**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ**

(**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**)» (**МАИ)**

Институт №3 «Системы управления, информатика и электроэнергетика»

Кафедра 304 Вычислительные машины, системы и сети

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ **№ 2**

**“Алгоритмы последовательного поиска”**

Группа: *М30-209Б-19*

Выполнил:

*Кузнецов И.И*

Проверил:

*Ким Роман Валерьевич*

**Москва 2020**

**Функция для генерации случайных массивов**

**static void RandomArray(T \*array, int size, T &key, int seed) {  
 srand(seed+key); //рекурсивно сбрасываем сид для генерации  
 //начальный сид - время  
 for (int i = 0; i < size; ++i) {  
 array[i] = 48 + rand() % 75;  
 //48 - пропускаем спец символы  
 //rand [0;74]; 48 - 122  
 }  
 key = rand() % 123;  
}**

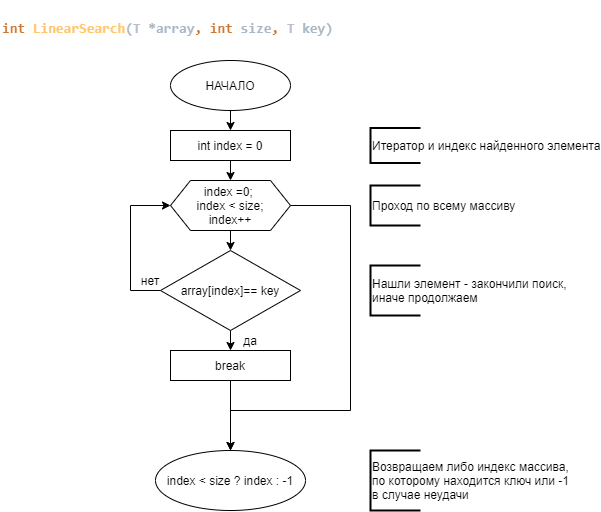
**Главная функция**

**int main() {  
 unsigned seed = unsigned(std::time(nullptr));  
 int size = 200;  
 int comparisons = 0;  
 char key;  
 char array[size];  
 for (int i = 0; i < 1000; ++i) {  
 RandomArray(array,size,key, seed);  
 sort(array, array+size); //если бинарный или поиск по таблице  
 BinarySearch(array, size, key, comparisons);  
 }  
 cout << "Comparisons: " << comparisons << endl;  
 cout << "Average number of comparisons: " << comparisons / 1000;  
 return 0;  
}**

**Описание функции линейного поиска**

Поиск производится путем прохода по всему массиву и сравнения на каждой итерации значения массива со значением искомого ключа, в случае успеха выходим из цикла и возвращаем индекс ключа в массиве, иначе ошибку (значение -1).

**Структурная схема алгоритма**

****

**Код алгоритма**

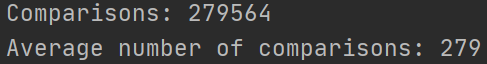
**static int LinearSearch(T \*array, int size, T key) {**  
 **int index = 0;  
 int comparisons = 0;  
 for (index = 0; index < size; index++) {  
 if (array[index] == key) break;**

**comparisons += 2; //index < size; array[index] == key  
 }  
 cout << "LinearSearch comparisons: " << comparisons << endl;  
 return index < size ? index : -1;**

**Результаты работы программы**

Размер массива: **200**

Количество экспериментов: **1000**

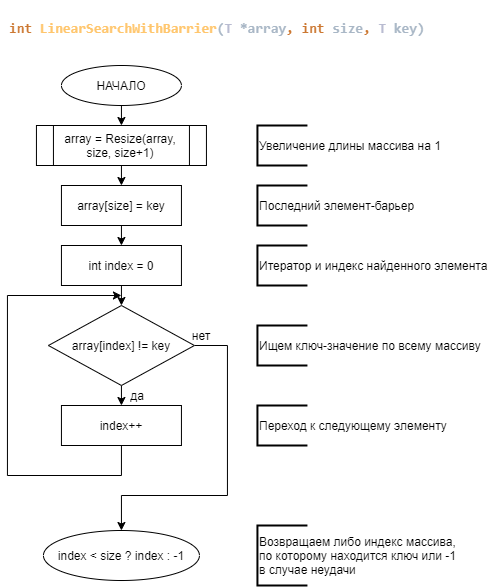


Среднее количество сравнений: **279**

**Описание функции линейного поиска с барьером**

Улучшенная версия последовательного поиска. Улучшение производится путем увеличения массива на 1 справа, на место пустой добавленной ячейки вставляется значение искомого ключа. Таким образом мы убираем лишнее сравнение "достигнут ли конец массива", что ускоряет работу алгоритма.

**Структурная схема алгоритма**

****

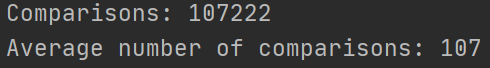
**Код алгоритма**

**static int LinearSearchWithBarrier(T \*array, int size, T key) {  
 int comparisons = 0;   
 array = Resize(array, size, size + 1);  
 array[size] = key;  
 int index = 0;  
 while (array[index] != key) {  
 index++;  
 comparisons++; //array[index] != key  
 }  
 cout << "LinearSearchWithBarrier comparisons: " << comparisons << endl;  
 return index < size ? index : -1;  
}**

**Результаты работы программы**

Размер массива: **200**

Количество экспериментов: **1000**



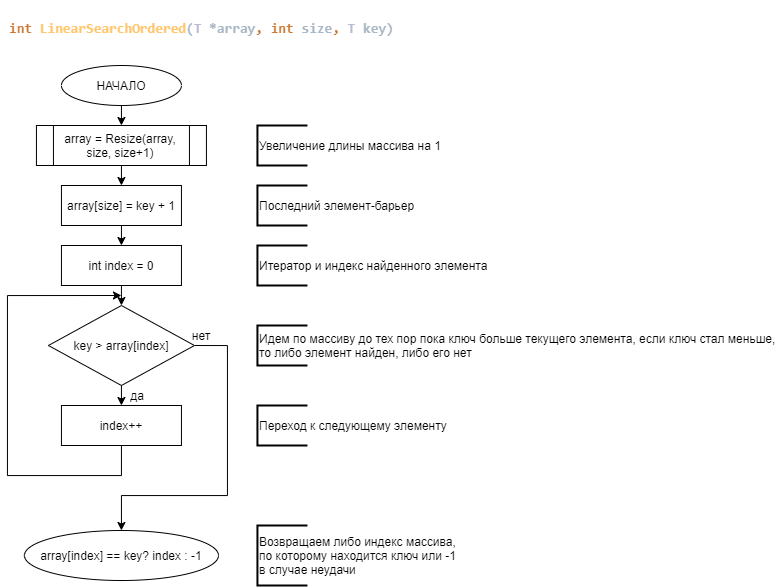
Среднее количество сравнений: **107**

**Описание функции последовательного**

**поиска в упорядоченной таблице**

Модификация последовательного поиска, но с учетом того, что поиск производится в упорядоченном массиве. На каждой итерации проверяем, больше ли наш ключ текущего значения, если больше, то идем далее, если нет, то элемент либо найден, либо его нет в последовательности. Также на месте size+1 элемента стоит барьер, который всегда больше ключа, это обеспечивает безопасность работы с памятью и убирает лишнюю проверку "достигнут ли конец массива"

**Структурная схема алгоритма**

****

**Код алгоритма**

**static int LinearSearchOrdered(T \*array, int size, T key) {  
 array = Resize(array, size, size + 1);  
 array[size] = key + 1;  
 int index = 0;  
 int comparisons = 0;  
 while (key > array[index]) {  
 comparisons++;  
 index++;  
 }  
 cout << "LinearSearchOrdered comparisons: " << comparisons << endl;  
 return array[index] == key ? index : -1;  
}**

**Результаты работы программы**

Размер массива: **200**

Количество экспериментов: **1000**



Среднее количество сравнений: **70**

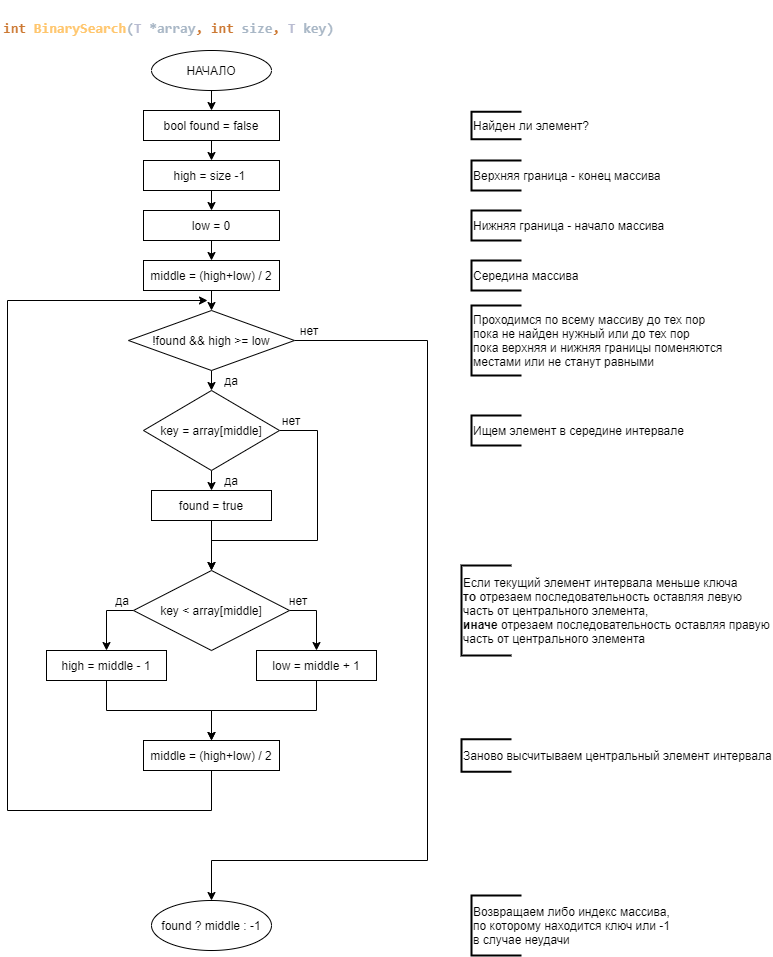
**Описание функции бинарного поиска**

**Шаг 1.** Определить номер среднего элемента массива **middle=(high+low)/2.**

**Шаг 2.** Если значение среднего элемента массива равно искомому, то возвращаем значение, равное номеру искомого элемента, и алгоритм завершает работу.

**Шаг 3.** Если искомое значение больше значения среднего элемента, то возьмем в качестве массива все элементы справа от среднего, иначе возьмем в качестве массива все элементы слева от среднего. Перейдем к **Шагу 1.**

**Структурная схема алгоритма**

****

**Код алгоритма**

**static int BinarySearch(T \*array, int size, T key) {  
 int comparisons = 0;  
 bool found = false;  
 int high = size - 1;  
 int low = 0;  
 int middle = (high + low) / 2;  
 while (!found && high >= low) {  
 comparisons += 2; //!found; high>=low;  
 if (key == array[middle]) {  
 found = true;  
 comparisons++;  
 } else if (key < array[middle]) {  
 high = middle - 1;  
 comparisons++;  
 } else {  
 low = middle + 1;  
 comparisons++;  
 }  
 middle = (high + low) / 2;  
 }  
 cout << "BinarySearch comparisons: " << comparisons << endl;  
 return found ? middle : -1;  
}**

**Результаты работы программы**

Размер массива: **200**

Количество экспериментов: **1000**

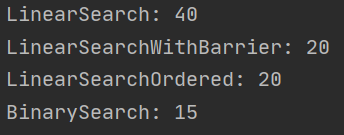


Среднее количество сравнений: **18**

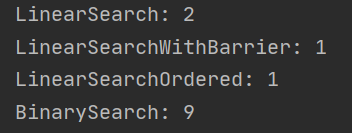
**Сопоставительный анализ**

**Для массива в 20 элемент *(количество сравнений)*:**

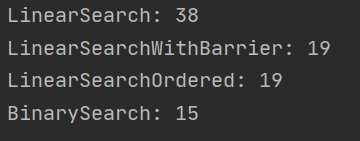
**Ключ поиска не найден:**

****

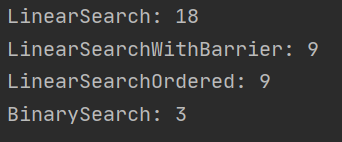
**Ключ поиска находится в начале массива:**

****

**Ключ поиска находится в конце массива:**

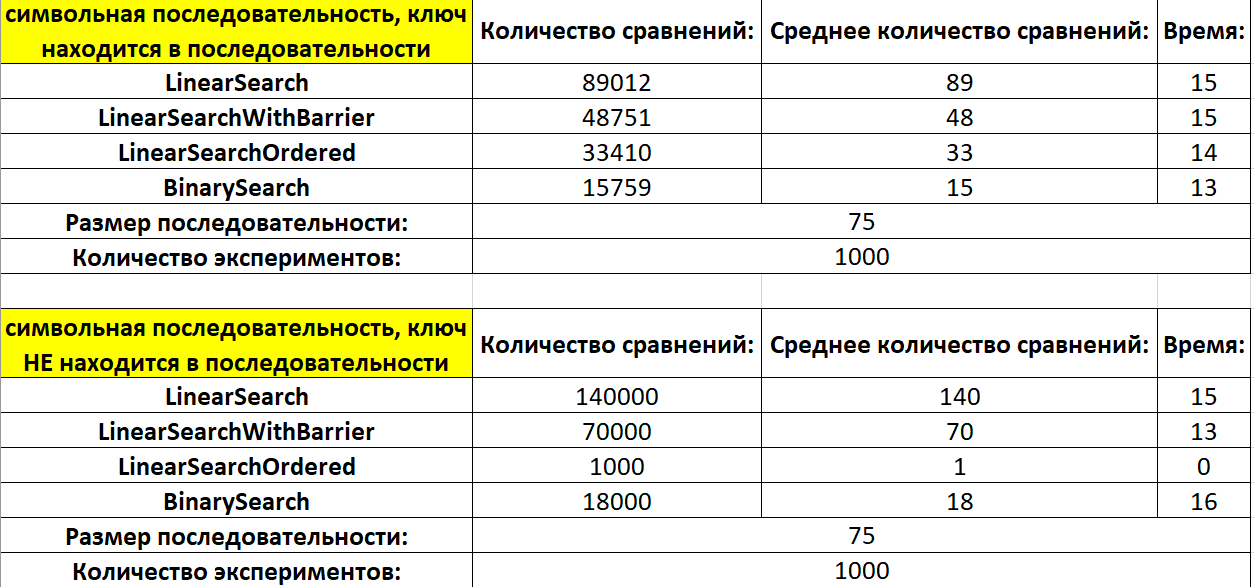
****

**Ключ поиска находится в середине массива:**

****

**Сопоставительный анализ с символьной последовательностью**

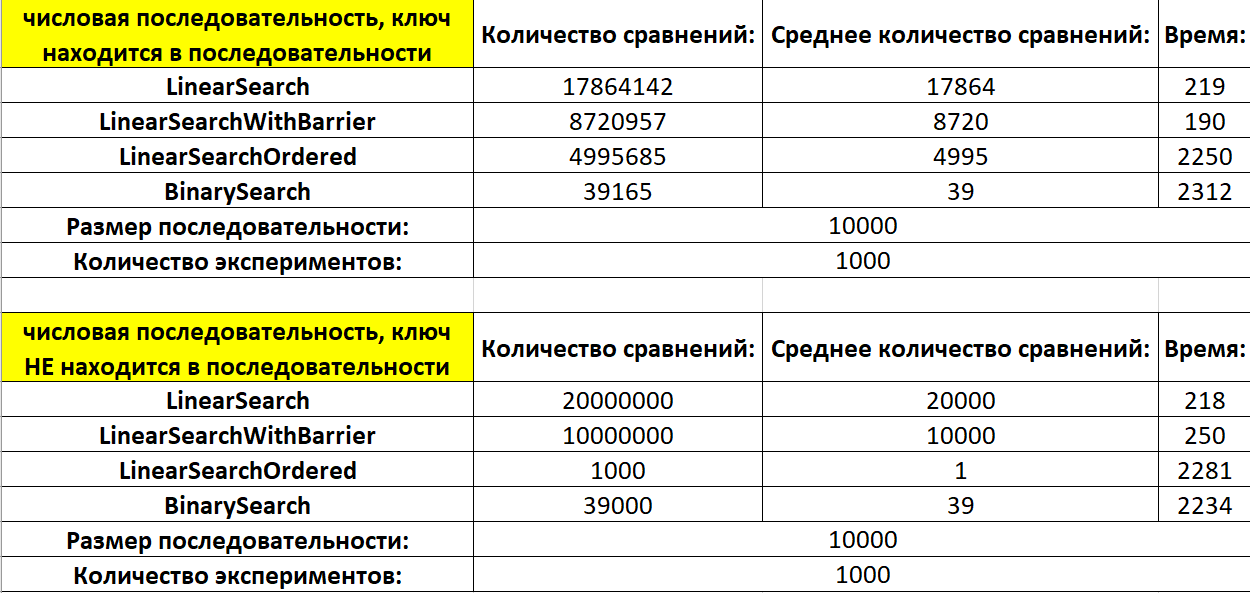
**Т.к. количество символов у нас ограничено, а повторное вхождение по условию задачи запрещено, ограничим количество последовательности**

****

По результатам работы, можно заметить, что обычный линейный поиск является самым медленным, линейный поиск с барьером, как и ожидалось работает в два раза быстрее, чем обычный линейный поиск.

Если рассматривать ситуацию с поиском в упорядоченном массиве, то выигрывает бинарный поиск, но, когда ключ отсутствует в последовательности линейный последовательный поиск явно опережает бинарный поиск.

Аналогичная ситуация происходит и в числовой последовательности, когда размер последовательности на порядок больше



**Вывод:**

Таким образом мы изучили бинарный поиск и несколько вариаций последовательного поиска, проанализировав полученные экспериментально данные выяснили: какой алгоритм работает быстрее, и за счет чего так происходит, а также разобрались в каких случаях тот или иной алгоритм работает лучше, то есть, когда данный алгоритм актуальнее использовать.