Министерство образования Республики Беларусь/

Учреждение образования «Белорусский государственный технологический университет»

ФИТ

Кафедра Информационных систем и технологий 1курс

Группа 2-2

**Структуры данных**

Тема:

Методы поиска. Set

Лабораторная работа 15,16

**Выполнил:**

Волковец Сергей Николаевич

**Проверил:**

Белодед Николай Иванович

Оглавление

[Теоретические сведения 3](#_Toc194475335)

[Код 7](#_Toc194475336)

## Теоретические сведения

**Линейный поиск** - это алгоритм, который мы уже неоднократно реализовывали. Он работает путем последовательного перебора всех элементов списка до тех пор, пока не будет найден искомый элемент или не будут перебраны все элементы списка.

Плюсы:

* Простота реализации.
* Работает быстро на небольших объемах данных.

Минусы:

* Неэффективен на больших объемах данных.
* Время выполнения может быть линейным в худшем случае.
* Необходимость перебирать все элементы списка в худшем случае.

**Двоичный поиск** - это алгоритм поиска элемента в отсортированном массиве. Он работает путем сравнения искомого элемента с элементом в середине массива. Если элемент в середине массива меньше искомого элемента, то поиск продолжается в верхней половине массива. Если элемент в середине массива больше искомого элемента, то поиск продолжается в нижней половине массива. Этот процесс повторяется до тех пор, пока искомый элемент не будет найден или пока не останется только один элемент в массиве.

Двоичный поиск имеет следующие преимущества:

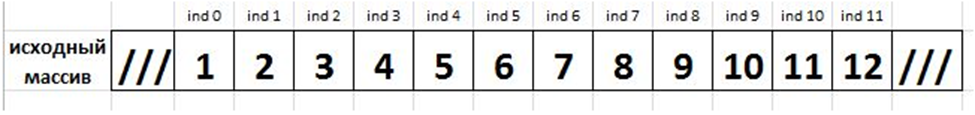
* Он работает быстрее, чем линейный поиск.
* Он может быть использован для поиска элементов в больших отсортированных массивах.

Недостатки двоичного поиска:

* Массив должен быть отсортирован перед использованием двоичного поиска.
* Двоичный поиск не может быть использован для поиска элементов в неотсортированных массивах.
* Двоичный поиск может быть менее эффективным, если массив содержит много повторяющихся элементов.

Рассмотрим алгоритм двоичного поиска подробнее на примере:

Предположим, что массив из 12-ти элементов отсортирован по возрастанию



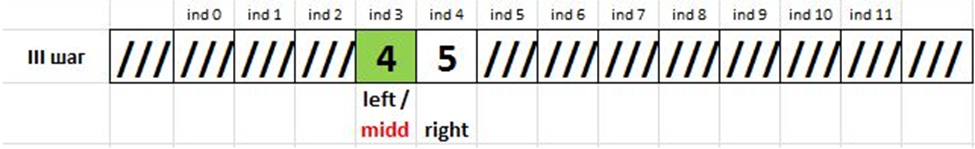
Пользователь задает искомое значение (ключ поиска). Допустим 4. На первой итерации массив делится на две части (ищем средний элемент – midd): (0 + 11) / 2 = 5 (0.5 отбрасываются). Сначала, проверяется значение среднего элемента массива. Если оно совпадает с ключом – алгоритм прекратит работу и программа выведет сообщение, что значение найдено. В нашем случае, ключ не совпадает со значением среднего элемента.



Если ключ меньше значения среднего элемента, алгоритм не будет проводить поиск в той половине массива, которая содержит значения больше ключа (т.е. от среднего элемента до конца массива). Правая граница поиска сместится (midd – 1). Далее снова деление массива на 2.



Ключ снова не равен среднему элементу. Он больше него. Теперь левая граница поиска сместится (midd + 1).



На третьей итерации средний элемент – это ячейка с индексом 3: (3 + 4) / 2 = 3. Он равен ключу. Алгоритм завершает работу.

Пример реализации двоичного поиска:

|  |
| --- |
| #include <iostream>  using namespace std;  int binarySearch(int arr[], int n, int x) {  int left = 0;  int right = n - 1; // левый и правый индексы подмассива, в котором мы ищем элемент x  while (left <= right) {       int mid = (left + right) / 2; // вычисляем середину подмассива       if (arr[mid] == x) { //  проверяем, равен ли элемент arr[mid] элементу x           return mid;       }       else if (arr[mid] < x) { //Если arr[mid] меньше, чем x, мы знаем, что x находится в правой половине подмассива           left = mid + 1;       }       else { // Если arr[mid] больше, чем x, мы знаем, что x находится в левой половине подмассива           right = mid - 1;          }      }  return -1;  }  int main() {      setlocale(LC\_CTYPE, "rus");  int arr[] = { 1, 3, 5, 7, 9, 11, 13, 15 };  int n = sizeof(arr) / sizeof(arr[0]); // размер массива n  int x = 7; // элемент x, который мы хотим найти  int index = binarySearch(arr, n, x);  if (index == -1) {       cout << "Элемент не найден." << endl;  }  else {       cout << "Элемент найден под индексом: " << index << endl;  }  // Вывод:  // Элемент найден под индексом: 3  return 0;  } |

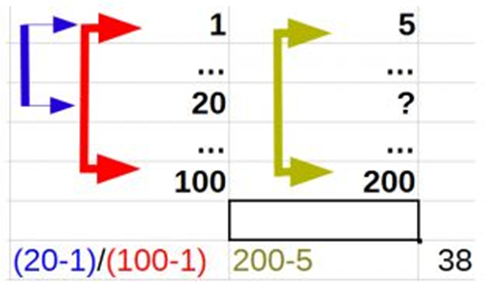
**Интерполирующий поиск** - это алгоритм поиска элемента в отсортированном массиве, который использует интерполяцию для нахождения близкого к искомому элемента индекса.

Он основан на принципе поиска в телефонной книге или словаре. Вместо сравнения каждого элемента с искомым, как при линейном поиске, данный алгоритм использует формулу для вычисления примерного местоположения искомого элемента в массиве. Затем он выполняет бинарный поиск вокруг этого местоположения для точного нахождения искомого элемента.

Формула расчета примерной позиции элемента в интерполяционном поиске основана на идее линейной интерполяции. Идея заключается в том, что если элементы массива равномерно распределены, то мы можем использовать линейную интерполяцию для приблизительного вычисления позиции искомого элемента в отсортированном массиве.

Интерполяция – это определение области поиска, путем вычисления подобия расстояний между искомым значением и всей областью.

Принцип работы этой формулы напоминает подобные треугольники в геометрии.



  Формула достаточно проста:

1. Вычисляется длина между номерами первого элемента и искомого.

2. Такая же длина считается между первым и последним номерами.

3. Длины между собой делятся, как раз и получая вычисление подобия.

4. То же самое происходит со значениями элементов – так же вычисляется расстояние между граничными значениями в массиве.

5. Полученная длина номеров элементов массива умножается на длину значений в этих (граничащих) элементах.

6. Прибавляется значение в первой ячейке массива.

Получается: 1 + (20-1)/(100-1) \* (200-5) ~ 38

Запишем эту формулу еще раз, но теперь в общем виде:

pos = low + (value - arr[low]) / (arr[high] - arr[low]) \* (high - low)

где pos - позиция, на которой, возможно, находится искомый элемент, low и high - индексы первого и последнего элементов в диапазоне, value - значение искомого элемента, arr - массив данных.

Пример реализации интерполирующего поиска:

|  |
| --- |
| #include <iostream>  using namespace std;  int interpolationSearch(int arr[], int n, int x) {  int left = 0;  int right = n - 1;// левый и правый индексы подмассива, в котором мы ищем элемент x  while (left <= right && x >= arr[left] && x <= arr[right]) {          int pos = left + ((x - arr[left]) \* (right - left)) / (arr[right] - arr[left]); // вычисляем позицию элемента          if (arr[pos] == x) { //  проверяем, равен ли элемент arr[mid] элементу x           return pos;       }       else if (arr[pos] < x) { //Если arr[mid] меньше, чем x, мы знаем, что x находится в правой половине подмассива           left = pos + 1;       }       else { // Если arr[mid] больше, чем x, мы знаем, что x находится в левой половине подмассива           right = pos - 1;          } }  return -1;  }  int main() {      setlocale(LC\_CTYPE, "rus");  int arr[] = { 1, 3, 5, 7, 9, 11, 13, 15 };  int n = sizeof(arr) / sizeof(arr[0]);// размер массива n  int x = 7; // элемент x, который мы хотим найти  int index = interpolationSearch(arr, n, x);  if (index == -1) {       cout << "Элемент не найден." << endl;  }  else {       cout << "Элемент найден под индексом: " << index << endl;  }  // Вывод:  // Элемент найден под индексом: 3  return 0;  } |

## Код

**Задание 1.**

Для своей БД реализовать 2 поиска по любому полю: бинарный и интерполяционный.

Код:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int binarySearch(const vector<int>& data, int target) {

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

while (low <= high) {

int mid = (low + high) / 2;

if (data[mid] == target) {

return mid;

}

else if (data[mid] < target) {

low = mid + 1;

}

else {

high = mid - 1;

}

}

return -1;

}

int interpolationSearch(const vector<int>& data, int target) {

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

while (low <= high && target >= data[low] && target <= data[high]) {

if (low == high) {

if (data[low] == target) return low;

return -1;

}

int pos = low + ((target - data[low]) \* (high - low) / (data[high] - data[low]));

if (data[pos] == target) {

return pos;

}

else if (data[pos] < target) {

low = pos + 1;

}

else {

high = pos - 1;

}

}

return -1;

}

int main() {

vector<int> database = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };

sort(database.begin(), database.end());

int target;

cout << "Введите число для поиска: ";

cin >> target;

int binaryResult = binarySearch(database, target);

if (binaryResult != -1) {

cout << "Найдено с помощью бинарного поиска на индексе: " << binaryResult << endl;

}

else {

cout << "Число не найдено с помощью бинарного поиска." << endl;

}

int interpolationResult = interpolationSearch(database, target);

if (interpolationResult != -1) {

cout << "Найдено с помощью интерполяционного поиска на индексе: " << interpolationResult << endl;

}

else {

cout << "Число не найдено с помощью интерполяционного поиска." << endl;

}

return 0;

}

**Задание 2.**

Написать программу, в которой нужно создать множество, добавить в него элементы и вывести их на экран в отсортированном порядке.

Код:

#include <iostream>

#include <set>

using namespace std;

int main() {

setlocale(LC\_ALL, "ru");

set<int> mySet;

mySet.insert(5);

mySet.insert(3);

mySet.insert(8);

mySet.insert(1);

cout << "Элементы множества в отсортированном порядке: ";

for (int num : mySet) {

cout << num << " ";

}

return 0;

}

Задание 3.

Написать программу, в которой нужно найти элемент с минимальным значением в множестве.

Код:

#include <iostream>

#include <set>

using namespace std;

int main() {

setlocale(LC\_ALL, "ru");

set<int> mySet;

mySet.insert(5);

mySet.insert(3);

mySet.insert(8);

mySet.insert(1);

cout << "Элемент с минимальным значением множества: " << \*mySet.begin();

return 0;

}

**Задание 4.**

Написать программу, в которой нужно создать два множества, объединить их и вывести на экран общие элементы.

Код:

#include <iostream>

#include <set>

using namespace std;

int main() {

using namespace std;

set<int> set1;

set<int> set2;

set1.insert(1);

set1.insert(2);

set1.insert(3);

set2.insert(3);

set2.insert(4);

set2.insert(5);

cout << "Общие элементы: ";

for (int num : set1) {

if (set2.count(num) > 0) {

cout << num << " ";

}

}

return 0;

}