Programming in C

Deadline: *6 March, 23:59*

Assignment 3: Text compression

Introduction

The friend you helped checking euro bank notes serial numbers was very impressed with the program you wrote for him. And as often happens in these cases he is asking for your help again. He came across some documentation that he needs for his work but it is compressed with a strange text compression method. The problem is that he cannot find the decompression utility, but luckily he knows how the compression method works. Your job is to help your friend once again by writing a C program that can decompress his text files.

Compression method

The idea behind the compression method is to take advantage of the repetition of words. Words that appear frequently in the text are stored in a list. These words in the input text can then be replaced with a index into the list, reducing the size of the encoded text. Words that are marked with a `*` are added to the list. The next time that a starred word appears in the input text it is replaced with the list index of the word.

In the following table we show how to the list changes as we decode a short piece of text. Words that have been read appear in boldface. The first six words are all marked with a star, so they are all added to the front of the list in the order they are read. After reading the word `programming*` we have the following input and list state:

As* soon* as* we* started* programming*, 2 found*
to our surprise* that it wasn't* 6 easy to* get
programs right 7 6 had thought. Debugging* had* 2 be
discovered. I can remember the exact instant when I*
realized that* a large part of my life from then on was
going 4 be spent in finding mistakes in my own
programs*.
— Maurice Wilkes discovers debugging, [1949]

0	programming
1	started
2	we
3	as
4	soon
5	As

Next we read the index 2 that matches the word `we` in the list. The decoder has now decoded the text `As soon as we started programming, we`. We continue reading and adding starred words until we reach the index 6. The input state and list now look like:

**As* soon* as* we* started* programming*, 2 found*
to our surprise* that it wasn't*** 6 easy to* get
programs right 7 6 had thought. Debugging* had* 2 be
discovered. I can remember the exact instant when I*
realized that* a large part of my life from then on was
going 4 be spent in finding mistakes in my own
programs*.
— Maurice Wilkes discovers debugging, [1949]

0	wasn't
1	surprise
2	found
3	programming
4	started
5	we
6	as
7	soon
8	As

The decoder looks up index 6 and prints the word `as`, and it continues adding starred words to the list and looking up the indexed words until the input text is decoded as:

```
As soon as we started programming, we found to our surprise that it
wasn't as easy to get programs right as we had thought. Debugging had
to be discovered. I can remember the exact instant when I realized
that a large part of my life from then on was going to be spent in
finding mistakes in my own programs.
— Maurice Wilkes discovers debugging, 1949
```

Note that as we add words to the front of the list the index used to replace the word `we` changes. First index 2 and later index 6 was used to reference `we` in the list.

Removing words from the table

The encoder decides the words it wants to add to the list by marking them with a star. So it could add all the new words it encounters to the list. The result would be a very large list and large indices into that list in the encoded text. The compression ratio would suffer and it would also slow down the encoder and decoder.

For this reason the encoder limits the number of words in the starred-list with the `-l` option. Frequently used words are added, but as the frequency of words in the text can change over time, we need the ability to remove words from the starred-list when their usage frequency drops. So the following extension was added to the compression scheme. Words tagged with a `~` are removed from the list and from that point on cannot be compressed with a index number until they are added again somewhere further on in the text.

To see how the encoder would use `~` to remove words from the starred-list, we examine the following text encoded with a starred-list of size 1:

```
test*, one, two, one* two, 0, two, test~, 0, two, test.
```

First `test` is added but it is later replaced by `one`, because the encoder noticed that `one` appears more often than `test`. The third occurrence of `one` is replaced with the index 0. The next occurrence of `test` is flagged with a `~` to signal that is has been removed from the starred-list. There was only `one` spot and that is currently occupied by `one`. Next we see that `one` is replaced with index 0 again but `test` cannot be replaced with an index as it not in the starred list. The decoded text is:

```
test, one, two, one two, one, two, test, one, two, test.
```

Assignment

For the assignment you need to implement the decoder for the compression scheme explained above. The *strict* requirement for this assignment is that you need to create a *linked list* data structure to store the starred-list. The linked list data allow you to efficiently insert and remove words from the starred list.

Implementation tips

Splitting lines

File input and string manipulation are not the strong points of the C language. We provide you with some template code (`template_code/decoder.c`) that splits lines into words based on a set of delimiter characters. If you decide to use the template code, check the manual page of `strpbrk()` so you know what the function does. If you decide to write your own main decoder function you should still use this definition of a set of characters that delimit a word:

```
#define DELIM "!?\",. \n"
```

This is the definition that the encoder uses to compress the text, using a different set of characters for decoding will break the encoding method.

Linked list

The functions and structures that implement the linked list should be stored in a separate C file and corresponding header file. The interface of your linked list should have functions that are similar to these:

```
struct list* list_init();
void list_add(struct list *l, char *d);
char* list_remove(struct list *l, char *d);
char* list_at_index(struct list *l, int index);
void list_print(struct list *l);
int list_cleanup(struct list *l);
```

Note that this list stores character pointers. You could decide to make your list more generic by using void pointers.

Miscellaneous

The `template_code` directory contains an `input` directory with test cases. Some encoded files will not contain the `~` markers to remove words. It is a good idea to make sure that your decoder works correctly for these tests before attempting test cases with tildes. We have included the encoder so you can create your own test cases. `make check` encodes and then decodes all the test cases and checks if they match using the tables sizes 0, 1 and 9. The test cases currently only pass the test with table size 0 because that encoding will not change the input text. The tests fail with table size 1 and 9 because the decoder template is not functional. The regular tests do not contain invalid list indices or stray *'s or ~'s. A good defensively written decoder should still check for these cases of course.

Bonus assignments

Order by frequency

The regular encoder appends starred words to the front of the word list. But there is also a version called `freq-encoder` that orders the list of starred words by their frequency. This mode has a slightly higher compression ratio as frequently occuring words will have lower indices. It saves a couple of characters when you use index 3 instead 33 or 101. The starred word list is ordered by frequency, and for words with the same frequency by decreasing word length. When these are all equal the alphabetical ordering is used. To decode such a file the decoder also needs to keep it's own starred-list ordered in the same way. Implement this method and test your program with test cases generated by the `freq-encoder`.

Write your own encoder

You can also decide to write your own encoder for this assignment. Make sure it behaves the same as the encoder provided for this assignment.

Improve the compression ratio

Instead of implementing the method described here, maybe you can improve on the encoding method and achieve better compression ratios than our encoder. Measure the compression ratio's for the large test cases with different table sizes. Think how the text replacement method could be improved, and then implement an encoder and decoder that uses your own replacement scheme and see if you can beat the best compression ratio achieved by the freq-encoder.