Last Time

- Iterators Recap
- Iterating over Iterators
Today

• Trees!
  • General Idea and Uses
  • Terminology
  • Some examples
    • Expression trees
  • Introduction to 
    structure5 BinaryTree class

• BinaryTree class implementation details
• Proofs and theory
• Traversing trees
Introducing Trees

• Our structures have had a linear organization
  • Stacks, queues
  • Even ordered vectors, ordered lists, arrays, vectors, lists are visualized linearly

• By linear we essentially mean that each element has at most one successor and at most one predecessor…
Branching Out: Trees

• A tree is a data structure where elements can have multiple successors (called children)
• But still only one predecessor (called parent)
Tree Logic (Natalie Jereminjenko) at Mass MoCA
House of Normandy, Battle of Hastings, 1066
Tree Features

• Trees express hierarchical relationships
  • Directed: root to leaf
• Root at the top
• Leaf at the bottom
• Interior nodes in middle
• Parent, 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: maximum depth (across all nodes)
Other Trees

- Phylogenetic tree
- Directories of files
- Game trees
  - Build a tree
  - Search it for moves with high likelihood of winning
- Expression trees
Expression Trees

4 * 2 + 3

(4 * 2 + 3) + ( (10 − 2)/ 4)
Introducing **Binary Trees**

- **Degree** of each node $\leq 2$
- Recursively defined. A tree can either be:
  - Empty
  - Root with left and right subtrees
- SLL: Recursive nature was captured by nodes (Node\(<E>\)) on inside
- Binary Tree: No “inner” node class; single BinaryTree class does it all
- (Not part of the structure hierarchy)
Binary Trees for (Math) Expressions

• 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: \((4*2)+3\)
    • Leaves are numbers
    • Non-leaf nodes are operators
      – We will apply each operator to its children (ex: left + right)
Example: Expression Trees

\[ 4 \times 2 + 3 \]

BinaryTree<String> fourTimesTwo =
    new BinaryTree<String>("\times",
    new BinaryTree<String>("4"),
    new BinaryTree<String>("2"));

BinaryTree<String> fourTimesTwoPlusThree =
    new BinaryTree<String>("+",
    fourTimesTwo,
    new BinaryTree<String>("3"));

Build using constructor
new BinaryTree<E>(value, leftSubTree, rightSubTree)
Evaluating Expression Trees

- Starting at the root,
  - Evaluate left subtree
  - Evaluate right subtree
  - Perform operation (+, -, *, /) with left and right
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;
    }
}
More Tree Terminology

• Some of the terminology is non-standard
• We will try to be consistent in this class, but…
  • We want to be able to communicate to our friends outside of Williams CS too!
• I *hate* jargon, but having a language for our data structures gives us the ability to express ideas and describe algorithms
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!