ECE 522 Group Project

Trees, Trees, and more Trees

Design Document

Group members: Xiaohan Liu
Yaokun Yu
Yongfan Wang

Dec 9th 2021

University of Alberta

Menu

1. Major Innovations

For Red-black tree:

1. *get_parent* function: if the parent node of the given node exists and this function can return the parent node.
2. g*et_grandparent* function: if the grandparent node of the given node exists and this function can return the grandparent node.
3. *search_succ* function: this function can return the successor node of the given node. The successor node is the deepest left child of the right child of the given node, which means the key of the successor node is the minimum of nodes of the right side of the given node. The successor node would be used in the deleting operation of red-black tree.
4. *search_node* function: it can check if there is a node with the key equaling to the given value and return the node if so.
5. Instead of assigning a black color to the nil leaves, I use the variables s_left_color and s_right_color to represent the color of the nil leaves. Given a node that needs to be deleted, those two variables would be 0 when the left child and the right child sibling node are nil leaves.

For AVL-tree:

1. *is_balanced* function: this function can be used to judge any binary tree to see if it is self-balanced, i.e. the difference between the height of two child-trees is no greater than one.
2. When doing the delete operation, always choose the smaller node to replace its father node node if both left and right child-nodes of the father node exist. This operation will ensure the consistency of the result after delete operation.

For common methods for binary trees.

1. *get_max* function: get the maximum number of the tree.
2. *get_min* function: get the minimum number of the tree.
3. *len* function: get the length of the tree.
4. *get_left* function: get the left child of the tree.
5. *get_right* function: get the right child of the tree.
6. *get_root* function: get root of the tree.

2. Detailed rationale for augmented decisions

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

In rea-black tree, Each node stores an extra field "color" ("red" or "black"), it is used to ensure that the tree remains balanced during insertions and deletions. Red-black tree can get the worst time complexity of insertion, deletion, and searching with O(logn), which means it beats the ordinary search trees that whose worst time complexity can reach O(n) during operations. Also, the properties of red-black tree can prevent it from degenerating into a Linked List during operations (a single chain containing 3 nodes is not possible in a red-black tree because it may disobey property 4 and property 5 of red-black trees), as ordinary search trees may do. In addition, compared with AVL tree, red-black tree can avoid the resources consumed by frequent rotation during insertion and deletion operations, thus obtaining higher efficiency. If more insertion or deletion operations are needed, we should use red-black trees.

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

(Some picture of function main)

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

Yes, we use Option<T> in my codes. This kind of error handling guarantees that the corresponding variable either has no value: None, or has a value: Some(T). With Option<T>, we can avoid forgetting to check for null/none in the code of this project, as is often the case with other programming languages, and this is one of the things that Rust does to ensure that its code is robust.

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.

Basically speaking, both of them are a special kind of binary tree, the height, only different in insert/delete actions or query. The traversal logic of them is the same as a binary tree. So we extract the functions such as getting children which can be reused into an independent mod. Both trees can use functions inside.

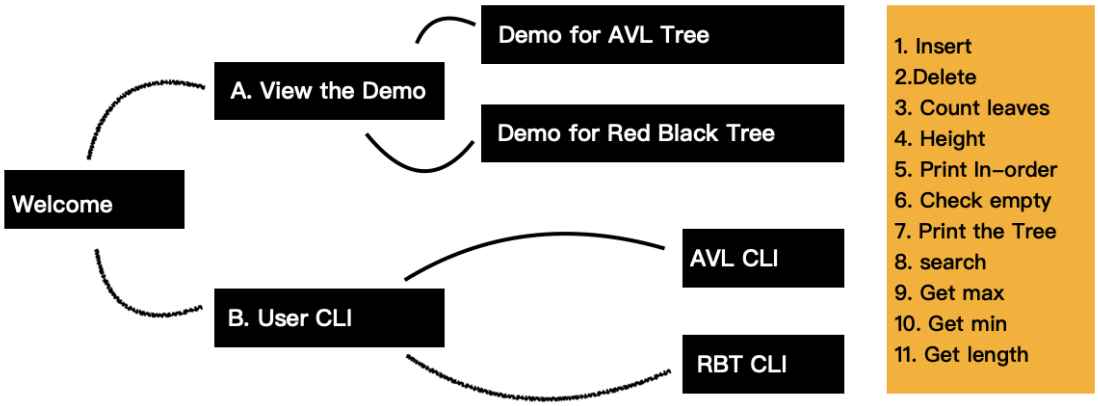
5- How do we construct our design to “allow it to be efficiently and effectively extended”?For example. Could your code be reused to build a 2-3-4 tree or B tree?

The structures of trees in our codes cannot be reused because a node of 2-3-4 trees or B trees can have more than two children. And the insertion and deletion functions need to be modified for new trees. During modification, we can take the advantage of similar concepts between old trees and new trees and make some codes reused. For example, we can take the 2 nodes of a 2-3-4 tree as a black node of the red-black tree in our project.

Also, a 3 node of the 2-3-4 tree can be regarded as a black node of the red-black with a left red child or a right red child. What's more, we can take the black node with two red children as a 4 node of a 2-3-4 tree. However, the functions we defined in base may be reused. For example, we can use the print function to print the 2-3-4 trees or B tree.

3. User Manual

1. Features at a glance



This system mainly provides 2 modules.

1. View Demo
2. Command Line Interface for Trees

1.1 View Demos of AVL Tree and Red Black Tree

Show a simple demo for AVL or Red Black Tree with multiple operations and functions.

1.2 Functions in Command Line Interface

For each tree, we provide 11 operations for users to use.

1.2.1 Functions for AVL Tree

1. Insert a node into AVL Tree
2. Delete a node from AVL Tree
3. Count the number of leaves in a tree
4. Return the height of a tree.
5. Print In-order traversal of the tree.
6. Check if the tree is empty
7. Print the Tree
8.Search a node in AVL Tree.
9. Get the max node in AVL Tree

10. Get the min node in AVL Tree
11. Get the length in AVL Tree

1.2.2 Functions for Red Black Tree

1. Insert a node into Red Black Tree
2. Delete a node from Red Black Tree
3. Count the number of leaves in a tree
4. Return the height of a tree.
5. Print In-order traversal of the tree.
6. Check if the tree is empty
7. Print the Tree
8. Search a node in the Red Black Tree.
9. Get the max node in Red Black Tree
10. Get the min node in Red Black Tree
11. Get the length in Red Black Tree

2. System instruction

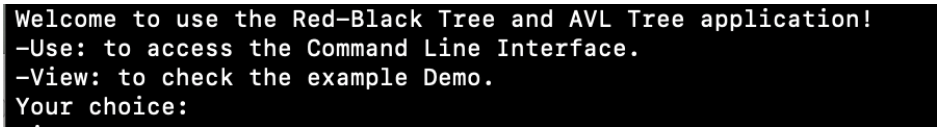
Enter the crate path.

- For macOS: In the terminal, use the ‘cd’ command.
- For Windows: In CMD, use the ‘cd’ command.
- For Linux: In shell, use ‘cd’ command.

Input the following command: cargo run (to enter the main interface of the system).

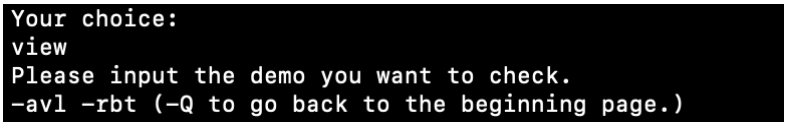
2.1 main interface

-Use: Input ‘Use’(lowercase is accepted), to access the Command Line Interface.(2.3)
-View:Input ‘View’(lowercase is accepted), to check the example Demo.(2.2)



2.2 View interface

The interface for example demo.
-avl: Input ‘avl’(uppercase is accepted), to check the preset demo for AVL Tree.
-rbt: Input ‘rbt’(uppercase is accepted), to check the preset demo for red-black Tree.
-Q to return to the main interface.



You can choose a second demo if you wish.
(-Q to return to the main interface.)

2.3 CLI for Trees

The main interface for trees.
Shows the Trees menu and Operation menu.
-AVL: Input ‘AVL’(lowercase is accepted), to access the AVL Tree command line interface.(2.3.1)
-RBT: Input ‘RBT’(lowercase is accepted), to access the Red Black Tree command line interface.(2.3.2)
-help: to get more help about the commands in this page.(2.4.1)
-Q: to return to the main interface

```
----------------------------------<<<<Trees, Trees, and More Trees>>>>----------------------------------
Trees menu
----------------
1-AVL Tree(avl)
2-Red-Black Tree(rbt)
----------------

Operation menu
----------------
1-insert
2-delete
3-height
4-count
5-empty
6-search
7-print
8-print in order
9-min
10-max
11-length
----------------
----------------------------Let's get start!----------------------------
--------------------------------------------------------------------------------------------
Now input a tree type to start.
(-Q to exit.-help to get more help about the commands.)
-AVL
-RBT
```

2.3.1 AVL Tree Interface

Operation menu for AVL Tree is shown.
-x: Input operation ‘x’ to commit that operation on AVL Tree.
-help: show the operation menu.(2.4.2)
-Q to return to the CLI for trees.

```
AVL
::<<<<<AVL Tree>>>>>::

Operation menu on trees:
---------------

(note: 1.Input the operation 2.press enter then input the value.)
---------------
-insert   -insert a node into the tree.
-delete    -delete a node from the tree.
-count    -count the number of leaves of the tree.
-height   -return the height of the tree
-empty    -check if the tree is empty
-search   - search from the tree).
-print    -print the in-order traversal of the tree
-printall    -print the tree
-max     - find the maximum value in the tree
-min     - find the get_minimum value in the tree
-length     - get the length of the tree
-Q     - exit and delete the current tree
```

2.3.2 Red-Black Tree Interface

Operation menu for Red-Black Tree is shown.

-x: Input operation ‘x’ to commit that operation on Red Black Tree.

-help: show the operation menu.(2.4.2)

-Q to return to the CLI for trees.

```
RBT
::<<<<<Red Black Tree>>>>>::

Operation menu on trees:
---------------

(note: 1.Input the operation 2.press enter then input the value.)
---------------
-insert   -insert a node into the tree.
-delete    -delete a node from the tree.
-count    -count the number of leaves of the tree.
-height   -return the height of the tree
-empty    -check if the tree is empty
-search   - search from the tree).
-print    -print the in-order traversal of the tree
-printall    -print the tree
-max     - find the maximum value in the tree
-min     - find the get_minimum value in the tree
-length     - get the length of the tree
-Q     - exit and delete the current tree

your operation:
(-Q to exit,-help for all operations)
```

2.4 Help

2.4.1 Command help for main CLI

The ‘help’ for main CLI.

```
Commands help:
---------------

-avl  - Get start with AVL Tree.
-rbt  - Get start with Red Black Tree.
-Q - Exit the program.
-help - Show more commands.
```

2.4.2 Operation menu for trees

The ‘help’ for operation on 2 trees.

```
Operation menu on trees:
---------------

(note: 1.Input the operation 2.press enter then input the value.)
---------------
-insert   -insert a node into the tree.
-delete    -delete a node from the tree.
-count    -count the number of leaves of the tree.
-height   -return the height of the tree
-empty    -check if the tree is empty
-search   - search from the tree).
-print    -print the in-order traversal of the tree
-printall    -print the tree
-max     - get the maximum value in the tree
-min     - get the get_minimum value in the tree
-length     - get the length of the tree
-Q     - exit and delete the current tree
```

3. Example Operation

1.Insert a node:

```
your operation:
(-Q to exit,-help for all operations)
insert
3
insert value 3 is in progress...
done!
```

2.Count the leaves:

```
your operation:
(-Q to exit,-help for all operations)
count
The number of leaves of the current avl tree is:2
```

3.Return the height:

```
your operation:
(-Q to exit,-help for all operations)
height
The height of the current avl tree is:2
```

4.Is tree empty?:

```
your operation:
(-Q to exit,-help for all operations)
empty
Q:Is this an empty tree? A: false
```

5.Search a number in the tree:

```
your operation:
(-Q to exit,-help for all operations)
search
8
search value 8 is in progress...
Q:8 exist in tree? A: true
```

6.Print the tree in order:

```
your operation:
(-Q to exit,-help for all operations)
print
The in-order traverse of your tree is:
Print in_order traverse:
3 4 8
```

7.Min:

```
your operation:
(-Q to exit,-help for all operations)
min
The min value in your tree is:3
```

8.Max:

```
your operation:
(-Q to exit,-help for all operations)
max
The max value in your tree is:8
```

9.Length:

```
your operation:
(-Q to exit,-help for all operations)
length
The length of your tree is:3
```

10.Print the tree:

```
your operation:
(-Q to exit,-help for all operations)
printall
Your current avl tree is:AVLTree {
    root: Some(
        RefCell {
            value: TreeNode {
                val: 4,
                height: 2,
                left: Some(
                    RefCell {
                        value: TreeNode {
                            val: 3,
                            height: 1,
                            left: None,
                            right: None,
                        },
                    },
                ),
                right: Some(
                    RefCell {
                        value: TreeNode {
                            val: 8,
                            height: 1,
                            left: None,
                            right: None,
                        },
                    },
                ),
            },
        },
    ),
}
```

4. Benchmarks

We use 9 different test cases.
Here is the result for iteration from [1000,5000,10000,20000,40000,70000,90000,100000,130000].
From the benchmark result, we can see that the red-black tree is the most efficient, for a tree size of 130000, the average insert and search time is 26ms.
And as a baseline, binary search tree performs worst, which has an average time of 96s.