CMSC 420: Tries

Justin Wyss-Gallifent

March 27, 2024

1 Overview

Tries are another class of data structure which is useful for storing data for which the keys are strings rather than integers.

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.

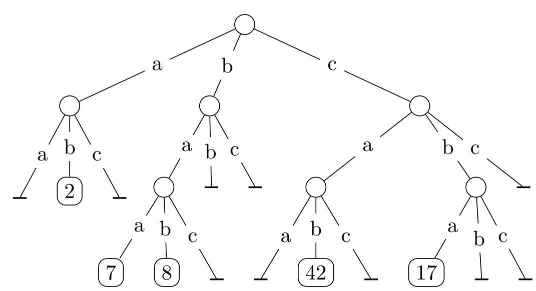
2.2 Character-by-Character

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 following key-value pairs:

$$\{\texttt{ab}, 2\}, \{\texttt{baa}, 7\}, \{\texttt{cab}, 42\}, \{\texttt{bab}, 8\}, \{\texttt{cba}, 17\}$$

The corresponding character trie would look like this:



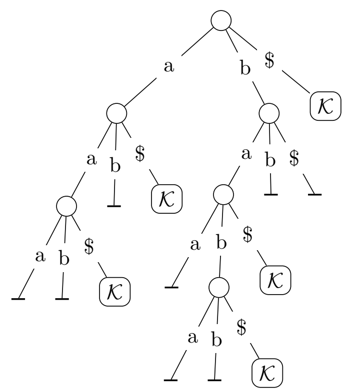
Note 2.2.1. Note that in what follows we won't worry about the values since the important thing is that we're storing the keys, which are strings. Whenever we write $\mathcal{K}$ we'll just mean some key.

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,(empty string)
>
> 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 `$` as seen above.

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.

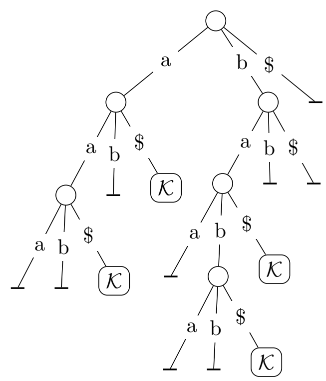
2.7 Insertion

Insertion is also straightfoward 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.

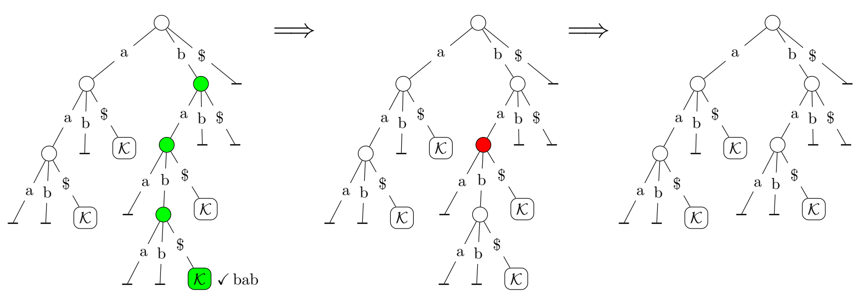
2.8 Deletion

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

The following variation is often called a *Radix Tree* but we're going to stick with the alternate term *Compressed Trie* or just *Trie*. Originally this data structure was called a PATRICIA Tree (Practical Algorithm to Retrieve Information Coded in Alphanumeric) but the term *Patricia Tree* is now most often used to refer to a specific sub-type of this structure.

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 (which typically contains a value) 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 `$` 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.

3.3 Proof of Existence and Uniqueness

We prove such a structure exists by constructing one.

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

(a) Group all the words with the same first letter.

(b) 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 subtrie.

(c) For each subtrie do the same thing but ignore the previous prefixes. In other words, recurse.

(d) 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.

QED

Now we prove the construction is unique.

Proof. 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 α (possibly the empty string if p_1 and p_2 are their respective roots) but such that p_1 and p_2 do not have exactly the same child branch labels.

Suppose p_1 has a child branch labeled β (possibly just \$) such that p_2 has no child branch labeled β. There are two possibilities:

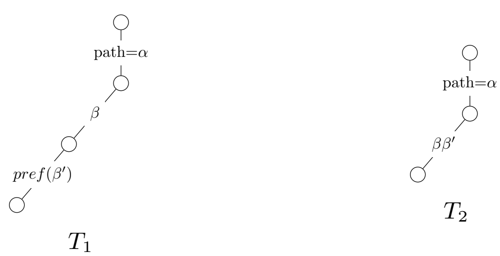
- p_2 has no child branches with β as a proper prefix.

  This tells us that a string with prefix $\alpha\beta$ is stored in T_1 but not in T_2, a contradiction.

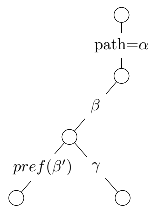
- p_2 has at least one child branch with β as a proper prefix, meaning a branch labeled $\beta\beta'$ with β' nonempty.

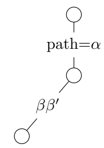
  Observe that neither β nor β' may be \$ because such $\beta\beta'$ are not valid branch labels. This tells us that a string with prefix $\alpha\beta\beta'$ is stored in T_2 and hence must be stored in T_1 which then requires that p_1's child has a branch label with prefix β'.

  But then p_1's child must have another child, say with label γ, meaning that a string with prefix $\alpha\beta\gamma$ is stored in T_1 and hence must be stored in T_2.

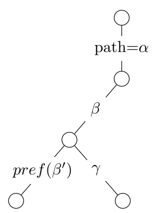


T_1

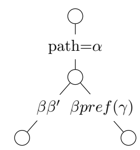


T_2

However then p_2 must have a child with label $\beta pref(\gamma)$.



T_1



T_2

This is a contradiction since p_2 has two child branches with the same prefix.

$\mathcal{QED}$

Here is the explicit construction of a compressed trie following the rules above: We'll put some keys back in here for illustration.

Example 3.1. Consider the key-value pairs:

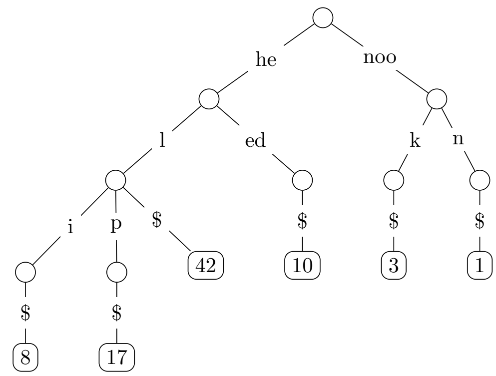
$$\{\texttt{heli}, 8\}, \{\texttt{heed}, 10\}, \{\texttt{help}, 17\}, \{\texttt{hel}, 42\}, \{\texttt{nook}, 3\}, \{\texttt{noon}, 1\}$$

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 `eli`, `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 procesure 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 3.3.1. Note that the maximum number of children a node can have equals the number of characters in the alphabet.

3.4 Search

Search in a compressed trie is straightforward. At each node we check the branches and find the matching substring of characters and follow the corresponding branch. If at any point there is no matching substring of characters or we do not reach a final `$` branch then the string is not in the compressed trie. If necessary the value(s) associated to the string can be stored in the final leaf node, or pointed to by it, etc.

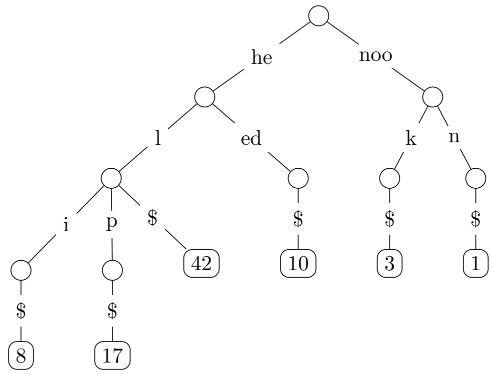
3.5 Insertion

Insertion into a compressed trie is fairly straightforward. Given a new string s, we start at the root and then:

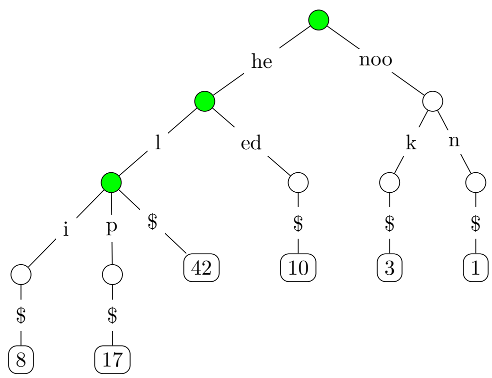
- If a prefix of s matches all of a branch label then we follow that branch and ask the same question at the child with the suffix of s treated as a new s. We continue doing this until we cannot, in which case one of the following applies.
- If a prefix of s matches a prefix of a branch label then we "shorten" and relabel that branch label to be just the common prefix, create a new node at the end with two branches, the first with branch label being the abandoned letters of the original branch and with subtrie being the original child and the second to a node with branch label being the suffix characters of s and with that node having its own single child containing the key and with branch labeled `$`.
- If no prefix of s matches any branch label partially then we create a new branch to a node with s as its branch label and with that node having its own single child containing the key and with branch labeled `$`.

Here are examples of all three:

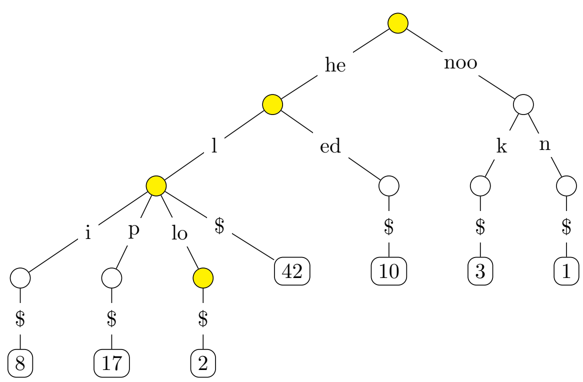
Example 3.2. Starting with the previous trie, here for reference:



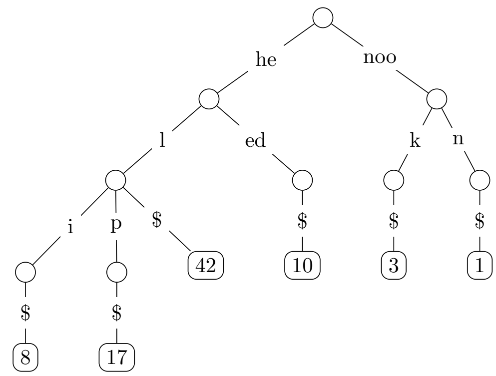
Suppose we wish to insert the pair $\{$`hello`$, 2\}$. The prefix `he` matches the `he` branch and the subsequent `l` matches the `l` branch. So far we are here:



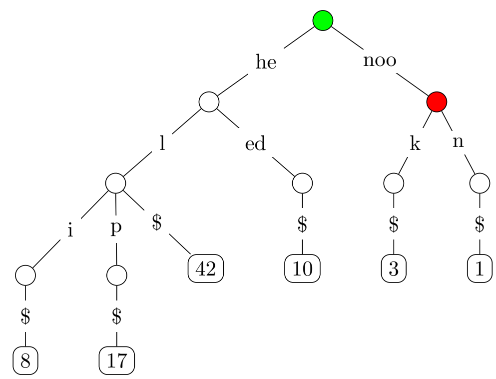
Now we are looking at just `lo` in our string which doesn't match any branch so we create a new branch:



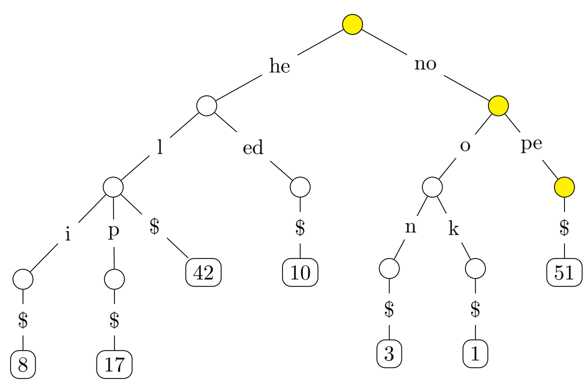
Example 3.3. Starting with the original trie, here for reference:



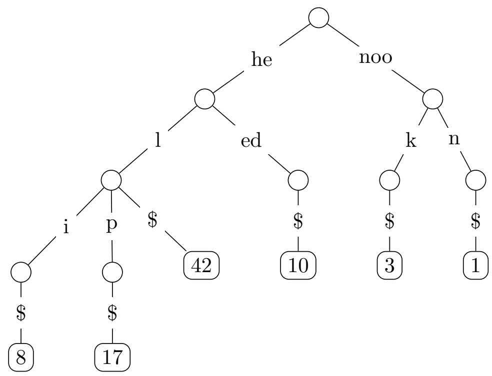
Suppose we wish to insert $\{$`nope`$, 51\}$. No prefix of `nope` matches a branch label exactly but the prefix `no` matches the branch label `noo` partially.



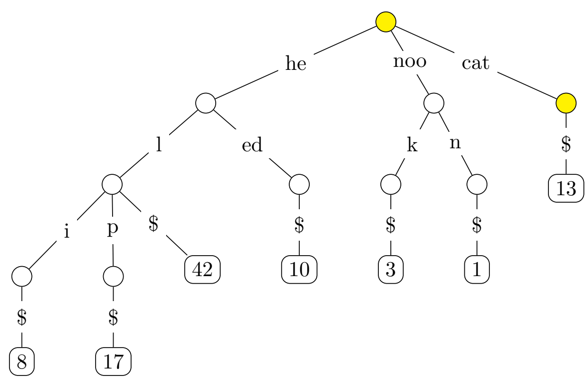
We relabel that branch as `no`, create a new child with two branch labels, one with the abandoned `o` and the other with `pe`. The original child is then the child of the `o` label while the child of the `pe` label gets one child labeled `$`:



Example 3.4. Starting with the original trie, here for reference:



Suppose we want to insert {`cat`, 13}. Since neither it nor any of its prefixes match the root's branch labels we just create a new child with branch label `cat` which also gets one child with branch label `$`.

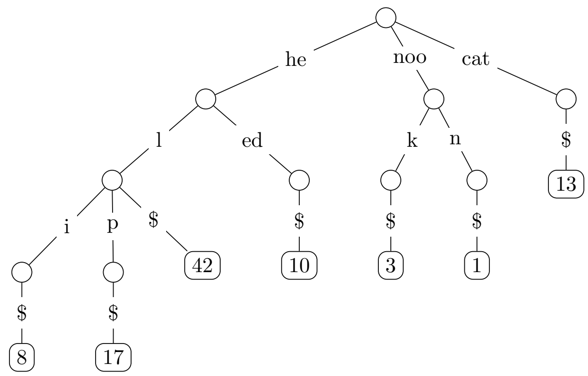


3.6 Deletion

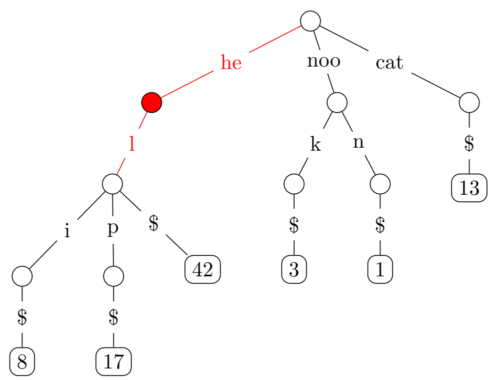
Deletion is also fairly straightforward:

- Locate the associated leaf node.
- Remove all branches back up the tree stopping when we reach a node with more than one child.
- If that node still has two or more children then we are done.
- If that node has one child, splice the branches above and below and concatenate the branch labels.

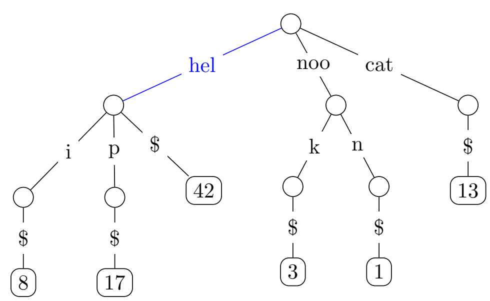
Example 3.5. Building off the previous example, here for reference:



Suppose we wish to delete the entry with key `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 best-case $\mathcal{O}(1)$ and worst-case $\mathcal{O}(\text{len}(s))$ where s is the deleted string. Taking node processing into account the time complexity is complicated business.

3.7 Height

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

Theorem 3.7.1. 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. $\mathcal{QED}$

Theorem 3.7.2. 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.

Proof. Well, not a full proof, but the outline:

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 $h = 2$. $\mathcal{QED}$

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 3.7.1. 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.

3.8 Time Complexity

We have seen that the height is bounded above by the length of the longest word plus 1 and is also bounded above by the number of strings plus 1. When we generally discuss complexity it is in terms of the number of items stored and so in this case $h = \mathcal{O}(n)$ where n is the number of strings stored. However we could also say $h = \mathcal{O}(k)$ where k is the length of the longest string being stored.

Using the former, we might then argue that search, insert, and delete are also worst-case $\mathcal{O}(n)$ because they simply need to traverse the trie (or just part of it) and because the operations performed during insert and delete are $\Theta(1)$.

This is not technically wrong but is rather an oversimplification because in reality at each node we need to perform a number of string comparisons and each of these string comparisons takes time depending on k, and while when we say $\mathcal{O}(n)$ we are explicitly ignoring k, this isn't always good because k can have an impact in real situations.

A few sample observations:

- If we are searching and we know the string s is in the trie then at each node we only need to compare the first character of s with the first character of each of the branch labels. This is $\mathcal{O}(1)$ because it is only bounded by the number of characters in the alphabet.
- If we are searching and we don't know the string s is in the trie then it's not sufficient to just compare the first character but rather we need to make sure the branch label matches the prefix of s exactly. This is $\mathcal{O}(k)$.
- What is going on for insertion and deletion?