МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и Структуры Данных»

Тема: Сортировки

Студент гр. 8304 _____ Птухов Д.А.

Преподаватель _____ Фиалковский М.А.

Санкт-Петербург

2019

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Птухов Денис Александрович

Группа 8304

Тема работы: сортировки

Исходные данные:
Написать программу для реализации сортировки слиянием (итеративным и рекурсивным подходом) и быстрой сортировки (итеративным и рекурсивным подходом)

Содержание пояснительной записки:

- Содержание
- Введение
- Сортировка слиянием – итеративный подход и рекурсивный подход
- Быстрая сортировка – итеративный и рекурсивный подход
- Тестирование
- Исходный код
- Использованные источники

Дата выдачи задания: 11.10.2019

Дата сдачи реферата:

Дата защиты реферата: 22.10.2019

Студент(ка) _____ Птухов Д.А.

Преподаватель _____ Фирсов М.А.

АННОТАЦИЯ

В данной работе была создана программа на языке программирования C++, которая сочетает в себе несколько функций: ввода/вывода массива и его сортировки. Также была проведена его оптимизация с целью экономии выделяемой в процессе работы памяти и улучшения быстродействия программы.

SUMMARY

In this work, a program was created in the C ++ programming language, which combines several functions: input / output of an array and its sorting. Also, its optimization was carried out in order to save the memory allocated in the process of working and improve the speed of the program.

СОДЕРЖАНИЕ

	Введение	5
1.	Сортировка слиянием	6
1.1.	Итеративный подход	6
1.2.	Рекурсивный подход	6
2.	Быстрая сортировка	7
2.1.	Итеративный подход	7
2.2.	Рекурсивный подход	7
3.	Тестирование	8
	Заключение	9
	Список использованных источников	10
	Приложение	11

ВВЕДЕНИЕ

Целью данной курсовой работы является закрепление знаний полученных на протяжении семестра. А именно – эффективное использование возможностей c++.

1. СОРТИРОВКА СЛИЯНИЕМ

1.1. Итеративный подход

Для решения поставленной подзадачи была реализована шаблонная функция mergeItSort, которая принимает сортируемый массив и функцию, необходимую для сравнения данных в полученном массиве. Далее при помощи двойного цикла осуществляется последовательная сортировка половинных блоков и их дальнейшее слияние в один отсортированный блок. Далее данные из отсортированного блока переписываются обратно в массив, и вышеописанный алгоритм повторяется, пока не будет отсортирован весь массив. Сложность данного алгоритма постоянна и равна O(n * log(n)), где n – размер исходного массива.

1.2. Рекурсивный подход

Для решения поставленной подзадачи была реализована шаблонная функция mergeRecSort, которая принимает сортируемый массив и функцию, необходимую для сравнения данных в полученном массиве. Далее входной массив делится на 2 равных блока и осуществляется рекурсивный вызов исходной функции от ранее сформированных блоков. Пользуясь тем, что массив длинной 1 считается отсортированным, был реализован выход из рекурсии. Далее аналогично пункту 1.1 происходит слияние двух ранее отсортированных половинных блоков в один. Сложность данного алгоритма постоянна и равна O(n * log(n)), где n – размер исходного массива.

2. БЫСТРАЯ СОРТИРОВКА

2.1. Итеративный подход

Для решения поставленной подзадачи была реализована шаблонная функция quickRecSort, которая принимает сортируемый массив и функцию, необходимую для сравнения данных в полученном массиве. Далее при помощи циклов и стека, в который сохраняются очередные рассматриваемые границы, осуществляется последовательная перестановка элементов, стоящих до опорного и больших его, с элементами, стоящими после опорного и меньших его. Выход из цикла осуществляется посредством проверки стэка на пустоту (то есть проверки на то что массив уже отсортирован). Сложность алгоритма варьируется от O(n * log(n)) до O(n * n). Она зависит от выбранного опорного эл-та.

2.2. Рекурсивный подход

Для решения поставленной подзадачи была реализована шаблонная функция quickRecSort, которая принимает сортируемый массив и функцию, необходимую для сравнения данных в полученном массиве. Далее в вышеописанной функции создается 3 массива smaller, bigger, equal, которые содержат элементы меньшие, большие и равные ранее выбранному опорному элементу. Выбор опорного элемента осуществляется при помощи конструкции rand() % arr.size(). Далее данные массивы заполняются и осуществляется рекурсивный вызов исходной функции от массивов smaller и bigger. Пользуясь тем, что массив длинной 1 считается отсортированным, был реализован выход из рекурсии. Далее входной массив заменяется на сумму массивов smaller, bigger, equal при помощи перегрузки оператора + для массивов. Сложность алгоритма варьируется от O(n * log(n)) до O(n * n). Она зависит от выбранного опорного эл-та.

3. ТЕСТИРОВАНИЕ

INPUT	OUTPUT
5 4 -10 7 8 15 19 100 8888 1 2	-10 1 2 4 5 7 8 15 19 100 8888
1	1
5 4 3 2 1	1 2 3 4 5
8 7 -10 16 38 13 19 5 -1	-10 -1 5 7 8 13 16 19 38
1 2 3 4 5	1 2 3 4 5
0 0 0 0 0 0	0 0 0 0 0 0
8.1 8.01 8.02 9.03 8.001 9.02 9.00001	8.001 8.01 8.02 8.1 9.00001 9.02 9.03
17 27 0 7 40	0 7 17 27 40
A 17 H AKDL	Array elements don't have same type
KKKK AAA A YH GHOSN	A AAA GHOSN KKKK YH
NDNG DKNbn JBG WNNJK GJNOANJ	DKNbn GJNOANJ JBG NDNG WNNJK
A B C D E	A B C D E
EEE EE E EEEE EEEEE	E EE EEE EEEE EEEEE
YO YOY YOYO YOYOY YOOOOOOOOOOOOO	YO YOOOOOOOOOOOOO YOY YOYO YOYOY
())(((())))) ((	((((()))) ()))(

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы были закреплены знания, полученные на протяжении семестра, а именно – эффективное использование возможностей c++.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D1%81%D0%BB%D0%B8%D1%8F%D0%BD%D0%B8%D0%B5%D0%BC
2. https://ru.wikipedia.org/wiki/%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0

ПРИЛОЖЕНИЕ

СОДЕРЖИМОЕ ФАЙЛА COURSEWORK.CPP

```
#include "Header.h"

ReturnType streamsCheck(std::ifstream& in, std::ofstream& out)
{
	return (in && out) ? ReturnType::Correct : ReturnType::IncorrectStreams;
}

TypeCode determineType(std::string const& checkString)
{
	TranformPair<int> intTransform = from_string<int>(checkString);
	if (intTransform.transformResult == true)
		return TypeCode::TypeInt;

	TranformPair<char> charTransform = from_string<char>(checkString);
	if (charTransform.transformResult == true)
		return TypeCode::TypeChar;

	TranformPair<double> doubleTransform = from_string<double>(checkString);
	if (doubleTransform.transformResult == true)
		return TypeCode::TypeDouble;

	return TypeCode::TypeString;
}

void readFileData(std::ifstream& in, StringVector& fileData)
{
	std::string currentFileString;

	while (std::getline(in, currentFileString))
	{
		if (currentFileString.back() == '\r')
			currentFileString.erase(currentFileString.end() - 1);

		fileData.push_back(currentFileString);
	}
}

int main(int argc, char** argv)
{
	srand(time(0));

	if (argc > 2)
	{
		std::ifstream in(argv[1]);
		std::ofstream out(argv[2]);
		ReturnType streamsCheckResult = ReturnType::Correct;

		streamsCheckResult = streamsCheck(in, out);
		if (streamsCheckResult == ReturnType::IncorrectStreams)
		{
			out << "Incorrect streams\n";
			return 0;
		}


		StringVector fileData;
		readFileData(in, fileData);
```

```
		for (auto it = fileData.begin(); it != fileData.end(); ++it)
		{
			std::string& arrStringForm = *it;

			auto searchResult = std::find(arrStringForm.begin(),
arrStringForm.end(), ' ');
			if (searchResult == arrStringForm.end())
			{
				out << arrStringForm << '\n';
				continue;
			}

			std::string firstELement(arrStringForm.begin(), searchResult);
			TypeCode type = determineType(firstELement);
			auto cmp = [](auto a, auto b) {return a < b; };
			auto reverseCmp = [](auto a, auto b) {return a > b; };

			switch (type)
			{
			case TypeCode::TypeInt:
			{
				std::vector<int> arr;
				ReturnType formResult = formArr(arr, arrStringForm);
				if (formResult == ReturnType::IncorrectFileData)
				{
					out << "Array elements don't have same type\n";
					continue;
				}

				quickRecSort(arr, cmp);
				for (auto i : arr)
					out << i << ' ';
				out << '\n';

				break;
			}
			case TypeCode::TypeChar:
			{
				std::vector<char> arr;
				ReturnType formResult = formArr(arr, arrStringForm);
				if (formResult == ReturnType::IncorrectFileData)
				{
					out << "Array elements don't have same type\n";
					continue;
				}

				quickRecSort(arr, cmp);
				for (auto i : arr)
					out << i << ' ';
				out << '\n';

				break;
			}
			case TypeCode::TypeDouble:
			{
				std::vector<double> arr;
				ReturnType formResult = formArr(arr, arrStringForm);
				if (formResult == ReturnType::IncorrectFileData)
				{
					out << "Array elements don't have same type\n";
					continue;
				}
```

```
                        quickRecSort(arr, cmp);
                        for (auto i : arr)
                              out << i << ' ';
                        out << '\n';

                        break;
                  }
                  case TypeCode::TypeString:
                  {
                        std::vector<std::string> arr;
                        ReturnType formResult = formArr(arr, arrStringForm);
                        if (formResult == ReturnType::IncorrectFileData)
                        {
                              out << "Array elements don't have same type\n";
                              continue;
                        }

                        quickRecSort(arr, cmp);
                        for (auto i : arr)
                              out << i << ' ';
                        out << '\n';

                        break;
                  }
                  default:
                        out << "Incorrect type!\n";
                        continue;
                  }
            }
      }

      return 0;
}
```

СОДЕРЖИМОЕ ФАЙЛА HEADER.H

```
#pragma once

#include "SortsHeader.h"
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>

using namespace sorts;
using StringVector = std::vector<std::string>;

enum class ReturnType
{
      IncorrectStreams,
      IncorrectType,
      IncorrectFileData,
      Correct
};

enum class TypeCode
{
      TypeInt,
      TypeChar,
      TypeDouble,
      TypeString
```

```
};

template<typename T>
struct TranformPair
{
	TranformPair(T newVal, bool newTranformResult) : value(newVal),
transformResult(newTranformResult)
	{ }

	T value;
	bool transformResult;
};

void readFileData(std::ifstream&, StringVector&);
ReturnType streamsCheck(std::ifstream&, std::ofstream&);
TypeCode determineType(std::string const&);

template <typename T>
TranformPair<T> from_string(std::string const& checkString)
{
	T value;
	std::istringstream stream(checkString);
	stream >> value;

	if (stream.fail() || stream.peek() != EOF)
		return TranformPair<T>(value, false);

	return TranformPair<T>(value, true);
}

template <typename T>
ReturnType reformArr(std::vector<T>& arr, std::string const& arrStringForm)
{
	auto startPosition = arrStringForm.begin();

	while (1)
	{
		auto searchResult = std::find(startPosition, arrStringForm.end(), '
');
		if (searchResult == arrStringForm.end())
		{
			TranformPair<T> transformToT =
from_string<T>(std::string(startPosition, arrStringForm.end()));
			if (transformToT.transformResult == false)
				return ReturnType::IncorrectFileData;

			arr.push_back(transformToT.value);
			return ReturnType::Correct;
		}
		TranformPair<T> transformToT =
from_string<T>(std::string(startPosition, searchResult));
		if (transformToT.transformResult == false)
			return ReturnType::IncorrectFileData;

		arr.push_back(transformToT.value);

		startPosition = searchResult + 1;
	}

	return ReturnType::Correct;
}

template <typename T>
```

```
ReturnType formArr(std::vector<T>& arr, std::string const& arrStringForm)
{
	ReturnType reformingResult = reformArr(arr, arrStringForm);
	if (reformingResult == ReturnType::IncorrectFileData)
		return ReturnType::IncorrectFileData;

	return ReturnType::Correct;
}

template <typename T, typename FUNC_T>
void sortArr(std::vector<T>& arr, FUNC_T const& cmp)
{
	quickItSort(arr, cmp);
}
```

СОДЕРЖИМОЕ ФАЙЛА SOURCEHEADER.H

```
#pragma once
#include <vector>
#include <stack>
#include <algorithm>
#include <time.h>
#include <memory>

namespace sorts
{
	template <typename T>
	std::vector<T> operator+(std::vector<T> const& left, std::vector<T> const&
right)
	{
		std::vector<T> result(left);
		for (auto i : right)
			result.push_back(i);

		return result;
	}

	template<typename T, typename FUNC_T>
	void quickRecSort(std::vector<T>& arr, FUNC_T const& cmp)
	{
		if (arr.size() <= 1)
			return;

		std::vector<T> smaller;
		std::vector<T> bigger;
		std::vector<T> equal;

		size_t supportIndex = rand() % arr.size();
		T supportingElement = arr[supportIndex];

		for (auto i : arr)
		{
			if (i == supportingElement)
				equal.push_back(i);
			else if (cmp(supportingElement, i))
				bigger.push_back(i);
			else
				smaller.push_back(i);
		}

		quickRecSort(smaller, cmp);
		quickRecSort(bigger, cmp);
```

```
		arr = smaller + equal + bigger;
	}

	template<typename T, typename FUNC_T>
	void mergeRecSort(std::vector<T>& arr, FUNC_T const& cmp)
	{
		if (arr.size() <= 1)
			return;

		std::vector<T> leftPart(arr.begin(), arr.begin() + arr.size() / 2);
		std::vector<T> rightPart(arr.begin() + arr.size() / 2, arr.end());

		mergeRecSort(leftPart, cmp);
		mergeRecSort(rightPart, cmp);

		std::vector<T> result;
		size_t leftArrIndex = 0;
		size_t rightArrIndex = 0;

		while (leftArrIndex < leftPart.size() && rightArrIndex <
rightPart.size())
		{
			if (cmp(leftPart[leftArrIndex], rightPart[rightArrIndex]))
			{
				result.push_back(leftPart[leftArrIndex]);
				leftArrIndex++;
			}
			else
			{
				result.push_back(rightPart[rightArrIndex]);
				rightArrIndex++;
			}
		}

		while (leftArrIndex < leftPart.size())
		{
			result.push_back(leftPart[leftArrIndex]);
			leftArrIndex++;
		}

		while (rightArrIndex < rightPart.size())
		{
				result.push_back(rightPart[rightArrIndex]);
				rightArrIndex++;
		}

		arr = std::move(result);
	}

	template<typename T, typename FUNC_T>
	void mergeItSort(std::vector<T>& arr, FUNC_T const& cmp)
	{
		for (size_t currenttBlockSize = 1; currenttBlockSize < arr.size();
currenttBlockSize *= 2)
		{
			for (size_t blockBorder = 0; blockBorder < arr.size() -
currenttBlockSize; blockBorder += 2 * currenttBlockSize)
			{
				size_t leftBorder = blockBorder;
				size_t middleBorder = leftBorder + currenttBlockSize;
				size_t rightBorder = (middleBorder + currenttBlockSize <
arr.size()) ? middleBorder + currenttBlockSize : arr.size();
```

```
                        std::vector<T> sortedBlock;

                        size_t leftArrIndex = 0;
                        size_t rightArrIndex = 0;
                        while (leftBorder + leftArrIndex < middleBorder &&
middleBorder + rightArrIndex < rightBorder)
                        {
                              T currentLeftArrElement = arr[leftBorder +
leftArrIndex];
                              T currentRightArrELement = arr[middleBorder +
rightArrIndex];

                              if (cmp(currentLeftArrElement,
currentRightArrELement))
                              {

      sortedBlock.push_back(currentLeftArrElement);
                                    leftArrIndex++;
                              }
                              else
                              {

      sortedBlock.push_back(currentRightArrELement);
                                    rightArrIndex++;
                              }
                        }

                        while (leftBorder + leftArrIndex < middleBorder)
                        {
                              sortedBlock.push_back(arr[leftBorder +
leftArrIndex]);
                              leftArrIndex++;
                        }

                        while (middleBorder + rightArrIndex < rightBorder)
                        {
                              sortedBlock.push_back(arr[middleBorder +
rightArrIndex]);
                              rightArrIndex++;
                        }

                        for (size_t insertIndex = leftBorder; insertIndex <
rightBorder; insertIndex++)
                        {
                              arr[insertIndex] = sortedBlock[insertIndex -
leftBorder];
                        }
                  }
            }
      }

      template<typename T, typename FUNC_T>
      void quickItSort(std::vector<T>& arr, FUNC_T const& cmp)
      {
            std::stack<int> indexStack;
            indexStack.push(arr.size() - 1);
            indexStack.push(0);

            int leftBorder = 0;
            int rightBorder = 0;
            int leftIndex = 0;
            int rightIndex = 0;
```

```
            do
            {
                  leftBorder = indexStack.top();
                  indexStack.pop();
                  rightBorder = indexStack.top();
                  indexStack.pop();

                  if (rightBorder - leftBorder == 1 && arr[leftBorder] >
arr[rightBorder])
                  {
                        std::swap(arr[leftBorder], arr[rightBorder]);
                  }
                  else
                  {
                        size_t supportIndex = rand() % arr.size();
                        T supportingElement = arr[supportIndex];

                        leftIndex = leftBorder;
                        rightIndex = rightBorder;
                        do
                        {
                              while (supportingElement > arr[leftIndex])
                                    leftIndex++;

                              while (arr[rightIndex] > supportingElement)
                                    rightIndex--;

                              if (leftIndex <= rightIndex)
                              {
                                    std::swap(arr[leftIndex], arr[rightIndex]);
                                    leftIndex++;
                                    rightIndex--;
                              }

                        } while (leftIndex <= rightIndex);
                  }

                  if (leftBorder < rightIndex)
                  {
                        indexStack.push(rightIndex);
                        indexStack.push(leftBorder);
                  }

                  if (leftIndex < rightBorder)
                  {
                        indexStack.push(rightBorder);
                        indexStack.push(leftIndex);
                  }

            } while (!indexStack.empty());
      }
}
```