Data Structure in Java - Midterm Review

Daniel Bauer

October 21, 2015

General Concepts

• Abstract Data Types vs. Data Structures.
• Recursion.
• Basic proofs by induction. Proofs by counter-example.

Java Concepts

You will not be required to write Java code during the midterm or final, but we expect you to be able to read short programs.

• Basic Java OOP: Classes / Methods / Fields. Visibility modifiers.
• Generics.
• Inner classes (static vs. non-static).
• Interfaces.
• Iterator/Iterable.
• Comparable.
• Mutable/Immutable objects.

Analysis of Algorithms

• Time vs. Space analysis.
• Big-O notation for asymptotic running time: \( O(f(n)) \), \( \Theta(f(n)) \), \( \Omega(f(n)) \).
• Typical growth functions for algorithms.
• Worst case, best case, average case.
• **Skills:** Compare growth of functions using big-O notation. Given an algorithm (written in Java), estimate the asymptotic run time (including nested loops and simple recursive calls).

### Lists

• The List ADT, including typical List operations.

• ArrayList:
  – running time for insert, remove, find at different positions in the list.
  – what to do when we run out of space.

• LinkedList:
  – single vs. double linked list.
  – running time for insert, remove, find at different positions in the list.
  – sentinel (head/tail) nodes.

• Lists in the Java Collections API.

• **Skills:** Develop simple list algorithms for additional operations (removing duplicates, etc.). Implement iterators.

### Stacks and Queues

• Stack ADT and operations (push, pop, peek). LIFO.

• Queue ADT and operations (enqueue, dequeue). FIFO.

• All operations run in $O(1)$.

• Stack implementation using LinkedList, ArrayList, plain arrays.

• Stack applications:
  – check if symbols are balanced.
  – reordering sequences (in-order to post-order, train cars, ...).
  – storing intermediate computations on a stack (evaluating post-order expressions).

• Method Call Stack, Stacks and recursion. Tail recursion.

• Queue implementation using LinkedList.

• Queue implementation using a Circular Array.

• Stacks and Queues in the Java Collections API.
• **Skills:** Use stacks and queues in applications. Implement multiple stacks in an array. Implement a queue using two stacks (or one stack + recursion).

**Trees**

• Binary trees and M-ary trees. Tree terminology (parent, children, root, leaves, path, depth, height)

• Different tree implementations (one field per child, array of children, list of children, siblings as linked list).

• Binary trees:
  – Full binary trees, complete binary trees.
  – Tree traversals: in-order, pre-order, post-order.
  – Implementing tree traversals using recursion or stacks.
  – Constructing an expression tree using a stack.
  – Relation between number of nodes and height of a binary tree. Maximum number of nodes in a binary tree depth $h$ is $2^h - 1$.

• **Skills:** Implement different tree traversals using recursion (different versions). Use these traversals to implement operations on trees.

• Examples for trees: expression trees, tries, search trees (to implement tree maps).

**Binary Search Trees**

• Map ADT.

• BST property.

• BST operations: contains, findMin, findMax, insert, remove

• Best case, worst case, average case runtime for operations (depends on height of tree)

• Implementing a Map using BSTs (tree map). Using a `Pair` class.

• **Skills:** Perform BST operations on paper. Different algorithms on BSTs. Returning all nodes in an interval. Checking that BST property is satisfied.

**AVL Trees**

• Balanced BSTs. AVL Balancing property.

• Maintaining AVL balance property on insert:
  – Outside imbalance, single rotation.
– Inside imbalance, double rotation.
– Verifying that a tree is balanced. Finding the location of an imbalance (bottom-up).

• Skills: Perform AVL rotations on paper, detect imbalances.

**B-Trees**

• Motivation for B-Trees: Store large search trees on disk. Replace expensive disk accesses with cheap linear search in memory.

• B-Tree balance property. B-Tree definition.

• 2-3-4 trees.

• Basic operations on B-trees: contains, insert, remove.

• Compute the ideal size of a B-Tree node (not needed for midterm).

• B+ trees.

• Skills: Perform easy B-Tree operations on paper.

**Hashing**

• Arrays as Maps.

• Basic concept of a hash map, hash functions, collisions.

• Good hash functions for Integers, Strings.

• Keys need to be immutable.

• Collision resolution: Separate chaining.

• Load factor of a hash map.

• Skills: Given a hash function, perform insertions into a Separate Chaining hashmap. Compute load factor. Design a new hash function for your own classes. (Universal hashing).