

# CS61B Lecture #21

## Administrative:

- For alternative test times, come to my office.

## Today:

- Trees

Readings for Today: *Data Structures, Chapter 5*

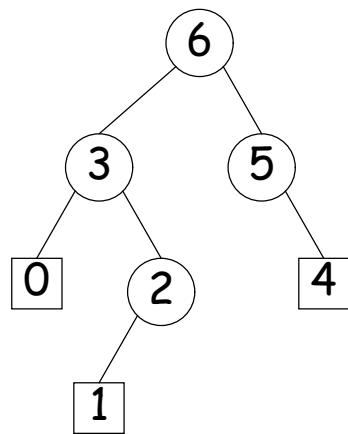
Readings for Next Topic: *Data Structures, Chapter 6*

# A Recursive Structure

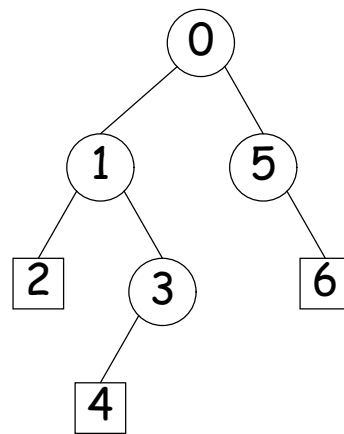
- Trees naturally represent recursively defined objects with more than one recursive instance.
- Common examples: expressions, sentences.
  - Expressions have definitions such as "an expression consists of a literal or two expressions separated by an operator."
- Also describe structures in which we recursively divide a set into multiple subsets.

# Fundamental Operation: Traversal

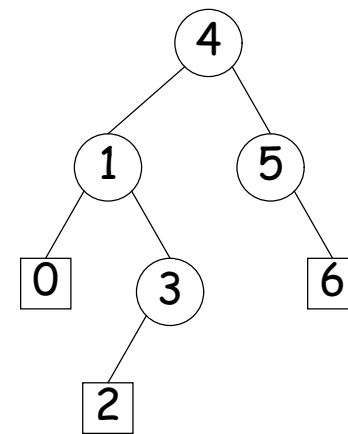
- *Traversing a tree* means enumerating (some subset of) its nodes.
- Typically done recursively, because that is natural description.
- As nodes are enumerated, we say they are *visited*.
- Three basic orders for enumeration (+ variations):
  - **Preorder**: visit node, traverse its children.
  - **Postorder**: traverse children, visit node.
  - **Inorder**: traverse first child, visit node, traverse second child (binary trees only).



*Postorder*



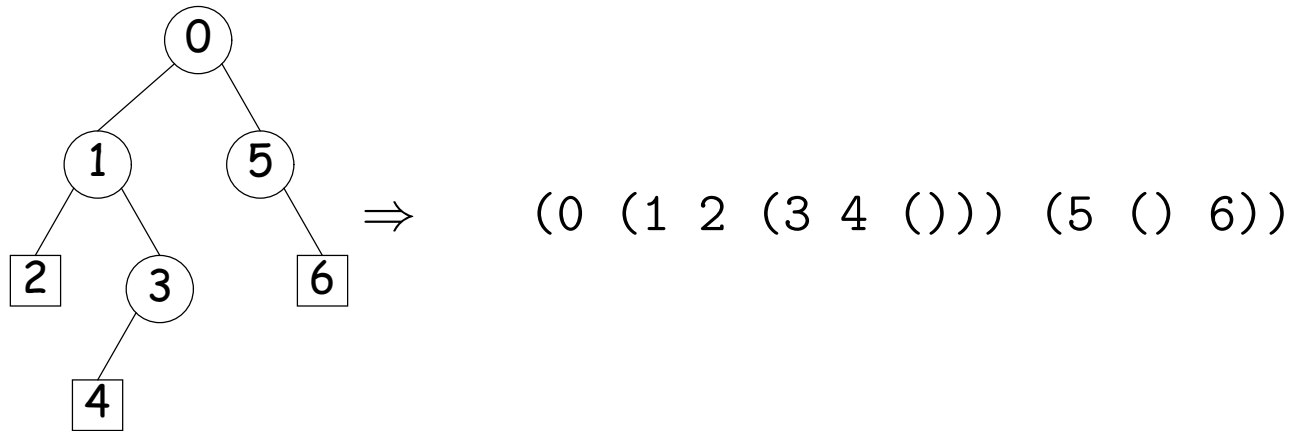
*Preorder*



*inorder*

## ... and Traversal Code (Preorder)

**Problem:** Convert



```
void printAsLisp (Tree T, PrintWriter p) {  
    if (T == null)  
        p.print (" (");  
    else if (T.degree () == 0)  
        p.print (" " + T.label ());  
    else {  
        p.print "(" + T.label ());  
        for (int i = 0; i < T.maxChild (); i += 1)  
            printAsLisp (T.child (i), p);  
        p.print (")");  
    }  
}
```

# Preorder Traversal Code: General

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

- What is Action?

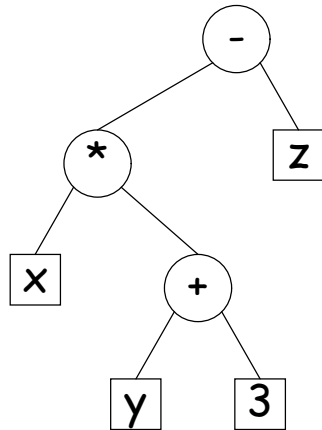
```
interface Action {
    void action (Tree T);
}
```

```
class Print implements Action {
    void action (Tree T) {
        System.out.print (T.label ());
    }
}

preorderTraverse (myTree,
                  new Print ());
```

# Inorder Traversal and Infix Expressions

Problem: Convert



$\Rightarrow ((x*(y+3))-z)$

```
void printExpr (Tree T, PrintWriter p) {
    if (T.degree () == 0)
        p.print (T.label ());
    else {
        p.print "(";
        printExpr (T.left (), p);
        p.print (T.label ());
        printExpr (T.right (), p);
        p.print (")");
    }
}
```

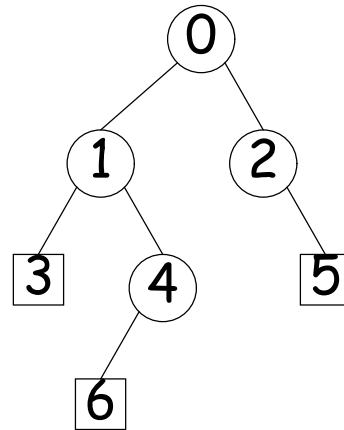
• **To think about:** how to get rid of all those parentheses?

# 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  $k$ ),  $h + 1 \leq N \leq \frac{k^{h+1}-1}{k-1}$ , 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.

# Level-Order (Breadth-First) Traversal

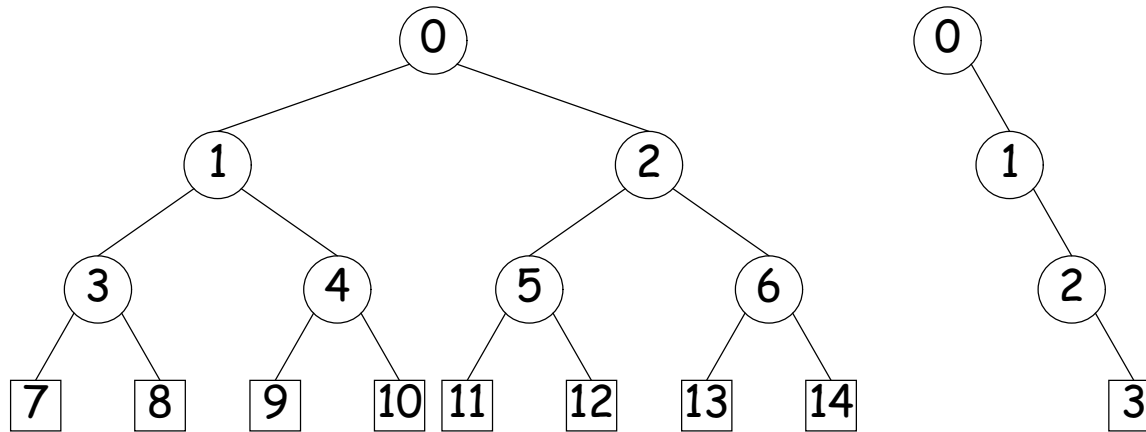
**Problem:** Traverse all nodes at depth 0, then depth 1, etc:



- One technique: *Iterative Deepening*. For each level,  $k$ , from 0 to  $h$ , 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);  
        }  
}
```

# 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  $(2^1 - 1) + (2^2 - 1) + \dots + (2^{h+1} - 1) = 2^{h+2} - h \in \Theta(N)$ , 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.

# Iterative Traversals

- Tree recursion conceals data: a *stack* of nodes (all the T arguments) and a little extra information. Can make the data explicit, e.g.:

```
void preorderTraverse2 (Tree T, Action whatToDo) {
    Stack s = new Stack ();
    s.push (T);
    while (! s.isEmpty ()) {
        Tree node = (Tree) s.pop ();
        if (node == null)
            continue;
        whatToDo.action (node);
        for (int i = node.maxChild ()-1; i >= 0; i -= 1)
            s.push (node.child (i));
    }
}
```

- 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.

# Iterators for Trees

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

```
class PreorderTreeIterator implements Iterator {
    private Stack s = new Stack ();

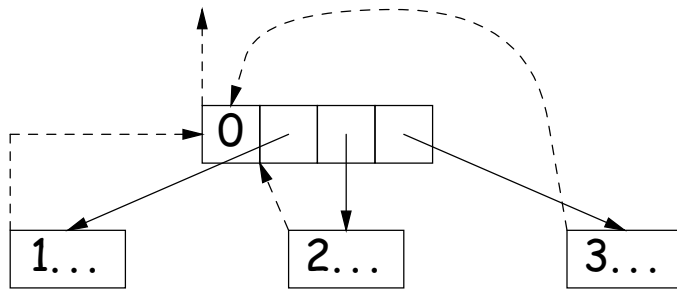
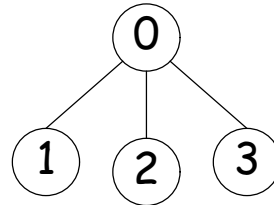
    public PreorderTreeIterator (Tree T) { s.push (T); }

    public boolean hasNext () { return ! s.isEmpty (); }
}

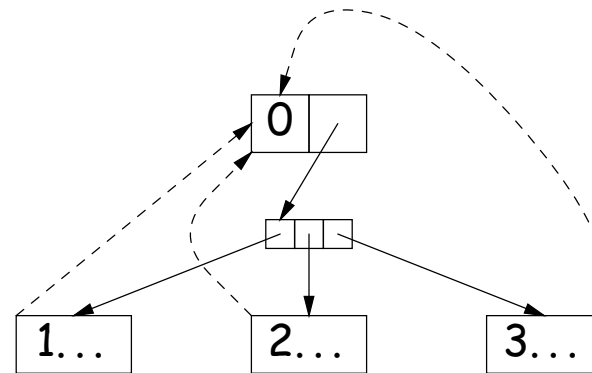
public Object next () {
    Tree result = (Tree) s.pop ();
    for (int i = result.maxChild ()-1; i >= 0; i -= 1)
        s.push (result.child (i));
    return result.label ();
}

void remove () { throw new UnsupportedOperationException (); }
}
```

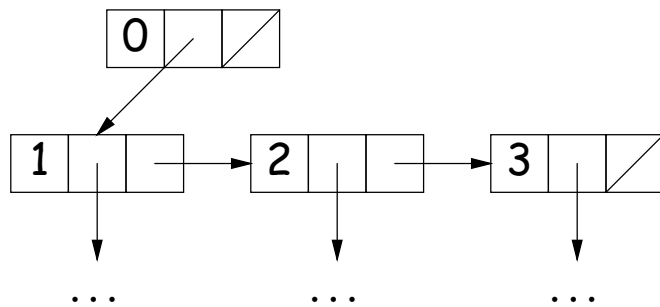
# Representation Choices



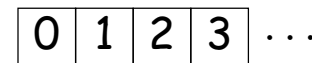
(a) Embedded child pointers  
(+ optional parent pointers)



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



(c) child/sibling pointers



(d) pre-order array  
(complete trees)