Suggested Final Project Topics

Here is a list of data structures 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!

If You Liked Range Minimum Queries, Check Out…

Range Semigroup Queries

In the range minimum query problem, we wanted to preprocess an array so that we could quickly find the minimum element in that range. Imagine that instead of computing the minimum of the value in the range, we instead want to compute $A[i] \bigstar A[i+1] \bigstar \ldots \bigstar A[j]$ for some associative operation $\bigstar$. If we know nothing about $\bigstar$ other than the fact that it's associative, how would we go about solving this problem efficiently? Turns out there are some very clever solutions whose runtimes involve the magical Ackermann inverse function.

Why they're worth studying: If you really enjoyed the RMQ coverage from earlier in the quarter, this might be a great way to look back at those topics from a different perspective. You'll get a much more nuanced understanding of why our solutions work so quickly and how to adapt those techniques into novel settings.

Lowest Common Ancestor and Level Ancestor Queries

Range minimum queries can be used to solve the ***lowest common ancestors*** problem: given a tree, preprocess the tree so that queries of the form "what node in the tree is as deep as possible and has nodes u and v as descendants?" LCA queries have a ton of applications in suffix trees and other algorithmic domains, and there's a beautiful connection between LCA and RMQ that gave rise to the first $\langle O(n), O(1) \rangle$ solution to both LCA and RMQ. The ***level ancestor query*** problem asks to preprocess a tree so that queries of the form "which node is k levels above node v in this tree" can be answered as efficiently as possible, and it also, surprisingly, admits a $\langle O(n), O(1) \rangle$ solution.

Why they're worth studying: The Fischer-Heun structure for RMQ was motivated by converting a problem on arrays to a problem on trees and then using the Method of Four Russians on smaller problem instances to solve the overall problem efficiently. Both the LCA and LAQ problems involve creative uses of similar techniques, which might be a great way to get a better sense of how these techniques can be more broadly applied.

Tarjan's Offline LCA Algorithm

The discovery that LCA can be solved in time $\langle O(n), O(1) \rangle$ is relatively recent. Prior to its discovery, it was known that if all of the pairs of nodes to be queried were to be specified in advance, then it was possible to solve LCA with $O(n)$ preprocessing and $O(1)$ time per query made.

Why it's worth studying: Tarjan's algorithm looks quite different from the ultimate techniques developed by Fischer and Heun. If you're interested in seeing how different algorithms for the online and offline cases can look, check this one out!

Area Minimum Queries

On Problem Set One, we asked you to design two data structures for the area minimum query problem, one running in time $\langle O(mn), O(\min\{m, n\}) \rangle$, the other in $\langle O(mn \log m \log n), O(1) \rangle$. It turns out that an $\langle O(mn), O(1) \rangle$-time solution is known to exist, and it's not at all what you'd expect.

Why they're worth studying: For a while it was suspected it was impossible to build an $\langle O(mn), O(1) \rangle$-time solution to the area minimum query problem because there was no way to solve the problem using a Cartesian-tree like solution – but then someone went and found a way around this restriction! Understanding how the attempted lower bound works and how the new data structure circumvents it gives an interesting window into the history of data structure design.

Decremental Tree Connectivity

Consider the following problem. You're given an initial tree T. We will begin deleting edges from T and, as we do, we'd like to be able to efficiently determine whether arbitrary pairs of nodes are still connected in the graph. This is an example of a dynamic graph algorithm: in the case where we knew the final tree, it would be easy to solve this problem with some strategic breadth-first searches, but it turns out to be a lot more complex when the deletes and queries are intermixed. By using a number of techniques similar to the Four Russians speedup in Fischer-Heun, it's possible to solve this problem with linear preprocessing time and constant deletion and query times.

Why it's worth studying: The key insight behind Fischer-Heun is the idea that we can break the problem apart into a top-level large problem a number of smaller bottom-level problems, then use tricks with machine words to solve the bottom-level problems. The major data structure for decremental tree connectivity uses a similar technique, but cleverly exploits machine-word-level parallelism to speed things up. It's a great example of a data structure using a two-layered structure and might be a cool way to get some more insight into just how versatile this technique can be.

Succinct Range Minimum Queries

A succinct data structure is one that tries to solve a problem in the theoretically minimum amount of space, where space is measured in *bits* rather than *machine words*. The Fischer-Heun data structure from lecture is a time-optimal data structure for RMQ, but uses a factor of $\Theta(\log n)$ more bits than necessary by storing $\Theta(n)$ total machine words. By being extremely clever with the preprocessing logic, it's possible to shrink the space usage of an RMQ structure down to just about the theoretical minimum.

Why it's worth studying: Succinct data structures are an exciting area of research and provide an entirely new perspective on what space efficiency means. In studying this data structure, you'll both see some new strategies for solving RMQ and get a taste of just how much redundancy can be removed from a data structure.

Pettie and Ramachandran's Optimal MST Algorithm

The minimum spanning tree problem is one of the longest-studied combinatorial optimization problems. In CS161, you saw Prim's and Kruskal's algorithms for finding MSTs, and may have come across Borůvka's algorithm as well. Since then, a number of algorithms have been proposed for solving MST. In 2002, Pettie and Ramachandran announced a major discovery – they had developed an MST algorithm that was within a constant factor of the optimal MST algorithm! There's one catch, though: *no one knows how fast it is!*

Why it's worth studying: The key insight behind Pettie and Ramachandran's MST algorithm is the observation that they can split a single large MST algorithm into a number of smaller MST problems that are so small that it's possible to do a brute-force search for the fastest possible algorithm for solving MST on those small cases. By doing some preprocessing work to determine the optimal MST algorithms in these cases (you read that right – one step of the algorithm is "search for the optimal algorithm of a certain variety!"), they end up with an algorithm that is provably optimal. The reason no one knows the runtime is that no one knows what the discovered algorithms actually look like in the general case. If you'd like to see a highly nontrivial application of a Four-Russians type technique and to get a good survey of what we know about MST algorithms, check this one out!

A warning: This particular topic will require you to do a decent amount of reading and research on related algorithms that are employed as subroutines. It's definitely more challenging than many of the other topics here. However, if you focus your project on one or two particular parts of the overall algorithm (for example, how the brute-force search for the optimal solution works, or how soft heaps can be used to get approximate solutions to MST that are then further refined), we think you'll have a great time with this.

If You Liked String Data Structures, Check Out...

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 it's 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.

Suffix Trees for Matrix Multiplication

In their paper *Fast Algorithms for Learning with Long N-grams via Suffix Tree Based Matrix Multiplication*, the authors (including the current chair of the CS Department!) devise a way to use suffix trees to speed up the sorts of matrix muliplications that arise in the context of algorithms involving *n*-grams (substrings containing *n* words or characters) from a piece of text. Although suffix trees are known for being a bit of a space hog, the fact that they can store $O(n^2)$ characters in $\Theta(n)$ space allows for some impressive compressions of large matrices.

Why it's worth studying: If you thought that suffix trees were a fun topic in and of themselves, this would be a great way to continue that exploration and to see how topics typically associated with data structures can be applied to machine learning and linear algebra.

Suffix Automata

You can think of a trie as a sort of finite automaton that just happens to have the shape of a tree. A suffix trie is therefore an automaton for all the suffixes of a string. But what happens if you remove the restriction that the automaton have a tree shape? In that case, you'd end up with a suffix automaton (sometimes called a *directed acyclic word graph* or *DAWG*), a small automaton recognizing all and only the suffixes of a given string. Impressively, this automaton will always have linear size!

Why they're worth studying: Suffix automata, like suffix trees, have a number of applications in text processing and computational biology. In many ways, they're simpler than suffix trees (the representation doesn't require any crazy pointer compress tricks, for example). If you liked suffix trees and want to see what they led into, 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.

LCP Induction

The SA-IS algorithm we explored in class can be modified to also produce LCP information for the suffix array, and to do much faster than traditional LCP construction algorithms. As you've seen, LCP information is extremely useful for all sorts of suffix array operations, so the practical speedup here is a big deal!

Why it's worth studying: Induced sorting is a very clever technique that touches on all sorts of nuances of the structure of a string's suffixes. If you're interested in exploring more of the hidden structure of strings and substrings, this would be a great way to do so while further solidifying your understanding of the SA-IS algorithm.

Ukkonen's Algorithm

Prior to the development of SA-IS as a way of building suffix arrays, the most popular algorithm for building suffix trees was Ukkonen's algorithm, which combines a number of optimizations on top of a relatively straightforward tree-building procedure to build suffix trees in time $O(m)$. Amazingly, the algorithm works in a streaming setting – it can build suffix trees incrementally as the characters become available!

Why it's worth studying: Ukkonen's algorithm is still widely-used and widely-taught in a number of circles because its approach works by exploiting elegant structures inherent in strings. In particular, Ukkonen's algorithm revolves around the idea of the *suffix link*, a link in a suffix tree from one node to another in a style similar to the suffix links in Aho-Corasick string matching. This algorithm is significant both from a historical and technical perspective and would be a great launching point into further study of string algorithms and data structures.

Levenshtein Automata

Levenshtein distance is a measure of the difference between two strings as a function of the number of insertions, deletions, and replacements required to turn one string into another. Although most modern spell-checkers and autocomplete systems are based on machine-learned models, many systems use Levenshtein edit distance as a metric for finding similar words. Amazingly, with a small amount of preprocessing, it's possible to build an automaton that will match all words within a given Levenshtein distance of an input string.

Why it's worth studying: The algorithms involved in building Levenshtein automata are closely connected to techniques for minimizing acyclic finite-state automata. If you're interested in seeing the interplay between CS154-style theory techniques and CS166-style string processing, this might be an interesting place to start!

FM-Indices

Suffix arrays were initially introduced as a space-efficient alternative to the memory-hogging suffix trees. But in some cases, even the suffix array might be too costly to use. The FM-index (officially, "***F***ull-text index in ***M***inute space," but probably more accurately named after the authors Ferragina and Manzini) is related to suffix arrays, but can often be packed into sublinear space. If you were a fan of the stringology bits that we covered when exploring suffix trees and suffix arrays (shared properties of overlapping suffixes, properties of branching words, connections between substrings and suffixes, etc.), then this would be a great way to dive deeper into that space.

Why it's worth studying: FM-indices interface with a number of other beautiful concepts from string data structures (Burrows-Wheeler transform) and succinct data structures (wavelet trees). Exploring this space will give you a real sense for what data structure design looks like when the goal is minimizing memory usage, in both a practical ("it needs to fit in RAM") and theoretical (can we do better than $\Theta(m)$?) sense.

If You Liked Balanced Trees, Check Out...

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.

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 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.

RAVL Trees

RAVL 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: RAVL 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 structure to explore.

B+ Trees

B+-trees are a modified form of B-tree that are used extensively both in databases and in file system design. Unlike B-trees, which store keys at all levels in the tree, B+ trees only store them in the leaves. That, combined with a few other augmentations, make them extremely fast in practice.

Why they're worth studying: B+ trees are used in production file systems (the common Linux `ext` family of file systems are layered on B+ trees) and several production databases. If you're interested in exploring a data structure that's been heavily field-tested and optimized, this would be a great one to look at!

Cache-Oblivious Binary Search Trees

To get maximal efficiency from a B-tree, it's necessary to know something about the size of the disk pages or cache lines in the machine. What if we could build a BST that minimized the number of cache misses during core operations without having any advance knowledge of the underlying cache size? Such a BST is called a *cache-oblivious binary search tree* and, amazingly enough, we know how to design them by adapting techniques from van Emde Boas trees.

Why they're worth studying: Cache-oblivious data structures are a recent area of research that has garnered some attention as cache effects become more pronounced in larger systems. If you're interested in seeing a theoretically elegant approach to combatting caches – and if you're interested in testing them to see how well they work in practice – this would be a great place to start.

R-Trees

R-trees are a variation on B-trees that store information about rectangles in 2D or 3D space. They're used extensively in practice in mapping systems, yet are simple enough to understand with a little bit of study.

Why they're worth studying: R-trees sit right at the intersection of theory and practice. There are a number of variations on R-trees (Hilbert R-trees, R* trees, etc.) used in real-world systems. If you're interested in exploring geometric data structures and potentially implementing some of your own optimizations on a traditional data structure, you may want to give these a try!

A Caveat: While the original paper on R-trees is a good introduction to the topic, there's been a lot of work in advancing R-trees since then. Much of the efficiency gains reported are *practical* improvements rather than *theoretical* improvements, meaning that a study of R-trees will likely require you to do your own investigations to determine which optimizations are worthwhile and why. You should be prepared to do a good amount of independent work verifying the claims that you find, since computer hardware has changed a lot since R-trees first hit the scene.

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 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 creative applications of balanced trees, this would be a great place to start.

A Caveat: Ropes are not as complicated as the other data structures on this list and many of the papers about ropes have incorrect runtime bounds. If you choose to use this data structure, we'll expect that you'll put in a significant amount of work comparing them to other approaches or developing novel ideas.

Succinct Trees

What is the the information-theoretic minimum number of bits necessary to represent a balanced tree in a way that lets you still perform queries with the regular time bounds? The standard representation of a binary search tree with n nodes, for example, uses $\Theta(n \log n)$ bits just for the tree structure, since each pointer requires a machine word to store. Amazingly, it's possible to compress the tree representation down to $2n + o(n)$ bits without sacrificing the runtime of performing standard operations on these trees.

Why they're worth studying: Succinct trees exist at the interplay between information theory (why can't you encode trees is less than $2n + o(n)$ bits?), data structure isometries (representing trees as cleverly-chosen bitvectors), and properties of trees (exploiting some hidden structure that was there all along). On top of this, these techniques can be used in practice to build highly compact suffix trees and suffix arrays, making it possible to push forward our understanding of computational biology with limited hardware.

Purely Functional Red/Black Trees

In purely functional programming languages, it's not possible to modify data structures after they've been constructed. Any updates that happen must be performed by building a new version of the data structure, possibly sharing some existing pieces of the original structure in a read-only way.

Although red/black trees are tricky to implement in imperative languages, it's possible to implement them in purely functional programming languages, maintaining all the existing time bounds, in about a page of code. This means that it's possible to implement red/black trees in a way that lets us "go back in time" to see what the data structure looked like in the past and to use them in programming languages and settings where updates aren't permitted.

Why they're worth studying: Purely functional red/black trees are a good first taste of what data structures look like in a purely functional setting. If you're interested in seeing what data structures might have evolved into if we thought about programming differently, or if you're a fan of functional languages and want to get a chance to experiment with data structure design in that space, this might be a great place to look for inspiration!

Sequence Heaps

In one sense, you can think of the binomial or Fibonacci heap as a data structure specifically optimized to get good results in Theoryland. What happens if you take the other approach and design a priority queue where all that matters is wall-clock runtime? With a good deal of engineering effort, you might end up with the sequence heap, a priority queue that's specifically designed to play nicely with caches. Amazingly, this data structure can outperform a regular binary heap, even though binary heaps are already packed densely into arrays.

Why they're worth studying: Practically speaking, sequence heaps are among the fastest priority queues in practice across a number of different workflows. Theoretical speaking, sequence heaps are based on a number of clever techniques for combining data streams together while playing well with caches. If you enjoyed our discussion of B-trees and the idea of optimizing for cache transfers rather than instruction counts, this would be a great data structure to investigate.

Iacono's Working Set Structure

Splay trees have the wonderful property that recently-accessed elements are much faster to look up than infrequently-accessed elements. John Iacono developed a data structure that also has this property (called the *working set property*) but which guarantees fast accesses in a worst-case rather than amortized-efficient sense. The strategy employed – using an ensemble of smaller data structures in concert with one another – is quite beautiful.

Why they're worth studying: Along with queaps and fishspears (described later), the working set structure is designed to provide good performance not just in the worst case, but in the sorts of cases that tend to come up in practice. We honestly don't know how fast this structure is in practice – does it compare favorably against a regular balanced BST or a splay tree? – and diving into that question would be a great way to gauge how well the mathematical theory matches practice.

Rotation Distance

Tree rotations are a fundamental primitive on binary search trees, and in fact, any two binary search trees with the same set of keys can be transformed into one another using tree rotations. That then sparks a question: what's the maximum number of rotations needed to turn one BST into another? By reframing binary search trees as triangulations of convex polygons and mapping everything into hyperbolic space, it's possible to get both an upper and lower bound to this question.

Why it's worth studying: Most papers on this topic, including the foundational paper *Rotation Distance, Triangulations, and Hyperbolic Geometry* and the more recent *The Diameter of Associahedra*, take the idea of isometries and run wild with them. You'll see connections between binary search trees, polygon triangulations, strings balanced strings of parentheses, etc. pop up, which, surprisingly, connects with the stack-based algorithm for building RMQ structures. The math in this subject area is fairly demanding, but the ideas are so beautiful that it shouldn't be hard to find a small piece of the bigger picture to dive deeper into.

Other Balanced Tree Variants

There are a gazillion ways you can maintain balance in a search tree. Here's a sampling of other approaches that didn't fit cleanly into any of the other categories.

- 2-3 trees
- AA-trees
- AVL trees
- (a, b) trees
- B* trees
- BB[α] trees
- Left-leaning red/black trees
- Rank-balanced trees
- Weak AVL (WAVL) trees
- Zip trees

The fact that we haven't scouted these out doesn't mean that they aren't interesting – it just means that we don't know much about them! If any of these spark joy, feel free to pursue them as a final project topic! We'd love to see what you find.

If You Liked Amortized-Efficient Data Structures, Check Out…

Hollow Heaps

Ever since the Fibonacci heap was introduced, there's been a push to find a simpler data structure meeting the same time bounds. Hollow heaps are a recent (2015) data structure that matches the theoretical bounds of the Fibonacci heap, but which rely on a much simpler set of structural properties.

Why they're worth studying: Hollow heaps are a fairly accessible modern data structure. Their design and analysis is reminiscent of that of the Fibonacci heap, with a few splashes of other concepts from amortization. Looking into hollow heaps would be a great way to do a compare-and-contrast across multiple data structures and to learn about the progress in the field since Fibonacci heaps first hit the scene.

Quake Heaps

Quake heaps, like hollow heaps, are a modern (2013) data structure designed to simplify the presentation of Fibonacci heaps. The key idea behind quake heaps is fundamentally different than that of Fibonacci heaps – they work by letting the heap get into an imbalanced state before "seismically" rebuilding the data structure from scratch. In doing so, they meet all the existing time bounds of Fibonacci heaps, but with a much simpler potential function.

Why they're worth studying: The major theoretical technique involved in quake heaps – globally rebuilding a structure when it's not in a good state – is a creative approach to designing a data structure. This technique can be repurposed in other places and might be a great launching point for revisiting older structures and looking for improvements.

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.

Fishspears

The *fishspear* is a priority queue with an unusual runtime performance – the amortized cost of a deletion is O(1), and the amortized cost of an insertion varies based on the relative size of the element removed. Removing smaller elements tends to be much faster than removing larger elements, so if you stream random elements through a fishspear and focus your deletion efforts primarily on small elements, the performance will be much better than what a binary heap will be able to match.

Why they're worth studying: We've spent a lot of time this quarter focusing on worst-case analyses of data structures (amortization is a variation on this theme), plus expected performance. The fishspear occupies an interesting place in that its analysis is somewhat output-sensitive (the longer an item is in the fishspear, the more expensive it is to insert it), allowing for a more nuanced understanding of the data structure. This would be a great launching point for a further exploration of how to analyze data structures beyond just thinking about the most pessimal case.

Queaps

The queap is a hybrid between a queue and a heap (hence the name) that's layered on top of a 2-3-4 tree. Like the fishspear, the queap's performance depends on how the data structure actually ends up getting used. Specifically, the queap has the property that the cost of removing an element depends purely on how many elements have been in the queap longer than the removed element. This means that if the elements inserted into the queap are mostly sorted, the runtime will approach that of a queue.

Why they're worth studying: Queaps combine a number of beautiful theoretical ideas – repurposing a 2-3-4 tree to act as a queue, analyzing data structures with respect not to the worst case but to the actual performance – etc. This would be a great way to combine lots of clever ideas from this class together into a single spot, and we'd be curious to see how well it holds up in practice!

Multisplay Trees

Tango trees (described earlier in the section on balanced trees) combine together multiple red/black trees and are never worse than a factor of O(log log n) than the best possible binary search tree. If you replace the red/black trees with splay trees, you get the *multisplay tree*, which is also never worse than a factor of O(log log n) from the best possible binary search tree. However, due to some of the properties of splay trees, we also know that multisplay trees have a number of other useful, amazing properties.

Why they're worth studying: You'd be amazed what we do and don't know about splay trees. If you're interested in getting a better sense for our understanding of splay trees and dynamic optimality, this would be an excellent starting point.

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.

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.

A Caveat: This data structure is surprisingly tricky to understand. The initial paper includes C code that gives the illusion that the structure is simple, but the math is deceptive and the motivation behind the data structure is not clear from the paper. We have complete faith that a team that chose to present this topic could do so in a way that makes them clear and intuitive, and we'd love to see what you come up with!

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.

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.)

Splay Tree Variations

Splay trees tend to work well in practice, but splaying can be quite expensive. Since splay trees have been introduced, there have been a number of variations proposed that attempt to match many of the theoretical guarantees of splay trees but with lower constant factors. These range from simple variations like only splaying every *k*th operation to probabilistically splaying on each access.

Why they're worth studying: Despite their nice theoretical guarantees, splay trees aren't often used in practice due to a combination of their amortized-efficient guarantees and the practical costs of splaying. If you're interested in exploring that annoying gap between theory and practice, consider checking these variations out!

The Geometric Lower Bound

There's been a recent development in the quest for the optimal binary search tree: a lower bound called the *geometric lower bound* based on casting binary search tree lookups in a geometric setting. This lower bound has been used to design a new type of dynamic binary search tree that is conjectured to be optimal and looks quite different from the other trees we've studied this quarter.

Why it's worth studying: If you had a dynamically optimal BST, how would you know? Lower-bounding techniques like the one developed here are one of the best tools we've got for reasoning about the limits of BSTs. If you're interested in seeing how these techniques are developed and would be up for a bit of a history lesson, this might be a great place to start.

Crazy Good Chocolate Pop Tarts

In their paper “De-Amortizing Binary Search Trees,” Bose et al. provide a data structure transformation that takes any amortized-efficient binary search tree and converts it into a new BST data structure that maintains all of the existing amortized time bounds and guarantees worst-case O(log *n*) runtimes for each operation. Their technique is based on a data structure called the chocolate pop tart, which they prove is “crazy good” for an appropriate definition of “crazy good.” Impressively, this means that there’s a transformation on splay trees that preserves all their amortized-efficient properties, yet guarantees that each operation is asymptotically no slower than a perfectly-balanced tree.

Why they’re worth studying: One of the major techniques employed in this transformation is the decomposition of a binary search tree into a series of chains, each of which is formed by linking the parent node to its heavier child. This closely mirrors the blue/red analysis we did of splay tree operations, yet converts it from an accounting trick into a real data structure design technique. This would be a great way to learn more about that technique while also seeing novel ways of simulating binary search trees.

The Deque Conjecture

Suppose that – for some reason – you decide to implement a deque using a splay tree. That is, you'll support the insertion and deletion of elements at the front or back of a sequence, using a splay tree as the underlying mechanism for storing elements. (This isn't as crazy an idea as it might initially seem; the *finger tree* data structure does precisely this and gets a lot of love in the functional programming community.) How fast would the deque be? The conjecture is that operations should run in amortized O(1) each. We haven't yet proved this, but we're getting closer!

Why it's worth studying: Exploring what we know about the deque conjecture is a great way to learn about the techniques people have explored in the quest for dynamic optimality. This would be a great launching point to learn more about, say, how we lower-bound the costs of sequences of operations on *any possible* binary search tree. And as a real kicker, the best upper bound we have right now is the mind-bogglingly-slowly-growing $O(\alpha^*(n))$. To understand what you're looking at, $\alpha(n)$ is the *Ackermann inverse function*, which grows so slowly that we need special notation to write out a value of n for which $\alpha(n) = 6$. The function $\alpha^*(n)$ is the *iterated Ackermann inverse function* – the number of times we need to apply α to a value before it drops to a constant. This might be the slowest-growing function ever used in a serious proof in computer science!

If You're Interested in Randomized Data Structures, Check Out...

The HyperLogLog Estimator

The HyperLogLog estimator is a data structure for estimating how many unique elements exist in a data stream. Impressively, it can get a high-quality estimate using an extremely small number of total bits, making it a popular choice in database implementations and data centers.

Why they're worth studying: Cardinality estimators have a long history and are used extensively in database systems to determine which of several different algorithms should be used when performing operations on huge data sets. Additionally, the mathematical analyses involved in cardinality estimators are quite clever and will give you a sense of how nontrivial the analysis of a randomized data structure can be.

A Caveat: The math behind the HyperLogLog data structure is fairly challenging and technical. We're confident that a hardworking team could find a way to simplify the presentation or at least make it more intuitive, and we'd be extremely grateful to any team that manages to do this!

Bloomier Filters

A *Bloomier filter* is based on the following, seemingly improbable idea: can you use a Bloom filter, which is used to approximately encode a set, to approximately encode a map? Amazingly, using random graph theory and a clever use of the XOR operator, it's possible to do this in a way where "false lookups" are comparably rare and the memory usage is kept to a minimum.

Why they're worth studying: On a theoretical level, Bloomier filters are a fascinating data structure. The main underlying idea is to use XORs in a way that makes it highly unlikely that missing keys will get an associated value, but which makes present keys uniquely map to a specific value. On a practical level, Bloomier filters have recently gotten popular in the compression of deep neural networks, where they represent a way to (noisly) encode weights between layers in very few bits.

Tabulation Hashing

In lecture, we've taken it as a given that we have good hash functions available. But how exactly do you go about building these hash functions? One popular approach – which has the endorsement of CS legend Don Knuth – is *tabulation hashing*, which breaks a key apart into multiple blocks and uses table-based lookups to compute a hash code. Although tabulation hashing only gives 3-independence, which doesn't sound all that great, a deeper analysis shows that using this hashing strategy in certain contexts will perform much better than what might initially appear to be the case.

Why it's worth studying: On a practical note, we think that learning more about how to build hash functions will give you a much better appreciation for the power of randomized data structures. On a theoretical note, the math behind tabulation hashing and why it's so effective is beautiful (but tricky!) and would be a great place to dive deeper into the types of analyses we've done this quarter.

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.

FKS Hashing

The cuckoo hashing strategy we described in class is one way of making a dynamic perfect hash table. As beautiful a technique as it is, it doesn't have the best utilization of memory, and it seems to require strong hash functions. Another technique for perfect hashing, the *FKS hash table*, uses weaker hash functions and is based on a surprising idea: what if we resolved hash collisions by using another hash table?

Why it's worth studying: FKS hashing is interesting in that it builds nicely on the mathematical techniques we'll cover for working with randomized data structures and that it was one of the earliest strategies devised for building perfect hash tables. Going from the static case to the dynamic case is a bit tricky, but is a great way of combining amortization and randomization in a single package.

Concurrent Hash Tables

Many simple data structures become significantly more complex when running in multithreaded environments. Some programming languages (most famously, Java) ship with an implementation of a hash table specifically designed to work in concurrent environments. These data structures are often beautifully constructed and rely on specific properties of the underlying memory model.

Why they're worth studying: Concurrent hash tables in many ways look like the hash tables we know and love, but necessitate some design and performance trade-offs. This would be a great way to see the disconnect between the theory of hash tables and the practice.

Cuckoo Hashing Variants

The original paper on cuckoo hashing suggested the hashing strategy we talked about in class: maintain two tables and displace elements by bouncing them back and forth between the tables. Since then, a number of variations have been proposed on cuckoo hashing. What if we use $k > 2$ tables instead of two tables? What if each table has its entries subdivided into a number of "slots" that can store multiple elements? Many of these variations on cuckoo hashing have proven to be extremely efficient in practice, while others have fantastic theoretical efficiency.

Why they're worth studying: As you saw from lecture, cuckoo hashing is a simple idea with a surprisingly complex analysis. Many of the updates to cuckoo hashing are known to work well in practice, but have been tricky to analyze in a mathematically rigorous fashion. If you're looking for a project where you'll get some exposure to really cool mathematical techniques while also getting the chance to try out the techniques in practice, this might be a great place to begin.

The AMS Sketch

The AMS sketch (named after its inventors Alon, Matias, and Szegedy) was one of the first modern streaming data structures and set the stage for much future development on the subject. The paper was on the complexity of estimating the frequency norms F_p of a data stream, which generalizes the L_1 and L_2 norms we explored in the context of the count and count-min sketches.

Why they're worth studying: The techniques used in the AMS sketch combines techniques from a number of different fields of mathematics: random matrix projections, streaming algorithms, and information theory, to name a few. Additionally, F_p-norm sketches are one of the few cases where we have strong lower bounds on the time and space complexity of any data structure. If you'd like to see some fun math that leads to proofs of correctness and optimality, this might be a great place to start.

Hopscotch Hashing

Hopscotch hashing is a variation on open addressing that's designed to work well in concurrent environments. It associates each entry in the table with a "neighborhood" and uses clever bit-masking techniques to quickly determine which elements in the neighborhood might be appropriate insertion points.

Why it's worth studying: Hopscotch hashing is an interesting mix of theory and practice. On the one hand, the analysis of hopscotch hashing calls back to much of the formal analysis of linear probing hash tables and therefore would be a good launching point for a rigorous analysis of linear probing. On the other hand, hopscotch hashing was designed to work well in concurrent environments, and therefore might be a good place to try your hand at analyzing parallel data structures.

Why Simple Hash Functions Work

Many of the data structures we talked about (cuckoo hashing, count sketches) required hash functions with strong independence guarantees. In practice, people tend to write pretty mediocre hash functions that don't meet these criteria, yet amazingly they tend to work out quite well. Why exactly is this? In 2007, Michael Mitzenmacher and Salil Vadhan published a paper explaining why, in most cases, weak hash functions work well by showing how they preserve the underlying entropy in the data source.

Why it's worth studying: Hashing is one of those areas where the theory and practice are quite different, and this particular line of research gives a theoretically rigorous explanation as to why this gap tends not to cause too many problems in practice. If you're looking for a more theory-oriented project that could potentially lead to some interesting implementation questions, this would be an excellent launching point.

The Johnson-Lindenstrauss Lemma

Many of the randomized data structures developed recently are based on a mathematical result called the *Johnson-Lindenstrauss lemma* which states, intuitively, that you can take high-dimensional data and project it into a relatively low-dimensional subspace without stretching the distances between those data points too much. This lemma was (transitively) the inspiration for the count sketch data structure and can be used as a building block for other techniques like locality sensitive hashing.

Why it's worth studying: The perspective shift involved in understanding the Johnson-Lindenstrauss lemma will give you a fundamentally different way of looking at many of the proofs about randomized data structures from this quarter. It's also a beautiful piece of mathematics, combining techniques from linear algebra and matrix theory with more basic concepts in data structures. A project that aimed to present this lemma in a simple way and tie it into the concepts from this quarter would be a great way to round out the quarter.

Distributed Hash Tables

Hash tables work well in the case where all the data is stored on a single machine, but what if you want to store data in a decentralized fashion with unreliable computers? There a number of techniques for building such distributed hash tables, many of which are used extensively in practice.

Why they're worth studying: We've typically analyzed data structures from the perspective of time and space usage, but in a distributed setting we need to optimize over entirely different quantities: the required level of communication, required redundancy, etc. Additionally, distributed hash tables are critical to peer-to-peer networks like BitTorrent and would be a great way to see theory meeting practice.

Fountain Codes

Imagine you want to distribute a large file to a number of receivers over a broadcast radio. You can transmit the data to the receivers, but they have no way of letting you know what they've received. How might you transmit the data in a way that makes new listeners have to wait as little as possible to get all the data? A family of techniques called fountain codes works by transmitting XORed blocks of data from the original source to a number of receivers, who can then work to decode the original message. By being strategic with how the data is sent, receivers will only need to listen for a surprisingly short time.

Why they're worth studying: Fountain codes are a great mix of theory and practice. Theoretically, they touch on a number of challenges in data storage and transmission and give rise to some interesting and unusual probability distributions. Practically, there's been talk of adopting fountain codes as a way of transmitting updates to devices like cell phones and cars in a way that doesn't require a huge number of end-to-end transmissions.

Hash Tables in Practice

Hash tables are used everywhere and their practical, as opposed to theoretical, efficiency is extremely important. Companies like Google have invested significant energy in designing fast, general-purpose hash tables for use in their systems, and professional programmers have often discovered or rediscovered new theoretical ideas that lead to practical improvements. What exactly are these hash tables doing that make them so fast?

Why they're worth studying: At the end of the day, the whole purpose of designing and building data structures is to improve our ability to solve problems in practice. And, as you've seen before, just because one data structure has an attractive big-O runtime doesn't mean that those hidden constant factors are worth it. This would be a great way to see what's actually done in The Real World, how it compares to the theoretical analyses we've done over the quarter, and (possibly) even build your own custom and fast hash table!

Learned Index Structures

A *learned index structure* is a hybrid between a classical data structure (often, a B-tree or a Bloom filter) and a deep learning model. The idea is to have the deep model learn some sort of common patterns present in input data sets and make an educated guess of how to handle an element, falling back on the classical data structure to avoid errors.

Why they're worth studying: Learned index structures are a fairly recent (2017) idea and it's unclear whether they're here to stay or just a passing fad. Investigating how these data structures perform in practice, and potentially adding your own contributions to the field, would be a great way to get a sense of the shape of things to come.

The CR-Precis

The count sketch and count-min sketch work by using randomly-selected hash functions to distribute elements (and, in the case of the count sketch, to choose the sign to associate with each element). A natural question then arises: can you get the same sort of guarantees from a similar data structure that doesn't use any randomization? Amazingly, the answer is *yes*, and it's thanks to the magic of number theory.

Why it's worth studying: In one sense, the CR-precis is familiar territory: it's literally a count-min sketch with a deterministic choice of hash functions. In another, it's a whole different world from the count-min sketch, since it has to somehow get all the benefits typically associated with randomization in a totally deterministic way. The math powering the CR-precis is beautiful and would be a great launching point for further exploration of the field of *derandomization*.

Locality-Sensitive Hashing

A locality-sensitive hash function is a hash function that, counterintuitively, attempts to make similar elements collide with one another. The idea is that elements that are close to one another in some space should have a high probability of a hash collision, while elements that are far apart should be kept separate. This can be used to do nearest-neighbor searching and as a subroutine in a huge number of modern algorithms.

Why they're worth studying: The mathematics behind locality-sensitive hash functions involves working with random matrix theory and studying what happens if you project higher-dimensional objects into lower-dimensional spaces. They're also one of the more exciting theoretical primitives to have been developed in recent times, so looking into LSH might also introduce you to a number of other beautiful techniques you wouldn't have otherwise encountered.

Min-Hashing

How do you determine how similar two documents are? That's a major question you'd need to answer if you were trying to build a search engine or indexing huge amounts of text. One technique for solving this problem involves techniques reminiscent of the cardinality estimation techniques we'll cover: hash the contents of the documents with different hash functions and take the smallest value produced. Amazingly, these minimum hash values reveal a huge amount about the documents in question.

Why they're worth studying: Min-hashing was initially developed to solve practical, real-world problems, and it's one of those beautiful approaches that combines theoretical elegance and pragmatic utility. Since they were initially developed, there have been a number of questions about how to generalize the idea to other problems and how to improve on the basic method, so this would be a great launching point for further exploration of the space.

Pagh, Pagh, and Rao's Optimal Approximate Membership Query Structure

In 2005, Pagh, Pagh, and Rao explored whether it was possible to improve on a few of the key weaknesses of standard Bloom filters (the need for multiple hash functions, the strength required to get those hash functions working properly, etc.). They devised a beautiful series of reductions between problems that gives rise to an asymptotically optimal data structure for approximate membership queries.

Why it's worth studying: This particular data structure combines together a number of powerful ideas from Theoryland. It's partially based on succinct data structures, data structures designed to use close to the information-theoretic minimum number of bits for solving a problem. It uses several reductions: one from approximate membership query to exact membership query, and one from storing sets to storing multisets. And it touches on data structure lower bounds – how do they know they can't push the space usage down further? This would be a great way to explore the theory behind advanced data structures and to see how all these ideas come together in one place.

If You're Interested in Integer Data Structures, Check Out...

van Emde Boas Trees

The *van Emde Boas tree* was one of the first data structures to support the same operations as a regular binary search tree (insertion, deletion, lookup, predecessor, and successor) for integers in sublogarithmic time. It supports all these operations in time O(log log U), where U is the upper bound on the integers stored. This is both exponentially faster than a regular BST and matches the time bounds of the later *y*-fast trie. However, the strategy that the vEB tree uses is fundamentally different than that of the *y*-fast trie, and it's been adapted for use in later data structures.

Why they're worth studying: It's possible to think about vEB trees in a number of different ways. You can either think of them as binary tries sliced through the middle, or as an optimization on a raw bitvector scan. That first approach is a novel perspective on tree searches and can be generalized to the *van Emde Boas layout*, a way of arranging nodes in a binary search tree to minimize cache misses. That second approach makes it possible to view vEB trees as an optimized version of the blocking decomposition strategy you've seen used for RMQ and elsewhere. Studying vEB trees with these perspectives in mind would be a great way to come full circle with the techniques we've covered this quarter.

Strees

The *Stree* is a highly optimized implementation of a vEB tree engineered to specifically be fast for 32-bit integer keys. The Stree was introduced in a paper by Dementiev et al as a way of exploring whether it was possible to adapt the general-purpose vEB tree into something that could practically, not just theoretically, outcompete a balanced BST. By combining a number of different strategies together into a unified whole, they were able to achieve this goal.

Why they're worth studying: Throughout the quarter, you've seen that some data structures are fast in Theoryland, some are fast IRL, and some are (coincidentally) fast in both. The Stree represents one way of starting with a jewel of Theoryland and turning it into something workable. If you're interested in tinkering and tweaking an existing structure to see if you can improve upon it – or generalize it to work for 64-bit machines – this would be a great starting point.

Radix Heaps

Fibonacci heaps are of great theoretical interest because they give O($m + n \log n$)-time implementations of both Dijkstra's algorithm and Prim's algorithm. Now, suppose that you'd like to use one of those algorithms and it happens to be the case that every edge in the graph has integral weight, and that weight is at most some number C. Could you take advantage of integer operations to speed things up? The answer is yes, and one of the first major steps toward doing so was the *radix heap*, which improves upon Dijkstra's algorithm for sparse graphs in this case.

Why they're worth studying: Radix heaps are interesting in that it's possible to start off with a fairly straightforward set of observations about how Dijkstra's algorithm operates to get an O($m + nC$)-time implementation, and then refine that first to O($m + n \log C$) and from there to O($m + n (\log C)^{1/2}$) through more and more creative observations. In that sense, this is a fairly accessible data structure that goes pretty deep into Dijkstra's algorithm. Later algorithms and data structures have improved upon the runtime even further, and this could also be a great launching point for further exploration.

Exponential Trees

Fusion trees were the first data structure designed for the transdichotomous machine model that are theoretically faster than comparison-based data structures in all cases, but they aren't the last word on the subject. The exponential tree provides a way to convert from static integer data structures to dynamic integer data structures, enabling fast, deterministic algorithms for integer sorting that are much faster than $O(n \log n)$.

Why they're worth studying: Studying exponential trees would be a great way to dive even deeper into the worlds of word-level parallelism (along the lines of the fusion tree) and trie-based integer algorithms (like the *y*-fast trie). They would also be a great way to study amortized analysis in depth, since the techniques employed there are similar to what was used in *y*-fast tries and fusion trees.

Priority Queues from Sorting

Given a priority queue, it's easy to build a sorting algorithm: just enqueue everything into the priority queue and then dequeue everything. It turns out that the converse is also possible – given a sorting algorithm, it's possible to construct an efficient priority queue that's internally backed by that sorting algorithm. There are a number of constructions that make this possible, most of which assume that the data are integers and many of which use a number of clever techniques.

Why they're worth studying: In trying to convert from a black-box sorting algorithm to a priority queue, it's often important to reason about the specific model of computation being used. Depending on whether randomization is permitted or what restrictions there are on the sorts of bitwise operations can be performed by the machine in constant time, the slowdown introduced in the construction can vary widely. If you're interested in both seeing a cool construction and learning about models of computation in the context of data structure design, this would be a great place to start.

Signature Sort

It's possible to sort in time $o(n \log n)$ if the items to sort are integers (for example, using radix sort). What are the limits of our ability to sort integers? Using advanced techniques, *signature sort* can sort integers in time $O(n)$ – assuming that the machine word size is $\Omega((\log n)^{2+\varepsilon})$.

Why it's worth studying: Signature sort employs a number of clever techniques: using bitwise operations to perform multiple operations in parallel, using tries to sort integers as though they were strings on a small alphabet, etc. This would be a great way to see a bunch of techniques all come together!

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.

Consider looking up: Full retroactivity with O(log *n*) slowdown; confluently persistent data structures.

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.

Consider looking up: Skew binomial random access lists, data-structural bootstrapping.

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.

Consider looking up: Concurrent skip lists, concurrent priority queues.

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.

Consider looking up: *k*-d trees, quadedges, fractional cascading.

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.

Consider looking up: Wavelet trees, succinct suffix trees.

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.

Consider looking up: van Emde Boas layout, cache-oblivious sorting.

Dynamic Graph Algorithms

It's not very hard to efficiently determine whether two nodes are reachable from one another. It's *much* harder to do this when the underlying graph is changing and you don't want to recompute things from scratch. Dynamic graph algorithms are data structures for solving classical graph problems (connectivity, MST, etc.) while the underlying graph updates. 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.

Consider looking up: Dynamic connectivity, top trees, disjoint-set forests.

Logical Data Structures

Suppose you need to store and manipulate gigantic propositional formula, or otherwise represent some sort of boolean-valued function. How could you do so in a way that makes it easy to, say, evaluate the function, or compose several functions together? A number of data structures have been designed to solve these problems, each of which have to contend with **NP**-hard or co-**NP**-hard problems yet work quite well in practice.

Consider looking up: Binary decision diagrams, majority-inverter graphs.

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.

Consider looking up: Wilbur's bounds, predecessor lower bound, BST dynamic optimality.