Suggested Final Project Topics

Here are a list of data structure and families of data structures we think you might find interesting topics for a final project. You're by no means limited to what's contained here; if you have another data structure you'd like to explore, feel free to do so!

Tango Trees

Is there a single best binary search tree for a set of data given a particular access pattern? We asked this question when exploring splay trees. Tango trees are a data structure that provably are at most O(log log n) times slower than the optimal BST for a set of data, even if that optimal BST is allowed to reshape itself between operations. The original paper on Tango trees ("Dynamic Optimality – Almost") is quite accessible.

Why they're worth studying: There's been a flurry of research on dynamic optimality recently and there's a good chance that there will be a major breakthrough sometime soon. By exploring Tango trees, you'll get a much better feel for an active area of CS research and will learn a totally novel way of analyzing data structure efficiency.

Approximate Distance Oracles

Computing the shortest paths between all pairs of nodes in a graph can be done in time O(n^3) using the Floyd-Warshall algorithm. What if you want to get the distances between *many* pairs of nodes, but not all of them? If you're willing to settle for an approximate answer, you can use subcubic preprocessing time to estimate distances in time O(1).

Why they're worth studying: Pathfinding is as important as ever, and the sizes of the data sets keeps increasing. Approximate distance oracles are one possible approach to try to build scalable pathfinding algorithms, though others exist as well. By exploring approximate distance oracles, you'll get a better feel for what the state of the art looks like.

Robin Hood Hashing

Robin Hood hashing is a hashing scheme that combines elements of uniform hashing (excellent in theory, hard to achieve in practice) with linear problem (poor theoretically, but excellent in practice). It's simple to implement and has an interesting property – the expected *worst-case* cost of a lookup is O(log log n), meaning that even slow lookups will not take too long.

Why they're worth studying: Robin Hood hashing is closely related to a family of theoretical results bounding the expected worst-case number of collisions in certain types of hash tables. Cuckoo hashing is motivated by results like these, and if you explore Robin Hood hashing, you'd likely build up a much deeper intuition for why these hashing approaches are so efficient.

Link/Cut Trees

Euler tour trees are one type of dynamic tree data structure. However, aside from the applications we saw in CS166, they're not widely used. One reason for this is that Euler tour trees are useful for aggregating information over trees and subtrees rather than information over paths. For many algorithms, especially maximum flow problems, other types of trees, such as Sleator and Tarjan's link/cut trees, are easier to use. They're also surprisingly fast in practice and are a natural generalization of splay trees.

Why they're worth studying: Link/cut trees show up as subroutines in a huge number of graph algorithms, are highly versatile, and have an analysis that generalizes nicely from splay trees. Exploring this data structure would be particularly interesting if you've taken CS261 or have a background in maximum flow algorithms.

Scapegoat Trees

Scapegoat trees are an amortized efficient binary search tree. They have a unique rebalancing scheme – rather than rebalancing on each operation, they wait until an insertion happens that makes the tree too large, then aggressively rebuild parts of the tree to correct for this. As a result, most insertions and deletions are extremely fast, and the implementation is amazingly simple.

Why they're worth studying: Scapegoat trees use a weight-balancing scheme commonly used in other balanced trees, but which we didn't explore in this course. They're also amazingly easy to implement, and you should probably be able to easily get performance numbers comparing them against other types of trees (say, AVL trees or red/black trees.)

Factor Oracles

Suffix trees are theoretically elegant, but in practice can be too large and too slow to be practical. Various other data structures have been proposed as alternatives to suffix trees, such as suffix arrays. The factor oracle is a recent, space-efficient data structure that can be used to solve several interesting problems on strings. Understanding where they come from and why they're useful can help give you a better appreciation for how string processing is done in practice.

Why they're worth studying: String algorithms in theory and in practice are often somewhat divorced. If you're interested in seeing the interplay between theory and practice, this would be an excellent starting point.

Soft Heaps

The soft heap data structure is an approximate priority queue – it mostly works like a priority queue, but sometimes corrupts the keys it stores and returns answers out of order. Because of this, it can support insertions and deletions in time O(1). Despite this weakness, soft heaps are an essential building block of a very fast algorithm for computing MSTs called Chazelle's algorithm. They're somewhat tricky to analyze, but the implementation is short and simple.

Why they're worth studying: Soft heaps completely changed the landscape of MST algorithms when they were introduced and have paved the way toward provably optimal MST algorithms. They also gave the first deterministic, linear-time selection algorithm since the median-of-medians approach developed in the 1970's.

Cardinality Estimators

A cardinality estimator is a data structure for solving the following problem: given a stream of elements, how many distinct elements were in that stream? Using some clever techniques involving hashing, it's possible to approximate the answer to this question to reasonable degrees. These techniques are frequently used in databases to estimate how much work will be done during a complex query.

Why they're worth studying: The actual algorithms that drive most cardinality estimators are surprisingly simple, but the analyses require clever and nuanced arguments about random numbers. If you're interested in seeing some cute math tricks applied to interesting algorithmic problems, this might be the data structure for you.

Pairing Heaps

Fibonacci heaps have excellent amortized runtimes for their operations – in theory. In practice, the overhead for all the pointer gymnastics renders them slower than standard binary heaps. An alternative structure called the pairing heap has worse theoretical guarantees than the Fibonacci heap, yet is significantly simpler and faster in practice.

Why they're worth studying: Pairing heaps have an unusual property – no one actually knows how fast they are! We've got both lower and upper bounds on their runtimes, yet it's still unknown where the actual upper and lower bounds on the data structure lie.

Finger Trees

A finger tree is a B-tree augmented with a "finger" that points to some element. The tree is then reshaped by pulling the finger up to the root and letting the rest of the tree hang down from the finger. These trees have some remarkable properties. For example, when used in a purely functional setting, they give an excellent implementation of a double-ended queue with amortized efficient insertion and deletion.

Why they're worth studying: Finger trees build off of our discussion of B-trees and 2-3-4 trees from earlier this quarter, yet the presentation is entirely different. They also are immensely practical and can be viewed from several different perspectives; the original paper is based on imperative programming, while a more recent paper on their applications to functional programming focuses instead on an entirely different mathematical framework.

Fusion Trees

The fusion tree is a data structure that supports the standard binary search tree operations (search, successor, predecessor) in time better than $O(\log n)$ in the case where the keys are integers. Interestingly, the runtime of fusion tree operations depends only on the number of entries in the tree, not the size of the universe (as is the case for van Emde Boas trees).

Why they're worth studying: Fusion trees are a clever combination of B-trees and word-level parallelism. If you study fusion trees in depth, you'll learn quite a lot about different models of computation (for example, the AC^0 model) and how models of computation influence data structure design.

Farach's Suffix Tree Algorithm

Many linear-time algorithms exist for directly constructing suffix trees – McCreight's algorithm, Weiner's algorithm, and Ukkonen's algorithm for name a few. However, these algorithms do not scale well when working with alphabets consisting of arbitrarily many integers. In 1997, Farach introduced an essentially optimal algorithm for constructing suffix trees in this case.

Why they're worth studying: Our approach to building suffix trees was to first construct a suffix array, then build the LCP array for it, and then combine the two together to build a suffix tree. Farach's algorithm for suffix trees is interesting in that it contains elements present from the DC3 algorithm and exploits many interesting structural properties of suffix trees.

Strict Fibonacci Heaps

Almost 30 years after the invention of Fibonacci heaps, a new type of heap called a strict Fibonacci heap was developed that achieves the same time bounds as the Fibonacci heap in the worst-case, not the amortized case.

Why they're worth studying: Strict Fibonacci heaps are the culmination of a huge amount of research over the years into new approaches to simplifying Fibonacci heaps. If you're interested in tracing the evolution of an idea through the years, you may find strict Fibonacci heaps and their predecessors a fascinating read.

Ravel Trees

Ravel trees are a variation of AVL trees with a completely novel approach to deletion – just delete the node from the tree and do no rebalancing. Amazingly, this approach makes the trees easier to implement and must faster in practice.

Why they're worth studying: Ravel trees were motivated by practical performance concerns in database implementation and a software bug that caused significant system failures. They also have some very interesting theoretical properties and use an interesting type of potential function in their analysis. If you're interested in exploring the intersection of theory and practice, this may be a good data structure to explore.

Binary Decision Diagrams

Binary decision diagrams (BDDs) are DAGs that represent boolean functions. They have a huge array of interesting use cases – you can use them to encode graphs space-efficiently, in SAT solvers, and even in program analysis. The basic operations on BDDs are relatively straightforward, and their mathematical and practical properties are quite surprising.

Why they're worth studying: BDDs and their variations are an interesting bridge between data structures and theoretical computer science. Many results on BDDs are closely related to results about tree computation and circuit complexity.

Ropes

Ropes are an alternative representation of strings backed by balanced binary trees. They make it possible to efficiently concatenate strings and to obtain substrings, but have the interesting property that accessing individual characters is slower than in traditional array-backed strings.

Why they're worth studying: Using ropes in place of strings can lead to impressive performance gains in many settings, and some programming languages use them as a default implementation of their string type. If you're interested in seeing applications of balanced trees outside of map and set implementation, this would be a great place to start.

The Burrows-Wheeler Transform

The Burrows-Wheeler transform is a transformation on a string that, in many cases, makes the string more compressible. It's closely related to suffix arrays, and many years after its invention was repurposed for use in string processing and searching applications. It now forms the basis for algorithms both in text compression and sequence analysis.

Why it's worth studying: The Burrows-Wheeler transform and its variations show up in a surprising number of contexts. If you'd like to study a data structure that arises in a variety of disparate contexts, this would be an excellent choice.

GADDAGs

GADDAGs are a data structure designed to speed up computerized Scrabble players. They're a type of automaton that optimizes searching for substrings with the assumption that those substrings would then be extended in a game of Scrabble to form new words. The techniques that go into GADDAGs are influenced by tries and suffix trees, and this might make for an interesting implementation project.

Why they're worth studying: GADDAGs combine techniques from automata theory (minimum-state DFAs), artificial intelligence (heuristic optimizations), and computer game playing. If you're interested in exploring how specialized data structures can significantly speed up AI algorithms, this would be a great choice.

Bloom Filters

Bloom filters are a data structure related to the Count-Min sketch. They're used to compactly represent sets in the case where false negatives are not permitted, but false positives are acceptable. Bloom filters are used in many different applications and are one of the more practical randomized data structures. *(A note: since Bloom filters are sometimes covered in CS161, if you choose to explore Bloom filters for your final project, we will expect you to also explore several variations of Bloom filters as well.)*

Why they're worth studying: Bloom filters have very interesting theoretical properties due to the interactions of multiple independent hash functions, yet are practically relevant as well. They'd be an excellent candidate for a project exploring the interplay of theory and practice.

General Domains of Interest

We covered many different types of data structures in CS166, but did not come close to covering all the different flavors of data structures. Here are some general areas of data structures that you might want to look into.

Persistent Data Structures

What if you could go back in time and make changes to a data structure? Fully persistent data structures are data structures that allow for modifications to older versions of the structure. These are a relatively new area of research in data structures, but there are some impressive results. In some cases, the best dynamic versions of a data structure that we know of right now are formed by starting with a static version of the structure and using persistence techniques to support updates.

Purely Functional Data Structures

The data structures we've covered this quarter have been designed for imperative programming languages where pointers can be changed and data modified. What happens if you switch to a purely functional language like Haskell? Many data structures that are taken for granted in an imperative world aren't possible in a functional world. This opens up a whole new space of possibilities.

Parallel Data Structures

Traditional data structures assume a single-threaded execution model and break if multiple operations can be performed at once. (Just imagine how awful it would be if you tried to access a splay tree with multiple threads.) Can you design data structures that work safely in a parallel model – or, better yet, take maximum advantage of parallelism? In many cases, the answer is yes, but the data structures look nothing like their single-threaded counterparts.

Geometric Data Structures

Geometric data structures are designed for storing information in multiple dimensions. For example, you might want to store points in a plane or in 3D space, or perhaps the connections between vertices of a 3D solid. Much of computational geometry is possible purely due to the clever data structures that have been developed over the years, and many of those structures are accessible given just what we've seen in CS166.

Succinct Data Structures

Pointer-based structures often take up a lot of memory. The humble trie uses one pointer for each possible character per node, which uses up a *lot* of unnecessary space! Succinct data structures are designed to support standard data structure operations, but use as little space as is possible. In some cases, the data structures use just about the information-theoretic minimum number of bits necessary to represent the structure, yet still support operations efficiently.

Cache-Oblivious Data Structures

B-trees are often used in databases because they can be precisely tuned to take advantage of disk block sizes. But what if you didn't know the page size in advance? Cache-oblivious data structures are designed to take advantage of multilayer memories even when they don't know the specifics of how the memory in the machine is set up.

Dynamic Graph Algorithms

This quarter, we talked about dynamic connectivity in trees and will (hopefully) explore incremental and fully-dynamic connectivity in graphs. Other data structures exist for other sorts of dynamic graph problems, such as dynamic connectivity in *directed* graphs, dynamic minimum spanning tree (maintaining an MST as edges can be added and deleted), etc. If you're interested to see what happens when you take classic problems in the style of CS161 and make them dynamic, this might be a great area to explore.

Lower Bounds

Some of the data structures we've covered this quarter are known to be optimal, while others are conjectured to be. Proving lower bounds on various data structures is challenging and in some cases showing that a particular data structure can't be improved takes much more work than designing the data structure itself. If you would like to go down a very different theoretical route, we recommend exploring the techniques and principles that go into lower-bounding the runtime of various data structures.