UC Berkeley – Computer Science
CS61B: Data Structures

Midterm #2, Fall 2020

This test has 8 questions worth a total of 960 points, and is to be completed in 130 minutes. The exam is closed book, except that you are allowed to use unlimited written notes. 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 your name. You may do this before the exam begins.**

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

Signature: _____

#	Points	#	Points
1	55	5	60
2	60	6	150
3	50	7	175
4	230	8	180
		TOTAL	960

Name: _____

SID: _____

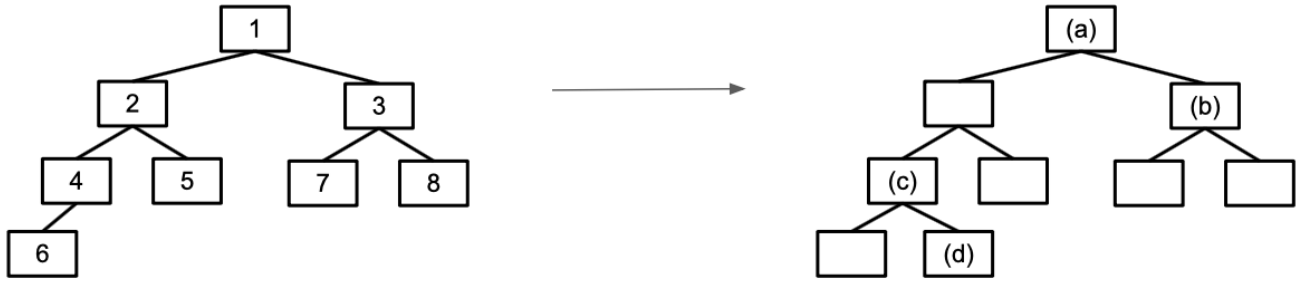
Class ID #_____: fa20-s_____

Optional. Mark along the line to show your feelings on the spectrum between ☹ and ☺.

Before exam: [☹_____☺].
After exam: [☹_____☺].

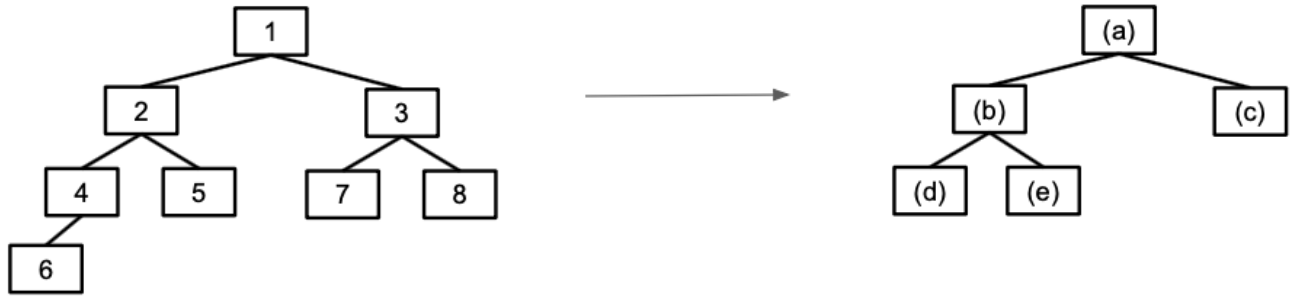
1. Heap Operations (55 points)

a) **(20 points).** Given the min-heap to the left below, suppose we call **insert(0).** The resulting min-heap will have 9 elements, as shown to the right below. What numbers are in spots a-d?



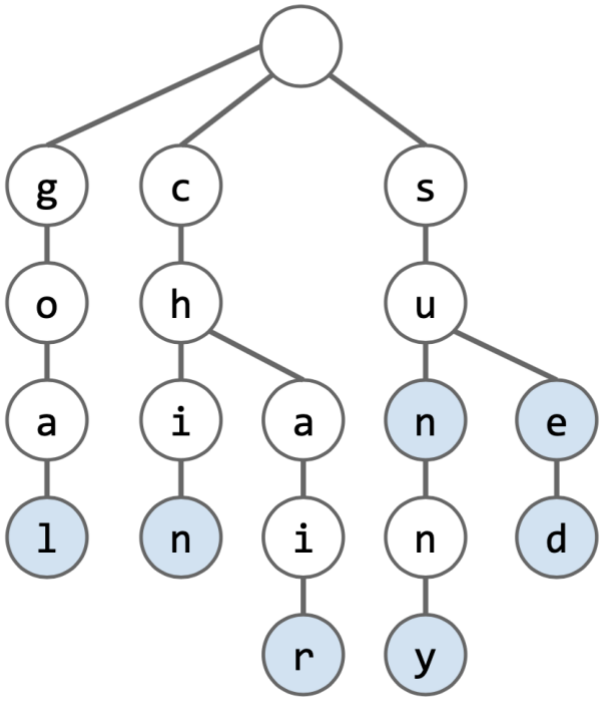
(a) ○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8
(b) ○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8
(c) ○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8
(d) ○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8

b) **(35 points)** Given the same initial min-heap from before (redrawn to the left below), suppose we call **removeMin three** times. The resulting min-heap will have five elements, as shown to the right below. What number is in each spot?



(a) ○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8
(b) ○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8
(c) ○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8
(d) ○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8
(e) ○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8

2. Trie Operations (60 points). Given the Trie below, answer the following questions. Recall that **isKey** is **true** for blue nodes and **isKey** is **false** for white nodes.



a) **(15 points)** If we call **keysWithprefix("su")**, what keys will get returned? Recall that **keysWithPrefix** finds all keys with the given prefix.

☐ sun ☐ chin ☐ sued ☐ goal ☐ sue ☐ sunn ☐ chair ☐ sunny ☐ n ☐ y ☐ e ☐ d

b) **(15 points)** What key, when added, creates the **fewest** nodes? If multiple keys produce the minimum, select all that apply.

☐ goals ☐ in ☐ goat ☐ soccer ☐ chain ☐ cheese ☐ at ☐ camel

c) **(30 points)** Suppose we add the function **shortestPrefixOf** that accepts a **String s** and returns the shortest key in the Trie that is a prefix of **s**. Calling **shortestPrefixOf("goalie")** on the Trie would return **"goal"**. A string is a prefix of itself, e.g. calling **shortestPrefixOf("goal")** would return **"goal"**. Assume **shortestPrefixOf** has been implemented correctly and as efficiently as possible. By efficient, we mean that it traverses no more nodes than necessary. What **String s** below would **maximize** the number of nodes needed to be traversed? If multiple words produce the maximum, select all that apply.

☐ sun ☐ sunnyday ☐ sunny ☐ chin ☐ sue ☐ goal ☐ sued ☐ sunn ☐ chair ☐ chairlift

3. Heap Iterator (50 points). Suppose we have an implementation of the **MinPQ** interface from lecture given below. This specific implementation only allows **Integer** items. Assume the priority of an **Integer** is itself. You should not assume that **XMinPQ** uses a heap, i.e. the underlying implementation is unknown and not provided.

```
public class XMinPQ implements MinPQ<Integer>, Iterable<Integer> {
    public void add(Integer x) {...}
    public Integer getSmallest() {...}
    public Integer removeSmallest() {...}
    public int size() {...}
    public Iterator<Integer> iterator() {
        return new XIterator();
    }
    private class XIterator implements Iterator<Integer> {
        public int x;
        public XIterator() {
            x = removeSmallest();
        }
        public boolean hasNext() {
            return x > 0;
        }
        public Integer next() {
            x = x - 1;
            return removeSmallest();
        }
    }
}
```

What is the result of the code below? **Reminder: The XMinPQ does not necessarily use a heap,** i.e. we know that it implements the priority queue abstract data type, but we do not know how.

```
public static void main(String[] args) {
    XMinPQ mpq = new XMinPQ();
    mpq.add(4);
    mpq.add(1);
    mpq.add(5);
    mpq.add(2);
    mpq.add(6);
    mpq.add(3);
    mpq.add(8);
    mpq.add(7);
    for (int i: mpq) {
        System.out.print(i + " "); // line 1
    }
    System.out.println();
    for (int i: mpq) {
        System.out.print(i + " "); // line 2
    }
}
```

Line 1: _____

Line 2: _____

4. Asymptotics. (230 points). Consider the code below. For each, give the **worst case** runtime in Theta notation as a function of N.

a) loops (30 points)

```
public static void loops(int N) {
    for (int i = 1; i < 2; i += 1) {
        for (int j = 2; j < 4; j = j * j)    {
           for (int k = 3; k < N; k = k * 2) {
               for (int m = 4; m < 8; m = m * m * m) {
                   System.out.println("hello");
               }
           }
        }
    }
}
```

○ Θ(1) ○ Θ(log log N) ○ Θ((log N)^2) ○ Θ(log N) ○ Θ(N) ○ Θ(N log N)
○ Θ(N^2) ○ Θ(N^2 log N) ○ Θ(N^3) ○ Θ(N^3 log N) ○ Θ(N^4) ○ Θ(N^4 log N)
○ Worse than Θ(N^4 log N) ○ Never terminates (infinite loop)

b) addToSet (40 points)

```
public static void addToSet(int N) {
    HashSet<RandomThing> m = new HashSet<>();
    for (int i = 0; i < N; i += 1) {
        // getNextRandomThing takes constant time
        RandomThing rt = getNextRandomThing();
        m.add(rt);
    }
}
```

Reminder! Throughout all 6 parts (a through f) of this asymptotics problem, you are giving the **worst case** in theta notation.

○ Θ(1) ○ Θ(log log N) ○ Θ((log N)^2) ○ Θ(log N) ○ Θ(N) ○ Θ(N log N)
○ Θ(N^2) ○ Θ(N^2 log N) ○ Θ(N^3) ○ Θ(N^3 log N) ○ Θ(N^4) ○ Θ(N^4 log N)
○ Worse than Θ(N^4 log N) ○ Never terminates (infinite loop)

c) nestedSum(nestedSum(x)) (40 points)

```
public static List<Integer> nestedSum(List<Integer> x) {
    List<Integer> newInts = new ArrayList<>();

    for (int i : x) {
        for (int j : x) {
            newInts.add(i + j);
        }
    }
    return newInts;
}


List<Integer> x = getIntegers(); // getIntegers returns a List of Integers
int N = x.size();

// Give the runtime for the following call to nestedSum(nestedSum(x))
List<Integer> x2 = nestedSum(nestedSum(x));
```

○ Θ(1) ○ Θ(log log N) ○ Θ((log N)^2) ○ Θ(log N) ○ Θ(N) ○ Θ(N log N)
○ Θ(N^2) ○ Θ(N^2 log N) ○ Θ(N^3) ○ Θ(N^3 log N) ○ Θ(N^4) ○ Θ(N^4 log N)
○ Worse than Θ(N^4 log N) ○ Never terminates (infinite loop)

d) g(N, 1) (40 points)

```
private static void g(int N, int ss) {
    if (ss > N) {
        return;
    }

    for (int i = 0; i < N; i += ss) {
        System.out.print(i + " ");
    }

    System.out.println();
    g(N, ss * 2);
}

// Give the runtime for the following call:
g(N, 1);
```

○ Θ(1) ○ Θ(log log N) ○ Θ((log N)^2) ○ Θ(log N) ○ Θ(N) ○ Θ(N log N)
○ Θ(N^2) ○ Θ(N^2 log N) ○ Θ(N^3) ○ Θ(N^3 log N) ○ Θ(N^4) ○ Θ(N^4 log N)
○ Worse than Θ(N^4 log N) ○ Never terminates (infinite loop)

e) g2(N, 1) (40 points)

```
private static void g2(int N, int ss) {
    if (ss > N) {
        return;
    }

    for (int i = 0; i < N; i += ss) {
        System.out.print(i + " ");
    }

    System.out.println();
    g2(N, ss * 2);
    g2(N, ss * 2);
}

// Give the runtime for the following call:
g2(N, 1);
```

○ Θ(1) ○ Θ(log log N) ○ Θ((log N)^2) ○ Θ(log N) ○ Θ(N) ○ Θ(N log N)
○ Θ(N^2) ○ Θ(N^2 log N) ○ Θ(N^3) ○ Θ(N^3 log N) ○ Θ(N^4) ○ Θ(N^4 log N)
○ Worse than Θ(N^4 log N) ○ Never terminates (infinite loop)

f) func (40 points)

```
public void func(int N) {
    int Q = 0;
    for (int i = 1; i < N; i *= 3) {
        Q += 1;
    }
    disco(Q);
}
public void disco(int Q) {
    if (Q == 0) {
        return;
    }
    for (int i = 0; i < Q; i += 1) {
        System.out.println("hi");
    }
    disco(Q - 1);
}
```

○ Θ(1) ○ Θ(log log N) ○ Θ((log N)^2) ○ Θ(log N) ○ Θ(N) ○ Θ(N log N)
○ Θ(N^2) ○ Θ(N^2 log N) ○ Θ(N^3) ○ Θ(N^3 log N) ○ Θ(N^4) ○ Θ(N^4 log N)
○ Worse than Θ(N^4 log N) ○ Never terminates (infinite loop)

5. Compress (60 Points)

Consider the **UnionFind** class as covered in lab.

```
public class UnionFind {
    private int[] parent;
    public int sizeOf(int v1) {
        int root = find(v1);
        return -1 * parent[root];
    }
    public boolean isConnected(int v1, int v2) {
        return find(v1) == find(v2);
    }
    public void connect(int v1, int v2) { ... }
    // Path compression is NOT implemented
    public int find(int v1) { ... }
}
```

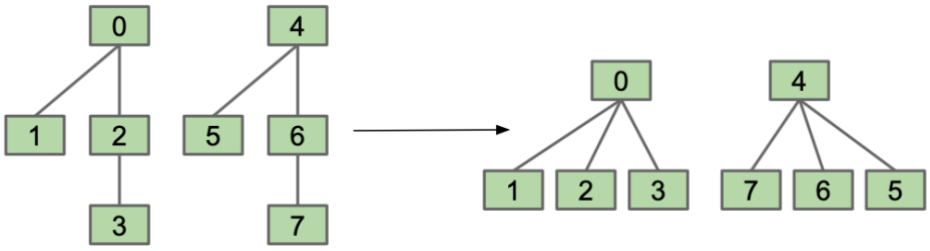
Suppose we add the method **compress** to the **UnionFind** class, which compresses the height of each tree to 1 by connecting every vertex to its respective root. For instance, suppose we have the parent array below

```
{-4, 0, 0, 2, -4, 4, 4, 6}
```

If we call **compress(),** the parent array would become

```
{-4, 0, 0, 0, -4, 4, 4, 4}
```

as vertices 3 and 7 connect to their respective roots. A visualization of this process can be seen below:

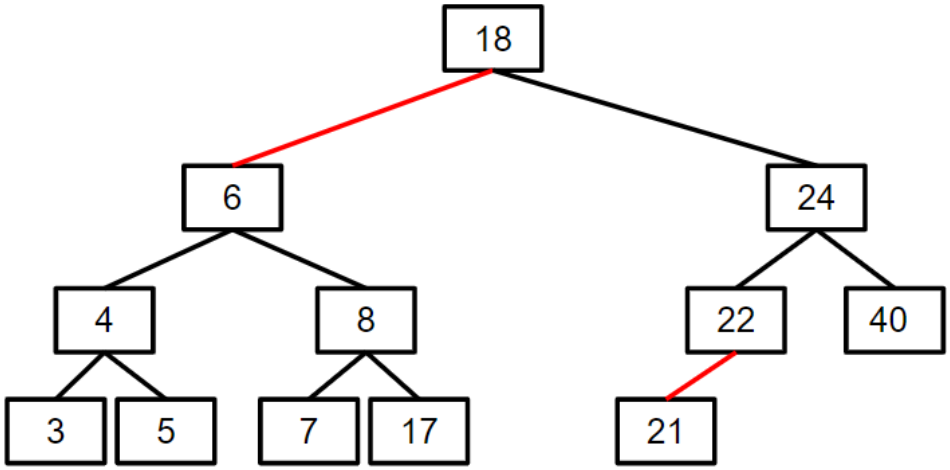


Note that the values of vertices 0 and 4 stay the same because they are roots. Also note that the values of vertices 1, 2, 5, and 6 happen to stay the same because they are already connected to their respective roots! Fill in the **compress** method for the **UnionFind** class below. Do **not** change the skeleton provided. No credit will be given to solutions which do not use the given skeleton, even if correct.

```
public class UnionFind {
    ...
    public void compress() {
        for (_____________________________) {
            if (_____________________________) {
                _____________________________;
            }
        }
    }
}
```

6. LLRB Insertions (150 points)

We will be using this left learning red-black tree (LLRB) for the following questions.



Recall from lecture and discussion that when we insert items into a LLRB tree, we sometimes need to perform balance operations. Recall that our balance operations are: **rotateRight**, **rotateLeft**, and **colorFlip**.

For each of the following prompts, give a single **int** value which, when inserted into the LLRB above, causes exactly the requested balance operation to occur. **You may only insert int values into the LLRB tree** and only a **single** balance operation may occur.

If multiple possible values would work, choose the **smallest, positive value**. **If there is no value that results in the given balance operation occurring by itself, simply write "None" in the answer box.**

Your inserted values should result in only a **single** balance operation. For example, if your value causes a sequence of rotations or color flips, it is not a valid answer.

Each question is independent. For each sub part, the LLRB tree you're inserting into is the exact, untouched LLRB you see above.

Reminder: If there are multiple values that could work, give the smallest positive value. That is, if 4 and 5 are both valid answers to a problem, you must give 4 for full credit.

a) (20 points) A single **rotateLeft** operation.	**int** value to insert:
b) (20 points) A single **rotateRight** operation.	**int** value to insert:
c) (20 points) A single **colorFlip** operation.	**int** value to insert:

Now consider the **corresponding 2-3 tree** for the LLRB shown above. Answer the following questions. The last one is particularly difficult.

d) (30 points) What is the **minimum** number of values that we must add to the **corresponding 2-3 tree** to increase its height? For this problem, values may be **any real value** (e.g. 3.6), not necessarily positive integers.	number of values:
e) (60 points) What is the **maximum** number of values that we can add to the corresponding 2-3 tree **without** increasing its height? For this problem, values may be **any real value** (e.g. 3.6), not necessarily positive integers.	number of values:

7. By The Numbers (175 points). For each scenario below, give the minimum and maximum. **If there is no min or max write "None"**.

Recall that the height of a tree is the number of links from the root to the farthest leaf, i.e. a tree with 2 nodes has height 1. Recall that the depth of a node in a tree is the number of links from the root to that node, i.e. the root of a tree has depth 0, its children have depth 1, and so forth. Assume for the entirety of this problem that duplicates in BSTs are not allowed.

a) (25 points) The number of nodes in a non-empty tree which is both a BST and a max-heap.

Min: _____ Max: _____

b) (25 points) The height of a BST with 7 nodes where the median is the root.

Min: _____ Max: _____

c) (25 points) The height of a WeightedQuickUnion object with 10 items, where all of the objects are connected.

Min: _____ Max: _____

d) (25 points) The depth of the median value in a heap of 15 objects, none of which are duplicates.

Min: _____ Max: _____

e) (25 points) The depth of the median value in a BST of 15 objects.

Min: _____ Max: _____

f) (25 points) The length of the longest list in a hash table with 10 buckets and 20 items, assuming each bucket is represented as a list.

Min: _____ Max: _____

g) (25 points) The height of the tallest tree in a hash table with 10 buckets and 7 items, assuming each bucket is stored as an LLRB.

Min: _____ Max: _____

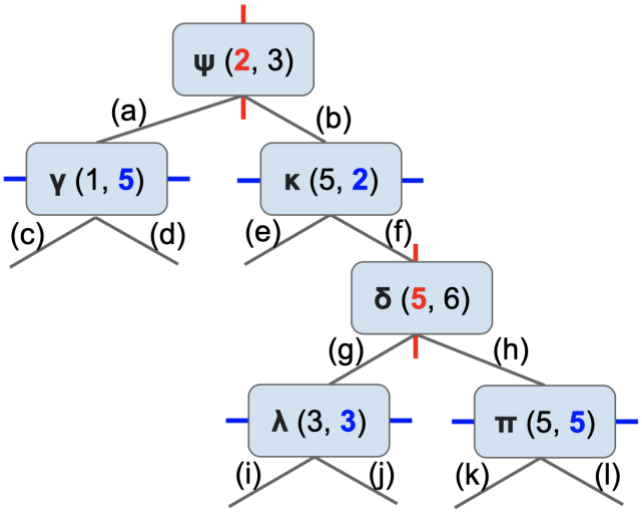
8. Window (180 points)

a) (30 points) Suppose we have a class called **Point** which represents a point in 2D space. Fill in the function **isInside** such that it returns **true** if the point is inside the window bounded by the given x and y coordinates. If a point lies exactly on the boundary, then it should be included.
Assume **x1** will *always* be less than or equal to **x2** and **y1** will *always* be less than or equal to **y2**.

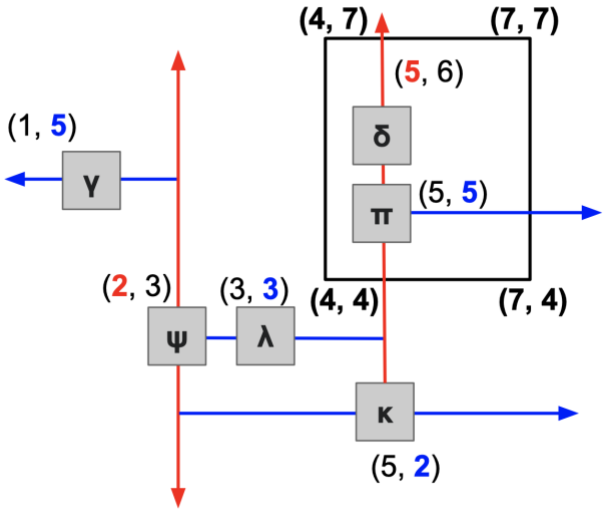
```
public class Point {
    public double x;
    public double y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public boolean isInside(double x1, double x2, double y1, double y2) {



    } // our code is 1 line, yours may be more or fewer
}
```

For the rest of this problem, we'll consider how we can use a KD-Tree to efficiently find all points in space that are inside a given window. In part (c), you'll be writing the function **List<Points> window(double x1, double x2, double y1, double y2)** which returns all points in the KD-Tree that are inside the given window. To see an example, suppose we are using the KD-Tree below and call **window(4, 7, 4, 7)**. The points returned are **(5, 5)** and **(5, 6).** Your method should be **efficient** and should **prune** branches that **definitely do not contain any points be in the given window**.

Tree Representation:



Graph Representation:



b) (50 points) Before we dive into the code, let's first understand the process of pruning a bit better. Using the KDTree above, which branches, if any, can be pruned in the call **window(4, 7, 4, 7)**? Note if we prune a branch higher up on a particular subtree, we don't prune the lower branches because we never consider them, i.e. if (h) can be pruned, (k) and (l) are **not** pruned. Also note that branches that do not point to a Node, e.g. (d) can be pruned.

☐ (a) ☐ (b) ☐ (c) ☐ (d) ☐ (e) ☐ (f) ☐ (g) ☐ (h) ☐ (i) ☐ (j) ☐ (k) ☐ (l)
☐ No branches can be pruned

c) (100 points) Fill in the **window** function so that it works as described above. You may use a **maximum of 3 lines** of code in the body of **window**. Your implementation for **windowHelper** *must* fit within the skeleton provided. Partial credit will be given to solutions that work correctly, but which do not prune correctly. If you think it will be useful, you may assume that the **isInside** method in the **Point** class works correctly even if you did not complete part a.

```
import java.util.*;

private static final boolean HORIZONTAL = false; // we compare x coordinates
private static final boolean VERTICAL = true; // we compare y coordinates
private Node root;

private class Node {
    private Point p;
    private boolean orientation; // true == VERTICAL, false == HORIZONTAL
    private Node leftChild;  // also refers to the "down" child
    private Node rightChild; // also refers to the "up" child

    public Node(Point givenP, boolean o) {
        p = givenP;
        orientation = o;
    }
}

    public List<Point> window(double x1, double x2, double y1, double y2) {
            ____________________________________________________
            ____________________________________________________
            ____________________________________________________
    } // You may use a maximum of 3 lines

    private void windowHelper(Node n, List<Point> points, double x1, double x2,
double y1, double y2) {
        if (_________________________________) {
            return;
        }
        if (_________________________________) {
            ________________________;
        }
        if (_________________________________) {
            if (_________________________________) {
                _________________________________;
            }
            if (_________________________________) {
                _________________________________;
            }
        } else {
            if (_________________________________) {
                _________________________________;
            }
            if (_________________________________) {
                _________________________________;
            }
        }
    }
```