Data Structures and Algorithms Project

Topic: Word Dictionary using Data Structures

Hriday Baghar (15BEC0467)
Shashwat Singh (16BCI0180)
Insaf Muhammed Ali (16BCI0144)

Completed Tasks

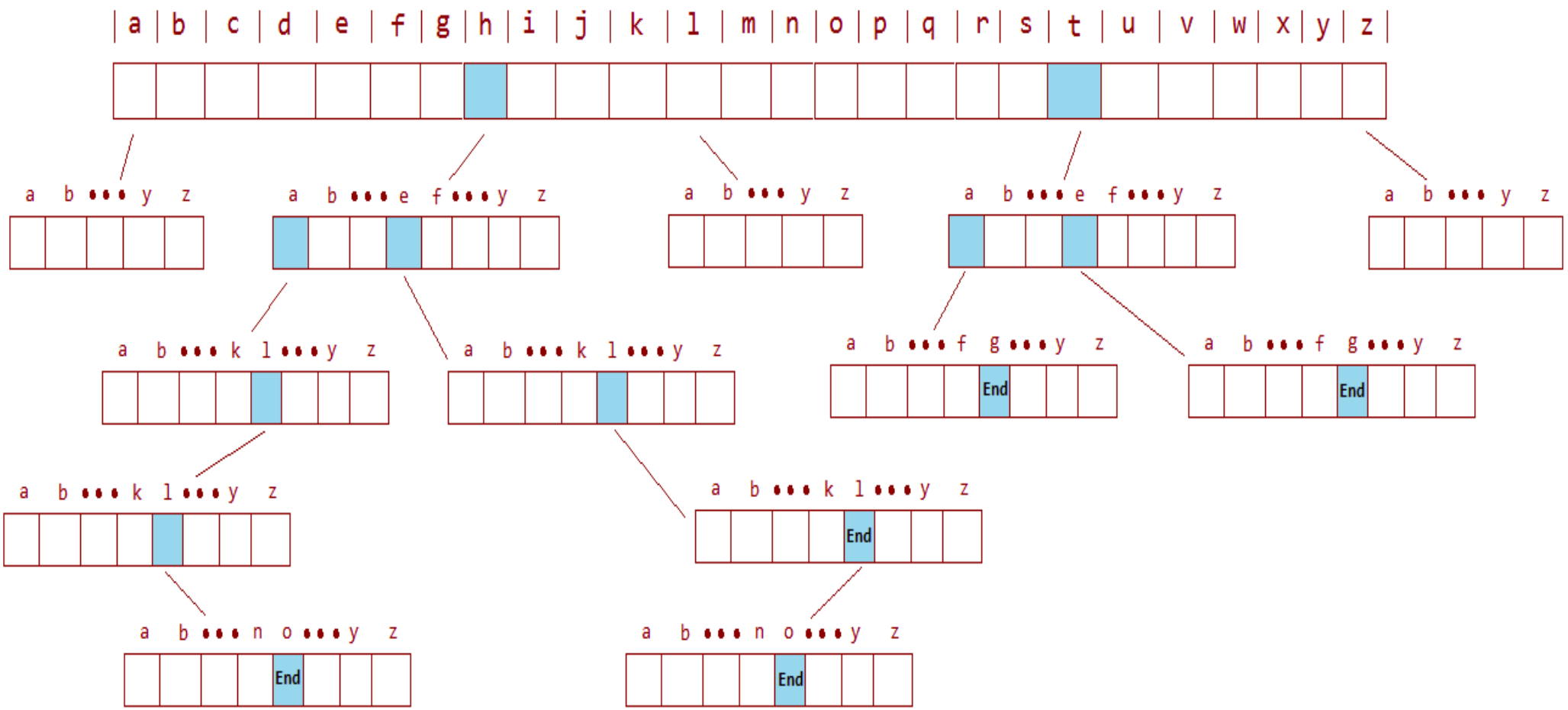
- Store word list from text file into trie
- Implement search algorithm that traverses trie and checks if word is found in trie
- Compute time taken for each search query
- Compare time taken with brute force approach (array implementation)

Trie

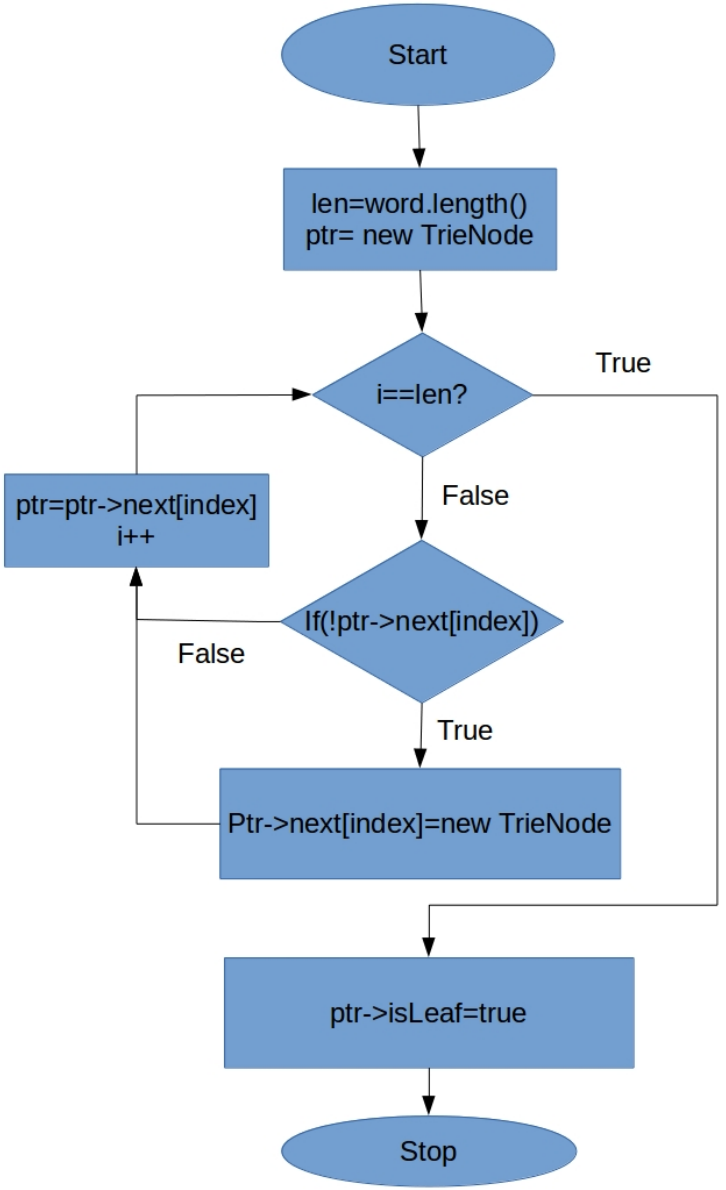
- Trie can be used for quick search of words in O(n) time where n is the length of the word.
- Even for word suggestions, trie is economical as we only traverse a small part when no match is found.
- The drawbacks are:
  - Space occupied is high as we must create 26 pointers (one for each alphabet) per node.
  - Word suggestions rely on matched prefix so if first character is incorrect, accuracy of suggestions decrease.

TRIE Datastructure Representation

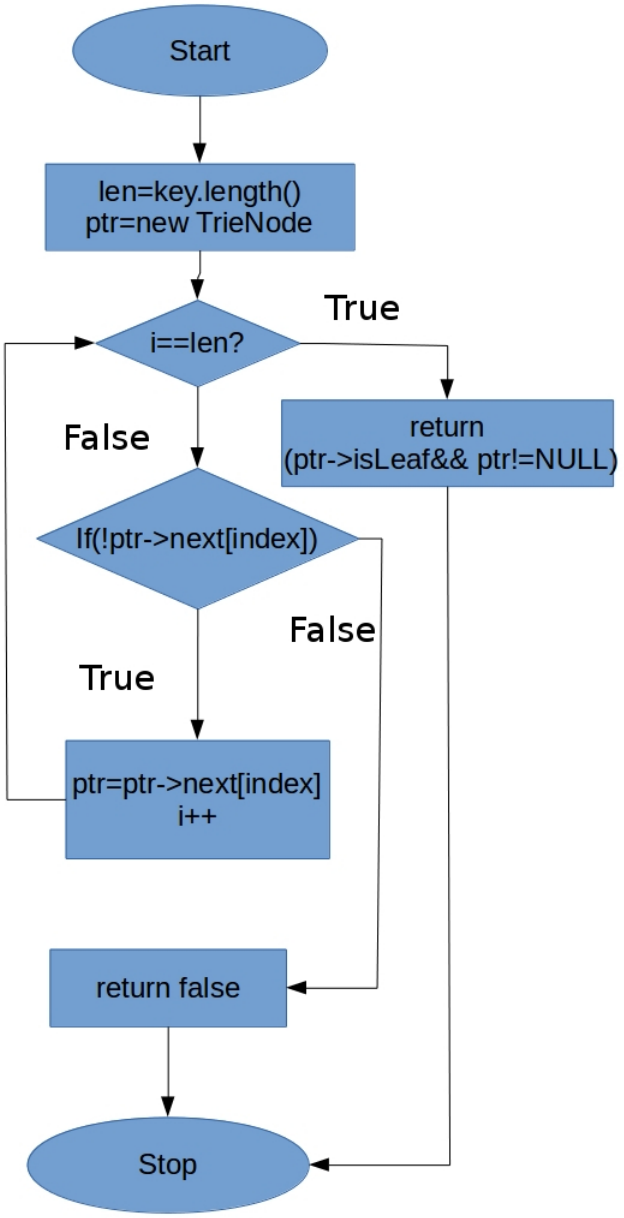
Store Words: **hello, hallo, hell, teg, tag**



Algorithm for Insertion of Word in Trie



Algorithm for Word Search in Trie



Next Steps

- Optimize trie structure to be space friendly (use TST etc)
- Return word suggestions from word list on not finding match (use Levenshtein Distance algorithm)

Ternary Search Tree

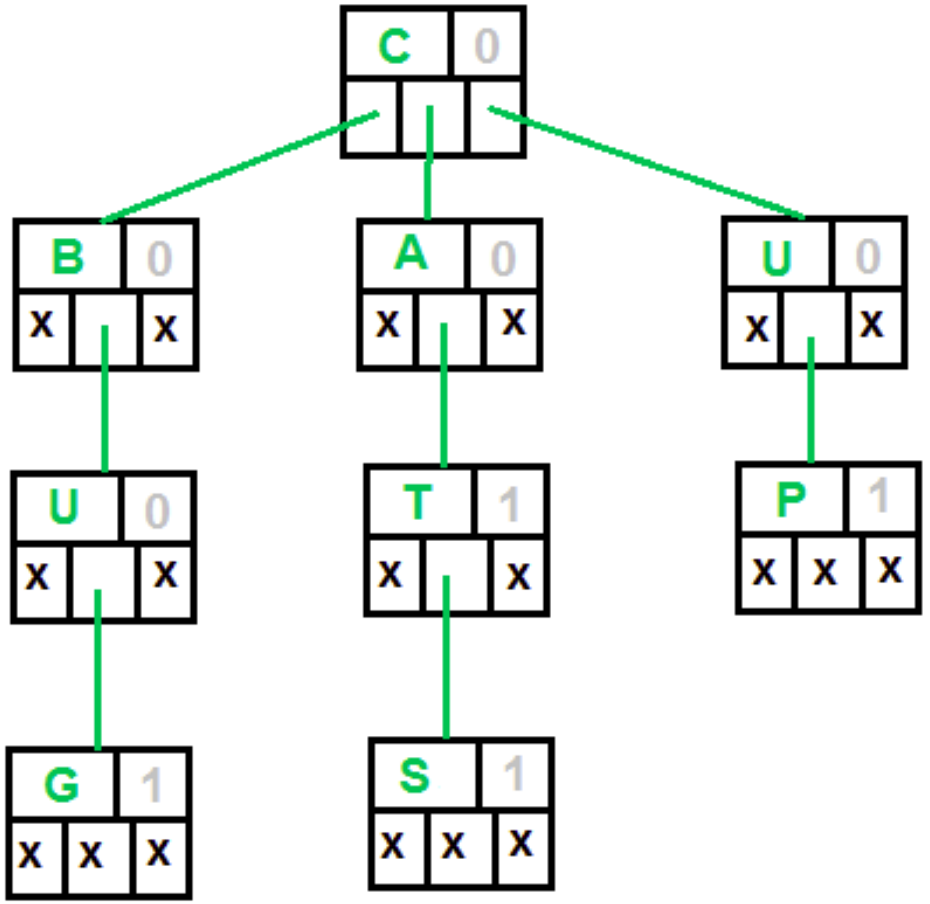
- Unlike trie(standard) data structure where each node contains 26 pointers for its children, each node in a ternary search tree contains only 3 pointers:

  1. The left pointer points to the node whose value is less than the value in the current node.

  2. The equal pointer points to the node whose value is equal to the value in the current node.

  3. The right pointer points to the node whose value is greater than the value in the current node.

- Apart from above three pointers, each node has a field to indicate data(character in case of dictionary) and another field to mark end of a string.



Ternary Search Tree for CAT, BUG, CATS, UP

Following are the 5 fields in a node

1) The data (a character)

2) isEndOfString bit (0 or 1). It may be 1 for nonleaf nodes (the node with character T)

3) Left Pointer

4) Equal Pointer

5) Right Pointer

Levenshtein Distance

- Levenshtein distance (LD) is a measure of the similarity between two strings, which we will refer to as the source string (s) and the target string (t). The distance is the number of deletions, insertions, or substitutions required to transform s into t. For example,
  - If s is "test" and t is "test", then LD(s,t) = 0, because no transformations are needed. The strings are already identical.
  - If s is "test" and t is "tent", then LD(s,t) = 1, because one substitution (change "s" to "n") is sufficient to transform s into t.
  - The greater the Levenshtein distance, the more different the strings are.

Steps to find Levenshtein Distance

- Step 1
  - Set n to be the length of s.
  - Set m to be the length of t.
  - If n = 0, return m and exit.
  - If m = 0, return n and exit.
  - Construct a matrix containing 0..m rows and 0..n columns.
- Step 2
  - Initialize the first row to 0..n.
  - Initialize the first column to 0..m.
  - Examine each character of s (i from 1 to n).
  - Examine each character of t (j from 1 to m).
  - If s[i] equals t[j], the cost is 0.
  - If s[i] doesn't equal t[j], the cost is 1.

Steps to find Levenshtein Distance

- Step 3
  - Set cell d[i,j] of the matrix equal to the minimum of:
    - a. The cell immediately above plus 1: d[i-1,j] + 1.
    - b. The cell immediately to the left plus 1: d[i,j-1] + 1.
    - c. The cell diagonally above and to the left plus the cost: d[i-1,j-1] + cost.
- Step 4
  - After the iteration steps (2,3) are complete, the distance is found in cell d[n,m].

Example of Levenshtein Matrix

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1	2	3	4
A	2	1	1	2	3	4
M	3	2	2	1	2	3
B	4	3	3	2	1	2
O	5	4	4	3	2	1
L	6	5	5	4	3	2