Obligatory Assignment

Lever oppgave

Leveres innen Fredag innen 23:59 **Poeng** 60 **Må leveres** en filopplasting
Filtyper pdf,zip,tar.gz,tar.bz,rar, og tar.bz2 **Tilgjengelig** etter 29 jan i 8:00

Overview

In this assignment you will implement a set ADT. Specifically, you will implement what is known as an *ordered* set, since the set relies on an ordering of elements and supports iterating over its elements in that order. The operations to be supported by the set are:

- Adding an element to the set.
- Getting the current size of the set.
- Checking whether a specific element is contained in the set.
- Getting the union of the set and another set.
- Getting the intersection of the set and another set.
- Getting the set difference between the set and another set. (Also known as the relative complement set).
- Iterating over the elements of the set, in sorted order.

There should be no bound on the number of elements that may be inserted into the set, short of running out of memory.

Algorithms

The primary concern of your implementation should be to implement the specified ADT interface correctly. The secondary concern is to do so with good worst-case complexity for the supported operations. You may find yourself in a situation where your implementation offers a trade-off in worst-case complexity between the various operations; if so, explain the trade-off and the motivation for making it in your report.

One of the simplest implementations you could go for would be to represent sets as sorted lists. This has the advantage of optimal complexity for iteration (as for *why* it's optimal, that should go into your report) but has worse complexity for adding elements than many alternatives. Another option is to use a search tree, but keep in mind that search trees can degenerate into linked lists for worst case inputs, unless you add a scheme for keeping them balanced. Implementing iterators for a tree-based representation is also non-trivial.

However you choose to implement it, you need to support iteration over the elements in sorted order. This suggests that the algorithms for set unions, intersections, and differences could be the same regardless of your representation: All of these can be implemented as variations of the basic merge algorithm. All in all, you should pick an approach that you're comfortable with and emphasize writing a good report; you will get little or no credit for sophisticated algorithms unless they are documented in your report. Starting out with something simple and refining it is probably a good idea.

Once the set-ADT is implemented, thourough performance evaluation should be performed and presented in the report. Ideally another algorithm should be implemented and evaluated, and compared to the first implementation. If you chose to start with lists, arrays could be used for comparison and vice versa. Remember to document interesting findings in the comparison.

Applications

The code comes with a complete test application that calculates various sets of integers (even numbers, odd numbers, primes and non-primes). This application, called *numbers*, may be used for testing (although you probably want to write some test code of your own, too). For a correct implementation of the set ADT, the output of the numbers application looks like the contents of the file *numbers-expected.txt* (located in the zip file with the assignment code). The application also serves as an example that illustrates how the ADT interface is used. Look here for examples of iteration, insertion, unions, etc.

You will also complete the implementation of a second application, *spamfilter*, which is to be a very simple and naive application for classifying e-mails (text files actually, but let's pretend they are e-mails) as spam or non-spam.

The algorithm used by the spamfilter application should be as follows. It starts out with a number of e-mails known to be spam, and a number of e-mails known NOT to be spam. These are pre-classified examples from which the algorithm will attempt to classify the remaining e-mails. The rule for classifying an e-mail as spam is this: If the e-mail contains a word that occurs in ALL of the spam e-mails and NONE of the non-spam e-mails, then it is spam; otherwise it is not spam. To phrase this in terms of sets, assume that each e-mail has been tokenized into a set of words. (Use the tokenize() function in spamfilter.c for this). Let S1,S2, ..., Sn be the spam word sets, and N1, N2, ..., Nm be the non-spam word sets. An e-mail M is then to be classified as spam if and only if:

$M \cap ((S1 \cap S2 \cap ... \cap Sn) - (N1 \cup N2 \cup ... \cup Nm)) \neq \emptyset$

(If that last symbol displays as a square, you are using a bad browser. It is supposed to be the empty set.)

The spamfilter application should accept three directory names on the command-line, specifying where to find the spam files, the non-spam files, and the files to be classified, respectively. So, assuming these files are to be found in the directories spamfiles, nonspamfiles, and mailfiles, the command-line would look like this:

./spamfilter spamfiles nonspamfiles mailfiles

The zip file with the assignment code also contains a sample data set for the spamfilter application. If you implement the above algorithm correctly, it should make the classifications in the file *spamfilter-expected.txt* (also located in the zip file with the assignment code) for the data set.

Code

Your starting point is the following set of files (found **here**):

- set.h - Specifies the interface of the set ADT. Do not modify this file.
- list.h- Specifies the interface of the list ADT from the previous assignment.
- linkedlist.c- An implementation of the list ADT based on doubly-linked lists.

- common.h - Defines utility functions for your convenience. Two functions have been added since the first assignment: list_files(), which recursively traverses a directory and returns a list of file names, and compare_strings(), which is a comparison function that may be used with sets of strings and lists of strings.
- common.c- Implements the functions defined in common.h.
- numbers.c - The numbers test program.
- spamfilter.c - A stub for the spamfilter program (complete this).
- assert_set.c - Functionality for testing if your set operations work as expected.
- Makefile - A Makefile for compiling the code.

As in the previous assignment (list ADT), you need to add a new source file that implements your set ADT, and edit the name of this source file into the Makefile, as the value of the SET_SRC variable.

We've bundled all of the source code in a zip file along with the sample spamfilter data set. The zip file is located in the files section.

Deliverables

You must develop your own understanding of the problem and discover a path to its solution. If you have any general questions, then you may ask someone else for help. But never copy another student's code or report.

There are two deliverables for this assignment: A report and the source code.

Hand in the two deliverables via Canvas, where you found this document.

The report must be in the pdf format. The source code must be placed in a compressed file, you can choose between the zip, rar, tar.gz or tar.bz2 formats.