Übung 02

Arbeitsaufwand insgesamt: 18h

Inhaltsverzeichnis

Inhaltsverzeichnis

Übung 02 1
Teil 1 – Mergesort 2
Lösungsidee: 2
Lösung: 3
Testfälle 4
Source Code: 7

Teil 1 – Mergesort

In dieser Übung soll der Sortieralgorithmus „external Mergesort“ (Auslagerung der zu Daten in Files) implementiert werden.

Lösungsidee:

Der Mergesort basiert auf der Idee mehrere Datensätze zu richtig zu rekombinieren („mergen“). Deswegen wird der Sortiervorgang damit begonnen die Ausgangsdaten auf zwei Files aufzuteilen (partitionieren). Dafür wird beim Schreiben der Files nach jedem Datensatz die Datei abgwechselt. D.h. erste Zahl in File 1, zweite Zahl in File 2 etc. In jedem File sollten dann annähernd gleich viele Zahlen sein (maximal eine Zahl mehr/weniger). Ich habe mich für Integer als Datentyp entschieden, weil sich diese gut einlesen und vergleichen lassen und es nicht aus der Angabe hervorging.

Der nächste, und wichtigste Schritt, ist das Mergen der Files.
Das geschieht in mehreren Läufen „p“, nach denen immer die Eingabe- und Ausgabefiles getauscht werden. Diese Läufe haben eine Länge „k“ (= Anzahl der Zahlen je File die verglichen werden), welche nach jedem Merge-Zyklus verdoppelt wird. Beim Vergleich der Zahlen wird immer die erste, noch nicht sortierte Zahl aus dem jeweiligen File mit dem aus dem anderen File verglichen und dann die kleinere in das momentane File geschrieben.

Grundsätzlich werden immer so viele Merge-Zyklen benötigt, bis die Lauflänge k gleich der Dateilänge ist.

Da es sich bei diesem Algorithmus um einen DNC-Algorithmus handelt, habe ich mich beschlossen diesen rekursiv zu realisieren. Ein Teil der Lösungen (partitionieren, swappen und die grundlegenden Klassen) wurde bereits im Unterricht besprochen, weshalb ich mich hier vorwiegend auf die merge() Methode beziehen werde, welche noch ausständig ist.

Zu Beginn kann gleich gesagt werden, dass, wie oben erwähnt, der Rekursionsboden „k >= “ ist.

Ist dieser nicht erreicht, wird ein Merge-Zyklus ausgeführt, die Files geswapped und wiederum rekursiv aufgerufen mit doppeltem k.

Nun ist noch zu klären was in einem Zyklus zu tun ist:

- Anfangs müssen die ersten Daten aus beiden Files ausgelesen werden
- Danach werden in einer Schleife immer wieder überprüft welche Zahl aus den beiden Files größer ist
- Beim Vergleich muss darauf geachtet werden, dass immer nur k Elemente aus jedem File verglichen werden. Wurden z.B. bereits k Elemente aus File 1 gelesen und ins aktuelle Ziel geschrieben, so müssen immer Zahlen aus File 2 verwendet werden, auch wenn diese kleiner sind als das momentane aus File 1. Das heißt es muss je File mitgezählt werden und nach jedem Lauf auf 0 gesetzt werden
- Zudem kann es passieren, dass ein File kleiner ist als das andere, weshalb beim Einlesen der Daten auf Fehler überprüft werden muss. In diesem Fall gibt es für die andere Zahl keinen Vergleich und es kann direkt die andere Zahl ins File geschrieben werden.

Durch diese Vorgehensweise wird garantiert, dass sich immer nur ein einziger Datensatz im Arbeitsspeicher befindet und der Mergesort sinngemäß funktioniert.

Als relevante File-Operationen habe ich mich für folgendes entschieden:

- Partitionieren (bereits in Übung erstellt)
- Nächste Zahl aus Inputstream holen
- File mit zufälligen Zahlen erstellen
- Fileinhalt kopieren
- Fileinhalt ausgeben
- Files löschen

Lösung:

Als C++ Projekt „Teil_1“ im Archiv
Source-Code am Ende dieses Dokuments

Testfälle

- Leeres File
- Referenzbeispiel Unterricht
- Ungerade Anzahl an Zahlen
- Negative Zahlen
- Große Files (1 000 000 Datensätze)
- Falscher Datentyp

Leeres File:

```
Nothing to do here, no elements found. Cleaning up...
```

Aufruf mit einem leeren File führt nicht zu einem Abbruch und das File bleibt leer.

Referenzbeispiel Unterricht:
Ausgang: 7 6 3 8 1 5 2 6

```
f0.txt:
7 3 1 2
f1.txt:
6 8 5 6

g0.txt:
3 8 2 6
g1.txt:
6 7 1 5

f0.txt:
1 2 5 6
f1.txt:
3 6 7 8

g1.txt:
1 2 3 5 6 6 7 8
```

Das File wird in den gleichen Schritten gemerged, wie im Unterricht.

Ungerade Anzahl an Zahlen:

```
f0.txt:
7 3 1 2 3333
f1.txt:
6 8 5 6

g0.txt:
6 7 1 5 3333
g1.txt:
3 8 2 6

f0.txt:
3 6 7 8 3333
f1.txt:
1 2 5 6

g0.txt:
1 2 3 5 6 6 7 8
g1.txt:
3333

f0.txt:
1 2 3 5 6 6 7 8 3333
```

Auch mit ungerader Anzahl an Daten funktioniert der Algorithmus.

Negative Zahlen:

```
f0.txt:
-1239 -3 -2 -15 -11
f1.txt:
-20 -5 -22222 -30 -11

g0.txt:
-1239 -20 -22222 -2 -11 -11
g1.txt:
-5 -3 -30 -15

f0.txt:
-1239 -20 -5 -3 -11 -1
f1.txt:
-22222 -30 -15 -2

g0.txt:
-22222 -1239 -30 -20 -15 -5 -3 -2
g1.txt:
-11 -1

f0.txt:
-22222 -1239 -30 -20 -15 -11 -5 -3 -2 -1
```

Negative Zahlen stören den Algorithmus nicht.

Große Files (1 000 000 Datensätze):

```
Printing has been skipped due to large amount of data.
```

Solch riesige Datenmengen lassen sich nicht gut darstellen und die Ausgabe verbraucht unnötig Ressourcen bzw. blockiert das Weiterarbeiten des Algorithmus.

Ein erster Blick auf das File zeigt jedoch, dass dieses sortiert zu sein scheint (zur Übersichtlichkeit wurden einige Zeilenumbrüche eingefügt):

```
-999998 -999995 -999994 -999992 -999991 -999986 -999986 -999985 -999985 -999980 -999979
-999977 -999977 -999976 -999974 -999973 -999971 -999969 -999968 -999967 -999965 -999958
-999952 -999952 -999952 -999952 -999948 -999947 -999946 -999945 -999937 -999936 -999936
-999935 -999934 -999933 -999932 -999930 -999928 -999927 -999927 -999925 -999925 -999924
-999923 -999922 -999921 -999921 -999919 -999917 -999917 -999917 -999916 -999913 -999911
-999911 -999909 -999906 -999903 -999900 -999894 -999891 -999890 -999883 -999882 -999882
-999881 -999879 -999879 -999876 -999875 -999874 -999870 -999865 -999863 -999861 -999861
-999860 -999856 -999852 -999848 -999843 -999841 -999837 -999836 -999833 -999832 -999829
-999828 -999825 -999822 -999813 -999812 -999806 -999805 -999804 -999801 -999801 -999795
-999791 -999788 -999788 -999785 -999784 -999782 -999780 -999780 -999779 -999779 -999778
-999773 -999772 -999767 -999765 -999763 -999758 -999758 -999757 -999754 -999753 -999752
-999751 -999749 -999748 -999745 -999745 -999744 -999738 -999738 -999734 -999731 -999727
-999723 -999723 -999722 -999720 -999719 -999718 -999714 -999712 -999706 -999705 -999705
-999701 -999699 -999698 -999696 -999695 -999694 -999689 -999685 -999683 -999682 -999676
...
```

Falscher Datentyp:

```
Nothing to do here, no elements found. Cleaning up...
```

Wiederum werden keine Nutzdaten gefunden und deshalb der Vorgang abgebrochen.

Source Code:

main.cpp

```
#include "MergeSort.h"
#include "FileManipulator.h"
#include<iostream>

int main()
{
	std::string const test1("empty.txt");
	std::string const test2("reference.txt");
	std::string const test3("uneven.txt");
	std::string const test4("negative.txt");
	std::string const test5("random_big.txt");
	std::string const test6("strings.txt");
	FileManipulator::create_random_file(test5, 1000000, -1000000, 1000000);

	// Initialize mergesort
	MergeSort ms = MergeSort();
	ms.c_wf[0][0] = "f0.txt";
	ms.c_wf[0][1] = "f1.txt";
	ms.c_wf[1][0] = "g0.txt";
	ms.c_wf[1][1] = "g1.txt";



	ms.sort(test1);
	//ms.sort(test2);
	//ms.sort(test3);
	//ms.sort(test4);
	//ms.sort(test5);
	//ms.sort(test6);

	return 0;
}
```

Mergesort.h

```
#ifndef MERGESORT_H
#define MERGESORT_H
#include "FileManipulator.h"

static std::size_t const LEFT = 0;       // Left file
static std::size_t const RIGHT = 1;      // Right file
static std::size_t const NUM_FILES = 2;
static std::size_t const PRINT_THRESHOLD = 50;

class MergeSort {

public:
	// Abbreviation for FileManipulator
	typedef FileManipulator fm;
	typedef int value_type;

	std::string c_wf[NUM_FILES][NUM_FILES]; // c_wf = choose writefile
	bool c_wf_swap;

	MergeSort()
	{
		c_wf_swap = false;
	}

	// Takes the given file, sorts its values and overwrites it.
	void sort(std::string const& filename);

private:
	// Returns the current filename for reading
	std::string const& getSrc(const size_t& i);
	// Returns the current filename for writing
	std::string const& getDst(const size_t& i);

	// Merge a partitioned file until it is sorted
	std::string merge(size_t const k, size_t const size);
	// One merge cycle
	void merge_cycle(size_t const size, size_t const run_length, std::ifstream&
in_left, std::ifstream& in_right, std::vector<std::ofstream*>& outputs);

	// Swaps between output/input files
	void swap();
};

#endif
```

Mergesort.cpp

```
#include "MergeSort.h"
#include<vector>
#include<fstream>

void MergeSort::sort(std::string const& filename)
{
	size_t k(1);
	std::vector<std::string> src;
	src.push_back(getSrc(LEFT));
	src.push_back(getSrc(RIGHT));

	// 1) partition
	size_t const n(fm::partition(filename, src));

	if (n > 0)
	{
		// 2) merge
		std::string sorted_filename(merge(k, n));

		// 3) copy result in source file
		if (n < PRINT_THRESHOLD)
		{
			std::cout << sorted_filename << ":\n";
			fm::print(sorted_filename);
		}
		else
		{
			std::cout << "Printing has been skipped due to large amount of
data.\n";
		}
		fm::copy(sorted_filename, filename);
	}
	else
	{
		std::cout << "Nothing to do here, no elements found. Cleaning up...\n";
	}

	// Remove all temporary files
	fm::clean_up({ getSrc(LEFT) ,getDst(LEFT) ,getSrc(RIGHT) ,getDst(RIGHT) });
}
```

```
std::string MergeSort::merge(size_t const run_length, size_t const size)
{
// If run_length greater or equal the amount of total datasets, stop recursion and return
filename.
	// Using size here, because we always handle run_length * 2 variables and therefor
have twice as many as per file.
	if (run_length >= size)
	{
		swap();
		return getDst(LEFT);
	}
	else
	{
		// Create all necessary input and output streams
		std::ifstream in_left(getSrc(LEFT));
		std::ifstream in_right(getSrc(RIGHT));

		std::vector<std::ofstream*> outputs;
		outputs.push_back(new std::ofstream(getDst(LEFT)));
		outputs.push_back(new std::ofstream(getDst(RIGHT)));

		merge_cycle(size, run_length, in_left, in_right, outputs);
		if (size <= PRINT_THRESHOLD)
		{
			std::cout << getSrc(LEFT) << ": \n";
			fm::print(getSrc(LEFT));
			std::cout << getSrc(RIGHT) << ": \n";
			fm::print(getSrc(RIGHT));
			std::cout << "\n";
		}

		// After one merging process, close all the filestreams.
		in_left.close();
		in_right.close();
		for (size_t n(0); n < outputs.size(); n++)
		{
			delete outputs[n];
		}

		// After one merging process, swap the input/output files and follow
recursion with double run_length
		swap();
		return merge(run_length * 2, size);
	}
}
```

```
void MergeSort::merge_cycle(size_t const size, size_t const run_length, std::ifstream&
in_left, std::ifstream& in_right, std::vector<std::ofstream*>& outputs)
{
	// Define a variable to easily switch between outputstreams
	size_t stream_index(1);

	// These two variables are used to store the current dataset (e.g, the first
uncompared number in the file) for each file
	value_type i1(-1);
	value_type i2(-1);

	// When reading from the file, it could happen that one file has no more
values.Hence we can check whether or not the number is valid later.
	bool ok1(fm::get_next_value(in_left, i1));
	bool ok2(fm::get_next_value(in_right, i2));

	// In every run only "k" numbers can be read from each file. Therefor we count how
often we read from each file.
	size_t amount1(0);
	size_t amount2(0);

	// Merging process (we write one element per cycle, hence we need <size> cycles)
	for (size_t n(0); n < size; n++)
	{
		// Change the output file after k run_length and reset read amounts
		if (n % (run_length * NUM_FILES) == 0)
		{
			stream_index = (stream_index + 1) % NUM_FILES;

			amount1 = 0;
			amount2 = 0;
		}
```

```
// In case we have already written too many entries from one file, always write the other
number.
		if (amount1 == run_length && ok2)
		{
			*outputs[stream_index] << i2 << " ";
			ok2 = fm::get_next_value(in_right, i2);
		}
		else if (amount2 == run_length && ok1)
		{
			*outputs[stream_index] << i1 << " ";
			ok1 = fm::get_next_value(in_left, i1);
		}
		else
		{
			// If input from both files could potentially be taken (based on
amount), check if they're valid...
			if (ok1 && ok2)
			{
				// ...and compare them
				if (i1 <= i2)
				{
					*outputs[stream_index] << i1 << " ";

					// Upon writing one number to the file, we need to
increase the amount counter and read the next number
					amount1++;
					ok1 = fm::get_next_value(in_left, i1);
				}
				else
				{
					*outputs[stream_index] << i2 << " ";
					amount2++;
					ok2 = fm::get_next_value(in_right, i2);
				}
			}
			// If one of the numbers couldn't be read, take the other one.
			else if (!ok1)
			{
				*outputs[stream_index] << i2 << " ";
				ok2 = fm::get_next_value(in_right, i2);
			}
			else if (!ok2)
			{
				*outputs[stream_index] << i1 << " ";
				ok1 = fm::get_next_value(in_right, i2);
			}
		}
	}
}

std::string const& MergeSort::getSrc(const size_t& i)
{
	return c_wf[c_wf_swap ? RIGHT : LEFT][i];
}
std::string const& MergeSort::getDst(const size_t& i)
{
	return c_wf[c_wf_swap ? LEFT : RIGHT][i];
}
```

```
void MergeSort::swap()
{
	c_wf_swap = !c_wf_swap;
}
```

FileManipulator.h

```
#ifndef FILEMANIPULATOR_H
#define FILEMANIPULATOR_H
#include<iostream>
#include<vector>
#include<string>
#include<fstream>

class FileManipulator {

public:
	typedef int value_type;

	// Writes the next value from <in> into <value>. Returns whether it was successful
or not.
	static bool get_next_value(std::ifstream& in, value_type& value);

	// Splits a file in two.
	static size_t partition(std::string const& src, std::vector<std::string> const&
dst);
	// Creates a file with random numbers
	static void create_random_file(std::string const& filename, size_t const size,
value_type const min, value_type const max);

	// Prints a file one number after another.
	static void print(std::string const& filename);
	// Prints a file one number after another using an inputstream.
	static void print(std::ifstream& in);

	// Copies the content of one file to another.
	// Caution: Overwrites files
	static void copy(std::string const& src, std::string const& dst);

	// Deletes all specified files
	static void clean_up(std::vector<std::string> files);
};

#endif
```

FileManipulator.cpp

```
#include "FileManipulator.h"
#include<random>
#include<chrono>
#include<fstream>
using std::chrono::system_clock;
using std::default_random_engine;
using std::uniform_real_distribution;
using std::uniform_int_distribution;

size_t FileManipulator::partition(std::string const& src, std::vector<std::string> const&
dst)
{
	std::ifstream in(src);
	std::vector<std::ofstream*> out;

	//create outputstreams based on input parameter dst
	for (size_t n(0); n < dst.size(); n++)
	{
		out.push_back(new std::ofstream(dst[n]));
	}

	size_t n(0); // amount of total elements. For arithmetic functions (like
determining the middle)
	value_type value;

	while (in >> value)
	{
		std::ofstream& outFile = *(out[n % dst.size()]);
		if (outFile)
		{
			outFile << value << " ";
		}
		n++;
	}

	// free the output streams
	for (size_t i(0); i < out.size(); i++)
	{
		delete out[i];
	}

	return n;
}
```

```
void FileManipulator::create_random_file(std::string const& filename, size_t const size,
value_type const min, value_type const max)
{
	// using the uniform random generator from std lib
	unsigned int seed = system_clock::now().time_since_epoch().count();
	default_random_engine generator(seed);
	uniform_int_distribution<int> distribution(min, max);

	std::ofstream out(filename);

	for (size_t n(0); n < size; n++)
	{
		out << distribution(generator) << " ";
	}

	out.close();
}

void FileManipulator::copy(std::string const& src, std::string const& dst)
{
	std::ifstream in(src);
	std::ofstream out(dst);

	// Read and write value by value.
	value_type value(0);
	while (in >> value)
	{
		out << value << " ";
	}

	in.close();
	out.close();
}

void FileManipulator::clean_up(std::vector<std::string> files)
{
	for (size_t i(0); i < files.size(); i++)
	{
		remove(files[i].c_str());
	}
}

void FileManipulator::print(std::string const& filename)
{
	std::ifstream in(filename);
	print(in);
	in.close();
}

void FileManipulator::print(std::ifstream& in)
{
	value_type value(0);
	while (in >> value)
	{
		std::cout << value << " ";
	}
	std::cout << std::endl;
}
```

```
bool FileManipulator::get_next_value(std::ifstream& in, value_type& value)
{
	if (in >> value)
	{
		return true;
	}
	else
	{
		value = -1;
		return false;
	}
}
```