Lab 7 - TreeSet / TreeMap

CS2040S Data Structures & Algorithms

AY20/21 Semester 1

Week 9

Week 8 One-day - Assigning Workstations

- There are workstations that lock itself after `m` minutes of inactivity
- There are `n` researchers that want to use these workstations
  - Arrives after `a` minutes and stays for `s` minutes
- Our protagonist is lazy, and wants to minimize the number of unlocks
  - “Lets not lock the workstation and let the next researcher take over!”
- How many “unlocks” can she save by doing this?

Week 8 One-day - Assigning Workstations

- Suboptimal solution

Time (inclusive)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
		Researcher 1											Researcher 4								
						Researcher 2				Researcher 3											

Week 8 One-day - Assigning Workstations

- Optimal Solution

Time (inclusive)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
		Researcher 1								Researcher 3											
						Researcher 2							Researcher 4								

Week 8 One-day - Assigning Workstations

- The problem demonstrates the Greedy Property
  - When there are more than one unlocked workstations available, assign researcher to the workstation that has minimum amount of time left
  - For this to work, you must process researchers in chronological order

Insight #1

- Easier to work with researchers coming in chronological order
  - Input order may not be chronological! Sort first
  - If we don’t work in chrono order
    - New researchers coming in may disrupt the allocation

Attempt #1

- Store a Set/List of workstations (available time & lock time)
- For each researcher in chronological order
  - For each workstation in list 

    - If current time > lock time
      - Fully locked already, no need to track separately.
      - Remove from set.
    - If current time ∈ [available time, lock time]
      - Tentatively assign to researcher, or overwrite if older than current candidate
    - If current time < available time
      - Still in use, keep for later.

Insight #2

We want to operate on the **oldest** unlocked workstation

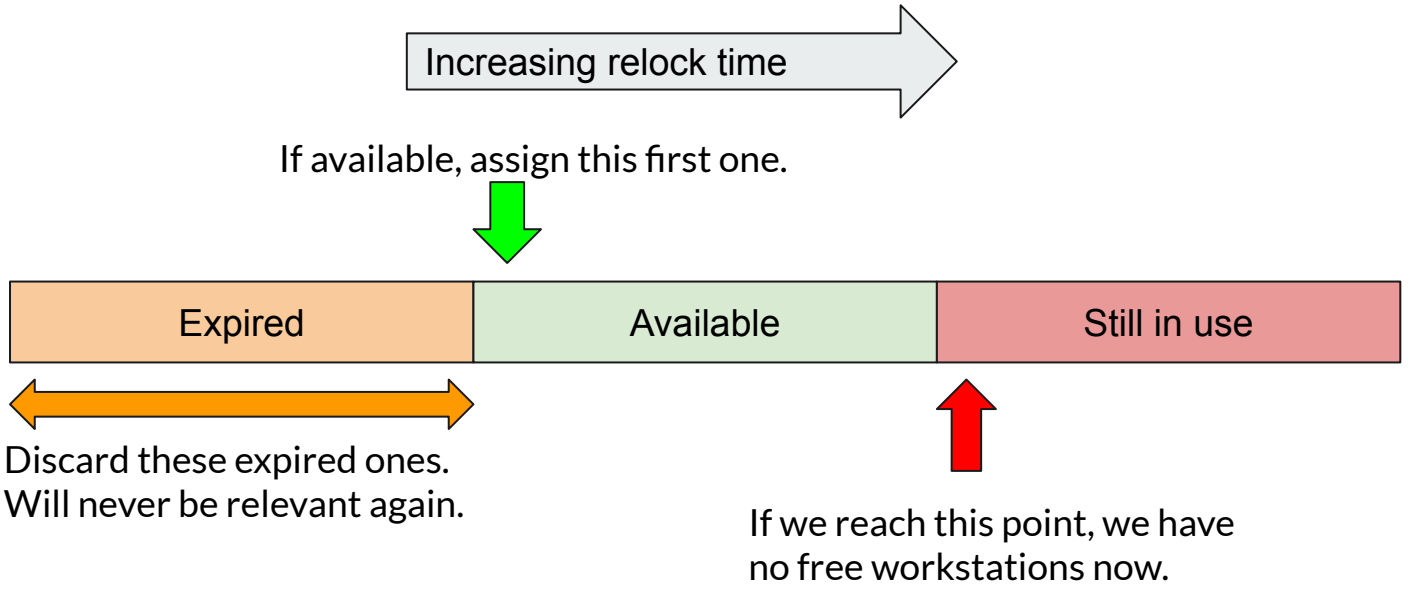
- If not expired, assign
- If still in use, or no workstations
    - Assign fresh one

Keep **oldest** unlocked workstation accessible at all times.

Priority Queue

- Keep track of the states of workstations
  - Only the ones that are unlocked/in use
    - Top element should be the **oldest unlocked**
  - Make use of priority queue, storing:
    - Time when workstation becomes locked again OR
    - Time when workstation becomes unused

Priority Queue Ordering



Attempt #2

- Store a PQ of workstations (available time & lock time)
- For each researcher in chronological order
  - Fast-forward to current time. Iterate through whole PQ:
    - Eagerly remove() any expired workstations
  - Look at the topmost workstation (if any)
    - If current time ∈ [available time, lock time]
      - Assign to researcher and break
    - If current time < available time, or if empty
      - **All remaining workstations (if any),** have greater/equal available time.
      - No point trying further, assign fresh machine and break

PQ.remove() is O(n) per item!

Eager Deletion

- Calling PQ's remove() method is slow
  - Does not exploit heap property
- Why are we even deleting elements?
  - Expired workstations may contaminate result from peek()
  - **IDEA:** Delete only when we see the "dead" item in peek()
    - Cheaper, O(log n) poll vs O(n) remove

Lazy Deletion

- Only way to get data from PQ, is via peek()
- IDEA: If peek() gives a “dead” element (i.e. stale data)
  - Poll() it away, and try again.
  - Continue on only with “live” element, or empty queue
- **Does not care** where the element is when marking as “dead”
  - Just happens that in this problem, all at top of PQ

Attempt #3

- Store a PQ of workstations (sorted by available time)
- For each researcher in chronological order
  - Peek at top workstation (earliest unlock time)
    - If current time > lock time
      - Fully locked already, no need to track separately.
      - Pop from PQ, retry again. 

    - If current time ∈ [available time, lock time]
      - Assign to researcher, and break
    - If current time < available time, or PQ is empty
      - **All remaining workstations (if any),** have greater/equal available time.
      - No point trying further, assign fresh machine and break

Slow?

- For a single researcher, we may delete **many** expired workstations!
  - O(n log n) worst case, per researcher
  - Possibly $O(n^2 \log n)$ over the whole program?
- Majority of cost is PQ poll()ing
- But overall, how many times do we call poll()?
  - How many items go into the PQ?

Lazy Deletion - Amortization

- How many elements enter/exit the PQ?
  - We add exactly 1 computer per researcher
    - (Treating reused computers as a new object)
    - N computers total
  - Each object we add, can be poll()'d only once
    - N add()s, N poll()s, over the **whole program execution**.
    - "Averaged"/Amortized to O(log N) per researcher
- We will see this technique again for Graph SSSP (Dijkstra)

Covered in 3230: Aggregate method of amortization

Lazy Deletion - Amortization

- Another way of thinking about lazy deletion costs
  - When we insert the item into the PQ
    - Pay the O(log n) insertion cost for this item
    - “Prepay” the O(log n) deletion cost for this item
  - When we poll() this item from the top
    - “Redeem” the O(log n) prepaid credits
    - Poll()ing does not “cost” anything

Covered in 3230: Accounting method of amortization

Week 8 One-day - Assigning Workstations

- For every researcher, check the first element in the priority queue
- 4 cases:
    1. Priority queue is empty
    2. Workstation is still in use
    3. Workstation is unlocked
    4. Workstation is already locked

Week 8 One-day - Assigning Workstations

Case 1: Priority Queue is Empty

- Always occurs for first researcher
- Unlock new workstation for this researcher
    - Add a new entry into the PQ

Week 8 One-day - Assigning Workstations

Workstation locks after 5 minutes of inactivity

Priority Queue

WS1
Unlocked until: 13

Researcher 1 enters at time 3, uses for 5 minutes

Queue empty! Unlock new workstation

Week 8 One-day - Assigning Workstations

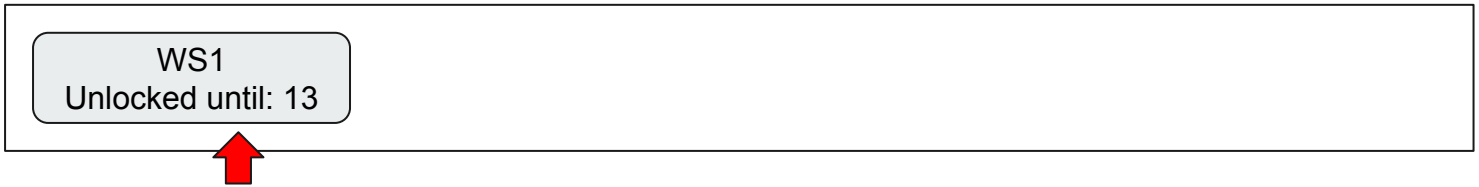
Case 2: Workstation is still in use

- Occurs if the first workstation in PQ is still in use
- Unlock new workstation for this researcher
    - Why?
        - The earliest unlocked workstation is still in use when current researcher comes in
        - No other workstation in PQ will be available as well

Week 8 One-day - Assigning Workstations

Workstation locks after 5 minutes of inactivity

Priority Queue



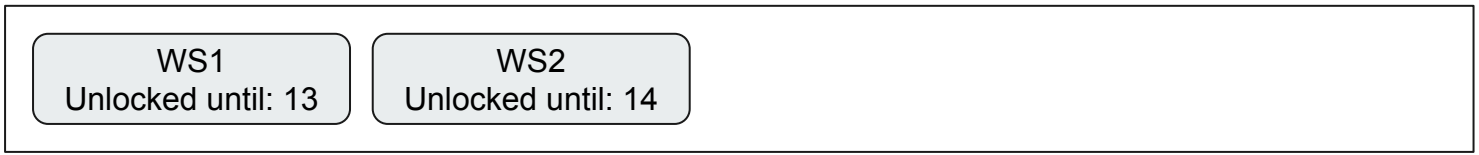
13 - 5 = 8
This workstation only becomes free at 8th minute

Researcher 2 enters at time 4, uses for 5 minutes

Week 8 One-day - Assigning Workstations

Workstation locks after 5 minutes of inactivity

Priority Queue



Researcher 2 enters at time 4, uses for 5 minutes

Unlock new workstation for researcher 2

Week 8 One-day - Assigning Workstations

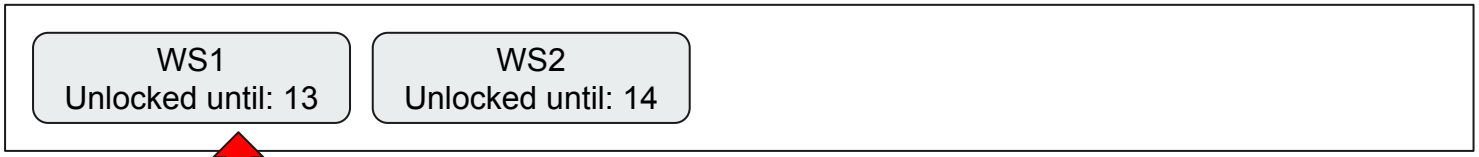
Case 3: Workstation is unlocked

- Assign current researcher to this workstation
  - Why?
    - Follow our Greedy Property - Assign researcher to the **oldest** workstation available

Week 8 One-day - Assigning Workstations

Workstation locks after 5 minutes of inactivity

Priority Queue



8 <= 12 <= 13
This workstation is available!

Researcher 3 enters at time 12, uses for 9 minutes

Week 8 One-day - Assigning Workstations

Workstation locks after 5 minutes of inactivity

Priority Queue

WS2
Unlocked until: 14

Researcher 3 enters at time 12, uses for 9 minutes

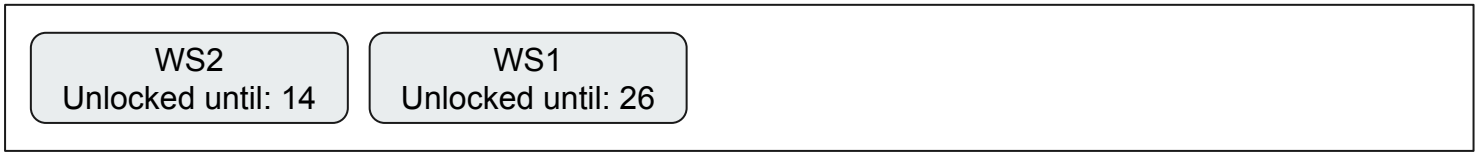
WS1
Unlocked until: 13

Assigned to researcher 3 & 1 unlock saved

Week 8 One-day - Assigning Workstations

Workstation locks after 5 minutes of inactivity

Priority Queue



Researcher 3 enters at time 12, uses for 9 minutes

WS1
Unlocked until: 26

Update the unlock time and push back into queue

Week 8 One-day - Assigning Workstations

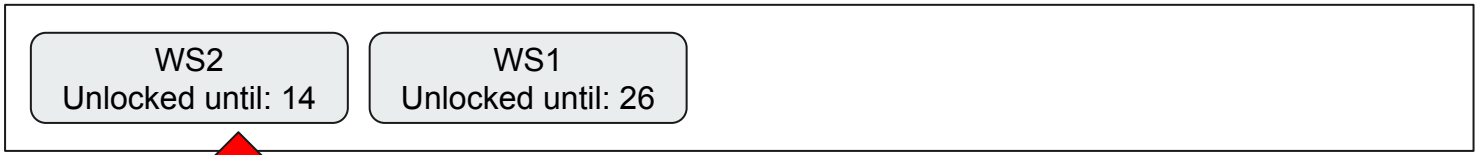
Case 4: Workstation is already locked

- Discard first workstation in PQ and check again
  - Why?
    - Researchers enter in chronological order - no other researchers will be able to use this workstation
    - Other workstations in PQ gets unlocked later - researcher may still be able to use them

Week 8 One-day - Assigning Workstations

Workstation locks after 5 minutes of inactivity

Priority Queue



14 < 15
This workstation is useless, discard

Researcher 4 enters at time 15, uses for 5 minutes

Week 8 One-day - Assigning Workstations

Workstation locks after 5 minutes of inactivity

Priority Queue

14 < 26 - 5 = 21
This is case 2!

Researcher 4 enters at time 15, uses for 5 minutes

After every discard, you will encounter 1 of the 4 states again!

TreeSet<E> / TreeMap<K, V>

- Uses a bBST as the underlying data structure
  - Red-black Tree
- Most methods are the same as HashSet / HashMap
  - HashSet / HashMap takes O(1) for most operations
  - TreeSet / TreeMap takes O(lg N) for most operations
- Additional methods available due to elements ordering in the data structure

TreeSet<E> / TreeMap<K, V>

- Use **comparison** methods **only** to order elements & check equality
  - Natural order: Comparable.compareTo
    - new TreeSet<MyClass>()
  - Explicit Comparator
    - new TreeSet<MyClass>(comparator)
- If compare(a, b) == 0, values are treated as equal
  - Store only **1 copy of a**
- Contrast to HashMap/Set using equals() and hashCode()

TreeSet<E> API: Set Operations

Method Signature	Description	Runtime
`boolean add(E e)`	Adds the specified element to this set if it is not already present.	O(lg N)
`void clear()`	Removes all of the elements from this set.	O(N)
`boolean contains(Object o)`	Returns true if this set contains the specified element. (Based off the object's `equals()` method)	O(lg N)
`boolean isEmpty()`	Returns true if this set contains no elements.	O(1)
`boolean remove(Object key)`	Removes the specified element from this set if it is present.	O(lg N)
`int size()`	Returns the number of elements in this set (its cardinality).	O(1)

The methods are exactly the same as HashSet<E>, just different runtimes.

TreeSet<E> API: NavigableSet Operations

Method Signature	Description		Runtime
`E ceiling(E e)`	Returns the least element in this set greater than **or equal** to the given element, or null if there is no such element.	`>= e`	O(lg N)
`E floor(E e)`	Returns the greatest element in this set less than **or equal** to the given element, or null if there is no such element.	`<= e`	O(lg N)
`E higher(E e)`	Returns the least element in this set **strictly** greater than to the given element, or null if there is no such element.	`> e`	O(lg N)
`E lower(E e)`	Returns the greatest element in this set **strictly** less than the given element, or null if there is no such element.	`< e`	O(lg N)
`E first()`	Returns the first (lowest) element currently in this set.		O(lg N)
`E last()`	Returns the last (highest) element currently in this set.		O(lg N)

TreeSet<E> API: NavigableSet Operations

```
TreeSet<Integer> primes = new TreeSet(Set.of(2,3,5,7,11));

assert 2 == primes.lower(3); // 2 < 3
assert 3 == primes.floor(3); // 3 <= 3

assert 5 == primes.higher(3); // 5 > 3
assert 11 == primes.ceil(8); // 11 >= 8

assert 2 == primes.first();
assert 11 == primes.last();
```

TreeSet<E> API: NavigableSet Views

Method Signature	Description		Runtime
`SortedSet headSet(E to)`	Returns a view of the portion of this set whose elements are strictly less than `to`.	`x < to` `(-∞, to)`	O(1)
`SortedSet tailSet(E from)`	Returns a view of the portion of this set whose elements are greater than or equal to `from`.	`from <= x` `[from, ∞)`	O(1)
`SortedSet subSet(E from, E to)`	Returns a view of the portion of this set whose elements range from `from`, inclusive, up til `to`, exclusive.	`from <= x < to` `[from, to)`	O(1)

Adding/removing in the view affects the original Set.
Inserting an element outside of the given range will throw an Exception.

Note that calling size() on any subset takes O(N) time instead of O(1).

TreeSet<E> API: NavigableSet Views

```
TreeSet<Integer> primes = new TreeSet(Set.of(2,3,5,7,11));

primes.headSet(7); // {2, 3, 5}
primes.tailSet(3); // {3, 5, 7, 11}
primes.subSet(3, 7); // {3, 5}
```

TreeMap<K, V> API: Map Operations

Method Signature	Description	Runtime
`V put(K key, V value)`	Associates the specified value with the specified key in this map. (Adds key to the `TreeMap` with the value.)	O(lg N)
`void clear()`	Removes all of the mappings from this map. (Clears the `TreeMap`.)	O(N)
`boolean containsKey(Object key)`	Returns true if this map contains a mapping for the specified key. (Based off the object's `equals()` method)	O(lg N)
`boolean containsValue(Object value)`	Returns true if this map maps one or more keys to the specified value. (Based off the object's `equals()` method)	**O(N)**
`V get(Object key)`	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.	O(lg N)

The methods are exactly the same as HashMap<K, V>, just different runtimes.

TreeMap<K, V> API: Map Operations

Method Signature	Description	Runtime
`boolean isEmpty()`	Returns true if this map contains no key-value mappings.	O(1)
`V remove(Object key)`	Removes the mapping for the specified key from this map if present.	O(lg N)
`int size()`	Returns the number of key-value mappings in this map.	O(1)
`Set<Map.Entry<K,V>> entrySet()`	Returns a `Set` view of the mappings contained in this map.	O(1)
`Set<K> keySet()`	Returns a `Set` view of the keys contained in this map.	O(1)
`Collection<V> values()`	Returns a `Collection` view of the values contained in this map.	O(1)

The methods are exactly the same as HashMap<K, V>, just different runtimes.
Similarly, it is not possible to iterate through a TreeMap directly (e.g. using for-loop).
You will need to use entrySet(), keySet(), or values() to access an iterable form of the data stored within.

TreeMap<K, V> API: NavigableMap Operations

Method Signature	Description		Runtime
`Map.Entry<K,V> ceilingEntry(K e)`	Returns the key-value mapping for the least key in this map greater than **or equal** to the given key, or null if there is no such entry.	`>= e`	O(lg N)
`Map.Entry<K,V> floorEntry(K e)`	Returns the key-value mapping for the greatest key in this map less than **or equal** to the given key, or null if there is no such entry.	`<= e`	O(lg N)
`Map.Entry<K,V> higherEntry(K e)`	Returns the key-value mapping for the least key in this map **strictly** greater than the given key, or null if there is no such entry.	`> e`	O(lg N)
`Map.Entry<K,V> lowerEntry(K e)`	Returns the key-value mapping for the greatest key in this map **strictly** less than the given key, or null if there is no such entry.	`< e`	O(lg N)

`<direction>Entry()` methods have `<direction>Key()` versions returning only the key.

TreeMap<K, V> API: NavigableMap Operations

Method Signature	Description	Runtime
`Map.Entry<K,V> firstEntry()`	Returns the key-value mapping for the first (lowest) key currently in this map.	O(lg N)
`Map.Entry<K,V> lastEntry()`	Returns the key-value mapping for the last (largest) key currently in this map.	O(lg N)

`<direction>Entry()` methods have `<direction>Key()` versions returning only the key.

TreeMap<K, V> API: NavigableMap Operations

```
TreeMap<String, Integer> marks = new TreeMap(
  Map.of("Tzerbin", 18, "Ryan", 16, "Zhi Jian", 17));
// Ordered as { "Ryan": 16, "Tzerbin": 18, "Zhi Jian": 17 }

// "Ryan" <= "Steve" in dictionary order
assert marks.floorKey("Steve").equals("Ryan");

// "Tzerbin" < "Zhi Jian" in dictionary order
assert marks.higherEntry("Tzerbin").getValue() == 17;
```

TreeMap<K, V> API: NavigableMap Views

Method Signature	Description		Runtime
`SortedMap<K,V> headMap(K toKey)`	Returns a view of the portion of this map whose keys are strictly less than toKey.	`x < to` `(-∞, to)`	O(1)
`SortedMap<K,V> tailMap(K fromKey)`	Returns a view of the portion of this map whose keys are greater than or equal to fromKey.	`from <= x` `[from, ∞)`	O(1)
`SortedMap<K,V> subMap(K fromKey, K toKey)`	Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.	`from <= x < to` `[from, to)`	O(1)

Adding/removing in the view affects the original Map.
Inserting an element outside of the given range will throw an Exception.

Note that calling size() on any submap takes O(N) time instead of O(1).

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

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

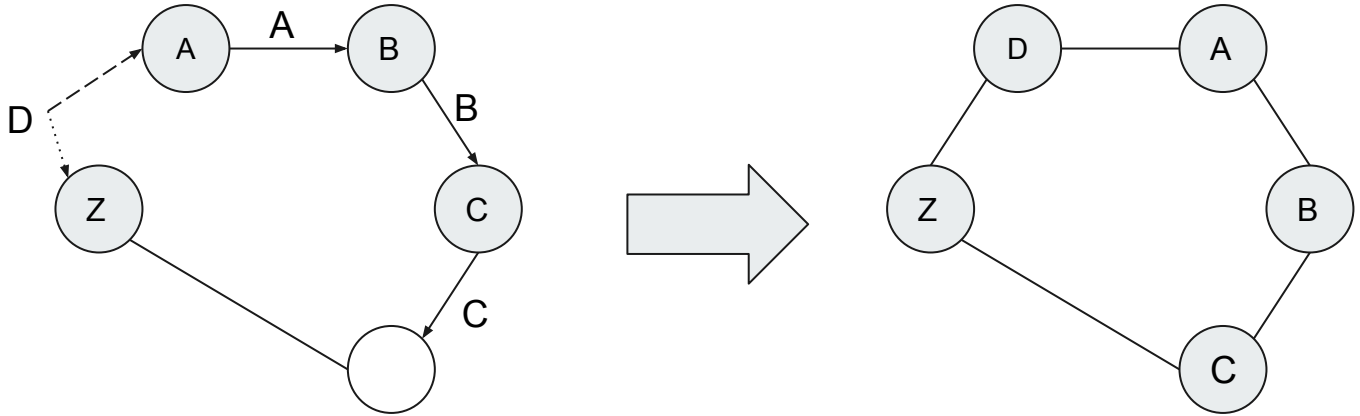
- Released this Tuesday
- Question 3 (Baby Names) is an optional lab
  - 1.5 extra marks, but total is capped at 27
  - You MUST implement using the Trie data structure to solve it (BST solution will not be accepted)

Take-Home Lab 3 - Ladice

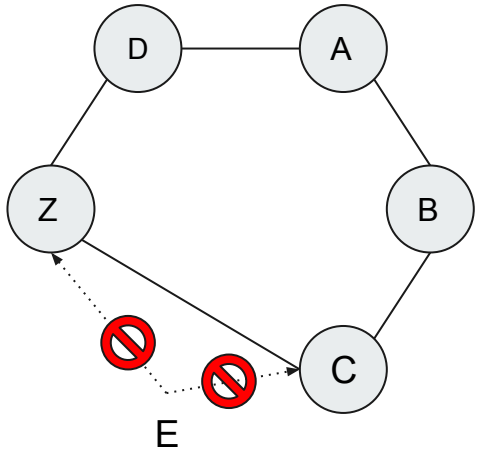
- N items, L drawers
  - Each item has 2 allowed positions A_i, B_i
- Try to insert each item
  - If successful, print LADICA
  - If discarded, print SMECE

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.

Ladice



Ladice



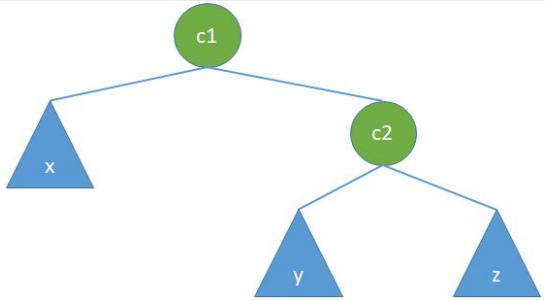
When is a “bunch” of drawers full?
(i.e. cannot push to make space?)

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.

Factor-Free Trees



- How does the in-order traversal correspond to the original tree?
- Which values could have been at the original root?
  - What happens if we have multiple possibilities?
  - In which order should we search?

Take-Home Lab 3 - Baby Names [Optional]

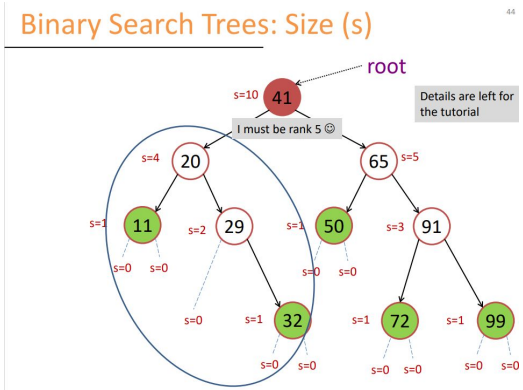
- Update and query a database of baby names, with gender suitability
- Q <= **500,000** queries
    0. Exit program.
    1. Given name and gender suitability, add suggestion
    2. Given name and gender suitability, remove suggestion
    3. Given [start] and [end], and suitability
        - Print number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

Counting

- { [start] <= name < [end] } is equal to
  - { [start] <= name } - { [end] <= name }
- If at "ABCDEFG", how do we count things like "ABCDE**G**A", but not "ABCDE**E**A"?
  - **All** of ABCDEG(...) should count
  - Part of ABCDEF(...) should count
  - None of ABCDEE(...) should count



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

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

Extras/Cut Content

Weekly meme from the original/official lab slides:

Yeah, no.