CS166 Handout 09
Spring 2016 April 28, 2016

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!

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 $A[i] \star A[i+1] \star \dots \star A[j]$ for some associative operation $\star$. If we know nothing about $\star$ 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.

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.

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!

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.

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, 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. On the other hand, if you're looking for a chance to really dive deep into a topic and see what we know about it, and if you're up for a challenge, this would be a fantastic topic to explore.

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

The Commentz-Walter String Matching Algorithm

The Commentz-Walter string matching algorithm is a generalization of the Boyer-Moore string searching algorithm, which (surprisingly) can match strings in sublinear time!

Why it's worth studying: Although Commentz-Walter is known to run in sublinear time on many inputs and to have a quadratic worst-case complexity, not much is known about the algorithm's theoretical properties. If you're interested in taking an algorithm that works well in practice and exploring exactly why that is, this would be a great starting point.

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 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.

Factor Oracles

Suffix trees are extremely powerful but use a ton of memory, and suffix arrays are more compact but a bit less flexible. Factor oracles are a more recent data structure designed to give the power and flexibility usually associated with suffix trees in significantly less space. They're used in a number of theoretical and practical applications.

Why they're worth studying: Interestingly, factor oracles work so well in practice because they sometimes give false positives on matches. If you're interested in seeing how relaxing the requirements on the string structure makes it possible to do extremely fast string matching, check this out!

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.

Suffix Trays

When we talked about the cost of performing searches in a suffix tree or suffix array, we implicitly assumed that the size of the alphabet was a constant and therefore could be ignored from the analysis. But what if we don't have a constant-sized alphabet? In that case, suffix trees and suffix arrays are still quite fast, but aren't asymptotically optimal. A hybrid structure between the two, called the *suffix tray*, achieves lookups in time $O(n + \log |\Sigma|)$, asymptotically better than either a suffix tree or suffix array.

Why they're worth studying: The main technique involved in suffix trays – slicing a tree into nodes of high degree and nodes of low degree – is a common technique in advanced data structures and is conceptually not too hard to understand. This would be an excellent starting point for learning more about this trick.

Kasai's LCP Algorithm

When we discussed suffix arrays, we mentioned the importance of computing LCP information for the strings in the suffix array. Although we mentioned Kasai's algorithm, which computes LCP information in time $\Theta(m)$, we never actually discussed how it worked. Studying up on this would make for an interesting final project.

Why it's worth studying: Kasai et al's original paper not only outlines how to build LCP arrays, but also talks about how to use them to simulate a number of suffix tree operations efficiently purely given a suffix array and LCP information. If you're interested in seeing just how much expressive power suffix arrays have – and you'd like to learn more about the structure of strings and suffixes – check this one out!

Double-Array Tries

As you saw in Problem Set Two, tries often use up quite a lot of space. On top of that, they require a lot of pointer indirections, which can lead to a lot of cache misses and therefore to slower-than-expected performance. In 1989, Jun-Ichi Aoe developed the double-array trie, a data structure for representing tries as a pair of arrays in a way that supports relatively efficient insertions and deletions.

Why they're worth studying: The double-array trie is a fundamentally different presentation of the trie data structure that bears little resemblance to what we've seen used so far. If you're interested in exploring an existing data structure in an entirely new light, we recommend checking this one out!

The Galil-Seiferas Algorithm

On Problem Set Two, you saw that Knuth-Morris-Pratt algorithm drop out as a special case of Aho-Corasick string-matching automata. This algorithm requires O(n) preprocessing and then supports string matching in time O(m). Since KMP precomputes a modified matching automaton for the pattern string, its space usage is O(n). The Galil-Seiferas algorithm is an algorithm that supports string matching in time O(m + n), but remarkably uses only O(1) storage space.

Why it's worth studying: The Galil-Seiferas algorithm relies on a number of beautiful theorems about strings and its analysis, while a bit tricky, is quite beautiful. If you're interested in getting a much deeper feel for the inherent structure of string-matching, this would be a great place to start.

Ukkonen's Algorithm

Prior to the development of DC3 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, giving a preprocessing/runtime setup that, in a sense, generalizes KMP.

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!

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.

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 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

When discussing different classes of balanced trees, we introduced the B-tree as a data structure well-suited for on-disk databases (and, recently, for in-memory tree structures). 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 are used to 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.) that are used in real-world systems and a number of open-source implementations available. 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!

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.

A Caveat: Ropes are not as complicated as some of the other data structures on this list and many of the papers about ropes give incorrect information about the runtime of the various operations on them. 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.

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.

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

Multisplay Trees

If you're interested in tango trees, you may also want to look at *multisplay trees*, which were developed shortly after tango trees and have several slightly nicer theoretical 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 tricky and the motivation behind the data structure is not clear from the paper. If you choose to explore this data structure, you'll need to do a fair amount of work to develop an intuition explaining why the data structure is designed the way it is.

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.

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.

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.)

Shadow Heaps

Shadow heaps are a variation on binary heaps that combine a traditional binary heap with some extra slack space in which elements are stored in a somewhat arbitrary order. By allowing for a little bit of disorder, these heaps support merges more efficiently than a traditional binary heap – at least, in an amortized sense.

Why they're worth studying: When we first talked about meldable heaps in class, I mentioned that I hadn't actually seen any ways of merging binary heaps in sublinear time. It turns out there's been a lot of research into this area and, while the linear lower bound still exists, there's quite a lot of work done in trying to speed things up as much as possible. This would be a great place to get started!

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!

Deamortization

Let's suppose you happen to have an amortized-efficient data structure lying around and want to convert it into a worst-case efficient data structure. In certain cases, there are simple mechanical transformations that can be made to the data structure to achieve exactly this result. There are a number of techniques for accomplishing this goal, and a survey would make for a great final project.

Why it's worth studying: Much of modern data structures research focuses on techniques for transforming or combining data structures together in clever ways to add in new features or to remove existing deficiencies. If you'd like to get a feel for what this looks like in practice, consider checking this topic 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.

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

Cardinality Estimators

A cardinality estimator is a data structure that takes in a set of data and tries to estimate how many different elements are in that data set while using an incredibly small amount of space – often, O(log log *n*) bits. They're used all the time in databases and web servers to estimate which algorithms out of a suite of possibilities would be best on a particular data set.

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.

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.

Quotient Filters

Recently, a new data structure called the *quotient filter* was proposed as an alternative to Bloom filters (described above) They maintain many of the same sorts of guarantees as Bloom filters, but are much more cache-friendly and therefore work faster in practice and in memory-constrained environments.

Why they're worth studying: Quotient filters combine a number of different data-structural techniques together, such as applying hash functions in unusual settings and using linear probing as a subroutine. They'd be a great way to review the topics from this course. Plus, the original paper introducing them has a highly entertaining name. ☺

Building Hash Functions

In lecture, we explored a number of different hash functions: 2-universal hash functions, pairwise-independent hash functions, etc. While we mentioned that these types of hash functions exist, we didn't actually touch on how to build them. How exactly would you go about constructing hash functions like these? What makes them work? And what does the gap between theory and practice look like?

Why it's worth studying: Most hash functions rely on a combination of number-theoretic and group-theoretic techniques to ensure that they meet the requisite goals. If you have a strong math background, you might be interested in learning more about these techniques.

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.

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.

L_p Norm Sketches

When we first discussed the count[-min] sketch, we modeled it as storing a frequency vector representing the distribution of the underlying elements in sublinear space. Imagine that we were interested not in storing the individual elements of that frequency vector, but rather in storing something like the L_1-norm or L_2-norm of that vector. This would help us determine information about the distribution of the elements – a low L_2-norm would indicate that the elements are rather uniform, for example. By using techniques reminiscent of those from the count[-min] sketch, it's possible to solve this problems in sublinear space to a reasonable degree of accuracy.

Why they're worth studying: The work done on L_p-norm sketches combines techniques from a number of different fields of mathematics: random matrix projections, streaming algorithms, and information theory, to name a few. Additionally, L_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.

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.

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.

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

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.

Thorup's Linear-Time Single-Source Shortest Paths Algorithm

In 1999, Mikeel Thorup published an algorithm for solving the single-source shortest paths problem in linear time on graphs with integer weights, beating the runtime of the best known implementation of Dijkstra's algorithm. This algorithm has been used as a subroutine in a number of more advanced data structures and algorithms and relies on a clever and nontrivial data structure.

Why it's worth studying: Thorup's algorithm relies on a clever combination of a macro/micro decomposition (using existing priority queue data structures layered on top of a number of smaller customized data structures) and would be a great way to see many of the techniques from this class all come together.

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: Succinct binary search trees, succinct priority queues.

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

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.

Consider looking up: Dynamic strongly-connected-components, top trees.

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 and their variants (ZDDs, ROBDDS, etc.)

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, BST dynamic optimality.