Project 3: Tree-Based Encryption

Due Friday, 2019/10/11, 11:59:59PM

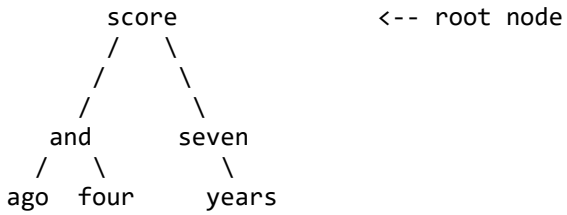
Project description

Encryption is a procedure for turning information which anyone can understand into information which only people with knowledge of a special key can understand. Decryption is the reverse of this process. Many simple encryption schemes are based on substitutions. For example, we could "rotate the alphabet" by 1 letter, letting b represent a, c for b, and so on. Thus the phrase "four score and seven years ago" would be encrypted into "gpvs tdpsf boe tfwfo zfbst bhp", using this simple substitution cipher. The key needed to encrypt information in this type of substitution cipher is the knowledge that each letter is shifted forward one (and the key for decryption is the to shift each letter backward one). When a message is not encrypted, it is called "cleartext."

Write a program which implements encryption and decryption using word substitutions rather than letter substitutions.

The codebook (the key) for this substitution cipher is a binary search tree of words. The tree stores all possible words that can be encrypted (or decrypted). The method for encryption is a representation of the path that one takes from the root of the tree to find the word in the tree.

Consider a binary search tree codebook that looks like this:



Then let us encrypt the phrase we had earlier ("four score and seven years ago") by translating each word into the the path taken from the root of the tree to the word. The word "four" translates to "r01", since we start at the root (r) and go to its left child (0), then the right child (1) of that node. The word "score" translates to "r", since it is the root word. The entire phrase is encrypted into "r01 r r0 r1 r11 r00". Each word is encrypted by starting at the root (r), and searching. Each time we move to a left child as we go down the tree, we add a zero (0), and each time we move to a right child, we add a one (1).

The tree is, of course, ordered. Thus the word in each tree node should be greater (in lexicographic sorted order) than the words in all nodes in its left subtree, and lesser than the words in all nodes in its right subtree.

Notice that depending on the codebook (which is the tree), the same input message could be encrypted in many different ways.

To build a codebook, insert the words into the codebook one at a time. One way to build the example codebook above is to start with an empty tree, and then insert the following words in the given order: 'score', 'and', 'seven', 'ago', 'four', 'years' (which you may notice is a level-order traversal of the final tree). However, that is not the only way to form that tree. The order the words are inserted can change the structure of the tree.

When inserting a word, if it is already in the codebook, then do nothing. Otherwise, place it as a new leaf node at the natural position of the codebook as a search tree.

Removing a word uses the following strategy. Assuming that the word to remove is at tree node N:

1. If N has no children, then N is simply removed.
2. If N has only one child, that child replaces N.
3. If N has both children, then the left-most node in the right child of N is removed and used to replace N.

Input

Input is a sequence of up to 10,000 lines, each containing one command. Input ends at end of file. Each command is formatted as one of the following:

- `i x` — insert cleartext word x into the codebook.
- `r x` — remove cleartext word x from the codebook.
- `e 'cleartext message'` — encrypt the given cleartext message. Each cleartext message consists of words separated by single spaces, and the entire message is surrounded by single quotes. There are only two single quotes (the first and last).
- `d 'encrypted message'` — decrypt the given encrypted message. Each encrypted message has the same format as a message to encrypt, but the words are encrypted.
- `p` — print the codebook in preorder format, visiting left children before right children.
- `q` — quit the program (stop processing input) — this is always the last command.

Every cleartext word in the input is 1 to 30 lowercase alphabet characters (a--z). Every encrypted word in the input is `r` followed by 0 to 30 `0` or `1` characters.

Output

For each print preorder command, print the tree in a preorder format, indenting each level of the tree by two spaces. For each encrypt (decrypt) command, print the encrypted (decrypted) message on one line. If the input asks the program to encrypt or decrypt a word that is not in the codebook, the program should print out '?'.

The reason for printing the codebook in preorder is because it would allow the receiver of the encrypted message to reconstruct the codebook. The receiver could take the codebook you printed and insert the words into his codebook in the order you printed them, and obtain the same codebook.

Notes

You may find the `getline(istream &, string &)` function useful for getting an entire line at a time (e.g. after you read a command to encrypt or decrypt). However, this is not necessary. Also, messages on the input do not span multiple lines; they are on the same line as the 'e' or 'd' commands.

Sample inputs

Several sample inputs are available here. You can get the correct output by capturing the output of the sample executable on these inputs.

- sample-3-1

- sample-3-2

Provided code

You must use the .h files provided here. I have provided a lot of code for you in the 'prof' file. You are not allowed to change the code in these files. Instead, write your code in and submit the 'student' file.

- bst-prof-proj3.h
- bst-student-proj3.h

Remember that when using templates, all of the code you write goes in the .h file. So turn in 2 files for this project: bst-student-proj3.h and driver-proj3.cpp. Your driver should `#include "bst-student-proj3.h"`, and that file should `#include "bst-prof-proj3.h"` (which it already does).

The EncryptionTree's two methods (encrypt and decrypt) method should not print anything; instead they should return values to the caller who can then print something.

Sample executables

Here are sample executables for you. When you design test cases, you can judge your output against the output from my correct solution.

- DOS executable
- Linux (Intel) executable
- MacOSX (Universal) executable

If you give a command-line argument to these executables, they print extra information about how they are operating. For example, this command executes the program like normal, redirecting input from a file called my_input.txt:

```
% project3_solution_dos.exe < my_input.txt
```

But here is the mode of operation that causes the program to print out what it is doing in more detail:

```
% project3_solution_dos.exe printMore < my_input.txt
```

The command-line argument doesn't have to be the word "printMore", it can be anything.

Structuring the project

Since this is a large project, it helps to have a plan of attack. The following milestones should be turned in via the upload site.

Step	Finish by	Milestone
1.	Thursday, October 3 at noon	WRITE and TEST all the unfinished methods in the BSTNode.
2.	Tuesday, October 8 at noon	WRITE and TEST the BST insert method, and the encrypt and decrypt methods in the EncryptionTree.

3.	Thursday, October 10 at noon	WRITE and TEST the BST remove method and the project driver. Finish early so you have time to debug.

Use your book. Your book has code for writing a binary search tree (section 4.3, see also section 4.6), but your code would be fairly different, since the book uses recursion but your code mostly would not use recursion. However, you should read and understand the code in the book first.

Final notes

Remember when writing this program to adhere to the coding style guidelines. This is an individual project, so you must work alone. No credit is given for a solution which does not pass all the hidden tests I create, or does not pass in the allowed time. For more detailed instructions, read the project submission guidelines.

Computer Science Department
Baylor University

valid html and css