MENU
▼

JOIN   **LOG IN**

COMPETE

DESIGN CHALLENGES

DEVELOPMENT CHALLENGES

DATA SCIENCE CHALLENGES

COMPETITIVE PROGRAMMING

LEARN

GETTING STARTED

DESIGN

DEVELOPMENT

DATA SCIENCE

COMPETITIVE PROGRAMMING

COMMUNITY

OVERVIEW

PROGRAMS

FORUMS

STATISTICS

EVENTS

BLOG

An Introduction to Binary Search and Red-Black Trees

By **cpphamza**– *TopCoder Member*

Discuss this article in the forums

As a programmer, you'll frequently come across tasks that deal with a number of objects — numbers, strings, people, and so forth — and that require you to store and process those objects. If you need to maintain a list of objects that are sorted and unique, and if you need to be able to quickly insert and retrieve objects to and from this list, the ideal data structure will be a tree set (or a tree map, if you consider each object a key and associate another object called a value to it).

Many programming languages provide built-in support for tree-based sets and maps — for instance, Java's TreeSet/TreeMap classes and the C++ Standard Template Library's set and map classes — but because of their common use, it's easy to misunderstand what actually happens behind the scenes, and how the underlying data structures actually work. That's what this article is all about.

We'll start off by looking at some of the general terms and concepts used in dealing with trees We'll then focus on binary search trees (BST), a special type of tree that keeps elements sorted (but doesn't guarantee efficient insertion and retrieval). Finally we'll look at red-black trees, a variation of binary search trees that overcome BST's limitations through a logarithmic bound on insertion and retrieval.

Trees terminology

A tree is a data structure that represents data in a hierarchical manner. It associates every object to a node in the tree and maintains the parent/child relationships between those nodes. Each tree must have exactly one node, called the **root**, from which all nodes of the tree extend (and which has no parent of its own). The other end of the tree – the last level down — contains the leaf nodes of the tree.
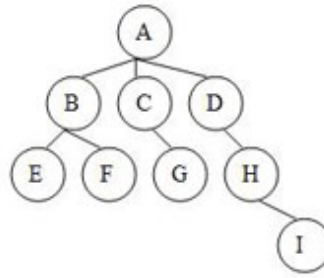
Figure 1. A tree

The number of lines you pass through when you travel from the root until you reach a particular node is the **depth** of that node in the tree (node G in the figure above has a depth of 2). The **height** of the tree is the maximum depth of any node in the tree (the tree in Figure 1 has a height of 3). The number of children emanating from a given node is referred to as its **degree** — for example, node A above has a degree of 3 and node H has a degree of 1.

Binary Search Tree (BST)

A binary search tree is a tree with one additional constraint — it keeps the elements in the tree in a particular order. Formally each node in the BST has two children (if any are missing we consider it a **nil node**), a left child and a right child. Nodes are rooted in place based on their values, with the smallest on the left and largest on the right.
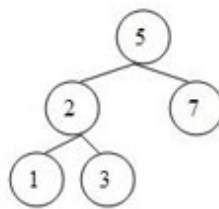

Figure 2. A binary search tree

Traversing BST

A common requirement when using sets and maps is to go through the elements in order. With binary search trees, traversing from left to right is known as **inordertree traversal**. In a tree where each node has a value and two pointers to the left and right children, inorder tree traversal can be thought of as:

```
Procedure Inorder_traversal(Node n)
    if(n == nil)
        return;
    Inorder_traversal(n.left_subtree);
    Print(n.value);
    Inorder_traversal(n.right_subtree);

…
Inorder_traversal(root);
```

Operations on BST (insertion, deletion and retrieval)

BST insertion:

When adding a new node to a binary search tree, note that the new node will always be a leaf in the tree. To insert a new node, all we will do is navigate the BST starting from the root. If the new node value is smaller than the current node value, we go left – if it is larger, we go right. When we reach a leaf node, the last step is to attach the new node as a child to this leaf node in a way that preserves the BST constraint.

For example, consider we want to add a new node with value 4 to the BST in Figure 1. Here are the steps we will go through:

> Let the current node = root = 5.

> As the new node is smaller than the current node (4 < 5), we will go left and set current node to 2.

> As the new node is greater than current node (4 > 2), we will go right and set the current node to 3.

> We've reached a leaf, so the last step is to attach the new node to the right of the current node. Here is how the new BST looks:
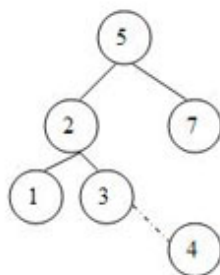

Figure 3. BST after adding node 4

BST Deletion:

Deleting a node from BST is a little more subtle. Formally there are three cases for deleting node n from a BST:

> If n has no children, we only have to remove n from the tree.

> If n has a single child, we remove n and connect its parent to its child.

> If n has two children, we need to :

Find the smallest node that is larger than n, call it m.

Remove m from the tree (if you have reached this case then m will always have no left child, though I'll leave the proof to the reader), so we apply one or the other of the above cases to do this.

Replace the value of n with m.

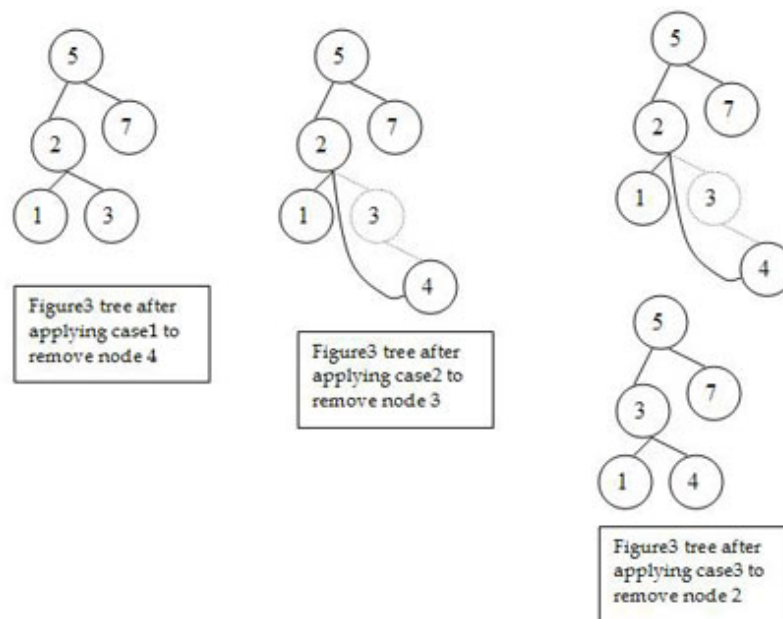Figure 4 shows these three cases in action.

Figure 4. BST deletion

BST Retrieval:

Retrieving an element from binary search trees requires simple navigation, starting from the root and going left, if the current node is larger than the node we are looking for, or going right otherwise.

Any of these primitive operations on BST run in O(h) time, where h is the tree height, so the smaller the tree height the better running time operations will achieve. The problem with BST is that, depending on the order of inserting elements in the tree, the tree shape can vary. In the worst cases (such as inserting elements in order) the tree will look like a linked list in which each node has only a right child. This yields O(n) for primitive operations on the BST, with n the number of nodes in the tree.

To solve this problem many variations of binary search trees exist. Of these variations, red-black trees provide a well-balanced BST that guarantees a logarithmic bound on primitive operations.

Red-black Trees

Red-black trees are an evolution of binary search trees that aim to keep the tree balanced without affecting the complexity of the primitive operations. This is done by coloring each node in the tree with either red or black and preserving a set of properties that guarantee that the deepest path in the tree is not longer than twice the shortest one.

A red-black tree is a binary search tree with the following properties:

1. Every node is colored with either red or black.

2. All leaf (nil) nodes are colored with black; if a node's child is missing then we will assume that it has a nil child in that place and this nil child is always colored black.

3. Both children of a red node must be black nodes.

4. Every path from a node n to a descendent leaf has the same number of black nodes (not counting node n). We call this number the **black height** of n, which is denoted by bh(n).
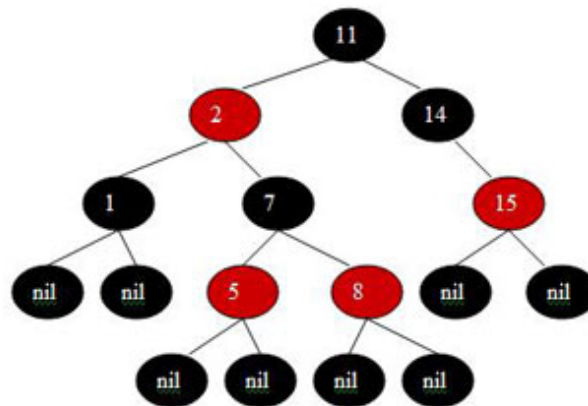
Figure 5 shows an example of a red-black tree.

Figure 5. an example of a red-black tree

Using these properties, we can show in two steps that a red-black tree which contains n nodes has a height of O(log n), thus all primitive operations on the tree will be of O(log n) since their order is a function of tree height.

1. First, notice that for a red-black tree with height h, bh(root) is at least h/2 by property 3 above (as each red node strictly requires black children).

2. The next step is to use the following lemma:

   Lemma: A subtree rooted at node v has at least $2^{bh(v)} - 1$ internal nodes

   Proof by induction: The basis is when h(v) = 0, which means that v is a leaf node and therefore bh(v) = 0 and the subtree rooted at node v has $2^{bh(v)}-1 = 2^0-1 = 1-1 = 0$ nodes.

Inductive hypothesis: if node v1 with height x has 2^bh(v1)-1 internal nodes then node v2 with height x+1 has 2^bh(v2)-1

For any non-leaf node v (height > 0) we can see that the black height of any of its two children is at least equal to bh(v)-1 — if the child is black, that is, otherwise it is equal to bh(v) . By applying the hypothesis above we conclude that each child has at least 2^[bh(v)-1]-1 internal nodes, accordingly node v has at least
2^[bh(v)-1]-1 + 2^[bh(v)-1]-1 + 1 = 2^bh(v)-1
internal nodes, which ends the proof.

By applying the lemma to the root node (with bh of at least h/2, as shown above) we get
n >= 2^(h/2) – 1
where n is the number of internal nodes of a red-black tree (the subtree rooted at the root). Playing with the equation a little bit we get h <= 2 log (n+1), which guarantees the logarithmic bound of red-black trees.

Rotations

How does inserting or deleting nodes affect a red-black tree? To ensure that its color scheme and properties don't get thrown off, red-black trees employ a key operation known as **rotation**. Rotation is a binary operation, between a parent node and one of its children, that swaps nodes and modifys their pointers while preserving the inorder traversal of the tree (so that elements are still sorted).

There are two types of rotations: left rotation and right rotation. Left rotation swaps the parent node with its right child, while right rotation swaps the parent node with its left child. Here are the steps involved in for left rotation (for right rotations just change "left" to "right" below):

Assume node x is the parent and node y is a non-leaf right child.

Let y be the parent and x be its left child.

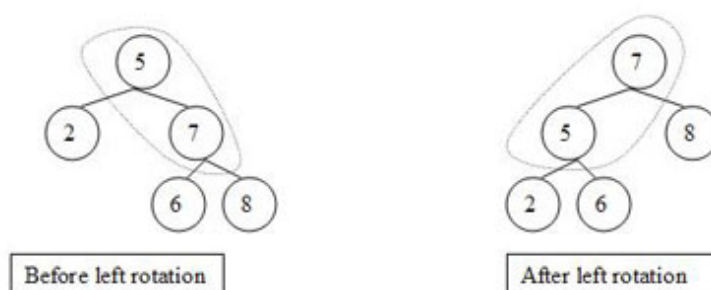Let y's left child be x's right child.

Figure 6. Left rotation

Operations on red-black tree (insertion, deletion and retrieval)

Red-black tree operations are a modified version of BST operations, with the modifications aiming to preserve the properties of red-black trees while keeping the operations complexity a function of tree height.

Red-black tree insertion:
Inserting a node in a red-black tree is a two step process:

1. A BST insertion, which takes O(log n) as shown before.

2. Fixing any violations to red-black tree properties that may occur after applying step 1. This step is O(log n) also, as we start by fixing the newly inserted node, continuing up along the path to the root node and fixing nodes along that path. Fixing a node is done in constant time and involves re-coloring some nodes and doing rotations.

Accordingly the total running time of the insertion process is O(log n). Figure 7 shows the red-black tree in figure 5 before and after insertion of a node with value 4. You can see how the swap operations modified the tree structure to keep it balanced.
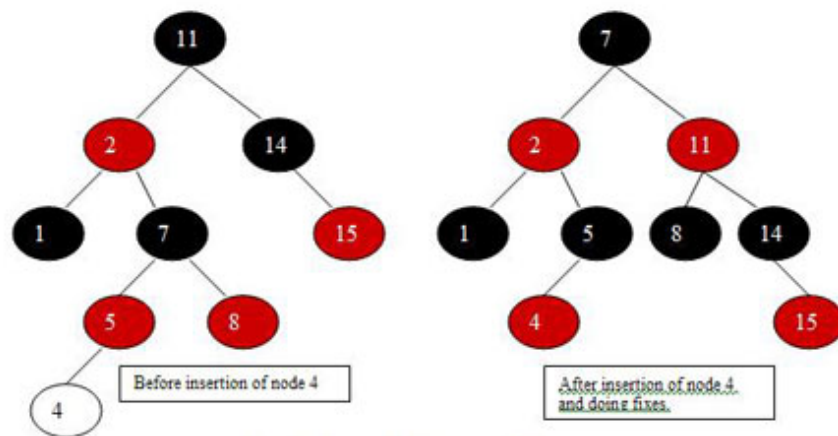

Figure 7. Insertion in red-black tree

Red-black tree deletion:
The same concept behind red-black tree insertions applies here. Removing a node from a red-black tree makes use of the BST deletion procedure and then restores the red-black tree properties in O(log n). The total running time for the deletion process takes O(log n) time, then, which meets the complexity requirements for the primitive operations.

Red-black tree retrieval:
Retrieving a node from a red-black tree doesn't require more than the use of the BST procedure, which takes O(log n) time.

Conclusion

Although you may never need to implement your own set or map classes, thanks to their common built-in support,

understanding how these data structures work should help you better assess the performance of your applications and give you more insight into what structure is right for a given task. For more practice with these concepts, check out these problems from the TopCoder archive that involve trees:

MonomorphicTyper (SRM 286)

PendingTasks (TCHS SRM 8)

RedBlack (SRM 155)

DirectoryTree (SRM 168)

EncodingTrees (SRM 261)

AntiChess (SRM 266)

IncompleteBST (SRM 319)

References

"Data Structures via C++: Objects by Evolution," by A. Michael Berman

"Fundamentals in Data Structures in C++, Second Edition," by Ellis Horowitz, Sartaj Sahni and Dinesh P. Mehta

"Introduction to Algorithms, Second Edition," by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein.

More Resources

Member Tutorials
Read more than 40 data science tutorials written by topcoder members.

Problem Set Analysis
Read editorials explaining the problem and solution for each Single Round Match (SRM).

Data Science Guide
New to topcoder's data science track? Read this guide for an overview on how to get started in the arena and how competitions work.

Help Center
Need specifics about the process or the rules? Everything you need to know about competing at topcoder can be

found in the Help Center.

Member Forums

Join your peers in our member forums and ask questions from the real experts - topcoder members!

OTHERS

SITEMAP

ABOUT US

CONTACT US

HELP CENTER

PRIVACY POLICY

TERMS

Topcoder is also on