Глава 4

АЛГОРИТМЫ ПОИСКА

ПЛАН

- 1. Задача поиска и базовые алгоритмы ее решения
- 2. Организация поиска на деревьях
- 3. Поиск в хеш-таблицах (рассмотрено ранее)
- 4. Поиск подстрок (?)

1 Задача поиска и базовые алгоритмы ее решения

- Постановка проблемы

$$f : X \to Y \qquad ?\, x \in X : f(x) = y$$

Элементы Y – ключи

Процедура поиска x – алгоритм поиска

Часто требуется, чтобы на множестве элементов было задано отношение линейного порядка

Результат поиска

- 1. Поиск завершился успешно: искомые объекты были найдены
- 2.Поиск оказался неудачным: искомые объекты не были обнаружены
- Сопутствующий вопрос: «А был ли мальчик?»

Классификация алгоритмов по некоторым критериям

- 1.Внешний и внутренний поиск

- 2. Статический (содержимое множества *X* остается неизменным) и динамический (множество *X* изменяется путем вставки либо удаления в него элементов) поиск

- 3. Поиск на базе сравнения ключей или их свойств

Элементарные методы поиска
(без учета внутренней структуры объектов)

Будем считать метод поиска элементарным, если для успешного завершения алгоритма элементы множества хранятся в элементарной структуре данных

Элементы из множества X записаны в массив и пронумерованы натуральными числами от 1 до $|X| = n$, если речь идет о дискретных объектах (или $|X| = |a-b|$ в непрерывном случае)

Базовые алгоритмы поиска: линейный, двоичный, тернарный (в т.ч. на основе «золотого сечения»), интерполяционный

1.1 Линейный поиск:

Последовательный просмотр элементов X

Реализация

```
Процедура 4.1.1. Linear Search
01:  Linear Search Impl(A[1, n], b)
02:        for index = 1 to n do
03:              if f(A[index]) = b then
04:                    return index
```

РС

$$T(n) = T(n-1) + c,\ n \geq 2,$$

$$T(1) = 1$$

1) $T(n) = T(n-1)$ – однор. ЛРС 1-го пор-ка

$\lambda = 1 \Rightarrow T_0(n) = C \cdot 1^n = C$

2) $T^*(n) = a \cdot 1^n \cdot n$ (т.к. $\lambda = 1$ – корень х.м.)

$T^* = an$

$T(n) = T(n-1) + c \Rightarrow an = a(n-1) + c$

$a = c$

$T^* = cn$

3) $T(n) = C + cn$, C – произвольная

c – заданная константа

$T(1) = 1 \Rightarrow C + c \cdot 1 = 1$

$C = 1 - c$

4) $\boxed{T(n) = 1 - c + cn}$

$n \to \infty \Rightarrow T(n) = \begin{cases} O(n) \\ \Omega(n) \\ \Theta(n) \end{cases}$

Объяснить

Оценочные характеристики

Основной недостаток – просмотр всех элементов множества, число которых может быть достаточно велико. Следовательно, не рекомендуется используется в ситуациях, когда процедуру поиска требуется повторять неоднократно

Положительная сторона – универсальность: линейный поиск применим как к упорядоченным множествам, так и к множествам, на которых не задано отношение порядка, а только ключ поиска

1.2 Двоичный поиск

на Y задано отношение линейного порядка

ИДЕЯ: элемент с искомым ключом находится в i-той позиции

РС

$$\forall j : i < j \Rightarrow f(x_i) \prec f(x_j)$$

$$\forall j : i < j \Rightarrow f(x_i) \succ f(x_j)$$

$$T(n) = T(n/2) + c, n \geq 2,$$

$$T(1) = 1$$

$$\begin{cases} T(n) = T(n/2) + c, \ n \geq 2 \\ T(1) = 1 \end{cases}$$

I. Строим мажоранту

Пусть $g(n) = a\log_2(n+1)$

1) Проверяем $g(1) \geq T(1) = 1$

$a\log_2 2 = a \geq 1 \Rightarrow \boxed{a = 1}$?

2) Проверяем $g(n) \geq g\left(\frac{n}{2}\right) + c, \ n \geq 2$

$a\log_2(n+1) \geq a\log_2\left(\frac{n}{2}+1\right) + c$

$a\left(\log_2(n+1) - \log_2\left(\frac{n+2}{2}\right)\right) \geq c$

$a\log_2\frac{2(n+1)}{n+2} \geq c \Rightarrow a \geq \frac{1}{\log_2\left(2 - \frac{2}{n+2}\right)}$ $c > c$

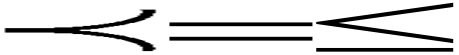
$0 < \log_2\left(2 - \frac{2}{n+2}\right) < 1 \Rightarrow \frac{1}{\log} > 1$

$a \geq 1, \ c = 1$

$\boxed{a = c}$

$g(n) = c\log_2(n+1) \underset{n\to\infty}{\Longrightarrow} g(n) = O(\log n)$

Реализация алгоритма

Y=A 

Проверка «отсортированности» *Y* для корректной работы

Процедура 4.1.2. Binary Search

```
01:  Binary Search Impl(A[1, n], b)
02:        lo = 1, hi = n
03:        while (hi – lo > 2) do
04:              middle = lo + ⌊(hi - lo)/2⌋
05:              if f(A[middle]) = b then
06:                    return middle
07:              if f(A[middle]) < b then lo = middle + 1
08:              if f(A[middle]) > b then hi = middle - 1
09:        return Linear Search(A[lo, hi], b)
```

Процедура 4.1.3. Check Binary Search Prerequisites

```
01:  for index = 1 to n - 1 do
02:        if f(A[index]) > f(A[index - 1]) then
03:              return false
04:  return true
```

Оценочные характеристики

Основное достоинство – время выполнения в худшем случае. Значит, подходит для задач, в которых процедуру поиска требуется повторять многократно для одного и того же множества ключей

Именно такой является ситуация, когда процедуру поиска требуется осуществлять не более чем $\lfloor \log n \rfloor$ раз, при условии, что массив данных не является упорядоченным и ключи не могут быть отсортированы за линейное время

Для применения алгоритма двоичного поиска необходимо применение алгоритма сортировки, трудоемкость которого, по крайней мере, составит O(nlogn)

Левосторонний (правосторонний) двоичный поиск

В ряде случаев требуется определить не только вхождение элемента с заданным ключом в искомый массив данных, но и левую и правую границы, между которыми находятся все элементы из X, удовлетворяющие заданному критерию

Цель левостороннего поиска – нахождение первого вхождения элемента с заданным ключом

Цель правостороннего – нахождение последнего вхождения искомого ключа в заданный массив

Практическое применение

Если не обнаружен элемент с заданным ключом, то должен вернуться индекс *i:*

$$f(x_{i-1}) < y < f(x_i)$$

$$f(x_{i-1}) < y \leq f(x_i)$$

$$f(x_{i-1}) \leq y < f(x_i)$$

Программная реализация

Левосторонний поиск

```
Процедура 4.1.4. Leftmost/Rightmost Binary Search
01:  Leftmost Binary Search Impl(A[1, n], b)
02:        lo = 1, hi = n
03:        while (hi - lo > 2) do
04:              middle = lo + ⌊(hi - lo)/2⌋
05:              if f(A[middle]) < b then
06:                    lo = middle + 1
07:              else
08:                    hi = middle
09:        return Linear Search(A[lo, hi], b)
10:
```

Правосторонний поиск

```
11:  Rightmost Binary Search Impl(A[1, n], b)
12:        lo = 1, hi = n
13:        while (hi - lo > 2) do
14:              middle = lo + ⌊(hi - lo)/2⌋
15:              if f(A[middle]) > b then
16:                    hi = middle
17:              else
18:                    lo = middle + 1
19:        return Linear Search(A[lo, hi], b)
```

Вещественный двоичный поиск

$$f(x) = y,\ \ x \in [a,b],\ \varepsilon - \textit{погрешность}$$

$$O(\log\left(\frac{b-a}{\varepsilon}\right)$$

Программная реализация
на основе идеи дискретного двоичного поиска

Возможно зацикливание?

Задано число итераций (какое?)

```
Процедура 4.1.5. Real Binary Search
01:  Real Binary Search Impl([l, u], y)
02:        lo = l, hi = u
03:        while (lo + ε < hi) do
04:              middle = (lo + hi) / 2
05:              if f(middle) + ε < y then
06:                    lo = middle
07:              else
08:                    hi = middle
09:        return (lo + hi) / 2
```

```
Процедура 4.1.6. Iterative Real Binary Search
01:  Iterative Real Binary Search Impl([l, u], y)
02:        lo = l, hi = u
03:        for iteration = 1 to number of iterations do
04:              middle = (lo + hi) / 2
05:              if f(middle) + ε < y then
06:                    lo = middle
07:              else
08:                    hi = middle
09:        return (lo + hi) / 2
```

1.3 Тернарный поиск
поиск минимума (максимума) функции

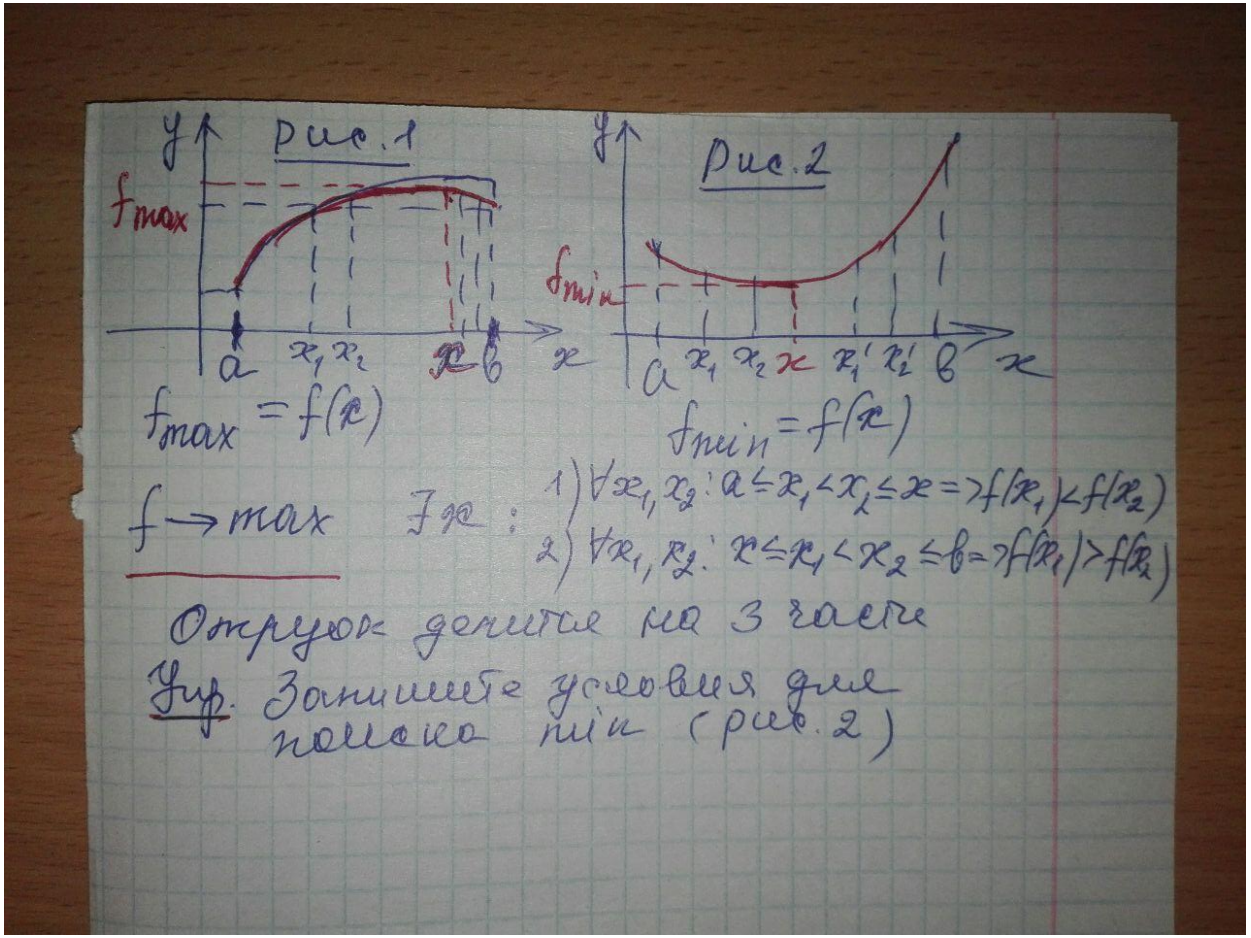
Пусть требуется найти максимум функции f(x) (**свойство функции**?)

Тогда должно существовать некоторое значение x, чтобы:

$$\forall x:\ a \le x_1 < x_2 \le x \Rightarrow f(x_1) < f(x_2)$$

$$\forall x:\ x \le x_1 < x_2 \le b \Rightarrow f(x_1) > f(x_2)$$

Пояснения


y
Рис.1
f_{max}
a
x_1 x_2
x
b
x
$f_{max} = f(x)$
y
Рис.2
f_{min}
a
x_1 x_2 x x_1' x_2'
b
x
$f_{min} = f(x)$
$f \to max$ $\exists x$:
1) $\forall x_1, x_2 : a \le x_1 < x_2 \le x \Rightarrow f(x_1) < f(x_2)$
2) $\forall x_1, x_2 : x \le x_1 < x_2 \le b \Rightarrow f(x_1) > f(x_2)$
Отрезок делится на 3 части
Уф. Запишите условия для поиска min (рис. 2)

РС и сложность

$$T(n) = T\left(\frac{2}{3}n\right) + c,\ n \geq 4,$$

$$T(1) = c_1, T(2) = c_2, T(3) = c_3$$

$$O(\log_{3/2} n)$$

$$O(\log_{3/2} \frac{b-a}{\varepsilon})$$

... условия для
поиска a и b (рис. 2)

$$g(n) = a\log_{\frac{3}{2}} n + b$$

1) $g(1) = b \geq c_1 \Rightarrow b = c_1$

$g(2) = a\log_{\frac{3}{2}} 2 + b \geq c_2 \Rightarrow a \geq \frac{c_2 - b}{\log_{\frac{3}{2}} 2} > \frac{1}{2}(c_2 - c_1)$

$\log_{\frac{3}{2}} 2 < 2$, $1 < \log_{\frac{3}{2}} 2$, $\frac{1}{2} < \frac{1}{\log} < 1$

$g(3) \geq c_3 \Rightarrow a\log_{\frac{3}{2}} 3 + b \geq c_3$

$\log_{\frac{3}{2}} 3 < 3$

$a \geq \frac{c_3 - b}{\log} > \frac{1}{3}(c_3 - b) > \frac{1}{3}(c_3 - c_1)$

$a = \max\{\frac{1}{2}(c_2 - c_1), \frac{1}{3}(c_3 - c_1)\}$

2) $a\log_{\frac{3}{2}} n + b \geq a\log_{\frac{3}{2}}(\frac{2}{3} \cdot n) + b + c = a\log_{\frac{3}{2}}(\frac{2}{3}) + a\log_{\frac{3}{2}} n + b + c$

$0 \geq -a + c \Rightarrow a \geq c = \dots = c$

Программная реализация

Для вещественных элементов

```
Процедура 4.1.7. Real Ternary Search
01:  Real Ternary Search Impl([l, u])
02:        lo = l, hi = u
03:        while (lo + ε < hi) do
04:              loThird = lo + (hi – lo) / 3
05:              hiThird = hi – (hi – lo) / 3
06:              if f(loThird) + ε < f(hiThird) then
07:                    lo = loThird
08:              else
09:                    hi = hiThird
10:        return (lo + hi) / 2
```

Для дискретных элементов

```
Процедура 4.1.8. Ternary Search
01:  Ternary Search Impl(A[1, n])
02:        lo = 1, hi = n
03:        while (hi – lo > 3) do
04:              loThird = lo + (hi – lo) / 3
05:              hiThird = hi – (hi – lo) / 3
06:              if f(A[loThird]) < f(A[hiThird]) then
07:                    lo = loThird
08:              else
09:                    hi = hiThird
10:        return Linear Search(A[lo, hi])
```

Оценочные характеристики

Недостатком тернарного поиска является сравнение значений функции в двух точках, определяющих границы каждой из третьих

Рассмотрим технику, которая позволяет получить значительный прирост в скорости путем «обсчета» каждой из рассматриваемых точек только один раз

Метод золотого сечения

Скрытую константу в оценке времени работы тернарного поиска можно значительно уменьшить, деля отрезок в отношении «золотого сечения», а не на 3 равные части

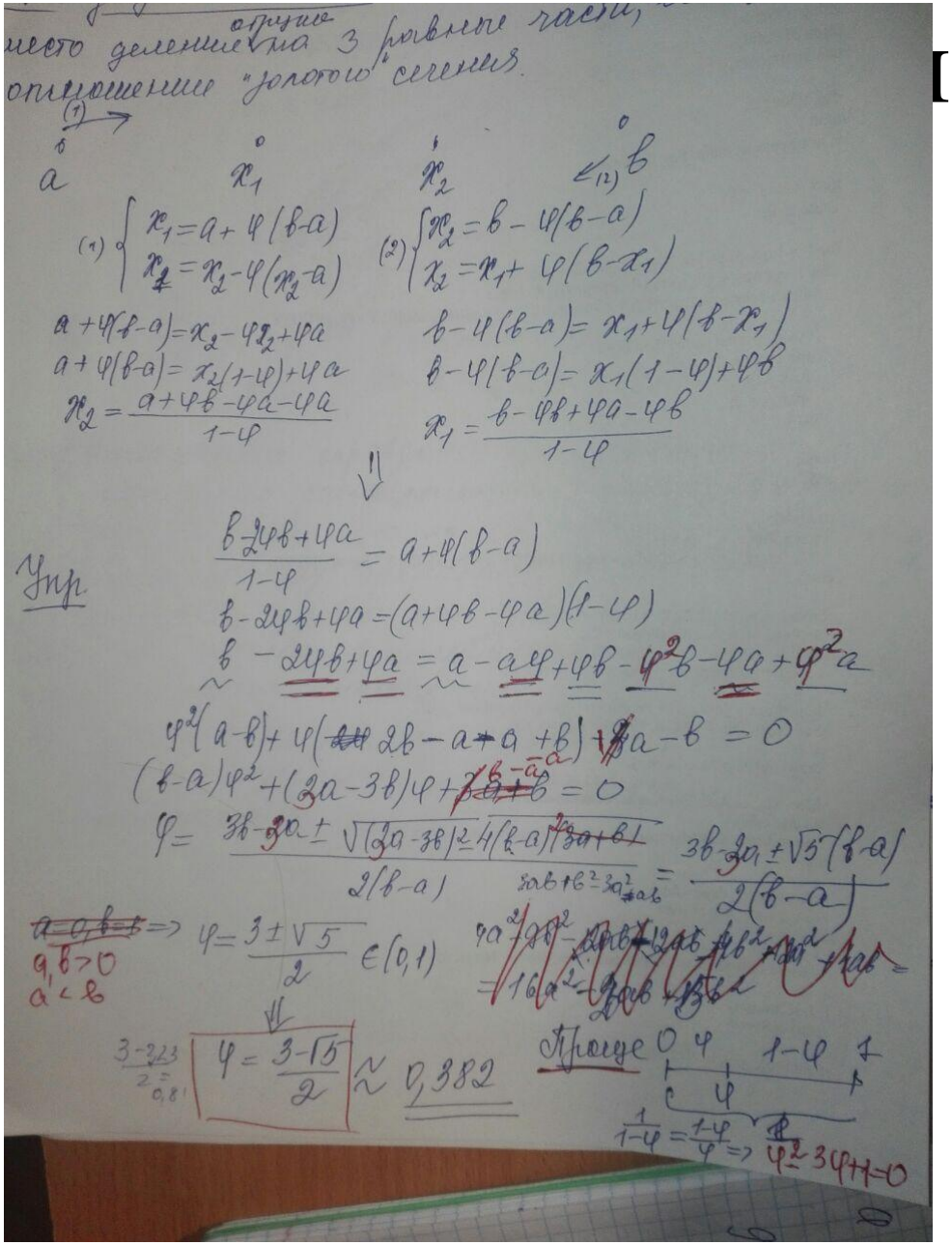
Постановка задачи

Пусть требуется найти максимум выпуклой функции f(x) на заданном отрезке [l,u]

Будем делить отрезок точками так, чтобы выполнялись следующие соотношения:

$$\begin{cases} x_1 = l + \varphi \cdot (u - l), & x_1 = x_2 - \varphi \cdot (x_2 - l), \\ x_2 = u - \varphi \cdot (u - l), & x_2 = x_1 + \varphi \cdot (u - x_1). \end{cases}$$

М ления


место деления на 3 равные части,
отрезки
с помощью "золотого" сечения.
(1)
(2)
a
x1
x2
b
(1) x1 = a + φ(b-a)
x2 = x2 - φ(x2-a)
(2) x2 = b - φ(b-a)
x2 = x1 + φ(b-x1)
a + φb - φa = x2 - φx2 + φa
a + φ(b-a) = x2(1-φ) + φa
x2 = (a + φb - φa - φa)/(1-φ)
b - φ(b-a) = x1 + φ(b-x1)
b - φ(b-a) = x1(1-φ) + φb
x1 = (b - φb + φa - φb)/(1-φ)
Упр.
(b - 2φb + φa)/(1-φ) = a + φ(b-a)
b - 2φb + φa = (a + φb - φa)(1-φ)
b - 2φb + φa = a - aφ + φb - φ²b - φa + φ²a
φ²(a-b) + φ(2b - a + a + b) + a - b = 0
(b-a)φ² + (3a - 3b)φ + b - a = 0
φ = (3b - 3a ± √((3a-3b)² - 4(b-a)(3a-b)))/(2(b-a))
= (3b - 3a ± √5(b-a))/(2(b-a))
a < b
a, b > 0
φ = (3 ± √5)/2 ∈ (0,1)
φ = (3 - √5)/2 ≈ 0,382
Лемма
1/(1-φ) = (1-φ)/φ ⇒ φ² + 3φ - 1 = 0

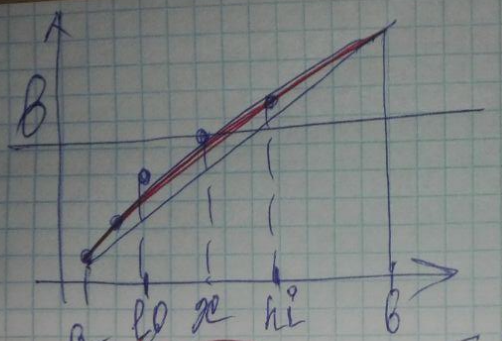
Процедура 4.1.9. Gold Section Ternary Search

```
01:  Gold Section Ternary Search Impl([l, u])
02:        φ = 2 − (1+√5)/2
03:        lo = l, hi = u
04:        if (lo + ε < hi) do
05:              loThird = lo + φ·(hi − lo)
06:              hiThird = hi - φ·(hi − lo)
07:              fLoThird = f(loThird)
08:              fHiThird = f(hiThird)
09:              do
10:                    if fLoTthird + ε < fHiThird then
11:                          lo = loThird
12:                          loThird = hiThird
13:                          hiThird = hi - φ·(hi − lo)
14:                          fLoThird = fHiThird
15:                          fHiThird = f(hiThird)
16:                    else
17:                          hi = hiThird
18:                          hiThird = loThird
19:                          loThird = lo + φ·(hi − lo)
20:                          fHiThird = fLoThird
21:                          fLoThird = f(loThird)
22:              while (lo + ε < hi)
23:        return (lo + hi) / 2
```

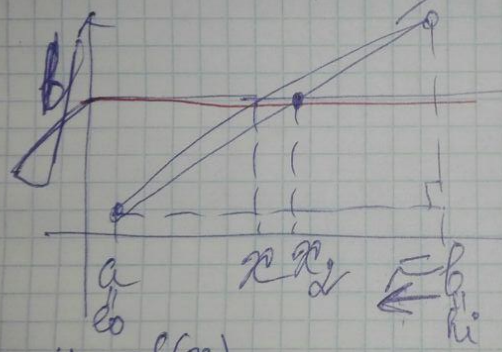
1.4 Интерполяционный поиск

Интерполяционный поиск, как и двоичный, используется для поиска данных в упорядоченных массивах

Их отличие состоит в том, что вместо деления области поиска на две примерно равные части, интерполяционный поиск производит оценку новой области поиска по расстоянию между ключом и текущими значениями граничных элементов



$y = f(x)$



$$\frac{f(b)-f(a)}{b-a} = \left(\frac{x-a}{y-f(a)}\right)^{-1}$$

$$x = a + \frac{f(b)-f(a)}{b-a}(y-f(a))$$

$$x = a + (y-f(a))\frac{b-a}{f(b)-f(a)}$$

$y = f(x)$ ← найти!

заданной ключ.

Программная реализация

```
Процедура 4.1.10. Interpolation Search
01:  Interpolation Search Impl(A[1, n], b)
02:        lo = 1, hi = n
03:        while (hi – lo > threshold) do
04:              x = lo + (b – A[lo]) * (hi – lo) / (A[hi] – A[lo])
05:              if A[x] = b then
06:                    return x
07:              if A[x] < b then lo = x + 1
08:              if A[x] > b then hi = x - 1
09:        return Binary Search(A[lo, hi], b)
```

Оценочные характеристики

Использование интерполяционного поиска оправдано в том случае, когда данные в массиве, в котором происходит поиск, распределены достаточно равномерно. Тогда трудоемкость приведенной процедуры может составить O(loglogn), что асимптотически лучше трудоемкости двоичного поиска

Если данные распределены не достаточно равномерно, а, например, экспоненциально, то трудоемкость интерполяционного поиска может составить O(n) в худшем случае

На практике интерполяционный поиск обычно используется для поиска значений в больших файлах данных, причем поиск продолжается до тех пор, пока не будет достигнут некий пороговый диапазон, в котором находится искомое значение

После этого обычно применяется двоичный либо линейный поиск

ПРИМЕР использования

Пусть есть словарь, состоящий из всех слов русского языка, упорядоченных по алфавиту

Задача состоит в том, чтобы достаточно быстро найти некоторое слово a1a2…an

Человек начнет поиск со страницы, которая содержит слова с префиксом искомого слова

В зависимости от места остановки поиска дальнейшие действия могут пропустить либо небольшое количество страниц, либо количество страниц, существенно большее половины допустимого интервала

Таким образом, основное отличие интерполяционного поиска от рассмотренных выше методов состоит в том, что в случае равномерно распределенных данных интерполяционный поиск учитывает разницу между «немного больше» и «существенно больше», тем самым позволяя сократить время поиска искомых данных до минимума

2. Организация поиска на деревьях

- Рассматриваются некоторые разновидности поисковых деревьев: бинарное поисковое дерево, АВЛ-дерево, 2-3-дерево, красно-черное дерево и самоперестраивающееся (splay) дерево
- Для каждого из видов деревьев приводятся базовые операции и оценивается их трудоемкость
- Для сбалансированных поисковых деревьев при описании базовых операций рассматриваются оценки трудоемкости процедуры поддержки инвариантов

2.1 Бинарные поисковые деревья

Пусть каждой вершине (узлу) бинарного дерева соответствует некоторое ключевое значение (например, целое число)

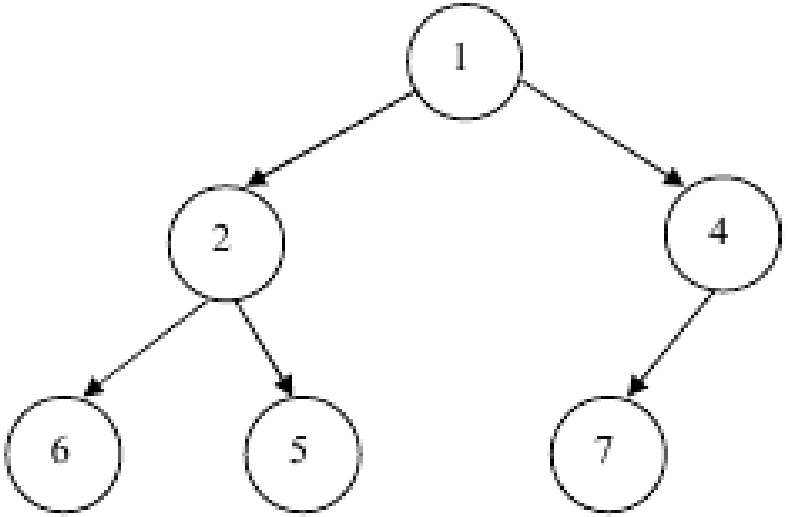
Определение

Бинарное дерево называется **деревом поиска**, если оно организовано так, что для каждой вершины v справедливо утверждение, что все ключи в левом поддереве вершины v меньше ключа вершины v, а все ключи в правом поддереве – больше

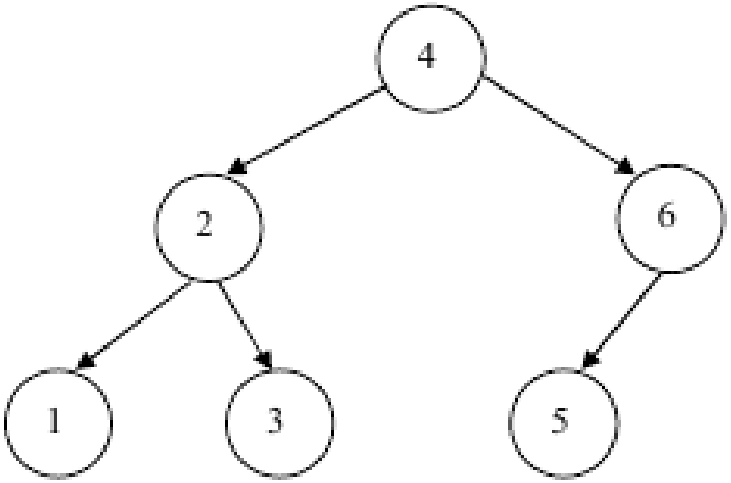
Для простоты будем считать, что ключ и данные – это одно и то же, и ключ имеет целочисленный тип

Сравните

Непоисковое дерево



Поисковое дерево



Способы обхода вершин

Многие алгоритмы, работая с бинарными корневыми деревьями, посещают каждую вершину дерева ровно один раз и в некотором порядке

Пусть дерево задано списковой структурой и известна процедура, которая выполняет некоторые необходимые действия при посещении вершины дерева, на которую ссылается указатель v

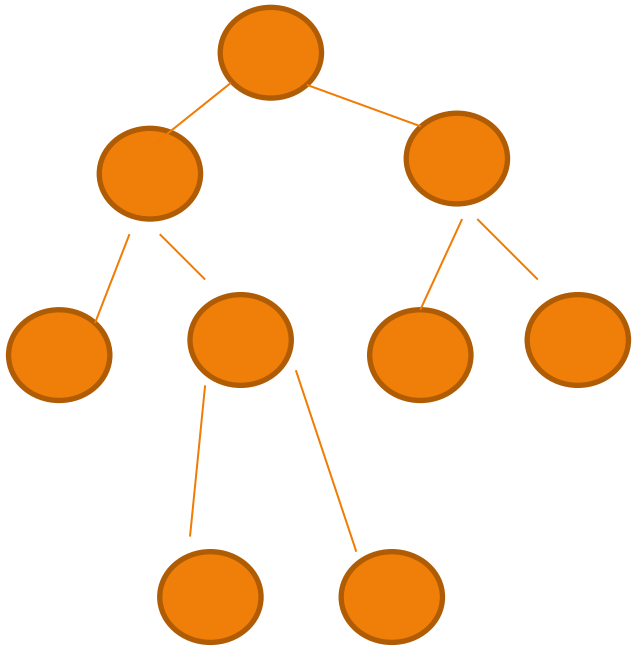
1) Прямой порядок обхода (сверху вниз)

Корень некоторого дерева посещается раньше, чем его поддеревья

Если после корня посещается его левое (правое) поддерево, то обход называется прямым левым (правым) обходом

Пример

- Корень
- Обход левого поддерева
- Обход правого поддерева



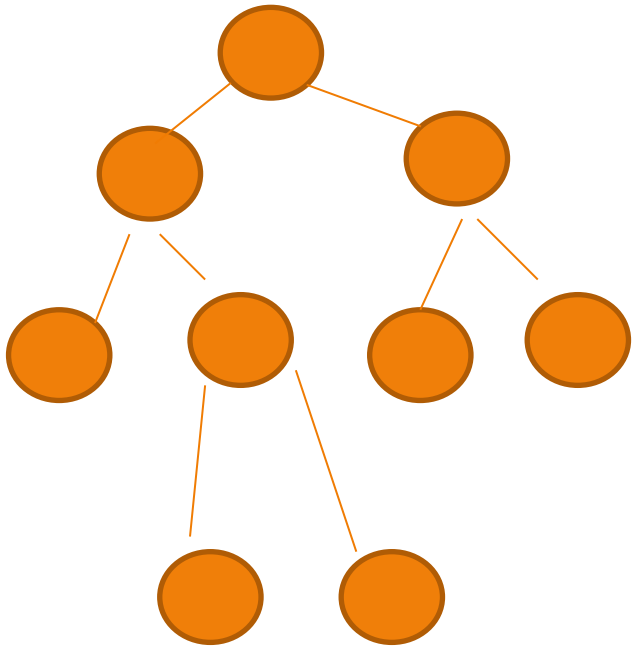
2) Обратный порядок обхода (снизу вверх)

Корень дерева посещается после его поддеревьев

Если сначала посещается левое (правое) поддерево корня, то обход называется обратным левым (правым) обходом

Пример

- Обход левого поддерева
- Обход правого поддерева
- Корень



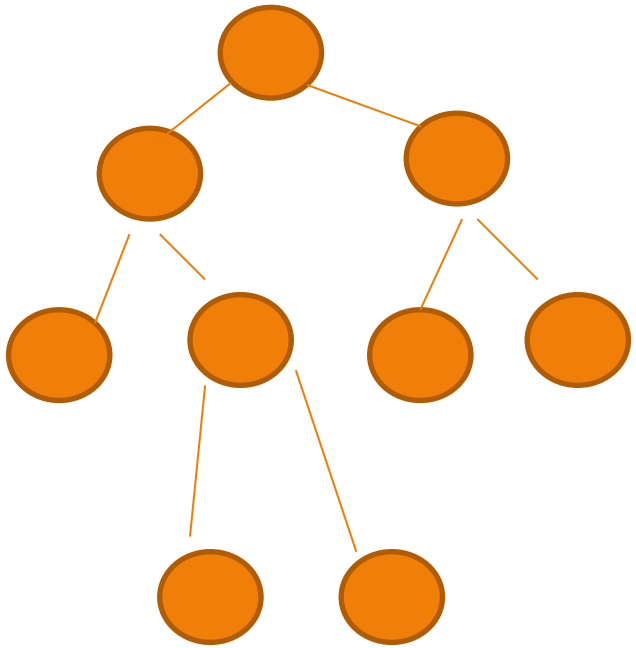
3) **Внутренний порядок обхода** (слева направо или справа налево, симметричный, поперечный)

Корень посещается после одного из его поддеревьев

Если корень посещается после его левого (правого) поддерева, то обход называется левым (правым) внутренним обходом

Пример

- Обход левого поддерева
- Корень
- Обход правого поддерева



Базовые операции над данными, представленными бинарными поисковыми деревьями

1)Поиск по ключу (поиск максимального, минимального элементов)

2)Добавление элемента с заданным ключом

3)Удаление элемента с заданным ключом

1) Поиск узла по ключу

- **Алгоритм** следует из определения дерева поиска

1.На каждом шаге значение искомого ключа сравнивается со значением ключа текущего узла, начиная с корня

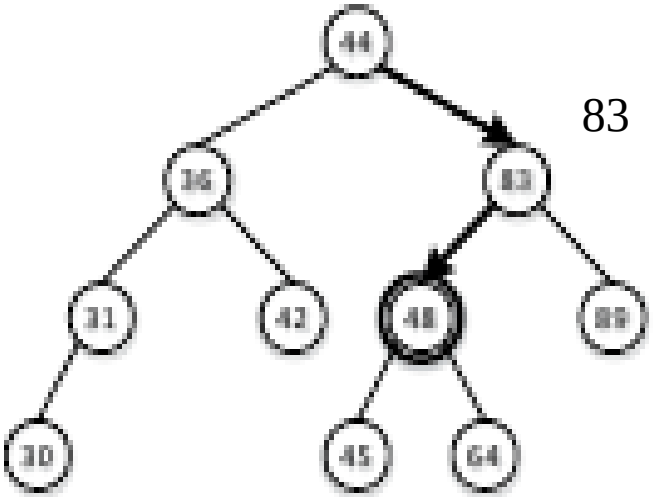
2.Если значение искомого ключа меньше, то поиск продолжается в левом поддереве, если *больше* — то в *правом*

3. Процесс продолжается, пока не будет найден узел с искомым ключом или пока поиск не достигнет того узла, ниже которого этот узел не может находиться

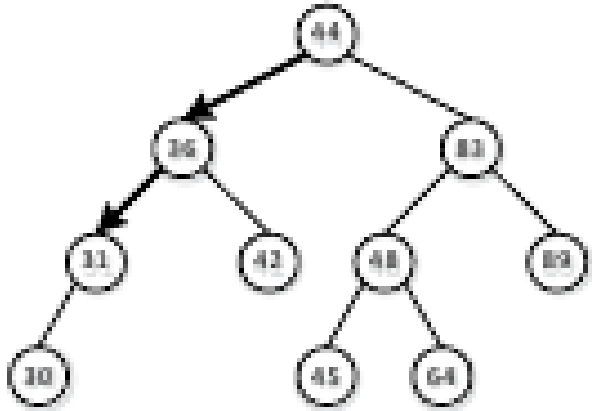
4.Если при поиске обнаруживается, что узел далее надо искать, например, в правом поддереве, а оно пусто, то искомого ключа в дереве нет

Пример

Ключ 48



Ключ 32



Рекурсивная реализация

```
TREE-SEARCH(T, k)

/* На вход подается дерево T, в котором производится поиск, и k
– значение ключа. Возвращается узел дерева, в котором находит-
ся искомый ключ, или NULL, если узла с искомым ключом в дереве
нет. */
1 x ← root[T]
2 if x = NULL или k = key[x] then /* нерекурсивная ветвь */
3         return x
4 if k < key[x] then
5         return TREE-SEARCH(left[x], k) /* вызываем функцию для
левого поддерева */
6 else
7         return  TREE-SEARCH(right[x],  k)  /*  вызываем  функцию
для правого поддерева */
```

Итеративная реализация

(более эффективная по времени и памяти)

```
INTERATIVE-TREE-SEARCH (T, k)

/* На вход подается дерево T, в котором производится поиск, и k
– значение ключа. Возвращается узел дерева, в котором находит-
ся искомый ключ, или NULL, если узла с искомым ключом в дереве
нет. */
1 x ← root[T]
2 while x ≠ NULL и k ≠ key[x] do
/* Просматриваем текущий узел, спускаясь по дереву вниз, пока не
найдем искомый ключ или не дойдем до пустого поддерева. */
3       if k < key[x] then
4               x ← left[x] /* присваиваем x его левого сына */
5       else
6               x ← right[x] /* присваиваем x его правого сына */
7 return x
```

Трудоемкость поиска узла

При поиске узла на каждой итерации происходим спуск на один уровень вниз, поэтому количество шагов, ограничено сверху высотой этого дерева

Время поиска в худшем случае – O(h),

где h – высота дерева

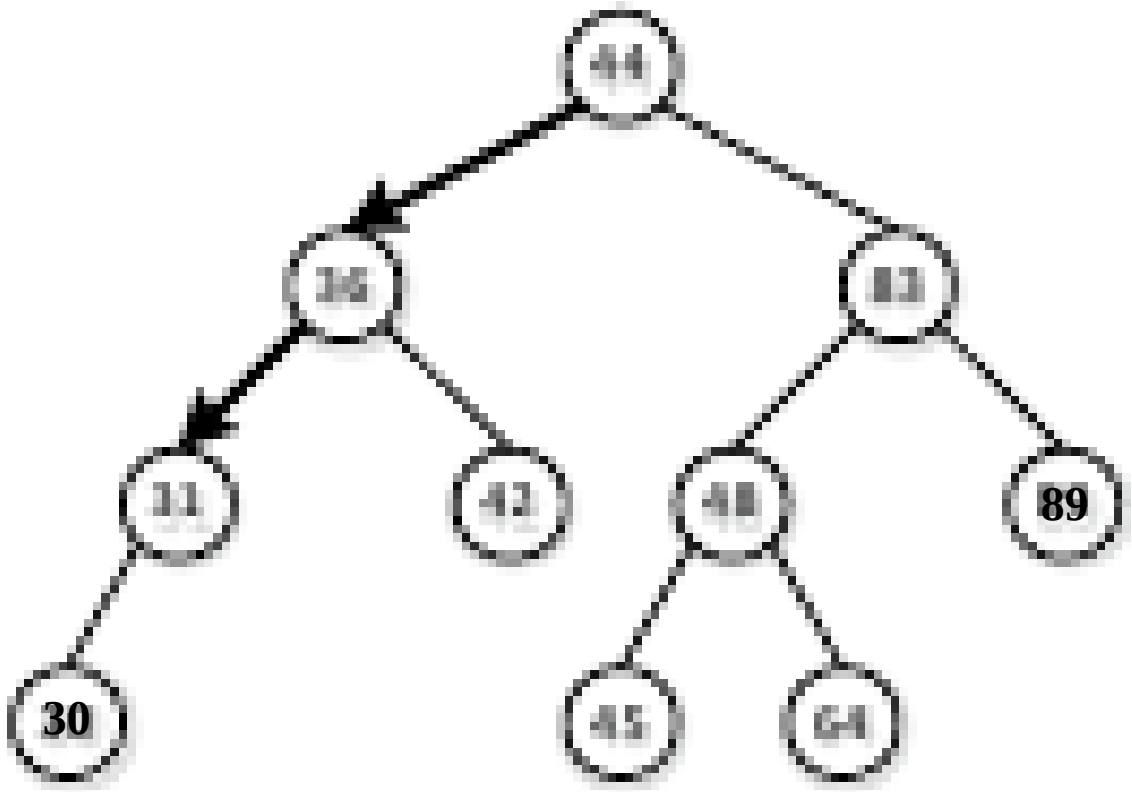
Поиск узла с минимальным (максимальным) ключом

Если требуется найти в дереве узел с минимальным (максимальным) ключом, то это будет самый левый (самый правый) узел в дереве

Найти этот узел можно, двигаясь от корня только налево (направо) до тех пор, пока у текущего узла существует левый (правый) сын

Время поиска – O(h), h – высота дерева

Пример



Реализация

Поиск минимума

```
TREE-MINIMUM(T)
/* На вход подается дерево T, и возвращается узел с минимальным
значением ключа в дереве. */
1 x ← root[T]
2 while left[x] ≠ NULL do /* ищем самый левый узел в дереве */
3         x ← left[x]
4 return x
```

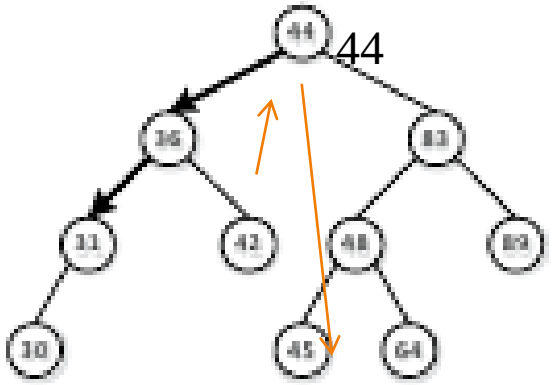
Поиск максимума

```
TREE-MAXIMUM(T)
/* На вход подается дерево T, и возвращается узел с максималь-
ным значением ключа в дереве. */
1 x ← root[T]
2 while right[x] ≠ NULL do /* ищем самый правый узел в дереве */
3         x ← right[x]
4 return x
```

Поиск предшествующего и последующего элементов узла x

Если правое поддерево узла x с ключом k не пусто, то последователем узла x будет самый левый узел его правого поддерева, потому как это он имеет минимальный ключ среди всех ключей, больших k, имеющихся в дереве.

Если правое поддерево узла x пусто, то надо искать, поднимаясь наверх, первого предка, для которого узел x окажется в левом поддереве



Реализация

```
TREE-SUCCESSOR(T, x)
/* На вход подается дерево T и узел x, и возвращается узел, который является его последователем. Если у узла нет последователя, то есть, узел является самым правым в дереве, то возвращается NULL. */
1 if right[x] ≠ NULL then
/* Если у узла есть правое поддерево, то возвращаем самый левый узел правого поддерева. */
2         return TREE-MINIMUM(right[x])
/* Если у узла нет правого поддерева, то поднимаемся по родителям наверх, пока не найдем того, для которого рассматриваемый узел – левый сын. */
3 y ← parent[x]
4 while y ≠ NULL и x = right[y] do
5         x ← y
6         y ← parent[y]
7 return y
```

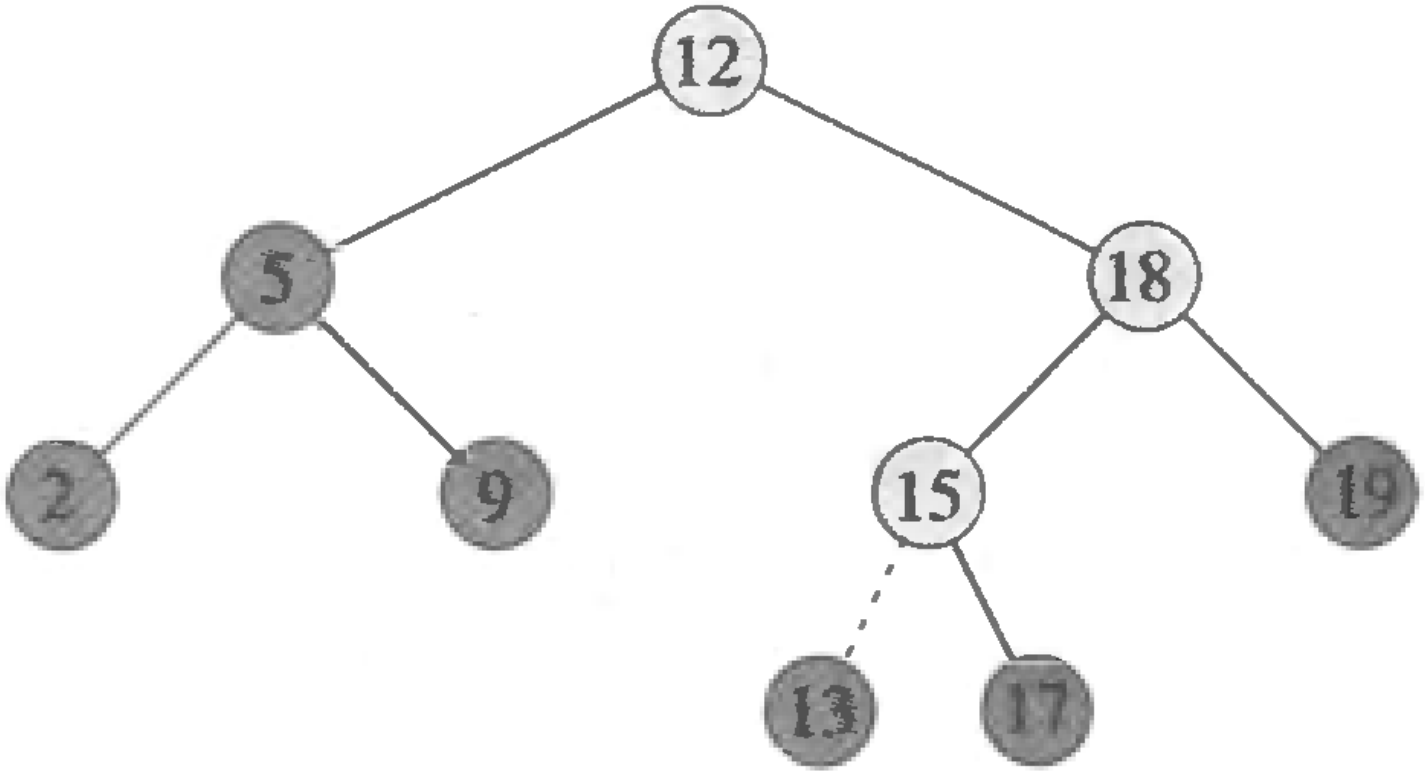
2) Добавление элемента с ключом *x*

Данную процедуру часто называют **поиском по дереву с включением**, так как для добавления некоторого элемента *x* необходимо сначала определить его местоположение в дереве

Если вершина с ключевым значением *x* уже существует, то добавление элемента не производится, так как в бинарном поисковом дереве все ключи различны

В противном случае поиск остановится на узле, к которому впоследствии будет присоединяться узел слева или справа в зависимости от значения его ключа

Пример



Реализация

```
TREE-SEARCH-INEXACT(T, k)

/* На вход подается дерево T, в котором производится поиск, и k
– значение ключа. Возвращается узел дерева, в котором находит-
ся искомый ключ, или тот узел, на котором остановился поиск. */
1 y ← NULL
2 x ← root[T]
3 while x ≠ NULL и k ≠ key[x] do /* продолжаем поиск до тех пор,
пока не найдем ключ или не дойдем до пустого дерева */
4       y ← x
5       if k < key[x] then
6               x ← left[x]
7       else
8               x ← right[x]
9  if x ≠ NULL /* ключ найден */
10      y ← x /* кладем в y узел x с ключом k вместо его родителя */
11 return y
```

```
TREE-INSERT(T, z)

/* На вход подается дерево T и узел z, который надо добавить в
дерево. */
1   y ← TREE-SEARCH-INEXACT(T, key[z])
2   parent[z] ← y
3 if y = NULL then /* если дерево было пусто, то z станет корнем */
4       root[T] ← z
/* Присоединяем z к y слева или справа в зависимости от значения
ключа. */
5 else if key[z] < key[y] then
6       left[y] ← z
7 else
8       right[y] ← z
```

Трудоемкость вставки

Требуется O(h) шагов для выполнения поиска и еще O(1) (константное значение) шагов для выполнения непосредственно операции добавления элемента, поэтому итоговое время выполнения – O(h)

3) Удаление элемента с заданным ключом *x*

Процедура удаления элемента сначала определяет вершину дерева *v* с ключевым значением *x*

Если найденная вершина имеет не более одного сына, то ее удаление из дерева производится непосредственным удалением вершины (как это традиционно выполняется в списковой структуре, когда для удаляемого элемента известен предыдущий и следующий элементы)

Трудности с удалением

У удаляемой вершины *v* *два* сына.

Данный случай сводится к предыдущему случаю

Для этого ключевое значение *x* вершины *v* нужно заменить либо на максимальный ключ в левом поддереве вершины *v* (левое удаление), либо на минимальный ключ в правом поддереве вершины *v* (правое удаление), а затем происходит удаление соответствующей вершины.

Каждая из указанных вершин имеет не более одного сына

Иллюстрация случаев удаления

1) Узел – лист (нет сыновей)

Узел удаляется, и соответствующее поддерево его родителя становится пустым

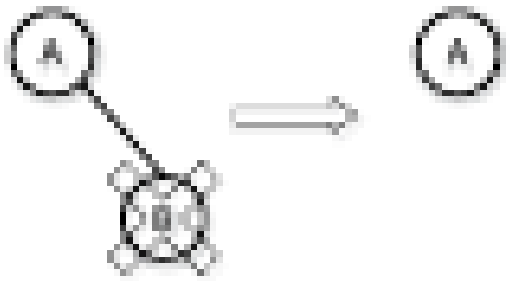


Иллюстрация случаев удаления

2) У узла **один** сын

Узел удаляется, а его сын переходит к его родителю

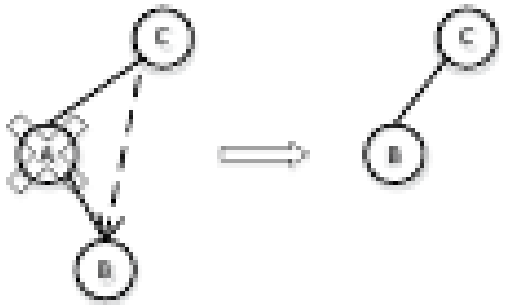
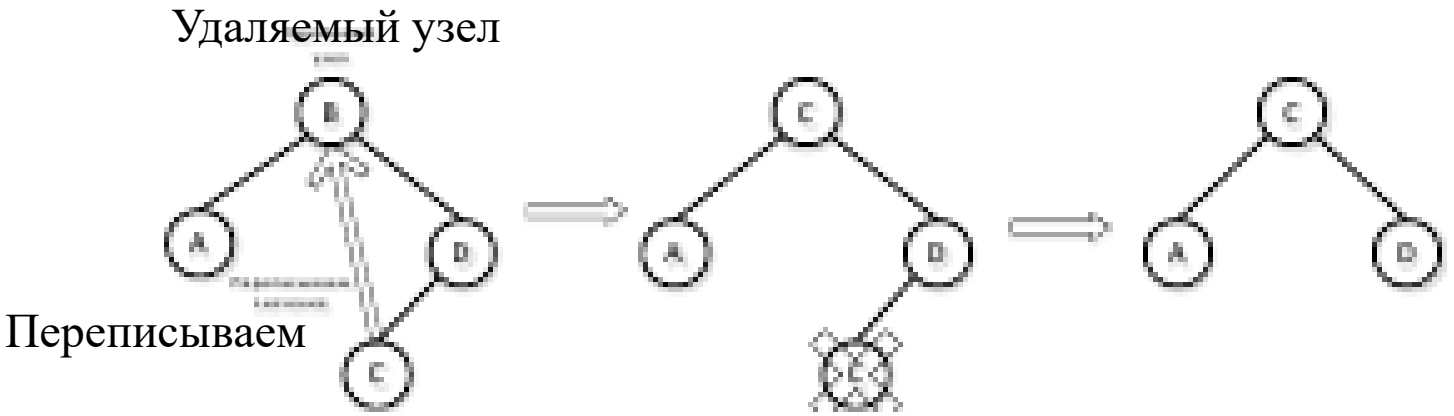


Иллюстрация случаев удаления

3) У узла два сына

Пусть B – удаляемый узел. Ищем его последователя C – узел с минимальным ключом в правом поддереве удаляемого узла. Переносим ключ узла C в узел B и сводим задачу к удалению узла C

Эта процедура является корректной, потому что после удаления узла B его место должен занять как раз его последователь



Балансировка бинарных деревьев поиска

Трудоемкость выполнения базовых операций зависит от высоты дерева h

Если бинарное поисковое дерево вытянуто в линейный список, то $h = \Omega(n)$

где n – количество вершин дерева

Поэтому особый интерес вызывают такие инварианты (свойства) поисковых деревьев, которые обеспечивают высоту деревьев h =O(logn), и при этом трудоемкость поддержания этих инвариантов не превышает O(logn), т.е. сбалансированность

Как построить дерево поиска минимальной высоты?

Если набор ключей известен заранее, то его надо упорядочить

Корнем поддерева становится узел с ключом, значение которого – медиана этого набора

Полный набор ключей не всегда известен заранее. Если ключи поступают по очереди, то построение дерева поиска будет зависеть от порядка их поступления, и дерево «вытянется» в высоту

Проблема поддержания свойства сбалансированности

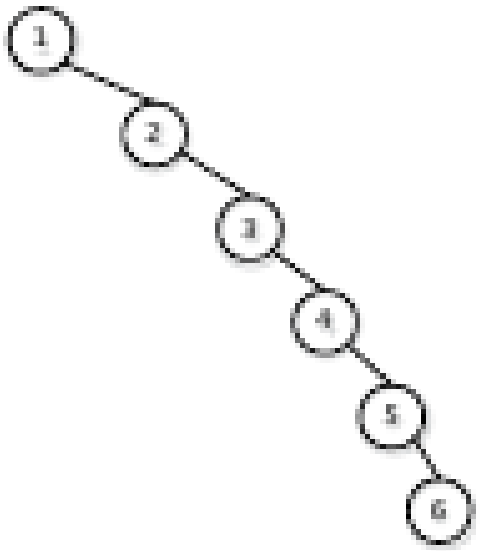
При добавлении очередного узла дерево понадобится перестраивать для уменьшения его высоты и на том же наборе узлов

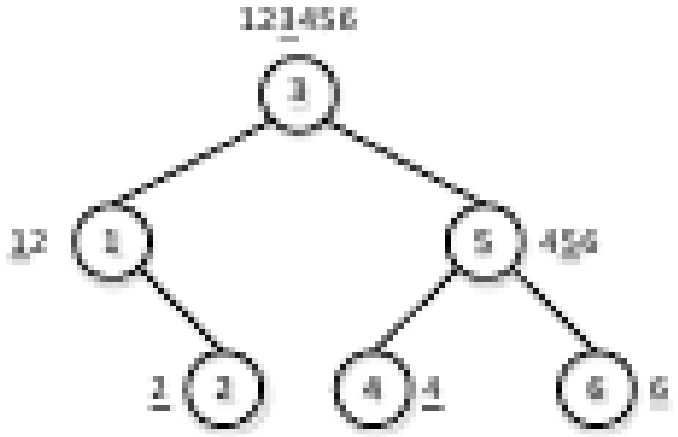
Идеальную балансировку поддерживать сложно

Если при добавлении очередного узла количество узлов в левом и правом поддеревьях какого-либо узла дерева станет различаться более, чем на 1, то дерево не будет являться идеально сбалансированным, и его надо будет перестраивать, чтобы восстановить свойства идеально сбалансированного дерева поиска

Поэтому обычно требования к сбалансированности дерева менее строгие

Деревья из одного набора ключей





3. АВЛ-деревья

АВЛ-дерево – это бинарное поисковое дерево, у которого для каждой вершины v выполняется следующее свойство: высота поддерева, корень которого – левый сын вершины v, отличается не более, чем на единицу, от высоты поддерева, корень которого – правый сын вершины v

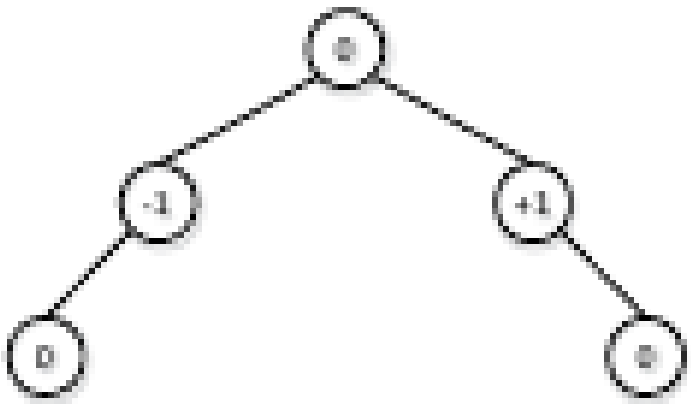
Показатель баланса

В каждом узле АВЛ-дерева, помимо ключа, данных и указателей на левое и правое поддеревья (левого и правого сыновей), хранится показатель баланса – разность высот правого и левого поддеревьев

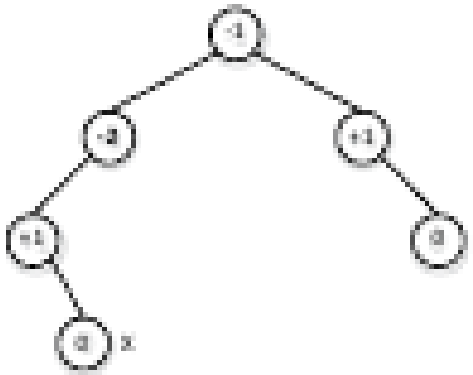
В некоторых реализациях этот показатель может вычисляться отдельно в процессе обработки дерева тогда, когда это необходимо

Пример

АВЛ-дерево



Не АВЛ-дерево



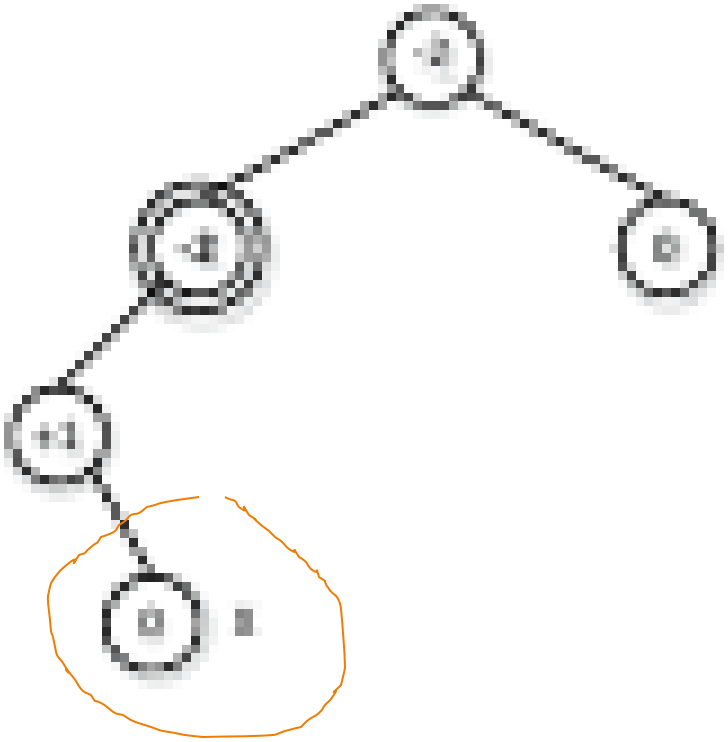
3.1 Вставка узла в АВЛ-дерево

1)Узел добавляется в дерево с помощью стандартного алгоритма вставки в двоичное дерево поиска

2)При добавлении нового узла разбалансировка может произойти сразу в нескольких узлах, но все они будут лежать на пути от этого добавленного узла к корню

3)Перестраивать надо поддерево с корнем в том из этих узлов, который является ближайшим к добавленному

Добавление узла x



Общее правило для добавляемых узлов, приводящих к разбалансировке

Чтобы найти корень поддерева, которое понадобится перестраивать, надо подниматься вверх по дереву от вновь добавленного узла до тех пор, пока не найдется первый (**опорный**) узел, в котором нарушена сбалансированность

После этого провести процедуру перестройки поддерева с корнем в этом узле с целью восстановления его сбалансированности

Остальная часть дерева останется в прежнем виде

Все дерево также станет сбалансированным — показатель баланса не будет превышать 1 по модулю во всех узлах дерева

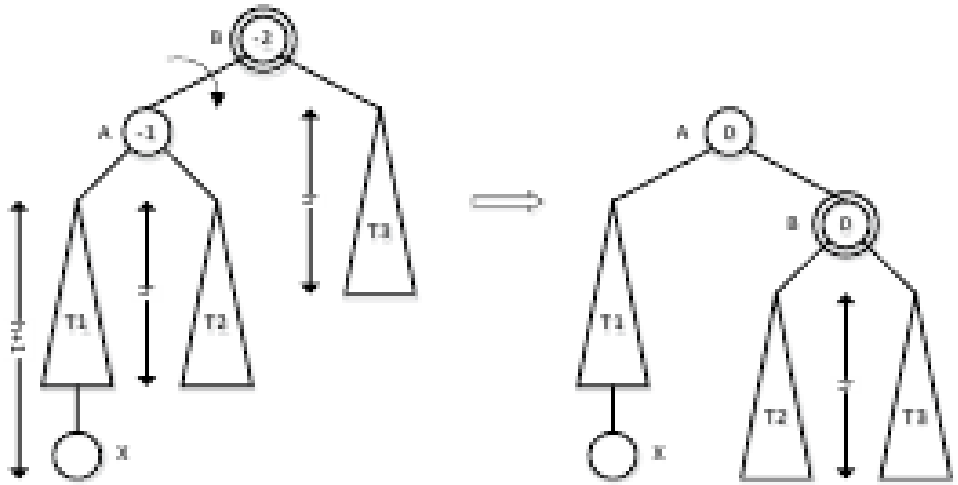
Классификация по случаям

В зависимости от того, в какое поддерево опорного узла был добавлен новый узел, рассмотрим четыре случая, которые можно разбить на две пары симметричных друг другу вариантов

В каждом из них баланс восстанавливается с помощью одного или двух поворотов

1) Добавление в левое поддерево левого сына опорного узла

Балансировка – произвести правый поворот (R): опорный узел (B) поворачивается направо относительно своего левого сына (A)



Реализация

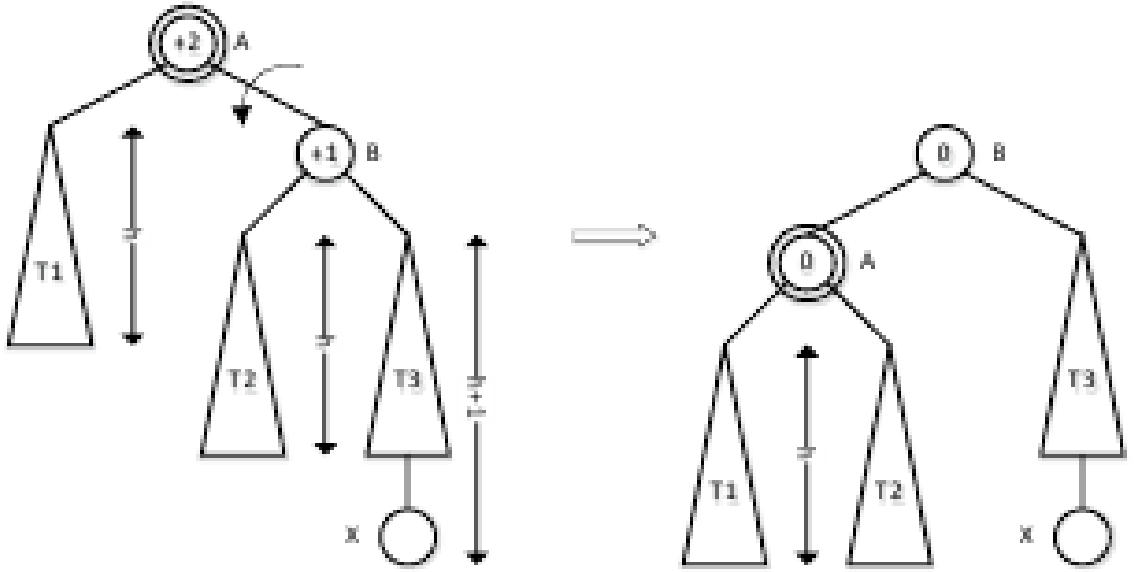
```
TREE-ROTATE-R(T, x)

/* На вход подается дерево T и опорный узел x. */
1   y ← left[x]
/* В строках 2–4 присоединяем T2 к x слева. */
2   left[x] ← right[y]
3   if right[y] ≠ NULL then
4         parent[right[y]] ← x
/* В строках 5–11 отсоединяем x от его родителя и присоединяем
y вместо x. */

5   parent[y] ← parent[x]
6   if parent[x] = NULL then
7         root[T] ← y
8   else if x = right[parent[x]] then
9         right[parent[x]] ← y
10  else
11        left[parent[x]] ← y
/* Соединяем x и y. */
12  right[y] ← x
13  parent[x] ← y
```

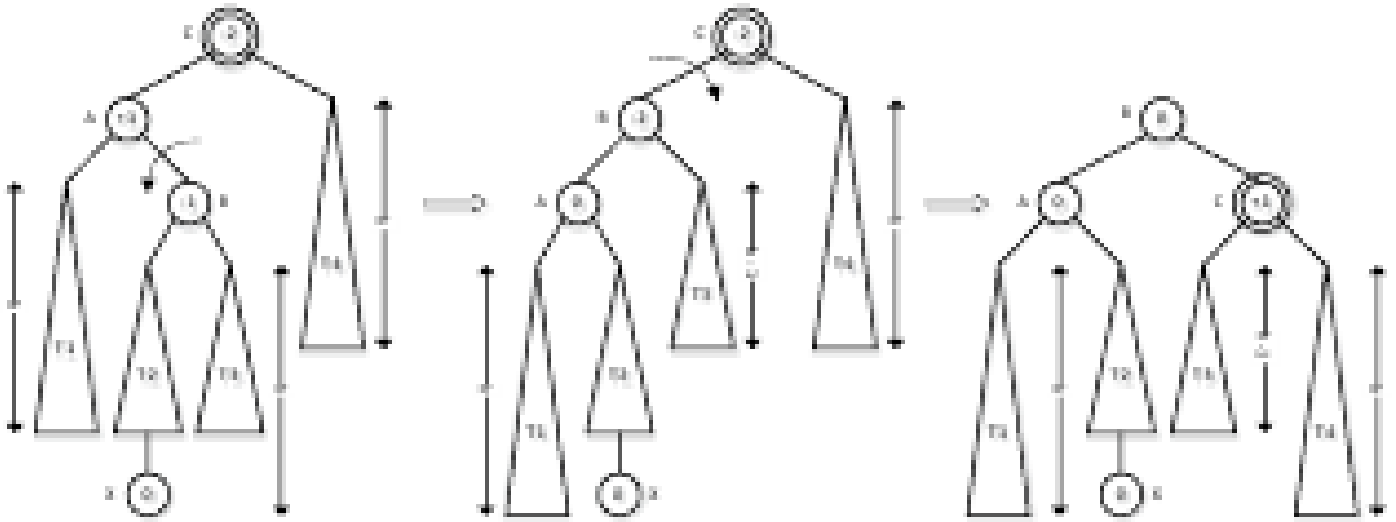
2) Добавление в правое поддерево правого сына опорного узла

- Симметричная операция случаю 1:



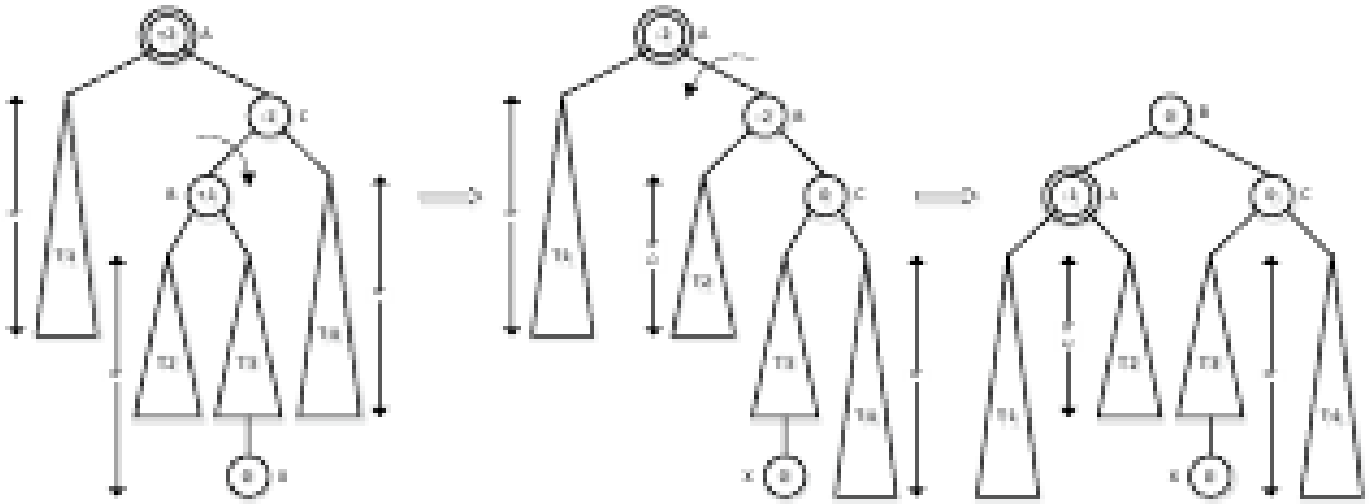
3) Добавление в правое поддерево левого сына опорного узла

Необходимо произвести двойной поворот — налево, потом направо (LR): сначала левый сын A опорного узла C поворачивается налево относительно своего правого сына B, а затем опорный узел C поворачивается направо относительно своего нового левого сына B



4) Добавление в левое поддерево правого сына опорного узла

Случай, симметричный рассмотренному



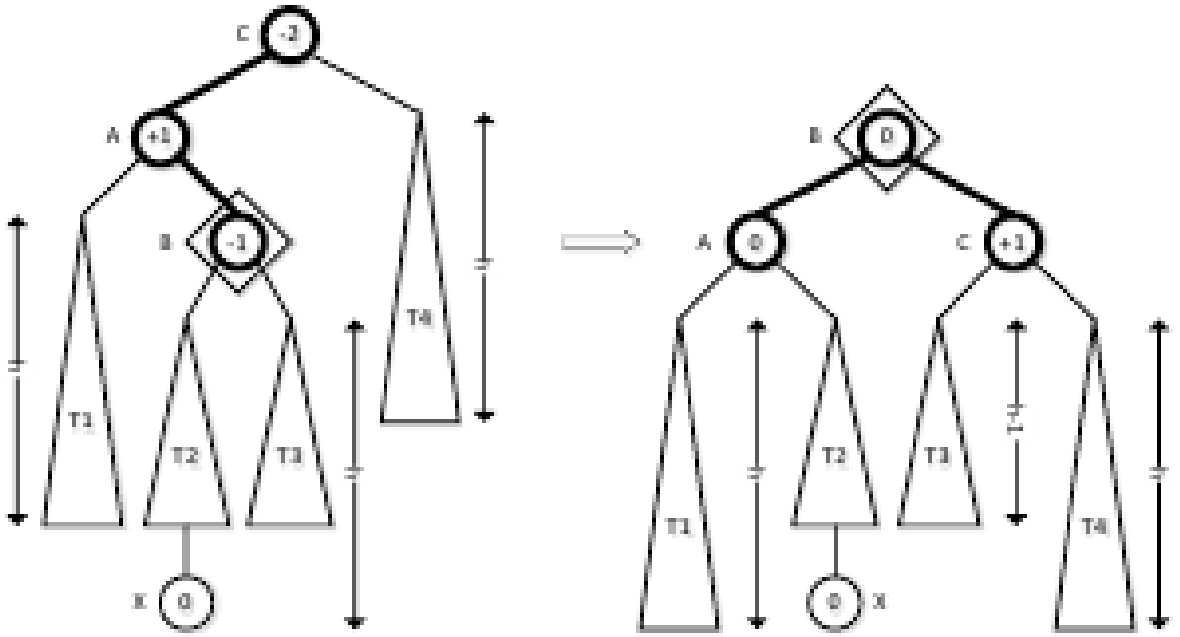
Вращение комбинации трех узлов

Эта комбинация включает в себя опорный узел и корни тех поддеревьев, куда добавился новый узел, нарушивший баланс

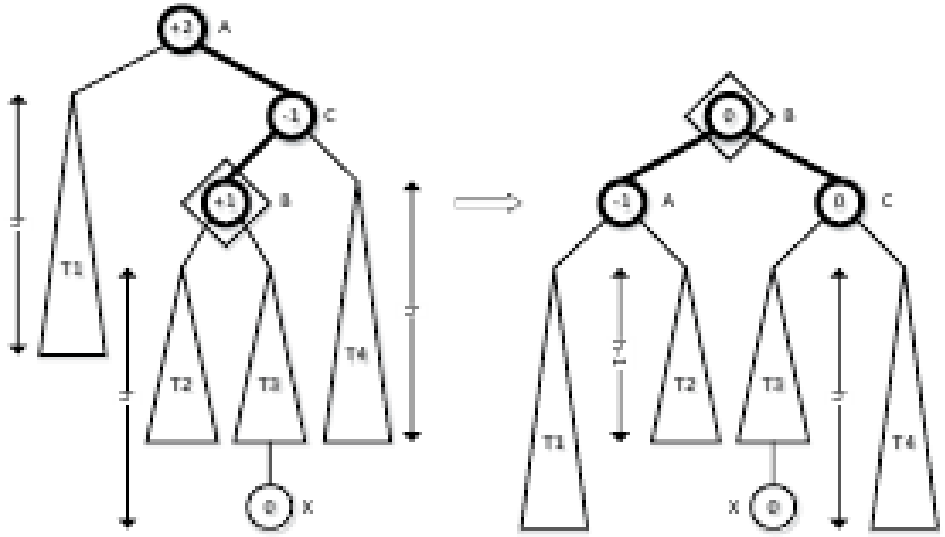
Если добавляем узел в левое поддерево правого сына опорного узла, то комбинация будет состоять из опорного узла, правого сына опорного узла и левого сына правого сына опорного узла

Тот узел, который был самым нижним в этой комбинации, после двойного поворота становится самым верхним, независимо от того, выполняется правый-левый поворот или левый-правый

LR


C
-2
A
+1
B
-1
T1
T2
T3
T4
X
0
B
0
A
0
C
+1
T1
T2
T3
T4
0
X

RL


+2
A
-1
C
+1
B
T1
T2
T3
T4
0
X
0
B
-1
A
0
C
T1
T2
T3
T4
X
0

Резюме по правилу поворотов

Одинарный поворот

Добавление нового узла, приводящее к разбалансировке, происходит в левое поддерево левого сына опорного узла или в правое поддерево правого сына опорного узла, т.е. если стороны сына и внука опорного узла одноименны

Двойной поворот

Добавление происходит в правое поддерево левого сына опорного узла или в левое поддерево правого сына опорного узла, т.е. стороны разноименны

Мнемоническое правило по направлению поворота

- добавление в левое поддерево левого сына опорного узла – правый (R);
- добавление в правое поддерево левого сына опорного узла – левый-правый (LR);
- добавление в левое поддерево правого сына опорного узла – правый-левый (RL);
- добавление в правое поддерево правого поддерева сына опорного узла – левый (L).

Вывод

Поворот производится в противоположную сторону

Визуально это правило выглядит так, что если самый глубокий узел (тот, который был добавлен последним) находится слева или справа, то производится одинарный поворот опорного узла относительно поддерева, содержащего этот узел, в противоположную сторону, чтобы выровнять высоты

Если самый глубокий узел находится посередине, то потребуется двойной поворот

3.2 Удаление узла из АВЛ-дерева

Удаление узла из АВЛ-дерева происходит так же, как и удаление узла из обычного двоичного дерева поиска

Чтобы после удаления сохранились свойства АВЛ-дерева, возможно, понадобится выполнить балансировку

Для этого надо подниматься вверх по пути от удаленного узла к корню и проверять в этих узлах баланс

Если в узле баланс нарушен, то надо выполнить соответствующий поворот – одинарный или двойной

Остановить просмотр можно на том узле, в котором показатель баланса не поменялся: это означает, что высота его поддерева, левого или правого, в котором производилось удаление, не изменилась

При балансировке после удаления используются те же виды поворотов, что и после вставки узла в дерево.

Теорема

- Пусть n – число внутренних вершин АВЛ-дерева, а h – его высота.
- Тогда справедливы следующие неравенства:

$$\log(n+1) \le h < 1{,}4404 \cdot \log(n+2) - 0{,}328$$

Высота АВЛ-дерева есть O(logn) , где n – число узлов дерева. Полученная оценка подтверждает тот факт, что высота АВЛ-дерева никогда не превысит высоту идеально сбалансированного дерева более чем на 45%.

Используя операции вращения, достаточно просто поддерживать инвариант, определенный для АВЛ-дерева.

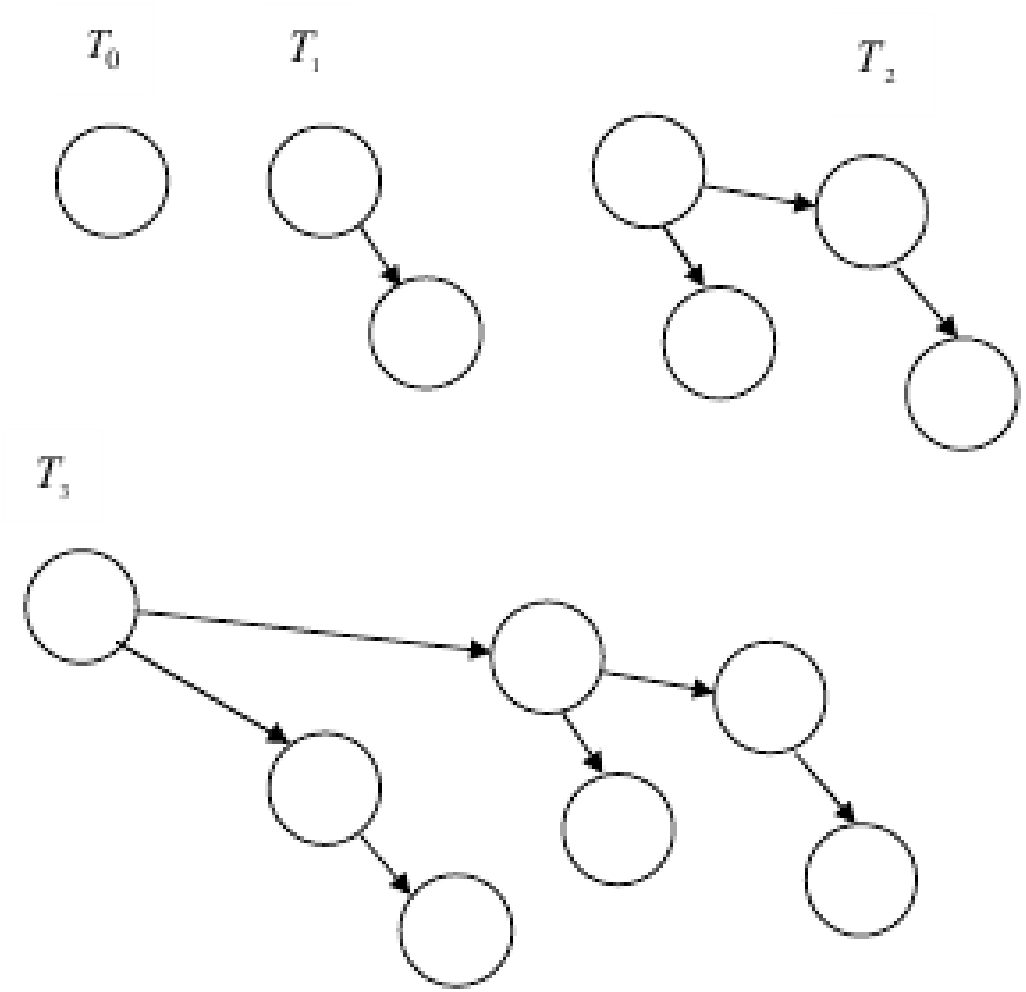
Доказательство

- 1. АВЛ-дерево – бинарное дерево

$$n \le 2^h - 1; h \ge \log(n+1).$$

- 2. Определим теперь минимальное количество внутренних вершин для АВЛ-дерева высотой h
- 3. Пусть Th – АВЛ-дерево высотой h с минимальным числом внутренних вершин.

Деревья Фибоначчи



Количество внутренних вершин

$$N_{h+1} = N_{h-1} + N_h + 1$$

$$F_i' = N_i + 1$$

Последовательность Фибоначчи

$$F_h = \frac{\Phi^h - \hat{\Phi}^h}{\sqrt{5}},$$

$$\Phi = \frac{1+\sqrt{5}}{2}, \hat{\Phi} = \frac{1-\sqrt{5}}{2}.$$

Связь F_i и F'_i

$$F_{2+h} = F'_h$$

$$F'_h = \frac{\Phi^{h+2}}{\sqrt{5}} - \frac{\hat{\Phi}^{h+2}}{\sqrt{5}}$$

$$n \geq N_h = F'_h - 1 \geq \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{h+2}}{\sqrt{5}} - \frac{\left(\frac{1-\sqrt{5}}{2}\right)^{h+2}}{\sqrt{5}} - 1 \geq \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{h+2}}{\sqrt{5}} - 2$$

Другая форма доказательства

$$\begin{cases} T(0) = 1, \ T(1) = 2, \\ T(h) = T(h-1) + T(h-2) + 1, \ n > 1 \end{cases}$$

$$T(h) + 1 = F(h)$$

$$\begin{cases} F(0) = 1, \ F(1) = 3, \\ F(h) = F(h-1) + F(h-2), \ n > 1, \end{cases}$$

$$F(h) = \left(1 + \frac{2}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^h + \left(1 - \frac{2}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^h$$

$$n \geq T(h) = \left(1 + \frac{2}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^h + \left(1 - \frac{2}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^h - 1$$

Основные операции с АВЛ-деревьями

1. Добавление вершины с заданным ключом

- Восстановление инварианта после выполнения операции добавления нового элемента

2. Удаление вершины с заданным ключом

- Восстановление инварианта после выполнения операции удаления элемента

3.3. 2-3-деревья

- Поисковое дерево называется 2-3-деревом, если оно обладает следующими свойствами (инвариантами):
- 1) каждая вершина x, не являющаяся листом, содержит два или три сына; при этом его сыновья классифицируются как левый ls(x), средний ms(x) и (возможно) правый сын rs(x);
- 2) все висячие вершины находятся на одной глубине.

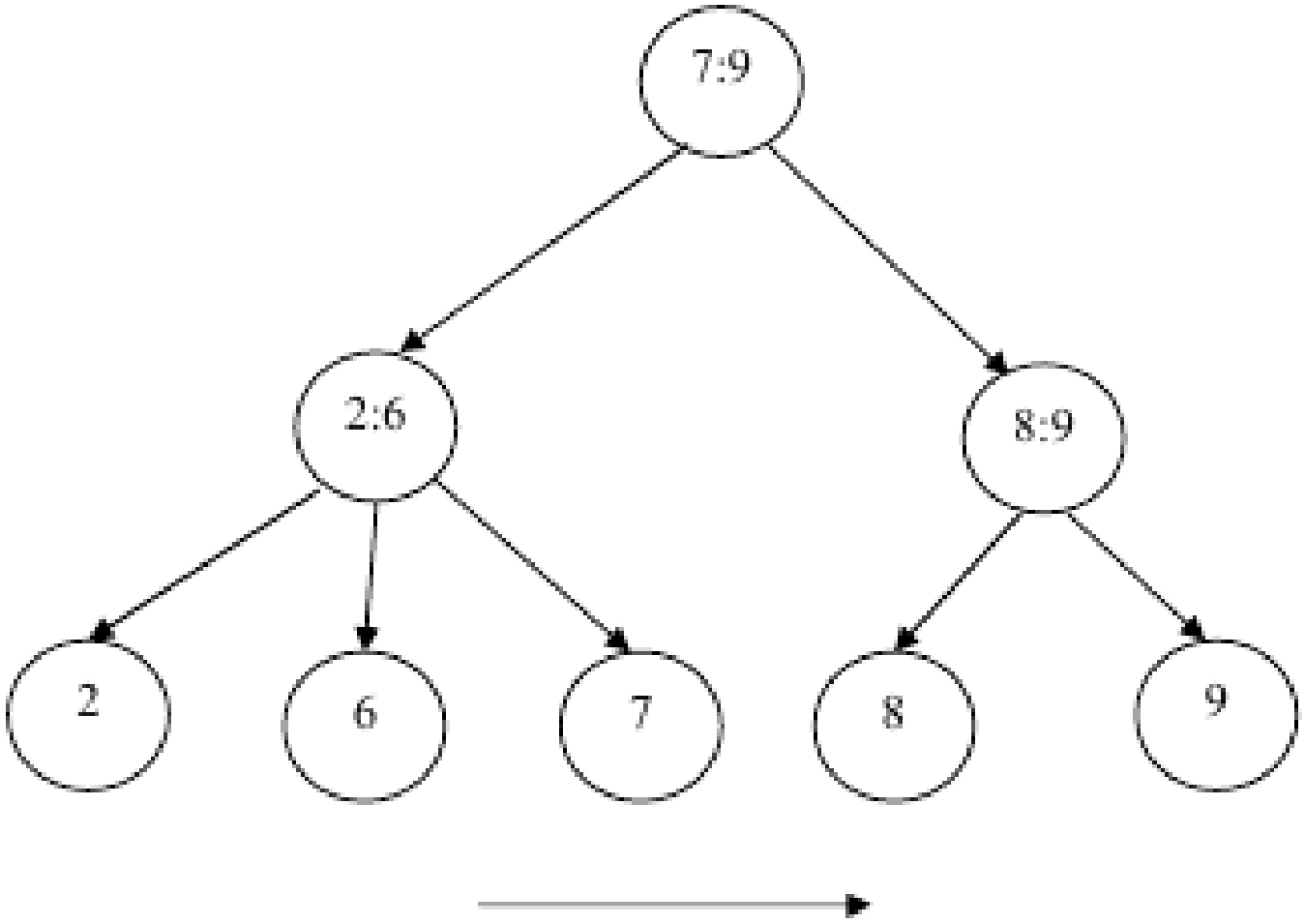
Свойства вершин

- Поскольку 2-3-дерево является поисковым, то для каждой вершины v выполняются следующие свойства:
- 1) значение ключей в поддереве, корень которого – левый сын вершины v, меньше значений ключей в поддереве, корень которого – средний сын вершины v;
- 2) значение ключей в поддереве, корень которого – средний сын вершины v, меньше значений ключей в поддереве, корень которого – правый сын вершины v.

Структура 2-3-дерева

- 1. Информация (ключи) хранится только в висячих вершинах, а все внутренние вершины – справочные.
- 2. Каждая внутренняя вершина i имеет две метки:
- l(i) – максимальное значение ключа в поддереве, корень которого – левый сын вершины i;
- m(i) – максимальное значение ключа в поддереве, корень которого – средний сын вершины i.
- Любое упорядоченное множество A можно представить в виде 2-3-дерева, если присваивать значения элементов этого множества висячим вершинам (элементы можно представить в порядке слева направо).

Пример



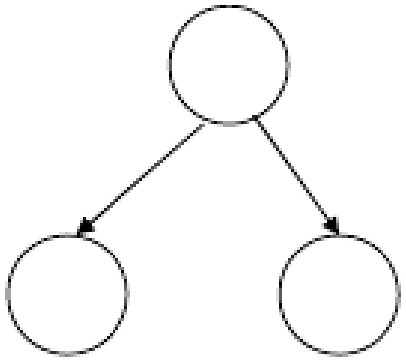
Теорема

Пусть n – общее количество вершин в 2-3-дереве (включая корень и листья);
l – количество листьев; h – высота дерева.
Тогда справедливы следующие неравенства:

$$2^h \le l \le 3^h,$$

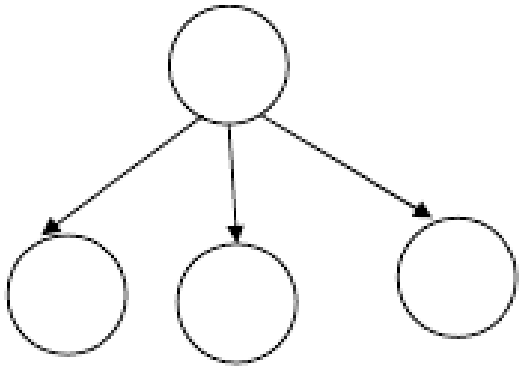
$$2^{h+1} - 1 \le n \le \frac{3^{h+1} - 1}{2}.$$

Доказательство (h=1)



$l = 2,$

$n = 3$



$l = 3,$

$n = 4$

$$2^1 \le l \le 3^1,\ 4-1 \le n \le \frac{9-1}{2}$$

Доказательство при *h*+1 (предположение справедливости при *h)*

1. (для количества листьев)

Обозначим через lh количество листьев в дереве Th , а через lhmin (lhmax) – минимально (максимально) возможное количество листьев в дереве Th .

Увеличение высоты дерева Th на единицу приводит к тому, что на глубине h+1 максимальное количество листьев l(h+1)max не превосходит 3lhmax (когда к каждому листу Th добавляется три новых сына) и не меньше величины 2lh min (когда к каждому листу Th добавляется два новых сына).

- Следовательно, выполнены следующие неравенства:

$$l_{h+1}^{\min} \le l_{h+1} \le l_{h+1}^{\max},$$

$$2l_h^{\min} \le l_{h+1} \le 3l_h^{\max}.$$

- 2.(для общего количества вершин)
- Провести рассуждения самостоятельно

Основные операции с 2-3-деревом

- 1. Поиск элемента с заданным ключом a
- 2. Добавление элемента с ключом a (предположим, что нам необходимо добавить вершину с ключом a в дерево, у которого имеется хотя бы одна вершина).
- 3. Удаление элемента с ключом a
- Трудоемкость – O(logn).

3 ПОИСК В ХЕШ-ТАБЛИЦАХ

Временная сложность неудачного поиска $\frac{1}{m} m\alpha = \alpha$

- **ТЕОРЕМА 1**. В ХТ с разрешением коллизий методом цепочек при равномерном хешировании среднее время поиска **отсутствующего** элемента (включая время на вычисление хеш-функции) равно $\Theta(1+\alpha)$.

Доказательство. В предположении простого равномерного хеширования любой ключ k, который еще не находится в таблице, может быть помещен с равной вероятностью в любую из m ячеек. Математическое ожидание времени неудачного поиска ключа k равно времени поиска до конца списка $T[h(k)]$, ожидаемая длина которого – $\mathrm{E}\left[n_{h(k)}\right] = \alpha$. Таким образом, при неудачном поиске математическое ожидание количества проверяемых элементов равно α, а общее время, необходимое для поиска, включая время вычисления хеш-функции $h(k)$, равно $\Theta(1+\alpha)$. ■

Временная сложность удачного поиска (удаления)

ТЕОРЕМА 2. Среднее время успешного поиска в хеш-таблице с цепочками и равномерным хешированием равно $\Theta(1+\alpha)$.

Доказательство (1-й способ -грубо)

Если ключ key присутствует в хеш-таблице, то он может с одинаковой вероятностью $1/\alpha$ находится в любом из α узлов связного списка $T[hash(key)]$. Если ключ находится в первом узле, то для его поиска требуется одно сравнение, если во втором – выполняется два сравнения и т. д. Математическое ожидание числа операций сравнения ключей при поиске узла равно

$$\frac{1}{\alpha}\cdot 1+\frac{1}{\alpha}\cdot 2+\cdots+\frac{1}{\alpha}\cdot\alpha\approx\frac{1+\alpha}{2}.$$

Следовательно, вычислительная сложность успешного поиска и удаления элемента из хеш-таблицы в среднем случае также равна $O(1+\alpha)$.

Теорема 3 (неудачный поиск)

Математическое ожидание числа проб при **поиске** в таблице с открытой адресацией **отсутствующего в ней элемента** не превосходит

$$\frac{1}{1-\alpha}$$

Хеширование предполагается **равномерным**.

Теорема 4 (удачный поиск)

Математическое ожидание числа проб при успешном поиске элемента в таблице при равномерном хешировании с $\alpha<1$, если считать, что ключ для успешного поиска в таблице выбирается случайным образом и все такие выборы равновероятны, не превосходит

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$