Binary Search Trees (Continued)

• Study Project 3 Solution
• Balanced Binary Search Trees
• Balancing Operations
• Reading: L&C 11.1 – 11.4
Study Project 3 Solution

- Project 3 was due before class today
- Discuss solution
Balanced Binary Search Trees

• The balance of a binary search tree is important for obtaining its efficiency
• If we add 3, 5, 9, 12, 18, and 20 to a binary search tree, we get a degenerate tree
• This is less efficient than a singly linked list because our code needs to check the null left pointer at each level while traversing it
• Operations are $O(n)$ instead of $O(\log n)$
• We want our binary search trees balanced
Balanced Binary Search Trees

• Degenerate tree for a binary search tree
Balancing Operations

• Brute force balancing methods work but are unnecessarily time consuming
• We could use an in-order traversal of the tree and move everything out to an array
• Then, we could use a recursive method to insert the middle element of the array as the root and subdivide the array into two halves
• Eventually, we will rebuild a balanced tree
Balancing Operations

- We prefer to use balancing operations after each add or remove element operation
- Semantics of balancing operations
  - Right rotation
  - Left rotation
  - Rightleft rotation
  - Leftright rotation
Balancing Operations

• Semantics of Right Rotation
  A. Make the left child of the root the new root
  B. Make former root the right child of the new root
  C. Make right child of the former left child of the former root the new left child of the former root
Balancing Operations

- **Semantics of Left Rotation**
  A. Make the right child of the root the new root
  B. Make former root the left child of the new root
  C. Make left child of the former right child of the former root the new right child of the former root

```
Initial Tree

   5
  /   \
3     10
   /  \
3    13
   /
 7

Step A

   10
  /   \
3     13
   /  \
3    15
   /
 7

Step B

   5
  /  \
3    15
   /
 7

Step C

   5
  /  \
3    13
   /
 7
```
Balancing Operations

• Semantics of Rightleft Rotation
  A. Right rotation around right child of root
  B. Left rotation around root

Initial Tree

After Right Rotation

After Left Rotation
Balancing Operations

- Semantics of LeftRight Rotation
  A. Left rotation around left child of root
  B. Right rotation around root

Initial Tree

After Left Rotation

After Right Rotation
Introduction to Project 4

• Study use of these Java Collections APIs
  – HashMap<K,V>
  – Map.Entry<K,V>
  – PriorityQueue<T>

• Study the concept of Huffman coding based on character frequency
  – A tree with the most frequent characters in leaves closer to the root
  – Following left child adds a 0 to the code
  – Following right child adds a 1 to the code
Sample Huffman Coding Tree

Symbol/Frequency | Binary Code Sequence (Read right to left)
--- | ---
E / 0.40 | 0
T / 0.35 | 10
I / 0.20 | 110
Others / 0.05 | 111

An E which occurs 40% of the time takes only 1 binary digit – Zero
A T which occurs 35% of the time takes only 2 binary digits – One and Zero
An I which occurs 20% of the time takes only 2 binary digits – One and One
All other characters which only occur 5% of the time take 3 or more digits
UML Diagram for Project 4

HuffmanTree
- encodeTable : HashMap
- decodeTable : HashMap
- compressionRatio : double

+ main(String[] args) : void
+ HuffmanTree (message : String)
+ encode (message : String) : String
+ decode (message : String) : String
+ getCompressionRatio (void) : double

java.util.HashMap
java.util.Set
java.util.Map.Entry
java.util.PriorityQueue

<<Interface>>
Comparable<HuffmanNode>

HuffmanNode
- value : String
- frequency : int
- zero : HuffmanNode
- one : HuffmanNode

+ HuffmanNode (message : String)
+ compareTo (that : HuffmanNode) : int
+ toString (void) : String
+ getValue (void) : String
+ getFrequency (void) : int
+ increment (void) : void
+ getZero (void) : HuffmanNode
+ getOne (void) : HuffmanNode