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 subtrees.* Again, nodes are the parents of their non-empty children.

– We'll see other varieties when consider-
ing 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*, *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).

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.

A Tree Type, 61A Style

```java
public class Tree<Label> {

    // This constructor is convenient, but unfortunately
causes
    // (harmless) warnings that we will explain
later.

    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;
```

```
}
```

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).
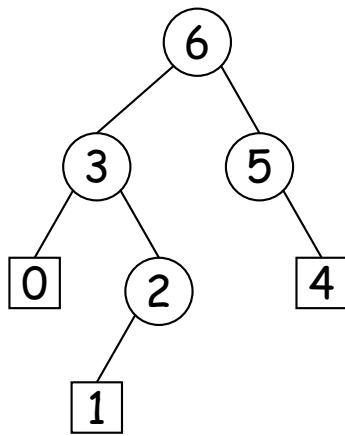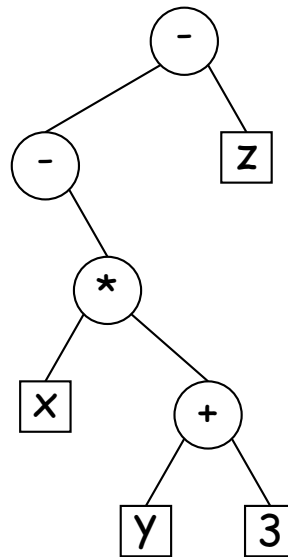
Postorder
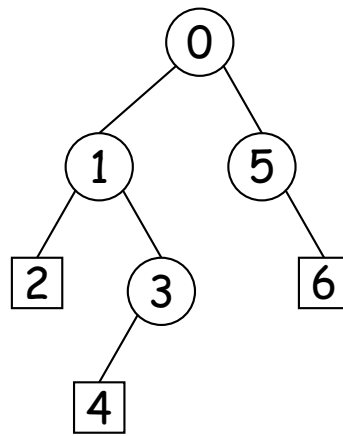
Preorder
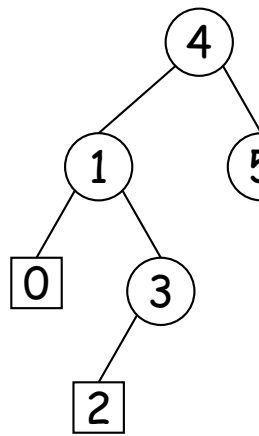
inorder

Preorder Traversal and Prefix Expressions

Problem: Convert   **into**   (– (– (*
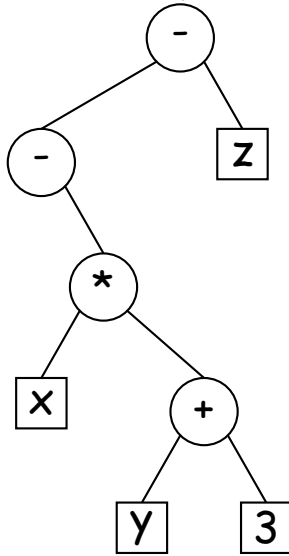
(Assume `Tree`<*Label*> is means "Tree whose labels have type *Label*.)

```java
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 Expressions
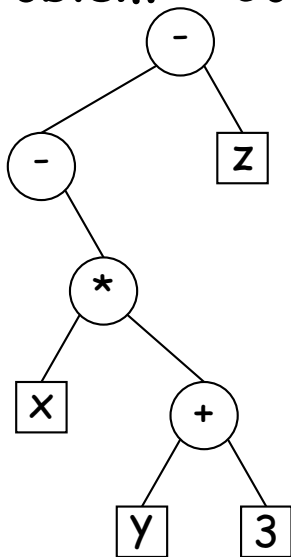
Problem: Convert

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 Expressions

Problem: Convert

$\Rightarrow$ 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());
  }
```

A General Traversal: The Visitor Pattern

```
void preorderTraverse(Tree<Label> T, Consumer<T
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<AType>` is a library interface that works as a function-like type with one void method, `accept`, which takes an argument of type `AType`.

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

```
    preorderTraverse(myTree, T -> System.out.print(T.labe
+ " "));
```
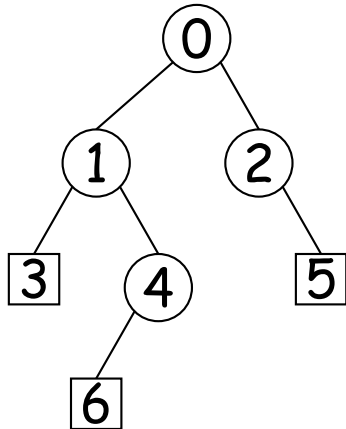
Iterative Depth-First Traversals

- 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, Consumer<Tree<Labe
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 $\Theta(\cdot)$ time as doing it recursively, and also the same $\Theta(\cdot)$ 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 (Breadth-First) Traversal

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

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 breadthFirstTraverse(Tree<Label> T, Consumer<Tree<L
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:* $\Theta(N)$, 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), $h+1 \le N \le \frac{k^{h+1}-1}{k-1}$, 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, worst-case time is proportional to the *height* of the tree—$\Theta(\lg N)$—assuming that tree is *bushy*—each level has about as many nodes as possible.

Recursive Breadth-First Traversal: Iterative Deepening

- 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 $\lg N$ space on bushy trees.

- Can we get breadth-first traversal in $\lg N$ space and $\Theta(N)$ time on bushy trees?

- 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);
    }
}
```

- So we do breadth-first traversal by repeated

(truncated) depth-first 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?
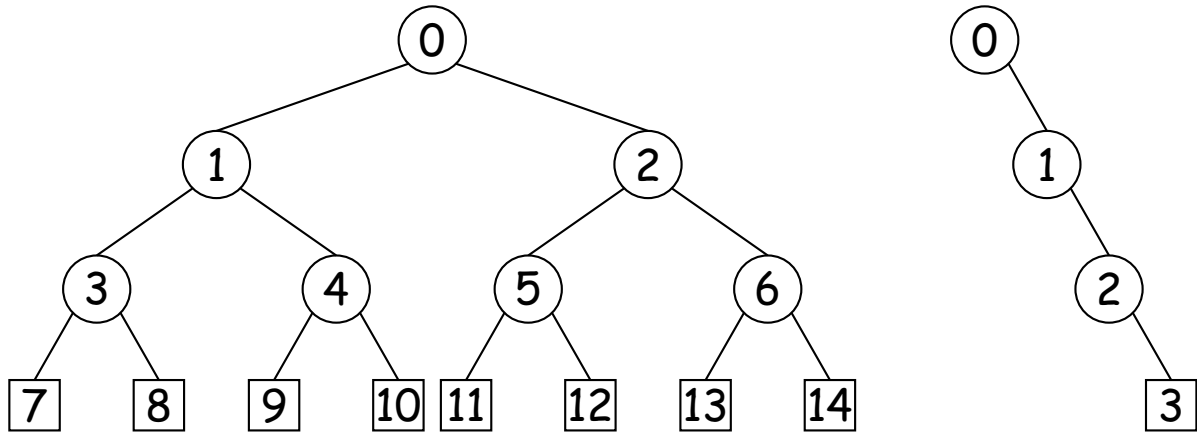
- Let h be height, N be # of nodes.

- 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.

```java
    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();
    }
  }
```
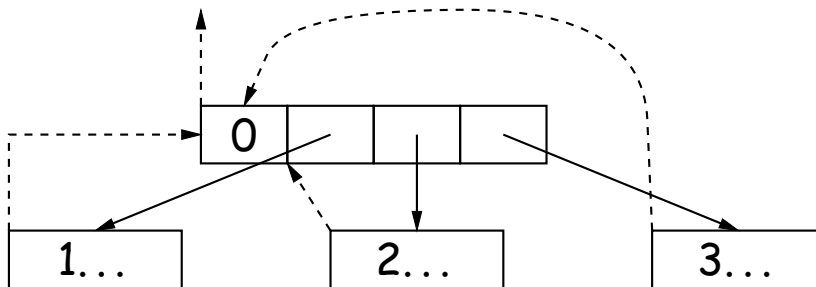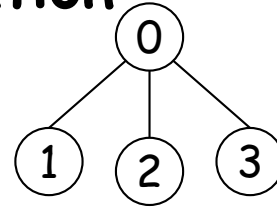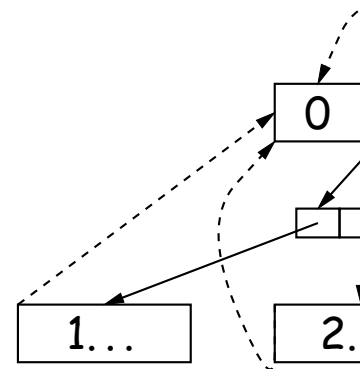
Example: (what do I have to add to class `Tree` first?)

```java
for (String label : aTree) System.out.print(label
    + " ");
```
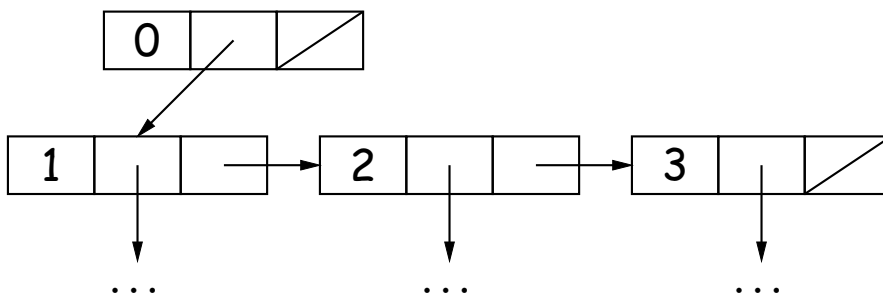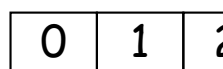
Tree Representation

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

(b) Array of
(+ optional par

(c) child/sibling pointers

(d) breadth
(complet