Precept Outline
- Amortized Runtime
- Accounting Method: Resizing Arrays

Relevant Book Sections
- Accounting Method: Union-Find
- Persistent Data Structures

A. Advanced Precept Problems

Part 1: Amortized Analysis

Let $c > 0$ be a constant such that

- removing an element from a non-empty array takes $\leq c$ units of time;
- adding an element to a non-full array requires $\leq c$ units of time;
- allocating an array of size n and copying $\leq n$ elements from one array to another takes $\leq cn$ units of time;

Prove that the amortized runtime of a resizable array with minimum capacity 2 and the double-when-full, halve-when-one-quarter-full resizing policy is $\leq 5c$.

Adapt your proof above to show a $\leq 4c$ amortized runtime under the resizing policy that triples when full and multiplies by $1/3$ when $1/9$ full.

Part 2: Persistent Data Structures

Construct two variants of partially persistent stacks that support the usual stack API (`push()` and `pop()` operate on the current state of the stack) as well as an `Iterator<Item> iterator(int i)` method, which returns an iterator for the i-th state of the stack.

They should satisfy optimal performance with respect to space ($\Theta(m)$ after m operations) and the following runtime guarantees:

- `push()`/`pop()` take $\Theta(1)$ and `iterator()` takes $O(m)$ time in the worst case;
- `push()`/`pop()` take $\Theta(1)$ amortized (but $\Theta(m)$ worst-case) time and `iterator()` takes $\Theta(1)$ worst-case time.