**Лекция 8.**

**Алгоритмы поиска. Последовательный поиск элемента. Бинарный поиск элемента. Последовательный поиск строки.**

Поиск - нахождение какой-либо конкретной информации в большом объеме ранее собранных данных.

Данные делятся на записи, и каждая запись имеет хотя бы один ключ. Ключ используется для того, чтобы отличить одну запись от другой.

Целью поиска является нахождение всех записей подходящих к заданному ключа поиска.

**Поиск элемента в массиве**

Для нахождения информации в неупорядоченных массиве нужен последовательный поиск, начинается с первого элемента и заканчивается при обнаружении подходящих данных или при достижении конца массива. Этот метод подходит для поиска неупорядоченной информации, но также можно использовать его и на отсортированных данных. Однако если данные уже отсортированы, можно применить двоичный поиск, находит данные быстрее.

**последовательный поиск**

Последовательный поиск очень легко запрограммировать. Приведенная ниже функция осуществляет поиск в массиве символов известной длины, пока не будет найден элемент с заданным ключом:

// Последовательный поиск

int function LinearSearch (Array A, int L, int R, int Key)

begin

for X = L to R do

if A [X] = Key then

return X

return 1; // Элемент не найден

end;

Функция возвращает индекс подходящего элемента, если таковой существует, или -1 в противном случае.

Понятно, что последовательный поиск в среднем просматривает n / 2 элементов. В лучшем случае он проверяет только один элемент, а в худшем - n. Если информация хранится на диске, поиск может занимать длительное время. Но если данные не упорядочены, последовательный поиск - единственно возможный метод.

**двоичный поиск**

Если данные, в которых осуществляется поиск, отсортированы, для нахождения элемента можно применять метод, намного превосходит предыдущий - двоичный поиск. В двоичного поиска есть и другие названия: дихотомическое поиск, логарифмический поиск, поиск распределением пополам. В нем применяется метод половинного деления. Сначала проверим средний элемент. Если он больше, чем искомый ключ, проверим средний элемент первой половины, в противном случае - средний элемент второй половины. Будем повторять эту процедуру до тех пор, пока искомый элемент не будет

найден или пока не останется очередного элемента.

Например, чтобы найти число 4 в массиве:

1 2 3 4 5 6 7 8 9

При двоичном поиске сначала проверяется средний элемент - число 5. Поскольку оно больше, чем 4, поиск продолжается в первой половине:

1 2 3 4 мая

Средний элемент теперь равен 3. Это меньше, чем 4, поэтому первая половина отвергается. Поиск продолжается во второй части:

5 апреля

На этот раз искомый элемент найден.

В двоичном поиске количество сравнений в худшем случае равна

log2n

В среднем случае количество сравнений значительно ниже, а в лучшем - равен 1. Двоичный поиск существенно быстрее линейный, относительно прост в реализации и общеупотребительный. Однако, в реальных программах случаи ошибочного использования линейного поиска в упорядоченных данных, приводящих к значительному уменьшению быстродействия.

Ниже приведена функция двоичного поиска.

// Итеративная версия

BinarySearch (A [0..N-1], value) {

low = 0

high = N - 1

while (low <= high) {

mid = (low + high) / 2

if (A [mid]> value)

high = mid - 1

else

if (A [mid] <value)

low = mid + 1

else

return mid // найдено

}

return -1 // Не найдено

}

Одним из вариантов реализации алгоритма является рекурсивная функция, получает массив, искомое значение и начальный и конечный индексы элементов в массиве.

// Рекурсивная версия

BinarySearch (A [0..N-1], value, low, high) {

if (high <low)

return -1 // Не найдено

mid = (low + high) / 2

if (A [mid]> value)

return BinarySearch (A, value, low, mid-1)

else if (A [mid] <value)

return BinarySearch (A, value, mid + 1, high)

else

return mid // найдено

}

// Итеративная версия с дополнениями

size\_t first = 0; / \* Первый элемент в массиве \* /

size\_t last = n; / \* Элемент в массиве, следующего за последним \* /

/ \* Если пересматривается не пустой фрагмент first <last \* /

size\_t mid;

if (n == 0)

{

/ \* Массив пустой \* /

}

else if (a [0]> x)

{

/\* не найдено; если вам надо вставить его со сдвигом - то в позицию 0 \* /

}

else if (a [n - 1] <x)

{

/\* не найдено; если вам надо вставить его со сдвигом - то в позицию n \* /

}

while (first <last)

{

/ \* ВНИМАНИЕ! В отличие от более простого (first + last) / 2, этот код устойчив к переполнений. \* /

mid = first + (last - first) / 2;

if (x <= a [mid])

{

last = mid;

}

else

{

first = mid + 1;

}

}

/ \* Если проверка n == 0 в начале отсутствует - значит, здесь раскомментировать! \* /

if (/ \* n! = 0 && \* / a [last] == ​​x)

{

/ \* Искомый элемент найден. last - искомый индекс \* /

} else

{

/ \* Искомый элемент не найден. Но если вам вдруг надо его вставить со смещением, то его место - last. \* /

}

**Поиск последовательности элементов в массиве**

Одно из самых простых задач поиска информации - **поиск точно заданной подстроки в строке.** Однако, эта задача чрезвычайно важно - оно применяется в текстовых редакторах, СУБД, поисковых машинах и тому подобное.

Поиск строки формально определяется следующим образом. Пусть задан массив Т из N элементов и массив W с M элементов, причем 0 <M≤N. Поиск строки обнаруживает первое вхождение W в Т, результатом будем считать индекс i, что указывает на первый с начала строки (с начала массива Т) совпадение с образцом

(Словом).

Пример. Нужно найти все вхождения образца W = abaa в текст T = abcabaabcabca.

****

Образец входит в текст только один раз, со смещение S = 3, индекс i = 4.

**Алгоритм прямого (последовательного) поиска**

Идея алгоритма:

1) I = 1,

2) сравнить I-й символ массива T с первым символом массива W,

3) совпадение → сравнить другие символы и т.д.,

4) расхождение → I = I + 1 и переход на пункт 2,

Условие окончания алгоритма:

1) подряд М сравнений удачные,

2) I + M> N, то есть слово не найдено.

Function Search (S: String; X: String; var Place: Byte): Boolean;

{Функция возвращает результат поиска в строке S}

{Подстроки X. Place - место первого вхождения}

var Res: Boolean; i: Integer;

Begin

Res = FALSE;

i = 1;

While (i <= Length (S) -Length (X) +1) And Not (Res) do

If Copy (S, i, Length (X)) = X then

begin

Res = TRUE;

Place = i

end

else i = i + 1;

Search = Res

End;

Пример. Нужно найти подстроку W = abcabd в тексте T = abcabcaabcabd.



Образец входит в текст только один раз, со сдвигом S = 7, индекс i = 8.

Сложность алгоритма:

Худой конец. Пусть массив T → {AAA .... AAAB} длина │T│ = N, вроде W → {A ... .AB} длина │W│ = M. Очевидно, что для выявления совпадения в конце строки потребуется сделать порядка N \* M сравнений, то есть O (N \* M).

Недостатки алгоритма:

1) высокая сложность - O (N \* M), в худшем случае - Θ ((N-M + 1) \* M)

2) после расхождения просмотр всегда начинается с первого символа образца и поэтому может включать символы T, которые ранее уже просматривались (если строка читается из вторичного памяти, то такие возвращения занимают много времени)

3) информация о тексте T, получаемая при проверке данного сдвига S, никак не используется при проверке следующих оползней.