Overview

In this assignment, we create a custom data structure in which, when we add or find any data, it becomes the root node of the tree. In deletion, it acts opposite to adding and finding, where we take that node as a leaf node and then delete it.

Our custom data structure, named WorkingSetTree, implements one interface called Searchable, which contains 7 abstract methods: add, find, remove, size, levelOf, serialize, and rebuild. These methods contain all the necessary steps needed for CRUD operations with any data structure.

This custom data structure only accepts inputs as strings; other inputs are not acceptable. Moreover, all inputs are considered case-insensitive. The most important point about this data structure is that it follows the rules of a binary search tree, where all the nodes on the left side of the root node are smaller than the root value, and all the nodes on the right side are larger than the root value.

Files and External Data

This implementation is divided into 4 Java classes:

Node: This class defines the basic structure of each node present in the tree. It maintains left, right, and parent pointers for that node and also contains some string data.

Searchable: This interface contains method signatures, and the implementation of these methods will be done by our Java class that contains the main method.

TreeFunctions: This class contains methods that are frequently needed during the implementation of methods present in the interface. Functions like calculating the height of the tree, traversing through the entire tree, and checking whether the tree is a valid BST or not after rebuilding has been done.

WorkingSetTree: This class contains our main method and implements the Searchable interface. It also contains the logic of rotations, such as left and right rotations of nodes during add, find, and delete operations.

Data structures and their relations to each other

Time and space efficiency are not requirements of the solution. Consequently, we choose the simplest data structures that will do the task.

Node:

- ➔ Stores information about each node, including the key value, references to its left and right children, and a reference to its parent. This parent reference is useful during left or right rotations.

Array :

- ➔ Stores the keys of nodes in a level order manner to represent the tree's structure during serialization.
- ➔ The same array format we take as a input during rebuilt of the tree.
- ➔ Stores the keys of nodes accessed during operations such as insertion, search, or deletion. This history is used for maintaining access patterns used during splay operations.

Assumptions

- ➔ Inputs to this data structure must be strings.
- ➔ Input strings are case-insensitive.
- ➔ During the rebuilding process, the root node is expected to be at the 1st index, and for any other node:
    - If it is at index x, then its left child is at 2x, and its right child is at 2x+1.
- ➔ Duplicate inputs are not allowed.
- ➔ Any operations related to integers will not be performed by this data structure.

Choices

- ➔ Node: Here, I maintain a parent pointer in each node, which makes it easy during left or right rotations in a tree.
- ➔ Null values are not allowed in the tree. If this choice is not taken, we may encounter many null pointer exceptions.
- ➔ A dynamic array is used to store the string values that were last accessed. Opting for a static size for this array could result in overflow at any time, potentially causing index out of bounds exceptions.

Key algorithms and design elements

add(key):

```
if tree is empty:
    create root node with key
else:
    traverse tree to find appropriate position for key
    insert key as leaf node in correct position
    perform necessary rotations and make that newly created node as the root node
```

find(key):

```
start from root node
traverse tree while key is not found:
    if current node's key matches target key:
        perform splay operation to bring node to root
        return node
    else if target key is less than current node's key:
        move to left child
    else:
        move to right child
if key not found:
    return null
```

Splay(node):

```
while node has parent:
    if node’s parent is root:
        If node is in left side of root
                perform single right rotation on root
        If node is in right side of root
                perform single left rotation on root
    else:
        If node is left child of parent and parent is left child of grandparent:
                perform right rotation on grandparent
                perform right rotation on parent
        else If node is right child of parent and parent is right child of grandparent:
                perform left rotation on grandparent
                perform left rotation on parent
        else If node is left child of parent and parent is right child of grandparent:
                perform right rotation on parent
                perform left rotation on grandparent
```

else

perform left rotation on parent

perform right rotation on grandparent

Serialize():

Calculate the height the tree

Create array of size 2^height of the tree

perform preorder traversal of tree and store keys in array

return array

Rebuild(keys):

if keys array is empty or null:

return null

recursively build tree from keys array

Check the newly created tree is BST or not

If checkBinaryTree()

set the root by newly created root node

return true;

else

return false;

Limitations

Due to rotations, this tree may sometimes become left or right skewed. As a result, space may not be effectively utilized, and the time complexity for some operations can reach O(n) in the worst case.