ГУАП

КАФЕДРА № 43

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ассистент |  |  |  | М. А. Мурашова |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ |
| «АЛГОРИТМЫ СОРТИРОВКИ» |
| по курсу: СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4134к |  |  |  | Костяков Н.А. |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2022

* 1. **Цель работы**

Целью работы является изучение алгоритмов внутренней сортировки и получение практических навыков их использования, и анализа их сложности.

**1.2 Задание на лабораторную работу** Использовать неупорядоченный массив A, содержащий n целочисленных элементов. Величина n определяется по согласованию с преподавателем. Дополнительно в программе должны быть реализованы следующие функции: 1) Поиск элемента либо по его порядковой позиции, либо по его содержимому; 2) Добавление/удаление элемента с последующей пересортировкой последовательности; 3) В программе должен быть реализован подсчет количества сравнений и перестановок, при осуществлении сортировки.

Вариант 17



**Расчет сложности алгоритма**

int i = first, j = last;

double tmp, x= get((first+last)/2);

do {

while (get(i) < x)

i++;

general\_comparison += 1;

while (get(j) > x)

j--;

general\_comparison += 1;

if (i <= j)

{

if (i < j)

{

tmp = get(i);

get\_el(i)->edit(get(j));

get\_el(j)->edit(tmp);

general\_permutation += 1;

}

i++;

j--;

general\_comparison += 1;

}

} while (i <= j);

if (i < last)

sort( i, last);

if (first < j)

sort(first, j);

**Теоретическая временная сложность**

**Лучший случай.**

В наиболее сбалансированном варианте при каждой операции разделения массив делится на две одинаковые (плюс-минус один элемент) части, следовательно, максимальная глубина рекурсии, при которой размеры обрабатываемых подмассивов достигнут 1, составит {\displaystyle \log \_{2}n}log2(n). В результате количество сравнений, совершаемых быстрой сортировкой, было бы равно значению рекурсивного выражения {\displaystyle C\_{n}=2\cdot C\_{n/2}+n}Cn=2\*Cn/2+n, что даёт общую сложность алгоритма O(n\*log2n){\displaystyle O(n\cdot \log \_{2}n)})Щщo

**Среднее.**

Среднюю сложность при случайном распределении входных данных можно оценить лишь вероятностно.

Прежде всего необходимо заметить, что в действительности необязательно, чтобы опорный элемент всякий раз делил массив на две *одинаковых* части. Например, если на каждом этапе будет происходить разделение на массивы длиной 75 % и 25 % от исходного, глубина рекурсии будет равна {\displaystyle \log \_{4/3}n}log4/3, а это по-прежнему даёт сложность {\displaystyle O(n\log n)}O(n log2n). Вообще, при любом *фиксированном* соотношении между левой и правой частями разделения сложность алгоритма будет той же, только с разными константами.

Будем считать «удачным» разделением такое, при котором опорный элемент окажется среди центральных 50 % элементов разделяемой части массива; ясно, вероятность удачи при случайном распределении элементов составляет 0,5. При удачном разделении размеры выделенных подмассивов составят не менее 25 % и не более 75 % от исходного. Поскольку каждый выделенный подмассив также будет иметь случайное распределение, все эти рассуждения применимы к любому этапу сортировки и любому исходному фрагменту массива.

Удачное разделение даёт глубину рекурсии не более {\displaystyle \log \_{4/3}n}log4/3 n. Поскольку вероятность удачи равна 0,5, для получения {\displaystyle k}k удачных разделений в среднем потребуется {\displaystyle 2\cdot k}2k рекурсивных вызовов, чтобы опорный элемент *k* раз оказался среди центральных 50 % массива. Применяя эти соображения, можно заключить, что в среднем глубина рекурсии не превысит {\displaystyle 2\cdot \log \_{4/3}n}2log4/3 n, что равно {\displaystyle O(\log n)}O(log n) А поскольку на каждом уровне рекурсии по-прежнему выполняется не более {\displaystyle O(n)}O(n) операций, средняя сложность составит O(n log n){\displaystyle O(n\log n)}Ooo=

**Худший случай.**

В самом несбалансированном варианте каждое разделение даёт два подмассива размерами 1 и {\displaystyle n-1}n-1, то есть при каждом рекурсивном вызове больший массив будет на 1 короче, чем в предыдущий раз. Такое может произойти, если в качестве опорного на каждом этапе будет выбран элемент либо наименьший, либо наибольший из всех обрабатываемых. При простейшем выборе опорного элемента — первого или последнего в массиве, — такой эффект даст уже отсортированный (в прямом или обратном порядке) массив, для среднего или любого другого фиксированного элемента «массив худшего случая» также может быть специально подобран. В этом случае потребуется {\displaystyle n-1}n-1 операций разделения, а общее время работы составит {\displaystyle \textstyle \sum \_{i=0}^{n}(n-i)=O(n^{2})}∑ (n-i)= O(n2)операций, то есть сортировка будет выполняться за квадратичное время. Но количество обменов и, соответственно, время работы — это не самый большой его недостаток. Хуже то, что в таком случае глубина рекурсии при выполнении алгоритма достигнет n, что будет означать n-кратное сохранение адреса возврата и локальных переменных процедуры разделения массивов. Для больших значений n худший случай может привести к исчерпанию памяти во время работы программы.

В таком случает теоретическая временная сложность составит

T=O(n \* max(O(n), O(n), (Omax(n, n))))= n\*n =n2

**Затраты по памяти**

4 int, 2 double, n int массив

**v=n\*4+4\*4+2\*8= 4n+32 байт**

**Теоретическая пространственная** сложность сотсавляет

V(n)= O(max(n\*4), (4\*4), (2\*8))= O(n)

**Листинг**

#include <iostream>

#include <ctime>

class A

{

public:

A();

~A();

class el

{

public:

el(double val) { value = val; }

~el();

void edit(int val) { value = val; };

el\* next = nullptr;

int value;

};

void add(int val);

void del(int index);

void del\_with\_sort(int index);

void show();

void show\_info();

void sort(int first, int last);

double get(int index);

double get\_data(int val);

int get\_size() { return size; };

el\* get\_el(int index);

int same\_el();

private:

el\* top= nullptr;

int size=0;

int general\_comparison = 0;

int general\_permutation = 0;

};

void A::add(int val)

{

if (top == nullptr)

{

top = new el(val);

size++;

}

else

{

el\* current = top;

while (current->next!=nullptr)

{

current = current->next;

}

current->next = new el(val);

size++;

}

}

void A::del(int index)

{

el\* current = top;

for (int i = 1; i < index; i++)

{

if (current==nullptr)

{

std::cout << "Error, out of range\n";

return;

}

current = current->next;

}

current->next = current->next->next;

size--;

}

void A::del\_with\_sort(int index)

{

del(index);

if (size>1)

{

sort(0, size-1);

}

}

void A::show()

{

for (el\* current = top; current!= nullptr; current = current->next)

{

std::cout << current->value << " ";

}

std::cout << '\n';

}

void A::show\_info()

{

std::cout << "general\_comporisons: " << general\_comparison << "\ngeneral\_permutations: " << general\_permutation << "\n";

general\_comparison = 0;

general\_permutation = 0;

}

void A::sort(int first, int last)

{

int i = first, j = last;

double tmp, x= get((first+last)/2);

do {

while (get(i) < x)

i++;

general\_comparison += 1;

while (get(j) > x)

j--;

general\_comparison += 1;

if (i <= j)

{

if (i < j)

{

tmp = get(i);

get\_el(i)->edit(get(j));

get\_el(j)->edit(tmp);

general\_permutation += 1;

}

i++;

j--;

general\_comparison += 1;

}

} while (i <= j);

if (i < last)

sort( i, last);

if (first < j)

sort(first, j);

}

double A::get(int index)

{

el\* current = top;

for (int i = 0; i < index; i++)

{

current = current->next;

}

return current->value;

}

double A::get\_data(int val)

{

int i = 0;

for (el\* current = top; current != nullptr; current = current->next)

{

if (current->value==val)

{

return i;

}

i++;

}

return -1;

}

A::el\* A::get\_el(int index)

{

el\* current = top;

for (int i = 0; i < index; i++)

{

current = current->next;

}

return current;

}

int A::same\_el()

{

int counte=0;

el\* current = top;

int ignore[14] = {};

for (int i = 0; i < size; i++)

{

int temp = current->value;

el\* compare = current->next;

for (int j = i; j < size-1; j++)

{

bool flag = 0;

if (temp == compare->value)

{

for (int q = 0; q < size; q++)

{

if (temp == ignore[q])

{

flag = 1;

}

}

if (flag==0)

{

ignore[counte] = temp;

counte++;

}

}

compare = compare->next;

}

current = current->next;

}

return counte;

}

A::A()

{

}

A::~A()

{

}

int main()

{

A array;

std::srand(time(0));

for (int i = 0; i < 14; i++)

{

array.add(rand()%10);

}

array.show();

std::cout << "1) to sort array\t2)to delete index\t3)to delete index with sort\n4)to find by index\t5)to find by value\n6)to find eq elements\n\n";

int var =0;

int index;

while (var<8)

{

std::cin >> var;

switch (var)

{

case 1:

array.sort(0, array.get\_size()-1);

array.show();

array.show\_info();

std::cout << '\n';

break;

case 2:

std::cout << "Enter index: ";

std::cin >> index;

array.del(index);

array.show();

std::cout << '\n';

break;

case 3:

std::cout << "Enter index: ";

std::cin >> index;

array.del\_with\_sort(index);

array.show();

array.show\_info();

std::cout << '\n';

break;

case 4:

std::cout << "Enter index: ";

std::cin >> index;

std::cout << array.get(index)<<'\n';

std::cout << '\n';

break;

case 5:

std::cout << "Enter value: ";

std::cin >> index;

std::cout << array.get\_data(index);

std::cout << '\n';

break;

case 6:

std::cout << "There are " << array.same\_el()<<" same elements\n";

break;

default:

break;

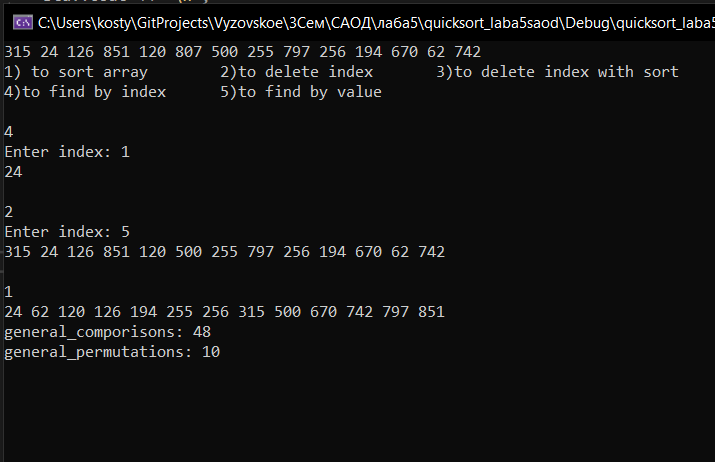
}

}

std::cout << array.get(13);

}

**Результат**



**Вывод**

Я освоил метод быстрой сортировки Хоара