

Data Structures and Algorithms

University of Washington, Autumn 2019

This exam has 5 questions worth a total of 45 points and is to be completed in 50 minutes. One double-sided, 8.5-by-11” sheet of notes is permitted. Electronic devices are prohibited. This exam is preprocessed by a computer when grading, so please **write darkly and write your answers inside the designated spaces**. **Write the statement below in the given blank and sign. You may do this before the exam begins.**

“I have neither given nor received any assistance in the taking of this exam.”

Signature: _____

Question	Points
1	1
2	6
3	13
4	11
5	14
Total	45

Name	
Student ID	
Name of person to left	
Name of person to right	

- **Work through the problems you are comfortable with first.**
- Not all information provided in a problem may be useful, and **you may not need all lines**.
- Take notes in any white space on this exam. Only the final answer region will be graded.
- Unless we specifically give you the option, the correct answer is not, ‘does not compile.’
- indicates that only one circle may be filled-in.
- indicates that more than one box may be filled-in.
- For answers that involve filling-in a or , fill-in the shape completely: or .

Designated Exam Relaxation Space

- (1 pt) **So It Begins** Write the statement on the front page and sign. Write your name, ID, and the names of your neighbors. Write your name in the given blank in the corner of every other page. Enjoy a free point.
- (6 pts) **Hashing** In the parts below, use the Point class. Mark NEI if there is NOT ENOUGH INFORMATION.

```
public class Point {
    public final int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int hashCode() {
        return this.x + this.y;
    }
}
```

(a) Mark all of the points that will collide with Point(1, 2) on a hash table with $M = 2$ buckets.

- | | | |
|--------------------------------------|--------------------------------------|--------------------------------------|
| <input type="checkbox"/> Point(1, 1) | <input type="checkbox"/> Point(3, 1) | <input type="checkbox"/> Point(1, 4) |
| <input type="checkbox"/> Point(2, 1) | <input type="checkbox"/> Point(1, 3) | <input type="checkbox"/> Point(1, 5) |

(b) Does there exist a point that is **guaranteed to collide** with Point(1, 2) on any choice of M buckets?

- Yes No NEI

If yes, mark all of the points that are guaranteed to collide.

- | | | |
|--------------------------------------|--------------------------------------|--------------------------------------|
| <input type="checkbox"/> Point(1, 1) | <input type="checkbox"/> Point(3, 1) | <input type="checkbox"/> Point(1, 4) |
| <input type="checkbox"/> Point(2, 1) | <input type="checkbox"/> Point(1, 3) | <input type="checkbox"/> Point(1, 5) |

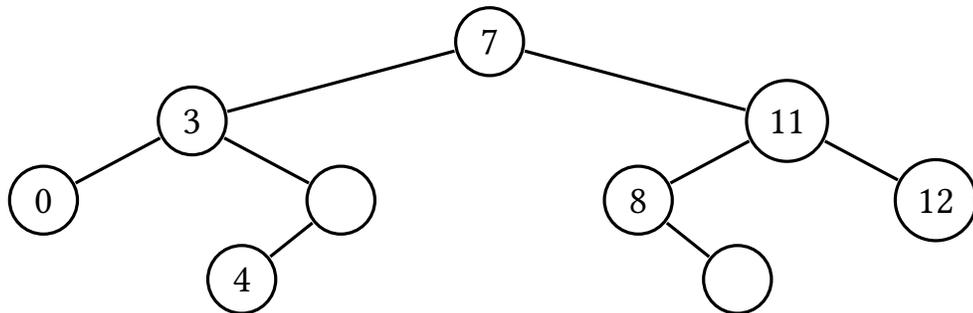
(c) Suppose two points are considered equals based on their x-values alone. For example, Point(1, 2) and Point(1, 3) are considered equals. When using the Point.hashCode defined above, will a hash table always, sometimes, or never return true when searching for a previously-inserted equals point?

```
public boolean equals(Object o) {
    Point other = (Point) o;
    return this.x == other.x;
}
```

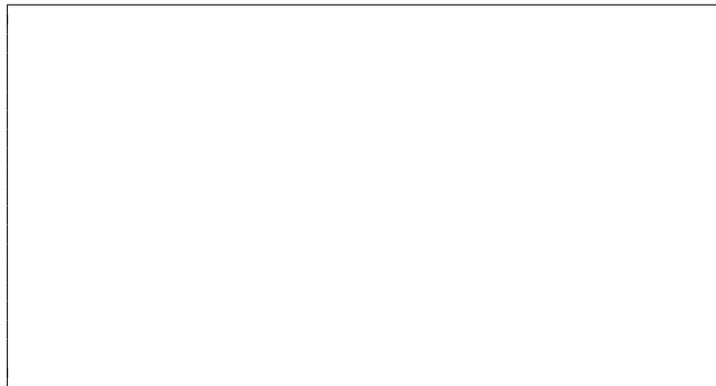
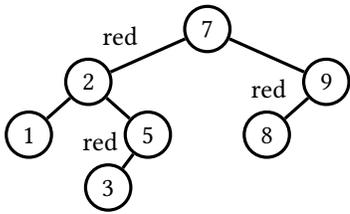
Always Never
 Sometimes NEI

3. (13 pts) **Trees**

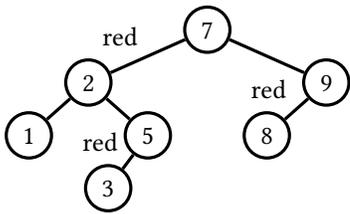
(a) Fill-in the blank nodes of the following binary search tree with valid **integer values**.



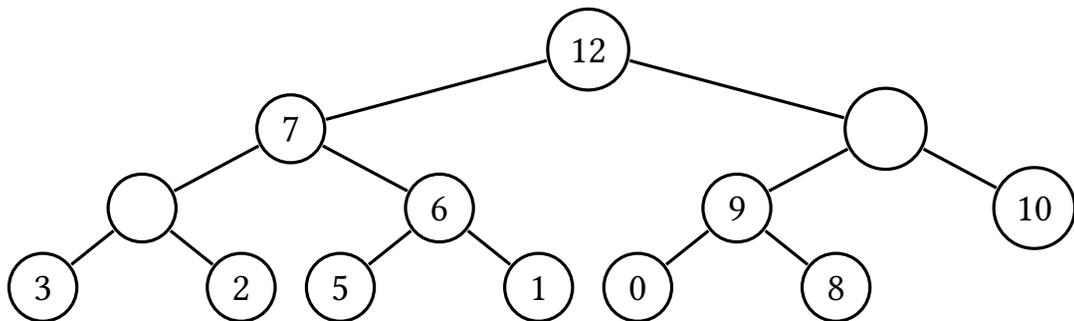
(b) Draw the 2-3 tree corresponding to the following left-leaning red-black tree.



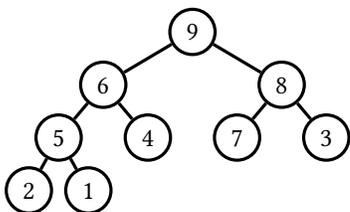
(c) Draw the left-leaning red-black tree after inserting 6. Label red edges **red**.



(d) Fill-in the blank nodes of the following binary max-heap with valid **integer values**.



(e) Draw the binary max-heap after removing the max value.



The implementation for `TrieSet.add` is shown below.

```
public void add(String key) {
    if (key == null) throw new IllegalArgumentException("Argument is null");
    root = add(root, key, 0);
}
private Node add(Node n, String key, int i) {
    if (n == null) n = new Node(false, R); // Assume R is defined elsewhere
    if (i == key.length()) n.isKey = true;
    else {
        char c = key.charAt(i);
        n.next.put(c, add(n.next.get(c), key, i + 1));
    }
    return n;
}
```

In lecture, we described in English an algorithm for collecting all the keys in a trie, `TrieSet.COLLECT()`.

<pre>function COLLECT() x ← new list of strings for each char c in root.next.keys() do COLHELP(c, x, root.next.get(c)) return x</pre>	<pre>function COLHELP(s, x, n) if n.isKey then x.add(s) for each char c in n.next.keys() do COLHELP(s + c, x, n.next.get(c))</pre>
--	--

TRIESORT works except on inputs with duplicate strings: only one of each duplicate is in the sorted output!

(c) Describe modifications to the `TrieSet` class so that `TRIESORT` works on inputs with duplicate strings. Write one English sentence per blank. Do not write outside the blanks. You may not need all the blanks.

class `TrieSet` – instance variable changes are not necessary.

class `Node`

i. _____

ii. _____

method `add(String key)` – changes are not necessary.

method `add(Node n, String key, int i)`

iii. _____

iv. _____

method `COLLECT()` – changes are not necessary.

method `COLHELP()` – assume that `keys()` returns keys in sorted order.

v. _____

vi. _____