CSE 373 Midterm Exam Solution

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."

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

Signature: Evil Kevin

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

Name	Evil Kevin
Student ID	1234567890
Name of person to left	Robbie Weber
Name of person to right	Hannah Tang

- **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. (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.

2. (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.

☐ `Point(1, 1)`
☑ `Point(2, 1)`
☐ `Point(3, 1)`
☐ `Point(1, 3)`
☑ `Point(1, 4)`
☐ `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.

☐ `Point(1, 1)`
☑ `Point(2, 1)`
☐ `Point(3, 1)`
☐ `Point(1, 3)`
☐ `Point(1, 4)`
☐ `Point(1, 5)`

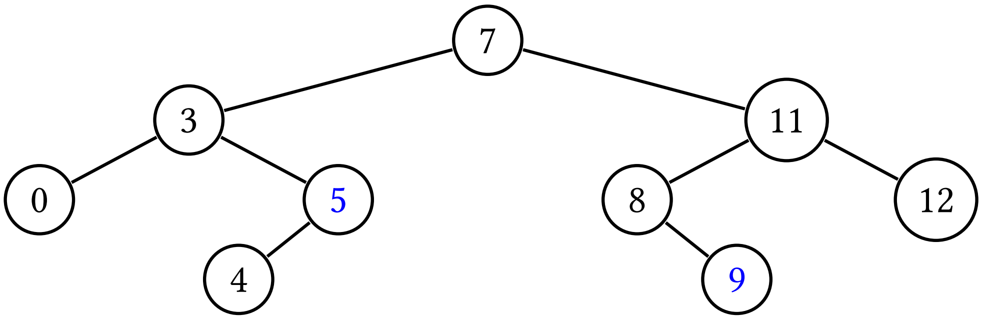
(c) Suppose two points are considered `equals` based on their `x`-values alone. For example, `Point(1, 2)` and `Point(1, 3)` are considered `equals`. When using the `Point.hashCode` defined above, will a hash table always, sometimes, or never return true when searching for a previously-inserted `equals`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

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

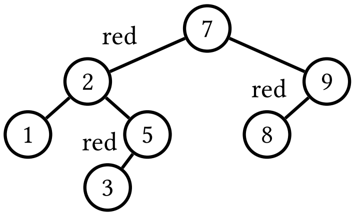
○ Always
● Sometimes
○ Never
○ NEI

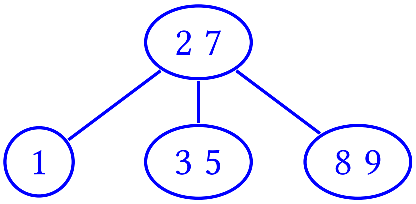
3. (13 pts) **Trees**

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

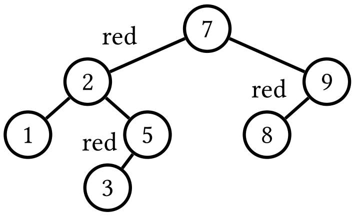


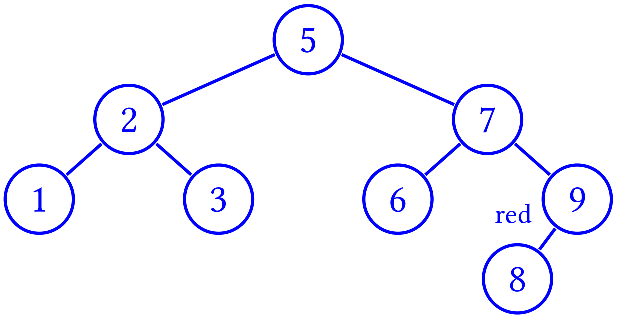
(b) Draw the 2-3 tree corresponding to the following left-leaning red-black tree.



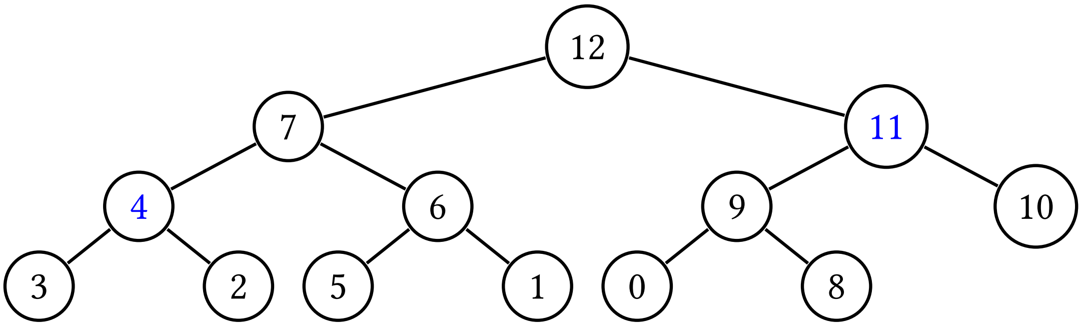


(c) Draw the left-leaning red-black tree after inserting 6. Label red edges **red**.

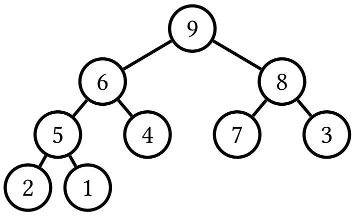


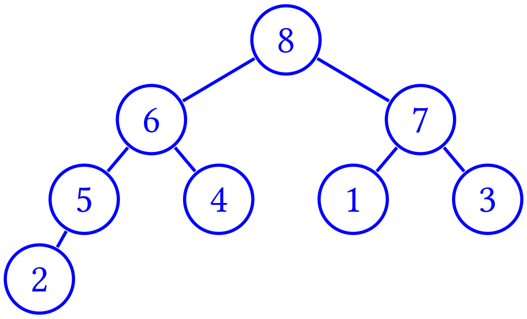


(d) Fill-in the blank nodes of the following binary max-heap with valid **integer values**.



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





4. (11 pts) **Algorithm Analysis**

(a) Suppose we know that the order of growth of a function is in $\Theta(N)$. Mark all of the true expressions.

☐ $O(1)$
☐ $O(\log N)$
☑ $O(N)$
☑ $O(N^2)$
☐ $\Theta(1)$
☐ $\Theta(\log N)$
☑ $\Theta(N)$
☐ $\Theta(N^2)$
☑ $\Omega(1)$
☑ $\Omega(\log N)$
☑ $\Omega(N)$
☐ $\Omega(N^2)$

Give the order of growth of the runtime in $\Theta(\cdot)$ notation as a function of N. Your answer should be simple with no unnecessary leading constants or summations.

(b) $\Theta(\underline{N^2\qquad})$

```
static void findFromMidpoint(int N) {
    for (int x = 0; x < N; x += 1) {
        int i = N / 2;
        while (i != x) {
            if (i > x) {
                i -= 1;
            } else {
                i += 1;
            }
        }
} }
```

(c) $\Theta(\underline{N \log N\quad})$

```
static void recursion(int N) {
    if (N > 1) {
        recursion(N / 2);
        for (int x = 0; x < N; x += 1) {
            System.out.println(x);
        }
        recursion(N / 2);
} }
```

(d) $\Theta(\underline{N^2\qquad})$

```
static void reverseDeque(ArrayDeque<Integer> deque) {
    // Circular ArrayDeque from HW 2 but without resizing
    int N = deque.size();
    for (int x = 0; x < N; x += 1) {
        System.out.println(deque.get(x));
    }
    if (N > 1) {
        int item = deque.removeFirst();
        reverseDeque(deque);
        deque.addLast(item);
} }
```

5. (14 pts) **Specialized Data Structures** In lecture, we implemented a trie using the `CharMap` data type.

(a) Give the order of growth of the runtime for `contains` on each `CharMap` implementation as a function of R, the size of the alphabet. (In lecture, $R = 128$ for ASCII.) Do not use L or N in your answer.

	Best Case	**Worst Case**
`DataIndexedCharMap`	$\Theta(1)$	$\Theta(1)$
`HashTableCharMap`	$\Theta(1)$	$\Theta(R)$
`BinarySearchTreeCharMap`	$\Theta(1)$	$\Theta(R)$
`LLRBTreeCharMap`	$\Theta(1)$	$\Theta(\log R)$
`UnorderedLinkedCharMap`	$\Theta(1)$	$\Theta(R)$

Below is the R-way `TrieSet` implementation using a `WeirdCharMap` instead of a `DataIndexedCharMap`.

```
public class TrieSet {
    private Node root;
    private static class Node {
        private boolean isKey;
        private WeirdCharMap<Node> next;
        private Node(boolean b, int R) {
            isKey = b;
            next = new WeirdCharMap<Node>(R);
        } } ...
```

(b) Suppose we have a `WeirdCharMap` with the following order of growth of the runtime for `contains`.

Best case $\Theta(\log R)$ **Worst case** $\Theta(\sqrt{R})$

Give the order of growth of the runtime for `TrieSet.contains` with `WeirdCharMap` as a function of

- R, the size of the alphabet;
- L, the length of the search string;
- N, the total number of strings stored in the trie.

Assume the search string is in the trie.

Best case $\Theta(L \log R)$ **Worst case** $\Theta(L\sqrt{R})$

We can use data structures to sort items. Consider the following sorting algorithm, TRIESORT.

```
function TRIESORT(stringsToSort)
    $t \leftarrow$ new TrieSet
    for each string $s$ in stringsToSort do
        $t$.add($s$)
    return $t$.collect()
```

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().

```
function collect()
    $x$ ← new list of strings
    for each char $c$ in root.next.keys() do
        colHelp($c$, $x$, root.next.get($c$))
    return $x$
```

```
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($c$))
```

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. Instead of storing a boolean `isKey`, store an `int count`.

ii. Initialize count to 0 in the constructor.

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

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

iii. Instead of marking the boolean `isKey`, increment the `int count`.

iv.

method collect() — changes are not necessary.

method colHelp() — assume that keys() returns keys in sorted order.

v. Instead of checking for n.isKey, check if `count > 0` and, if so, add count copies of s to x.

vi.