Precept Outline

- Review of Lectures 13 and 14:
  - Hash Tables
  - K-d Trees

Relevant Book Sections

- Book chapters: 3.4

A. Review: Hash Tables + K-d Trees

Your preceptor will briefly review key points of this week's lectures.

B. Hash Tables

Solve exercises 1 through 6 in the Hash Tables Ed lesson.

C. K-d Trees

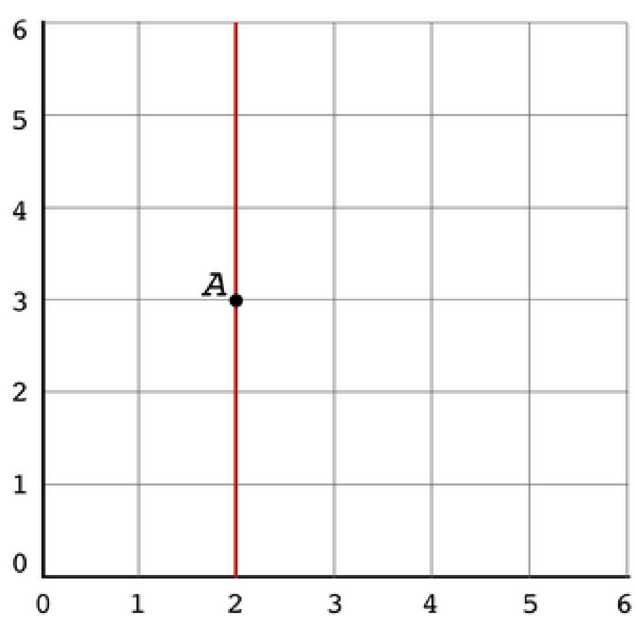
Draw the Kd-tree that results from inserting the following points:

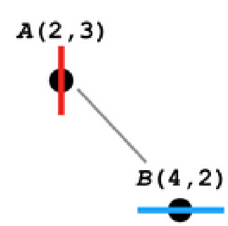
A	B	C	D	E	F	G
$(2,3)$	$(4,2)$	$(4,5)$	$(3,3)$	$(1,5)$	$(4,4)$	$(1,1)$

Additionally, 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 thereof.

Recall: 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**.

Use the images below to draw your grid/tree.



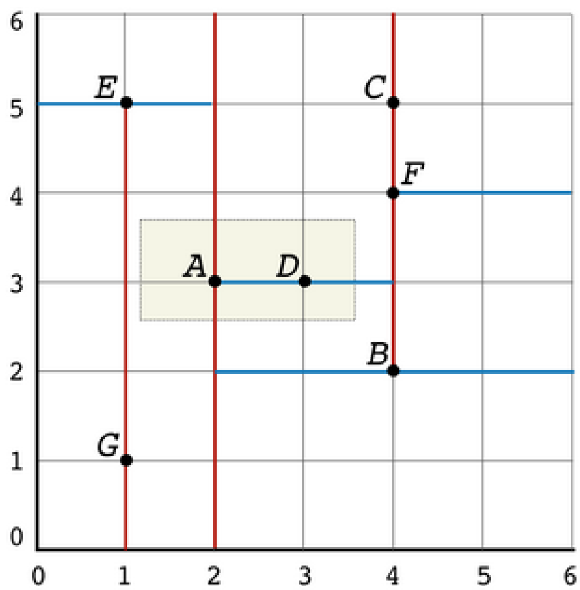


Determine each point's bounding box and fill them in the table below.

A	$[-\infty, \infty] \times [-\infty, \infty]$
B	
C	
D	
E	
F	
G	

Number the (non-null) nodes in the sequence they are visited by a *range query* with the rectangle shown below. Which subtrees are pruned? (Some null subtrees may be pruned, and some may not be.)

Remember. The range search algorithm recursively searches in both the left and right subtrees unless the bounding box of the *current* node does not intersect the query rectangle. If both do, our convention is to visit the left one first.



D. Assignment Overview: K-d Tree

Your preceptor will introduce and give an overview of your fourth assignment. Please don't hesitate to ask questions!

Summary of the assignment.

- Implement a `PointST` class, a symbol table whose keys are two-dimensional points. Don't overthink this part of the assignment. It's worth 25% of the grade but should take much less than that proportion of time – pay attention to the lax performance requirements. It has nothing to do with k-d trees!
- Implement a `KdTreeST` class, a 2d-tree to implement a symbol table with two-dimensional point keys.
- There is also a `readme.txt` where you have to perform a simple timing experiment and answer one theoretical question about pruning.

Here are a few implementation tips:

- **Avoid duplicating code**. Use private helper methods to achieve that. For example, a `compare` helper method can be used to encapsulate the logic of checking the orientation and doing the comparison using the correct coordinate. This makes the code cleaner and more concise.
- **Avoid reinventing the wheel**. Do not re-implement the `RectHV` and `Point2D` methods!
- **Use the visualizer classes**. They are there to help, and are extremely useful debugging tools!
- **Duplicate points**. The `KdTreeST` is a symbol table, so duplicates are not allowed. Make sure to handle that in the `put` method! The expected behavior when inserting a a key that has been inserted before is to replace the old value with the new value.
- **Timing**. To count the number of calls per second:
  - Use `put()` to build the tree but don't time this part.
  - Time so that the method is repeatedly called for at least 2 seconds.
  - Example: If the method is called N times in 4.62 seconds, then report $\frac{N}{4.62}$ as the number of calls per second.
  - Expect a faster method to have a higher number of calls per second.
- **Recursion**.
  - If your recursive method returns a value, make sure to catch the return value and use it whenever you make a recursive call.
  - If your recursive method receives an argument that needs to be updated, note that the following works:

    ```
    void myMethod(Type1 arg1, Type2 result) {
        result.setX(someValue);
        ...
    }
    ```

    But the following does not work:

    ```
    void myMethod(Type1 arg1, Type2 result) {
        result = new Type2(someValue);
        ...
    }
    ```

  - In the first example, `result` is a reference, and the method is changing X in the object referenced by `result`. In the second example, `result` is a reference to the object passed as an argument. By setting `result = ...`, we make the reference `result` point to a new object but don't change the original object.