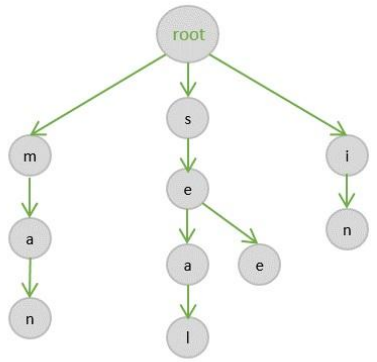
Tries

Matt Kyle, Evan O’Neill, Jacob Pierce, and Nitish Salvi

What are Trie Trees?

- AKA prefix trees or digital trees
- Tree-based data structures
    - store and retrieve strings efficiently
- Applications involving large datasets of strings
    - autocomplete and spell checking



Trie Data Structure

Source

Key Features of Trie Trees:

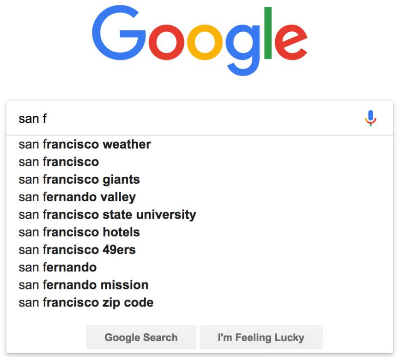
- Stores strings
    - Represents them as paths in the tree
    - Each node is a character in the string
- The root node represents the empty string
- Descendants of a node share a common prefix

Benefits of Trie Trees:

- Efficient string search and retrieval
- Space-efficient storage
  - Stores strings compactly via prefixes
- Support for dynamic data
  - insert, delete, or update strings

Applications of Trie Trees:

- Autocomplete
  - Traverses tree with partially entered strings as the prefix
- Spell checking
  - Can efficiently check word spelling against tree
- Word frequency analysis
  - Given a curtain word the tree can check its existence


san f

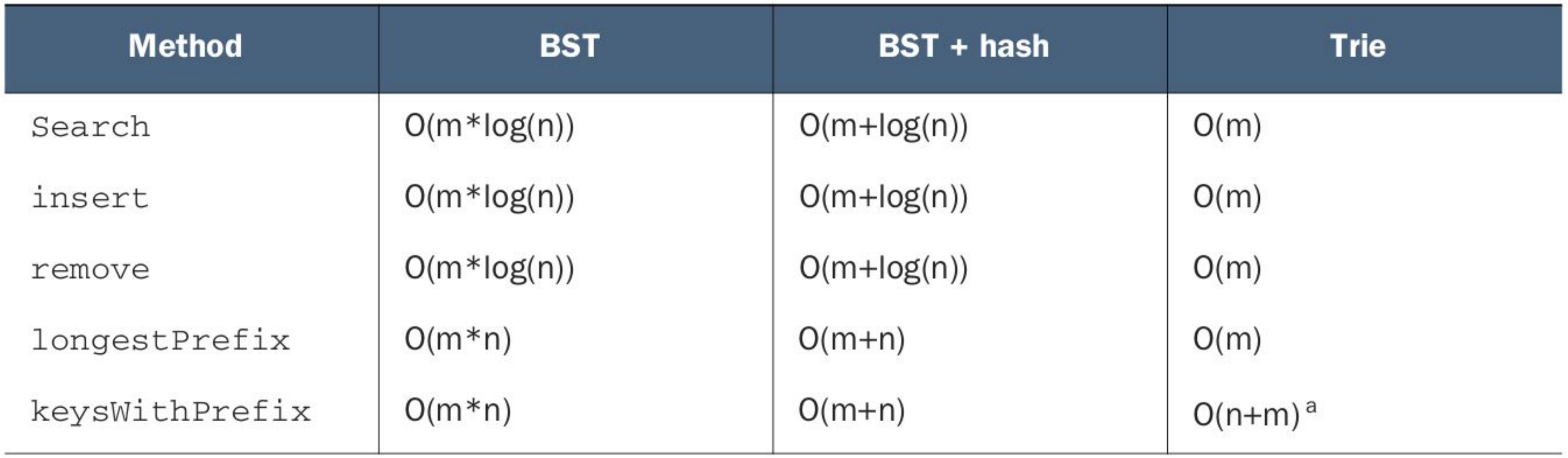
san francisco weather
san francisco
san francisco giants
san fernando valley
san francisco state university
san francisco hotels
san francisco 49ers
san fernando
san fernando mission
san francisco zip code

Source

Trie Functions

- Search:
    - Traverse a tree-like structure from the root by comparing characters of the key with nodes.
    - Determines if the key exists.
- Insertion
    - Traverse the tree from the root by adding nodes for each character of the key. The last node is marked to indicate the end of a word (often ‘$’)
    - Stores the key.
- Deletion
    - Traverse to the key’s end, removes nodes along the path. E
    - Eliminates the branches and key.

Trie vs other data structures



Method	BST	BST + hash	Trie
`Search`	O(m*log(n))	O(m+log(n))	O(m)
`insert`	O(m*log(n))	O(m+log(n))	O(m)
`remove`	O(m*log(n))	O(m+log(n))	O(m)
`longestPrefix`	O(m*n)	O(m+n)	O(m)
`keysWithPrefix`	O(m*n)	O(m+n)	O(n+m) [a]

Image from: https://stackoverflow.com/questions/4737904/difference-between-tries-and-trees

Quick recap

- As we see, the Trie is an efficient data structure that can be utilized in multiple search applications.
- We can also observe the unique features that the Trie provides.
- Understanding how the functions work allow for proper understanding of the implementation and why it used.

Our project/Goal

- Trie Tree implementation
  - insert/search/read file/output file
- Autocomplete
  - Input text file (for filling tree, searching tree, or incomplete text to be autocompleted)
  - Output text file with corresponding data

Simple trie implementation

- Insertion is simultaneously searching
  - If the char exists in the tree: continue
  - Else add
- Printing the tree
  - Creating a recursive print method in DOT format
  - Printing an already full tree
  - Re-understanding the word

Implementing the goal

- Autocomplete
    - Searching the tree to find longest substring
    - Making a list of all possible autocomplete options (knowing the list size)
- User Interface
    - Ask the user to fill/search/autocomplete
    - User inputs

Getting it to work

- Inserting & printing
  - Testing output size ~250 | Actual 2,650
  - A->C, A->T | A -> C, C -> T
  - Printing to .dot
- Autocomplete
  - Traversing Tree Structure
  - Balancing prefilling list

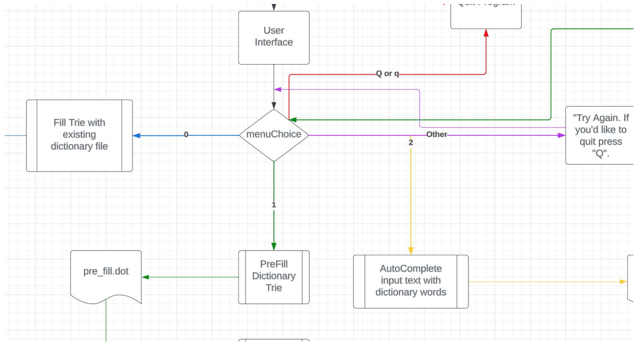
User Interface

- Iterations
  - No options, file is used to fill tree
  - File can fill, search or autocomplete
  - If you don't fill the tree auto-fills with a dictionary
  - Looping menu
  - User input option
  - Secondary file option
- Debugging
  - Distinguishing multi-word strings
  - Not having double the amount of code and slipping a user made string in for the file
  - Cin vs getline

Flowchart

- To aid in visualization on all of the possible paths/outcomes of the user input in the main function, a flowchart was created.
- The flowchart shows:
  - Choices user can make
  - Outcomes of user choices
  - Files created by certain functions
  - Menus prompted by the program

  - Link to flowchart: https://lucid.app/lucidchart/dcd03a0f-d83c-4ca3-bf8a-da2cc1256667/edit?page=0_0&invitationId=inv_cbbcba8b-0b9d-4bb0-9cc7-1af2ccd227ad#


User Interface
Q or q
Fill Trie with existing dictionary file
0
menuChoice
Other
2
"Try Again. If you'd like to quit press "Q".
1
pre_fill.dot
PreFill Dictionary Trie
AutoComplete input text with dictionary words

Operating

Via an input file the user can fill the tree, search a prefilled tree, or use autocomplete against the prefilled tree

Whatever choice is made an output file is made and the user can continue playing with the tree or exit

Assuming the user continues they have the option to use a file or input any string they want with the console. If they choose file they can then choose to use the input file or a new file

The options then loop and the user will choose to add to the tree, search the tree, or use autocomplete

Let's Take a Look at The Code

DOT Visualizer: Graphviz Online (dreampuf.github.io)

Repository: matt8011/Term-Project: Group 1's Term Project Repository (github.com)

Thank you!
Questions?