CS61B Lecture 20

Tuesday, March 10, 2020

Announcements

- Sections are being cancelled after today.
- Professor Hilfinger's Office Hours (OH) will continue as scheduled
- Project and homework deadlines are being extended.

Trees

Trees naturally represent a recursively defined, hierarchial objects with more than one recursive subpart for each instance.

Common examples will include expressions and sentences. Expressions have definitions such as "an expression consists of a literal or two expressions separated by an operator."

Also describe structures in which we recursively divide a set of multiple subsets.

Formal Definitions

Trees come in a variety of flavors defined recursively. Here are a few definitions:

- As defined by CS61A: A tree consists of a label value and zero or more branches (or children), each of them a tree.
- As defined by CS61A, alternative definition: A tree is a set of nodes (or vertices), each of which has a label value and one or more child nodes, such that no node descends (directly or indirectly) from itself. A node is the parent of its children.

We can also think of each node as having a fixed purpose. The most notable one we have seen so far is the binary tree. We will see other varieties when considering graphs.

- Positional trees: A tree is either empty or consists of a node containing a label value and an indexed sequence of zero or more children, each a positional tree. If every node has two positions, we have a binary tree and the children are its left and right subtrees. Again, nodes are the parents of their non-empty children.

Characteristics of a Tree

The **root** of a tree is a non-empty node with no parent in that tree (its parent might be in some larger tree that contains that tree as a subtree). Thus, every node is the root of a (sub)tree.

The **order**, **arity**, or **degree** of a node (tree) is its number (maximum number) of children. The **nodes** of a k-ary tree each have at most k children. A **leaf** node has no children (no non-empty children in the case of positional trees).

The **height** of a node in a tree is the largest distance to a leaf. That is, a leaf has height 0 and a non-empty tree’s height is one mor e than the maximum height of its children. The height of a tree is the height of its root.

The **depth** of a node in a tree is the distance to the root of that tree. That is, in a tree whose root is R, R itself has depth 0 in R, and if node S 6= R is in the tree with root R, then its depth is one greater than its parent's.

These definitions will be useful when discussing the efficiency of this data structure.

The `61ATree`

```java
public class Tree<Label> {
    // This constructor is convenient, but unfortunately requires this
    // SuppressWarnings annotation to prevent (harmless) warnings
    // that we will explain later.
    @SuppressWarnings("unchecked")
    public Tree(Label label, Tree<Label>... children) {
        label = label;
        kids = new ArrayList<>(Arrays.asList(children));
    }
    public int arity() { return kids.size(); }
    public Label label() { return label; }
    public Tree<Label> child(int k) { return kids.get(k); }
    private Label label;
    private ArrayList<Tree<Label>> kids;
}
```

Here is the `Tree` class as we studied it in CS61A.

Interesting in Java, while it has implementations for `ArrayList` and `LinkedList`, it does not have a formal definition for trees. Everything that uses Trees, include `TreeSet` and `TreeMap`, there's no "tree"-ness in it except in its implementation; there are no usual tree operations available to you, the programmer.

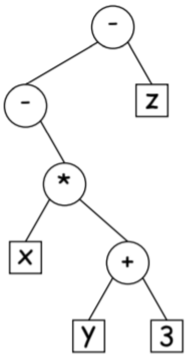
Tree Traversal

Traversing a tree means enumerating (some subset of) its nodes. Typically done recursively, because that is natural description. As nodes are enumerated, we say they are visited. Three basic orders for enumeration (+ variations):

- Preorder: visit node, traverse its children.
- Postorder: traverse children, visit node.
- Inorder: traverse first child, visit node, traverse second child (binary trees only).

Preorder Traversal and Prefix Operations

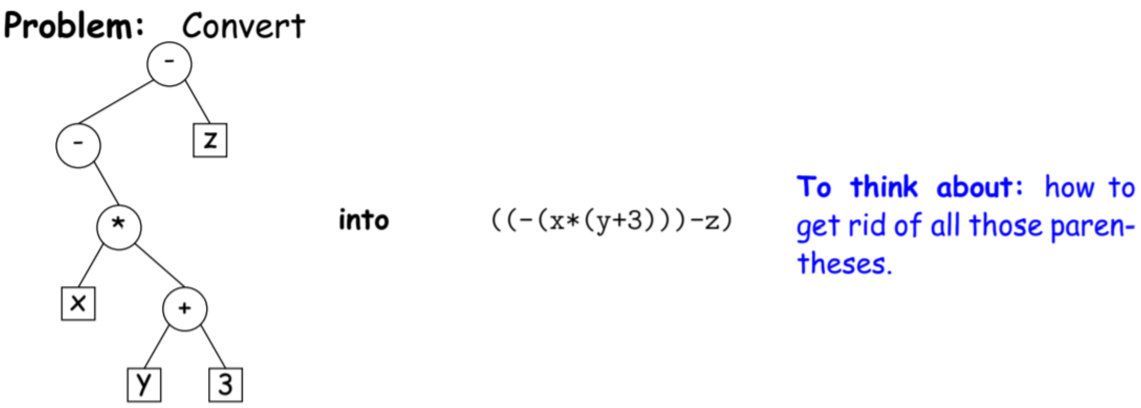
Problem: Convert



into (- (- (* x (+ y 3))) z)

```
static String toLisp(Tree<String> T) {
    if (T.arity() == 0) return T.label();
    else {
        String R; R = "(" + T.label();
        for (int i = 0; i < T.arity(); i += 1) {
            R += " " + toLisp(T.child(i));
            return R + ")";
        }
    }
```

Inorder Traversal and Infix Operations

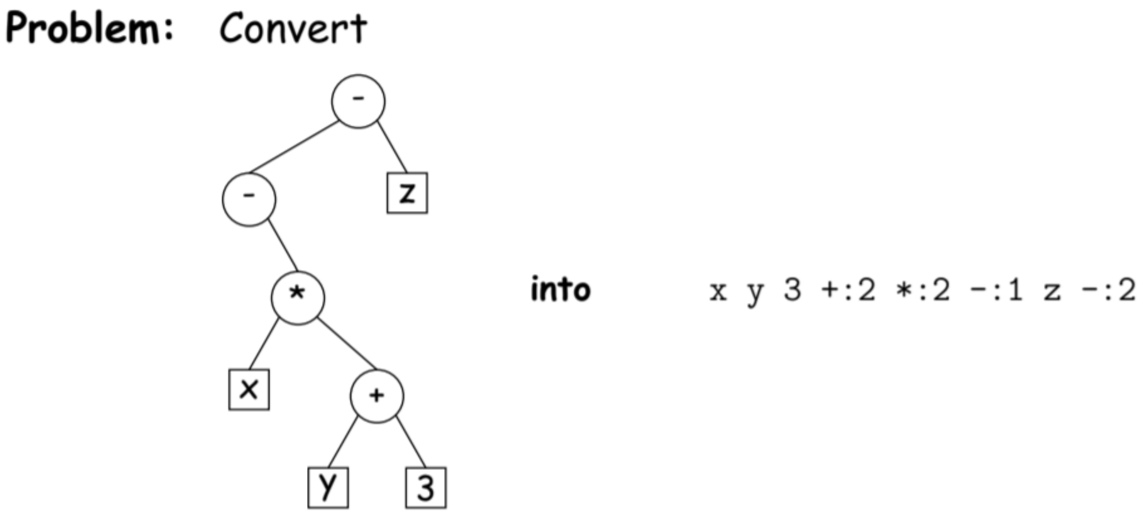

Problem: Convert

into ((-(x*(y+3)))-z)

To think about: how to get rid of all those parentheses.

```
static String toInfix(Tree<String> T) {
    if (T.arity() == 0) {
        return T.label();
    } else if (T.arity() == 1) {
        return "(" T.label() + toInfix(T.child(0)) + ")";
    } else {
        return "(" toInfix(T.child(0)) + T.label() + toInfix(T.child(1)) + ")";
    }
}
```

Postorder Traversal and Postfix Operations


Problem: Convert

into x y 3 +:2 *:2 -:1 z -:2

```
static String toPolish(Tree<String> T) {
    String R; R = "";
    for (int i = 0; i < T.arity(); i += 1) {
        R += toPolish(T.child(i)) + " ";
        return R + String.format("%s:%d", T.label(), T.arity());
    }
}
```

The Visitor Pattern

```
void preorderTraverse(Tree<Label> T, Consumer<Tree<Label>> visit) {
    if (T != null) {
        visit.accept(T);
        for (int i = 0; i < T.arity(); i += 1) {
            preorderTraverse(T.child(i), visit);
        }
    }
}
```

`java.util.function.Consumer` is a library interface that works as a function-like type with one `void` method, `accept`, which takes an argument of type `AType`.

Using Java 8 lambda syntax, I can print all labels in the tree in preorder with: `preorderTraverse(myTree, T -> System.out.print(T.label() + " "));`

Iterative Depth-First Traversal

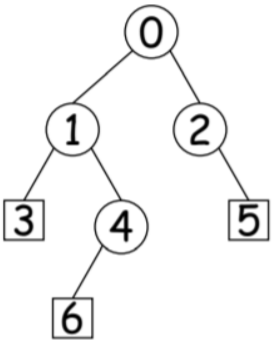
Tree recursion conceals data: a stack of nodes (all the T arguments) and a little extra information. We can make the data explicit:

```
void preorderTraverse2(Tree<Label> T, Consumer<Tree<Label>> visit) {
    Stack<Tree<Label>> work = new Stack<>();
    work.push(T);
    while (!work.isEmpty()) {
        Tree<Label> node = work.pop();
        visit.accept(node);
        for (int i = node.arity()-1; i >= 0; i -= 1) {
            work.push(node.child(i)); // Why backward?
        }
    }
}
```

This traversal takes the same Big Theta time as doing it recursively, and also the same Big Theta space. That is, we have substituted an explicit stack data structure (`work`) for Java's built-in execution stack (which handles function calls).

Level-Order Traversal

Problem: Traverse all nodes at depth 0, then depth 1, etc:



A simple modification to iterative depth-first traversal gives breadthfirst traversal. Just change the (LIFO) stack to a (FIFO) queue:

```java
void breadthFirstTraverse(Tree<Label> T, Consumer<Tree<Label>> visit) {
    ArrayDeque<Tree<Label>> work = new ArrayDeque<>(); // (Changed)
    work.push(T);
    while (!work.isEmpty()) {
        Tree<Label> node = work.remove(); // (Changed)
        if (node != null) {
            visit.accept(node);
            for (int i = 0; i < node.arity(); i += 1) { // (Changed)
                work.push(node.child(i));
            }
        }
    }
}
```

Times

The traversal algorithms have roughly the form of the boom example in §1.3.3 of Data Structures —an exponential algorithm. However, the role of M in that algorithm is played by the height of the tree, not the number of nodes. In fact, easy to see that tree traversal is linear to N, where N is the # of nodes: Form of the algorithm implies that there is one visit at the root, and then one visit for every edge in the tree. Since every node but the root has exactly one parent, and the root has none, must be N − 1 edges in any non-empty tree.

In a positional tree, is also one recursive call for each empty tree, but # of empty trees can be no greater than kN, where k is arity. For k-ary tree (where the max # children is k):

$$h + 1 \le N \le \frac{k^{h+1} - 1}{k - 1}$$

where h is height.

So,

$$h \in \Omega(log_k N) = \Omega(lgN) \text{ and } h \in O(N)$$

Many tree algorithms look at one child only. For them, worst-case time is proportional to the height of the tree, linear to lg N, assuming that tree is bushy—each level has about as many nodes as possible.

Recursive Breadth-First Traversal

Previous breadth-first traversal used space proportional to the width of the tree, which is Theta N for bushy trees, whereas depth-first traversal takes logarithmic space on bushy trees.

Can we get the best of both worlds, so that we get breadth-first traversal in logarithmic space and linear time on bushy trees? We can do this by, for each level, k, of the tree from 0 to `lev`, call `doLevel(T, k)`:

```
void doLevel(Tree T, int lev) {
    if (lev == 0) {
        // visit T
    } else  {
        for each non-null child, C, of T {
            doLevel(C, lev-1);
        }
    }
}
```

In this method, we do breadth-first traversal by repeated (truncated) depthfirst traversals: iterative deepening

In `doLevel(T, k)`, we skip (i.e., traverse but don’t visit) the nodes before level k, and then visit at level k, but not their children.

Iterative Deepening Time?

In general, the number of leaves at a level in a tree is approximately equal to the number of nodes in all the levels before it.

This results in the interesting fact that if you pursue a depth-first traversal, level-by-level, you get the same order as a breadth-first traversal, and even though you seem to do a fair bit more work, it is only a constant factor more.

Iterators for Trees

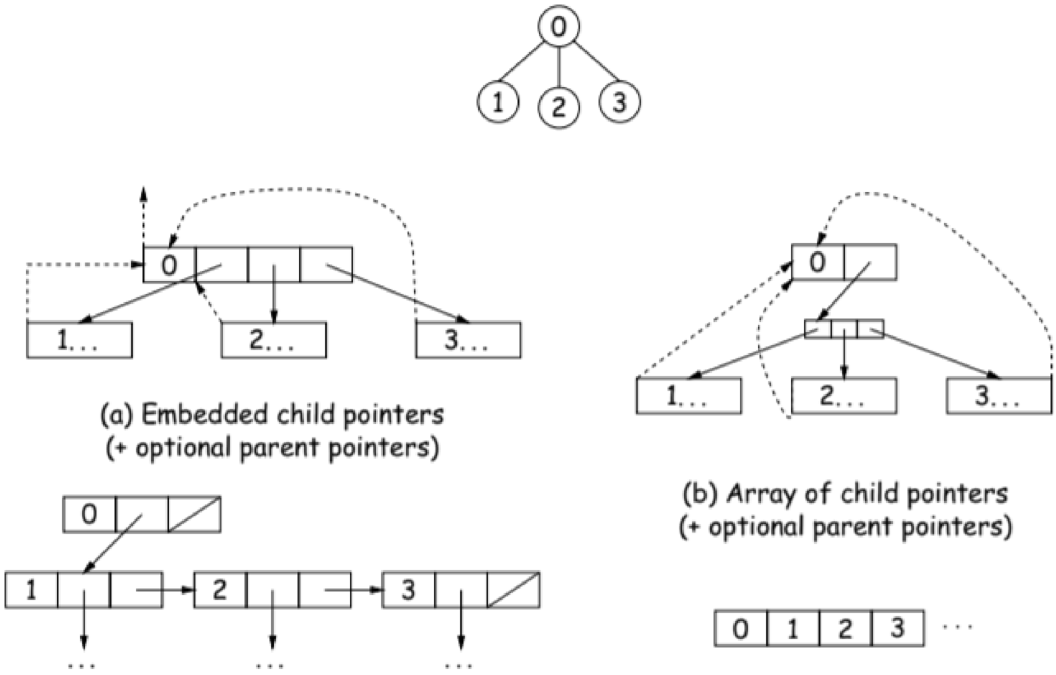
Frankly, iterators are not terribly convenient on trees. But we can use our ideas from iterative methods to do so:

```
class PreorderTreeIterator<Label> implements Iterator<Label> {
    private Stack<Tree<Label>> s = new Stack<Tree<Label>>();
    public PreorderTreeIterator(Tree<Label> T) { s.push(T); }
    public boolean hasNext() { return !s.isEmpty(); }
    public T next() {
        Tree<Label> result = s.pop();
        for (int i = result.arity()-1; i >= 0; i -= 1) {
            s.push(result.child(i));
        }
        return result.label();
    }
}
```

We could also modify this method to take in `<Tree<Label>>` and return the nodes instead of just labels.e

Tree Representation

Tree Representation



(a) Embedded child pointers
(+ optional parent pointers)

(b) Array of child pointers
(+ optional parent pointers)

(c) child/sibling pointers

(d) breadth-first array
(complete trees)