CMSC 420: Lecture EX1
Review for the Midterm Exam

- The Midterm Exam will be asynchronous and online. The exam will be made available through Gradescope for a 48-hour period starting at **12:00am the morning of Thu, Oct 29** and running through **11:59pm the evening of Fri, Oct 30**. The exam is designed to be taken over a 90-minute time period, but to allow time for scanning and uploading, you will have **2 hours** to submit the exam through Gradescope once you start it.

- The exam will be open-book, open-notes, open-Internet, but it must be done on your own without the aid of other people or software. (You may use a simple arithmetic calculator, but I don't expect that you will need one.)

- Do not discuss any aspects of the exam with classmates during the exam's 48-hour time window, even if you have both submitted. This includes its content, its difficulty, and its length.

- If any questions arise while you are taking the exam, please either email me (`mount@umd.edu`) or make a *private* Piazza post. (Do *not* ask your classmates.) If you are unsure about how to interpret a problem and I do not respond in a timely manner, please do your best and write down any assumptions you are making. There will be no "trick" questions on the exam. Thus, if a question doesn't make sense or seems too easy or too hard, please check with me.

- If you experience any technical issues while taking the exam, **don't panic**. Save you work (ideally in a manner that attaches a time stamp), and contact me by email (`mount@umd.edu`) as soon as possible. I understand that unforeseen events can occur, and I will attempt make reasonable accommodations.

So far, we have studied a wide variety of data structures for ordered dictionaries. While they are all aimed at solving the same basic operations (insert, delete, find) they illustrate various aspects of data-structure design: worst-case and asymptotic analyses, randomized data structures, and external-memory data structures.

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

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

Binary Search Trees: Standard (unbalanced) binary search trees. Good expected-case performance ($O(\log n)$) for random insertions. If biased replacement is used during deletion (selecting the replacement node exclusively from one subtree, left or right), long sequences of random deletion can lead to $O(\sqrt{n})$ tree height.

AVL Trees: Height-balanced trees. Use of single- and double-rotations to balance the tree. Worst-case time for all dictionary operations is $O(\log n)$. At most one rotation is performed after each insertion, but deletion may result in rotations going all the way up to the root.

2-3 Trees: (These are equivalently B-trees of order 3). They have variable-width nodes with either 2 or 3 children per node. 2-3 trees are rebalanced using the operations split (convertine one 4-node into two 2-nodes) and merge (combining a 1- and 2-node into a 3-node) and adoption or key-rotation (moves a key and subtree between two adjacent siblings). Operations run in $O(\log n)$ worst-case time.

Red-Black Trees: These are a binary encodings of 2-3 and 2-3-4 trees. We presented AA-trees, a simplified variant of red-black trees. Rather than using colors, AA-trees use level numbers to encode colors. It employs three restructuring operations (update-level, skew, and split) to maintain balance. Operations run in $O(\log n)$ worst-case time.

Treaps: A randomized binary search tree, which uses random priorities assigned to each node so that the tree structure is equivalent to a binary search tree under random insertions. The expected running time of dictionary operations is $O(\log n)$, where the expectation is over the random choices. Treaps can be interpreted as a geometric data structure called a priority search tree.

Skip lists: Another randomized search structure, which is based on linked lists with variable height nodes. Dictionary operations can be performed in $O(\log n)$ expected-case time, where the expectation is over the random choices. The analysis of the search process is based on an interesting technique, called *backwards analysis*, which involves analyzing the algorithm's operation in reverse.

Splay Trees: A self-adjusting data structure, which uses no balance information. It is based on an operation, called *splay*, which brings a node to the root while reorganizing the trees structure. Through a complicated potential argument (which we did not present), it can be shown that the amortized running time of dictionary operations is $O(\log n)$. The data structure also has a number of other interesting operations, including static optimality, efficient finger-search, and the working-set properties.

B-Trees: A variable-width tree, where (typical nodes) have between $\lceil m/2 \rceil$ to m children, for a B-tree of order m (where m is a constant). These are widely used for external-memory (disk storage), by setting the node size to match the size of a disk page. As with 2-3 trees, nodes are balanced through the operations of split, merge, and adoption (key-rotation). The worst-case tree height is roughly $O(\log_{m/2} n)$, which is extremely small when m is large. A more practical variant is the B^+-tree, which is an extended version of the B-tree.

Scapegoat Trees: Another data structure with amortized efficiency of $O(\log n)$. Unlike earlier data structures, which use incremental rebalancing operations, this one rebuilds subtrees whenever they become unbalanced. The amortized analysis relies on the fact that large subtrees are rarely rebuilt, because, once balanced, it takes a large number of operations to render the subtree unbalanced again.