Тема 5

Алгоритмы поиска

Поиск – процесс **отыскания информации во множестве данных** (обычно представляющих собой записи) путем просмотра специального поля в каждой записи, называемого ключом

Целью поиска является **отыскание записи** (если она есть) **с данным значением ключа**

Предметы (объекты), составляющие множество, называются его **элементами**

Элемент множества будет называться **ключом**, и обозначаться латинской буквой k_i с индексом, указывающим номер элемента

Пусть задано множество ключей $\{k_1, k_2, k_3, ..., k_n\}$
Необходимо отыскать во множестве ключ k_i
Поиск может быть **завершен в двух случаях:**

- ключ во множестве **отсутствует**
- ключ **найден во множестве**

Множество данных, в котором производится поиск, описывается как массив фиксированной длины:
A: array[1..n] of ItemType;

Обычно ItemType описывает **запись** с некоторым полем, играющим роль **ключа**, а сам массив представляет собой таблицу

Последовательный (линейный) поиск

Множество элементов просматривается **последовательно** в порядке, **гарантирующем просмотр всех элементов** множества
Если **будет найден** искомый элемент, просмотр прекращается **с положительным результатом**; если же **элемент не будет найден**, алгоритм должен выдать **отрицательный результат**

Алгоритм можно применять тогда, когда **нет никакой дополнительной информации о расположении данных** в рассматриваемой последовательности

<u>Сводится к последовательности шагов:</u>

1. [**Начальная установка**] Установить i:=1
2. [**Сравнение**] Если $k = k_i$, алгоритм заканчивается удачно
3. [**Продвижение**] Увеличить i на 1
4. [**Конец файла?**] Если $i \le N$, то вернуться к шагу 2. В противном случае алгоритм заканчивается неудачно


```
function LineSearch
(Key: ItemType, n: integer; var A: array[1..n] of ItemType): boolean;
{Функция линейного поиска,}
{если элемент найден, то возвращает значение true, иначе – false}
var i: integer;
begin
        i := 1;
        while (i <=n ) and (A[i]<>Key) do i := i + 1;
        if A[i]=Key then LineSearch := true
                else LineSearch := false;
end;
```

В среднем требует n/2 итераций цикла

Это означает временную сложность алгоритма поиска, пропорциональную O(n)

Никаких ограничений на порядок элементов в массиве алгоритм не накладывает

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

Модификация поиска – оптимизация цикла:
В цикле *while* производятся два сравнения: (i<=n) и (A [i] <>Key).
Избавимся от первого, введя в массив так называемый «барьер», положив A[n+1] := Key

```
function LineSearchWithBarrier
(Key: ItemType, n: integer; var A: array[1..n+1] of ItemType): boolean;
{Функция линейного поиска с барьером,}
{если элемент найден, то возвращает значение true, иначе - false}
var i: integer;
begin
        I := 1;
        A[n+1] := Key;
        while A[i]<>Key do I := I + 1;
        if I <= n then LineSearchWithBarrier := true
                else LineSearchWithBarrier := false;
end;
```

Такая функция будет работать быстрее, но временная сложность алгоритма остается такой же – O(n)

Бинарный поиск

Этот алгоритм поиска предполагает, что множество хранится, как некоторая упорядоченная (например, по возрастанию) последовательность элементов, к которым можно получить прямой доступ посредством индекса

Массив отсортирован – каждый последующий ключ больше (во всяком случае – не меньше) предыдущего:

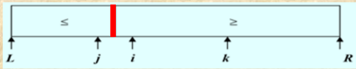
$\{k_1 \le k_2 \le k_3 \le k_4, \ldots, k_{n-1} \le k_n\}$

Суть метода: отыскиваемый ключ сравнивается с центральным элементом множества, если он меньше центрального, то поиск продолжается в левом подмножестве, в противном случае – в правом

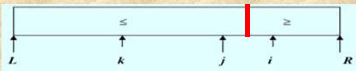
Медианой последовательности из n элементов считается элемент, значение которого меньше (или равно) половины n элементов и больше (или равно) другой половины

В алгоритме Хоара для нахождения медианы используется операция разделения с $L = 1$, $R = n$ и с $a[k]$, выбранным в качестве разделяющего значения x.
Получаются значения индексов i и j, такие, что

1. Разделяющее значение x было слишком мало – в результате граница между двумя частями ниже искомого значения k. Процесс разделения следует повторить для элементов $a[i], \ldots, a[R]$



2. Выбранная граница x была слишком велика – граница выше искомого значения k. Операцию разбиения следует повторить на подмассиве $a[L], \ldots, a[j]$



3. Значение k лежит в интервале $j < k < i$ – элемент $a[k]$ разделяет массив в заданной пропорции

Пример 1. Во множестве элементов отыскать ключ равный 653.
Поиск ключа $K = 653$ осуществляется за четыре шага:
[061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908]
061 087 154 170 275 426 503 509 [512 612 653 677 703 765 897 908]
061 087 154 170 275 426 503 509 [512 612 653] 677 703 765 897 908
061 087 154 170 275 426 503 509 512 612 [653] 677 703 765 897 908

Пример 2. Дано упорядоченное множество элементов
{7,8,12,16,18,20,30,38,49,50,54,60,61,69,75,79,80,81,95,101,123,198}
Найти во множестве ключ $K = 61$

Шаг 1. $N_{эл\text{-}та} = [n / 2]+1 = [22 / 2] + 1 = 12$
{7, 8, 12, 16, 18, 20, 30, 38, 49, 50, 54, 60, 61, 69, 75, 79, 80, 81, 95, 101, 123, 198}
$K \vee k_{12}$ → 61 > 60 → Дальнейший поиск в правом подмножестве

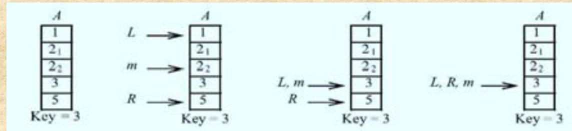
Шаг 2. $N_{эл\text{-}та} = [n / 2] +1 = [10 / 2] +1 = 6$
{61, 69, 75, 79, 80, 81, 95, 101, 123, 198}
$K \vee k_{18}$ → 61<81 → Дальнейший поиск в левом подмножестве

Шаг 3. $N_{эл\text{-}та} = [n/2] +1 = [5/2] +1 = 3$
{61, 69, 75, 79, 80}
$K \vee k_{15}$ → 61<75 → Дальнейший поиск в левом подмножестве

Шаг 4. $N_{эл\text{-}та} = [n/2] +1 = [2/2] +1 = 2$
{61, 69}
$K \vee k_{14}$ → 61<69 → Дальнейший поиск в левом подмножестве

Шаг 5. {61}
$K \vee k_{13}$ → 61=61 → Искомый ключ найден под номером 13

Имеем массив A = {1, 2, 2, 3, 5}
Сначала область поиска – часть массива (L, R), где L = 1, а R = n
Индекс среднего элемента: m := (L+R) div 2
Если Key > A_m → *Key* находится в одном из элементов с индексами от L + m до R, следовательно, можно присвоить L := m + 1
В противном случае можно положить R := m



На каждом шаге область поиска будет сокращаться вдвое

Как только L станет равно R, т. е. область поиска сократится до одного элемента, можно будет проверить этот элемент на равенство искомому и сделать вывод о результате поиска


```
function BinarySearch
    (Key: ItemType, n: integer; var A: array[1..n] of ItemType): boolean;
{Функция двоичного поиска,}
{если элемент найден, то возвращает значение true, иначе – false}
var L, m, R: integer;
begin
    L := 1; R := n;
    while (L <> R) do
    begin
        m := (L+R) div 2;
        if Key > A[m] then L := m+1 else R := m;
    end;
    if A[L]= Key then BinarySearch := true
                 else BinarySearch := false;
end;
```

Область поиска на каждом шаге сокращается вдвое, а это означает временную сложность алгоритма, пропорциональную O(log n)

Фибоначчиев поиск

Анализируются элементы, находящиеся в позициях, равных числам Фибоначчи: {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …}
Поиск продолжается до тех пор, пока не будет найден интервал между двумя позициями с расположением отыскиваемого ключа

Фибоначчиев поиск предназначается для поиска аргумента K среди расположенных отсортированных ключей $K_1 < K_2 < \ldots < K_n$

Предположим, что $n+1$ есть число Фибоначчи F_{k+1}

1 – Начальная установка. Установить $i{:=}F_k$, $p{:=}F_{k-1}$, $q{:=}F_{k-2}$. (p и q обозначают последовательные числа Фибоначчи)
2 – Сравнение. Если $K < K_i$, то перейти на шаг 3; если $K > K_i$, то перейти на шаг 4; если $K = K_i$, алгоритм заканчивается удачно
3 – Уменьшение i. Если $q = 0$, алгоритм заканчивается неудачно. Если $q \ne 0$, то установить $i := i - q$, заменить (p, q) на $(q, p - q)$ и вернуться на шаг 2.
4 – Увеличение i. Если $p = 1$, алгоритм заканчивается неудачно. Если $p \ne 1$, установить $i := i + q$, $p := p - q$, $q := q - p$ и вернуться на шаг 2.

Пример. Дано исходное множество ключей
{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}
Пусть отыскиваемый ключ равен 42 ($K = 42$).
Последовательное сравнение отыскиваемого ключа будет проводиться с элементами исходного множества, расположенных в позициях, равных числам Фибоначчи: {1, 2, 3, 5, 8, 13, 21,…}
Шаг 1. $K \vee k_1$ → 42>3 → отыскиваемый ключ сравнивается с ключом, стоящим в позиции, равной числу Фибоначчи
Шаг 2. $K \vee k_2$ → 42>5 → сравнение продолжается с ключом, стоящим в позиции, равной следующему числу Фибоначчи
Шаг 3. $K \vee k_3$ → 42>8 → сравнение продолжается
Шаг 4. $K \vee k_5$ → 42>11 → сравнение продолжается
Шаг 5. $K \vee k_8$ → 42>19 → сравнение продолжается
Шаг 6. $K \vee k_{13}$ → 42>35 → сравнение продолжается
Шаг 7. $K \vee k_{18}$ → 42<52 → найден интервал, в котором находится отыскиваемый ключ, т.е отыскиваемый ключ может находится в исходном множестве между 13 и 18 позициями
т.е. {35, 37, 42, 45, 48, 52}
В найденном интервале поиск вновь ведется в позициях, равных числам Фибоначчи

Черников Б.В. 13

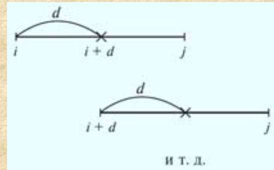
Интерполяционный поиск

Исходное множество – упорядочено по возрастанию весов
Первоначальное сравнение осуществляется на расстоянии шага d, который определяется по формуле

$$d = \left[\frac{(j-i)(K-K_i)}{K_j-K_i}\right]$$

i – номер первого элемента;
j – номер последнего элемента;
K – ключ; K_i, K_j – значения ключей в i и j позициях; [ ] – целая часть от числа

Идея: шаг поиска d меняется по формуле после каждого этапа



Алгоритм заканчивает работу при $d = 0$, при этом анализируются соседние элементы, после чего принимается окончательное решение о результатах поиска.
Метод прекрасно работает, если исходное множество – арифметическая прогрессия или множество, приближенное к ней

Черников Б.В. 14

Пример 1. Дано множество ключей:
{2, 9, 10, 12, 20, 24, 28, 30, 37, 40, 45, 50, 51, 60, 65, 70, 74, 76}
Пусть искомый ключ $K = 70$.
Шаг 1. Определим шаг d для исходного множества ($i = 1$; $j = 18$):
d = [(18 – 1)·(70 – 2) / (76 – 2)] = 15
Сравниваем ключ, стоящий под номером 16 с искомым ключом:
$k_{16} \vee K$ → 70 = 70 → ключ найден

Пример 2. Дано множество ключей:
{4, 5, 10, 23, 24, 30, 47, 50, 59, 60, 64, 65, 77, 90, 95, 98, 102}
Требуется отыскать ключ $K = 90$.
Шаг 1. Определим шаг d для исходного множества ($i = 1$; $j = 17$):
d = [(17 – 1)·(90 – 4) / (102 – 4)] = 14
Сравниваем ключ, стоящий под номером 15 с ключом:
$k_{15} \vee K$ → 95 > 90 → следовательно, сужается область поиска
{4, 5, 10, 23, 24, 30, 47, 50, 59, 60, 64, 65, 77, 90, 95}
Шаг 2. Определим шаг d для множества ключей ($i = 1$; $j = 15$):
d = [(15 – 1) ·(90 – 4) / (95 – 4)] = 13
Сравниваем ключ, стоящий под порядковым номером 14 с отыскиваемым ключом:
$k_{14} \vee K$ → $k_{14} = K$ → 90 = 90 → ключ найден

Черников Б.В. 15

Поиск по бинарному дереву

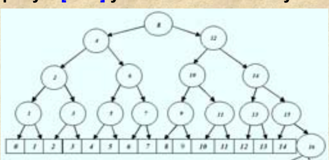
Дерево двоичного поиска для множества чисел S – это размеченное бинарное дерево, каждой вершине которого сопоставлено число из множества S, а все пометки удовлетворяют правилу: если больше – направо, если меньше – налево

Пример. Для набора чисел {7, 3, 5, 2, 8, 1, 6, 10, 9, 4, 11} получится бинарное дерево

Чтобы правильно учесть повторения чисел, можно ввести дополнительное поле, которое будет хранить количество вхождений для каждого числа

Черников Б.В. 16

Бинарное дерево для бинарного поиска среди n записей можно построить следующим образом: при $n = 0$ дерево сводится к узлу 0. В противном случае корневой узел – $[n / 2]$, левое поддерево соответствует бинарному дереву с $[n / 2] – 1$ узлами, а правое – дереву с $[n / 2]$ узлами и числами в узлах, увеличенными на $[n / 2]$



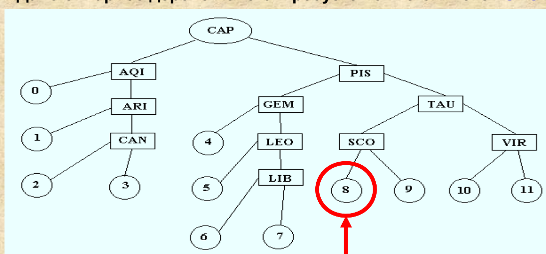
Бинарное дерево для n=16

Число порождений отдельного узла (число поддеревьев данного корня) – его степень
Узел с нулевой степенью называют листом или концевым узлом
Максимальное значение степени всех узлов дерева – степень дерева

Алгоритм поиска по бинарному дереву: вначале аргумент поиска сравнивается с ключом, находящимся в корне. Если аргумент совпадает с ключом, поиск заканчивается. Если не совпадает, то в случае, когда аргумент меньше ключа, поиск продолжается в левом поддереве, когда больше ключа – в правом поддереве. Увеличив уровень на 1, повторяют сравнение, считая текущий узел корнем

Черников Б.В. 17

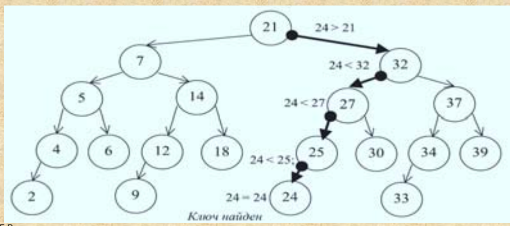
Дано бинарное дерево поиска. Требуется отыскать ключ SAG



1. По первой букве латинского алфавита – название SAG больше чем CAP → поиск будем вести в правой ветви
2. SAG больше PIS → вправо
3. SAG меньше TAU → влево
4. SAG меньше SCO → узел 8
SAG должно находиться в узле 8

Черников Б.В. 18

Пример. Дано исходное множество ключей
{2, 4, 5, 6, 7, 9, 12, 14, 18, 21, 24, 25, 27, 30, 32, 33, 34, 37, 39}
Множество ключей должно быть упорядочено по возрастанию
Переходим от линейного списка к построению бинарного дерева поиска, где корень дерева – центральный элемент множества:
$N_ц = [N / 2] + 1$, где N – количество элементов множества
Вершиной по левой ветке является центральный элемент левого подмножества, а правой – правого подмножества, и т. д.
Отыскиваемый ключ $K = 24$.
Исходное множество имеет 19 элементов, $N_ц = [19 / 2] + 1 = 10$

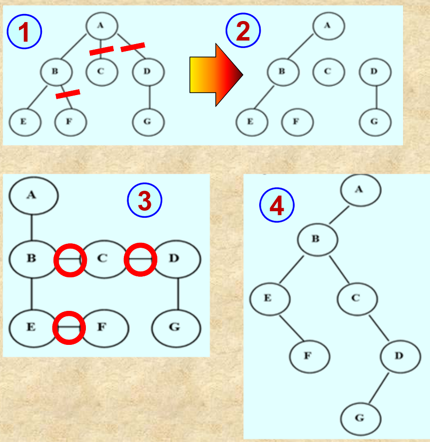


Преобразование произвольного дерева в бинарное

Упорядоченные деревья степени 2 – бинарные деревья
Бинарное дерево состоит из конечного множества элементов (узлов), каждый из которых либо пуст, либо состоит из корня (узла), связанного с двумя различными бинарными деревьями, называемыми левым и правым поддеревом корня
Деревья, у которых d > 2, называются d-арными

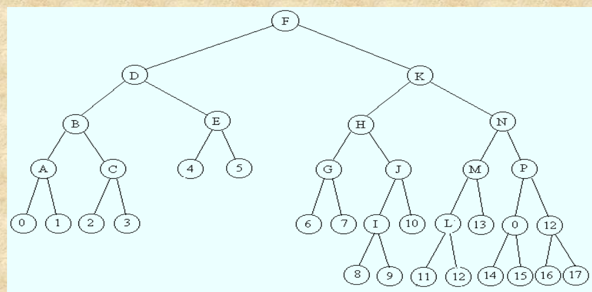
Преобразование произвольного дерева с упорядоченными узлами в бинарное дерево:

1. В каждом узле исходного дерева вычеркиваем все ветви, кроме самых левых ветвей
2. Соединяем горизонтальными ветвями узлы одного уровня, которые являются «братьями» в исходном дереве (если несколько узлов имеют общего предка, то такие узлы – «братья»)
3. Левым потомком каждого узла x считается непосредственно находящийся под ним узел (если он есть), а в качестве правого потомка – соседний справа «брат» для x, если таковой имеется

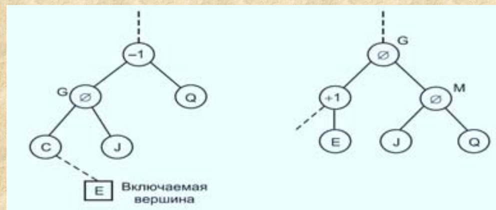


Сбалансированное бинарное дерево

Бинарное дерево – сбалансированное (B-balanced), если высота левого поддерева каждого узла отличается от высоты правого не более чем на 1



Показатель сбалансированности узла, то есть разность высот правого и левого поддерева обозначается $B = +1; 0; -1$



При восстановлении баланса дерева по высоте учитывается показатель B

Символы указывают:

- левое поддерево выше правого (+1)
- поддеревья равны по высоте (∅)
- правое поддерево выше левого (–1)

Поиск хешированием

Функция хеширования

В рассмотренных методах поиска число итераций в лучшем случае было пропорционально O(log n)
Надо найти такой метод поиска, при котором число итераций не зависело бы от размера таблицы, а в идеальном случае поиск сводился бы к одному шагу

Идеально быстрый поиск – таблица прямого доступа
При создании таблицы выделяется память для хранения всей таблицы и заполняется пустыми записями
Затем записи вносятся в таблицу – каждая на свое место, определяемое ее ключом
При поиске ключ используется как адрес и по этому адресу выбирается запись.
Если выбранная запись пустая, то записи с таким ключом вообще нет в таблице

Пространство ключей – множество всех **теоретически возможных** значений ключей записи
Пространство записей – множество тех **ячеек памяти**, которые выделяются для хранения таблицы

Из соображений экономии памяти целесообразно назначать размер пространства записей равным размеру фактического множества записей или превосходящим его незначительно

Необходимо иметь некоторую функцию, обеспечивающую отображение точки из пространства ключей в точку в пространстве записей, т. е. преобразование ключа в адрес записи: $a := h(k)$, где a – адрес, k – ключ

Такая функция называется **функцией хеширования** (другие ее названия – **функция перемешивания**, **функция рандомизации**)

Идеальной хеш-функцией является такая функция, которая для любых **двух неодинаковых ключей** дает **неодинаковые адреса**:
$k1 \neq k2 \rightarrow h(k1) \neq h(k2)$

Ситуация, при которой **разные ключи отображаются в один и тот же адрес записи**, называется **коллизией**, или **переполнением**, а такие ключи называются **синонимами**

Коллизии составляют основную проблему для хеш-таблиц

Если хеш-функция, преобразующая ключ в адрес, может порождать коллизии, то однозначной обратной функции: $k := h'(a)$, позволяющей восстановить ключ по известному адресу, существовать не может → **ключ должен обязательно входить в состав записи хешированной таблицы как одно из ее полей**

Требования к хеш-функции:
- должна обеспечивать **равномерное распределение** отображений фактических ключей по пространству записей
- должна порождать **как можно меньше коллизий** для данного фактического множества записей
- **не должна отображать какую-либо связь** между значениями ключей в связь между значениями адресов;
- должна быть **простой** и **быстрой для вычисления**

1. **Деление по модулю числового значения ключа на размер пространства записи.** Результат интерпретируется как адрес записи. Плохо соответствует первым трем требованиям. Обычно применяется как последний шаг в более сложных функциях хеширования для приведения результата к размеру пространства записей
2. **Функция середины квадрата.** Значение ключа преобразуется в число, это число затем возводится в квадрат, из него выбираются несколько средних цифр и интерпретируются как адрес записи
3. **Функция свертки.** Цифровое представление ключа разбивается на части, каждая из которых имеет длину, равную длине адреса. Над частями производятся какие-то арифметические или поразрядные логические операции, результат которых интерпретируется как адрес
4. **Функция преобразования системы счисления.** Ключ, записанный как число в некоторой системе счисления P, интерпретируется как число в системе счисления Q > P. Обычно выбирают Q = P+1. Это число переводится из системы Q обратно в систему P, приводится к размеру пространства записей и интерпретируется как адрес

Алгоритм поиска хешированием

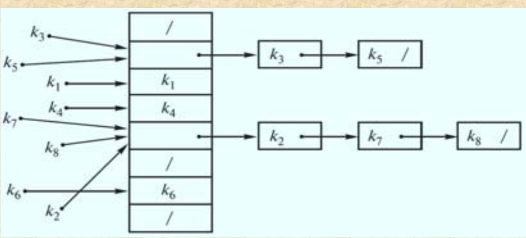
В основе поиска лежит переход от исходного множества к множеству хеш-функций $h(k)$
Хеш-функция имеет следующий вид: $h(k) = k \bmod (m)$,
где k – ключ; m – целое число; *mod* – целочисленный остаток от деления

Пример. Пусть дано множество {9, 1, 4, 10, 8, 5}.
Определим для него хеш-функцию $h(k)= k \bmod (m)$.
1. Пусть $m = 1$ → $h(k)$ = {0, 0, 0, 0, 0, 0} → Множество хеш–функций состоит из нулей
2. Пусть $m = 20$ → $h(k)$ = {9, 1, 4, 10, 8, 5} → Множество хеш-функций повторяет исходное множество
3. Пусть m равно половине максимального ключа $m = [K_{max} / 2]$ → $m = [10 / 2] = 5$ → $h(k)$ = {4, 1, 4, 0, 3, 0}

Хеш-функция указывает адрес, по которому следует отыскивать ключ

Для разных ключей хеш-функция может принимать **одинаковые значения**, такая ситуация называется **коллизией**

Поиск хешированием заключается в **устранении коллизий методом цепочек**



Пример 1. Дано множество ключей {7,13,6,3,9,4,8,5}. Найти ключ $K=27$
Хеш-функция равна $h(k) = K \bmod (m)$;
$m = [13/2] = 6$ (т.к. 13 – максимальный ключ)
$h(k)$ = {1, 1, 0, 3, 3, 4, 2, 5}

h(k)	Цепочки ключей
0	6
1	7, 13
2	8
3	3, 9
4	4
5	5

Попарным сравнением множества хеш-функций и множества исходных ключей, заполняем таблицу. При этом хеш-функция указывает адрес, по которому следует отыскивать ключ

Если отыскивается ключ $K = 27$, тогда $h(k) = 27 \bmod 6 = 3$
Это значит, что ключ $K = 27$ может быть только в 3-й строке
Но его там нет → данный ключ отсутствует в исходном множестве

Пример 2. Дано множество ключей {7, 1, 8, 5, 14, 9, 16, 3, 4}
Найти ключ $K = 14$.
Хеш-функция равна $h(k) = K \bmod (m)$;
$m = [16/2] = 8$ (т.к. 16 – максимальный ключ)
$h(k)$ = {7, 1, 0, 5, 6, 1, 0, 3, 4}

h(k)	Цепочки ключей
0	8, 16
1	1, 9
3	3
4	4
5	5
6	14
7	7

Поиск осуществляется по таблице: $K = 14$

$h(k) = 14 \bmod 8 = 6$ → ключ $K = 14$ может быть только в 6-й строке

Поиск по бору

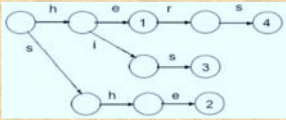
Бор (trie, луч, нагруженное дерево) – структура данных для хранения набора строк, представляющая из себя «подвешенное» дерево с символами на ребрах (по одному на каждом ребре)

Строки получаются прохождением из корня по ребрам, записывая соответствующие им символы, до терминальной вершины
Размер бора линейно зависит от суммы длин всех строк, а поиск в бору занимает время, пропорциональное длине образца

Бор представляет собой *m*-арное дерево. Каждый узел уровня *h* представляет множество всех ключей, начинающихся с определенной последовательности из *h* литер. Узел определяет *m*-путевое разветвление в зависимости от (*h* + 1)-ой литеры

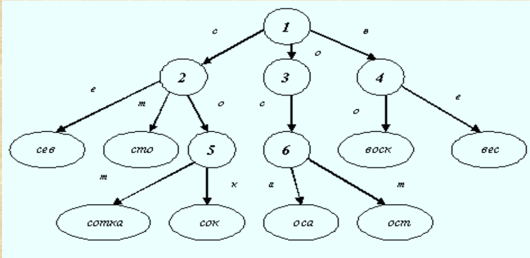
Символы	Узлы				
	1	2	3	...	N

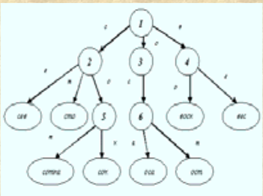
- В первом узле записывается первая буква или цифра ключа
- Во втором узле к ней добавляется еще один символ и т.д.
- Если слово, начинающееся с определенной буквы (цифры), – единственное, то оно сразу записывается в первом узле



В графическом виде бор представляет собой граф – пример бора для хранения словаря {he, she, his, hers}

Пример. **Пусть задано множество слов: {воск, сок, оса, сто, сев, век, ост, сотка}**



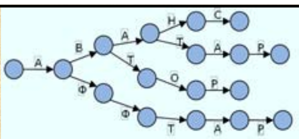


Символы	Узлы					
	1	2	3	4	5	6
С	(2)		(6)			
О	(3)	(5)		воск		
Т		сто			сотка	ост
К					сок	
А						оса
В	(4)					
Е		сев		вес		

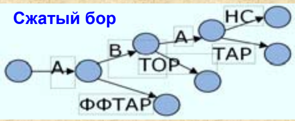
Пусть нужно найти слово ост

1. **Слово начинается на букву «о» → смотрим пересечение первого столбца со строкой, обозначенной буквой «о»**
2. **Записана ссылка (3) – обращаемся к третьему столбцу «бора»**
3. **Вторая буква слова ост – буква «с» → на пересечении третьего столбца и строки с буквой «с» записана ссылка (6), которая отправляет нас на просмотр шестого столбца бора**
4. **В этом столбце видим два слова: ост и оса. Поскольку третьей буквой искомого слова является буква «т», то на строке, обозначенной этой буквой, находим слово ост**

Бор – классическая структура данных для хранения набора строк. Размер бора линейно зависит от суммы длин всех строк, а поиск в бору занимает время, пропорциональное длине образца



Сжатый бор



Сжатый бор – отличается от бора следующим: если у какой-то вершины исходящая степень 1, то эту вершину, ребро, входящее в нее, и ребро, исходящее из нее, можно объединить в одно ребро с более чем одним символом

l-слабый бор – улучшение сжатого бора. Глубина вершины – длина пути от корня до этой вершины (глубина корня равна 0). l-слабый бор имеет структуру сжатого бора в вершинах с глубиной, меньшей l.

На уровне l ветвление заканчивается, и все суффиксы дописываются на ребрах, исходящих из вершин глубины l. В частности, из вершины глубины l могут выходить ребра, помеченные строками, начинающимися с одной буквы

2-слабый бор

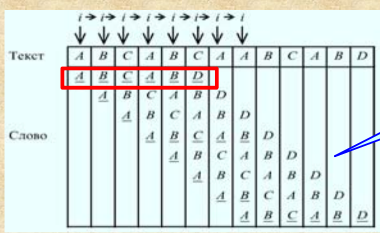
Алгоритмы поиска словесной информации

Пусть задан массив Txt из N элементов, называемый текстом, и массив Wrd из M элементов, называемый словом (0 < M < N)
Описать их можно как строки.
Поиск слова обнаруживает первое вхождение Wrd в Txt

Прямой поиск строки

Посимвольное сравнение текста со словом

- **Начальный момент → сравнение первого символа текста с первым символом слова, второго символа текста со вторым символом слова и т. д.**
- **Совпадение всех символов → фиксируется факт нахождения слова.**
- **В противном случае →«сдвиг» слова на одну позицию вправо и повторяется посимвольное сравнение, т. е. сравнивается второй символ текста с первым символом слова, третий символ текста со вторым символом слова и т. д.**



Подчеркнуты символы, подвергшиеся сравнению

Такие «сдвиги» слова повторяются, пока конец слова не достигнет конца текста или не произошло полное совпадение символов слова с текстом (т. е. слово найдено)

```
function DirectTxtSearch(var Wrd: TWrd;
var Txt: TText;
var Position: integer): boolean;
{Функция поиска слова Wrd в тексте Txt,}
{если слово найдено, то возвращает значение true}
{и позицию Position начала первого слова Wrd,}
{иначе – false и Position не изменяется}
var i: integer; {Индекс начала слова в тексте}
    j: integer; {Индекс текущего символа слова}
begin
        i := 0;
        repeat
                j := 1; i := i + 1;
                {Осуществляем посимвольное сравнение}
                while (j <= M) and (Txt[i+j-1] = Wrd[j]) do j := j+1;
        until (j = M+1) or {Совпало все слово}
                        (i >= N-M+1); {Конец слова за концом текста}
        {Оценка результатов поиска}
        if j = M+1 then begin DirectTxtSearch := true; Position := i; end
                else begin DirectTxtSearch := false; end;
end;
```

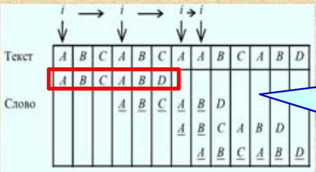
В худшем случае алгоритм будет малоэффективен: его сложность будет пропорциональна $O((N – M)\cdot M)$

Алгоритм Кнута – Морриса – Пратта (КМП)

В 1970 г. Д. Кнут, Д. Морис и В. Пратт изобрели алгоритм (КМП-алгоритм), фактически требующий только O(N) сравнений даже в самом плохом случае

Идея: После частичного совпадения начальной части слова с символами текста известна пройденная часть текста и можно «вычислить» некоторые сведения (на основе самого слова), с помощью которых затем быстро продвинуться по тексту

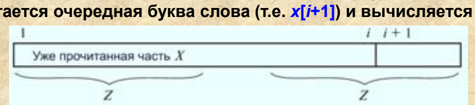
Если j определяет позицию в слове, содержащую первый несовпадающий символ (как в алгоритме прямого поиска), то величина сдвига Shift определяется как *j – LenSuff – 1*.
Значение LenSuff определяется как размер самой длинной последовательности символов слова, непосредственно предшествующих позиции *j* (суффикс), которая полностью совпадает с началом слова.
LenSuff зависит только от слова и не зависит от текста.
Для каждого j будет своя величина сдвига



При каждом несовпадении пары символов слово сдвигается на переменную величину, и меньшие сдвиги не могут привести к полному совпадению

Алгоритм получает на вход слово
X = *x*[1]*x*[2] ... *x*[*n*]
и просматривает его слева направо буква за буквой, заполняя при этом массив натуральных чисел *L*[1] ... *L*[*n*],
где *L*[*i*] = длина слова *L*(*x*[1]...*x*[*i*])
Таким образом, *L*[*i*] есть длина наибольшего начала слова *x*[1]...*x*[*i*], одновременно являющегося его концом

Пример. Используя алгоритм КМП, определить, является ли слово *A* подсловом слова *B*?
Решение. Применим алгоритм КМП к слову *A#B*,
где # – специальная буква, не встречающаяся ни в *A*, ни в *B*.
Слово *A* является подсловом слова *B* тогда и только тогда, когда среди чисел в массиве *L* будет число, равное длине слова *A*.
Предположим, что первые *i* значений *L*[1]...*L*[*i*] уже найдены.
Читается очередная буква слова (т.е. *x*[*i*+1]) и вычисляется *L*[*i*+1]



Определить начала *Z* слова *x*[1]...*x*[*i*+1], одновременно являющиеся его концами – из них следует выбрать самое длинное.
Рассмотрим все начала слова *x*[1]...*x*[*i*], являющиеся одновременно его концами. Из них выберем подходящие – те, за которыми следует буква *x*[*i*+1]. Из подходящих выберем самое длинное. Приписав в его конец *x*[*i*+1], получим искомое слово *Z*. НО!!! Все слова, являющиеся одновременно началами и концами данного слова, можно получить повторными применениями к нему функции *L*


```
i:=1; l[1]:=0;
{таблица l[1]..l[i] заполнена правильно}
while i <> n do
begin
        len:= l[i]
        {len – длина начала слова x[1]..x[i], которое является
        его концом; все более длинные начала оказались
неподходящими}
        while (x[len+1]<>x[i+1]) and (len>0) do
        begin {начало не подходит, применяем к нему функцию l}
                len:=l[len];
        end;
        {нашли подходящее слово или убедились в его отсутствии}
        if x[len+1]=x[i+1] then do
        begin           {x[1]..x[len] – самое длинное подходящее
начало}
                l[i+1]:=len+1;
        end
                        else begin l[i+1]:= 0; end;  {подходящих нет}
        i:=i+1;
end;
```


Пример. Запишем алгоритм, проверяющий, является ли слово *X* = *x*[1]...*x*[*n*] подсловом слова *Y* = *y*[1]...*y*[*m*].
Решение. Вычисляем таблицу *L*[1]...*L*[*n*]

```
j:=0; len:=0;
        {len – длина максимального начала слова X,
        одновременно  являющегося концом слова y[1]..j[j]}
while (len<>n) and (j<>m) do
begin
        while (x[len+1]<>y[j+1]) and (len>0) do
        begin   {начало не подходит, применяем к нему функцию l}
                len: = l[len];
        end;
        {нашли подходящее слово или убедились в его отсутствии}
        if x[len+1]=y[j+1] then do
        begin   {x[1]..x[len] – самое длинное подходящее начало}
                len:=len+1;
        end
                        else begin len:=0; end; {подходящих нет}
        j:=j+1;
end;
{если len=n, то слово X встретилось;
иначе мы дошли до конца слова Y, так и не встретив X}
```

Алгоритм Боуера – Мура (БМ)

КМП-алгоритм дает выигрыш тогда, когда неудаче предшествовало некоторое число совпадений – тогда слово сдвигается более чем на единицу

Метод, предложенный Р. Боуером и Д. Муром в 1975 году (БМ-алгоритм), не только улучшает обработку самого плохого случая, но дает выигрыш в промежуточных ситуациях

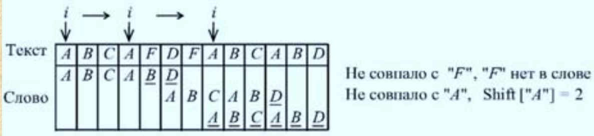
Идея: Сравнение символов идет с конца слова, а не с начала
Перед поиском на основе слова формируется некоторая таблица. Пусть для каждого символа x из алфавита величина $Shift_x$ – расстояние от самого правого в слове вхождения x до правого конца слова.

Пусть обнаружено расхождение между словом и текстом, причем символ в тексте, который не совпал – *x*.

Тогда слово сразу можно сдвинуть вправо так, чтобы самый правый символ слова, равный *x*, оказался бы в той же позиции, что и символ текста *x*.

Этот сдвиг будет на число позиций, большее единицы.

Если несовпадающий символ текста *x* в слове вообще не встречается, то сдвиг становится даже больше: сдвигаем вправо так, чтобы ни один символ слова не накладывался на символ *x*




```
function
   BMTxtSearch(var Wrd: TWrd; var Txt: TText; var Position: integer): boolean;
{Функция поиска слова Wrd в тексте Txt, если слово найдено,
{то возвращает значение true и позицию Position начала первого слова Wrd, иначе –
false и Position не изменяется}
var
i,                                  {Индекс начала слова в тексте}
j: integer;                         {Индекс текущего символа слова}
ch: char;
Shift: array[' '..'я'] of integer;  {Массив смещений}
begin
{Заполнение массива Shift}
        for ch:=' ' to 'я' do Shift[ch] := M;
        for j:=1 to M do Shift[Wrd[j]] := M–j;
        {Поиск слова Wrd в тексте Txt}
        i := 1; {Начало слова совпадает с началом текста}
        repeat
                j := M+1; {Сравнивать будем с последнего символа}
        {Посимвольное сравнение слова, начиная с правого символа}
                repeat j := j–1; until (j < 1) or (Wrd[j] <> Txt[i+j–1]);
                if j >= 1 then i := i + (j + Shift[Txt[i+j-1]] – M); {Сдвиг слова вправо}
        until (j < 1) or (i > N–M+1);
{Оценка результатов поиска}
        if j < 1 then begin BMTxtSearch := true; Position := i; end
                else begin BMTxtSearch := false; end;
end;
```


Этот алгоритм в типичной ситуации читает лишь небольшую часть всех букв слова, в котором ищется заданный образец

Пример. Отыскивается образец *abcd*.
Посмотрим на четвертую букву слова в тексте: если, например, это буква *e*, то нет никакой необходимости читать первые три буквы (в образце буквы *e* нет, поэтому он может начаться не раньше пятой буквы)

Почти всегда, кроме специально построенных примеров, данный алгоритм требует значительно меньше *O(N)* сравнений

В самых благоприятных обстоятельствах, когда последний символ слова всегда попадает на несовпадающий символ текста, число сравнений пропорционально O(N / M)