CMSC 420: Lecture 17
Tries and Digital Search Trees

Strings and Digital Data: In earlier lectures, we studied binary search trees, which store keys from an ordered domain. Each node stores a splitter value, and we decide whether to visit the left or right subtree based on a comparison with the splitter. Many times, data is presented in digital form, that is, as a sequence of binary bits. These may be organized into groups, for example as characters in a string. When data is presented in this form, an alternative is to design trees that branch in a radix-based manner on these digital values. This can be advantageous with respect to the data structure's speed and the simplicity of performing update operations. Generically, we refer to these structures as *digital search trees*. In this lecture, we will investigate a few variations on this idea.

Tries: The trie (pronounced "try") and its variations are widely used for storing string data sets. Tries were introduced by René de la Briandais in 1959, and the term "trie" was later coined by Edward Fredkin, derived from the middle syllable of the work "retrieval." (It is said that Fredkin pronounced it the same as "tree," but it was changed to "try" to avoid confusion.)

Let us assume that characters are drawn from an alphabet Σ of size k. In its simplest form, each internal node of a trie has k children, one for each letter of the alphabet. We cast each character to an integer in the range $\{0, \ldots, k-1\}$, which we can use as an index to an array to access the next child. This way, we can search for a word of length m by visiting m nodes of our tree. Let us also assume Java's convention that we index the characters of a string starting with 0 as the first (leftmost) character of the string. Given a string `str`, we will refer to its leftmost character as `str[0]`, and in Java this would be `str.charAt(0)`.

An example is shown in Fig. 1, where we store a set of strings over the alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ where $\texttt{a} = 0$, $\texttt{b} = 1$, and $\texttt{c} = 2$.

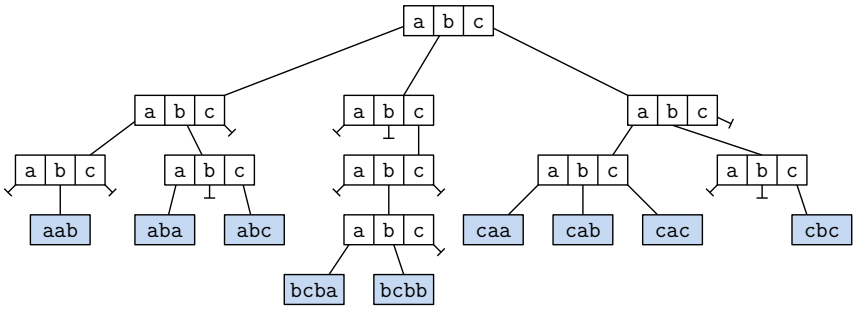

Fig. 1: A trie containing the set $\{\texttt{aab}, \texttt{aba}, \texttt{abc}, \texttt{bcba}, \texttt{bcbb}, \texttt{caa}, \texttt{cab}, \texttt{cac}, \texttt{cbc}\}$.

We would like each distinct search path to terminate at a different leaf node. This is not generally possible when one string is a prefix of another. A simple remedy is to add a special *terminal character* to the end of each string. (Later, we will use `$` as our terminal character, but for now we will simply avoid storing keys where one is a prefix of another.)

The drawing shown in Fig. 1 is true to the trie's implementation, but this form of drawing is rather verbose. When drawing tries and other digital search trees, we will adopt the manner shown below, where we label edges with the character. But remember that this is not a different representation or a different data structure, it is just a different way of drawing the tree.

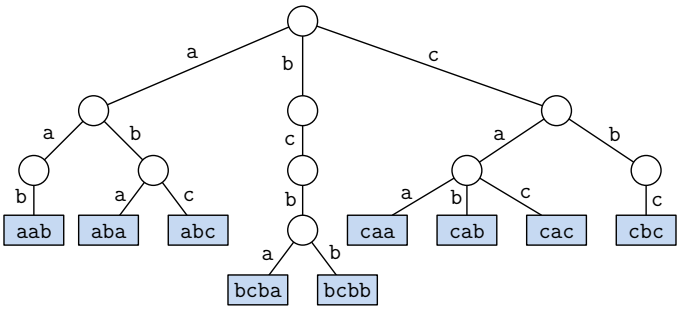

Fig. 2: Alternative method of drawing a trie.

Analysis: It is easy to see that we can search for a string in time proportional to the number of characters in the string. This can be a significant savings over binary search trees. For example, suppose that you wanted to store a dictionary of English words. A large dictionary can have around 500,000 entries, and $\log_2 500,000 \approx 19$, but the length of the average English word is well less than 10.

Space is an issue, however. In the worst case, the number of nodes in the data structure can be as high as the total number of characters in all the strings. Observe, for example, that if we have one key containing 100 characters, we must devote a search path for this string, which will require 100 nodes and $100k$ total space, where $k = |\Sigma|$. Clearly, this is an absurd situation, but it points to the main disadvantage of tries, namely their relatively high space requirements. (By the way, if you wanted to perform exact-match searches, hashing would be both more time and space efficient.)

de la Briandais Trees: One idea for saving space is to convert the k-order trie into a form that is similar to the first-child/next-sibling representation that we introduced for multiway trees. A node in this structure stores a single character. If the next character of the string matches, we descend to the next lower level and advance to the next character of the search string. Otherwise, we visit the next node at this same level. Eventually, we will either reach the desired leaf node or else we will run off the end of some level. If the latter happens, then we report that the key is not found. These are called *de la Briandais trees* (see Fig. 3).

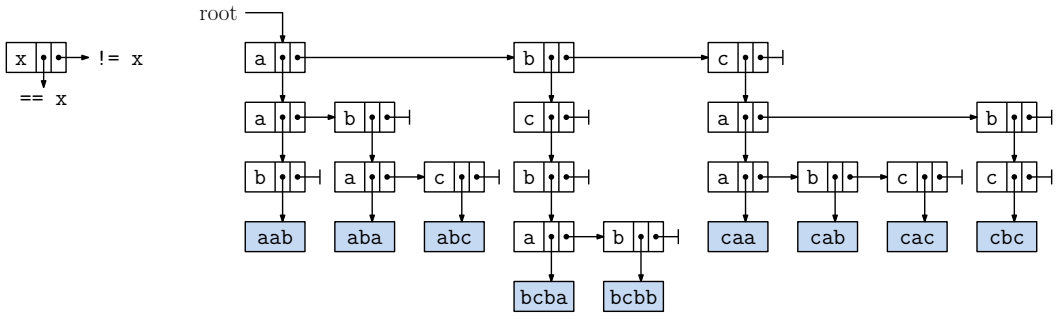

Fig. 3: A de la Briandais tree containing the same strings as in Fig. 1.

The de la Briandais tree trades off a factor of k in the space for a factor of k in the search time. While the number of nodes can be as large as the total number of characters in all the strings, the size of each node is just a constant, independent of k. In contrast, the search time can be as high as $O(k)$ per level of the tree, in contrast to $O(1)$ per level for the traditional

trie.

Patricia Tries: One issue that arises with tries and digital search trees is that we may have long paths where the contents of two strings are the same. Consider a very degenerate situation where we have a small number of very long keys. F, that we have the keys "`dysfunctional`" and "`dysfunctioning`" (see Fig. 4). The standard trie structure would have a long sequence where there is no branching. This is very wasteful considering that each of these nodes requires $O(k)$ space, where k is the size of our dictionary.

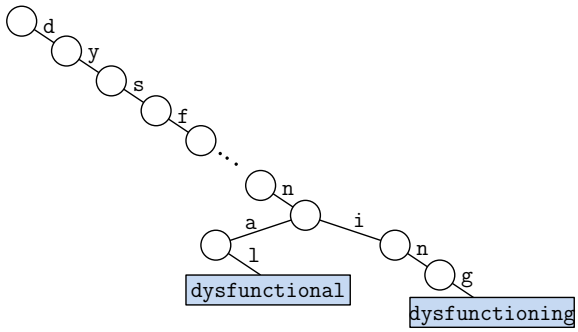

Fig. 4: Space wastage due to long degenerate paths in traditional tries storing "`dysfunctional`" and "`dysfunctioning`".

When keys are closely clustered (in the sense that they share lengthy common prefexes), digital search trees can suffer from this phenomenon. The solution is to perform *path compression*, which succinctly encodes long degenerate paths. This is the idea behind a trie variant called a *patricia trie*. (The word 'patricia' is an acronym for *Practical Algorithm To Retrieve Information Coded In Alphanumeric*.) This concept was introduced in the late 1960's by Donald R. Morrison, and was independently discovered by Gernot Gwehenberger.

A patricia trie uses the same node structure as the standard trie, but in addition each node contains an *index field*. This field is the index of the *discriminating character*, which is used to determine how to branch at this node. This index field value increases as we descend the tree. (A traditional trie can be viewed as a special case where this index increases by one with each level.) In the patricia trie, the index field is set to the next index such where there is a nontrivial split (that is, the next level where the node has two or more children). Note that once the search path has uniquely determined the string of the set, we just store a link directly to the leaf node. An example is shown in Fig. 5. Observe we start by testing the 0th character. All the strings that start with "`e`" are on the left branch from the root node. Since they all share the next symbol in common, we continue the search with the character at index 2.

As we did with standard tries, there is a more intuitive way of drawing patricia tries. Rather than listing the index of the discriminating character using for branching, we instead list the entire substring from the (see Fig. 6). This is convenient because we can read the substrings from the drawing without having to refer back to the original strings. As in the previous case, this is not a different representation or a different data structure, just a different drawing.

Analysis: The patricia trie is superior to the standard trie in terms of both worst-case space and worst-case query time. For the space, observe that because the tree splits at least two ways with each node, and easy induction argument shows that the total number of nodes is

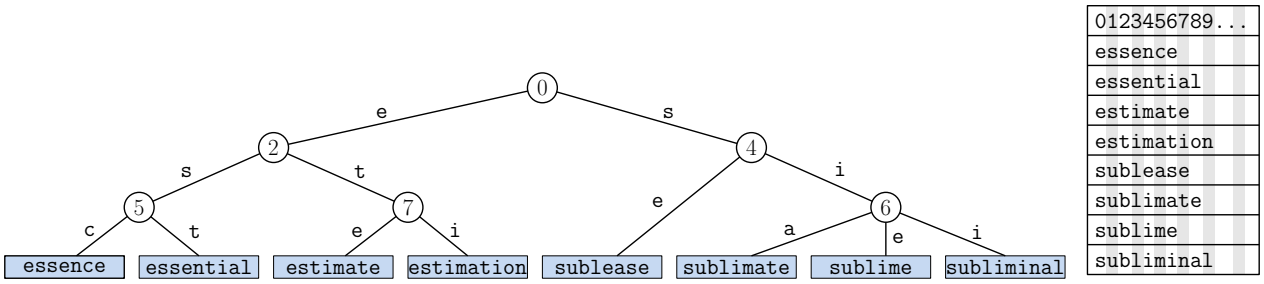

Fig. 5: A patricia trie for a set of strings. Each node is labeled with the index of the character and each edge is labeled with the matching character for this branch.

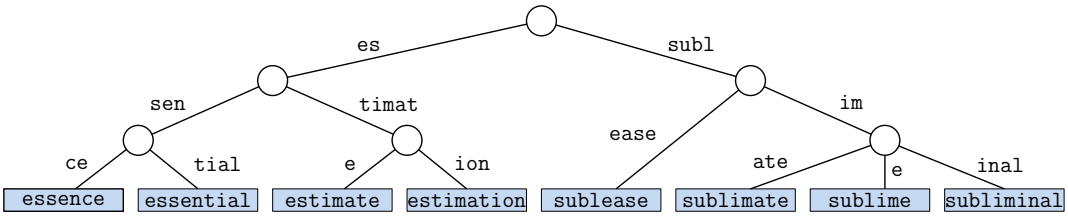

Fig. 6: Alternative method of drawing a patricia tree.

proportional to the number of strings (not the total number of characters in these strings, which was the case for the standard trie). As with the standard trie, the worst-case search time is equal to the length of the query string, but note that it may generally be much smaller, since we only query character positions that are relevant to distinguishing between two keys of the set.

Suffix trees: So far, we have been considering data structures for storing a set of strings. Another common application involves storing a single, very long string, for which we want to perform substring queries. For example, "Given the string "`abracadabra`", how often does the substring "ab" occur?" Such queries are useful in applications such as genomics, where queries are applied to a genome sequence.

Consider a string $S =$ "$a_0 a_1 \ldots a_{n-1}$\$", which we call the *text*. For $0 \leq i \leq n$, define the *ith suffix* to be the substring $S_i =$ "$a_i a_{i+1} \ldots a_{n-1}$\$". We have intentionally placed a special terminator character "\$" at the end of the string so that every suffix is distinct from any other substring appearing in S.

For each position i, $0 \leq i \leq n$, there is a minimum length substring starting at index i that uniquely identifies S_i. For example, in the string "`yabbadabbadoo$`", there are two suffixes (S_1 and S_6) that start with "`abbad`", and so this substring does not uniquely identify a suffix. But there is only one suffix (namely S_6) that starts with "`abbado`", and therefore this substring is minimum length unique identifier of a suffix.

To make this more formal, for $0 \leq i \leq n$, define the ith *substring identifier*, denoted id_i, to be the shortest substring starting at index i that is unique among all substrings in the text string. for position i. Note that because the end of string character is unique, every position has a unique substring identifier. An example is shown in the following figure. The set of all substring identifiers for this string are shown on the left side of Fig. 7.

A *suffix tree* for a text S is a patricia trie in which we store each of the $n + 1$ substring identifiers for the string S. We illustrate this on the right side of Fig. 7. We have adopted

the convention from Fig. 6 for drawing the patricia trie. Each leaf of the tree is associated with the suffix S_i it identifies, and it is labeled with the associated index i.

Index	Substring ID	Index	Substring ID
0	y	7	bbado
1	abbada	8	bado
2	bbada	9	ado
3	bada	10	do
4	ada	11	oo
5	da	12	o$
6	abbado	13	$

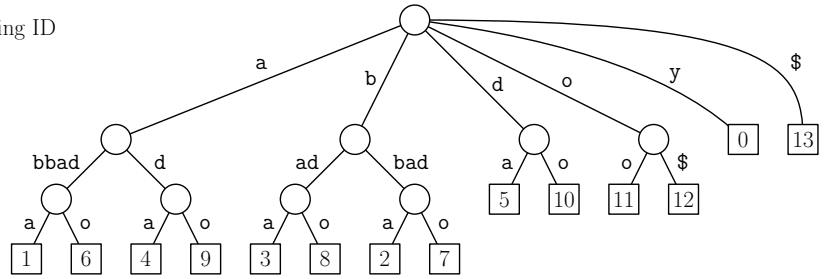

Fig. 7: A suffix tree for the string "yabbadabbadoo$". (Labels are placed on the edges, but this can be implemented using a standard trie or patricia trie.)

Efficient Construction: Generally, the size and construction time of a patricia trie are both proportional to the total length of the strings being inserted. If you are given a text S of length n, there are n suffixes of lengths $0, 1, 2, \ldots, n$. Thus, the total number of characters over all suffixes is $\sum_{i=1}^{n} i = O(n^2)$. This would suggest that the suffix tree has size $O(n^2)$ and takes $O(n^2)$ time to compute. While we will not present the proof, it can be shown that, due to the the special structure of the suffix tree, it has $O(n)$ size, and it can be built in $O(n)$ time.[1]

By the way, suffix trees are not usually the method of choice when storing suffix information. There is a related data structure, called a *suffix array*, which encodes the same information more succinctly (by a constant factor), and is the method that is usually used in practice when answering the same sorts of queries.

Suffix-Tree Queries: As an example of answering a query, suppose that we want to know how many times the substring "abb" occurs within the text string S. To do this we search for the string "abb" in the suffix tree. If we fall out of the tree, then it does not occur. Otherwise the search ends at some node u. The number of leaves descended from u is equal to the number of times "abb" occurs within S.

In our example from Fig. 7, the search ends on the leftmost path in the tree, midway along the edge labeled "bbad". Since the subtree rooted here has two nodes, the answer to the query is 2. By traversing the entire subtree, we can report that the two instances start at indices 1 and 6. (Recall that we index the string starting with 0.)

Geometric Digital Search Trees: Before leaving the topic of digital search trees, we should mention an interesting connection to geometric data structures. In an earlier lecture we discussed point quadtree and point kd-trees as two ways of storing geometric point sets. We can extend the idea of digital search trees to the geometric setting. We will do this in a 2-dimensional setting, but the generalization to arbitrary dimensions is straightforward.

Our approach will be to define a transformation that maps a 2-dimensional point (x, y) into a string, in a manner that preserves geometric structure. To do this, let us first assume that we have applied a scaling transformation so our point coordinates lie within the interval $[0, 1)$.

[1] Recall that the size of a patricia tree is proportional to the number of strings, irrespective of their sizes. This directly bounds the suffix tree's size. The trick for constructing the tree efficiently is to start with the last suffix S_n = "$" and work backwards to the first suffix, using the partially built tree to assist in the construction.

(For example, we can add a sufficiently large number that our coordinates are positive, and then we can scale by dividing by the largest possible coordinate value.) By doing this, we may assume that each point (x, y) satisfies $0 \leq x, y < 1$.

Our next step is to represent coordinate as a binary fraction. For example, the decimal point $(x, y) = (0.356, 0.753)$ can be represented in binary form as $(0.01011\ldots, 0.11000\ldots)$. We can treat the binary strings $x = 01011\ldots$ and $y = 11000\ldots$ as strings over the 2-symbol alphabet $\Sigma = \{0, 1\}$.

But, how do we convert two coordinates into a single digital string? Recall that in kd-trees, we alternated splitting on x and then y. This suggests that we can map two coordinates x and y into a single binary string by *alternating* their binary bits. More formally, if $x = 0.a_1a_2a_3\ldots$ and $y = 0.b_1b_2b_3\ldots$ in binary, we create the binary string $a_1b_1a_2b_2a_3b_3\ldots$. In our previous example, given our point $(x, y) = (0.356, 0.753)$, we would interleave the bits of their binary fractions to obtain the binary string "`0111001010...`". We we refer to this as the *bit interleaving transformation*. Let us apply this transformation to every point in our 2-dimensional point data set. We can then store the resulting binary strings in a digital search tree, like a patricia trie.

Wow! Is this crazy transformation from geometric data to digital data meaningful in any way? It turns out that not only is meaningful, it actually corresponds to one of the most fundamental geometric geometric data structures, called a *PR kd-tree*. In a 2-dimensional PR kd-tree, we assume that the data lies within a square. Let's assume that this square is our unit square $[0, 1)^2$. At the root, we split vertically through the midpoint of this square, creating two children (West and East). Each child is associated with a rectangular cell. For each child, if its cell contains two or more points, we split the cell horizontally through its midpoint into two cells (South and North). We repeat the process, splitting alternately between x and y, always splitting through the midpoint of the current cell, until the cell has either zero or one points. An example of the resulting subdivision of space is shown in Fig. 8(a).

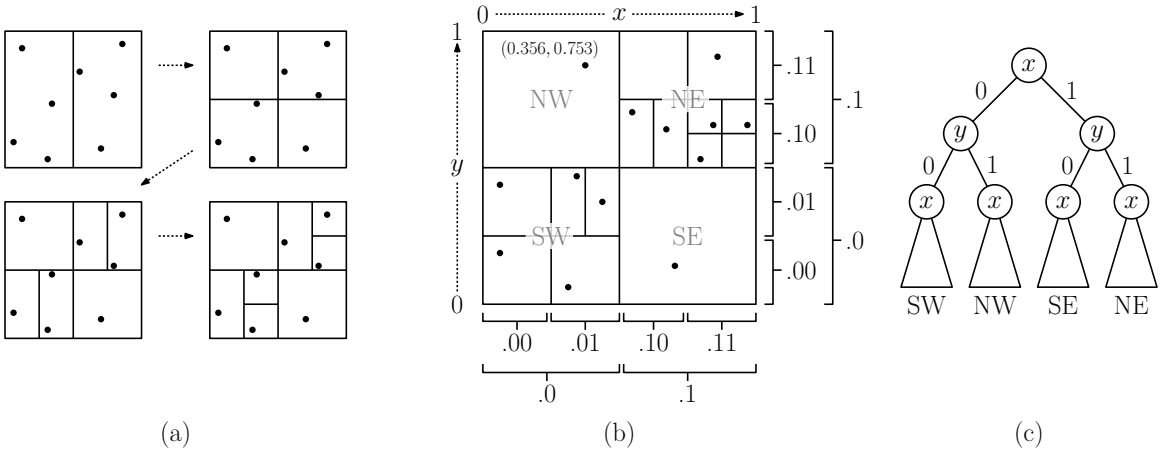

Fig. 8: The PR kd-tree as a digital search tree using bit interleaving.

Observe that after two levels of splitting in our PR kd-tree, we have subdivided the original square into four quadrants, each of half the side length of the original square. We can label these SW, NW, SE, and NE (see Fig. 8(b)). Let's consider what can be said about the points lying in any one of these quadrants. For concreteness let's consider the point with coordinates $(0.356, 0.753)$ in the NW quadrant (see Fig. 8(b)). Since it lies in this quadrant

we know that the leading bits of its x-coordinate is 0 (smaller than half) and the leading bit of its y-coordinate is 1 (larger than half). Therefore, its digital code starts with "`01...`". Observe that our digital tree will put this point and all points in the NW quadrant into the 01 grandchild subtree of the root. It is easy to verify that each of the four grandchildren of the root correspond to each of the four quadrants. Furthermore, as we descend the tree, with each two levels, the digital search tree partitions the points into subtrees in exactly the same manner as the PR kd-tree does.

In summary, the PR kd-tree data structure, is equivalent to a digital search tree for the points after applying the bit-interleaving transformation! While we think of a patricia tree as a data structure for storing strings, it turns out that this data structure can also be interpreted as geometric data structure, and it can be used for answering most of the same queries (such as orthogonal range and nearest neighbor queries) that the kd-tree can answer.