Our lives and times have ever made
Just footnotes[1] to a tardigrade

[1] *Nebuloid-Gamma-Cluster TG34765.14 - A galaxy of standard configuration (like the Milky Way). Existed from 9123E123 to 9123E129 qsecs.*

The Prefix Tree

In under 200 lines of readable, commented code, (around 25-50 chars per line, I think), you can implement a brand new kind of exciting data structure. It is a data structure that is very useful to have in your inventory because it has uses that other data structures have yet to fill efficiently (and as elegantly).

It goes by many names in many places, including *prefix trees* and retrieval trees.

We'll just use the word *trie* in this spec, which, I believe, comes from the middle of "retrieve".

We tries. Only then we will succeeds

Here is an interesting problem. Suppose we have a lot of words, and want to find all words that share the same prefix, what's a good way to do it?

If you aren't familiar with this problem, consider yourself lucky. Take a break now, go for a walk and think about it before resuming.

One way is to create a hash-table (a dictionary, which we'll look at next quarter) which maps every possible prefix into a set of strings that can follow it. But that would be awfully wasteful, yes?

The trie presents another (I think, elegant) way. By now you're already familiar with the idea of using vector indices to encode data.[2] Consider the way you stored the cache in Quest 2 (Hanoi), for instance.

In a trie, the data you want to store is not stored explicitly under a data label, but is instead implicit in the presence or absence of a particular child.

You can think of the trie as an abstraction over a general tree (the koala quest) with a fixed number of children. In this spec, I'm using the vector of children technique to represent this particular kind of general tree.

[2] There is a subtle detail to appreciate here: Using *indices* to store data is fundamentally different from using the cells themselves. In the latter (common) case, the cell holds the data. In the former, it's implicit in one of many paths leading from that cell to another. This is like putting labels on the edges rather than nodes in a graph.

We'll apply yet another perspective shift to this representation and assume that the data element of a child node is equal to the index into the parent node's vector that led to it.

Then:

1. Every node in the trie can be reached from the root by following a prefix, P, of one or more words encoded in the trie.[3]
2. If the k'th element of the vector of node pointers contained in a node is NULL, then *P appended with the character whose ascii value is k* is NOT a prefix of any word encoded in the trie.
3. If it is not NULL, then it points to a unique node that can be reached by following a prefix constructed by appending to P the character whose ascii value is k.
4. A valid whole sentence ends in the 0th element of its terminal node's vector.

Check out Figure 1, in which the vector is 256 elements large. This allows that trie to encode any ASCII string.

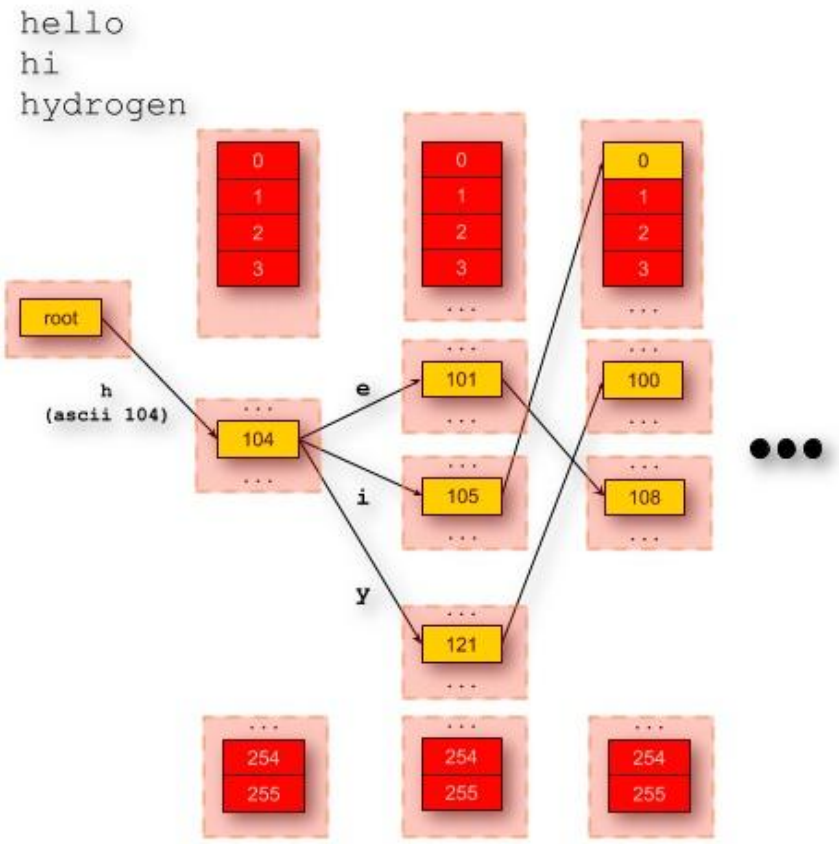


Figure 1: A Trie. Red cells are not live (null pointers). Each cell is within a vector contained in a trie node

[3] Assume that by *following* I mean that you traverse an ordered sequence of nodes from the root corresponding to a valid prefix.

What kind of storage overhead or gain are we talking about? (Compare to the next best working representation you can think of - e.g. a dictionary). What's the trade-off?

Starter code

Again, this is a relatively easy quest once you get past the *pig bicture.*[4]

So the quest master said "No can do".

But he let me snoop around and take photos with an old camera I found. Here's a fuzzy photo of the class def.

```
class Trie {
private:
    struct Node {
        std::vector<Trie::Node *> next;

        ~Node();

        void insert(std::string s);
        const Node *traverse(std::string s) const;
        bool lookup(std::string s) const;
        size_t get_completions(std::vector<std::string>& completions, size_t limit) const;
    } *_root;

    // Private Trie:: helper. Returns the interior node traversing s from _root
    const Node *traverse(std::string s) const { return _root->traverse(s); }

public:
    Trie();
    ~Trie();

    void insert(std::string s) { _root->insert(s); }
    bool lookup(std::string s) const { return _root->lookup(s); }
    size_t get_completions(std::string s, std::vector<std::string>& completions, size_t limit) const;
    size_t trie_sort(std::vector<std::string>& vec) const;

    std::string to_string(size_t n) const;
    std::ostream& operator<<(std::ostream& os) { return os << to_string(100); }

    friend class Tests; // Don't remove
};
```

More details can be found in the following miniquest descriptions.

[4] pig bictures stand in your line of sight of the big picture.

Your first miniquest - Make a trie

Implement the trie constructor:

```
Trie::Trie()
```

All this needs to do is allocate a brand new `_root` node and return.

And you get a reward for that? Hmm…

Your second miniquest - Insert

Implement the private `Node` method:

```
void Trie::Node::insert(string s)
```

which is needed to complete the `Trie::insert()` method. It should insert the given string into the trie at the current node (duh!). But it must be iterative. How might I be able to tell in my testing code? You may decide to recurse and sneak through. But it'll be a worse implementation. (Why?)

Tip: Exploit the fact that, in c++, the underlying c-style representation of any string contains a NUL[5] delimited sequence of the ASCII values of the characters of the string. It fits the bill PERFECTLY because it lets us directly use the NUL character (ascii value 0) as the index of the node into which the very last character of the string should branch. Can it get any better?

Here is a fuzzy photo of the reference implementation. Of course, it doesn't have to look like this. But it may help you come up with yours:

```
teratively insert the given string into the trie starting at the
urrent node

 Trie::Node::insert(string s) {
Trie::Node *curr = this;
for (const char *str = s.c_str(); *str; str++) {
    char ch = *str;
    if ((size_t) ch >= curr->next.size())
        curr->next.resize(ch+1);

    if (curr->next[ch] != nullptr)
        curr = curr->next[ch];
    else
        curr = curr->next[ch] = new Trie::Node;
}

// the \0
if (curr->
    curr->

if (curr->next[0] == nullptr)
    curr->next[0] = new Trie::Node;
```

[5] NUL is usually taken to mean the NUL character (byte) whose value is 00000000. Sometimes you'll find it coded as a single (escaped) character, '\0'. In C, this is a "formal" invalid character and used as a sentinel in string processing algorithms. Don't confuse it with NULL. That is the memory address 0 (null pointer).

At a high level, you want to start at the head of your string, jumping to your next nodes which are got by indexing into your `_next` vectors by the character in the string you must move over to get there. At the end of the string (at the node indexed into from the last character), jump into the 0th child.

Here's a useful tip: Watch out for duplicates. Unfortunately, you'll only know what I mean after you get bitten. But this tip might help you recognize the bite sooner before it ruins your experience.

Your third miniquest - Traverse

Implement the private `Node` helper method

```
const Node *Trie::Node::traverse(string s)
```

which is needed to complete `Trie::traverse(string s)`.

Uh Oh! No photo here.

Thankfully, the logic is very similar to `insert()`.

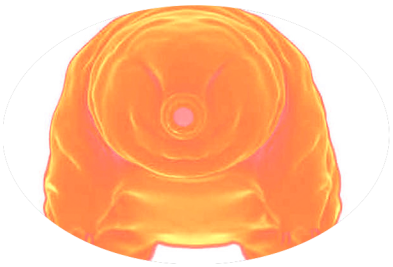
It should return a pointer to the node reached by traversing the trie from the root, one node per character in the supplied string `s`, in order.

As an example, traversing the trie for "`hydro`" starting at the root node in Figure 1 should return a pointer to the Trie node you would eventually end up at by entering it on the ascii value of the last letter '`o`'.

If the string (or its superstring) cannot be located in the trie, `traverse()` must return a null pointer. You can return a null pointer as soon as you know for sure a prefix cannot be found in the trie. What's the earliest in your search when you can know this for sure?

You'll sure be glad you implemented this method when it comes time to implement `lookup()`.

Your fourth miniquest - Lookup



"Hey! did I just hear my name?"

Implement the private `Node` method

```
bool Trie::Node::lookup(string s)
```

which is needed to complete `Trie::lookup()`.

It should return `true` if the string `s` is *wholly* contained within the trie. This means its last encoded (but not visible) character must be the sentinel (`\0`).

Leverage `traverse()` to get to the internal node that ends at the last visible character of the string. This node should encode the sentinel as a valid continuation. That is, `next[0]` at this node should be non-null.

As an example in Figure 1, the string "`hydr`" will end in a node whose `next[0]` will be null (since "`hydr`" is not a valid word. However, "`hi`" will end in a node whose `next[0]` will be non-null.

If `lookup("my-string")` succeeds, then it means that "`my-string\0`" is a valid prefix contained in the trie.

Your fifth miniquest - Destructor

Implement the destructors:

```
Trie::Node::~Node()

Trie::~Trie()
```

The trie destructor only has to delete `_root` and let the node destructor do everything else. It's ok for the node destructor to be recursive… in many use cases of a trie. (Why?)

All that the destructor needs to do is to delete each of the allocated children. There's nothing else to do. But if your bookkeeping is not up to snuff or your implementation is confusing, you may end up with nasty memory related errors.

Side note on memory errors

Many times you can get "lucky" and receive a fatal segmentation violation error when you read from memory that doesn't belong to you (or memory that has been freed by you). But luck is a fickle companion for computer programs. You can't rely on it.

How then can you minimize the likelihood of segfaults?

Although asking the run-time system to monitor ALL of your unallocated heap for illegal access is unfeasible, it is indeed possible, and a wise investment by the run-time system, to monitor a single location continuously. This location is memory address 0, the null pointer (`nullptr`).

You can exploit this fact to come up with a useful design technique:

1. In the first iteration of your program (or product), always explicitly set all deleted pointers to the value NULL.
2. This will guarantee that your program will crash AS QUICKLY AS POSSIBLE after an illegal access, thereby GREATLY SIMPLIFYING your debugging.
3. Users of your product will be quick to report crashes, and their reports will be more accurately representative of the bug in your code.

4. Release millions (figuratively) of copies of your app (instrumented thus). Get many bug reports and fix as many of them as possible.
5. When the bug reports slow down to a trickle, revisit your code and pull out the null assignments, getting a small performance improvement to reward your faithful users.

While we're on the subject of destructors, here is a worthwhile exercise to try in your spare time. Instrument the destructor at the appropriate points with the following lines:

- `static int r_depth = 0;`
- `cout << "~Node() Recursion depth " << ++r_depth <<endl;`
- `cout << "~Node() Descending from depth " << r_depth <<endl;`
- `cout << "~Node() Back to depth " << --r_depth << endl;`

Look at the output when you delete a trie containing many sentences. Share your insights on our subreddit. Did it help you in making your program leak-proof?

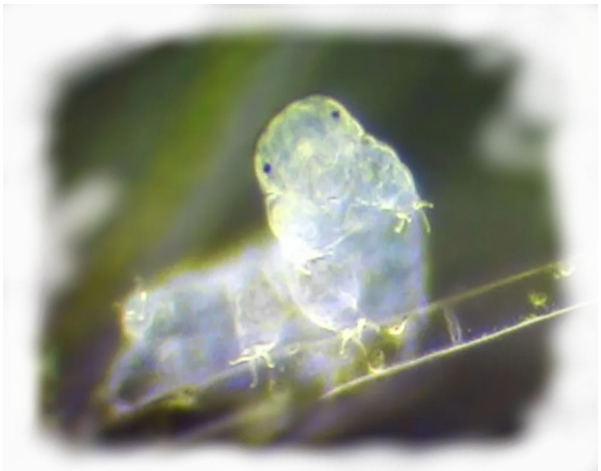
Make sure you remove the instrumentation before you submit your code into the questing site.

Your sixth miniquest - Get completions at node

Implement the private `Node` method:

```
size_t Trie::Node::get_completions(
    vector<string> &completions, size_t limit
) const
```

Clear the vector, `completions`, and fill it with up to `limit` strings that are possible completions of the empty string at `this` node. Essentially, this is the same as a *breadth-first traversal*.



It involves a number of subtleties. Read on. Some of it may not make sense right now. That's ok. I suggest that you read it anyway until it seems to not make any sense without a great deal of effort. Then try to rewind all the way back to the description of this miniquest and say "*Screw this spec. I'm just gonna do this by myself. It sure looks easier than trying to understand this gibberish.*"

If you end up solving the miniquest your way, collect the reward and move on. Otherwise, read the section below and you'll find it makes a whole lot more sense even though your hardcopy would have been unchanged!

You can achieve a breadth first traversal by *processing* the nodes of any general tree using the strategy below. You didn't get to do this in the Koala quest. But you get to do it here:

1. Start with a queue that only has the start node
2. Until the queue becomes empty:
    a. Pop the front node.
    b. Add all the children of that node to the end of the queue
    c. Process the node's data element (if appropriate)

(What happens if you had accidentally typed "stack" instead of "queue"?)

If you do this with a tree, you'll find that you traverse it in level order left-to-right.

All well and good if each node can tell you what its own data element is. But what do we do in situations like ours where a node's data element is implicitly hidden in the index of the parent's branch that led into it? This information is lost in the recursive invocation in which the "`this`" object points to the child and not its parent.

The way I handle it in this quest is by pushing packages of two things on the queue instead of single nodes. These packages will each contain the node to be processed, but also, importantly, the much needed history information of the prefix that led into it, which I've called `partial`.

BTW, a `struct` in c++ is simply a class in which every member is public by default.

At each node, consider all possible continuations (every ASCII char) and handle each of the following 3 cases for each child in sequence:

1. The character that leads into that child is NUL (`\0`)
2. The character has no continuation out of the current node
3. The character is a valid continuation out of the current node

Here is the starter code for `Trie::Node::get_completions()`.

```
size_t Trie::Node::get_completions(vector<string> &completions, size_t limit) const {
    struct Continuation {
        const Trie::Node *node;
```

```
        string partial;
        Continuation(const Node *p, string s) : node(p), partial(s) {}
    };

    // All string descendants of this node are completions
    queue<Continuation> unprocessed_nodes;

    // TODO - Your code here

    while(!unprocessed_nodes.empty()) {
        Continuation cont = unprocessed_nodes.front();
        unprocessed_nodes.pop();

        // TODO - Your code here

    }

    return completions.size();
}
```

Your seventh miniquest - Get completions in Trie

Implement

```
size_t Trie::get_completions(
     string s, vector<string> &completions, size_t limit
 ) const
```

If you had managed to get to this miniquest, you'll find it's nothing at all. Simply traverse your trie's root on the given string, and then generate all possible completions of the empty string at that node.

Of course, *if you can't traverse a trie with a given string, then you can only return a nothin' thing.*

Your eighth miniquest - To string

Implement

```
string Trie::to_string(size_t limit)
```

No complex logic here. The returned string should contain, in order:

1. The line: "`# Trie contents`"
2. Up to `limit` entries from the list of all completions of the empty string (`""`) in the trie, at one per line. The completions must be obtained using your previously completed `get_completions()` method call.
3. If there are more entries left after having added `limit` entries, add the line "`...`"

Your ninth miniquest - Trie Sort

Eazy peazy points. Just for surviving thus far, I guess.

Implement:

```
size_t Trie::trie_sort(vector<string>& vec) const;
```

It should clear and suitably size `vec` to hold all the distinct strings in the trie. Then it should fill it with all the completions of the empty string (no limit). Finally, it should return the size of this vec.

If you look at the result you'll see the encoded strings ordered a particular way. What is this order? Can you think of any practical applications for such a permutation of a list? How long does it take (as a function of the number of input strings) to generate a trie-sorted sequence of n strings?

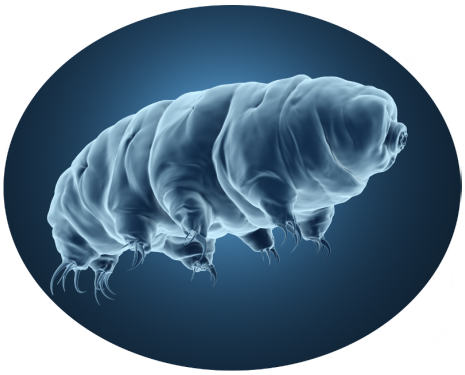
For extra fun, (if you're so inclined) share a Unix/Linux one-liner (not necessarily a single command) that will trie-sort an input list of strings.

How long is a well written solution?

Well-written is subjective, of course.

But I think you can code up this entire quest in about 200 lines of clean self-documenting code (about 25-50 chars per line on average), including additional comments where necessary.

Submission



When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to https://quests.nonlinearmedia.org
2. Enter the secret code for this quest in the box.
3. Drag and drop your `source` files into the button and press it.
4. Wait for me to complete my tests and report back (usually a minute or less).

Points and Extra Credit Opportunities

Extra credit points await well thought out and helpful discussions.

Happy hacking,

&