Binary Search Trees (Part 1)

Robin Dawes

February 17, 2021

INTRODUCES one of the all-time classic problems: searching a set for a particular value. Binary trees are proposed as an effective structure to support algorithms to solve this and related problems.

.

Binary Search Trees (aka Lexically Ordered Binary Trees)

NOW WE TURN to the most popular application of binary trees ... one that is found throughout computing.

Suppose we have a collection S of values and we want to perform the SEARCH operation on S. This operation comes in two flavours:

- Given x, is x in S?
- Given x, what is the location of x in S?

As always in our study of data structures, our concern is choosing the best structure in which to store S to facilitate answering these questions.

Most often we are interested in the second question because we want to do something with x, such as access or modify information associated with x. If we are really only interested in the first question then there are structures that are particularly suited to answering that ... as we will see later.

Dramatic foreshadowing!

First let's try to establish the complexity classification of the search problem. To do so we will be a bit specific about the types of algorithm we will consider: we will focus on comparison-based algorithms - ie. algorithms that are based on comparing the target value x to elements of S.

It may seem unlikely that there can be a search algorithm that *does not* compare x to elements of S ... but we'll let that question slide for now.

Suppose we have a comparison-based search algorithm A that is guaranteed to find the correct answer for a target value x and a set S. We can think of the steps this algorithm follows as:

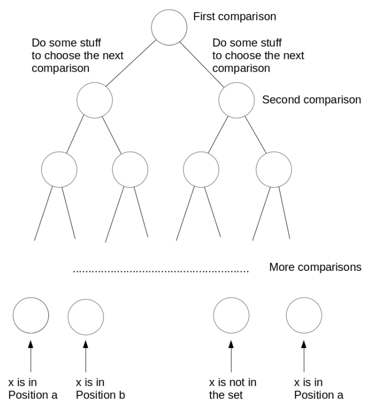
```
compare x to some element of S
    based on the comparison result ...
    compare x to some other element of S
        based on the comparison result ...
        compare x to some other element of S
            etc.
```

until we either find the value x or determine that it cannot be in S.

We can illustrate the set of **all possible** execution traces of A with an execution tree. This tree is specific to the algorithm A. To establish our lower bound on comparison-based searching we will have to build an argument that applies to the execution trees of all possible algorithms.

You may remember this approach from our study of the Ω classification of comparison-based sorting.

Note the outcomes at the bottom of the figure. Different sequences of comparisons can lead to the same outcome, which is demonstrated in the figure by the repeated presence of "x is in Position a" as an outcome. Similarly, the outcome "x is not in the set" may be the result of many possible sequences of comparisons that the algorithm might execute. The figure may give the impression that all the potential comparison sequences have the same length - this is not necessarily the case. If an early comparison actually finds x (or determines that x is not present) it is reasonable to allow the algorithm to stop at that point.



Each execution of the algorithm (for a specific target value x and set S) will follow a path down through this execution tree until it either finds x or determines that it is not there. Note that the only way to be absolutely sure that x is in S is to actually find an element of S that equals x. Thus the execution tree must contain at least as many "comparison nodes" as there are elements of S - if there is some element of S that is **never** compared to x in any execution of the algorithm, then we cannot always know for sure whether or not that element equals x. (We can think of an evil adversary who knows our algorithm and knows we are searching for x - the adversary arranges things so that x is placed in the element of S that our algorithm doesn't look at - so we never find x even though it is there.

If you don't like the idea of an evil adversary, just think of Murphy's Law: if we use an algorithm that never looks at some particular element of the set, then Murphy's Law says that sooner or later that's the element that we should have looked at.

So we know the execution tree for our algorithm A must contain at least n comparison nodes. But each execution of A will only visit some of those nodes - each execution represents a path down through the execution tree from the root to the point where the answer to the question is known. Our important question is "What is the longest sequence of comparisons A will ever need to complete a search?" If we can determine this as a function of n, this will tell us that for every value of n, there is some set of values that will require this many comparisons to get the right answer. This gives us a lower bound on the complexity of A. And since A is a completely unspecified comparison-based search algorithm, it will give us a lower bound on all comparison-based search algorithms.

Our question is equivalent to asking what the length of the longest path is from the root to the bottom of the execution tree. This will be different for each possible search algorithm A, but we can put a lower bound on it. We know the execution tree for A contains at least n comparison nodes. The top level of the tree has 1 node. The next level has no more than 2 nodes. The level below that has no more than 4 nodes (we say "no more than" because some of the branches may be early exits from A - for example, when x is found). Thus if there are k levels, there are no more than

$$1 + 2 + 4 + \cdots + 2^{k-1} = \sum_{i=0}^{k-1} 2^i = 2^k - 1$$

nodes in the tree.

This gives $n \leq 2^k - 1$ since we know the tree contains at least n nodes.

From this we get $k \geq \log_2(n+1)$ which gives $k > \log_2 n$.

So we conclude that the execution tree for **every** comparison-based search algorithm has at least $\log_2 n$ levels ... so no comparison-based search algorithm can have complexity less than $\mathrm{O}(\log n)$. In other words, comparison-based searching is in $\Omega(\log n)$.

So what does this have to do with the binary tree data structure?

We are already familiar with a structure that lets us search for x very efficiently - our old friend the simple one-dimensional array. If we store S in sorted order in an array, we can search S in $O(\log n)$ time using binary (or trinary, or k-ary) search.

End of story? We already know an algorithm whose O complexity matches the Ω classification of the problem. There's nothing left to be done, right?

Well, not quite. If our set S is fixed and unchanging, a sorted array is perfectly fine. But many applications involve sets that are modifiable - we need to add new values and delete existing values. In these situations a sorted array is not a good choice at all: inserting or deleting values in a sorted array takes $O(n)$ time. It's not much good having a fast search algorithm if our set-update algorithms are much much slower.

For contrast, consider storing the set in a linked list (unsorted). Now the complexity of adding a new value is in $O(1)$, but searching and deleting items are both in $O(n)$.

The question then becomes: is there a data structure that allows searching, adding and deleting to **all** be completed in $O(\log n)$ time?

The answer is yes, and of course since this discussion is lodged in the "Binary Tree" section of the notes you will have guessed that this is the structure. But to facilitate the search operation we need to be more precise about how the values in S will be stored in a binary tree.

It's actually going to take quite a bit of hard work before we can prove we can attain the $O(\log n)$ goal - buckle up and hang on!

When we store information in a generic binary tree there is no rule that says the information must be stored according to a specific pattern or rule. However in order to use a binary tree to address the "search" problem we enforce a simple rule for the placement of the values in the tree: small values go to the left and large values go to the right. We can formalize this as follows:

DEFINITION: A BINARY SEARCH TREE (BST) for a set S is a binary tree in which each vertex contains an element of the data set.

The values are arranged to satisfy the following additional property: at each vertex, all values in the left subtree are $\leq$ the value stored at the vertex, and all values stored in the right subtree are $>$ the value stored in the vertex. Note that we use "$\leq$" for the left subtree to accommodate the possibility of having duplicate values in the tree.

A BST is a more complex structure than either a one-dimensional array or a linked list. Why should we use it?

In order to make a case for using a BST as our structure of choice for Search/Insert/Delete situations we need to determine the complexity of algorithms for the Search, Insert and Delete operations, and then argue that they are superior to the algorithms for the same operations on an array or list.

BST_Search

Because of the ordering of the values in the vertices, searching a BST works just like binary search on a sorted array. We start at the root - if it contains the value we want, we are done. If not, we go to the left child or right child as appropriate.

Our design goal for implementing this data structure (and all subsequent ones) is that the user - in this case, the program which is calling the search function - should not need to know any details about the implementation of the structure. For example, the user should not need to know that the root of the tree is identified by an attribute called "root".

In these notes I'm using a typical object-oriented language syntax in which instances of classes possess methods which are accessed by appending the method name to the instance name. So if **`T`** is an instance of class **`Binary_Search_Tree`**, and all instances of this class own a method called Search, then we can call that function on **`T`** with **`T.Search(x)`** where x is the value we are searching for.

We need to decide which flavour of search we are going to implement ("if x is there, return True" versus "if x is there, return its location"). We will opt for the latter since it is neither easier nor more difficult with the BST structure. If x is in **`T`**, we return a pointer to the vertex containing it. If x is not in **`T`**, we return a nil pointer.

Here is a simple iterative version of the binary search tree algorithm as it might fit in a **`Binary_Search_Tree`** class.

```
Class Binary_Search_Tree():
#instance variable:
root : Binary_Tree_Vertex

def Search(x):
    current = this.root        # current is a Binary_Tree_Vertex pointer
    while current != nil:
        if current.value == x:
            return current
        elif current.value > x:
            current = current.left_child
        else:
            current = current.right_child
    return nil          #  x is not in the set
```

If we don't like multiple return points we can write the algorithm like this:

```
def Search(x):
    current = this.root
    while (current != nil) && (current.value != x):
        if current.value > x:
            current = current.left_child
        else:
            current = current.right_child
    return current
```

We can also implement the search algorithm recursively. We can use a "wrapper" function so that the interface does not change In this version, **`Search(x)`** and **`rec_Search(x)`** are both instance methods of the **`Binary_Search_Tree`** class.

Design principle: the user should not need to know whether our algorithm is iterative or recursive.

```
def Search(x):        # this method initiates the recursion
    return rec_Search(this.root,x)

def rec_Search(current,x):
    if current == nil:
        return nil
    elif current.value == x:
        return current
    elif current.value > x:
        return rec_Search(current.left_child,x)
    else:
        return rec_Search(current.right_child,x)
```

You should convince yourself that the iterative and recursive versions of the search algorithm do indeed achieve the same result. It is easy to see that they have the same complexity since they visit exactly the same sequence of vertices.

Which of the two is better? To my eye the recursive version is marginally more elegant, but that's debatable. The iterative version is probably a bit more efficient - this is because (according to conventional wisdom) a function call typically takes longer to execute than an iteration of a loop. This means that even though the two algorithms have the same complexity, the constant multiplier for the iterative version may be smaller than the constant multiplier for the recursive version.

Caveat: as I once discovered by experimenting in Python with recursive versus iterative implementations of Quicksort, it seems that recursive implementations of some algorithms may be faster than iterative implementations of the same algorithms. I encourage you to conduct some experiments to explore this question for yourself. Don't always trust conventional wisdom!

Another consideration is that it is often easier to prove correctness of recursive algorithms because we can use simple inductive proofs.

Regardless of the difference in speed, I prefer the recursive version. As we will see when we look at more sophisticated algorithms for BSTs, there are times when using recursion is much, much cleaner than using iteration.

Thinking about trees as recursive objects is a valuable exercise. Sometimes, even if the eventual goal is an iterative algorithm the best way to get there is to start by constructing a recursive algorithm, then convert the recursive calls into loops.

I noted above that the two versions of the Search algorithm for Binary Search Trees have the same complexity ... but what is it? We'll defer that question for a while, but at this point we can observe that on each iteration of the loop (or in each recursive call) we do a constant amount of work, and the number of iterations (recursive calls) is bounded above by the number of levels in the tree.

We can think of the Binary Search Tree Search algorithm - either the recursive or the iterative version - as a modification of one of the three traversal algorithms we explored earlier. Which one?

BST_Insert

LET'S TURN TO the problem of inserting new values in the set. When we insert a new value we need to put it in a position where we will be able to find it when we search for it. So we can start by comparing the new value to the root value. If it is $>$ than the root value, we need to put the new value in the right subtree ... because that is where BST_Search will look for it. Similarly, if it is $\leq$ the root value, it needs to go into the left subtree. And of course, capitalizing on the recursive structure of BSTs, we conduct exactly the same decision process at whichever of the two children we go to.

Wait a minute ... this sounds suspiciously like the Search algorithm. It is! The main work in the Insert algorithm is finding the proper place to add the new value, and that is almost exactly the same as Search. The only difference is that we continue the search until we find an empty place (ie. a nil pointer).

This means that if we find the value already present in the tree we continue the search (since we are allowing duplicates in our set) - thus we will inevitably reach a point where we "fall off" the tree. The point at which we fall off the tree is the unique proper location for the new leaf containing the new value.

One iterative version of the algorithm looks something like this. Note that we have to treat an empty tree as a special case because the root value will be "nil" so we cannot compare the new value to the root value. Also in this situation the new vertex (containing the new value) becomes the root, whereas in all other cases it is attached as a child of an existing vertex.

```
def Insert(x):
    if this.root == nil:
        this.root = new Binary_Tree_Vertex(x)
    else:
        current = this.root
        done = false # declaring a Boolean variable
        while not done:
            if current.value >= x: # x belongs on the left side
                if current.left_child == nil: # the new vertex needs to be the left child of current
                    current.left_child = new Binary_Tree_Vertex(x)
                    done = true
                else: # keep going down the tree
                    current = current.left_child
            else: # x belongs on the right side
                if current.right_child ==  nil: # the new vertex needs to be the right child of current
                    current.right_child = new Binary_Tree_Vertex(x)
                    done = true
                else: # keep going down the tree
                    current = current.right_child
```

Here's what is happening in this algorithm. As we work our way down the tree, we compare the new value x to the value in the current vertex and decide to go left or right. But we can't just jump down to the new level because if it happens to be a nil pointer then we have successfully found the insertion point, but by jumping down a level we have lost the link to the tree vertex which needs to be the parent of the new vertex. So we "test the water" by checking to see if the appropriate child of current is a nil pointer. If it is, then we create the new vertex and attach it to current. If the child is not a nil pointer then we move down to it in the normal way.

I wouldn't actually ever use this iterative method since the recursive method is so much cleaner. Behold!

```
def Insert(x):
    this.root = rec_Insert(this.root,x)

def rec_Insert(current,x):
    if current == nil:
        current = new Binary_Tree_Vertex(x)
    elif current.value >= x:
        current.left_child = rec_Insert(current.left_child,x)
    else:
        current.right_child = rec_Insert(current.right_child,x)
    return current
```

I remember feeling a sense of awe when I first saw this - it looks too simple to be correct! It makes beautiful use of the recursive structure of the tree to eliminate the need to treat the root as a special case, and it does away with the nested ifs. Furthermore it illustrates a very sound design principle for recursive algorithms that modify binary trees:

We were easily impressed in the Dark Ages.

*A method that potentially modifies a tree should return a pointer to the top of the modified tree, **even if it didn't change**.*

The beauty of this is that we can apply this principle at every vertex in the search path (that is, it applies to subtrees as well as to the whole tree). We call the recursive method on either the right or left child of the current vertex and simply attach the returned pointer to the modified subtree in place of whatever was there before.

For our insertion algorithm, in almost all cases this will be the same connection as was already there, but in the one crucial situation where we have found the insertion point an existing nil pointer gets replaced by the pointer to the new Vertex. Then we work our way back out of the recursion, re-attaching the vertices as we go.

See how this works at the root: we call the recursive method on the subtree that starts at the root (ie. the whole tree) and whatever we get back as the top of the modified tree is made the root. If the tree was empty, this will be a pointer to the new Vertex. If the tree was not empty, it will be a pointer to the previous (and unchanged) root Vertex. Either way, it is correct.

Next we will look at more complex algorithms for modifying trees, in which the subtrees may be very different after the changes have been made. At that point the power of saying "I know the recursive call will return a pointer to the top of the fixed subtree, so I can just attach it and return" will become more apparent.