REVIEW TOPICS FOR MIDTERM AND FINAL

Midterm contains all the topics up to sorting.

Asymptotic Growth Rate: Compare functions defined analytically. Compare recurrent sequences. Analyze the time complexity of algorithms from their pseudocode.

1. Analyze functions and sequences. *
   A. Given an closed expression of a function, find its $\Theta(g(n))$ growth rate in its simplest form.
   B. Compare two functions in terms of their asymptotic growth rate.
   C. Given a recurrent sequence, define its asymptotic growth rate.
2. Analyze code. *Find time complexity from a given Python, C++ or pseudocode fragment.*
   A. Apply assumptions about common data structures (lists, sets, dictionaries) in Python.
   B. Express time complexity for a code snippet from the inside out.
   C. Write a recurrence to express time complexity of a recursive algorithm.
   D. Solve a recurrence using the Master's theorem.
3. Analyze Other Complexity Measures. *Estimate other criteria besides worst-case time complexity.*
   A. Evaluate the space complexity for an algorithm, its asymptotic growth rate.
   B. Evaluate the amortized time complexity, if some operation is applied many times.
   C. Evaluate the number of comparisons needed for sorting, searching or ranking algorithms.

Lists, Stacks, Queues:

1. Typical implementations for Lists, Stacks, Queues.
   A. Given an implementation, draw the memory state at a certain moment, e.g. an array or a linked list.
   B. Create a singly linked list implementation of some ADT method.
   C. Create a doubly linked list implementation of some ADT method.
2. Implement a data structure in pseudocode, compare the implementation alternatives.
   A. Express dependent ADT operations in terms of simpler ADT operations.
   B. Given a list/stack/queue algorithm pseudocode, find its time complexity.
   C. Given a problem description, implement the algorithm at ADT Level.
3. Write algorithms using Lists, Stacks or Queues. *Algorithms can call list-like data structures using their ADT functions.*
   A. Write algorithms and estimate the time complexity of algorithms processing expressions.

B. Write algorithms using stack to navigate a tree-like structure.

Tree-like Structrues:

1. Tree concepts.

   A. **Use the concepts of non-rooted trees (plain graph level), rooted trees, ordered trees.**

   B. Use the concepts of binary and n-ary trees. For binary trees distinguish full, complete and perfect trees.

   C. Use the concept of binary search tree (labels/keys compare according to the in-order traversal order).

   D. Encode multiway trees with binary trees (and binary trees into multiway trees).

2. Priority Queues and Heaps.

   A. **Define priority queue ADT, analyze various non-heap ways to implement it.**

   B. Define a heap data structure, compute parents and children, peform insert and delete-min (or delete-max).

   C. Use priority queues to build Huffman prefix code given the alphabet of messages and their probabilities.

3. Tree traversals and Backtracking.

   A. **Use BFS traversal order.**

   B. Use DFS traversal (for pre-order, in-order, post-order visiting of the nodes).

   C. Solve algorithmic tasks using backtracking.

N-ary Search Trees:

1. Regular BSTs

   A. Insert, delete and find keys in a binary search tree.

   B. Answer the questions about their properties.

   C. Perform various flavors of DFS traversals (in-order, pre-order, post-order), find in-order predecessors and successors.

   D. Reason about the expected height of a BST, if you insert keys in certain order.

2. Self-balancing Search Trees.

   A. Draw AVL Trees, answer questions about their properties (worst-case depth etc.), insert and delete keys.

   B. Insert, delete and find keys in multiway search trees.

   C. Draw 2-4 Trees, answer questions about their properties, insert and delete keys.

3. Create and Use Augmented Trees. *Extra information for any node can be computed from other attributes of the node and its children.*

   A. Consider different ways to augment trees ()

   B. Computing $\textsc{Rank}(v)$ – how many nodes w in the given tree satisfy the inequality $w.key \leq v.key$.

   C. Computing $\textsc{Count}(a, b)$ – how many keys are between a and b.

Sorting:

1. Time-complexity for sorting algorithms.

   A. Use Stirling's formula to evaluate factorials and binomial coefficients.

B. **Count comparisons in a decision tree to find the lower bound of comparisons needed.**

C. Analyze some inefficient algorithms such as Bubblesort.

2. Various sorting algorithms:

   A. Criteria how to compare sorting algorithms (efficient/inefficient, stable/unstable, online/offline, in-place/outplace, behavior for random or specific inputs).

   B. Use Mergesort, draw memory states, analyze complexity, count comparisons.

   C. Use Heapsort, draw memory states, analyze complexity, count comparisons.

   D. Use Quicksort, draw memory states, analyze complexity, count comparisons.

3. Linear-time sorting in special cases:

   A. Use Radix sort, draw memory states, analyze time.

   B. Use Counting sort, draw memory states, analyze time.