2 December

Announcements

- Will publish two practice assignments, both based on trees. Totally optional and just for studying.
- Email me ASAP if you want individualized feedback on any coding assignments.
- Labs this week are graded for attendance, will be free time to work on project with TA feedback

Abstract Data Types

- algebraic structures: domain, collection of operations, set of constraints
- ex: Collection, Set, List, Stack, Queue, Map, Tree, Graph, Priority Queue
- Stack
  - domain: S (stack states), X (type of values in stack)
  - operations: push, pop
  - constraints: push(S,x); V <- pop(S); equivalent to V <- x
    - can also be implemented with pre- and post-conditions
    - Ex: precondition of pop is `the stack is not empty` and the postcondition is `top element returned, stack size decreases by one`
  - Can implement with array, linked list, etc
- Queue
  - similar but different constraints:
    - if n = Q.size() > 0; Q.push(t); pop n times; then `t = Q.front()`
- Map
  - domain: D (map states), X (type of values in stack)
  - synonyms: associative array, symbol table, dictionary, hashmap
  - operations: put, get, remove
  - constraints:
    - `get(k, put(j, v, D)) = if k == j then v else get (k,D)`
    - `get(k, new()) = fail` where `fail` is an exception or default value
    - `remove(k, put(j, v, D)) = if k == j then remove(k,D) else put(j, v, remove(k, D))`
    - `remove(k, new()) = new()`
  - Can implement with hash tables, self-balancing binary search trees
- Self-Balancing Binary Trees
  - pre-condition and post-condition of put and delete are that balance factors of all nodes are +1 / 0 / 1

Why?

- software engineering: users can interact with defined top-level behaviors while engineers are free to optimize "under the hood"
  - if you build it, they will use it (esp if your customers are developers)
- correctness of code
  - type checking, algebraic verification, 'design by contract'
  - for larger codebases or more complex projects, debugging at compile-time is easier
  - bugs can be dependent on program state ("heap pollution")

Type Checking and Generics

This will compile, but is not type-safe:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Type-safe version:

```
List<String> list = new ArrayList<>();
list.add("hello");
String s = list.get(0);
```

Recall that type information is not available at runtime in Java, so we CANNOT do the following:

```
if (someList instanceof List<String>) {
 // ...
}
```

We can check for instances of a *class* but not a *type* (ADT).

Bounded Type Parameters

Bad:

```
public <T> void sort(List<T> list) {
    // ...
}
```

Good:

```
public <T extends Comparable<T>> void sort(List<T> list) {
    // ...
}
```