# Department of Electrical and Computer Engineering

# ELECTENG 203FC   Software Design 1E  Project

**Due Date:  10 am 31 May 2004**

## 1. Background

It is often useful to store a group of files so that they can be 'backed up', easily transferred to another directory or to a different computer. This process is known as *archiving*.
During the archiving process, the original data files are compressed into one shorter file so that they use less disk space and download faster.  This is known as *file compression*.
WinZip, arj, pkzip, gzip etc are examples of commonly used utilities which perform the archiving and compression/decompression.

## 2. Project Purpose

The purpose of this project is to implement a simple text file compressor and archiver under Linux.  Text files in a specified directory must be compressed and archived in such a way that the original files can be later recovered on a different computer.

## 3. Functional Objectives

3.1 Archive text files, with a specified file extension, in a specified directory.

3.2 The files inside the archive should be in a compressed form.  Files must be compressed using the **Lempel-Ziv Storer Szymanski (LZSS) algorithm** as outlined in appendix A.

3.3 The data in the archive must be able to be decompressed and the files saved in their original directory.

## 4. Operating Conditions

The utility must run under the Linux operating system.  It must be invoked from the command line under a terminal window (absolutely no graphical interfaces).  Command line switches should be implemented to select various options.

### 4.1 Command line options

The utility must be invoked using the following command line syntax:

```
lzss -c|-d  archive  filelist ...
```

**Notes:**

(a) `-c|-d` is the first parameter expected after the executable filename. This parameter is mandatory. The bar | indicates one **or** the other parameter must be present.
`-c` means to create a compressed archive (*archive*) of the files specified in `filelist`.
`-d` means to restore the files, from the archive, to their original directory as specified in *archive*.

(b) *archive* is the name of the archive file which is to be created when using the `-c` switch. The archive name is mandatory and always follows the `-c` or `-d` switch.

(c) The *filelist* comprises a list of filenames which are to be added to the archive when using the `-c` switch. A space separates each file in the list. Wildcards such as `*.c` for all files ending in `.c` for example are also allowed.

(d) Incorrect number of parameters or invalid command line switches must be detected and appropriate action taken to ensure the program does not crash or produce unexpected results.

## 4.2 Examples invoking from the command line

```
lzss -c myfiles file1.c file2.c file3.c
```
create archive name 'myfiles' and store file1.c, file2.c and file3.c inside the archive.

```
lzss -d myfiles
```
unpack files file1.c, file2.c and file3.c from the 'myfiles' archive.

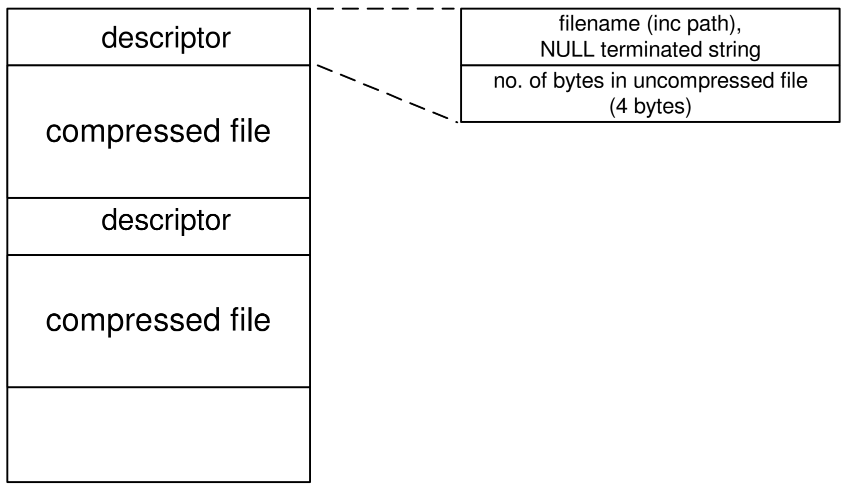
```
lzss -c onlytxt  document/*.txt
```
create an archive named 'onlytxt' and store in it all files ending in .txt from the document directory.

```
lzss -d onlytxt
```
unpack files from onlytxt archive, creating the directory document (if it doesn't already exist) and storing the all files ending in .txt in that directory.

```
lzss -h
```
show help informing user about format of command line options.

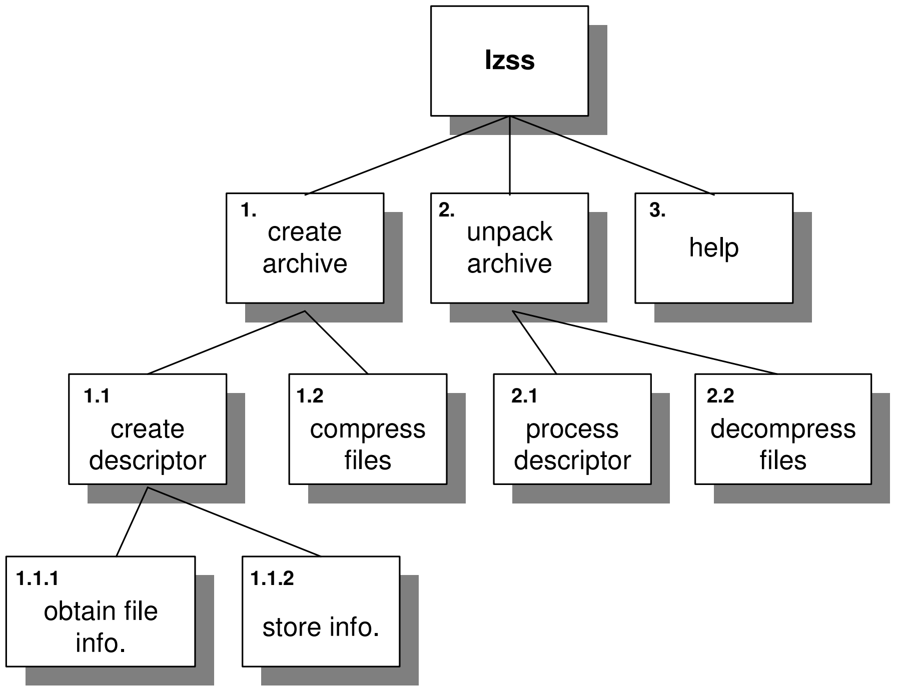
## 5. Structure of the archive file

In simple terms, the archive file is a collection of compressed data files separated from each other by a *descriptor*. The *descriptor* contains information about the path/name of the original file and the length of the file so that it can be decompressed correctly. The filename and path is an ASCII string. The length of the file is stored as 4 bytes enabling text files of up to $2^{32}$-1 characters to be compressed. The first MSbyte of the length is stored first and the LSByte is stored last. Graphically, the structure of the archive file appears as below.


descriptor
compressed file
descriptor
compressed file
filename (inc path), NULL terminated string
no. of bytes in uncompressed file (4 bytes)

## 6. Design and Implementation

A diagram showing the main components of the utility and their relationship to other components is shown below.



Each block in the hierarchy represents a task which is a simpler component of the parent task. Blocks do not necessarily translate directly into C functions.

**Specifications of the main blocks**

*1. Create archive*

If the user selects the –c switch in the command line, this block will be performed. Creating the archive involves creating the file descriptor for a file within the specified directory and then compressing that file into the archive.

*1.1 Create descriptor*

This block involves determining the length of the file to be compressed and then writing the name of the file and file length to the archive file.

*1.1.1 Obtain file info*

This block primarily involves getting the length of the file to be compressed. The name of the file can be obtained directly from the *argv* parameter.

*1.1.2 Store info*

This block involves writing the info as the file descriptor in the archive file.

*1.2 Compress files*

Each file specified on the command line is compressed using the LZSS algorithm (see Appendix A). The compressed file is written to the archive file.

*2. Unpack archive*

If the user selects –d switch on the command line, this block will be performed. Unpacking the archive involves reading each header from the archive and then decompressing the encoded text.

*2.1 Process descriptor*

The block involves reading the *descriptor* information (filename and file length) from the archive file, and creating a new destination directory if it doesn't already exist.

*2.2 Decompress files*

This block involves decoding the compressed text in the archive using the LZSS algorithm.
The decompressed file is written to the destination directory.

*3 Help*

This block prints a message on the screen when the user does not enter the correct number if parameters or the parameters are not in the specified order.

## 7. Design Strategy

It is very important that you **design** your program on paper before you begin coding. Putting in a bit of effort to think about *structure* will save you grief in the long term. There will be parts of the project that you cannot implement until the theory is covered in lectures so we will give you some general guidance as to what is important and what can wait for later.

Design the overall structure of the program. This means considering which tasks are best implemented as a C *function*. Remember a well structured program should have *modularity*. If there are functions that you cannot implement at present, leave them as **stubs** (i.e. functions which just return 'dummy' data). Convert the design into pseudocode and flowcharts which can later be converted into C code.

The implementation of this project is best tackled in two stages.

**Stage 1**: You should initially implement the project without the compression and decompression functions as these are tricky and it will be better to leave them until you have more confidence with the debugger. For stage 1 you can replace blocks 1.2 and 2.2 in the hierarchy with simpler functions that just copy text from one file to the archive and vice-versa. The end result of stage 1 will be a fully-functioning archiver.

**Stage 2**: Compression and decompression using LZSS

Start by implementing the compression routine first. Choose a very small window (say 20 characters long) and a smaller lookahead buffer (say 4 characters) so you can easily test and debug the function. Perform a 'desk-check' to check that the encoded characters are what you would expect them to be. Once all seems right, increase the length of the window and lookahead buffer. Use 512 characters for window and lookahead buffer and 16 character lookahead buffer.

**Integration and Testing**:

Replace the blocks 1.2 and 2.2 with compression and decompression functions. Test the program on some relatively large text files. Do not use binary files as they are unlikely to work.

## 8. Submission

You are required to hand-in for marking **TWO** source files. One source file should be the archiver (without compression/decompression) and the second file should be the archiver program with compression and decompression. Each source file should be fully documented according to the format described in lectures.

A **printed copy of your source files** should include a cover page with signed *declaration of originality*. Please hand this into the student services stall on level 3 before the due date.

You must also electronically submit both source files to the Assignment Dropbox.

Please **do not** use a word processor (such as MS Word) for formatting comments after you have written the program. Comments should be written directly into the source document using KEdit or KDevelop only. A MS Word document is **not** a text file and therefore cannot be recompiled.

Failure to hand-in/submit either printed copy of your source code or source code file will result in an automatic zero mark for the project.

## 9. Marking Schedule

Marks will be allocated in two broad categories. There are no additional marks for features which are over and above the specifications in this document. Use the following checklist as a guide to your achievements.

**1. Functionality** – i.e. how well does your program work ?

- ☐ Command line arguments handled correctly
- ☐ Archives single file correctly
- ☐ Archives multiple files correctly
- ☐ Compresses single file correctly
- ☐ Decompresses single file correctly
- ☐ Archives and compresses multiple files
- ☐ Decompresses and unpacks multiple files
- ☐ All directory paths are recreated correctly during unpacking
- ☐ Completeness of the project (i.e. how much actually works)

**2. Quality of the programming** – e.g. efficiency, documentation.

- ☐ Well formed, easy to read prologue comments
- ☐ Comments explain the purpose of the code
- ☐ Efficient use of loops and selections
- ☐ Source code is printed neatly
- ☐ Variable names are meaningful
- ☐ Appropriate number of functions and good modularity
- ☐ No spelling errors
- ☐ Good use of indenting blocks of code for readability

Approximately half of the project marks will be derived from functionality and the other half from quality of the programming and documentation.

## Appendix A: Implementing the LZSS algorithm

The Lempel-Ziv algorithm achieves compression because of the fact that most text files have repetitive strings of text. It would be more compact to represent the repeated text by a coded word (*pointer* – but not a C pointer) rather than repeat the text every time.

### A simple example

Consider the following short text file.

```
The quick brown fox jumps over the lazy dog. The fox jumps high while
the dog lies still.
```

There are 89 characters in the file (incl. spaces). We would consider this file as *uncompressed*. By inspection of the above text, it is possible to identify repetitive strings of text. For example, the string "`fox jumps`" appears twice in the text, as does the words "`The `" and "` dog`". We could achieve compression by replacing the occurrence of "`fox jumps`" by a short *pointer*. The pointer would be a coded-pair containing offset from the start of the file and length of match. Thus, we could replace the second occurrence of "`fox jumps`" by <16,9> where the letter 'f' appears at offset 16 from the start of the file and the match is 9 characters long. (Note we begin from offset 0 in the file). Similarly, the text "`The `" would be replaced with <0,4> and "` dog`" replaced with <39,4>. A similar code can be generated for other matching text phrases.

If we were to view this compressed file it might look like,

```
The quick brown fox jumps over the lazy dog. <0,4><16,9> high while
the<39,4> lies still.
```

Assuming each character is 1 byte, and the pointer is coded as 2 bytes (1 byte for offset and 1 byte for length of match) then in this example, the compressed file would be **77** bytes long. Larger files tend to have much more repetition and thus better compression is achieved than in this short file.

### The Lempel-Ziv Storer Szymanski (LZSS) algorithm

The example above is rather inefficient because all characters must be searched for a possible match. This is very time consuming for a large file. A better approach is to only search a smaller 'window' of characters and to only match a certain maximum number of characters. This means the offset and match length can be coded into fewer bits. As characters are read from the file they are entered into a short *lookahead buffer* and it is this buffer which is matched against characters already in the window. As new characters are read from the file, they are inserted into the lookahead buffer and the window and lookahead buffer are shifted one byte to the right. This is known as a sliding window.

## An example of LZSS

Consider a window length of N=20 characters and a lookahead buffer of F=4 characters.

The file to be compressed is,

`abcdefghijklmnopdefg1234opd5678`

You will note that there are two obvious repetitive patterns in the text, "`defg`" and "`opd`".
The window is implemented in C as an array of size 20 characters. Two indices are used to point to the start of the window (`wp`) and start of the lookahead buffer (`lk`). As a new character is read from the file, the indices are incremented which has a similar effect as moving the window one character to the right.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | | **wp** | **lk** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | **a** | **b** | **c** | **d** | | 0 | 16 |
| **e** | | | | | | | | | | | | | | | | a | **b** | **c** | **d** | | 1 | 17 |
| **e** | **f** | | | | | | | | | | | | | | | a | b | **c** | **d** | | 2 | 18 |
| **e** | **f** | **g** | | | | | | | | | | | | | | a | b | c | **d** | | 3 | 19 |
| **e** | **f** | **g** | **h** | | | | | | | | | | | | | a | b | c | d | | 4 | 0 |
| e | **f** | **g** | **h** | **i** | | | | | | | | | | | | a | b | c | d | | 5 | 1 |
| e | f | **g** | **h** | **i** | **j** | | | | | | | | | | | a | b | c | d | | 6 | 2 |
| e | f | g | **h** | **i** | **j** | **k** | | | | | | | | | | a | b | c | d | | 7 | 3 |
| e | f | g | h | **i** | **j** | **k** | **l** | | | | | | | | | a | b | c | d | | 8 | 4 |
| e | f | g | h | i | **j** | **k** | **l** | **m** | | | | | | | | a | b | c | d | | 9 | 5 |
| e | f | g | h | i | j | **k** | **l** | **m** | **n** | | | | | | | a | b | c | d | | 10 | 6 |
| e | f | g | h | i | j | k | **l** | **m** | **n** | **o** | | | | | | a | b | c | d | | 11 | 7 |
| e | f | g | h | i | j | k | l | **m** | **n** | **o** | **p** | | | | | a | b | c | d | | 12 | 8 |
| e | f | g | h | i | j | k | l | m | **n** | **o** | **p** | **d** | | | | a | b | c | d | | 13 | 9 |
| e | f | g | h | i | j | k | l | m | n | **o** | **p** | **d** | **e** | | | a | b | c | d | | 14 | 10 |
| e | f | g | h | i | j | k | l | m | n | o | **p** | **d** | **e** | **f** | | a | b | c | d | | 15 | 11 |
| e | f | g | h | i | j | k | l | m | n | o | p | **d** | **e** | **f** | **g** | a | b | c | d | | 16 | 12 |
| e | f | g | h | i | j | k | l | m | n | o | p | d | e | f | g | **1** | **2** | **3** | **4** | | 0 | 16 |
| **o** | f | g | h | i | j | k | l | m | n | o | p | d | e | f | g | 1 | **2** | **3** | **4** | | 1 | 17 |
| **o** | **p** | g | h | i | j | k | l | m | n | o | p | d | e | f | g | 1 | 2 | **3** | **4** | | 2 | 18 |
| **o** | **p** | **d** | h | i | j | k | l | m | n | o | p | d | e | f | g | 1 | 2 | 3 | **4** | | 3 | 19 |
| **o** | **p** | **d** | **5** | i | j | k | l | m | n | o | p | d | e | f | g | 1 | 2 | 3 | 4 | | 4 | 0 |
| o | p | d | **5** | **6** | **7** | **8** | l | m | n | o | p | d | e | f | g | 1 | 2 | 3 | 4 | | 7 | 3 |
| o | p | d | 5 | **6** | **7** | **8** | | m | n | o | p | d | e | f | g | 1 | 2 | 3 | 4 | | 8 | 4 |

The top row of the table shows the index corresponding to each cell in the window.
The shaded cells represent the part of the window to be searched. The white cells represent the content of the lookahead buffer. The starting index of the window is given by the `wp` column and the starting index of the lookahead buffer is given by the `lk` column.

Initially, the window is filled with spaces (blanks) and the lookahead buffer is filled with the first 4 characters from the file. There is no match in this case, so the first character 'a' is written to the compressed file. The wp and lk pointers are incremented and the next character 'e' is read from the file. This character overwrites the *oldest* character in the window. When `wp=4`, the `lk` pointer is incremented modulo N to that it wraps around the start of the buffer. When `wp=16`, the string "`defg`" is in the lookahead buffer. This matches exactly with the string in the window starting at offset 3 relative to the start of the window, `wp`. This is then encoded as <3,4> and written to the compressed file. The pointers are then advanced modulo N by 4 characters. The next match occurs when the lookahead buffer contains "`opd5`". In this case the match length is 3 and the offset of the match relative to the start of the window is 6.

The contents of the compressed file would be :

```
a b c d e f g h i j k l m n o p <3,4> 1 2 3 4 <6,3> 5 6
```

## LZSS Implementation Guidelines

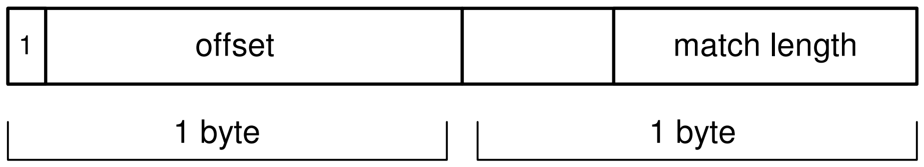
1. In this project the sliding window is to be implemented as an array of N characters. The lookahead portion of the window is F characters. The size of the search window is N-F characters.

2. Don't forget you need to increment the window pointer using modulo N arithmetic so that it wraps around to the start of the array.

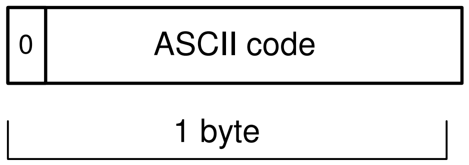
3. The lookahead pointer can be *derived* from the window pointer, thus avoiding the need for a separate variable for `lk`.

4. An encoded string is represented by 2 bytes. The first byte contains the offset of the first matching character in the window. For a window of size 512, this field must be 9 bits long. A 1 in the most-significant bit position indicates an encoded-pair and a 0 in the MSbit represents an unencoded ASCII character. The second byte contains the remainder of the offset (2 bits) and the length of match. For a lookahead buffer size of 16, there must be 4 bits for this field.

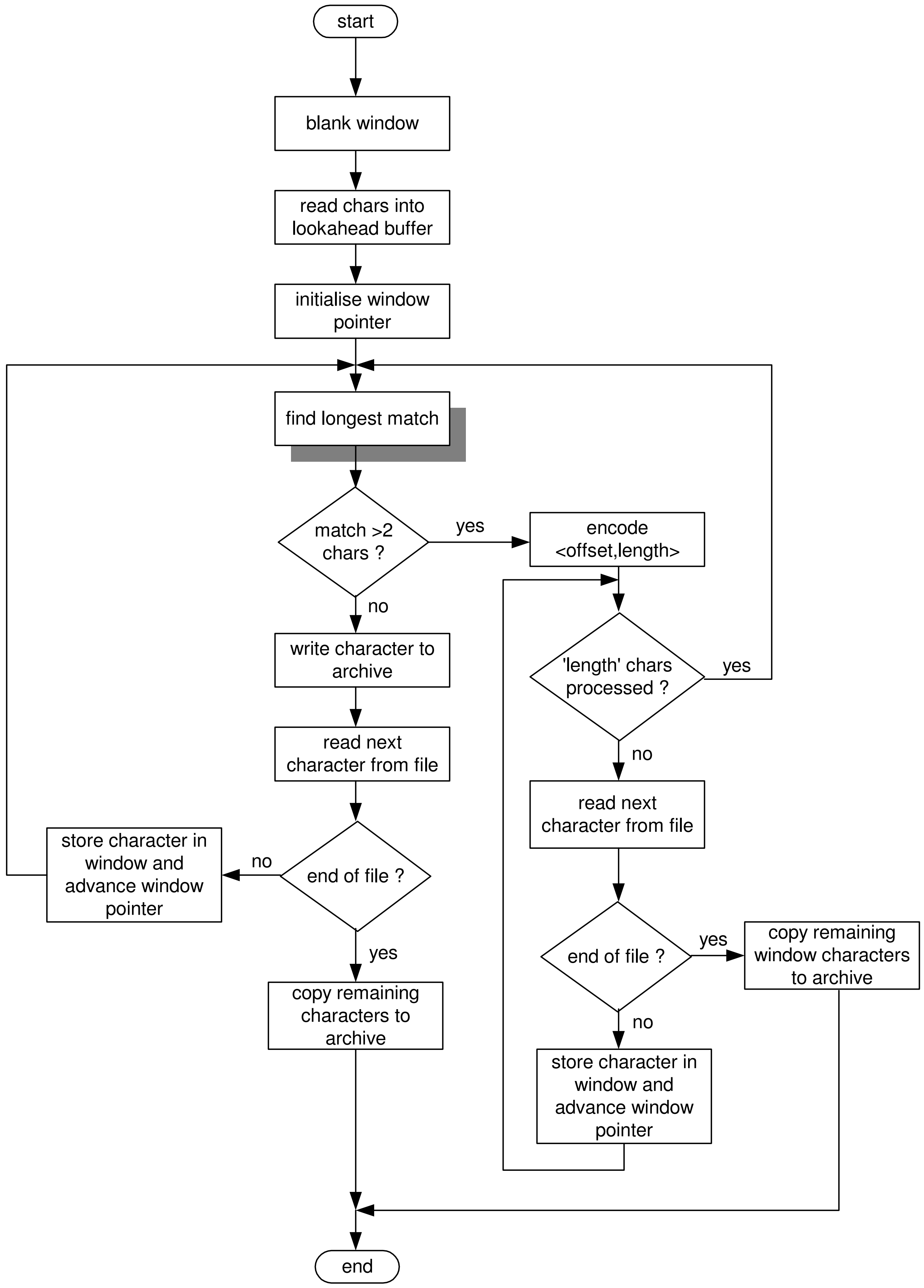
   (a) Encoded pair



   (b) Unencoded character



**Flowchart of LZSS algorithm:**


start
blank window
read chars into lookahead buffer
initialise window pointer
find longest match
match >2 chars ?
yes
encode <offset,length>
no
write character to archive
'length' chars processed ?
yes
no
read next character from file
read next character from file
store character in window and advance window pointer
no
end of file ?
end of file ?
yes
copy remaining window characters to archive
yes
no
copy remaining characters to archive
store character in window and advance window pointer
end