



## Topic Notes: Trees

We've spent a lot of time looking at a variety of *linear* structures. There was a natural linear ordering of the elements in arrays, vectors, linked lists. We then put some restrictions on those structures, looking at stacks and queues and ordered linear structures.

Just like we can write programs that can branch into a number of directions, we can design structures that have branches.

Today, we'll start looking at our first more complicated structure: *trees*.

In a linear structure, every element has unique successor.

In trees, an element may have many successors.

We usually draw trees upside-down in computer science.

You won't see trees in nature that grow with their roots at the top (but you can see some at Mass MoCA).

---

## Examples of Trees

---

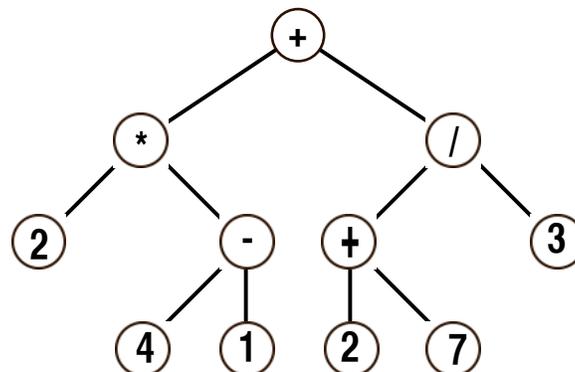
### Expression trees

One example of a tree is an *expression tree*:

The expression

$$(2 * (4 - 1)) + ((2 + 7) / 3)$$

can be represented as

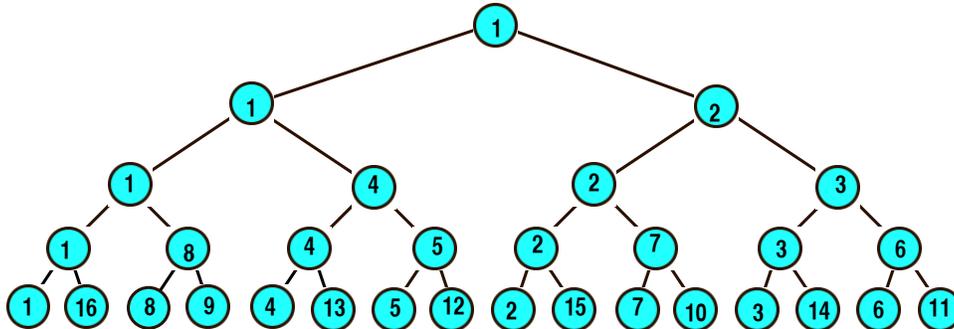


Once we have an expression tree, how can we evaluate it?

We evaluate left subtree, then evaluate right subtree, then perform the operation at root. The evaluation of subtrees is recursive.

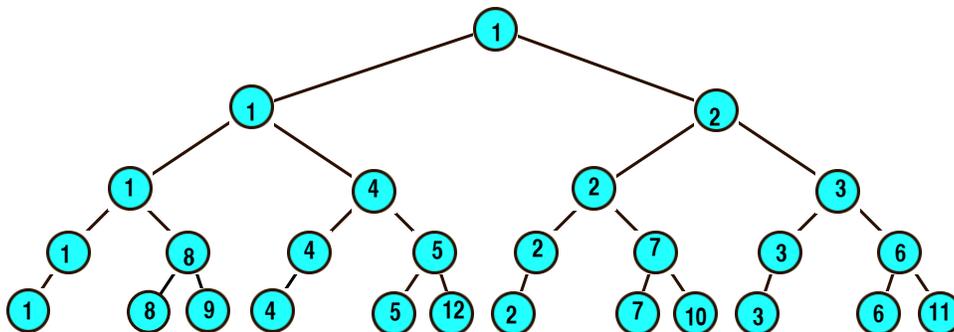
## Tournament Brackets

Another example is a tree representing a tournament bracket:



(a *complete* and *full* tree)

or



(neither complete nor full)

## Tree of Descendants

The text looks at an example of a pedigree chart – looking at a person’s ancestors. Instead, we can look at a person’s descendants. (Example drawn in class).

## Definitions and Terminology

There are a lot of terms we will likely encounter when dealing with tree structures:

A *tree* is either empty or consists of a *node*, called the *root node*, together with a collection of (disjoint) trees, called its *subtrees*.

- An *edge* connects a node to its subtrees
- The roots of the subtrees of a node are said to be the *children* of the node.
- There may be many nodes without any successors: These are called *leaves* or *leaf nodes*. The others are called *interior nodes*.
- All nodes except root have unique predecessor, or *parent*.
- A collection of trees is called a *forest*.

Other terms are borrowed from the family tree analogy:

- sibling, ancestor, descendant

Some other terms we'll use:

- A *simple path* is series of distinct nodes such that there is an edge between each pair of successive nodes.
- The *path length* is the number of edges traversed in a path (equal to the number of nodes on the path - 1)
- The *height of a node* is length of the longest path from that node to a leaf.
- The *height of the tree* is the height of its root node.
- The *depth of a node* is the length of the path from the root to that node.
- The *degree of a node* is number of its direct descendents.
- The idea of the *level* of a node defined recursively:
  - The root is at level 0.
  - The level of any other node is one greater than the level of its parent.

Equivalently, the level of a node is the length of a path from the root to that node.

We will limit our initial discussion to *binary trees* – trees whose nodes are all have degree  $\leq 2$ .

We will also orient the trees: each subtree of a node is defined as being either the *left* or *right*.

---

## Binary Tree Interface

There are many possible specifications of binary trees that allow reasonable insertion and deletion of elements.

We will consider the one provided in the structure package first, and think about other possibilities later.

Unlike what we have seen to this point, the structure package does not define an interface for binary trees and then use implement that in one or more concrete classes. Structure simply implements class `BinaryTree`, so we will look right at that.

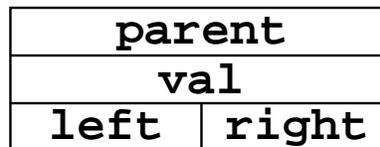
Unlike the linked list implementations, where we do not give the users of the structures access to the actual list nodes, the binary tree exposes more of its structure to users. The actual recursive data structure is given directly to users. The implementation needs to ensure that changes that might be made by users cannot render a tree invalid.

The tree is constructed of instances of class `BinaryTree`. Each `BinaryTree` object has the fields it needs to store its value and the parent and child references.

**See Structure Source:**

`/home/cs501/src/structure5/BinaryTree.java`

A node has 4 fields, which we might draw as follows:



Plenty of things to notice and think about here:

- We have three constructors.
  1. The first is used to create an “empty” `BinaryTree`. We will see this constructor used by the other constructors to create these empty trees in place of having `null` references to represent empty subtrees. This allows most methods to be called on these, eliminating lots of special cases. We could also use `null` to represent empty trees, but this would mean some extra code in several methods. Note that only empty tree nodes may contain a `null` value. Regular tree nodes must contain non-`null` values.
  2. The second constructor creates a tree node with no children (a leaf node) containing a particular value.
  3. The third constructor creates a tree node that may have children. Note that if a user of this constructor specifies a `null` child, it is replaced with an empty tree instance.
- We have accessors to retrieve the children or parent of a node.
- Note that the value and subtree links can be set by the user, but the parent reference is set only in a `protected` method. This is done to make sure we don't put the tree into a (bad) state where a parent points to a child but the child doesn't point back to the parent.

- We have a variety of other (self-explanatory) methods to retrieve information about a tree: `size`, `root`, `height`, `depth`, `isFull`, `isEmpty`, `isComplete`. Note the recursive nature of many of these methods.
- 

## Binary Tree Example

We can construct a simple binary tree to represent and evaluate an arithmetic expression using the `BinaryTree` implementation:

```
((4+3)*(10-5))/2
```

### See Example:

```
/home/cs501/examples/BinaryExpressionTree
```

There are two versions of this program:

1. `BinaryExpressionTree.java` stores the operators and values to be used as `Strings`.
  - This lets us use a `BinaryTree<String>`.
  - Since some are operators and some are numbers, we need to check and treat as appropriate, based on the contents of the `String`.
2. `BinaryExpressionTreeObject.java` stores the operators as `Characters` and the numbers as `Integers`.
  - Here we instead use a `BinaryTree<Object>`, since that's the type that can represent both a `Character` and an `Integer`.
  - We check the actual type of the value retrieved with the `instanceof` operator and use the value as appropriate

Another option would be to define a common type along the lines of the `Tokens` from the postscript lab.

In both cases, note the `treeString` method that prints our binary tree in a nice format.

---

## Tree Traversals

Iterating over all values in our linear structures is usually fairly easy. Moreover, one or two orderings of the elements are the obvious choices for our iterations. Some structures, like an array or a `Vector`, allow us to traverse from the start to the end or from the end back to the start very easily. A `SinglyLinkedList`, however, is most efficiently traversed only from the start to the end.

For trees, there is no single obvious ordering. Do we visit the root first, then go down through the subtrees to the leaves? Do we visit one or both subtrees before visiting the root?

We will consider 4 standard *tree traversals* for our binary trees:

1. *preorder*: visit the root, then visit the left subtree, then visit the right subtree.
2. *in-order* visit the left subtree, then visit the root, then visit the right subtree.
3. *postorder*: visit the left subtree, then visit the right subtree, then visit the root.
4. *level-order*: visit the node at level 0 (the root), then visit all nodes at level 1, then all nodes at level 2, etc.

For example, consider the preorder, in-order, and postorder traversals of the expression tree we looked at in the example code:

- preorder leads to prefix notation:  
`/ * + 4 3 - 10 5 2`
- in-order leads to infix notation:  
`4 + 3 * 10 - 5 / 2`
- postorder leads to postfix notation:  
`4 3 + 10 5 - * 2 /`

The iterator concept fits nicely with tree traversals, but since the code for the iterators in the text is somewhat complex, so we will first consider traversals without iterators.

In our first traversal examples, we will build a small binary tree of `Integer` values and call methods that perform the traversal. Here, “visiting” a tree node involves passing its value to the method `process`.

### See Example:

`/home/cs501/examples/BTTraversals`

First, note the construction of the tree. We build the tree from bottom up, but do not store the subtrees in local variables during construction – we simply construct them in the parameters of the constructor for the next level up.

Now, consider each of the traversal implementations. The in-order, preorder, and postorder traversals work exactly as we would expect. Each is recursive, and we visit the subtrees and the root node as defined for each ordering.

The level-order traversal is a bit trickier. We need to visit the root of each subtree before doing anything in the next level. This calls for a queue!

For all of the others, we used a stack, just without thinking about it. We took advantage of the call stack to support the recursion!

---

## Tree Iterators

The `structure` package has implementations of iterators for each of these four traversals. Whereas in the `do{pre,post,in}order` methods above, we were able to take advantage of the computer’s run-time stack, we need to have a stack explicitly declared and used in the iterator implementations.

The complexity of the iterators varies with the type of traversal.

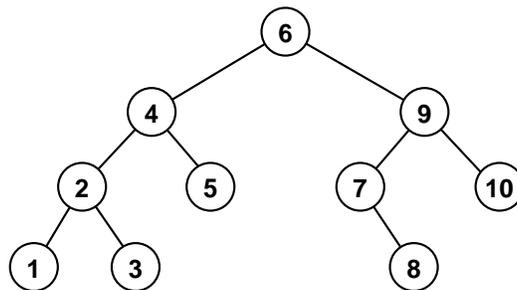
We need to make sure we get things onto the stack (or queue, in the case of the level order) in the right order.

At any time, we want the stack/queue to contain the tree nodes that still need to be visited.

Visiting a non-leaf node will result in additional nodes being added to the stack/queue.

In all cases, iteration can continue as long as something remains on the stack/queue.

We will consider this example tree:



In each case, recall that we need to satisfy the iterator interface (actually, the `AbstractIterator` in structure) by providing:

1. a constructor
2. a `reset` method
3. a `hasNext` method
4. a `next` method
5. a `get` method

We look at each in turn.

### See Structure Source:

```
/home/cs501/src/structure5/BTPreorderIterator.java
```

Our preorder traversal visits the root first, followed by the left subtree, then the right subtree.

Recall that we implicitly used the run-time stack for the non-iterator traversal code. Here, we manage the stack.

At any point, we want the tree node on the top of the stack to be the next tree node that needs to be visited.

We want to visit the root, then the left subtree, then the right subtree.

So to start or reset our iteration, we initialize the stack with the root node.

A `next` operation just involves popping, then processing the node on top of the stack, then pushing its right and left subtrees (in that order, since we want to process the left first).

Finally, we can implement `hasNext` by just checking if the stack is empty, which will tell us that the traversal has been completed.

**See Structure Source:**

```
/home/cs501/src/structure5/BTInorderIterator.java
```

For this traversal, we need to visit the left subtree, then the root, then the right subtree.

Here, the first thing we want to visit is the deepest, leftmost child. So we need to initialize the state of our iterator so that that node (the deepest, leftmost child) is on top of the stack. To do this, we push the root, and all of the left subtrees until we come to a node which doesn't have a left subtree.

A `next` operation here involves popping the top value off the stack to be returned, then dealing with its right subtree. The first thing there that needs to happen is again its leftmost branch, so we need to push the right subtree then all of its left children.

**See Structure Source:**

```
/home/cs501/src/structure5/BTPostorderIterator.java
```

Here, we visit the left subtree, then the right subtree, and finally the root.

This is the tricky one. First, if there is a left subtree, we need to push down through those left subtrees as far as we can. If any node has no left subtree but has a right subtree, push that instead. Continue to a leaf.

A `next` operation involves popping the top value to be returned. If the thing we just popped is a left child, push the sibling and its left children (or right when there is no left) until we get to a leaf again.

**See Structure Source:**

```
/home/cs501/src/structure5/BTLevelorderIterator.java
```

Here, we visit the tree level by level.

This one is actually quite easy. We have a queue instead of a stack.

We start by enqueueing the root, as this is the first thing we want to visit.

When we visit a node, we enqueue its children.

The `BTTraversals` example also demonstrates the use of these iterators.

---

## Example Application: Huffman Compression

We will now briefly consider a *greedy algorithm* concerned with the generation of encodings.

The problem of *coding* is assignment of bit strings to alphabet characters. *Codewords* are the bit strings assigned for characters of alphabet.

We can categorize codes as one of:

1. *fixed-length encoding* (e.g., ASCII)
2. *variable-length encoding* (e.g., Morse code)

The idea here is to reduce the amount of space needed to store a string of characters. Usually, we store characters with 8 bits each, meaning we can store up to  $2^8 = 256$  different characters.

However, many strings don't use that many different characters. If we had a string that used only 12 unique characters, we could define patterns of 4 bits each to represent them and save half of the space.

The idea behind variable length encoding is that we can do even better if we use short strings of bits to represent frequently seen characters and infrequent characters with longer strings. This results in a smaller total number of bits needed to encode a message.

To do this, we need to come up with a code and a way to translate text into code and then back again.

But...these variable length codes introduce a problem. If each character's code can have a different length, how do we know when the code for one character has ended and the next has begun?

In Morse code, how can we tell if the sequence "dot dash dash" is supposed to represent "AT", "ETT", "EM" or just the one character "W"?

This is possible because Morse code is not a binary code at all – it does have dots and dashes (which are one and three time units, respectively, of the sound), but it also has pauses of varying length to separate the individual dots and dashes (a period of silence equal in duration to the sound of a "dot"), to separate letters (silence for the duration of a dash), and to separate words (silence for the duration of 7 dots).

A strictly binary code cannot have these other "markers" to separate letters or words. Therefore, we would construct a *prefix-free code*, one where no codeword is a prefix of another codeword.

This leads to an interesting problem: if the frequencies of the character occurrences for the string to be encoded are known, what is the best binary prefix-free code?

Consider this procedure to generate a translation system, known as a *Huffman coding*.

Count the number of each character in the string to represent and create a single-node binary tree with that character and its count as the value. Repeatedly take the smallest two trees in the collection and combine them to a new tree which has the two trees as subtrees and label the root with the sum of their counts. Continue combining trees (both the original one-element trees and the trees created) in this manner until a single tree remains.

Consider the phrase:

no... try not... do... or do not... there is no try...

We count the letters up:

n=4, o=7, . =15, \_ =10, t=5, r=4, y=2, d=2, h=1, e=2, i=1, s=1

and build the tree.

Once we have that, we can use it to construct our encoded (compressed) string.

To decode, we just trace the bit patterns through the tree. When we encounter a leaf, we know the next letter. We then start tracing at the root again.

Note that the construction is a greedy procedure: we simply take the tree from our collection that has the smallest number of characters represented.