# Trie Adventure

## Node Definitions For A Smaller Footprint

Originally Developed by
Keith Hellman

Heavily Modified by
Anastasia Sokol

Updated on June 2, 2024

## Introduction

We have studied the abstract trie data structure, a specialized tree designed for the quick search of sequenced data such as words, and we also implemented a trie in the programming project. Both the lecture and project version store a pointer to each of the 26 possible subsequent lowercase letters in *each node*, which is *208 bytes* on most machines, and means that storing the word ”algorithms” takes almost two kilobytes!

In this project you will explore some specialized node structures to decrease the memory footprint of the trie structure. As you go, write responses to the deliverables along with any properly formatted graphs or equations that the box asks for. This project has a longer prompt that some of the past assignments, but most of the content is there to guide you through the project, and the deliverables are equivalent to – if not less than – other analysis projects in this class.

**tl;dr** This project is about decreasing the memory footprint of the trie class. Provide the deliverables. *It is not as much work as it may seem.* If you get stuck or have questions do not hesitate to reach out for help after class or in office hours. Good luck!

## The Default Trie

The default trie is just the structure that was explored in lecture and in the trie programming project. To reduce the amount of redundant code you have to write, the `Trie` class defined in `trie.hpp` is written to work with any type of node that fulfils a specific interface that is laid out in detail later on. A node that fulfils that interface using the lecture method is already implemented in `nodes/node.hpp`. This interface style is also known as dependency injection and is important for idiomatic c++.

The first thing we need to do is gather some data about how the trie class is being used. This can provide insight into how the class can be optimized, as well allow us to solve some future computations. To do this you will need to implement the methods below in the `StatisticsNode` structure defined at `nodes/statistics.hpp`, then use the `StatisticsTrie` class defined at `statisticstrie.hpp` to gather the data that you need.

- `std::size_t StatisticsNode::size()`

  This function will return all of the terminal nodes at-or-below the current node. In other words, the number of words stored at-or-below the current node. Consider the recursive nature of this function when writing your implementation.

- `std::size_t StatisticsNode::count_children()`

  This function will return the number of ”existing” children in the current node. For the default trie, this means the number of pointers in the array `children` that are not `nullptr`.

- `std::size_t StatisticsNode::count_descendants()`

  This function will return the number of children *at-or-below* the current node. This is equivalent to the descendants below the current node plus the children of the current node.

- `void StatisticsNode::record_histogram(std::array<std::size_t, 26> &histogram)`

  Adds to a histogram the nodes at-or-below the current node. To implement this increment the value representing the number of children at the current node, for example `++histogram[count_children()]` then call `record_histogram` for all of the non-`nullptr` children of the current node.

After implementing these methods you can run the statistics tests to verify your implementation, then use **SOMETHING** to record the statistics for later use. Finally, add the deliverable to your analysis document.

**Deliverable One** Plot three histograms you got from your statistics recordings as bar charts. Discuss the general shape of the data as well as any major differences between the three charts.

**A Bitmasked Trie**

**A Deep-Node Trie**

**Empirical Measurements**

**Rubric**

**Language Feature Explanations**