CO322: Data Structures and Algorithms

Lab 6 : Trie Data Structure - Part I

Date: 2017/04/27
Time: 3 - 5pm

Introduction

In computer science a *trie*, or *prefix tree*, is an ordered multi-way tree data structure that is used to store strings over an alphabet. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. Each node contains an array of pointers, one pointer for each character in the alphabet and all the descendants of a node have a common prefix of the string associated with that node. The root is represents the empty string and key values are normally not associated with every node, but only associated with leaves.

A trie allows strings with similar character prefixes to use the same prefix data and store only the tail as separate data. One character of the string is stored at each level of the tree.

The term *trie* comes from “retrieval”. Due to this etymology it is pronounced “tri”, although some encourage the use of “try” in order to distinguish it from the more general tree.

In the example shown, keys are listed in the nodes and values below them. Each complete English word has an arbitrary integer value associated with it.

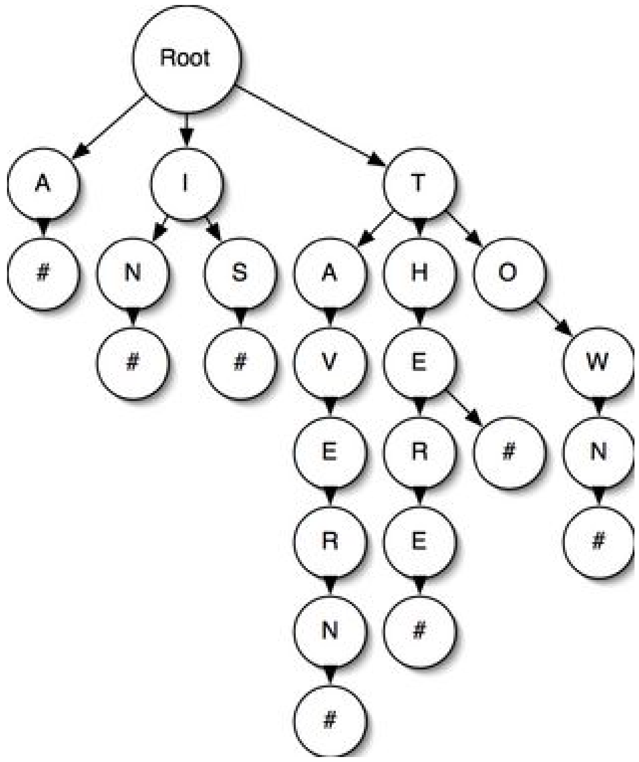


Figure 1

A common application of a trie is storing a predictive text or autocomplete dictionary, such as found on a mobile telephone. Such applications take advantage of a trie's ability to quickly search for, insert, and delete entries.

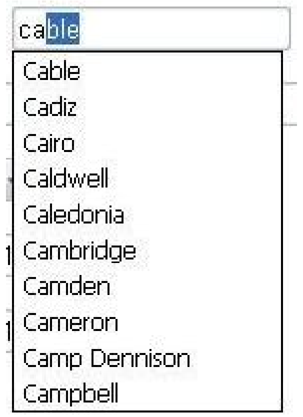


Figure 2

Lab task

In this lab you have to implement a trie containing complete words, and use it for word prediction. Your program must provide a list of complete word suggestions for a given word prefix. A dictionary file which contains 354935 words has been provided to you with lab resources. You should use that dictionary to populate your trie structure.

User should be able to enter a word(or part of a word) and hit enter key to get the list of suggested words as shown below.

```
Enter keyword: compute

==============================================================

******************** Possible Words *******************
computed
computerese
computeris
computerization
computerizati
computerizes
computerizes
computerize
computeriz
computeriz
computerlike
computern
computers
computes
compute
==============================================================
Enter keyword:
```

Figure 3

Implementation

You have to implement a trie data structure using C language. Skeleton of the implementation has been provided to you. There are two source files and one header file. Few functions have been provided as a guideline for your implementation. Please feel free to use your own function/s other than the ones provided.

Note: You are not expected to handle edge cases in your code.

Submission

Submit your code as a tar ball via Moodle. Your submission must contain the files AutoComplete.c, AutoCompleteImpl.c, AutoCompleteImpl.h which are provided to you and any other source files that you need for the implementation.

Note: Please avoid submitting separate files or any other format than tar

Deadline: 3rd of May, 11.55 pm

Lab 6 : Trie Data Structure - Part II

Compressed Tries (Radix Tree)

Introduction

Take a close look at the trie in Figure 4. This trie has several branch nodes that do not partition the elements in their subtrie into two or more nonempty groups. We can often improve both the retrieval time and storage space metrics of a trie by combining parent nodes having only one child, with that child node. The resulting trie is called a **compressed trie**, also known as Radix Tree or Patricia (*Practical Algorithm to Retrieve Information Coded in Alphanumeric*) Tree. Compressed Tries (Radix trees) consumes less memory with same functionality at the same runtime complexity, which is preferred in memory critical applications such as mobile apps.

Compressing the tree

The idea of the compressed trie is to convert long chains of single-child edges to one single edge. Example, suppose we have three words “sell”, “stop” and “stock” in our regular trie (Figure 4). We can see that “s” node has two child nodes and “s->e” and “s->t” nodes have only one child. Also the “s->e->l” node which is the child of “s->e” node also has only one child, while “s->t->o” node
which is the child of “s->t” node has two children.

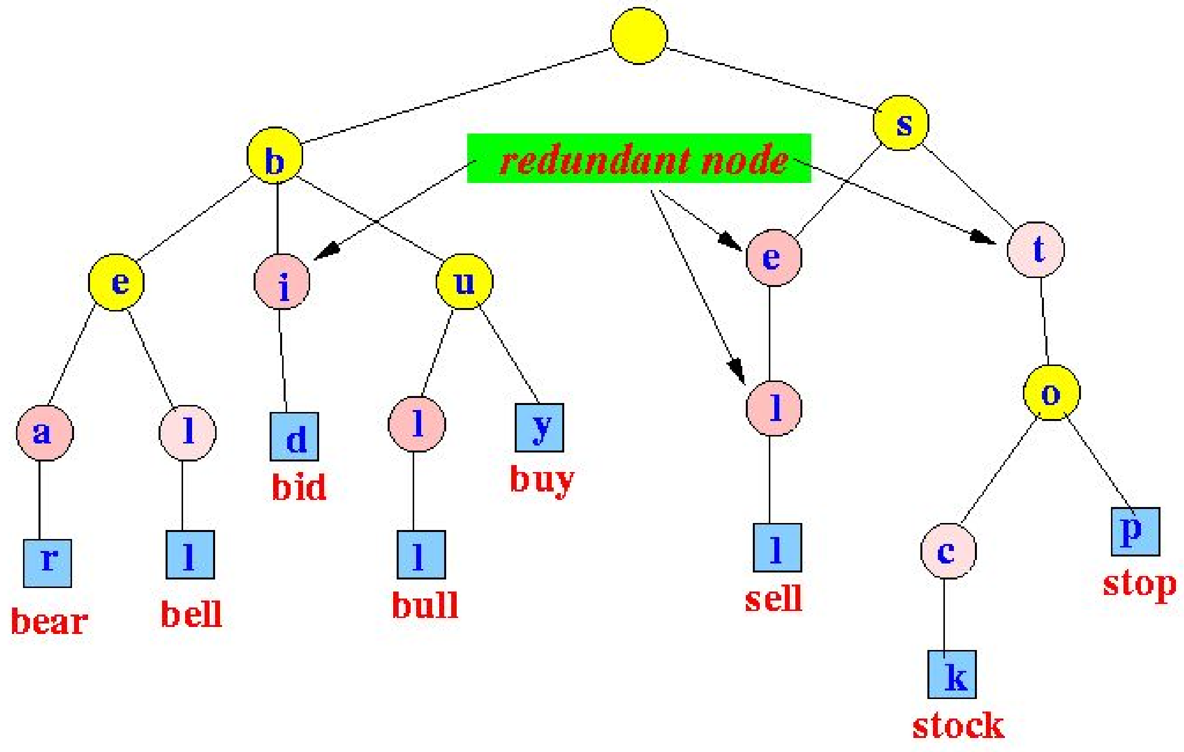


Figure 4

We can compress those branches as illustrated in Figure 5.

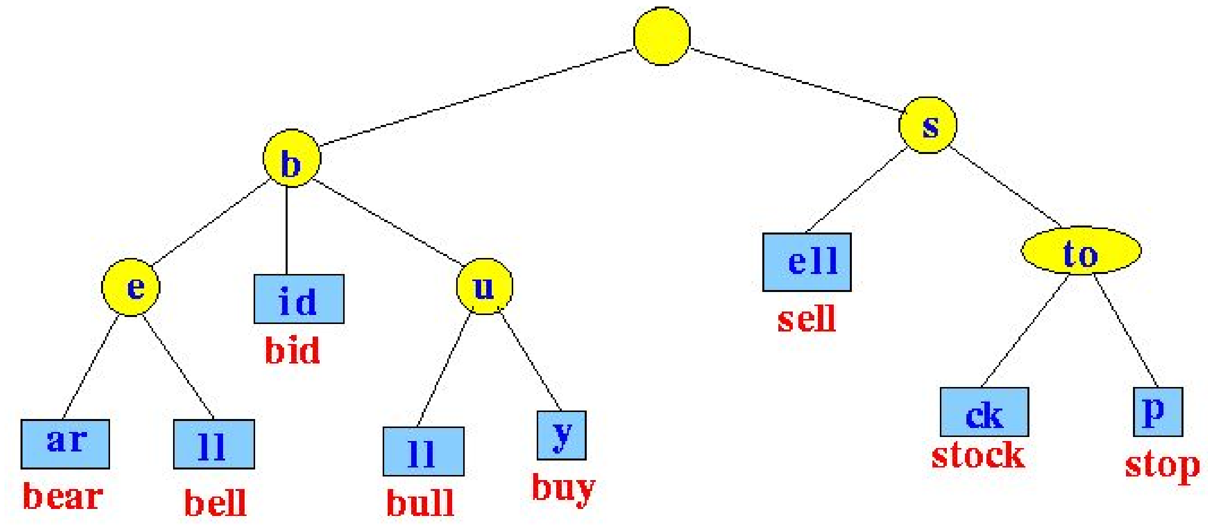
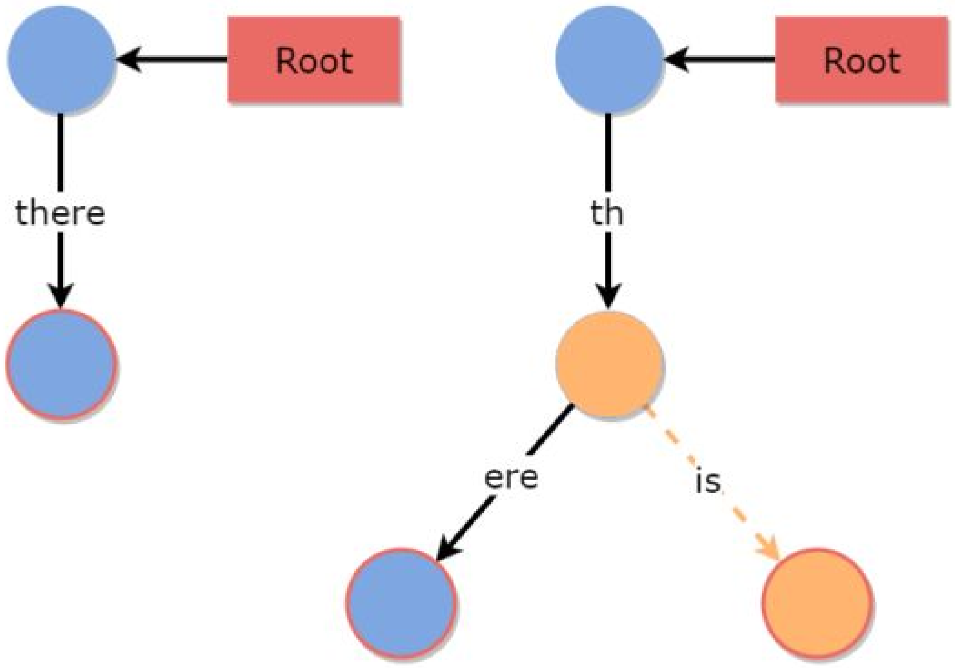


Figure 5

Insert new word

Let's consider a Radix Tree which contains word "there" and we need to add "this" word. To insert that word, we need to create new node from where it needs a new branch. Following diagram depicts that scenario.



Lab task

Your tasks in this lab

1. Implement Compressed Trie (Radix Tree). Take a copy of your original trie (part 1 of this lab) implementation and improve it to its compressed version. It should provide word insertion, searching facilities and should return the same word suggestions provided by your original trie for any given word.
2. Compare performance of the original trie and the compressed trie. Measure time to generate suggested word list for word “the” in both scenarios and report it.
3. Discuss memory consumptions of a regular trie and a Radix Tree and compare them.

Submission

Submit your code as a tar ball via Moodle. Your submission must contain the files AutoComplete.c, AutoCompleteImpl.c, AutoCompleteImpl.h which are provided to you and any other source files that you need for the implementation.

Note: Please avoid submitting separate files or any other format than tar

Deadline: 10rd of May, 11.55 pm