Comparing Data Structures

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>add(val x)</th>
<th>lookup(val 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>Linked List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Binary Tree</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>BST</td>
<td>$O(\text{height})$</td>
<td>$O(\text{height})$</td>
<td>$O(\text{height})$</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 descendent null node contains the same number of black nodes.

Announcements

- Prelim 1 is Tonight, bring your student ID
  - 5:30PM EXAM
  - OLH155: netids starting aa to dh
  - OLH255: netids starting di to jj (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)
RB Tree Quiz

- Which of the following are red-black trees?

A) YES  B) NO  C) YES  D) NO

Class for a RBNode

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 = v; color = c;
    }

    // Null if the node is the root of the tree.
    // Either might be null if the subtree is empty.

    // Either might be null if the subtree is empty.

    // Case 1: parent is black
    // Case 2: parent is red uncle is black
    // Case 3: parent is red uncle is black
    // Case 4: parent is red uncle is red

    // Rotations

    leftRotate;
    rightRotate;

    // Fixing the tree

    fixTree(RBTree t, RBNode n)

    case 1: Case 2: Case 3: Case 4:

    parent is black
    parent is red
    parent is red
    parent is red

    uncle is black
    uncle is black
    uncle is black
    uncle is red

    Case 3:
    parent is red
    uncle is black
    uncle is red

    // Rotations

    leftRotate;
    rightRotate;

    // Fixing the tree

    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
                n.parent.color = BLACK;
                n.parent.parent.color = RED;
                leftRotate(n.parent.parent);
            } else if(uncle.color == RED) { // Case 4
                n.parent.color = BLACK;
                uncle.color = BLACK;
                n.parent.parent.color = RED;
                n = n.parent.parent;
            } else { // Case 4
                n.parent.color = RED;
                uncle.color = BLACK;
                n.parent.parent.color = RED;
                n = n.parent.parent;
            }
        }
    }

    t.root.color = BLACK; // fix root

    }
Search

- 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 $\leq \log(n + 1)$
- 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.

<table>
<thead>
<tr>
<th>n</th>
<th>$\log(n+1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.584</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2.321</td>
</tr>
<tr>
<td>5</td>
<td>2.584</td>
</tr>
<tr>
<td>6</td>
<td>2.807</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>3.169</td>
</tr>
<tr>
<td>9</td>
<td>3.221</td>
</tr>
<tr>
<td>10</td>
<td>3.249</td>
</tr>
</tbody>
</table>

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

$h$ 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</td>
<td>(O(n))</td>
<td>(O(1))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Linked List</td>
<td>(O(1))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Binary Tree</td>
<td>(O(1))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>BST</td>
<td>(O(\text{height}))</td>
<td>(O(\text{height}))</td>
<td>(O(\text{height}))</td>
</tr>
<tr>
<td>RB Tree</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
</tr>
</tbody>
</table>

Application of Trees: Syntax Trees

- 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 ASTs.

Applications of Trees: Syntax Trees

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

A Java expression as a string.

```
2 1 * 1 + 0
```

An expression as a tree.

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
- **Post-order traversal**: 1. Visit the left subtree (in post-order) 2. Visit the right subtree 3. Visit the root
- **In-order traversal**: 1. Visit the left subtree (in-order) 2. Visit the root 3. Visit the right subtree
Pre-order, Post-order, and In-order

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

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

Printing contents of BST (In-Order Traversal)

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

/** 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.

1 * 10 + - 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.

* 10 + - 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.

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))))
```

Iterator/Iterable

- 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