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 \( \Sigma \) 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 Diagram]

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 furthest ancestor which has more than one (non-null) child and we remove the corresponding 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 Formal Definition of a Compressed Trie

The following formal definition of a compressed trie can be glossed over if all you need to do is get stuff done. However this definition makes some theorems easier.

**Definition 3.2.1.** A compressed trie is a tree satisfying:

- The branches are labeled with strings.
- No two sibling branch labels share a prefix.
- Each leaf node is connected to its parent by a branch labeled “$”, otherwise “$” doesn’t appear anywhere.
- The only nodes which may have just one child are parents of “$” labeled-branches and the root.

**Definition 3.2.2.** Given a set of strings $S$, a compressed trie corresponding to $S$ is a compressed trie which has the property that for all $x \in S$ the string $x+“$ appears exactly once in the compressed trie as a unique path of labels from the root to a leaf and moreover every path of labels from the root to a leaf spells $x+“$ where $x \in S$.

**Theorem 3.2.1.** Given a set of strings $S$, the structure of the compressed trie corresponding to $S$ exists and is unique.

**Proof.** First we show that such a structure exists. We do this by explicitly constructing one as follows:

1. Group all the words with the same first letter.
2. Within each group, take the longest shared prefix, that prefix will be the label for a branch from the root. All of those words will be in that branch’s subtree.
3. For each subtree do the same thing but ignore the previous prefixes. In other words, recurse.
4. When complete, add a final branch labeled “$” to each node with the property that the path of labels from the root to that node corresponds to a string in $S$. 


Next we prove that such a structure is unique. Suppose $T_1$ and $T_2$ are both compressed tries corresponding to $S$ but that they differ from one another. The fact that they differ from one another means that we can find nodes $p_1 \in T_1$ and $p_2 \in T_2$ such that the path labels from $T_1$’s root to $p_1$ and the path labels from $T_2$’s root to $p_2$ are identical and both produce some string $\beta$ (possibly the empty string if $p_1$ and $p_2$ are the roots) but such that (WLOG) $p_1$ has a child branch label which $p_2$ does not.

There are now two possibilities:

• Suppose $p_1$ has a child branch labeled $\alpha$ (possibly “$\$”) but $p_2$ has no child branch label with prefix $\alpha$. This would mean a string with prefix $\beta \alpha$ (possibly a string and its terminating “$\$”) is stored in $T_1$ but not in $T_2$, which contradicts the fact that both correspond to $S$.

• Suppose $p_1$ has a child branch labeled $\alpha$ and $p_2$ has a child branch label with prefix $\alpha$, say $\alpha \alpha'$. In this case neither $\alpha$ nor $\alpha'$ can be “$\$” because $\alpha \alpha'$ cannot exist as a branch label. Then since a string with prefix $\beta \alpha \alpha'$ is stored in $T_2$ it must also be stored in $T_1$, which means that $p_1$’s child must have a child label with prefix $\alpha'$ but then it must also have another child, say with label $\gamma$ (possibly “$\$”), but then a string with prefix $\beta \alpha \gamma$ (possibly a string and its terminating “$\$”) must be stored in $T_1$ which means it must also be stored in $T_2$ but this would be another child of $p_2$ but then $p_2$ has child branches labeled both $\alpha \alpha'$ and $\alpha \gamma$, a contradiction.

QED

Here is the explicit construction of a compressed trie following the rules above:

Example 3.1. 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 “noo”.

Thus there are two branches, one corresponds to “he” and contains “heli”, “heed”, “help”, and “hel” and the other corresponds to “noo” 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 procedure with “k” and “n”, these will group into the group “k” and the group “n”, each getting a branch, etc.

The end result is:
Note 3.2.1. Note that the maximum number of children a node can have equals the number of characters in the alphabet.

3.3 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 highly 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 best-case $O(1)$ and worst-case $O(\text{len}(s))$ 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.

3.4 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 $s$ (treated as a new $s$).
• 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 $s$ as its branch label.

Here are examples of all three:

**Example 3.2.** 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 3.3. 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 3.4. 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 best-case $O(1)$ and worst-case $O(\text{len}(s))$ where $s$ is the inserted string. Taking node processing into account the time complexity is complicated business.
3.5 Deletion

Deletion is also rather tricky. Here are the rules:

• We remove the final branch (including the $ part) corresponding to the word.

• If the parent of that branch has two or more children remaining, we are done.

• If the parent of that branch has just one child then we remove the parent and concatenate the branch labels.

Example 3.5. 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 “I”. We move that sibling up and concatenate the branch labels “he” and “I”:
As with search and insert, the time complexity is not trivial to analyze. Ignoring node processing the time complexity is best-case $O(1)$ and worst-case $O(\text{len}(s))$ where $s$ is the deleted string. Taking node processing into account the time complexity is complicated business.

3.6 Height

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

**Theorem 3.6.1.** 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. \(\Box\)

**Theorem 3.6.2.** For a compressed trie $T$ if $h(T)$ is the height and $n(T)$ is the number of strings then $h(T) \leq n(T) + 1$. This includes final branches to “$\$”.

*Proof.* The proof can be done by structural induction or strong induction.

For the structural approach before proceeding note that we proved that compressed trie structures are unique. This means that we don’t need to prove the property is preserved during deletion because deleting a string from a compressed trie results in exactly the same compressed trie that we would obtain by simply rebuilding it using insertion on the remaining strings.

So then we simply prove that the property is true for some base case(s) and prove that it is preserved when we insert a string.

Observe that the base case(s) are quirky since when $n = 0$ we have an empty compressed trie with height $-1$ but when $n = 1$ we have $h = 2$.

\(\Box\)

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 3.6.1.** 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 3.6.** If our compressed 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 $10 + 1 = 11$ while the second theorem tells us that the height will be at most $100 + 1 = 101$. The first is more helpful.

**Example 3.7.** If our compressed trie stores up to 100 strings of 5000 bits each then the first theorem tells us that the height will be at most $5000 + 1 = 5001$ while the second theorem tells us that the height will be at most $100 + 1 = 101$. The second is more helpful.