Lil' Trie Adventure
(alternative node definitions for smaller memory footprints)

March 30, 2024

Abstract

We have studied *tries*, a specialized tree designed for the quick search of sequenced data such as words or IP addresses. Our lecture and 3P version of lowercase word tries allocated 26 pointers (208 bytes!) to store the `.children[]` of **each node**.

This project has you consider more specialized node structures designed to reduce the overall memory footprint of the 3P word trie and investigates the run-time trade-off in doing so.

Part 1

Clone the `LilTrieAdventure` repo according to the Canvas assignment instructions. Drop in your `trie` header and source file from the Trie programming project and follow the `README.md` instructions to make sure you can build `run-main`.

> **Take Note 1**: The `CMakeLists.txt` expects your header and source to be `trie.h` and `trie.cpp`, but you can change these names if you choose.
> Additionally, this write-up assumes you've used a nested class structure for your `trie` class, with the "internal" node structure named `node`. (This approach wasn't a requirement of 3P, you may have opted to use to a single object definition that was itself a node.) You aren't required to use the former but you'll need to make some mental translations for clean compiles. You're permitted to change any of the functions provided in `liltrie.h` — we are interested in your results, not so much the code gymnastics to get them.[a]
>
> [a] This would be a wonderful project to learn about C++ polymorphism, user-defined iterators and virtual base class design — but these are beyond the scope of this course and assignment. It makes me sad `:/`

The first thing we will do is add some statistics collecting member functions to your `trie` and `node` classes. ***We'll refer to the base 3P trie node structure as `node`, you may have called it something else. That's not a big deal.*** The `trie::` versions of these functions are used by the provided code, they should all initiate a recursive chain of calls on the root's `node::` children to"get the ball rolling."

- `void trie::finalize() {}` and `void node::finalize() {}`: for now these function can do nothing, you may choose to use them in Part 4 of this analysis to reduce the amount of new code you have to write. (But these need to be implemented for compilations.)
- `size_t trie::size()` should report the total number of words stored in the trie (recall a word exits where `node::is_terminal` is `true`). You may want to implement a `size_t node::size()` helper function as well, which could return the number of words at or below the current node. `trie::size()` will be used to help verify correct implementations of your compressed trie node and the changes you make to your 3P trie node.
- `size_t node::node_children()` reports the number of children a node has. You'll need to iterate through `.children[]` and return the count of non-`nullptr`s in the array.
- `size_t trie::descendants()` and `size_t node::descendants()` reports the total number of non-`nullptr` child edges in the trie (`trie::`) or at and below the current node (`node::`). The calculation of the latter is the sum of `node::descendants()` for all of the calling object's children plus `node::node_children()` of the calling object.

- `void node::children_histogram( vector<size_t>& histo )` should increment the `node_children()` valued index in `histo` by one:

```
histo[this->node_children()]++;
```

  and recursively call `child_histogram` on its children with the same `histo` parameter.

Once you have the node member functions implemented and compiling cleanly, change the zero in `main-lta.cpp`'s `#if` line to a `1` and rebuild. The `test_node_functions<Y>` parameterized function in `main()` will make sure your implementations compile cleanly. You will probably want to populate the string of vectors with data from the `wordlists` files for a more thorough test.

Expected values for the various word dictionaries used in this project are in the `test_node_functions` directory of the repo.

> **Take Note 2**: If `test_node_functions<Y>` is called with a dictionary less than 2000 words or with sample=false, it will put all the dictionary vector's words into the trie. If the dictionary has more than 2000 words and `sample=true`, it will put a random number of words into the trie and test `.contains()` for both included and un-included words. The default value for the function is `sample=false`.
> `test_node_functions<Y>` always **randomizes** the order in which words go into your trie and the order in which they are tested for appropriate `.contains()` results.

Now we want to collect some statistics about the structure of our uncompressed trie after it has stored a dictionary of words. We will build a discrete count (or "cardinal") histogram of the `node_children()` results in our trie. A *discrete data histogram* records either the absolute number or the population percentage of a set of distinct keys or attributes in a population. In this case the population is all the nodes in a trie, the keys are the number of children trie nodes have for a particular dictionary. Calculating the histogram values is best done with a helper function, which has been provided to you in `liltrie.h`.

```
void collect_child_histogram<yourTrie>( yourTrie& root, vector<string> words,
       vector<size_t> histogram_array );
```

Provide `collect_child_histogram<Y>` an empty trie of your creation, a vector of strings read from one of the project dictionaries (you've done similar word reading in 2A) and the `size_t` vector for the histogram.

`collect_child_histogram<Y>` has two optional parameters, if `sample_size=COLLECT_CH_FULLSIZE` then all the dictionary vector words are used, if `sample_size=COLLECT_CH_RANDOMSIZE` a random number of words are sampled, and if `sample_size>1`) a precise number of words are sampled. The default for `collect_child_histogram<Y>` is to sample a random number of words from the dictionary vector.

If the Boolean parameter `show=true`, `collect_child_histogram<Y>` will display to the console the histogram result as both counts and population fractions.

Expected values for the smaller word dictionaries (size < 2000) are in the project's `child_histogram` directory of the repo. If your implementation results from calling `collect_child_histogram()` don't match up, the issue is likely in your `descendants()` or `node_children()` functions. Of course, you can always inspect `liltrie.h` for `collect_child_histogram()` implementation details.

The value at `child_histogram[i]` is the number of nodes in the trie that have `i` children (or the fraction of nodes in the trie with `i` children, if you are looking at the frequency histogram results).

Part 2 – a smaller trie node

Our first alternative node structure to investigate stems from the observation that the majority of tree nodes have in fact a small number of children. Our new node structure takes advantage of this feature by using a fixed size smaller array to hold the children pointers for the majority of nodes. We'll call the size of this fixed child pointer array T. If a node has more than T children, an additional array of `LETTERS-T` pointers is allocated and maintained to hold the overflow (`LETTERS` is the size of the language alphabet, in our case 26).

```
struct trie_Tnode {
  bool is_terminal;
  char fixed_letters[T];
  struct trie_Tnode* fixed_children[T];
  typedef trie_Tnode* trie_Tnode_ptr;
  struct trie_Tnode_ptr* *overflow_children;
};
```

In this case, we have used a `typedef` for a `trie_Tnode` pointer, which should make reading and writing this code more straightforward for students. (It can be a more approachable way for students to think of "pointers to pointers.") The `typedef` is used only at compile time, so it does not increase the memory footprint of a `trie_Tnode`.

You may not have to implement `trie_Tnode` for this project, so hold off on writing code for it just now! But we do want to explain how it works so you can be convinced it is a valid structure for a compressed trie.

We have included a `fixed_letters[]` array so we know which characters are associated the with edges stored in `fixed_children`. (In 3P, we knew this information implicitly by adding `'a'` to the index of our children array.) When trying to find the edge associated with a character, we would follow these steps:

Access a `trie_Tnode` letter edge

a. First consider the `T` `fixed_letters[]` values. If any of these are for the edge we seek, we are done.
b. If the `overflow_children` pointer is `nullptr`, then we don't have an edge for the letter we seek and we are done.
c. Otherwise, we iterate through the `LETTERS-T` elements of `overflow_children`, associating `'a'` with index 0, `'b'` with index 1 and so forth. **EXCEPT** (and this is the tricky part) we skip over the `T` letters held in `fixed_letters`. So if `T=2` and `fixed_letters={'a', 'c'}`, `overflow_children[0]` is associated with the `'b'` edge, `overflow_children[1]` is associated with the `'d'` edge.

The size of this alternative node in memory depends on T and the total number of child edges stored in the node (C).

$$nodeSize(C) = \begin{cases} sizeof(bool) + T \times sizeof(char) \\ \quad + T \times sizeof(trie\_Tnode*) + sizeof(trie\_Tnode\_ptr*) & C \leq T \\ \\ sizeof(bool) + T * sizeof(char) \\ \quad + T \times sizeof(trie\_Tnode*) + sizeof(trie\_Tnode\_ptr*) & \text{otherwise} \\ \quad + (LETTERS - T) \times sizeof(trie\_Tnode\_ptr) \end{cases}$$

data member	memory allocated	size in bytes
`is_terminal`	allocated with object	`sizeof(bool)`

`fixed_letters[0]`
`fixed_letters[1]`
⋮
`fixed_letters[T-1]`
	allocated with object	`T*sizeof(char)`

`fixed_children[0]`
`fixed_children[1]`
⋮
`fixed_children[T-1]`
	allocated with object	`T*sizeof(trie_Tnode*)`

`overflow_children`	allocated with object, initialized to `nullptr`, valued with `new` in `.insert()` as needed	`sizeof(trie_Tnode_ptr*)`

How much space can be saved with this `trie_Tnode` structure? It depends on T (of course) so the task is to determine the optimal T given the `child_histogram` valued from `collect_child_histogram()`.

Let N be the number of nodes in the trie, `N=trieObj.descendants()`. Let s be the number of nodes in the trie with $\leq T$ children (so $N - s$ is the number of nodes in the trie with $> T$ children). The memory footprint of a trie using `trie_Tnode` nodes is

$$footPrint = s \times nodeSize(T) + (N - s) \times nodeSize(T + 1)$$

Clearly, we could use this equation to find the optimal T value for a particular dictionary, which is to say the T that produces the smallest trie $footPrint$.

Part 3 - empirical evidence

Include the `struct trie_Tnode` definition in your program source and write two functions: `node_footprint` and `trie_Tnode_footprint`. All the `*_footprint` functions in this project will take the same parameter list, we show the details with `trie_Tnode_footprint`:

```
size_t trie_Tnode_footprint( size_t N, const vector<size_t>& child_histogram );
```

where `N` is the total number of nodes in a trie and `child_histogram` is the data determined from `collect_child_histogram()`.

`trie_Tnode_footprint()` should return the smallest footprint a trie would have using `trie_Tnode` nodes — you don't have to report the optimal T, but you are welcome to print it to the console for debug or testing.

The `node_footprint()` calculation is much simpler, it is based on the Trie 3P node definition and is simply:

```
return N*(sizeof(bool) + LETTERS*sizeof(node*))
```

data member	memory allocated	size in bytes
`is_terminal`	allocated with object	`sizeof(bool)`

`children[0]`
`children[1]`
⋮
`children[LETTERS-1]`
	allocated with object	`LETTERS*sizeof(node*)`

Write a loop to run an experiment (tweak the following as you see fit):

1. Read the words from the project dictionary (`dictionary.txt`) into a vector.
2. For some reasonably large value of `EXPERIMENTS`:

```
vector<size_t> histogram_data[LETTERS+1];
while( EXPERIMENTS-- ) {
  trie = myTrie();
  collect_child_histogram( trie, words_vector, histogram_data);
  size_t N = trie.descendants();
  node_footprint = node_footprint( N, histogram_data );
  Tnode_footprint = trie_Tnode_footprint( N, histogram_data );
  resultsFile << N << " " << node_footprint << " " << Tnode_footprint << endl;
}
```

A second, even smaller trie node

Both the 3P trie node definition and the `trie_Tnode` definition over-allocate space for children (the `trie_Tnode` does this only when the number of children is not exactly T). We can reduce the footprint of a trie node even more with some bit-twiddling and allocating a `children[]` array for only the number of children needed for each node.

We'll call this final node structure a `trie_Mnode`, where M stands for "mask" or more accurately "bitmask". A bitmask uses a standard integer data type but treats each bit as a Boolean value. Our `LETTERS=26`, so we can have a flag for each character, when that flag is on or "up", then our node has a valid pointer for the character edge in its `children` array. A four byte integer (`int`) has 32 bits, so we have more flags than needed for characters, let's use the last flag as our `is_terminal` Boolean.

The mask layout (labeling the bits with indices 0–31) would be

bit index	0	1	2	3	4	5	···	23	24	25	···	31
associated with	'a'	'b'	'c'	'd'	'e'	'f'	···	'x'	'y'	'z'	(unused)	`is_terminal`

(See the Helpful hints appendix to this write-up for bitwise manipulation of masks — but not before you've read through Part 4!)

The `trie_Mnode` data declaration would be:

```
struct trie_Mnode {
  int mask;
  typedef trie_Mnode* trie_Mnode_ptr;
  trie_Mnode_ptr* children;
};
```

The memory footprint of this data structure depends on the absolute number of children a node has, C. There is no flexible parameter like T in `trie_Tnode`. The memory footprint of `trie_Mnode` is simply:

```
sizeof(int) + sizeof(trie_Mnode_ptr*) + C*sizeof(trie_Mnode*)
```

Deliverable 1
Incorporate a `trie_Mnode_footprint` function into your application code and generate a scatterplot with N on the independent horizontal axis and the results of our three trie node memory footprints each in their own series. Provide this final three series scatterplot in your submission.

Deliverable 2
Summarize your scatterplot results by answering these questions.

a. How much space savings would you expect to gain for storing a dictionary of 100,000 words?

b. Is the percentage of expected savings consistent across a range of dictionary sizes?

c. Which $O(n)$ relationships that we've studied ($O(1), O(n), O(n \log n), \ldots$) might this space savings belong to? **We aren't asking you to derive this relationship! Just comment on which it might be.**

Runtime considerations

While we have decreased the memory footprint of the trie, have we changed its runtime complexity? If so, to what extent? Consider this: with `node` constructed tries (as in 3P) there is immediate access to the child pointer of a letter. It is simply `.children[letter-'a']`, that's $\mathcal{O}(1)$ for those of you who are counting `:)`

With `trie_Tnode`, things aren't as simple (recall the Access a `trie_Tnode` letter edge algorithm mentioned before). If `n=LETTERS` — what is the $\mathcal{O}(n)$ of this algorithm? What are its average (expected) and worst case runtime complexities? Hint: an overly simple solution is **not** $\mathcal{O}(1)$ — but you might do better...

On the other hand, we see substantial space savings with both `trie_Tnode` and `trie_Mnode`, which might translate to better cache performance and mitigate some of the worsened runtime complexity. It's "time" to let the our machines weigh in on this question!

Part 4 — runtime performance

How do these two different compressed node trie implementations fair at runtime? Is it worth it to use substantially more memory for better performance? The last question depends on the application needs and its runtime environment (is it on a small embedded device with limited memory?). In a professional setting you would want to collect some performance measures to guide your decision — and this is precisely what you'll do in this last part of the project.

We ask you to implement **only one** of the two compressed node definitions and report your findings compared to your original trie definition (the 3P node structure). There are two ways you can go about this task. Regardless of your approach you will need to run your timing experiments on both your 3P `node` structure and your compressed node trie.

On the fly

You can write a brand new trie class and manage the compressed node structure from start to finish. This means implementing the constructor, destructor, `.insert()` and `.contains()` (at least). There is more memory management required for this scheme, but it's not too much to ask from students in this course. We call this "on the fly" because you'll be allocating memory as needed in `.insert()`.

Using `.finalize()`

You can also copy your current trie class definition (or, if you're into the whole class hierarchy approach, you could inherit from your original trie class) and implement or rework the `.finalize()` and `.contains()` functions for the class. The advantage to this approach is that it doesn't require any memory management and you won't have to debug `.insert()` and constructor logic for a brand new class.

The key to this solution is the `.finalize()` member function. This is called by all the routines in `liltrie.h` after all words have been `.insert()`'d into a trie. When `.finalize()` is called, you can

morph the data stored in your 3P node structures into the member variables associated with your chosen compressed trie node. You may see how this would work in code right off the bat, if not see item V. in the Helpful hints appendix to this write-up.

Generating paired timed comparisons

When you have your compressed node compiling cleanly, test it with `test_node_functions<Y>` and varying sized dictionaries. Compare the results against your original trie implementation and make sure the reported metrics match up.

Finally, you're ready for the last bit of data generation in CSCI220!

Use the `time_trials<Y,O>` function defined in `liltrie.h` to generate a pair of `std::chrono::duration<double>` values. The returned pair's `.first` is the runtime for the `Y` trie class, it's `.second` is the runtime for the `O` trie class. The function should be provided with empty tries for comparison (the `Y` trie being your 3P equivalent, the `O` trie being your compressed node implementation), a dictionary of words and an optional Boolean `finalize` parameter. The default value for `finalize` is true. Your invocation might look like:

```
myTrie     trie3P;
trie_Tnode trieO;
auto secs = time_trials( trie3P, trieO, dictionary )
size_t W = trie3P.size();
cout << W << " " << secs.first << " " << secs.second << endl;
```

If you have implemented an "on the fly" compressed node, you can provide `finalize=false` and the timing results will reflect the total load time as well (all the `.insert()` calls).

You will provide a scatterplot of these timing results for your submitted write-up to this project. Of course this could be a simple two series plot with W on the horizontal access and runtime on the vertical axis. However care has been taken in `time_trials<Y,O>` to make sure that each trie sees the same order of words for both `.insert()`'ions and `.contain()`s tests — so it is legitimate to compare these values directly within one experiment.

Consider the following four styles of scatterplot graphs to show your results, if t_{3P} and t_O are the `.first` and `.second` values for one experiment:

a. You could plot the difference in time vs trie size:

$$(W, t_O - t_{3P})$$

b. The percentage increase in runtime by the compressed node:

$$(W, (t_O - t_{3P})/t_{3P})$$

c. The difference in time normalized by the number of words in the trie:

$$(W, (t_O - t_{3P})/W)$$

d. Or you could plot the coordinate pair (t_{3P}, t_O) along with the line of identity $(y = x)$. If these points lie on or near the line of identity it means the runtime performance is nearly identical. If the points lie above $y = x$, then t_O is consistently $> t_{3P}$.

Deliverable 3
Choose your preferred graphic presentation(s), make sure the axis, title and labels are accurate and provide a plot of your runtime results. **Also**, be sure to state whether you have coded an "on-the-fly" or `.finalize()` solution.

Deliverable 4
Among the implementations you've provided runtime results for, which would you prefer to use in a real world application? What development or runtime factors might influence your choice?

Deliverable 5
Finally, copy and paste a nicely formatted version of the `.contains()` logic of your compressed node implementation *and any other functions it may call.* Use a fixed width font, single line spacing and consistent block indentation for this in your report.

Helpful hints

I. For "on the fly" implementations, You want to avoid memory leaks and corrupted data, fortunately our runtime needs for a compressed trie node are simple: allocate memory as needed in `.insert()` and make sure our destructor frees all memory for a node. You **really** want to avoid the stray typo or synapse misfire and accidentally copy construct or assign a node, because without these procedures properly implemented you will likely have very hard to debug issues when testing large tries. Recall that C++ will provide default (aka, dumb, wrong) versions of these "big 3" functions for you. You can prevent this by providing declarations for these in your class definition, and then **not** providing procedure definitions. So your `trie_Mnode` should have the following prototypes:

```
trie_Mnode( const trie_Mnode& trie );
trie_Mnode& operator=( const trie_Mnode& rhs );
```

Providing these will prevent C++ from using its default versions. Now when you *would have* introduced memory errors, you'll be greeted with linker errors when building. And hopefully you'll say to yourself: "Wait a minute — I shouldn't need an assignment operator!"

II. For "on the fly" implementations of `trie_Mnode`, you can use the following bitwise operators for turning bits on and testing their state:

```
// bit is 0 to 31
static inline bool bit_on( int mask, unsigned bit )
{ return mask & (1<<bit); }

static inline int set_bit_on( int mask, unsigned bit )
{ return mask | (1<<bit); }
```

III. For "on the fly" implementations, remember that

```
children = new trie_Xnode_ptr[Q];
```

does not initialize the elements to `nullptr` — you have to do that yourself.

IV. For the `trie_Tnode` compressed node, a small insight into the Access a `trie_Tnode` letter edge algorithm can make a substantial improvement in the node's runtime performance. We need to avoid a one-by-one search of `.overflow_children[]`, which seems difficult because for each letter slot to consider, that letter might be in the `.fixed_letters[]` array and should be ignored. Here is a nice idea:

If this were a 3P trie node, we would know the slot in `.overflow_children[]` associated with a letter, it would be `letter-'a'`. And, in fact, this **is** slot we should inspect if all the values in `.fixed_letters[]` come **after** the query letter in the alphabet! Some examples:

A. if `.fixed_letters[] = {'s', 'b', 'q'}` and we are looking for `'a'`, it is still in slot 0 of `.overflow_children[]` because s, b, and q are all after a in the alphabet.

B. with the same `.fixed_letters`, if we are looking for `'r'`, it won't be at slot `'r'-'a'`, because b and q come before r and their edges are stored in `.fixed_children[1]` and `.fixed_children[2]` respectively. The r slot of `.overflow_children[]` is `'r'-'a'-2`.

C. one more: `'z'` won't be at index 25 of `.overflow_children[]` because s, b, and q have their edges stored in `.fixed_children[]`. So the z edge is at `.overflow_children[LETTERS-T-1]` (or `'z'-'a'-3`).

Do you see the pattern? If we are looking for an edge associated with the letter L

a. first calculate the variable `int L_in_overflow=L-'a';`

b. when inspecting each letter F in `.fixed_letters`, decrement `L_in_overflow` by one if $F < L$.

c. now, if you have to consider `.overflow_children[]` (L is not in `.fixed_letters`, and `.overflow_children != nullptr`), the slot to inspect is `.overflow_children[L_in_overflow]`.

V. Here are more details for implementing the `finalize()` approach for runtime measurements. Recall from the initial description, the idea here is to leverage the code and structure already written (and working!) for 3P in order get runtime measurements with the least amount of code to write and debug.

The steps for implementing `.finalize()` "fake" compressed nodes are:

(a) Copy your 3P trie class header and source files as starting points to your compressed node implementation.

(b) Keep all the data members of your original class, **augment** the class' definition with the data members you'll need to "fake" your compressed node at runtime (details for `trie_Tnode` are on page V. and a similar example for `trie_Mnode` is on page V.).

(c) Implement `finalize()` for the node to convert the 3P stored data into the design used by your compressed node (again, an example for each are below).

(d) Change your `.contains()` implementation to only use the information stored for your particular compressed node.

The examples below show possible "fake" compressed node implementations that will permit accurate runtime measurements. In each case we try to reuse the object footprint of the 3P trie node definition to the greatest extent.

Each compressed node example shows how data from an arbitrary node would be translated from the 3P structure to the "fake" compressed mode structures. In both cases we show the result of `.finalize()` would be for a terminal node containing child edges associated with the letters a, c, h, q, t and v.

Example: converting from `node` to `trie_Tnode`

Our "`trie_Tnode` augmented" 3P `node` structure would look like:

```
struct fake_trie_Tnode {
  static const int T=17;   // pick a better value guided by your histograms :)
  /** node data members **/
  bool is_terminal;
  fake_trie_Tnode* children[LETTERS];
  /** need to know which letters are in the ``fixed'' block */
  char fixed_letters[T];
};
```

For the example, we'll suppose `trie_Tnode::T=4`. We pretend the first `T` slots of `children` are `trie_Tnode::fixed_children`, and indexes `T` through `LETTERS-T` of `children` are `trie_Tnode::overflow_children` — **your `.contains()` should be re-written appriopriately!**

`node::` data member	**value**	moved to `fake_trie_Tnode::` data member(s)
`is_terminal`	true	`is_terminal` (same)
`children[0]`	0x40007780	`fixed_letter[0]='a'; children[0]` = 0x40007780;
`children[1]`	`nullptr`	none
`children[2]`	0x40007880	`fixed_letter[1]='c'; children[1]` = 0x40007880;
⋮		
`children['h'-'a']`	0x40007980	`fixed_letter[2]='h'; children[2]` = 0x40007980;
`children['q'-'a']`	0x40007c80	`fixed_letter[3]='q'; children[3]` = 0x40007c80;
⋮		
`children['t'-'a']`	0x40007b80	`children['t'-'a'-4]` = 0x40007b80;
`children['v'-'a']`	0x40007a80	`children['v'-'a'-4]` = 0x40007a80;

Take Note 3: The special case of `children[0]` being unchanged in this example is because this node has an `'a'` edge. This is certainly *not* gauranteed for every node in a trie.

Example: converting from `node` to `trie_Mnode`

A "`trie_Mnode` augmented" 3P `node` structure might look like:

```
struct fake_trie_Mnode {
  /** node data members **/
  bool is_terminal;
  fake_trie_Mnode* children[LETTERS];
  /** need a mask for trie_Mnode::contains() and destructor logic */
  int mask;
};
```

For `trie_Mnode`, we simply need to

a. "Crunch" the pointers spread out in `children` down to the first 6 slots that would have been allocated for `trie_Mnode::children`,
b. and store which letter edges exist as well as the value of `is_terminal` into `mask`.

node:: data member	value	moved to fake_trie_Mnode:: data member(s)
is_terminal	true	set_bit_on(mask,31);
children[0]	0x40007780	set_bit_on(mask,0); children[0] = 0x40007780;
children[1]	nullptr	none
children[2]	0x40007880	set_bit_on(mask,2); children[1] = 0x40007880;
⋮		
children['h'-'a']	0x40007980	set_bit_on(mask,'h'-'a'); children[2] = 0x40007980;
children['q'-'a']	0x40007c80	set_bit_on(mask,'q'-'a'); children[3] = 0x40007c80;
⋮		
children['t'-'a']	0x40007b80	set_bit_on(mask,'t'-'a'); children[4] = 0x40007b80;
children['v'-'a']	0x40007a80	set_bit_on(mask,'v'-'a'); children[5] = 0x40007a80;

Part V. wrap-up!

Take Note 4: Note that in both cases, we have "overloaded" the `.children` array of edge pointers to avoid the need to manage memory with `new`'s and `delete[]`'s. After `.finalize()` is finished, we must access `.children[]` as if it were **both** `.fixed_children[]` and `.overflow_children[]` of the `trie_Tnode` or `.children[]` of `trie_Mnode`; in either case the first array location may or may not be an edge for the letter a!

Take Note 5: For both `fake_trie_Tnode` and `fake_trie_Mnode` `.finalize()` implementations, you are moving pointer values around in the your `.children[]` array. **Be sure to nullify the pointer's old location when doing so!** For instance, when finalizing the q node in the previous example, you would want to write:

```
children[3] = children['q'-'a']
children['q'-'a'] = nullptr;
```

Of course, the `'q'` and `3` would probably be stored in looping variables of your logic. Nullify the old pointer locations correctly and *mostly likely*, the code you wrote for `collect_child_histogram<yourTrie>` will still work correctly (since you likely just traversed `.children[]` looking for non-null pointers....). Checking that you get identical histogram data from your original Trie object and your new `.finalize()`'d version is a reassuring sanity test before generating runtime results.