Final Review

CSE373 - Help Section
Preserving Abstraction

Aliasing and mutation

- Client was able to update something inside the abstraction because client had an alias to it!
class BankAccount {
    private Person owner;
    private float balance;
    public BankAccount(Person o, float b) {
        if(o == null || o.birthdate == null){
            throw new IllegalArgumentException();
        }
        owner = o; balance = b;
    }
    public long getOwnerAge() {
        Date now = new Date();
        long millisecondsPerYear = 365*24*60*60*1000;
        return (now.getTime() - owner.birthdate.getTime()) / millisecondsPerYear;
    }
}
class BankAccount {
    private Person owner;
    private float balance;
    public BankAccount(Person o, float b) {
        if(o == null || o.birthdate == null){
            throw new IllegalArgumentException();
        }
        owner = o; balance = b;
    }
    public long getOwnerAge() {
        Date now = new Date();
        long millisecondsPerYear = 365*24*60*60*1000;
        return (now.getTime() - owner.birthdate.getTime()) /
               millisecondsPerYear;
    }
}

Person p = new Person();
p.name = "Bob";
p.birthdate = new Date(1988, 10, 17);
BankAccount acct = new BankAccount(p, 10.0);
p.birthdate = null;
acct.getOwnerAge();

Fixation: The constructor of BankAccount should do a deep copy of the Person object passed in.
Multi-Threading

Parallelism idea

- Example: Sum elements of a large array
- Idea: Have 4 threads simultaneously sum 1/4 of the array
  - Warning: This is an inferior first approach, but it’s usually good to start with something naïve works

- Create 4 thread objects, each given a portion of the work
- Call start() on each thread object to actually run it in parallel
- Wait for threads to finish using join()
- Add together their 4 answers for the final result
Multi-Threading

1. Create Threads
2. Call `start()` to run them in parallel
3. Wait for threads to finish with `join()`
4. Add together their returns to get the final result

```java
class SumThread extends java.lang.Thread {
  int lo, int hi, int[] arr; // arguments
  int ans = 0; // result
  SumThread(int[] a, int l, int h) { ... }
  public void run(){ ... } // override
}

int sum(int[] arr){ // can be a static method
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for(int i=0; i < 4; i++){ // do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
    ts[i].start();
  }
  for(int i=0; i < 4; i++) { // combine results
    ts[i].join(); // wait for helper to finish!
    ans += ts[i].ans;
  }
  return ans;
}
```
Multi-Threading

Problem:
The current code is entirely sequential because a separate thread of execution is never created (i.e. `start()` is NEVER called).

Fixation:
```
left.start();
right.run();
left.join();
```
(a) While processing a list of objects, **check if you have processed a particular object before.**

**Hashtable**

(b) Store a **list of students and their grades**. You must also provide an efficient way for a client to see all students **sorted in alphabetical order by name**. Give the running time for this operation as well.

**AVL Tree**

(c) Process a digital image to divide the image up **into groups** of pixels of the same color.

**Union-Find**

<table>
<thead>
<tr>
<th>Functions</th>
<th>insert()</th>
<th>find()</th>
<th>remove()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>O(1) (push)</td>
<td>/</td>
<td>O(1) (pop)</td>
</tr>
<tr>
<td>Queue</td>
<td>O(1)</td>
<td>/</td>
<td>O(1)</td>
</tr>
<tr>
<td>Hashtable</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>AVL Tree</td>
<td>O(logN)</td>
<td>O(logN)</td>
<td>O(logN)</td>
</tr>
<tr>
<td>Priority Queue</td>
<td>O(logN)</td>
<td>/</td>
<td>O(logN)</td>
</tr>
<tr>
<td>Union-Find</td>
<td>Union: O(1), Find: O(logN)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(d) Compute a frequency analysis on a file. That is, **count the number** of times each character occurs in the file, and **store the results**.

**Hashtable**

(e) Store the activation records (i.e. objects containing the return address and local variable associated with a function call) for nested function calls.

**Stack**

<table>
<thead>
<tr>
<th>Functions</th>
<th>insert()</th>
<th>find()</th>
<th>remove()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>O(1) (push)</td>
<td>/</td>
<td>O(1) (pop)</td>
</tr>
<tr>
<td>Queue</td>
<td>O(1)</td>
<td>/</td>
<td>O(1)</td>
</tr>
<tr>
<td>Hashtable</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>AVL Tree</td>
<td>O(logN)</td>
<td>O(logN)</td>
<td>O(logN)</td>
</tr>
<tr>
<td>Priority Queue</td>
<td>O(logN)</td>
<td>/</td>
<td>O(logN)</td>
</tr>
<tr>
<td>Union-Find</td>
<td>Union: O(1), Find: O(logN)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Sorting

<table>
<thead>
<tr>
<th></th>
<th>Best Case</th>
<th>Worst Case</th>
<th>Average Case</th>
<th>Additional Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$\sim O(n*\log n)$</td>
<td>$\sim O(n*\log n)$</td>
<td>$O(n*\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$\sim O(n*\log n)$</td>
<td>$O(n*\log n)$</td>
<td>$O(n*\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Quick Sort (simple)</td>
<td>$O(n*\log n)$</td>
<td>$O(n^2)$</td>
<td>$O(n*\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Quick Sort (good pivot)</td>
<td>$O(n*\log n)$</td>
<td>$O(n*\log n)$</td>
<td>$O(n*\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Bucket Sort</td>
<td>$O(n+K)$</td>
<td>$O(n+K)$</td>
<td>$O(n+K)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>