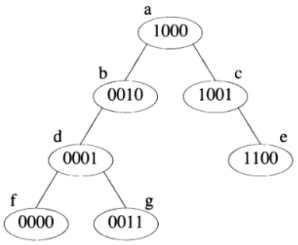
- A trie stores a collection $\mathcal{C}$ of keys, where each key is a string. The ADT of a trie has the following three operations: (i) whether a a query string belongs to $\mathcal{C}$, (ii) inserting a new string into $\mathcal{C}$, and (iii) deleting a string from $\mathcal{C}$. The word trie is derived from re*trie*val. For a fixed alphabet Σ, every string in $\mathcal{C}$ is defined over Σ. We assume $|\Sigma|$ is a constant.

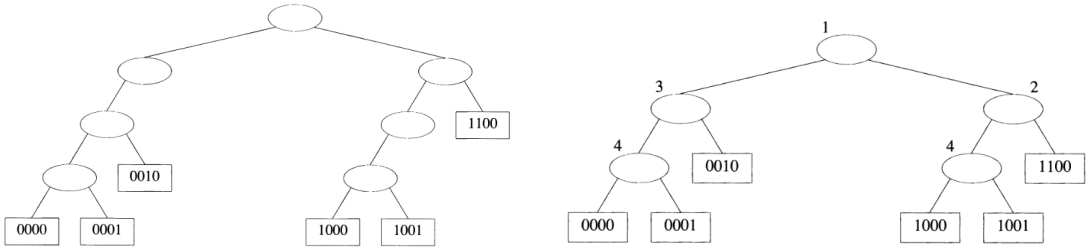
  Objective: Then, the data structure should take at most $O(n)$ space to store all the strings in $\mathcal{C}$, where n is the sum of lengths of all the strings in $\mathcal{C}$. To insert, delete, or to search for a key of length k, each of these operations should take $O(k)$ time.

- *Digital (binary) search tree*: A binary tree in which each node contains a unique key, and the $(i+1)^{st}$-bit of every key in the left subtree (resp. right subtree) of any node at any level i is equal to 0 (resp. 1).



  Since every key in $\mathcal{C}$ is stored exactly at one node, the space complexity is as desired. The search and insert operations are same as for BST search and delete, except that subtree to search/insert is determined by a bit in the search key. Considering the search key is compared with the key stored at every node along the search path, the search/insertion algorithm takes $O(k^2)$ time. Here k is the number of bits in the input key. The deletion algorithm first searches for the given key. If the key is at a leaf, then that node is deleted from the tree. However, when the input key is located at any internal node v, the deleted key is replaced by a key from any leaf ℓ in the subtree rooted at v before ℓ gets removed from the tree. Hence, the deletion algorithm also takes $O(k^2)$ time. The tries defined below improve these time complexities.

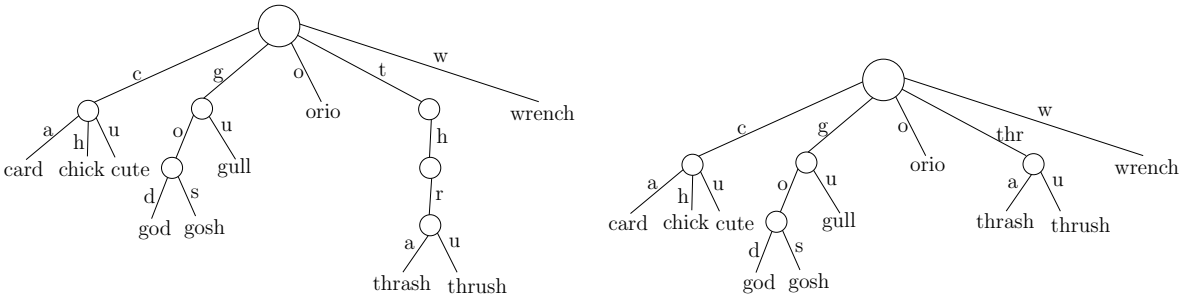
- *Binary trie*: This is a binary tree with only leaf nodes storing keys. A unique key is stored at each leaf node. The internal nodes help in guiding the search: for any node v at any level i, the $(i+1)^{st}$-bit of every key stored in the left subtree (resp. right subtree) of any node at any level i is equal to 0 (resp. 1). Hence, in searching for any key k, the $(i+1)^{st}$-bit of k is used in branching at node belonging to any level i. And, as usual, an unsuccessful search falls off the binary trie. From this, it is immediate that the search time for a query string of length k is $O(k)$. To facilitate storing keys of different lengths, each at a unique leaf of a binary trie, a special symbol (ex. \$) can be appended to every key. Essentially, this special symbol forces each key to be at a unique leaf node.



  Compressed binary trie: This trie improves the space of a binary trie by pruning every node that has only one child. This is achieved by applying the following till no node with single child exists in the tree: for

a node v with its parent $v.p$ and its only child $v.c$, make $v.c$ as a child of $v.p$ and remove v. Besides, with every internal node v, a bit number i is saved with v; that is, at v, the i^{th}-bit of a key is used for branching at v. This leaves each edge of the tree implicitly labeled with a string. For example, in the right figure, the edge joining the root and its left child has 00 label; this says, the prefix of every key stored in the left subtree of the root is 00; and, the branching at the left child of root is based on bit number 3 of the key. For any leaf node ℓ that is deleted, while walking along the path P from ℓ to root, P is compressed by removing nodes with one child on P. Due to these path compressions, by induction, if n keys are stored in a trie, than that trie has at most $n-1$ internal nodes. Like in binary tries, the query time in searching for a key of length k is $O(k)$.

- *Multiway trie*: Any node will have at most $|\Sigma|$ number of children. At any node on the j^{th} level, branching is determined by $(j+1)^{st}$ digit of the key. Hence, the i^{th}-child of an internal node in level j points to the subtrie having keys whose $(j+1)^{st}$ digits are the same. Depending on the application, a branching node may be implemented as an array, as a BBST, or as a hash table. Again, typically, a special symbol (ex. \$) is appended to each key, so that to store keys of different lengths at leaves of the trie.



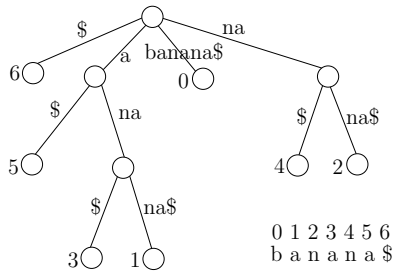
Compressed multiway trie: This is a multiway trie, but nodes with one child are compressed. This require saving the digit number based on which branching happens at each node. (Instead, in some implementations, number of levels of branching to be skipped is stored with the node.) Assuming $|\Sigma|$ is a constant, the total number of nodes is at most a constant multiple of the number of leaves in the trie. Hence, for a set $\mathcal{C}$ of strings, this trie takes $O(\sum_{s_i \in \mathcal{C}} |s_i|)$ space. And, each edge is labeled with a string; however, instead of storing a string s with any edge e, to save space, two pointers, pointing into the input corresponding to first and last digits of s are saved with e. The same is with the leaves: none of the leaf nodes store strings; instead, each refers to a location in the input. It is immediate that the time to search for a query string of length k takes $O(k)$ time. For any node v, concatenation of strings associated to each edge along the simple path from root to v in that order is called the path label of v.

- Homework: Given a collection $\mathcal{S}$ of strings in lexicographic order, preprocess $\mathcal{S}$, so that to output the location of any query string P in the lexicographic ordering of $\mathcal{S} \cup P$ in $O(|P|)$ time.

- Objective: Given a text string T of length n, preprocess T and build a data structure so that to answer queries of the following form: given a pattern string P, determine whether P is a substring of T.

  Observation: A string P is a substring of T if and only if P is a prefix of some suffix of T.

  A *suffix trie* (also called, a *suffix tree*) $\mathcal{T}$ of T is a compressed ordered multiway trie comprising all suffixes of T. The T is appended with a special symbol \$ not in T. This helps in no suffix of T being a proper prefix of another suffix of T; in turn, every suffix of T occurring at a distinct leaf node of $\mathcal{T}$ and each leaf node of $\mathcal{T}$ storing a unique suffix of T. As mentioned, this also lets a trie storing keys of different lengths, each at a unique leaf node. Hence, a suffix tree of T has exactly $n+1$ leaves, one for each suffix of T and one for \$. And, since it is a compressed multiway trie, every internal node has at least two children. Every edge e of $\mathcal{T}$ has two pointers into T, delimiting the non-empty substring of T that e is labeled with. (In the below figure, edges are labeled with strings only for convenience.) Each leaf ℓ of $\mathcal{T}$ refers to an index in T from where the suffix corresponding to ℓ starts. Further, the inorder traversal of $\mathcal{T}$ lists leaf nodes of $\mathcal{T}$ such that suffixes referred by those leaf nodes are in lexicographic order. Hence, this trie is an ordered trie.



  A naive algorithm to compute a suffix trie, inserts i^{th}-suffix into current suffix trie $\mathcal{T}_{i-1}$ in the i^{th} step. To form trie $\mathcal{T}_i$, algorithm starts from the root and follows the unique path matching digits in the i^{th}-suffix. If the traversal does not end at an internal node, algorithm creates an internal node v and initiates branching at v and (re-)labels edges appropriately. This algorithm takes $O(|T|^2)$ time to construct a suffix tree $\mathcal{T}$ of T, and $\mathcal{T}$ consumes $O(|T|)$ space. Using $\mathcal{T}$, determining whether P is a substring of T takes $O(|P|)$ time.

- A generalized suffix trie $\mathcal{T}$ of text strings T_1 and T_2 is a suffix trie comprising suffixes of both T_1 and T_2 at its leaves. This could be accomplished by constructing a suffix trie $\mathcal{T}'$ for $T_1\$T_2\#$, and pruning every node v of $\mathcal{T}'$ if the path label of v has symbols from both T_1 and T_2. Let $\mathcal{T}$ be the resulting trie. Each leaf of $\mathcal{T}$ represents either a suffix from one of the two strings or a suffix that occurs in both the strings, and every suffix of T_1 and every suffix of T_2 is at a leaf of $\mathcal{T}$. This definition can be extended by obvious means so that the generalized suffix trie comprises suffixes of more than two strings at its leaves.

- A few more applications of suffix tries:

- Find the longest substring of T that appears at least $m > 1$ times in T:

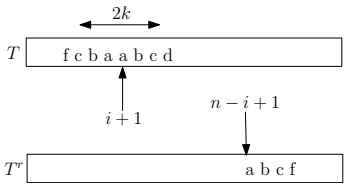
  While traversing the suffix trie, find an internal node v with number of children greater than or equal to m and the path label of v is the longest. This algorithm takes $O(|T|)$ time.

  (A variant of this problem is the longest repeated substring problem. In this problem, m is not given, but the substring of interest occurs at least twice.)

- Find the longest common substring of strings T_1 and T_2:

  Compute a generalized suffix trie $\mathcal{T}$ for T_1 and T_2. With a depth-first traversal of $\mathcal{T}$, mark each internal node v of $\mathcal{T}$ with 1 (resp. 2) if T_v has at least one leaf representing a suffix from T_1 (resp. T_2). With another traversal of $\mathcal{T}$, identify an internal node of $\mathcal{T}$ which is marked with both 1 and 2 and for which the path label is of maximum length. This takes $O(|T_1| + |T_2|)$ time. (There is a well-known dynamic programming (DP) based algorithm for this problem, which uses only $O(\min(|T_1|, |T_2|)$ space, but takes $O(|T_1| \cdot |T_2|)$ time. And, with a slight modification, that DP based algorithm can compute a longest common subsequence of two input strings.)

- Finding all even-lengthed maximal palindromes in T:



here, L is a b c, and k is 3

  Reverse text T to obtain T^r. Compute a generalized suffix trie $\mathcal{T}$ for T and T^r. Preprocess $\mathcal{T}$ in linear time to answer lowest common ancestor queries in $O(1)$ worst case time. With a depth-first traversal of $\mathcal{T}$, for every $i \in [1, n-1]$, with i, store a pointer to leaf node $v_{i+1,T}$ of $\mathcal{T}$ that represents the suffix $s_{i+1,T}$ starting at $i+1$ in T and a pointer to leaf node v_{n-i+1,T^r} that represents the suffix s_{n-i+1,T^r} starting at $n-i+1$ in T^r. And, for every $i \in [1, n-1]$, find the lowest common ancestor v of nodes $v_{i+1,T}$ and v_{n-i+1,T^r} in $\mathcal{T}$. If the path label L of v has nonzero length k (implying L of length k is a prefix of both $s_{i+1,T}$ and s_{n-i+1,T^r}), then there is a maximal palindrome of length $2k$ whose mid point is in between i and $i+1$ digits of T. This algorithm takes $O(|T|)$ time.

- When $|\Sigma|$ is not necessarily a constant, the suffix trie for a string T requires $O(|T| \cdot |\Sigma|)$ space, and searching for a pattern P using this trie takes $O(|P|)$ time. The algorithms by Weiner or McCreight to compute suffix tries take linear time when $|\Sigma|$ is a constant; however, these algorithms maintain several auxiliary data structures. On the other hand, the famous Ukkonen's (online) algorithm is complicated and takes linear time when $|\Sigma|$ is a constant. The algorithm by Farach is an optimal algorithm for all alphabet sizes, but it is also complicated.

  Hence, as an alternative, a data structure called suffix array was proposed, which is space efficient, even when $|\Sigma|$ is not a constant. A *suffix array* comprises indices of all the suffixes of $T\$$ listed in lexicographic order. A suffix array of $T\$$ could be computed by listing indices stored at the leaves of suffix trie of $T\$$ by traversing it in inorder. Naturally, the size of suffix array is $O(|T|)$. For example, the suffix array corresponding to $banana\$$ is $6\,5\,3\,1\,0\,4\,2$. Given a suffix array $\mathcal{A}$ of $T\$$, to determine whether a query string P is a substring of T, one could do a binary search in $\mathcal{A}$ and find the smallest index i such that P is a prefix of suffix corresponding to suffix index stored in $\mathcal{A}[i]$. Obviously, the query algorithm takes $O(|P| \lg |T|)$ time. The i^{th} entry of a longest common prefix (LCP) array stores the LCP of the i^{th} and $(i+1)^{st}$ suffixes in the sorted order of suffixes. For example, the LCP array corresponding to suffix array of $banana\$$ is $0\,1\,3\,0\,0\,2$. With the help of LCP arrays, the query time can be improved to $O(|P| + \lg |T|)$ (details omitted). Though the pattern matching queries on suffix tries are answered in only $O(|P|)$ time, where $|P|$ is the length of the pattern string, suffix arrays are space-efficient.

  There are simpler linear time algorithms to compute suffix array of T directly (without using suffix trie to compute it). And, there are linear time algorithms to compute a suffix trie of T given the suffix array corresponding to T. The other advantages of suffix arrays include an implicit data structure, $|\Sigma|$ does not play a role in time and space complexities (esp., useful if $|\Sigma|$ is large and cannot be considered as a constant), better cache locality, and efficient parallel implementations. Considering these advantages, to store suffixes of a string, in practice, suffix arrays are preferred over suffix trees.

References:

- Fundamentals of Data Structures by E. Horowitz, S. Sahni, and S. A.-Freed.

- Wikipedia.

Courtesy: All the figures in this note are from the above two resources.