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. Printing the whole tree red black 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(1);
```

3. Printing subtree of a red-black tree.

```
Usage:
red_black_tree.print_subtree(1, &(node_v as usize));
```

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 as node_ptr<K, V> 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));
```

2. Post-order traversal of tree

The function postorder_trav_print() shows the result from post-order traversal of the AVL tree.

```
Usage:

AVL_tree.postorder_trav_print();
```

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 trees are given as follows:

Creating a new tree [Function name: new()]
Insertion into a tree [Function name: insert()]
Deleting a node from a tree [Function name: delete()]
Counting number of leaves in a tree [Function name: len()]
Returning height of a tree [Function name: get_height()]
In-order, Pre-order and Post-order traversals in a tree [Function names: inorder_trav_print(), preorder_trav_print(), postorder_trav_print()]
Searching for nodes in a tree [Function name: find_node()]

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:

Our Red-black and AVL trees work as per the design requirements and can successfully produce the expected results. However, there are some limitations in our system that can be noted as follows,

1. Our system was developed using std::ptr which means that there is a lot of unsafe code blocks that might lead to unknown errors during development. Although after testing the system for multiple times, we were not able to find any problem, we'd say a safer approach would be using RefCell and Rc. In our defense, we can say that using standard pointer might lead to a faster performance.

2. Unfortunately, our AVL tree does not handle a node deletion with two nodes properly, and is advised to only remove nodes with one child or no children.

3. No clear function was implemented in our system to free the memory after we have produced the trees. So, it's all dependent on the compiler. We plan to add this feature in the future versions of our system.

We have uploaded our code on GitHub in a private repository, that we plan to release as a crate and an open source system in the future once it is finalized.

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 of type K and value of type V 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();
```

3. 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();
```

4. 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());
```

5. 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());
```

6. 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();
```

7. 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 the key value of 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();
```

3. 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();
```

4. 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());
```

5. 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());
```

6. 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();
```

7. 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 perform benchmarking of Red-black and AVL trees in two steps. First, we compare the results for both trees in case of insertion and search time for a set of values. Then we perform comparison only on insertion to see if one tree produces better result consistently than the other.

Comparison of Benchmark Results for Insertion and Search:

In the bar chart, the x-axis denotes the number of elements inserted into the tree and y-axis denotes the time taken to complete insertion and search in milliseconds. By comparing the results between Red-black tree and AVL tree, we can see that Red-black tree has better performance and takes lesser time to insert and search elements compared to AVL tree. The reason behind this may be due to fewer rotations in Red-black tree while performing insertion. Although, an AVL tree has a faster searching capability, we can see that the performance in insertion was showcased more in this test.

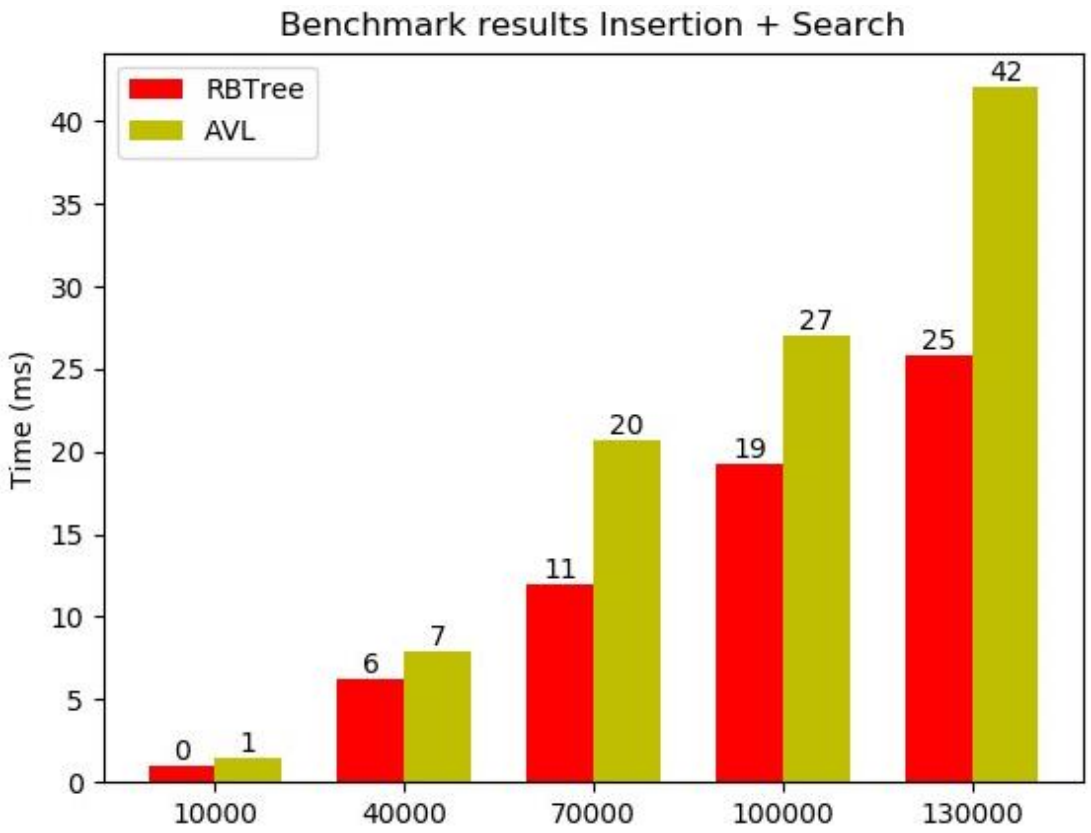

Figure 1: Benchmark results for Insertion and Search

Comparison of Benchmark Results for Insertion:

We compared the results for insertion of 10,000, 40,000, 70,000, 100,000 and 300,000 elements intro the Red-black and AVL trees. It is seen that performance of Red-black trees dominate over AVL trees in terms of insertion.

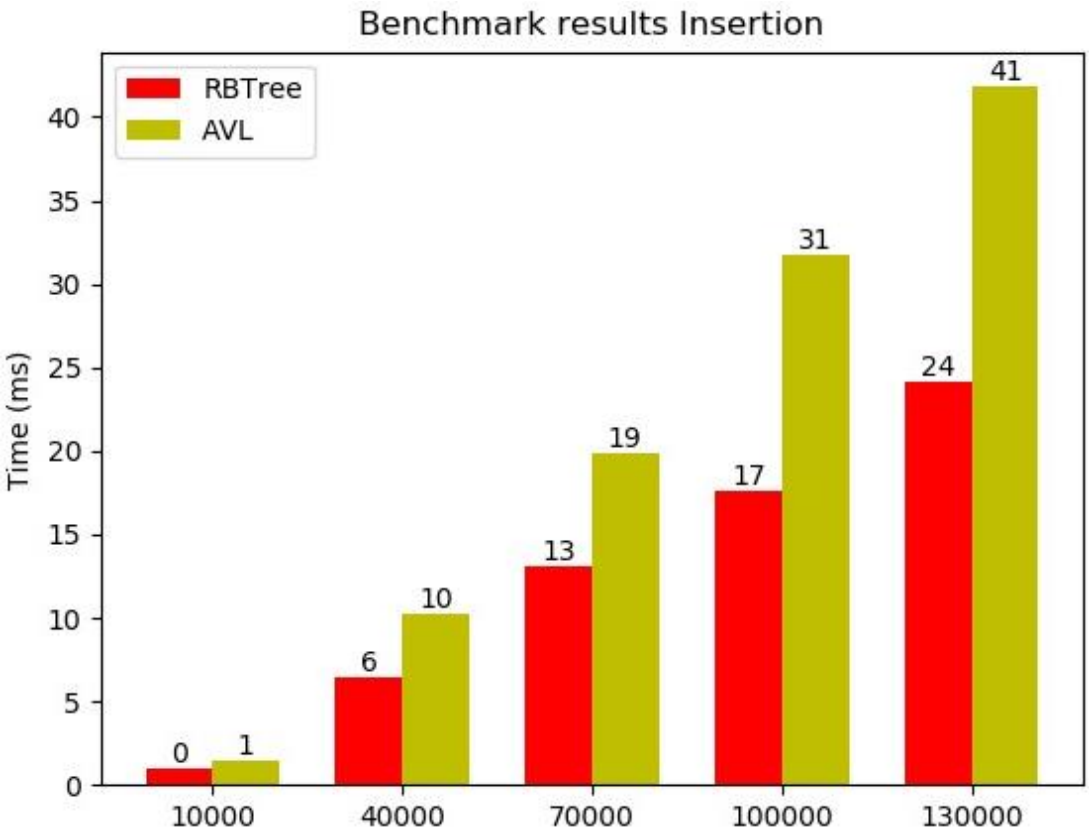

Figure 2: Benchmark results for Insertion

Conclusion:

In this project, we have implemented two important data structures, Red-black and AVL trees that are capable to store and retrieve large amount of data with great speed. On performing the benchmark of these trees for their insertion and search functions, we have observed that the Red-black tree shows better performance than AVL tree in terms of insertion and searching for elements. We have also accommodated an additional test case for insertion of elements in which also the Red-black tree shows better performance. From these results, we can infer that Red-black tree is a more efficient data structure in our system.

We know that the search time for Red-black and AVL tree is O(log(n)), where as search time for Binary tree is O(n). Using an additional data structure such as, binary tree in the benchmark would provide us an idea about the improvement in speed in these trees compared to the conventional data structures. So, it would be beneficial to perform comparison with additional data structures to measure the efficiency across different systems.