CS 499 Algorithms and Data Structures

Tyler Beckham

Southern New Hampshire University

09/22/2020

“For my original project I used a simple Binary Search Tree to parse through the data once it was aligned, in order from smallest to largest, in an array. This is a very efficient method for searching through large amounts of ordered data since, if structured correctly, we start our search in the middle and cut our data in half.”

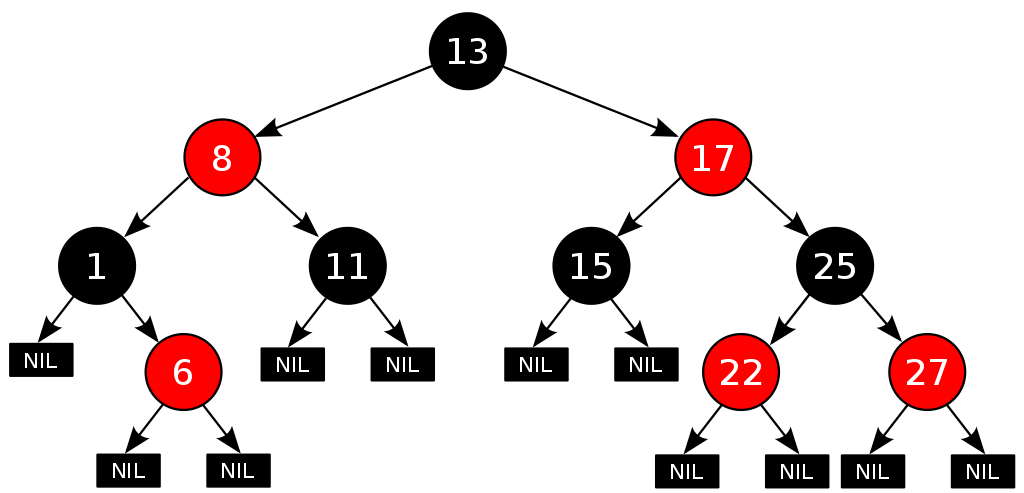
image provided by <https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm>

“The problem we run into with using an ordinary BST is when data is constantly coming in and out of the tree. This happens to be a real world scenario if we look at online retailers or bidding websites. Since the original project calls for the use of a Binary Search Tree to help solve the fictional websites data problem we will need to expand the complexity of our BST to better handle data coming in and out constantly as items are purchased and removed, or added and put on sale.

Throughout my research into Binary Search Trees I have found various different approaches to this data structure and algorithm. Some focus on balancing the tree so that data is always consistent while others focus more on speed. I needed a consistently fast tree that could handle data moving in and out and so I found what's called a Red-Black Binary Search Tree. According to geeksforgeeks.org, “Most Binary Search Tree operations take O(h) time where h is the height of the tree. The cost of these operations may become O(n) for a skewed binary tree. If we make sure that the height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of a Red-Black tree is always O(Logn) where n is the number of nodes in the tree.”(*Red-Black Tree: Set 1 (Introduction)* 2019). What this means is that because of the rules that dictate how a Red-Black tree works we can guarantee that on any path from root to leave no one path will ever be more than twice as long as any other. This balancing isn’t perfect but when you are dealing with large amounts of data that are constantly coming in and out of the tree, this type of approximate balance is beneficial.”

“Above I mentioned the rules that dictate how a Red-Black tree works. The following are the rules followed by a visual representation of the data structure.

1. Every node has a color either red or black.
2. Root of the tree is always black.
3. All leaves (NIL in the pic) are black.
4. There are no two adjacent red nodes (red nodes can not have red parent or child nodes)
5. Every path from a node (including the root) to any of its descendant NULL nodes has the same number of black nodes.



By using a Red-Black tree we can have more options when it comes to searching through our data which in turn can speed up the time it takes to retrieve our look-up. The binary search already cuts the data amount in half when it searches, but now we can cut that search in half again by using the red attribute that every other branch is assigned. This fact, when combined with a good algorithm lets us essentially look at only a quarter of the data at a time instead of half of the data.”

“The following code examples are for creating a basic Red-Black Binary Search Tree in Python. All code examples provided by <https://brilliant.org/wiki/red-black-tree/>, link to website is at the end.”

Basic Node and Tree class

class node:

def \_\_init\_\_(self, key):

self.key = key

self.red = True

self.left = None

self.right = None

self.parent = None

class RBTree:

def \_\_init\_\_(self):

self.root = None

Search Function:

def search(self, key):

currentNode = self.root

while currentNode != None and key != currentNode.key:

if key < currentNode.key:

currentNode = currentNode.left

else:

currentNode = currentNode.right

return currentNode

Insert Function:

def insert(self, key):

node = Node(key)

*#Base Case - Nothing in the tree*

if self.root == None:

node.red = False

self.root = node

return

*#Search to find the node's correct place*

currentNode = self.root

while currentNode != None:

potentialParent = currentNode

if node.key < currentNode.key:

currentNode = currentNode.left

else:

currentNode = currentNode.right

*#Assign parents and siblings to the new node*

node.parent = potentialParent

if node.key < node.parent.key:

node.parent.left = node

else:

node.parent.right = node

self.fixTree(node)

This method is an important distinction from the basic BST as it looks at the whole structure of the tree and adjusts each node as needed to keep in line with the rules mentioned above.

Fix Tree Method to decide what kind of case we are in and act appropriately:

def fixTree(self, node):

while node.parent.red == True and node != self.root:

if node.parent == node.parent.parent.left:

uncle = node.parent.parent.right

if uncle.red:

*#This is Case 1*

node.parent.red = False

uncle.red = False

node.parent.parent.red = True

node = node.parent.parent

else:

if node == node.parent.right:

*#This is Case 2*

node = node.parent

self.left\_rotate(node)

*#This is Case 3*

node.parent.red = False

node.parent.parent.red = True

self.right\_rotate(node.parent.parent)

else:

uncle = node.parent.parent.left

if uncle.red:

*#Case 1*

node.parent.red = False

uncle.red = False

node.parent.parent.red = True

node = node.parent.parent

else:

if node == node.parent.left:

*#Case 2*

node = node.parent

self.right\_rotate(node)

*#Case 3*

node.parent.red = False

node.parent.parent.red = True

self.left\_rotate(node.parent.parent)

self.root.red = False

Left Rotate Method (Right Rotate is the same just in the other direction)

def left\_rotate(self, node):

sibling = node.right

node.right = sibling.left

*#Turn sibling's left subtree into node's right subtree*

if sibling.left != None:

sibling.left.parent = node

sibling.parent = node.parent

if node.parent == None:

self.root = sibling

else:

if node == node.parent.left:

node.parent.left = sibling

else:

node.parent.right = sibling

sibling.left = node

node.parent = sibling

Sources:

Red-Black Tree: Set 1 (Introduction). (2019, November 27). Retrieved September 25, 2020, from https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/

Red-Black Tree. (2020, September 25). Retrieved September 25, 2020, from https://brilliant.org/wiki/red-black-tree/

Data Structure - Binary Search Tree. (n.d.). Retrieved September 25, 2020, from https://www.tutorialspoint.com/data\_structures\_algorithms/binary\_search\_tree.htm