Precept Outline
- Review of Lectures 13 and 14:
  - Hash Tables

Relevant Book Sections
- Book chapters: 3.4

A. Review: Hash Tables

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: Hash table analysis

Suppose we have an initially *empty* hash table of size $2n$ with indices 0 to $2n-1$, where $n \geq 10$. The table stores integer keys from the set $\{0, 1, 2, \ldots, 4n-1\}$ and uses the hash function $h(x) = x \bmod 2n$.

What is the worst-case *total* running time, in Θ-notation as a function of n, of a sequence of n `insert` operations in each of the following two cases? Recall that a hash table implements a symbol table, so duplicate keys are not allowed (inserting an existing key updates the associated value).

1. The hash table is implemented using *separate chaining*.
2. The hash table is implemented using *linear probing*.

C. Algorithm Design

Part 1: Server Downtime

You are the system administrator for a university network. During the exam period, students submitted n reports of server downtime. Each report is recorded as a time interval $[s_i, e_i]$, representing the start and end times in milliseconds from the beginning of the month. Because multiple students reported the same outages, many of these intervals overlap. You need to calculate the *total number of milliseconds* that the network was actually down.

Example. Suppose $n = 4$ and the reported downtime intervals are $[1000, 5000]$, $[4000, 8000]$, $[10000, 12000]$, and $[7000, 9000]$. These merge into two actual, continuous downtime periods: $[1000, 9000]$ and $[10000, 12000]$. The total offline duration is $(9000 - 1000) + (12000 - 10000) = 8000 + 2000 =$ **10000** milliseconds.

Design an algorithm to compute the total offline duration in $O(n \log n)$ time in the worst case. *Hint: consider the sweep-line algorithm.*

D. Optional Bonus Problems

Part 1: Duplicate Network Packet

You are analyzing a chronological log of network traffic. You are given an array `packets` of length n, where each element is an integer representing the ID of a received packet. The array is ordered chronologically by the exact time the packets arrived.

Because of network unreliability, many packets are re-transmitted and appear in the log multiple times. We wish to design an algorithm that finds the *first* packet ID (chronologically, by its first arrival) that is a *duplicate* (*i.e.*, it is received more than once in the log). It should return `-1` if every packet in the log is strictly unique. Specifically, we wish to implement the method `int firstDup(int[] packets)`.

Example. If the log is `[4, 5, 2, 7, 5, 4, 2, 8]`, the algorithm should return `4`. (Packets 4, 5, and 2 are duplicates. While the second 5 arrives before the second 4, packet 4 was the chronologically *first* packet to arrive that eventually gets duplicated).

1. Design an algorithm implementing `firstDup` that runs in $O(n)$ time in expectation and uses $O(n)$ memory.
2. Identify the logical flaw in the following proposed in-place implementation of `firstDup`: *First, sort the array using an in-place sort. Then, scan the sorted array to find an entry that is equal to its immediate neighbor, and return it.*
3. Describe an $O(n \log n)$ worst-case time, $O(n)$ memory implementation of `firstDup`.

Part 2: Designing Hashes

Suppose that you are implementing a symbol table as a hash table, where the keys are the following integers:

`5, 20, 40, 55, 64`

You decide to use the hash function `h(i) = i % c`, where `c` is the capacity of the hash table. What is the smallest value of `c` you can pick such that there are no collisions?

Since you have mastered hashing integers, you decided to tackle a bigger challenge: strings. Consider the following two Java methods that hash a string `s`:

```
public int hashMethod1(String s) {
    s = s.toLowerCase();
  int hash = 0;
  for (int i = 0; i < s.length(); i++)
    hash = hash + (s.charAt(i) - 'a') % 13;
  return hash;
}
```

```
public int hashMethod2(String s) {
    s = s.toLowerCase();
  int hash = 0;
  for (int i = 0; i < s.length(); i++)
    hash = (hash * 13) + (s.charAt(i) - 'a');
  return hash;
}
```

For each of them, find a pair of distinct strings of length 5 that hash to the same value.

Part 3: 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):

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

Part 4: K-d tree tracing

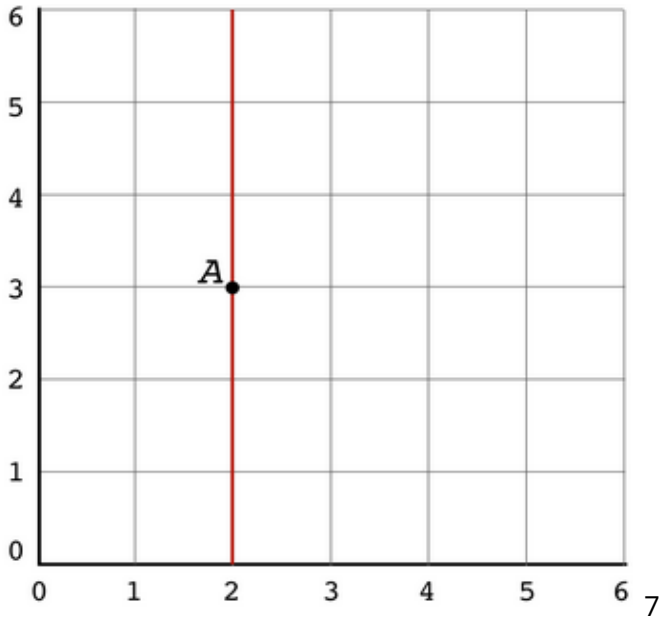
Draw the 2d-tree that results from inserting the following points:

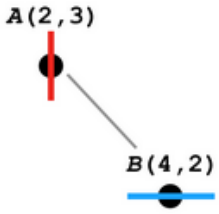
A	B	C	D	E	F	G
$(2, 3)$	$(4, 2)$	$(4, 5)$	$(3, 3)$	$(1, 5)$	$(4, 4)$	$(1, 1)$

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?

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.

