Administrative Details

• Lab 7 Today: PostScript
  • No partners this week
  • Review before lab; come to lab with design doc
  • Check out the javadoc pages for the 3 provided classes
    • Token – A wrapper for semantic PS elements,
    • Reader – An iterator to produce a stream of Tokens from standard input or a List of Tokens,
    • SymbolTable – A dictionary with String keys and Token values: For user-defined names
Last Time:

- Ordered Structures
- Trees
  - Structure, Terminology, Examples
Today

• Trees
  • Implementation
  • Recursion/Induction on Trees
  • Applications
  • Traversals
Tree Features

- Hierarchical relationship
- **Root** at the top
- **Leaf** at the bottom
- **Interior nodes** in middle
- Parents, children, ancestors, descendants, siblings
- **Degree (of node)**: number of children of node
- **Degree (of tree)**: maximum degree (across all nodes)
- **Depth** of node: number of edges from root to node
- **Height** of tree: maximum depth (across all nodes)
Expression Trees

\[ 4 \times 2 + 3 \]

\[ (4 \times 2 + 3) + \left( \frac{(10 - 2)}{4} \right) \]
Introducing Binary Trees

• Degree of each node at most 2

• Recursive nature of tree
  • Empty
  • Root with left and right subtrees

• SLL: Recursive nature was captured by hidden node (Node<E>) class

• Binary Tree: No “inner” node class
  • Single BinaryTree class does it all
  • Is it a tree or a node?
    • It’s a node that’s a root of a tree!
  • And it’s not part of Structure hierarchy!
Expression Trees

Build using constructor

```java
new BinaryTree<E>(value, leftSubTree, rightSubTree)
```

```java
BinaryTree<String> fourTimesTwo = new BinaryTree<String>
    ("*", new BinaryTree<String>("4"), new BinaryTree<String>("2"));
```

```java
BinaryTree<String> fourTimesTwoPlusThree = new BinaryTree<String>
    ("+", fourTimesTwo, new BinaryTree<String>("3"));
```
Expression Trees

- General strategy
  - Make a binary tree (BT) for each leaf node
  - Move from bottom to top, creating BTs
  - Eventually reach the root
  - Call “evaluate” on final BT

- Example
  - How do we make a binary expression tree for
    
    \[(4 + 3) \times (10 - 5)/2\]

  - Postfix notation: 4 3 + 10 5 - * 2 /
int evaluate(BinaryTree<String> expr) {

    if (expr.height() == 0)
        return Integer.parseInt(expr.value());

    else {
        int left = evaluate(expr.left());
        int right = evaluate(expr.right());
        String op = expr.value();
        switch (op) {
            case "+" : return left + right;
            case "-" : return left - right;
            case "*" : return left * right;
            case "/" : return left / right;
            Assert.fail("Bad op");
        }
        return -1;
    }
}
Full vs. Complete (non-standard!)

- **Full** tree – A full binary tree of height \( h \) has *leaves only* on level \( h \), and each internal node has exactly 2 children.

- **Complete** tree – A *complete* binary tree of height \( h \) is *full* to height \( h-1 \) and has all leaves at level \( h \) in leftmost locations.

All full trees are complete, but not all complete trees are full!
Implementing BinaryTree

- BinaryTree<E> class
  - Instance variables
    - BinaryTree: parent, left, right
    - E: value
  - left and right are never null
    - If no child, they point to an “empty” tree
      - Empty tree T has value null, parent null, left = right = T
    - Only empty tree nodes have null value
Implementing BinaryTree

- BinaryTree class
  - Instance variables
    - BT parent, BT left, BT right, E value
A small tree

EMPTY != null!
Implementing BinaryTree

- Many (!) methods: See BinaryTree javadoc page
- All “left” methods have equivalent “right” methods
  - public BinaryTree()
    - // generates an empty node (EMPTY)
    - // parent and value are null, left=right=this
  - public BinaryTree(E value)
    - // generates a tree with a non-null value and two empty (EMPTY) subtrees
  - public BinaryTree(E value, BinaryTree<E> left, BinaryTree<E> right)
    - // returns a tree with a non-null value and two subtrees
  - public void setLeft(BinaryTree<E> newLeft)
    - // sets left subtree to newLeft
    - // re-parents newLeft by calling newLeft.setParent(this)
  - protected void setParent(BinaryTree<E> newParent)
    - // sets parent subtree to newParent
    - // called from setLeft and setRight to keep all “links” consistent
Implementing BinaryTree

• Methods:
  • public BinaryTree<E> left()
    • // returns left subtree
  • public BinaryTree<E> parent()
    • // post: returns reference to parent node, or null
  • public boolean isLeftChild()
    • // returns true if this is a left child of parent
  • public E value()
    • // returns value associated with this node
  • public void setValue(E value)
    • // sets the value associated with this node
  • public int size()
    • // returns number of (non-empty) nodes in tree
  • public int height()
    • // returns height of tree rooted at this node
  • But where’s “remove” or “add”?!?!
BT Questions/Proofs

• Prove
  • The number of nodes at depth $n$ is at most $2^n$.
  • The number of nodes in tree of height $n$ is at most $2^{(n+1)}-1$.
  • A tree with $n$ nodes has exactly $n-1$ edges
  • The size() method works correctly
  • The height() method works correctly
  • The isFull() method works correctly