Sigrid Klein, 6. November 2022 Aufwand: ca. 7h

1. External-Mergesort

1.1. Lösungsidee

File_manipulator:

Fill_randomly: Verwendet random für Zufallszahlen, um damit zufällige chars zu erzeugen. Dazu verwendet es real_distribution für einen zufälligen double zwischen 0 und 1. Ist diese Zahl kleiner als 0.5 wird der Buchstabe ein Kleinbuchstabe, ansonsten ein Großbuchstabe. Mit int_distribution wird der exakte Buchstabe berechnet.

Partition: Liest alle Werte aus dem Ausgangsfile ein und schreibt sie abwechselnd auf die output files.

Merge_sort:

Allgemeiner Algorithmus:

Idee:

Beim externen Mergesort lädt man immer nur zwei Objekte in den Arbeitsspeicher. Diese werden dann verglichen und dementsprechend weiter behandelt. Deshalb können damit große Objekte von externen Quellen sortiert werden, ohne zuerst den ganzen Datenbestand zu speichern.

Vorgehensweise:

Im ersten Schritt werden die Objekte auf zwei Dateien (f0.txt, f1.txt) aufgeteilt -> partition funktion vom file_manipulator. Der Algorithmus verwendet insgesamt 4 Hilfsdateien (f0.txt, f1.txt, g0.txt, g1.txt).

Im nächsten Schritt wird immer ein Element des ersten files (f0) und ein Element des zweiten files (f1) miteinander verglichen und aufsteigend sortiert in das erste outputfile gespeichert (g0). Das je zweite Element von f0 und f1 werden ebenfalls verglichen und in g1 gespeichert. Jetzt werden nun alle Elemente verglichen und immer abwechselnd in g0 und g1 gespeichert. Hierbei enstehen sortierte Läufe von 2, weil die 2 verglichenen Elemente immer nebeneinander im File liegen.

Da nun alle Elemente umsortiert in g0 und g1 liegen werden im nächsten Schritt, wenn diese Elemente wieder weiter sortiert werden, werden die files f0 und f1 überschrieben, da diese nur einen irrelevanten Zwischenschritt speichern.

Man kann nun in Läufen von zwei sortieren. Das heißt man nimmt je die ersten zwei von beiden files und merget diese, sodass Läufe der Länge 4 enstehen.

Dies wiederholt man, bis ein Lauf alle Elemente enthält. Das File, in dem dieser Lauf enthalten ist, ist dann das Ergebnis und kann zum Beispiel in eine gewünschte Datei kopiert werden.

Merge:

Zwei sortierte Läufe werden zu einem vereinigt, in dem immer der vorderste noch nicht evaluierte Wert der beiden Läufe verglichen wird. Der kleinere Wert wird in eine neue Datei gespeichert und

von diesem Lauf wird der nächste Wert betrachtet. Dieses Vorgehen wiederholt sich, solange beide Läufe Werte enthalten. Wenn von einem Lauf keine Werte mehr übrig sind, werden alle restlichen Werte des anderen Laufs noch abgespeichert und das Vorgehen ist fertig.

1.2. Quelltext

1.2.1. file_manipulator.h:

```
#if ! defined FILE_MANIPULATOR_H
#define FILE_MANIPULATOR_H

#include <iostream>
#include <string>
#include <fstream>
#include <vector>

class file_manipulator {

public:
	using value_t = std::string;
	using cont_t = std::vector<value_t> ;


	static void fill_randomly(value_t const& dst, std::size_t n = 100, int const
no_letters = 1);
	static void print(value_t const& src, std::ostream& out);
	static std::size_t partition(value_t const& src, cont_t const& dst);
	static void copy(value_t const& src, value_t const& dst);

};
```

1.2.2. file_manipulator.cpp

```
#include "file_manipulator.h"

#include <random>
#include <chrono>

using clck = std::chrono::system_clock;
using rnd = std::default_random_engine;
using real_dist = std::uniform_real_distribution<double>;
using int_dist = std::uniform_int_distribution<int>;



void file_manipulator::fill_randomly(value_t const& dst, std::size_t n, int const
no_letters) {

	unsigned int seed = clck::now().time_since_epoch().count();
	rnd generator{ seed };
	real_dist dis_r{ 0.0,1.0 };  //decides between upper and lower -case letters
	int_dist dis_i{ 0, 25 };  //decides the actual letter
	std::ofstream out{ dst };
```

```
	// generating n sets of random chars [A-Za-z] as long as the output stream is
fine
	while (out && (n-- > 0)) {
		// generating no_letters chars for each set
		for (int i{ 0 }; i < no_letters; ++i) {
			char c;
			if (dis_r(generator) < 0.5) {
				c = ('a' + dis_i(generator));
			}
			else {
				c = ('A' + dis_i(generator));
			}
			out << c;
		}
		out << ' ';
	}
}

std::size_t file_manipulator::partition(value_t const& src, cont_t const& dst) {
	std::ifstream in { src };
	std::vector<std::ofstream*> out;

	// putting the output streams for all files in the vector
	for (std::size_t i{ 0 }; i < dst.size(); ++i) {
		out.push_back(new std::ofstream(dst[i]));
	}

	std::size_t n{ 0 };
	value_t value;
	// as long as the input streams delievers new input -> the whole file gets read
	while (in >> value) {
		// the current outputstream calculated with n (a counter)
		std::ofstream& outFile{ *(out[n % dst.size()]) };
		if (outFile) {
			outFile << value << ' ';
		}
		n++;
	}

	// deleting the streams from the heap -> closing the streams
	for (std::size_t i{ 0 }; i < out.size(); ++i) {
		delete out[i];
	}
	return n;
}

void file_manipulator::print(value_t const& src, std::ostream& out) {

	std::ifstream in{ src };

	value_t value;

	// as long as the input streams delievers new input -> the whole file gets read
	while (in >> value) {
		if (out) {
			// each value gets printed on the given output stream
			out << value << ' ';
		}
	}

}

void file_manipulator::copy(value_t const& src, value_t const& dst) {
	std::ofstream out{ dst };
```

```
	// printing the source on the dst via output stream
	print(src, out);
}
```

1.2.3. merge_sort.h

```
#if ! defined MERGE_SORT_H
#define MERGE_SORT_H

#include "file_manipulator.h"

#include <string>
#include <vector>

class merge_sorter {
public:
	static void sort(std::string const& infilename, std::string const& outfilename);

private:
	// to remember what files the algorithm is currently working on
	enum file_types {
		g,
		f
	};

	// the helper files
	static file_manipulator::cont_t f_files;
	static file_manipulator::cont_t g_files;

	static size_t sort_step(size_t k, file_manipulator::cont_t const & src,
file_manipulator::cont_t const& dst);
	static void merge(size_t const k, std::ofstream* out,
std::vector<std::ifstream*> in, file_manipulator::value_t& value1);
};


#endif
```

1.2.4. merge_sort.cpp

```
#include "merge_sort.h"

// initializing the helper files
file_manipulator::cont_t merge_sorter::f_files = { "f0.txt", "f1.txt" };
file_manipulator::cont_t merge_sorter::g_files = { "g0.txt", "g1.txt" };


void merge_sorter::sort(std::string const& infilename, std::string const& outfilename)
{
	size_t const n{ file_manipulator::partition(infilename, f_files) }; //n is the
amount of strings
	size_t k{ 1 }; //current run length
	file_types current_srcfiles{ f }; //the current source files (f or g)
	while (k < n) {
		if (current_srcfiles == f) {
			k = sort_step(k, f_files, g_files);
```

```
                        current_srcfiles = g; //changing the current source files to the
opposite
                }
                else {
                        k = sort_step(k, g_files, f_files);
                        current_srcfiles = f; //changing the current source files to the
opposite
                }

        }
        // copying the right file into the result file
        if (current_srcfiles == f) {
                file_manipulator::copy(f_files[0], outfilename);
        }
        else {
                file_manipulator::copy(g_files[0], outfilename);
        }
}


size_t merge_sorter::sort_step(size_t k, file_manipulator::cont_t const& src,
file_manipulator::cont_t const& dst) {
        std::vector<std::ofstream*> out;
        std::vector<std::ifstream*> in;

        // putting the output streams for both dst files in the vector
        for (std::size_t i{ 0 }; i < dst.size(); ++i) {
                out.push_back(new std::ofstream(dst[i]));
        }
        // putting the input streams for both src files in the vector
        for (std::size_t i{ 0 }; i < src.size(); ++i) {
                in.push_back(new std::ifstream(src[i]));
        }

        size_t current_out_ind{ 0 }; // to remember which output stream to use

        file_manipulator::value_t value;

        while (*(in[0]) >> value) {

                merge(k, out[current_out_ind], in, value);

                current_out_ind = (current_out_ind + 1) % 2; // changing the output
stream to the other one
        }

        //deleting all the streams from the heap
        for (std::size_t i{ 0 }; i < out.size(); ++i) {
                delete out[i];
        }
        for (std::size_t i{ 0 }; i < in.size(); ++i) {
                delete in[i];
        }

        return k * 2; // doubleing the run length
}

void merge_sorter::merge(size_t const k, std::ofstream* out,
std::vector<std::ifstream*> in, file_manipulator::value_t& value1) {
        file_manipulator::value_t value2;
        size_t read_count_0{ 1 }; // to count how many values have been read from input
file index 0
        size_t read_count_1{ 0 }; // to count how many values have been read from input
file index 1
```

```
	bool good{ true };

	// for every value in the run in inputfile index 1
	while (read_count_1 < k && *in[1] >> value2) {
		++read_count_1;
		// all the values from inputfile 0 in the run, that are smaller than the
current value from inputfile 1
		while (good && value1 < value2) {
			*out << value1 << ' ';
			if (read_count_0 < k && *in[0] >> value1) {
				++read_count_0;
			}
			else {
				good = false;
			}
		}
		*out << value2 << ' ';
	}

	// all the values from inputfile 0 in the run that are left
	while (good) {
		*out << value1 << ' ';
		if (read_count_0 < k && *in[0] >> value1) {
			++read_count_0;
		}
		else {
			good = false;
		}
	}


}
```

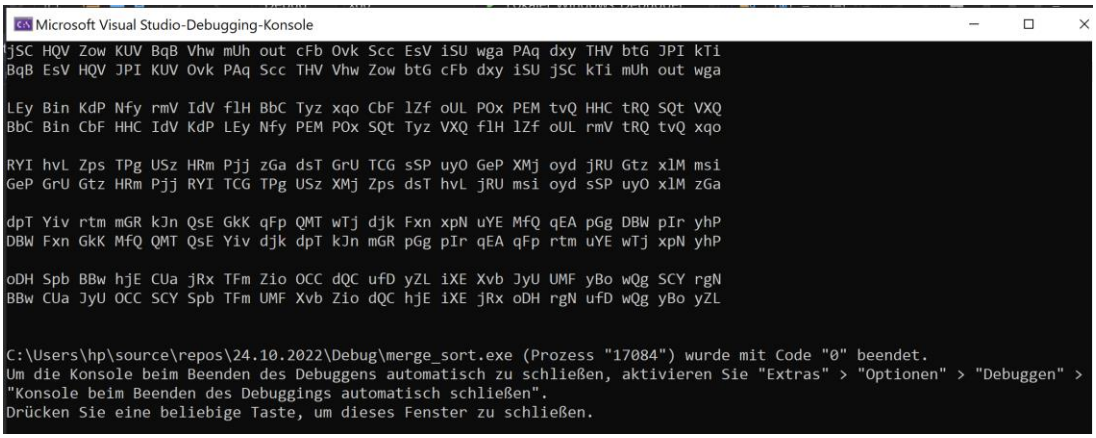
1.3. Testfälle

Random:

5 Dateien werden zufällig mit 20 Sets aus 3 Buchstaben gefüllt und dann sortiert:

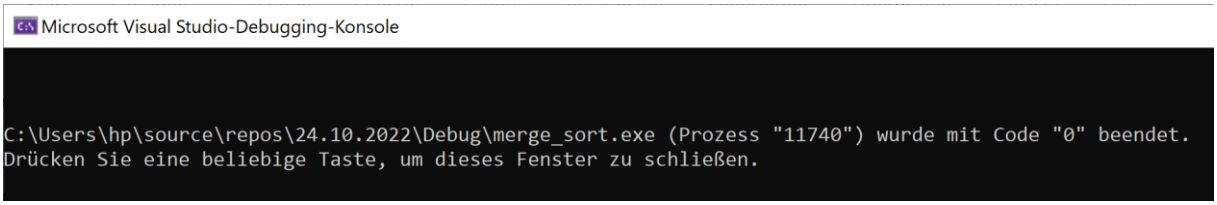
Auf der Konsole wird je die unsortierte und die sortierte Datei untereinander ausgegeben.

Hierbei sind folgende Dateien entstanden: random0.txt, random0_result.txt, random1.txt, random1_result.txt, random2.txt, random2_result.txt, random3.txt, random3_result.txt, random4.txt, random4_result.txt



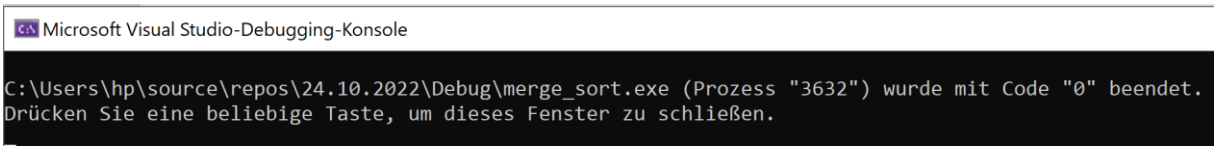
Empty:

Wenn das File leer ist, wird trotzdem eine Datei für das Ergebnis erstellt. Diese ist leer (empty_result.txt). Es wurde ebenfalls die print Funktion für die unsortierte und sortierte Datei aufgerufen:



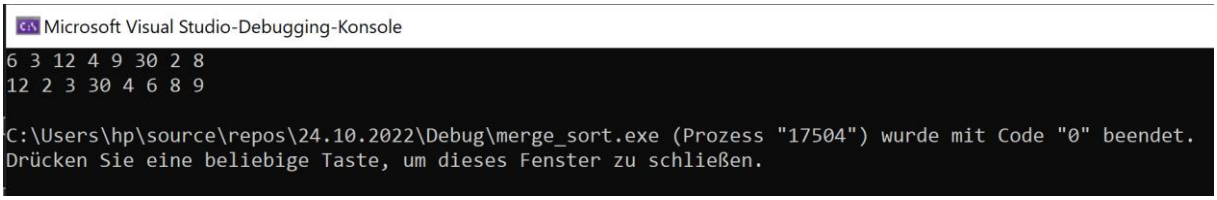
Nicht vorhanden:

Wenn das File nicht vorhanden ist, wird trotzdem eine Datei für das Ergebnis erstellt. Diese ist leer (nonexisting_result.txt). Es wurde ebenfalls die print Funktion für die unsortierte und sortierte Datei aufgerufen, es wurde aber nur ein Zeilenumbruch gemacht:



Zahlen:

Bei diesem Test wurden in einem File einstellige und zweistellige Zahlen gemischt. Da der Algorithmus auf Zeichenketten operiert, sortiert er in erster Stelle auf die erste Ziffer und nicht auf die Länge, was zufolge hat, dass bspw. 12 vor 2 gereiht ist. Ausgangsdatei: numbers.txt, Ergebnis: numbers_result.txt



Umlaute:

Beim letzten Test wurde ein File mit Umlauten verwendet. Das File wurde richtig sortiert, in der Ausgabe mit print, werden die Umlaute jedoch „falsch“ dargestellt. (weird.txt, weird_result.txt)