Design Document for Red Black Tree and AVL Tree

ECE 522

Group 1

**Outline**

* Major Innovations – Additional to the project specifications
* Design Decision Rationale
* System’s limitations
* User manual
* Benchmarking
* Conclusion

**Major Innovations:**

**Red Black Tree:**

1. Preorder traversal of tree

This line of code prints out the pre-order traversal of red-black tree.

Usage:

red\_black\_tree.preorder\_trav\_print();

2. Postorder traversal of tree

The function postorder\_trav\_print() shows the result from post-order traversal of the red-black tree.

Usage:

red\_black\_tree.postorder\_trav\_print();

3. Print the whole tree.

The function print\_tree() iterates through the entire red-black tree and prints the whole tree starting from root node.

Usage:

space(u32) ->space between nodes

red\_black\_tree.print\_tree();

**AVL Tree:**

1. Searching for nodes in an AVL tree.

When the value of a node to be searched is provided to the find\_node function, it iterates through the tree and returns the pointer to the node if it exists in the tree. Otherwise, it outputs a value of node\_ptr(0x0), indicating that the node is not present in the tree.

Usage:

println!(“{:?}, AVL\_tree.find\_node(value of node as usize));

**Design Decision Rationale:**

1. What does a red-black tree provide that cannot be accomplished with ordinary binary search trees?

An ordinary binary search tree has a worst-case scenario of getting unbalanced which will result in O(N) time for inserting, removing and searching operations on the nodes. It is very easy to get into a trap to create an unbalanced tree and once an ordinary binary search tree gets unbalanced, then it cannot be repaired.

The Red-black tree overcomes limitations of ordinary binary search trees by adding the ability to rebalance the tree after insertion or deletion operation. So, the same operations can be performed in O(log(n)) time. Also, the balancing of tree makes searching much faster than the Binary search trees.

2. Please add a command-line interface (function main) to your crate to allow users to test it.

We have a “main.rs” file which allows users to see how the functions can be used by our code. It provides the users with a list of options from where they can choose if they want to work with a Red-black tree or an AVL tree. We have also provided a list of operations such as, insertion, deletion of nodes, counting number of leaves as options from where they can simply type the number in the list and produce the required outputs. When users want to opt out of a function, they can type (-1) for Exit and return to the previous state.

3. Do you need to apply any kind of error handling in your system (e.g., panic macro, Option<T>, Result<T, E>, etc..)

An important feature of our program is the use of Option<T> that allows the tree to either contain nodes within itself or even remain empty. When a tree is initialized, it does not contain any nodes in the beginning, unless a user explicitly inserts node into it. Here the Option<T> pointer allows the tree to be initialized even without values of nodes in it.

Result<T, E> has been used to handle error with return values. Besides, we have used ‘Error’ from the standard I/O library to efficiently handle error due to duplicate nodes and also undefined errors in the program.

4. What components do the Red-black tree and AVL tree have in common? Don’t Repeat Yourself! Never, ever repeat yourself – a fundamental idea in programming.

The common components in Red black Tree and AVL tree are given as follows:

• Creating a new tree (function name: new)

• Tree operations i.e. Rotate Left, Rotate Right. (function names: rotate\_left, rotate\_right)

• Minimum value of the tree. (function name: min)

• Searching values in the tree. (function names: find/contains)

• Getting the height of the tree. (function name: height)

• Obtaining number of leaves in a tree. (function names: count\_leaves, print\_leaves)

5. How do we construct our design to “allow it to be efficiently and effectively extended”? For example. Could your code be reused to build a 2-3-4 tree or B tree?

Our code can be reused to implement a 2-3-4 tree or a B-tree.

A 2-3-4 tree has 2 or 3 or 4 nodes and the same number of data elements in it. We can modify our existing code for TreeNode such that it can implement a 2-3-4 tree as follows,

struct TreeNode<K: Ord, V> {

key:[ K;4],

value: [V;4],

parent: node\_ptr<K, V>,

left: node\_ptr<K, V>,

right: node\_ptr<K, V>,

level: usize,

}

A B-tree can have multiple keys and children. In that sense, we can implement a B-tree using a vector in our TreeNode structure as follows,

struct TreeNode<K: Ord, V> {

key: Vec<usize>,

value: Vec<usize>,

parent: node\_ptr<K, V>,

left: node\_ptr<K, V>,

right: node\_ptr<K, V>,

level: usize,

}

**System’s Limitations:**

**\*\***

**User Manual:**

**Red Black Tree:**

1. Creating a red black tree.

This line of code initializes an empty red black tree.

Usage:

let mut red\_black\_tree: RBTree<usize, usize> = RBTree::new();

2. Inserting a node into the red-black tree

The function ‘insert()’ pushes the key and value of a node into the red-black tree. If a node already exists in the tree, it doesn’t allow the duplicate node to be inserted.

Usage:

red\_black\_tree.insert(key of node, value of node).unwrap();

2. Deleting a node from red-black tree

The remove\_node() function deletes the node with value of the node from the red black tree, where the node to be deleted is taken as usize of the node.

Usage:

red\_black\_tree.remove\_node(&( value of node as usize)).unwrap();

3. Counting the number of leaves in a tree.

This function gets length of a tree and prints out the number of leaves in the red black tree.

Usage:

println!(“The number of leaves in the red black tree is: {}, red\_black\_tree.len());

4. Returning height of the tree.

The function get\_height() returns the height of the red black tree.

Usage:

println!(“The height of the tree is: {:?}”, red\_black\_tree.get\_height());

5. Printing out in-order traversal of the tree.

This line of code prints out the in-order traversal of red-black tree.

Usage:

red\_black\_tree..inorder\_trav\_print();

6. Checking if the tree is empty.

The function is\_empty() returns a Boolean value indicating emptiness of the red black tree. If there are no nodes in the tree, it returns a ‘true’ value, else it returns ‘false’.

Usage:

println!(“Is the red black tree empty?: {}”, red\_black\_tree.is\_empty());

**AVL Tree:**

1. Creating an AVL tree.

This line of code initializes an empty AVL tree.

Usage:

let mut AVL\_tree: AVLTree<usize, usize> = AVLTree::new();

2. Inserting a node into the AVL tree

The insert() function pushes a new node into the AVL tree. Here, a node with key 5 and value 5 is inserted into the AVL tree.

Usage:

AVL \_tree.insert(5, 5).unwrap();

2. Deleting a node from AVL tree

This function deletes a specified from the AVL tree. For example, the node 7 is deleted here from an existing AVL tree.

Usage:

AVL \_tree.remove\_node(&( 7 as usize)).unwrap();

3. Counting the number of leaves in a tree.

This function gets length of a tree and prints out the number of leaves in the AVL tree.

Usage:

println!(“The number of leaves in the AVL tree is: {}, AVL \_tree.len());

4. Returning height of the tree.

The function get\_height() returns the height of the AVL tree.

Usage:

println!(“The height of the AVL tree is: {:?}”, AVL \_tree.get\_height());

5. Printing out in-order traversal of the tree.

The inorder\_trav\_priint() function prints out the AVL tree by performing an in-order traversal.

Usage:

AVL \_tree..inorder\_trav\_print();

6. Checking if the tree is empty.

The function is\_empty() returns a Boolean value indicating emptiness of the AVL tree. If there are no nodes in the tree, it returns a ‘true’ value, else it returns ‘false’.

Usage:

println!(“Is the AVL tree empty?: {}”, AVL \_tree.is\_empty());

**Benchmarking:**

\*We need a function to search for the elements in the AVL tree.

**Conclusion:**