Министерство Образования Республики Молдова

Молдавский государственный университет  
Факультет: Математика и информатика

Лабораторная работа №1

По курсу **ASDC**

Работу выполнил студент группы I1702:   
**Симонян Абгар**

Работу проверила:  
**Каролина Опинка**

**Задания**

Создать текстовый файл, который содержит не менее 50 записей, не менее 5 полей и не менее 2 типов данных, ключевое поле должно быть уникальным и неупорядоченным.

Внедрить несколько методов поиска в таблицах и сделать поиск по ключевому полю из текстового файла и для каждого метода сделать следующее:

- Анализировать среднею теоретическую и практическую длинну поиска. Описать каждый алгоритм метода поиска.

1 Последовательнный метод поиска

3 Поиск в неупорядоченных древовидных таблицах

2 Бинарный метод поиска

3 Поиск в хеш-таблицах

1 Реферат на тему „Поиск”

**Решение**

1. ) **Последовательнный метод поиска**

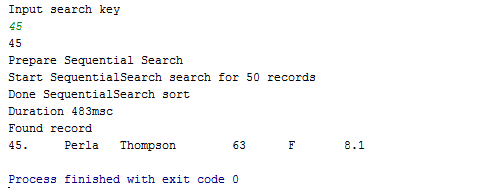
Линейный, последовательный поиск — [алгоритм](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC" \o "Алгоритм) нахождения заданного значения произвольной функции на некотором отрезке. Данный алгоритм является простейшим алгоритмом поиска и, в отличие, например, от [двоичного поиска](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA" \o "Двоичный поиск), не накладывает никаких ограничений на функцию и имеет простейшую реализацию.

**Реализация**

**for** (**int** i = 0; i < service->recordCount; i++) {  
 **if** (array[i]->key == searchFor) {  
 foundElem = array[i];  
 found = **true**;  
 **break**;  
 }  
}

Как видим - реализация это обычный **for** цикл.

**Пример работы алгоритма:**



**Вычислительная сложность алгоритма** в любом случае [O](https://ru.wikipedia.org/wiki/%C2%ABO%C2%BB_%D0%B1%D0%BE%D0%BB%D1%8C%D1%88%D0%BE%D0%B5_%D0%B8_%C2%ABo%C2%BB_%D0%BC%D0%B0%D0%BB%D0%BE%D0%B5" \o ")(n).

В связи с малой эффективностью по сравнению с другими алгоритмами линейный поиск обычно используют, только если отрезок поиска содержит очень мало элементов, тем не менее, линейный поиск не требует дополнительной памяти или обработки/анализа функции, так что может работать в потоковом режиме при непосредственном получении данных из любого источника. Также линейный поиск часто используется в виде линейных алгоритмов поиска максимума/минимума.

1. **)Бинарный метод поиска**

**Двоичный** (**бинарный**) **поиск**(также известен как метод деления пополам и [дихотомия](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%85%D0%BE%D1%82%D0%BE%D0%BC%D0%B8%D1%8F" \o "Дихотомия)) — классический [алгоритм](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC" \o "Алгоритм) поиска элемента в отсортированном массиве (векторе), использующий дробление массива на половины. Используется в [информатике](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0" \o "Информатика), [вычислительной математике](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0" \o "Вычислительная математика) и [математическом программировании](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5" \o "Математическое программирование).

**Реализация**

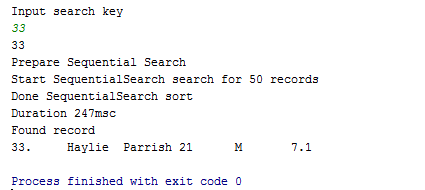
DataModel \*BinarySearch::binarySearch(DataModel \*array[], **int** left, **int** right, **int** searchFor) {  
  
 **if** (right >= left) {  
 **int** mid = left + (right - left) / 2;  
  
 *// If the element is present at the middle  
 // itself* **if** (array[mid]->key == searchFor) {  
 **this**->found = **true**;  
 **return** array[mid];  
 }  
  
 *// If element is smaller than mid, then  
 // it can only be present in left subarray* **if** (array[mid]->key > searchFor)  
 **return**

binarySearch(array, left, mid - 1, searchFor);  
  
 *// Else the element can only be present  
 // in right subarray* **return** binarySearch(array, mid + 1, right, searchFor);  
 }  
  
 *// We reach here when element is not  
 // present in array* std::cout << **"Nothing found"** << std::endl;  
 **return nullptr**;  
}

foundElem = binarySearch(array, 0, n - 1, searchFor);

1. Определение значения элемента в середине структуры данных. Полученное значение сравнивается с ключом.
2. Если ключ меньше значения середины, то поиск осуществляется в первой половине элементов, иначе — во второй.
3. Поиск сводится к тому, что вновь определяется значение серединного элемента в выбранной половине и сравнивается с ключом.
4. Процесс продолжается до тех пор, пока не будет найден элемент со значением ключа или не станет пустым интервал для поиска.

**Пример работы алгоритма:**



**Вычислительная сложность алгоритма:**

В случае, когда первое значение mid совпадает с ключом, тогда считается, что алгоритм выполнился за свое лучшее время O(1). В среднем и худшем случае время работы алгоритма составляет O(logn), что значительно быстрее, чем у линейного поиска, требующего линейного времени.

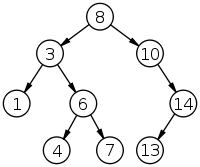
**3.)Поиск в неупорядоченных древовидных таблицах**

**Поиск в неупорядоченных древовидных таблицах** это метод поиска в двоичном дереве поиска для которого выполняются следующие дополнительные условия (свойства дерева поиска):

- Оба поддерева — левое и правое — являются двоичными деревьями поиска.

- У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X.

- У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X.



**Алгоритм**:

Если дерево пусто, сообщить, что узел не найден, и остановиться.

Иначе сравнить K со значением ключа корневого узла X.

Если K=X, выдать ссылку на этот узел и остановиться.

Если K>X, рекурсивно искать ключ K в правом поддереве Т.

Если K<X, рекурсивно искать ключ K в левом поддереве Т.

**Реализация**

Структура дерева поиска

**struct** TreeTable {  
  
 **int** key;  
 DataModel \*data;  
  
 **struct** TreeTable \*left, \*right;  
};

Сначала заполняет дерево поиска

**void** UnsortedTreeTableSearch::populateTreeTableStruct

(DataModel \*array[], **int** recordCount) {  
  
 **for** (**int** i = 0; i < recordCount; i++) {  
 **if** (i == 0) {  
 *// Allocate memory for new node* **struct** TreeTable \*initialPoint =

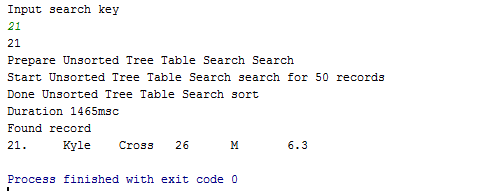
(**struct** TreeTable \*) malloc(**sizeof**(**struct** TreeTable));  
  
 *// Assign data to this node* initialPoint->key = array[i]->key;  
 initialPoint->data = array[i];  
  
 *// Initialize left and right children as NULL* initialPoint->left = **nullptr**;  
 initialPoint->right = **nullptr**;  
  
 **this**->rootNode = initialPoint;  
 } **else** {  
 TreeTable \*searchPoint = rootNode;  
 **int** key = array[i]->key;  
 **bool** searching = **true**;  
 **bool** left = **true**;  
  
 **while** (searching) {  
 **if** (searchPoint->key < key) {  
 **if** (searchPoint->left != **nullptr**) {  
 searchPoint = searchPoint->left;  
 } **else** {  
 searching = **false**;  
 left = **true**;  
 }  
 } **else** {  
 **if** (searchPoint->right != **nullptr**) {  
 searchPoint = searchPoint->right;  
 } **else** {  
 searching = **false**;  
 left = **false**;  
 }  
 }  
 }  
  
 *// Allocate memory for new node* **struct** TreeTable \*newPoint =

(**struct** TreeTable \*) malloc(**sizeof**(**struct** TreeTable));  
  
 *// Assign data to this node* newPoint->key = array[i]->key;  
 newPoint->data = array[i];  
  
 *// Initialize left and right children as NULL* newPoint->left = **nullptr**;  
 newPoint->right = **nullptr**;  
 **if** (left) {  
 searchPoint->left = newPoint;  
 } **else** {  
 searchPoint->right = newPoint;  
 }  
  
 }  
 }  
}

Алгоритм поиска

**auto** start = std::chrono::high\_resolution\_clock::now();  
std::cout << **"Start Unsorted Tree Table Search search for "** << n << **" records"** << std::endl;  
  
  
**int** i = 0;  
TreeTable \*searchPoint = rootNode;  
  
**bool** search = **true**;  
  
**while** (search) {  
  
 **if** (searchPoint->key == searchFor) {  
 search = **false**;  
 foundElem = searchPoint->data;  
 **this**->found = **true**;  
 } **else if** (searchPoint->key >= searchFor) {  
 **if** (searchPoint->right != **nullptr**) {  
 searchPoint = searchPoint->right;  
 } **else** {  
 search = **false**;  
 }  
 } **else** {  
 **if** (searchPoint->left != **nullptr**) {  
 searchPoint = searchPoint->left;  
 } **else** {  
 search = **false**;  
 }  
 }  
 i++;  
}  
  
**auto** end = std::chrono::high\_resolution\_clock::now();  
std::cout << **"Done Unsorted Tree Table Search sort"** << std::endl;

**Пример работы алгоритма:**



**Вычислительная сложность алгоритма:**

В среднем сложность алгоритма составляет O(logn) В худшем случае сложность данного алгоритма O(n).

**4. )Поиск в хэш таблицах**

**Хеш-табли́ца**— это [структура данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85" \o "Структура данных), реализующая интерфейс [ассоциативного массива](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2" \o "Ассоциативный массив), а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу. После создания Хэш-таблицы поиск данных идёт сначала по таблице а затем по ассоциации найденного элемента получаем результат в настоящей таблице данных.

**Реализация**

Хэш таблица

**int** hashTable[100];

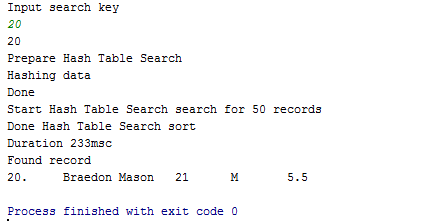
Хэширование данных

**void** HashTableSearch::hashData(DataModel \*array[], **int** amountOfRecordsToHash) {  
  
 std::cout << **"Hashing data"** << std::endl;  
  
 **for** (**int** i = 0; i < amountOfRecordsToHash; i++) {  
 **this**->hashTable[i] = array[i]->key;  
 }  
  
 std::cout << **"Done"** << std::endl;  
}

Поиск в хэш-таблице

hashData(array, n);  
  
**auto** start = std::chrono::high\_resolution\_clock::now();  
std::cout << **"Start Hash Table Search search for "** << n << **" records"** << std::endl;  
  
**for**(**int** i = 0; i < n; i++){  
 **if**(searchFor == **this**->hashTable[i]){  
 foundElem = array[i];  
 **this**->found = **true**;  
 **break**;  
 }  
}  
  
**auto** end = std::chrono::high\_resolution\_clock::now();  
std::cout << **"Done Hash Table Search sort"** << std::endl;

**Пример работы алгоритма:**



**Вычислительная сложность алгоритма:**

В среднем сложность алгоритма составляет O(1) В худшем случае сложность данного алгоритма O(n).

**Завершение**

Поиск данных — раздел [информатики](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0" \o "Информатика), изучающий алгоритмы для поиска и обработки информации как в структурированных (см. напр. [базы данных](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B7%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85" \o "База данных)) так и [неструктурированных](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5" \o "Неструктурированные данные) (напр., текстовый документ) данных. Поиск данных неразрывно связан с понятием [фильтрации данных](https://ru.wikipedia.org/wiki/%D0%A4%D0%B8%D0%BB%D1%8C%D1%82%D1%80_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)" \o "Фильтр (информатика)).

В общем случае, [поиск](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA" \o "Поиск) означает исследование чего-либо с целью нахождения неочевидной, утерянной или спрятанной части. Поиск данных обычно связан с обработкой некоторого хранилища данных, прочесть или осознать которые последовательно не представляется возможным, с целью найти интересующее постановщика задачи подмножество этих данных (или установить их отсутствие). Алгоритмы эффективного поиска существовали задолго до появления компьютеров и применялись, к примеру, для нахождения книг в библиотеках. Существует неинформированный поиск данных, когда алгоритмы могут обрабатывать любые данные независимо от их сути, например, побитовый поиск. Часто, более эффективными является информированный поиск, например, системы автоматизированного нахождения отпечатков пальцев, фонетический поиск в текстах и т. д.

Выбор алгоритма поиска сводится к тому как структурированы данные, как много данных нужно попадают под поиск, сложность алгоритма.