CS61B Lecture #20: Trees
A Recursive Structure

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

- Common examples: expressions, 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 into multiple subsets.
Formal Definitions

• Trees come in a variety of flavors, all defined recursively:

  – **61A style**: A tree consists of a *label* value and zero or more *branches* (or *children*), each of them a tree.

  – **61A style, 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.

  – **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 sub-trees*. Again, nodes are the parents of their non-empty children.

  – We’ll see other varieties when considering graphs.
Tree Characteristics (I)

- 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* of a node (or tree) is its number of children.

- A *leaf* node has no children (no non-empty children in the case of positional trees).

- The number of children of a node is the *order* of the node.

- The nodes of a *k-ary tree* each have at most $k$ children. (I sometimes use the term *arity* for the order a node or maximum order of its nodes.)
Tree Characteristics (II)

• The **height** of a node in a tree is the smallest distance to a leaf. That is, a leaf has height 0 and a non-empty tree’s height is one more 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 $S \neq R$ is in the tree with root $R$, then its depth is one greater than its parent’s.
Fundamental Operation: 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).

```
Postorder          Preorder          inorder
6                  0                  4
  3    5                1    5                1
    0  2  4            2  3  6            0  3  6
      1               4               2
```

Preorder Traversal and Prefix Expressions

Problem: Convert

\[- \ (\ (- \ (* \ x \ (+ \ y \ 3))) \ z)\]

into

\[- \ (- \ (* \ x \ (+ \ y \ 3))) \ z\]

(Assume Tree<Label> is means “Tree whose labels have type Label.”)

static String toLisp(Tree<String> T) {
    if (T == null) return "";
    else if (T.degree() == 0) return T.label();
    else {
        String R;  R = "";
        for (int i = 0; i < T.numChildren(); i += 1)
            R += " " + toLisp(T.child(i));
        return String.format("(%s%s)", T.label(), R);
    }
}
Inorder Traversal and Infix Expressions

Problem: Convert

\[ - \frac{-(x(y+3))}{z} \]

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

```java
static String toInfix(Tree<String> T) {
    if (T == null)
        return "";
    if (T.degree() == 0)
        return T.label();
    else {
        String left = toInfix(T.left()), right = toInfix(T.right());
        return String.format("(\%s\%s\%s)", left, T.label(), right);
    }
}
```
Postorder Traversal and Postfix Expressions

Problem: Convert

\[- ( - x * y + 3 : 2 * : 2 z : 1 - : 2 ) \Rightarrow x y 3 + : 2 * : 2 - : 1 z - : 2 \]

```java
static String toPolish(Tree<String> T) {
    if (T == null)
        return "";
    else {
        String R; R = "";
        for (int i = 0; i < T.numChildren(); i += 1)
            R += toPolish(T.child(i)) + " ";
        return String.format("%s%s:%d", R, T.label(), T.degree());
    }
}
```
A General Traversal: The Visitor Pattern

```java
void preorderTraverse(Tree<Label> T, Action<Label> whatToDo)
{
    if (T != null) {
        whatToDo.action(T);
        for (int i = 0; i < T.numChildren(); i += 1)
            preorderTraverse(T.child(i), whatToDo);
    }
}
```

- What is Action?

```java
interface Action<Label> {
    void action(Tree<Label> T);
}
```

Now, using Java 8 lambda syntax, I can print all labels in the tree in preorder with:

```java
preorderTraverse(myTree,
    (Tree<String> T) -> System.out.print(T.label()));
```
Iterative Depth-First Traversals

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

```java
void preorderTraverse2(Tree<Label> T, Action whatToDo) {
    Stack<Tree<Label>> s = new Stack<>();
    s.push(T);
    while (!s.isEmpty()) {
        Tree<Label> node = s.pop();
        if (node != null) {
            whatToDo.action(node);
            for (int i = node.numChildren()-1; i >= 0; i -= 1)
                s.push(node.child(i)); // Why backward?
        }
    }
}
```

• To do a breadth-first traversal, use a queue instead of a stack, replace `push` with `add`, and `pop` with `removeFirst`.

• Makes breadth-first traversal worst-case linear time in all cases, but also linear space for “bushy” trees.
Level-Order (Breadth-First) Traversal

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

```
  0
 / \
1   2
|   |
3   4
|   |
  6
```

Traverse: 0, 1, 2, 3, 4, 5, 6
Breadth-First Traversal Implemented

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

```java
void preorderTraverse2(Tree<Label> T, Action whatToDo) {
    ArrayDeque<Tree<Label>> s = new ArrayDeque<>(); // (Changed)
    s.push(T);
    while (!s.isEmpty()) {
        Tree<Label> node = s.remove(); // (Changed)
        if (node != null) {
            whatToDo.action(node);
            for (int i = 0; i < node.numChildren(); i += 1) // (Changed)
                s.push(node.child(i));
        }
    }
}
```
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: $\Theta(N)$, 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 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 (max # children is $k$), $h + 1 \leq N \leq \frac{k^{h+1}-1}{k-1}$, where $h$ is height.

So $h \in \Omega(\log_k N) = \Omega(\lg N)$ and $h \in O(N)$.

Many tree algorithms look at one child only. For them, time is proportional to the height of the tree, and this is $\Theta(\lg N)$, assuming that tree is bushy—each level has about as many nodes as possible.
Recursive Breadth-First Traversal: Iterative Deepening

- For each level, $k$, of the tree from 0 to $h$, call `doLevel(T,k)`:  
  ```java
  void doLevel(Tree T, int lev) {
      if (lev == 0)
          visit T
      else
          for each non-null child, C, of T {
              doLevel(C, lev-1);
          }
  }
  ```

- We do breadth-first traversal by repeated (truncated) depth-first traversals.

- 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?

- Let $h$ be height, $N$ be # of nodes.
- Count # edges traversed (i.e, # of calls, not counting null nodes).
- First (full) tree: 1 for level 0, 3 for level 1, 7 for level 2, 15 for level 3.
- Or in general $(2^1 - 1) + (2^2 - 1) + \ldots + (2^{h+1} - 1) = 2^{h+2} - h \in \Theta(N)$, since $N = 2^{h+1} - 1$ for this tree.
- Second (right leaning) tree: 1 for level 0, 2 for level 2, 3 for level 3.
- Or in general $(h + 1)(h + 2)/2 = N(N + 1)/2 \in \Theta(N^2)$, since $N = h + 1$ for this kind of tree.
Iterators for Trees

- Frankly, iterators are not terribly convenient on trees.
- But can use ideas from iterative methods.

    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.numChildren()-1; i >= 0; i -= 1)
                s.push(result.child(i));
            return result.label();
        }
        void remove() { throw new UnsupportedOperationException(); }
    }

Example: (what do I have to add to class Tree first?)

    for (String label : aTree) System.out.print(label + " ");
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)