Lecture 25: Tries and Digital Search Trees

Reading: Section 5.3 in Samet's notes. (The material on suffix trees is not covered there.)

Strings and Digital Data: Earlier this semester we studied data structures for storing and retrieving data from an ordered domain through the use of binary search trees, and related data structures such as skip lists. Since these data structures can store any type of sorted data, they can certainly be used for storing strings. However, this is not always the most efficient way to access and retrieve string data. One reason for this is that unlike floating point or integer values, which can be compared by performing a single machine-level operation (basically subtracting two numbers and comparing the sign bit of the result) strings are compared lexicographically, that is, character by character. Strings also possess additional structure that simple numeric keys do not have. It would be nice to have a data structure which takes better advantage of the structural properties of strings.

Character strings arise in a number of important applications. These include language dictionaries, computational linguistics, keyword retrieval systems for text databases and web search, and computational biology and genetics (where the strings may be strands of DNA encoded as sequences over the alphabet $\{C, G, T, A\}$).

Tries: As mentioned above, our goal in designing a search structure for strings is to avoid having to look at every character of the query string at every node of the data structure. The basic idea common to all string-based search structures is the notion of visiting the characters of the search string from left to right as we descend the search structure. The most straightforward implementation of this concept is a *trie*. The name is derived from the middle syllable of "retrieval", but is pronounced the same as "try". Each internal node of a trie is k-way rooted tree, which may be implemented as an array whose length is equal to the number of characters k in the alphabet. It is common to assume that the alphabet includes a special character, indicated by '.' in our figures, which represents a special end of string character. (In languages such as C++ and Java this would normally just be the null character.) Thus each path starting at the root is associated with a sequence of characters. We store each string along the associated path. The last pointer of the sequence (associated with the end of string character) points to a leaf node which contains the complete data associated with this string. An example is shown in the figure below.

The obvious disadvantage of this straightforward trie implementation is the amount of space it uses. Many of the entries in each of the arrays is empty. In most languages the distribution of characters is not uniform, and certain character sequences are highly unlikely. There are a number of ways to improve upon this implementation.

For example, rather than allocating nodes using the system storage allocation (e.g., `new`) we store nodes in a 2-dimensional array. The number of columns equals the size of the alphabet, and the number of rows equals the total number of nodes. The entry $T[i, j]$ contains the index of the j-th pointer in node i. If there are m nodes in the trie, then $\lceil \lg m \rceil$ bits suffice to represent each index, which is much fewer than the number of bits needed for a pointer.

de la Brandais trees: Another idea for saving space is, rather than storing each node as an array whose size equals the alphabet size, we only store the nonnull entries in a linked list. Each entry of the list contains a character, a child link, and a link to the next entry in the linked list for the node. Note that this is essentially just a first-child, next-sibling representation of the trie structure. These are called *de la Brandais* trees. An example is shown in the figure below.

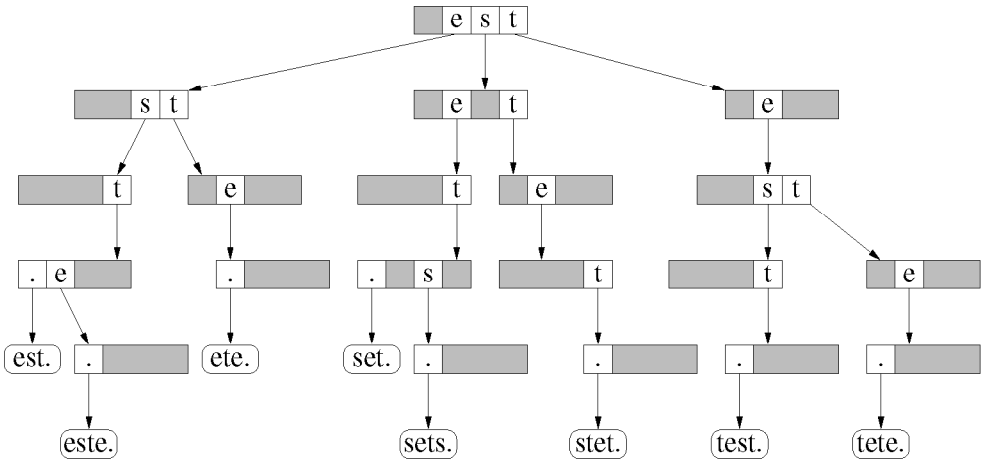

Figure 75: A trie containing the strings: est, este, ete, set, sets, stet, test and tete. Only the nonnull entries are shown.

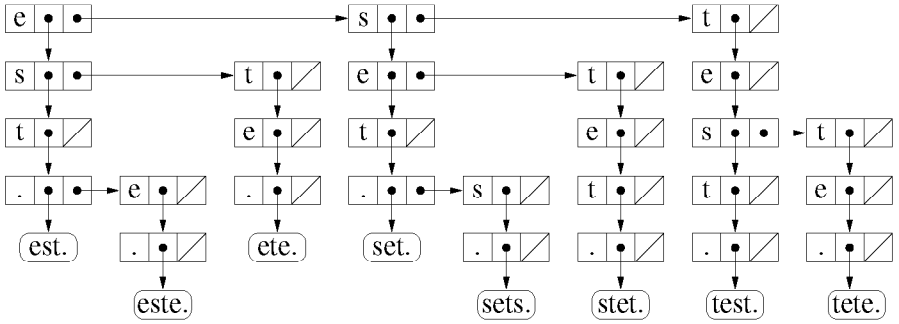

Figure 76: A de la Brandais tree containing the strings: est, este, ete, set, sets, stet, test and tete.

Although de la Brandais trees have the nice feature that they only use twice as much pointer space as there are characters in the strings, the search time at each level is potentially linear in the number of characters in the alphabet. A hybrid method would be to use regular trie nodes when the branching factor is high and then convert to de la Brandais trees when the branching factor is low.

Patricia Tries: In many applications of strings there can be long sequences of repeated substrings. As an extreme example, suppose that you are creating a trie with the words "demystificational" and "demystifications" but no other words that contain the prefix "demys". In order to stores these words in a trie, we would need to create 10 trie nodes for the common substring "tification", with each node having a branching factor of just one each. To avoid this, we would like the tree to inform us that after reading the common prefix "demys" we should skip over the next 10 characters, and check whether the 11th is either 'a' or 's'. This is the idea behind *patricia tries*. (The word 'patricia' is an acronym for *Practical Algorithm To Retrieve Information Coded In Alphanumeric*.)

A patricia trie uses the same node structure as the standard trie, but in addition each node contains an *index field*, indicating which character is to be checked at this node. This index field value increases as we descend the tree, thus we still process strings from left to right. IIowever, we may skip over any characters that provide no discriminating power between keys. An example is shown below. Note that once only one matching word remains, we can proceed immediately to a leaf node.

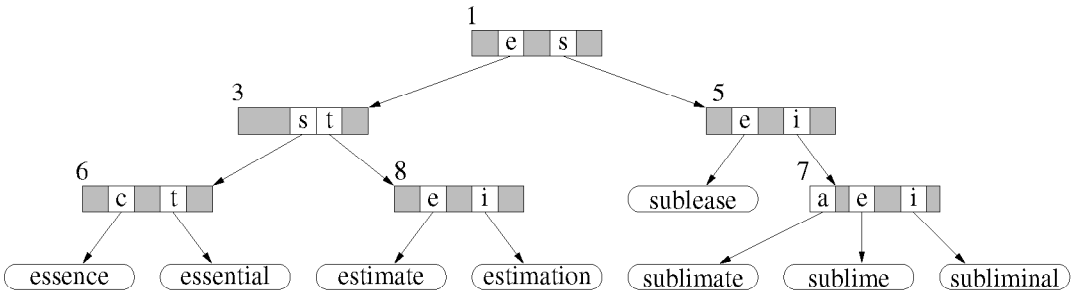

Figure 77: A patricia trie for the strings: essence, essential, estimate, estimation, sublease, sublime, subliminal.

Observe that because we skip over characters in a patricia trie, it is generally necessary to *verify* the correctness of the final result. For example, if we had attempted to search for the word "survive" then we would match 's' at position 1, 'i' at position 5, and 'e' at position 7, and so we arrive at the leaf node for "sublime". This means that "sublime" is the only possible match, but it does not necessarily match this word. Hence the need for verification.

Suffix trees: In some applications of string pattern matching we want to perform a number of queries about a single long string. For example, this string might be a single DNA strand. We would like to store this string in a data structure so that we are able to perform queries on this string. For example, we might want to know how many occurrences there are of some given substring within this long string.

One interesting application of tries and patricia tries is for this purpose. Consider a string $s =$ "$a_1 a_2 \ldots a_n$.". We assume that the $(n+1)$-st character is the unique string termination character. Such a string implicitly defines $n+1$ suffixes. The i-th suffix is the string "$a_i a_{i+1} \ldots a_n$.". For each position i there is a minimum length substring starting at position i which uniquely identifies this substring. For example, consider the string "yabbadabbadoo".

Position	Substring identifier
1	y
2	abbada
3	bbada
4	bada
5	ada
6	da
7	abbado
8	bbado
9	bado
10	ado
11	do
12	oo
13	o.
14	.

Figure 78: Substring identifiers for the string "yabbadabbadoo.".

The substring "y" uniquely identifies the first position of the string. However the second position is not uniquely identified by "a" or "ab" or even "abbad", since all of these substrings occur at least twice in the string. However, "abbada" uniquely identifies the second position, because this substring occurs only once in the entire string. The *substring identifier* for position i is the minimum length substring starting at position i of the string which occurs uniquely in s. 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.

A *suffix tree* for a string s is a trie in which we store each of the $n + 1$ substring identifiers for s. An example is shown in the following figure. (Following standard suffix tree conventions, we put labels on the edges rather than in the nodes, but this data structure is typically implemented as a trie or a patricia trie.)

As an example of answering a query, suppose that we want to know how many times the substring "abb" occurs within the 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 the example, this is 2.) By storing this information in each node of the tree, we can answer these queries in time proportional to the length of the substring query (irrespective of the length of s).

Since suffix trees are often used for storing very large texts upon which many queries are to be performed (e.g. they were used for storing the entire contents of the Oxford English Dictionary and have been used in computational biology) it is important that the space used by the data structure be $O(n)$ where n is the number of characters in the string. Using the standard trie representation, this is not necessarily true. (You can try to generate your own counterexample where the data structure uses $O(n^2)$ space, even if the alphabet is limited to 2 symbols.) However, if a patricia trie is used the resulting suffix tree has $O(n)$ nodes. The reason is that the number of leaves in the suffix tree is equal to $n + 1$. We showed earlier in the semester that if every internal node has two children (or generally at least two children) then the number of internal nodes is not greater than n. Hence the total number of nodes is $O(n)$.

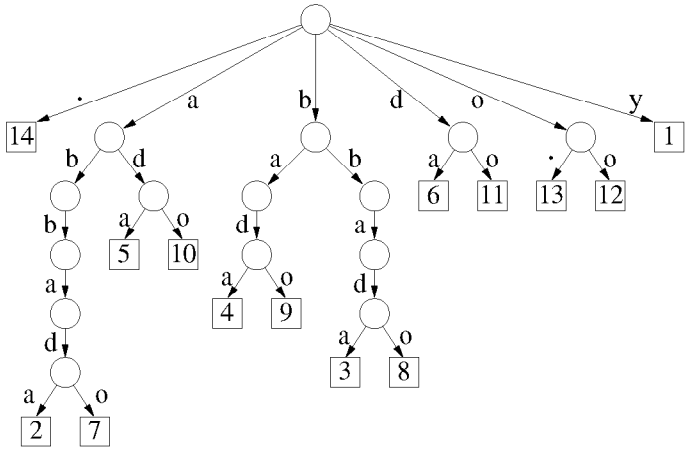

Figure 79: A suffix tree for the string "yabbadabbadoo.".

Lecture 26: Hashing

Reading: Chapter 5 in Weiss and Chapter 6 in Samet's notes.

Hashing: We have seen various data structures (e.g., binary trees, AVL trees, splay trees, skip lists) that can perform the dictionary operations `insert()`, `delete()` and `find()`. We know that these data structures provide $O(\log n)$ time access. It is unreasonable to ask any sort of tree-based structure to do better than this, since to store n keys in a binary tree requires at least $\Omega(\log n)$ height. Thus one is inclined to think that it is impossible to do better. Remarkably, there is a better method, at least if one is willing to consider expected case rather than worst case performance.

Hashing is a method that performs all the dictionary operations in $O(1)$ (i.e. constant) expected time, under some assumptions about the hashing function being used. Hashing is considered so good, that in contexts where just these operations are being performed, hashing is the method of choice (e.g. symbol tables for compilers are almost always implemented using hashing). Tree-based data structures are generally prefered in the following situations:

- When storing data on *secondary storage* (e.g. using B-trees),
- When knowledge of the relative *order* of elements is important (e.g. if a `find()` fails, I may want to know the nearest key. Hashing cannot help us with this.)

The idea behind hashing is very simple. We have a table containing m entries. We select a *hash function* $h(x)$, which is an easily computable function that maps a key x to a "virtually random" index in the range `[0..m-1]`. We will then attempt to store the key in index $h(x)$ in the table. Of course, it may be that different keys are mapped to the same location. This is called a *collision*. We need to consider how collisions are to be handled, but observe that if the hashing function does a good job of scattering the keys around the table, then the chances of a collision occuring at any index of the table are about the same. As long as the table size is at least as large as the number of keys, then we would expect that the number of keys that are map to the same cell should be small.