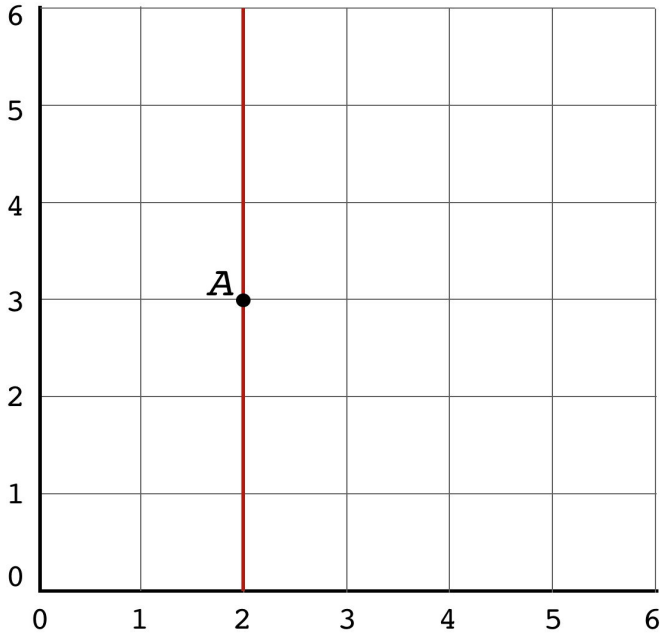
EXERCISE 1: Kd-Trees

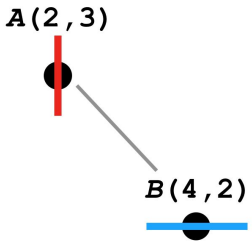
(a) Draw the Kd-tree that results from inserting the following points:

`[A(2, 3), B(4, 2), C(4, 5), D(3, 3), E(1, 5), F(4, 4), G(1, 1)]`

Draw each point on the grid, as well as the vertical or horizontal line that runs through the point and partitions the plane, or a subregion of it.

Note: While inserting, go left if the coordinate of the inserted point is less than the coordinate of the current node. Go right if it is greater than ***or equal***.



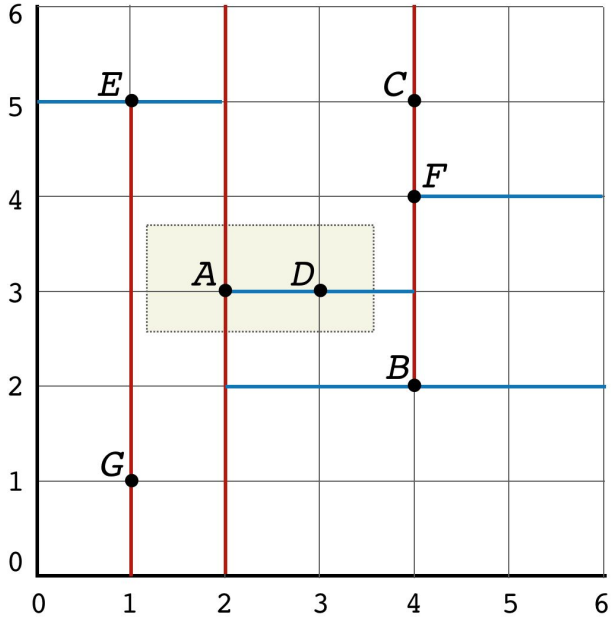


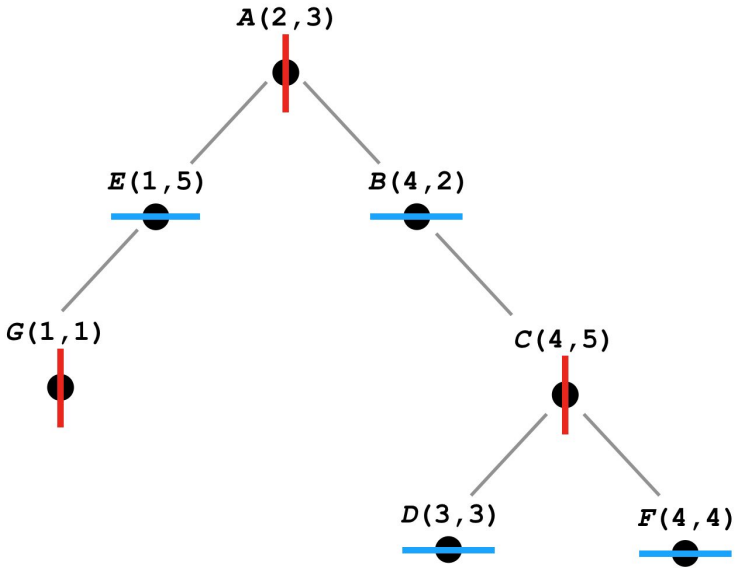
(b) Give each point's bounding rectangle.

`A(2, 3)`	$[(-\infty, -\infty), (+\infty, +\infty)]$
`B(4, 2)`	
`C(4, 5)`	
`D(3, 3)`	
`F(4, 4)`	$[(4, 2), (+\infty, +\infty)]$
`E(1, 5)`	$[(-\infty, -\infty), (2, +\infty)]$
`G(1, 1)`	

(c) Number the tree nodes according to the visiting order when performing a *range query* using the rectangle shown below. Label pruned subtrees with ✘.

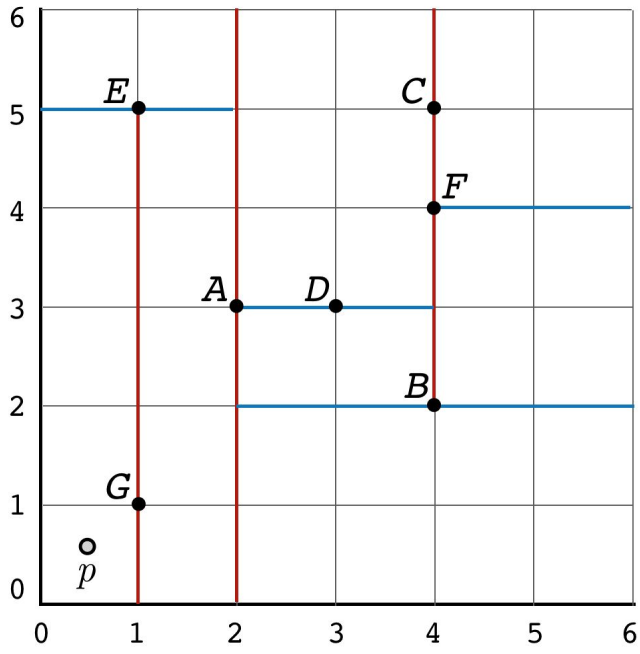
Remember. The range search algorithm recursively searches in both the left and right subtrees unless the bounding rectangle of the *current* node does not intersect the query rectangle.

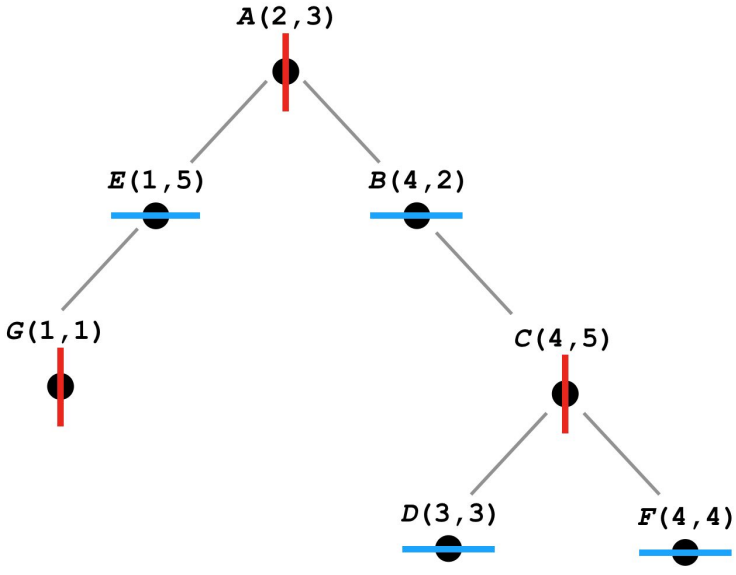




(d) Number the tree nodes according to the visiting order when performing a *nearest neighbor (NN) query* using the point p shown below. Label pruned subtrees with ✘.

Remember. The NN algorithm recursively searches in *both* the left and right subtrees unless the distance between *p* and the bounding rectangle of the *current* node is larger than the distance between *p* and the nearest point found so far.





EXERCISE 2: Graph Traversal

Note. You can also use the online version of this exercise, which allows testing your code and receiving instant feedback:

https://stepik.org/lesson/217879

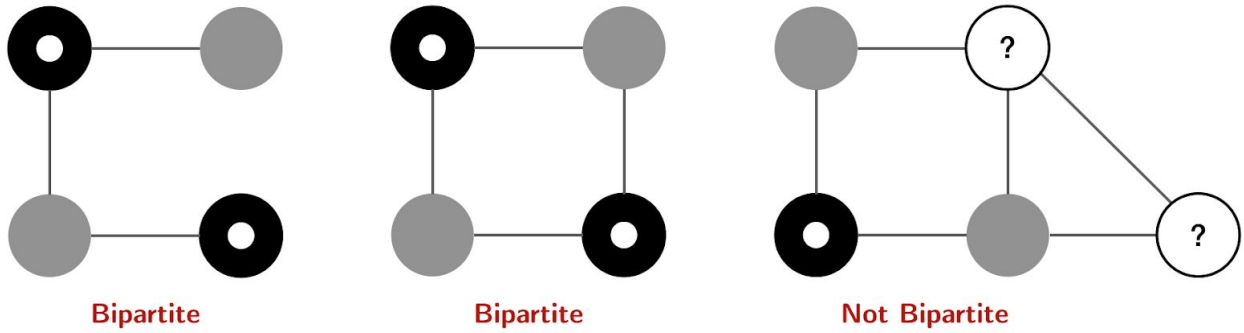
The online version also has extra exercises for the bored!

(a) *Cycle Detection.* Consider the following Breadth-First Search code. What modifications should be made in order for the `hasCycle()` method to return `true` if the graph has a simple cycle and `false` otherwise? Assume that the graph is simple, connected and undirected.

Def. A *cycle* is a path with at least one edge whose first and last vertices are the same. A *simple cycle* is a cycle with no repeated edges or vertices (except the requisite repetition of the first and last vertices).

```
1  private static boolean hasCycle(Graph G) {
2        boolean[] marked = new boolean[G.V()];
3        int[] edgeTo = new int[G.V()];
4
5        Queue<Integer> q = new Queue<Integer>();
6        marked[0] = true;
7        q.enqueue(0);
8
9        while (!q.isEmpty()) {
10             int v = q.dequeue();
11             for (int w : G.adj(v)) {
12                   if (!marked[w]) {
13                         edgeTo[w] = v;
14                         marked[w] = true;
15                         q.enqueue(w);
16                   }
17             }
18       }
19 }
```

(b) *Testing Bipartiteness.* A graph is *bipartite*, if the vertices can be colored using two colors, such that no two adjacent vertices have the same color. Modify the following BFS code in order to check if the given graph is bipartite. Assume that the graph is simple, connected and undirected.



Bipartite Bipartite Not Bipartite

```
1  private static boolean isBipartite(Graph G) {
2        boolean[] marked = new boolean[G.V()];
3        marked[0] = true;
4
5        Queue<Integer> q = new Queue<Integer>();
6        q.enqueue(0);
7
8        while (!q.isEmpty()) {
9              int v = q.dequeue();
10             for (int w : G.adj(v)) {
11                   if (!marked[w]) {
12                         marked[w] = true;
13                         q.enqueue(w);
14                   }
15             }
16       }
17 }
```