Swinburne University of Technology

School of Science, Computing and Engineering Technologies

MIDTERM COVER SHEET

Subject Code: COS30008
Subject Title: Data Structures and Patterns
Assignment number and title: Midterm
Due date: Thursday, April 27, 2023, 23:59
Lecturer: Dr. Markus Lumpe

Your name:_____ **Your student ID:**_____

Check Tutorial	Tues 08:30	Tues 10:30	Tues 12:30 BA603	Tues 12:30 ATC627	Tues 14:30	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30	Thurs 08:30	Thurs 10:30

Marker's comments:

Problem	Marks	Obtained
1	52	
2	74	
3	108	
Total	234	

Extension certification:

This assignment has been given an extension and is now due on _____

Signature of Convener:_____

Midterm: LZW Compression

In 1984, Terry A. Welch published a paper titled "A Technique for High-Performance Data Compression". It describes the LZW compression algorithm, an improvement of LZ77 and LZ78 previously invented by Abraham Lempel and Jacob Ziv. LZW is a lossless compression algorithm that uses a string table to record the string sequences encountered in the data. The table usually has 4096 entries, which gives rise to 12-Bit codes. The first 256 entries are prepopulated with the characters of the 8-Bit ASCII alphabet. At every stage of the algorithm, the recognized input sequence is a prefix string occurring in the string table until the next character yields a sequence with no code. In this case, the algorithm inserts the new string into the table, makes the last character the new prefix, and outputs the code for the sequence (without the last character). This process is repeated until all input data has been processed.

For the purpose of this midterm, we apply two changes. First, we reduce the table size to 1024 entries. This means, we generate 10-Bit codes. In addition, a table with 1024 entries can be stored on the stack. So, no extra C++ memory management is needed. Second, we limit the input alphabet to 7-Bit ASCII. Hence, the algorithm uses only the first 128 entries for prepopulated data. Operationally, indices 128 to 1023 remain for storing prefix strings.

The LZW algorithm requires two dedicated data types: prefix strings and LZW string table. The LZW table has a prefix property that guarantees that for every string in the table its prefix string is also in the table. If a string ωK, composed of some string ω and some character K, is in the table, then ω is in the table. The exception is single character strings. They do not have a prefix. We use -1, no prefix, to denote this fact programmatically. Single character strings correspond to the prepopulated entries for ASCII characters.

LZW uses a greedy parsing process in which the data is examined character-serially in one pass, and the longest recognized input string is parsed off each time. A recognized string is one that exists in the table. Strings are added to the table when the parsed input string extended by the next character forms a new string. Each added string is assigned a unique code, the index into the LZW string table where this string is stored.

The following pseudo code (due to Terry A. Welch) describes the LZW compression algorithm:

```
      Initialize table to contain single-character strings.
      Read first input character K and assign it to prefix string ω.
Step: Read next input character K.
      If no such K (input exhausted):
            Send code(ω) to output; EXIT.
      If ωK exists in string table:
            Assign ωK to ω; repeat Step.
      Else (ωK not in string table):
            Send code(ω) to output;
            Add ωK to string table;
            Assign K to ω; repeat Step;
```

Figure 1 below illustrates the process. It relies on the 7-Bit ASCII encoding for characters 'a', 'b', and 'c', which is 97, 98, and 99, respectively.

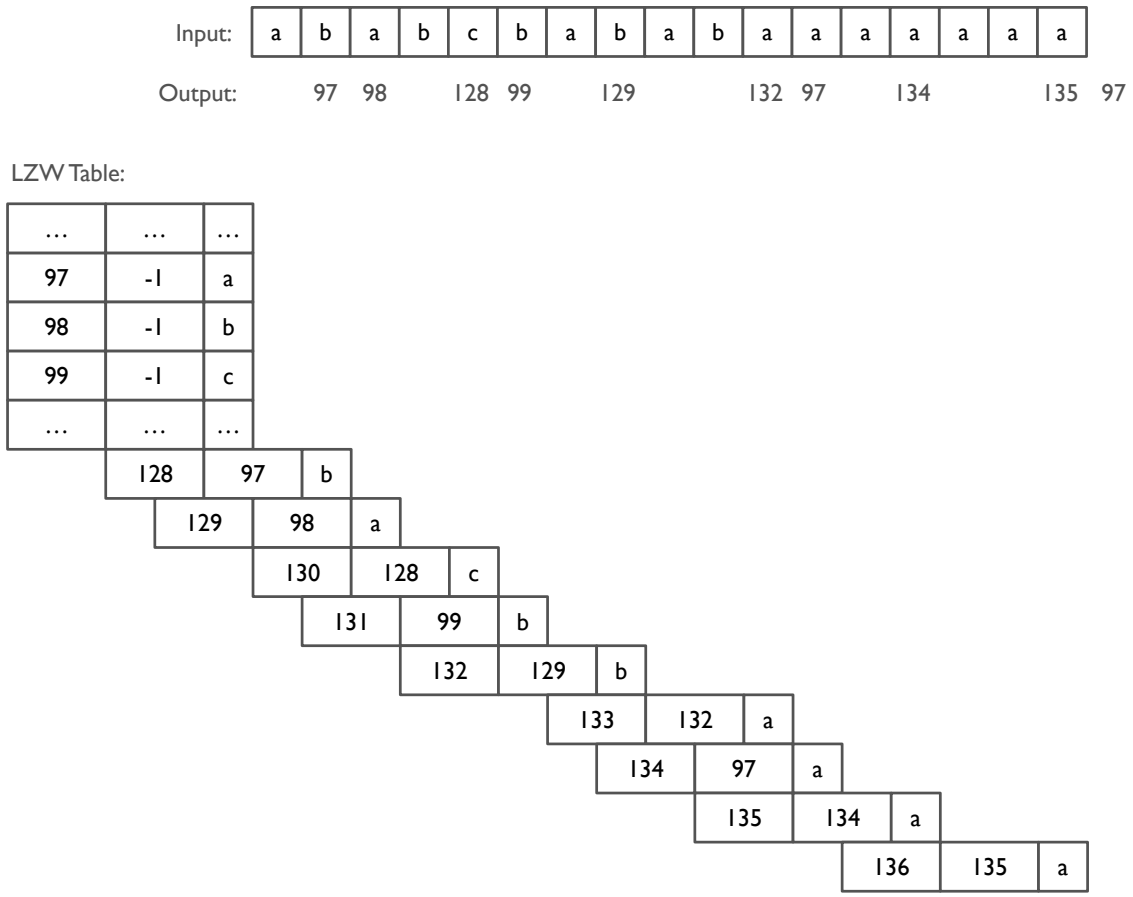

Figure 1: LZW Compression Example.

The LZW string table is prepopulated with 128 entries. At position 97, 98, and 99, we find the single-character strings for 'a', 'b', and 'c' which have no prefix (i.e., -1).

We consider the input sequence "ababcbababaaaaaaa", a string with 17 characters.

The algorithm starts by reading the first character K = 'a' and sets the prefix to ω = (-1, a). We enter the loop. Next, we read K = 'b' and create a local prefix string ω' = ωK. The value of ω' is (97, b), which corresponds to the current input "ab". The prefix string ω' does not exist in the LZW table. Hence, we add it. Its code becomes 128. We output code 97 and set ω = (-1, b), the prefix of the last character parsed and continue. Now, we read the next character K = 'a', which results in a local prefix string ω' = (98, a). Again, this string is not in the table. We add it and output code 98 and set ω = (-1, a).

If ω' is in the string table, we keep reading input. This happens for the second sequence of "ab". Its prefix string has index 128. Parsing the second 'a' results in ω = (-1, a). Reading the fourth character K = 'b' yields a local prefix string ω' = (97, b) which has code 128. So, we need to parse the fifth character K = 'c'. We obtain a new string ω' = (128, c), which is assigned code 130. We output 128 and set prefix ω = (-1, c).

The last code is emitted after the input is exhausted. Please note the output lag. The prefix at this point is ω = (-1, a). Its code is 97. So, we need to output 97. The algorithm stops.

When we use an iterator for the implementation of LZW compression, we need to set ω = -1 at this point also. This is a programmatic trick to signal that we have processed all characters and all codes. A LZW iterator is at the end, if and only if, all characters have been parsed and all codes have been sent to output and no further character K is available for processing.

The actual implementation requires three abstractions: `PrefixString`, `LZWTable`, and `LZWCompressor`. The latter is a standard forward iterator.

This midterm is comprised of three problems. The test driver (i.e., `Main.cpp`) uses `P1`, `P2`, and `P3` as variables to enable/disable the test associated with a corresponding problem. To enable a test just uncomment the respective **`#define`** line. For example, to test problem 2 only, enable **`#define`** `P2`:

```
// #define P1
#define P2
// #define P3
```

In Visual Studio, the code blocks enclosed in **`#ifdef`** `PX` … **`#endif`** are grayed out, if the corresponding test is disabled. The preprocessor definition **`#ifdef`** `PX` … **`#endif`** enables conditional compilation. XCode does not use this color coding scheme.

Problem 1

The LZW compression algorithm relies on an abstraction called prefix strings. Technically, prefix strings are tuples. To increase the utility of the abstraction, we also incorporate the actual code of the prefix string into the tuple. For example, we write (97, 65535, a) to represent the single-character prefix string for 'a'. Please note that 65535 stands for -1 when it is interpreted as `unit16_t`, unsigned 16-Bit short integer. Similarly, we write (128, 97, b) to mean the prefix string "ab" stored at index 128 in the LZW table.

The codes for prefix strings do not exceed 2^{10}. For this reason, we use `unit16_t` as the underlying data type. We often wish to use -1 to denote "no prefix". The value -1 is a signed integer that does not really work with `unit16_t`. So, we need to use a static type cast for the value -1 to convert it to `uint16_t`: **`static_cast`**`<unit16_t>(-1)`.

The following class specification suggests a possible solution for prefix strings:

```
#pragma once

#include <cstdint>
#include <ostream>

class PrefixString
{
private:
  uint16_t fCode;
  uint16_t fPrefix;
  char fExtension;

public:
  PrefixString( char aExtension = '\0' ) noexcept;
  PrefixString( uint16_t aPrefix, char aExtension ) noexcept;

  uint16_t getCode() const noexcept;
  void setCode( uint16_t aCode ) noexcept;

  uint16_t w() const noexcept;
  char K() const noexcept;

  PrefixString operator+( char aExtension ) const noexcept;
  bool operator==( const PrefixString& aOther ) const noexcept;

  friend std::ostream& operator<<( std::ostream& aOStream,
                                   const PrefixString& aObject );
};
```

Class `PrefixString` defines two constructors, the first has a default argument. The first constructor initializes single-character strings. Single character prefix strings use -1 as unsigned value for `fPrefix`. Both constructors have to initialize all member variables. Initially, `fCode` must be set to -1 as an unsigned value.

Prefix strings allow their code to be fetched and updated. The latter is required when we perform a lookup in the LZW table or add a prefix string to the LZW table.

The getters `w()` and `K()` work in the expected way. The function `w()` returns the prefix, whereas `K()` returns the extension. In the LZW terminology, K is called the extension to prefix string ω. So, any `PrefixString` object effectively represents a value ωK.

The **`operator+`**`()`, prefix string extension, creates a new prefix string. This operation is only allowed on `PrefixString` objects whose `fCode` is not equal to -1. That is, prefix strings that are already present in the LZW table. Prefix string extension creates a new prefix string whose ω component is the receiver object's code (i.e., `fCode`) and K is set to the extension character. For example, single-character prefix string (97, -1, a) extended by extension

character 'b' is (97, -1, a) + 'b' = (-1, 97, b). The code of the result must be -1 as we do not yet know whether or not this string is in the LZW table.

The **`operator==`**`()` defines equivalence of prefix strings. The equivalence of codes is implied by the equivalence of prefix strings. That is, $\omega_1 K_1 == \omega_2 K_2$, if $\omega_1 == \omega_2$ and $K_1 == K_2$. We must not test the code. So, (128, 97, b) == (-1, 97, b) is true. That is, (-1, 97, b) represents that same prefix string as (128, 97, b). In (-1,97,b) we have not yet set the code.

The output **`operator<<`**`()` assigns `PrefixString` objects a textual representation. It yields a comma-separated list with no spaces enclosed in parentheses.

To test your implementation of `PrefixString`, uncomment **`#define`** `P1` and compile your solution.

The test driver should produce the following output:

```
Test PrefixString:

0 string ::= code= 65535, w = 65535, K =
A string ::= code= 97, w = 65535, K = a
BA string ::= code= 128, w = 98, K = a
lW == lStringBA? true
All strings:
lString0 = (65535,65535,)
lStringA = (97,65535,a)
lStringB = (98,65535,b)
lStringAB = (127,97,b)
lStringBA = (128,98,a)
lW = (128,98,a)

PrefixString test complete.
```

Problem 2

In the LZW compression algorithm, the LZW table serves as dictionary of prefix strings. The prefix property guarantees that for every string in the LZW table its prefix string is also in the LZW table. The exception is single character strings. They do not have a prefix. The prefix property must hold at all times.

Using instances of class `PrefixString`, we can define class `LZWTable` as follows:

```
#pragma once

#include <cstdint>

#include "PrefixString.h"

class LZWTable
{
private:
  PrefixString fEntries[1024];
  uint16_t fIndex;
  uint16_t fInitialCharacters;

public:

  LZWTable( uint16_t aInitialCharacters = 128 );

  void initialize();

  const PrefixString& lookupStart( char aK ) const noexcept;

  bool contains( PrefixString& aWK ) const noexcept;
  void add( PrefixString& aWK ) noexcept;
};
```

Class `LZWTable` allows for some extra flexibility. We can use any number of initial single-character prefix strings. However, as a constraint of this midterm, it must always be limited to 128.

The value `fIndex` represents the next available slot in `fEntries`. We may exhaust the LZW table eventually. However, in the midterm we can safely assume, that the table is large enough to store all required prefix strings. No range check is required for adding prefix strings.

The constructor has to initialize all member variables and populate the single-character prefix strings. The 7-Bit ASCII table defines 128 characters which need to be represented as single-character prefix strings. The codes of the single-character prefix strings have to be set accordingly. Ideally, this setup should be implemented in method `initialize()`. This method is public for a reason. A client of `LZWTable` may need to call `initialize()` also.

The method `lookupStart()` returns a reference to the single-character prefix string that corresponds to argument `aK`. The lookup has to guarantee that there exists such a prefix string. Rely on standard practice for this requirement. An explicit type conversion from `char` to `unit16_t` is optional. The C++ compiler does it implicitly.

The method `contains()` checks whether a given prefix string is already in the LZW table. The argument is a read-write reference to a `PrefixString` object. The method must check a safety requirement. The argument prefix string must have a valid ω component. That is, `contains()` cannot be called with single-character prefix strings as arguments. The method `contains()` searches backwards in `fEntries`. The search ends if a match is found or if the search index reaches the value of the ω component. The ω component is a lower bound for searches in the LZW table. Remember, if a string ωK, composed of some string ω and some character K, is in the table, then ω is in the table. If a match is found, then `contains()`

copies the match into the reference argument and returns true. Otherwise, method `contains()` returns false.

Finally, method `add()` inserts a prefix string at the end of the LZW table. Again, the argument prefix string must have a valid ω component. That is, `add()` cannot be called with a single-character prefix string as argument. Upon a successful insertion of the given prefix string, the method `add()` must set the code of the argument prefix string to the insertion index. For this reason, the argument is a read-write reference to a `PrefixString` object. Think about the order here. What is the correct approach?

To test your implementation of `LZWTable`, uncomment **`#define`** `P2` and compile your solution.

The test driver should produce the following output:

```
PrefixString test complete.
Test LZW Table:

LZW Table contains 128 entries.
Next available index is 128.
lA = (97,65535,a)
Is lW_1 = (65535,97,b) in LZW table? No.
lW_1 = (128,97,b)
Is lW_2 = (65535,97,b) in LZW table? Yes.
lW_2 = (128,97,b)

LZWTable test complete.
```

Problem 3

Finally, we wish to define a forward iterator that performs LZW compression. We are only interested in the LZW code sequence, not an implementation that literally compresses input data.

The iterator for LZW compression has to maintain a number of instance variables to allow for a faithful mapping of Terry Welch's algorithm:

- `fTable`: the LZW table,
- `fW`: the current prefix string ω,
- `fInput`: the input data (a `std::string`),
- `fIndex`: the iterator index into the input data,
- `fK`: the current input character K, and
- `fCurrentCode`: the code of the current prefix string (i.e, `code(ω)`).

Conceptually, a forward iterator that performs LZW compression is like any forward iterator, with one exception. In LZW compression, we are always one character ahead, see Figure 1. That is, the first code becomes available after we have parsed two characters already, and the last code becomes available one step after the last character has been parsed. This lag has to be properly factored in into the implementation of the iterator methods.

The specification of the forward iterator for LZW compression is given below:

```
#pragma once

#include <string>
#include <cstdint>

#include "PrefixString.h"
#include "LZWTable.h"

class LZWCompressor
{
private:
  LZWTable fTable;
  PrefixString fW;
  std::string fInput;
  size_t fIndex;
  char fK;
  uint16_t fCurrentCode;

  bool readK() noexcept;
  void start() noexcept;
  uint16_t nextCode() noexcept;

public:

  LZWCompressor( const std::string& aInput );

  const uint16_t& operator*() const noexcept;

  LZWCompressor& operator++() noexcept;
  LZWCompressor operator++(int) noexcept;

  bool operator==( const LZWCompressor& aOther ) const noexcept;
  bool operator!=( const LZWCompressor& aOther ) const noexcept;

  LZWCompressor begin() const noexcept;
  LZWCompressor end() const noexcept;
};
```

The crucial methods are the private member functions: `readK()`, `start()`, and `nextCode()`. The function `readK()` reads the next K, sets member variable `fK`, and returns

true if the input is not exhausted. If there is no such K, then `readK()` sets `fK` to -1 (i.e., EOF) and returns false. It is import that `fK` is set to -1 at the end. It signals that all input characters have been parsed, but there is still a final code that needs to be returned by the iterator.

The method `start()` implements the initialization part of the LZW compression algorithm. This methods should be called in the constructor and in `begin()`. The method `start()` has to initialize the LZW table, read the first K and set the prefix string ω (i.e, `fK` and `fW`), and use `nextCode()` to set `fCurrentCode`. The last step corresponds to the first iteration through the pseudo code loop starting at label `Step`.

The method `nextCode()` implements the pseudo code loop of the LZW algorithm. The features defined in `PrefixString` and `LZWTable` should allow you to faithfully map the pseudo code with one exception. The implementation requires a while-loop. However, the test "`If no such K`" has to be done twice in an iterator implementation. Once before the while-loop and as part of the while-loop condition testing. If it fails before the while-loop, then the iterator has to stop. The end condition has been reached and the next code is -1. The test "`If no such K`" can simply be mapped to `fK == -1`. If `fK != -1`, then there may be more characters in the input, so we use `readK()` and evaluate its return value as part of the while-loop condition. If `readK()` returns true, then the while-body follows the pseudo code sequence (`ωK` in the pseudo code requires a local variable in the while-loop body). If `readK()` returns false, then we do not enter the while-loop, but we have to set `fCurrentCode` to the code of the prefix string in `fW`, the `EXIT` condition in the pseudo code.

The constructor, as usual, initializes all member variables and then calls `start()` to move the iterator forward at least one character.

The deference operator simply returns the current code.

The increment operators advance the iterator by setting the next code. Follow standard practice for the implementation of both variants.

Two iterators are equal, if and only if, they refer to the same collection (i.e., input data) and are positioned on the same element. Here the same element test is composed of three features: the index into the input, the current K, and the current code. For not equal, follow standard practice.

The `begin()` and `end()` methods have to return corresponding iterator copies. These methods are part of the range-loop protocol. The method `begin()` has to return an iterator that is positioned at the first code, whereas method `end()` returns an iterator that cannot yield any more codes.

Please note that the LZW algorithm assumes at least one input character. This requirement is not tested in the iterator though. We simply require the input data to be well-formed, that is, to contains at least one character.

To test your implementation of `LZWCompressor`, uncomment **`#define`** `P3` and compile your solution.

The test driver should produce the following output:

```
Test LZW Compression:

Input String:ababcbababaaaaaaa
LZW Codes:
97
98
128
99
129
132
97
134
135
97

Compression Ratio: 1.36/1
Overhead in Bits: 4
Space Saving: 23.5294%

LZW Compression test complete.
```

The solution of this problem set requires approx. 260-280 lines of low density C++ code.

Finally, you can test your implementation with other input data. It should work as long as only 896 additional prefix strings are being generated, the capacity of the LZW table. Compression ratio and overhead may differ, however, as the actual values depend on the patterns in the input data. The performance of LZW compression depends on the probability of prefix strings.

Submission deadline: Thursday, April 27, 2023, 23:59.

Submission procedure: Follow the instruction on Canvas. Submit electronically the PDF of the printed code for classes `PrefixString`, `LZWTable`, and `LZWCompressor`, combined with the cover page of the midterm test. In addition, upload the source files to Canvas (i.e., `PrefixString.cpp`, `LZWTable.cpp`, and `LZWCompressor.cpp`).

The sources need to compile in the presence of the solution artifacts provided on Canvas.