Trie | Insert & Search Data Structure

Group Members: Adrian Sicaju Ruiz, Daniel Maiello, Kenny Sullivan

Introduction

For our group project, we decided to implement the Data Structure, Trie. A trie is a type of tree data structure that is used for locating specific keys in a set. These keys are often strings with links between nodes defined not by just the set but its individual characters. A prime example of a Trie Data Structure is a Google Search bar that will attempt to autocomplete once a word has been typed and based on the word, it will attempt to autocomplete the next character to complete the word and so on. Our goal was to

1.) Store words in an input text file and treat it like a dictionary and also count the number of words (node) that are stored in the dictionary.

2.) Implement an Insert/Search method that will return and show the count of words.

3.) Implement a GUI that will model a Google search bar and display a count of repetitions on a word in a dictionary.

4.) And finally generate a DOT file for visualization.

Methods

Tries are a useful data structure that is based on a prefix of a string. Tries represent the Retrieval of data. The prefix of the string is considered strictly at the beginning of a string. For example, if you were to type in “abacaba” into a search bar it would visually look like so:

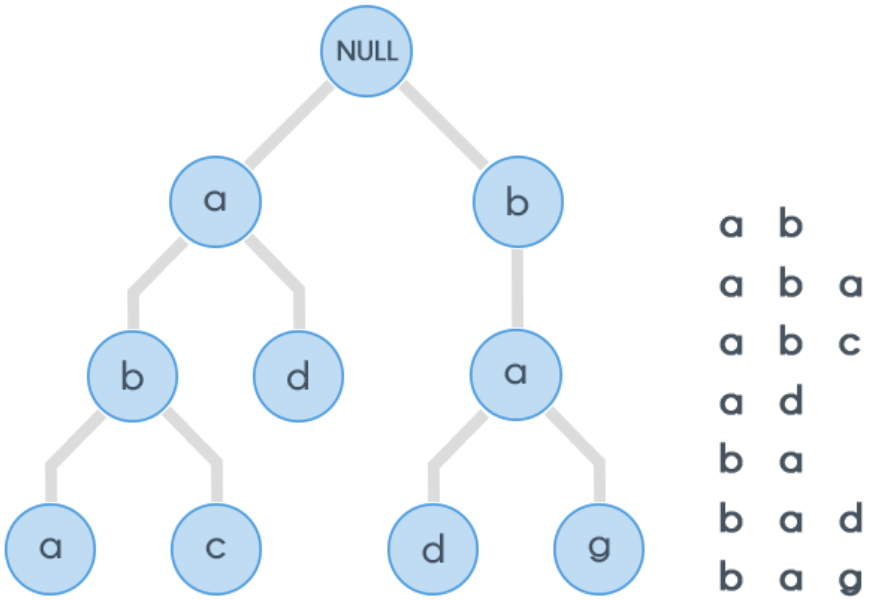


Fig. 1

Tries are used for a group of strings. When multiple strings are fed into the trie data structure, we can solve a variety of problems. If we considered an English dictionary with a single character string, to regularly solve this problem without a trie we would match the prefix of the given string with every word in the dictionary which is an expensive process that would take a long time to compute. With tries, we could process each entry and search the length of the longest prefix within the existing words of the dictionary. A trie consists of a special node which is the root node that contains 26 edges for each letter in the alphabet. So any insertion string that goes into the data structure starts from the root node and are direct children of length one, which is the root node. Then, all prefixes with length 2 become children of the previous existing node and so on.

To start the process of the project we first researched the data structure itself. After reading some articles and referencing some helpful videos we were able to conclude that tries were very similar to linked lists, heaps, binary search trees and basically every other tree based data structure. Which led us to the files from our previously completed labs. Using much of the code we created in labs, such as the node header and .cpp files, allowed for us to quickly and accurately implement much of the base for the trie data structure. Referencing the insert and such methods as part of multiple structures we implemented also provided us with a stable starting base for our implementation. Having previously worked with and solved many of the similar

problems this topic presented us, our group had a firm understanding and competence of how the code will, and should, run during our implementation. For example, when implementing the insert and search work done in the linked list lab, binary search tree and heaps was useful for referencing these functions as we had created them here and all of these data structures have similarities when creating the nodes and overall trees for them.

When researching the trie data structure we found that it is a fairly simple data structure consisting of nodes holding character value that form an efficient method for finding keys in large sets of data. This provides tries with plenty of functionality across all aspects of the digital world. Applications for the data structure, originally with the term being coined by Edward Fredkin, include predictive text softwares and autocompletions for example, search engines like Google use tries and they can also be implemented to correct your spelling of texts. The best, worst, and average case run times for tries are all O(n) where n is the length of the string that is being inserted or searched for because to search for or insert a key the trie must traverse all the way to the last child, leaf node, to reach the end of the desired operation. While Tries are certainly efficient with speed of operations they do come with some pitfalls. The most glaring being the trouble with memory. While our technology advances and so too does our memory and storage capacities are still extensively large in the space the structure takes up. This is because for every node created in the Trie an array of characters must be initialized, pointing to null to begin with, containing the size of the dictionary the user wishes to use. If the creator is using the ASCII library then every node will have 128 pointers to null to be stored; even more characters if it is the extended ASCII library. There are methods to make tries more memory efficient such as creating bitwise tries or compressing the trie where it becomes a more static data structure.

Radix trees are also a way to optimize the space of a trie by compacting the tree. This is done by searching the tree for nodes that only have one child node attached to them. When these nodes are found the child and current node are compressed back into its parent node allowing for less overall nodes spanning away from the root node of the trie.

Implementation

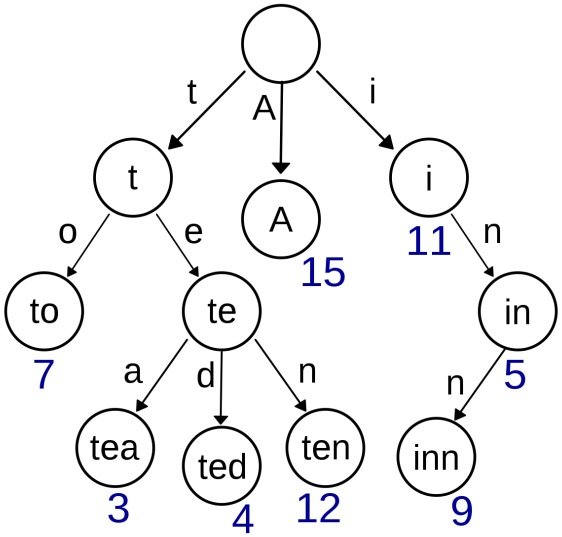
To begin our implementation we created all files and code on CS50 as it was the most universally available environment for collaboration between our group members. This was also decided because of the familiarity all members had with the environment. Then came the process of creating simple header files for the nodes and trie structure itself.

One of the main functions of this data structure is the Insertion process. The way this works is, when given a word, for example “pot”, the trie starts at the root node and examines each of the children that the root node has. When examining each of these children, it is checking to see if the current node in the tree already has a child with the key “h”. If there is no child with the existing key, that child is then created, and that child becomes the current node. Then the children nodes of that node are examined the same way, looking for the child with the key “e”.

Once the algorithm has either fully created all nodes required for that tree or sees that word already exists in the tree, it returns back to the root node and then completes the process.

Another main function of the Trie implementation is the search function. The search process is very similar to the insertion process. The code is first given a key, or piece of data to search for in the tree. The function then starts at the root node, and checks all of its children nodes to see if a child with the first element of the input key exists. If not, the child node is created, but if the child node does exist, the program moves to that node and then starts again. The program searches for the following elements in the input key. If the last element of the key is found, the program then checks to see if that node is marked as the end of a word, or a leaf node. If the node is found as a leaf node, then the function will return true as that exact word was in the Trie, as well as the count recorded for how many times that word is stored in the Trie. If the node is found, but not as a leaf node, then the function will return false because, although the word in its entirety may exist, the key was not inserted as its own key so it can not be referred to as a complete word. If the program comes to a point where it is at an element in the key (not at the end) and it has no other child nodes to examine, it will return false because the requested word was not found in the trie.

A minor, but important, process of trie implementation is the counter. The counter process of a Trie is important because it is the process that tells you how many times a key is inserted or stored in a trie. For example, if the key “hello” were inserted into the trie, it would be given an initial count of 1, because that key is stored only once in the trie. If the same key were given to the Trie to be inserted, the trie would use the insert function process, but it would then get to the last element in the key and realize that it does not need to insert any more elements. Instead of returning back and finishing its process, the counter would increment by 1. By doing this, whenever a key is searched for in a trie, it will show that the key appears either multiple times, or a single time in the trie.



An extra function that we added in was to use the tries functionality to autocomplete prefixes given to a dictionary. To do this we implemented a function that accepts a desired prefix to autocomplete for and checks whether it exists in the dictionary or not. Then it calls a helper function that then searches for and outputs all strings in the dictionary that have a matching prefix that the user could be looking for recursively.

Contributions

Kenny: When working on this project, my main contribution was writing and debugging the main trie functions and operations. One of the biggest problems we ran into was accessing the data of specific nodes. Dan and myself had originally implemented the code using minimal nodes and instead created a Trie of Tries. While this was originally successful, it created problems. The solution for this problem was the reimplement the trie class, but prioritize the creation and pointers of nodes. By doing this, we were able to clean up the way the code looked, as well as being able to more dynamically access parts of the node class that we needed. I also worked heavily on the readFile function and the Insert function. When implementing these, we had little difficulty once we were able to directly access the data of Nodes.

Dan: For this project, I mainly worked on the code and debugging for all of the node files, trie files, and main files. I worked mainly on implementing insert, search and both autocomplete functions such as how to recursively call through the trie to find the word with the matching prefix being queried. Challenges I faced with the code were mainly within the autocomplete function and print function. The autocomplete function I was able to write and compile correctly but was stopping at the first conditional statement for checking child nodes when run. Kenny was able to help me debug by figuring out the boolean we were testing for was reversed and we remedied the problem. As well as partially contributing to Kenny solving our counter problem for searching repeated words. Other than coding based contributions I researched much of the data structure and wrote the methods portion of this report. While pushing revised editions of the code such as output corrections and comments to the GitHub repository I wrote the “readme” file for which a user can expect to work with the code provided in the repository.

Adrian: My contribution to this project was to take charge in being able to implement the code itself into a functioning GUI that represented what our code would theoretically do. What I initially did was use visual studio community and attempted to convert the C++ code into C# in an attempt for it to be compatible with the Windows Form GUI. One of my biggest problems was trying to get it to properly convert to C#. I also took part in doing research on the Trie method, learning how to write the algorithm, the dot file, fully understanding and reiterating what Trie’s can do while also cleaning and adding additional input into our cpp and header files. I also contributed to creating the Github repository, uploading files, sorting our report and powerpoint and also research the basis of what Trie data structure is and how we use it in real world applications.

List of References

S. Singh, “Applications of trie data structure,” *OpenGenus IQ: Computing Expertise & Legacy*, 02-Apr-2020. [Online]. Available: https://iq.opengenus.org/applications-of-trie/. [Accessed: 09-Dec-2021].

“Trie (keyword tree) tutorials & notes: Data structures,” *HackerEarth*. [Online]. Available: https://www.hackerearth.com/practice/data-structures/advanced-data-structures/trie-keyword-tree/tutorial/. [Accessed: 09-Dec-2021].

“Visualize C++ data structures using Graphviz and the dot language,” *Nirav's Blog*, 08-Dec-2019. [Online]. Available: https://nirav.com.np/2019/12/08/visualize-c-data-structures-using-graphviz-and-dot-language.html. [Accessed: 09-Dec-2021].

“Trie,” *Wikipedia*, 07-Dec-2021. [Online]. Available: https://en.wikipedia.org/wiki/Trie#Autocomplete. [Accessed: 09-Dec-2021].

G. Sen, “What is the Trie data structure and where do you use it?,” *YouTube*, 09-May-2007. [Online]. Available: https://www.youtube.com/?gl=DE. [Accessed: 09-Dec-2021].

U. Agarwal, “Trie Data Structure,” *Medium*, 24-Apr-2020. [Online]. Available: https://medium.com/underrated-data-structures-and-algorithms/trie-data-structure-fd9a2aa6fcb8. [Accessed: 09-Dec-2021].

“Drawing graphs with - graphviz - graph visualization software.” [Online]. Available: https://www.graphviz.org/pdf/dotguide.pdf. [Accessed: 09-Dec-2021].

S. Singh, “Applications of trie data structure,” *OpenGenus IQ: Computing Expertise & Legacy*, 02-Apr-2020. [Online]. Available: https://iq.opengenus.org/applications-of-trie/. [Accessed: 09-Dec-2021].

“C# graphical user interface tutorial,” *C# Tutorial , C# Help , C# Source Code*. [Online]. Available: http://csharp.net-informations.com/gui/gui_tutorial.htm. [Accessed: 09-Dec-2021].

“DOT File Visualizer,” *Graphviz online*. [Online]. Available: https://dreampuf.github.io/GraphvizOnline/#digraph%20G%20%7B%0A%0A%20%20subgraph%20cluster_0%20%7B%0A%20%20%20%20style%3Dfilled%3B%0A%20%20%20%20color%3Dlightgrey%3B%0A%20%20%20%20node%20%5Bstyle%3Dfilled%2Ccolor%3Dwhite%5D%3B%0A%20%20%20%20a0%20-%3E%20a1%20-%3E%20a2%20-%3E%20a3%3B%0A%20%20%20%20label%20%3D%20%22process%20%231%22%3B%0A%20%20%7D%0A%0A%20%20subgraph%20cluster_1%20%7B%0A%20%20%20%20

node%20%5Bstyle%3Dfilled%5D%3B%0A%20%20%20%20b0%20-%3E%20b1%20-%3E%20b2%20-%3E%20b3%3B%0A%20%20%20%20label%20%3D%20%22process%20%232%22%3B%0A%20%20%20%20color%3Dblue%0A%20%20%7D%0A%20%20start%20-%3E%20a0%3B%0A%20%20start%20-%3E%20b0%3B%0A%20%20a1%20-%3E%20b2%3B%0A%20%20b2%20-%3E%20a3%3B%0A%20%20a3%20-%3E%20a0%3B%0A%20%20a3%20-%3E%20end%3B%0A%20%20b3%20-%3E%20end%3B%0A%0A%20%20start%20%5Bshape%3DMdiamond%5D%3B%0A%20%20end%20%5Bshape%3DMsquare%5D%3B%0A%7D. [Accessed: 09-Dec-2021].