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
node 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-*
ain, 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
Thus, every node is the root of a (sub)tree.

rity, or *degree* of a node (tree) is its number (maximum
hildren.

a *k-ary tree* each have at most k children.

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

Tree Characteristics (II)

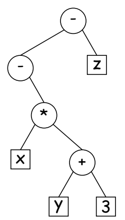
f a node in a tree is the largest distance to a leaf. That
s height 0 and a non-empty tree’s height is one more
ximum height of its children. The height of a tree is the
root.

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.

A Tree Type, 61A Style

```
ree<Label> {

nstructor is convenient, but unfortunately requires this
sWarnings annotation to prevent (harmless) warnings
 will explain later.
arnings("unchecked")
e(Label label, Tree<Label>... children) {
 = label;
= new ArrayList<>(Arrays.asList(children));


 arity() { return _kids.size(); }

el label() { return _label; }

e<Label> child(int k) { return _kids.get(k); }

bel _label;
rayList<Tree<Label>> _kids;
```

der Traversal and Prefix Expressions



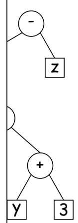
into `(- (- (* x (+ y 3))) z)`

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

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

 R = "(" + T.label();
 = 0; i < T.arity(); i += 1)
 + toLisp(T.child(i));
 ")";
```


der Traversal and Postfix Expressions

ert



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

```
toPolish(Tree<String> T) {
 = "";
 0; i < T.arity(); i += 1)
lish(T.child(i)) + " ";
tring.format("%s:%d", T.label(), T.arity());
```


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, Consumer<Tree<Label>> visit) {
abel>> work = new Stack<>();
;
.isEmpty()) {
> node = work.pop();
pt(node);
 = node.arity()-1; i >= 0; i -= 1)
ush(node.child(i));  // Why backward?
```

al takes the same $\Theta(\cdot)$ time as doing it recursively, and
e $\Theta(\cdot)$ space.

ave substituted an explicit stack data structure (`work`)
ilt-in execution stack (which handles function calls).

undamental Operation: Traversal

tree means enumerating (some subset of) its nodes.

e recursively, because that is natural description.

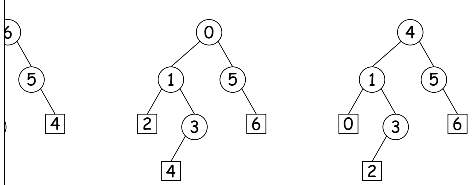
enumerated, we say they are *visited*.

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 Infix Expressions

ert

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

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

```
toInfix(Tree<String> T) {
) == 0) {
abel();
.arity() == 1) {
 T.label() + toInfix(T.child(0)) + ")";

 toInfix(T.child(0)) + T.label() + toInfix(T.child(1)) + ")";
```


neral Traversal: The Visitor Pattern

```
erTraverse(Tree<Label> T, Consumer<Tree<Label>> visit)

ull) {
cept(T);
 i = 0; i < T.arity(); i += 1)
derTraverse(T.child(i), visit);
```

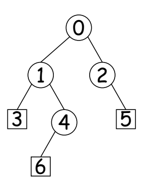
`unction.Consumer<AType>` is a library interface that
unction-like type with one void method, `accept`, which
ument of type `AType`.

ava 8 lambda syntax, I can print all labels in the tree in
h:

```
averse(myTree, T -> System.out.print(T.label() + " "));
```

el-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:

```
rstTraverse(Tree<Label> T, Consumer<Tree<Label>> visit) {
ree<Label>> work = new ArrayDeque<>();  // (Changed)
;
.isEmpty()) {
> node = work.remove();    // (Changed)
= null) {
accept(node);
nt i = 0; i < node.arity(); i += 1) // (Changed)
k.push(node.child(i));
```


Times

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

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

y 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
rees can be no greater than kN, where k is arity.

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

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

lgorithms look at one child only. For them, worst-case
ortional to the *height* of the tree—$\Theta(\lg N)$—assuming
bushy—each level has about as many nodes as possible.

eadth-First Traversal: Iterative Deepening

adth-first traversal used space proportional to the *width*
which is $\Theta(N)$ for bushy trees, whereas depth-first
es $\lg N$ space on bushy trees.

breadth-first traversal in $\lg N$ space and $\Theta(N)$ time on

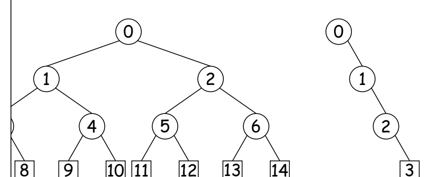
l, k, of the tree from 0 to lev, call `doLevel(T,k)`:

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


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

eadth-first traversal by repeated (truncated) depth-
als: *iterative deepening.*

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).

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

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

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

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

Iterators for Trees

ators are not terribly convenient on trees.

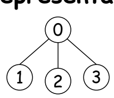
deas from iterative methods.

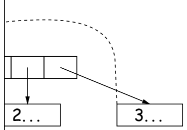
```
rderTreeIterator<Label> implements Iterator<Label> {
Stack<Tree<Label>> s = new Stack<Tree<Label>>();

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

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

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

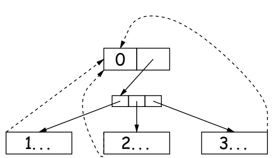
```
ring label : aTree) System.out.print(label + " ");
```

Tree Representation

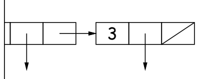




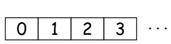
ed child pointers
parent pointers)



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



sibling pointers



(d) breadth-first array
(complete trees)