iRAT #004 UT5

Data structures and Algorithms in Java - Willey

- [ ] **13.1** *Notation for character string*
- [ ] **13.3** *Tries*
- [ ] **13.3.1** *Standard Tries*
- [ ] **13.3.2** *Compressed Tries*

13. 1 Notation for character string

We denote the alphabet from which the characters come from as $\Sigma = \{...\}$.

Although we assume a fixed size for Σ, the size for it can be pretty significant, as with Java's treatment to the UNICODE alphabet.

A substring for a given P string can be defined as $P[i...j]$ meaning the substring of P from index i to index j inclusive, with i>0 and j < n - 1, n being the length of P. If i>j then by convention that substring is null.

- We define a prefix of P as $P[0...j]$ with $0 < j < n - 1$.
- We define a suffix of P as $P[i...n - 1]$ with $0 < i < n - 1$.
- Proper substrings cannot start on index 0 or end at index n-1.

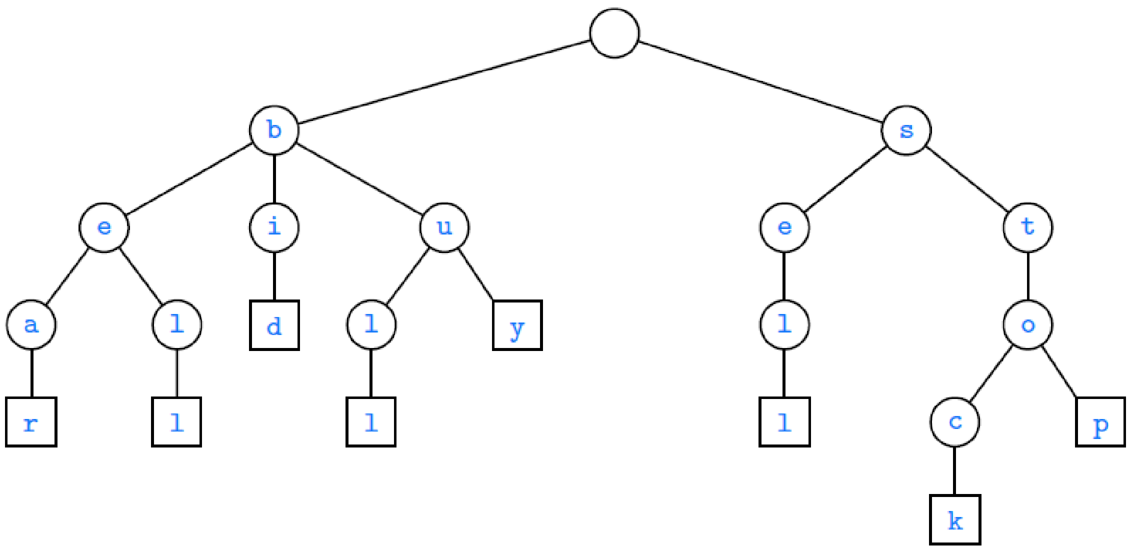
13.3 Tries

A trie is a tree-based data structure for storing strings in order to support fast pattern matching.

In a collection S we are given a set of strings all defined by the same alphabet.

The primary query operations that $tries$ support are pattern matching, and prefix matching given a string X, looking for all strings in S that begin with X.

13.3.1 Standard Tries



Let S be a set of strings all defined by Σ in such a way that no string in S is a substring of another.

A standard trie for S is an ordered tree T with the following properties:

- each node of T, except the rot, is labeled with a character from Σ
- the children of an internal node of T have distinct labels.
- T has leaves, each associated with a string of S, such that the concatenation of the labels of the nodes from root to leaf yields the string associated with such leaf.

Trie T represents the string of S with paths from root to leaf. The fact that no string in S is a substring of another, ensures that each string in uniquely associated with a leaf from T. We can always satisfy by adding a special character that is not present in the original Σ at the end of each string.

There is an edge going from the root to one of its children for each character that's first in some string of S. A path from the root to an internal node v at depth k corresponds to a k-character prefix, meaning $X[0...k-1]$ of a string X of S.

A trie concisely stores common prefixes that exists among the strings of S.

As a special case, if there's only two characters in Σ, then the trie becomes a binary tree, with some internal nodes only having one child (making it an improper binary tree).

Runtime analysis

Although it is possible for an internal node to have up to $|\Sigma|$ children, in practice the number is likely to be much smaller. The average degree of nodes decreases as we go deeper into the trie.

Preposition 13.4

A standard trie storing a collection S of strings of total length n from the alphabet Σ:

- The height of T equals the length of the longest String in S.
- Every internal node has up to $|\Sigma|$ children.
- T has S leaves, meaning, it has the number of leaves as S has number of Strings.
- The number of nodes of T is at most n + 1.

The worst-case scenario for the number of nodes of a trie occurs when no two strings share a common non-empty prefix. Except for the root, every internal node has one child.

If we perform a search in a trie T for a String X of S, we trace down from root to c. If this path can be traced and c is a leaf, then X is a String of S. Conversely, if the path it not traceable or it ends in an internal node of T, X is not a String of S.

At first glance, running time for a String length m is $O(m * |\Sigma|)$, because we visit at most m + 1 nodes of T and spend $(|\Sigma|)$time at each node determining the child having the subsequent character as label.

The upper bound $(|\Sigma|)$ is achievable even if the children of a node are out of order, since there are at most $|\Sigma|$ children.

We can improve running time by mapping character to children using a secondary search table, hash table or a direct lookup table of size $|\Sigma|$ (given that it is small enough) at each node.

For these reasons, we expect a search for an m-length String to run in $O(m)$ time.

Pattern matching

- Word matching: determines whether a given pattern matches one of the words of the trie exactly.

Word matching differs from pattern matching because we cannot match arbitrary sub-strings, only full words. To achieve this we need to add all the text's words to a trie. An extension of this paradigm can support prefix-matching. Although suffix-matching, for example, cannot be executed efficiently.

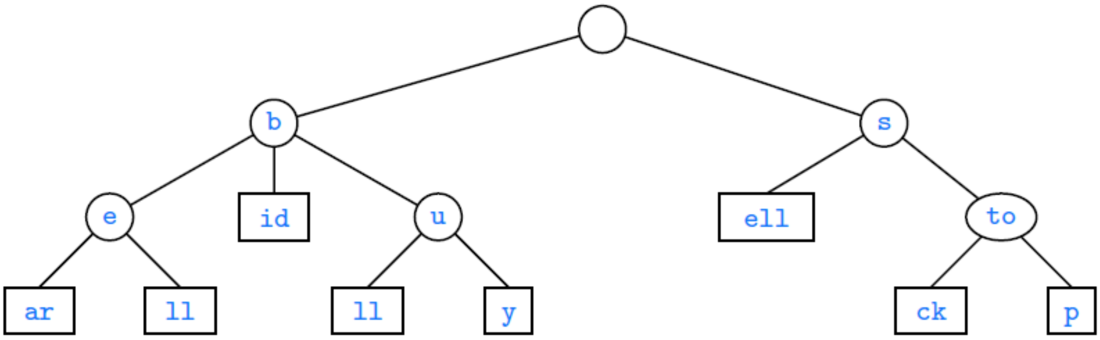
Based on the assumption that no String of S is a sub-string of another, we can insert words one at a time by tracing the path of a given string X and creating new nodes once we get stuck with a new character from $|\Sigma|$. The runtime to insert X with length m is worst-case $O(m * |\Sigma|)$ or expected $O(m)$ is using secondary hash tables at every node.

Creating a trie for a set S takes $O(n)$ time, n being the total length of all Strings contained in S.

Compressed tries or Patricia tries also exist but are not included in the mandatory reading.

13.3.2 Compressed Tries

A compressed trie is similar to a standard trie but it ensures that each internal node in the trie has at least two children. It enforces this rule by compressing chains of single-child nodes into individual edges. We say that an internal node v of T is redundant if v has one child and is not
the root.



This additional compression scheme allows us to reduce the total space for the trie itself from O(n) for the standard trie to O(s) for the compressed trie, where n is the total length of the strings in S and s is the number of strings in S. Searching in a compressed trie is not necessarily faster than in a standard tree, since there is still need to compare every character of the desired pattern with the potentially multicharacter labels while traversing paths in the trie.