# Lecture 27: Final Review

**Final Overview:** This semester we have covered a number of different data structures. The fundamental concept behind all data structures is that of arranging data in a way that permits a given set of queries or accesses to be performed efficiently. The aim has been at describing the general principles that lead to good data structures as opposed to giving a "cookbook" of standard techniques. In addition we have discussed the mathematics needed to prove properties and analyze the performance of these data structures (both theoretically and empirically).

Some of the important concepts that I see that you should take away from this course are the following:

**Mathematical Models/Objects:** Before you design any data structure or algorithm, first isolate the key mathematical elements of the task at hand. Identify what the mathematical objects are, and what operations you are performing on them. (E.g. Dictionary: insertion, deletion, and finding of numeric keys. Ray Tracing: insertion, and ray tracing queries for a set of spheres in 3-space.)

**Recursive Subdivision:** A large number of data structures (particularly tree based structures) have been defined recursively, by splitting the underlying domain using a key value. The algorithms that deal with them are often most cleanly conceptualized in recursive form.

**(Strong) Induction:** In order to prove properties of recursively defined objects, like trees, the most natural mathematical technique is induction. In particular, strong induction is important when dealing with trees.

**Balance:** The fundamental property on which virtually all good data structures rely is a notion of information balancing. Balance is achieved in many ways (rebalancing by rotations, using hash functions to distribute information randomly, balanced merging in Union-Find trees).

**Amortization:** When the running time of each operation is less important than the running time of a string of operations, you can often arrive at simpler data structures and algorithms. Amortization is a technique to justify the efficiency of these algorithms, e.g. through the use of potential function which measures the imbalance present in a structure and how your access functions affect this balance.

**Randomization:** Another technique for improving the performance of data structures is: if you can't randomize the data, randomize the data structure. Randomized data structures are simpler and more efficient than deterministic data structures which have to worry about worst case scenarios.

**Asymptotic analysis:** Before fine-tuning your algorithm or data structure's performance, first be sure that your basic design is a good one. A half-hour spent solving a recurrence to analyze a data structure may tell you more than 2 days spent prototyping and profiling.

**Empirical analysis:** Asymptotics are not the only answer. The bottom line: the algorithms have to perform fast on my data for my application. Prototyping and measuring performance parameters can help uncover hidden inefficiencies in your data structure.

**Topics:** Since the midterm, these are the main topics that have been covered.

**Priority Queues:** We discussed leftist heaps and skew-heaps. Priority queues supported the operations of insert and deleteMin in $O(\log n)$ time. Leftist and skew heaps also supported the operation merge. An interesting feature of leftist heaps is that the entire tree does

not need to balanced. Only the right paths need to be short, in particular the rightmost path of the tree is of length $O(\log n)$. Skew heaps are a self-adjusting version of leftist heaps.

**Disjoint Set Union/Find:** We discussed a data structure for maintaining partitions of sets. This data structure can support the operations of merging to sets together (Union) and determining which set contains a specific element (Find).

**Geometric data structures:** We discussed kd-trees and range trees for storing points. We also discussed interval trees for storing intervals (and a variant for storing horizontal line segments). In all cases the key concepts are those of (1) determining a way of subdividing space, and (2) in the case of query processing, determining which subtrees are relevant to the search and which are not. Although kd-trees are very general, they are not always the most efficient data structure for geometric queries. Range trees and interval trees are much more efficient for their specific tasks (range queries and stabbing queries, respectively) but they are of much more limited applicability.

**Tries:** We discussed a number of data structures for storing strings, including tries, de la Brandais tries, patricia tries, and suffix trees. The idea behind a trie is that of traversing the search structure as you traverse the string. Standard tries involved a lot of space. The de la Brandais trie and patricia trie were more space-efficient variants of this idea. The suffix trees is a data structure for storing all the identifying substrings of a single string, which is used for answering queries about this string.

**Memory management and garbage collection:** We discussed methods for allocating and deallocating blocks of storage from memory. We discussed the standard method for memory allocation and the buddy system. We also discussed two methods for garbage collection, mark-and-sweep and stop-and-copy.

**Hashing:** Hashing is considered the simplest and most efficient technique (from a practical perspective) for performing dictionary operations. The operations insert, delete, and find can be performed in $O(1)$ expected time. The main issues here are designing a good hash function which distributes the keys in a nearly random manner, and a good collision resolution method to handle the situation when keys hash to the same location. In open addressing hashing, clustering (primary and secondary) are significant issues. Hashing provides the fastest access for exact searches. For continuous types of queries (range searching and nearest neighbor queries), trees are preferable because they provide ordering information.