**АДТ**

**Структура данных**

**Целью** предмета является изучение основных структур данных, описание этих структур и операций обработки данных.

**Структура данных** – это основной компонент программы, т.к. каждая из программ выполняет обработку некоторой дискретной информации.

- допустимые операции над различными структурами необходимы для таких разделов: операционные системы, компиляторы, искусственный интеллект, база данных и граф. приложения.

**Абстрактный тип данных (АТД/АДТ)**

Для описания **логической** структуры данных удобно использовать **АДТ**.

**АДТ** – это совокупность данных и операций над ними.

*Формат АДТ:*

1. Название/имя типа
2. Описание типа данных
3. Список операций над данными

При описание каждой операции выполняется 5 действий:

* -входные значения
* -предусловие
* -процесс
* -возвращаемое значение
* -постусловие

**Пример АДТ:**

*АДТ Circle*

*Данные: radius –real*

*Операции:*

*Конструктор:*

*нач. значение: r – real*

*предусловие: if r<0…. Выход*

*процесс: radius=r;*

*Операция Aria:*

*вход: нет*

*предусловие: нет*

*процесс: PI\*radius\*radius*

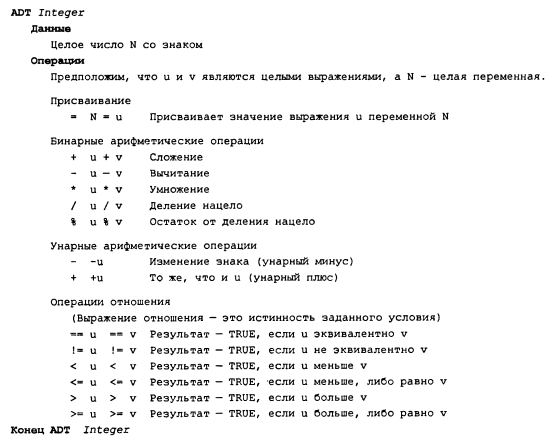
*выход: вернуть площадь*

*постусловие: нет*

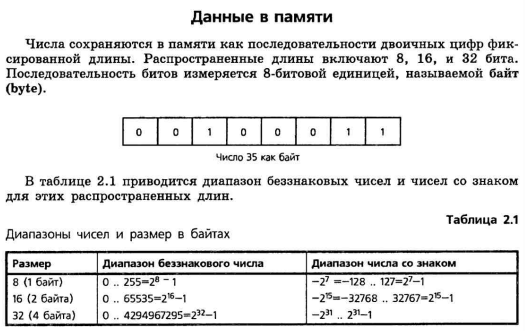
*end АДТ.*

**АДТ базовых типов данных**

- целочисленные типы



Компьютерная система сохраняет целые числа в блоках памяти с фиксированной длиной. В компьютере целые хранятся как двоичные (бинарные) числа, состоящие из различных последовательностей цифр 0 и 1.

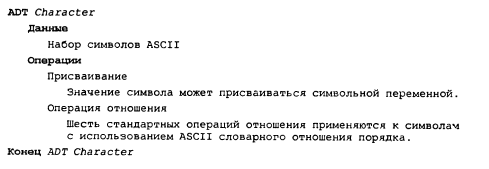


**Компьютерная память** – это последовательность байтов, к которой обращаются с помощью адресов 0, 1, 2, 3 и так далее.

**Адрес** целого в памяти это местоположение первого байта последовательности.

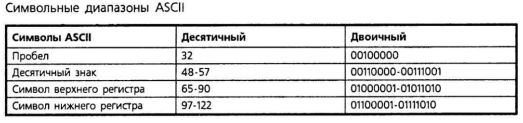
Целыми типами являются: ***int***, ***short int*** и ***long int***.

- символьные типы



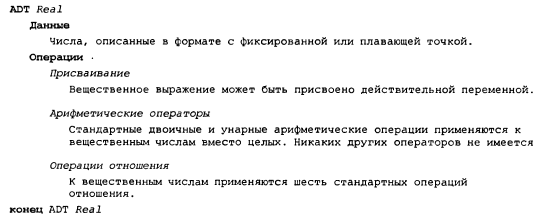
Большинство компьютерных систем используют ASCII для символьного представления. Символы ASCII хранятся как 7-битный целый код в 8-битовом числе (2^7 =128 различных кодов подразделяются на 95 печатаемых и 33 управляющих символа).





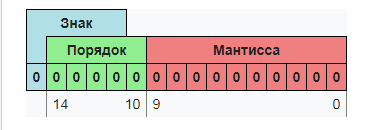
Для хранения символа используется тип **char.**

- вещественные типы



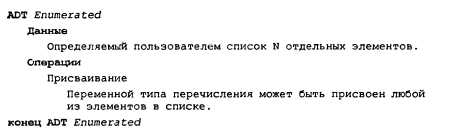
Вещественные числа обычно представляются в виде **чисел с плавающей запятой.** Числа с плавающей запятой — один из возможных способов представления действительных чисел, который является компромиссом между точностью и диапазоном принимаемых значений, его можно считать аналогом экспоненциальной записи чисел, но только в памяти компьютера.

Число с плавающей запятой состоит из набора отдельных двоичных разрядов, условно разделенных на так называемые **знак**, **порядок** и **мантиссу** . В наиболее распространённом формате (стандарт IEEE 754) число с плавающей запятой представляется в виде набора битов, часть из которых кодирует собой мантиссу числа, другая часть — показатель степени, и ещё один бит используется для указания знака числа (00 — если число положительное, 11 — если число отрицательное).

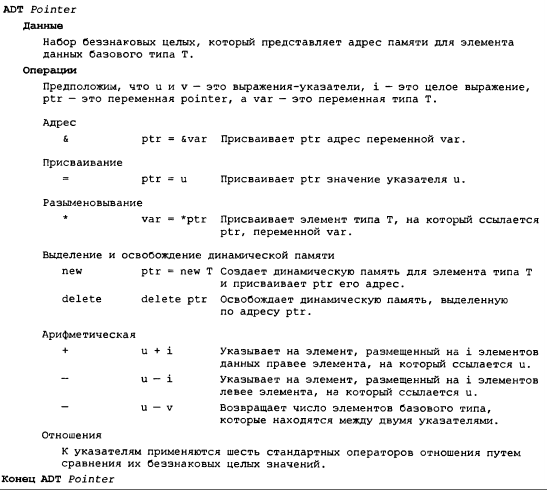


С++ придерживает три вещественных типа данных: **float**, **double** и **long double**.

- типы перечисления

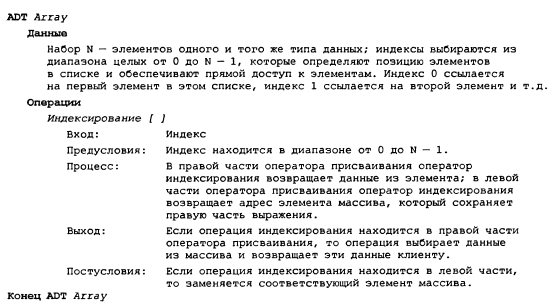


- указатели

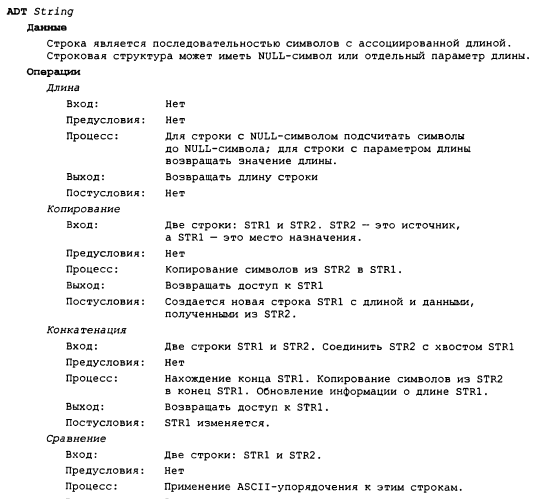


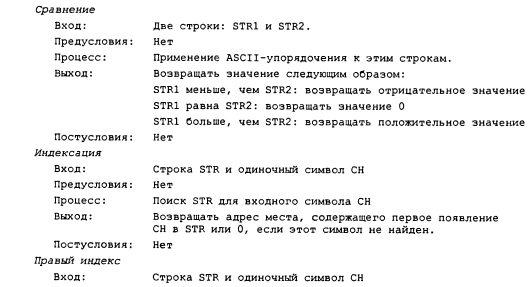
Значение указателя – адрес памяти, использующий 16, 32 или более битов в зависимости от машинной архитектуры.

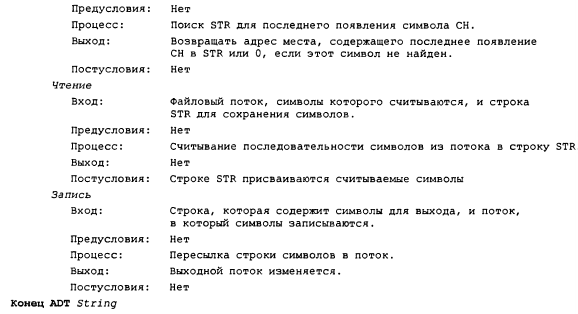
- массивы



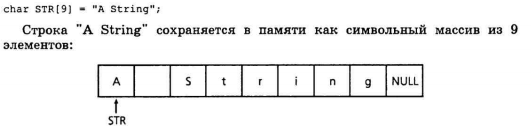
- строковые константы и переменные



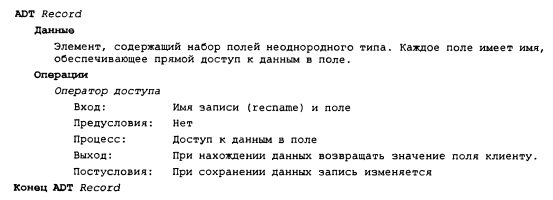




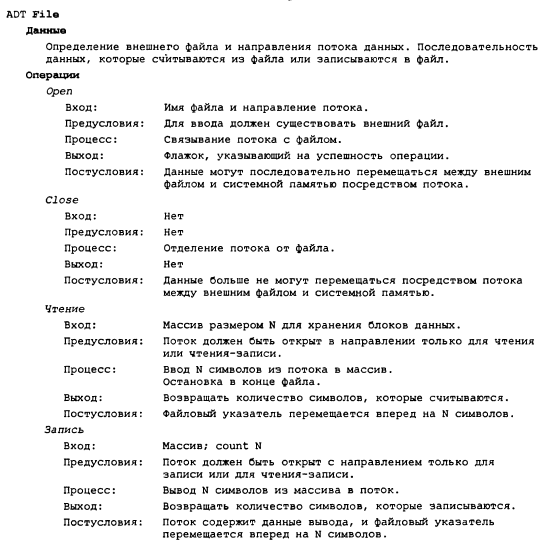
**Строка в С++** -это строка с нулевым завершающим символом, в которой NULL-символ обозначается символом 0 в коде ASCII. Компилятор определяет строковую константу как последовательность символов, заключенную в кавычки. **Строковая переменная** – это символьный массив, который содержит последовательность символов с NULL-символом в конце.

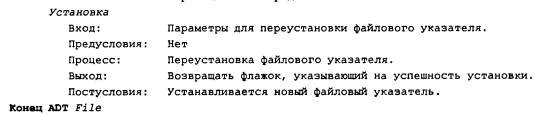


- записи



- файлы





**Алгоритмы**

<https://www.intuit.ru/studies/courses/16740/1301/lecture/25624>

<http://ani-studio.narod.ru/BOX/Flash/Study/Automation/HTML-Themes/Theme11.htm>

<https://www.yaklass.ru/materiali?mode=cht&chtid=474>

<https://studfile.net/preview/3276183/>

***Алгоритм –***

- это последовательность чётко определенных действий, выполнение которых ведёт к решению задачи;

- это совокупность действий, приводящих к достижению результата за конечное число шагов.

**Свойства алгоритмов:**

* *Дискретность* – это разбиение алгоритма на ряд отдельных законченных действий (шагов).
* *Детерминированность/определенность* - любое действие алгоритма должно быть строго и недвусмысленно определено в каждом случае.
* *Конечность* – каждое действие в отдельности и алгоритм в целом должны иметь возможность завершения.
* *Массовость* – один и тот же алгоритм можно использовать с разными исходными данными.
* *Результативность* – алгоритм должен приводить к достоверному решению.

**Основная цель алгоритмизации** – составление алгоритмов для ЭВМ с дальнейшим решением задачи на ЭВМ.

Существует несколько способов записи алгоритмов:

1. словесная (запись на естественном языке);
2. псевдокоды (полуформализованные описания алгоритмов на условном алгоритмическом языке, включающие в себя как элементы языка программирования, так и фразы естественного языка, общепринятые математические обозначения и др.);
3. графическая (изображения из графических символов – блок-схема);
4. программная (тексты на языках программирования – код программы).

**Различают три основных вида алгоритмов:**

* линейный алгоритм - это алгоритм, в котором действия выполняются однократно и строго последовательно.
* разветвляющийся алгоритм -  это алгоритм, в котором в зависимости от условия выполняется либо одна, либо другая последовательность действий.
* циклический алгоритм -  это алгоритм, команды которого повторяются некое количество раз подряд.

**Формальные модели алгоритмов**

В теоретических подходах к построению строгого определения понятия алгоритма исторически выделились три основные направления**:**

**Первое направление** связано с рассмотрением алгоритмов, позволяющих вычислить значение числовых функций, зависящих от целочисленных значений аргументов – такие функции получили название *вычислимых*.

**Второе направление** связано с машинной математикой. Основная идея этого направления состоит в том, что алгоритмические процессы – это процессы, которые может совершать соответствующим образом устроенная *«машина» (*данный подход развивался в работах Э.Поста и А.Тьюринга).

**Третье направление** связано с понятием нормальных алгоритмов, введенным и разработанным российским математиком А.А.Марковым в начале 50-х гг. XX в.

***Вычислительная сложность*** — понятие в  [теории алгоритмов](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%BE%D0%B2), обозначающее функцию зависимости объёма работы, которая выполняется некоторым алгоритмом, от размера входных данных.

Одним из способов оценки трудоемкости (Tn) является*подсчет количества выполняемых операций*. Рассмотрим в качестве примера алгоритм поиска минимального элемента массива.

1. начало; поиск минимального элемента массива array из N элементов
2. min := array[1]
3. для i от 2 до N выполнять:
4. если array[i] < min
5. min := array[i]
6. конец; вернуть min

При выполнении этого алгоритма будет выполнена:

1. N — 1 операция присваивания счетчику цикла i нового значения;
2. N — 1 операция сравнения счетчика со значением N;
3. N — 1 операция сравнения элемента массива со значением min;
4. от 1 до N операций присваивания значения переменной min.

Точное количество операций будет зависеть от обрабатываемых данных, поэтому имеет смысл говорить о наилучшем, наихудшем и среднем случаях. При этом худшему случаю всегда уделяется особое внимание, в том числе потому, что «плохие» данные могут быть намеренно поданы на вход злоумышленником.

Понятие среднего случая используется для оценки поведения алгоритма с расчетом на то, что наборы данных равновероятны. Однако, такая оценка достаточно сложна

**Асимптотические обозначения**

Подсчет количества операций позволяет сравнить эффективность алгоритмов. Однако, аналогичный результат можно получить более простым путем. Анализ проводят с расчетом на достаточно большой объем обрабатываемых данных (n→∞), поэтому ключевое значение имеет скорость роста функции сложности, а не точное количество операций.

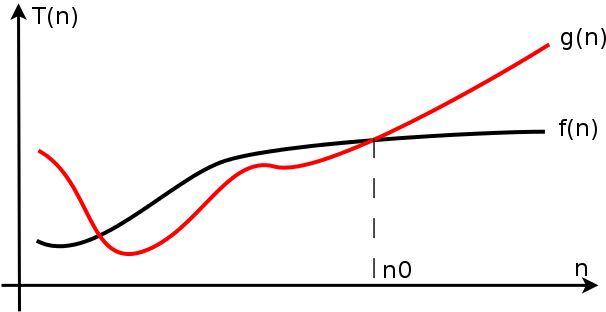
При анализе скорости роста игнорируются постоянные члены и множители в выражении, т.е. функции fx=10⋅x2+20 и gx=x2 эквивалентны с точки зрения скорости роста. Незначащие члены лишь добавляют «волнистости», которая затрудняет анализ.

В оценке алгоритмов используются специальные асимптотические обозначения, задающие следующие классы функций:

* O(g) — функции, растущие медленнее чем g;
* Ω(g) — функции, растущие быстрее чем g;
* Θ(g) — функции, растущие с той же скоростью, что и g.

Запись fn=O(gn) означает принадлежность функции f классу O(g), т.е. функция f ограничена сверху функцией g для достаточно больших значений аргумента. ∃n0>0,c>0:∀n>n0,fn≤c⋅gn.

Ограниченность функции g снизу функцией f записывается следующим образом: gn=Ω(fn). Нотации Ω и O взаимозаменяемы: fn=O(gn)⇔gn=Ω(fn).



Асимптотические обозначения «О большое» и «Омега большое»

**Рекурсия**

**Рекурсия** — это такая организация выполнения работы функции, при которой данная функция вызывает сама себя.

Рекурсию можно назвать простой, если в функции присутствует лишь один рекурсивный вызов. Такую рекурсию можно назвать еще рекурсией первого порядка. Но рекурсивный вызов может появляться в функции более, чем один раз. В таких случаях можно выделить следующие виды рекурсии:

1. **параллельная рекурсия** - тело определения функции function\_1 содержит вызов некоторой функции function\_2, несколько аргументов которой являются рекурсивными вызовами функции function\_1.  
   (defun function\_1 … (function\_2 … (function\_1 …) … (function\_1 …) … ) … )
2. **взаимная рекурсия** - в теле определения функции function\_1 вызывается некоторая функции function\_2, которая, в свою очередь, содержит вызов функции function\_1.  
   (defun function\_1 … (function\_2 … ) … ) (defun function\_2 … (function\_1 … ) … )
3. **рекурсия более высокого порядка** - в теле определения функции аргументом рекурсивного вызова является рекурсивный вызов.  
   (defun function\_1 … (function\_1 … (function\_1 …) … ) … )

**Пример**: написать функцию умножения двух чисел, используя только операцию сложения

int mul(int n, int m)

{

if (m == 0)

return 0;

return (n + mul(n, m - 1));

}

**Жадные алгоритмы**

В жадном алгоритме(greedy algorithm) всегда делается выбор, который кажется самым лучшим в данный момент — т.е. производится локально оптимальный выбор в надежде, что он приведет к оптимальному решению глобальной задачи. Решение принимаемое на каждом шаге должно быть оптимальным только на текущем шаге и должно приниматься без учета предыдущих или последующих решений.

Алгоритмы, предназначенные для решения задач оптимизации, обычно представляют собой последовательность шагов, на каждом из которых предоставляется некоторое множество выборов. Определение наилучшего выбора, руководствуясь принципами динамического программирования, во многих задачах оптимизации напоминает стрельбу из пушки по воробьям; другими словами, для этих задач лучше подходят более простые и эффективные алгоритмы.

Признаки того, что задачу возможно решить при помощи жадного алгоритма:

1.Задачу можно разбить на подзадачи;

2. Величины, рассматриваемые в задаче, можно дробить так же на подзадачи;

3.Сумма оптимальных решений для двух подзадач даст оптимальное решения для всей задачи.

Свойства жадного выбора

Свойства жадного выбора говорят о том, что глобальное оптимальное решение можно получить, делая локальный оптимальный выбор, т.е мы выбираем тот выбор, который будет являться лучшим на данный момент.

После каждого жадного выбора задача сводится к простой.

**Оптимальная подструктура**

Оптимальная подструктура проявляется в задаче, если в её оптимальном решение содержится оптимальное решение подзадачи (оно применимо как в динамическом программирование, так и в жадных алгоритмах).

Динамическое программирование в теории вычислительных систем – это способ решения сложных задач путём разбиения их на сложные подзадачи.

Динамическое программирование – это метод решения задач путём составления последовательности из подзадач. Таким образом, что:

1. первый элемент последовательности имеет тривиальное решение
2. последний элемент этой последовательности – это исходная задача
3. каждая подзадача этой последовательности может быть решена

**Алгоритмы поиска. Последовательный поиск. Поиск с сужением зоны. Распределяющий поиск**

Поиск — обработка некоторого множества данных с целью выявления подмножества данных, соответствующего критериям поиска.

Ключ поиска – это поле записи, по значению которого происходит поиск. Ключи используются для отличия одних записей от других. Целью поиска является нахождение всех записей (если они есть) с данным значением ключа.

Под поиском в массиве будем понимать задачу нахождения индекса, по которому в массиве располагается некоторый заданный элемент. Тривиальный алгоритм поиска заключается в последовательном переборе элементов массива до тех пор, пока не будет обнаружен искомый или не будут просмотрены все элементы массива. В случае, когда нет никакой дополнительной информации о характере расположения элементов в массиве, такой алгоритм представляется единственно возможным. Нетрудно подсчитать, что, в худшем случае, для поиска в массиве, состоящем из N элементов, потребуется N операций сравнения.

Все алгоритмы поиска делятся на:

* поиск в неупорядоченном множестве данных;
* поиск в упорядоченном множестве данных.

Таким образом, определим общий алгоритм поиска данных:

**Шаг 1.** Вычисление элемента, что часто предполагает получение значения элемента, ключа элемента и т.д.

**Шаг 2.** Сравнение элемента с эталоном или сравнение двух элементов (в зависимости от постановки задачи).

**Шаг 3.** Перебор элементов множества, то есть прохождение по элементам массива.

**Последовательный (линейный) поиск**

**Последовательный (линейный) поиск** – это простейший вид поиска заданного элемента на некотором множестве, осуществляемый путем последовательного сравнения очередного рассматриваемого значения с искомым до тех пор, пока эти значения не совпадут.

Идея этого метода заключается в следующем. Множество элементов просматривается последовательно в некотором порядке, гарантирующем, что будут просмотрены все элементы *множества* (например, слева направо). Если в ходе просмотра *множества* будет найден искомый элемент, просмотр прекращается с положительным результатом; если же будет просмотрено все множество, а элемент не будет найден, *алгоритм* должен выдать отрицательный результат.

*Алгоритм последовательного поиска*

**Шаг 1.** Полагаем, что *значение* переменной *цикла* i=0.

**Шаг2.**Если *значение* элемента массива x[i] равно значению ключа key, то возвращаем *значение*, равное номеру искомого элемента, и *алгоритм* завершает работу. В противном случае *значение* переменной *цикла* увеличивается на единицу i=i+1.

**Шаг 3.** Если i<k, где k – число элементов массива x, то выполняется Шаг 2, в противном случае – работа алгоритма завершена и возвращается *значение* равное -1.

При наличии в массиве нескольких элементов со значением key данный *алгоритм* находит только первый из них (с наименьшим индексом).

Временная сложность последовательного поиска пропорциональна **O(n)**. Никаких ограничений на порядок элементов в массиве данный алгоритм не накладывает.

Недостатком рассматриваемого алгоритма поиска является то, что в худшем случае осуществляется просмотр всего массива. Поэтому данный алгоритм используется, если множество содержит небольшое количество элементов.

Достоинства последовательного поиска заключаются в том, что он прост в реализации, не требует сортировки значений множества, дополнительной памяти и дополнительного анализа функций. Следовательно, может работать в потоковом режиме при непосредственном получении данных из любого источника.

//последовательный поиск

template <typename T>

int SearchLinear(T arr[], int n, T key)

{

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

{

if (arr[i] == key)

return i;

}

return -1;

}

**Бинарный (двоичный) поиск**

**Бинарный (двоичный, дихотомический) поиск** – это *поиск* заданного элемента на упорядоченном множестве, осуществляемый путем неоднократного деления этого *множества* на две части таким образом, что искомый элемент попадает в одну из этих частей. *Поиск* заканчивается при совпадении искомого элемента с элементом, который является границей между частями *множества* или при отсутствии искомого элемента.

*Бинарный поиск* применяется к отсортированным множествам и заключается в последовательном *разбиении множества* пополам и поиска элемента только в одной половине на каждой итерации.

Таким образом, идея этого метода заключается в следующем. *Поиск* нужного значения среди элементов упорядоченного массива (по возрастанию или по убыванию) начинается с определения значения центрального элемента этого массива. *Значение* данного элемента сравнивается с искомым значением и в зависимости от результатов сравнения предпринимаются определенные действия. Если искомое и центральное значения оказываются равны, то *поиск* завершается успешно. Если искомое *значение* меньше центрального или больше, то формируется *массив*, состоящий из элементов, находящихся слева или справа от центрального соответственно. Затем *поиск* повторяется в новом массиве .

*Алгоритм бинарного поиска*

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

**Шаг2.**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 (в зависимости от характера упорядоченности).

В процессе работы алгоритма *бинарного поиска* размер фрагмента, где этот *поиск* должен продолжаться, каждый раз уменьшается примерно в два раза. Это обеспечивает сложность алгоритма пропорциональную **O(log n)**, где n – количество элементов *множества*.

Достоинством данного алгоритма является относительная быстрота выполнения поиска, по сравнению с алгоритмом *последовательного поиска*. Недостаток заключается в том, что *бинарный поиск* может применяться только на упорядоченном множестве.

//Поиск с сужением зоны(бинарный)

template <typename T>

int SearchBinary(T arr[], int n, T key)

{

int low, high, mid;

low = 0;

high = n - 1;

while (low <= high)

{

mid = (low + high) / 2;

if (key < arr[mid])

high = mid - 1;

else

if (key > arr[mid])

low = mid + 1;

else

return mid;

}

return -1;

}

template <typename T>

void sort(T arr[], int count)

{

int tmp;

for (int i = 0; i < count; i++)

{

for (int j = count - 1; j > i; --j)

{

if (arr[j] < arr[j - 1])

{

tmp = arr[j - 1];

arr[j - 1] = arr[j];

arr[j] = tmp;

}

}

cout << arr[i] << "?";

}

}

Для ключей, которые распределены не вполне случайно, производительность интерполяционного поиска еще выше. А его граничным случаем является метод распределяющего поиска.

Распределяющий поиск выполняется в 2 этапа.

**Первый этап – инвертирование.**

//Распределяющий поиск

template <typename T>

T\* Search(T arr[], int n, int\* min, int\* max)

{

\*min = \*max = arr[0];

for (int i = 1; i < n; i++)

{

if (arr[i] > \* max) \*max = arr[i];

if (arr[i] < \*min) \*min = arr[i];

}

if (\*max - \*min > \_ALLOCA\_S\_THRESHOLD) return NULL;

int\* ret = new int[\*max - \*min + 1];

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

{

ret[arr[i] - \*min] = i;

}

return ret;

}

**Второй этап – поиск.**

int\* ainv = Search(arr, 10, &min, &max);

result = -1;

if (key >= min && key <= max) result = ainv[key - min];

Вычислительная сложность:

1. O(N) - на подготовку
2. O(M) - на поиск М элементов
3. T(N,M)=O(N)+O(M)=**O(N)**

<https://prog-cpp.ru/algorithm-sort/>

<https://top-technologies.ru/ru/article/view?id=24620>

<https://works.doklad.ru/view/0C0aaW85LvE.html>

<https://www.intuit.ru/studies/courses/648/504/lecture/11466>