Lecture 13: Binary (Search) Trees I

built on 2019/06/04 at 13:24:01

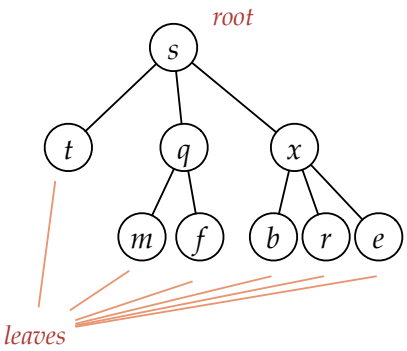
1 Trees and Binary Trees

A *tree* is a hierarchical structure with a few notable properties. Each item in a tree is generically called a *node*. There's a special top element known as the *root*. Every node has zero or more *children*. When a node doesn't have any children, it is called a *leaf*. In Computer Science, we often draw trees with its root node at the top and their descendants in subsequent layers below that.

More formally, a tree T is a set of nodes N where elements are kept and a parent-child relationship given by a function $p\colon N \to N \cup \{\bot\}$ satisfying the following properties:

- If the tree is nonempty, there is a *unique* node $r \in N$, called the root, such that $p(r) = \bot$, that is, it has no parent.
- Every node other than r has a parent in N, that is, $p(v) \neq \bot$ for all $v \neq r$. The children of a node w are those v such that $p(v) = w$.
- The parent chain of every vertex other than the root itself contains the root r. In other words, for every $v \in N$, $v \neq r$, $p(p(\cdots p(v))) = r$.

As an example, consider the following tree diagram and its corresponding formal description:



In this particular example, the set of nodes N is $\{s, t, q, x, m, f, b, r, e\}$, and the parent-child relationship p can be expressed as follows:

Node	*Parent*
s	$p(s) = \bot$
t	$p(t) = s$
q	$p(q) = s$
x	$p(x) = s$
m	$p(m) = q$

Node	*Parent*
f	$p(s) = q$
b	$p(t) = x$
r	$p(q) = x$
e	$p(x) = x$

Trees are used both to "physically" represent data and to conceptually express ideas (help shape our thought, though a real tree is perhaps never stored anywhere). In the former case, we need to be able to materialize them—we need to represent them in our program. *How to represent a tree structure in a computer program?*

With this definition in mind, we could represent a tree by storing this p function, for example, as a dictionary. As a forward reference to what we're discussing next, this allows for efficiently looking up parents but it would be hard to determine the children of a particular node.

In code, we could store the p function as a `Map`, say a `HashMap` (an equivalent of `dict` in Python). The goal here would be if we have `Map<Integer, Integer> p` that represents p, then `p.get(u)` should return the identity of the parent of u.

2 General Tree Representation

By keeping the parent information, we can walk from a node to the root easily. This is because for every node, we can quickly find out who the parent for this node is. However, answering the following question is inefficient: *given a node, who are the children?* The basic problem is, we don't have readily accessible data in that direction. Hence, to fix this, we'll simply keep data in the direction of children.

But how do you keep the children of a node? Keep in mind that the tree might change: there will be new nodes, and there will be some nodes that will be deleted. Inspired by the linked list discussion, we will make an object for each node, like so:

```
class TreeNode {
  // you store data here, but in addition to that, we'll keep the children
  List<TreeNode> children;
}
```

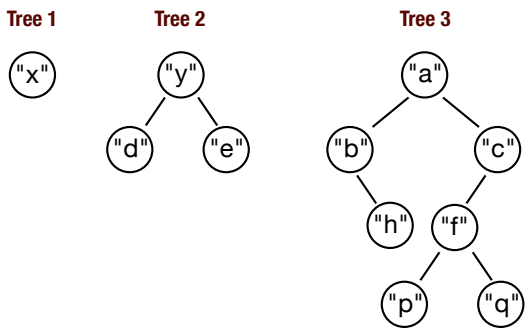
Yet when it comes to data organization, we often deal with trees that have a bounded number of children. One particular flavor that will show up time and again is a family of trees that have no more than 2 children. These are known as binary trees. A binary tree is a tree as just described but with one extra requirement: every node can have at most 2 children.

3 Binary Tree Representation

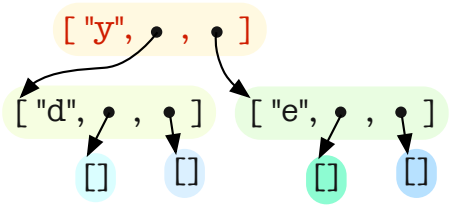
Let's turn our focus to binary trees as they are what we'll encounter most of the time in this class.

There are many ways to encode a tree structure in a program you write. Indeed, modern languages such as Python, Java, C/C++, etc. provide various mechanisms for expressing them. We have seen one such idea during our discussion about binary heaps. In that case, we are storing a special kind of trees, so we could exploit a specific encoding—allowing us to collapse a tree structure into a flat list. In general, our intuition suggests that a tree structure probably isn't as flat as a list. Here, we'll look at a general representation: a representation using a node class and reference.

As our running example, we'll consider the following trees:



Using nodes and references: To motivate this representation, let's consider the second tree (above). We would like for each node to store its data, together with its two children, as in the following diagram:



In particular, a node has "arrows," denoting references, pointing to its children nodes.

Hence, we can define a class that represents each node individually. We will then link them up to form a tree. Specifically, our class will store information about this very node itself—for instance, the key and value associated with this node. It will store what the left subtree and the right subtree are (as references). For example, we can start with the following snippet:

```
class TreeNode<E> {
  E key;
  TreeNode<E> left;
  TreeNode<E> right;
```

```
  TreeNode(E key_, TreeNode<E> left_, TreeNode<E> right_) {
    key=key_; left=left_; right=right_;
  }

  TreeNode(E key_) {
    key=key_; left=null; right=null;
  }
}
```

In this representation, we can therefore represent the trees in our examples as follows:

```
TreeNode<String> tree1 = new TreeNode<>("x");
TreeNode<String> tree2 = new TreeNode<>("y", new TreeNode<>("d"),
                                             new TreeNode<>("e"));
TreeNode<String> tree3 = new TreeNode<>("a",
          new TreeNode<>("b", null, new TreeNode<>("h")),
          new TreeNode<>("c",
                new TreeNode<>("f", new TreeNode<>("p"), new TreeNode<>("q")),
                null));
```

Exercise: Write a function that takes in this representation and returns the depth of the tree (i.e., the length of the deepest path in the tree)?

Example Solution:

```
static <E> int height(TreeNode<E> u) {
  if (u==null) return 0;
  else return 1 + Math.max(height(u.left), height(u.right));
}
```

Now that we know how to represent a binary tree in your code, we'll put that knowledge to use. We'll see two important ideas in tree algorithms: (1) the ability to walk the tree in different order, and (2) data organization strategies that will make navigating trees easier (next time).

4 Tree Traversal

We'll study a systematic way of accessing nodes in a tree. Given a tree T, a traversal of T prescribes an ordering in which nodes of T are visited. When a node is visited, the specific action taken at that node depends on the particular application. This could be a simple print statement, incrementing a counter, or some complex computation.

For binary trees, there are *three* common traversal patterns that we'll consider in turn: (1) preorder traversal, (2) inorder traversal, and (3) postorder traversal. All these patterns are best described recursively.

Preorder Traversal: First, visit the node itself—then visit both children recursively and return. That is,

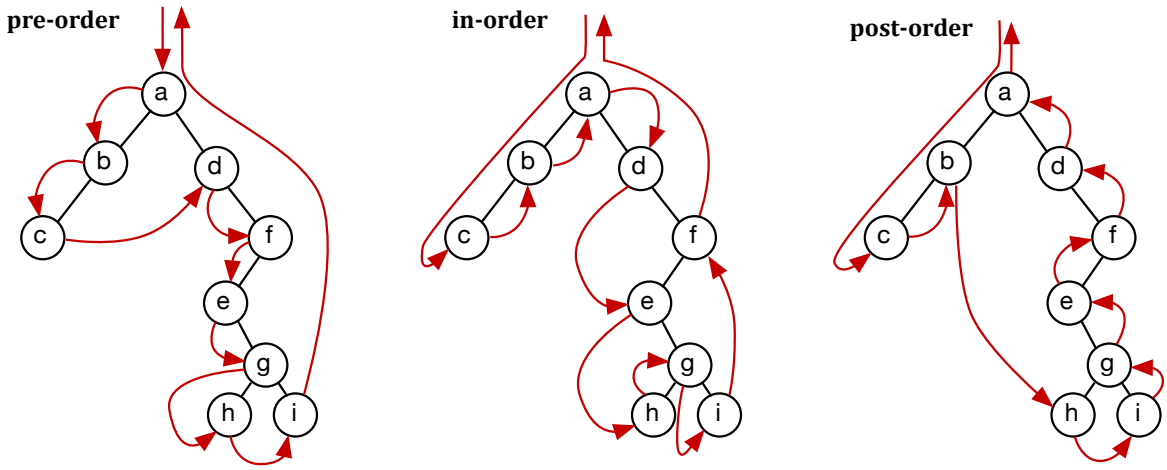
```
pre(node):
   visit(node)
   pre(node.left)
   pre(node.right)
```

Inorder Traversal: First, visit the left child—then visit the node itself, and visit the right child before returning. That is,

```
inorder(node):
   inorder(node.left)
   visit(node)
   inorder(node.right)
```

Postorder Traversal: First, visit both children recursively—then visit the node itself and return. That is,

```
post(node):
   post(node.left)
   post(node.right)
   visit(node)
```



Exercise: Can you come up with a systemtic way to traverse a tree using these patterns?

4.1 Why Should I Care?

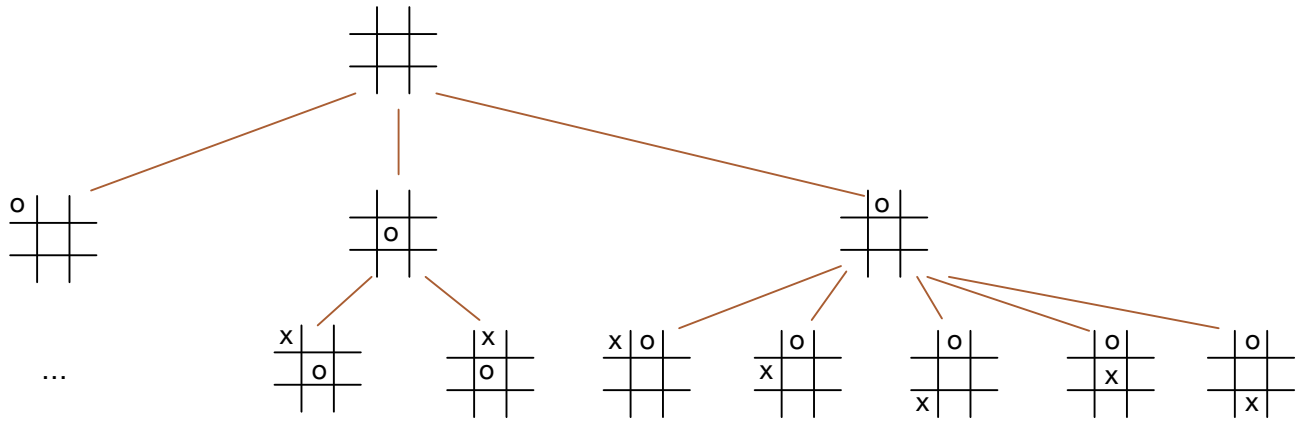
We have seen numerous applications of the for-loop for lists. Think of this as a for-loop for trees, but since trees aren't flat; there are many patterns to use.

- Example 1: An arithmetic expression can be represented as a tree. Then, we need to traverse that tree to evaluate it or to manipulate it in some way.
- Example 2: The table of contents of a book can be seen as a tree. Printing this out is basically tree traversal.
- Example 3: In an all too real example, an html page has of a DOM (document-object model) tree. If you want a program to go over these objects (to update color or other property), that's tree traversal.

5 Breadth-First Traversal

Another common tree traversal pattern is to visit all positions at depth d prior to visiting the positions at depth $d + 1$. This is known as a *breadth-first traversal*.

One common use of this pattern is game playing using what is known as a game tree. *What is a game tree?* It is a tree that represents all possible choices of moves that the players could make during a game. For example, the tree below is a partial game tree for Tic-Tac-Toe that starts from the empty 3×3 board.



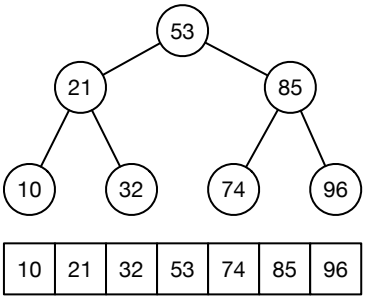
The reason for searching in this kind of tree is so that we know what move we should make in order to minimize our chance of getting into a losing configuration.

Food for thought: How would you implement such a traversal in Java? *(Hint: Use a queue)*

6 Advertisement: Binary Search Tree, Not Just Any Binary Tree

Let's first gather a bit of intuition for where we are going. Consider the following sorted array and a tree that we superimpose on:

This figure shows a perfect binary tree on 7 nodes. These nodes coincidentally (or perhaps not) are labeled according to numbers from the sorted array below the tree. In many ways, this should remind us of binary search. When we look up a key using binary search, the walk from root to that key in the tree is exactly the comparisons we make in binary search.



10	21	32	53	74	85	96