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

Part 1: Linear probing vs. separate chaining

Consider the following sequence of (integer) insertions into an empty hash table of capacity 11, where the hash function `h(i)` returns `i % 11`:

`15, 90, 53, 100, 34, 65, 20`

Suppose that the symbol table is implemented as a *linear-probing* hash table. Which positions of the `keys[]` array contain null values after all 7 insertions?

Suppose we perform the same sequence of insertions with the same hash function, but use a *separate-chaining* table instead. What linked list sizes are present in the table after all 7 insertions?

Part 2: Attacking "bad" hashes

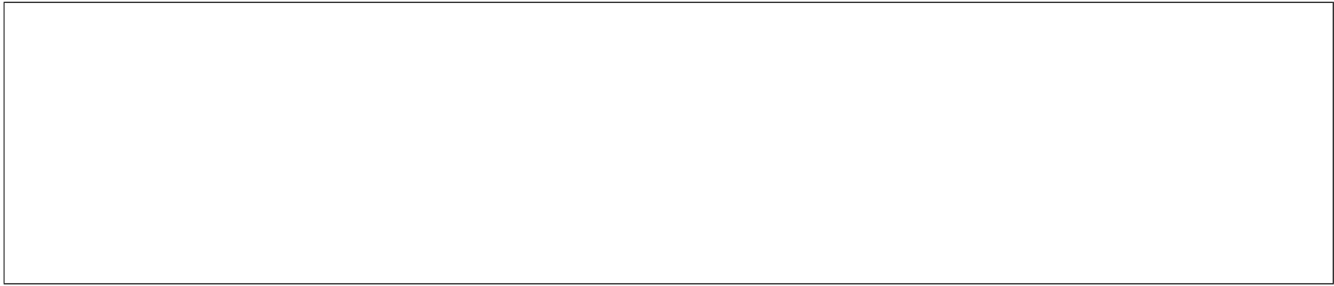
What is the worst-case runtime order of growth of a sequence of n insertions into a linear-probing hash table of size n? What about a separate-chaining hash table? Assume that the arrays are not resized.

For each of the hash functions h_n below, exhibit a set of n integers $x_1, \ldots, x_n$ which all hash to the same value. (Recall that $x \pmod n$ is the remainder of the division of x by n, which is value of `x % n` when `x` is non-negative.)

(a) $h_n(x) = x \pmod n$.
(b) $h_n(x) = x(x-1)(x-2)\cdots(x-n+1) \pmod n$.
(c) $h_n(x) = 2x + 5 \pmod n$. (You may assume that division and multiplication work as expected $\bmod n$: if $2y \pmod n$ and $z \pmod n$ are equal, then so are $y \pmod n$ and $z/2 \pmod n$.)

Finally, give a set of n *strings* that all hash to the same value under the following hash function (which was built into early versions of Java):

```
1 public int hashCode() {
2   int hash = 0;
3   int skip = Math.max(1, length() / 8);
4   for (int i = 0; i < length(); i += skip)
5     hash = (hash * 37) + charAt(i);
6   return hash;
7 }
```



C. K-d Trees

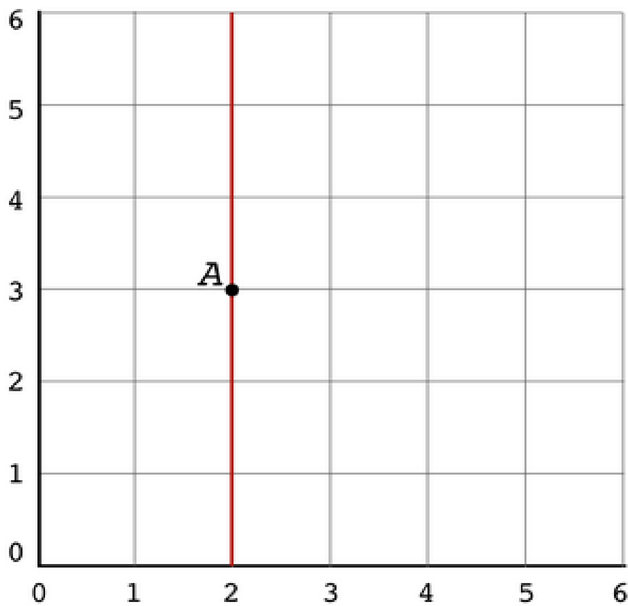
Draw the 2d-tree that results from inserting the following points:

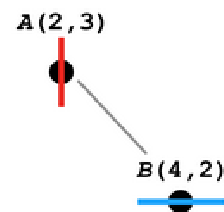
A	B	C	D	E	F	G
$(2,3)$	$(4,2)$	$(4,5)$	$(3,3)$	$(1,5)$	$(4,4)$	$(1,1)$

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: We take the convention that while inserting, we move left if the coordinate of the inserted point is less than the coordinate of the current node; and go right if it is greater than **or equal**.

Use the images below to draw your grid/tree.



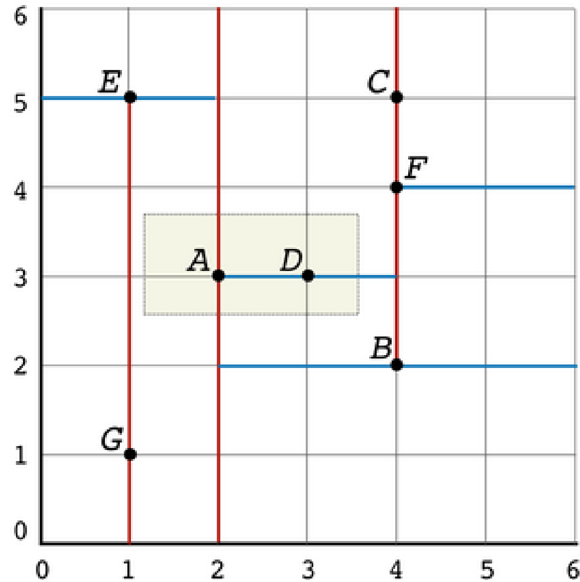


Determine each point's bounding box and fill them into the table below.

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!

Here are a few implementation tips:

- **Avoid duplicating code**. Use private helper methods to do so. 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 instance methods of `RectHV` and `Point2D`!
- **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:

```
1 void myMethod(Type1 arg1, Type2 result) {
2     result.setX(someValue);
3 }
```

    But the following does not:

```
1 void myMethod(Type1 arg1, Type2 result) {
2     result = new Type2(someValue);
3 }
```

E. Optional Bonus Problem

Part 1: Pairwise Uniform Hash Functions

Note: This exercise uses concepts of probability theory that you may not be used to.

In our discussion of hash functions, there is an elephant in the room that we never really addressed: we adopt the uniform hashing assumption, but none of the examples of hash functions we've seen satisfy it – they're all deterministic, after all, and no deterministic function can.

We'll consider one of the possible "fixes" to this problem (the other, widely used in practice, is to forgo the uniform hashing assumption entirely and make an assumption on the "unpredictability" of the output; see, e.g., the AES cipher).

As a warm-up, consider the hash function $h\colon \{0,1\} \to \{0,1\}$ generated as follows: sample two uniformly random and independent bits $a, b \sim \{0,1\}$ and set $h(x) = (a \cdot x) \oplus b$ (where $\oplus$ is the XOR operation, given by $0 \oplus x = x$ and $1 \oplus x = 1 - x$ for $x \in \{0,1\}$).

What is the probability that $(h(0), h(1)) = (x, y)$ for each $x, y \in \{0,1\}$?

Now let $m < n$ and consider the hash function $h\colon \{0,1\}^n \to \{0,1\}^m$, where for each i, the ith bit of $h(x)$ is generated by sampling $a_i, b_i \sim \{0,1\}$ and setting $h_i(x) = (a_i \cdot x_i) \oplus b_i$. (All a_i and b_i are uniform and independent.)

Show that it is **not** the case that for any four bit strings $x \neq x' \in \{0,1\}^n$ and $y, y' \in \{0,1\}^m$, the probability that $h(x) = y$ and $h(x') = y'$ is the same.

Consider the following alternative definition of $h\colon \{0,1\}^n \to \{0,1\}^m$: for each i, the ith bit of $h(x)$ is generated by sampling $a_{i1}, a_{i2}, \ldots, a_{in}, b_i \sim \{0,1\}$ and setting $h_i(x) = (a_{i1} \cdot x_1) \oplus \cdots \oplus a_{in} \cdot x_n \oplus b_i$. (All a_{ij} and b_i are uniform and independent).

Show that h satisfies the *pairwise* uniform hashing assumption: for any four bit strings $x \neq x' \in \{0,1\}^n$ and $y, y' \in \{0,1\}^m$, the probability that $h(x) = y$ and $h(x') = y'$ is the same.

Finally, show that h does not satisify the 4-wise uniform hashing assumption: find four strings $x, y, z, w \in \{0,1\}^n$ such that, *for every* h, the value of $h(w)$ is completely determined by $h(x)$, $h(y)$ and $h(z)$.