COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

Lecture 19 Trees(2): April 16th, 2019

Lecturer: Dr. Andrew Hines *Scribes: Ankhit Pandurangi and Stephen Gaffney*

1.1 Overview-Binary Search Tree

1.1.1 Definition

A binary search tree (BST) is a tree abstract data structure. More specifically it is a sorted dynamic data structure, this means that the size of the binary search tree is only limited by the amount of free memory the process has. Like all trees the BST consists of nodes and edges.

Each node has a unique value (Cannot have duplicate values for nodes) and nodes are arranged and sorted by these values. Each node must have two links out of the node and no more. If the node does not have children then these links are null. Else they are the edge to a children node.

The concept behind BST is relatively straightforward and intuitive. Start with a root node and if the value is less than the root node then follow the edge to the left else if it is greater than the root node value then go to the right. This process is repeated within each subnode depending on the desired operations that are outlined below.

The major advantages of using BST are rapid searches and relatively cheap addition of new nodes. However some of the drawbacks are that only 2 children nodes possible, no duplicate values are allowed and deletions are a little more cumbersome. For these reasons BST are used widely in applications where rapid searches are required. Balanced trees attempt to improve on BST.

1.1.2 Application

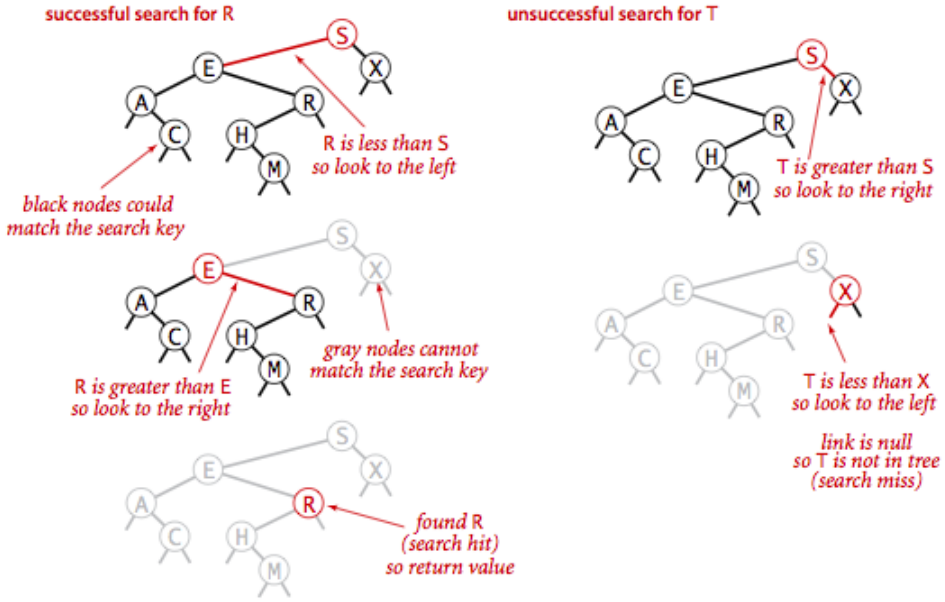
1. Used in many search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries.
2. Binary Space Partition - Used in almost every 3D video game to determine what objects need to be rendered.
3. Binary Tries/Radix Tree - Used in almost every high-bandwidth router for storing router-tables.

1.1.3 Operations

1. There are 3 main operations for BST; insertion, search and deletion. Like all data structures the time complexity of these vary and there are best, average and worst case scenarios for each.

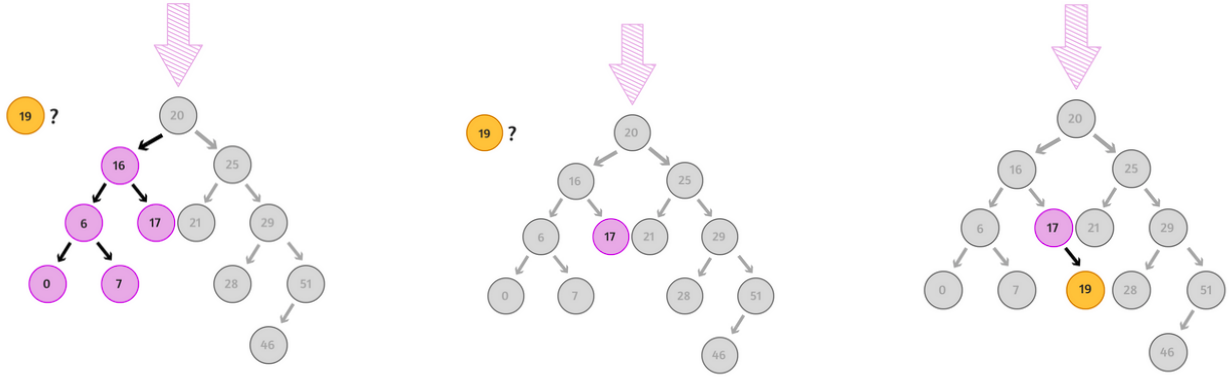
- **Search** -Starting from the root node:
    1. Compare the desired value to the value of the root node.
    2. If value is less than root node value then follow edge to left.

3. Or if it is greater than root node value so follow edge to right.
4. Else if the value matches the root node value then we have a Hit and we can return this node.
5. This process is repeated for each sub node after following the appropriate edge.
6. If comparing to null value then the operation has reached a base of the tree and this tree does not contain the value and should return false.



*

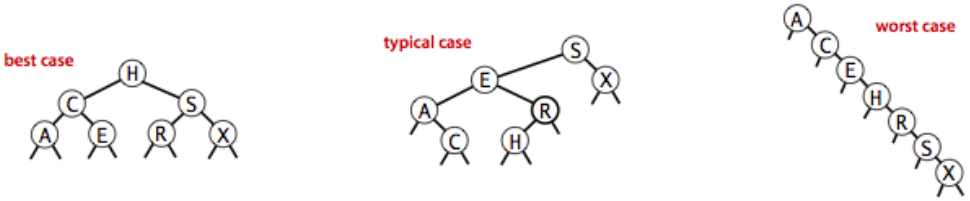
- **Insert: Following the exact same steps as the search operation:**
  - Initially search the tree for the value to be inserted, if it already exists then return an error as duplicates cannot exist.
  - If it returns null then change this null link to point to the new node to be inserted. New node is now inserted and will have two null links extending out of the node.



–

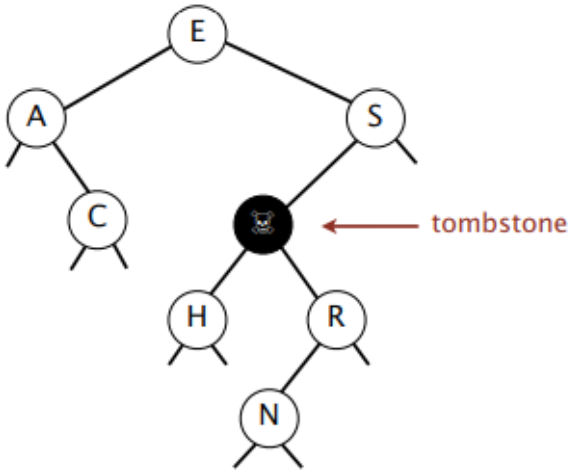
For both search and insert the time complexity depends on the shape of the tree. There are best, average/typical and worst case scenarios for both. Examples of the three can be seen below.

- **Best** - Tree is completely balanced, each node has two children nodes. Big O is O(logn) as only half of the nodes must be traversed at most.
- **Typical** - Not completely balanced but a mix of nodes with 2 and 1 children nodes. Big O is between O(logn) and O(n).
- **Worst** - Unbalanced and every node only has 1 child node. Big O is O(n) as whole tree must be traversed.



–

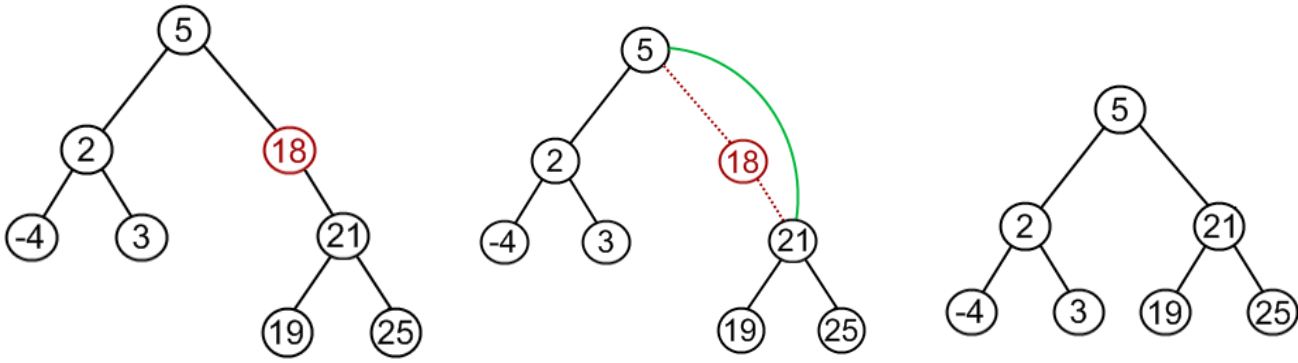
- **Deletion (Lazy Approach):**
  - Search for node to be deleted
  - Replace node with flag to signify that node has been deleted and should only be used for reference.
  - This flag node keeps its value and all sub nodes.



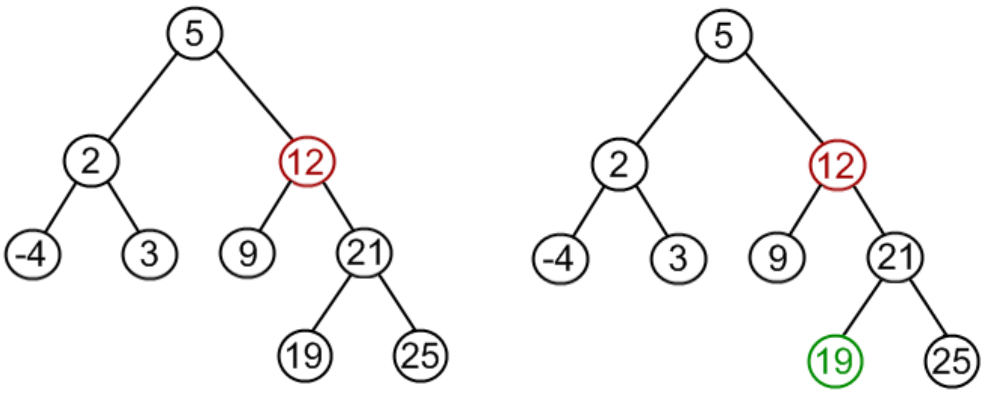
–]

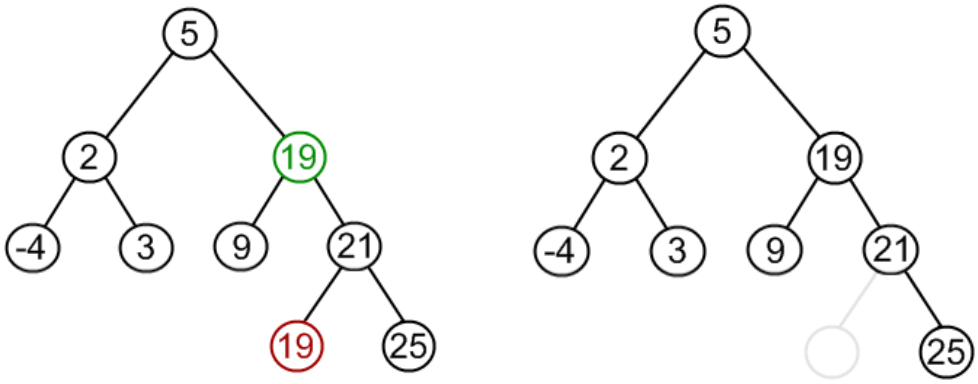
- **Deletion (Hibbard) 3 Cases::**
  - **Case 1**: Case 1 (0 children nodes):
    - The node to be deleted has zero children nodes.
    - Simply set the parent link to null.
    - Garbage collector will get rid of linkless node.

- **Case 2 (1 child node):**
  - The node to be deleted has one child node.
  - Set the link from parent node to now point to the child node of the node to be deleted e.g bypassing the deleted node to its child node.
  -



- **Case 3 (2 children nodes):**
  - The node to be deleted has two children nodes.
  - Replace the node to be deleted with the Leftmost child node of the Right child node
  - Replace the node to be deleted with the Rightmost child node of the Left child node
  - Delete the child node that was swapped with the node to be deleted.
  - This swap ensures that the sorting remains intact and the the sub-nodes are not lost on deletion.
  -





–

2 Overview-Balanced Search Tree (2-3 Tree)

2.1 Definition

1. 2-3 tree is an implementation of the Binary Search Tree. It differs from the Binary Search Tree in that it is perfectly balanced. This means that every path from the root node to the leaf has the same length. Lets explore what this means for time complexity.

   I Since we know that length of tree is the same from the root to all leaf nodes, the operations of lookup, insert, and delete will be, worst case, O(log(n)) operations.

   II However, as every insertion and deletion may require a restructuring of the tree itself, this means that insert and delete may inherently be more complicated than in a Binary Search Tree.

2. So what is the structure of a 2-3 tree?

   I Every non-leaf node has 2 or 3 children

   II All of the leaf nodes will be at the same level

   III If the node is a 2-node, it will have 1 key (A) and 2 children. The child with a value less than A will be the left child, the child with a value greater than A will be the right child.

   IV If the node is a 3-node, it will have 2 keys (A and B), and 3 children. The child with a value less than A will the be the left child, the child with a value in between A and B will be the middle child, and a child with a value greater than B will be the right child.

2.2 Application

1. 2-3 Trees are not applied too often in actuality. The complexity of implementing a 2-3 tree comes from its different node type, so instead, Red Black Trees are used in practical settings. This means that you will see very few 2-3 trees beyond an educational setting.

2.3 Operations

1. The main operations available for a 2-3 tree are:

   - **Search** -The lookup for a 2-3 tree is very similar to lookups in a Binary Search Tree. For finding a key K in the tree T:

     I If the tree is empty, return false. If the node is a leaf node, attempt to find the value.

     II If k is less than or equal to the value of the Node, search in the left subtree.

     III If it is a 3 node, if K is greater than key A and less than key B, search the middle subtree

     IV If K is greater than the value of the Node, search the right subtree. **If the value is still not found, return False.**

   This operation has a worst case time of O(log(n)), as the height of the tree is O(log(n))

   - **Insertion** - Insertion in a 2-3 tree always happens at a leaf node, but it is generally more complicated due to the various cases to consider. There are 3 cases which we must explore:

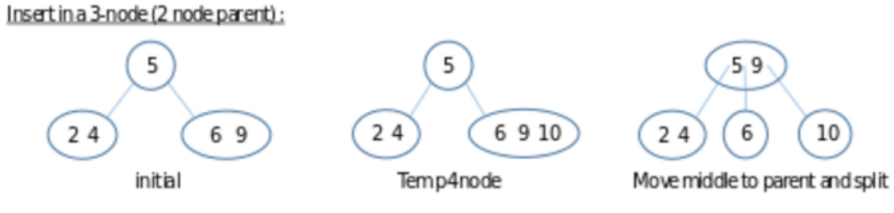
     – **Case 1**: Insertion into a leaf node with one value and 2 node parent. This is the simplest case of insertion, demonstrated here:

     – Simply insert the node into the available slot so that the structure is maintained.

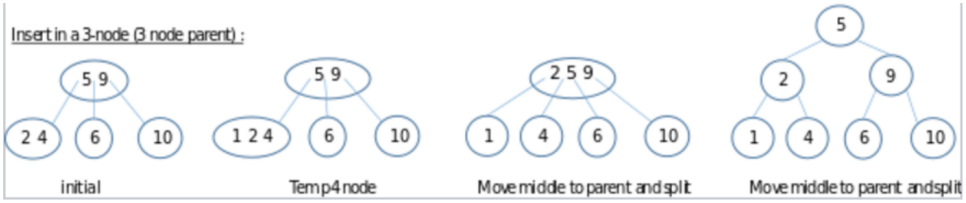
     –

   - **Case 2**: Insertion into a 3-node leaf with 2 node parent. This is more complicated, as you must split the data.

     –



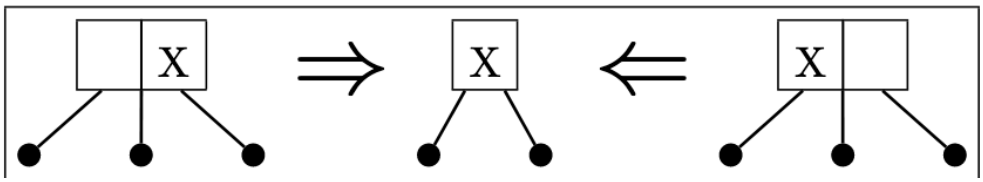
- **Case 3**: Insertion into a 3-node leaf with 3-node parent. This is the most complicated, as there is 2 layers of splitting involved as two temporary 4 node leaves will occur.

  – 

- **Deletion in a 2-3 tree:** - Deletion in a 2-3 tree is like insertion in that we have to consider various cases. This operation is more complex to understand due to the rebalancing of the tree that must occur once a trees node is deleted. Sometimes, deletion can leave a hole in a 2-3 Tree. This occurs when a value must be removed from a 2-node or a 3-node, so we must remove the hole and, in more complicated cases, replace it with its in-order predecessor or in-order successor.

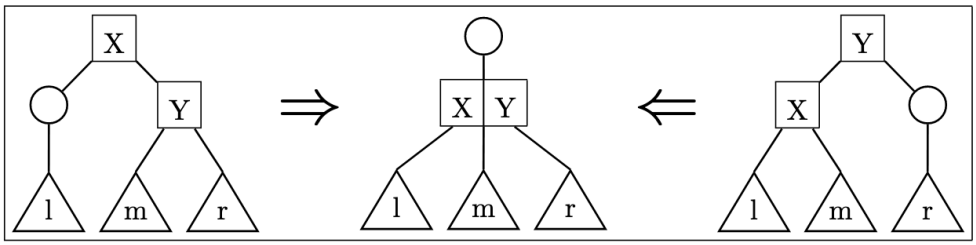
  – **Case 1**: Deleting a value and removing a hole from a 3-Node.

  – Simply insert the node into the available slot so that the structure is maintained.

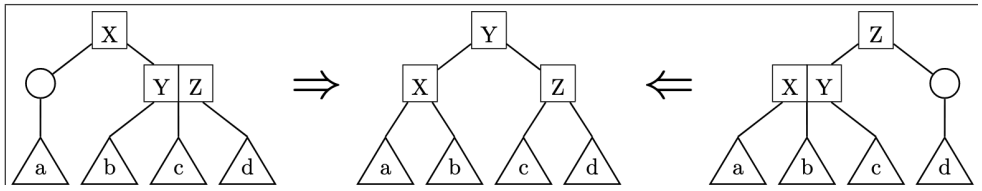
  – 

    1. ***HOWEVER, WHEN A 2 NODE IS DELETED, A HOLE IS LEFT BEHIND WHICH WE MUST DELETE. IT WILL PROPAGATE UP THE TREE UNTIL WE CAN DELETE IT.***

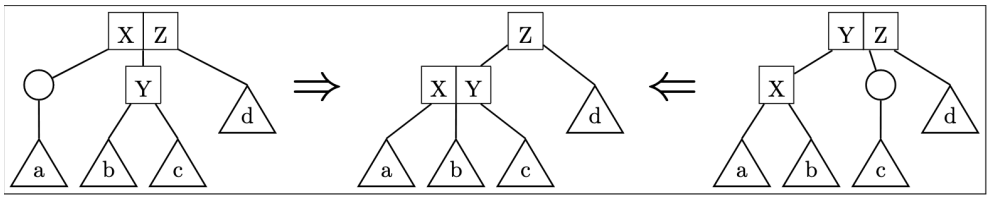
  – **Case 2**: Removing a hole with a 2-node sibling and 2-node parent.

  * 

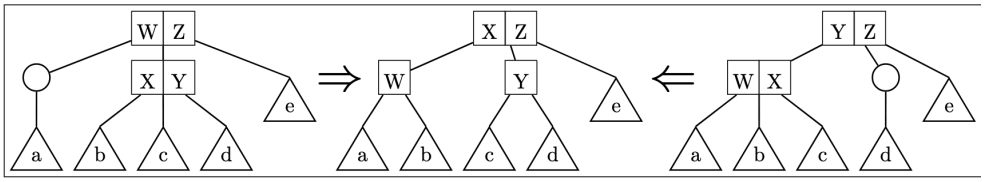
  – **Case 3**: Case 3: Removing a hole with a 3-node sibling and 2-node parent.

* 

– **Case 4**: Removing a hole with a 2-node sibling and 3-node parent.

* 

– **Case 5**: Removing a hole with a 3-node sibling and 3-node parent.

* 

These are the operations possible with a 2-3 Tree, all of which, again, have a worst case O(log(n)).

3 Overview-Red Black Trees

3.1 Definition

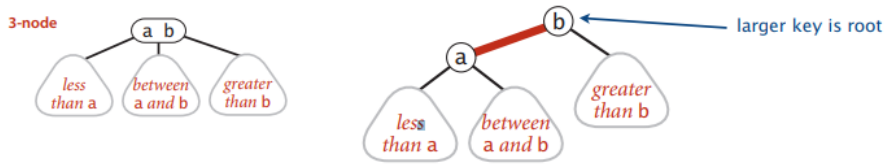
Red black trees are a data structure that improve on the binary search tree and the 2-3 tree. It attempts to maintain the efficiency of operations of the BST and maintain the guaranteed balance provided by the 2-3 tree. It also cuts down on the complexity of the 2-3 tree. A red black tree is simply a binary tree that satisfies the following red-black properties:

- Every node is either red or black, represented by a boolean bit (True/False)
- The root is black.
- Every leaf (NIL) is black.
- If a node is red, then both its children are black.
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.
- Two red nodes cannot connect directly

- Right red nodes must be rotated

If the above properties are met then the red black tree is ensured to be balanced and worst case time complexities of the elementary BST operations can be avoided.

Same tree implemented in 2-3 tree and red black tree:



3.2 Application

1. Valuable in time-sensitive applications such as real-time applications.
2. Valuable building blocks in other data structures which provide worst-case guarantees; for example, many data structures used in computational geometry can be based on red-black trees.
3. The Completely Fair Scheduler used in current Linux kernels and epoll system call implementation uses red-black trees.

3.3 Operations

- Red black trees use the elementary operations of BST; insertion, search and deletion.

However as red black trees are balanced the time complexity is much improved.

- Search: Big O is O(logn) for best and worst case.
- Insertion: Big O is O(logn) for best and worst case.
- Deletion: Big O is O(logn) for best and worst case.

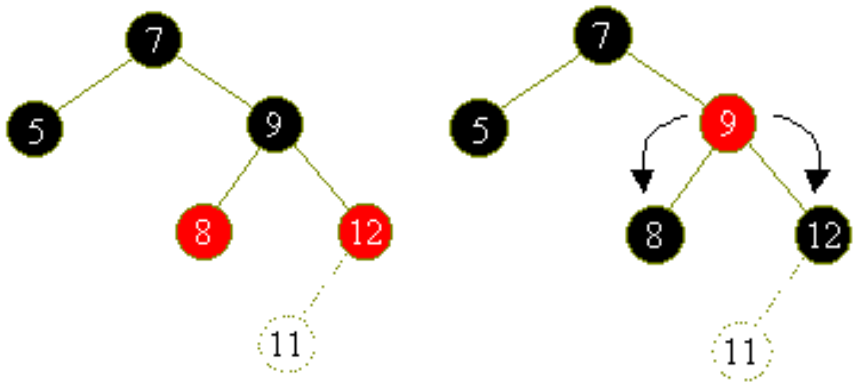
When a new node is inserted it is generally red to prevent a path having an additional black node and violating the path to null rule. As a result of deletions and insertions the tree may become unbalanced or have a double red violation. Hence two additional operations are needed to ensure that the balance and rules of red black trees are maintained. These are restructuring operations and consist of recoloring and rotating.

- **Extra Operations:**

– **Recoloring:** Recoloring is generally needed when a new insertion occurs as the new node will be red. If two red nodes are linked to a black node then a simple swap of the colours is normally sufficient. Multiple recolorings back up to the root node may be needed to ensure the red black tree rules are preserved.

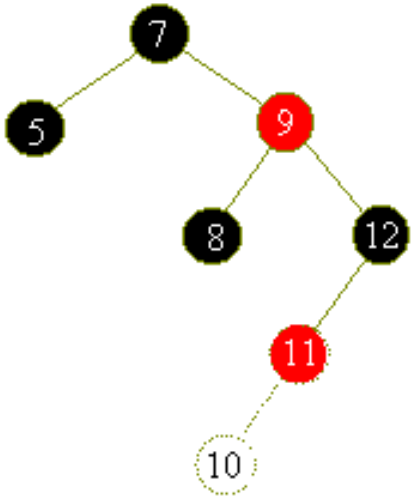
  Simple recoloring maintains integrity of the red black tree after the insertion of 11 in the sample below. Note: 11 is red.

–



– However when we attempt to insert 10 into the tree, it cannot maintain integrity with simple recoloring as seen below. For this reason an additional operation of rotating and restructuring (combination of recoloring and rotating) is needed and is outlined below.

–



– **Restructuring:** Restructuring (combination of recoloring and rotating) has a big O of O(1) for the rotations and O(logn) worst case for the recolorings. Restructure whenever the red child's red parent's sibling is black or null.

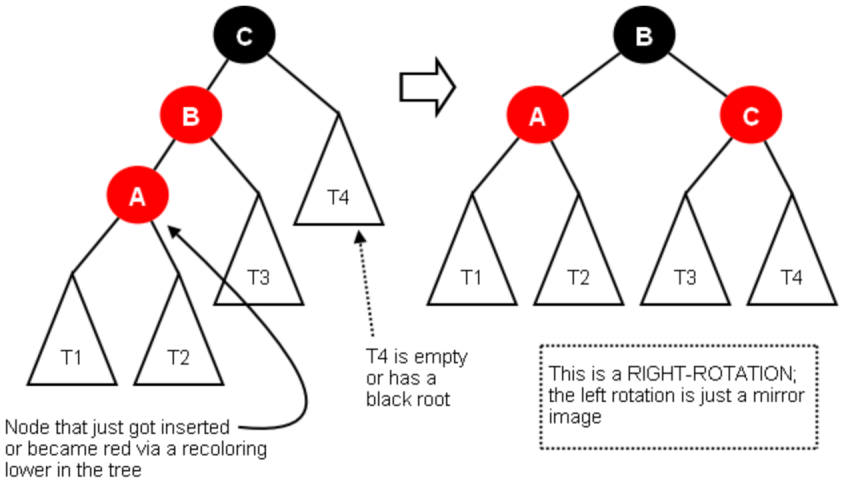
– There are 4 cases within this extra operation:

  * Right Rotation

* Left Rotation
* Right-Left Rotation
* Left-Right Rotation

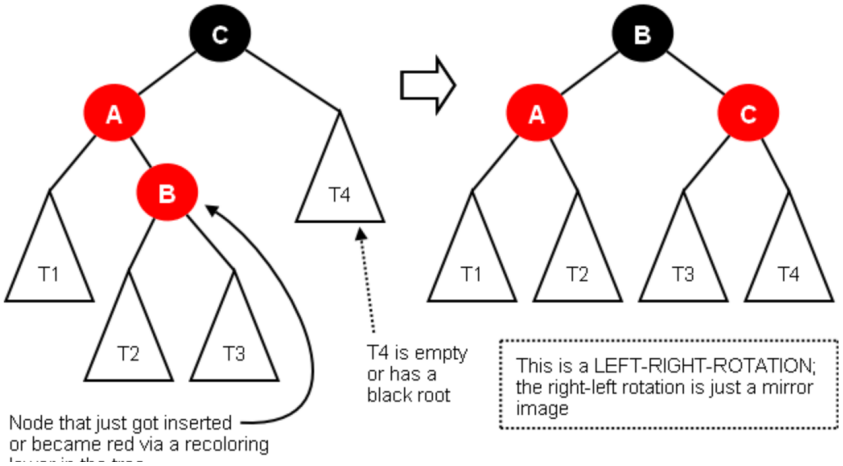
– Here is an example of a Right Rotation Restructuring:

*



– Here is an example of a Left Right Rotation Restructuring:

*



Here is a handy visual: https://www.cs.usfca.edu/ galles/visualization/RedBlack.html

3.4 References

- **Binary Search Trees**: https://algs4.cs.princeton.edu/32bst/
- **Data Structures: Binary Search Trees Explained** https://medium.com/@mbetances1002/data-structures-binary-search-trees-explained-5a2eeb1a9e8b
- **Binary search tree**: http://www.algolist.net/Data_structures/Binary_search_tree
- **CSC 378: Data Structures and Algorithm Analysis** http://www.dgp.toronto.edu/ jstewart/378notes/17rbInsertion/
- **Red-Black Tree — Set 1 (Introduction)** https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/
- **Binary Trees: Red Black**. https://towardsdatascience.com/red-black-binary-tree-maintaining-balance-e342f5aa6f5
- **2-3 Trees** https://pages.cs.wisc.edu/ deppeler/cs400/readings/23Trees/index.html
- **2-3 Trees (Search and Insert)** https://www.geeksforgeeks.org/2-3-trees-search-and-insert/
- **Day 76: 23 tree** https://medium.com/100-days-of-algorithms/day-76-2-3-tree-f20935b0e78b