Task

- Explain remove
- Explain what gets assigned lowest
- Explain Balance after removal

Remove from AbstactBalancedBinarySearchTree :

public N remove(T key)

The method takes in a key of comparable type T

N node = findNode(n_root, key);

Calls the method findnode from the extended AbstractBinarySearchTree and saves the result in the variable “N node” (N is the type)

Findnode finds if the given key exits, returning null if not found starting from the head and recursively searching the tree.

This method returns the node that is found to have the key that is being searched for or null.

if (node != null)

Next, “remove” checks if the result returned from findnode is null or not.

N lowest = node.hasBothChildren() ? removeHibbard(node) : removeSelf(node);

If “N node” isn’t null, “remove” then checks if the returned node from findnode has two children by calling .hasbothchildren (from AbstractBinaryNode).

If the return value from checking if “N node” has two children is true, then
“remove” runs removeHibbard(node) and saves that result in “N lowest”
else it runs removeSelf(node) and saves that result in “N lowest”

.hasbothchildren checks if N has both a left and right child

removeHibbard(node) swaps the smallest node of the right subtree of the node being deleted with the node being deleted. Returns the parent node of the smallest node of the right child of the node being deleted with updated children.

Does this by first getting the right node of the node being removed (i.e the successor), and the left most node of the right node of the node being removed (i.e the “min” node of successor)(“min” could be the successor node itself if it has no left child)

Continues by checking that min and successor are not the same to determine how to handle some child assignments before the official swap of “min” and the node being removed.

removeHibbard returns the parent of the “min” node with the swap of min and node already done. (if min !=successor)Parent left child should be either null or the right child of “min”

removeSelf(node) is used in the case when the node being removed has only 1 child or none. It returns the parent of the node being removed with update children.

get the parent of the node being removed (save in parent) and creates a node named “child” = null (to be used to switch with node)

if the node being removed has a left child, have “child” = node.getLeftChild() else if the node being removed has a right child “child” = node.getRightChild().

Then switch node and child.

if (lowest != null && lowest != node) balance(lowest);

Assures that if lowest (the result from removeHibbard or removeSelf) is not null (not the head of the tree) and if lowest doesn’t equal the node being removed, then to balance the tree.

How you balance depends on if you are using ALV or RedBlack.

ALV balance

Summary:

Takes in the parameter of “node” that will begin the balancing

If the node is null, it doesn’t need to be balanced – end

If node isn’t null, get the balance factor of the tree. If that factor is 2 or -2, the tree is unbalanced.

Depending on the balancefactor you rotate appropriately.

There are 4 cases resulting in

1) rotating left
2) rotating right, rotating left (zig zag case1)
3) rotating right
4) rotating left, rotating right (zig zag case2)

then recursively iterate up the tree until you reach the head.

RedBlack balance

Summary:

If the node seeking to be balanced is the root, set it to black

Else If the node being passed in’s parent is red, then get and store the uncle of node.

If the uncle isn’t null and is red, run balanceWithRedUncle else run balanceWithBlackUncle

balanceWithRedUncle sets the parent and uncle to black , grand parent to red, and then recursively to balance (uncle)

balanceWithBlackUncle If grandparent isn’t null rotate appropriately for the 4 cases. Set parent to black, and the grandparent to red.