CompSci 201, L21: Balanced Binary Search Trees
Logistics, Coming up

• APT9 (more tree problems) due this Wed., 11/9

• P5 Huffman due next Monday 11/14

• APT10 (greedy) next Wed., 11/16
APT Quiz 2

• Will release after APT10 is due, likely next Thursday 11/17 or Friday morning 11/18.

• Due by any time Monday 11/21.
  • Last and only thing due week of Thanksgiving break

• Logistics: 2 hours, 3 problems, adjusted scale, like the first quiz.

• Content: APT sets 6-10. LinkedList and Trees for sure.
Binary Search Tree Invariant

A binary tree is a binary **search** tree if *for every node*:

- Left subtree values are all less than the node’s value

**AND**

- Right subtree values are all greater than the node’s value

According to some ordering (comparable or comparator)

Enables efficient search, similar to binary search!
Use BST to make a DIY version of a TreeSet

```java
1 public class TreeNode {
2     int info;
3     TreeNode left;
4     TreeNode right;
5     TreeNode(int x){
6         info = x;
7     }
8     TreeNode(int x, TreeNode lNode, TreeNode rNode){
9         info = x;
10        left = lNode;
11        right = rNode;
12     }
13 }
```

Remember that a TreeSet stores values with no duplicates in sorted order as a binary search tree.
Support **add, contains, sorted order**

```java
74  public static void main(String[] args) {
75      BST mySet = new BST();
76      mySet.add(8); mySet.add(4); mySet.add(12);
77      mySet.add(6); mySet.add(10); mySet.add(15);
78      mySet.add(15); mySet.add(15);
79
80      System.out.println(mySet.contains(6));  true
81      System.out.println(mySet.contains(7));  false
82
83      System.out.println(mySet);
84     } [4, 6, 8, 10, 12, 15]
```

Note lack of duplicate 15’s
add wrapper

```java
11   public boolean add(int x) {
12       if (root == null) {
13           root = new TreeNode(x);
14           return true;
15       }
16       return insert(x, root);
17   }
```

Just initializes root when first node is added, otherwise calls a helper method insert.
Recursive **insert**, pictures

- 8
- 4
  - Is < 8
- 12
  - Is > 8
- 6
  - Is < 8
  - And > 4
- 10
  - Is > 8
  - And < 12
- 15
  - Is > 8
  - And > 12

- If less, insert or recurse left
- If greater, insert or recurse right
- If equal? Duplicate, do nothing
Recursive `insert` code

```java
private boolean insert(int x, TreeNode t) {
    if (t.info == x) {
        return false;
    }
    if (x < t.info) {
        if (t.left == null) {
            t.left = new TreeNode(x);
            return true;
        }
        return insert(x, t.left);
    }
    if (t.right == null) {
        t.right = new TreeNode(x);
        return true;
    }
    return insert(x, t.right);
}
```
contains wrapper

```java
19 public boolean contains(int x) {
20     return search(x, root);
21 }
```

Just initializes a recursive search at the root.
Recursive search

```java
private boolean search(int x, TreeNode t) {
    if (t == null) {
        return false;
    }
    if (t.info == x) {
        return true;
    }
    if (x < t.info) {
        return search(x, t.left);
    }
    return search(x, t.right);
}
```
Runtime complexity of BST add/contains on balanced tree

```java
29 private boolean search(int x, TreeNode t) {
30     if (t == null) {
31         return false;
32     }
33     if (t.info == x) {
34         return true;
35     }
36     if (x < t.info) {
37         return search(x, t.left);
38     }
39     return search(x, t.right);
40 }
```

Completely balanced tree:
- \( T(N) = T(N/2) + O(1) \)
- Solution is \( O(\log(N)) \), same as binary search
Runtime performance of BST on perfectly unbalanced tree

```java
private boolean search(int x, TreeNode t) {
    if (t == null) {
        return false;
    }
    if (t.info == x) {
        return true;
    }
    if (x < t.info) {
        return search(x, t.left);
    }
    return search(x, t.right);
}
```

Perfectly unbalanced tree:
- $T(N) = T(N-1) + O(1)$
- Solution is $O(N)$, search in linked list
Experiment: How much difference does it make empirically to do 100,000 random searches?

Timings in milliseconds
See example code in coursework.cs.duke.edu/cs-201-fall-22/diytreeset

<table>
<thead>
<tr>
<th>N</th>
<th>sorted order BST</th>
<th>random order BST</th>
<th>java.util TreeSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>370</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>2,000</td>
<td>715</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>4,000</td>
<td>1422</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>8,000</td>
<td>2905</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>16,000</td>
<td>5991</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>32,000</td>
<td>Runtime exception</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>64,000</td>
<td>...</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1,000,000</td>
<td>...</td>
<td>15</td>
<td>24</td>
</tr>
</tbody>
</table>
Red-Black Tree

Red-Black Trees are **binary search trees** that satisfy the following properties:

1. Every node is red or black,
2. The root is black,
3. A red node cannot have red children, and
4. From a given node, all paths to null descendants must have the same number of black nodes.

null is considered to be a black node
Red-Black Trees in `java.util`

**Class TreeMap<K,V>**

```java
java.lang.Object
to java.util.AbstractMap<K,V>
to java.util.TreeMap<K,V>
```

**Type Parameters:**
- K - the type of keys maintained by this map
- V - the type of mapped values

**All Implemented Interfaces:**
- `Serializable`, `Cloneable`, `Map<K,V>`, `NavigableMap<K,V>`

```
public class TreeMap<K,V>
extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serial
```

A Red-Black tree based `NavigableMap` implementation.
More red-black trees in `java.util`

**Class TreeSet<E>**

```
java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractSet<E>
            java.util.TreeSet<E>
```

**Type Parameters:**
E - the type of elements maintained by this set

**All Implemented Interfaces:***
`Serializable, Cloneable, Iterable<E>, Collection<E>

```
public class TreeSet<E>
extends AbstractSet<E>
implements NavigableSet<E>, Cloneable, Serializ

A NavigableSet implementation based on a TreeMap.
```
A “family” tree connection

public class TreeMap<K, V>
extends AbstractMap<K, V>
implements NavigableMap<K, V>, Cloneable, Serializable

A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.

My (doctoral) adviser’s adviser’s adviser (we don’t know each other)
Understanding Red-Black Tree Properties

- Root node is red.
- NOT a valid red-black tree.

- Root node is black.
- No red nodes with red children.
- Path (22,null) has 2 black nodes, but path (22,11,null) has 3.
- NOT a valid red-black tree.

- Root node is black.
- Red node has no red children.
- All paths from a node to null leaves have the same number of black nodes.
  - All paths from 22 to null leaves have 2 black nodes.
  - All paths from 11 to null leaves have 1 black node.
- Tree is a valid red-black tree.

Trick: Not a binary search tree at all!

Reference: ZyBook 21
Not all binary search trees can be colored as red-black trees.
red-black tree properties guarantee approximate balance

• Note that the runtime complexity of add/contains (a.k.a. insert and search) in a binary search tree is proportional to the height of the tree.

• **Claim.** Any red-black tree with N nodes has height that is $O(\log(N))$. 
Proof sketch (not going to sweat the details)

1. At least half of the nodes on any root to leaf path are black (because red nodes cannot have red children).

2. All root to leaf paths have the same number of black nodes (property 4)

3. 1+2 imply that all root to leaf paths have within a factor 2 of the same number of nodes.
Proof sketch (not going to sweat the details)

4. Worst case for $h = \text{height}$: longest root to leaf path is twice as long as other root to leaf paths.

5. 4 Implies worst case tree is complete for $\sim h/2$ levels.

6. Implies number of nodes is $N > 2^{h/2}$.

7. Rearranging, $h < 2 \log_2(N)$

More precise proof yields $h \leq 2 \log_2(N+1)$
WOTO

Go to duke.is/64tcz

Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!
How do Red-Black Trees Work

Remember, red black trees are also binary search trees (BST).

- **contains/search** – Exact same as BST, no change!
- **add/insert** – Two steps:
  1. Run regular BST add/insert
  2. Color the new node red
  3. Fix the tree to reestablish red-black tree properties
public class RBTreeNode {
    int info;
    RBTreeNode parent;
    RBTreeNode left;
    RBTreeNode right;
    boolean red;
}

RBTreeNode(int x, boolean red) {...
RBTreeNode(int x, RBTreeNode parent, boolean red) {...
RBTreeNode(int x, RBTreeNode lNode, RBTreeNode rNode, RBTreeNode parent, boolean red) {
    info = x;
    left = lNode;
    right = rNode;
    this.parent = parent;
    this.red = red;
}
Search is the same

```java
private boolean search(int x, RBTreeNode t) {
    if (t == null) {
        return false;
    }
    if (t.info == x) {
        return true;
    }
    if (x < t.info) {
        return search(x, t.left);
    }
    return search(x, t.right);
}
```
Insert looks the same...

```java
private boolean insert(int x, RBTTreeNode t) {
    if (t.info == x) {
        return false;
    }
    if (x < t.info) {
        if (t.left == null) {
            t.left = new RBTTreeNode(x, true);
            RBTTreeBalance(t.left);
            return true;
        }
        return insert(x, t.left);
    }
    if (t.right == null) {
        t.right = new RBTTreeNode(x, true);
        RBTTreeBalance(t.right);
        return true;
    }
    return insert(x, t.right);
}
```

Except for the root, always add new nodes as red initially.

Need to re-balance (reestablish red-black tree properties) after insertion.
Terminology: Getting to know the family “Tree”

- Node (N)
- Sibling (S)
- Parent (P)
- Grandparent (G)
- Uncle (U)
Recoloring

• We always insert a new node as red, see the 5 node here.
  • (This way never violates the black nodes on paths property)

• Violates RBT property: red child of red parent.

• Fix by recoloring?
Recoloring

If parent and uncle of new node are both red, color both black and color grandparent red.

Before Recoloring:

- Node 4 (P) is red
- Node 8 (G) is black
- Node 12 (U) is red
- Node 15 (N) is red

After Recoloring:

- Node 4 (P) becomes black
- Node 8 (G) becomes red
- Node 12 (U) becomes black
- Node 15 (N) becomes black
Can’t fix all problems by recoloring

• Suppose we just inserted 5 here.

• “Looks” like we could just recolor...
  • Set 4 red, 6 black?
Sometimes need to rotate

• Looks good, but...

• Violating path property now. Need to rotate: actually change the tree structure.
Left Rotation

Not a valid red-black tree

Valid red-black tree
Right Rotation

Before:

```
        80
       /  
     61   92
    /    /  
   40   77   
  / 
16 43
```

After:

```
        61
       /  
     40   80
    /    /  
   16   43   77 92
```

Algorithm:

```
RBTTreeRotateRight(tree, tree->root)
```
Case Analysis

• Full rebalance algorithm proceeds by cases:
  • Cases vary by color and position of node, parent, grandparent, uncle.
  • Deal with cases by recoloring, left rotations, and right rotations.

• Remove has case analysis as well.

• Want the details? See the ZyBook (or CLRS Intro to Algorithms for the standard reference).