Assignment 4

COS 212

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Deadline: 05/05/2017, 23:00

General instructions:

- This assignment should be completed individually, no group effort is allowed.
- Be ready to upload your assignment well before the deadline as no extension will be granted.
- You may not import any of Java's built-in data structures. Doing so will result in a mark of 0. You may only make use native of arrays where applicable. If you require additional data structures, you will have to implement them yourself.
- If your code does not compile you will be awarded a mark of 0. Only the output of your program will be considered for marks but your code may be inspected for plagiarism.
- Read the entire assignment before you start coding.
- You will be afforded three upload opportunities.

After completing this assignment:

Upon successful completion of this assignment you will have implemented your own Patricia tree structure which supports insertions, deletion and searching on bit level.

Background:

PATRICIA Trees

There are a number of optimizations one can make to speed up searching within a trie and/or save space. Patricia trees (also known as Radix trees) were first mentioned by Morrison in 1968 in his journal article "Patricia trees". The word `Patricia` is an acronym for **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric. Patricia trees attempt to speed up the search process and save space by capitalizing on words with shared prefixes. Amongst other components, a trie structure's height is

determined by the length of the longest common prefix. Patricia trees actually collapse prefixes into single nodes and they become very efficient when there are a large number of words sharing common prefixes.

Consider the trie in *figure 1*. You will notice that every key in the trie shares a prefix with other keys. All keys starting with the letter `C` share the prefix `CA`, all the keys starting with `S` share the prefix `SUPER` and all keys starting with the letter `T` have the prefix `TRA`. If you know that all keys starting with the letter `S` share the prefix `SUPER` the question is then whether it's really necessary to entertain a trie of height 7 and require 5 internal nodes just to represent the prefix `SUPER`. Why not represent the prefix with a single internal node? Fortunately Patricia trees provide a solution. *Figure 2* depicts a Patricia tree containing the same keys obtained from collapsing the trie from *figure 1*.

In broad terms, a Patricia tree recognizes the fact that internal nodes with only one child may be an indication of a prefix shared by all keys containing that character up to that point and as such they could potentially be merged with their children. Consider the Patricia tree in *figure 2*. In this tree each of the indices in the root node now represent entire prefixes although the prefixes themselves are not explicitly stored. For example, the `S` index represents the prefix `SUPER` and the `T` index the prefix `TRA`. The numbers next to the characters may be interpreted as an indication of the index of the next character that should be considered in a string or the length of the prefix from there on when searching for a key. If one searches for the key `SUPERGENE` searching will commence at the root node and the first character in the string (at index 0) will be considered. There is an index corresponding to the first character `S`. The number next to `S` indicates the index of the character which should next be considered if a traversal is made to the child of `S`, in this case 5. Notice how the this is the index of the first character which distinguishes all of the keys starting with the prefix `SUPER`. When the traversal is made to the child of `S`, then the character `G` (at index 5) is examined next. At this point it is determined that the child at this index is a leaf node in which case the final test will simply be to see if the entire key in this leaf node matches the string searched for. Because prefixes cannot be built up through traversals, entire keys, and not just suffixes, need to be stored in the leaf nodes to perform a successful final comparison. Search for the key `SUPERGENE` in the trie in *figure 1* and compare the computational effort for searching for the same key in the tree in *figure 2* (hint: consider the path lengths).

Digital Trees

A digital tree is a binary tree variant of a trie. Digital trees do not process keys character for character as is the case for conventional tries, but they operate the bit string representation of keys. Another difference is that all of the nodes, not just the leaf nodes, contain keys. Traversing these structures could be seen as a combination of traversing tries, in that the bit string are read from left to right, one bit at a time, and binary search trees, in that the value of the current bit determines if the traversal will proceed down the left or the right branch of a node. Traversal starts at the root node and the first bit in the bit string is read. If the bit is a 0, traversal proceeds to the left child and if it is a 1, traversal proceeds to the right child and from this point the next bit in the string is considered. The height of a digital tree is also determined by the length of the longest prefix as is the case in conventional tries. Digital trees will often be a lot higher that a trie structure which contains the same keys. Can you think of the reasons why? Furthermore, unlike other tree structures that are balanced to optimize their searching efficiency, digital trees might be highly unbalanced.

It might seem counter intuitive to use an unbalanced structure that is much higher than its trie equivalent and for some applications that would be the case like, for example, spell checkers. However, there are certain applications for which digital trees have shown themselves to actually be very efficient.

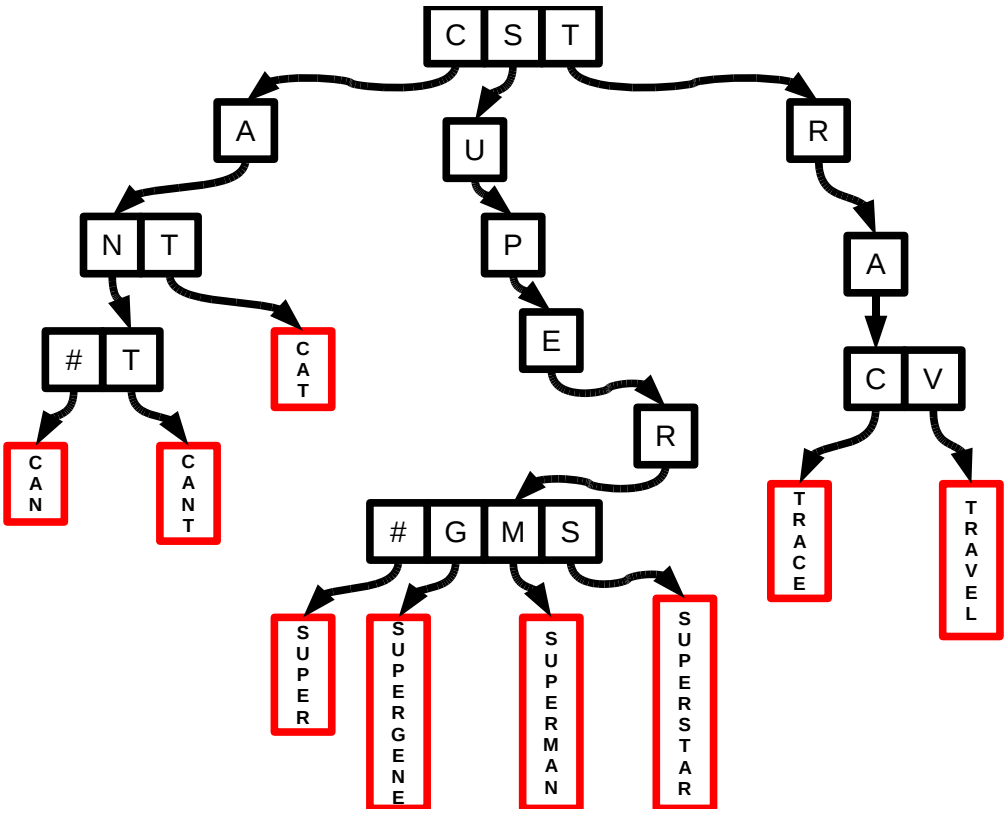

Figure 1: A trie structure.

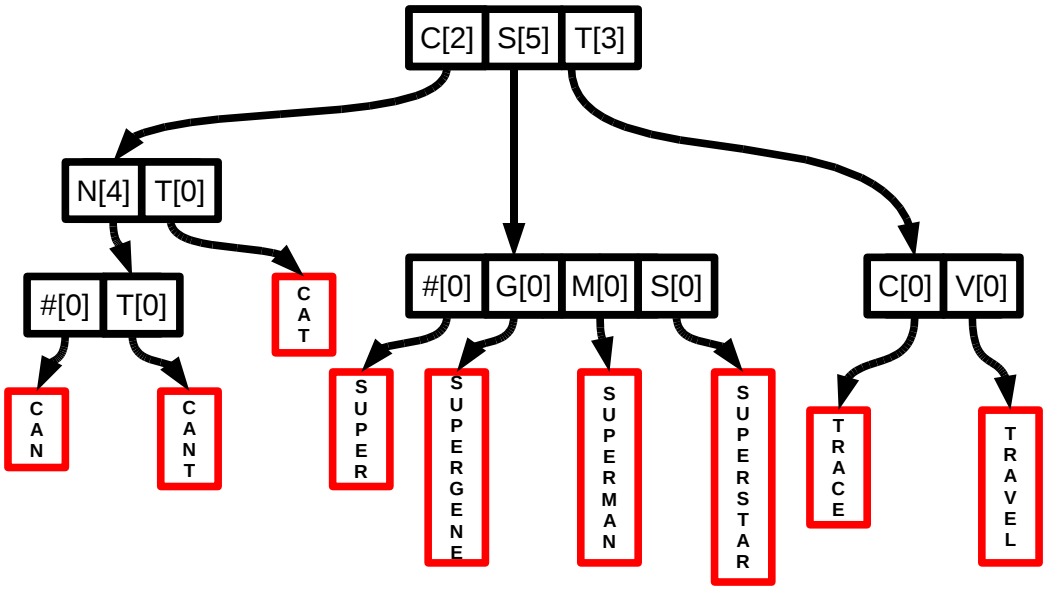

Figure 2: A Patricia tree.

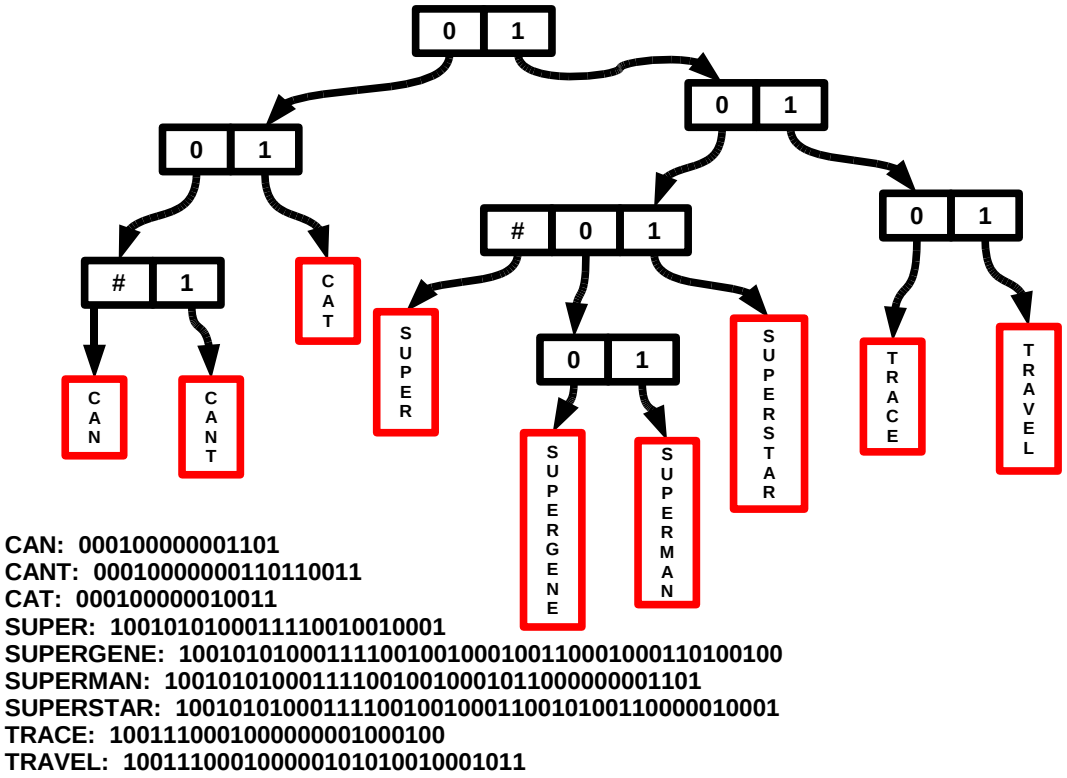

Figure 3: A Digital Patricia tree.

Your task:

You will have to implement a Digital-Patricia tree which is a Patricia tree that borrows concepts from digital trees. It has to behave as a Patricia tree but operate on bit level. The tree in *figure 3* shows the Digital-Patricia tree equivalent for the trie from *figure 1* and the Patricia tree from *figure 2*. It might not seem as if anything was gained, however, remember that instead of characters, bits are now stored in the internal nodes. The lengths of the prefixes are omitted from the figure and the bit strings represented by the structure are next to the tree.

Download the archive `assignment4.tar.gz` from the course website. This archive contains partial code which you must complete in order to create a fully functional Digital-Patricia tree. Each method is described in the comments within the partial code given to you. You have to write code to test your implementation in the `Main.java` file. This file will be replaced for marking purposes. In addition you, must create your own makefile which will compile your code and subsequently allow for the code to be run. Your makefile will be used with the following commands:

```
make
make run
```

The run rule should start your program.

Implementation Details:

Insertion and Output:

You will have to implement the `insert` method of the `DigitalPatriciaTree` class. This method accepts a string (the key) to be inserted into the structure. It should return `true` for a successful insert

and `false` otherwise. You will have to make some modifications to the normal trie insert procedure to accomplish this. (HINT: Look at how internal nodes with a single child can be combined together.) You will need to translate the keys received as strings into bit strings. Your implementation only has to cater for uppercase letters of the alphabet. Assign a number to each letter starting with `A` assigned 0, `B` assigned 1, `C` assigned 3 and so on. The 5-bit binary number of each of the integers will then be the translation for a character into a bit string. For example: The key `CAT` is translated into the bit string `000100000010011` (C = 00010, A = 00000 and T = 10011).

In order to speed up the time it takes you to populate your tree, you may consider doing some file I/O in your main. Create a text file containing the words which you would like to populate your tree with, and in your main read them one by one from the file to insert them into the tree. IMPORTANT: Your tree itself should not depend on file I/O. So if you do decide to read words from a file, make sure that all of this code only appears in your main.

Nodes:

You will also have to design and implement your own internal and leaf node classes. For marking purposes, you will have to be able to generate string representations of your nodes. For your implementation you must use the token `#` to indicate your end-of-word marker. You may assume that no words containing the token `#` will be inserted. The string representation of a node should depict only the bits applicable at the node and not any unused bits. The bits should appear in square brackets and the bits must be ordered in ascending order from left to right. If the end-of-word marker is applicable to the node, it must be shown as the first element in your node. Your strings should not contain any white space. In *figure 3*, the string representation of the root would be:

```
[0][1]
```

The parent node of the keys `SUPER` and `SUPERSTAR` would be:

```
[#][0][1]
```

Leaf nodes are simply strings representing their keys. For example, the leaf node containing the key `SUPERMAN` is simply the string `SUPERMAN`, once again without any white space.

You may add any additional members and/or variables to your node classes that you might need, like for example, a data member to indicate the length of a prefix, or the index of the next bit to be tested in the string, etc.

Searching:

In order to complete this step, you will have to implement the `search` method for the `DigitalPatriciaTree` class. The method returns a string representation of the nodes traversed in the path while searching for the key.

Consider the tree in *figure 3*. Searching for the word `SUPER` should return the string:

```
[0][1],[#][0][1],[0][1],SUPER
```

The nodes are comma-separated and the strings contain no additional white space.

If the word searched for is not in the tree, then the string should be ended with the token `!`. For example, searching for the word `CANS` should yield the string:

```
[0][1],[0][1],[#][1],!
```

The root in a non-empty tree will always appear as the first node in the returned strings. An empty tree is simply indicated with an exclamation mark.

Deletion:

You have to implement the `remove` method in the `DigitalPatriciaTree` class. The method returns a search string (see previous section) to depict the path followed to reach the key to be deleted. If the key exists, then the returned string should be identical to the string that would be generated if the key was simply searched for and the same applies if the key does not exist.

In addition, your tree would need to be restructured if possible. It must be able to collapse just as it was allowed to grow. Your nodes might merge with their children or nodes might end up with unnecessary characters. These details need to be seen to. (HINT: Nodes with one child?)

Marking and Submission Instructions:

Your `Main.java` file will be replaced for marking purposes. You must add all of your code, including additional files that you might have created, to a single archive. Your archive should be named uXXXAssignment5.zip, where XXX is your student/staff number. Your archive must be uploaded to the link `Assignment 5` for marking before the deadline.