Announcements

- Prelim 1 is Tonight, bring your student ID
  - 5:30PM EXAM
    - OLH155: netids starting aa to dh
    - OLH255: netids starting di to ji
    - PHL101: netids starting jj to ks (Plus students who switched from the 7:30 exam)

- 7:30PM EXAM (314 Students)
  - OLH155: netids starting kt to rz
  - OLH255: netids starting sa to wl
  - PHL101: netids starting wm to zz (Plus students who switched from the 5:30 exam)
Comparing Data Structures

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>add(val x)</th>
<th>lookup(int i)</th>
<th>search(val x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><img src="2130" alt="Array Diagram" /></td>
<td>2 1 3 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linked List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><img src="2130" alt="Linked List Diagram" /></td>
<td>2 1 3 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Tree</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><img src="123" alt="Binary Tree Diagram" /></td>
<td>1 2 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>$O(\text{height})$</td>
<td>$O(\text{height})$</td>
<td>$O(\text{height})$</td>
</tr>
<tr>
<td><img src="123" alt="BST Diagram" /></td>
<td>1 2 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Binary Search Trees
Red-Black Trees

- Self-balancing BST
- Each node has one extra bit of information "color"
- Constraints on how nodes can be colored enforces approximate balance
Red-Black Trees

1) A red-black tree is a binary search tree.
2) Every node is either red or black.
3) The root is black.
4) If a node is red, then its (non-null) children are black.
5) For each node, every path to a descendant null node contains the same number of black nodes.
Which of the following are red-black trees?

A) YES
B) NO
C) YES
D) NO
class RBNode<T> {
    private T value;
    private Color color;
    private RBNode<T> parent;
    private RBNode<T> left, right;

    /** Constructor: one-node tree with value x */
    public RBNode (T v, Color c) { value= d; color= c; }

    ...
}

Null if the node is the root of the tree.
Either might be null if the subtree is empty.
Insert(RBTree t, int v){
  Node p;
  Node n = t.root;
  while(n != null){
    p = n;
    if(v < n.value){n = n.left}
    else{n = n.right}
  }
  Node vnode = new Node(v, RED)
  if(p == NULL){
    t.root = vnode;
  } else if(v < p.value){
    p.left = vnode; vnode.parent = p;
  } else{
    p.right = vnode; vnode.parent = p;
  }
  fixTree(t, vnode);
}
Case 1: parent is black
Case 2: parent is red uncle is black node on outside
Case 3: parent is red uncle is black node on inside
Case 4: parent is red uncle is red
Rotations

leftRotate

rightRotate
```c
fixTree(RBTree t, RBNode n) {
    while (n.parent.color == RED) { // not Case 1
        if (n.parent.parent.right == n.parent) {
            Node uncle = n.parent.parent.left;
            if (uncle.color == BLACK) { // Case 2 or 3
                if (n.parent.left == n) {
                    rightRotate(n);
                } //3
                n.parent.color = BLACK;
                n.parent.parent.color = RED;
                leftRotate(n.parent.parent);
            } else { //uncle.color == RED // Case 4
                n.parent.color = BLACK;
                uncle.color = BLACK;
                n.parent.parent.color = RED;
                n = n.parent.parent;
            }
        } else {...} // n.parent.parent.left == n.parent
    }
    t.root.color = BLACK; // fix root
}
```
Red-black trees are a special case of binary search trees

Search works exactly the same as in any BST

Time: $O(\text{height})$
What is the max height?

- Observation 1: Every binary tree must have a null node with depth \( \leq \log(n + 1) \)
What is the max height?

- Observation 1: Every binary tree must have a null node with depth $\leq \log(n + 1)$
What is the max height?

- Observation 1: Every binary tree must have a null node with depth $\leq \log(n + 1)$

- Observation 2: In a red-black tree, the number of red nodes in a path from the root to a null node is less than or equal to the number of black nodes.
What is the max height?

- **Observation 1:** Every binary tree must have a null node with depth $\leq \log(n + 1)$
- **Observation 2:** In a red-black tree, the number of red nodes in a path from the root to a null node is less than or equal to the number of black nodes.
- **Observation 3:** The maximum path length from the root to a null node is at most 2 times the minimum path length from the root to a null node.
What is the max height?

- Observation 1: Every binary tree must have a null node with depth $\leq \log(n + 1)$

- Observation 2: In a red-black tree, the number of red nodes in a path from the root to a null node is less than or equal to the number of black nodes.

- Observation 3: The maximum path length from the root to a null node is at most 2 times the minimum path length from the root to a null node.

$$h = \max_{root \rightarrow null} \text{path len} \leq 2 \cdot \min_{root \rightarrow null} \text{path len} \leq 2 \log(n + 1)$$

$h$ is $O(\log n)$
## Comparing Data Structures

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>add(val x)</th>
<th>lookup(int i)</th>
<th>search(val x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array [2, 1, 3, 0]</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Linked List [2, 1, 3, 0]</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Binary Tree [1, 2, 3]</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>BST [1, 2, 3]</td>
<td>$O(\text{height})$</td>
<td>$O(\text{height})$</td>
<td>$O(\text{height})$</td>
</tr>
<tr>
<td>RB Tree [1, 2, 3]</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
Most languages (natural and computer) have a recursive, hierarchical structure.

This structure is implicit in ordinary textual representation.

Recursive structure can be made explicit by representing sentences in the language as trees: Abstract Syntax Trees (ASTs).

ASTs are easier to optimize, generate code from, etc. than textual representation.

A parser converts textual representations to AST.
Applications of Trees: Syntax Trees

A Java expression as a string:

```
2 * 1 - (1 + 0)
```

An expression as a tree:

```
2 * 1 - (1 + 0)
```

"parsing"
Pre-order, Post-order, and In-order

Pre-order traversal:
1. Visit the root
2. Visit the left subtree (in pre-order)
3. Visit the right subtree

$\begin{array}{c}
\ast \\
2 & 1 \\
\end{array} \quad \begin{array}{c}
- \\
1 & 1 \\
\end{array} \quad \begin{array}{c}
+ \\
\quad 0 \\
\end{array}$

$- \ast 2 1 + 1 0$
Pre-order, Post-order, and In-order

Pre-order traversal

Post-order traversal
1. Visit the left subtree (in post-order)
2. Visit the right subtree
3. Visit the root

23
Pre-order, Post-order, and In-order

Pre-order traversal
Post-order traversal
Post-order traversal

In-order traversal
1. Visit the left subtree (in-order)
2. Visit the root
3. Visit the right subtree

```
- * 2 1 + 1 0
2 1 * 1 0 + -
2 * 1 - 1 + 0
```
Pre-order, Post-order, and In-order

To avoid ambiguity, add parentheses around subtrees that contain operators.

Pre-order traversal

Post-order traversal

In-order traversal

-  \(*\) 2 1 + 1 0

2 1 * 1 0 + -

(2 * 1) - (1 + 0)
Because of ordering rules for a BST, it’s easy to print the items in alphabetical order

- Recursively print left subtree
- Print the node
- Recursively print right subtree

```java
/** Print BST t in alpha order */
private static void print(TreeNode<T> t) {
    if (t == null) return;
    print(t.left);
    System.out.print(t.value);
    print(t.right);
}
```
In Defense of Postfix Notation

- Execute expressions in postfix notation by reading from left to right.
- Numbers: push onto the stack.
- Operators: pop the operands off the stack, do the operation, and push the result onto the stack.

2 1 * 1 0 + -
In Defense of Postfix Notation

- Execute expressions in postfix notation by reading from left to right.
- Numbers: push onto the stack.
- Operators: pop the operands off the stack, do the operation, and push the result onto the stack.

1 * 1 0 + -

2
In Defense of Postfix Notation

- Execute expressions in postfix notation by reading from left to right.
- Numbers: push onto the stack.
- Operators: pop the operands off the stack, do the operation, and push the result onto the stack.

\[ * \ 1 \ 0 \ + \ - \]

1
2
In Defense of Postfix Notation

- Execute expressions in postfix notation by reading from left to right.
- Numbers: push onto the stack.
- Operators: pop the operands off the stack, do the operation, and push the result onto the stack.

1 0 + -

2
In Defense of Postfix Notation

- Execute expressions in postfix notation by reading from left to right.
- Numbers: push onto the stack.
- Operators: pop the operands off the stack, do the operation, and push the result onto the stack.
In Defense of Postfix Notation

- Execute expressions in postfix notation by reading from left to right.
- Numbers: push onto the stack.
- Operators: pop the operands off the stack, do the operation, and push the result onto the stack.
In Defense of Postfix Notation

- Execute expressions in postfix notation by reading from left to right.
- Numbers: push onto the stack.
- Operators: pop the operands off the stack, do the operation, and push the result onto the stack.
In Defense of Postfix Notation

- Execute expressions in postfix notation by reading from left to right.
- Numbers: push onto the stack.
- Operators: pop the operands off the stack, do the operation, and push the result onto the stack.

In about 1974, Gries paid $300 for an HP calculator, which had some memory and used postfix notation! Still works.

a.k.a. “reverse Polish notation”
In Defense of Prefix Notation

- Function calls in most programming languages use prefix notation: like `add(37, 5)`.
- Some languages (Lisp, Scheme, Racket) use prefix notation for everything to make the syntax simpler.

```
(define (fib n)
  (if (<= n 2)
      1
      (+ (fib (- n 1) (fib (- n 2)))))
```
There's a pair of Java interfaces designed to make data structures easy to traverse.

You could modify a tree to implement iterable, implement an (in-order, post-order, etc.) iterator and then use a for each loop to traverse the tree!

In recitation this week, you will modify your linked list from A3 to implement iterable.