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

• This Wednesday, April 5
  • APT 8 due

• Next Monday, April 10
  • Project P5: Huffman due

• Next Wednesday, April 12
  • APT Quiz 2 due:
    • Covers linked list, sorting, trees
    • No regular APTs this week, just the quiz
Reminder: What is an APT Quiz?

• Set of 3 APT problems, 2 hours to complete.
  • Will be available starting this Saturday afternoon (look for a Sakai/email announcement)
  • Must complete by 11:59 pm Wednesday 4/12 (so start before 10)

• Start the quiz on Sakai assessments tool, begins your timer and shows you the link to the problems and submission page.
  • Will look/work just like the regular APT page, just with only 3 problems.
Reminder: What is allowed?

Yes, allowed
• Zybook
• Course notes
• API documentation
• VS Code
• JShell

No, not allowed
• Collaboration or sharing any code.
• Communication about the problems at all during the window.
• Searching internet, stackoverflow, etc. for solutions.
Reminder: Don’t do these things

1. Do not collaborate. Note that we log all code submissions and will investigate for academic integrity.

2. Do not hard code the test cases (if(input == X) return Y, etc.).
   We show you the test cases to help you debug. But we search for submissions that do this and you will get a 0 on the APT quiz if you hard code the test cases instead of solving the problem.
Reminder: How is it graded?

Not curved, adjusted. 3 problems, 10 points each.

<table>
<thead>
<tr>
<th>Raw score R out of 30.</th>
<th>Adjusted score A out of 30.</th>
<th>100 point grade scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>27 &lt;= R &lt;= 30</td>
<td>A = R</td>
<td>90 – 100</td>
</tr>
<tr>
<td>24 &lt;= R &lt;= 26</td>
<td>A = 26</td>
<td>~87</td>
</tr>
<tr>
<td>21 &lt;= R &lt;= 23</td>
<td>A = 25</td>
<td>~83</td>
</tr>
<tr>
<td>18 &lt;= R &lt;= 20</td>
<td>A = 24</td>
<td>80</td>
</tr>
<tr>
<td>15 &lt;= R &lt;= 17</td>
<td>A = 23</td>
<td>~77</td>
</tr>
<tr>
<td>12 &lt;= R &lt;= 14</td>
<td>A = 22</td>
<td>~73</td>
</tr>
<tr>
<td>9 &lt;= R &lt;= 11</td>
<td>A = 21</td>
<td>70</td>
</tr>
<tr>
<td>6 &lt;= R &lt;= 8</td>
<td>A = 20</td>
<td>~67</td>
</tr>
<tr>
<td>3 &lt;= R &lt;= 5</td>
<td>A = 19</td>
<td>~63</td>
</tr>
<tr>
<td>1 &lt;= R &lt;= 2</td>
<td>A = 18</td>
<td>60</td>
</tr>
</tbody>
</table>

Can still get in the B range even if you can’t solve one; don’t panic!

Only going to get a 0 if you collaborate or hard code test cases. Don’t do it!
Binary Heap Wrapup

Reminder: You can see a simple DIY implementation of a binary heap-based priority queue at coursework.cs.duke.edu/cs-201-spring-23/diybinaryheap
decreaseKey Operation?

• Suppose we decrease the 13 to 5.
• Violates heap property
• Fix like in the add operation:
  While violating heap property:
  • Swap with parent
decreaseKey NOT in java.util

• decreaseKey is important for some algorithms, but not supported in many standard libraries (including the java.util PriorityQueue)

• Why not?
  • Note that binary heap does not support efficient search
  • In order to do decreaseKey in $O(\log(n))$ time, need to store references/indices of all the “nodes.”
  • Adds overhead, not done in java.util
Alternative Implementation: Binary Search Tree

• If your keys happen to be unique...
• Can support $O(\log(n))$ add & remove (smallest) using a binary search tree!
• Smallest is leftmost child
PriorityQueue (with unique keys) using a java.util TreeSet

import java.util.TreeSet;

public class BSTPQ<T extends Comparable<T>> {
    private TreeSet<T> bst;

    public BSTPQ() { bst = new TreeSet<>(); }
    public void add(T element) { bst.add(element); }
    public int size() { return bst.size(); }
    public T peek() { return bst.first(); }

    public T remove() {
        T returnValue = bst.first();
        bst.remove(returnValue);
        return returnValue;
    }

    public void decreaseKey(T oldKey, T newKey) {
        bst.remove(oldKey);
        bst.add(newKey);
    }

    first gives smallest element in TreeSet in O(log(n)) time
    Can decreaseKey by removing and then re-adding, both O(log(n)) time for a TreeSet
Disadvantages to using a Binary Search Tree for your priority queue?

1. All elements must be unique

2. Not array-based, uses more memory and has higher constant factors on runtime

3. Much harder to implement with guarantees that the tree will be balanced.
Binary Search Tree
Review and Runtime

See videos of live coding a DIYTreeSet as a binary search tree:

Part 1: Getting started, traversal, iterator
Part 2: add and contains

And here is the code: coursework.cs.duke.edu/cs-201-spring-23/diytreeset
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!
Recursive search, pictures, pseudocode

boolean search(int x, TreeNode t) {
    • If t == null: Return false
    • If x == t.info: Return true
    • If x < t.info: search left
    • Else: search right

Searching for 10

10 > 8, so search right
Recursive **search**, pictures, pseudocode

```java
boolean search(int x, TreeNode t) {
    • If t == null: Return false
    • If x == t.info: Return true
    • If x < t.info: search left
    • Else: search right
```
Recursive search, pictures, pseudocode

```java
boolean search(int x, TreeNode t) {
    // If t is null, return false
    if (t == null) return false;
    // If x is equal to t.info, return true
    if (x == t.info) return true;
    // If x is less than t.info, search the left subtree
    else if (x < t.info) search(t.left);
    // Otherwise, search the right subtree
    else search(t.right);
    // Since 10 is equal to 10, return true
    return true;
}
```

10 == 10 so return true
Recursive search code

```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
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);
}
```

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
Another perspective: Balanced BST has height $O(\log(n))$

$n = 2^0 + 2^1 + \cdots + 2^h$
$= 2^{h+1} - 1$
$\rightarrow h = \log_2(n + 1) - 1$

Results in $O(\log(n))$ best case performance.

Geometric series formula:
$$\sum_{i=0}^{k-1} ar^i = \frac{a(1 - r^k)}{1 - r}$$
Another perspective: Unbalanced BST has $O(n)$ height

For example, results from:
Sort(values)
For each e in values:
insert(e)

Results in $O(n)$ worst case performance.
**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 DIY binary search tree</th>
<th>random order DIY binary search tree</th>
<th>sorted order 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>
Average Case: Random Binary Search Tree has $O(\log(n))$ expected height

• Given $x_1, \ldots, x_n$ unique keys
• Let $\sigma(x_1, \ldots, x_n)$ be a uniform random permutation

• Theorem 12.4 CLRS (restated):
$$\mathbb{E}_\sigma[h_n] \leq \log_2 \left( \frac{n^3 + 6n^2 + 11n + 6}{24} \right) \text{ is } O(\log(n)).$$
Stronger statements about random binary search trees

• At most
  \[ h_n \to_{n \to \infty} 4.3 \log_2(n) \text{ with high probability} \]
  

• Empirical performance.  
  Note that for \( n = 1 \) million:
  
  • \( 2\log_2(n) \approx 40 \)
  
  • \( 3\log_2(n) \approx 60 \)
Red-Black Tree: A Balanced Binary Search Tree
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.
Red-Black Trees in `java.util`

**Class TreeMap<K,V>**

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

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

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>`

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

A `NavigableSet` implementation based on a `TreeMap`. This allows for efficient searching, insertion, and deletion of elements based on their natural ordering.
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

- Too many black nodes on right compared to left paths
- Too many black nodes on right compared to left paths
- Too many black nodes on right compared to left paths
- Red node with red child not allowed
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.
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
RBTreeNode

```java
class RBTreeNode {
    int info;
    RBTreeNode parent;
    RBTreeNode left;
    RBTreeNode right;
    boolean red;
}
```

Just like a regular TreeNode except:
- Store parent reference
- Store color
Search is the same

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

RBTTreeRotateRight(tree, tree → root)

4/3/23

Compsci 201, Spring 2023, L21: Balanced Binary Search Trees
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).