Bytom, 04.06.2020

Semester: II Semester
Group: 1
Section: s1

COMPUTER PROGRAMMING LABORATORY

Author: Radosław Rzeczkowski
E-mail: radorze093@student.polsl.pl
Tutor: Karolina Nurzyńska, PhD, DSc

1. Task topic

18. Compression LZ78

Write a program using LZ78 algorithm to compress a text file. The program is to communicate with the user using the following call parameters:

- -f [input file]
- -o [output file]
- -c for compression
- -d for decompression

Exaample:

```
kompresuj -d -f skompresowany.dat -o plik.txt
```

2. Project analysis

In this project a simple singly-linked list is the main data structure as it provides clarity, relatively low memory consumption and a great representation of the dictionary structure used by the LZ78 Compresssion algorithm.

The LZ78 Compression algorithm is by definition the one used to tackle this task of file compression and decompression. It offers good compression ratios that depend on how repetitive the input is. When tested on regular text files it achieved compression levels of even above 50%. The algorithm works by building a dictionary of read sequences of characters. Every time a new character is read from the text file it is looked up in the dictionary. If it's not found then it gets added to the dictionary with an approprieate key (in ascending order from 1) and an id of 0, which means that it's a new character and it isn't preceeded by any sequence of characters. If it is found in the dictionary then the next character is read and is appended to the previous one. Now this sequence is looked up in the dictionary. If it's not found then it gets added with an appropriate key and an id equal to the key of the character/sequence of characters it's appended to. If it's found then the process of appending a new character from the file and looking up the new sequence is repeated, until a new sequence is found. In this way the compressed file consists of number and character pairs and is decompressed by appending the sequence of characters found at an index from a pair and the character it was paired with.

This project showed me the practical implications of the byte sizes of different data types. More on that can be found at the end of this report in Conclusions.

3. External specification

This program uses LZ78 Compression algorithm to compress and decompress text files. The files to be compressed should be `.txt` files and the resultant compressed files should be saved as a `.dat` files. This program should be used for decompression of only the `.dat` files that have been generated by this program. The resultant file from decompression should be saved as a `.txt` file.

This program is to be called from the command line. To do so, open the command line console and navigate to the directory where the program is located. The program will first

have to be compiled using `gcc -o output_program functions.c main.c` command. After the `-o` flag in place of `output_program` the user puts the name of the executable program file.

Then the program can be run using `output_program`. If run like this a Help page for the program will be displayed because no arguments were passed to the program and it requires always three arguments to run correctly. Every time wrong or insufficient arguments are passed this Help information will be displayed.

The arguments are passed to the program using flags. The flags recognized by this program are `-c`, `-d`, `-f`, `-o`. The flags may be given in any order.

`-c` this flag stands for *compression* and is used to tell the program that it needs to compress the passed file.

`-d` this flag stands for *decompression* and is used to tell the program that it needs to decompress the passed file.

`-f` this flag needs to be followed by the name of the input text file for compressing or input binary data file for decompressing. This file needs to be in the same directory as the program.

`-o` this flag needs to be followed by the name of the output binary data file for compressing or output text file for decompressing. This file will be created by the program if not present in the current directory. **CAUTION:** If the output file is present in the current directory its contents will be overwritten and lost.

Example of correct call of the program compiled to a file `de_compress` for compression of the file `my_text.txt` into file `my_text_compressed.dat` :

```
> de_compress -c -f my_text.txt -o my_text_compressed.dat
```

Example of correct call of the program compiled to a file `de_compress` for decompression of the file `my_text_compressed.dat` into file `my_text_decomp.txt` :

```
> de_compress -o my_text_decomp.txt -d -f my_text_compressed.dat
```

4. Internal specification

The internal specification is appended at the end of this report in the form of code documentation generated by Doxygen.

5. Testing

The program was tested by passing wrong and incomplete command line arguments. It was also tested with arguments passed in different order.

The program is meant to be used on `.txt` and `.dat` files so it was tested on such files. The program was tested on input text files of sizes between 0 and 500 KB. It was observed that files as large as 450 KB and higher caused a stack overflow related crash of the program on some systems (Windows 10) due to the default memory limits enforced by the gcc compiler. The program worked fine on such files when compiled and ran in Ubuntu command line or on Windows when compiled with bigger memory limits. A larger memory limit can be set in the gcc compiler using the `-Wl,--stack,10485760` flags (the number here is an arbitrary number, a smaller number will probably be required). The input binary file was tested with correct and incorrect formats and the program responds accordingly.

The compression rates depend on the size of the input file and repetitive it is. Large natural text files from the Canterbury Corpus set (http://corpus.canterbury.ac.nz/descriptions/) achieved compression ratios of about 50%. Highly repetitive input could even get compression rates as high as 70% - 80%.

6. Conclusions

This project showed me the practical implications of the byte sizes of different data types. It was very interesting to write a compression program that works fine in theory, according to the algorythm, but due to the way the data was being saved in practise that generates „compressed" files five times larger then the original file. It was very fun to understand why that was happening and find a solution, learning some new things and refreshing some knowledge from the first semester lectures along the way.

Computer Programming Semester Project - LZ78 Compression

Generated by Doxygen 1.8.17

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 list Struct Reference

A singly linked list structure representing a LZ78 dictionary entry.

`#include <structures.h>`

Data Fields

- unsigned int id
- char entry
- unsigned int key
- char * str
- struct list * p_next

3.1.1 Detailed Description

A singly linked list structure representing a LZ78 dictionary entry.

3.1.2 Field Documentation

3.1.2.1 entry

`char entry`

The character in this entry.

3.1.2.2 id

```
unsigned int id
```

The key of the linked entry.

3.1.2.3 key

```
unsigned int key
```

The key of this entry.

3.1.2.4 p_next

```
struct list* p_next
```

The pointer to the next entry in the dictionary.

3.1.2.5 str

```
char* str
```

The string encoded with this entry.

The documentation for this struct was generated from the following file:

- structures.h

Chapter 4

File Documentation

4.1 functions.c File Reference

A source code file containing the functions used by the program.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include "functions.h"
```

Functions

- struct list * list_add (struct list *head, unsigned int _id, char _entry, char *_str, unsigned int _key)

  Adds an element to a list of list structs.
- struct list * list_free (struct list *head)

  Frees the memory allocated to the list.
- void list_print (struct list *head)

  Prints the whole list to the console (used for debugging).
- struct list * list_search_string (struct list *head, char *_str)

  Searches a list linearly for a given string and returns a pointer to the element containing the string.
- void init_bin (FILE *out_bin, unsigned int count)

  Initializes the binary output file in which the compressed input file will be stored.
- void list_write_bin (FILE *out_bin, struct list *head)

  Writes the whole list pointed by its head pointer as binary data to a file.
- void compress (char *in_txt_name, char *out_bin_name)

  Coordinates the whole compression process.
- unsigned int check_input (FILE *in_bin)

  Checks the format of the input binary file and reads the amount of entries saved in it.
- struct list * list_search_key (struct list *head, unsigned int _key)

  Searches the list for a given numerical key and returns a pointer to the element containing it.
- void list_write_txt (FILE *out_txt, struct list *head)

  Writes the decompressed data to the output text file.
- void decompress (char *in_bin_name, char *out_txt_name)

  Coordinates the whole process of decompressing a file.
- int read_cmd (int count, char *args[ ], char *file_names[ ])

  Reads the command line arguments and processes them.
- void print_help ()

  Prints the Help for the program to the console.

4.1.1 Detailed Description

A source code file containing the functions used by the program.

Author

Radoslaw Rzeczkowski

Date

06.2020

4.1.2 Function Documentation

4.1.2.1 check_input()

```
unsigned int check_input (
            FILE * in_bin )
```

Checks the format of the input binary file and reads the amount of entries saved in it.

This function checks the correctness of the format of the input binary file and reads the amount of entries to be read from it. The compressed file should start with "LZ78" saved in the first 3 bytes. That should be followed by the number of entries to be read from the file saved in the next 4 bytes. That is exactly the way that

See also

init_bin initializes the binary file prior to output of the compressed data.

Parameters

in_bin	The pointer to the stream from which data will be read.

Returns

The amount of entries (character and numerical variable pair) to be read from the file.

4.1.2.2 compress()

```
void compress (
            char * in_txt_name,
            char * out_bin_name )
```

Coordinates the whole compression process.

This function opens the file streams, reads the input file, calls other functions to construct the LZ78 dictionary, print the compressed data to a given output file and free the allocated memory leaving no memory leaks.

Parameters

in_txt_name	The name of the text input file.
out_bin_name	The name of the output binary file.

4.1.2.3 decompress()

```
void decompress (
            char * in_bin_name,
            char * out_txt_name )
```

Coordinates the whole process of decompressing a file.

This function opens the file streams, reads the input file, calls other functions to reconstruct the LZ78 dictionary, print the decompressed data to a given output file and free the allocated memory leaving no memory leaks.

Parameters

in_bin_name	The name of the binary input file.
out_txt_name	The name of the output text file.

4.1.2.4 init_bin()

```
void init_bin (
            FILE * out_bin,
            unsigned int count )
```

Initializes the binary output file in which the compressed input file will be stored.

The output binary file is initialized by writing at the beginning "LZ78" using 3 bytes followed by the amount of entries that will be stored in this file using 4 bytes.

Parameters

out_bin	The pointer to the stream which will be initialized.
count	The amount of entries to be stored in the file.

4.1.2.5 list_add()

```
struct list * list_add (
            struct list * head,
```

```
            unsigned int _id,
            char _entry,
            char * _str,
            unsigned int _key )
```

Adds an element to a list of list structs.

Parameters

head	A pointer to the list.
_id	The key of the linked entry.
_entry	The character of the entry.
_str	The string that has been encoded using this entry.
_key	The key of this entry.

Returns

Caution. Returns the head pointer of the list only for the first element added. For all other elements returns a pointer to the current element in order to save time adding new elements.

Warning

See Returns.

4.1.2.6 list_free()

```
struct list * list_free (
            struct list * head )
```

Frees the memory allocated to the list.

Parameters

head	The head pointer of the list to be freed.

Returns

A null pointer.

4.1.2.7 list_print()

```
void list_print (
            struct list * head )
```

Prints the whole list to the console (used for debugging).

Parameters

head	The head pointer of the list to be printed.

4.1.2.8 list_search_key()

```
struct list * list_search_key (
            struct list * head,
            unsigned int _key )
```

Searches the list for a given numerical key and returns a pointer to the element containing it.

Parameters

head	The head pointer of the list to be searched.
_key	The value of key that will be searched for.

Returns

The pointer to the element that contains the _key value in its key field.

4.1.2.9 list_search_string()

```
struct list * list_search_string (
            struct list * head,
            char * _str )
```

Searches a list linearly for a given string and returns a pointer to the element containing the string.

Parameters

head	The head pointer of the list to be searched.
_str	The string to be found.

Returns

A pointer to the element containing the given string in its str field. When not found returns a null pointer.

4.1.2.10 list_write_bin()

```
void list_write_bin (
            FILE * out_bin,
            struct list * head )
```

Writes the whole list pointed by its head pointer as binary data to a file.

From each entry of the list only the id and entry fields are written to the file. The entry always uses one byte (it's a character) but the id uses the minimal amount of bytes in which it can be stored. For example it can use one byte for values up to 255 by saving the value as an unsigned char datatype and above that value start using two bytes in unsigned short. The exact max values of each datatype are included in the limits.h header. This is done to achieve high compression ratios.

Parameters

out_bin	The pointer to the stream in which the data will be written.
head	The head pointer of the list to be written in a file.

4.1.2.11 list_write_txt()

```
void list_write_txt (
            FILE * out_txt,
            struct list * head )
```

Writes the decompressed data to the output text file.

This function writes the str field from each element of the list into a text file. This makes up the entirety of the decoded data and the resulting file is the same as the original file that was compressed.

Parameters

out_text	The pointer to the file in which the text will be written.
head	The head pointer of the list to be written into a file.

4.1.2.12 read_cmd()

```
int read_cmd (
            int count,
            char * args[],
            char * file_names[] )
```

Reads the command line arguments and processes them.

This function reads the command line arguments passed to the program, checks their correctness and in case of any wrong/insufficient input causes the Help for the program to be displayed in console.

Parameters

	count	The amount of arguments passed to the program (argc).
	args	The arguments passed to the program (argv).
out	*file_names*	The array of two strings in which the read file names will be saved (0 for input file name, 1 for output file name).

Returns

The action to be taken by the program. For 1 compression, for 2 decompression and for -1 printing of the Help due to wrong arguments.

4.2 functions.h File Reference

A header file containing the functions used by the program.

```
#include "structures.h"
```

Functions

- struct list * list_add (struct list *head, unsigned int _id, char _entry, char *_str, unsigned int _key)
  Adds an element to a list of list structs.
- struct list * list_free (struct list *head)
  Frees the memory allocated to the list.
- void list_print (struct list *head)
  Prints the whole list to the console (used for debugging).
- struct list * list_search_string (struct list *head, char *_str)
  Searches a list linearly for a given string and returns a pointer to the element containing the string.
- void init_bin (FILE *out_bin, unsigned int count)
  Initializes the binary output file in which the compressed input file will be stored.
- void list_write_bin (FILE *out_bin, struct list *head)
  Writes the whole list pointed by its head pointer as binary data to a file.
- void compress (char *in_txt_name, char *out_bin_name)
  Coordinates the whole compression process.
- unsigned int check_input (FILE *in_bin)
  Checks the format of the input binary file and reads the amount of entries saved in it.
- struct list * list_search_key (struct list *head, unsigned int _key)
  Searches the list for a given numerical key and returns a pointer to the element containing it.
- void list_write_txt (FILE *out_txt, struct list *head)
  Writes the decompressed data to the output text file.
- void decompress (char *in_bin_name, char *out_txt_name)
  Coordinates the whole process of decompressing a file.
- int read_cmd (int count, char *args[ ], char *file_names[ ])
  Reads the command line arguments and processes them.
- void print_help ()
  Prints the Help for the program to the console.

4.2.1 Detailed Description

A header file containing the functions used by the program.

Author

Radoslaw Rzeczkowski

Date

06.2020

4.2.2 Function Documentation

4.2.2.1 check_input()

```
unsigned int check_input (
            FILE * in_bin )
```

Checks the format of the input binary file and reads the amount of entries saved in it.

This function checks the correctness of the format of the input binary file and reads the amount of entries to be read from it. The compressed file should start with "LZ78" saved in the first 3 bytes. That should be followed by the number of entries to be read from the file saved in the next 4 bytes. That is exactly the way that

See also

init_bin initializes the binary file prior to output of the compressed data.

Parameters

in_bin	The pointer to the stream from which data will be read.

Returns

The amount of entries (character and numerical variable pair) to be read from the file.

4.2.2.2 compress()

```
void compress (
            char * in_txt_name,
            char * out_bin_name )
```

Coordinates the whole compression process.

This function opens the file streams, reads the input file, calls other functions to construct the LZ78 dictionary, print the compressed data to a given output file and free the allocated memory leaving no memory leaks.

Parameters

in_txt_name	The name of the text input file.
out_bin_name	The name of the output binary file.

4.2.2.3 decompress()

```
void decompress (
            char * in_bin_name,
            char * out_txt_name )
```

Coordinates the whole process of decompressing a file.

This function opens the file streams, reads the input file, calls other functions to reconstruct the LZ78 dictionary, print the decompressed data to a given output file and free the allocated memory leaving no memory leaks.

Parameters

in_bin_name	The name of the binary input file.
out_txt_name	The name of the output text file.

4.2.2.4 init_bin()

```
void init_bin (
            FILE * out_bin,
            unsigned int count )
```

Initializes the binary output file in which the compressed input file will be stored.

The output binary file is initialized by writing at the beginning "LZ78" using 3 bytes followed by the amount of entries that will be stored in this file using 4 bytes.

Parameters

out_bin	The pointer to the stream which will be initialized.
count	The amount of entries to be stored in the file.

4.2.2.5 list_add()

```
struct list* list_add (
            struct list * head,
            unsigned int _id,
            char _entry,
            char * _str,
            unsigned int _key )
```

Adds an element to a list of list structs.

Parameters

head	A pointer to the list.
_id	The key of the linked entry.

Parameters

_entry	The character of the entry.
_str	The string that has been encoded using this entry.
_key	The key of this entry.

Returns

Caution. Returns the head pointer of the list only for the first element added. For all other elements returns a pointer to the current element in order to save time adding new elements.

Warning

See Returns.

4.2.2.6 list_free()

```
struct list* list_free (
            struct list * head )
```

Frees the memory allocated to the list.

Parameters

head	The head pointer of the list to be freed.

Returns

A null pointer.

4.2.2.7 list_print()

```
void list_print (
            struct list * head )
```

Prints the whole list to the console (used for debugging).

Parameters

head	The head pointer of the list to be printed.

4.2.2.8 list_search_key()

```
struct list* list_search_key (
            struct list * head,
            unsigned int _key )
```

Searches the list for a given numerical key and returns a pointer to the element containing it.

Parameters

head	The head pointer of the list to be searched.
_key	The value of key that will be searched for.

Returns

The pointer to the element that contains the _key value in its key field.

4.2.2.9 list_search_string()

```
struct list* list_search_string (
            struct list * head,
            char * _str )
```

Searches a list linearly for a given string and returns a pointer to the element containing the string.

Parameters

head	The head pointer of the list to be searched.
_str	The string to be found.

Returns

A pointer to the element containing the given string in its str field. When not found returns a null pointer.

4.2.2.10 list_write_bin()

```
void list_write_bin (
            FILE * out_bin,
            struct list * head )
```

Writes the whole list pointed by its head pointer as binary data to a file.

From each entry of the list only the id and entry fields are written to the file. The entry always uses one byte (it's a character) but the id uses the minimal amount of bytes in which it can be stored. For example it can use one byte for values up to 255 by saving the value as an unsigned char datatype and above that value start using two bytes in unsigned short. The exact max values of each datatype are included in the limits.h header. This is done to achieve high compression ratios.

Parameters

out_bin	The pointer to the stream in which the data will be written.
head	The head pointer of the list to be written in a file.

4.2.2.11 list_write_txt()

```
void list_write_txt (
            FILE * out_txt,
            struct list * head )
```

Writes the decompressed data to the output text file.

This function writes the str field from each element of the list into a text file. This makes up the entirety of the decoded data and the resulting file is the same as the original file that was compressed.

Parameters

out_text	The pointer to the file in which the text will be written.
head	The head pointer of the list to be written into a file.

4.2.2.12 read_cmd()

```
int read_cmd (
            int count,
            char * args[],
            char * file_names[] )
```

Reads the command line arguments and processes them.

This function reads the command line arguments passed to the program, checks their correctness and in case of any wrong/insufficient input causes the Help for the program to be displayed in console.

Parameters

	count	The amount of arguments passed to the program (argc).
	args	The arguments passed to the program (argv).
out	*file_names*	The array of two strings in which the read file names will be saved (0 for input file name, 1 for output file name).

Returns

The action to be taken by the program. For 1 compression, for 2 decompression and for -1 printing of the Help due to wrong arguments.

4.3 main.c File Reference

The main source file of this program.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include "functions.h"
```

Functions

- int **main** (int argc, char ∗argv[ ])

4.3.1 Detailed Description

The main source file of this program.

Author

Radoslaw Rzeczkowski

Date

06.2020

4.4 structures.h File Reference

A header file containing the structures used by the program.

Data Structures

- struct list

  A singly linked list structure representing a LZ78 dictionary entry.

4.4.1 Detailed Description

A header file containing the structures used by the program.

Author

Radoslaw Rzeczkowski

Date

06.2020

Index