CS2040 Lab 7
TreeSet/TreeMap

Lab 7 – TreeSet/TreeMap

- Java provides the TreeSet/TreeMap API for handling a balanced BST
  - Internally uses a Red-Black tree (not examinable) instead of an AVL tree
- Most methods are the same as HashSet/HashMap
- Additional methods available in TreeSet/TreeMap since the elements are sorted (second slide of TreeSet/third slide of TreeMap API)
- Differences between HashSet/HashMap and TreeSet/TreeMap:
  - HashSet/HashMap takes O(1) time for most operations, while TreeSet/TreeMap takes O(log n) time
  - HashSet/HashMap does not store elements in sorted order, while TreeSet/TreeMap maintains a sorted order all the time

Lab 7 – TreeSet/TreeMap

- Like PriorityQueue, the use of TreeSet/TreeMap will require you to implement the Comparable interface in your custom classes, or pass in a Comparator which compares the keys of the TreeSet/TreeMap eg:
  - `TreeMap<Integer, String> tm = new TreeMap<Integer, String> (comp); // comp is a Comparator which compares Integers`
- Java TreeSet/TreeMap will also **not support duplicate keys**; if a key is already present in the TreeSet/TreeMap, then inserting the same key again does nothing (TreeSet), or replaces the value associated with that key (TreeMap)

Lab 7 – TreeSet

Method name	Description	Time
.add(YourClass element)	Adds *element* to the TreeSet	O(log n)
.clear()	Clears the TreeSet	O(n)
.contains(Object o)	Checks if *o* is in the TreeSet, based off the object's equals() method	O(log n)
.isEmpty()	Checks if the TreeSet is empty	O(1)
.remove(Object o)	Removes *o* if it is in the TreeSet, based off the object's equals() method	O(log n)
.size()	Returns the number of elements in the TreeSet	O(1)

The methods in this slide are exactly the same as in HashSet

Lab 7 – TreeSet

Method name	Description	Time
.ceiling(YourClass element)	Returns the first element equals to or larger than *element* in the TreeSet	O(log n)
.floor(YourClass element)	Returns the first element equals to or smaller than *element* in the TreeSetx`	O(log n)
.first()	Returns the smallest element in the TreeSet	O(log n)
.headSet(YourClass element)	Returns a set of elements smaller than *element*	O(1)
.last()	Returns the largest element in the TreeSet	O(log n)
.subSet(YourClass start, YourClass end)	Returns a set of elements between *start* (inclusive) to *end* (exclusive)	O(1)
.tailSet(YourClass element)	Returns a set of elements larger than *element*	O(1)

Note: calling .size() on any subset (from headSet(), subSet() or tailSet()) takes O(n) time instead of O(1)

Lab 7 – TreeMap

Method name	Description	Time
.put(YourClass key, YourClass value)	Adds *key* to the TreeMap with the value *value*	O(log n)
.clear()	Clears the TreeMap	O(n)
.containsKey(Object o)	Checks if key *o* is in the TreeMap, based off the object's equals() method	O(log n)
.containsValue(Object o)	Checks if value *o* is in the TreeMap, based off the object's equals() method	O(n)
.get(Object o)	Gets the value corresponding to the key *o*	O(log n)
.isEmpty()	Checks if the TreeMap is empty	O(1)
.remove(Object o)	Removes the entry with key *o* if it is in the TreeMap, based off the object's equals() method	O(log n)
.size()	Returns the number of elements in the TreeMap	O(1)

The methods in this slide are exactly the same as in HashMap

Lab 7 – TreeMap

Method name	Description	Time
.entrySet()	Returns a set of all entries in the TreeMap	O(1)
.keySet()	Returns a set of all keys in the TreeMap	O(1)
.values()	Returns a collection of all values in the TreeMap	O(1)

Unlike TreeSet, it is not possible to iterate through a TreeMap (eg. enhanced for-loop) directly. You will need to use the above methods to access an iterable form of the data stored within

Lab 7 – TreeMap

Method name	Description	Time
.ceilingEntry(YourClass element)	Returns the first entry with a key equals to or larger than *element* in the TreeMap	O(log n)
.floorEntry(YourClass element)	Returns the first entry with a key equals to or smaller than *element* in the TreeMap	O(log n)
.firstEntry()	Returns the smallest entry in the TreeMap	O(log n)
.headMap(YourClass element)	Returns a map of entries smaller than *element*	O(1)
.lastEntry()	Returns the largest entry in the TreeMap	O(log n)
.subMap(YourClass start, YourClass end)	Returns a map of entries between *start* (inclusive) and *end* (exclusive)	O(1)
.tailMap(YourClass element)	Returns a map of entries larger than *element*	O(1)

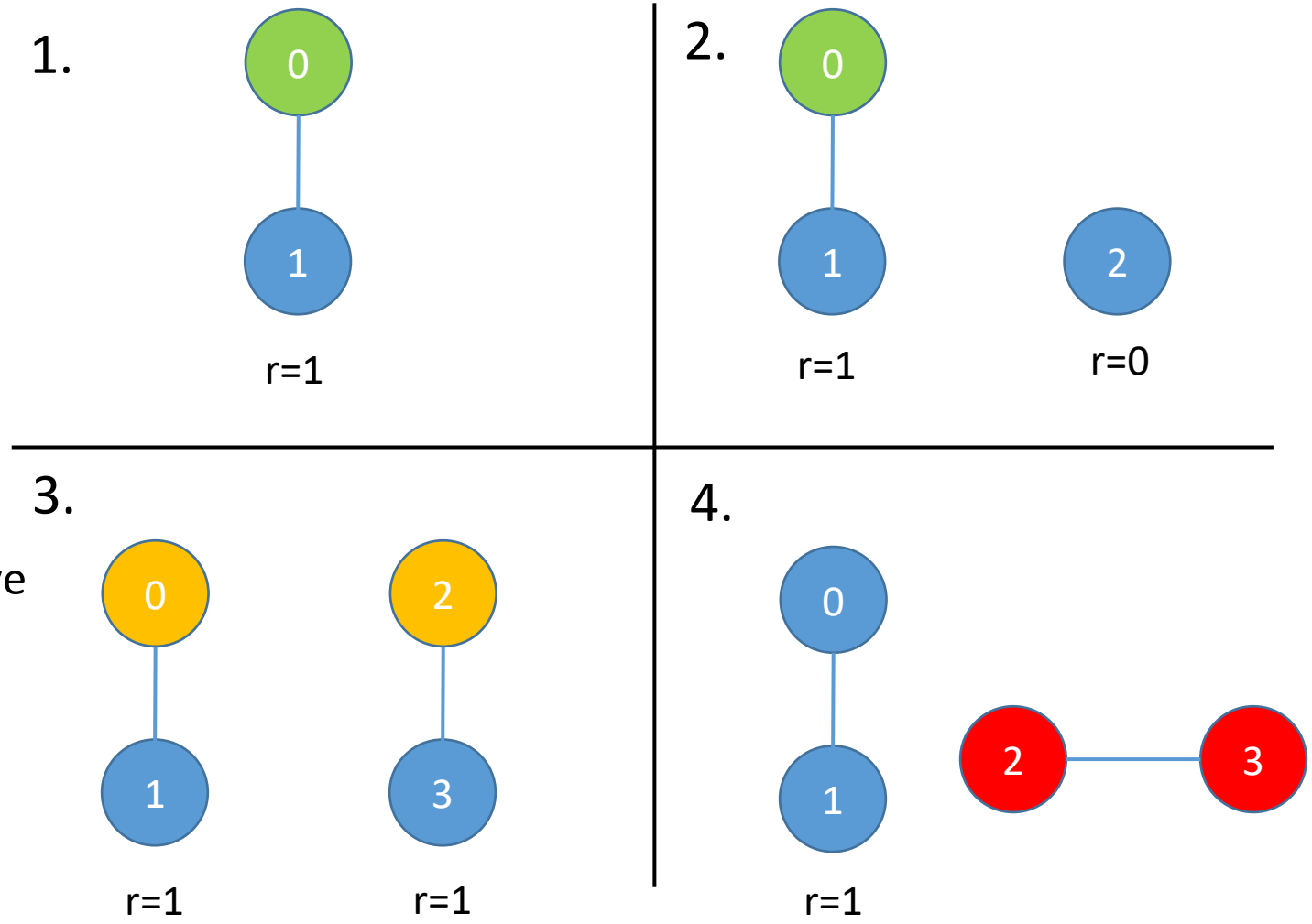
Note: calling .size() on any submap (from headMap(), subMap() or tailMap()) takes O(n) time instead of O(1)

Lab 7 – UFDS

- UFDS is not provided by the Java API, so you'll have to implement this yourself
  - Some code has already been provided in the lecture

Lab 7 – UFDS

- 1. The representative item of a set can be thought of as the root of a set
- 2. For union-by-rank, if we call unionSet() on two sets of different ranks, we pick the representative item of the set with larger rank to be the new representative item
- 3. If both sets have the same rank, it is fine to pick either representative item as the new one
- 4. If a set has no clearly defined representative item, something has gone horribly wrong



Take-Home Assignment 3 – Almost Union Find

- Implement an ADT that supports operations similar to UFDS
- Additionally, support an operation that moves a single item from one set to another (if they are not in the same set already)
  - This operation may be the most difficult one to support; you may want to consider alternate approaches, if the most literal approach seems too difficult to implement
- Also support a way to query a specific item, which returns both the number of items, as well as the total sum of all items in the set which the item queried belongs to
  - This operation may require the use of the *long* data type
- Note that there can be multiple testcases per input.
  - Use Kattio's io.hasMoreTokens() with a while loop

Take-Home Assignment 3 – Nicknames

- Given a series of names, determine how many names begin with a particular series of characters
- You should use your own implementation of an AVL tree to solve this
  - Attempts to use a TreeSet/TreeMap to solve this instead of a custom AVL tree will not be awarded any marks
  - Chances are such an attempt would TLE anyway
- Note that a query for "all strings beginning with a certain string x" can be rewritten as a query for all strings in the range[x, y) or [x, z] in lexicographical order, where y and z are strings to be determined

One-Day Assignment 6 – Planks

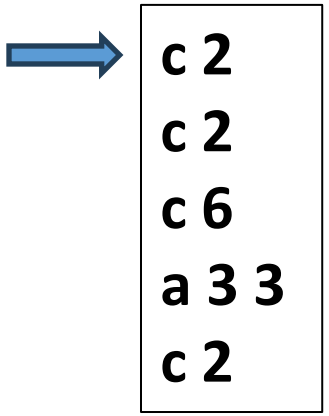
- Planks have a weight **W** and length **L**
- We must support two types of queries
  - Add a new plank
  - Pull out 2 planks **A** and **B**, and calculate the effort **E** required
- Plank **A** is the longest plank with length <= **X**
  - If there are ties, choose the plank with lightest weight
- Plank **B** is the shortest plank with length >= **X**
  - If there are ties, choose the plank with heaviest weight
  - Note that you will choose Plank **A** before Plank **B**
- Consider the maximum possible value of the effort **E** – what variable type do you need to store your value(s)?

One-Day Assignment 6 – Example

```
a 5 1
a 3 2
a 4 7
a 4 1
a 8 7
a 3 2
a 2 1
a 4 2
```

Weight	5	3	4	4	8	3	2	4
Length	1	2	7	1	7	2	1	2

One-Day Assignment 6 – Example



Weight	5	3	4	4	8	3	2	4
Length	1	2	7	1	7	2	1	2

A B

Weight	5	4	4	8	3	2
Length	1	7	1	7	2	1

E = (1 + 3 + 4) x (1 + |2 - 2|) = 8

One-Day Assignment 6 – Example

```
c 2
c 2
c 6
a 3 3
c 2
```

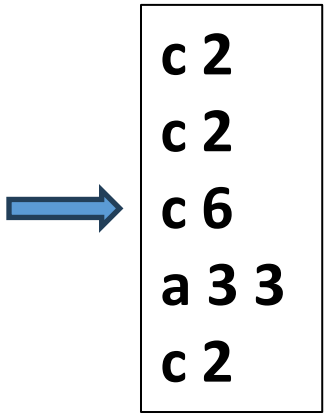
Weight	5	4	4	8	3	2
Length	1	7	1	7	2	1

B A

Weight	5	4	4	2
Length	1	7	1	1

$$\mathbf{E} = (1 + 3 + 8) \times (1 + |2 - 7|) = 72$$

One-Day Assignment 6 – Example



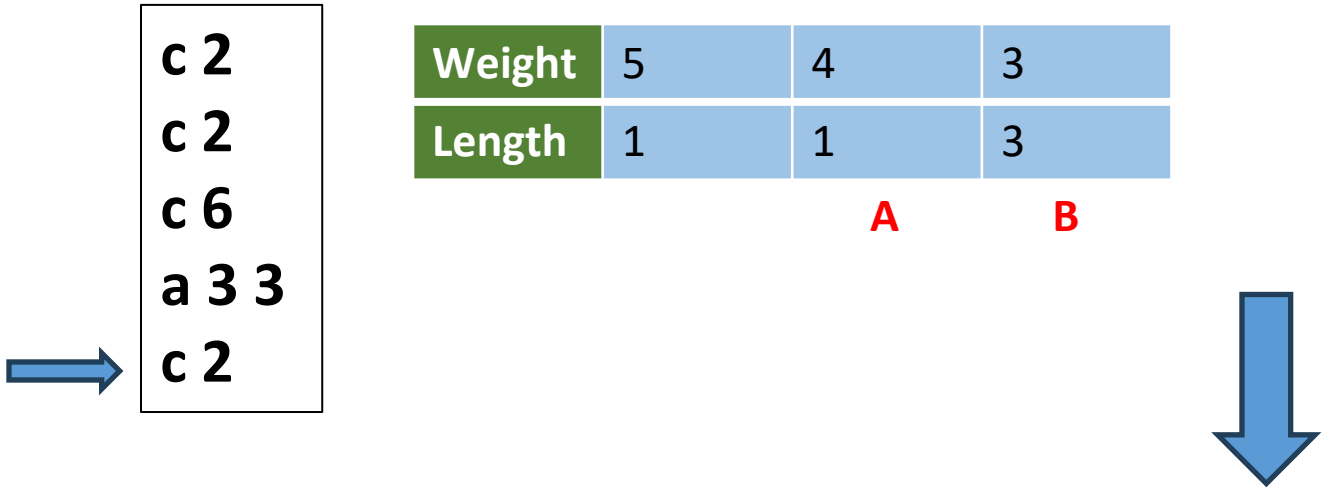
Weight	5	4	4	2
Length	1	7	1	1

B A

Weight	5	4
Length	1	1

E = (1 + 2 + 4) x (1 + |1 - 7|) = 49

One-Day Assignment 6 – Example



Weight	5	4	3
Length	1	1	3
		A	B

Weight	5
Length	1

$$\mathbf{E} = (1 + 4 + 3) \times (1 + |1 - 3|) = 24$$

One-Day Assignment 6 – General Idea

- General idea:
  - The use of TreeSet/TreeMap will require you to implement the Comparable interface in your custom classes (if any), which will require implementing the following method:
    - `public int compareTo(YourClass o)`
    - Alternatively, you can use a Comparator if you are more used to that
- Important note:
  - TreeSet/TreeMap does not support duplicate elements (it is a set/map after all), which means that the most intuitive way of storing planks as follows **will not work**:
  - TreeSet<IntegerPair>, where in IntegerPair:
    - Value 1: weight
    - Value 2: length
  - Since duplicate elements are not supported, if there are multiple planks with the exact same weight and length, it will only appear at most once in the TreeSet/TreeMap