CMSC 420: Lecture Final
Final Review

- Like the Midterm Exam, the Final 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 Wed, Dec 16** and running through **11:59pm the evening of Thu, Dec 17** (Eastern Time). The exam is designed to be taken over a 2-hour time period, but to allow time for scanning and uploading, you will have **2.5 hours** to submit the exam through Gradescope once you start it.
- I will release practice problems and solutions. (There will be a posting on Piazza.)
- There will be a *review session* at 2-3pm on Tue, Dec, 15. (There will be a Piazza posting with the link.)
- 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 during the exam period (Dec 16–17), 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 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, and external-memory data structures.

Up to the Midterm Exam: The final exam will be comprehensive, but the focus will be on material since the midterm (in a ratio of roughly 2/3 to 1/3). Here is a summary of the coverage prior to the midterm:

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).

Ordered Dictionaries: We studied a wide variety of tree-based data structures for ordered dictionaries. These support the operations of insert, delete, and find, and various ordered extensions of these operations (e.g., find-up, get-min, range queries).

- **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:** (These are equivalently B-trees of order 3). Variable-width nodes with either 2 or 3 children per node. Operations run in $O(\log n)$ worst-case time.
- **Red-Black Trees:** Binary encodings of 2-3 and 2-3-4 trees. 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.
- **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.
- **Splay Trees:** A self-adjusting data structure, which uses no balance information. 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). These are widely used for external-memory (disk storage), by setting the node size to match the size of a disk page. The worst-case tree height is roughly $O(\log_{m/2} n)$, which is extremely small when m is large.
- **Scapegoat Trees:** Another data structure with amortized efficiency of $O(\log n)$. Achieves balance by rebuilding subtrees whenever they become unbalanced.

Hashing: A very fast data structure for *unordered dictionaries*. Keys are scattered through the use of a *hash function* and various *collision resolution* strategies are applied to handle collisions. We studied separate chaining, linear probing, quadratic probing, and double hashing.

Quadtrees and kd-trees: Data structures for storing multi-dimensional data. We explained how to perform insertion and answer queries for point kd-trees. We showed that orthogonal range searching queries can be answered in $O(\sqrt{n})$ time, assuming that the tree is balanced.

Persistence: In the programming assignment, we studied how to make search trees persistent (that is, supporting queries in the past) through the use of subtree copying and temporal nodes.

Tries and Digital Search Trees: We studied various data structures related to digital search trees, including tries, Patricia tries, and suffix trees. These are used for storing digital, that is, string-like, data and are useful for answering substring queries efficiently.

Memory Management: Data structures for allocating and deallocating variable sized blocks of memory. We discussed the standard (unstructured) approach based on storing variable sized nodes and a more structured approach called the *buddy system*.

The world of data structures is vast, and there are many more topics that we could have delved into, including many more ways to store geometric data, storing high-dimensional data for applications in machine learning, storing temporal and time-series data sets, and how to compress data sets and retrieve information from these compressed forms.