CSE 326: Data Structures
Assignment #2
April 9, 1999
Due: Monday, April 19

A Compression Algorithm

A self-adjusting list is a list data structure which reorganizes itself dynamically based in the sequence of operations performed. One example of a self-adjusting list is a list that implements the "Move-To-Front" rule:

- When a new item is inserted, it is always inserted at the front of the list.
- When an item is accessed by a *find* operation, it is moved to the front of the list without changing the relative order of the other items.

The motivation for such a strategy (moving a requested item to the front of the list) is that if that element is referenced again in the near future, the cost of the *find* will be significantly lower, since it will be closer to the front of the list. Indeed, in many applications, there is "locality of reference" in the accesses to a list, i.e., when an item is referenced, it is much more likely to be referenced again in the near future, and hence the "Move-To-Front" Rule improves performance considerably.

This type of self-adjusting list forms the basis for the following **text compression algorithm**. The algorithm takes as input a document consisting of N words, only $k << N$ of which are distinct. Given the document, D, we constructed the compressed document $C(D)$ as follows:

1. In one pass over the document, the k distinct words in the document are inserted into a linked list L.
2. The initial segment of $C(D)$ is obtained by printing out the list L, from front to the back.
3. In another pass over the document, the rest of $C(D)$ is constructed as follows: For each consecutive word w in D:

   (a) Search the list L for the word w.

   (b) Let i be the position of w in the list L.

   (c) Append an efficient representation of the integer i followed by a separator character to $C(D)$.

(d) Move w to the front of the list L.

For example, if the document D is:

```
Humpty Dumpty sat on the wall
Humpty Dumpty had a great fall
all the kings horses and all the kings men
could not put Humpty Dumpty back together again
```

then $C(D)$ is:

```
Humpty
Dumpty
sat
on
the
wall
had
a
great
fall
all
kings
horses
and
men
could
not
put
back
together
again

0 1 2 3 4 5 5 5 6 7 8 9 10 8 11 12 13 4 4 4 14 15 16 17 14 14 18 19 20
```

For ease of understanding in the above example, the words in the printed word list (step 2 of the algorithm) are given one list element per line and are followed by an empty line, and the integers representing the locations of each word in the list have been given in ASCII digits using whitespace as a separator. A real compression algorithm would output a compact binary representation.

On receipt of the compressed document $C(D)$, the document can be reconstructed as follows:

1. Read the initial printed list from $C(D)$ and construct a corresponding linked list L.
2. In one pass over the the rest of $C(D)$, reconstruct D as follows:
   (a) Initialize D to the empty string
   (b) While $C(D)$ is not empty
      i. Read the next integer, say j, from $C(D)$.
      ii. Find the j-th element of the list L, say the word w.
      iii. Append word w to D, followed by a space.
      iv. Move w to the front of the list L.

An alternative compression algorithm

An alternative to self-adjusting lists would be to use a list where the words are ordered according to their frequency in the document, so the most frequently occurring word is at the front of the list, the second most frequently occurring word is next and so on.

This idea yields the following compression algorithm. Given the document, D, the compressed document $C(D)$ is constructed as follows:

1. Construct a linked list L of the k distinct words in the document D ordered by decreasing frequency of occurrence in the document.
2. The initial segment of $C(D)$ is obtained by printing out the list L, from the front to the back.
3. In another pass over the document, the rest of $C(D)$ is constructed as follows: For each word w in D
   (a) Let i be the position of w in the list L.
   (b) Append an efficient representation of the integer i followed by a separator character to $C(D)$.

Project

For this assignment you are to implement the two compression algorithms just described. The first should be a program called `mtfzip` (Move-To-Front Zip). Your program should work as follows:

```
orcas% mtfzip file.txt file.mfz
Input size: N bits
Output Size: M bits
Compression Ratio: X%
orcas%
```

Here, N is the number of bits in the input, M is the number of bits in the output file, and X is the compression ratio $X = 100M/N$. `file.txt` is the name of the input file and `file.mfz` is the name of the output file (the extensions `.txt` and `.mfz` are not required).

Also implement another program `fszip` (Frequency Sort Zip) that uses the second compression algorithm discussed. The output of `fszip` should be in the same format as `mtfzip`.

Your solution may treat all types of contiguous whitespace in the input document as a single space. Also you do not have to support punctuation characters like `'`, `.`, etc. You program should work on documents that have no punctuation, where the words are formed from any of the 26 alphabetical characters.

For computing the number of bits in the input and the output, you can assume that the input is a sequence of ASCII characters and that each of these is represented using one byte. Both compression programs should output a compact binary representation for the integers (representing the positions of words in the list). This representation must have the property that the integer values can be recovered unambiguously from $C(D)$ in decompression.

Electronic Turnin

There will be an electronic turnin for your programs. To turn it in, on orcas run the `turnin` command like:

```
turnin -v -c cse326 Makefile *.cpp *.h
```

or a whole directory with:

```
turnin -v -c cse326 hw2/
```

Feel free to turn in a `README` file as well. It is very important that you turn in all of your files at once. **If you forget some files the first time then turn everything in again. The `turnin` program only keeps your last turn in.** Do not turn in object files `.o` or your compiled main programs.

Paper Turnin

The following should be turned at the beginning of class on Monday, April 19:

1. A hardcopy of your complete source code.
2. A written discussion of algorithmic and data structure choices that you made in your implementation. Also, give the asymptotic complexity of your program.
3. A brief explanation of the encoding you are using for the integers.
4. A written comparison of the compression performance of `mtfzip` and `fszip`. In particular, try to describe input patterns such that `mtfzip` gets better compression that `fszip` and vice versa.
5. In implementing your compression program, you will be implementing several data structures. Provide a test program and output that demonstrates that these data structures work.

Extra Credit

For extra credit, you can:

1. Implement the decompression algorithm.
2. Extend your program to properly deal with different types of whitespace and punctuation characters. With your paper turnin, explain how you are solving this problem and how you would reconstruct D from $C(D)$.