Administrative Details

- CS Colloquium?!?!
  - Meets (almost) every Friday at 2:30pm
  - Guest speaker presents their research
  - Next Friday (4/20) we will have an information session instead of a normal speaker
    - Discussion of courses offered next semester
    - Advising about majoring in CS
      - We can sign major declaration sheets there
  - Food!
Last Time

• The `structure5 BinaryTree` class
  • implementation details

• Quick proofs and theory
  • Number of nodes at a given depth $d$ is at most $2^d$
  • The total number of nodes in a tree of height $n$ is at most $2^{(n+1)} - 1$
  • A tree with $n$ nodes has exactly $n-1$ edges

• Implications: if a tree is balanced (full or complete), the height is $\log_2 n$
Today

• Traversing trees
  • Pre/post/in/level-order traversals
  • Iterators for each traversal strategy

• Alternative Tree Representations
  • Array-based binary trees
Implementing structure 5 BinaryTree

- BinaryTree\(<E>\) class
  - Instance variables
    - BinaryTree: parent, left, right
    - E: value
  - public BinaryTree()
  - public BinaryTree(E value)
  - public BinaryTree(E value, BinaryTree\(<E>\) left, BinaryTree\(<E>\) right)

```
parent
value
left
right
```

```
null
null
this
this
```

EMPTY BT
Implementing BinaryTree

• Connect BinaryTree nodes:
  • public void setLeft(BinaryTree<E> newLeft)
  • public void setRight(BinaryTree<E> newLeft)
  • protected void setParent(BinaryTree<E> newParent)

• Navigate edges:
  • public BinaryTree<E> left()
  • public BinaryTree<E> right()
  • public BinaryTree<E> parent()

• Interact with BinaryTree data:
  • public E value()
  • public void setValue(E value)

• Get an Iterator:
  • public Iterator<E> iterator()
  • public Iterator<E> preorderIterator()
  • public Iterator<E> postorderIterator()
  • public Iterator<E> levelorderIterator()
Tree Traversals

• In linear structures, there are only a few basic ways to traverse the data structure
  • Start at one end and visit each element
  • Start at the other end and visit each element

• How do we traverse binary trees?
  • (At least) four reasonable mechanisms
Tree Traversals

In-order: “left, node, right”
Terry, Ricky, Mikey, Marky, Danny, Davey

Pre-order: “node, left, right”
Marky, Ricky, Terry, Mikey, Danny, Davey

Post-order: “left, right, node”
Terry, Mikey, Ricky, Davey, Danny, Marky,

Level-order: visit all nodes at depth $i$ before depth $i+1$
Marky, Ricky, Danny, Terry, Mikey, Davey
Tree Traversals

- **Pre-order**
  - Each node is visited before any children. Visit node, then each node in left subtree, then each node in right subtree. (node, left, right)
  - $+*237$

- **In-order**
  - Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree. (left, node, right)
  - $2*3+7$

("pseudocode")
Tree Traversals

- **Post-order**
  - Each node is visited after its children are visited. Visit all nodes in left subtree, then all nodes in right subtree, then node itself. (left, right, node)
  - \(23 \times 7 +\)

- **Level-order** (not obviously recursive!)
  - All nodes of level \(i\) are visited before nodes of level \(i+1\). (visit nodes left to right on each level)
  - \(+ \times 723\)

("pseudocode")
Tree Traversals

public void preOrder(BinaryTree t) {
    if(t.isEmpty())
        return;
    visit(t); // some method
    preOrder(t.left());
    preOrder(t.right());
}

For in-order and post-order: just move visit(t)!

But what about level-order???
Level-Order Traversal
Level-Order Traversal

- Green
  - Blue
  - Violet
    - Orange
    - Yellow
      - Indigo
      - Red

G
Level-Order Traversal

- Green
  - Blue
    - Orange
      - Indigo
    - Yellow
  - Violet
    - Red

G
Level-Order Traversal

G B
Level-Order Traversal

G B V
Level-Order Traversal

G B V O
Level-Order Traversal

G B V O Y
Level-Order Traversal

G B V O Y I
Level-Order Traversal

G B V O Y I R
Level-Order Traversal

• How does level-order work?
  • We visit the nodes level by level, left to right
• Hint: we will use a linear structure…
Level-Order Traversal

Green

Blue

Violet

Orange

Yellow

Indigo

Red
Level-Order Traversal

- Green
  - Blue
  - Violet
    - Orange
    - Yellow
      - Indigo
      - Red

- todo queue
Level-Order Traversal

G
Level-Order Traversal

Green
  Blue
  Violet
    Orange
    Yellow
      Indigo
      Red

G B

todo queue
Level-Order Traversal

G B V
Level-Order Traversal

G B V O
Level-Order Traversal

G B V O Y
Level-Order Traversal

G B V O Y I
Level-Order Traversal

G B V O Y I R
public static <E> void levelOrder(BinaryTree<E> t) {
    if (t.isEmpty()) return;

    // The queue holds nodes for in-order processing
    Queue<BinaryTree<E>> q = new QueueList<BinaryTree<E>>();
    q.enqueue(t); // put root of tree in queue

    while (!q.isEmpty()) {
        BinaryTree<E> next = q.dequeue();
        visit(next);
        if (!next.left().isEmpty()) q.enqueue(next.left());
        if (!next.right().isEmpty()) q.enqueue(next.right());
    }
}
Iterators

• Provide iterators that implement the different tree traversal algorithms

• Methods provided by BinaryTree class:
  • preorderIterator()
  • inorderIterator()
  • postorderIterator()
  • levelorderIterator()
Implementing the Iterators

- **Basic idea**
  - Should return elements in same order as corresponding traversal method shown
  - Recursive methods don’t convert as easily: must phrase in terms of `next()` and `hasNext()`
    - Similar to how we implemented `SkipIterator`: do some prep work before returning from `next()`
  - So, let’s start with `levelOrder`!
Level-Order Iterator

```java
public BTLevelorderIterator(BinaryTree<E> root) {
    todo = new QueueList<BinaryTree<E>>();
    this.root = root; // needed for reset
    reset();
}

public void reset() {
    todo.clear();
    // empty queue, add root
    if (!root.isEmpty()) todo.enqueue(root);
}
```
public boolean hasNext() {
    return !todo.isEmpty();
}

public E next() {
    BinaryTree<E> current = todo.dequeue();
    E result = current.value();
    if (!current.left().isEmpty())
        todo.enqueue(current.left());
    if (!current.right().isEmpty())
        todo.enqueue(current.right());
    return result;
}
Pre-Order Iterator

• Basic idea
  • Should return elements in same order as processed by pre-order traversal method
  • Must phrase in terms of next() and hasNext()
  • We “simulate recursion” with stack
    • The stack holds “partially processed” nodes
Pre-Order Iterator

• Outline: node - left tree – right tree
  1. Constructor: Push root onto todo stack
  2. On call to next():
     • Pop node from stack
     • Push right and then left children of popped node onto stack
     • Return node’s value
  3. On call to hasNext():
     • return !stack.isEmpty()
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

- Green
  - Blue
  - Violet
    - Orange
    - Yellow
      - Indigo
      - Red
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

Green

Blue

Violet

Orange Yellow

Indigo Red

todo stack
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

```
Green
   Blue
     Violet
        Orange
             Yellow
                    Indigo
                           Red

Blue
   Violet
todo stack
```

G
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

G B
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

G B V
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

G B V O
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

G B V O I

Red
Yellow
todo stack
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

G B V O I R
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

```
  Green
   /    /
  Blue  Violet
   /   /    /
Orange Yellow Indigo Red
```

todo stack

G B V O I R Y
Pre-Order Iterator

```java
public BTPreorderIterator(BinaryTree<E> root) {
    todo = new StackList<BinaryTree<E>>();
    this.root = root;
    reset();
}

public void reset() {
    todo.clear(); // stack is empty; push on root
    if (!root.isEmpty()) todo.push(root);
}
```
public boolean hasNext() {
    return !todo.isEmpty();
}

public E next() {
    BinaryTree<E> old = todo.pop();
    E result = old.value();
    if (!old.right().isEmpty())
        todo.push(old.right());
    if (!old.left().isEmpty())
        todo.push(old.left());
    return result;
}
Tree Traversal (Practice) Problems

• Prove that levelOrder() is correct: that is, that it touches the nodes of the tree in the correct order (Hint: induction by level)
• Prove that levelOrder() takes O(n) time, where n is the size of the tree
• Prove that the PreOrder (LevelOrder) Iterator visits the nodes in the same order as the PreOrder (LevelOrder) traversal method
In-Order Iterator

- **Outline:** left - node - right
  1. Push left children (as far as possible) onto stack
  2. On call to `next()`:
     - Pop node from stack
     - Push right child and follow left children as far as possible
     - Return node’s value
  3. On call to `hasNext()`:
     - return `!stack.isEmpty()`
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.

B G I
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.

B G I O R V Y
Post-Order Iterator

• Left as an exercise…
Alternative Tree Representations

- Total # “slots” = 4n
  - Since each BinaryTree maintains a reference to left, right, parent, value
- 2-4x more overhead than vector, SLL, array, ...
- But trees capture successor and predecessor relationships that other data structures don’t...
Array-Based Binary Trees

- Encode structure of tree in array indexes
  - Put root at index 0
- Where are children of node i?
  - Children of node i are at $2i+1$ and $2i+2$
  - Look at example
- Where is parent of node j?
  - Parent of node j is at $(j-1)/2$
ArrayTree Tradeoffs

• Why are ArrayTrees good?
  • Save space for links
  • No need for additional memory allocated/garbage collected
  • Works well for full or complete trees
    • Complete: All levels except last are full and all gaps are at right
    • “A complete binary tree of height \( h \) is a full binary tree with 0 or more of the rightmost leaves of level \( h \) removed”

• Why bad?
  • Could waste a lot of space
  • Tree of height of \( n \) requires \( 2^{n+1} - 1 \) array slots even if only \( O(n) \) elements