Lab 8 - Graph DS & Traversal

CS2040S Data Structures & Algorithms

AY20/21 Semester 1

Week 10

One-Day Assignment 6 – Kattis's Quest

- Given a list of quests and a given energy level
  - Calculate the amount of gold gained in that play session
- N <= 10^5 commands
  - Add a quest to the list
  - Simulate with energy X

With starting energy X:

1. Find largest energy quest, with energy cost C <= current X
   - If tie, pick largest gold reward G.
2. Clear quest
   - Remove it from quest board permanently
   - Decrease X by C.
   - Add G gold.
3. Repeat until not enough energy

One-Day Assignment 6 – Kattis's Quest

1. add E G
   - Add quest with energy E and gold reward G to pool
2. query X
   - Enter with energy X
   - Repeatedly clear quests (as per strategy)
   - Print out total gold earned

Questions

- How to store list of quests?
  - How to handle quests with same E and G values?
- How to find quest with largest energy <= X?
  - (E=4, G=5) < (4,6) < (5,1)
  - How to find maximum possible reward?
  - Reward at most 10^5

Kattis's Quest - Operations

- What kind of data structure do we want to store the quests in?
- What operations do we need to do quickly (ideally O(1) or O(log n)?)
    1. Add a quest (energy E, gold G)
    2. Repeatedly:
        - Find and extract quest with
            a. max energy <= X
            b. max gold.

Storing Quests

- Assume for now, all quests have different (E,G)
- We want to find “max energy under some upper bound”
  - IDEA: TreeSet.floor?
  - TreeSet<Integer> storing energy values?
    - But max reward is important too

TreeSet

- Store a TreeSet<Quest>
  - Quest is (energy, gold), comparing energy then gold.
- TreeSet.floor(***Z***) will give give largest quest ≤ ***Z***, in O(log n)
  - Energy ≤ ***Z***'s energy
  - If equal energy, gold ≤ ***Z***'s gold
    - But we want maximum gold, with no limit!

Upper Bounding

- Maybe ***Z*** = (remaining energy, "*infinity*" gold)
  - "If equal energy, highest gold at most ***Z***'s gold"

    ⇒ If equal energy, highest gold ≤ "*infinity*"

    ⇒ If equal energy, highest gold
  - Exactly what we need!

Upper Bounding

- What should *“infinity”* be?
  - int cannot store ∞
  - Instead, store large enough value ≥ any possible reward
    - Integer.MAX_VALUE
    - 10^5 (i.e. maximum possible reward)
- We will see this trick again in Graph SSSP

Duplicates

- Current assumption: No quests with same (E,G)
  - Not guaranteed to hold!
  - Need to handle possibility of multiple *copies* of (E,G)
  - TreeSet only stores 1 single copy
  - If we remove 1 copy out of 2, the other should still be available

Solutions

1. Store a unique ID field, and use it to tiebreak equal (E,G)
    - Quest is { energy E, gold G, id K }

Use current loop index i as ID.

- Instead of TreeSet<Quest>

2. Use TreeMap<Quest, Integer>
    - (E,G) -> Number of copies

Need to create if adding to nonexistent
Need to remove if removing from 1 left

3. Use TreeMap<Integer, PriorityQueue<Integer>>
    - E -> { Max-PriorityQueue of G values for this E }
    - PriorityQueue allows duplicates

O(log n) TreeMap.floor
O(log n) PriorityQueue.poll

Kattis's Quest - Operations

- What kind of data structure do we want to store the quests in?
- What operations do we need to do quickly (ideally O(1) or O(log n)?)
  1. Add a quest (energy E, gold G) O(log n)
  2. Repeatedly:
     - Find and extract quest with O(log n) per quest
       a. max energy <= X
       b. max gold.

Each quest enters/exits our DS at most once, so O(log n) per quest * n quests
Total is O(n log n) time

Graph DS

- Java do not have API that supports graphs directly
  - Instead, you need manually code graph data structures out using what you have learnt the previous weeks

Graph DS

- Define an edge to contain 3 different pieces of information:
  - w - Weight of each edge. Edge has weight of 1 if it is an unweighted graph.
  - u - Vertex which the edge is pointing from.
  - v - Vertex which the edge is pointing to.
- We also use V and E to refer to total number of vertices and edges in a graph G

Graph DS - Adjacency Matrix

- Declare an adjacency matrix to represent graphs:
  - `int[][] adjMatrix = new int[V][V];`
- Store an edge:
  - `adjMatrix[u][v] = w;`
- Entry in the array with value 0 is ***usually*** assumed to indicate that edge (u, v) is not present in graph

Graph DS - Adjacency List (Unweighted Graph)

- Declare an adjacency list to represent unweighted graphs:
  - ```
    ArrayList<ArrayList<Integer>> adjList =
    new ArrayList<ArrayList<Integer>>();
    ```
- Store an edge:
  - ```
    adjList.get(u).add(v);
    ```

Graph DS - Adjacency List (Weighted Graph)

- Declare an adjacency list to represent weighted graphs:
  - ```
    ArrayList<ArrayList<IntegerPair>> adjList =
    new ArrayList<ArrayList<IntegerPair>>();
    ```
- Store an edge:
  - ```
    adjList.get(u).add(new IntegerPair(v, w));
    ```

Graph DS - Adjacency List

- Unlike 2D arrays, 2D ArrayLists require you to manually add the second layer of ArrayLists

```
ArrayList<ArrayList<Integer>> adjList =
        new ArrayList<ArrayList<Integer>>();
for (int i = 0; i < V; i ++) {
    adjList.add(new ArrayList<Integer>());
}
```

- Ensure that you are adding a **new** `ArrayList` per call, NOT adding the same `ArrayList` multiple times

Graph DS - Edge List (Unweighted Graph)

- Declare an edge list to represent unweighted graphs:
  - `ArrayList<IntegerPair> edgeList = new ArrayList<IntegerPair>();`
- Store an edge:
  - `edgeList.add(new IntegerPair(u, v));`

Graph DS - Edge List (Weighted Graph)

- Declare an edge list to represent weighted graphs:
  - `ArrayList<IntegerTriple> edgeList = new ArrayList<IntegerTriple>();`
- Store an edge:
  - `edgeList.add(new IntegerTriple(u, v, w));`

Graph Traversal - BFS

- Breadth-First Search (BFS)
  - Tends to employ a queue
  - Need to mark a vertex as visited when added to the queue instead of when removed from the queue
    - Prevent enqueuing the same vertex multiple times

Graph Traversal - DFS

- Depth-First Search (DFS)
  - Tends to employ a implicit stack via recursion
  - May result in running out of computer memory that OS provides
    - Can use iterative DFS using an explicit stack instead

One-Day Assignment 7 - Weak Vertices

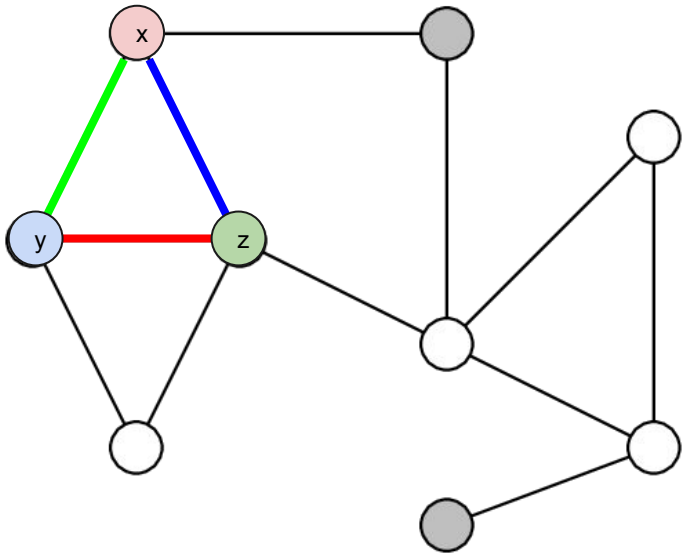
- Given T (≤ 100) undirected unweighted graphs
  - Each graph is given in adjacency matrix form
    - N ≤ 20 vertices per graph
  - Print out all “weak” vertices in ascending order
    - Vertex not part of any triangle => “weak”

How to find all triangles in a graph?

Back-of-envelope estimate:
100M ops / 100 graphs = 1M ops/graph
V = 20, so V^4 ≈ 160K fits fine

Triangles

- (x,y,z) is triangle in graph
  - ⇔ (x,y), (y,z), (x,z) exist
  - ⇔ We have the length-3 cycle
    - $x \to y \to z \to x$
- For any vertex x,
  - How to find if some triangle contains x?
  - What kind of graph DS operations do we want?
  - How fast can we answer that?



Assignment Guidelines

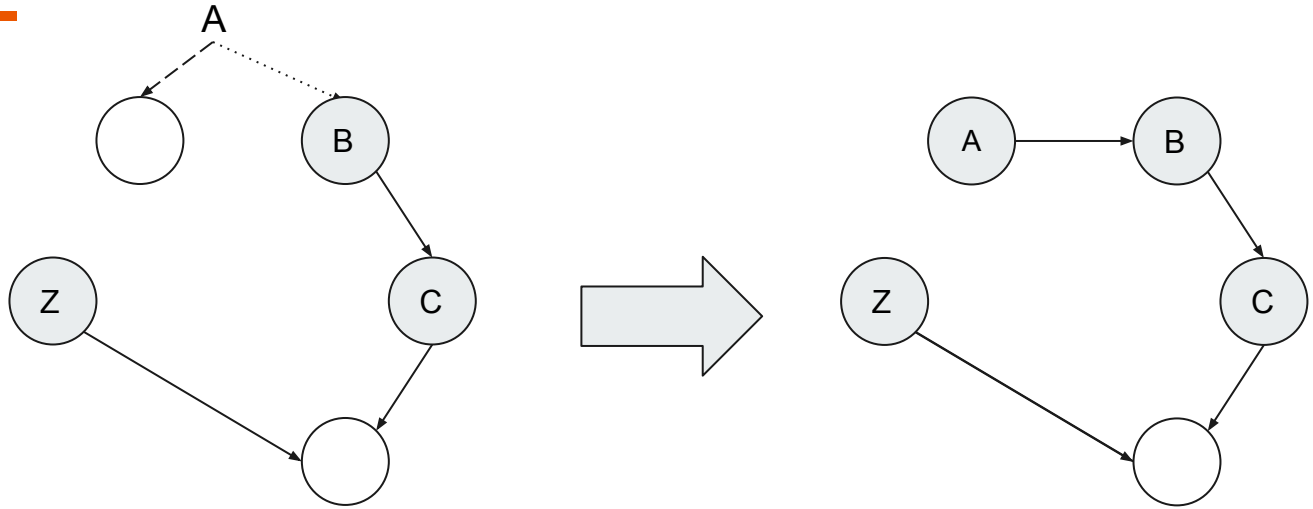
- Include your **name** and **student number** in comments at the top of your code.
- You are allowed (and encouraged) to discuss algorithms
  - List down all your collaborators in your source code
- **You are NOT allowed to:**
  - **Copy another person's code**
  - **Look at another person's code**
  - **Use another person's code as a base for your own code**
- Plagiarism checks will be in place

Take-Home Lab 3 - Ladice

- N items, L drawers
  - Each item has 2 allowed positions A_i, B_i
- Try to insert each item
  - If successful, print LADICA
  - If discarded, print SMECE

1. Try position A_i.
2. Try position B_i.
3. Try repeatedly pushing the item already at A_i, to its other positions, until a free space is reached.
   If we loop, continue to next rule.
4. Try repeatedly pushing the item already at B_i, to its other positions, until a free space is reached.
   If we loop, continue to next rule.
5. Discard the item.

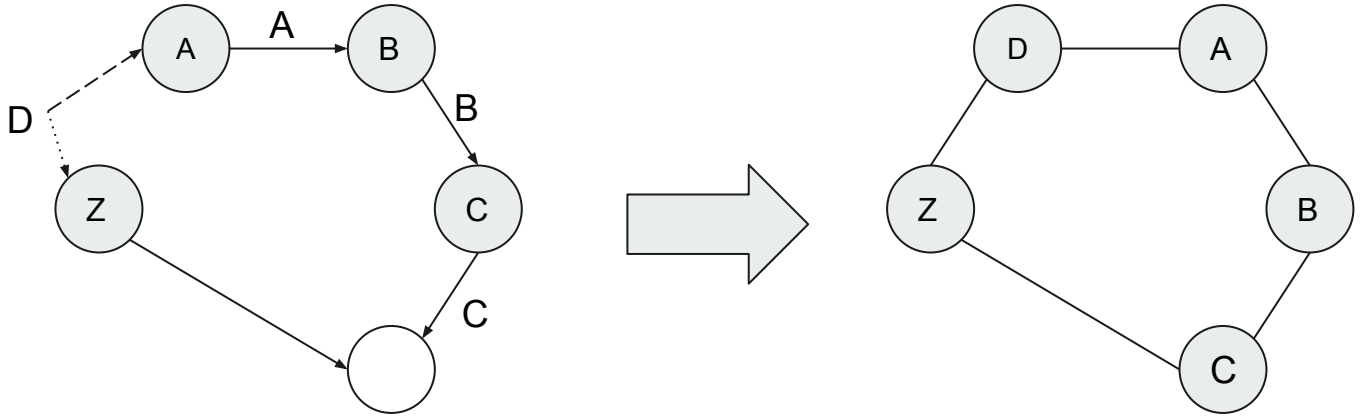
Pushing Stuff Back



Can we track where we can “push” items to make space?

Not full yet, still have empty drawer
Can still push stuff

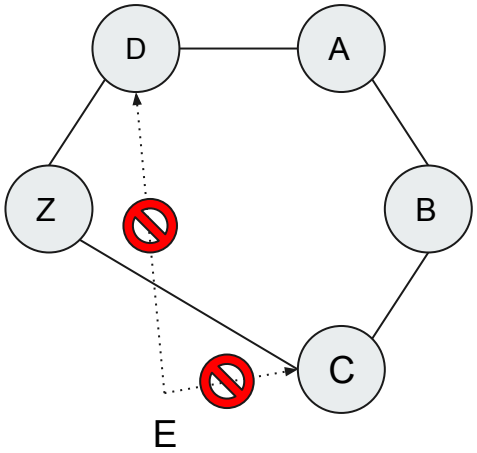
Pushing Stuff Back, Redux



Not full yet, still have empty drawer
Can still push stuff

Full, we can't push forward/backward any of these items.

Pushing Stuff But Failing



When is a “bunch” of drawers **full**?
(i.e. cannot push to make space?)

Here, drawers holding {A,B,C,D,Z} are full.

NEW: In hindsight, these look like a graph...

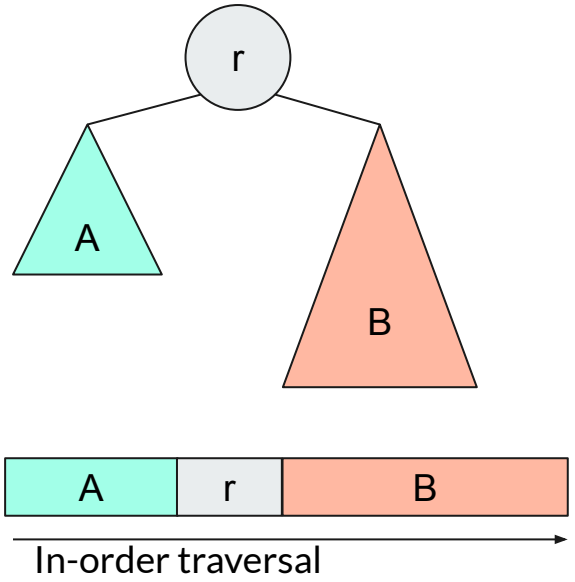
Take-Home Lab 3 - Factor-Free Tree

- A **factor-free** tree is a binary tree, with natural numbers at each node
  - Any node value is coprime to all of its ancestor node values
- Given an **in-order** traversal of this binary tree
  - Reconstruct a valid factor-free tree
  - Does not have to be the original, any valid is OK

Factor-free trees are made just for this problem, they are not a 2040S DS.

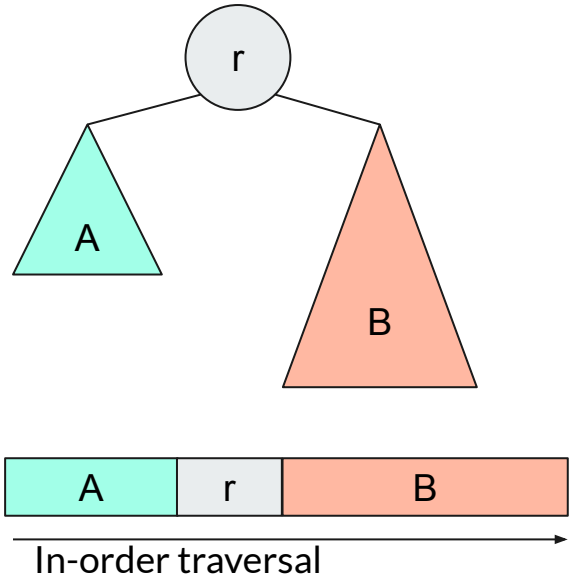
Tree Construction

- Property of in-order traversal:
  - [left subtree] (root) [right subtree]
- Maybe we can borrow a page from the perfect binary tree construction.
  - Find an *appropriate* root (position *r*)
  - Split and recurse on left/right halves
    - Produce left/right subtrees



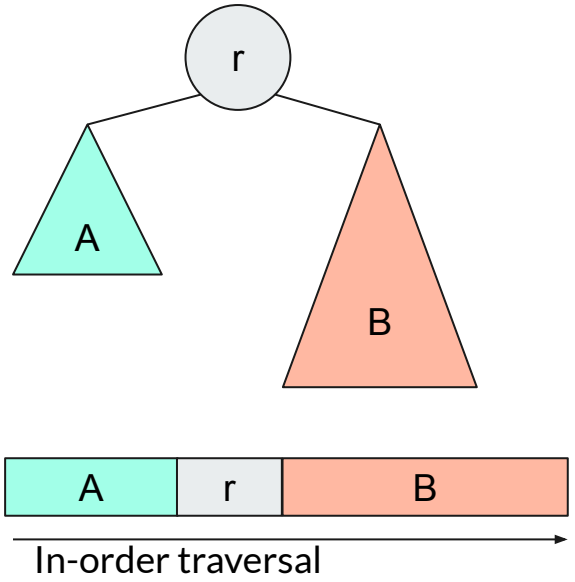
Tree Construction

- Can we find **the original** root (if any)?
- Can we find **a possible** root?
  - Root has to be coprime with all its descendants



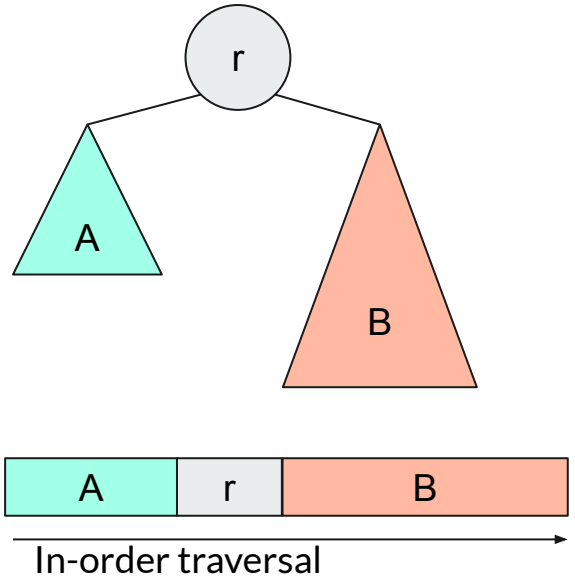
Tree Construction

- Work => Finding the correct root
- What's our recurrence relation?
  - O(1) work + *uneven* split => O(n) total
    - Very good!
  - O(n) work + even split => O(n log n) total
    - Not too bad
  - O(n) work + *uneven* split => $O(n^2)$ total
    - How did we end up here?
    - Can we avoid this case?



Tree Construction

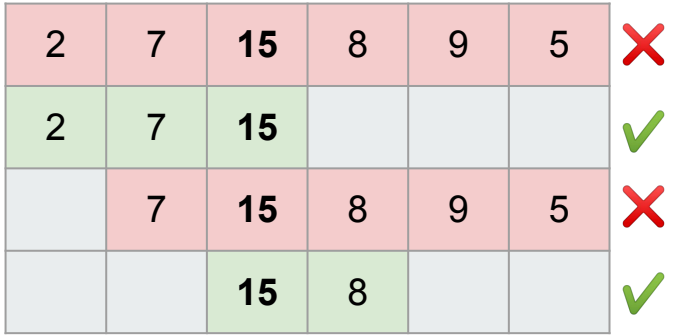
- If we pick a different root than the original tree, can we end up failing later?
  - Prove! Or just assume for now.
    - What does it mean to *fail*?
    - If it fails, what did the original tree do right? Is this possible?



Tree Construction

- How do we quickly find if an element is a possible root?
- Each time we recurse on some subarray [l,r]
  - “Rootness” of element may change!
  - Why can it change?
  - ***How*** exactly does it change?
- Is there a “summary” for each element:
  - Given subarray bounds [l,r]
  - Can this element can be root?

Can 15 be a root?



Next improvement:
How to compute this “summary” quickly?
What are the relevant factors affecting 15’s “summary”?

Coprimality

- How to test if two numbers are coprime?
  - GCD, O(log M) for numbers at most M.
- Given N numbers, how to test pairwise coprimality?
  - GCD(a,b) != 1 iff some prime p is a common divisor

Primes

- How to prime factorize single number?
  - Trial division: O(sqrt(M)) per factor
- How to prime factorize many numbers (value at most M)?
  - IDEA: For any number K ≥ 1
    - Shrink it by dividing by **smallest prime factor**
  - Now: Compute smallest prime factors for a lot of numbers?
  - Have we seen something similar in CS1231?

Take-Home Lab 3 - Baby Names [Optional]

- Update and query a database of baby names, with gender suitability
- Q <= **500,000** queries
    0. Exit program.
    1. Given name and gender suitability, add suggestion
    2. Given name and gender suitability, remove suggestion
    3. Given [start] and [end], and suitability
        - Print number of names satisfying { [start] <= name < [end] }

FOCUS ON PREVIOUS PROBLEMS/OTHER MODS FIRST

This problem is much harder, but offers decent practice in implementing DSes.

Query

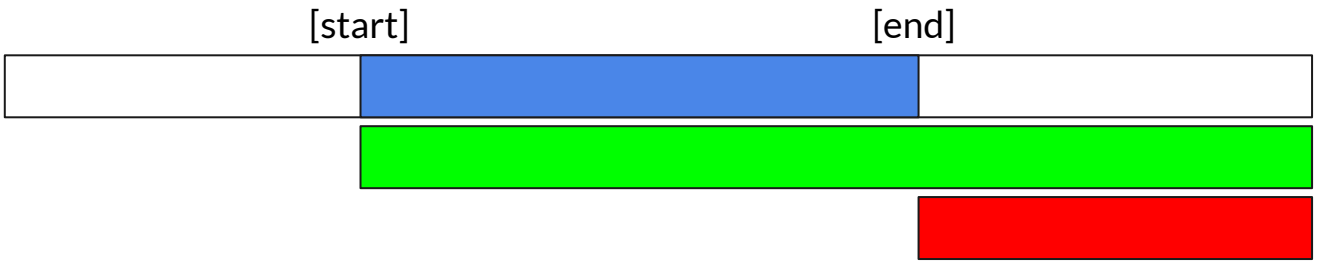
- Kattis sample input gives you 1-letter queries ‘A’ and ‘Z’
  - **Not true in general!**

followed by two strings: *start*, *end*,

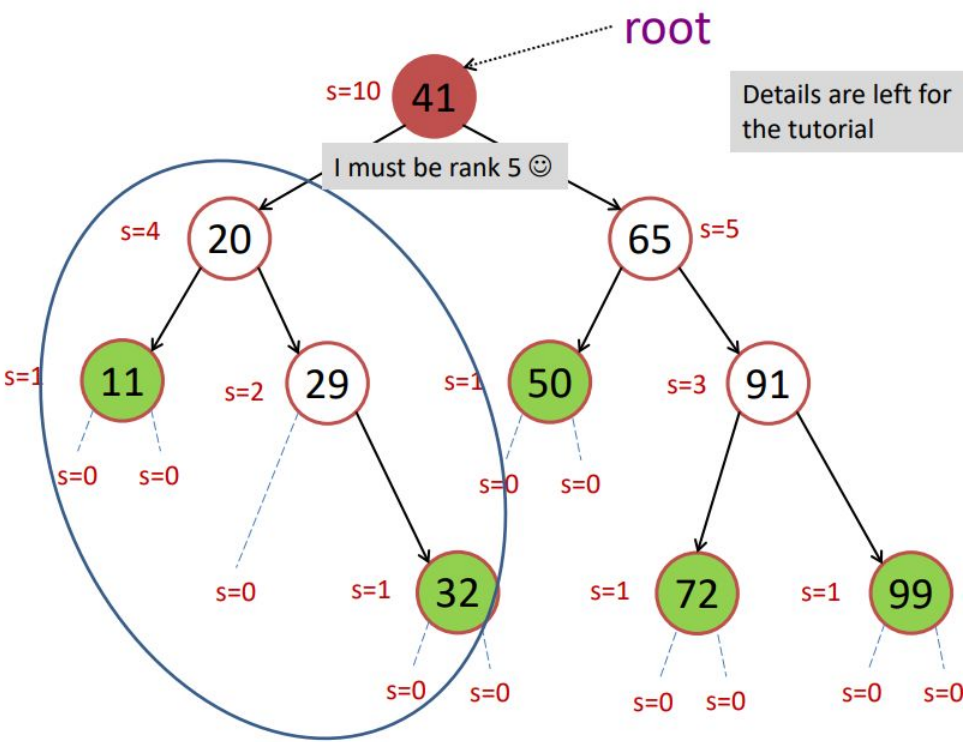
    - Only guarantee is at most 30 letters.
  - We can ask: In {“Andy”, “Brian”, “Charlie”}
    - How many names are (“Ada” <= name < “Anna”)?
    - Answer: 1 (“Andy”)

Counting

- We want to answer queries of the form
  - Size of { [start] <= name < [end] }
- We can rewrite these as:
  - Size of { [start] <= name } - Size of { [end] <= name }



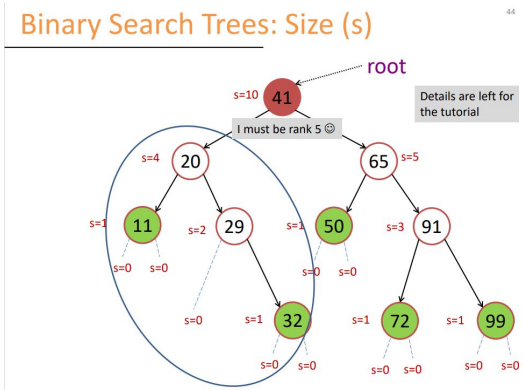
Binary Search Trees: Size (s)



Since this image grew much bigger than last time, this is probably an important hint.

Counting

- If [start] is “EFG”, already went into “E” node
  - How do we **count** things like “E**G**A”, but not “E**E**A”?
  - Count whole of everything after E**F**:
    - EG(...), EH(...), …
  - Part of EF(...) should count
    - How to count part?
  - Do not count anything before E**F**:
    - EE(...), ED(...), ...



Performance Caveats!

#	NAME	SCORE	RUNTIME	DATE	ID
1	Matthew Ng	100	0.65 s	2019-10-15 09:05:36	4775481@other site
2	*Hidden user*	100	0.68 s	2020-09-10 17:02:44	6065896
3	Andrew Godbout	100	0.68 s	2020-09-10 17:07:01	6065926
4	Ryan Chew	100	0.69 s	2020-08-07 06:42:27	5924988
5	Chow Yuan Bin	100	0.86 s	2020-09-23 15:31:49	6140048
6	Enzio Kam Hai Hong	100	0.87 s	2020-10-09 03:42:06	6243256
7	Steven Halim	100	0.91 s	2019-10-12 02:24:25	4753870@other site
8	Lim Daekoon	100	0.99 s	2020-03-15 10:17:20	5466844

- Most likely to just scrape past 0.99s
- Question is highly specific to particular trie implementation
- 5×10^5 queries, storing 2×10^5 names
    - A lot of things to read in/write out **(hint hint)**

Trie

- Java API does not contain a trie class
- Some suggestions:
  - Avoid implementing trie “optimizations” for longer strings
    - E.g. path compression, qp-trie, etc.
    - Names are at most 30 characters
  - Take advantage of small radix
    - Names consist of only uppercase characters A-Z