Computer science problems.

About the problems. For this year's problem set you will explore a data structure *prefix tree* (also known as *trie*) and its variants *radix tree (radix trie)* and *Patricia tree*.

What you need to do. For these problems we ask you to write a program (or programs), as well as write some "paper-and-pencil" solutions (use any text editor that you see fit, or scan an actual handwritten solution; convert the result to pdf format if possible). You may use any programming language you want for your programs, as long as its full implementation is available at no cost and with an easy installation for both a Mac and Windows. It is best to implement each problem as a separate function so that we can run them separately. We will be looking for the following in your submissions:

- Correct code that we can run. You need to send us all your code files, including the header files for languages like C++. If you are using standard libraries, make sure to include all "import" statements, as required by the language you are using. Make sure to send the files under the correct names, including the file extension (.java, .c, etc). Make sure that the file names do not contain any identifying information about you, such as your first or last name.

- Test data for your code that you have used (you can write it in comment or in a separate file). Make sure to test your code well – you don't want it to fail our tests!

- Code documentation and instructions. If you are submitting your answers to non-code problems in a separate file, also make sure that it does not have your name in the contents or in the file name. The only place where you specify your name is the zip file with your solutions which must be of the form `yourlastname-CS-solution.zip` (replace `yourlastname` by your actual last name). **Make sure that you use zip compression, and not any other one, such as tar**. In the beginning of each file specify, in comments:

  1. Problem number(s) in the file. If you have a file with "helper" functions, mark it as such.
  2. The *programming language*, including the *version* (Java 1.7 or 1.8, for instance), the *development framework* (such as Visual Studio) that you used, unless you were using just a plaintext editor (notepad, emacs, etc), and the *platform* (such as Windows, Mac, Linux)
  3. Instructions for running your program (how to call individual functions, pass the input (if any), etc), either in comments in your program file or as a separate file, clearly named. Your program may get input from the user (i.e. it asks to enter some data and then reads it) or you may store the data in specific variables within your program. You need to clearly explain how to input or set the data.

4. Some of your code may be commented out if it is not used in the final run of your program. Make sure it is clear what needs to be uncommented to run code for each of the problems.
5. All of your test data.
6. If you were using sources other than the ones listed here (i.e. textbooks, online resources, etc) for ideas for your solutions, please clearly credit these contributions. This is a courtesy to work of others and a part of ethics code for scholars.

- Clear, understandable, and well-organized code. This includes:

1. Clear separation between problems; comments that help find individual problems and explain how to run the corresponding functions.
2. Breaking down code into functions that are clearly named and described (in comments), using meaningful names for variables and function parameters. Your code should be as self-explanatory as possible. While using comments helps, naming a variable `average` is better that naming it `x` and writing a comment "x represents the average".
3. Minimization of code repetition. Rather than using a copy-paste approach, use functions for repeated code and reuse these functions.
4. Using well-chosen storage structures (use an array or a list instead of ten variables, for instance) and well-chosen programming constructs (use loops or recursion when you can, rather than repeated code).
5. While we are not asking for the fastest program (it's better to make it more readable), you should avoid unnecessary overhead.

For this problem set you will implement and explore a data structure known as a *prefix tree* and its space-optimized variants known as a *radix tree* (sometimes called *radix trie* or *compact prefix tree*) and *Patricia tree*. Prefix trees were first described by René de la Briandais in 1959. However, the name "trie" was suggested later by Edward Fredkin, and was based on the word re*trie*val. It can be pronounced as "try" or as "tree". *Patricia trees* are a variant of radix trees and is named after an algorithm by Donald Morrison called PATRICIA, which is an acronym for "Practical Algorithm to Retrieve Information Coded in Alphanumeric".

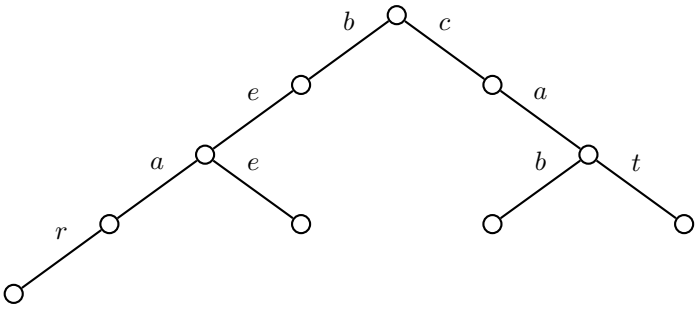
In this problem set we will be using the term "tree", and not "trie".

For more information on prefix trees and many more advanced data structures see [1].

A prefix tree is a tree for storing strings (or any other items that can be viewed as strings over some alphabet, such as phone numbers, IP addresses, etc.) that allows quick searching, insertion, and deletion. Each node can have as many children as the size of the alphabet. The root of the tree corresponds to an empty string, and each node contains a string that's a common prefix of all

strings in the subtree rooted at that node. A straightforward implementation adds one symbol per level of the tree.

For instance, the following prefix tree stores words `bee, bear, cab, cat`:



Notations. For all the problems we denote the number of characters in the alphabet by M and assume that the tree contains N strings of lengths $l_1, \ldots, l_N$, respectively.

Problem 1. One small issue with this way of storing strings of arbitrary length is that there needs to be some way of indicating whether a node is just a prefix of strings below it or has a complete string in it. For instance, in the tree above the node `be` could be a word "be" that was inserted in the tree, or just the common prefix of `bee` and `bear`.

- Give another example of a small prefix tree of English words in which there is ambiguity of whether a node corresponds to a word in the tree.
- Draw your example as a tree and mark nodes that contain words by double-circling. Note that leaf nodes always contain a word.

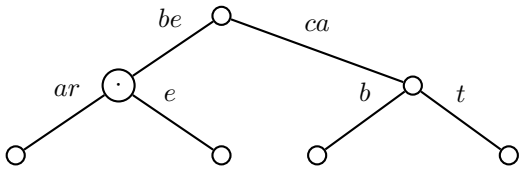
Problem 2.

1. Draw a prefix tree with words `lead, leaf, lot, key, kin, king, kiwi`, make sure to mark nodes that contain words.
2. What is the maximum number of nodes to traverse in order to find a string in the tree or determine that it's not there? Describe this procedure for strings `leaf` and `least`.
3. What is the maximum number of nodes to traverse in order to insert a new string into the tree? Describe the process of adding the string `least`. Generalize this procedure to any string, be careful with strings in inner nodes.
4. What is the maximum number of nodes to traverse in order to remove a string from the tree? Describe the process of removing the strings `lead` and `king`.

Problem 3. Compute storage requirements for the prefix tree: count memory that's used for storing string characters. Also count the number of edges

and of nodes. For what kind of a set of N strings is the storage maximal? For what set of N strings is it minimal?

Problem 4. Radix tree is based on prefix trees, but it reduces the number of nodes by merging every node that is the only child of its parent with the parent node. For instance, the tree in Problem 1 can be compressed into the following:



This is the most beneficial for sparse trees that have long sequences since it reduces the number of nodes that need to be traversed. The radix r of the tree is the maximum number of children a node may have. It corresponds to what is considered the smallest comparable portion of the key that determines the item's position. For instance, if the items are case-insensitive strings of letters in the English alphabet then the radix of that tree is $r = 26$.

1. Draw a picture of a radix tree corresponding to the tree in Problem 2.

2. What is the radix of a tree storing US phone numbers, compared digit-by-digit?

Problem 5. Assume an alphabet of M characters and the comparisons done character-by-character. For instance, if I am comparing a string `leaf` to a string `least`, I would make four character comparisons to determine that they aren't equal, assuming that I start at the beginning; however, if I came from a node that shares their common prefix `lea` then I only need to make one more comparison to determine that they aren't the same.

1. How much storage is required for storing N strings of lengths $l_1, \ldots, l_N$? Count the storage of the characters and also the number of nodes and edges. For what kind of a set of N strings is the storage maximal? For what set of N strings is it minimal?

2. Describe an algorithm for searching for a string in a given radix tree. How many character comparisons does it make in the worst case? How many nodes does it traverse in the worst case?

3. Describe an algorithm for adding a string to a given radix tree. How many character comparisons does it make in the worst case? How many nodes does it visit (by traversing or creating) in the worst case?

4. Describe an algorithm for removing a string from a given radix tree. How many character comparisons does it make in the worst case? How many nodes does it visit in the worst case? How many nodes would you need to delete in different cases?

Problem 6 (implementation). Implement a radix tree to store DNA segments. Assume that DNA encoding consists only of letters `A, C, G, T`. Here is an example of a DNA fragment: `CTGCACGTGTCCCTGAAGGCTTCCAGAGGAAGCTTTACA`. Segments can be between 10 and 100 characters long. You can use randomly generated strings for testing, although in reality there are dependencies in DNA sequences. No symbols other than `A, C, G, T` may appear in the data. Note that a valid fragment may also be a prefix of another valid fragment. Your implementation must have:

- A method to add a string.
- A method to delete a string.
- A method to search for a string.
- A method to find out how many strings are in the tree.
- A method to find out how many nodes are in the tree.
- A method to print out all of the strings in alphabetical order.
- A method to print all nodes in the tree and what prefixes they correspond to.

Your goal should be to optimize the storage, so make sure that you are storing any repeated data only once. Also make sure that your way of looking for a string is efficient and avoids unnecessary comparisons and lookups. Explain, in comments, your implementation choices.

Problem 7. For your implementation please compute an expected amount of storage (characters and any overhead generated by storing nodes and edges) to store N randomly generated sequences of `A, C, G, T` of lengths between 10 and 100. Also estimate the storage that a non-optimized prefix tree (as in Problem 2) would have for the same data.

Problem 8. In your implementation, assuming that there are N strings stored in the tree, how many steps would it take, on average, to find a string of length 50 (or determine that it's not in the tree)? Please list what these steps are and what is the best case and the worst case.

Problem 9. Inputs to the radix tree are often treated as strings of zeros and ones. For $r = 2$ strings are compared bit-by-bit and a node corresponds to a binary string of common prefixes. For instance, if you have two strings 010101 and 010100, their parent node would have the common prefix 01010, and the strings themselves would be the two descendants of that node.

If inputs to a radix tree are strings of zeros and ones and radix of a tree is higher than 2, the radix then must be $r = 2^a$ for a positive integer a, and strings are compared in chunks of a bits. For instance, if we consider the same two strings and $r = 4$ then the strings consist of three two-bit chunks: 010101 consists of the chunks $01, 01, 01$, and 010100 consists of $01, 01, 00$. Thus their common prefix is two chunks $01, 01$, and they differ in the last chunk: 01 versus 00.

Explain the tradeoffs between the two representations, $r = 2$ and $r = 4$. What are the factors that would determine which of the two you would use? Describe a data set for which $r = 2$ is preferred and a data set for which $r = 4$ (or higher) is preferred.

Problem 10. In 1968 Donald Morrison proposed a further optimization of radix trees known as Patricia trees. The original paper [2] has all of the details of the implementation and is quite complicated. A simplified explanation of a binary Patricia tree is given here:
`http://users.monash.edu/~lloyd/tildeAlgDS/Tree/PATRICIA/`.
The idea is that there is no need to compare every character in order to determine where a string is (or could be) positioned in the tree: only a few characters in the string determine its position. Thus each internal node only stores a number: the index of the position that needs to be checked. If the position has a 0 (or a, as in the example at the link), the search continues to the left. If it's a 1 (a b in the example), the search continues to the right. Once a leaf node is reached the rest of the characters in the search string are compared. If any of them don't match, the string isn't actually there.

Note that if a string is a prefix of other strings in the tree, the corresponding internal node would have to indicate that and store the string. This is illustrated in the last picture at the link.

1. Draw a Patricia tree over the alphabet $0, 1$ that has strings `10110, 110, 100, 001, 00110, 001011`

2. Describe the search in this tree for the following strings: 0, 00110, 01110 (the search determines whether the string is in the tree). For each string also show the number of character comparisons performed and the number of visited nodes.

3. Prove that in a non-empty tree if no string is a prefix of any other string in the tree then each internal node has 2 children (you might want to use induction for your proof).

4. How does the number of nodes and comparisons compares to a straightforward radix tree with $r = 2$? What are the tradeoffs between the two implementations, and when would you recommend one, and when the other one?

Problem 11 (design, implementation, experimenting). There are other variations of prefix trees, such as suffix trees that store strings based on a common suffix (i.e. ending) rather than prefix.

Now that you have learned various variations of prefix trees, you need to design and experiment with your own version for storing English text. Specifically, your program will have to take input as a plaintext file and store all words in the text (converted to lower-case) into a tree. If a word appears twice, it should only be stored once. However, do not preprocess your data to eliminate duplicates or sort: try to add each word into the tree, and if it's there, just do not add it again.

You may convert words into any other encoding. For instance, you may treat them as ASCII characters, or encode a as five zeros, b as four zeros followed by a 1, etc. You don't need to actually use machine-level bitwise operations.

In order to easier get statistics on character comparisons, please set up a counter for character comparisons and increment it every time you perform a comparison. Note that depending on your implementation, the characters may be English characters or binary.

As an example of data, use any wikipedia page, such as
`https://en.wikipedia.org/wiki/Grace_Hopper`
(note that not all words are dictionary words, and some may be acronyms). The data set that your program will be tested on is similar to that.

Your implementation needs to support insertion, search (that returns true or false, depending on whether a string is in the tree), and deletion.

Your solution will be evaluated for correctness. Correct solutions will be also evaluated on the following:

1. The total number of bits necessary to store characters. Note that if you use a binary encoding, 0 and 1 count as one bit each, even if in practice they take more in your implementation. Letters of the English alphabet are counted as 5 bits, unless your encoding is specifically different.

2. The total storage needed to store nodes and edges. For instance, if edges or nodes contain string fragments that are not a part of the data storage above, they would be counted. Numbers, arrays (filled in or empty), pointers to other nodes also need to be counted.

3. The number of bitwise comparisons to create the tree for a given file (i.e. the number of comparisons multiplied by the length of the character encoding).

4. The number of bitwise comparisons to find a string in the tree. Your program will be tested on data in which roughly a half of the strings are in the tree, and the half aren't.

5. The total number of nodes visited when tested on a search data set as specified in the previous item.

Make sure that your program clearly specifies how to input a file for testing (both to create a tree and to test if strings appear in the tree).

References

[1] Peter Brass. *Advanced Data Structures*. Cambridge University Press, New York, NY, USA, 1st edition, 2008.

[2] Donald R. Morrison. Patricia: Practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, October 1968.