Лабораторная работа

Задание 4. Внутренние сортировки.

Вариант 12.

|  |  |
| --- | --- |
| Выполнил студент гр.5130904/30002 | Мадьяров Г.С. |
| Преподаватель | Череповский Д.К. |

Оглавление

[Общая постановка задачи. 2](#_Toc164074136)

[Описание алгоритма решения. 2](#_Toc164074137)

[Требования и тест план. 3](#_Toc164074138)

[Приложение A. Скриншоты работы программы. 4](#_Toc164074139)

[Приложение B. Код программы. 4](#_Toc164074140)

[Вывод. 7](#_Toc164074141)

# Общая постановка задачи.

Для выполнения индивидуального задания по внутренним сортировкам нужно:

• Разработать алгоритм решения индивидуальной задачи.

• Реализовать алгоритм на языке С++:

• написать программу,

• выполнить отладку и тестирование программы.

• Подготовить отчет по заданию, состоящий из разделов:

1. Общая постановка задачи

2. Описание алгоритма решения

3. Сравнение результаты экспериментальной оценки временной сложности с

теоретическими для массивов, состоящих из 1000, 10000, 100000 и 500000 элементов.

Приложение 1. Код программы.

Замечание:

Выполнить тесты для лучшего, худшего и среднего случаев.

Для тестирования использовать функцию, проверяющую упорядоченность последовательности.

Для генерации тестов среднего случая можно использовать псевдослучайные числа.

**Вариант: 12. Сортировка слиянием без рекурсии.**

# Описание алгоритма решения.

Сортировка слиянием (mergesort) является эффективным алгоритмом общего назначения. Сортировка слиянием - это алгоритм "разделяй и властвуй" который был изобретен John von Neumann в 1945 году. Подробное описание и анализ сортировки слиянием снизу вверх появились в отчете Голдстайна и фон Неймана еще в 1948 году.

Алгоритм:

Метод сортировки слиянием снизу вверх использует итерационную технику. Он начинается с одноэлементного массива, а затем объединяет и сортирует два соседних элемента. Объединенные массивы объединяются и сортируются снова, пока не останется только одна единица отсортированного массива. Другими словами, рекурсия не используется в реализации "снизу вверх". Он начинается в нижней части дерева и продвигается вверх, перебирая фрагменты и объединяя их.

Изображение выглядит как снимок экрана, Прямоугольник, прямоугольный, линия

Автоматически созданное описание

Сложность во времени.

Поскольку мы выполняем n слияния для log(n) проходов, сложность по времени составляет O(n log n).

# Требования и тест план.

|  |  |  |
| --- | --- | --- |
| Требование | Данные | Ожидаемый результат |
| Реализация сортировки слиянием без рекурсии. | rand() | Вывод результатов экспериментальной оценки временной сложности для массивов из 1000, 10000, 100000 и 500000 элементов в консоль. |

# Приложение A. Скриншоты работы программы.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рис.1. Экспериментальная оценка временной сложности.

Теоретическая оценка временной сложности.

Для массива из 1000 элементов: O(n log n) = O(1000 \* log(1000)) = O(9965,78)

Для массива из 10000 элементов: O(n log n) = O(10000 \* log(10000)) = O(132877,12)

Для массива из 100000 элементов: O(n log n) = O(100000 \* log(100000)) = O(1,66 \* 10^6)

Для массива из 500000 элементов: O(n log n) = O(500000 \* log(500000)) = O(9,47 \* 10^6)

Теоретическая сложность не коррелирует с практической. Возможно дело в недостаточно рандомной генерации массива чисел.

# Приложение B. Код программы.

Source.cpp

#include "MergeSort.h"

#include <iostream>

#include <chrono>

using namespace std;

const int SIZE = 10;

const int SIZE\_1 = 1000;

const int SIZE\_2 = 10000;

const int SIZE\_3 = 100000;

const int SIZE\_4 = 500000;

int main()

{

cout << "--- Iterative Merge Sorting ---" << endl;

MergeSort arr(SIZE);

arr.fillingArr();

cout << "array (size 10): ";

arr.output();

cout << endl << "Is the array sorted? " << arr.isOrder() << endl;

arr.iterativeMergeSorting();

cout << "array after Iterative Merge Sorting (size 10): ";

arr.output();

cout << endl << "Is the array sorted? " << arr.isOrder() << endl << endl;

MergeSort arr1(SIZE\_1); // size 1000

arr1.fillingArr();

auto begin1 = std::chrono::steady\_clock::now();

arr1.iterativeMergeSorting();

auto end1 = std::chrono::steady\_clock::now();

auto time1 = std::chrono::duration\_cast<std::chrono::microseconds>(end1 - begin1);

std::cout << "The time of sorting (size 1000): " << time1.count() << " microseconds" << endl;

cout << "Is the array sorted? " << arr1.isOrder() << endl;

MergeSort arr2(SIZE\_2); // size 10000

arr2.fillingArr();

auto begin2 = std::chrono::steady\_clock::now();

arr2.iterativeMergeSorting();

auto end2 = std::chrono::steady\_clock::now();

auto time2 = std::chrono::duration\_cast<std::chrono::microseconds>(end2 - begin2);

std::cout << endl << "The time of sorting (size 10000): " << time2.count() << " microseconds" << endl;

cout << "Is the array sorted? " << arr2.isOrder() << endl;

MergeSort arr3(SIZE\_3); // size 100000

arr3.fillingArr();

auto begin3 = std::chrono::steady\_clock::now();

arr3.iterativeMergeSorting();

auto end3 = std::chrono::steady\_clock::now();

auto time3 = std::chrono::duration\_cast<std::chrono::microseconds>(end3 - begin3);

std::cout << endl << "The time of sorting (size 100000): " << time3.count() << " microseconds" << endl;

cout << "Is the array sorted? " << arr3.isOrder() << endl;

MergeSort arr4(SIZE\_4); // size 500000

arr4.fillingArr();

auto begin4 = std::chrono::steady\_clock::now();

arr4.iterativeMergeSorting();

auto end4 = std::chrono::steady\_clock::now();

auto time4 = std::chrono::duration\_cast<std::chrono::microseconds>(end4 - begin4);

std::cout << endl << "The time of sorting (size 500000): " << time4.count() << " microseconds" << endl;

cout << "Is the array sorted? " << arr4.isOrder() << endl;

return EXIT\_SUCCESS;

}

MergeSort.h

#pragma once

#include <iostream>

using namespace std;

class MergeSort

{

private:

int\* data;

size\_t data\_size;

public:

MergeSort(size\_t size);

~MergeSort();

void fillingArr();

void iterativeMergeSorting();

int\* merge(int\* m1, int\* m2, int len\_1, int len\_2);

bool isOrder();

void output();

};

MergeSort.cpp

#include "MergeSort.h"

MergeSort::MergeSort(size\_t size)

{

data\_size = size;

data = new int[size];

}

MergeSort::~MergeSort()

{

delete[] data;

}

void MergeSort::fillingArr()

{

std::srand(std::time(0));

for (int i = 0; i < data\_size; i++)

{

data[i] = std::rand() % 10;

}

}

void MergeSort::output()

{

for (int i = 0; i < data\_size; i++)

{

std::cout << data[i] << " ";

}

}

bool MergeSort::isOrder()

{

bool isOrder = true;

for (int i = 0; i < data\_size - 1; i++)

{

if (data[i] > data[i + 1])

{

isOrder = false;

}

}

return isOrder;

}

void MergeSort::iterativeMergeSorting()

{

int n = 1, k, rem;

int\* arr1;

while (n < data\_size) {

k = 0;

while (k < data\_size) {

if (k + n >= data\_size) break;

rem = (k + n \* 2 > data\_size) ? (data\_size - (k + n)) : n;

arr1 = merge(data + k, data + k + n, n, rem);

for (int i = 0; i < n + rem; ++i)

data[k + i] = arr1[i];

delete[] arr1;

k += n \* 2;

}

n \*= 2;

}

}

int\* MergeSort::merge(int\* arr1, int\* arr2, int len1, int len2)

{

int\* merge\_arr = new int[len1 + len2];

int n = 0;

while (len1 && len2) {

if (\*arr1 < \*arr2) {

merge\_arr[n] = \*arr1;

arr1++;

--len1;

}

else {

merge\_arr[n] = \*arr2;

++arr2;

--len2;

}

++n;

}

if (len1 == 0) {

for (int i = 0; i < len2; ++i) {

merge\_arr[n] = \*arr2;

n++;

arr2++;

}

}

else {

for (int i = 0; i < len1; ++i) {

merge\_arr[n] = \*arr1;

n++;

arr1++;

}

}

return merge\_arr;

}

# Вывод.

Разработал алгоритм для сортировки слиянием без рекурсии на языке C++, сравнил теоретическую и экспериментальную временную сложность.