Project 3A: Lil' Trie Adventure

(alternative node definitions for smaller memory footprints)

Introduction

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.

For this assignment, please provide responses for the "deliverables" in this prompt. These will primarily consist of text in complete sentences, along with tables, charts, diagrams, and equations as needed or directed by the write-up. Your answers should be concise but complete. Diagrams or charts must be clearly labeled and readable; use of a software plotting tool (Excel, matplotlib, gnuplot, tikz, etc.) is required. Equations must be properly typeset (e.g., using the equation editor in your word processing software or an equation environment in LaTeX). The rubric for each deliverable is at the end of the write-up.

Part 1 — tooling up with child histograms

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, 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 (***W***) stored in the trie (recall a word exists 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 changes of your 3P trie node.

  We will refer to the result of `.size()` as the variable ***W*** in this project.

Once you have these node member functions implemented and compiling cleanly, change the zero in `main-test.cpp`'s `#if` line to a `1` and rebuild (`make run-test`). 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 project 3P trie after it has stored a dictionary of words. We will build a discrete count (or "cardinal") histogram of the number of children each node has in a trie.

A *discrete data histogram* records either the absolute number or the population percentage of a set of distinct attributes in a population. In this case the population is all the nodes in a trie, the attribute is the number of children the trie nodes have for a particular dictionary. For this task you will implement one or two more member functions:

- `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.
- `void trie::children_histogram( vector<size_t>& histo )` should call `void node::children_histogram( vector<size_t>& histo )` for all of the trie's children (do this recursively!). `node::children_histogram` should increment the `node_children()` valued index in `histo` by one:

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

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

> (You could choose to simply calculate the number of children in a node within `children_histogram`, in which case you won't need `node_children` for this project.)

Test these newly added functions with something like:

```
yourTrie trie;
vector<size_t> histo(LETTERS+1);
test_child_histogram( trie, words, histo );
```

Where `words` is a vector of words from one of the project dictionaries in the `wordlists/` directory of the repository (you've done similar word reading in 2A).

Compare the output results with the appropriately named files in `child_histogram/` directory.

The value at `histo[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). For instance, from the `child_histogram/words8.txt` file, we see none of the eight words have sub-word prefixes because there are eight nodes with zero children (nodes without children are always terminal nodes, and there are eight words in the dictionary generating the histogram). There are 53 nodes with only one child, two nodes with two children and one node with six children.

The "3P node" footprint

Let's now use the histogram we can collect to calculate the total memory footprint of the trie node structure presented in lecture and used in 3P as well as two other alternative trie designs.

> **Take Note 3**: We'll use the following constants (conveniently defined in `liltrie.h`) to avoid inconsistent alignment and word sizes of each student's individual machines: INTBYTES, PTRBYTES, CHARBYTES and BOOLBYTES which are the sizes of (you guessed it) a signed or unsigned integer, a pointer, a `char` data type and a `bool`. **Don't** generate analysis data using `sizeof()` and don't worry about uncounted bytes due to alignment issues. Just pretend you're targeting an ARM Cortex-M like processor :)

The 3P trie node has a Boolean flag `is_terminal` and `LETTERS=26`[1] pointers in a static array. Regardless of the number of children stored in a 3P trie node, its size is always

$$LnodeSize = BOOLBYTES + LETTERS \times PTRBYTES \tag{1}$$

If N is the number of nodes in the trie (the result of calling `liltrie.h`'s `nodecount()` on the trie or its histogram):

$$LnodeFootprint(N) = N \times LnodeSize() \tag{2}$$

[1] Conveniently defined in `liltrie.h`.

Part 2 — smaller footprints

We now consider two other node structures we expect will generate smaller memory footprints than the 3P trie node.

The T node

Our first alternative node structure stems from the observation that the majority of tree nodes actually have a small number of children. We take advantage of this by using a small fixed size 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.

```
struct trie_Tnode {
  static const int T = to_be_determined;
  bool is_terminal;
  char fixed_letters[T];
  typedef trie_Tnode* trie_Tnode_ptr;
  trie_Tnode_ptr fixed_children[T];
  trie_Tnode_ptr* overflow_children;
};
```

Here we have used a `typedef` statement to define the symbol `trie_Tnode_ptr` as an "alias" for `trie_Tnode*`. This may make reading and writing this code more straightforward for you, as otherwise you will be working with "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 choose not to implement `trie_Tnode` for this project, so hold off on writing code for it just now!

The size of this alternative node in memory depends on the value of T chosen for the structure and the total number of child edges stored in the node (C):

$$TnodeSize(C,T) = \begin{cases} BOOLBYTES + T \times CHARBYTES + \ T \times PTRBYTES + PTRBYTES & \text{if } C \leq T \\ \\ BOOLBYTES + T \times CHARBYTES + \ T \times PTRBYTES + PTRBYTES \\ + (LETTERS - T) \times PTRBYTES & \text{otherwise} \end{cases} \tag{3}$$

The additional $(LETTERS - T) \times PTRBYTES$) addend is because the array of pointers will be allocated only when $C > T$.

How much space can be saved with this `trie_Tnode` structure? It depends on T (of course) so the task is to determine an ideal T for a given language. You will complete this task later in the project. For now, let us suppose we have determined an ideal value of T to use for a particular language dictionary.

Again, if N is the number of nodes in the trie, we can 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 built with `trie_Tnode`s is

$$TnodeFootprint(N, T, s) = s \times TnodeSize(1, T) + (N - s) \times TnodeSize(LETTERS, T) \quad (4)$$

Take Note 4: A quick explanation: there are s nodes with T or less children. All these nodes will consume the same amount of memory (by the design of `trie_Tnode`). Their contribution to the footprint is $s \times TnodeSize(1)$. Likewise, there are $N - s$ with more than T children, again all of these nodes will consume the same amount of memory. Their contribution to the total footprint is $(N - s) \times nodeSize(LETTERS)$. We choose 1 and $LETTERS$ as convenient values that always happen to select smaller or larger version of the node regardless of Ts value. We could have used T and $T + 1$ to the same effect.

An even smaller trie node structure

Both the 3P trie node definition and the `trie_Tnode` definition over-allocate space for children. We can reduce the size of a trie node even more with some bit-twiddling and allocating a `children[]` array for only the number of children needed at 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. We'll assume $INTBYTES \geq 4$ and thus we have at least 32 bits. We'll use $LETTERS = 26$ of the Booleans to track if a particular letter has an edge in the node and one more bit to store the `is_terminal` property of the node.

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 the `trie_Mnode` 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 a `trie_Mnode` is simply:

$$MnodeSize(C) = INTBYTES + (C + 1) \times PTRBYTES \quad (5)$$

Recall that the structure has a "built-in" pointer (`children`), and it will point to C `trie_Mnode_ptrs` in memory, hence the $C + 1$ number of pointers, not just C.

The total footprint of an M node trie can be calculated from its histogram of child counts, H:

$$MnodeFootprint(H) = \sum_{c \in H} MnodeSize(c) \times H[c] \quad (6)$$

Part 3 — empirical evidence

Find an ideal *T* for the `trie_Tnode`

Using `main.cpp`, `ideal_tnode_histogram`, `trie.size()`, the `dictionary.txt` provided in the git repository for this project and equation (4) calculate how the total `trie_Tnode` trie footprint, normalized by the number of words in the trie, varies with $1 \leq T < LETTERS$. (`main.cpp` is a minimal program, add code to it for these calculations, use `make run-main` to build your program.)

Use `ideal_tnode_histogram` as such:

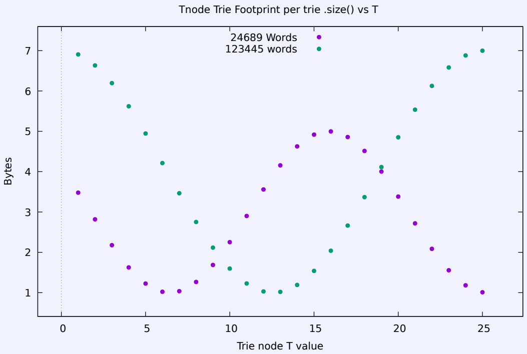
```
yourTrie trie;
vector<size_t> histo(LETTERS+1);
ideal_tnode_histogram( trie, dict, 30000, histo );
```

This will value the `histo` vector with a random sample of 30,000 words from `dict`, a `vector<string>` you populated with `dictionary.txt`.

Take Note 5: To be clear: you are not implementing any **new nodes** (yet). You are using your pre-existing trie code to store words from a dictionary and collecting child histogram data from it. From this histogram we can do a fair bit of analysis without needing to implement any new data structures!

Deliverable 1

a. Show a properly labeled **scatterplot** with at least five sets of calculations for the $(T, footprint/W)$ relationship. Use distinct symbols for each series and put T on the independent axis. There are a 172,823 words in `dictionary.txt`, show **at least four** series, all on one graph, for an evenly sampled number of words (such $W = 30K, 60K, 90K, 120K, 150K$).
Your plot (with just two series) might look as such:



(Though we hope your data is more consistent than our made up values!)

b. Supported by the evidence in your scatterplot, what value(s) of T would be a good choice to minimize the total "T node" trie footprint? If you don't feel your scatterplot supports such a conclusion, you need more experiments or there is something amiss in your code.

Now, given the ideal T you have discovered for `dictionary.txt`, we want to investigate how the footprints of the 3P node, an ideal Tnode and the Mnode compare for variously sized word sets.

Change your code in `main.cpp` to run `collect_child_histogram` with all the words from `dictionary.txt` at least 100 times.

```
for( size_t i=0; i<AT_LEAST_100; ++i ) {
    yourTrie trie;
    vector<size_t> histo(LETTERS+1);
    collect_child_histogram( trie, dict, histo );
    // calculate three data points
}
```

A random sample of the dictionary words will be used for each invocation. The result will be a wide range of word counts over all the runs. For each run of `collect_child_histogram`, determine the number of words stored in the trie (W) with `trie::size()`, and show in a scatterplot the three coordinate pairs:

$$(W, LnodeFootprint(N)) \quad (W, TnodeFootprint(N, T, s)) \quad (W, MnodeFootprint(H)) \qquad (7)$$

Where:

- The dependent values are from equations (2,4,6) respectively
- T is now a constant value, your ideal T
- H is the child histogram of the trie, valued by `collect_child_histogram`
- `N = nodecount(H) /** defined in liltrie.h */`
- $s = \sum_{i=0}^{T} H[i]$

Deliverable 2
Show a properly labeled **scatterplot** of these coordinate pairs (7). Use three distinct symbols in your plot, treating each trie node definition as a plot series.

Part 4 — the adventure

We have investigated how the overall memory footprint of a trie depends on the core trie node data structure. Your results should provide convincing evidence that substantial memory savings can be had. But are these savings in memory for free? We shouldn't think so, as both of the proposed alternative node structures will require more complicated algorithms for word insertion and tree traversals (eg: `contains()`, prefix finding and `trie::size()`, to name a few).

Consider this: with 3P node tries there is immediate access to the child pointer of a letter. It is simply `.children[letter-'a']`, or $\mathcal{O}(1)$ for those of you who are counting :)

With the `trie_Tnode`, we must first look in `fixed_letters` one letter at a time for a particular edge and if it isn't found, we must then consider `overflow_children`. That sounds like it'll be at least two loops at every node of a search. Performance penalties associated with smaller nodes seem a legitimate concern.

Your task for the last part of this analysis is to implement **just one** of the two alternative node structures and compare its runtime performance against the 3P trie.

Take Note 6: **First**, don't trash your 3P trie! You'll need it for the runtime tests later in the project..
Second, if you add new header or source files to the project (alternatively you could just keep stickin' everything in `trie.h` and `trie.cpp`) you'll need to add their filenames alongside the `trie.h` and `trie.cpp` entries in `CMakeLists.txt` and possibly re-initialize your `build` directory with the `cmake` command.
Lastly, once you have your lil'trie class working compiling cleanly and working with the `time_trials<Y,O>` function (described below) — you should be ready for generating data. `time_trials<Y,O>` does very thorough job of making sure each trie is returning the anticipated results from `contains` tests. You shouldn't have to implement `trie::size` or `trie::children_histogram` for the time trials.

You have **four choices**, which seems strange because there are only two node structures to choose from, let us explain:

Choice: an on the fly node

You can write a brand new `trie` class and manage the 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 managing memory as needed in `.insert()`.

If you're just itchin' to implement one more data structure from scratch, follow the appropriate link:
Tnode "on the fly" instructions
Mnode "on the fly" instructions

Choice: nodes using `.finalize()`

You can also copy your current trie class definition[2] and only have to implement or rework the `.finalize()` and `.contains()` logic of your new class. The advantage to this approach is that it doesn't require any memory management and you won't have to debug `.insert()`, constructor or destructor logic for a brand new class.

If this sounds more to your liking, follow the appropriate link:
Tnode "finalize" instructions
Mnode "finalize" instructions

[2] Or, if you're into the whole class hierarchy approach, you could inherit from your original trie class.

Part 5 — runtime performance

When you have your lil'trie class compiling cleanly, 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, its `.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 trie, the `O` trie being your *other* lil'trie class implementation), a dictionary of words and an optional Boolean `finalize` parameter. The default value for `finalize` is true.

`time_trials<Y,O>` will take a random number of words from the dictionary, half will go into a trie, `.finalize()` is called on the trie, and then the trie is searched for all the words (half the queries will fail, half must succeed). The elapsed time is measured during the searching phase of the experiment (post `.finalize()`).

Take Note 7: If you have implemented an "on the fly" node, you can provide `finalize=false` and the timing results will reflect the total "load time" as well (all the `.insert()` calls). If you have implemented a "finalized" trie, you should use the default value for the `finalize` argument.

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.count() << " " << secs.second.count() << endl;
```

Take Note 8: **Help! My machine is a beast!** If your machine is so fast that `time_trials<Y,O>` doesn't generate reliable, repeatable run times, take a look inside `lil_trie.h`. Increment the `MY_MACHINE_IS_A_BEAST` counter until you get reliable, repeatable timing results.

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 lil'trie class:

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

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 includes the original 3P trie), which would you prefer to use in a real world application? What development or runtime factors might influence your choice?
Consider, for instance, the following scenarios:

a. A desktop editor that uses a trie to provide word completion hints.
b. A small resource-constrained (memory, storage space) customer service kiosk that uses a trie, populated by customer input, for completion hints to many different types of information: street addresses, ZIP codes, nine digit phone numbers, email addresses, ...

Deliverable 5
Finally, copy and paste a nicely formatted version of the `.contains()` logic of your lil'trie class 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.

Rubrics

Rubric for Deliverable 1a

S Scatterplot has a title, correctly labeled axes. Plot shows results over $1 \leq T < 26$ for five differently sized tries over the range $[1\text{K}, 172\text{K}]$. Data points **should not be connected with lines** (this is discrete data, there is no footprint for Tnode tries with $T = 13\frac{1}{2}$).

N Plot does not meet S criteria, has missing data, is poorly formatted or has poorly chosen axis ranges or scales.

U No plot

Rubric for Deliverable 1b

S Correct response written clearly and in agreement with their plot.

N A response that does not meet S criteria.

U No response.

Rubric for Deliverable 2

S Scatterplot has a title, correctly labeled axes. Plot shows results over a wide range of words. Data points **may be connected with lines** (it is reasonable to extrapolate between two points). Different symbols for each of the three trie node types must be used to denote actual data points.

N Plot does not meet S criteria, has missing data, is poorly formatted or has poorly chosen axis ranges or scales.

U No plot

Rubric for Deliverable 3

S Scatterplot has a title, correctly labeled axes. Plot shows results for a wide range of independent axis values. Data points **should not be connected with lines**.

N Plot does not meet S criteria, has missing data, does not support conclusions, is poorly formatted or has poorly chosen axis ranges or scales.

U No plot

Rubric for Deliverable 4

S Response addresses the question, is correct and supported by features of the data or scatterplot(s).

N Same as S, but there are errors/misunderstandings in the discussion.

U No discussion

Rubric for Deliverable 5

S Response is the student's Mnode or Tnode `.contains()` function **and any helper functions it might call**. Indentation of programming blocks and the use of `{}`s is consistent throughout. A fixed with font is used.

N Insufficient, incorrect, or too much source provided. Formatting does not meet S standards. A variable width or difficult to read font is used.

U No source provided.

Appendix: Tnode tries "on the fly"

For "on the fly" implementations, you will need to implement the following for your node class:

i. Constructor and destructor

ii. `.insert()`

iii. `.contains()`

iv. If you run into development issues and wish to use `test_node_functions` for testing, you'll need to implement the `.size()` member function.

v. If you wish to use `collect_child_histogram` for testing or debug, you'll need both `.size()` and `child_histogram` member functions.

vi. For "on the fly" implementations, you want to avoid memory leaks and corrupted data. Fortunately our runtime needs for your chosen lil'trie class are simple: allocate memory as needed in `.insert()` and make sure the 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 "delete" at the end of these declarations for in your class definition. So (for example) your node class called `trie_Xnode` should have the following prototypes:

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

Providing these will prevent C++ from using its default versions. You should do the same for `trie_Tnode` on the fly implementations.

Here is a quick synopsis of the `trie_Tnode` structure and its workings:

```
struct trie_Tnode {
  static const int T = to_be_determined;
  bool is_terminal;
  char fixed_letters[T];
  typedef trie_Tnode* trie_Tnode_ptr;
  trie_Tnode_ptr fixed_children[T];
  trie_Tnode_ptr* overflow_children;
};
```

1. The `fixed_letters[i]` holds the character value for the edge pointer stored at `fixed_children[i]`, so `fixed_letters` should be initialized to a *sentinel value* such as zero to indicate "no edge."

2. `overflow_children` as well as `fixed_children` elements should be initialized to `nullptr`.

3. In the `insert()` method, if `fixed_letters[T-1]` is not the sentinel value, `overflow_children` needs to be allocated

```
overflow_children = new trie_Tnode_ptr[LETTERS-T];
```

and the new edge placed in `overflow_children` according to (11).

4. The location of a letter edge pointer depends on the edge character ϵ being sought and the contents of `fixed_letters`:

$$\epsilon \text{ location} = \begin{cases} \mathit{fixed\_children}[i] & \text{if } \epsilon = \mathit{fixed\_letters}[i] \\ \mathit{overflow\_children}[j] & \text{otherwise, where } j = \epsilon - \text{'a'} - \alpha \end{cases} \tag{8}$$

where `'a'` is the ASCII code for letter a and α is the number of characters in `fixed_letters` that are less than ϵ:

$$\alpha = \sum_{t \in \mathit{fixed\_letters}} (\epsilon < t)$$

We borrow some CS semantics in the last equation and think of $\epsilon < t$ as being 1 if true, 0 if false.

> **Take Note 9**: **Important!** this location equation (11) for any letter edge must be used not only when searching for a letter edge, but also when `insert`'ing new edges into `overflow_children`. Calculate α *while* you traverse `fixed_letters`, so the value is known when you need it for `overflow_children` operations!

The `insert` method

"On the fly" Tnode insertion should do the following (assuming the next letter value to be considered is ϵ):

a. Scan through `fixed_letters` for the value ϵ, accumulating α as you go.

b. If ϵ is found in `fixed_letters` at index `i`, the pre-existing edge for ϵ is at `fixed_children[i]`.

c. Otherwise, if `fixed_letters` is not full, store ϵ and a newly allocated edge pointer at the first empty slots of `fixed_letters` and `fixed_children`. If `fixed_letters` is full, store a newly allocated edge pointer at index j of `overflow_children`.

Appendix: Mnode tries "on the fly"

For "on the fly" implementations, you will need to implement the following for your node class:

i. Constructor and destructor

ii. `.insert()`

iii. `.contains()`

iv. If you run into development issues and wish to use `test_node_functions` for testing, you'll need to implement the `.size()` member function.

v. If you wish to use `collect_child_histogram` for testing or debug, you'll need both `.size()` and `child_histogram` member functions.

vi. For "on the fly" implementations, you want to avoid memory leaks and corrupted data. Fortunately our runtime needs for your chosen lil'trie class are simple: allocate memory as needed in `.insert()` and make sure the 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 "delete" at the end of these declarations for in your class definition. So (for example) your node class called `trie_Xnode` should have the following prototypes:

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

Providing these will prevent C++ from using its default versions. You should do the same for `trie_Tnode` on the fly implementations.

Here is a quick synopsis of the `trie_Mnode` structure and its workings. You should be familar with the terminology and concepts in the bitmask primer before embarking on this part of the project.

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

1. Bits 0–25 will be "up" or "on" if the cooresponding letter `'a'–'z'` has an edge out of the node. The edge pointers are kept in `children`.

2. Bit 31 is used to store the `is_terminal` property of the node.

3. The Mnode goal is to minimize a trie footprint, so `children` will always be perfectly sized for

$$\alpha = \sum_{i=0}^{i<LETTERS} \text{bit\_on(mask,i)} \tag{9}$$

children at a node.

4. Letter ϵ's edge pointer in `children` depends on how many alphabetical order letters precede ϵ in the node:

$$\delta = \epsilon \text{ index} = \sum_{i=0}^{i<\epsilon-\text{'a'}} \text{bit\_on(mask,i)} \tag{10}$$

where `'a'` is the ASCII code for letter a.

For example: if ϵ = 's' is the smallest letter with an edge in the node, its edge pointer index in `children` is 0. The letter edge index of ϵ = 's' in a node also containing edges for `'g'`, `'q'`, `'u'` and `'w'` is 2, the edge for `'w'` is 3.

5. On any insertion of a new letter edge, the `children` array must be grown by one (we suppose the letter value is again ϵ:

   i. Determine the initial number of children in the node (α, eqn 9) and allocate a new `trie_Mnode_ptr` type array of size $\alpha + 1$.

   ii. Turn the appropriate bit for ϵ on in the mask (`set_bit_on`) and find ϵ's index in the newly allocated array (12).

   iii. If $\alpha > 0$, copy any pre-existing edge pointers for indices $< \delta$ from `children` to the same index in the newly allocated array.

   iv. Allocate a `new trie_Mnode` at index δ in the new array.

   v. If $\delta < \alpha$, copy any pre-existing edge pointers for indices $\delta \leq i < \alpha$ to index $i + 1$ in the new array

   vi. `delete children`, and store the new array at `children`.

Appendix: Tnode tries using `finalize`

For `finalize` implementations, you should make a copy of your 3P trie class as a starting point. Here are the changes you should expect to make to your lil'trie class:

i. You will add some data members to the class, they won't be used until the testing harness loads all the words into the trie and calls `finalize`.

ii. Of course, you'll need to implement a `finalize` function (more details soon).

iii. You'll need to change `contains` function logic.

iv. If you run into development issues and wish to use `test_node_functions` for testing, you'll need to implement the `.size()` member function.

v. If you wish to use `collect_child_histogram` for testing or debug, you'll need both `.size()` and `child_histogram` member functions.

First, a quick recap of the memory saving strategy of this trie. An "on the fly" Tnode would have this structure:

```
struct trie_Tnode {
  static const int T = to_be_determined;
  bool is_terminal;
  char fixed_letters[T];
  typedef trie_Tnode* trie_Tnode_ptr;
  trie_Tnode_ptr fixed_children[T];
  trie_Tnode_ptr* overflow_children;
};
```

where:

1. The `fixed_letters[i]` holds the character value for the edge pointer stored at `fixed_children[i]`, so `fixed_letters` should be initialized to a *sentinel value* such as zero to indicate "no edge."
2. `overflow_children` as well as `fixed_children` elements should be initialized to `nullptr`.
3. In the `insert()` method, if `fixed_letters[T-1]` is not the sentinel value, `overflow_children` needs to be allocated

   ```
   overflow_children = new trie_Tnode_ptr[LETTERS-T];
   ```

   and the new edge placed in `overflow_children` according to (11).
4. The location of a letter edge pointer depends on the edge character ϵ being sought and the contents of `fixed_letters`:

   $$\epsilon \text{ location} = \begin{cases} \textit{fixed\_children}[i] & \text{if } \epsilon = \textit{fixed\_letters}[i] \\ \textit{overflow\_children}[j] & \text{otherwise, where } j = \epsilon - \text{'a'} - \alpha \end{cases} \tag{11}$$

   where `'a'` is the ASCII code for letter a and α is the number of characters in `fixed_letters` that are less than ϵ:

   $$\alpha = \sum_{t \in \textit{fixed\_letters}} (\epsilon < t)$$

> We borrow some CS semantics in the last equation and think of $\epsilon < t$ as being 1 if true, 0 if false.

We won't have to write an `insert` method, as we'll use the same code as the 3P trie. Our `finalize` method simply has to record the character values associated with the first *T* edges in `children` into a `fixed_letters` array.

We need add only two new data members to our Tnode class: the `static const unsigned T` value and the `char fixed_letters[T]` array. You will of course set *T* to be your chosen optimal constant determined in ***Deliverable 1b***. The `fixed_letters` array should have all its elements valued to a **sentinel value**, some non-letter ASCII code such as 0, in the constructor for the node.

The `finalize` method

In the `finalize` method, put character values of the first *T* edges into `fixed_letters`.

```
    unsigned c=0;
    for( unsigned i=0; i<LETTERS && c < T; ++i ) {
        if( children[i] != nullptr ) {
            fixed_letters[c++] = 'a' + i;
        }
    }
}
```

The `contains` method

Recall that in the 3P trie, we know immediately where a letter edge might be. If we are looking for a `'q'` edge, it is at `children['q'-'a']`. The `finalize` logic didn't move any pointers around in `children`, so in theory we could do the same edge location calculation in our "fake" Tnode. But this would make Tnodes appear to perform just as well as 3P trie nodes — and our instinct says that can't be.

We must be sure to mimic the work done in a true Tnode implementation in our `contains` logic.

If we are seeking an edge for the character ϵ with ASCII code `e`:

a. For each element of `fixed_letters`: first check if it is equal to ϵ, if it is the edge pointer is at `children[e-'a']`. Its important to stop this search if you encounter the sentinel value, as this means there are $< T$ edges in the node and an edge for ϵ will not be found.

b. If ϵ is not found in `fixed_letters` and `fixed_letters[T-1]` is **not** the sentinel value, inspect `children[e-'a']`. If it is `nullptr` an edge for ϵ does not exist in the node, otherwise it does and you've found its value.[3]

[3] We know, there are two comparisons at each `fixed_letters` element, the second one involved in the calculation of α — we've chosen to not require the modeling of this step.

Appendix: Mnode tries using `finalize`

For `finalize` implementations, you should make a copy of your 3P trie class as a starting point. Here are the changes you should expect to make to your lil'trie class:

i. You will add some data members to the class, they won't be used until the testing harness loads all the words into the trie and calls `finalize`.

ii. Of course, you'll need to implement a `finalize` function (more details soon).

iii. You'll need to change `contains` function logic.

iv. If you run into development issues and wish to use `test_node_functions` for testing, you'll need to implement the `.size()` member function.

v. If you wish to use `collect_child_histogram` for testing or debug, you'll need both `.size()` and `child_histogram` member functions.

You should be familar with the terminology and concepts in the bitmask primer before embarking on this part of the project.

First, a quick recap of the memory saving strategy of this trie. An Mnode trie uses the minimum amount of memory for a trie node. An integer `mask` member variable uses bits as Boolean values to keep track of letters associated with child edges and whether or not the node is "terminal" (ends a word). We'll mimick the use of a minimally sized `children` array by packing all the edge pointers created during "word loading" to the front of `children` in the `finalize` logic.

1. Bits 0–25 will be "up" or "on" if the cooresponding letter `'a'`–`'z'` has an edge out of the node. The edge pointers are kept in `children`.
2. Bit 31 is used to store the `is_terminal` property of the node.
3. Letter ϵ's edge pointer in `children` depends on how many alphabetical order letters precede ϵ in the node:

   $$\delta = \epsilon \text{ index} = \sum_{i=0}^{i<\epsilon-\text{'a'}} \text{bit\_on(mask,i)} \tag{12}$$

   where `'a'` is the ASCII code for letter a.

   For example: if ϵ = 's' is the smallest letter with an edge in the node, its edge pointer index in `children` is 0. The letter edge index of ϵ = 's' in a node also containing edges for `'g'`, `'q'`, `'u'` and `'w'` is 2, the edge for `'w'` is 3.

We need add only one new data member to our Mnode class: an integer `mask`. It should be initialized to zero in the constructor.

The `finalize` method

In the `finalize` method, you'll need to:

a. Set bit index **31** "on" or "up" in the node's `mask` if `is_terminal` is true.

b. Compress the non-`nullptr` values in `children` to the front of `children`, and record the ones that exist in `mask`:

```
if( is_terminal ) set_bit_on(mask,31);
unsigned c=0;
for( unsigned i=0; i<LETTERS; ++i ) {
    if( children[i] != nullptr ) {
        set_bit_on(mask,i);
        children[c++] = children[i];
    }
}
while( c < LETTERS ) children[c++] = nullptr;
```

The `contains` method

The `contains` method should first check if the $\epsilon -$ 'a' bit of `mask` is "on" or "up" (where ϵ is the letter value of the next character in the word). If the flag is up, a child edge exists and is at the δ index of `chidren` (equation 12).

Appendix: A quick primer on bitmasks and their manipulation

We often refer to the bits in a bitmask as “flags": a 1 represents an “up” flag, representing `true`, whereas 0 represents `false`. In this project, an “up" flag for some character tells us the `children` array will have a pointer for this character edge. 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

A bitmask in an Mnode that terminated a word and had child edges for the letters `'b'`, `'f'` and `'p'` would have a `mask` value of `0x80008022` in a debugger print out, the leftmost `0x8` nibble is the “up" `is_terminal` flag, the last `0x2` nibble is the flag for character `'b'`. The `children` data member would hold three `trie_Mnode` pointers, the first for the `'b'` edge, the second for the `'f'` edge, the last for the `'p'` edge.

To manipulate the bit values of an integer mask, you can use the following functions found in `bitwiseops.h`:

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

// bit is 0 to 31, sets the bit to "on"
static inline void set_bit_on( int& mask, unsigned bit )
{ mask |= (1<<bit); }

/***
  * You shouldn't need to turn bits in an Mnode mask "off" for this project,
  * but in case you are wondering...
  */
// bit is 0 to 31, sets the bit to "off"
static inline void set_bit_off( int& mask, unsigned bit )
{ mask ^= (1<<bit); }
```