Huffman-compression

Rónai-Kovács Martin
(MNZEBU)

Version 1.3
04/24/2020

Table of Contents

Table of contents

Huffman-compression

This project includes an OOP modell for Huffman-compression, including a test program and documentation.

Namespace Index

Namespace List

Here is a list of namespaces:

gtest_lite

Hierarchical Index

Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

BitBuffer....5
Huffman....10
Letter....11
List< T >....13
Node....14
End....8
Path....16

Credits:

In this project I used memtrace.h, memtrace.cpp and gtest_lite.h files from https://infocpp.iit.bme.hu

The project includes a test file (“O_Captain.txt”) containing a poem by Whalt Whitman.

Full, now open source project can be found on https://github.com/martin20005/Huffman-compression (my github repository), where anyone can create issues, additional changes or start a discussion in the topic publicly.

Specification:

Task:

The user gives a text file; the program will (considering the User's command) compress or extract it (using the Huffman-algorithm).

User interface:

On a simple console interface, the user types in a command (either "compress", "extract" or "exit") and a file's path, then the program starts execution. It's optional to give a resulting destination. Absolute paths should be used.

The program gives information after each command (except for "exit") (e.g. "Invalid path" or "File extracted")

An example:

```
compress "home/user/bigfile.txt"
extract "home/user/bigfile.txt.huff" "home/user/bigagain.txt"
exit
```

The example shows that not giving a result path makes the program extend the source's name with ".huff". Similarly, the same case with command "extract" creates a file ending with "_ext.txt".

Handling errors:

The program guarantees not causing any harm input files.

Class Index

Class List

Here are the classes with brief descriptions:

BitBuffer 5
.....Enables to push bitwise data from and to streams and variables
End 8
.....A Node to represent a leaf of a binary (Huffman-)tree
Huffman 10
.....Enables to compress streamdata using Huffman algorithm
Letter 11
.....Represents a letter along with it’s H-code and H-code’s length
List< T > 13
.....General list
Node 14
.....Rerpesents a binary tree’s nodes
Path 16
.....A Node to represent a non-leaf of a binary (Huffman-)tree

File Index

File List

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

bitbuffer.h .....
gtest_lite.h .....
huffman.h .....
letter.h .....
list.h .....
memtrace.h .....
node.h .....

BitBuffer Class Reference

`#include <bitbuffer.h>`

Public Member Functions

BitBuffer (int size_of_chunk_b=8 *sizeof(char))
BitBuffer (const **BitBuffer** &buffer)
const **List**< char > & **data** () const
char **currentChunk** () const
int **count** () const
int **sizeOfChunk** () const
int **countOfFullChunks** () const
bool **isEmpty** ()
bool **isOpen** ()
template<typename T > void **push** (T value_bit_container, int count_of_bits)
char **pop** ()
void **close** ()
void **leakAfter** (int num_of_chunks)
int **leakAfter** () const
void **leak** (std::ostream &stream_out)
void **fill** (std::istream &stream_in)
bool **bit** ()

Detailed Description

This class can hold bits. Useful when size of values to be stored is not constant.

Constructor & Destructor Documentation

BitBuffer::BitBuffer (int *size_of_chunk_b* = `8*sizeof(char)`)

Buffer

Parameters:

size_of_chunk_b	Number of bits each chunk can store (character's size by default)

BitBuffer::BitBuffer (const BitBuffer & *buffer*)

Buffer

Parameters:

buffer	A buffer to be copied (resulting Buffer is editable)

Member Function Documentation

bool BitBuffer::bit ()

Returning the first (not yet returned) bit from buffer

Returns:

True = 1, False = 0 bit

void BitBuffer::close ()

Closing the buffer This makes sure that all data_ is stored properly; using **push()** will not be allowed anymore.

int BitBuffer::count () const[inline]

Returns:
Count of valuable bits in **currentChunk()**

int BitBuffer::countOfFullChunks () const[inline]

Returns:
Number of full chunks.

char BitBuffer::currentChunk () const[inline]

Returns:
Last (not yet full) chunk of data_. The last **count()** bits are values.

const List<char>& BitBuffer::data () const[inline]

Returns:
data_ stored so far

void BitBuffer::fill (std::istream & *stream_in*)

Getting some data_ from file After call, at least 1 full character will be stream_in the buffer.

bool BitBuffer::isEmpty ()[inline]

Returns:
true, if there is no **pop()**-able data_ in the buffer

bool BitBuffer::isOpen ()[inline]

Returns:
Whether buffer is open (bits can be **push()**-ed)

void BitBuffer::leak (std::ostream & *stream_out*)

Leaking data_ to a file When there'are more chunks than the leak-size, those all go to the file

void BitBuffer::leakAfter (int *num_of_chunks*)

Setting the leak-size

int BitBuffer::leakAfter () const

Get leak-size

char BitBuffer::pop ()

Returning the first chunk The returned chunk will be removed from buffer.

Returns:

The first chunk

template<typename T > void BitBuffer::push (T *value_bit_container*, int *count_of_bits*) [inline]

Adding bits to the buffer Pushes the given bits into the buffer (current_chunk_).

Parameters:

value_bit_container	It contains the value-bits (fitted to the right_child_=LSB).
count_of_bits	Number of significant bits.

int BitBuffer::sizeOfChunk () const [inline]

Returns:

Count of bits in each chunk.

The documentation for this class was generated from the following files:

0 bitbuffer.h
1 bitbuffer.cpp

End Class Reference

`#include <node.h>`
Inherits **Node**.

Public Member Functions

End (**Letter** &**letter**, long frequency)
End (const **End** &end)
End ()
Letter & **letter** ()
Node * **left** () const
Node * **right** () const
bool **operator==** (**End** &end)
bool **operator==** (char ch)
void **operator++** (int)

Additional Inherited Members

Detailed Description

Represent a **Node** of a binary (**Huffman**) tree, which IS a leaf

---

Constructor & Destructor Documentation

End::End (Letter & *letter*, long *frequency*)

Constructor

Parameters:

letter	The represented **Letter**
frequency	Frequency of the letter in input

End::End (const End & *end*)

Copy constructor

End::End ()

Empty end-node; useful for arrays

Member Function Documentation

Node * End::left () const

left_child_ child

Letter & End::letter ()

Represented letter

void End::operator++ (int)

Incrementing the frequency

bool End::operator== (End & *end*)

Equality operator (true if the same letter is represented)

bool End::operator== (char *ch*)

True if **End** represents this character

Node * End::right () const

right_child_ child

The documentation for this class was generated from the following files:

2 node.h
3 node.cpp

Huffman Class Reference

`#include <huffman.h>`

Public Member Functions

void **compress** (istream &stream_in, ostream &stream_out)
void **extract** (istream &stream_in, ostream &stream_out)

Detailed Description

Class able to execute Huffman-compression or extraction

Member Function Documentation

void Huffman::compress (istream & *stream_in*, ostream & *stream_out*)

Compression of file

void Huffman::extract (istream & *stream_in*, ostream & *stream_out*)

Extraction of .huffman_code_ file

The documentation for this class was generated from the following files:

4 huffman.h
5 huffman.cpp

Letter Class Reference

`#include <letter.h>`

Public Member Functions

Letter (char ch)
Letter (const **Letter** &letter)
Letter ()
char **original** () const
void **original** (char ch)
long long int **huffman** () const
void **huffman** (long long int h, unsigned char l)
unsigned char **length** () const
Letter & **operator=** (const **Letter** &letter)
bool **operator==** (const **Letter** &letter) const

Detailed Description

Bounds original character and Huffman-code together

Constructor & Destructor Documentation

Letter::Letter (char *ch*)[inline], [explicit]

Constructor Only the original character is set

Letter::Letter (const Letter & *letter*)[inline]

Copy constructor

Letter::Letter ()[inline]

isEmpty **Letter** for creating arrays

Member Function Documentation

long long int Letter::huffman () const[inline]

Returns:
The Huffman-code of character

void Letter::huffman (long long int *h*, unsigned char *l*)[inline]

Setting the Huffman-code (length of H-code always needed!)

unsigned char Letter::length () const[inline]

Returns:
The length of Huffman-code

Letter& Letter::operator= (const Letter & *letter*)[inline]

Assignment

bool Letter::operator== (const Letter & *letter*) const[inline]

Equal if the original characters equal

char Letter::original () const[inline]

Returns:

The original character

void Letter::original (char *ch*)[inline]

Setting the original character

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

6 letter.h

List< T > Class Template Reference

Public Member Functions

List (const **List** &list)
int **count** () const
void **add** (T element)
void **removeAt** (int idx)
void **remove** (T &element)
int **findFirst** (T &element)
T & **operator[]** (int idx) const
List & **operator+=** (T element)

Member Function Documentation

template<class T> List& List< T >::operator+= (T *element*)[inline]

Addition of element with operator

template<class T> T& List< T >::operator[] (int *idx*) const[inline]

Returns:
Element with index 'idx'

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

7 list.h

Node Class Reference

`#include <node.h>`
Inherited by **End**, and **Path**.

Public Member Functions

Node ()
Node (**Node** ***left**, **Node** ***right**, long **weight**=0)
Node *& **left** ()
void **left** (**Node** *l)
Node *& **right** ()
void **right** (**Node** *r)
long **weight** () const
void **weight** (long new_weight)

Protected Attributes

Node * **left_child_**
Pointer to left_child_ child.

Node * **right_child_**
Pointer to right_child_ child.

long **weight_**
Weight of node.

Detailed Description

Represents a **Node** of a binary (**Huffman**) tree

Constructor & Destructor Documentation

Node::Node ()

Constructor Basically an empty node (useful for creating a list of '**Node**'-s)

Node::Node (Node * *left*, Node * *right*, long *weight* = 0)

Constructor Making a parent for 2 children nodes

Member Function Documentation

Node *& Node::left ()

Returns left_child_ child's pointer's reference

void Node::left (Node * *l*)

Setter for the left_child_ child

Returns:
The new left_child_'s pointer

Node *& Node::right ()

Returns right_child_ child's pointer's reference

void Node::right (Node * *r*)

Setter for the right_child_ child

Returns:

The new right_child_'s pointer

long Node::weight () const

Weight of node

void Node::weight (long *new_weight*)

Setting the weight of node

The documentation for this class was generated from the following files:

8 node.h
9 node.cpp

Path Class Reference

`#include <node.h>`
Inherits **Node**.

Public Member Functions

Path (**Node** ***left**, **Node** ***right**)

Additional Inherited Members

Detailed Description

Represents a **Node** of a binary (**Huffman**) tree, which is NOT a leaf

---

Constructor & Destructor Documentation

Path::Path (Node * *left*, Node * *right*)

Merging 2 nodes The new (parent) **Node** has a count equal to the sum of the children's count

---

The documentation for this class was generated from the following files:

10 node.h
11 node.cpp

Testing

Cases

All classes of the project have a testing section in main.cpp. To enable auto-testing ***"#define HUFFMAN_TEST"*** must be used.

Buffer

Constructor

Uses push(), tests whether copying works correctly.

Push

Checks if push() really saves data.

Pop

Checks if pop() returns correctly and whether it removes the popped data.

Bit

Tests getting bitwise data

Letter

Constructor

Checks if constructors can be called and work properly.

Functions

Uses functions of the class, while testing them.

List

Ctor_Add_Count_Remove

Checks these functions of List (and operators).

Huffman

Compress_Extract

Uses the 2 public functions of class Huffman. Human interaction needed to check whether "Invalid path" was put to output (and whether compression was resulting in file). But it's semantic error; syntactic errors are still result in Fail.

The test:

```
#define HUFFMAN_TEST
/* … */
#ifdef HUFFMAN_TEST
// ----------------------------  Buffer  ---------------------------- //
TEST(Buffer, Constructor) {
    BitBuffer bb0(8);
    bb0.push('a', 4);
    bb0.close();
    BitBuffer bb1 = BitBuffer();
    EXPECT_EQ(sizeof(char)*8, (unsigned long) bb1.sizeOfChunk());
    BitBuffer bb2(bb0);
    EXPECT_EQ(bb0.pop(), bb2.pop());
} END
```

```
TEST(Buffer, Push) {
    BitBuffer test1 = BitBuffer();
    test1.push('a' >> 4, 4);
    test1.push('a', 4);
    test1.close();
    EXPECT_EQ('a', test1.data()[0]);
} END
TEST(Buffer, Pop) {
    BitBuffer test2 = BitBuffer();
    test2.push('a', 8);
    test2.close();
    EXPECT_EQ(1, test2.countOfFullChunks());
    EXPECT_EQ('a', test2.pop());
    EXPECT_EQ(0, test2.countOfFullChunks());
} END
TEST(Buffer, Bit) {
    BitBuffer test3 = BitBuffer();
    test3.push(170, 8);
    test3.close();
    int cnt = 0;
    for (int i = 0; i < test3.sizeOfChunk(); ++i) {
        cnt += test3.bit() ? 1 : 0;
    }
    EXPECT_EQ(4, cnt);
} END
// ----------------------------  Letter  ---------------------------- //
TEST(Letter, Constructor) {
    Letter l0 = Letter();
    EXPECT_EQ((char) 0, l0.original());
    EXPECT_EQ((long long int) 0, l0.huffman());
    EXPECT_EQ((unsigned char) 0, l0.length());
    Letter l1 = Letter('a');
    EXPECT_EQ('a', l1.original());
} END
TEST(Letter, Functions) {
    Letter l0('a');
    l0.huffman('b', 4);
    Letter l1 = l0;
    EXPECT_EQ(l0.original(), l1.original());
    EXPECT_EQ(l0.huffman(), l1.huffman());
    EXPECT_TRUE(l0 == l1);
} END
// ------------------------------ List ------------------------------ //
TEST(List, Ctor_Add_Count_Remove) {
    List<int> tester = List<int>();
    EXPECT_EQ(0, tester.count());
    tester.add(1);
    tester += 2;
    EXPECT_EQ(2, tester.count());
    EXPECT_EQ(1, tester[0]);
    tester.removeAt(0);
    EXPECT_EQ(2, tester[0]);
} END
```

```
// ------------------------------ Node ------------------------------ //
TEST(Node, General) {
    Node node;
    Letter letter = Letter('a');
    End end = End(letter, 0);
    end++;
    EXPECT_EQ(1, (int) end.weight());
    EXPECT_EQ('a', end.letter().original());
    node.left(&end);
    EXPECT_TRUE(node.left() == &end);
    Path path = Path(&end, &end);
    EXPECT_EQ(2, (int) path.weight());
} END
// ---------------------------  Huffman  ---------------------------- //
TEST(Huffman, Compress_Extract) {
    string HOME = "";
    //string HOME = std::getenv("HOME") ? std::getenv("HOME") : ".";
    string source = HOME + "../O_Captain.txt";
    string destiny = HOME + "../O_Captain.huff";
    string extdest = HOME + "../O_Captain_ext.txt";
    std::ifstream source_file;
    std::ofstream destiny_file;
    Huffman h2 = Huffman();
    source_file.open(source);
    destiny_file.open(destiny);
    h2.compress(source_file, destiny_file);
    source_file.close();
    destiny_file.close();
    source_file.open(destiny);
    destiny_file.open(extdest);
    h2.extract(source_file, destiny_file);
    source_file.close();
    destiny_file.close();
} END
#endif
```