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(3, 1)`	[✓ `Point(1, 4)`]
[✓ `Point(2, 1)`]	☐ `Point(1, 3)`	☐ `Point(1, 5)`

A hash table with $M = 2$ buckets has bucket indices 0 and 1. `Point(1, 2)` has a hash code of $2 + 1 = 3$ so it has a bucket index of 1. `Point(2, 1)` shares the same hash code so it also has the same bucket index. `Point(1, 4)` has the hash code 5 which also shares the same bucket index.

(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(3, 1)`	☐ `Point(1, 4)`
[✓ `Point(2, 1)`]	☐ `Point(1, 3)`	☐ `Point(1, 5)`

The only time points will be guaranteed to collide with other points on any choice of M buckets is when their hash codes are exactly the same. From the example above, we know that `Point(2, 1)` shares the same hash code as `Point(1, 2)`.

(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?

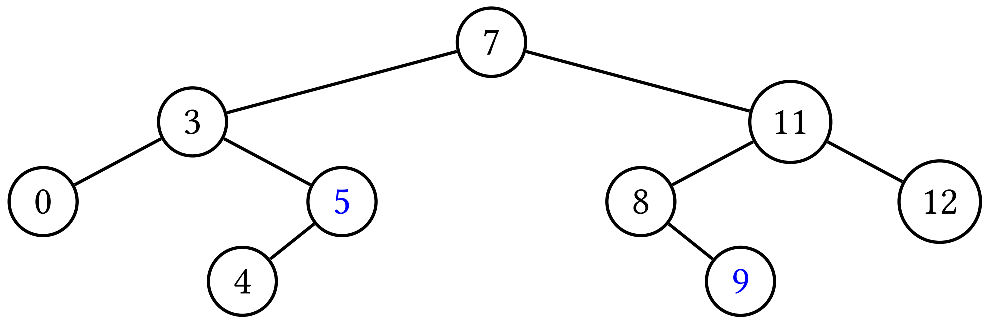
```
public boolean equals(Object o) {
    Point other = (Point) o;
    return this.x == other.x;
}
```

Always Never [✓ Sometimes] NEI

Consider the example given in the problem with `Point(1, 2)` and `Point(1, 3)`. These two points are considered `equals` but they have different hash codes. We want to make sure that we can find `Point(1, 3)` in the same bucket index as `Point(1, 2)` and vice-versa. This only sometimes occurs depending on the choice of M buckets. For example, if we only have $M = 1$ bucket, then this always works. (But this means that the size of the bucket will be N, where N is the total number of items–it's no better than unordered linked nodes.) On other choices of M buckets, this won't work. In general, we are sometimes able to find a previously-inserted `equals` point, but only when we're lucky and the bucket indices happen to be the same.

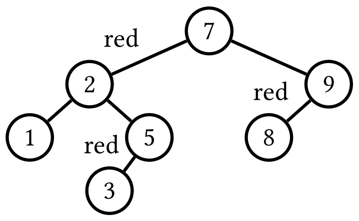
3. (13 pts) **Trees**

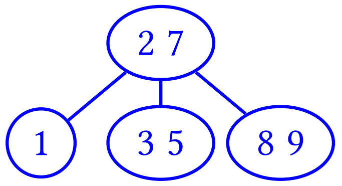
   (a) Fill-in the blank nodes of the following binary search tree with valid **integer values**.



The values in a binary search tree are ordered: imagine flattening the tree and putting all of the numbers on a line.

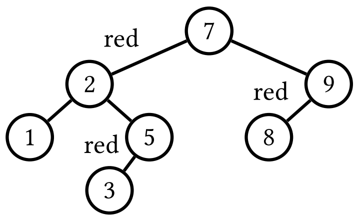
   (b) Draw the 2-3 tree corresponding to the following left-leaning red-black tree.

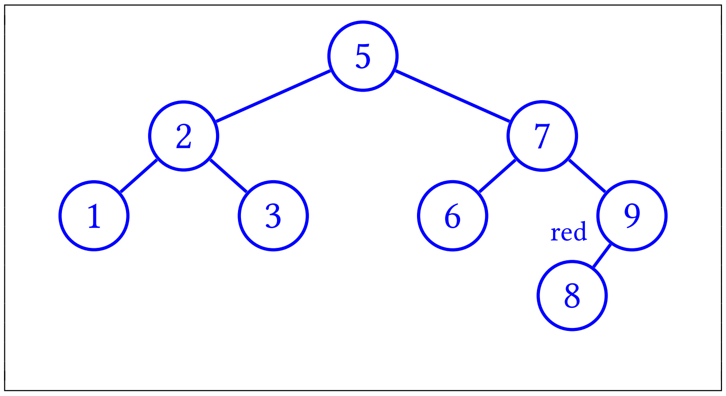




Each red link in the left-leaning red-black tree connects two values in a 3-node (2 keys, 3 non-null children). The 2-nodes (1 key, 2 non-null children) can be copied over directly. We can check our work by making sure that each leaf node is the same depth from the root since we know that 2-3 trees are perfectly balanced.

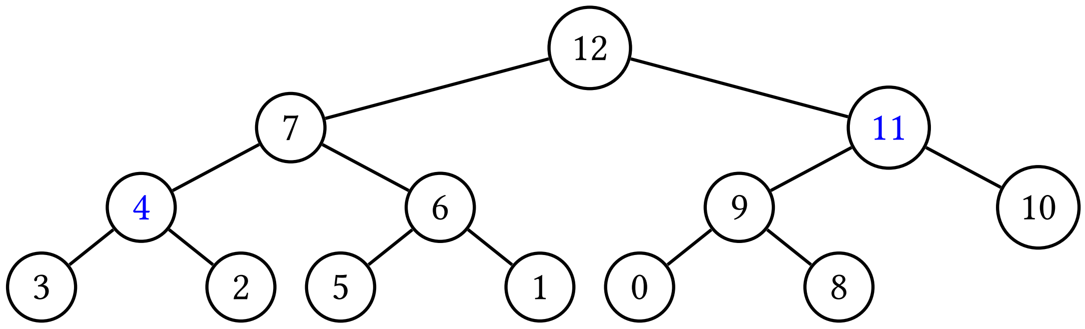
   (c) Draw the left-leaning red-black tree after inserting 6. Label red edges **red**.





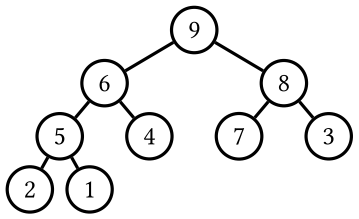
Inserting into a left-leaning red-black tree directly is somewhat tricky and prone to error when rotating and color flipping by hand. The better approach is to use the 1-1 correspondence between 2-3 trees and LLRB trees. We can use the 2-3 tree drawn above as a starting point, insert 6 into the 2-3 tree, and then convert the result back to an LLRB tree. Inserting 6 results in splitting the 3-5-6 node and promoting 5, which results in splitting the 2-5-7 root node and promoting 5 again.

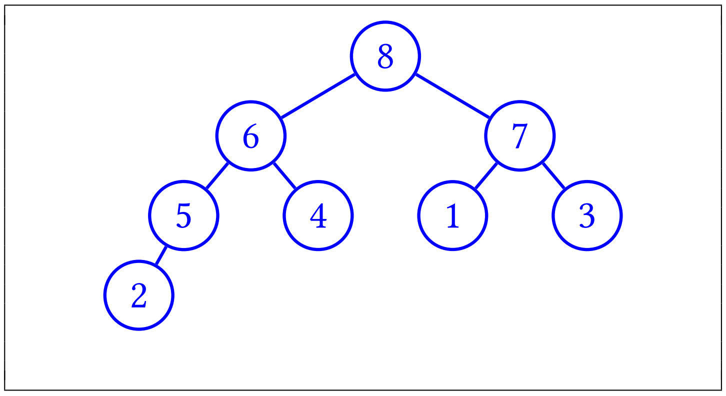
(d) Fill-in the blank nodes of the following binary max-heap with valid **integer values**.



Use the max-heap invariant: the parent node value must be greater-than or equal-to both of its child node values.

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





1. Swap the root (node-with-value-9) with the rightmost leaf on the last level (node-with-value-1).
2. Remove the rightmost leaf on the last level. This leaf has no children.
3. Sink the new root node to its proper place in the heap to restore invariants.

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.

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

The (informal) definition of $\Theta(N)$ states that the order of growth of the function is bounded above and below by a linear function, N, for all large inputs. This implies $O(N)$, that the order of growth of the function is bounded above by a linear function, N, for all large inputs. This also implies $\Omega(N)$ for the lower bound. If the order of growth of a function is bounded above by a linear function, it's also bounded above by a quadratic function, N^2, for all large inputs, so it is also $O(N^2)$. We can apply the same logic for lower bounds too to get $\Omega(\log N)$ and $\Omega(1)$.

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) $\Theta(N^2)$

```
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;
            }
        }
    } }
```

Use one of the two methods for analyzing iteration: exact count of operations or geometric explanation. Suppose our cost model is the number of != operations. The first `N / 2` iterations of the `for` loop form a right triangle with base of length `N / 2`. The remaining `N / 2` iterations of the `for` loop form another right triangle with base of length `N / 2`. Individually, each of these triangles account for about $N^2/4$ work. Since there are two triangles, the overall work is about $N^2/2 \in \Theta(N^2)$.

(c) $\Theta(N \log N)$

```
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);
    } }
```

The analysis for this is similar to merge sort. Ignoring recursive calls, the initial call to `recursion` takes linear time with respect to the original input, N. Since the problem is being broken into two halves of equal size, the two calls on the next level of the tree recursion each perform $N/2$ work. When summed together, these calls together take linear time with respect to the original input, N. The pattern continues for $\log_2 N$ levels. Since each level contributes N work, the sum of all $\log_2 N$ levels gives us the overall work as $\Theta(N \log N)$.

(d) $\Theta(N^2)$

```
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);
} }
```

Let's apply the same strategy of ignoring the work contributed by recursive calls at first. Given a circular `ArrayDeque` (without resizing), each deque operation takes constant time, so the initial call to `reverseDeque` takes linear time with respect to the original deque of size N. Before making the recursive call, we remove the first item from the deque so that the first recursive call to `reverseDeque` only handles a deque of size $N - 1$. This pattern continues until the deque is empty, resulting in the following summation.

$$N + (N - 1) + (N - 2) + \cdots + 3 + 2 + 1 \in \Theta(N^2)$$

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`	$\Theta(1)$	$\Theta(R)$

For `DataIndexedCharMap`, the best case and worst case runtimes are constant since each character in the alphabet is given its own array index.

The best case for `contains` on each data structure is constant time since the search character can be found at the "front" of the data structure. In a hash table, this is the front of a bucket. For a binary search tree and LLRB tree, this is the root node. For unordered linked nodes, this could happen to be in the first node.

A worst case scenario for `contains` on unordered linked nodes occurs when the search character is at the last node. For a hash table, we can apply this same logic to searching in a bucket if all of the R characters collide into a single bucket. For a binary search tree, imagine inserting into the map in order, which results in a spindly structure. Searching for the last character in this spindly structure will take linear time with respect to the size of the alphabet. The difference with a left-leaning red-black tree is that it performs rotations and color flips to balance itself to logarithmic height, so searching is also limited to logarithmic runtime.

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** $\Theta(L\sqrt{R})$

Searching for a string in an R-way `TrieSet` depends on the length of the string, not the total number of strings in the trie. For each character in the string, we need to find its value (the next node) in the `WeirdCharMap next`.

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

```
function TRIESORT(stringsToSort)
    $t \longleftarrow$ 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()
    $x \leftarrow$ 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. __________

We know that TRIESORT works except on inputs containing duplicate strings. This is because our trie is a set: it contains a `boolean isKey` to mark whether or not a prefix (represented by each node) is actually a word in the set. But the problem with storing a boolean is that it only tells us whether or not the word is in the set, not how many copies of it need to be sorted.

The resulting abstract data type is called a **multiset**: a modification of the set ADT that allows for multiple instances of each of its elements. Interestingly, the runtime for TRIESORT is $O(LN)$. This can be potentially better than merge sort. TRIESORT is the data structure analogue to the radix sort algorithm we'll see later in the course.