3 вариант

Цель работы – исследование структур данных, методов их обработки и оценки.

Задача:

Работая с таблицей использовать методы

Поиска - вычисление адреса

Упорядочения - Шелла

Корректировки - Замена данных

Даны N записей вида: код материала, дата поступления, номер склада, количество поступившего материала, стоимость материала

## Структура данных

В качестве исходной структуры данных был выбран массив записей.

Реализация:

struct Material

{

unsigned short int id;

time\_t date;

unsigned short int storage;

unsigned short int number;

unsigned short int cost;

};

Material materials[500];

## Определение объема памяти

Объем памяти отводимый под 1 элемент:

V1 = 4Vshort int + Vtime\_t = 4 \* 2 + 8 = 16 байт

Размер всего массива:

V = V1 \* N = 16 N

## 

## Анализ алгоритма поиска

Согласно заданию используется метод вычисления адреса.

Реализация:

Material searchByAddr(unsigned int n, Material arr[]) {

unsigned int addr = addrFunc(n);

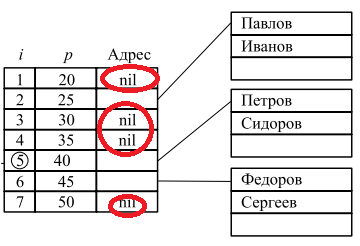
return arr[addr];

}

Так как использован массив (таблица прямого доступа), длина поиска и средняя длина поиска равны единице (S = 1, L = 1).

Но данный метод может являться крайне ресурсоемким с точки зрения памяти, если аргумент адресной функции изменяется так, что невозможно написать функцию изменяющуюся в сопоставимых с количеством записей пределах, вследствие чего большое число ячеек памяти будет зарезервировано программой, но не будет использовано для хранения входных данных.

Иллюстрация на примере из методического пособия:



## Анализ алгоритма сортировки

Согласно заданию использована сортировка Шелла, реализованная на С++ (сортировка по цене материала)

void ShellSort(Material A[], int n)

{

int i, j, d;

Material buff;

d = n;

d = d / 2;

while (d > 0)

{

for (i = 0; i < n - d; i++)

{

j = i;

while (j >= 0 && A[j].cost > A[j + d].cost)

{

buff = A[j];

A[j] = A[j + d];

A[j + d] = buff;

j--;

}

}

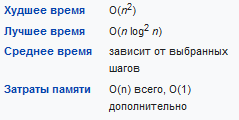
d = d / 2;

}

}

Число сравнений в методе Шелла C <= 0.5 N3/2

Также, некоторые количественные характеристики сортировки Шелла



## Анализ алгоритма корректировки

В задании требуется реализовать корректировку заменой данных.

1. В случае если требуется корректировка одного параметра (на примере стоимости)

Реализация:

void ChangeCost(Material arr[], int n, unsigned short int new\_cost) {

arr[n].cost = new\_cost;

}

t1 = t[] + t= = 2 + 2 = 4 такта

1. В случае замены всех параметров

Реализация:

void ChangeAll(Material arr[], int n, unsigned short int new\_id, time\_t new\_date, unsigned short int new\_storage,

unsigned short int new\_number, unsigned short int new\_cost) {

arr[n].id = new\_id;

arr[n].date = new\_date;

arr[n].storage = new\_storage;

arr[n].number = new\_number;

arr[n].cost = new\_cost;

}

t = 5 \* (t[] + t=) = 5 \* (2 + 2) = 20 тактов

## Вывод

Алгоритм корректировки является оптимальным, алгоритм сортировки близок к оптимальному, но неэффективен в “худшем случае” исходного массива. Алгоритм поиска эффективен по времени, но может быть крайне неэффективен по памяти и не универсален (сильно зависит от параметров поиска).

## Альтернативные варианты структуры и методов ее обработки

В качестве структуры данных сохранен массив, так как дерево пришлось бы перестраивать каждый раз при смене критерия сортировки, а поиск элемента в списке может быть осуществлен только последовательно, что требует временных затрат.

## Альтернативный алгоритм поиска

В качестве альтернативного алгоритма поиска используется двоичный поиск, так как он быстрее последовательного и при этом более универсален, чем метод вычисления адреса (позволяет не оставлять незаполненных элементов в массиве).

Реализация двоичного поиска (пример - по стоимости):

int BinarySearch(Material arr[], int n, int cost, Material& result)

{

int left = 0;

int right = n;

int midd = 0;

result = arr[0];

while (left <= right)

{

midd = (left + right) / 2;

std::cout << left << " " << midd << " " << right << "\n";

if (cost < arr[midd].cost)

right = midd - 1;

else if (cost > arr[midd].cost)

left = midd + 1;

else {

result = arr[midd];

return midd;

}

if (left > right)

return -1;

}

}

Среднее число операций сравнения в таком случае



## Альтернативный алгоритм сортировки

В качестве альтернативного алгоритма сортировки выбрана сортировка слиянием, имеющая большую эффективность по времени, но более ресурсоемкая с точки зрения памяти.

Реализация:

void Merge(Material\* A, int first, int last)

{

int middle, start, final, j;

Material\* mas = new Material[500];

middle = (first + last) / 2;

start = first;

final = middle + 1;

for (j = first; j <= last; j++)

if ((start <= middle) && ((final > last) || (A[start].cost < A[final].cost)))

{

mas[j] = A[start];

start++;

}

else

{

mas[j] = A[final];

final++;

}

for (j = first; j <= last; j++) A[j] = mas[j];

delete[]mas;

};

void MergeSort(Material\* A, int first, int last)

{

{

if (first < last)

{

MergeSort(A, first, (first + last) / 2);

MergeSort(A, (first + last) / 2 + 1, last);

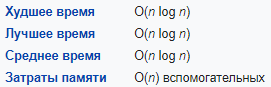
Merge(A, first, last);

}

}

};

Характеристики сортировки слиянием:



## Вывод

В ходе пересмотра исходной структуры данных набора методов и набора методов был сделан выбор в пользу более общих алгоритмов поиска и сортировки, позволяющих в среднем более эффективно работать с произвольными входными данными(в рамках заданных ограничений).

## Сравнительная таблица

|  |  |  |
| --- | --- | --- |
|  | Исходный вариант | Альтернативный вариант |
| Структура данных | массив записей | массив записей |
| Объем памяти | 16 N | 16 N |
| Метод корректировки | заменой значения | заменой значения |
| Метод поиска | вычисления адреса | двоичный |
| Алгоритм сортировки | сортировка Шелла | сортировка слиянием |
| Сложность корректировки | tmin = 4  tmax = 20 | tmin = 4  tmax = 20 |