Theory: Binary search tree in Java

33 minutes 0 / 5 problems solved

Before diving into the implementation of binary search trees in Java, let's quickly remind ourselves some basic definitions. A **binary search tree** is a node-based tree that fulfills the following conditions:

- for any node, the values of the nodes in its left subtree are less than the value of this node;
- for any node, the values of the nodes in its right subtree are greater than or equal to this node;
- any subtree of the binary search tree is also a binary search tree.

As simple as that: we have a treelike data structure, where smaller children go to the left and bigger go the the right, and the process continues from the root of the tree down to the leaves. On average, every basic operation on a binary search tree takes O(log n) time to complete, so this data structure allows us to find solutions rather quickly, which is surely a big advantage.

§1. The implementation in Java

Let's take a look at the data structure.

```
static class Node {
    int key;
    int value;
    Node left;
    Node right;
    Node parent;

    public Node(int key, int value, Node parent) {
        this.key = key;

        this.value = value;

        this.parent = parent;

    }

}
```

The '**key**' field contains the number of a node. This field is used during the construction process. The values we work with are located in the '**value**' field. Note that keys cannot recur while values can! Since we're working with a binary tree, having two pointers – one for the left child and one for the right – is enough. The '**parent**' field is also quite important: it is used in the 'delete' algorithm as you will see a bit later.

Now that we have looked at the structure, let's move on to the three basic operations on a binary search tree: node search, node insertion and node removal.

§2. Node search

To look for a specific node, we use a key. The result of the search can be either finding a node with that key or checking that it doesn't exist in the given tree. This is how this operation looks in the code:

Current topic:

Binary search tree in Java

Topic depends on:

✓ Binary search tree

✗ Trees in Java

Table of contents:

```
Node search(Node t, int key) {
    if (t == null || t.key == key)
        return t;
    if (key < t.key)
        return search(t.left, key);
    else
        return search(t.right, key);
}


public Node search(int key) {

    return search(root, key);

}
```

As you can see, we use two different search methods: private and public. If we want to conduct a search for a certain key, we need to know the root of the tree. For the user interface, this information is unnecessary, so the user passes the key to the public method, and then the internal private method finds the required node with the pointer to the root.

The algorithm is basically a recursive search for the node. It makes use of the main property of a binary search tree: all nodes in the left subtree are less than the parent, and all nodes in the right subtree are greater or equal. The results of the comparison (key < t.key) dictate in which subtree we should continue the search. The algorithm goes on until we either find the node with the given key or reach a leaf without having found the required node: (t == null || t.key == key).

The complexity of the algorithm is equal to the depth of the tree: O(log n) on average and O(n) in the worst case.

§3. Node insertion

```
Node insert(Node t, Node p, int key, int value) {
    if (t == null) {
        t = new Node(key, value, p);
    } else {
        if (key < t.key)
            t.left = insert(t.left, t, key, value);
        else
            t.right = insert(t.right, t, key, value);
    }

    return t;

}



public void insert(int key, int value) {

    root = insert(root, null, key, value);

}
```

Quite logically, the key of the node we're inserting is not contained within the tree. This means that during the recursive traversal by that key, we eventually get down to a node that should become a parent of the new one. All the while we keep a pointer to the parent (at the start it's null). This way, we always know the parent of our node. Same as with the previous algorithm, the complexity is O(log n) on average and O(n) in the worst case.

§4. Node removal

This algorithm may seem a bit more difficult compared to the previous two: depending on the position of the node we need to remove, there are a few different options. If you don't feel confident enough, review the previous topic about binary search trees.

Let’s first examine a supporting method that we will need for the removal operation. While removing the node, we don’t want any data lost, so we employ this method. Its purpose is to put a child into its parent node, hence it can only be used when the given node has only one child.

```
void replace(Node a, Node b) {
    if (a.parent == null)
        root = b;
    else if (a == a.parent.left)
        a.parent.left = b;
    else
        a.parent.right = b;
    if (b != null)
        b.parent = a.parent;

}
```

The method allows us to replace node a (the one we want to remove) with node b while keeping the correct structure. For that, we first need to check if node a is in the left or the right subtree; children are moved together with the new node.

Now that we have reviewed the basics and looked at the supporting method, we can finally move on to the removal algorithm. Take a look at the code:

```
void remove(Node t, int key) {
    if (t == null)
        return;
    if (key < t.key)
        remove(t.left, key);
    else if (key > t.key)
        remove(t.right, key);
    else if (t.left != null && t.right != null) {
        Node m = t.right;

        while (m.left != null)

            m = m.left;

        t.key = m.key;

        t.value = m.value;

        replace(m, m.right);

    } else if (t.left != null) {

        replace(t, t.left);

    } else if (t.right != null) {

        replace(t, t.right);

    } else {

        replace(t, null);

    }

}


public void remove(int key) {

    remove(root, key);

}
```

To remove the node, we should first reach it using recursion (this part is the same as with the search). When we find the node, there are several possible cases to consider:

1. The node we want to delete is a leaf. This is the easiest case: just replace the node with ‘null’.

2. The node we want to delete has either the left or the right child (t.left != null || t.right != null). In this case, simply replace the node with either its left or right child.
3. The node we want to delete has both left and right children (t.left != null && t.right != null). Here you should look for the leftmost node in the right subtree (the rightmost in the left is also an option), replace it with the node you need to delete and finally delete that node from the right subtree.

The complexity of the algorithm is also O(log n) on average and O(n) in the worst case.

Report a typo

9 users liked this theory. **3** didn't like it. **What about you?**

Start practicing

Comments (5) Hints (0) Useful links (0) Show discussion