ProgrammingAbstractions-Lecture25

Instructor (Julie Zelenski):Okay, we’re experiencing a little technical difficult here that’s going to be resolved very quickly because Jason went to get the magic piece of equipment, but in the meantime, I’ll just fake it with my big personality up here. Okay, so thing’s that are going on: Pathfinder is due this Friday. So how many people have been making good progress on Pathfinder? You have one of the algorithms already implemented and working? Anybody with one already done? Almost done? Okay, maybe a little bit of a slow start on this one, but – and then there’s two pieces that you do have to get ready coming in this Friday. Do remember that you can use a late day on it if you absolutely have to, but we don’t really recommend it, because it does kinda put you behind getting ready for finals. And our final is at the very end of finals week, so on Friday. So a week and something from today in the 12:15 slot. I don’t know what room we’ll be scheduled in, but you can guarantee it won’t be Terman, that’s all I can tell you for sure. And when we do know, we’ll let you know. Anything else administratively I need to tell you guys? Okay, Bishop.

Student:[Inaudible]

Instructor (Julie Zelenski):There is an extremely limited run optional finals. You can talk your way into one Thursday if you can convince us of the compelling need for said thing. So send an email to Jason. I know a couple of you told me at the beginning of the quarter you were in this bind about being able to make it to the Friday exam. And so we – against my better judgment, I allowed myself to be talked into that. Okay, all right, I’m gonna have to just do this kinda blind, because the piece I need is not here, so I’m going to actually just talk to you about what it says on my slides and then when Jason arrives we’ll plug it back in and see what’s going on. Oh, there’s Jason. There we go. So what I’m going to do today is actually something kinda fun. And so I’m gonna take a case study of a particular data structure design, and the one I’m looking at there is the lexicon. So how many of you actually ever decided to just open up the lexicon file, just to see what was in there at some point during the quarter. We used it a couple of times. Anybody want to tell us what they learned by opening up that file? Anything interesting?

Student: It’s a data structure called a DAWG.

Instructor (Julie Zelenski):It’s a data structure called a DAWG. You’re like, well that’s a pretty funny name for something. And was the code particularly well-[inaudible], easy to read, easy to understand? No. Who wrote that code? She should totally be fired. Yeah, so this what I wrote kinda a long time ago, actually, to solve this particular problem. And actually I would like to key through kinda how it is I came to discover the DAWG and why it is the DAWG was the choice that made sense for what the need pattern we had for the lexicon was. So let me just refresh what the lexicon does. It is a special case of kinda set or map. It is a container where you’re going stick things in, they’re going to be unique, right? There’s no need for any sort of duplicate. The keys are always strings. And in fact, they’re English words, so that’s going to be an important thing to remember, that they’re not just arbitrary sequences of letters, right? They have certain patterns to them and certain sizes that might actually be something we could capitalize on. When we know something about the domain, we're able to kinda special case or tune or data structure to solve that problem very well, in particular. It has no associated value, so it doesn't have dictionary definitions or synonyms or anything like that, right?

So it really just is exists or doesn't exist that we were interested in supporting. And it also has, in addition to the kind of being able to add words and check if a word is contained, it's also this prefixing operation that we had talked a little about in Boggle, and that came back on the midterm. Like, why was prefixing important, right? It was critical for pruning these dead-end paths, these searches on the Boggle board that otherwise would really have taken a lot of time and kinda led to nothing of any value. So what I'm gonna to talk you through is some of the obvious choices based on data structures we already know about, and see where they would go. So this is a thought experiment. It's not that I wrote all these to find out. Actually, I mostly did a lot of this on paper, trying to figure out what options we had and how they would play out and what kind of trade-offs I'd have to make to make them work. So the simplest case of any kind of collection is to think about whether vector or array can do the job for you. In this case, having a vector that's kept in sorted – alphabetically sorted – order seems like a pretty good, easy way to get this set up and running and how would it behave? So in order to do a contains (word) on this data structure, what algorithm are you going to use to do your searching? Binary search, right? It's in sorted order, right?

We've got to take advantage of that, right? If we do a linear search I'm looking for the word "mediocre," I'm going to be trucking through, you know, thousands of words before I find it. So definitely want to be using binary search. Equals-equals, comparing my strings, less than, greater than, dividing in half. And so it should run a logarithm time. Logarithm time, one of those greater performers that you learned in the PQ assignment, hopefully. It's just basically not measurable on today's computer. So even for entries of 100,000, 500,000, basically for free. How do we do contains (prefix)? Can we also do a fast prefix search given this data structure? I hope so right? Thinking about the same kind of binary search. In this case, though, looking at substring, right? So comparing I'm looking for a three-character substring, only looking at the first three characters, decide which way to go. Once I find something that matches in the first three characters, I'm done. So again, still using the sorted property to quickly narrow in on where it could be. When it's time to add a new word to this, if someone asked you to put in the word "abalone," I gotta use that binary search to find out where to go. So in logarithmic time I can tell you where the new word goes, but then to put it into position, there's gonna be some shuffling.

And that's where this operation starts to bog down, given the idea that you might be moving half or more of your items, on average, to make that space for that one to open up. It is going to require a linear time operation to get new words into the data structure. Okay, probably still, though, a reasonable approach, right? These are the operations that get done immense numbers of times in Boggle, right? You're doing a ton of contains (word)'s and contains (prefix)'s as you're exhaustively searching that big board. But the add's are done kinda once at the beginning. You build that big dictionary and then you subsequently use it. And so if it took a little bit more time to built that thing, sorta an n-squared operation to kinda load it up, well, maybe that would be okay. The other we want to look at is space usage, and that's – and some of this is a little bit of an interesting artifact of the time I was working on this. It turns out space was actually a limiting factor in what we had available to us, in terms of core memories of our machines and what we could take up for our dictionary. In this case, it has per entry the size of the string, which we don't exactly know the string it represented. It's an abstraction to us, but we can make the assumption of kinda a simplification, that it's probably about the length of the string times of the size of the character.

And there might be a little bit more housekeeping associated with it, but that's a pretty good estimate of what it's taking. So in fact, it's the string data itself that's being represented in this vector without much else overhead. So if we say words are about average of about eight characters, about eight bytes per word. So if we have 100,000 we expect to have 800,000 bytes invested in this sort of vector arrangement. Excess capacity, a couple other things that might tweak that a little bit, but that gives us at least a good starting point to think about how this compares to alternatives. Okay, so then if we said we know that sortedness – that binary search is the real advantage of keeping it in sorted order, but the contiguous nature of the array kinda bites us in terms of insert. If we reorganize into that binary search tree to give ourselves the dynamic flexibility of wiring in the left and right halves through pointers, as opposed to contiguous memory, then we know we can also get add to run in logarithmic time. So looking at these operations, contains (word) is a tree search that's using equals-equals, left-right, deciding which way to go, and also performs logarithmically if the tree is balanced. Contains (prefix), same deal.

Keep looking at the [inaudible], working our way down, knowing which way to go. And then add can also perform in logarithmic time because it does the same search, falling off the bottom and then inserting the new word there, or finding it along the way and not having any work to do. So we can get all our operations done in logarithmic time, which we know to be kinda fast enough to actually not be worth fighting any harder for. Where did we pay for this? Nothing really comes for free, right? Overhead, right? And overhead, in this case, memory is certainly one of the places where there's going to be a pretty big cost that we now have the string data for each entry, plus a left pointer and a right pointer. And if we're doing the work to keep it balanced, which we probably ought to do if we want to avoid the degenerate cases of it kinda getting totally out of whack and degenerating into linear time, we're going to have a couple more bits, you know, bytes thrown into that factor, too. And so if we're assuming that there's ten bytes of overhead, four bytes in each of these plus another two for the balance factor, then we've added ten bytes of overhead on top of the eight bytes for the word.

Getting a little bit hefty. Something to worry about. Also in terms of code complexity, which I didn't put a bullet up for, but it's also worth kinda keeping – weighing it in your mind also which is that building the fancier, more complicated data structure probably meant you spent more time debugging it, more time developing it, right? You will have – it will be harder for someone later to maintain and deal with because it's actually just more complicated code that somebody looking at a sorted vector isn't likely to actually – like, they want to add a new operation, let's say they add remove. You're able to take a word out of the lexicon. The lexicon doesn't currently support that. Somebody adding that on the sorted vector implementation probably won't find themselves too tripped up. Someone who has to do a remove out of a binary search tree, removing it, reattaching the subtrees and not destroying the balancing is a pretty complicated operation. So we've made – we've invested in this data structure that actually will keep on creating further development hassles for any modifications that are down the road for this data structure. So it is a big investment in your brainpower to keep it working.

So let's think about what a hash table can do for us. A hash table is the last thing that we looked at on Friday, kinda just moving away from this whole idea of sortedness and kinda moving it out of bound off to this idea of a hashing function. So if I have a hash table that has a certain number of buckets that uses a link list for chaining to handle the collision, then what I would have is my words just scattered around my table, allowing me to have quick access by hashing to the right bucket and then working my way down the bucket to C. So the contains (word) would do a standard hash table lookup. Take the word, you know, "mediate," run it through the hash function. It says oh, it's bucket three in this case. I go to bucket three. I walk down the link list. I find it or not. I can tell you whether the word "mediate," if it's in the table, had to be in bucket three. No questions asked. In that case, the expected time for that is going to be n over b, where n is the number of entries, b is the number of buckets. If we have buckets set to be in the rough order of the number of entries, then we're going to get constant access there.

Now we want to do contains (prefix). I want to know if there's any words in this table that begin with "med." Where do I look? If I run "med" through the hash function, do I expect that I would get the same number as other words that begin with "med," like "mediate" and "median" and "mediator?"

Student:

Yes.

Instructor (Julie Zelenski):You say yes. All right, let's take that to an extreme. If I expect that if a prefix in any longer words all have the same thing, then should it be the case, for example, I think about the simplest case of a prefix, "m." Should all the words that begin with "m" hash to the same place? If they did, we're in trouble, right? Now, if all prefixes did hash to the same place, then what I'm going to end up with is a really heavily clustered table where "a," "b," "c," you know, "z" – there's 26 buckets that have all the "z" words, all the "y" words, all the what not, and none of the other buckets would be used. All right, so in fact, actually, as part of the good behavior of a hash function, we're hopping that, actually, it does jumble things up. And then a word that looks like one but has another letter added on the end, we're hoping it goes somewhere totally different, that "median" and "medians" hopefully go to two totally disparate places. That if there were patterns where words that were substrings of each other all got hashed to the same location, we would end up with heavy clustering. Because there are a lot of words that repeat parts of the other words in the dictionary. So in fact, if I want to know if there's any words that begin with "med," I have no a priori information about where to look for them. They could be anywhere.

I could hash "med" and see if "med" itself was a word, right? But let's say it was something that itself was just the beginning of something. "St," "str" – there's a lot of words that begin with that, but "str" itself is not a word. So hashing to that bucket and looking for it would only tell me if there's exactly "str," and then occasionally there might be some "str" words also there. But other than that, we just gotta look. And where do we look? Everywhere. Everywhere, everywhere, everywhere. That in order to conclude there is no word that begins with the prefix we have, the only way to be confident of that is to either find something, or to have searched the entire table and not found it. And so it does require this completely linear, exhaustive look through the entire contents of the hash table to know something, for example, is not a prefix. You might find one quickly. So in the best case, you might be in the first couple of buckets, happen upon one. But no guarantees, overall. And certainly when it's not a prefix, it's going to require a complete look-see of everything. That's kinda a bummer. Adding a word, back to the fast behavior again. Hashing in the bucket, checking to see if it's already there, adding if needed [inaudible]. So it gets good times on these two, but then kinda really bottoms out when it comes to supporting that prefix search.

Its space usage is kinda roughly comparable to that of the binary search tree. We've got the string data itself, and we've got the four-byte pointer on the link list. There's also the four-byte bucket pointer, which is over in this other structure here, this array full of buckets. If that number, though, is roughly equal to the number of cells, then you can kinda just count it as each entry had this 12-byte cell, plus a four-byte bucket pointer somewhere that you can kinda think of as associated on a per entry basis to tell you it's about eight bytes of overhead on top of the string. So 16 bytes for each entry in the table.

Okay, that's what we have so far. We've got pretty good performance on contains (word) no matter what we choose. We can get either logarithmic or constant time with either of these three data structure. So perfectly acceptable performance there. We cannot get contains (prefix) to run well on a hash table. Just doesn't happen the way hashing works. And then add can be fast on the two complicated pointer-based data structures to the right, but can't be fast on the sorted vector. And then we see kinda some trade-offs here in terms of space versus time, that there's very little space overhead in the sorted vector case, but really blossoms into two times, almost three times, the storage needed.

In the VST and hash table case, the code also got a lot more complicated when we moved up to those data structures. So then I'll tell you, actually, that it turns out that the one type of performance that was interesting to us when I was exploring this, but in fact, more important at the time, was the memory usage. That the memory usage was really the big obstacle that we had. The Official Scrabble Players Dictionary II is actually the source for the word list that we're using. It has about 128,000 words, I think, is exactly the number it has. And the average length of those is eight characters. And the file itself is over a megabyte. So on disk, if you were just to look at the listing of all the words, it's a megabyte – a little over a megabyte there. If we were loading it into the sorted vector, we'd get something that was just roughly equal to that. It would take about a megabyte of space, plus a little bit of noise. The ones that double up that take it up to two, two and a half megabytes total. At the time we were working, and I know this is going to seem completely archaic to you, but we had these stone tablets that we worked on. And they typically had main memories of about a megabyte, maybe, like in an exciting case, four megabytes; four whole megabytes of RAM.

So it was not possible that we could dedicate one or two megabytes of your main memory to just your lexicon. It just wasn't – there just wasn't enough space to go around. So we're working in this very tight environment in terms of that. So a little bit more like, for example, today's world of embedded devices. If you're adding something for an iPod or a cell phone you have a lot less gargantuan memory. So your average desktop machine, having a gigabyte of RAM, now you're like, whatever, a megabyte here, ten megabytes there, free, all the memory you want. And so this seems kinda like a silly thing to focus on, but this was 1995, and it turns out, yeah, there was certainly more interest in keeping it tight to the memory. The Boggle lexicon that we have actually takes under a third of a megabyte. The memory that's used while it's working and searching and checking for words and prefixes is actually smaller than the on-disk form of the data we started with. So it actually does a kind of a bit of a compression as part of its strategy for storage, which is pretty interesting to think about. So I'll tell you, actually, the truth about how we first did Boggle, because it's kinda a little interesting historical perspective on it.

But Boggle was given by a colleague of mine for the first time, Todd Feldman. He was the guy who came up with the idea, which is a great idea. And he said, "Oh, this is a great assignment." And when he wrote it, he said, "Oh, I need a lexicon." So as part of kinda putting the assignment together, he was like, "Oh, we have to get some lexicons." So he looked around and he found a word list that had about 20,000 words. And I think we first found this word list and we said, "Oh, it's just impossible." It's going to take a megabyte or more and we just don't have that kind of space. So we need much smaller data files. So we had a data file that had about 20,000 words, which then takes about 200k. And he actually wrote it using a hash table. So given he wrote it as a hash table, he just didn't support contains (prefix). He just didn't put it in, and – because you can't do it efficiently. So it actually took a smaller amount of space and had a smaller word list and it wouldn't do contains (prefix). So then people wrote Boggle. And as you learned from that midterm question, you can write Boggle without contains (prefix). It spends a lot more time looking down these data ends, but it eventually bottoms out when it runs off the board and uses all the cubes.

And it actually, in some ways, almost made the program a little more satisfying to write in one odd way, which is, when you're – so it's the computer's turn, right, that uses the prefix pruning. So you would type in your words and it would be perfectly fine finding it, but it doesn't use the prefix part when it's doing the human's turn. You're typing in words; it's finding them. You're typing in the words; it's find them. Then you would go to do the computer's turn and you'd say, "Okay, go." And now it would take a really long time. It'd be hunting around the board. It would be working so hard your fan would be coming on. You'd feel like oh, my computer's doing something. And then it would show up with twice as many words as you found or ten times as many words as you found, whatever. And in fact, then you would say, "Well, at least it had to work hard to beat me." And then you felt like I wrote this program that causes my processor to get busy, right? I feel good. And in fact, actually, the word list was pretty small. So in fact, it didn't – it wasn't even actually as likely to beat you because it didn't know as many words. 125,000 words is an enormous number of words.

The average – I've heard different factors on this, but the average person's vocabulary, English vocabulary, is about 20,000 to 30,000 words. It tends to be about 10,000 higher if you've gone to college, so maybe 30,000 to 40,000. The fact that it knows 80,000 more words than the average college graduate means it knows a lot of obscure words, too. So in fact, not only does it very quickly find all those words and then produce these ridiculous words that you've never heard of, it just – it's almost like it's taunting you by doing it in a millisecond. And back then it took a while, so it was actually like okay, it beat me, but it had to work hard. But then – so then I took over Boggle. I taught x maybe a quarter or two later and I said, "I'm going to write a lexicon. I'm going to go up and make a little project for myself, because I don't have enough work to do." And my goal was to make one that would do prefix, but that also would be tight enough on memory that we could actually load a much bigger dictionary file, because I thought I'd actually make the game even more fun to play.

All right, so here's where I started. This is a little excerpt from the middle of the dictionary. "Stra," yeah, that's totally the thing you notice, right, when you look at this thing. You see "straddle," "straddler," "straddlers," "straddles," "straddling," and then these words that you can't believe you're allowed to do this, but apparently you can put "er" and "ier" and "ies' and "ing" on almost anything out there and it makes a valid word. So although you never really thought about the fact that "straightforward," "straightforwardly," "straightforwardness," are all there, there is a huge amount of repetition. And this is where I said we're going to use some information about the domain. These aren't just arbitrary strings. They're not just random sequences of letters that are picked out of the air, that they develop over time. They're word roots. They're suffixes. There's these prefixes that mean things that actually show you that looking at this little excerpt out of the middle, there's an awful lot of words that repeat portions of other words. And that may be part of what we can capitalize on. That instead of really recording "straightaway" and "straightaways," repeating those first 11 characters and adding an "s," there's some way that I can unify the data that I have here.

The same way when we try to find codes, you have the same passage of code repeated two or three times. We're always saying unify it, move it into a helper and call it in three places. It's like can we do the same thing with our data? Can we share the parts of the words that are in common and only distinguish where they are different? Take a look at this. This is a little portion of something called a trie. A trie is spelled t-r-i-e and it's actually from "retrieval," but sadly, it's still pronounced tri, just to confuse you. And it is a variant of a tree. In this case it's a tree that has 26 children coming out of any particular node, one for each letter of the alphabet. And so starting from the root node at the top, all the words that begin with “a” are actually down the A branch, and then there’s a B branch. There’d be a C branch and a D branch. And so the idea is that all the words, like if you think of the levels of the tree are actually like – these are all the zero if characters of the word.

And then that second level is all the characters that follow that zero if character. So here’s the “ab” words and the “ac” words and the “ax” words. Over here is the “st” words, but there’s not a “sx,” so there’s some places that actually aren’t filled out in this tree. And as you keep going down further, so that the depth you would trace to a tree would actually be tracing through the characters in sequence that make words. And so in this form, tracing out a path through this tree from the root down to a particular node tells you about prefixes. So there are words that begin with “a” and words that begin with “ac” and “ace,” and then there’s actually a little notation there that’s done by a visual of the bold and circle around it that says, “Oh, and the path that led here from the root is a word.” So “a” is a word and “ace” is a word and “aces” is a word. “Act” is a word and so is “acted” and “actor.” And in this case, the “act” part is totally shared, so everything that comes off of “act,” “acting,” “acted,” “actor,” “actors,” can all share that initial part. And for a word like “act,” it doesn’t seem that profound.

But you start thinking about words like “strawberry” or “straightforward” or “stratosphere” and realize that “stratospheric” and “strawberries” and “straightforwardly” can all leverage the other ten or 12 characters that were already invested in the root word, and that “straightjacket” and “straightforward” can even share “straight.” But there’s actually a lot of prefixes that can be unified across the space of the dictionary. So knowing that words have these patterns helps us to do this. So let’s build that. So the first form of this – and this, again, is a thought experiment. I did not actually write it this way because it would just be absolutely astronomically expensive. But it was a good way to kinda think about where I needed to go. I designed a new kinda trie node that had the letter and a little Boolean flag of is this the word, so whether it had the dark circle around it. And then it had a 26-member children array, so pointers to other nodes like this one. So totally recursive, staring from that root and then the idea is to use those 26 children as “a” being in the first slot and “z” in the last slot to make it really easy to kinda just trace letter by letter down to the levels of the tree.

So I’d have this one root node that pointed to the initial one, and then all the words that begin with “a” are off that first lot; all from “z” off the last slot. So when I want to know if a word is in the dictionary, what do I have to do in this data structure? How do I find it? I want to find the word “far.” Where do I look? Left? Right? Down? Which way? It would be great if you would tell me. Then I would understand what my data structure is.

Student:

[Inaudible]

Instructor (Julie Zelenski):Exactly. So at the very top I say okay, “f” is my first letter. Match my “f,” get me to the F node, and now it says recursively find “ar” from here. So it's very recursive. It'll be like find the first letter and then recursively find the substring from here, working down. So find the "a" out of the next node, find an "r" out of the next node, and then when you get to that node, you'll check this little "is word Boolean," and say okay, was that path I traced marked in the dictionary as leading to something that's known to be a word? The prefix looks exactly the same, except for that last check for the as word. So given that nodes exist in this, because further down there there's some words, I just trace out a sequence of letters "str," and as long as there's something there, then I know that away from there are subsequent children beneath it that lead to words. And adding a word does the same operation. If I need to add the word "strawberry," then I start looking for "s," "t," "r," and at some point, I find some places where the nodes don't exist. Then I start creating them from there on down. So it is tracing what exists and then adding new nodes off the bottom, and then marking the final one as yes, this is a word.

What's interesting about this is that all three of these operations don't actually make any reference to how many words are in the dictionary. They are all dependent on the length of the word you're searching for or adding into the dictionary. If there's ten words in there, if there's 1,000 words in there, if there's a million words in there. But in no way does the big O depend on how big or how loaded the data structure is. That's kinda wacky. The longest word, according to the Oxford English Dictionary? There you go. You're not going to make that on the Boggle board anytime soon because it's only got 16 cubes. Even Big Boggle can't really make that puppy.

But just so you know, 45 is not so many, right? And, in fact, actually, here's a little thought for you. Not only is it not dependent on num words, it actually has a little bit of an inverse relationship with num words, especially the add operation. That as the data structure becomes more loaded, there's typically less work to do in add. So if "strawberry" is in there, and you go to add "strawberries," then actually you already have "strawberr" to depend on. You just need to build off the thing. So the fact that the more words that are in there, the more likely that some of the nodes you already need are already in place, and then you can just kinda blow past them and just add your new nodes. Now, that's odd. You think of most data structures as really as clearly as they get heavier, it takes more work to install new things in it rather than less.

Student:

[Inaudible].

Instructor (Julie Zelenski):Well, the thing is, they're organized by letter, and that 26-number array. So in fact, I don't even have to look. So I just go straight to the slot. If I'm looking for strawberry, I'll look for "s" and I go straight to the "t" slot and then the "r" slot. So in fact, if the mode was already in place, I'm not searching for it. It really just is exactly in the "r" slot. That's why there's 26 of them.

Student:

In each array?

Instructor (Julie Zelenski):Each array is 26 and they're A through Z and if there's an empty slot we're using a null. So in this case, it's optimized to make it super fast. I don't even have to look through the letters to find the one that matches. There's no matching. It just goes straight to the "r" slot, "z" slot, the "s" slot. We're going to see this is going to be a consequence we can't tolerate, but it is, at least, the starting point for thinking about how to build a data structure. So the space usage is where this thing really, really – so this is an extreme example of a space-time trade-off, right? We've got this thing that optimizes beautifully for OAV length of the word. So that means typically eight operations on average are going to be needed to add or to search for something. The space we're using here is 106 bytes per nodes, so that's this 26-member, four-byte array, plus two bytes for the character and the Boolean that are up there. One hundred and six bytes is a lot of memory, and the tree has about a quarter of a million nodes. So given the 125,000 words that I said are in the input, if you build it into a tree, they take about 250,000 nodes. That's an interesting sort of number to think about, though. You have 125,000 words. They take 250,000 nodes.

That tells you, actually, that kinda on average, each word has about two unique nodes, that even words like "straightforward" and "straightforwardly," on average, are sharing enough of their prefix – common prefix parts – that there's very little unique data added by any particular word in there that averages across the whole thing. That's pretty neat to know. However, this is just not going to fly. So here I was telling you that my main memory had four megabytes, maybe, on a good day, and now I've actually built a data structure that's going to require six times that in terms of storage. Okay, we gotta squeeze that down. So the first thing we can do is to realize that those 26 – allocated a full 26-member array, of which I only plan on using some of the slots, is pretty wasteful. So instead of saying I'm committing to a 26 per thing, I'll say well, how about if I have an array that actually is tight to the number of children coming out of this node? So the very first node will have 26. There are words that start with every letter in the alphabet. But from there, it very quickly winnows down. You have the letter "x." You do not need the full 26 children coming off of "x." There's only a few letters that follow "x" in the English language that are going to need to be fleshed out.

You need the "xa," you need an "xe," but you don't need "xj," you don't need "xk," a whole bunch of things like that. So if I change the structure here to instead of being a 26-member array of node pointers, I make it to be a pointer to a set of nodes pointers, so this is a dynamic array that I'm keeping track of the number of children in a second field here that the first one will be allocated to 26, the second will be allocated to eight. This is going to change a little bit of our running times, because now we won't be able to go exactly to the "s" slot. We'll have to go look for it. So on each step down the tree, it'll be looking through that sized array at this slot to find the right match. But it adds a constant factor, basically, on top of OAV length of the word. So the space issues of this is – we're just not storing all those nulls. And there were a lot of nulls. That the node itself is not just ten bytes, and then there's some space in this dynamic array out there, which is the number of children times the four-byte pointer, that on average, a node has six children. Across the 250,000 nodes in the data structure that's being billed from this, so really 26 was way overkill. About 20 of them had nulls for most of the nodes. And many of them, for example, at the very bottom of the tree, have completely none. They're all nulls.

So you get to a word like "straightforwardly." There's nothing after that "y," so it has zero children coming out the back of that one. And so certainly having 26 null children pointing away from that was a waste that we can tighten up. So when we get it down to this we'll have 44 bytes – or 34 bytes for the per node. Still a quarter million nodes. Still eight and half megabytes. So we're not winning any prizes just yet. But we are making good progress. That got us down a factor of three or four right there. So I still have the same number of nodes. All I changed was the pointers to the subsequent nodes further down the tree. So I – the kind of letters and how the letters connect to each other – the connectivity is still pretty much the same. It's just that in any particular node, how many children does it have? I was storing a whole bunch of nulls and now I'm not storing those nulls. So the number of nodes is still exactly as it was before. I'm going to do one other little squishing thing on this data structure and then I'm going to flip gears for a second. So still using kinda my CS side of things, how can I squash things down? I'm going to use a technique that we used in the heap. So in the case of the binary heap that we implemented into the PQ, you'll remember that we drew pictures in the handout that showed a heap with oh, I have a parent with two children, which has four grandchildren and so on. But that we actually compressed that down into an array, and in the array there was this pattern of here is the root node. So we'll call this level zero. And then the next two nodes are level one. And the next four nodes are level two and so on. So there's one here, two here, there's four here, there's eight here. And we kinda flattened it down by generation, so looking at what was at the top and what was underneath it.

When we did that, we removed all the space for pointers. Although we drew those pictures in the heap as though there were pointers there, in fact, there were not. There were no pointers being stored in the heap in this version of the PQ. And so that gets us back to kind of array like storage requirements, but by flattening it down into this array, and still kinda using the tree conceptually to work our way through it, we're getting the advantages of that traversal that's based on height of tree, not length of vector. So if we do the same thing with the trie, it's a little bit more complicated because there were a couple things about heap that made this a little easier. Heap always had two children, so there were these rules about how the tree was filled in that meant you could always compute parent and child index using this kinda divide by two, multiple by two exchange operation. In this case, what we're going to have to do is just lay it down by generation, but we don't know how big a generation's gonna be, so we're actually gonna have to keep track of it manually. So the root node will have all A through Z. So this will be level one, which is A through Z here, and then level two is – there'll be a bunch of slots here, which is the A followed by some letter, and then there'd be the B's followed by some letter, and then there'd be the C's followed by some letter, and so on. And then further down is level three, which are the three-character words. So these are all the single-character paths, two-character paths, three-character paths, kinda flattened down. And so if I look at what I changed here, my first node says it has no letter that's not the root node, it's not a word, and it says it's children begin at index one. So the next location. So it's not a multiply by two and add one or something like that.

It’s okay, here’s where my children begin. They’re at slot one in the array. And then there are 26 of them. So that means that A, B, C, D are kinda in order there. If I look at A, “a” is a word, and it says its children begin at index 27 and there are 12 of them, so there are 12 characters out of the 26 that can follow an “a” in the sequence of a valid word. And then B’s children begin at index 39, which is 27 plus 12. So that’s where A’s children are sitting here at index 27 to 38 and then 39 to – I think B has eight or so children – and C and so on. So we flattened it all down by generation into an array. That means all the space for pointers has now been removed. One serious consequence of this, though, is that the pointers got us flexibility, that that insert and remove was dependent on the idea that pointers really being there, pointers buying us the ability to insert and delete without doing a shuffle. That now, once I have flattened it, I have suddenly bought back the same problems that arrays have, which is if I have the words here that have “ab” for abalone, and “ac” for act and I don’t happen to have any words with “ad.” And I try to add the word “add,” that in order to put “ad” in there, we’re going to have to move everything over and then there’s going to be a lot of shuffling and what not. So I’m starting to build a data structure that’s losing some flexibility but saving space. So I’m, in some sense, focusing my attention on how can I build a data structure that’s really fast to search, even if it was a little harder to build? Because it turns out the thing I most care about is making sure it can actually very, very efficiently look up words. Maybe building it once, being a little slow, isn’t so painful. So the space issues in this gets us down to just ten bytes per node. All the pointers are gone. So there was 24 bytes of pointers that we’ve just eliminated. And that means we’ve got two and a half megabyte memory footprint right now. So we’re kinda getting close to hitting the range for binary tree and hash tables. Now I’m going to go back to the domain again. So I worked on it kinda from the CS side, thinking about things I know about data structures and reorganizing arrays and heaps and pointers. I’m going to go back and look at that domain again and see if there’s something else I can kinda take advantage of. So this is a different cross-section of the dictionary showing the words “adapted,” “adaptor,” “adaptors,” “adapting,” “adaption,” adaptions,” “adapts.” So eight words that are all built off the “adapt” root with a bunch of suffixes.

I look sorta over a little bit further. I’ve got “desert,” “deserted,” “deserter,” “deserting,” “desertings,” “desertion,” “desertions,” “deserts.” Okay, and “detect” and “insert” and “perfect” and “invent” and “interrupt” that all are verbs for which you can apply the past tense in the gerund form and the “ion’s” that tack onto that root word to give you these suffixes. And each of these words that has shown up here has exactly the same set of suffixes that can be applied to it. So I did this thing about prefixes. I went out of my way to find that – so all these words could share that same prefix, “assert,” “asserter,” “assertings,” “asserting.” But is there some way that I could take this idea of “corrupt” and “assert” and “insert,” and then once I get down to that “t,” where they end, is there a way where they can also share their backend parts? That those “ing’s” and “ion’s,” can they be unified, as well? Well, why not? Why not? So here is the first version of what’s going to be the DAWG, the Directed Acyclic Word Graph. It takes the idea of unifying the prefixes and now applies it to the suffixes. That there are a lot of words that end in the same letters, as well as start in the same letters. Why not just apply the same kind of unification and sharing on the back end?

So here comes in “adapt” and “adopt” and “based” and “port” and out the back end is “ports” and “portion” and “porting” and “porter” and “adopter” and “adopted” and “basting” and “bastion,” all coming off that sharing all the endings, because they have the same words that are viable and the same connections that are between them. So the idea that all sharing the “er” and the “er” leading in to the “s.” And that same “s,” for example, being shared from “portion” and “baste” and “adopt” and “adopter,” all coming into the same ending of oh, these things that end in “s” are words. Here’s an “s” that is a word for them to share. So it is directed. So it is a graph. So once we remove this – the idea that there’s a single path from the root to any node, we’re no longer a tree structure. So we can get to “t” by going through “adapt” or “adopt” or “port,” and so that no longer fits the definition of what is a tree. It’s become a graph. We can get to these nodes from different places. The arcs go one way in this case. We can’t go back up the tree to find the words in reverse. There are no cycles. So acyclic, right, you can’t just wank around some portion and get this many bananas as you want. And they have been with certain nodes that are marked, in this case, the same way they were in the trie.

This path from the root node to here does represent a word in this situation. So building this is a little bit complicated. I’ll give you a little bit of the idea of the intuition for it, and I will not get too deep into it. But largely the way you do it is you build the trie, so that it doesn’t have the unification of the suffixes; it only unifies the prefixes. And then you work backwards on the suffixes. So in some sense you look at all the words that end, for example, in “s” in the dictionary. And you find all the ones that just end in one “s” that goes nowhere, and you say look, this “s” looks like every other “s” in this thing. All these words that are just plurals or verbs that can have an “s” tacked on. Let’s share that “s.” And then so you take however many s’s there were, 1,000 of them, and you say these are all the same “s.” And then what you do is you look at all the nodes that lead into that “s.” So you’re kinda traversing the graph in reverse, and you say well look, here’s a bunch of people who lead in on an “e.” And if that “e” is a word or not, there might be some group that has “e” that’s also a word, so like the word “apple” and “apples.” But sometimes the “e” is not, where that wasn’t a stopping place, let’s say, for “parties” it wasn’t. So all the ones that are a word you can unify on the “e” that was a word, and all the ones that aren’t are that way can unify on that.

So you just work your way backwards from there. So you say what letters led into that “e?” And so kinda from the end of the words backwards, you’re looking for all the people who can end in an “s,” and end in an “e,” and end in an “ies,” and unify them up, making sure that they have all the same outgoing connections. So for example, “running” has an “ing” at the end. So does “swimming,” but there’s “swimmingly” and there’s not “runningly.” So you have to be careful about unifying things that they would kinda have to have all identical properties from this point down to the bottom of the tree to be unifiable. But it’s a pretty neat process. It’s actually what’s called DFT subset minimization, so if you want to learn a little bit about it, that’s the word to look up. And so if I do that, and I build this as a DAWG, I’m still using the same structure definition here, but what I am doing is unifying suffixes as well as prefixes and that, actually, is gonna change the number of nodes that I need drastically. It goes down to 80,000 from my initial 250,000. And if you think about the number of words again, there were 125,000 words in the dictionary, the DAWG has just 80,000 words.

So it means, actually, once you start unifying the suffixes, it's like most words don't even have – or many of the words don't even have nodes of their own that are unique to them whatsoever. They just exist kinda as portions of other words that have all been joined together and compressed into one package. So 80,000 nodes total, about two-thirds of the number of words total. So we have ten nodes – ten-byte nodes and 80,000 of them. We are down to under a megabyte. So we've crossed that little interesting barrier of the data structure on disk. The data that was fed into it, those words, was over a megabyte and now, actually, we have created a representation that actually is compressed, that uses less space than the text file did in the first place. And that's really because it is finding all these ways in which words are like each other and sharing, making use of the existing structure in the DAWG to add new words without adding bulk. So once I've done this I have to say that I've made the data structure even less flexible. So to be clear, each of these steps has a cost somewhere else that made the code more complicated.

It also made it so that now that I have all this sharing back here, that when I'm adding a new word, so I go back to look at this picture here, that I go in to add a word and I say oh, actually, the word "portioning" is a word. Well, "adoptioning" is not. So that will necessitate if I go in to add the word "portioning," that I actually have to split it off of its existing unified branch, because if I just tacked on the "ing" on the end of "portioning" there, that it turns out that "adaptioning" and "adoptioning" would suddenly become words, too. So actually I have to be extra careful once I've got it unified this way that when I add new words that I don't actually accidentally add some additional words that I didn't plan on because these paths have been shared. So it requires a little bit more careful management. Again, the goal, though, here, in this case, was to try to build a data structure that was kinda freeze-dried. So I build it off the data structure – the input data file I had, that then would operate very, very efficiently, and I wasn't planning on making a lot of runtime changes to it.

I get this down to this and then the last thing I'm going to do to it is go back to kinda the CS side of it and see if I can squeeze it down kinda in terms of just data structure, what's being stored per node. So I have 80,000 of these. So typically, when you look at a structure, it's very rare that the idea if you change something from an int to a char, so it was a two-byte thing into a one-byte thing, that you're going to have any profound impact on the total size needed. But if you happen to know you're going to make a lot of these structures, tens of thousands, almost 100,000 of them, then it turns out every byte counts. If I can get this thing down from, in this case, ten bytes to eight bytes, or to six bytes or four bytes, then it will cut my memory by quite a factor. So what I did in the final version of it is I used a little technique called bit packing, which is to actually stash stuff in the absolute minimum possible space that I can fit it into, taking advantage of things like, for example, if there are only 26 letters in the alphabet, a character is actually a full eight-bit value, which can hold numbers from zero to 255. And I just don't need that much space. I'm not using the yen sign, I'm not using the parentheses, I'm not using the digits. So having, in some sense, the capacity to store all those distinct characters is actually a cost I'm paying on every character that I don't need to. So in fact, I squeezed it down by dividing up into the sub-bit level. This something that's talked about a little bit in Chapter 15 if you're at all curious. It's not what I consider core material for 106B, but I'm just sorta mentioning it to kinda complete the picture. Where five bits, given that a bit is either zero or one, that I have five of them connected together, then I have 25 different patterns that I can represent in that amount of capacity there, and that's 32 different patterns, which is enough for my characters, plus a little slack.

I use exactly one bit for is word, rather than a full Boolean. All I need to know is yes or no. I also changed the way I did knowing that you were the last child of a generation, rather than having A say, "there are 12 children here." I just said, go to index 27 and start reading, and one of them will tell you it's the last. So each of them says, "No, I'm not the last," "No, I'm not the last." And then when you get to the last one, it will be marked as "yes," and that's how you'll know – that's where A ends and the next one beings. And so if I can get it down in this way I've taken what was a ten-byte structure down to four. And it turns out there's often reasons why trying to get it down to a multiple of two is actually a critical barrier. That if it were five bytes, it's actually likely to be eight, just because of what's called alignment and padding restrictions. So if were at seven and I squeeze down to six, it might not make any change. And if I squeeze down to five, it still might not make any change. It might be, actually, artificially decided that in each of the structures is going to be eight behind the scenes. But getting it down to four actually tends to be another one of those critical barriers where okay, four actually will be smaller than something that was ten.

And so I take my 80,000 nodes, I get them to be four bytes each, and then I now have a data structure that's about a third of a megabyte in memory, as opposed to the over a megabyte on disk in the text form. And so that's how it does what it does. So if you're going to look at the code, you'll see that there is this complicated bit pack structure that it does look for things char byte using a recursive character-by-character strategy of start at the top, find the matching character, then go to the children index, and then work your way down the children to find the next matching character, and so on to find its way through the data structure. So [inaudible] is where the child index, or for example, the idea that A says my children start at 27 or B starts at 39, and so I just need a number there, but I'm using a slightly smaller than an integer, just to make them all fit very tightly.

So I'll tell you a couple of interesting things about it, actually, before I close this off, which is the lexicon.dat file actually just is an in-memory representation of what the DAWG data structure looks like. And so in fact, it's very easy to take the whole data structure. It doesn't have any pointers in it. Pointers, actually, tend to be a little bit – make data structures harder to rebuild and to dehydrate to disk because they have to kinda be wired up in memory. You don't know what memory you're going to get from new and what the addresses are. This one, actually, doesn't have any pointers. Everything is actually kinda self-contained. It's just one big array of these blocks. And all the information is relative in there. It says go to the index 27, index 45. So you can take that whole block and just write it to disk, 300 bytes worth, read it back in 300 bytes worth and it kinda is rehydrated without any extra effort. So that’s actually what the lexicon.dat file is, it’s just a dehydrated form of the DAWG having been build in memory. That makes it very, very fast to load. At that point, though, it’s super inflexible. If you want to add a word to it, the way the structure is build, you’d have to kinda add nodes and make sure it wasn’t glomming on to some existing words that were nearby, and so it would be a lot of work to actually edit the DAWG in place. In fact, when you ask it to add a word, it actually doesn’t edit that one at all. You say I’d like to put a new word in. It says I’m just – this one’s so perfect the way it is. I’ve built this one, I love this one, I’m very happy with it. If you want to put some other words into it, it just keeps a second data structure on the side, an auxiliary one over here, where it sticks the additional words you ask it to put in.

I have to say that seems like a very obvious thing to do, but in fact, I didn’t even think of that for years. We had the lexicon. I said you can’t use the lexicon to make new word lists or edit them. You can only load this one, beautiful, perfect one. You can’t do anything else. And at some point people kept saying, “I wish I could put some words in the lexicon. I wish I could use it for the word list to keep track of the human and computer players.” I’m like, “Well, you just can’t do it. You don’t understand. It’s very complicated. You just don’t understand my pain.” Whatever. Whatever. I had excuses. And then, really, after some number of years of just being a baby about it, I’m like oh, of course, why don’t I just keep another structure off to the side? It’s an abstraction. It’s a word list. You don’t need to know what I do. The fact that I keep a DAWG for some of the words and just an ordinary set for some of the other words is really completely irrelevant. And as long as when you ask for words I look in both and I tell you, “Yes, it’s there,” why do you need to know where I keep it? Okay, so it took me a while to buy abstraction, even though I teach it all the time. It just wasn’t totally making sense.

So a neat little thing, just kinda keeping both behind. And so it turns out that I learned about this, actually, from a paper that I read at the – in the early ‘90s about somebody building a Scrabble player, and they wanted a dictionary that supported kinda finding plays on the board and being able to extend words. And so knowing things about root words and how they extend and prefixes and suffixes was kinda the original motivation. And this kind of data structure is actually really great for any kind of word playing. So you have anagrams that you want to rearrange or words that you want to extend past suffix and prefix that this kind of data structure is really optimized for doing that kind of traversal and kinda leveraging the existing patterns in words to save space while doing so. So it was a really fun little project to do. It was kinda neat to think about. Even though today, if you needed a lexicon, you could just use a set. It’s fast enough. It takes a ton of memory, but you know what, memory’s cheap.

So this is not the kind of thing you would need to do today except when you were in some kinda embedded, low memory, very, very tight processor environment, but it still makes for kinda interesting sort of artifact to think about well, how you get there and what kind of things you can use. So Wednesday we will talk about some big picture design, some kinda wrap stuff, and I will see you then.

[End of Audio]

Duration: 52 minutes