Министерство Образования Республики Беларусь

Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий

Кафедра Информационных систем и технологий

**СТРУКТУРЫ ДАННЫХ**

**Поиск записи по значению/индексу.**

**Лабораторная работа № 15, 16.**

Студент:

1курса 2 группы 1 подгруппы

Марушко Тимофей Фёдорович

Проверяющий:

Белодед Николай Иванович

**ОГЛАВЛЕНИЕ**

[**ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ 3**](#_Toc194480948)

[**1. ЛИНЕЙНЫЙ ПОИСК 3**](#_Toc194480949)

[**2. БИНАРНЫЙ ПОИСК 6**](#_Toc194480950)

[**3. ИНТЕРПОЛЯЦИОННЫЙ ПОИСК 10**](#_Toc194480951)

[**РЕШЕНИЕ ОСНОВНЫХ ЗАДАЧ 13**](#_Toc194480952)

[**РЕШЕНИЕ ДОПОЛНИТЕЛЬНЫХ ЗАДАЧ 16**](#_Toc194480953)

Теоретические сведения

1. Линейный поиск

**Линейный поиск** — это простой алгоритм поиска элемента в массиве или списке, который заключается в том, чтобы последовательно перебирать все элементы и сравнивать их с целевым элементом до тех пор, пока не будет найден элемент или не будут пройдены все элементы.



Основные преимущества линейного поиска:

1. Простота реализации: линейный поиск является одним из самых простых алгоритмов поиска и может быть легко реализован на любом языке программирования без использования дополнительных структур данных или библиотек.
2. Универсальность: линейный поиск может быть применен к любому типу данных и любой форме контейнера, включая неупорядоченные массивы, списки, файлы и т.д.
3. Надежность: линейный поиск всегда найдет целевой элемент, если он присутствует в контейнере.

Основные недостатки линейного поиска:

1. Низкая эффективность: в худшем случае, когда целевой элемент находится в конце контейнера или отсутствует в нем, линейный поиск должен пройти через все элементы, что может занять значительное время для больших контейнеров.
2. Неэффективность при частом поиске: если требуется выполнить множество поисковых операций в большом контейнере, линейный поиск будет неэффективен, поскольку каждый поиск потребует просмотра всех элементов.
3. Непригодность для упорядоченных контейнеров: если контейнер упорядочен, можно использовать более эффективные алгоритмы поиска, такие как двоичный поиск, которые могут быть значительно быстрее линейного поиска.

Обычно линейный поиск применяется для одиночного поиска в небольшом массиве, который не отсортирован. В других случаях, лучше и эффективней сначала отсортировать массив и применять другие алгоритмы поиска. Например, двоичный (бинарный) поиск либо другие.

….

int linearSearch(int arr[], int n, int x) {

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

if (arr[i] == x) {

return i;

}

}

return -1;

}

int main() {

setlocale(LC\_ALL, "rus");

int arr[] = { 45, 12, 78, 23, 56, 89 };

….

}



В этой программе определена функция linearSearch, которая принимает массив arr, его размер n и целевой элемент x. Функция последовательно перебирает элементы массива и сравнивает их с целевым элементом. Если элемент найден, функция возвращает его позицию в массиве; если элемент не найден, функция возвращает -1.

Предположим, вы ищете фамилию человека в телефонной книге. Она начинается с буквы «К». Конечно, можно начатся самого начала и перелистывать страницы, пока вы не доберетесь до буквы «К». Но скорее всего для ускорения поиска лучше раскрыть книгу на середине: ведь буква «К» должна находиться где-то ближе к середине телефонной книги. Теперь допустим, что вы вводите свои данные при входе на Facebook. При этом Facebook необходимо проверить, есть ли у вас учетная запись на сайте. Для этого ваше имя пользователя нужно найти в базе данных. Допустим, вы выбрали себе имя пользователя “karlrnageddon”. Facebook может начать с буквы А и проверять все подряд, но разумнее будет начать с середины.

Перед нами типичная задача поиска. И во всех этих случаях для решения задачи можно применить один алгоритм: бинарный поиск.

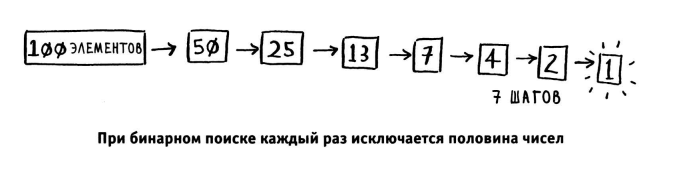
1. Бинарный поиск

**Бинарный поиск** — это алгоритм поиска элемента в упорядоченном массиве или списке, который заключается в последовательном делении массива на половины и проверке, находится ли целевой элемент в левой или правой половине. Алгоритм повторяется, пока не будет найден целевой элемент, или пока не будет проверено, что его в массиве нет.



Рассмотрим пример того, как работает бинарный поиск. Я загадал число от 1до100. Вы должны отгадать мое число, использовав как можно меньше попыток. Предположим, вы начинаете перебирать все варианты подряд: 1, 2, 3, 4… Это плохой способ угадать число. Это пример простого поиска. При каждой догадке исключается только одно число. Если я загадал число 99, то, чтобы добраться до него, потребуется 99 попыток!

Существует другой, более эффективный способ. Начнем с 50. Все эти числа слишком малы, но вы только что исключили половину чисел! Теперь вы знаете, что все числа 1-50 меньше загаданного. Следующая попытка: 75. На этот раз перелет ... Но вы снова исключили половину оставшихся чисел. С бинарным поиском вы каждый раз загадываете число в середине диапазона и исключаете половину оставшихся чисел. Следующим будет число 63 (по середине между 50 и 75). Так работает бинарный поиск.



Сколько времени экономит применение бинарного поиска? Если список состоит из 100 чисел, может потребоваться до 100 попыток, используя линейный поиск. Для списка из 4 миллиардов чисел потребуется до 4 миллиардов попыток. Таким образом, максимальное количество попыток совпадает с размером списка. С бинарным поиском дело обстоит иначе. Если список состоит из 100 элементов, потребуется не более 7 попыток. Для списка из 4 миллиардов элементов потребуется не более 32 попыток. Впечатляет, верно? Бинарный поиск выполняется за логарифмическое время.



Несмотря на то, что бинарный поиск является очень эффективным алгоритмом поиска элемента в упорядоченном массиве, у него **есть некоторые недостатки**:

1. Бинарный поиск требует, чтобы массив был упорядочен. Если массив не упорядочен, то перед началом поиска потребуется произвести сортировку, что может занять значительное время.
2. Бинарный поиск не работает с динамическими структурами данных, такими как связанные списки, потому что доступ к элементам в них не может быть осуществлен быстро.
3. Бинарный поиск занимает больше памяти, чем линейный поиск, потому что требуется дополнительная переменная для хранения среднего индекса.
4. Если массив содержит повторяющиеся элементы, то бинарный поиск может вернуть любой из них, а не обязательно первый или последний. Это может быть нежелательным в некоторых случаях.
5. В худшем случае, сложность бинарного поиска может достигать O(n), когда все элементы массива одинаковы.
6. Бинарный поиск неэффективен для поиска элемента в небольших массивах, так как в таком случае линейный поиск может быть быстрее.

**Основные шаги бинарного поиска**:

1. Определить границы поиска в массиве, обычно это начало и конец массива.
2. Вычислить средний индекс, как сумму границ, деленную на 2. Этот индекс будет использоваться для разделения массива на две части: левую и правую.
3. Сравнить целевой элемент со средним элементом в массиве. Если целевой элемент меньше, чем средний элемент, то дальнейший поиск будет осуществляться только в левой половине массива, иначе - в правой.
4. Если целевой элемент найден, то поиск завершается, иначе алгоритм повторяется для соответствующей половины массива.
5. Если границы поиска сомкнулись, то элемент не найден в массиве.

Пример:

// функция с алгоритмом двоичного поиска

int Search\_Binary(int arr[], int left, int right, int key)

{

int midd = 0;

while (1)

{

midd = (left + right) / 2;

if (key < arr[midd]) // если искомое меньше значе-ния в ячейке

right = midd - 1; // смещаем правую границу по-иска

else if (key > arr[midd]) // если искомое больше значе-ния в ячейке

left = midd + 1; // смещаем левую границу поиска

else // иначе (значения равны)

return midd; // функция возвращает индекс ячейки

if (left > right) // если границы сомкнулись

return -1;

}

}

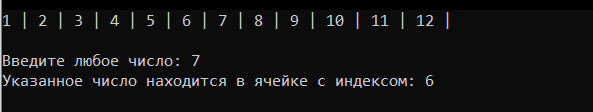
int main()

{

setlocale(LC\_ALL, "rus");

….

}



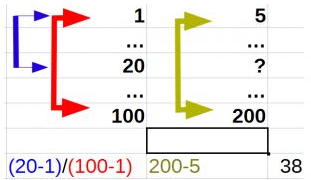
1. Интерполяционный поиск

**Интерполяционный поиск** — это алгоритм поиска элемента в упорядоченном массиве, который использует формулу интерполяции для оценки приблизительного местоположения целевого элемента в массиве и выбирает соответствующий интервал для дальнейшего поиска.

Простыми словами, интерполяционный поиск работает так:

1. Определить границы поиска в массиве, обычно это начало и конец массива.
2. Оценить приблизительное местоположение целевого элемента в массиве с помощью формулы интерполяции. Формула может быть разной в зависимости от конкретной реализации, но обычно она использует значения начального и конечного элементов массива, а также значение целевого элемента.
3. Сравнить целевой элемент со значением, найденным по формуле интерполяции. Если целевой элемент меньше, чем значение из формулы, то дальнейший поиск производится в левой части массива, иначе - в правой части массива.
4. Повторять шаги 2-3 до тех пор, пока целевой элемент не будет найден или пока не будет установлено, что он отсутствует в массиве.

Интерполяционный поиск может быть эффективным, еси элементы в массиве распределены равномерно и значения между ними линейно увеличиваются или уменьшаются. Однако, если распределение элементов не является равномерным, то интерполяционный поиск может работать медленнее, чем бинарный поиск.



Примерно так это будет выглядеть в виде картинки. Формула интерполяции достаточно проста – вычисляется длина между номерами первого элемента и искомого (задаваемого точнее). Такая же длина считается между первым и последним номерами. Длины между собой делятся, как раз и получая вычисление подобия. То же самое происходит со значениями элементов – так же вычисляется расстояние между граничными значениями в массиве.

Линейная интерполяция может быть выражена следующей формулой:

position = low + ((high - low) / (arr[high] - arr[low])) \* (x - arr[low]),

где: low - индекс начального элемента интервала поиска; high - индекс конечного элемента интервала поиска; arr - упорядоченный массив; x - значение целевого элемента, которое мы ищем в массиве; position - приблизительное местоположение целевого элемента в массиве.

Пример:

int main()

{

//Массив значений в котором пойдет поиск

int MyArray[]{ 1, 2, 4, 6, 7, 89, 123, 231, 1000, 1235 };

int x = 0; //Текущая позиция массива, с которым сравнивается искомое

int a = 0; //Левая граница области, где ведется поиск

int b = 9; //Правая граница области, где ведется поиск

int WhatFind = 123; //Значение, которое нужно найти

bool found; //Переменая-флаг, принимающая True если искомое найдено

/\*\*\*\*\*\*\*\*\*\*\*\* Начало интерполяции \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//Цикл поиска по массиву, пока искомое не найдено

//или пределы поиска еще существуют

for (found = false; (MyArray[a] < WhatFind) && (MyArray[b] > WhatFind) && !found; )

{

//Вычисление интерполяцией следующего элемента, который будет сравниваться с искомым

x = a + ((WhatFind - MyArray[a]) \* (b - a)) / (MyArray[b] - MyArray[a]);

//Получение новых границ области, если искомое не найдено

if (MyArray[x] < WhatFind)

a = x + 1;

else if (MyArray[x] > WhatFind)

b = x - 1;

else

found = true;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Конец интерполяции \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//Если искомое найдено на границах области поиска, показать на какой границе оно

if (MyArray[a] == WhatFind)

cout << WhatFind << " founded in element " << a << endl;

else if (MyArray[b] == WhatFind)

cout << WhatFind << " founded in element " << b << endl;

else

cout << "Sorry. Not found" << endl;

return 0;

}



Таким образом сам цикл просто вычисляет по формуле область массива, где может находиться искомое используя этот самый принцип интерполяции в С++, подбирая подобия так сказать. Если вычисленное не равно искомому, значит нужно сдвинуть границы области, где проходит вычисление. Если вычисленное больше – сдвигается правая граница области поиска, если меньше – левая. Так отрезая (как в бинарном поиске) кусок массива за куском, постепенно достигается нужная ячейка массива, ну или границы области поиска сужаются до таких величин, в пределах которого уже искать нечего, когда дистанция между границами равна 1 (т.е. между точкой А и В нет более элементов для вычисления) решение говорит о том, что значение в массиве не найдено.

Решение ОСНОВНЫХ ЗАДАЧ

**Условие:**

Для вашей БД реализовать 3 вида поиска: стандартный алгоритм поиска подстроки (в лр. рассматривался как первый алгоритм), поиск подстроки с помощью алгоритма Кнута-Морриса-Пратта, поиск подстроки с использованием string::find(). Поиск производить по одному типу данных, например, по цене.

**Решение:**

*#include* <iostream>

*#include* <vector>

*#include* <fstream>

*#include* <sstream>

*#include* <algorithm>

*using* *namespace* std;

*struct* *Phone* {

*int* id;

*string* model;

*double* price;

};

*struct* *Teacher* {

*int* id;

*string* name;

vector*<string>* subjects;

vector*<int>* phoneIds;

};

vector*<Teacher>* teachers;

vector*<Phone>* phones;

*void* loadDataFromFile(*const* *string&* teacherFile, *const* *string&* phoneFile, *const* *string&* foreigingkeysFile) {

*ifstream* teacherStream(teacherFile);

*ifstream* phoneStream(phoneFile);

*ifstream* foreigingkeysStream(foreigingkeysFile);

*string* line;

*while* (getline(phoneStream, line)) {

*stringstream* ss(line);

*int* id;

*string* model;

*double* price;

ss >> id >> model >> price;

phones.push\_back({ id, model, price });

}

*while* (getline(teacherStream, line)) {

*stringstream* ss(line);

*int* id;

*string* name;

*int* subjectCount;

ss >> id >> name >> subjectCount;

*Teacher* teacher;

teacher.id *=* id;

teacher.name = name;

*for* (*int* i *=* 0; i *<* subjectCount; *++*i) {

*string* subject;

ss >> subject;

teacher.subjects.push\_back(subject);

}

teachers.push\_back(teacher);

}

*while* (getline(foreigingkeysStream, line)) {

*stringstream* ss(line);

*int* teacherId, phoneId;

ss >> teacherId >> phoneId;

*for* (*auto&* teacher : teachers) {

*if* (teacher.id *==* teacherId) {

teacher.phoneIds.push\_back(phoneId);

*break*;

}

}

}

cout << "Data loaded from files successfully!" << endl;

}

*void* findPhoneByPriceLinear(*double* price) {

cout << "Searching phone with price: " << price << endl;

*for* (*auto&* p : phones) {

*if* (p.price *==* price) {

cout << "Found: Model: " << p.model << " | Price: " << p.price << "$" << endl;

*return*;

}

}

cout << "Phone with the given price not found." << endl;

}

*void* findPhoneByPriceBinary(*double* price) {

vector*<Phone>* sortedPhones *=* phones;

sort(sortedPhones.begin(), sortedPhones.end(), [](*const* *Phone&* a, *const* *Phone&* b) {

*return* a.price *<* b.price;

});

cout << "Searching phone with price: " << price << " (Binary Search)" << endl;

*int* low *=* 0, high *=* sortedPhones.size() *-* 1;

*while* (low *<=* high) {

*int* mid *=* low *+* (high *-* low) */* 2;

*if* (sortedPhones[mid].price *==* price) {

cout << "Found: Model: " << sortedPhones[mid].model << " | Price: " << sortedPhones[mid].price << "$" << endl;

*return*;

}

*if* (sortedPhones[mid].price *<* price) {

low *=* mid *+* 1;

}

*else* {

high *=* mid *-* 1;

}

}

cout << "Phone with the given price not found." << endl;

}

*void* findPhoneByPriceInterpolation(*double* price) {

vector*<Phone>* sortedPhones *=* phones;

sort(sortedPhones.begin(), sortedPhones.end(), [](*const* *Phone&* a, *const* *Phone&* b) {

*return* a.price *<* b.price;

});

cout << "Searching phone with price: " << price << " (Interpolation Search)" << endl;

*int* low *=* 0, high *=* sortedPhones.size() *-* 1;

*while* (low *<=* high *&&* sortedPhones[low].price *<=* price *&&* sortedPhones[high].price *>=* price) {

*int* pos *=* low *+* ((price *-* sortedPhones[low].price) *\** (high *-* low)) */* (sortedPhones[high].price *-* sortedPhones[low].price);

*if* (pos *<* low *||* pos *>* high) {

*break*;

}

*if* (sortedPhones[pos].price *==* price) {

cout << "Found: Model: " << sortedPhones[pos].model << " | Price: " << sortedPhones[pos].price << "$" << endl;

*return*;

}

*if* (sortedPhones[pos].price *<* price) {

low *=* pos *+* 1;

}

*else* {

high *=* pos *-* 1;

}

}

cout << "Phone with the given price not found." << endl;

}

*void* printInfo() {

*for* (*const* *auto&* teacher : teachers) {

cout << "Teacher ID: " << teacher.id << " | Name: " << teacher.name << " | Subjects: ";

*for* (*const* *auto&* subject : teacher.subjects) {

cout << subject << " ";

}

cout << "| Phones: ";

*for* (*int* phoneId : teacher.phoneIds) {

*for* (*const* *auto&* phone : phones) {

*if* (phone.id *==* phoneId) {

cout << phone.model << " (Price: " << phone.price << "$) ";

*break*;

}

}

}

cout << endl;

}

}

*void* printAllPhones() {

*for* (*const* *auto&* phone : phones) {

cout << "Phone ID: " << phone.id << " | Model: " << phone.model << " | Price: " << phone.price << "$" << endl;

}

}

*void* findTeacherByName(*const* *string&* name) {

*bool* found *=* *false*;

*for* (*const* *auto&* teacher : teachers) {

*if* (teacher.name == name) {

cout << "Found Teacher: ID: " << teacher.id << " | Name: " << teacher.name << " | Subjects: ";

*for* (*const* *auto&* subject : teacher.subjects) {

cout << subject << " ";

}

cout << endl;

found *=* *true*;

*break*;

}

}

*if* (*!*found) {

cout << "Teacher with the given name not found." << endl;

}

}

*int* main() {

*int* choice;

loadDataFromFile("teachers.txt", "phones.txt", "foreigingkeys.txt");

*while* (*true*) {

cout << "\nUniversity System\n"

<< "1. Search Phone by Price (Linear Search)\n"

<< "2. Search Phone by Price (Binary Search)\n"

<< "3. Search Phone by Price (Interpolation Search)\n"

<< "4. Print printInfo\n"

<< "5. Search Teacher by Name\n"

<< "9. Exit\n"

<< "Choose option: ";

cin >> choice;

cin.ignore();

*double* price;

*string* name;

*switch* (choice) {

*case* 1: {

cout << "Enter price: ";

cin >> price;

findPhoneByPriceLinear(price);

*break*;

}

*case* 2: {

cout << "Enter price: ";

cin >> price;

findPhoneByPriceBinary(price);

*break*;

}

*case* 3: {

cout << "Enter price: ";

cin >> price;

findPhoneByPriceInterpolation(price);

*break*;

}

*case* 4:

printInfo();

*break*;

*case* 5:

cout << "Enter teacher name: ";

getline(cin, name);

findTeacherByName(name);

*break*;

*case* 9:

*return* 0;

*default*:

cout << "Invalid choice!" << endl;

}

}

}