CMSC 420: Lecture EX1
Review for Midterm 1

The exam will be in class, this Thursday, Mar 9. It will be closed-book, closed-notes, but you will be allowed one “cheat sheet” of notes, front and back.

So far, we have studied a wide variety of data structures for a diverse set of applications. We have considered the material from both a theoretical and practical perspectives. We have illustrated various aspects of data-structure design and analysis, including worst-case and asymptotic analyses, randomized data structures.

**Basic Data Structures:** Sequential and linked allocation, amortized analysis, multilists and sparse matrices.

**Amortized Analysis:** We studied the concept of amortized analysis, where rather than analyzing the running time of each individual operation, we instead consider the total running time for a sequence of operations, and then take the average. This is useful for data structures that are fast most of the time, but occasionally take a long time (e.g., due to expanding or reorganizing themselves).

**Trees:** Representations of rooted trees, binary trees and traversals, full and extended binary trees, threaded binary trees, complete binary trees (and array allocation).

**Disjoint Set Union/Find:** This is a tree-based data structure for maintaining a collection of disjoint sets, where the two operations are those of merging two sets (union) and finding the set that contains a given element (find). We showed that a simple version of the data structure supports operations in \(O(\log n)\) time. If path compression is used, the total time to perform a sequence of \(m\) operations is \(O(m \cdot \alpha(m, n))\), thus the amortized time per operation is \(O(\alpha(m, n))\). Recall that this is the extremely slow-growing inverse Ackerman’s function, which theoretically grows to infinity, but it does so so slowly that its value is essentially a constant for all practical purposes.

**Priority Queues:** A priority queue is a data structure that stores elements, each associated with a key, called its *priority*. At a minimum it supports the operations of *insert* and *extract-min*, ideally each in \(O(\log n)\) time.

- **Binary Heap:** This is the most basic priority queue data structure. Since it is based on a left-complete tree, it can be stored inside an array. Operations *insert* and *extract-min* can be performed in \(O(\log n)\) time.

- **Leftist Heap:** Additionally supports the operation of merging two heaps together in \(O(\log n)\) time. This was done by organizing the tree so that its rightmost path is short, never longer than \(O(\log n)\). Introduces the notion of the npl or null path length to organize the tree structure.

**Ordered Dictionaries:** We studied a number of tree-based data structures for ordered dictionaries. These support the operations of insert, delete, and find.

- **Binary Search Trees:** Standard (unbalanced) binary search trees. Good expected-case performance (\(O(\log n)\)) for random insertions.
**AVL Trees:** Height-balanced trees. Use of single- and double-rotations to balance the tree. Worst-case time for all dictionary operations is $O(\log n)$.

**2-3 Trees:** Variable-width nodes with either 2 or 3 children per node. Operations run in $O(\log n)$ worst-case time. This data structure was maintained through three operations, split, merge, and adoption (or key rotation).

**Red-Black Trees:** Binary encodings of 2-3 and 2-3-4 trees. Operations run in $O(\log n)$ worst-case time. We presented a variant called the **AA tree**, in which balance is maintained by two operations, split and skew.