CS361 Project 2

Behnoud Alaghband[1], Robert McDaniels[2], Dmitri Azih[3], and Dylan Rivas[4]

[1]Department of Computer Science, University of New Mexico
[2]Department of Computer Science, University of New Mexico
[3]Department of Computer Science, University of New Mexico
[4]Department of Computer Science, University of New Mexico

April 2024

Project Goals

This project focuses on implementing binary search tree data structure in Java while applying the Singleton design pattern to represent empty trees (`Nil`). This data structure supports operations such as insertion, deletion, tree creation from array, finding the minimum node, and verifying whether the tree is a binary search tree (BST).

BST Implementation

The project employs the interface `BT` to define the structure and behavior of a binary search tree data structure. This interface includes methods for insertion and deletion, computing the height of a tree, converting the tree to a string representation, and checking if the tree is a BST. Inside the interface are two classes, `Nil` and `Node`, implemented to represent empty trees and nodes, respectively.

`Node`

The `Node` class represents nodes in the binary tree. Each node contains an integer value, references to left and right subtrees (left and right), and implements the methods defined in the `BT` interface.

`Nil`

The `Nil` class represents empty trees in the BST implementation. It uses a Singleton pattern to ensure that only one instance of `Nil` exists throughout the program. Doing so, optimizes memory usage by reusing the same empty tree instance wherever needed. The `Nil` class provides inherits methods from the `BT` interface.

`Static Methods & Example Usage`

`BT` includes static methods like `fromArray()` for constructing a BST from an array of integers.
`BT` also includes a main method demonstrating the creation of a binary search tree with sample nodes and printing its height and string representation.

1 Binary Tree `toString` Implementation

Method was written to override the method `toString()` in the `Node` class, such that the structure of the binary tree can be translated to the following string fdigitormat:

`Node(num, string_of_left_subtree, string_of_right_subtree)`

For example, the tree shown above is translated to the string:

`Node(3, Node(1, Nil, Node(4, Nil, Nil)), Node(2, Nil, Nil))`

Below outlines the implementation of the `toString()` method for the `Node` and `Nil` subclass.

1.1 Explanation of Implementation

Listing 1: Implemenation for `Node` subclass

```
public String toString() {
    return "Node(" + num + ", " + left + ",
    " + right + ")";
}
```

Listing 2: Implemenation for `Nil` subclass

```
public String toString() {
    return "Nil";
}
```

This method returns the string that starts with "Node" which is the data part of the node (num). This method calls the left child and then the right child, but if each of them is Nil, it returns Nil.

1.2 Running Time

As this method performs pre-order traversal and each node is visited once, the running time of this method is $O(n)$.

2 Binary Tree Validation

An abstract method `isBST` was written to the `BT` interface, which is overridden by its subclasses (`Node` and `NIL`). This method verifies if the binary tree is a BST under certain restrictions. **Restrictions:** The method must not use loops or check if a reference is null or not.
Below outlines the implementations of `isBST()` for each subclass.

2.1 Explanation of Implementation

Listing 3: Implemenation for `Node` subclass

```
public boolean isBST() {
    return (
    left.compare(num) <= 0 &&
    right.compare(num) >= 0 &&
    left.isBST() &&
    right.isBST());
}
```

Listing 4: Implemenation for `Nil` subclass

```
public boolean isBST() {
    return true;
}
```

This method checks if a tree satisfies the binary tree properties. By the definition of a BST, any given node's left subtree must be less than or equal to that parent node and all node values of right subtree must be greater than or equal to that parent node. `left.compare(num)` $\leq$ `0` compares the parent node's value to the left child and it returns -1 if it is less, 0 if it is equal, and 1 if it is greater. The only valid results are 0 and -1. Similarly, `right.compare(num)` $\geq$ `0` checks if the right subtree has greater or equal value that must result for 1 and 0 for correctness of BST. Recursive calls are made to left.isBST() and right.isBST() to check even the current subtrees satisfy the BST properties.

2.2 Running Time

This method must traverse through every node in the tree to check if the BST properties hold true, therefore, the running time is $O(n)$.

3 Binary Search Tree Construction

A static method `fromArray()` was written for the `BT` interface. The method takes an array of integer values and creates a BST by consecutively calling `insert()` on these values into an empty tree (represented as `Nil`). The method returns a reference of a `Node` object.
Below outlines the implementation of this static method and the `insert()` method, explaining how each function contributes to the construction of the BST.

3.1 Explanation of Implementation

Listing 5: Helper method for `fromArray()` for inserting values into tree

```
public BT insert(int n) {
    if(n <= num) {
        left = left.insert(n);
    }
    else {
        right = right.insert(n);
    }
    // Always return this for nodes
    return this;
}
```

Listing 6: Helper method implementation for Nil specifically

```
public BT insert(int n) {
    return new Node(n, nil, nil);
}
```

Listing 7: Static method implementation for BT interface

```
public static BT fromArray(int[] arr) {
    BT tree = nil;
    for(int v : arr) {
        tree = tree.insert(v);
    }
    return tree;
}
```

This method is for converting the array of integers to a binary search tree with inserting each element of the array to the tree in sequence. We iterated over array arr and for each integer value v inserted that value to initially empty BST while maintaining it properties. This method returns final BST after all insertions. For Nil, when a node is "inserted", instead a new one is created to build a tree of size 1 (height 0).

3.2 Running Time

The running time of insert() depends on the height of the tree that is $O(\log n)$ for n nodes. The total running time of inserting n nodes to BST is $O(n \log n)$ as each insertion takes $O(\log n)$ time.

4 Binary Search Tree Node Deletion

Wrote a method to delete the node that has the given key from a binary search tree (BST), assuming the tree initially conforms to the properties of a binary search tree. Below are implementation details for special cases.

4.1 Explanation of Remove() Implementation

Listing 8: Finds the mininum node from within a given tree (Node)

```
public Node min() {
    return left.compare(num) < 0 ? left.min() : this;
}
```

Listing 9: Removes given key from tree

```
public BT remove(int n) {
    // Found the key, replace with the minimum of the
       right subtree
    if(num == n) {
        // nil.min() is not valid
        if(right == nil) return left;
        Node succ = right.min();
        right.remove(succ.num);
        succ.left = left;
        succ.right = right;
        return succ;
    }
    // Otherwise, recurse
    else if(n < num) {
        left = left.remove(n);
    }
    else {
        right = right.remove(n);
    }
    // This tree level didn't change
    return this;
}
```

This method is for deleting a given node while maintaining the BST properties. It begins by checking if the current node's key matches the key to be removed. If there's a match, the method addresses 3 cases: if the node has no children, it's simply removed returning nil; if it has one child, it is replaced with the minimum node from its right subtree, or if there is no right subtree, it is replaced by left; and if it has two children, it is replaced with the minimum node from its right subtree.

The algorithm uses recursion to traverse the tree and handle removal operations in the appropriate subtree based on comparisons with node keys.

4.2 Running Time

The running time for a balanced BST is $O(\log n)$, however if the BST is unbalanced, resembling a linked list, then the running time is $O(n)$. This is due to the algorithm recursively traversing down the appropriate subtree based on comparisons with the node's value.

5 Contribution

- Robert McDaniels: Created BST implementation, which includes the interface, classes, and methods
- Behnoud Alaghband: Added explanations of implementations explanations and running times, creator of report
- Dmitri Azih: Added explanations of implementations and running times, finalized report
- Dylan Rivas: Review and revision of implementations and report