UC Berkeley – Computer Science
CS61BL: Data Structures

Final, Summer 2016

This test has 11 questions worth a total of 65 (with 5 as extra credit) points. The exam is closed book, except that you are allowed to use three double-sided pages of notes as a cheat sheet. No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided.

Write the statement out below in the blank provided and sign. You may do this before the exam begins. **Any plagiarism, no matter how minor, will result in points deducted from your exam.**

"I have neither given nor received any assistance during the taking of this exam."

_____

_____

Signature: _____

Write your name and student ID on the front page. Write the names of your neighbors. Write and sign the given statement. Once the exam has started, write your login in the corner of every page.

Name:	Saralahan Kyao	**Your Login:**	cs61bl-ab
SID:	_____	**Name of person to left:**	Chalisee Fahwajiarns
TA:	영글 선생님	**Name of person to right:**	Chalisee Fahwajiarns

Notes:

- **All graphs are simple graphs: two vertices can only have one edge between them, and there are no self-edges.**
- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- There are a lot of problems on this exam. Work through the ones with which you are comfortable first. Do not get overly captivated by interesting design issues or complex corner cases you're not sure about.
- Not all information provided in a problem may be useful.
- Unless otherwise stated, you can use any standard library classes & methods, and can assume imports happen automatically.
- Unless otherwise stated, all given code on this exam should compile. All code has been compiled and executed before printing, but in the unlikely event that we do happen to catch any bugs during the exam, we'll announce a fix. Unless we specifically give you the option, the correct answer is not 'does not compile.'

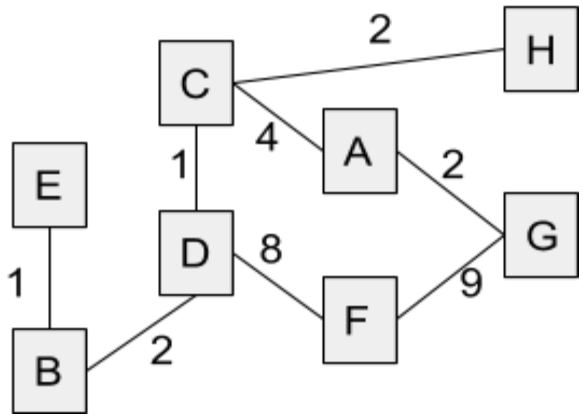
Optional. Mark along the line to show your feelings on the spectrum between 😟 and 😂.

Before exam: [😟 |_____ 😂]

After exam: [😟 _____|_ 😂]

1. Montague's (8 pts)

a. For the graph below, write the order in which vertices are visited using the specified algorithm. Each node's neighbors are given in alphabetical order. The starting point, vertex A, is provided for you.

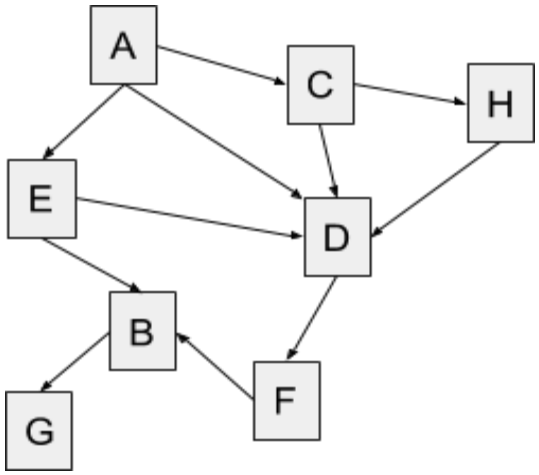


DFS: A C D B E F G H

BFS: A C G D H F B E

Dijkstra's: A G C D H B E F

b. Given the following graph,

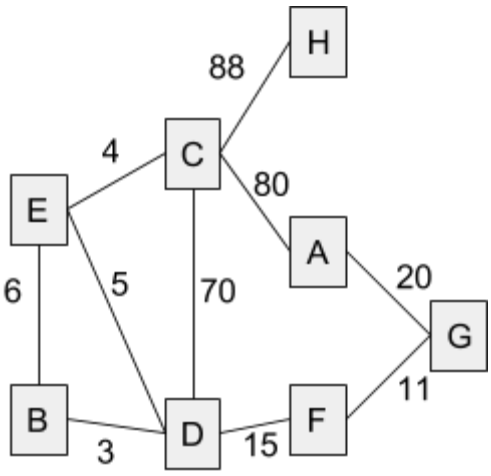


What is a valid topologically sorted ordering of the vertices?

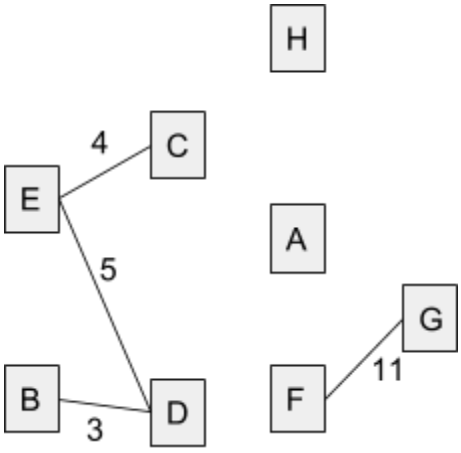
A C E H D F B G
A C H E D F B G
A E C H D F B G

How many possible orderings are there? 3 (as you can see above)

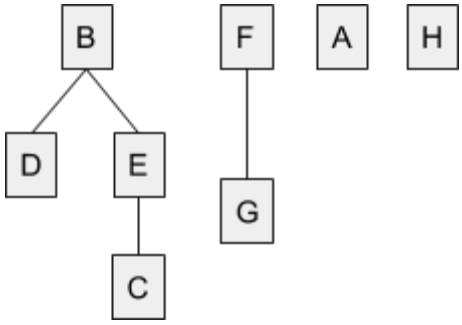
Parts **c-e** refer to the following weighted undirected graph to the right.



c. Draw the intermediate state of the graph during execution of Kruskal's Algorithm after four edges have been added:



d. Draw a **tree representation** of the union-find (disjoin sets) data structure, **without path compression**, at the same point in the algorithm:



e. Now suppose the algorithm has reached completion, but designers want to secretly add another edge to the graph of integer weight `w`. For each of the following conditions, circle whether adding an edge of that weight to the graph will, might, or will not change the edges in a MST:

If `w < 4`, a MST **[will / might / will not]** change.

If `5 < w < 7`, a MST **[will / might / will not]** change.

If `20 < w`, a MST **[will / might / will not]** change.

If `90 < w`, a MST **[will / might / will not]** change.

For w < 4, it will become a lightest edge, and must be added by Kruskal's.

For 5 < w < 20, it may be redundant with another edge already connecting two parts.

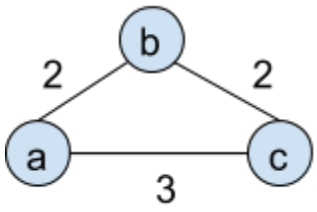
For 90 < w, it will be the heaviest edge, and the graph is already connectable without using it.

2. Kimchi Garden (6 pts)

For any **weighted, undirected** graph, determine whether each statement is true (T) or false (F) and circle your choice. If you choose false, provide a counterexample in the space below the statement.

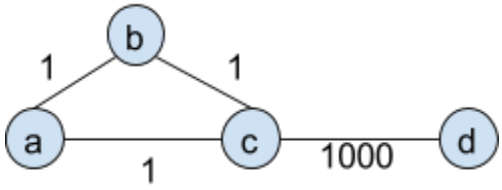
a. Adding 1 to each edge weight will not change any MSTs. **T**

b. Adding 1 to each edge weight will not change the shortest paths between vertices. **F**

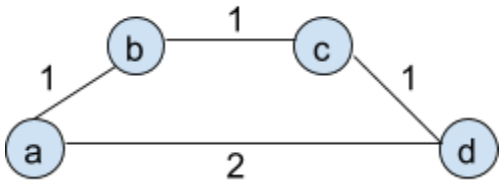


Consider a-c in the example.

c. If a graph of V vertices has more than |V| - 1 edges, and there is a unique edge of highest weight, it cannot be a part of *any* MST. **F**

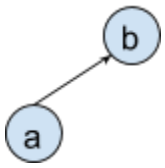


d. The shortest path between two nodes s and t must be a part of *some* MST. **F**



e. If we could sort a list in O(1) time, the runtime of Kruskal's would asymptotically improve. **T**

f. If there is a path between all pairs of vertices in a directed graph, there must be a cycle in the graph. **F**



3. Tamon Tea (2 pts)

Recall the class definition of `IntNode` as defined below:

```
public class IntNode {
    int item;
    IntNode next;
    public IntNode(int item, IntNode next) {
        this.item = item;
        this.next = next;
    }
}
```

Write the `breakTheLoop` method below, which takes in an `IntNode`. You are guaranteed that the list this node belongs to is **circular** (the "last" element of the list points to the "first"), it is in **sorted increasing** order, and all its items are **unique**. However, the node passed in to `breakTheLoop` can point to any node in the circular list.

`breakTheLoop` should destructively turn the list into a non-circular list where the first node contains the lowest number and return the first node. You may iterate through the circular list at most once. You may not need all the lines below.

```
public static IntNode breakTheLoop(IntNode n) {
    assert n != null;
    while (n.item < n.next.item) {
        n = n.next;
    }
    IntNode toReturn = n.next;
    n.next = null;
    return toReturn;
}
```

4. Jasmine Thai Lunch Special (5 pts)

a. Provide the best case and worst case runtimes in theta notation in terms of N for the following operations and data structures. Assume `N` to be the number of nodes in the tree. Additionally, each node correctly maintains the size of the subtree rooted at it.

Operations	BST	Red-Black tree
`// Returns true if the object is in the tree` `boolean contains(T o);`	Best: Θ(1) (top of tree) Worst: Θ(N) (spindly tree)	Best: Θ(1) (at top of tree) Worst: Θ(log N)
`// Inserts the given object.` `void insert(T o);`	Best: Θ(1) Worst: Θ(N)	Best: Θ(log N) Worst: Θ(log N)
`// Returns the ith smallest object in the tree.` `T getElement(int i);`	Best: Θ(1) Worst: Θ(N)	Best: Θ(1) Worst: Θ(log N)

b. Professor Sarallahan Kyao at Harbuvard University decided to use a self-balancing binary search tree. However, the code became somehow corrupt and you, as a faithful assistant, have the job of fixing it.

At each node, we keep an `int` variable called `height`, which keeps track of the height of the subtree rooted at the node. A leaf node has a height of 0, and each non-leaf node has a height of `1 + max(left.height, right.height)`. Professor Kyao demands that for each node, the left child's height and right child's height differ by no more than 1. That is, `Math.abs(left.height - right.height) <= 1`. Thus, whenever one of our left or right subtrees is not balanced, or their height differs by more than 1, we should rebalance.

Fill in the blanks below so that the code correctly calls `rebalance()` as needed.

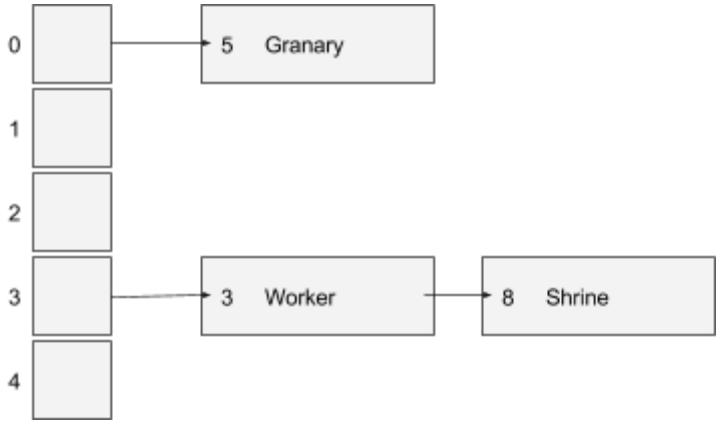
```
class WhatTree {
    Node root;
    static class Node {
        int item, height;
        Node left, right;


        Node(int item, int height) {
            this.item = item;
            this.height = height;
        }
    }

    public void insert(int item) {
        root = insert(root, item);
    }

    // Continued on next page...
```

```
    private static Node insert(Node node, int item) {
        if (node == null) {
            return new Node(item, 0);
        }
        if (node.item == item) {
            return node;
        } else if (node.item < item) {
            node.right = insert(node.right, item);
        } else {
            node.left = insert(node.left, item);
        }
        node.height = findHeight(node);

        if (!isBalanced(node)) {
            node = rebalance(node);
        }
        return node;
    }

    private static boolean isBalanced(Node n) {
        if (n == null) return true;
        if (Math.abs(n.left.height - n.right.height)) > 1) {
            return false;
        } else if (!isBalanced(n.left) || !isBalanced(n.right)) {
            return false;
        }
        return true;
    }

    // Finds the height of a node.
    private static int findHeight(Node n) { ... }

    // Assume this works properly
    private static boolean rebalance(Node n) { ... }

}
```

5. SF Soup Company (4 pts)

a. Consider a `HashMap<Integer, String>` with an underlying array of size 5. Draw the resulting structure after the following operations.

```
put(3, "monument");
put(8, "shrine");
put(3, "worker");
put(5, "granary");
```



Worker **replaces** monument, as their keys are the same.

b. Two instances of a class are **deep-equals** if all its fields are `.equals()`. Consider the `FastMap`, a variant of a `HashMap` that avoids collisions. A `FastMap` requires the following for every key K inserted into it: (1) Instances of K have a unique final int id field, which is set upon construction to the number of unique instances of K created so far. (2) The hashCode() method for K returns id. (3) The equals(Object o) method for K returns whether the two objects are deep equals. (4) Once a FastMap is instantiated, no more instances of K can be constructed.

The `FastMap`'s underlying array size is initialized to be the number of instances created (the highest `id`) and a `FastMap` does not resize.

What is the worst case runtime of `put`, `get`, and `remove` (they are the same) on a `FastMap`?

Runtime: O(1)

If the `FastMap` does indeed prevent collisions, justify your answer. Otherwise, give a counterexample.

Since each id is unique, hashCodes are also unique. Since the FastMap's underlying array is the size of the maximum id, nothing needs to share the same bucket. Thus there are no collisions.

Alternatively, note that we're limited by the size of an integer. If more than 2^{32} instances of K are created, the id will overflow and start looping around, reusing ids. At this point, the correctness of FastMap is broken. Any runtime and explanation pointing out this fact receives full credit.

6. Momo Masala (7 pts)

a. For each of the following scenarios below, fill in the blank with the letter corresponding to the sorting algorithm that would perform the best, with respect to **total runtime**, on sorting a **large array** (unless otherwise specified). Then provide the tight **asymptotic runtime** of the sort you chose, on the array, in Big-O notation, where N is the length of the array (for quicksort, provide the average/majority case runtime).

A. Insertion Sort B. Mergesort C. In-Place Quicksort D. Radix sort E. Bubble Sort

		Letter	Runtime
1.	Java `ints`. Fixed length integers should be sorted with radix sort	D	O(N)
2.	Floating point numbers of any length (e.g. 2.5, 3.1415) Arbitrary length rules out radix sort, so we use quicksort, which is best on arrays.	C	O(N log N)
3.	Comparable objects. Quicksort is fastest on arrays.	C	O(N log N)
4.	Comparable objects, given that there are O(N) inversions. We can do insertion sort in linear time.	A	O(N)
5.	Comparable objects, given that there are $O(N^{1.5})$ inversions. N log N is better than $N^{1.5}$	C	O(N log N)
6.	Strings of length $O(\log(N)^2)$. Radix sort with a radix in $O(\log(N)^2)$ will have a constant number of passes.	D	O(N)
7.	Java `ints`, and the array length is less than 40. Insertion sort is best when the list is short	A	O(1)
8.	Linked List of Comparable objects. Mergesort is best on linkedlists	B	O(N log N)

b. Suppose you are given a list of N objects, comparable in O(1) time. Based on the data set you are working with, you know the objects can only be in one of $O(4^N)$ possible permutations. Give an asymptotic lower bound on the running time[1] of sorting this list. Briefly justify your answer.

Lower bound: Ω(N)
Justification: A sorting algorithm must ask Q questions in Ω(Q) time and receive 2^Q possible answers, and $2^Q >= O(4^N)$ to cover all possible permutations. Thus, Q >= O(N), and the algorithm is in Ω(N).

What does this imply about the space complexity of a sorting algorithm that has the runtime of the lower bound? Provide an asymptotic upper bound of the space complexity. Justify your answer.

Upper bound: O(N)
Justification: An algorithm can only use as much space as it has time. Thus an algorithm taking O(N) time must take O(N) space.

[1] Disclaimer: in the decision tree model of computing. Don't worry about it.

Now suppose you know the objects can only be in $O(N^2)$ possible permutations. Again, provide an asymptotic lower bound on the running time of sorting this list. Justify your answer.

Lower bound: $\Omega(N)$
Justification: With the previous reasoning, we'd have that a $2^Q >= O(N^2)$, giving that $Q >= O(\log N)$, giving that our algorithm is in $\Omega(\log N)$. However, a sorting algorithm cannot run in $\Omega(\log N)$ time as it must inspect the entire input, and therefore the lower bound is still $\Omega(N)$.

7. Snack Shack: The Pork Belly Cubano (8 pts + 5 EC)

Provide a brief description of how to solve each of the following graph problems efficiently. Provide a worst-case runtime bound in Big-Θ notation in terms of V and E. Let WUG mean weighted, undirected graph.

a. Find a path from node s to node t in a strongly connected, directed graph.
Description: Run DFS starting from node s. Terminate at t, saving back pointers.
Comment: BFS is also acceptable. Running Dijkstra's or A* is unnecessary as we don't want a shortest path.
Runtime: $\Theta(|V| + |E|)$. $\Theta(|E|)$ is also okay because it's connected.

b. On a connected WUG, find the shortest path from s to t that goes through a given edge `e=(u,v)`.
Description: Run Dijkstra's from s to u, and v to t with (u, v) removed. Run Dijkstra's from s to v, and u to t with (u, v) removed. Take the minimum of the two.
Runtime: $\Theta(|E| \log |V|)$

c. Determine if a cycle exists in a directed graph.
Description: Run DFS. If a node encountered is in the call stack (can be maintained as a set), there's a cycle.
Runtime: $\Theta(|V| + |E|)$

d. Given a connected WUG, find the maximum spanning tree, the spanning tree of maximal weight.
Description: Run Kruskal's, with the list of edges sorted in reverse, from heaviest to lightest.
Runtime: $\Theta(|E| \log |E|)$
e. (Extra Credit, 2 pts) Find the longest path between any two nodes in a weighted, directed, acyclic graph.
Description: Suppose we want to find the longest path from s to t. Topologically sort the graph. Initialize all the distances to -Infinity. Starting from s, iterate through the vertices in sorted order. Update distances according to $dist(v) = \max_{\text{all edges } e = (u, v)} \{ dist(u) + dist(u, v) \}$. This is okay and even handles negative edges as iterating through in sorted order ensures all nodes u pointing into v are visited before v. Return dist(t).
Comment: Dijkstra's **does not work**.
Runtime: $\Theta(|V| + |E|)$
f. (Extra Credit, 3 pts) Given a weighted, undirected graph where the weights are bounded by k, give an algorithm and data structure to find the shortest s-t path in $O((|V|+|E|) \log(k))$ time.
Description: Observe that at any point during execution of Dijkstra's, the values in the heap will always range from HEAP_MIN to HEAP_MIN + k. Maintain an indexed (on 0 through k) heap of size k, where each node just stores a list of the entries that have that priority (to save on growing the size of the heap). Thus the heap operations in Dijkstra's will only take O(log k) time.

8. Little Gem Belgian Waffles (5 pts)

a. Less than a hundred years after returning to the Undying Lands, Elrond has found a new hobby: forging legendary water bottles to be given away at Valar career fairs. He is an ardent fan of Java, so he represents each of his creations as a `WaterBottle implements Comparable<WaterBottle>` and ranks them on a large number of quality factors. This way, when he sorts them, the water bottles are ranked from worst to best.

He wants to send the highest `k` quality water bottles of his `n` water bottles back to his daughter Arwen, where `k` is still undecided. He doesn't want to sort the entire list of water bottles, because that would be too slow. Help him implement a function that runs in **`Θ(n log k)`** time that will find the `k` highest quality water bottles in descending order.

Recall that `PriorityQueue`, `ArrayDeque`, and `LinkedList` implement `Queue`. `ArrayList` and `LinkedList` implement `List`. *You may not need all lines.*

```
public static List<WaterBottle> topK(List<WaterBottle> all, int k) {
    List<WaterBottle> ret = new ArrayList<>();
    Queue<WaterBottle> q  = new PriorityQueue<>();
    for (WaterBottle b : all) {
        q.add(b);
        if (q.size() > k) {
            q.poll();
        }
    }
    while (!q.isEmpty()) {
        ret.add(q.poll());
    }
    return ret;
}
```

Give a brief justification of the runtime:

The `PriorityQueue` maintains a size of `k`. We insert into and remove from the queue `n` times for a runtime of `Θ(n log k)`.

9. Kamado Sushi (4 pts)

The quicksort algorithm can be modified for finding the `k`th smallest element in an array. This is called the quickselect algorithm as briefly discussed in lab, and finds the item at sorted index `j = k - 1`. The partition step is the same as quicksort, and it only differs in the recursive call, recursing on the partition that contains the `k`th smallest element.

a. Using the first element of the list as the pivot, show the way the list is partitioned at each step when `quickselect(j = 5)` is called, trying to find the sixth smallest element. Circle the partition that is recursed on.

`42 93 50 39 81 94 23 28 95 89`

Left Partition	Pivot Partition	Right Partition	j
39 23 28	42	93 50 81 94 95 89	5

50 81 89	93	94 95	1
	50	81 89	1
	81	89	0

b. What is the best, worst, and average case runtime of quickselect?

Best: $\Theta(n)$

Worst: $\Theta(n^2)$

Average: $\Theta(n)$

c. Quickselect can be used to find the median of an array. What if, instead of selecting a pivot at random when quicksorting, instead we use quickselect to find the median and use that as the pivot? What is the worst, best, and average case runtime of quicksort now?

Best: $\Theta(n \log n)$

Worst: $\Theta(n^2 \log n)$

Average: $\Theta(n \log n)$

10. Chengdu Restaurant (4 pts)

For the below regex problems, assume these are **non-java** strings, so double escaping is not necessary.
a. Consider the regex `"[hello]\w+rld"`. Circle all strings that the regex fully matches.

`"helloworld"` `"hworld"` `"smallworld"` `"hellowrld"` `"otherworld"`

b. Consider the regex `"[a-d0-9]+\\[^\d]{2,5}"`. Circle all strings that the regex fully matches.

`"d00d+42"` `"61b\party"` `"aa61bb+cc"` `"000+ooo"` `"9a9\\+hue"` `"4242+meaning"`

For parts **c-d**, write regular expressions that will fulfill the conditions below. Be sure to escape special characters.

c. Match any occurrences of a string that has an "A" at the beginning and end, and greedily, any positive number of "B" in between.
Fully matched inputs: `"ABA"`, `"ABBBBBBA"`
Non-matching inputs: `"AAAABBBB"`, `"AAAA"`, `""`, `"BBBABBB"`, `"AABABABABABA"`

Regex: `AB+A`

d. Match a valid alphanumeric lowercase email address.
Fully matched inputs: `"alan42@cs61bl.io"`, `"party@gmail.com"`
Non-matching inputs: `"dun@goofed"`, `"CAPITALS@stop.shouting"`, `"n.onalph_anum@r1c.com"`

Regex: `[a-z0-9]+@[a-z0-9]+\.[a-z]+`

Parrot Party (This parrot speaks to me on a transcendental level)

This is a designated ExamFunOnlyZone™©. Draw or write whatever you want!

11. Ellenos Real Greek Yogurt (7 pts)

a. We can implement a `PriorityQueue` an array-based binary heap. Consider a heap where the keys are integers, and the priority is the key itself. Write out the values of the array representation after the following calls:

```
Starting array:          [ X , 0  , 6  , 1  , 9  , 7  , 4  , 13 , 17 , 12 ]
removeMin():             [ X , 1  , 6  , 4  , 9  , 7  , 12 , 13 , 17 ]
add(3):                  [ X , 1  , 3  , 4  , 6  , 7  , 12 , 13 , 17 , 9  ]
changePriority(17, 2):   [ X , 1  , 2  , 4  , 3  , 7  , 12 , 13 , 6  , 9  ]
```

b. Recall that `changePriority` on a `PriorityQueue` takes linear time. This means that we cannot use `changePriority` as given in Dijkstra's algorithm, otherwise the runtime will not be satisfactory. Augment the `ArrayHeap` that you wrote in lab18 such that `changePriority` takes O(log n) time. You may not need all lines.

```
public class ArrayHeap<T> {
    private ArrayList<Node> contents;
    /** Put any additional fields here and initialize them in the constructor. */
    private HashMap<T, Node> map;

    public ArrayHeap() {
        contents = new ArrayList<>();
        contents.add(null);
        map = new HashMap<>();
    }
    private class Node {
        T item; double priority;
        /** Put any additional fields here and initialize them in the constructor. */
        int index;

        private Node(T item, double priority, int index) {
            this.item = item;
            this.priority = priority;
            this.index = index;
        }
    }

    /** Bubbles up the node currently at the given index. */
    private void bubbleUp(int index) { ... }

    /** Bubbles down the node currently at the given index. */
    private void bubbleDown(int index) { ... }

    // Continued on next page...
```

```
    /** Always called whenever nodes are swapped by bubbleUp and bubbleDown */
    private void swap(int index1, int index2) {
        Node node1 = contents.get(index1);
        Node node2 = contents.get(index2);
        contents.set(index1, node2);
        contents.set(index2, node1);
        node1.index = index2;
        node2.index = index1;
    }

    /** Inserts an item with the given priority value. Same as enqueue, or offer. */
    public void insert(T item, double priority) {
        Node n = new Node(item, priority, contents.size());
        contents.add(n);
        bubbleUp(contents.size() - 1);
        map.put(item, n);
    }

    /** Returns and removes the Node with the smallest priority value. */
    public Node removeMin() {
        swap(1, contents.size() - 1);
        Node results = contents.remove(contents.size() - 1);
        bubbleDown(1);
        map.remove(results.item);
        return results;
    }

    /** Changes the node in this heap with the given item to have the given
     * priority. You can assume the heap will not have two nodes with the same
     * item. You can assume item is indeed in the heap. */
    public void changePriority(T item, double priority) {
        Node n = map.get(item);
        double prevPriority = n.priority;
        n.priority = priority;
        if (n.prevPriority < priority) {
            bubbleDown(n.index);
        } else {
            bubbleUp(n.index);
        }
    }

    /** Equally correct implementation for changePriority */
    public void changePriority2(T item, double priority) {
        Node n = map.get(item);
        n.priority = priority;
        bubbleDown(n.index);
        bubbleUp(n.index);
    }
}
```