Computer Science I
Binary Search Trees

CSCI-141
Lecture

11/20/2018

1 Problem

We want to be able to quickly find information when the quantity of information may grow over time. We also want to be able to print the information in order.

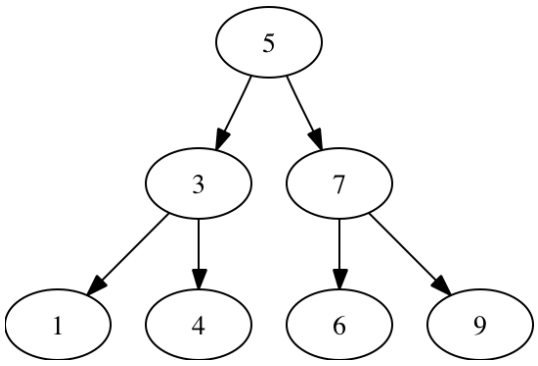
We learned how to *binary search* a sorted, Python list for an element in $O(\log(N))$ time. The problem with sorting a Python list and using *binary search* is that adding or removing an element requires us to shift many values in the list. A *binary search* works well only if the list of elements that we want to search does not change after the program starts.

1.1 Introduction to Binary Search Trees

If we wish to maintain a sorted order for a collection of elements that is updated dynamically (i.e. elements are added or removed at run-time), a *binary search tree* is often a better alternative. In the ideal case, search still takes $O(\log(N))$ operations in a binary search tree. Update operations may be faster than for a Python list[1].

Binary search trees are comprised of nodes. The nodes we'll consider have three attributes: a value, and two references to other nodes: one to a node with a smaller value, and the other to a node with a larger value. These references are often called the 'left' and 'right' child of a node. If the left or right child does not refer to another node, we will use `None` as the reference value. There are three types of nodes in a binary search tree: 1) the **root** (the 'top' node in the tree, with no parents), 2) **internal nodes** (with a parent and one or two children), and 3) **leaf** nodes, which have no children. The references between nodes are also known as *links* or *edges*.

Here is an example of a binary search tree that contains this sequence of numbers: $[1, 3, 4, 5, 6, 7, 9]$. Let's name the tree `tree1`.



1. The list representation used in binary search can be understood as a representation of a binary search tree that is less flexible but more compact.

1.2 Problem Tasks

If we want to use a binary search tree, our tasks are the following:

1. Create a Python representation of binary (search) trees.
2. Develop code, analyze worst-case run-time complexity, and devise test cases for the following:
   (a) A recursive algorithm that converts a binary search tree to a string that contains the values of the tree in sorted order.
   (b) A recursive algorithm that searches for an element in a binary search tree.

2 Analysis and Design

2.1 Definitions, Representation and Algorithms

2.1.1 Definitions and Terminology

We would like to characterize binary search trees beyond an individual example. If we observe that a subdivision of a tree is also a tree, we will see that trees are *recursive structures*. Also, a general definition involves components that are invisible: empty trees. We use `None` to represent an empty tree. Consequently, a binary tree has one of two possible choices...

Definition 1 *A* binary tree *is one of the following:*

- *An empty tree, or*
- *A non-empty tree, which has the following parts:*
  - *A data value;*
  - *a left sub-tree, which is a* binary tree*; and*
  - *a right sub-tree, which is a* binary tree*.*

Definition 2 *A* binary search tree*, bst, is a binary tree with the following property: If bst is a non-empty tree, then*

- *the left sub-tree of bst is a binary search tree, and all data values of the left sub-tree of bst are less than the data value of bst, and*
- *the right sub-tree of bst is a binary search tree, and all data values of the right sub-tree of bst are greater than the data value of bst.*

The domain of trees requires understanding the following tree terminology:

Definition 3 *A* node *is another name for a non-empty tree. (Informally, the emphasis is on the circle rather than the arrows.)*

Definition 4 *A* leaf *is a node whose left sub-tree and right sub-tree are both the empty tree. (In the picture, a leaf has no arrows coming out of it, since empty trees are invisible.)*

Definition 5 *An* internal node *is a node that is not a leaf. (In the picture, an internal node has at least one arrow coming out of it.)*

Definition 6 *Node n_1 is a* child *of node n_2 if and only if either n_1 is the left sub-tree of n_2 or n_1 is the right sub-tree of n_2.*

Definition 7 *Node n_1 is the* parent *of node n_2 if and only if n_2 is a child of n_1.*

Definition 8 *The* root *is the node that has no parent. (Since trees grow downwards in computer science, the root is the node at the top of the picture, and it has arrows going out but not coming in.)*

Definition 9 *Informally, a binary tree is* balanced *if the left and right sub-trees are both bushy, and roughly equally so. The tree in the picture is balanced.*

2.1.2 Python Binary Tree Representation

The binary tree structure defined above involves two fundamental concepts: structure defined by choices, and structure defined by parts. Using structures defined by choices involves almost no new Python mechanisms; we can pass different kinds of value through the same parameter, and we can return different kinds of values from functions.

The following Python code illustrates how to represent the binary tree. We use a structure to define our binary tree. Using `dataclasses`, we specify the self-references using the string name of the structure as the type for both the left and right fields, indicating that both the left and right references are themselves binary trees.

```
from dataclasses import dataclass
from typing import Union, Any

@dataclass
class BinaryTree:
    '''Class to represent a Binary Tree'''

    left: Union[None, 'BinaryTree']
    value: Any
    right: Union[None, 'BinaryTree']
```

The use of type `Any` for the value of the tree node provides a flexible representation

in which any type of data can be stored as the value. The `left` or `right` fields of a `BinaryTree` instance can be assigned the value `None`, which corresponds to the left or right sub-tree being an empty tree.

An initial, empty tree can be created with the statement:

```
tree = None
```

The following expression represents a tree that simply contains the value 1.

```
BinaryTree(None, 1, None)
```

The following expression represents a tree called `tree2`, which contains the values 1, 3, and 4.

```
tree2 = BinaryTree(BinaryTree(None, 1, None), \
                   3,                         \
                   BinaryTree(None, 4, None)
```

This structure is a convenient mechanism, but we need the ability to determine which of the possible choices we have.

How can a function decide what *type* of value it receives? An empty tree is represented by the constant `None`. A non-empty tree is an instance of the `BinaryTree`.

We can use Python's `isinstance` function to check the type of a value. Here's how:

```
>>> isinstance( tree2, BinaryTree )
True
>>> isinstance( [1,3,4], BinaryTree )
False
```

While the definition is a representation of a binary tree, does it also represent a binary search tree? Yes, it does, as long as we are careful only to write expressions that satisfy the properties of a binary search tree. For the rest of the lecture, we will be careful to write expressions that produce a binary search tree. Nevertheless, one might imagine that some programmers might forget. Indeed, while we can't enforce the binary search tree property in the data structure, we could write a function (which we might think of as an enhanced builder/constructor) that is guaranteed to return a binary search tree[2].

2.1.3 Algorithms and the Structural Recursive Design Pattern

How do we go about writing algorithms for structures defined both by choices and by parts? In particular, how do we write algorithms for binary trees? It turns out that a design pattern provides a scaffolding template for our code.

---

2. Even a binary search tree might not be all that we want. If we require only that the function returns a binary search tree, the tree returned might be quite unbalanced and look more like a linked list!

For structures defined by choices, we need to determine which choice we encounter. The Python version of the *structural recursive design pattern* tells us to use `if` statements and comparison with `None` to make that determination.

For structures defined by parts, we need to operate on the values of some or all of the parts. The *structural recursive design pattern* stipulates that a function should be called recursively on the recursive parts of the structure.

If we follow this design pattern, a function on binary trees will have the following form, a *code pattern*:

```
function bt_pattern( tree ):

    if tree is None :
        return ...

    else :
        return ... bt_pattern( tree.left )
                 ... tree.value
                   ... bt_pattern( tree.right ) ...
```

2.2 Solution: Converting Trees to Strings

2.2.1 From Examples to Test Cases (and Testing)

We would like to convert a binary search tree to a string representing the values in ascending order. Consider the following example:

```
>>> bst_to_string( tree2 )
'1 3 4 '
>>> bst_to_string( tree2 ) == '1 3 4 '
True
```

Without writing any definition, we have just showed what the code should do after it has been written. Using the function signature, we can identify examples that can become the test cases and test code for validating the solution. Below are descriptions of inputs for cases on which to test our function; these are organized around the input tree's structure.

What is the expected, correct result in each case?

1. Empty tree.
2. Tree with only one node.
3. Root and left child only.
4. Root and right child only.
5. Root with left and right child only (as in `tree2`).
6. A larger example such as `tree1`.

Given these cases, it is easy to write test functions even before writing the code[3].

3. This approach to development goes by the phrase of "Write the tests first" in the de-

2.2.2 Code

Let's start with the code pattern. We need to fill in the code for the empty tree case. What string should represent the empty tree?

To fill in the second case, let's assume the recursive calls work properly; this means that `bst_to_string( tree2.left )` produces `'1 '`, and `bst_to_string( tree2.right )` produces `'4 '`. What operations can we use to combine `'1 '`, `3`, and `'4 '` to get the result above?

Answers to these questions lead us to the following *pseudocode*:

```
function bst_to_string( tr ):
    """
        bst_to_string : BinarySearchTree -> String
        bst_to_string produces a string representing the
        in-order content of the binary search tree, tr.
    """
    if tr is an empty tree :
        return ''
    else :
        return bst_to_string( tr.left )  \
               +  str( tr.value ) + ' ' \
               +  bst_to_string( tr.right )
```

Since the input is a binary search tree, it requires all values less than the data value to be in the left sub-tree and all values greater than the data value to be in the right sub-tree. Therefore concatenating their string representations *in order* produces the desired result.

2.2.3 Substitution Tracing

Recall that substitution tracing involves writing a function call with actual arguments, an equal sign, and then substituting the calls with the expression returned by the function. Here we trace the functions defined above.

$$
\begin{aligned}
&\texttt{bst\_to\_string}(\texttt{BinaryTree}(\texttt{None}, 1, \texttt{None})) && = \\
&\texttt{bst\_to\_string}(\texttt{None}) + \texttt{'1 '} + \texttt{bst\_to\_string}(\texttt{None}) && = \\
&\qquad \texttt{''} + \texttt{'1 '} + \texttt{bst\_to\_string}(\texttt{None}) && = \\
&\qquad \texttt{''} + \texttt{'1 '} + \texttt{''} && = \\
&\qquad \texttt{'1 '}
\end{aligned}
$$

---

velopment process model originally known as "Extreme Programming". If you're interested, see http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap.

```
bst_to_string(BinaryTree(
              BinaryTree(None, 1, None),
              3,
              BinaryTree(None, 4, None)))                              =

              bst_to_string(BinaryTree(None, 1, None))
              + '3 '+
              + bst_to_string(BinaryTree(None, 4, None))               =

              bst_to_string(None) + '1 ' + bst_to_string(None)
              + '3 '
              + bst_to_string(None) + '4 ' + bst_to_string(None)       =

              '' + '1 ' + '' + '3 ' + '' + '4 ' + ''                   =
              '1 ' + '3 ' + '4 '                                       =
              '1 3 4 '
```

2.2.4 Complexity

What is the time complexity of this function? It visits every sub-tree exactly once. When visiting a sub-tree, if the node is not empty, it performs a string conversion and two string concatenations. String conversion takes essentially a constant amount of time. However, string concatenation typically takes time proportional to the length of the string, and the length of the string is proportional to the number of nodes in the sub-tree. Let N be the number of data values; N is proportional to the number of visits[4]. If the tree is balanced, the time complexity is $O(N \log(N))$. However, for unbalanced trees, at each visit there can be work that is proportional to the number of data values, and the worst case time complexity is therefore $O(N^2)$.

2.2.5 Observation: Making a Traversal

The `bst_to_string` function prints values in sorted order because the tree is in binary search tree order. This function performs what we call a *tree traversal* –specifically an *in-order traversal* in which we print the value at a node *after* we have printed the values of the left sub-tree, and *before* we print the values of the right sub-tree. The print action

4. N is the same as the number of nodes, but there are also empty trees to be counted. If a tree has some values, the number of empty trees will be twice the number of leaves. Since the number of leaves is less than or equal to the number of nodes, we can bound the sub-trees and thus the visits by a constant.

is known as the traversal's *visit* to the node. Printing is just one choice for action; in the general case a traversal may do arbitrary actions while visiting the node.

Other traversals include the *pre-order traversal*, where the visit occurs first, before traversing the sub-trees, and the *post-order traversal*, where the visit occurs last, after traversing the sub-trees.

2.2.6 Implementation

See `bst.py` for binary search tree `bst_to_string` and test functions.

2.3 Solution: Searching the Tree

2.3.1 From Examples to Test Cases (and Testing)

We would like to search a binary search tree to determine whether some value exists in the tree. Consider the following examples:

```
>>> bst_search( tree2, 1 )
True
>>> bst_search( tree2, 2 )
False
>>> bst_search( tree2, 3 )
True
>>> bst_search( tree2, 2 ) == False
True
```

Using the function signature, we can identify examples of use that can become the test cases and test code for validating the solution.

Below are inputs for cases on which to test our function; these are organized around the input tree's structure and the search value.

What is the expected, correct result in each case?

1. Empty tree and any value.
2. Tree with only one node and the value at that node.
3. Tree with only one node and a value not at that node.
4. Root and left child only, and a value in the tree.
5. Root and left child only, and a value not the tree.
6. Root and right child only, and a value in the tree.
7. Root and right child only, and a value not in the tree.
8. Root with left and right child only (as in `tree2`), and a value in the tree.
9. Root with left and right child only (as in `tree2`), and a value not in the tree.
10. A larger example such as `tree1`, and a value in the tree.
11. A larger example such as `tree1`, and a value not in the tree.

2.3.2 Code

Let's start again with the code pattern. We need to fill in the code for the empty tree case. What's the answer when searching in the empty tree?

To fill in the second case, let's assume the recursive calls work on the components. The recursive calls answer the question: "Is the value we're seeking in a sub-tree?" While we could combine the results of the recursive calls using the logical operator `or`, we can do better than that. We can avoid searching in a particular sub-tree if we know in advance that it can't possibly be there. We can rely on the definition of binary search trees, which guarantees that values smaller than the node value will be in the left sub-tree and values larger than the node value will be in the right sub-tree. How can we write this three way test?

The answers to these questions lead us to this *pseudocode*:

```
function bst_search( tr, value ):
    """
        bst_search: BinarySearchTree * Number -> Boolean
        bst_search returns whether or not
        value is present in the binary search tree, tr.
    """
    if tr is an empty tree :
        return False
    else :
        if value < tr.value:
            return bst_search( tr.left, value )
        elif value > tr.value:
            return bst_search( tr.right, value )
        else: # value == tr.value
            return True
```

2.3.3 Substitution Tracing

$$\begin{aligned} \texttt{bst\_search}(\texttt{tree2}, 1) &= \\ \texttt{bst\_search}(\texttt{BinaryTree}(\texttt{None}, 1, \texttt{None}), 1) &= \\ \texttt{True} & \end{aligned}$$

$$\begin{aligned} \texttt{bst\_search}(\texttt{tree2}, 2) &= \\ \texttt{bst\_search}(\texttt{BinaryTree}(\texttt{None}, 1, \texttt{None}), 2) &= \\ \texttt{bst\_search}(\texttt{None}, 2) &= \\ \texttt{False} & \end{aligned}$$

2.3.4 Complexity

What is the time complexity of this function? If the tree is balanced, about half of the tree is on the left and about half the tree is on the right. A constant time decision then eliminates half of the possibilities. Dividing something in half recursively leads to a running time of $O(\log(N))$.

However, we don't know whether or not the tree is balanced. For example, the left sub-tree *might be empty*! In that case, eliminating the left sub-tree does not eliminate any possibilities. In general, the worst case time complexity is the same as linear search, $O(N)$.

2.3.5 Implementation

For binary search tree search and test functions, see `bst.py`.