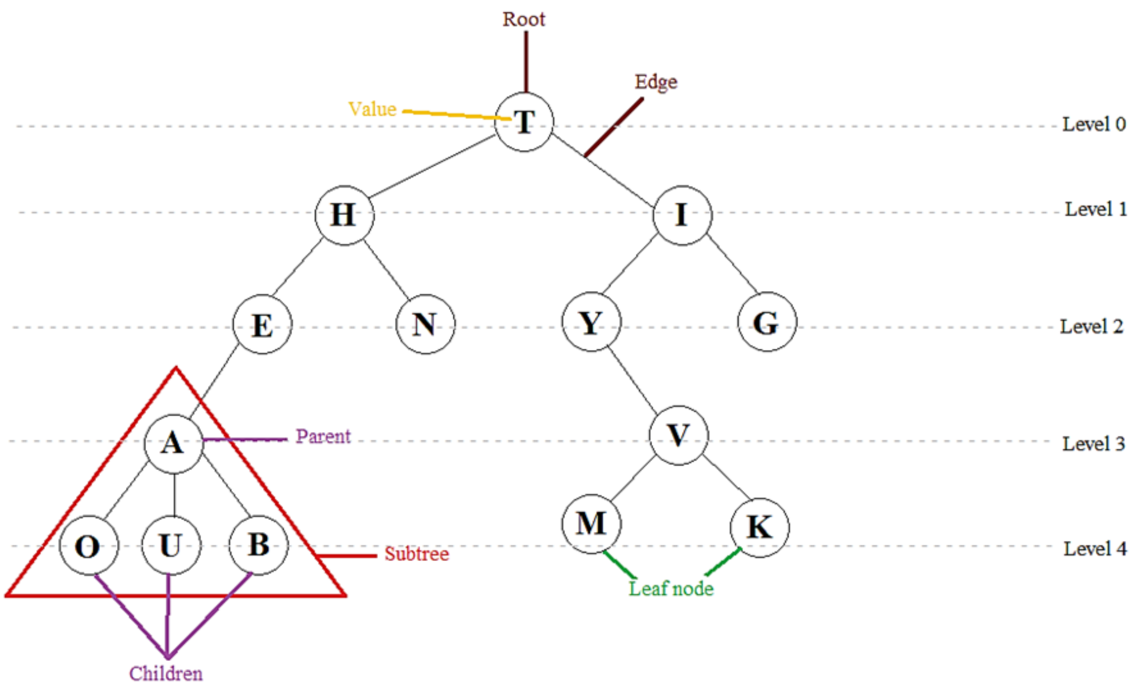


Figure 1. *Example of a tree data structure*

d. *Ordered tree*

There are two basic types of tree structure: unordered tree and ordered tree. An unordered tree is a tree with a tree structure, in which between the children of a node, there is no order. And vice versa, a tree in which the children of a node follow a defined order is called an ordered tree. Ordered trees have many applications in the tree data structure.

1.2 Binary search tree (BST)

In this subject, I learned about several types of trees in tree data structures. One of them is the binary search tree.

Binary search tree is a binary tree (the tree in which each node has no more than two children) on which each node is assigned a key such that for each node k:

- All keys on the left subtree must be smaller than the key on node k
- All keys on the right subtree must be larger than the key on node k

Let's look at an example of a binary search tree.

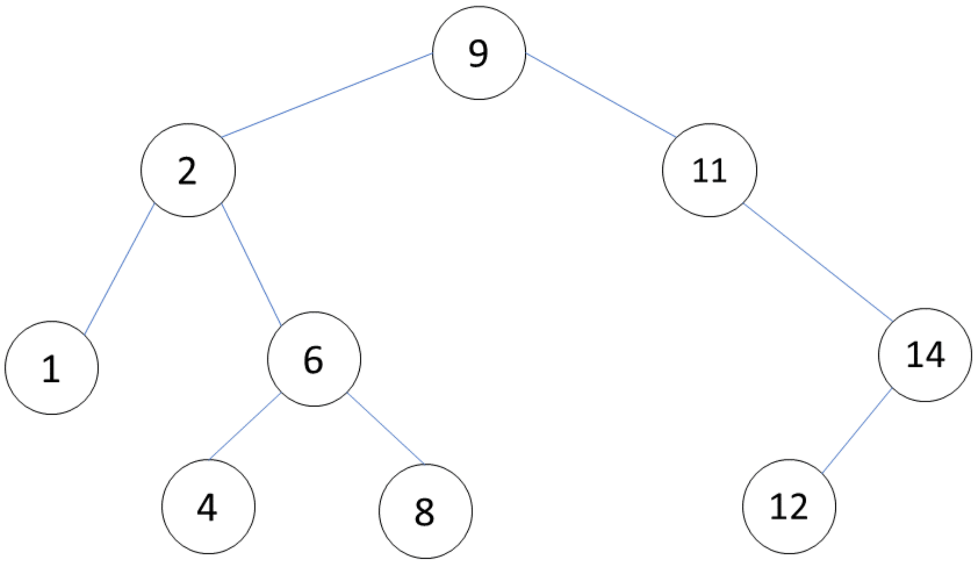


Figure 2. *Example of Binary search tree*

Some operations used in BST

i. <u>Searching</u>

To search for a node, we begin checking at the root node.

If the tree has nothing, the key we seek will not exist.

Otherwise, if the key we find is equal to the key at the root, the search is successful and we will return the root node.

If the key we find is smaller or bigger than the key at the root node, we will search the left subtree or the right subtree respectively.

This process repeats until subtree has nothing left.

If found, we will return the node, otherwise the key we are searching does not exist.

ii. Insertion

A new key is always inserted at leaf.

We start searching a key from root till we hit a leaf node.

Once a leaf node is found, the new node is added as a child of the leaf node.

iii. Deletion

When deleting a node, there are 3 cases:

- *If the node is a leaf node:* Simply delete that node.
- *If the node wants to delete only 1 child:*
    - Copy its children
    - Delete the node we want
    - Replace it with its children
- *If the node that wants to delete has 2 child nodes:*
    - Find its predecessor or its successor to swap it with the one that needs to be deleted and then delete it.
    - Because predecessor node or successor node have fewer than two children, deleting them is attributable to the previous two cases.

1.3 Red-black tree

Overview

The red-black tree is a binary search tree. It is a self-balancing tree and is used in computer science. Color elements (RED, BLACK) are the basis for balance. Its original structure was launched in 1972 by Rudolf Bayer. He called them the "B-tree" and the present name is given since 1978 by Leo J. Guibas and Robert Sedgewick. It is a complex structure but gives good time results in the worst case scenario. The operations on them are like search (insertion), insert (insertion), and delete (removal) in O (log n) time, where n is the number of tree elements.

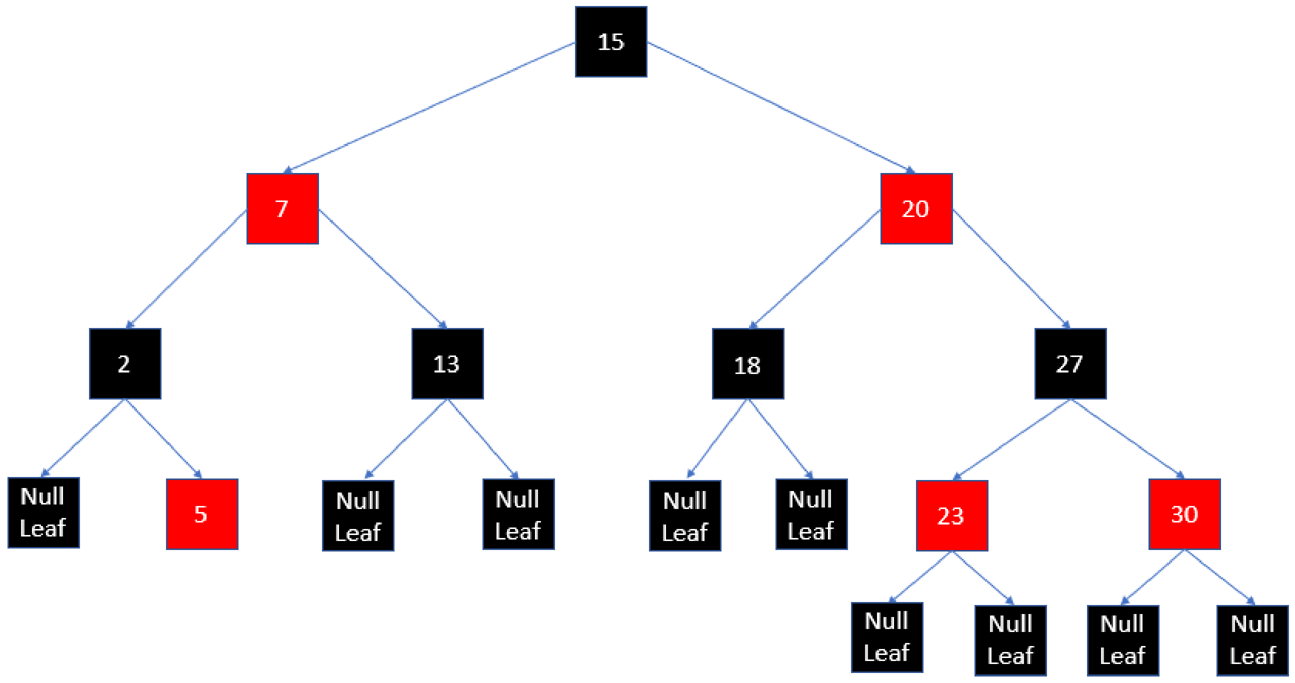


Figure 3. *Example of Red-black tree*

1.4 B-tree

Overview

When data is to large to fit in main memory, number of disk accesses is very important. One access for external retrieval required thousands of times greater than for internal retrieval. The goal is to scale back the amount of disk accesses, since each access for external retrieval takes so much time compared to internal one. The concept is to make tree's height as small as possible. That's where B-Trees become useful.

In computing, a B-Tree is a tree that enables searching, sequential access, insertion, deletion in logarithmic time. It's a generalization of the Binary search tree, din which a node can have more two children. Unlike a self-balancing Binary search tree, B-Trees are optimized for reading and writing large data systems. It's commonly utilized in system of large database or file. The B-Tree was created by Rudolf Bayer and EdMcCreight.

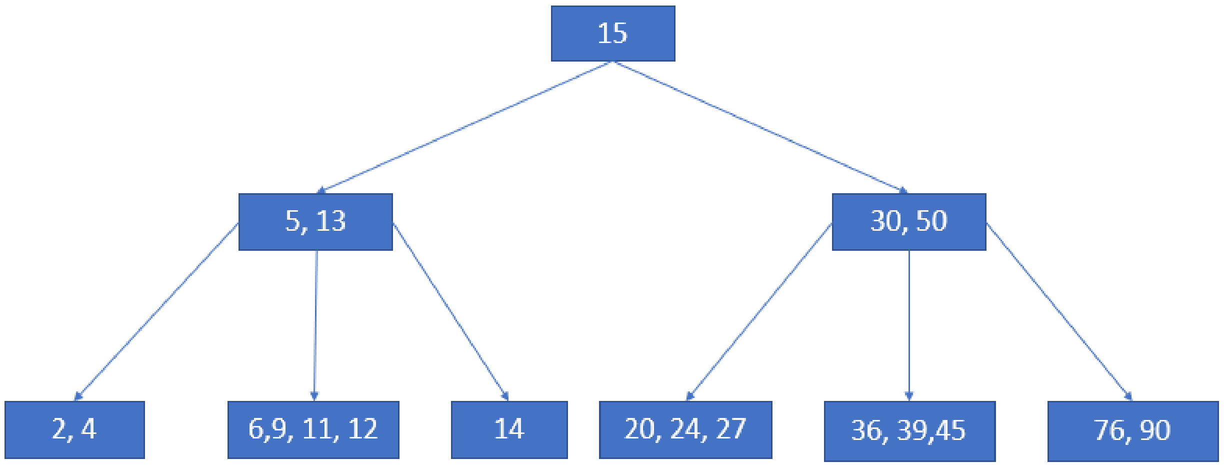


Figure 4. *Example of B-tree*

CHAPTER 2: RED-BLACK TREE

2. 1 Definition

The red-black tree is similar to a binary search tree in that it is made up of nodes and each node has at most 2 child nodes. However, there are properties specific to red-black tree:

1. Each node is red or black. All leaf nodes are black and don’t contains value (NULL).
2. The root node is always black.
3. If a node is red, then its parent or child can’t be red.
4. Path from a specific node to any of its children contains the same number of black nodes.

Black height is the number of black nodes on the path from node x to leaf (excluding x).

The red-black tree with N internal nodes have a maximum height of 2logn (N+1).

The height of the tree <= 2 * black height.

Red-black tree has infite height is $O(logn)$ and it take $O(logn)$ time to search, add or remove a node in the tree. An examle of a red-black tree is shown below:

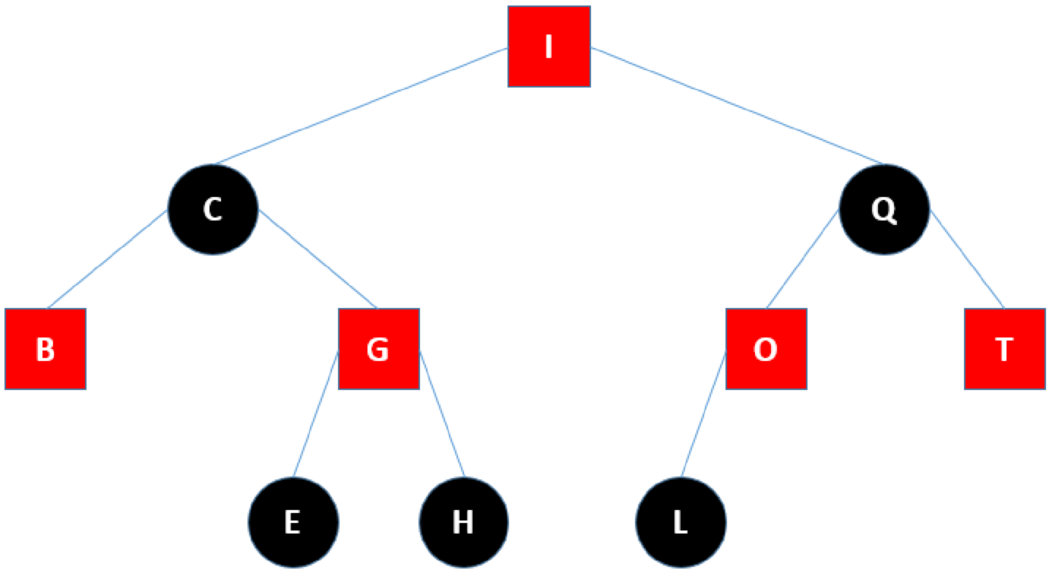


Figure 5. *Examle of a red-black tree*

2. 2 Operations

Just like with a binary search tree, red-black tree can be represented by the following operations:

- Insert a key (Insertion)
- Check if the key exists in the tree (Search)
- Delete a key (Removal)
- Print all keys sorted in order (Print)

Search and Print operations of a red-black tree are similar to the Search and Print operations of a binary search tree, due to two reasons:

- Red-black tree is binary search tree.

- An operation that does not change the structure of a tree will not affect the satisfaction of its properties.

Therefore, in this report, we only mention two operations: Insertion and Removal. It is necessary to use the tree rotation described below:

Right rotation

Suppose R0 is the root node of a tree and has a left child as L.

The rotation convert L to the root of the tree and the old R0 to become a right child of L. Then L's former right child, LR, is detached from L, in order to preserve the properties of the binary search tree, we must let R0's left point to LR.

Left rotation

Suppose R0 is the root node of a tree and has a right child as R.

The rotation convert R to the root of the tree and the old R0 to become a left child of R. Then R's former left child, RL, is detached from R, in order to preserve the properties of the binary search tree, we must let R0's right point to RL.

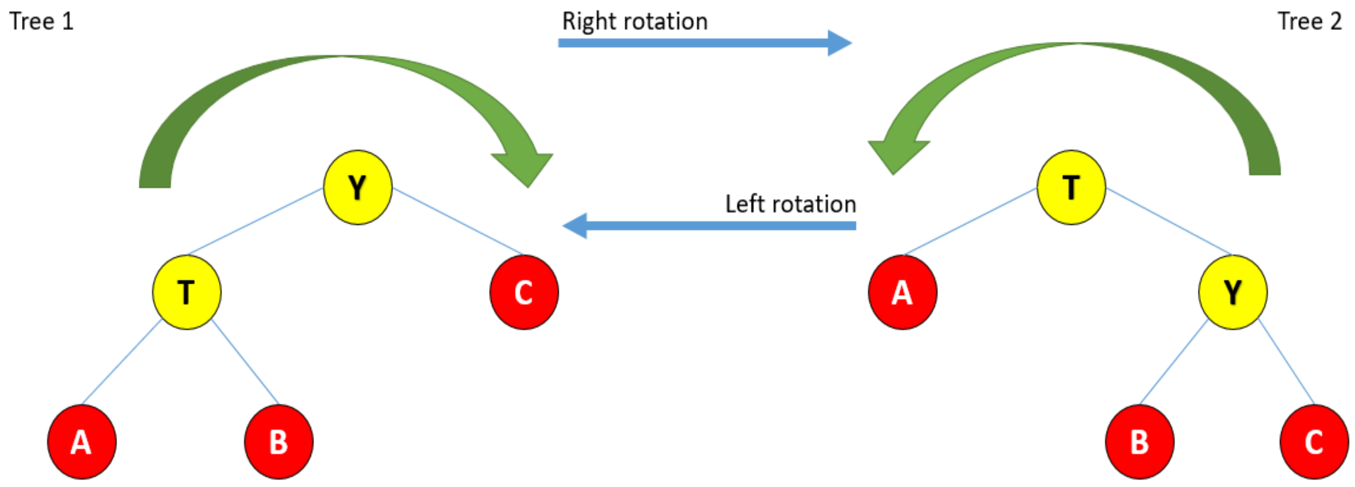


Figure 6. *The right rotation turns tree 1 into tree 2 and the left rotation changes back. Sub-trees of A, B, C can be NULL*

a. Insertion

The goal of the insertion is to insert key K into tree T, maintaining T's red-black tree properties.

If T is an empty tree, replace it with a black node containing K. This ensures that the root property is satisfied.

If T isn't an empty tree, use the insertion algorithm of the binary search tree to insert a node containing K to the tree. The difference is that the leaf node of a red-black tree is null, but the insertion algorithm of a binary search tree inserts the key as a leaf node, so instead of that, this insertion will replace the leaf node with the new node contains K and add to that node 2 black leaf nodes that do not contain value. Then color that node to red. This might wrong for the property that a red node's parent is black, though. So, now we've three potential cases that we would must cater to.

- K's parent (P) is black
- K's parent (P) is red (so it can't be the root of the tree) and K's uncle (U) is red
- K's parent (P) is red and K's uncle (U) is black or null

<u>Case 1</u>

If P is black, then the property of the tree is kept, so no further action is needed.

<u>Case 2</u>

In case both of K's parents and uncles are red, it can be said that K's grandparent (G) is black, we will recoloring P, U, G. The result is that P and U's colors will become red and G's will will become black. (Unless G is the root, in that case, to keep the root property intact, we leave G black.).

When we recoloring, it may have created a same situation between G and G's parent. If that’s the case, then we recursively start recoloring at G and G's parent (instead of K and K's parent).

The Figure 6 shows the case on before and after rectified.

K: new node which need to be inserted.

P, U, G are K's parent, uncle and grandparent respectively.

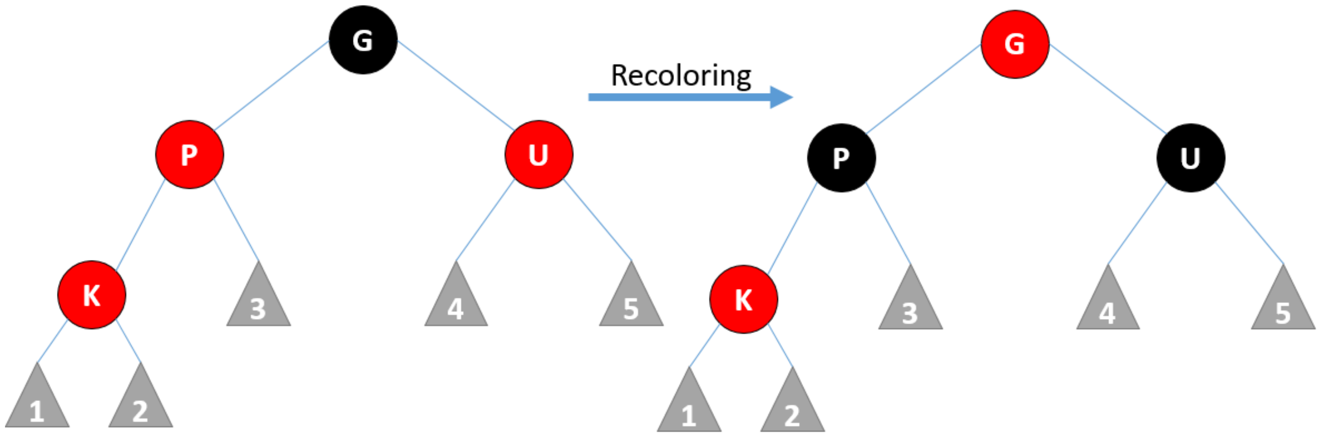


Figure 7. *Image shows the case on before and after recoloring in case 3.2 of insert operation*

Case 3

If K's uncle U is black or null, then we will do a restructuring of K, P, G. To do a restructuring, we first put K, P, and G in order; let's call this order A, B, and C. We then make B the parent of A and C, color B black, and color A and C red.

There are two possibilities for the relative ordering of K, P, and G. The way a restructuring is done for each of the possibilities is shown below.

- Parent P is red, but uncle U is black, K is the right child of P, and P is the left child of G.
    - In this case, performing a left rotation converts the roles of the new node K and the parent P.

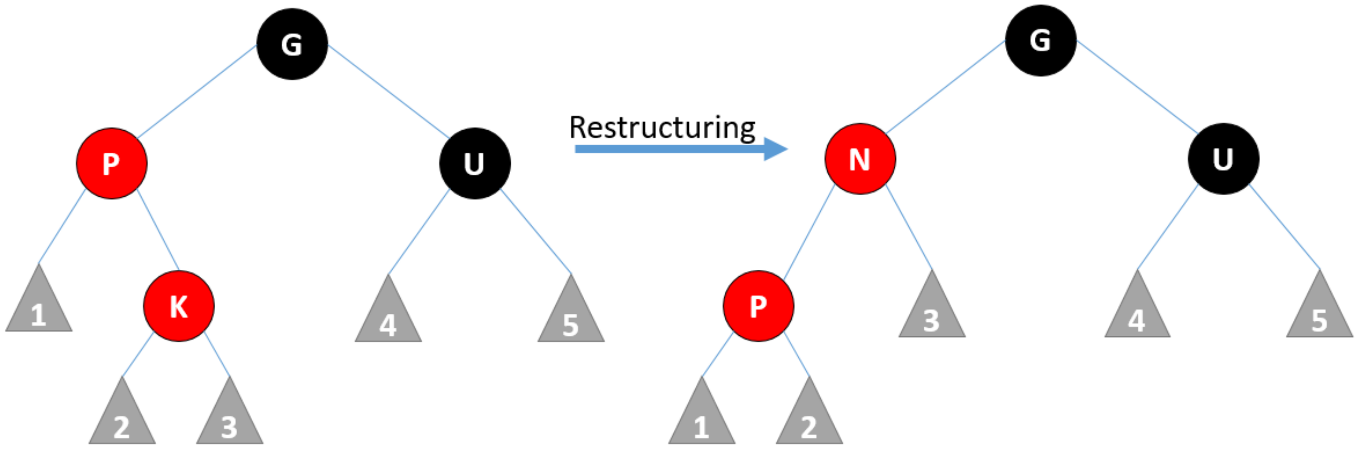


Figure 8. *Image shows the case on before and after restructuring in case 3.1 of insert operation*

- Parent P is red, but the U is black, K is the left child of P, and P is the left child of G.
    - In this case, a right rotation at node G.

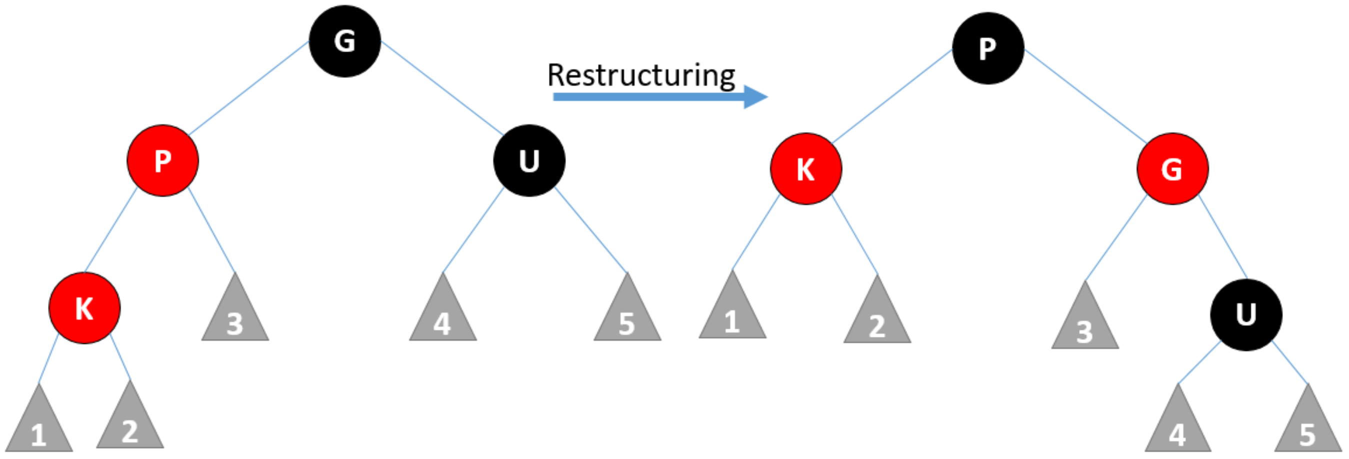


Figure 9. *Image shows the case on before and after restructuring in case 3.2 of insert operation*

If U is null, then in the case of the two possibilities above, the subtree of U (numbered 4, 5) will be assigned as null.

b. Removal

The removal is comparable in feel to the insertion, but more complicated.

These are the removal steps:

- Use binary search tree deletion algorithm.
- In case of deleting a node with at most one child (not a null leaf).
- If we delete a red node, we can make sure its child is a black node. All paths passing through a deleted node simply remove a red node so property 4 does not change.
- In addition, both the parent and child nodes of the deleted node are black, so property 3 remains the same.

Another simple case is when deleting a black node with only one red child. When deleting the node, the properties 3 and 4 are broken, but if the color of the child node is black again, they are restored.

The complex case occurs when both the node is deleted and its child node is black. We will start by replacing the deleted node with its child.

Call the deleted node as X and the replacement child node as C. Also, we call C’s parent as P, C’s sibling as S (another child of the new parent node) and S_L indicates S's left child, and S_R indicates S's right child (they exist because S cannot be a leaf).

If both X and C are black nodes, after deleting X, the path passing through C will decrease a black node. Therefore, in violation of Property 4, the tree needs to be balanced. There are the following cases (We convention C is the left child of father P for cases 2,5,6. If it is a right child, left and right will be interchangeable) :

Case 1

C is the new root. In this case we stop. We have removed a black node from all paths and the new root is black. No properties have been violated.

Case 2

S is red. In this case swap the colors of P and S, and then rotate left at P, it will make S become C's grandparent node.

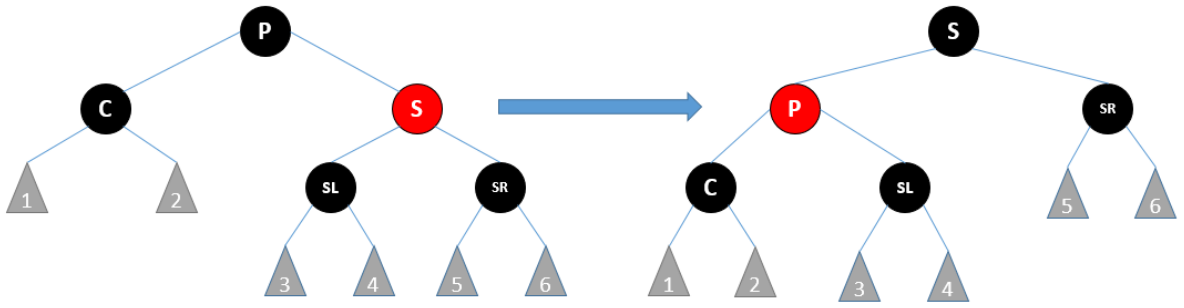


Figure 10. *Image shows the before and after restructuring in case 2 of removal operation*

Case 3

P, S, and S's children are black. In this case, we recoloring S as red. As a result, every path through S has less than a black node. Because deleting X makes all paths passing C reduce a black node, so after coloring the S, they are equal. However, all paths passing through P now have less than one black node compared to roads not passing P, so property 4 (Path from a specific node to any of its children contains the same number of black nodes) will be violated. To fix it we rebalance at P.

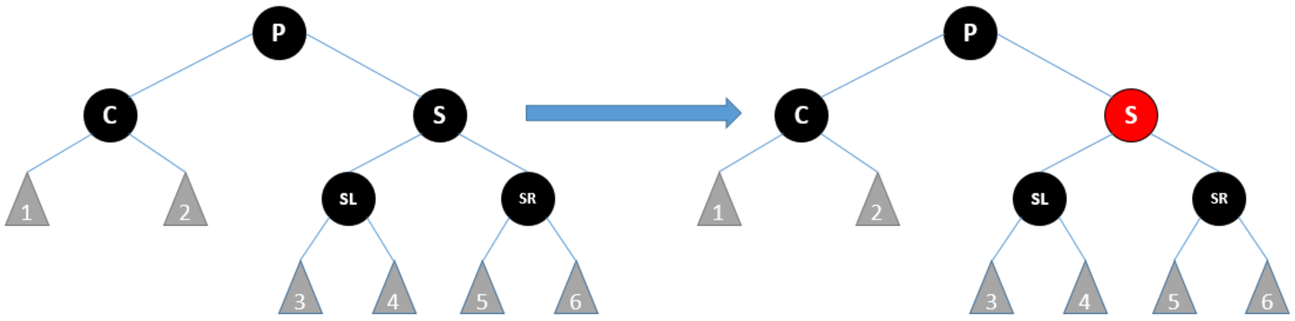


Figure 11. *Image shows the before and after recoloring in case 3 of removal operation*

Case 4

S and S's children are black but P is red. In this case, we reverse the colors of S and P. This doesn't affect the number of black nodes on the paths that don't pass through

C, but adds a black node on the paths that pass through C, instead of the black node X which is deleted on these paths.

Case 5

S is black, S's left child is red, S's right child is black, and C is P's left. In this case we rotate right at S, then S's left child becomes S's parent and C is its new sibling. Then we switched colors of S and its new parent. All paths will have the same number of black nodes, but now that C has a black sibling whose right child is red, we move to Case 6.

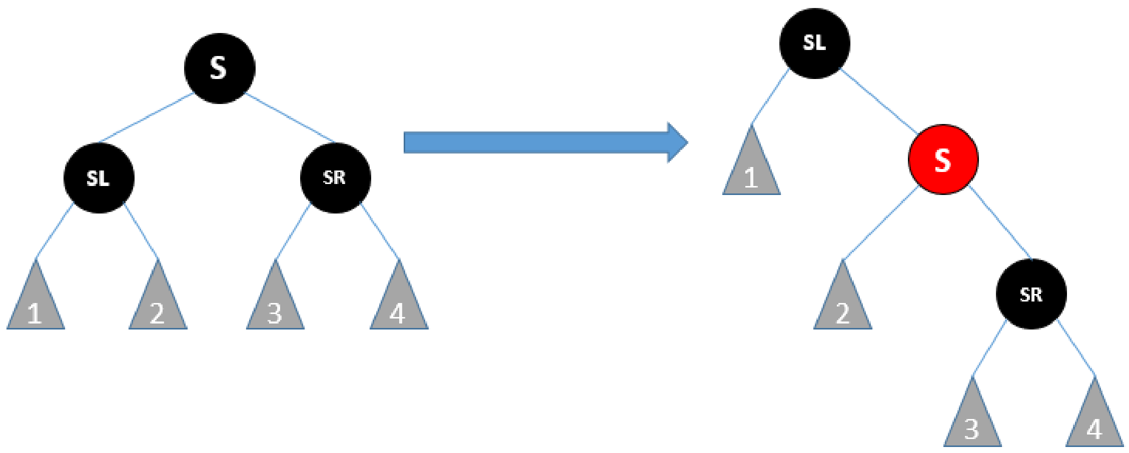


Figure 12. *Image shows the before and after rotation in case 5 of removal operation*

Case 6

S is black, S's right child is red and C is the left child of P. In this case we rotate left at P, then S becomes parent of its right child and P. We change colors of P and S, and assign the right child of S as black. C now has an predecessor black node: either P has just been black, or P has been black or S has just become its grandparent is black node. Thus the way through C has a black node.

Meanwhile, with a path that does not pass through C, there are two possibilities:

- Go through the sibling node of C.

  - Then both before and after rotation it must go through S and P, When changing colors, these two nodes exchanged colors. So this path has not changed the number of black nodes.
- Go through S's right child.
  - At that time before rotating it passed through S, S's parent, and S's right child, but after rotate it just passed S's and S's right child.
  - At this point, S is colored in the old color of P while S's right child changes color from red to black. As a result, the number of black nodes on this path does not change.

Thus, the number of black nodes on the paths is unchanged. Therefore properties 3 and 4 have been restored.

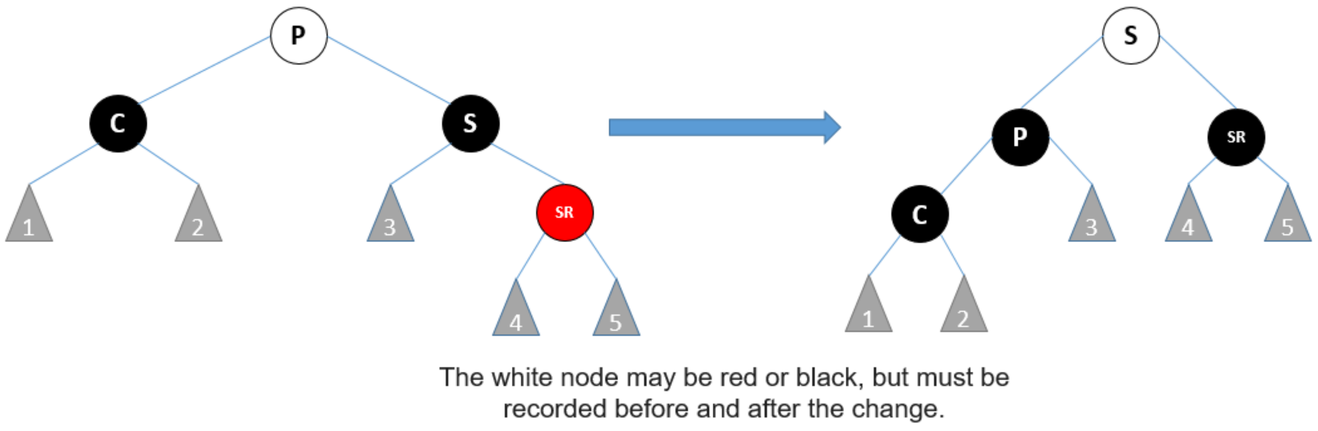
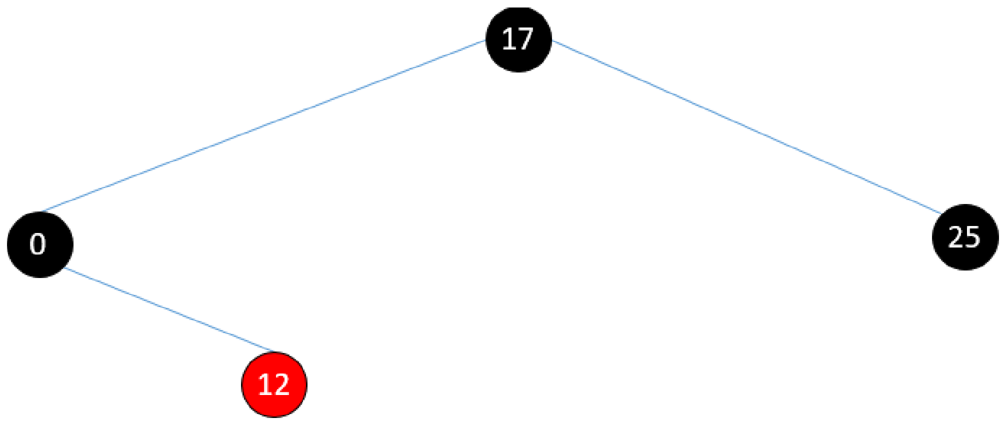

P
C
S
SR
1
2
3
4
5
S
P
SR
C
3
4
5
1
2
The white node may be red or black, but must be recorded before and after the change.

Figure 13. *Image shows the case on before and after restructuring in case 6 of removal operation*

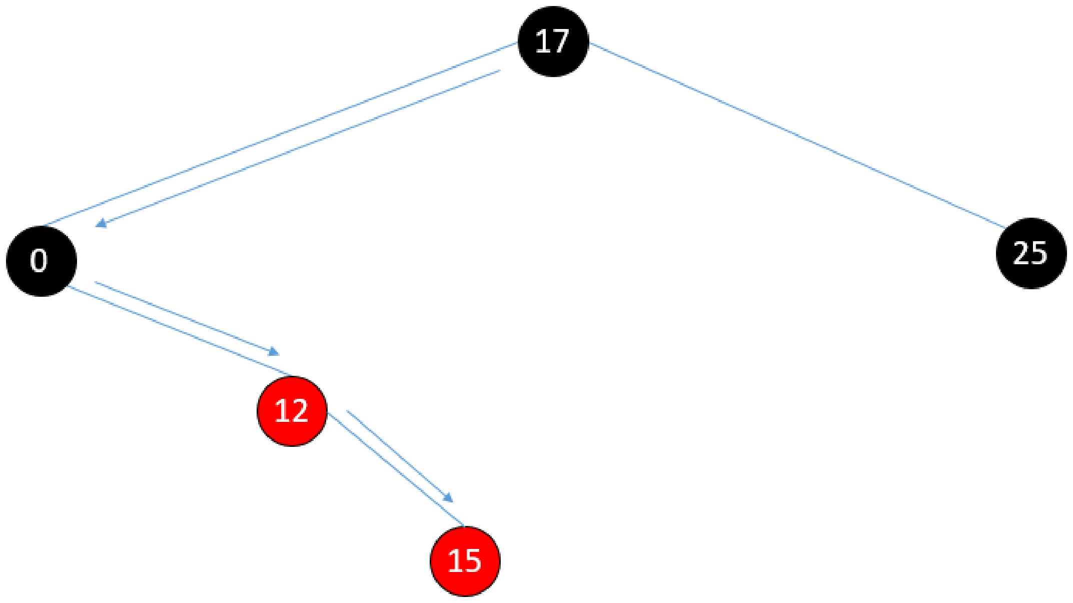
Example for the operations

<u>Insertion:</u>

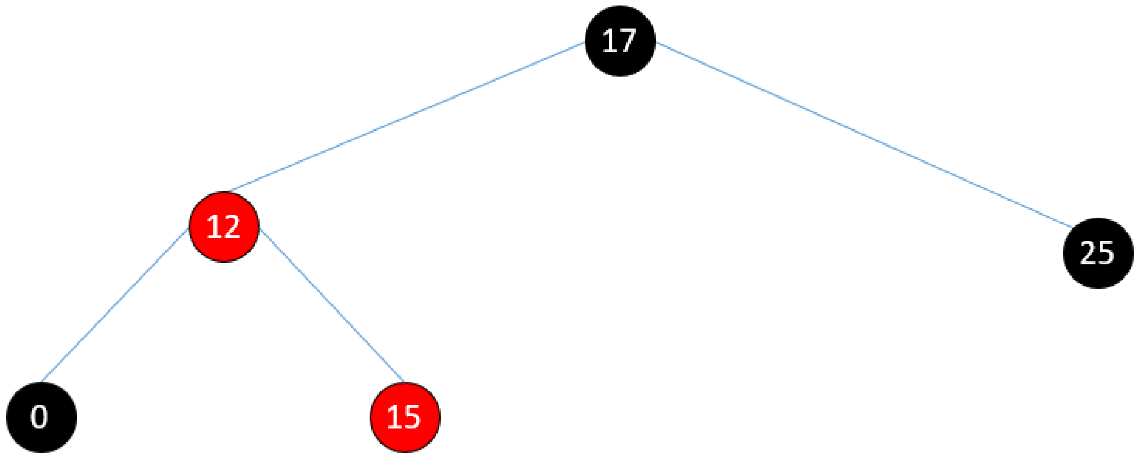
Start with this tree.



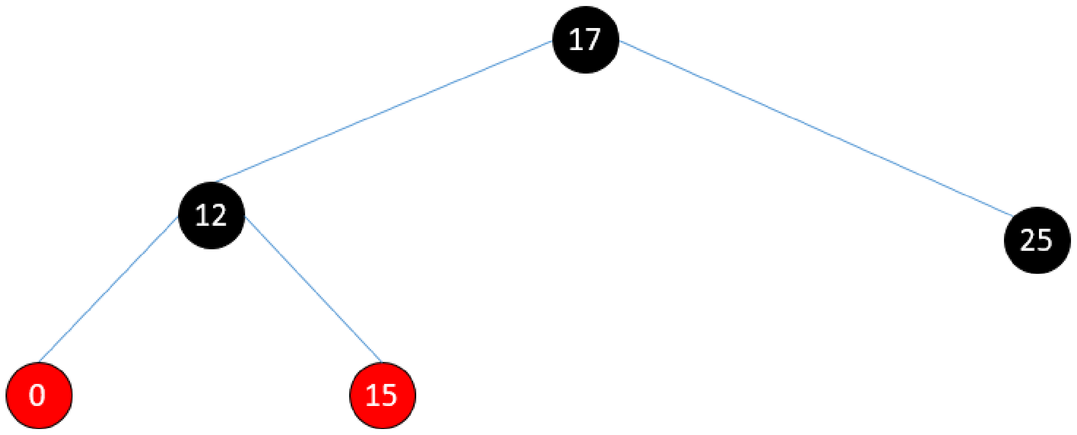
We add 15 to the tree.



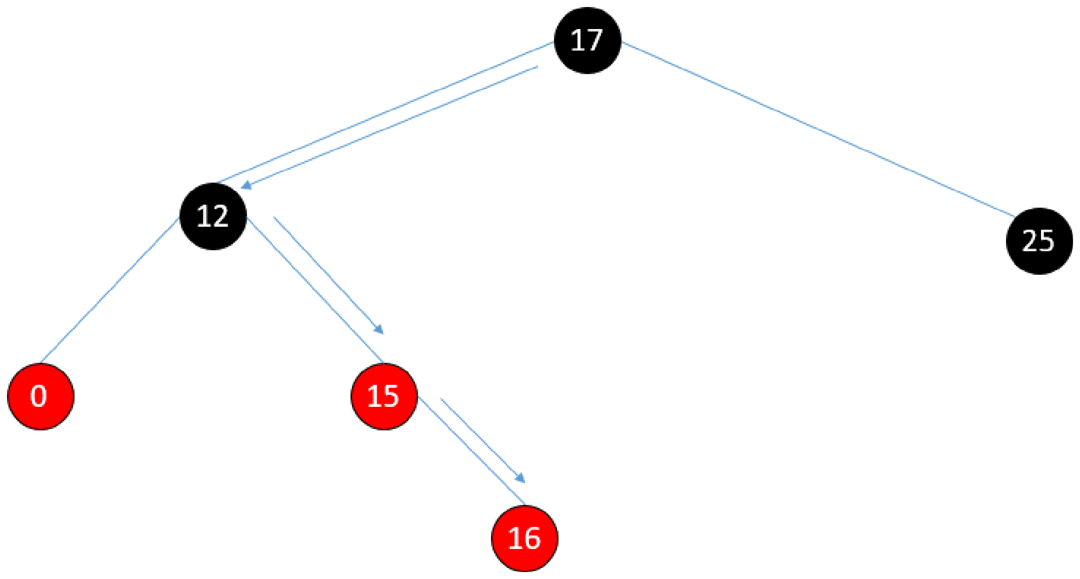
Left rotation takes place. 12 goes up one level and 0 becomes the left child of 12.



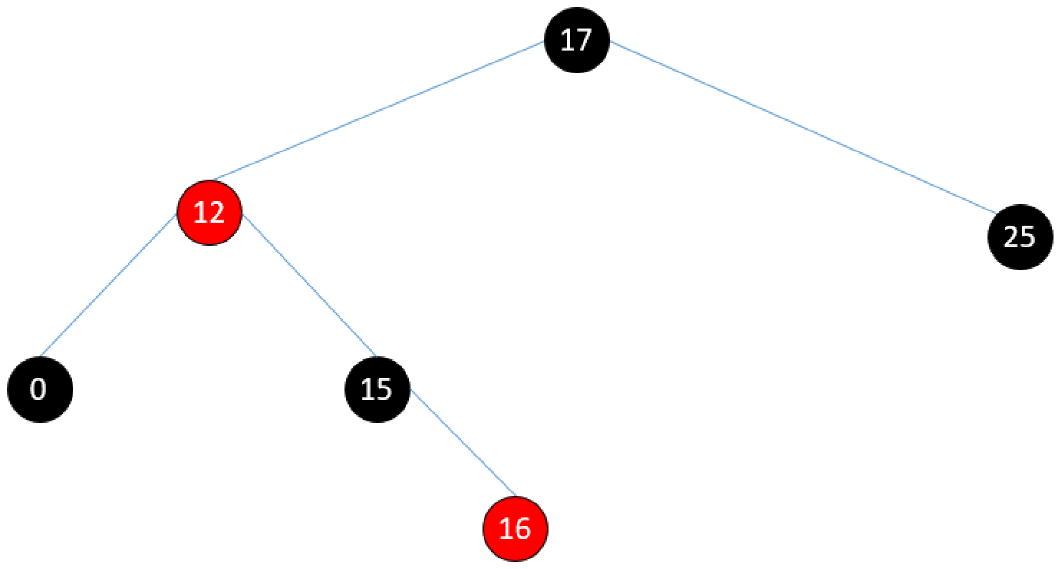
In the next step some color adjustments are made so that the color properties are not violated.



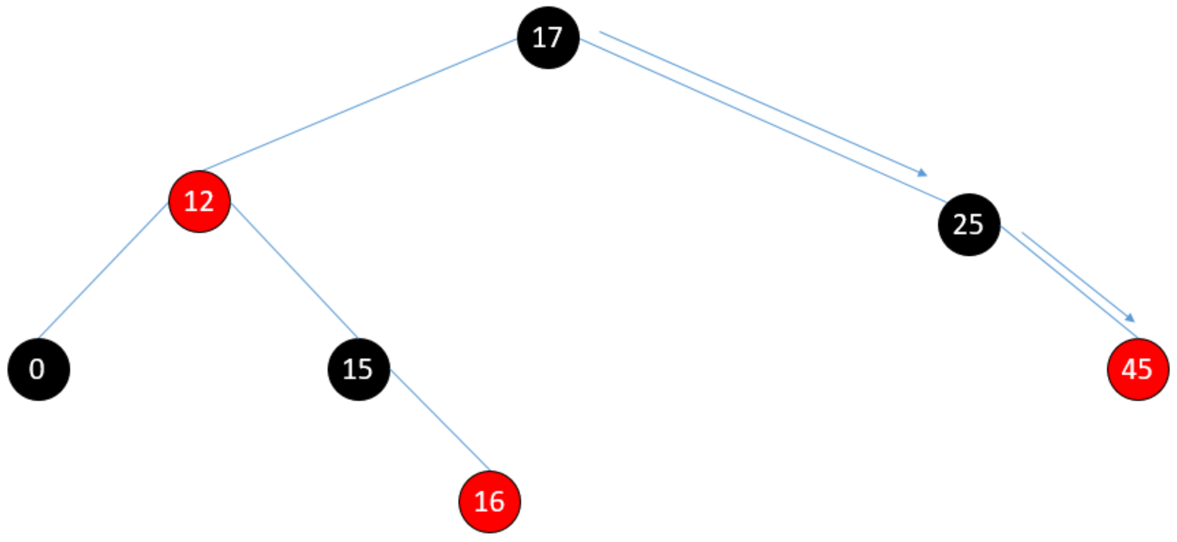
Next, add 16 to the tree.



Color adjustment takes place so that the color properties are not violated.

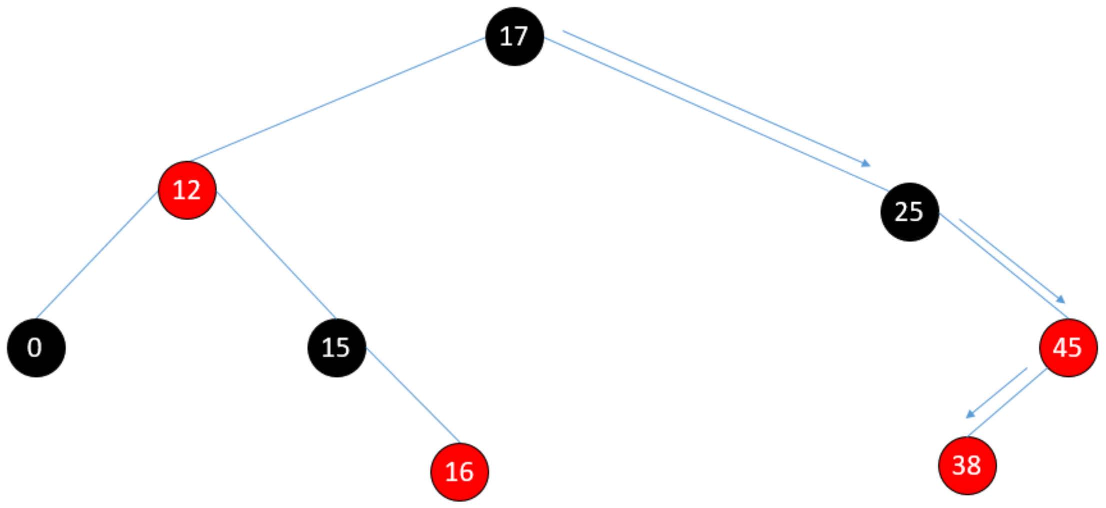


Now, add 45 to the tree.

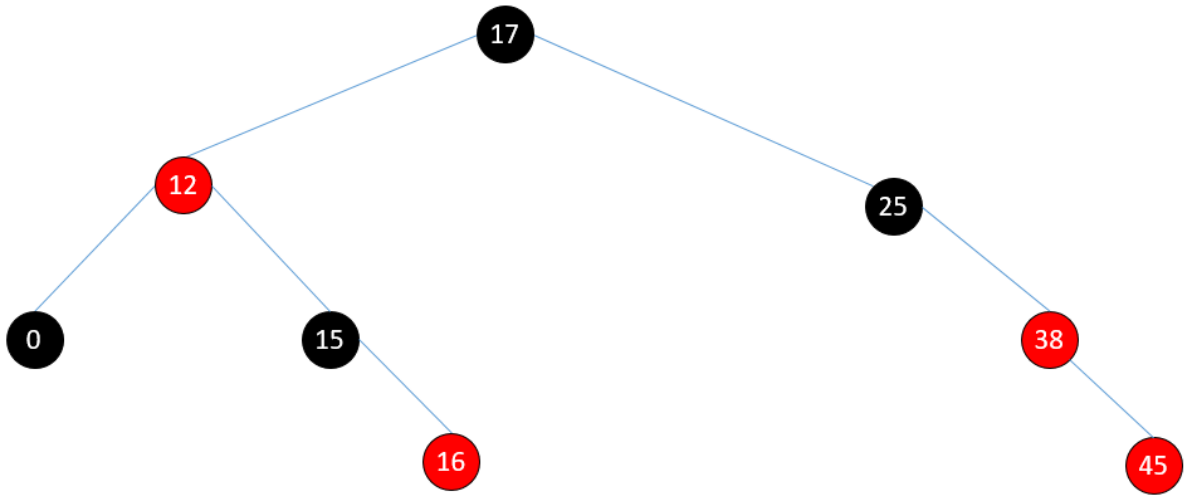


The insertion of this node does not violate any of the red-black tree proprieties, so there are nothing to do.

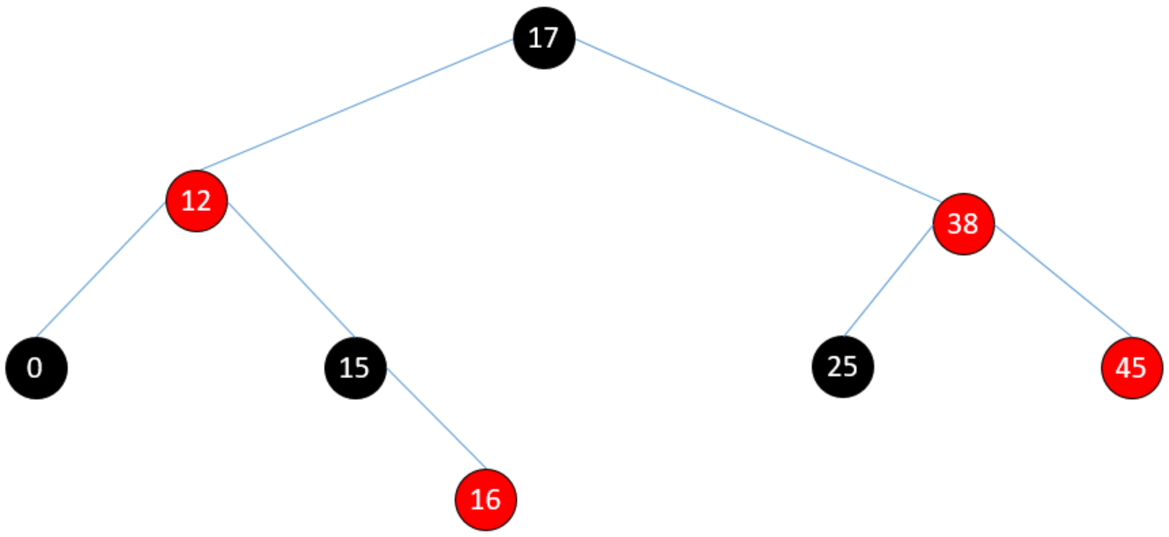
Now, we add 38 to the tree.



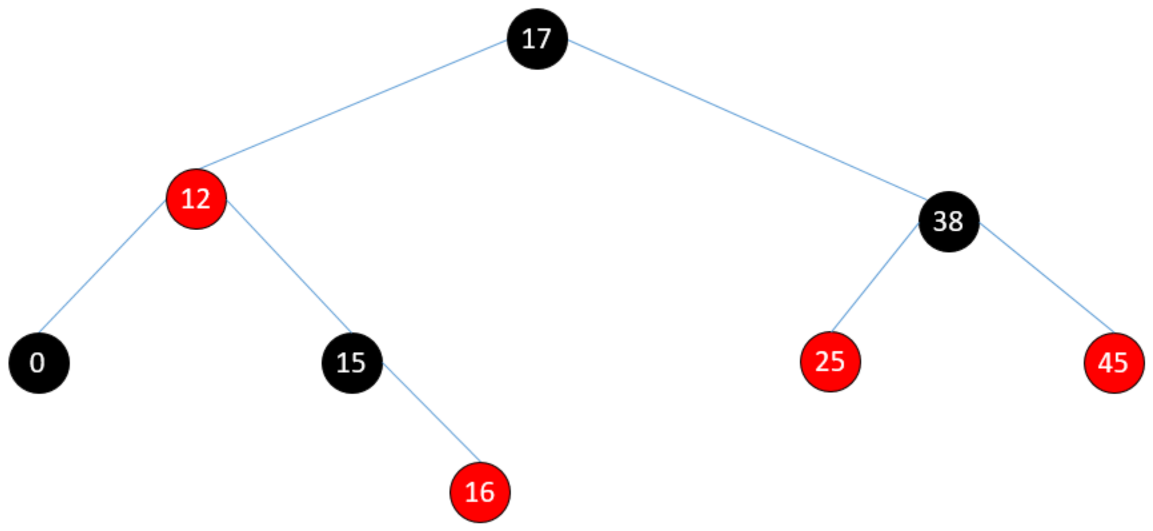
Right rotation takes place, then 38 goes up one level and 45 becomes 38’s right child.



Left rotation takes place, in which 38 goes up one more level and the number 25 becoming 38's left child.

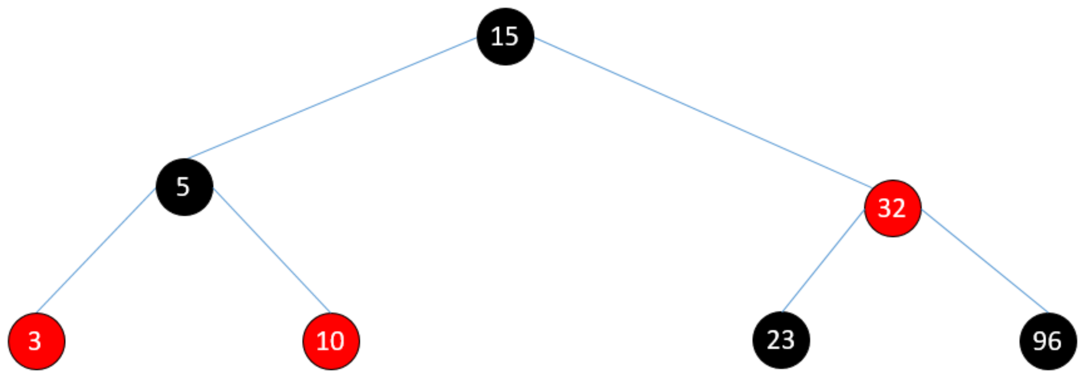


Next step is recoloring so that the color properties are not violated.

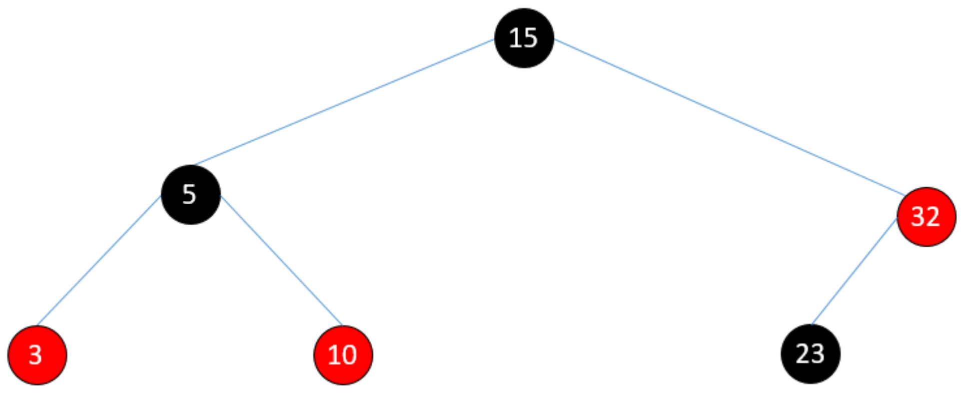


Removal:

Start with this tree.



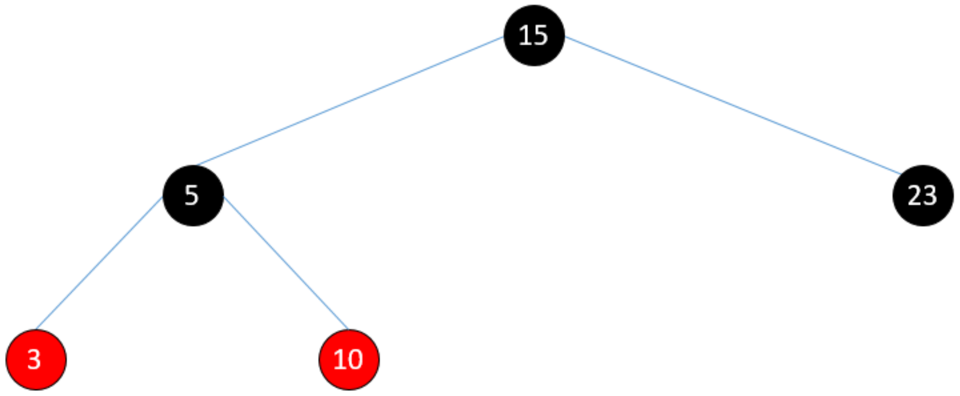
We delete number 96.



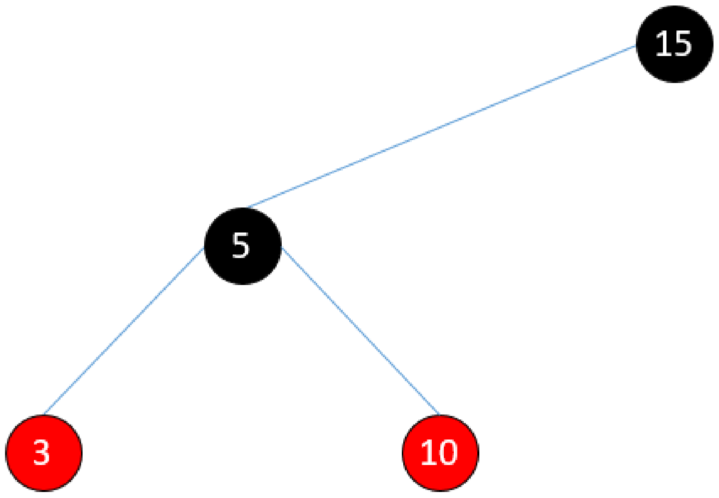
After deleting number 96, the node with key 32 remains with two right side black children that belonged to number 96. (Every leaf is black, but those leafs are NULL and are not represented in these drawings.)

We recolor in order to fix the violation:

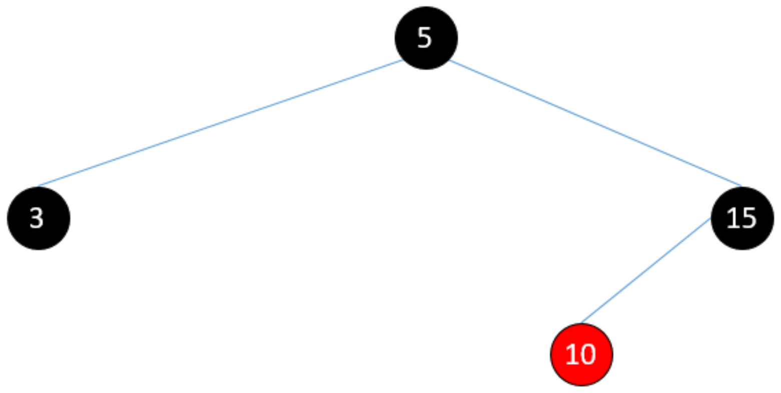
Next step we delete number 32. There aren't double black node emerges as in the previous. So the structure of the tree doesn't need to be changed.



Next step we delete number 23. It end up with a tree like this:



The tree is unbalanced therefore we need to restructure it by using right rotation at number 5. Also some recoloring is needed after that. The final tree looks like this:



2. 3 Implementation

Every node of the tree contain these attribute: key, parent, left child, right child, color. A red-black tree's node structure would be:

```
class Node {
        int key; // holds the key
        Node parent; // pointer to the parent
        Node left_child; // pointer to left child
        Node right_child; // pointer to right child
        int color_num; // 1 . Red, 0 . Black
}
```

a. Insertion

The following pseudocode combine all cases in the previous chapter about insertion operation. Using a loop and if-else condition to perform the insert operation.

```
RedBlack-INSERT(T, K)
    BinarySearchTree-INSERT(T, K)                // standard BST insertion
    while K.parent.color_num == RED
```

```
        if K.parent == K.parent.parent.right_child
          U = K.parent.parent.left_child                    // Uncle
          if U.color_num == RED                             // Case 2
            U.color_num = BLACK
            U.parent.color_num = BLACK
            U.parent.parent.color_num = RED
            K = K.parent.parent
          else if K == K.parent.right_child                 // Case 3.1 and 3.2
              K = K.parent
              RIGHT-ROTATE(T, K)
            K.parent.color_num = RED
            K.parent.parent.color_num = BLACK
            LEFT-ROTATE(T, K.parent.parent)
      else (same but exchange "right" and "left")
    T.root.color_num = BLACK
```

b. Removal

The following pseudocode combine all cases in the previous chapter about removal operation. Using a loop and if-else condition to perform the remove operation.

```
RedBlack-DELETE(T, C)
    BinarySearchTree-DELETE(T, C)
    while C ≠ T.root and C.color_num == BLACK
        if C == C.parent.left_child
            S = C.parent.right_child
            if S.color_num == RED                              // case 2
                S.color_num = BLACK
            C.parent.color_num = RED
            LEFT-ROTATE(T, C.parent)
            S = C.parent.right_child
        if S.left_child.color_num == BLACK
            and S.right_child.color_num == BLACK    // case 3 , case 4
            S.color_num = RED
            C = C.parent
        else if S.right_child.color_num == BLACK
                S.left_child.color_num = BLACK          // case 5
                S.color_num = RED
                RIGHT-ROTATE(T, s)
                S = C.parent.right_child
            S.color_num = C.parent.right_child            // case 6
            C.parent.color_num = BLACK
            C.right_child.color_num = BLACK
            LEFT-ROTATE(T, C.parent)
            C = T.root
    else (same but exchange "right" and "left")
C.color_num = BLACK
```

CHAPTER 3: B-TREE

3.1 Definition

Some of the important properties of B tree are:

- B-tree node has more than two child nodes.
- A node of B-tree may contain more than one key.

When using B-trees, we predetermine an integer t as a minimum size. Each internal node of a B-tree (except the root node) has t-1 to 2t-1 keys. Coefficient 2 ensures that nodes can be split or combined.

- Every leaf node has the same depth.
- The number of keys for each node is within a predetermined range. This interval is determined by the parameter $t \geq 2$.
- Each internal, non-root node has at least t-1 key. If the tree is not empty, the root node must have at least one key.
- Each node has a maximum of 2t-1 keys.

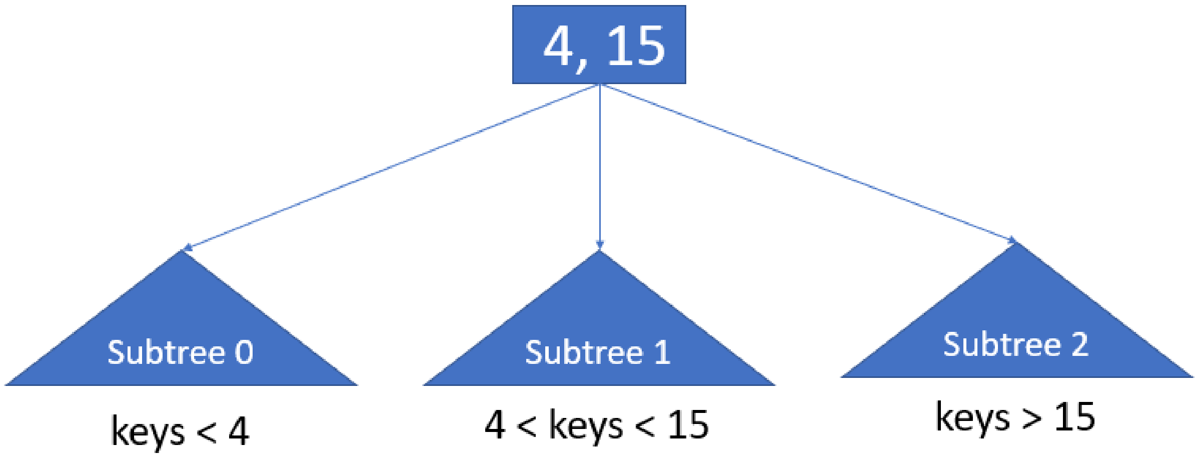


Figure 14. *B-tree in general*

B-tree with n keys take $O(logn)$ time to add or remove a key in the tree.

3.2 Operations

a. Searching

B-tree searching is similar to BST searching. For example, a B-tree as shown below:

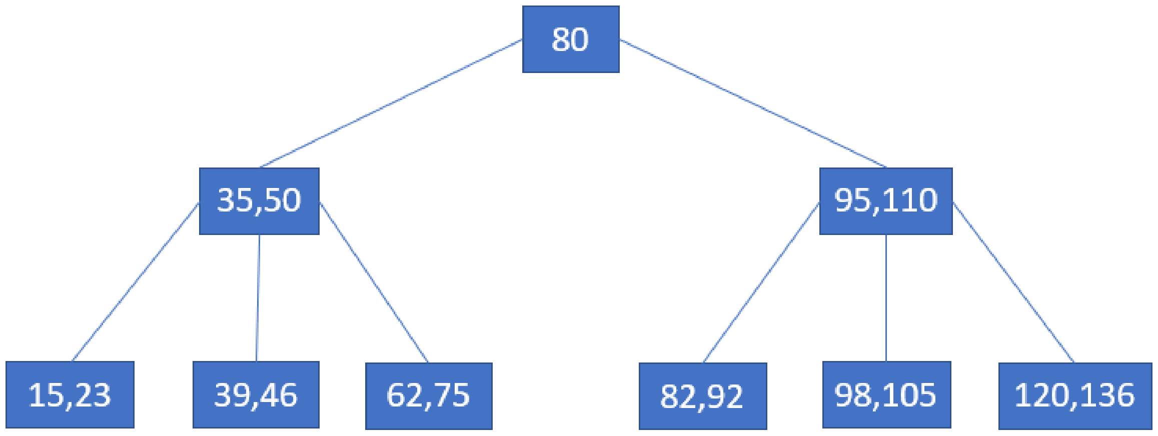


Figure 15. *Example for searching in B-tree*

If we find node 46 in that tree, the process will take place as follows:

1. Compare node 46 with root 80, because 46 <80, so we move to the subtree to the left of node 80 to continue searching.

2. Because 35 <46 <50, we move to the right subtree of 35.

3. 46> 39, so we move right. Compare 46.

4. Found it. Returns results.

The searching operation of a B-tree is based on the height of the tree. This algorithm has the complexity $O(n)$ to search for any node in the B-tree.

b. *Insertion*

The insertion is performed at the leaf node. The following is the process of inserting a node into the B-tree.

1. Traversal the B-tree to find a suitable leaf node location to insert the node into.
2. If the leaf node contains fewer than t-1 keys, insert new nodes in ascending order.
3. If the leaf node contains t-1 keys then:
   - Insert a new key in ascending order.
   - Take out the middle key of that node.

- Divide the keys in that node into 2 new nodes with t-1 keys.
- Push the middle key just taken out, into its parent node.
- If the parent node has t-1 keys, repeat the procedure above.

Example:

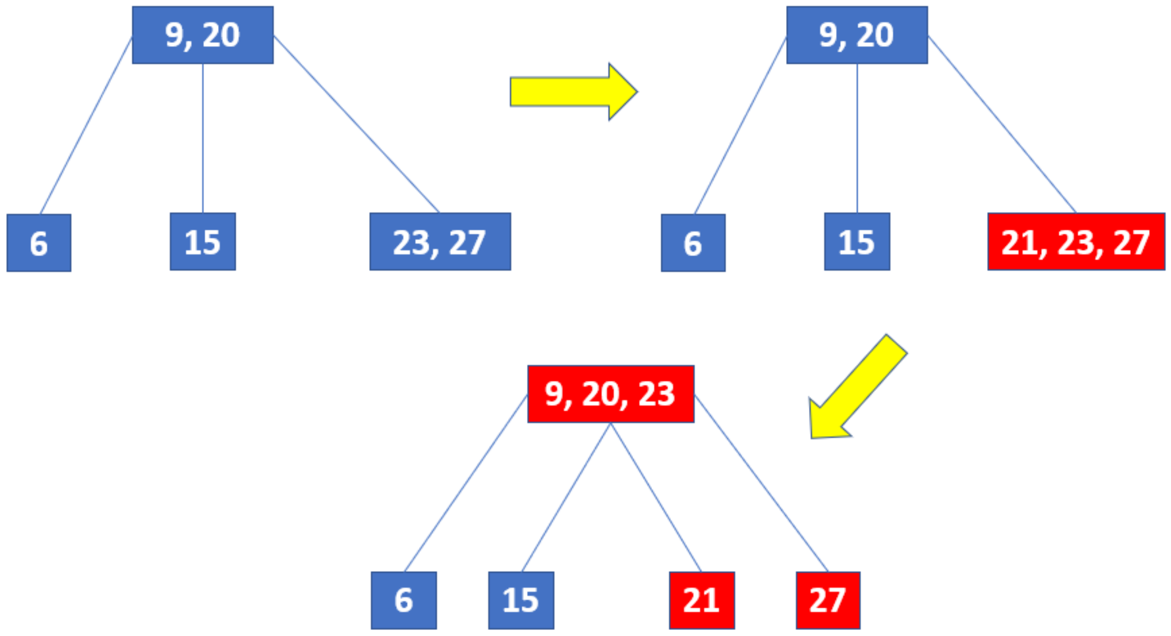


Figure 16. *Example for B-tree Insertion Operation (insert 21 to the tree)*

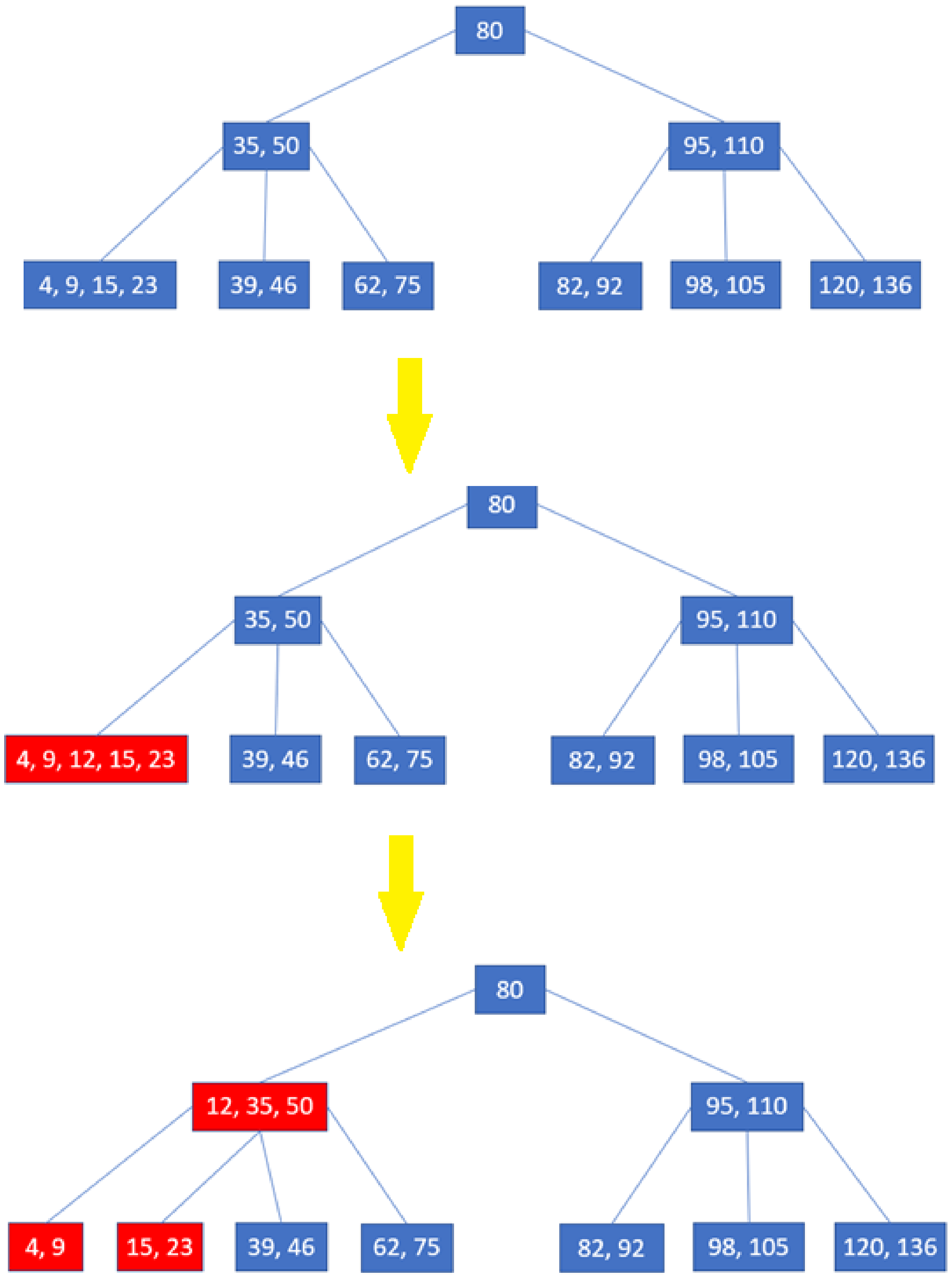


Figure 17. *Example for B-tree Insertion Operation (insert 12 to the tree)*

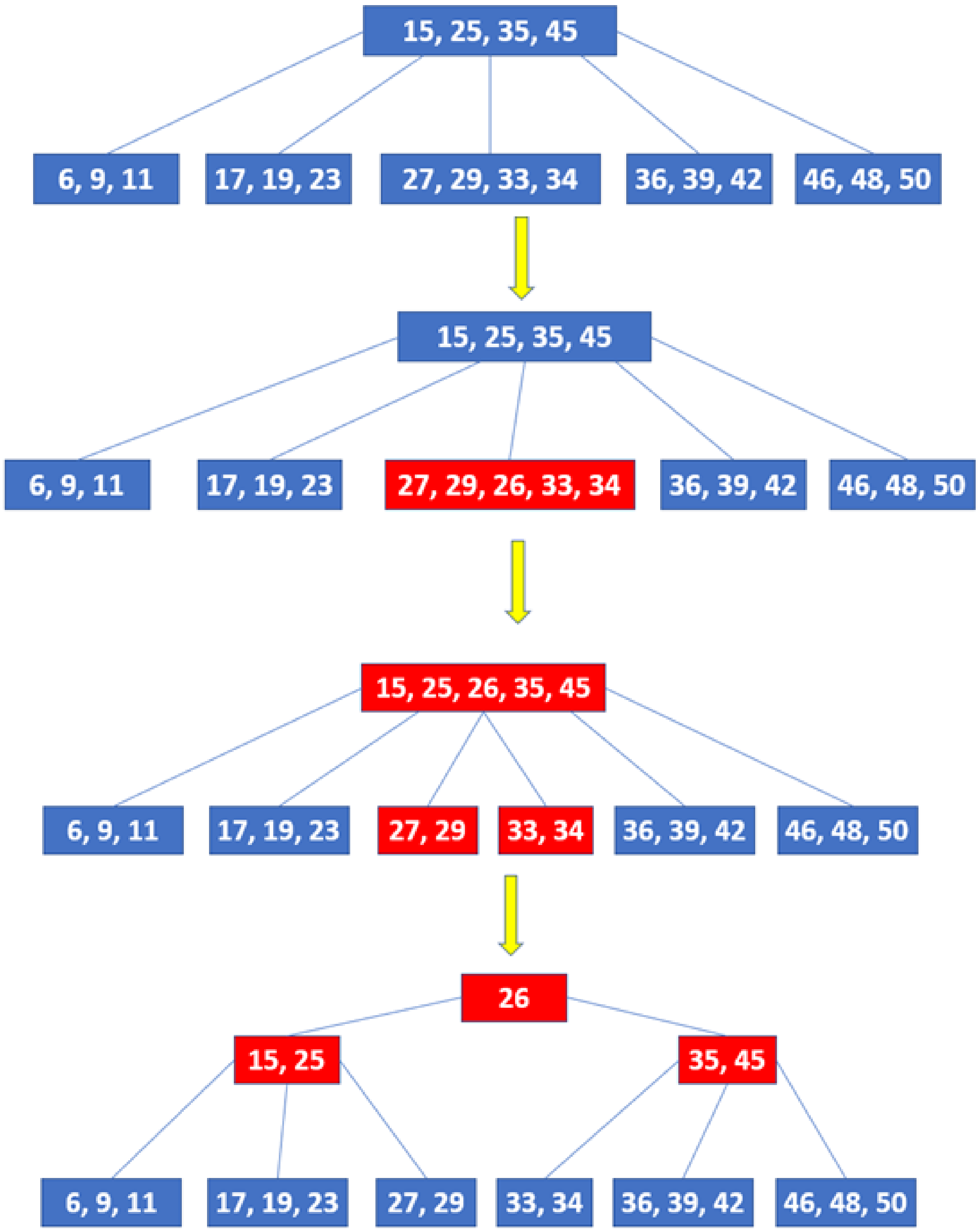


Figure 18. *Example for B-tree Insertion Operation (insert 26 to the tree)*

c. Deletion

Deletion is also done at the leaf nodes. The node to delete is either a leaf node or an internal node. Follow the algorithm below to delete a node from B-tree.

Delete a key from a leaf node

1. Locate the key to delete
2. Simply delete it from the node if the key is in a leaf node.
3. Rebalance the tree with section “Rebalance” below. (if need)

Delete a key from an internal node

The internal node's key is used to make a separating value between two sub-trees under it called the separator. So before deleting the internal node's key, we need to find another key to make it into the separator instead. That key can be the smallest key on the left subtree or the largest key on the right subtree. The algorithm is described as follows:

1. Choose a new separator. (may be the smallest key of the left subtree or the largest key of the right subtree)
2. Remove the selected key from its leaf node and replace the key that will be deleted with that key to make a new separator.
3. If the leaf node is deficient (less than the number of keys required), we will rebalance the tree with the "Rebalance" section below, start from the leaf tree.

Rebalance

The balancing is done from leaf node to root until all nodes satisfy the properties of B-tree.

Deleting a key in a node can cause it to be deficient, so some keys need to be reallocated so that all keys reach a minimum size. Typically, this allocation will require sibling node keys that have a greater number of keys than the minimum required by a node. If no sibling node has extra keys, then the deficient node will have to merge with the sibling node.

Merging can affect the parent node, making the parent node deficient and need to be rebalanced. This merge and balancing continues from root to root. Because the minimum key attribute does not apply to root, it is okay to root the node is deficient. The algorithm for balancing is as follows:

1. <u>If the right sibling node of the deficient node exists and has more than the number of keys needed for a node:</u>
   a. If the right sibling node of the deficient node exists and has more than the number of keys needed for a node:
   b. Copy separator from the parent node to the end of the deficient node (the separator is moved down, so the deficient node has the required number of keys for a node)
   c. Replace the recently moved separator with the smallest key of the right sibling node (the right sibling node loses one key but still has the required number of nodes for a node)

Now the tree is balanced.

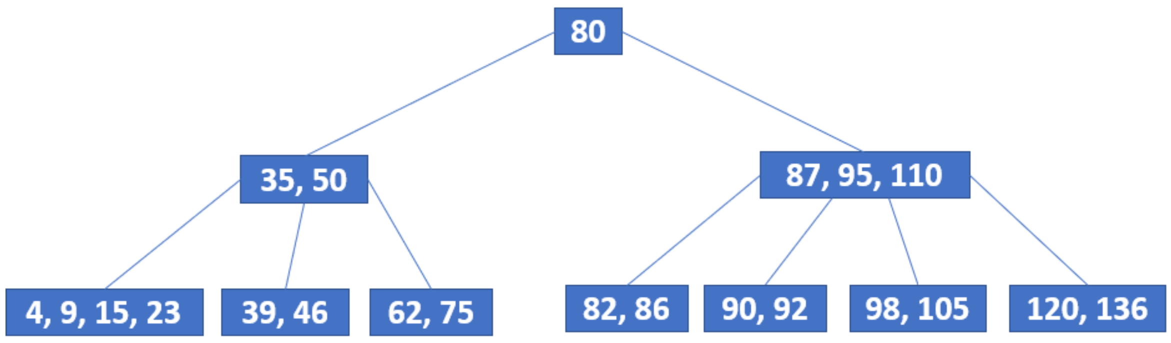
2. <u>If the node sibling on the left of the deficient node exists and has more than the number of keys needed for a node:</u>
   a. If the node sibling on the left of the deficient node exists and has more than the number of keys needed for a node:
   b. Copy separator from the parent node to the start of the deficient node (the separator is moved down, so the deficient node has enough keys needed for a node)
   c. Replace the recently moved separator with the largest key of the left sibling node (the left sibling node loses one key but still has the required number of keys for a node)

Now the tree is balanced.

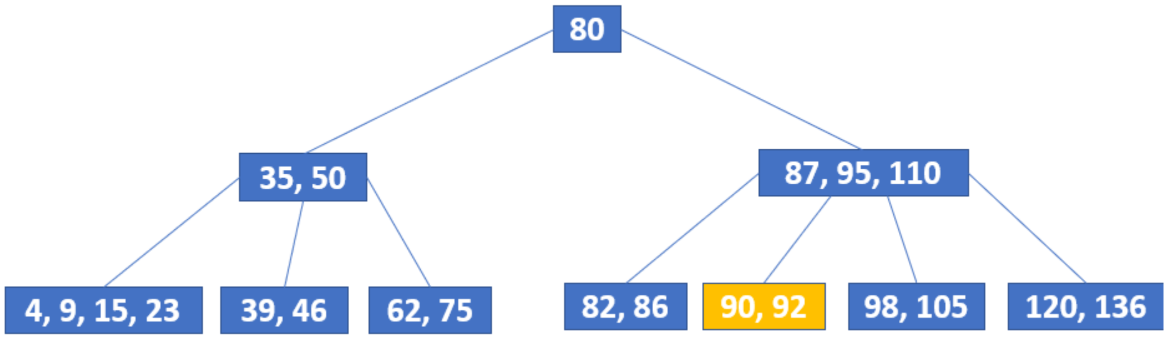
3. <u>If the sibling node on both sides of the deficient node only has the required number of keys for a node, merge the deficient node with the sibling node has the same separator:</u>
   a. Copy the separator of the parent node to the left node (the left node can be either the deficient node or the sibling node with just the minimum number of keys for a node)
   b. Move all keys of the right node to the left (the left node now has the maximum number of keys a node can have, the right node is empty)
   c. Delete the separator on the parent node along with the right child. This causes the parent node to lose a key and can result in the following:
      i. If the parent node is root: after clearing the key it is empty, release it and let the newly merged node become the new root.
      ii. If the parent node has fewer than the minimum number of keys required for a node, perform the balancing at the parent node.

Example:

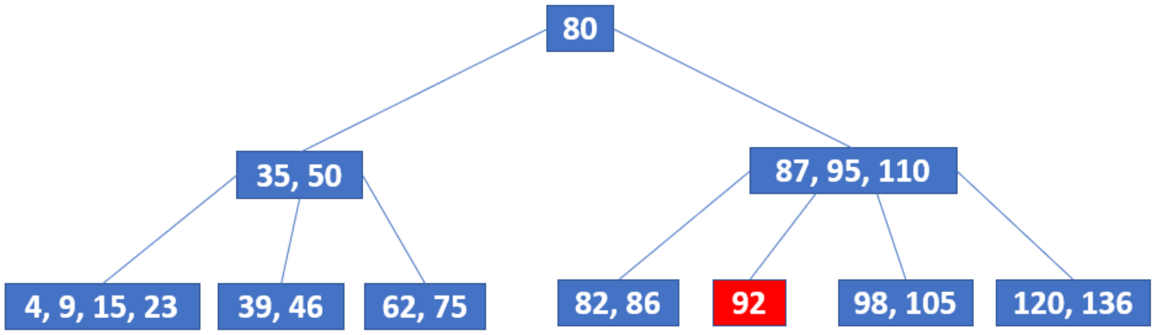
We have a B-tree as shown follows:



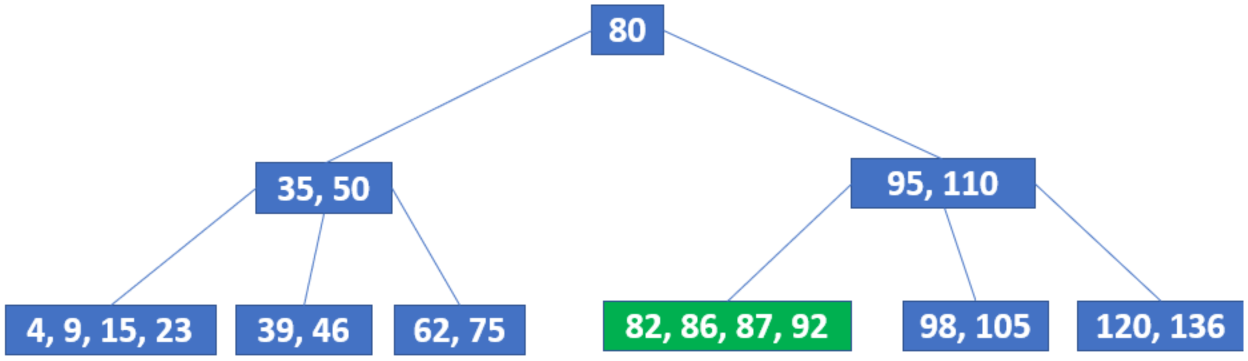
Now, we delete 90 in the right child of key 87.



Then, the node has one left key is 92, so that node is deficient. Moreover, its siblings don’t have spare key to share.



Therefore, we merge the node has key 92 with its right sibling and a separator taken from its parent i.e. 87.



Generally:

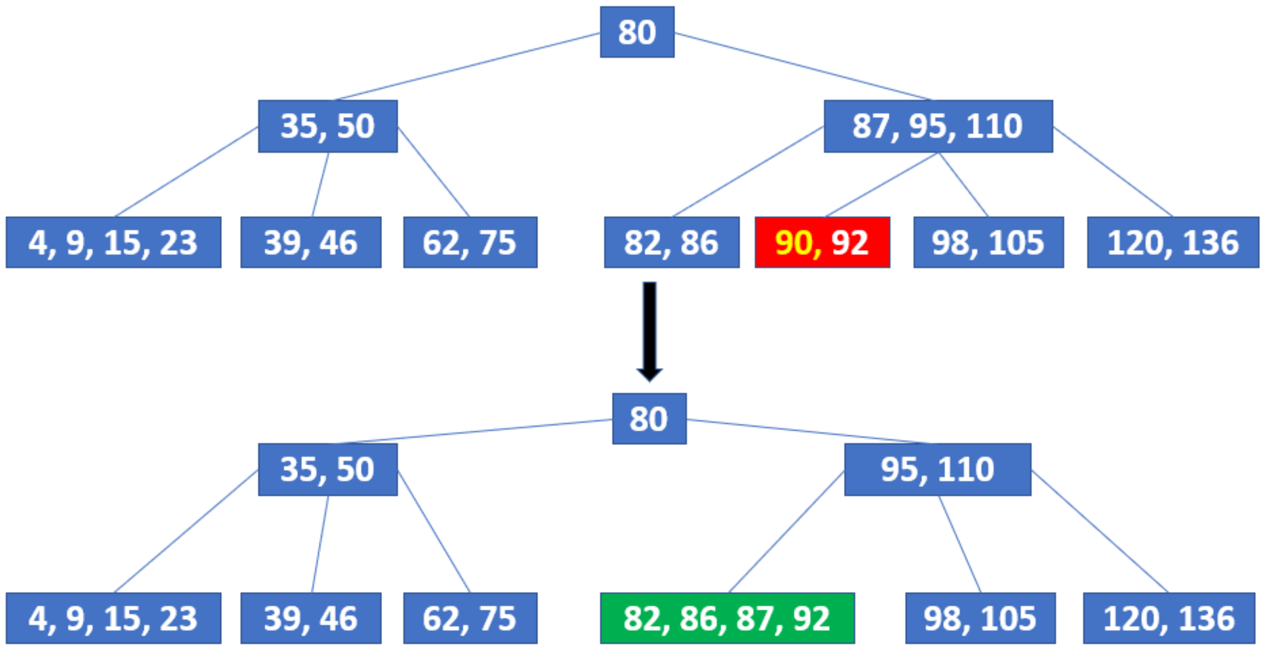


Figure 19. *Example for B-tree Deletion Operation (delete 90 from the tree)*

3.3 Implementation

3.4 The number of disk accesses is measured in terms of the amount of pages of data that require to be read from or written to the disk. We note that access time isn't constant--it depends on the space between the present track and therefore the desired track and also on the initial rotational state of the disk. However, we will use the number of pages that are read or written as approximately the first order of the entire disk access time.

3.5 In a typical B-tree application, the number of information handled is so large that each one the info don't fit into main memory directly. Tree B algorithms copy selected pages from the disk into PRN main memory and write to the changed disk pages. The size of the main memory is not limited to the size of the B tree because the B tree algorithm only needs a continuous number of pages in main memory at any time.

a. Create B-tree

We need to create an empty root node to initialize the B-tree.

```
B_TREE_CREATE (B)
        x = ALLOWCATE ();
        leaf[x] = True
        n[x] = 0
        DISK_WRITE (x)
        root[B] = x
```

This assumes there is an ALLOCATE function that returns a node with key, c, leaf fields, etc., and each node has a unique "address" on the disk.

b. Insertion

Next, to insert a key k into a B-tree B. It calls two other functions, B_TREE_SPLIT, that splits a node, and B_TREE_INSERT_NONFULL, that handles inserting into a node that isn't full.

```
B_TREE_INSERT (B, k)
        r = root[B]
        if n[r] = 2t - 1 then
                // we have to split it because root is full
                x = ALLOCATE ()
                root[B] = x     // new root
                leaf[x] = False // have children
                n[x] = 0        // for now
                c(1)[x] = r //  the old root now is child
                B_TREE_SPLIT (x, 1, r) // split r
                B_TREE_INSERT_NONFULL (x, k) // x isn’t full
        else
```

```
                    B_TREE_INSERT_NONFULL (r, k)
        endif
```

Let’s take a look at B_TREE_INSERT_NONFULL function.

```
        B_TREE_INSERT_NONFULL (x, k)
                i = n[x]
                if leaf[x] then
// shift everything over to the "right" up to the point where the new key k should go
                        while i >= 1 and k < key(i)[x] do
                                key(i+1)[x] = key(i)[x]
                                i--
                        end while
                        // stick k in its right place and increase n[x]
                        key(i+1)[x] = k
                        n[x]++
                else    // find child where new key belongs:
                        while i >= 1 and k < key(i)[x] do
                                i--
                        end while
                        // if k is in c(i)[x], then k <= key(i)[x] (from the definition)
                        // we'll go back to the last key (least i) where we found this
                        // to be true, then read in that child node
                        i++
                        DISK_READ (c(i)[x])
                        if n[c(i)[x]] = 2t - 1 then
                                // split this child node because it’s full
                                B_TREE_SPLIT_CHILD (x, i, c(i)[x])
                // now c(i)[x] and c(i+1)[x] are the new children,
                // and key(i)[x] may have been changed.
                // we'll see if k belongs in the first or the second
                if k > key(i)[x] then i++
            end if
            // call recursively to do the insertion
            B_TREE_INSERT_NONFULL (c(i)[x], k)
    end if
```

Now let's see how to split a node. When we slit a node, we always do it to its parents; two new nodes appear and the parent has one more child than before.

```
    B_TREE_SPLIT (x, i, y)
        z = ALLOCATE ()
        // new node is a leaf if old node was
        leaf[z] = leaf[y]
        // we since y is full, the new node must have t-1 keys
        n[z] = t - 1
        // copy over the "right half" of y into z
        for j in 1..t-1 do
            key(j)[z] = key(j+t)[y]
        end for
        // copy over the child pointers if y isn't a leaf
        if not leaf[y] then
            for j in 1..t do
                c(j)[z] = c(j+t)[y]
            end for
        end if
```

```
// having "chopped off" the right half of y, it now has t-1 keys
    n[y] = t - 1
    // shift everything in x over from i+1, then stick the new child in x;
    // y will half its former self as c(i)[x] and z will
    // be the other half as c(i)+1[x]
    for j in n[x]+1 downto i+1 do
        c(j+1)[x] = c(j)[x]
    end for
    c(i+1) = z
    // shifted keys
    for j in n[x] downto i do
        key(j+1)[x] = key(j)[x]
    end for
    // to accommodate the new key we're bringing in from the middle of y
    // (if you're wondering, since (t-1) + (t-1) = 2t-2, where
    // the other key went, its coming into x)
    key(i)[x] = key(t)[y]
    n[x]++
    // write everything to disk
    DISK_WRITE (y)
    DISK_WRITE (z)
    DISK_WRITE (x)
```

c. Deletion

Assume that the B_TREE_DELETE function is required to remove the key k from the root subtree at x. This function is structured to ensure that whenever B_TREE_DELETE is called recursively on an x node, the number of keys in x is at least the minimum t. This condition requires an extra key compared to the minimum required by normal B-tree conditions, so sometimes a key may have to be moved to a child node before recursively descending to that child.

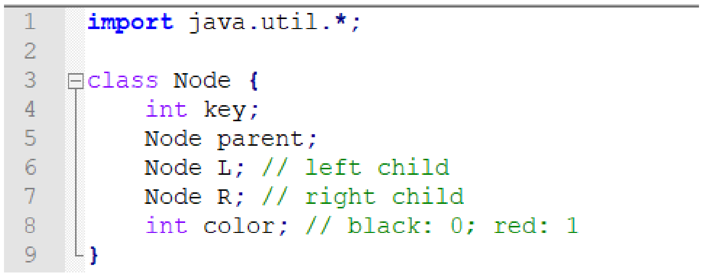
```
B_TREE_DELETE(x, k)
    if not leaf[x] then
    y ← PREDECESSOR _CHILD(x)
    z ← SUCCESSOR_CHILD(x)
    if n[y] > t − 1 then
        k' ← FIND_PREDECESSOR(k, x)
        MOVE (k', y, x)
        MOVE (k, x, z)
        B_TREE_DELETE (k, z)
    else if n[z] > t − 1 then
        k' ← FIND_SUCCESSOR(k, x)
        MOVE (k', z, x)
        MOVE (k, x, y)
        B_TREE_DELETE (k, y)
    else
        MOVE (k, x, y)
        MERGE(y, z)
        B_TREE_DELETE (k, y)
    else (leaf node)
     y ← PREDECESSOR _CHILD(x)
     z ← SUCCESSOR_CHILD (x)
     w ← root(x)
     v ← Root_Key(x)
        if n[x] > t − 1 then REMOVE (k, x)
```

```
else if n[y] > t − 1 then
    k' ← FIND_PREDECESSOR (w, v)
    Move-Key(k', y,w)
    k' ← FIND_SUCCESSOR (w, v)
    Move-Key(k',w, x)
    B_TREE_DELETE (k, x)
else if n[w] > t − 1 then
    k' ← FIND_SUCCESSOR (w, v)
    Move-Key(k', z,w)
    k' ← FIND_PREDECESSOR (w, v)
    MOVE (k',w, x)
    B_TREE_DELETE (k, x)
else
    s ← FIND_SIBLING(w)
    w' ← root(w)
        if n[w'] = t − 1 then
            MERGE (w',w)
            MERGE (w, s)
            B_TREE_DELETE (k, x)
        else
            MOVE (v,w, x)
            B_TREE_DELETE (k, x)
```

CHAPTER 4: DEMO

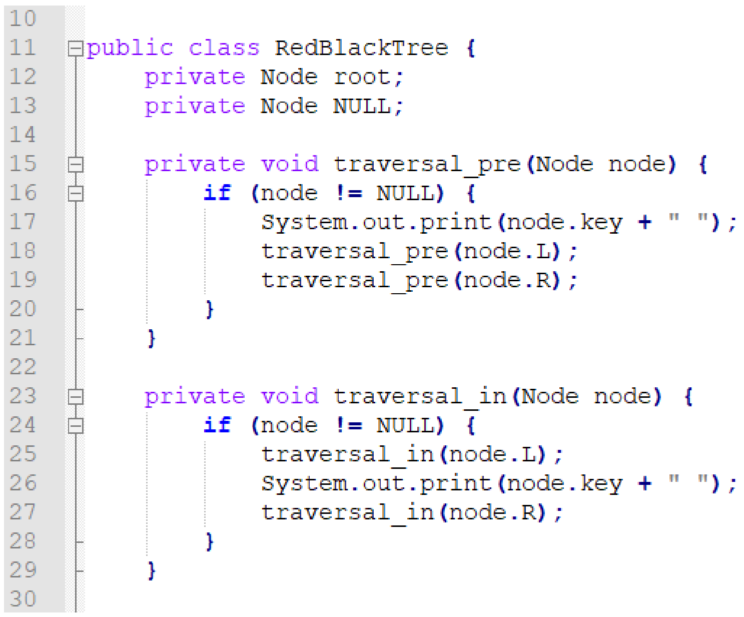
In this section, I will implement the Red-Black tree data structure, below is the code to do this. These codes are written in the Java language.

First, we import the full java.util package with the command: *import java.util. *;* Then, create class Node to store properties of a node in tree.

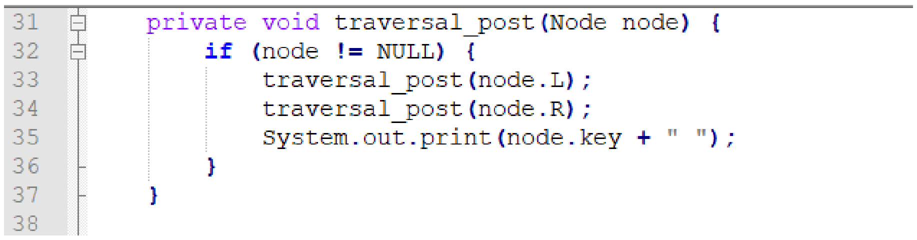


```java
import java.util.*;

class Node {
    int key;
    Node parent;
    Node L; // left child
    Node R; // right child
    int color; // black: 0; red: 1
}
```

Start to create method. For traversal the tree, we can use 3 way: preorder, inorder, postorder respectively traversal_pre(), traversal_in(), traversal_post() in these code.

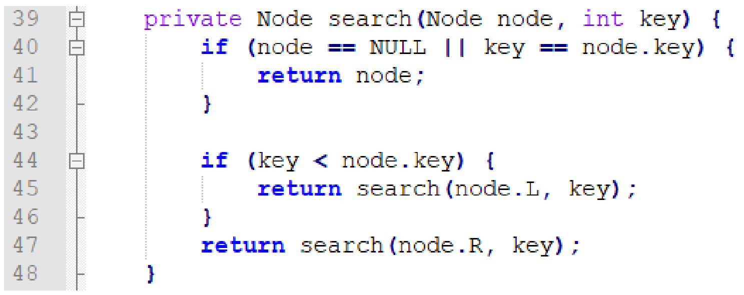


```java

public class RedBlackTree {
    private Node root;
    private Node NULL;

    private void traversal_pre(Node node) {
        if (node != NULL) {
            System.out.print(node.key + " ");
            traversal_pre(node.L);
            traversal_pre(node.R);
        }
    }

    private void traversal_in(Node node) {
        if (node != NULL) {
            traversal_in(node.L);
            System.out.print(node.key + " ");
            traversal_in(node.R);
        }
    }

```



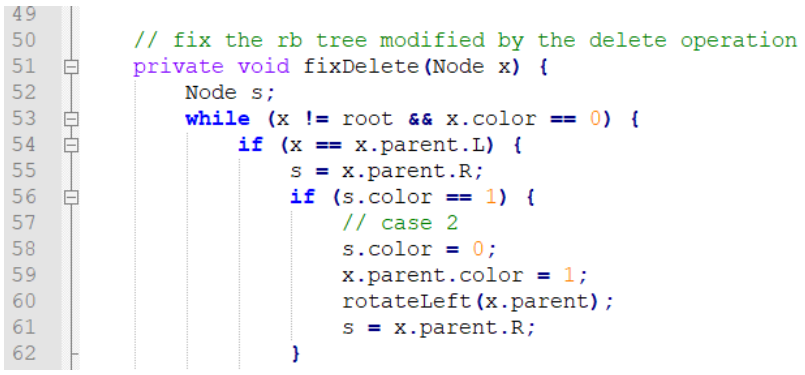
```java
    private void traversal_post(Node node) {
        if (node != NULL) {
            traversal_post(node.L);
            traversal_post(node.R);
            System.out.print(node.key + " ");
        }
    }

```

The following method use to search node in the tree with the given key. It using if-else condition and recursion to compare given key with key in tree until find one.

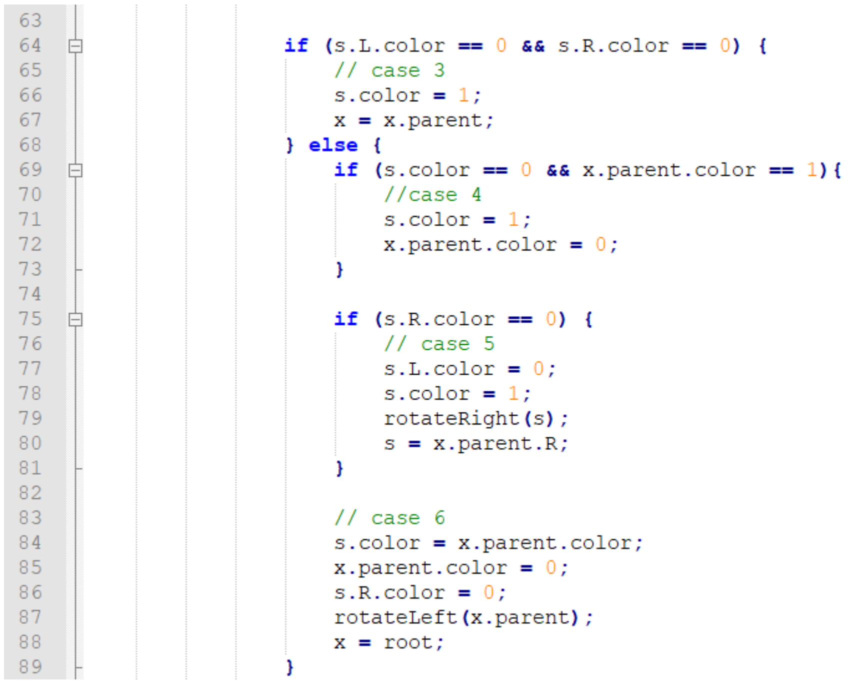


```
private Node search(Node node, int key) {
    if (node == NULL || key == node.key) {
        return node;
    }

    if (key < node.key) {
        return search(node.L, key);
    }
    return search(node.R, key);
}
```

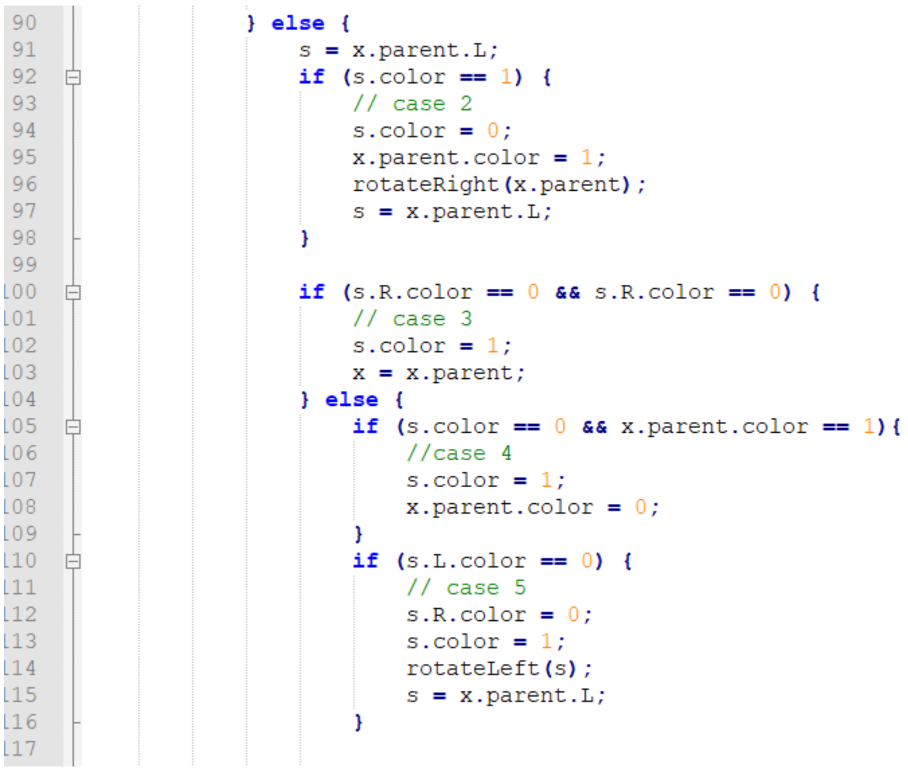
The following method is the method used to handle the cases listed in chapter 2 of the removal operation.



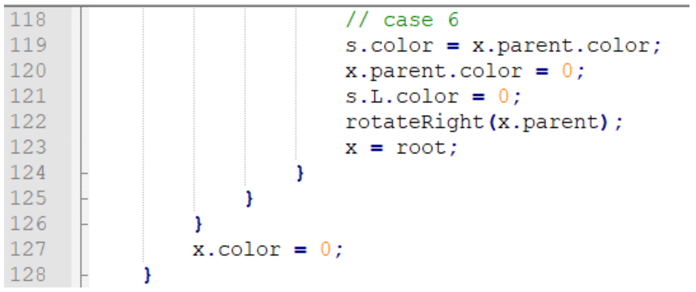
```

// fix the rb tree modified by the delete operation
private void fixDelete(Node x) {
    Node s;
    while (x != root && x.color == 0) {
        if (x == x.parent.L) {
            s = x.parent.R;
            if (s.color == 1) {
                // case 2
                s.color = 0;
                x.parent.color = 1;
                rotateLeft(x.parent);
                s = x.parent.R;
            }
```



```

            if (s.L.color == 0 && s.R.color == 0) {
                // case 3
                s.color = 1;
                x = x.parent;
            } else {
                if (s.color == 0 && x.parent.color == 1){
                    //case 4
                    s.color = 1;
                    x.parent.color = 0;
                }

                if (s.R.color == 0) {
                    // case 5
                    s.L.color = 0;
                    s.color = 1;
                    rotateRight(s);
                    s = x.parent.R;
                }

                // case 6
                s.color = x.parent.color;
                x.parent.color = 0;
                s.R.color = 0;
                rotateLeft(x.parent);
                x = root;
            }
```

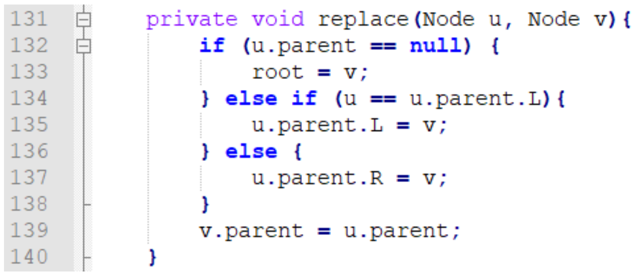


```
            } else {
                s = x.parent.L;
                if (s.color == 1) {
                    // case 2
                    s.color = 0;
                    x.parent.color = 1;
                    rotateRight(x.parent);
                    s = x.parent.L;
                }

                if (s.R.color == 0 && s.R.color == 0) {
                    // case 3
                    s.color = 1;
                    x = x.parent;
                } else {
                    if (s.color == 0 && x.parent.color == 1){
                        //case 4
                        s.color = 1;
                        x.parent.color = 0;
                    }
                    if (s.L.color == 0) {
                        // case 5
                        s.R.color = 0;
                        s.color = 1;
                        rotateLeft(s);
                        s = x.parent.L;
                    }

```



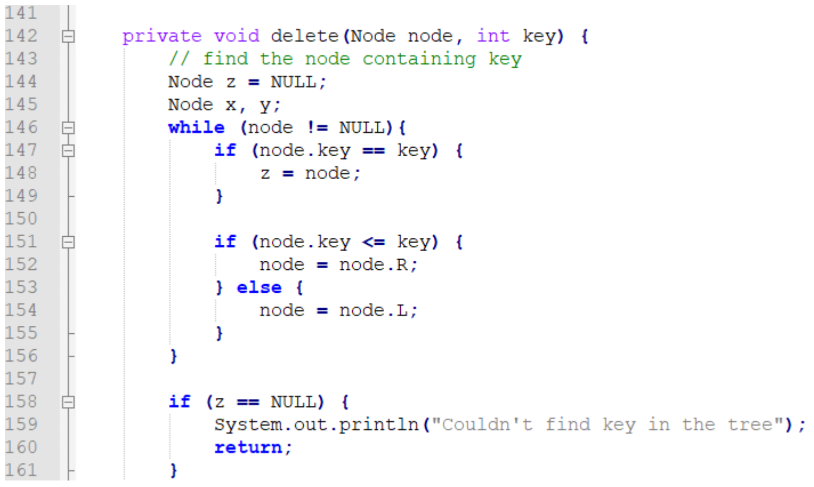
```
                    // case 6
                    s.color = x.parent.color;
                    x.parent.color = 0;
                    s.L.color = 0;
                    rotateRight(x.parent);
                    x = root;
                }
            }
        }
        x.color = 0;
    }
```

This next method is used to replace nodes with child nodes.

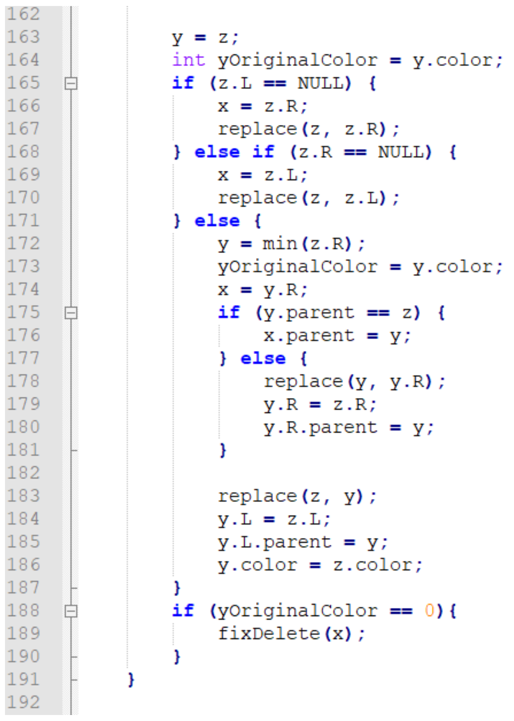


```
    private void replace(Node u, Node v){
        if (u.parent == null) {
            root = v;
        } else if (u == u.parent.L){
            u.parent.L = v;
        } else {
            u.parent.R = v;
        }
        v.parent = u.parent;
    }
```

The next method will be called when you want to delete a node from the tree.

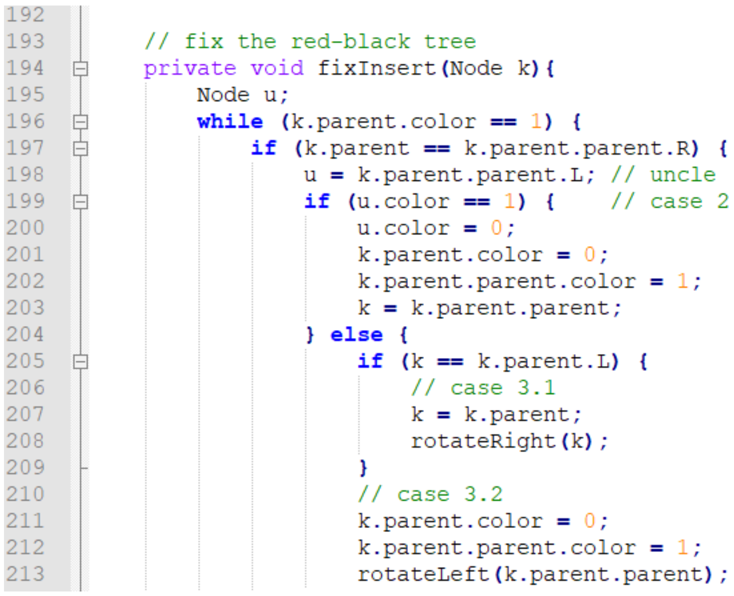


```
    private void delete(Node node, int key) {
        // find the node containing key
        Node z = NULL;
        Node x, y;
        while (node != NULL){
            if (node.key == key) {
                z = node;
            }

            if (node.key <= key) {
                node = node.R;
            } else {
                node = node.L;
            }
        }

        if (z == NULL) {
            System.out.println("Couldn't find key in the tree");
            return;
        }
```

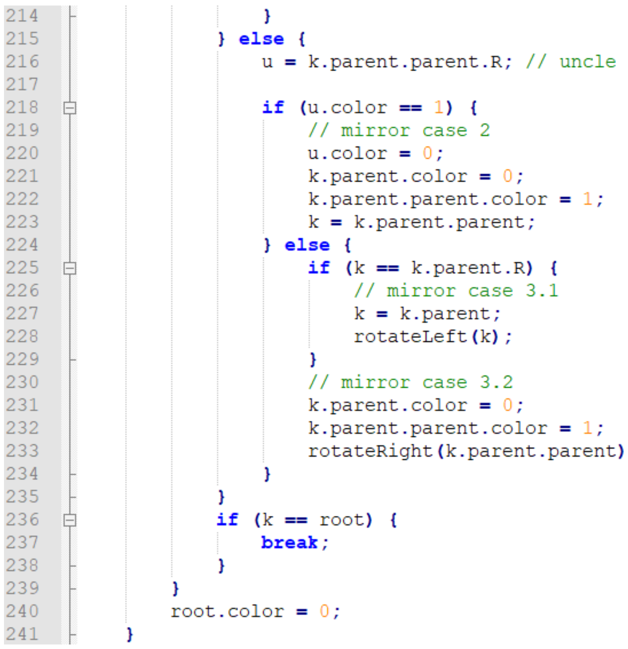


```

        y = z;
        int yOriginalColor = y.color;
        if (z.L == NULL) {
            x = z.R;
            replace(z, z.R);
        } else if (z.R == NULL) {
            x = z.L;
            replace(z, z.L);
        } else {
            y = min(z.R);
            yOriginalColor = y.color;
            x = y.R;
            if (y.parent == z) {
                x.parent = y;
            } else {
                replace(y, y.R);
                y.R = z.R;
                y.R.parent = y;
            }

            replace(z, y);
            y.L = z.L;
            y.L.parent = y;
            y.color = z.color;
        }
        if (yOriginalColor == 0){
            fixDelete(x);
        }
    }

```

The following method is the method used to handle the cases listed in chapter 2 of the insertion operation.

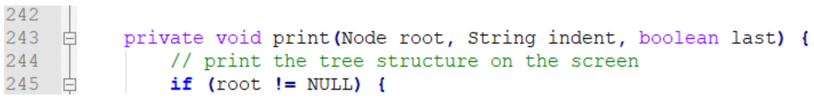


```

    // fix the red-black tree
    private void fixInsert(Node k){
        Node u;
        while (k.parent.color == 1) {
            if (k.parent == k.parent.parent.R) {
                u = k.parent.parent.L; // uncle
                if (u.color == 1) {    // case 2
                    u.color = 0;
                    k.parent.color = 0;
                    k.parent.parent.color = 1;
                    k = k.parent.parent;
                } else {
                    if (k == k.parent.L) {
                        // case 3.1
                        k = k.parent;
                        rotateRight(k);
                    }
                    // case 3.2
                    k.parent.color = 0;
                    k.parent.parent.color = 1;
                    rotateLeft(k.parent.parent);
```

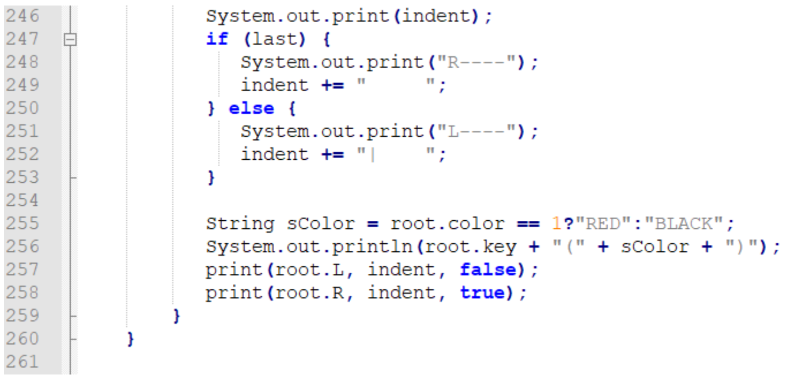


```
                }
            } else {
                u = k.parent.parent.R; // uncle

                if (u.color == 1) {
                    // mirror case 2
                    u.color = 0;
                    k.parent.color = 0;
                    k.parent.parent.color = 1;
                    k = k.parent.parent;
                } else {
                    if (k == k.parent.R) {
                        // mirror case 3.1
                        k = k.parent;
                        rotateLeft(k);
                    }
                    // mirror case 3.2
                    k.parent.color = 0;
                    k.parent.parent.color = 1;
                    rotateRight(k.parent.parent);
                }
            }
            if (k == root) {
                break;
            }
        }
        root.color = 0;
    }
```

The following print method is used to represent the B-tree in a visualized form.

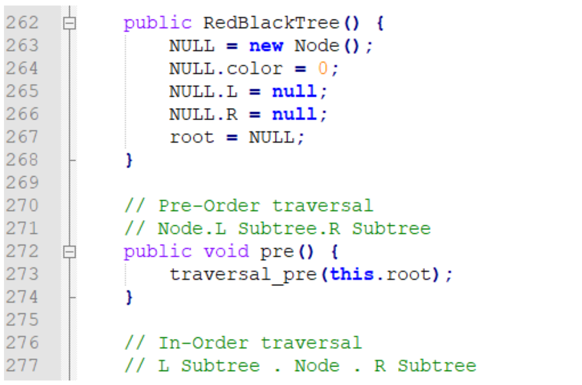


```

    private void print(Node root, String indent, boolean last) {
        // print the tree structure on the screen
        if (root != NULL) {
```

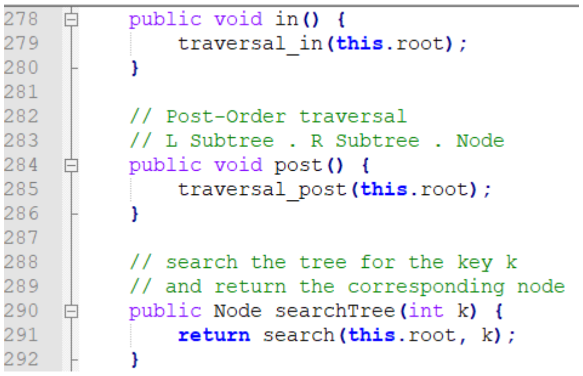


```
            System.out.print(indent);
            if (last) {
                System.out.print("R----");
                indent += "     ";
            } else {
                System.out.print("L----");
                indent += "|    ";
            }

            String sColor = root.color == 1?"RED":"BLACK";
            System.out.println(root.key + "(" + sColor + ")");
            print(root.L, indent, false);
            print(root.R, indent, true);
        }
    }

```

The following methods are used to call the above methods to execute at the beginning of the tree construction.

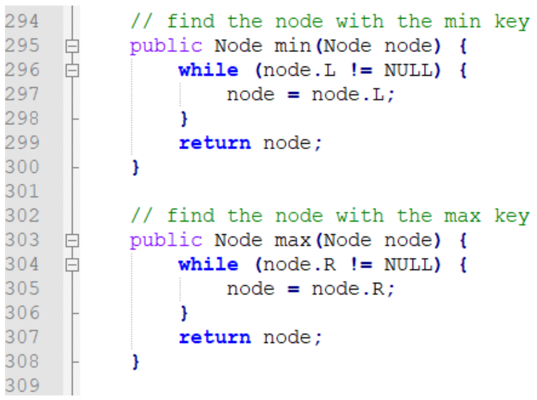


```
    public RedBlackTree() {
        NULL = new Node();
        NULL.color = 0;
        NULL.L = null;
        NULL.R = null;
        root = NULL;
    }

    // Pre-Order traversal
    // Node.L Subtree.R Subtree
    public void pre() {
        traversal_pre(this.root);
    }

    // In-Order traversal
    // L Subtree . Node . R Subtree
```

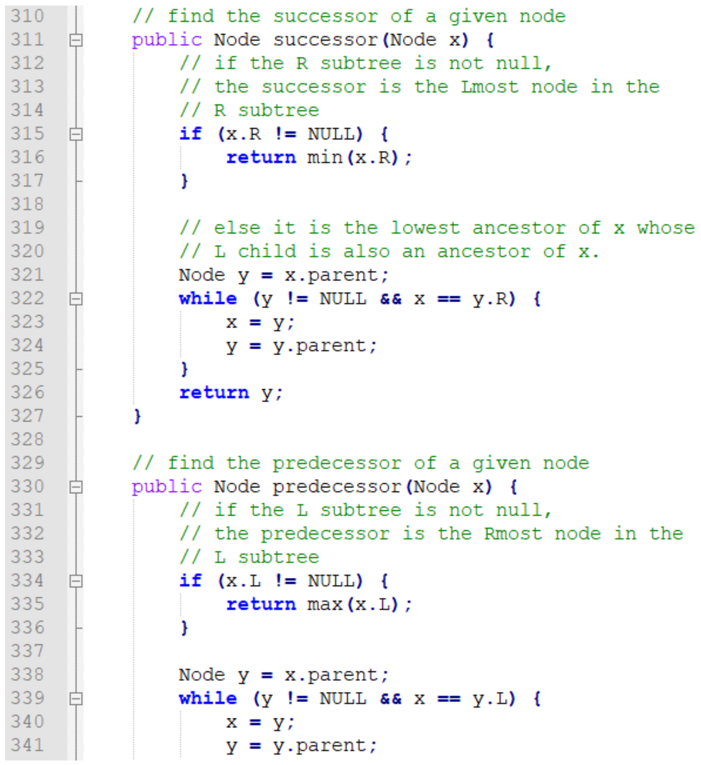


```
    public void in() {
        traversal_in(this.root);
    }

    // Post-Order traversal
    // L Subtree . R Subtree . Node
    public void post() {
        traversal_post(this.root);
    }

    // search the tree for the key k
    // and return the corresponding node
    public Node searchTree(int k) {
        return search(this.root, k);
    }
```

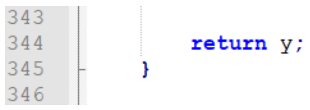
The following methods are used to find the node with the minimum key, maximum key and find the successor, predecessor of a given node.



```java
// find the node with the min key
public Node min(Node node) {
    while (node.L != NULL) {
        node = node.L;
    }
    return node;
}

// find the node with the max key
public Node max(Node node) {
    while (node.R != NULL) {
        node = node.R;
    }
    return node;
}

```

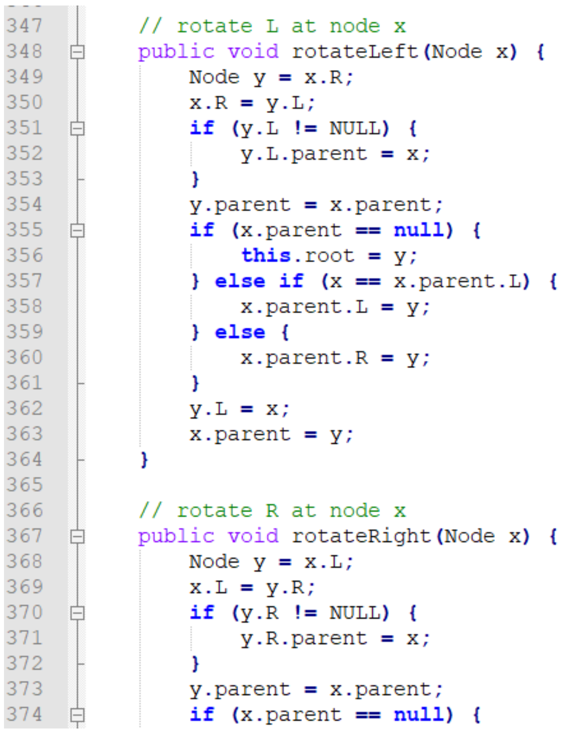


```java
// find the successor of a given node
public Node successor(Node x) {
    // if the R subtree is not null,
    // the successor is the Lmost node in the
    // R subtree
    if (x.R != NULL) {
        return min(x.R);
    }

    // else it is the lowest ancestor of x whose
    // L child is also an ancestor of x.
    Node y = x.parent;
    while (y != NULL && x == y.R) {
        x = y;
        y = y.parent;
    }
    return y;
}

// find the predecessor of a given node
public Node predecessor(Node x) {
    // if the L subtree is not null,
    // the predecessor is the Rmost node in the
    // L subtree
    if (x.L != NULL) {
        return max(x.L);
    }

    Node y = x.parent;
    while (y != NULL && x == y.L) {
        x = y;
        y = y.parent;
```

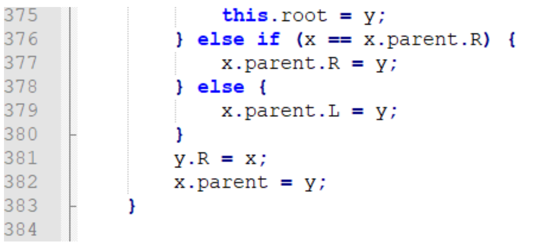


```java

    return y;
}

```

The following are the methods used to rotate the tree right and left based on what is written in chapter 2 rotation operation.

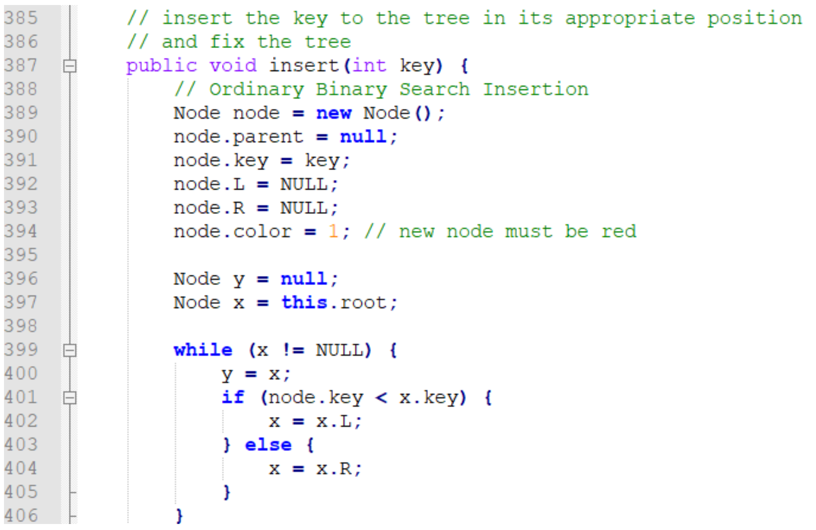


```
    // rotate L at node x
    public void rotateLeft(Node x) {
        Node y = x.R;
        x.R = y.L;
        if (y.L != NULL) {
            y.L.parent = x;
        }
        y.parent = x.parent;
        if (x.parent == null) {
            this.root = y;
        } else if (x == x.parent.L) {
            x.parent.L = y;
        } else {
            x.parent.R = y;
        }
        y.L = x;
        x.parent = y;
    }

    // rotate R at node x
    public void rotateRight(Node x) {
        Node y = x.L;
        x.L = y.R;
        if (y.R != NULL) {
            y.R.parent = x;
        }
        y.parent = x.parent;
        if (x.parent == null) {
```

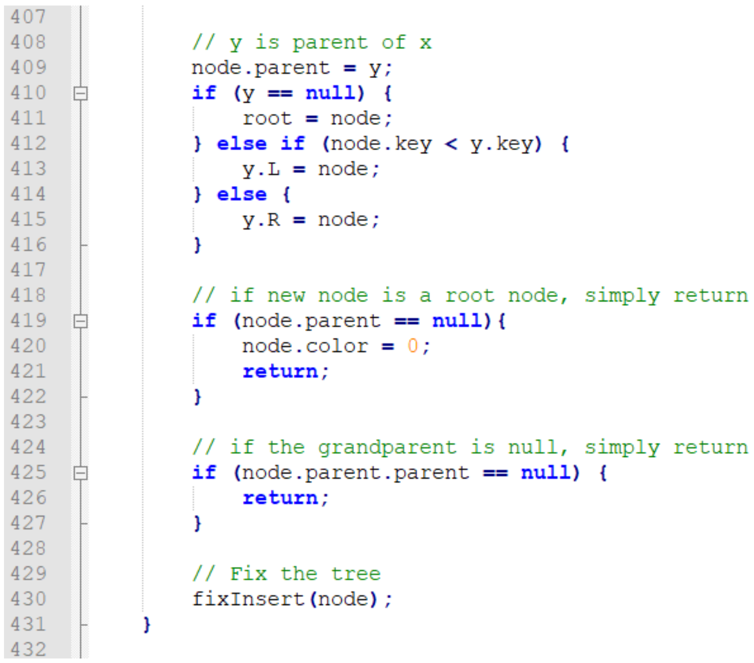


```
            this.root = y;
        } else if (x == x.parent.R) {
            x.parent.R = y;
        } else {
            x.parent.L = y;
        }
        y.R = x;
        x.parent = y;
    }
```

The next method will be called when you want to insert a node from the tree.

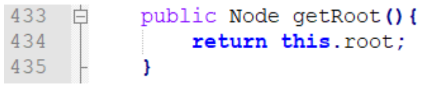


```
    // insert the key to the tree in its appropriate position
    // and fix the tree
    public void insert(int key) {
        // Ordinary Binary Search Insertion
        Node node = new Node();
        node.parent = null;
        node.key = key;
        node.L = NULL;
        node.R = NULL;
        node.color = 1; // new node must be red

        Node y = null;
        Node x = this.root;

        while (x != NULL) {
            y = x;
            if (node.key < x.key) {
                x = x.L;
            } else {
                x = x.R;
            }
        }
```



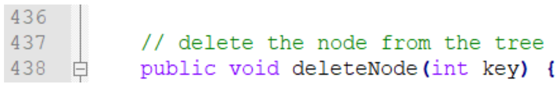
```java

        // y is parent of x
        node.parent = y;
        if (y == null) {
            root = node;
        } else if (node.key < y.key) {
            y.L = node;
        } else {
            y.R = node;
        }

        // if new node is a root node, simply return
        if (node.parent == null){
            node.color = 0;
            return;
        }

        // if the grandparent is null, simply return
        if (node.parent.parent == null) {
            return;
        }

        // Fix the tree
        fixInsert(node);
    }

```

This method use to get root node of the tree.

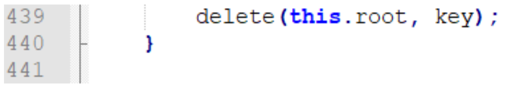


```java
    public Node getRoot(){
        return this.root;
    }
```

The next method use to delete a single node in tree when having the given key.

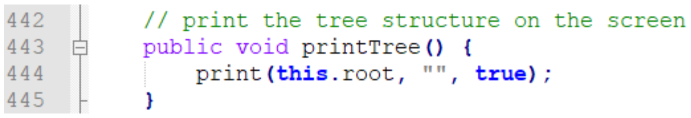


```java

    // delete the node from the tree
    public void deleteNode(int key) {
```



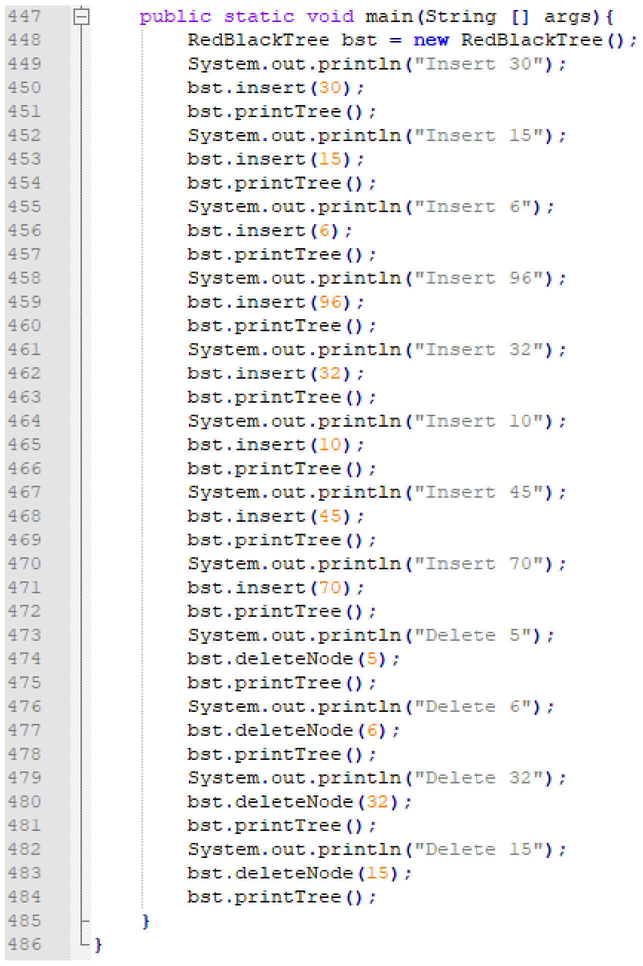
```java
        delete(this.root, key);
    }

```

The last method here using when you want to call print method above to print a tree in visualized form.



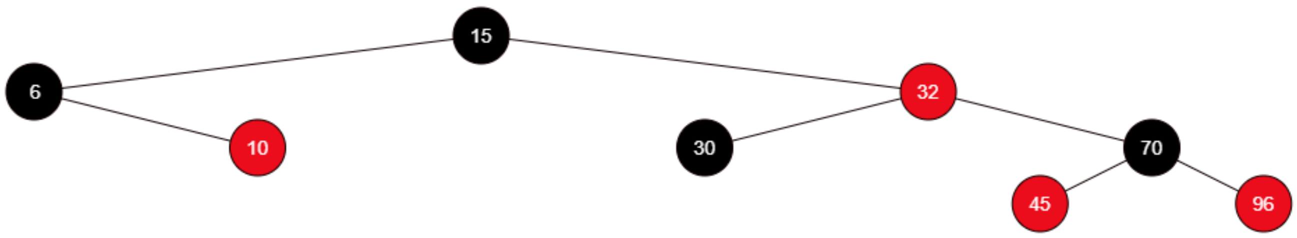
```java
    // print the tree structure on the screen
    public void printTree() {
        print(this.root, "", true);
    }
```

The main is as follows:

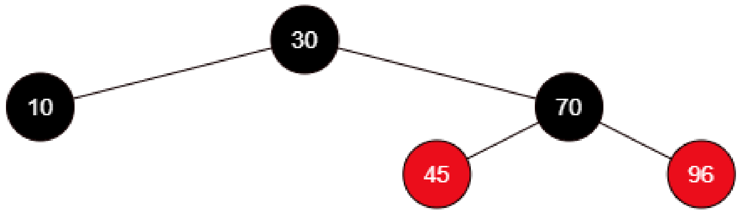


```
    public static void main(String [] args){
        RedBlackTree bst = new RedBlackTree();
        System.out.println("Insert 30");
        bst.insert(30);
        bst.printTree();
        System.out.println("Insert 15");
        bst.insert(15);
        bst.printTree();
        System.out.println("Insert 6");
        bst.insert(6);
        bst.printTree();
        System.out.println("Insert 96");
        bst.insert(96);
        bst.printTree();
        System.out.println("Insert 32");
        bst.insert(32);
        bst.printTree();
        System.out.println("Insert 10");
        bst.insert(10);
        bst.printTree();
        System.out.println("Insert 45");
        bst.insert(45);
        bst.printTree();
        System.out.println("Insert 70");
        bst.insert(70);
        bst.printTree();
        System.out.println("Delete 5");
        bst.deleteNode(5);
        bst.printTree();
        System.out.println("Delete 6");
        bst.deleteNode(6);
        bst.printTree();
        System.out.println("Delete 32");
        bst.deleteNode(32);
        bst.printTree();
        System.out.println("Delete 15");
        bst.deleteNode(15);
        bst.printTree();
    }
}
```

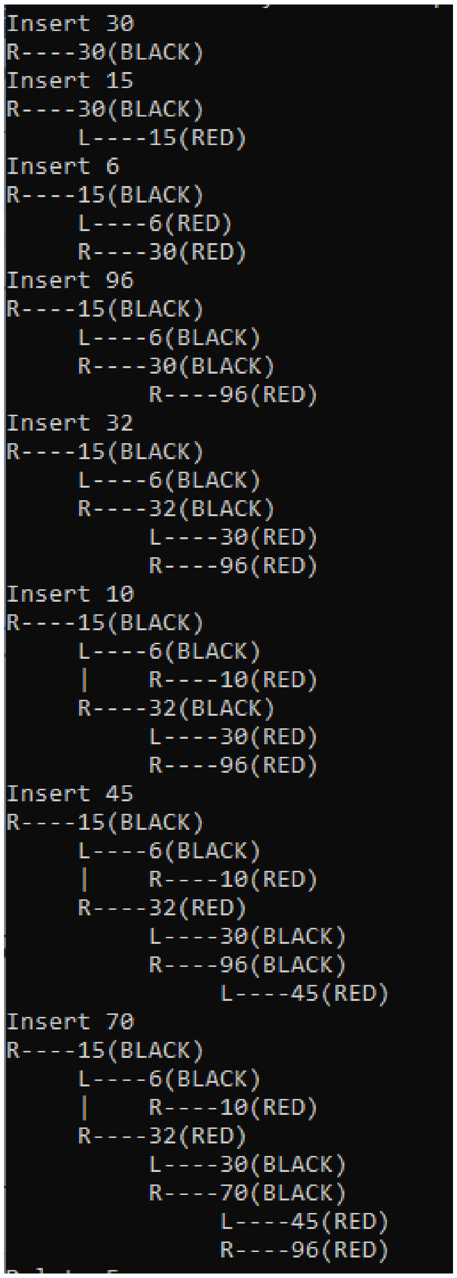
The tree we build by insert method is:



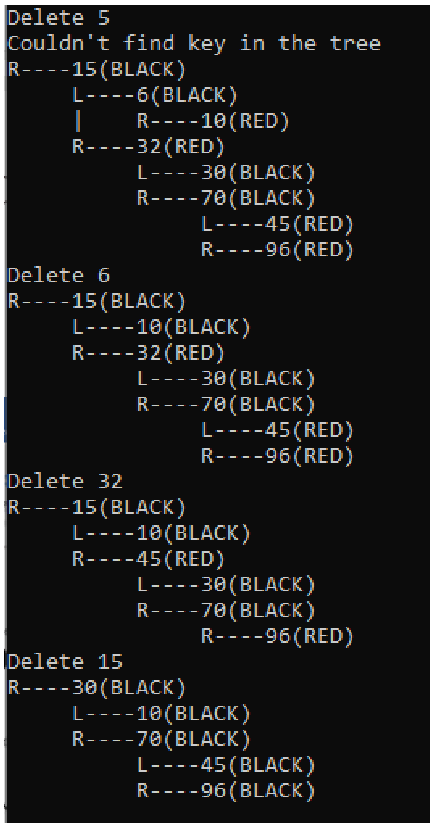
The tree after remove 5, 6, 32, 15 is:



The result after running the code is:



```
Insert 30
R----30(BLACK)
Insert 15
R----30(BLACK)
     L----15(RED)
Insert 6
R----15(BLACK)
     L----6(RED)
     R----30(RED)
Insert 96
R----15(BLACK)
     L----6(BLACK)
     R----30(BLACK)
          R----96(RED)
Insert 32
R----15(BLACK)
     L----6(BLACK)
     R----32(BLACK)
          L----30(RED)
          R----96(RED)
Insert 10
R----15(BLACK)
     L----6(BLACK)
     |    R----10(RED)
     R----32(BLACK)
          L----30(RED)
          R----96(RED)
Insert 45
R----15(BLACK)
     L----6(BLACK)
     |    R----10(RED)
     R----32(RED)
          L----30(BLACK)
          R----96(BLACK)
               L----45(RED)
Insert 70
R----15(BLACK)
     L----6(BLACK)
     |    R----10(RED)
     R----32(RED)
          L----30(BLACK)
          R----70(BLACK)
               L----45(RED)
               R----96(RED)
```



```
Delete 5
Couldn't find key in the tree
R----15(BLACK)
     L----6(BLACK)
     |    R----10(RED)
     R----32(RED)
          L----30(BLACK)
          R----70(BLACK)
               L----45(RED)
               R----96(RED)
Delete 6
R----15(BLACK)
     L----10(BLACK)
     R----32(RED)
          L----30(BLACK)
          R----70(BLACK)
               L----45(RED)
               R----96(RED)
Delete 32
R----15(BLACK)
     L----10(BLACK)
     R----45(RED)
          L----30(BLACK)
          R----70(BLACK)
               R----96(RED)
Delete 15
R----30(BLACK)
     L----10(BLACK)
     R----70(BLACK)
          L----45(BLACK)
          R----96(BLACK)
```

The full code is in file RedBlackTree.java. To run this file, I use command line of Windows with the command: *javac RedBlackTree.java && java RedBlackTree*

CHAPTER 5: CONCLUSION AND DISCUSSIONS

In this report, I have completed the assignment goal.

In chapter 1, I introduced the tree data structure as well as the binary search tree I learned. In addition, I will briefly introduce two tree data structures that will be presented in chapter 2 and chapter 3, red-black trees and b-trees.

Coming to chapter 2, I have raised the concept of red-black tree and also talked about the necessary properties of the tree. In addition, based on the request made, I presented the insert and remove operation and briefly presented its implementation with pseudocode. In each of the operations I also have given examples to be able to visualize this tree more clearly.

In chapter 3, I also talked about B-tree concepts, operations, and implementation. Examples have also been added to make B-tree representations more intuitive.

In chapter 4, I used the Java language to write code to implement how to create Red-Black trees with methods to insert or delete a node in the tree.

After making this report, I gained knowledge of tree data structures and their algorithms. At the same time, I also know the practical applications of these two types of tree data structures.

By making this report, I also gained the ability to search and read documents and improve my English proficiency.

In my opinion, in the process of making this report, I found it easier to read and understand the B-tree than the red-black tree. This is because there are red and black in the red tree, distinguishing between nodes, making it difficult to insert or remove a node than when doing it for a B-tree. This is one of my thoughts after completing this report.

REFERENCES

[1] Tree (data structure). En.wikipedia.org. https://en.wikipedia.org/wiki/Tree_(data_structure). Published 2020. Accessed May 25, 2020.

[2] Red–black tree. En.wikipedia.org. https://en.wikipedia.org/wiki/Red%E2%80%93black_tree#History. Published 2020. Accessed May 25, 2020.

[3] RedBlackTree.Andrew.cmu.edu. https://www.andrew.cmu.edu/user/mm6/95-771/examples/RedBlackTreeProject/dist/javadoc/redblacktreeproject/RedBlackTree.html?fbclid=IwAR3FdFaUgHVfxyjJOv3xwjE4MKfMpuWg4pi9kdgakXWrEsbW1-xVlqe3wys. Published 2020. Accessed May 25, 2020.

[4] Red-Black Trees. Pages.cs.wisc.edu. http://pages.cs.wisc.edu/~hasti/cs367-common/readings/Red-Black-Trees/index.html. Published 2020. Accessed May 25, 2020.

[5] Red-Black Tree | Set 1 (Introduction) - GeeksforGeeks. GeeksforGeeks. https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/. Published 2020. Accessed May 25, 2020.

[6] Binary Search Tree | Set 2 (Delete) - GeeksforGeeks. GeeksforGeeks. https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/?ref=rp. Published 2020. Accessed May 25, 2020.

[7] Binary Search Tree | Set 1 (Search and Insertion) - GeeksforGeeks. GeeksforGeeks. https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/. Published 2020. Accessed May 25, 2020.

[8] Binary search tree. En.wikipedia.org. https://en.wikipedia.org/wiki/Binary_search_tree#Traversal. Published 2020. Accessed May 25, 2020.

[9] Software.ucv.ro. http://software.ucv.ro/~mburicea/lab8ASD.pdf?fbclid=IwAR39HfsSUVCuEdO8w

HKqJSWjIN0qW4HJ59mGtpX-tk4b1OjEhc7u3ewsgm4. Published 2020. Accessed May 25, 2020.

[10] B Tree - javatpoint. www.javatpoint.com. https://www.javatpoint.com/b-tree. Published 2020. Accessed May 25, 2020.

[11] B-tree. En.wikipedia.org. https://en.wikipedia.org/wiki/B-tree#Insertion. Published 2020. Accessed May 25, 2020.

[12] Bibeknam/algorithmtutorprograms. GitHub. https://github.com/Bibeknam/algorithmtutorprograms/blob/master/data-structures/red-black-trees/RedBlackTree.java?fbclid=IwAR1n1LeSrD3GpEOqMBYUjvKmCbW_LkCyrPmZLzxhIHIi4qVkcxSd8f4AVUM. Published 2020. Accessed May 25, 2020.

[13] CS241: Data Structures & Algorithms II. Cpp.edu. https://www.cpp.edu/~ftang/courses/CS241/notes/b-tree.htm. Published 2020. Accessed May 25, 2020.

[14] CTDL: B-Tree - Quoc-Hung Ngo. Sites.google.com. https://sites.google.com/site/ngo2uochung/courses/ctdl-btree. Published 2020. Accessed May 25, 2020.

[15] Intro to Algorithms: CHAPTER 19: B-TREES. Staff.ustc.edu.cn. http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap19.htm. Published 2020. Accessed May 25, 2020.

[16] Introduction of B-Tree - GeeksforGeeks. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-of-b-tree-2/?ref=rp. Published 2020. Accessed May 25, 2020.

[17] Red Black Trees (with implementation in C++, Java, and Python). Algorithm Tutor. https://algorithmtutor.com/Data-Structures/Tree/Red-Black-Trees/?fbclid=IwAR3pdjaPNUfa40BTexBJZPElax-

saFZ1F2X_LCyRn0_eFvLrQuZxopw9EwI. Published 2020. Accessed May 25, 2020.

[18] Red-Black Tree | Brilliant Math & Science Wiki. Brilliant.org. https://brilliant.org/wiki/red-black-tree/. Published 2020. Accessed May 25, 2020.

SELF-EVALUATION

Requirements	**Score /10**	**Level 1**	**Level 2**	**Level 3**	**Self-evaluation**	**Reason(s)**
		0 score	**1/2 score**	**Full score**		
1/ Report	**8.0**					
In right format	1.0	Wrong format and outlines	Some errors	In right format and outlines, no error	1.0	
Chapter 1	1.0	Not enough content, bad written, no example	Full contents, not very well written, not enough examples	Full contents, well written, with examples	0.5	Not enough examples
Chapter 2	2.0	Not enough content, bad written, no example	Full contents, not very well written, not enough examples	Full contents, well written, with examples	1.0	Not enough examples
Chapter 3	2.0	Not enough content, bad written, no example	Full contents, not very well written, not enough examples	Full contents, well written, with examples	1.0	Not enough examples

Requirements	**Score /10**	**Level 1**	**Level 2**	**Level 3**	**Self-evaluation**	**Reason(s)**
		0 score	**1/2 score**	**Full score**		
Chapter 4	1.0	Not enough content, bad written, no example	Full contents, not very well written, not enough examples	Full contents, well written, with examples	0.5	Not enough examples
Chapter 5	0.5	Not enough content, bad written	Full contents, not very well written	Full contents, well written	0.25	Not very well written
References	0.5	No reference	Wrong format, < 3 references	Right format, ≥ 3 references	0.5	
2/ Demo	**2.0**					
Contents	1.5	Implement less than half of the operations	Implement half of the operations	Implement all of the operations	1.5	
Program	0.5	Cannot be compiled	Runtime error for 1 fomula or algorithm	Can be run correctly with no error	0.5	
Total	10.0	Result:			6.75	