The final exam will be **Mon, May 15, 4:00-6:00pm in IRB 0324** (the Antonov Auditorium). The exam will be closed-book, closed-notes, but you are allowed 3 “cheat-sheets” (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, and external-memory data structures.

**Up to the Midterm Exam:** The final exam will be comprehensive, but with a focus on the latter part of the semester. Rough estimates (but don’t rely on these): Before Midterm 1 (30%), between midterms (30%), after Midterm 2 (40%). Here is a summary of topics covered:

**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).

**Disjoint Set Union/Find:** Supports \( m \) Union-Find operations on a set of size \( n \) in amortized time \( O(\alpha(m, n)) \). (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 **extractMin**, ideally each in \( O(\log n) \) time. We discussed two variants:

- **Binary Heap:** The most well-known priority queue data structure, used in HeapSort. Since it is based on a left-complete tree, it can be stored inside an array, and uses the operations sift-up and sift-down.

- **Leftist Heap:** Supports the operation of merging two heaps together. This was done by organizing the tree so that its rightmost path is short, never longer than \( O(\log n) \).

**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.

**AA- and 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.

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.

Tries and Suffix 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.

Bloom Filters: This data structure is used for answering membership queries in very large sets. It does not explicitly store the keys of the set, but rather bit patterns to determine which elements are in the set. With low probability, it may commit false positive errors (claiming that an element is in the set when it is not). It never commits false negative errors. The algorithm is space efficient (using roughly as many bits as there are elements in the set) and time efficient (taking essentially constant time per query).

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.