CS61B Lecture #20: Trees

A Recursive Structure

lly represent recursively defined, hierarchical objects
an one recursive subpart for each instance.

mples: expressions, sentences.

ns have definitions such as "an expression consists of a
two expressions separated by an operator."

e structures in which we recursively divide a set into
sets.

Formal Definitions

n a variety of flavors, all defined recursively:

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

, alternative definition: A tree is a set of *nodes* (or
each of which has a label value and one or more *child*
ch that no node descends (directly or indirectly) from
ode is the *parent* of its children.

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

other varieties when considering graphs.

Tree Characteristics (I)

a tree is a non-empty node with no parent in that tree
ight be in some larger tree that contains that tree as
hus, every node is the root of a (sub)tree.

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

has no children (no non-empty children in the case of
es).

of children of a node is the *order* of the node.

f a *k-ary tree* each have at most k children. (I some-
e term *arity* for the order a node or maximum order of

Tree Characteristics (II)

of a node in a tree is the smallest distance to a leaf.
af has height 0 and a non-empty tree's height is one
e maximum height of its children. The height of a tree
of its root.

f a node in a tree is the distance to the root of that
, in a tree whose root is R, R itself has depth 0 in R,
$S \neq R$ is in the tree with root R, then its depth is one
its parent's.

undamental Operation: Traversal

tree means enumerating (some subset of) its nodes.

e recursively, because that is natural description.

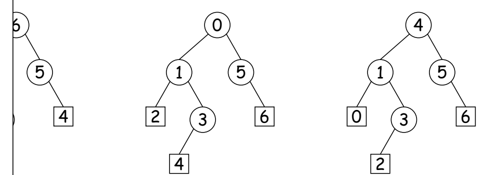
enumerated, we say they are *visited.*

orders for enumeration (+ variations):

visit node, traverse its children.

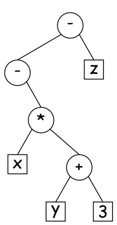
: traverse children, visit node.

traverse first child, visit node, traverse second child
ees only).



rder *Preorder* *inorder*

der Traversal and Prefix Expressions



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

<*Label*> is means "Tree whose labels have type *Label*.)

```
 toLisp(Tree<String> T) {
l) return "";
legree() == 0) return T.label();

 R = "";
 = 0; i < T.numChildren(); i += 1)
 + toLisp(T.child(i));
ing.format("(%s%s)", T.label(), R);
```


der Traversal and Infix Expressions

ert

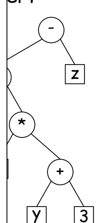
into ((-(x*(y+3)))-z)

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

```
 toInfix(Tree<String> T) {
l)

() == 0)
abel();

t = toInfix(T.left()), right = toInfix(T.right());
ing.format("(%s%s%s)", left, T.label(), right);
```


der Traversal and Postfix Expressions

ert



⇒ x y 3 +:2 *:2 -:1 z -:2

```
 toPolish(Tree<String> T) {
l)

 R = "";
 = 0; i < T.numChildren(); i += 1)
olish(T.child(i)) + " ";
ing.format("%s%s:%d", R, T.label(), T.degree());
```


neral Traversal: The Visitor Pattern

```
erTraverse(Tree<Label> T, Action<Label> whatToDo)

ull) {
.action(T);
 i = 0; i < T.numChildren(); i += 1)
derTraverse(T.child(i), whatToDo);
```

ion?

```
on<Label> {
Tree<Label> T);
```

va 8 lambda syntax, I can print all labels in the tree in

```
se(myTree,
   (Tree<String> T) -> System.out.print(T.label()));
```


terative Depth-First Traversals

on conceals data: a *stack* of nodes (all the T arguments) xtra information. Can make the data explicit:

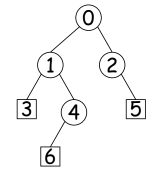
```
raverse2(Tree<Label> T, Action whatToDo) {
abel>> s = new Stack<>();

Empty()) {
> node = s.pop();
= null) {
Do.action(node);
nt i = node.numChildren()-1; i >= 0; i -= 1)
ush(node.child(i));  // Why backward?
```

adth-first traversal, use a queue instead of a stack, with add, and pop with removeFirst.

dth-first traversal worst-case linear time in all cases, r *space* for "bushy" trees.

l-Order (Breadth-First) Traversal

erse all nodes at depth 0, then depth 1, etc:



adth-First Traversal Implemented

cation to iterative depth-first traversal gives breadth- Just change the (LIFO) stack to a (FIFO) queue:

```
raverse2(Tree<Label> T, Action whatToDo) {
ree<Label>> s = new ArrayDeque<>();  // (Changed)

Empty()) {
> node = s.remove();     // (Changed)
= null) {
Do.action(node);
nt i = 0; i < node.numChildren(); i += 1) // (Changed)
ush(node.child(i));
```


Times

l algorithms have roughly the form of the boom example *Data Structures*—an exponential algorithm.

role of M in that algorithm is played by the *height* of the number of nodes.

to see that tree traversal is *linear:* $\Theta(N)$, where N nodes: Form of the algorithm implies that there is one root, and then one visit for every *edge* in the tree. node but the root has exactly one parent, and the root st be $N-1$ edges in any non-empty tree.

tree, is also one recursive call for each empty tree, but trees can be no greater than kN, where k is arity.

ee (max # children is k), $h+1 \le N \le \frac{k^{h+1}-1}{k-1}$, where h is

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

gorithms look at one child only. For them, time is pro- the *height* of the tree, and this is $\Theta(\lg N)$, assuming *bushy*—each level has about as many nodes as possible.

eadth-First Traversal: Iterative Deepening

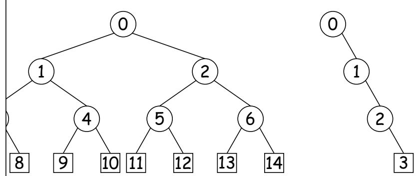
el, k, of the tree from 0 to h, call doLevel(T,k):

```
el(Tree T, int lev) {
= 0)

h non-null child, C, of T {
el(C, lev-1);
```

lth-first traversal by repeated (truncated) depth-first

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

Iterative Deepening Time?



ght, N be # of nodes.

es traversed (i.e, # of calls, not counting null nodes).

ee: 1 for level 0, 3 for level 1, 7 for level 2, 15 for level

$(2^1-1)+(2^2-1)+\ldots+(2^{h+1}-1) = 2^{h+2}-h \in \Theta(N)$, $^{+1}-1$ for this tree.

t leaning) tree: 1 for level 0, 2 for level 2, 3 for level 3.

$(h+1)(h+2)/2 = N(N+1)/2 \in \Theta(N^2)$, since $N=h+1$ of tree.

Iterators for Trees

ators are not terribly convenient on trees.

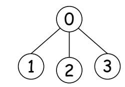
deas from iterative methods.

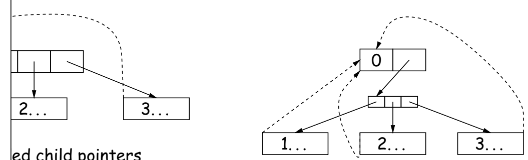
```
rderTreeIterator<Label> implements Iterator<Label> {
Stack<Tree<Label>> s = new Stack<Tree<Label>>();

reorderTreeIterator(Tree<Label> T) { s.push(T);  }

olean hasNext() { return !s.isEmpty(); }
 next() {
abel> result = s.pop();
nt i = result.numChildren()-1; i >= 0; i -= 1)
sh(result.child(i));
 result.label();

ove() { throw new UnsupportedOperationException(); }
```

t do I have to add to class Tree first?)

```
ring label : aTree) System.out.print(label + " ");
```

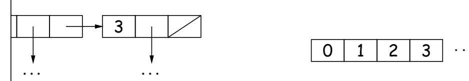

Tree Representation





ed child pointers
parent pointers)

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



sibling pointers

(d) breadth-first array
(complete trees)