CmpSci 187: Programming with Data Structures
Spring 2015

Lecture #16, Binary Search Trees

John Ridgway

March 31, 2015

1 Trees

Why Binary Search Trees?

- We've seen that *binary search* is a powerful technique to search sorted arrays, finding an element in $O(\log n)$ time rather than $O(n)$ for linear search. But inserting or deleting into arrays is in general $O(n)$ due to shifting elements.
- Linked structures are good for inserting and deleting quickly, once you have a pointer to the place where it is happening, but they don't allow the *random access* to arbitrary elements that we need to do binary search.

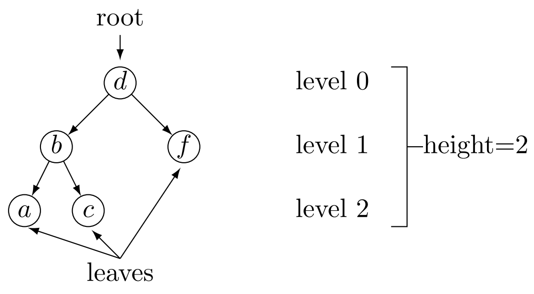
Binary Search on a Linked List?

- We could speed up our access to the middle elements of a linked list by adding more pointers, say from the beginning of the list to the middle, or from the middle to the three-quarters mark. Doing this right, we can add the abilities we need to get to the element we want in $O(\log n)$ time.
- The best way to arrange this sort of thing turns out to be a tree structure.

Tree Vocabulary

- We've seen *trees* in the directory structures of operating systems, and in the organization of a textbook or a company.
- A *binary tree* is one where each node may have a *left child* or a *right child* or both. There is one node called the *root*, and every node except for the root is the child of exactly one other node, its *parent*. A *leaf* is a node that has no children of its own.

- We use genealogical language to describe trees: *parent*, *ancestor*, *descendant*, sometimes even *uncle* or *niece'*.
- We say that the root node of a tree is at *level* 0, its children are at level 1, its grandchildren at level 2, and so forth.
- The highest level number for any node in the tree is the tree's *height*. The height can also be defined as the length of the longest path from the root to any leaf.



b is a left child of d, f is its right child
a, b, and c are all descendants of d

Clicker Question #1

Remember that a node in a binary tree may have a left child, a right child, both, or neither, and that the height of a binary tree is the length of the longest path from the root to a leaf. What are the possible sizes of a binary tree of height 3?

A. must be 15

B. anywhere 1 to 15

C. anywhere 8 to 15

D. anywhere 4 to 15

The BST Rule

- A *binary search tree* is a binary tree where every node is assigned an element called its contents. These elements must be comparable.
- The BST Rule is that for every node x, the element at x is greater than the elements in x's left child and all descendants of the left child, and is less than the elements in x's right child and all descendants of the right child.

The BST Rule

- Thus x's element splits the values less than or equal to x in the left subtree from the values greater than or equal to x in the right subtree.
- This allows us to find an arbitrary element by a form of binary search. We first look at the root node. If its value is equal to our target, we win.

Binary Search in a BST

- The more interesting case is when the target is not equal to the root's value. If the target's value is smaller, we recursively search the left child of the root, since we know that if the target is in the tree at all, it must be in the left subtree.
- Similarly, if the target is greater than the root's value, we recursively search the right subtree.
- If the target is in the tree, we will eventually find it by recursing in this manner. We always choose a subtree that must contain the target if the target is there at all.
- The number of times that we recurse is limited by the height of the tree.
- If we recurse to a node that is `null`, we can be confident that the target is not in the tree.

Clicker Question #2

Suppose that a binary search tree of `Integer`s has two nodes of value 7, called u and v. (We allow BST's to have nodes of equal value.) Suppose further that neither u nor v is an ancestor of the other. Which of these is possible?

A. No node other than u or v has value 7.

B. There are three nodes of value 7 and no more.

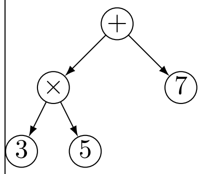
C. Nodes u and v are both children of a node with value 8.

D. Either u or v is the root of the tree.

Traversals

- Just as we had prefix, infix, and postfix strings to represent formulas, we have three ways to traverse the nodes of a binary tree in order: *pre-order*, *in-order*, and *post-order*.
- If our tree is the parse tree of a formula with binary operators and values at the leaves, each order gives the order of the characters in the corresponding string.

- Each traversal is defined recursively, so that we define how to *visit* each node.
- "Visiting" a node means processing it in some way, and our recursive definition of "traversing a node" x will give us a way to visit all the nodes in the subtree under x (x and all its descendants).
- A pre-order traversal of x first visits x, then traverses its left child, then traverses its right child.
- An in-order traversal traverses the left child, visits the node, and traverses the right child.
- A post-order traversal traverses the left child, traverses the right child, and finally visits the node.



In-order traversal:
$3 \times 5 + 7$

Pre-order traversal:
$+ \times 3\ 5\ 7$

Post-order traversal:
$3\ 5 \times 7\ +$

The BSTInterface

- The basic operations of a BST are exactly those of a list.
- The main difference is that we define methods to return three different kinds of iterator.

Code for BSTInterface

```
public interface
    BSTInterface<T extends Comparable<T>>
{
  boolean isEmpty();
  int size();
  boolean contains(T element);
  boolean remove(T element);
  T get(T element);
  void add(T element);
  Iterator<T> preOrderIterator();
  Iterator<T> inOrderIterator();
  Iterator<T> postOrderIterator();
}
```

An Application Example

- One of DJW's applications of lists was to read pairs of player names and golf scores, place them into a sorted list, and then print them sorted with smaller scores first.
- We can get the same results by placing the pairs into a BST instead of into a sorted list.
- The list version took $O(n)$ for most operations – will this do better?

Code for the Application

```
BSTInterface<Golfer> golfers =
  new BinarySearchTree<Golfer>();
System.out.println("Golfer name: ");
String name = conIn.nextLine();
while (!name.equals("")) {
  System.out.println("Score: ");
  int score = conIn.nextInt();
  String skip = conIn.nextLine();
  golfers.add(new Golfer(name, score));
  System.out.println("Name: ");
  name = conIn.nextLine(); }
System.out.println("The results are: ");
Iterator<Golfer> iterator =
  golfers.inOrderIterator();
while (iterator.hasNext()) {
  System.out.println(iterator.next()); }
```

Beginning the Implementation

- Like a list, a tree is a linked structure whose nodes are allocated dynamically. We begin by defining a class for the nodes, which looks a lot like the classes for singly or doubly linked lists.
- Note that any binary tree with elements at every node could use a similar type of node – this is a *search* tree node because `T` must be a class that implements the `Comparable<T>` interface.

Code for `BSTNode<T>`

(Getters and setters omitted.)

```
public class BSTNode<T> {
  private T data;
  private BSTNode<T> left;
  private BSTNode<T> right;

  public BSTNode(T data, BSTNode<T> left,
                 BSTNode<T> right)
  {
    this.data = data;
    this.left = left;
    this.right = right;
    }
  }
```

Code for `BinarySearchTree`

```
  public class
      BinarySearchTree<T extends Comparable<T>>
      implements BSTInterface<T>
  {
    private BSTNode<T> root;

    public boolean isEmpty() {
      return root == null;
    }
```

Computing the Size

- We could keep a count of the nodes in the tree and update it on any insert or delete. But it's useful to have the size of any *subtree* available, and we can calculate that (including the size of the root's subtree, which is the whole tree) with a recursive helper method.
- The size of a node's subtree is 1 plus the size of its left subtree plus the size of its right subtree.
- But computing this runs a risk of NPE's.

Clicker Question #3

Which of the bodies for `recSize(tree)` will make it return the size of any binary tree?

A. `return 1 + recSize(tree.getLeft()) + recSize(tree.getRight( ));`

B. `if (tree == null) return 0; if (tree.getLeft() != null) return 1+recSize(tree.getLeft()); if (tree.getRight() != null) return 1+recSize(tree.getRight());`

C. `if (tree == null) return 0; return 1 + recSize(tree.getLeft()) + recSize(tree.getRight());`

D. none of these

Recursive Code for Size

If our helper method is prepared to encounter an empty subtree, this approach works fine and gives us simple, correct code.

```
    public int size() {
      return subtreeSize(root);
    }

    private int subtreeSize(BSTNode<T> node) {
```

```
    if (node == null) {
      return 0;
    } else {
      return 1 + subtreeSize(node.getLeft())
               + subtreeSize(node.getRight());
    }
  }
```

Finding Size Iteratively

- We can visit all the nodes of the tree by what amounts to a depth-first search with a stack. Since each node will go onto the stack exactly once, we just keep a counter and increment it for each node pushed.
- This method is more complicated than the recursive one and not faster, but it illustrates a technique that might come in handy in other circumstances.

Iterative Code for Size

```
public int size() {
  int count = 0;
  if (root != null) {
    Stack<BSTNode<T>> hold =
      new LinkedStack<BSTNode<T>>();
    BSTNode<T> currNode;
    hold.push (root);
    while (!hold.isEmpty()) {
      currNode = hold.pop(); count += 1;
      if (currNode.getLeft() != null)
        hold.push(currNode.getLeft());
      if (currNode.getRight() != null)
        hold.push(currNode.getRight()); } }
  return count; }
```

A Recursive `get` Method

- We know how to search a BST – if the element we seek is at the current node we have found it, and if not we know that we have to search either the left or the right subtree.
- The job that we want to make recursive is "find this element in the subtree under this node". The base cases are a `null` node (return `null`) and a node with the given element (return the element).

Recursive `get`

The `contains` method is very similar.

```
public T get(T t) {
  return getFromTree(t, root);
}
```

```
private T getFromTree(T t, BSTNode<T> node) {
  if (node == null) return null;
  if (t.compareTo(node.getData()) < 0)
    return getFromTree(t, node.getLeft());
  if (t.compareTo(node.getData()) == 0)
    return node.getData();
  return getFromTree(t, node.getRight());
}
```

Clicker Question #4

Consider the recursive `get` method that is analogous to the recursive `contains` method. Suppose there are multiple entries in the tree with the same value. Which of these statements is true?

A. `get` might return any of the entries

B. `get` must return the leftmost of the entries

C. `get` cannot return the value of a leaf node

D. None of the above