1 Overview

Tries are another class of data structure which is useful for storing strings.

2 Character Trie

2.1 Terminology

The term trie is very general and so for this section we’ll use the term character trie. The reason for this modification will become immediately clear.

2.2 Character-by-Character

Perhaps the easiest way to store string would be as follows. Suppose we have a list Σ of \( n \) possible characters. We construct a tree in which the internal nodes are simply intersections with one branch per character. The strings are stored in leaf nodes only and the leaf node for a string is found by following the corresponding branches. In the following the horizontal bar nodes are taken to be null.

Example 2.1. For example, suppose \( \Sigma = \{a, b, c\} \) and we wish to store the words:

\[
\text{ab, baa, cab, bab, cba}
\]

The corresponding character trie would look like this:

![Character Trie Example](image)

Note 2.2.1. Notice that we don’t actually need to store the strings in the leaf nodes since the string is found by concatenating the branch labels from the root to the leaf.

2.3 String Endings

You might immediately see a serious issue with the previous example. Where would we store the string “baab”? We’d have to replace the “baa” leaf with an internal node but then we can’t store “baa”. The issue that we see is that we can’t store two strings such that one is a prefix of the other.
One solution to this is to have a terminating character, something like “$”. If each string ends with “$” then no string will be a prefix of another.

The above example is pretty icky if we do this, but here is a simpler one. In this case we have also used the convention of not putting the key in the leaf node but just putting “$” to indicate the string has terminated:

**Example 2.2.** For example, suppose $\Sigma = \{a, b\}$ and we wish to store the words:

\[a, aa, ba, bab\]

We use the “$” leaf node label and branch label whenever a string terminates:

One small benefit to this approach is that we can store the empty string as just “$”.

### 2.4 Data is Branches!

Under this new approach note that the data is no longer stored in the nodes, either leaf or internal. Rather it each datum is a path from the root to a “$” node.

### 2.5 Height

The height of a character trie is easily seen to be the length of the longest string stored plus one for the “$”. While this might seem very appealing the trade-off is that character tries tend to have lots of children, even if they’re null pointers, and hence tend to be very wide. This slows down tree performance immensely.

### 2.6 Search

Search in such a character trie is straightforward, we simply follow the tree letter-by-letter until we reach the corresponding leaf, or don’t, in which case we respond accordingly.
2.7 Insertion

Insertion is also straightforward with a caveat. We follow the tree until we fall out of the tree. At that point the path through the tree generates a prefix of the insert string. At that point we have to build out a new branch of the tree deep enough to account for all the remaining characters in the string.

2.8 Deletion

Deletion can be a bit sneaky. We first follow the appropriate path to the leaf to ensure that the string exists in the tree. Once we reach the leaf we travel back to the last node which has more than one (non-null) child and we remove the appropriate branch.

Example 2.3. Revisiting this tree from earlier:

To remove the word “bab” we first travel to the leaf to ensure that the word is there. Then we travel back up to the last node which has more than one (none-null) child and we remove that branch:
3 Compressed Trie

3.1 Terminology

There is some confusing history in the terminology of the next data structure. Originally they emerged from a 1968 document about what were called PATRICIA Trees (Practical Algorithm to Retrieve Information Coded in Alphanumeric) but the term Patricia Tree is now most often used to refer to a type of what are often called Radix Trees. However Radix Trees most often have a maximum number of children associated with them and our data structure will not. Radix Trees themselves are a subset of what are known as Compressed Tries in which there is no child limit. So we’ll use the term Compressed Trie.

The term compressed trie is somewhat preferable for us because it speaks to the relationship to the previous trie we looked at.

3.2 Substrings

The crux of the matter here is that instead of branching by single character we branch by maximal shared substrings of text. In addition we insist that no node has just one child except for when that child is the string terminator.

If this reads a bit confusingly, think about it this way:

1. Group all the words with the same first letter.
2. Within each group, take the longest shared prefix, that prefix will be a branch. All of those words will be in that branch’s subtree.
3. For each subtree do the same thing but ignore the previous prefixes.

Example 3.1. For example consider the strings:

heli, heed, help, hel, nook, noon

We group these by first letter and note that in the “h” group the longest shared prefix is “he” and in the “n” group the longest shared prefix is “nook”. Thus there are two branches, one corresponds to “he” and contains “heli”, “heed”, “help”, and “hel” and the other corresponds to “nook” and contains “nook” and “noon”.

For the left branch we repeat the procedure with “li”, “ed”, “lp”, and “l”, these will group into the group “li”, “lp”, “l” and the group “ed”, each getting a branch, etc.

For the right branch we repeat the process with “k” and “n”, these will group into the group “k” and the group “n”, each getting a branch, etc.

The end result is:
3.2.1. **Note**

Note that there is no theoretical maximum to the number of children that a node could have. In practical terms, however, there are only so many words with matching substrings which might align so there is a practical limit.

In any case the children of each node must be stored in variable-length list rather than in an easy per-child structure.

In addition as we’ll see, this makes time complexity a bit tricky.

### 3.3 Height

We have the following theorems regarding the height of a compressed trie:

**Theorem 3.3.1.** The height of a compressed trie is bounded by the length of the longest word plus 1.

*Proof.* The compressed trie for a set of strings is no higher than the character trie for the same set of strings. \( \square \)

**Theorem 3.3.2.** The height of a compressed trie containing \( n \) strings is bounded by \( h \leq n + 1 \). This includes final branches to “$”.

*Proof.* The basic outline of the proof is by structural induction on \( n \). We show that:

- A compressed trie with zero strings has height \( h = 0 \) and \( h = 0 \leq 0 + 1 = n + 1 \).

- If \( h, n \) are before insertion and \( h', n + 1 \) are after insertion then observe that insertion of a string will either result in \( h' = h + 1 \) or \( h' = h \). Why? Either way \( h' \leq (n + 1) + 1 \) will hold.

- If \( h, n \) are before deletion and \( h', n + 1 \) are after deletion then observe that deletion will either result in \( h' = h - 1 \) or \( h' = h \). Why? The latter might be worrisome but it only occurs when \( h < n + 1 \). Why? Either way \( h' \leq (n - 1) + 1 \) will hold.
In general the height of a compressed trie will depend on the characteristics of the words being stored. For that reason an in-depth analysis is quite challenging.

**Note 3.3.1.** It may seem odd to have two theorems which both give upper bounds but they are relevant in different circumstances. In a (very!) short list we may be more focused on the second theorem whereas in a longer list we may be more focused on the first theorem.

**Example 3.2.** If our trie stores up to 100 common english words each with at most 10 letters then the first theorem tells us that the height will be at most \(10 + 1 = 11\).

**Example 3.3.** If our trie stores up to 100 strings of 5000 bits each then the second theorem tells us that the height will be at most \(100 + 1 = 101\).

### 3.4 Search

Search in a compressed trie is straightforward. At each node we find the matching substring of characters and follow that branch. We do this until we reach the corresponding “$” leaf, or don’t, in which case we respond accordingly.

Since each branch corresponds to a substring it follows that the depth we would need to go when searching is bounded above by the number of characters in the string. This is great because a high compressed tree does not imply a slow search time.

However we have the additional issue that there is no easy upper bound for the number of children that a node may have. It’s certainly true that a node may not have more children than the number of characters which make up the strings, plus one for the “$”, but that’s about all we can say. This means that the decision process at a node (which branch to follow) is bounded but not necessarily constant.

If we ignore the time complexity of the node processing then the time complexity of search is \(O(\text{len}(s))\) where \(s\) is the query string.

If we want to take the node processing into account then we need more information regarding the strings themselves. An extremely crude approximation would be to say that a node may not have more children than the number of possible substrings but given that we have not limited the length of the strings this is infinite, and hence impractical.

### 3.5 Insertion

Insertion into a compressed trie can be a bit tricky. Given a new string \(s\), we proceed as follows, starting at the root:
• If a prefix of $s$ matches a branch label exactly then we follow that branch and recurse at the child with the right substring of $s$.

• If a prefix of $s$ matches a branch label partially then we relabel (shorten) that branch label to be the prefix of $s$ which matches it, replace the child with a new child with two children, one with branch label being the abandoned letters and with subtree being the original child and one with branch label being the post-prefix characters of $s$ and leading to “$\$”. This is a bit confusing! See the second example below.

• If no prefix of $s$ matches any branch label partially then we create a new branch with that entire word as its branch label.

Here are examples of all three:
Example 3.4. Starting with the previous tree, here for reference:

Suppose we wish to insert “hello”. The prefix “he” matches the “he” branch and the subsequent “l” matches the “l” branch. So far we are here:

Now we are looking at just “lo” in our string which doesn’t match any branch so we create a new branch:
Example 3.5. Starting with the original tree, here for reference:

Suppose we wish to insert “nope”. No prefix of “nope” matches a branch label exactly but the prefix “no” matches the branch label “noo” partially.

We relabel that branch as “no”, create a new child with two branch labels, one with the abandoned “o” and the other with “pe”. The original child is then the child of the “o” label while the child of the “pe” label is just “$”:
Example 3.6. Starting with the original tree, here for reference:

Suppose we want to insert “cat”. Since neither it nor any of its prefixes match the root’s branch labels we just create a new child with branch label “cat” and which leads to “$”:

As with search, the time complexity is not trivial to analyze. Ignoring node processing the time complexity is $O(\text{len}(s))$ where $s$ is the inserted string. Taking node processing into account the time complexity is complicated business.

3.6 Deletion

Deletion is also rather tricky. Here are the rules:

- We remove the final branch corresponding to the word.
- If the parent has two or more children remaining, we are done.
- If the parent has just one child then we splice the child and concatenate the branch labels.
Example 3.7. Building off the previous example, here for reference:

Suppose we wish to delete “heed”. We remove the final branch and notice that the parent node has one child now:

We find it has only one sibling which follows the labels “he” and “l”. We move that sibling up and concatenate the branch labels “he” and “l”:

As with search and insert, the time complexity is not trivial to analyze. Ignoring node processing the time complexity is $O(\text{len}(s))$ where $s$ is the deleted string. Taking node processing into account the time complexity is complicated
business.