

§1. Стек

Пусть дано множество значений V .

Тогда $V^n = \overbrace{V \times V \times \dots \times V}^n$ – множество кортежей длины n , составленных из элементов V . Пусть $V^* = V^0 \cup V^1 \cup V^2 \cup \dots$, $V^+ = V^* \setminus V^0$.

Стек – это структура данных, представляющая кортеж $s \in V^*$ с тремя операциями:

1. $\text{Push} : V^* \times V \rightarrow V^*$ – добавление элемента $x \in V$ в стек;
2. $\text{Pop} : V^+ \rightarrow V^* \times V$ – удаление элемента из стека;
3. $\text{StackEmpty} : V^* \rightarrow 2$ – проверка стека на пустоту.

Говорят, что стек реализует метод доступа к элементам LIFO («Last In – First Out», «последним пришёл – первым вышел»).

Реализация стека через массив:

стек s представляется структурой $\langle s.data, s.cap, s.top \rangle$, где $s.data$ – массив размера $s.cap$, $s.top$ – размер стека:

$$s.data : \left\langle \underbrace{\bullet, \bullet, \bullet, \bullet, \bullet}_{s.top}, \circ, \circ, \circ, \circ \right\rangle.$$

При добавлении элемента в стек $s.top$ увеличивается на единицу, а при удалении – уменьшается на единицу.

Если $s.top = s.cap$ и нужно добавить элемент, то либо генерируют сообщение об ошибке, либо увеличивают размер массива (выделяют новый массив большего размера и копируют в него содержимое старого).

Все операции над стеком работают за время $O(1)$.

```

1  InitStack(out  $s$ , in  $n$ )
2       $s.data \leftarrow$  новый массив размера  $n$ 
3       $s.cap \leftarrow n$ 
4       $s.top \leftarrow 0$ 

6  StackEmpty(in  $s$ ):  $empty$ 
7       $empty \leftarrow s.top = 0$ 

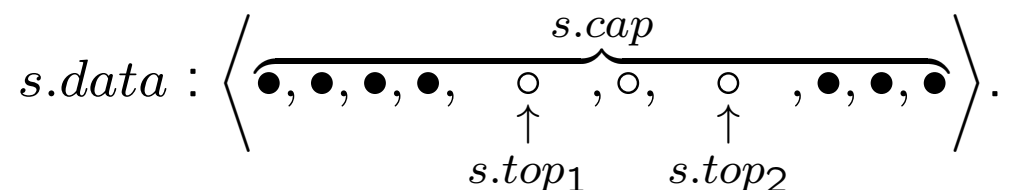
9  Push(in  $s$ , in  $x$ )
10     if  $s.top = s.cap$ : error "переполнение"
11      $s.data[s.top] \leftarrow x$ 
12      $s.top \leftarrow s.top + 1$ 

14 Pop(in  $s$ ):  $x$ 
15     if StackEmpty( $s$ ): error "опустошение"
16      $s.top \leftarrow s.top - 1$ 
17      $x \leftarrow s.data[s.top]$ 

```

На одном массиве можно реализовать сразу два стека: один будет расти с младших индексов массива, а другой – со старших.

В этом случае стек s представляется структурой $\langle s.data, s.cap, s.top_1, s.top_2 \rangle$:



Все операции над двойным стеком работают за время $O(1)$.

Инициализация двойного стека и проверки на пустоту:

```

1 InitDoubleStack(out  $s$ , in  $n$ )
2      $s.data \leftarrow$  новый массив размера  $n$ 
3      $s.cap \leftarrow n$ 
4      $s.top_1 \leftarrow 0$ 
5      $s.top_2 \leftarrow n - 1$ 
6 StackEmpty1(in  $s$ ):  $empty$ 
7      $empty \leftarrow s.top_1 = 0$ 
8 StackEmpty2(in  $s$ ):  $empty$ 
9      $empty \leftarrow s.top_2 = s.cap - 1$ 

```

Добавление и удаление элементов:

```
1 Push1(in s, in x)
2     if s.top2 < s.top1: error "переполнение"
3     s.data[s.top1] ← x
4     s.top1 ← s.top1 + 1

6 Push2(in s, in x)
7     if s.top2 < s.top1: error "переполнение"
8     s.data[s.top2] ← x
9     s.top2 ← s.top2 + 1

11 Pop1(in s): x
12     if StackEmpty1(s): error "опустошение"
13     s.top1 ← s.top1 - 1
14     x ← s.data[s.top1]

16 Pop2(in s): x
17     if StackEmpty2(s): error "опустошение"
18     s.top2 ← s.top2 - 1
19     x ← s.data[s.top2]
```

§2. Очередь

Очередь – это структура данных, представляющая кортеж $q \in V^*$ с тремя операциями:

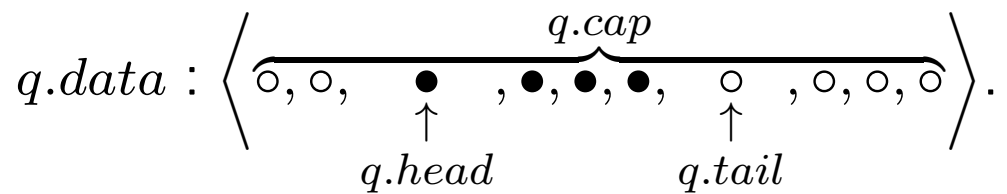
1. Enqueue : $V^* \times V \rightarrow V^*$ – добавление элемента $x \in V$ в очередь;
2. Dequeue : $V^+ \rightarrow V^* \times V$ – удаление элемента из очереди;
3. QueueEmpty : $V^* \rightarrow 2$ – проверка очереди на пустоту.

Говорят, что очередь реализует метод доступа к элементам FIFO («First In – First Out», «первым пришёл – первым вышел»).

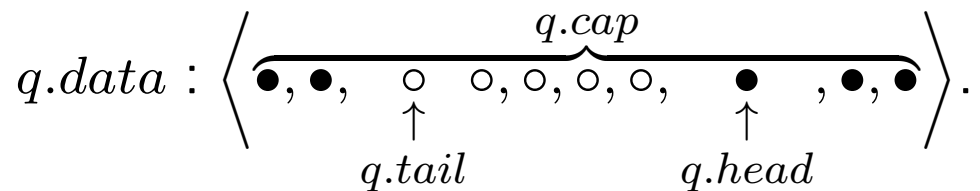
Реализация очереди через массив (*кольцевой буфер*):

очередь q представляется структурой $\langle q.data, q.cap, q.count, q.head, q.tail \rangle$, где:

- $q.data$ – массив размера $q.cap$,
- $q.count$ – количество элементов в очереди,
- $q.head$ – индекс в массиве, показывающий, откуда брать элемент для удаления;
- $q.tail$ – индекс в массиве, показывающий, куда класть элемент при добавлении.



При добавлении элемента индекс $q.tail$ увеличивается на единицу, при удалении элемента индекс $q.head$ увеличивается на единицу. Если индекс становится равным $q.cap$, он обнуляется. Поэтому возможна ситуация:



Все операции над очередью работают за время $O(1)$.

```

1 InitQueue(out  $q$ , in  $n$ )
2      $q.data \leftarrow$  новый массив размера  $n$ 
3      $q.cap \leftarrow n$ ,  $q.count \leftarrow 0$ ,  $q.head \leftarrow 0$ ,  $q.tail \leftarrow 0$ 

5 QueueEmpty(in  $q$ ): empty
6      $empty \leftarrow q.count = 0$ 

8 Enqueue(in  $q$ , in  $x$ )
9     if  $q.count = q.cap$ : error "переполнение"
10     $q.data[q.tail] \leftarrow x$ 
11     $q.tail \leftarrow q.tail + 1$ 
12    if  $q.tail = q.cap$ :  $q.tail \leftarrow 0$ 
13     $q.count \leftarrow q.count + 1$ 

15 Dequeue(in  $q$ ):  $x$ 
16    if QueueEmpty( $q$ ): error "опустошение"
17     $x \leftarrow q.data[q.head]$ 
18     $q.head \leftarrow q.head + 1$ 
19    if  $q.head = q.cap$ :  $q.head \leftarrow 0$ 
20     $q.count \leftarrow q.count - 1$ 

```


Очередь можно реализовать через двойной стек:

```
1 InitQueueOnStack(out  $s$ , in  $n$ )
2     InitDoubleStack( $s$ ,  $n$ )

4 QueueEmpty(in  $s$ ): empty
5     empty  $\leftarrow$  StackEmpty1( $s$ ) and StackEmpty2( $s$ )

7 Enqueue(in  $s$ , in  $x$ )
8     Push1( $s$ ,  $x$ )

10 Dequeue(in  $s$ ):  $x$ 
11     if StackEmpty2( $s$ ):
12         while not StackEmpty1( $s$ ):
13             Push2( $s$ , Pop1( $s$ ))
14      $x \leftarrow$  Pop2( $s$ )
```

§3. Очередь с приоритетами

Пусть задано множество значений V и множество ключей K . Назовём *словарной парой* элемент множества $P = K \times V$.

Пусть на множестве ключей определено отношение строгого порядка \triangleleft .

Очередь с приоритетами – это структура данных, представляющая кортеж $q \in P^*$ с пятью операциями:

1. $\text{Insert} : P^* \times P \rightarrow P^*$ – добавление пары $x \in P$ в очередь;
2. $\text{Maximum} : P^+ \rightarrow P$ – поиск пары с максимальным ключом (если в q есть несколько пар с максимальным ключом, возвращается одна произвольно выбранная из них пара);
3. $\text{ExtractMax} : P^+ \rightarrow P^* \times P$ – удаление пары с максимальным ключом;

4. $\text{IncreaseKey} : P^+ \times P \times K \rightarrow P^+$ – замена ключа у пары, входящей в очередь, на больший ключ;
5. $\text{QueueEmpty} : P^* \rightarrow 2$ – проверка очереди на пустоту.

Реализация очереди через пирамиду:

очередь q представляется структурой $\langle q.\text{heap}, q.\triangleleft, q.\text{cap}, q.\text{count} \rangle$, где:

- $q.\text{heap}$ – массив размера $q.\text{cap}$, составленный из указателей на тройки $\langle \text{index}, k, v \rangle$, в которых index – индекс указателя на тройку в массиве $q.\text{heap}$;
- $q.\triangleleft$ – отношение порядка на множестве ключей;
- $q.\text{count}$ – количество элементов в очереди.

Инициализация очереди:

```
1 InitPriorityQueue(out  $q$ , in  $n$ , in  $\triangleleft$ )
2    $q.heap \leftarrow$  новый массив размера  $n$ 
3    $q.\triangleleft \leftarrow \triangleleft$ ,  $q.cap \leftarrow n$ ,  $q.count \leftarrow 0$ 
```

Поиск максимального элемента тривиален и выполняется за $O(1)$:

```
1 Maximum(in  $q$ ):  $ptr$ 
2   if  $q.count = 0$ : error "очередь пуста"
3    $ptr \leftarrow q.heap[0]$ 
```

Проверка на пустоту очереди:

```
1 QueueEmpty(in  $q$ ):  $empty$ 
2    $empty \leftarrow q.count = 0$ 
```

Добавление элемента в очередь выполняется за $O(\lg n)$, потому что приходится поддерживать выполнение условий пирамиды:

```
1 Insert(in q, in ptr)
2     i  $\leftarrow$  q.count
3     if i = q.cap: error "переполнение"
4     q.count  $\leftarrow$  i + 1
5     q.heap[i]  $\leftarrow$  ptr
6     while i > 0 and q.heap[(i - 1)/2].k (q. $\triangleleft$ ) q.heap[i].k:
7         q.heap[(i - 1)/2]  $\leftrightarrow$  q.heap[i]
8         q.heap[i].index  $\leftarrow$  i
9         i  $\leftarrow$  (i - 1)/2
10    q.heap[i].index  $\leftarrow$  i
```

Удаление максимального элемента выполняется за $O(\lg n)$ из-за вызова процедуры `Heapify`:

```
1 ExtractMax(in  $q$ ):  $ptr$ 
2     if  $q.count = 0$ : error "очередь пуста"
3      $ptr \leftarrow q.heap[0]$ 
4      $q.count \leftarrow q.count - 1$ 
5     if  $q.count > 0$ :
6          $q.heap[0] \leftarrow q.heap[q.count]$ 
7          $q.heap[0].index \leftarrow 0$ 
8         Heapify( $q$ , 0,  $q.count$ ,  $q.heap$ )
```

Алгоритм `Heapify` должен быть слегка модифицирован с тем, чтобы при обмене двух элементов массива обновлять их индексы:

$$\begin{aligned} \mathcal{P}[i] &\leftrightarrow \mathcal{P}[j] \\ \mathcal{P}[i].index &\leftarrow i \\ \mathcal{P}[j].index &\leftarrow j \end{aligned}$$

Увеличение ключа у одного из элементов очереди выполняется за $O(\lg n)$:

```
1 IncreaseKey(in q, in ptr, in k')
2      $i \leftarrow ptr.index$ 
3      $ptr.k \leftarrow k'$ 
4     while  $i > 0$  and  $q.heap[(i - 1)/2].k \ (q.\triangleleft) \ k'$  :
5          $q.heap[(i - 1)/2] \leftrightarrow q.heap[i]$ 
6          $q.heap[i].index \leftarrow i$ 
7          $i \leftarrow (i - 1)/2$ 
8      $ptr.index \leftarrow i$ 
```

(Параметр i задаёт индекс элемента в массиве $q.heap$.)

§4. Связанный список

Транзитивное замыкание отношения R – это минимальное транзитивное отношение, включающее R .

Циклическая перестановка на множестве M – это биекция множества M на себя, транзитивное замыкание которой совпадает с $M \times M$.

Пример. Транзитивное замыкание перестановки $\mathcal{P} = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \end{pmatrix}$ – это отношение $\{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle, \langle 2, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 2 \rangle, \langle 3, 3 \rangle\}$.
Поэтому \mathcal{P} – не циклическая перестановка.

Пример. Транзитивное замыкание перестановки $\mathcal{P} = \begin{pmatrix} 0 & 1 & 2 \\ 2 & 0 & 1 \end{pmatrix}$ – это отношение $\{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 0, 2 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 0 \rangle, \langle 2, 1 \rangle, \langle 2, 2 \rangle\}$.
Поэтому \mathcal{P} – циклическая перестановка.

Связанный список – это структура данных, представляющая тройку $\langle M, nil, Link \rangle$, в которой:

- M – конечное непустое множество элементов;
- $nil \in M$ – элемент, называемый *ограничителем*;
- $Link \subseteq M \times M$ – циклическая перестановка.

Пусть L – множество связанных списков, V – множество значений, подмножества которого являются множествами элементов списков.

Для каждого элемента списка определены две операции:

1. $Prev : L \times V \rightarrow V$ – получение предыдущего элемента списка;
2. $Next : L \times V \rightarrow V$ – получение следующего элемента списка.

Мы будем называть $\text{Next}(\langle M, nil, Link \rangle, nil)$ ГОЛОВНЫМ ЭЛЕМЕНТОМ, а $\text{Prev}(\langle M, nil, Link \rangle, nil)$ – ХВОСТОВЫМ ЭЛЕМЕНТОМ.

Определим основные операции над списками:

1. $\text{ListEmpty} : L \rightarrow 2$ – проверка списка на пустоту;
2. $\text{ListLength} : L \rightarrow \mathbb{N}$ – вычисление длины списка (ограничитель не учитывается);
3. $\text{ListSearch} : L \times V \rightarrow 2$ – поиск элемента x ;
4. $\text{InsertAfter} : L \times V \times V \rightarrow L$ – вставка нового элемента после уже существующего элемента;
5. $\text{InsertBefore} : L \times V \times V \rightarrow L$ – вставка нового элемента до уже существующего элемента;

6. $\text{InsertBeforeHead} : L \times V \rightarrow L$ – вставка нового элемента перед головным элементом;
7. $\text{InsertAfterTail} : L \times V \rightarrow L$ – вставка нового элемента после хвостового элемента.
8. $\text{Delete} : L \times V \rightarrow L$ – удаление элемента;
9. $\text{DeleteAfter} : L \times V \rightarrow L$ – удаление элемента, следующего за указанным элементом;
10. $\text{DeleteBefore} : L \times V \rightarrow L$ – удаление элемента, расположенного перед указанным элементом;
11. $\text{DeleteHead} : L \rightarrow L$ – удаление головного элемента;
12. $\text{DeleteTail} : L \rightarrow L$ – удаление хвостового элемента.

§5. Двухнаправленный кольцевой список с ограничителем

Реализация двухнаправленного кольцевого списка с ограничителем:

- каждый элемент списка представляется структурой $\langle v, prev, next \rangle$, где $v \in V$, $prev$ – указатель на предыдущий элемент, $next$ – указатель на следующий элемент;
- список представляет собой указатель на элемент-ограничитель $\langle ?, prev, next \rangle$, где $?$ – произвольное значение из V (не используется).

Инициализация списка, проверка на пустоту и вычисление длины:

```
1 InitDoubleLinkedList(out l)
2     l ← указатель на новый элемент
3     l.prev ← l, l.next ← l
4 ListEmpty(in l): empty
5     empty ← l.next = l
6 ListLength(in l): len
7     len ← 0, x ← l
8     while x.next ≠ l:
9         len ← len + 1
10    x ← x.next
```

Поиск элемента с заданным значением возвращает указатель на найденный элемент или указатель на элемент-ограничитель, если ничего не найдено:

```
1 ListSearch(in  $l$ , in  $v$ ):  $x$ 
2      $x \leftarrow l.next$ 
3     while  $x \neq l$  and  $x.v \neq v$ :
4          $x \leftarrow x.next$ 
```

Вставка нового элемента y после уже существующего элемента x :

```
1 InsertAfter(in  $x$ , in  $y$ )
2      $z \leftarrow x.next$ 
3      $x.next \leftarrow y$ 
4      $y.prev \leftarrow x$ 
5      $y.next \leftarrow z$ 
6      $z.prev \leftarrow y$ 
```

InsertBefore, InsertBeforeHead и InsertAfterTail – аналогично.

Удаление элемента x :

```
1 Delete (in  $x$ )
2      $y \leftarrow x.prev$ 
3      $z \leftarrow x.next$ 
4      $y.next \leftarrow z$ 
5      $z.prev \leftarrow y$ 
6     — Для порядка отсоединим  $x$ :
7      $x.prev \leftarrow \text{NULL}$ 
8      $x.next \leftarrow \text{NULL}$ 
```

DeleteAfter, DeleteBefore, DeleteHead и DeleteTail – аналогично.

Нетрудно сообразить, что операции ListLength и ListSearch работают за время $O(n)$, где n – длина списка. Все остальные операции работают за константное время.

Принципиальные отличия списка от массива – отсутствие возможности быстрого доступа к i -тому элементу, но зато размер списка не фиксирован, и, кроме того, вставка и удаление элементов выполняются за константное время.

§6. Однонаправленный список

Реализация однонаправленного списка:

- каждый элемент списка представляется структурой $\langle v, next \rangle$, где $v \in V$, $next$ – указатель на следующий элемент; у хвостового элемента $next = \text{NULL}$.
- список представляет собой указатель на первый элемент.

Инициализация списка, проверка на пустоту и вычисление длины:

```
1 InitSingleLinkedList(out l)
2     l ← NULL

4 ListEmpty(in l): empty
5     empty ← l = NULL

7 ListLength(in l): len
8     len ← 0, x ← l
9     while x ≠ NULL:
10         len ← len + 1
11         x ← x.next
```

Поиск элемента с заданным значением возвращает указатель на найденный элемент или NULL, если ничего не найдено:

```
1 ListSearch(in  $l$ , in  $v$ ):  $x$ 
2      $x \leftarrow l$ 
3     while  $x \neq \text{NULL}$  and  $x.v \neq v$ :
4          $x \leftarrow x.next$ 
```

Вставка нового элемента y после уже существующего элемента x :

```
1 InsertAfter(in  $x$ , in  $y$ )
2      $z \leftarrow x.next$ 
3      $x.next \leftarrow y$ 
4      $y.next \leftarrow z$ 
```

Вставка нового элемента y перед головным элементом:

```
1 InsertBeforeHead(in/out  $l$ , in  $y$ )
2      $y.next \leftarrow l$ 
3      $l \leftarrow y$ 
```

Эффективные реализации InsertBefore и InsertAfterTail невозможны.

Удаление элемента, следующего за x (он должен существовать):

```
1 DeleteAfter(in  $x$ )
2      $y \leftarrow x.next$ 
3      $x.next \leftarrow y.next$ 
4     -- Для порядка отсоединим  $y$ :
5      $y.next \leftarrow \text{NULL}$ 
```

Удаление головного элемента (он должен существовать):

```
1 DeleteHead(in / out  $l$ )
2      $y \leftarrow l$ 
3      $l \leftarrow y.next$ 
4     -- Для порядка отсоединим  $y$ :
5      $y.next \leftarrow \text{NULL}$ 
```

Эффективные реализации Delete, DeleteBefore и DeleteTail невозможны.

Для однонаправленных списков актуально сочетание операций Search и Delete, позволяющее выполнять поиск и удаление найденного элемента:

```
1 SearchAndDelete(in/out  $l$ , in  $v$ ):  $x$ 
2      $y \leftarrow \text{NULL}$ 
3      $x \leftarrow l$ 
4     while  $x \neq \text{NULL}$ :
5         if  $x.v = v$ :
6             if  $y = \text{NULL}$ :
7                 DeleteHead( $l$ )
8             else :
9                 DeleteAfter( $y$ )
10            return
11             $y \leftarrow x$ 
12             $x \leftarrow x.next$ 
```

Операции ListLength, ListSearch и SearchAndDelete выполняются за время $O(n)$, где n – длина списка. Операции ListEmpty, InserAfter, InsertBeforeHead, DeleteAfter и DeleteHead – за $O(1)$.

§7. Ассоциативный массив

Пусть K – множество ключей, на котором задано отношение строгого порядка \triangleleft , а V – множество значений. Тогда $P = K \times V$ – множество словарных пар.

Ассоциативный массив (словарь) – это структура данных, представляющая конечное множество $M \subseteq P$ такое, что $M : K \rightarrow V$ – функция (т.е. каждая словарная пара из M имеет уникальный ключ).

Пусть A – множество ассоциативных массивов.

Для каждой словарной пары ассоциативного массива определены две операции:

1. $\text{Prec} : A \times P \rightarrow P$ – получение предыдущей пары;
2. $\text{Succ} : A \times P \rightarrow P$ – получение следующей пары.

Определим основные операции над ассоциативными массивами:

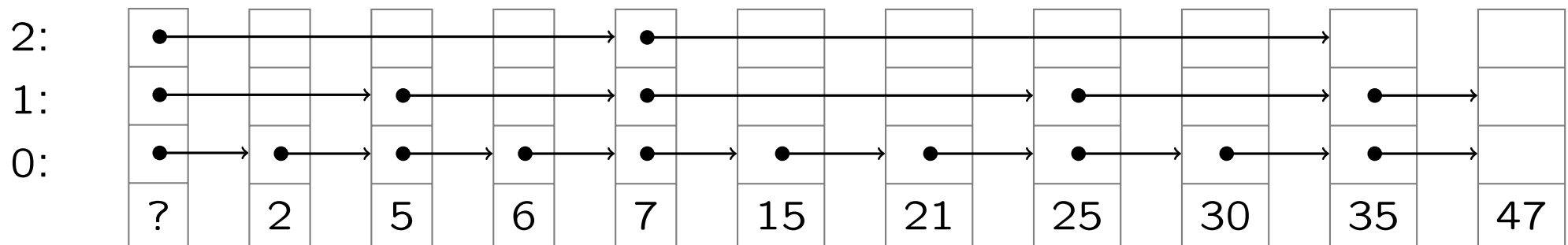
1. $\text{MapEmpty} : A \rightarrow 2$ – проверка массива на пустоту.
2. $\text{MapSearch} : A \times K \rightarrow 2$ – поиск словарной пары с заданным ключом;
3. $\text{Lookup} : A \times K \rightarrow V$ – получение значения словарной пары с заданным ключом;
4. $\text{Insert} : A \times P \rightarrow A$ – добавление новой словарной пары;
5. $\text{Delete} : A \times K \rightarrow A$ – удаление словарной пары с заданным ключом;
6. $\text{Reassign} : A \times P \rightarrow A$ – замена значения, связанного с заданным ключом;

7. Minimum : $A \rightarrow P$ – поиск словарной пары с минимальным ключом;
8. Maximum : $A \rightarrow P$ – поиск словарной пары с максимальным ключом;
9. Sequence : $A \rightarrow P^*$ – получение отсортированной последовательности входящих в ассоциативный массив словарных пар.

§8. Список с пропусками

Список с пропусками – это реализация ассоциативного массива, основанная на следующей идее:

- отсортированная в соответствии с отношением строгого порядка \prec последовательность словарных пар представляется в виде m однонаправленных списков, пронумерованных от 0 до $m - 1$;
- в список 0 входят абсолютно все словарные пары, в список 1 входит приблизительно каждая вторая словарная пара, в список 2 – приблизительно каждая четвёртая, и т.д.;
- для поиска пары с заданным ключом мы сначала движемся по списку $m - 1$, затем «спускаемся» на уровень ниже и движемся по списку $m - 2$, и т.д. На каждом уровне «скорость» движения падает приблизительно в два раза.



Замечание. Если словарная пара входит в список с номером i , то она обязана входить также во все списки с номерами от 0 до $i - 1$.

Каждый элемент списка с пропусками представляется в виде структуры $\langle k, v, next \rangle$, где
 $\langle k, v \rangle$ – словарная пара,
 $next$ – массив указателей размера m .

Если $next[i] = \text{NULL}$, то элемент либо не входит в список с номером i , либо он – последний в списке с номером i .

Сам список представляет собой указатель на дополнительный служебный элемент, который играет роль элемента-ограничителя в двунаправленном списке.

Инициализация списка с пропусками, проверка на пустоту:

```
1 InitSkipList(in  $m$ , out  $l$ )
2      $l \leftarrow$  новый элемент
3      $l.next \leftarrow$  новый массив указателей размера  $m$ 
4      $i \leftarrow 0$ 
5     while  $i < m$ :
6          $l.next[i] \leftarrow \text{NULL}$ 
7          $i \leftarrow i + 1$ 

9 MapEmpty(in  $l$ ):  $empty$ 
10     $empty \leftarrow l.next[0] = \text{NULL}$ 
```

Операция получения следующей словарной пары:

```
1 Succ(in  $x$ ):  $y$ 
2      $y \leftarrow x.next[0]$ 
```

Операция Прес допускает эффективную реализацию только на двунаправленных списках с пропусками.

Операция Skip используется при реализации других операций над списками с пропусками. Она формирует массив указателей p размера m такой, что $p[i]$ – указатель на элемент в списке с номером i , после которого может располагаться элемент с ключом k . Элемент $p[i]$ выбирается таким образом, что:

- $p[i].k \triangleleft k$, либо $p[i] = l$;
- $\neg (p[i].next[i].k \triangleleft k)$, либо $p[i].next[i] = \text{NULL}$.

```

1 Skip (in l, in m, in  $\triangleleft$ , in k, in/out p)
2      $x \leftarrow l$ 
3      $i \leftarrow m - 1$ 
4     while  $i \geq 0$ :
5         while  $x.next[i] \neq \text{NULL}$  and  $x.next[i].k \triangleleft k$ :
6              $x \leftarrow x.next[i]$ 
7          $p[i] \leftarrow x$ 
8          $i \leftarrow i - 1$ 

```

Можно показать, что алгоритм Skip в среднем работает за время $O(\lg n)$, если размер $m > \lg n$.

Поиск в списке с пропусками:

```
1 MapSearch(in  $l$ , in  $m$ , in  $\triangleleft$ , in  $k$ ):  $verdict$ 
2      $p \leftarrow$  новый массив указателей размера  $m$ 
3     Skip( $l$ ,  $m$ ,  $\triangleleft$ ,  $k$ ,  $p$ )
4      $x \leftarrow$  Succ( $p[0]$ )
5      $verdict \leftarrow x \neq \text{NULL}$  and  $x.k = k$ 

7 Lookup(in  $l$ , in  $m$ , in  $\triangleleft$ , in  $k$ ):  $v$ 
8      $p \leftarrow$  новый массив указателей размера  $m$ 
9     Skip( $l$ ,  $m$ ,  $\triangleleft$ ,  $k$ ,  $p$ )
10     $x \leftarrow$  Succ( $p[0]$ )
11    if  $x = \text{NULL}$  or  $x.k \neq k$ :
12        panic
13     $v \leftarrow x.v$ 
```

При вставке нового элемента в список с пропусками количество t списков, в которые будет входить новый элемент, определяется псевдослучайным образом. Если r – псевдослучайное целое число, то $t = 1 + \max \{x \mid r = 0 \pmod{2^x}\}$.

```

1 Insert(in  $l$ , in  $m$ , in  $\triangleleft$ , in  $k$ , in  $v$ )
2      $p \leftarrow$  новый массив указателей размера  $m$ 
3     Skip( $l$ ,  $m$ ,  $\triangleleft$ ,  $k$ ,  $p$ )
4     if  $p[0].next[0] \neq \text{NULL}$  and  $p[0].next[0].k = k$ :
5         panic

7      $x \leftarrow$  новый элемент
8      $x.next \leftarrow$  новый массив указателей размера  $m$ 
9      $r \leftarrow \text{rand}() * 2$  — чётное псевдослучайное целое число
10     $i \leftarrow 0$ 
11    while  $i < m$  and  $r \bmod 2 = 0$ :
12         $x.next[i] \leftarrow p[i].next[i]$ 
13         $p[i].next[i] \leftarrow x$ 
14         $i \leftarrow i + 1$ 
15         $r \leftarrow r/2$ 
16    while  $i < m$ :
17         $x.next[i] \leftarrow \text{NULL}$ 
18         $i \leftarrow i + 1$ 

```

Удаление элемента с ключом k из списка с пропусками:

```
1 Delete(in  $l$ , in  $m$ , in  $\triangleleft$ , in  $k$ )
2      $p \leftarrow$  новый массив указателей размера  $m$ 
3     Skip( $l$ ,  $m$ ,  $\triangleleft$ ,  $k$ ,  $p$ )
4      $x \leftarrow$  Succ( $p[0]$ )
5     if  $x = \text{NULL}$  or  $x.k \neq k$ :
6         panic

8      $i \leftarrow 0$ 
9     while  $i < m$  and  $p[i].next[i] = x$ :
10          $p[i].next[i] \leftarrow x.next[i]$ 
11          $i \leftarrow i + 1$ 
```

Замена значения с заданным ключом, получение минимального элемента и формирование отсортированной последовательности элементов:

```
1 Reassign(in  $l$ , in  $m$ , in  $\triangleleft$ , in  $k$ , in  $v$ )
2      $p \leftarrow$  новый массив указателей размера  $m$ 
3     Skip( $l$ ,  $m$ ,  $\triangleleft$ ,  $k$ ,  $p$ )
4      $x \leftarrow \text{Succ}(p[0])$ 
5     if  $x = \text{NULL}$  or  $x.k \neq k$ :
6         panic
7      $x.v \leftarrow v$ 

9 Minimum(in  $l$ ):  $x$ 
10     $x \leftarrow \text{Succ}(l)$ 

12 Sequence(in  $l$ , in/out  $queue$ )
13     $x \leftarrow \text{Succ}(l)$ 
14    while  $x \neq \text{NULL}$ :
15        Enqueue( $queue$ ,  $x$ )
16     $x \leftarrow \text{Succ}(x)$ 
```

§9. Бинарное дерево поиска

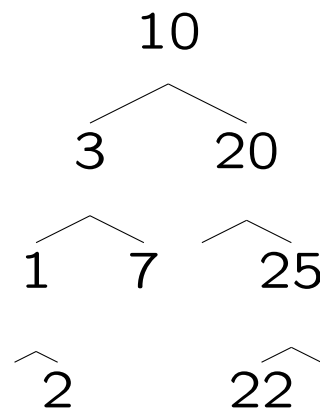
Бинарное дерево поиска является ещё одной реализацией ассоциативного массива. Бинарное дерево состоит из *вершин*, соединённых направленными *дугами*. В каждой вершине хранится одна словарная пара.

Из одной вершины может исходить не более двух дуг, поэтому у вершины не может быть более двух *дочерних вершин*. Вершины, из которых не исходит ни одной дуги, называются *листьями*. Единственная вершина дерева, в которую не входит ни одна дуга, называется *корнем*.

Высота дерева – это максимальное количество дуг между корнем и листом дерева.

Для каждой вершины дерева одна дочерняя вершина, если она существует, является *левой вершиной*, а другая, если существует, является *правой вершиной*. Вершина по отношению к своим дочерним вершинам является *родительской вершиной*.

Бинарное дерево поиска обладает важным свойством: если k — ключ некоторой вершины дерева, то ключи всех вершин в её левом поддереве предшествуют k , и, кроме того, k предшествует ключам всех вершин в её правом поддереве.



Вершину дерева представляют структурой $\langle k, v, parent, left, right \rangle$, в которой:

- $\langle k, v \rangle \in P$ – словарная пара;
- $parent$ – указатель на родительскую вершину;
- $left$ и $right$ – указатели на левую и правую дочерние вершины.

Если вершина x – корень, то $x.parent = \text{NULL}$. Если вершина x не имеет левой дочерней вершины, то $x.left = \text{NULL}$, а если не имеет правой дочерней вершины, то $x.right = \text{NULL}$.

Само дерево представляется указателем на корень. В случае пустого дерева этот указатель равен NULL .

Инициализация дерева и проверка на пустоту тривиальны:

```
1 InitBinarySearchTree(out t)
2     t ← NULL

4 MapEmpty(in t): empty
5     empty ← t = NULL
```


Операция поиска словарной пары с минимальным ключом выполняется путём спуска от корня до листа по дугам, ведущим к левым дочерним вершинам. Для пустого дерева Minimum будет возвращать NULL.

```
1 Minimum(in  $t$ ):  $x$ 
2     if  $t = \text{NULL}$ :
3          $x \leftarrow \text{NULL}$ 
4     else :
5          $x \leftarrow t$ 
6         while  $x.\text{left} \neq \text{NULL}$ :
7              $x \leftarrow x.\text{left}$ 
```

Операция Maximum реализуется аналогично. Время выполнения операций Minimum и Maximum – $O(h)$, где h – высота дерева.

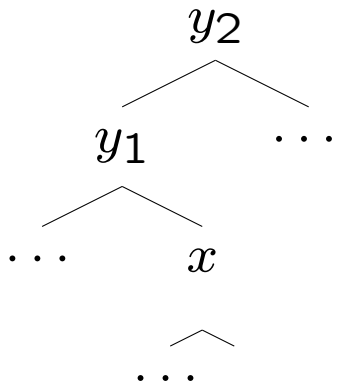
Утверждение. Minimum всегда возвращает указатель на вершину, не имеющую левой дочерней вершины, а Maximum всегда возвращает указатель на вершину, не имеющую правой дочерней вершины.

Справедливость утверждения прямо следует из алгоритма.

Операция получения следующей словарной пары работает за время $O(h)$:

```
1 Succ(in  $x$ ):  $y$ 
2   if  $x.right \neq \text{NULL}$ :
3        $y \leftarrow \text{Minimum}(x.right)$ 
4   else:
5        $y \leftarrow x.parent$ 
6       while  $y \neq \text{NULL}$  and  $x = y.right$ :
7            $x \leftarrow y$ 
8            $y \leftarrow y.parent$ 
```

Иллюстрация. (случай, когда $x.right = \text{NULL}$)



Операция PreS реализуется аналогично.

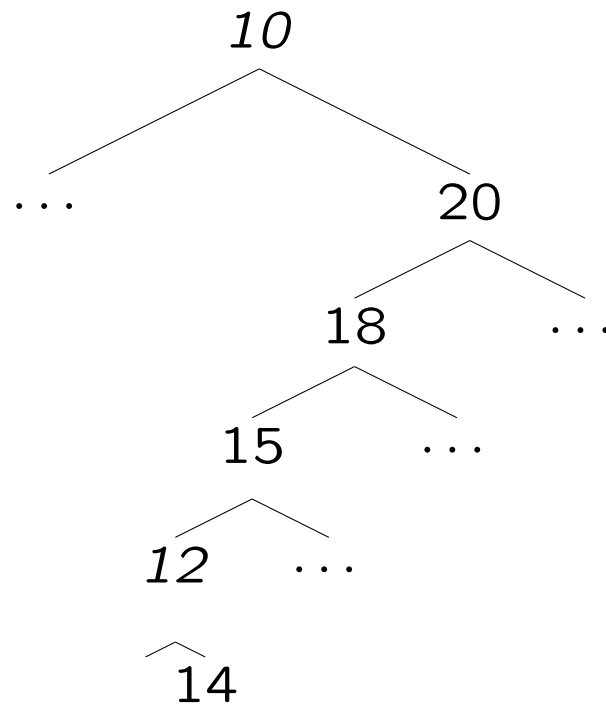
Утверждение. Если вершина x в бинарном дереве поиска имеет правую дочернюю вершину, то вершина $\text{Succ}(x)$ не имеет левой дочерней вершины.

Действительно, рассмотрим вершину x , у которой $x.\text{right} \neq \text{NULL}$. Тогда, согласно алгоритму Succ , за вершиной x будет следовать вершина $\text{Minimum}(x.\text{right})$, у которой, как известно, нет левой дочерней вершины.

Утверждение. Если вершина x в бинарном дереве поиска имеет левую дочернюю вершину, то вершина $\text{Prec}(x)$ не имеет правой дочерней вершины.

Доказывается аналогично.

Пример. (Вершина 10 имеет правую дочернюю вершину 20, значит вершина $12 = \text{Succ}(10)$ не имеет левой дочерней вершины.)



Поиск в бинарном дереве работает за время $O(h)$:

```
1 Descend(in  $t$ , in  $\triangleleft$ , in  $k$ ):  $x$ 
2      $x \leftarrow t$ 
3     while  $x \neq \text{NULL}$  and  $x.k \neq k$ :
4         if  $k \triangleleft x.k$ :
5              $x \leftarrow x.\text{left}$ 
6         else :
7              $x \leftarrow x.\text{right}$ 

9 MapSearch(in  $t$ , in  $\triangleleft$ , in  $k$ ):  $\text{verdict}$ 
10     $\text{verdict} \leftarrow \text{Descend}(t, \triangleleft, k) \neq \text{NULL}$ 

12 Lookup(in  $t$ , in  $\triangleleft$ , in  $k$ ):  $v$ 
13     $x \leftarrow \text{Descend}(t, \triangleleft, k)$ 
14    if  $x = \text{NULL}$ :
15        panic
16     $v \leftarrow x.v$ 
```

Вставка новой словарной пары в бинарное дерево работает за время $O(h)$:

```
1 Insert(in/out  $t$ , in  $\triangleleft$ , in  $k$ , in  $v$ )
2    $y \leftarrow$  новая вершина,  $y.k \leftarrow k$ ,  $y.v \leftarrow v$ 
3    $y.parent \leftarrow \text{NULL}$ ,  $y.left \leftarrow \text{NULL}$ ,  $y.right \leftarrow \text{NULL}$ ,
4   if  $t = \text{NULL}$ :
5        $t \leftarrow y$ 
6   else:
7        $x \leftarrow t$ 
8       loop:
9           if  $x.k = k$ :
10              panic
11          if  $k \triangleleft x.k$ :
12              if  $x.left = \text{NULL}$ :
13                   $x.left \leftarrow y$ ,  $y.parent \leftarrow x$ 
14                  break
15               $x \leftarrow x.left$ 
16          else:
17              if  $x.right = \text{NULL}$ :
18                   $x.right \leftarrow y$ ,  $y.parent \leftarrow x$ 
19                  break
20               $x \leftarrow x.right$ 
```

Вспомогательная операция ReplaceNode меняет в бинарном дереве вершину x (вместе с растущим из неё поддеревом) на вершину y (которая тоже может быть корнем поддерева). Операция допускает $y = \text{NULL}$: в этом случае происходит удаление поддерева с корнем в x .

```
1 ReplaceNode(in/out  $t$ , in  $x$ , in  $y$ )
2     if  $x = t$ :
3          $t \leftarrow y$ 
4         if  $y \neq \text{NULL}$ :  $y.\text{parent} \leftarrow \text{NULL}$ 
5     else :
6          $p \leftarrow x.\text{parent}$ 
7         if  $y \neq \text{NULL}$ :
8              $y.\text{parent} \leftarrow p$ 
9         if  $p.\text{left} = x$ :
10             $p.\text{left} \leftarrow y$ 
11        else :
12             $p.\text{right} \leftarrow y$ 
```

Нетрудно заметить, что операция ReplaceNode в общем случае не сохраняет свойство бинарного дерева поиска.

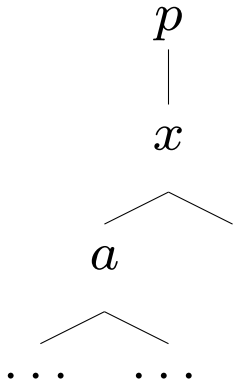
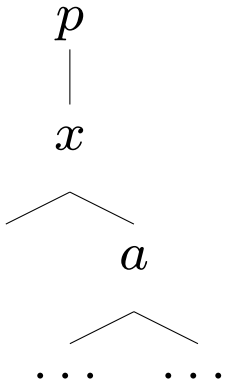
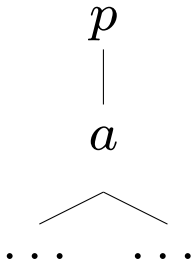
При удалении вершины x из бинарного дерева возможны три случая:

1. Вершина x – листовая. Для удаления достаточно обнулить указатель на неё у её родительской вершины:

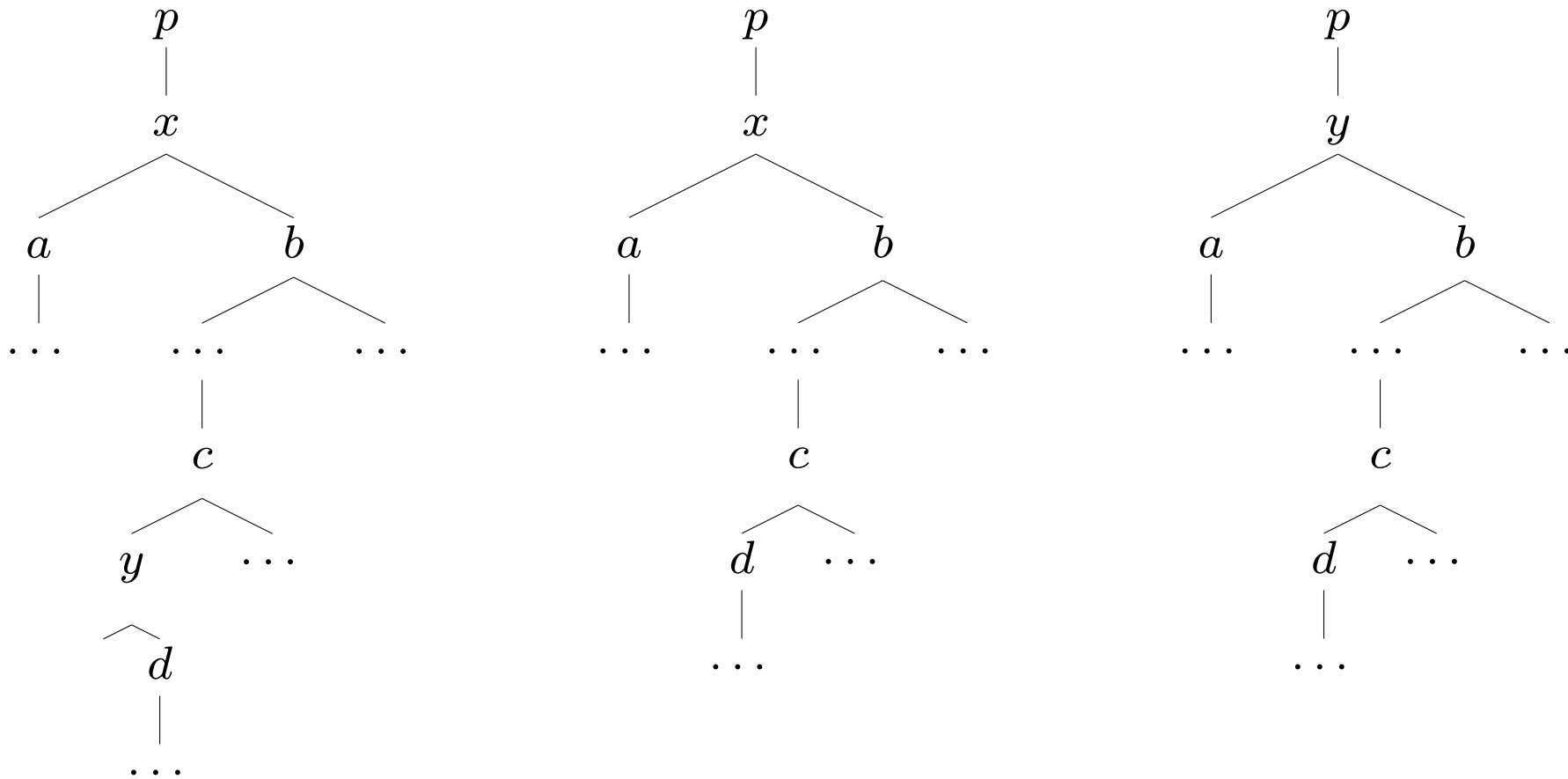
$\text{ReplaceNode}(t, x, \text{NULL})$.

2. У вершины x есть одна дочерняя вершина a . Для удаления достаточно заменить x на a :

$\text{ReplaceNode}(t, x, a)$.

До (a – левая)	До (a – правая)	После
		

3. У вершины x есть две дочерние вершины a и b . Пусть $y = \text{Succ}(x)$. Удалим вершину y из дерева, воспользовавшись схемой из случая 2 (у вершины y нет левой дочерней вершины).



Присоединим вершины a и b вместе с растущими из них поддеревьями к вершине y . При этом a становится левой дочерней ($a \triangleleft x \triangleleft y$), а b — правой дочерней ($y \triangleleft b$).

Заменяем в дереве x на y .

```

1 Delete(in/out  $t$ , in  $\triangleleft$ , in  $k$ )
2      $x \leftarrow \text{Descend}(t, \triangleleft, k)$ 
3     if  $x = \text{NULL}$ :
4         panic
5     if  $x.\text{left} = \text{NULL}$  and  $x.\text{right} = \text{NULL}$ :
6         ReplaceNode( $t$ ,  $x$ ,  $\text{NULL}$ )
7     else if  $x.\text{left} = \text{NULL}$ :
8         ReplaceNode( $t$ ,  $x$ ,  $x.\text{right}$ )
9     else if  $x.\text{right} = \text{NULL}$ :
10        ReplaceNode( $t$ ,  $x$ ,  $x.\text{left}$ )
11    else :
12         $y \leftarrow \text{Succ}(x)$ 
13        ReplaceNode( $t$ ,  $y$ ,  $y.\text{right}$ )
14         $x.\text{left}.\text{parent} \leftarrow y$ 
15         $y.\text{left} \leftarrow x.\text{left}$ 
16        if  $x.\text{right} \neq \text{NULL}$ :  $x.\text{right}.\text{parent} \leftarrow y$ 
17         $y.\text{right} \leftarrow x.\text{right}$ 
18        ReplaceNode( $t$ ,  $x$ ,  $y$ )

```

Удаление вершины выполняется за время $O(h)$.

Замена значения с заданным ключом:

```
1 Reassign(in/out  $t$ , in  $\triangleleft$ , in  $k$ , in  $v$ )
2      $x \leftarrow \text{Descend}(t, \triangleleft, k)$ 
3     if  $x = \text{NULL}$ :
4         panic
5      $x.v \leftarrow v$ 
```

Рекурсивное формирование отсортированной последовательности вершин бинарного дерева:

```
1 Sequence(in  $t$ , in/out  $queue$ )
2     if  $t \neq \text{NULL}$ :
3         Sequence( $t.\text{left}$ ,  $queue$ )
4         Enqueue( $queue$ ,  $t$ )
5         Sequence( $t.\text{right}$ ,  $queue$ )
```

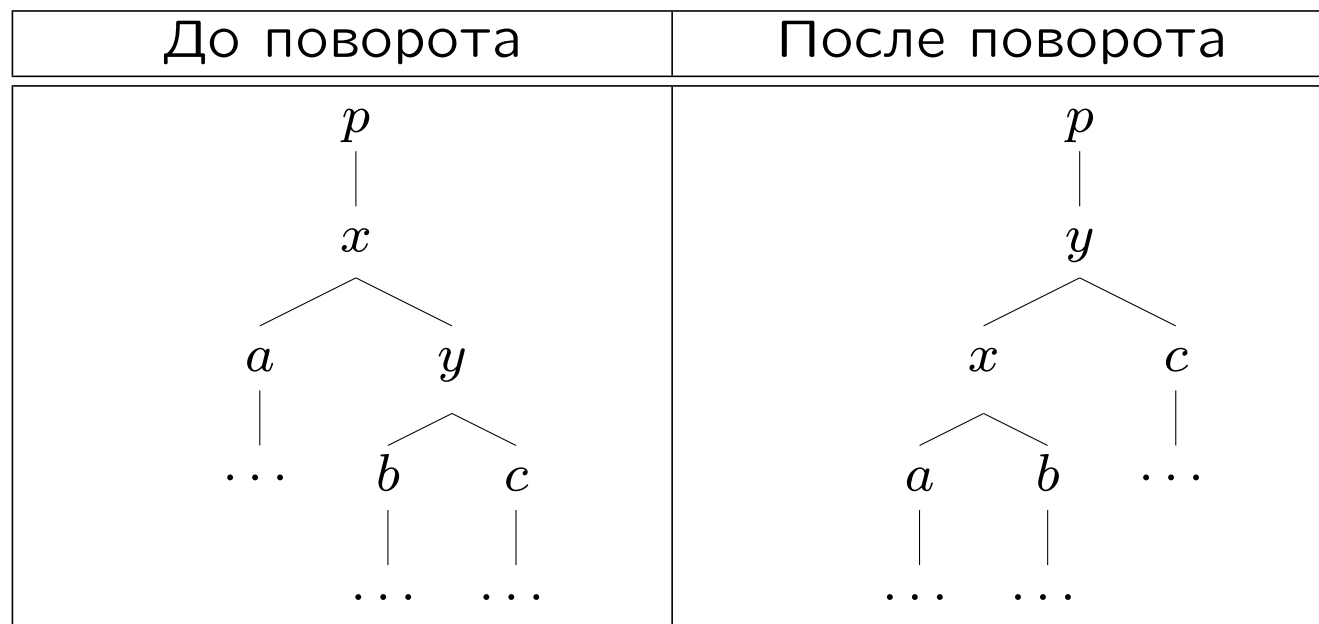
Нерекурсивное формирование отсортированной последовательности вершин бинарного дерева:

```
1 Sequence(in  $t$ , in/out  $queue$ )
2     InitStack( $s$ , ...)
3      $x \leftarrow t$ 
4     loop :
5         while  $x \neq \text{NULL}$  :
6             Push( $s$ ,  $x$ )
7              $x \leftarrow x.left$ 
8         if StackEmpty( $s$ ) :
9             break
10         $x \leftarrow \text{Pop}(s)$ 
11        Enqueue( $queue$ ,  $x$ )
12         $x \leftarrow x.right$ 
```

В этом алгоритме предполагается, что либо мы знаем общее количество вершин в дереве, либо стек умеет расти во время работы алгоритма.

Для реализации балансировки бинарного дерева нам потребуется операция поворота поддерева, растущего из некоторой вершины.

Если x – вершина бинарного дерева, и y – её правая дочерняя вершина, то поворот поддерева с корнем x *влево* модифицирует дерево таким образом, что y становится родительской вершиной для x .



```

1 RotateLeft(in/out  $t$ , in  $x$ )
2      $y \leftarrow x.right$ 
3     if  $y = \text{NULL}$ :
4         panic
5     ReplaceNode( $t$ ,  $x$ ,  $y$ )
6      $b \leftarrow y.left$ 
7     if  $b \neq \text{NULL}$ :
8          $b.parent \leftarrow x$ 
9      $x.right \leftarrow b$ 
10     $x.parent \leftarrow y$ 
11     $y.left \leftarrow x$ 

```

Операция правого поворота RotateRight реализуется аналогично.

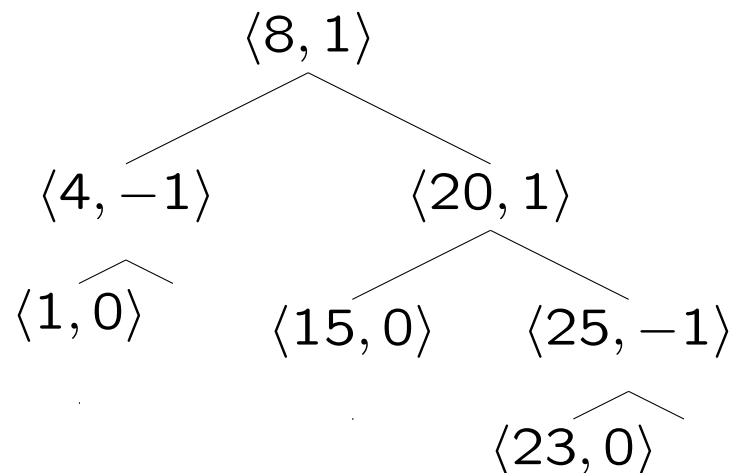
§10. AVL-дерево

Г.М. Адельсон-Вельский, Е.М. Ландис:

Бинарное дерево поиска называется *сбалансированным*, если высоты дочерних поддеревьев каждой из его вершин отличаются не более, чем на единицу. (Такие деревья ещё называют AVL-деревьями).

Вершина AVL-дерева представляют как $\langle k, v, balance, parent, left, right \rangle$, где *balance* – разность высот правого и левого поддеревьев.

Пример. (в вершинах дерева изображены пары $\langle k, balance \rangle$)



Пусть дерево с корнем x поворачивается влево. Посмотрим, как меняются балансы в вершинах дерева при повороте.

До поворота	После поворота
<div> <div>x</div> <div> <div>a</div> <div>y</div> </div> <div> <div>...</div> <div> <div>b</div> <div>c</div> </div> <div> <div>...</div> <div>...</div> </div> </div> </div>	<div> <div>y</div> <div> <div>x</div> <div>c</div> </div> <div> <div> <div>a</div> <div>b</div> </div> <div>...</div> </div> <div> <div>...</div> <div>...</div> </div> </div>

Очевидно, что поддеревья с корнями a , b и c не меняются, и балансы всех входящих в них вершин сохраняются.

Сначала разберёмся с балансом вершины x .

До поворота	После поворота

$$x.balance_2 = height(b) - height(a).$$

Если $y.balance_1 \leq 0$, то

$$x.balance_1 = (1 + height(b)) - height(a),$$

$$x.balance_2 = x.balance_1 - 1.$$

Если $y.balance_1 > 0$, то

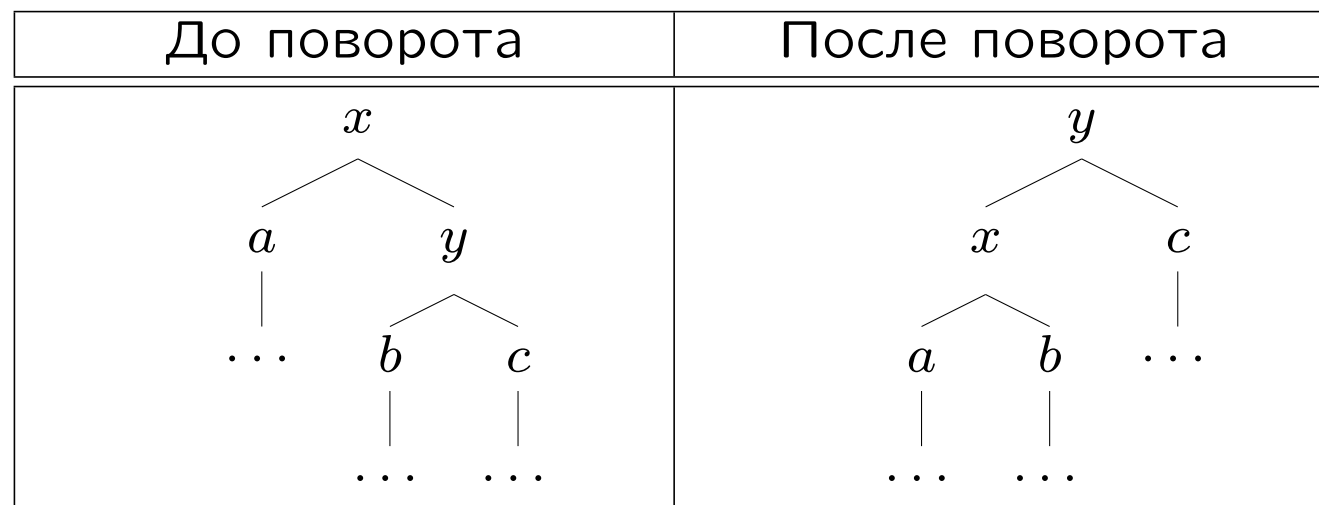
$$x.balance_1 = (1 + height(c)) - height(a),$$

$$height(c) = height(b) + y.balance_1,$$

$$x.balance_1 = (1 + height(b) + y.balance_1) - height(a),$$

$$x.balance_2 = x.balance_1 - 1 - y.balance_1.$$

Теперь посчитаем новый баланс вершины y .



$$y.balance_1 = height(c) - height(b).$$

Если $x.balance_2 \geq 0$, то

$$y.balance_2 = height(c) - (1 + height(b)) = y.balance_1 - 1.$$

Если $x.balance_2 < 0$, то

$$y.balance_2 = height(c) - (1 + height(a)),$$

$$height(a) = height(b) - x.balance_2,$$

$$y.balance_2 = height(c) - (1 + height(b) - x.balance_2) = y.balance_1 - 1 + x.balance_2.$$

Таким образом, в алгоритм RotateLeft для АВЛ-дерева нужно дописать изменение балансов вершин x и y :

```
1 RotateLeft(in/out  $t$ , in  $x$ )
2     . . .
3      $x.balance \leftarrow x.balance - 1$ 
4     if  $y.balance > 0$ :
5          $x.balance \leftarrow x.balance - y.balance$ 
6      $y.balance \leftarrow y.balance - 1$ 
7     if  $x.balance < 0$ :
8          $y.balance \leftarrow y.balance + x.balance$ 
```

Если воспроизвести те же рассуждения для поворота вправо, мы получим следующее дополнение к алгоритму RotateRight:

```
1 RotateRight(in/out  $t$ , in  $x$ )
2     . . .
3      $x.balance \leftarrow x.balance + 1$ 
4     if  $y.balance < 0$ :
5          $x.balance \leftarrow x.balance - y.balance$ 
6      $y.balance \leftarrow y.balance + 1$ 
7     if  $x.balance > 0$ :
8          $y.balance \leftarrow y.balance + x.balance$ 
```

Обратите внимание: алгоритмы RotateLeft и RotateRight работают несколько неправильно – они не изменяют балансы родительских вершин. В дальнейшем мы увидим, что это и не нужно.

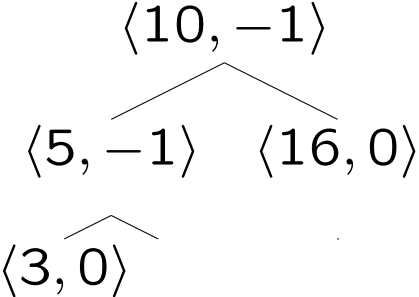
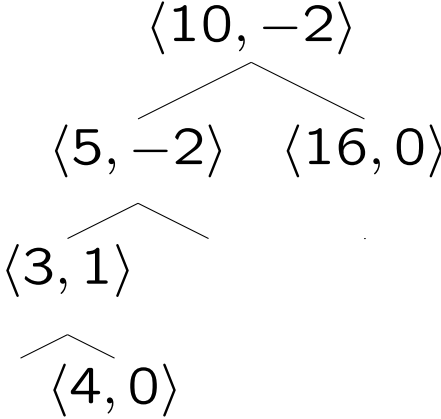
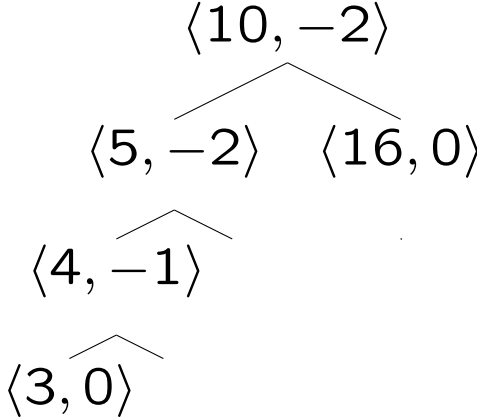
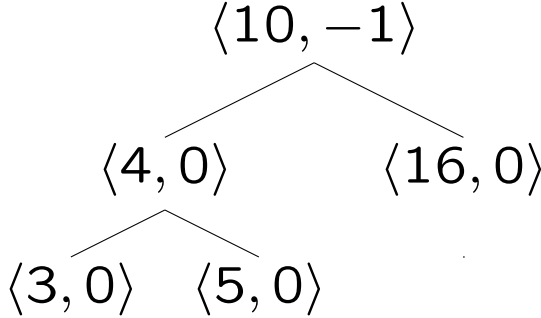
При добавлении новой словарной пары к AVL-дереву оно может перестать быть сбалансированным.

Однако, мы покажем, что такое дерево всегда можно сбалансировать за один или два поворота.

Пример. (балансировка за один поворот)

Исходное дерево	После добавления вершины с ключом 1	После поворота поддерева, растущего из 5, вправо
$ \begin{array}{c} \langle 10, -1 \rangle \\ \swarrow \quad \searrow \\ \langle 5, -1 \rangle \quad \langle 16, 0 \rangle \\ \swarrow \\ \langle 3, 0 \rangle \end{array} $	$ \begin{array}{c} \langle 10, -2 \rangle \\ \swarrow \quad \searrow \\ \langle 5, -2 \rangle \quad \langle 16, 0 \rangle \\ \swarrow \\ \langle 3, -1 \rangle \\ \swarrow \\ \langle 1, 0 \rangle \end{array} $	$ \begin{array}{c} \langle 10, -1 \rangle \\ \swarrow \quad \searrow \\ \langle 3, 0 \rangle \quad \langle 16, 0 \rangle \\ \swarrow \quad \searrow \\ \langle 1, 0 \rangle \quad \langle 5, 0 \rangle \end{array} $

Пример. (балансировка за два поворота)

Исходное дерево	После добавления вершины с ключом 4	После поворота поддерева, растущего из 3, влево	После поворота поддерева, растущего из 5, вправо
			

Утверждение. Если после добавления новой вершины к АВЛ-дереву мы получаем несбалансированное дерево, то его можно сбалансировать за один или два поворота, причём высота дерева после балансировки будет совпадать с его высотой до добавления новой вершины.

Докажем это утверждение индуктивно по высоте дерева, какой она была до добавления новой вершины.

1. Высота $h = 0$.

Дерево состоит из единственной вершины. Добавление новой вершины не может сделать его несбалансированным, поэтому утверждение тривиально верно.

2. Высота $h = 1$.

Случаи, когда после добавления вершины дерево оказалось несбалансированным и для балансировки требуется один поворот:

Исходное дерево	После добавления вершины z	После поворота поддерева, растущего из $x...$
$\langle x, -1 \rangle$ $\langle y, 0 \rangle$	$\langle x, -2 \rangle$ $\langle y, -1 \rangle$ $\langle z, 0 \rangle$...вправо $\langle y, 0 \rangle$ $\langle z, 0 \rangle \quad \langle x, 0 \rangle$
$\langle x, 1 \rangle$ $\langle y, 0 \rangle$	$\langle x, 2 \rangle$ $\langle y, 1 \rangle$ $\langle z, 0 \rangle$...влево $\langle y, 0 \rangle$ $\langle x, 0 \rangle \quad \langle z, 0 \rangle$

В итоге дерево сбалансировано и его высота не поменялась.

...

2. Высота $h = 1$.

...

Случаи, когда после добавления вершины дерево оказалось несбалансированным и для балансировки требуется два поворота:

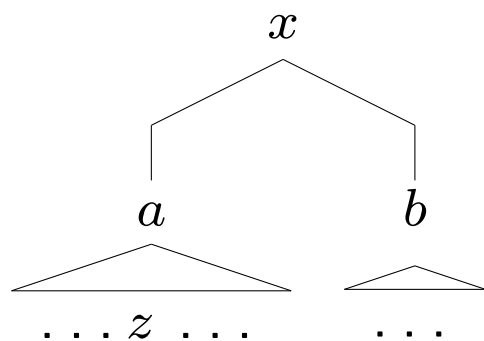
Исходное дерево	После добавления вершины z	После поворота поддерева, растущего из $y...$	После поворота поддерева, растущего из $x...$
$\langle x, -1 \rangle$ $\langle y, 0 \rangle$	$\langle x, -2 \rangle$ $\langle y, 1 \rangle$ $\langle z, 0 \rangle$...влево $\langle x, -2 \rangle$ $\langle z, -1 \rangle$ $\langle y, 0 \rangle$...вправо $\langle z, 0 \rangle$ $\langle y, 0 \rangle$ $\langle x, 0 \rangle$
$\langle x, 1 \rangle$ $\langle y, 0 \rangle$	$\langle x, 2 \rangle$ $\langle y, -1 \rangle$ $\langle z, 0 \rangle$...вправо $\langle x, 2 \rangle$ $\langle z, 1 \rangle$ $\langle y, 0 \rangle$...влево $\langle z, 0 \rangle$ $\langle x, 0 \rangle$ $\langle y, 0 \rangle$

В итоге дерево сбалансировано и его высота не поменялась. (Обратите внимание: поворот вокруг y не влияет на баланс x .)

3. Предположим, что для деревьев высоты меньше h утверждение справедливо. Докажем, что оно справедливо и для деревьев высоты h .

Пусть имеется дерево высоты h с корнем x . Так как дерево сбалансировано, и $h > 1$, то у вершины x есть две дочерние вершины a и b .

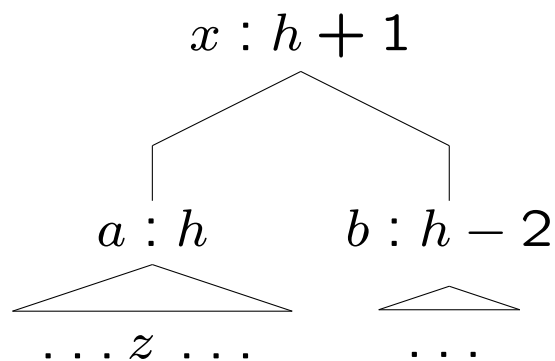
Добавим в дерево новую вершину z . Рассмотрим случай, когда она оказывается в поддереве a . Случай, когда z оказывается в поддереве b , мы рассматривать не будем, потому что он во всём аналогичен.



Если поддерево a в результате добавления z станет несбалансированным, то, учитывая, что его высота меньше h , его можно сбалансировать одним или двумя поворотами. При этом после балансировки оно станет прежней высоты, и баланс дерева x не изменится.

Поэтому будем считать, что поддерево a осталось сбалансированным, и балансировка нарушена только в вершине x .

Обратим внимание на высоту деревьев a и b . Если бы высота дерева b превышала или хотя бы была равна высоте дерева a , то при добавлении в дерево a новой вершины z баланс в вершине x никак не мог бы быть нарушен. Тем самым, высота a больше высоты b .



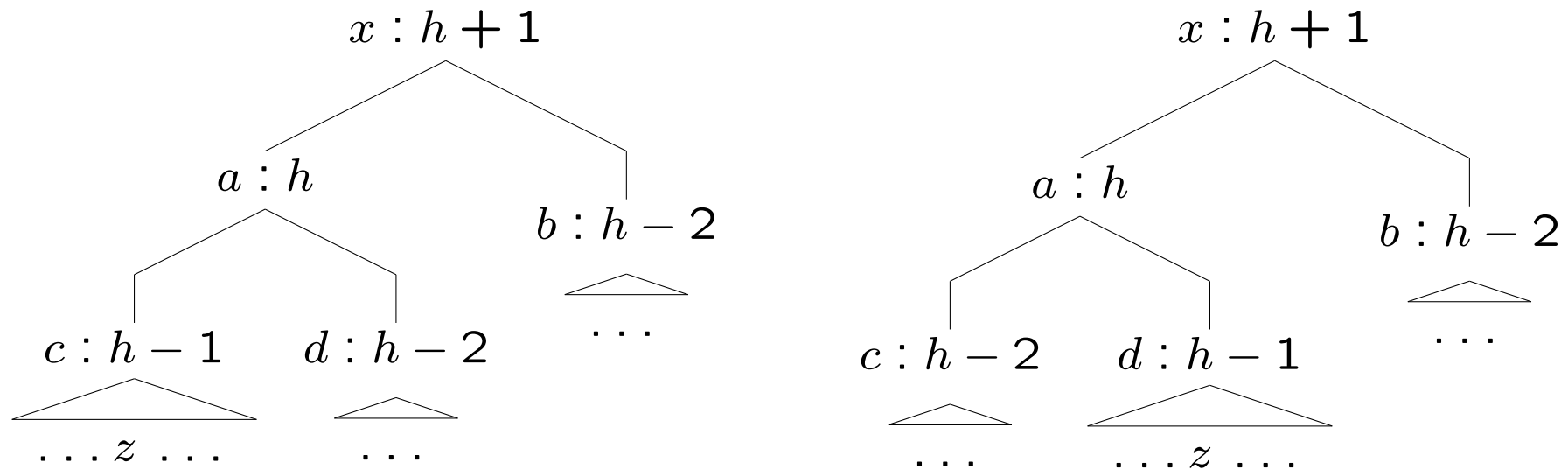
Так как до добавления вершины z дерево x было сбалансировано, а после добавления стало несбалансировано, то высота a была ровно на 1 больше высоты b .

Итак, $height_1(a) = h - 1$, $height(b) = h - 2$.

Соответственно, после добавления z высота a выросла на 1 и стала равна $height_2(a) = h$. (Если бы высота a не выросла, то баланс в x не был бы нарушен.)

Так как $h > 1$, то $height_1(a) > 0$. После добавления z дерево a осталось сбалансировано, и его высота выросла. Поэтому высоты левого и правого поддеревьев вершины a до добавления z были равны, и тем самым у вершины a есть две дочерние вершины c и d .

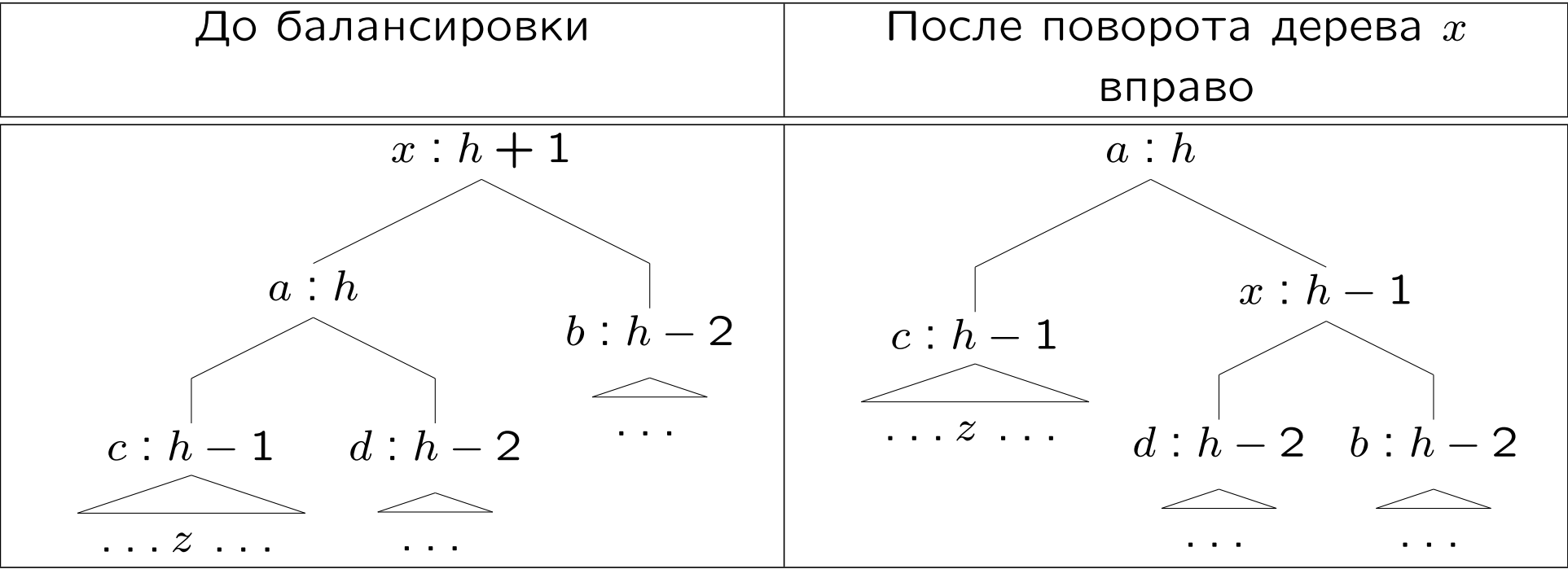
В зависимости от того, в какое поддереву вершины a попадает новая вершина z , возможны два случая:



В первом случае баланс вершины a равен -1 , и, как мы увидим далее, для балансировки дерева x потребуется один поворот.

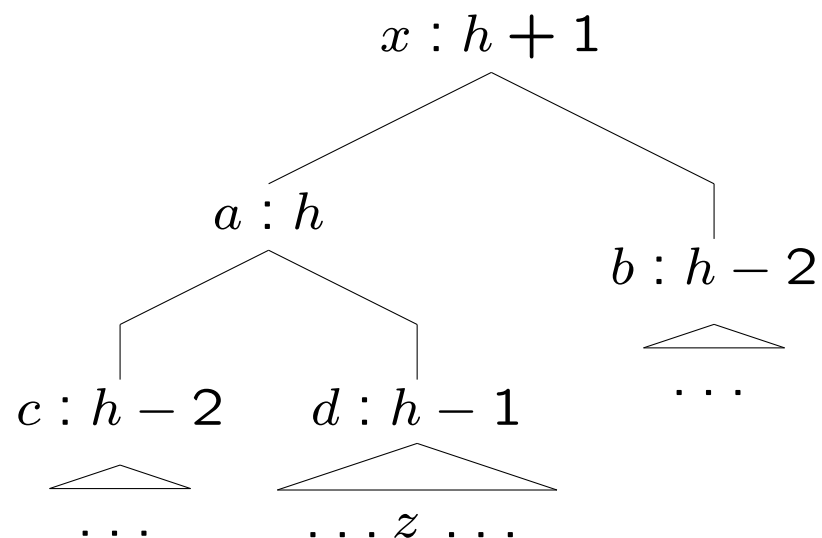
Во втором случае баланс вершины a равен 1 , и потребуется два поворота.

В первом случае балансировка выполняется путём правого поворота дерева x :



В итоге дерево сбалансировано (баланс в корне равен 0), и его высота не поменялась.

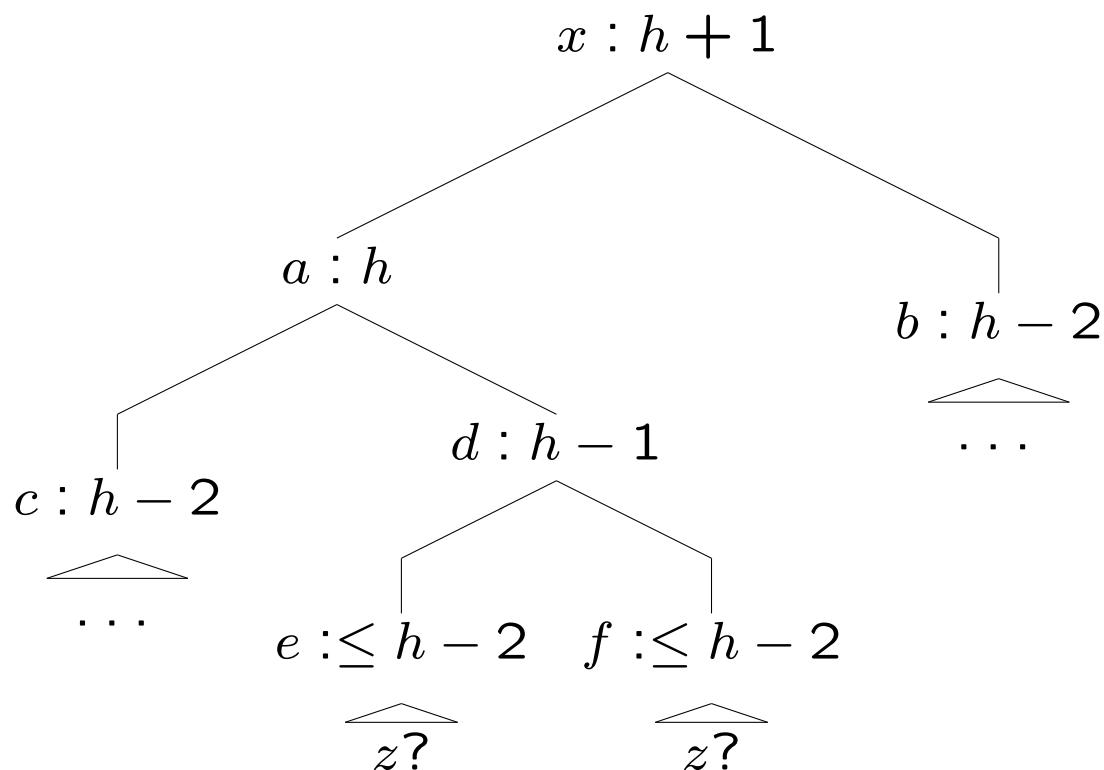
Рассмотрим второй случай:



После добавления вершины z дерево d осталось сбалансированным. Если бы это было не так, мы бы применили к нему процедуру балансировки (согласно гипотезе индукции), в результате которой его высота бы не изменилась, и баланс в вершине x остался бы прежним.

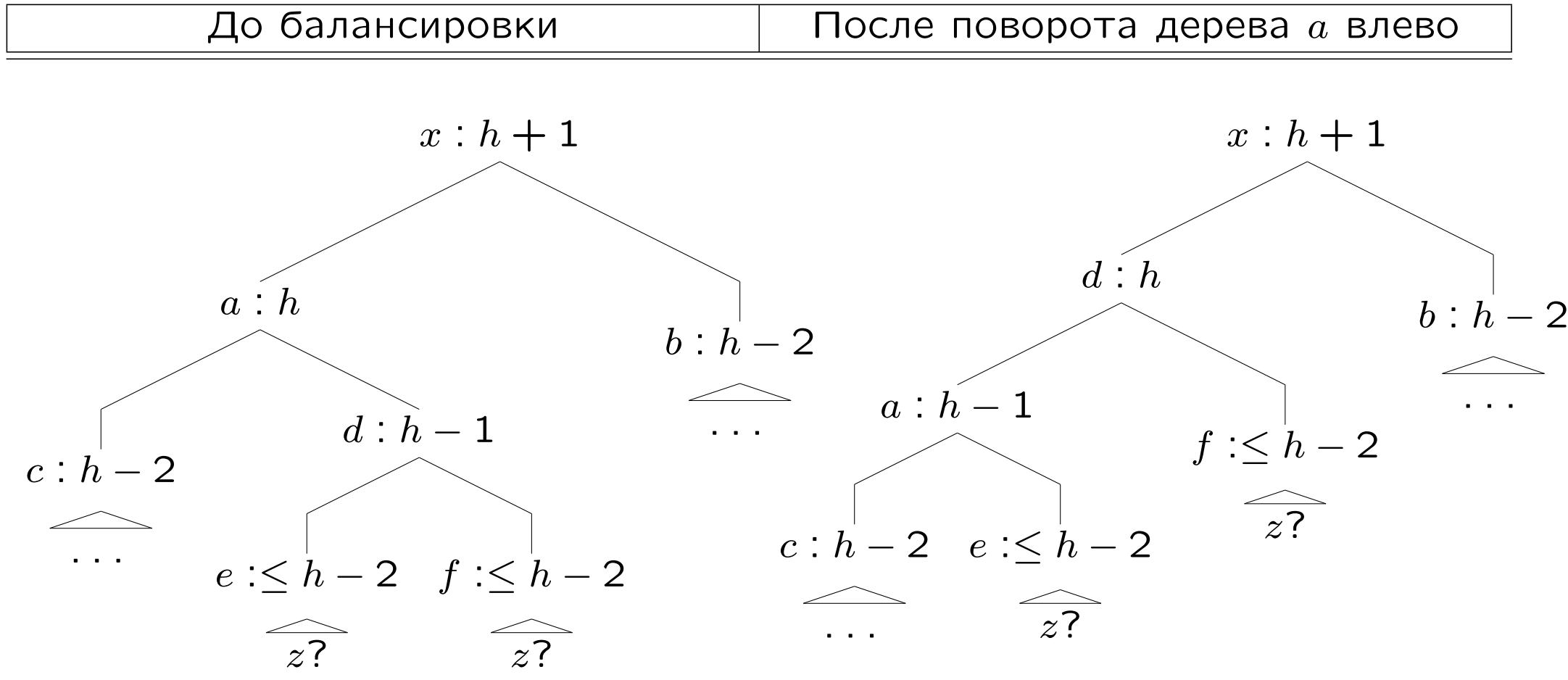
Кроме того, высота дерева d после добавления z выросла, а это означает, что поддеревья дерева d до добавления z имели одинаковую высоту $h - 3$. (Да, при $h = 2$ их высота равна -1 , то есть они просто-напросто пусты.)

Пусть d имеет два дочерних поддерева e и f , которые могут быть пусты. Отразим это на схеме:



Мы не знаем, в какое поддерево — e или f — попадёт вершина z . Для дальнейших рассуждений это неважно. Главное, что высота деревьев e и f после добавления z не превышает $h - 2$.

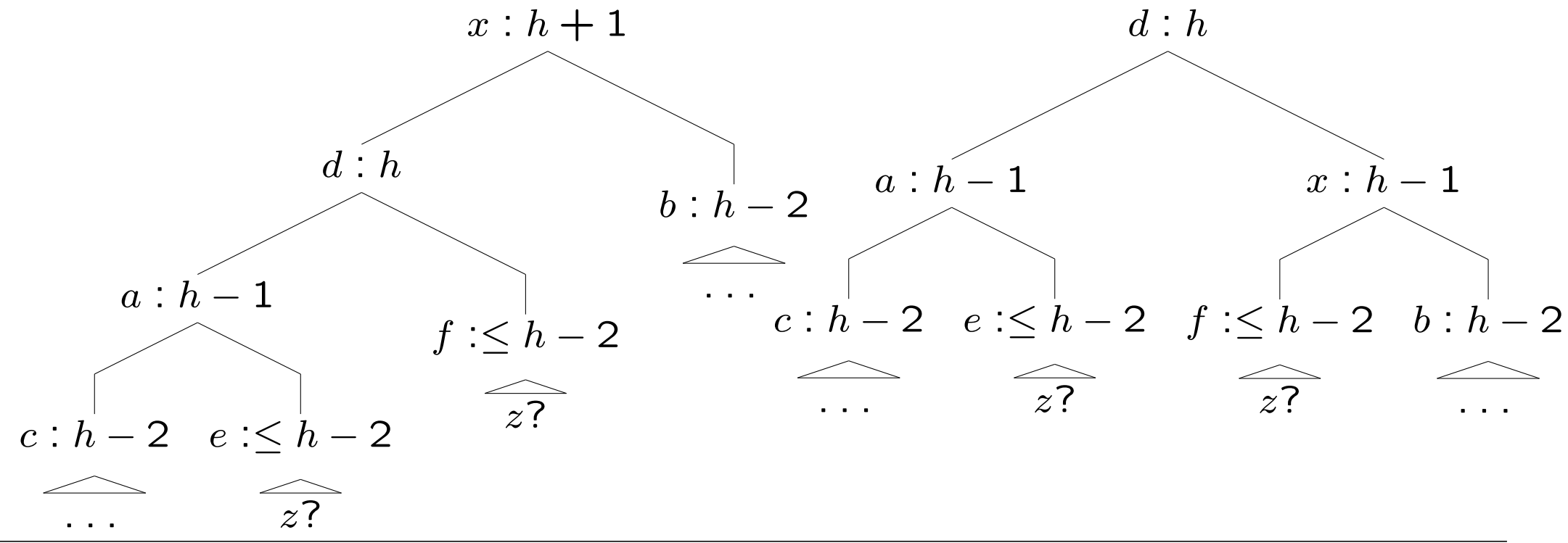
Выполним левый поворот дерева a .



Обратите внимание, что высота дерева a до поворота равна высоте дерева d после поворота. Значит поворот не влияет на баланс вершины x , и мы можем использовать `RotateLeft`.

Теперь выполним правый поворот дерева x .

После первого поворота	После поворота дерева x вправо
------------------------	----------------------------------



В итоге дерево сбалансировано (баланс в корне равен 0), и его высота осталась прежней.

Утверждение доказано.

```

1 InsertAVL(in/out  $t$ , in  $\triangleleft$ , in  $k$ , in  $v$ )
2      $a \leftarrow \text{Insert}(t, \triangleleft, k, v)$ 
3      $a.balance \leftarrow 0$ 
4     loop :
5          $x \leftarrow a.parent$ 
6         if  $x = \text{NULL}$ : break
7         if  $a = x.left$ :
8              $x.balance \leftarrow x.balance - 1$ 
9             if  $x.balance = 0$ : break
10            if  $x.balance = -2$ :
11                if  $a.balance = 1$ : RotateLeft( $t, a$ )
12                RotateRight( $t, x$ )
13                break
14        else :
15             $x.balance \leftarrow x.balance + 1$ 
16            if  $x.balance = 0$ : break
17            if  $x.balance = 2$ :
18                if  $a.balance = -1$ : RotateRight( $t, a$ )
19                RotateLeft( $t, x$ )
20                break
21         $a \leftarrow x$ 

```

§11. Таблица с прямой адресацией

При малой мощности множества ключей допустим очень простой способ реализации ассоциативного массива в виде так называемой таблицы с прямой адресацией.

Пусть $M : K \longrightarrow V$ – ассоциативный массив, и задано взаимнооднозначное соответствие $g : K \longrightarrow \mathbb{N}_m$, где $m = |K|$. При этом биекция g выбрана таким образом, что $k_1 \triangleleft k_2 \Leftrightarrow g(k_1) < g(k_2)$.

Назовём *таблицей с прямой адресацией* массив T размера m такой, что

$$\forall k \in K, T[g(k)] = \begin{cases} \text{NULL}, & \text{если } \nexists \langle k, v \rangle \in M; \\ v, & \text{если } \exists \langle k, v \rangle \in M. \end{cases}$$

Здесь NULL – специальное значение, которое обозначает, что в словаре отсутствует словарная пара с данным ключом.

Реализация операций над таблицей с прямой адресацией тривиальна:

```
1 InitDirectAddressTable(in  $m$ , out  $t$ )
2      $t \leftarrow$  новый массив указателей размера  $m$ 

4 MapEmpty(in  $t$ , in  $m$ ):  $empty$ 
5      $i \leftarrow 0$ 
6     while  $i < m$ :
7         if  $t[i] \neq \text{NULL}$ : return false
8          $i \leftarrow i + 1$ 
9     return true

11 MapSearch(in  $t$ , in  $g$ , in  $k$ ):  $verdict$ 
12      $verdict \leftarrow t[g(k)] \neq \text{NULL}$ 

14 Lookup(in  $t$ , in  $g$ , in  $k$ ):  $v$ 
15      $v \leftarrow t[g(k)]$ 
16     if  $v = \text{NULL}$ :
17         panic
```

Операция MapEmpty выполняется за $O(m)$, если не хранить количество занятых элементов.

Операции Succ и Minimum (а также Prec и Maximum, которые реализуются аналогично) выполняются за $O(m)$.

```
1 Succ(in t, in m, in g, in k): k'
```

```
2      $i \leftarrow g(k) + 1$ 
```

```
3     while  $i < m$  and  $t[i] = \text{NULL}$ :
```

```
4          $i \leftarrow i + 1$ 
```

```
5     if  $i = m$ :
```

```
6         panic
```

```
7      $k' \leftarrow g^{-1}(i)$ 
```

```
9 Minimum(in t, in m, in g): k
```

```
10      $i \leftarrow 0$ 
```

```
11     while  $i < m$  and  $t[i] = \text{NULL}$ :
```

```
12          $i \leftarrow i + 1$ 
```

```
13     if  $i = m$ :
```

```
14         panic
```

```
15      $k \leftarrow g^{-1}(i)$ 
```

Операции Insert, Delete и Reassign выполняются за константное время:

```
1 Insert(in t, in g, in k, in v)
```

```
2      $i \leftarrow g(k)$ 
```

```
3     if  $t[i] \neq \text{NULL}$ :
```

```
4         panic
```

```
5      $t[i] \leftarrow v$ 
```

```
7 Delete(in t, in g, in k)
```

```
8      $i \leftarrow g(k)$ 
```

```
9     if  $t[i] = \text{NULL}$ :
```

```
10        panic
```

```
11     $t[i] \leftarrow \text{NULL}$ 
```

```
13 Reassign(in t, in g, in k, in v)
```

```
14      $i \leftarrow g(k)$ 
```

```
15     if  $t[i] = \text{NULL}$ :
```

```
16        panic
```

```
17     $t[i] \leftarrow v$ 
```


Операция Sequence выполняется за $O(m)$:

```
1 Sequence(in  $t$ , in  $m$ , in  $g$ , in/out  $queue$ )
2      $i \leftarrow 0$ 
3     while  $i < m$ :
4         if  $t[i] \neq \text{NULL}$ :
5             Enqueue( $queue$ ,  $\langle g^{-1}(i), t[i] \rangle$ )
6      $i \leftarrow i + 1$ 
```

§12. Хеш-таблица

Основным отличием хеш-таблицы от таблицы с прямой адресацией является то, что в ней не существует взаимнооднозначного соответствия между ключами и индексами элементов массива, то есть несколько ключей могут отображаться в один и тот же элемент.

Пусть K – множество ключей. Мы будем называть *хеш-функцией* отображение $h : K \longrightarrow \mathbb{N}_m$, где $m < |K|$. При этом случай, когда $k_1 \neq k_2$, но $h(k_1) = h(k_2)$, называется *коллизией*.

В силу возможности коллизий каждый элемент хеш-таблицы должен обеспечивать возможность хранения сразу нескольких словарных пар.

Пусть $M : K \longrightarrow V$ – ассоциативный массив, и задана хеш-функция $h : K \longrightarrow \mathbb{N}_m$. Тогда *хеш-таблица* – это массив T размера m такой, что

$$\forall i \in \mathbb{N}_m : T[i] = \{\langle k, v \rangle \mid h(k) = i\}.$$

Качественная хеш-функция должна равномерно распределять словарные пары по элементам хеш-таблицы. В этом случае время поиска в хеш-таблице равно $O(1 + n/m)$, где n – количество словарных пар, хранящихся в таблице.

При построении хеш-функции нужно руководствоваться следующими принципами:

- функция не должна коррелировать с закономерностями, которым могут подчиняться существующие данные;
- для вычисления функции должен использоваться весь ключ целиком.

Естественно, что хеш-функция не сохраняет порядка на множестве ключей: $k_1 \triangleleft k_2 \not\Rightarrow h(k_1) < h(k_2)$. Поэтому операции Pres, Succ, Minimum, Maximum и Sequence не могут быть реализованы для хеш-таблицы.

Пример. Пусть множество ключей – это множество строк.

Неудачные хеш-функции:

1. $h(k)$ возвращает длину строки k (очевидно, что в общем случае длины строк не равновероятны; кроме того, для вычисления $h(k)$ используется не вся информация, содержащаяся в ключе – игнорируются конкретные значения символов строки);

2. $h(k)$ возвращает код первого символа строки k (недостаток: для вычисления хеш-функции не используются остальные символы строки);

3. $h(k) = \left(\sum_{i=0}^{len(k)-1} k[i] \right) \bmod m$, где m – количество элементов в хеш-таблице (недостаток: $h(k)$ не зависит от перестановки символов в строке).

Удачная хеш-функция:

$h(k) = \left(\sum_{i=0}^{len(k)-1} p^{i+1} \cdot k[i] \right) \bmod m$, где p – простое число.

Рассмотрим реализацию основных операций над хеш-таблицей:

```
1 InitHashTable(in  $m$ , out  $t$ )
2      $t \leftarrow$  новый массив однонаправленных списков размера  $m$ 

4 MapEmpty(in  $t$ , in  $m$ ):  $empty$ 
5      $i \leftarrow 0$ 
6     while  $i < m$ :
7         if not ListEmpty( $t[i]$ ): return false
8          $i \leftarrow i + 1$ 
9     return true

11 MapSearch(in  $t$ , in  $h$ , in  $k$ ):  $verdict$ 
12      $verdict \leftarrow$  ListSearch( $t[h(k)]$ ,  $k$ )  $\neq$  NULL

14 Lookup(in  $t$ , in  $h$ , in  $k$ ):  $v$ 
15      $p \leftarrow$  ListSearch( $t[h(k)]$ ,  $k$ )
16     if  $p = \text{NULL}$ : panic
17      $v \leftarrow p.v$ 
```

Здесь операция ListSearch выполняет поиск элемента списка словарных пар по ключу.

```

1 Insert(in  $t$ , in  $h$ , in  $k$ , in  $v$ )
2      $i \leftarrow h(k)$ 
3     if ListSearch( $t[i]$ ,  $k$ )  $\neq$  NULL:
4         panic
5     InsertBeforeHead( $t[i]$ ,  $\langle k, v \rangle$ )

7 Delete(in  $t$ , in  $h$ , in  $k$ )
8     SearchAndDelete( $t[h(k)]$ ,  $k$ )

10 Reassign(in  $t$ , in  $h$ , in  $k$ , in  $v$ )
11      $p \leftarrow$  ListSearch( $t[h(k)]$ ,  $k$ )
12     if  $p = \text{NULL}$ :
13         panic
14      $p.v \leftarrow v$ 

```

Здесь операция SearchAndDelete должна паниковать в случае, если элемент списка с заданным ключом не найден.

§13. Бор

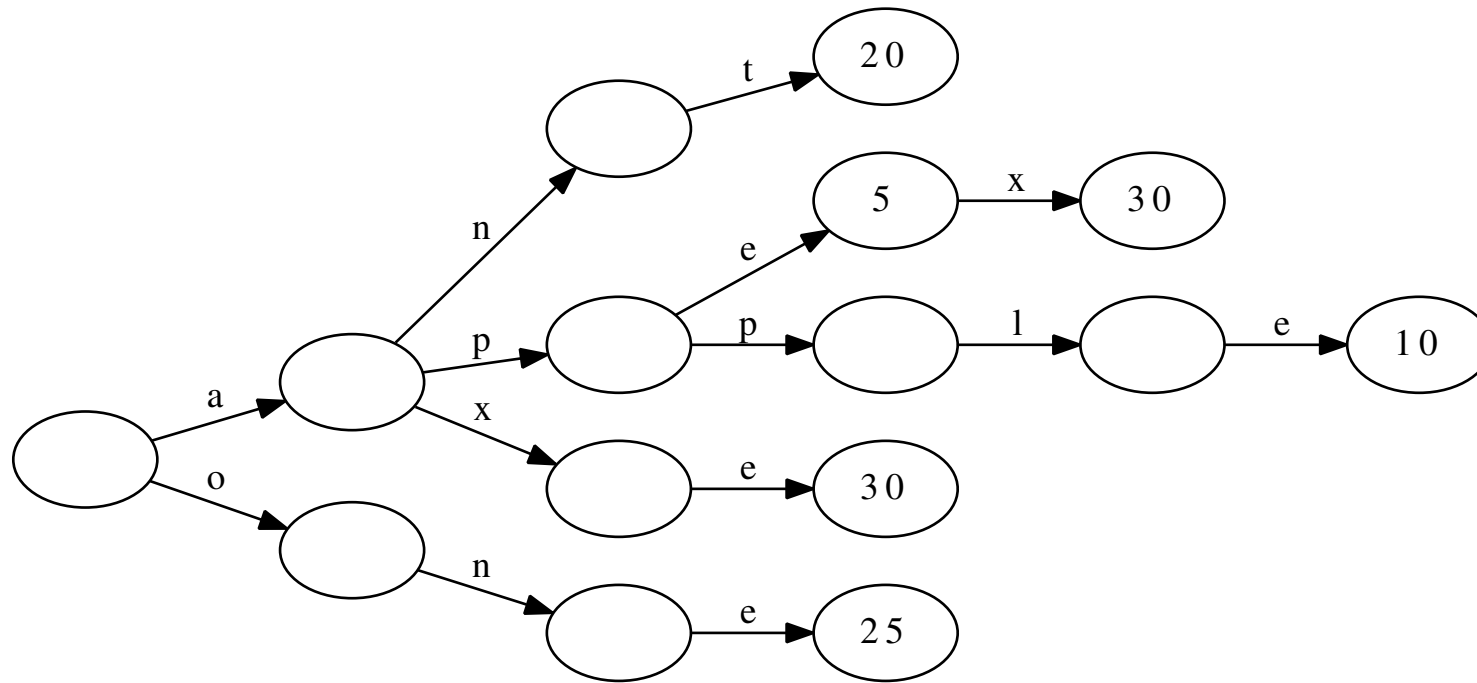
Пусть ключами ассоциативного массива являются строки, т.е. в общем случае конечные последовательности натуральных чисел из алфавита \mathbb{N}_m . (В §§13–14 буква m будет обозначать размер алфавита.)

Если отказаться от модели сравнений и разрешить операциям доступ к ключам, то такой ассоциативный массив можно реализовать в виде бора.

Бор (trie) – это дерево, каждая дуга которого помечена символом из алфавита \mathbb{N}_m . При этом метки всех дуг, исходящих из одной вершины, должны различаться.

Каждая вершина бора соответствует некоторой строке, причём эта строка не хранится в вершине, а определяется последовательностью меток дуг, ведущих от корня бора к этой вершине. Очевидно, что корень бора соответствует пустой строке.

Пример. (Бор, содержащий словарные пары $\langle ant, 20 \rangle$, $\langle ape, 5 \rangle$, $\langle apex, 30 \rangle$, $\langle apple, 10 \rangle$, $\langle axe, 30 \rangle$, $\langle one, 25 \rangle$.)



Вершину бора представляют структурой $\langle v, parent, arcs \rangle$, в которой:

- v – указатель на значение, привязанное к строке, соответствующей вершине (указатель $v = \text{NULL}$, если вершина является служебной);
- $parent$ – указатель на родительскую вершину;
- $arcs$ – массив указателей на дочерние вершины (если из вершины исходит дуга, помеченная символом x , то $arcs[x]$ содержит указатель на вершину, в которую эта дуга входит; в противном случае $arcs[x] = \text{NULL}$).

Рассмотрим реализацию основных операций над бором.

Инициализация бора заключается в создании корневой вершины, имеющей статус служебной ($v = \text{NULL}$):

```
1 InitTrie(out  $t$ )
2      $t \leftarrow$  новая вершина
```

Замечание. Запись «новая вершина» означает выделение памяти для структуры вершины и заполнение этой памяти нулями.

Бор пуст тогда и только тогда, когда его корень – служебный, и из него не исходит ни одной дуги:

```
1 MapEmpty(in  $t$ ): empty
2     if  $t.v \neq \text{NULL}$ : return false
3      $i \leftarrow 0$ 
4     while  $i < m$ :
5         if  $t.\text{arcs}[i] \neq \text{NULL}$ : return false
6          $i \leftarrow i + 1$ 
7     return true
```

Вспомогательная операция Descend выполняет спуск от корня бора до вершины x , соответствующей максимальному префиксу строки k , присутствующему в боре. Операция возвращает указатель на вершину x и длину соответствующего ей префикса строки k .

```
1 Descend(in t, in k): x, i
2      $x \leftarrow t, \quad i \leftarrow 0$ 
3     while  $i < \text{len}(k)$ :
4          $y \leftarrow x.\text{arcs}[k[i]]$ 
5         if  $y = \text{NULL}$ : break
6          $x \leftarrow y$ 
7          $i \leftarrow i + 1$ 
```

Операция Descend аналогична одноимённой операции бинарного дерева поиска, но работает за время $O(\text{len}(k))$. Тем самым, спуск по бору не зависит от количества словарных пар в нём.

Отметим, что высота бора равна длине самого длинного ключа в нём, и бор не нуждается в балансировке.

Операции MapSearch и Lookup для бора используют в своей работе операцию Descend и, соответственно, тоже работают за время $O(\text{len}(k))$:

```
1 MapSearch(in  $t$ , in  $k$ ):  $verdict$ 
2      $x, i \leftarrow \text{Descend}(t, k)$ 
3      $verdict \leftarrow i = \text{len}(k)$  and  $x.v \neq \text{NULL}$ 

5 Lookup(in  $t$ , in  $k$ ):  $v$ 
6      $x, i \leftarrow \text{Descend}(t, k)$ 
7     if  $i \neq \text{len}(k)$  or  $x.v = \text{NULL}$ : panic
8      $v \leftarrow x.v$ 
```

Обратите внимание на то, что Descend по ключу k может вернуть указатель на служебную вершину, не соответствующую ни одной словарной паре. Поэтому приходится проверять, пусто ли поле v найденной вершины.

Операция Minimum возвращает указатель на вершину, ключ которой лексикографически наименьший. Для этого выполняется спуск от корня бора до первой неслужебной вершины по самым левым дугам:

```
1 Minimum(in  $t$ ):  $x$ 
2     if  $t.v \neq \text{NULL}$ :
3         return  $t$ 
4      $i \leftarrow 0$ 
5     while  $i < m$ :
6         if  $t.\text{arcs}[i] \neq \text{NULL}$ :
7             return Minimum( $t.\text{arcs}[i]$ )
8          $i \leftarrow i + 1$ 
9     return NULL
```

Время работы операции Minimum – $O(h)$, где h – высота бора.

Отметим, что в операции Minimum используется хвостовая рекурсия, которая может быть элементарно превращена в цикл.

Операция Maximum возвращает указатель на вершину, ключ которой лексикографически наибольший. Для этого выполняется спуск от корня бора до листовой вершины по самым правым дугам:

```
1 Maximum( in  $t$  ):  $x$ 
2      $i \leftarrow m - 1$ 
3     while  $i \geq 0$  :
4         if  $t.arcs[i] \neq \text{NULL}$  :
5             return Maximum(  $t.arcs[i]$  )
6          $i \leftarrow i - 1$ 
7     if  $t.v = \text{NULL}$  : — Может быть в корне пустого бора
8         return NULL
9     return  $t$ 
```

Время работы операции Maximum — $O(h)$, где h — высота бора. В операции Maximum также используется хвостовая рекурсия.

Операции Insert и Reassign выполняются за время $O(\text{len}(k))$:

```
1 Insert(in t, in k, in v): x
2      $x, i \leftarrow \text{Descend}(t, k)$ 
3     if  $i = \text{len}(k)$  and  $x.v \neq \text{NULL}$ :
4         panic
5     while  $i < \text{len}(k)$ :
6          $y \leftarrow$  новая вершина
7          $x.\text{arcs}[k[i]] \leftarrow y$ 
8          $y.\text{parent} \leftarrow x$ 
9          $x \leftarrow y$ 
10         $i \leftarrow i + 1$ 
11     $x.v \leftarrow v$ 

13 Reassign(in t, in k, in v)
14     $x, i \leftarrow \text{Descend}(t, k)$ 
15    if  $i \neq \text{len}(k)$  or  $x.v = \text{NULL}$ :
16        panic
17     $x.v \leftarrow v$ 
```

Операция Delete выполняется за время $O(\text{len}(k))$:

```
1 Delete(in t, in k)
2      $x, i \leftarrow \text{Descend}(t, k)$ 
3     if  $i \neq \text{len}(k)$  or  $x.v = \text{NULL}$ :
4         panic
5      $x.v \leftarrow \text{NULL}$ 
6     while  $x.\text{parent} \neq \text{NULL}$  and  $x.v = \text{NULL}$ :
7          $j \leftarrow 0$ 
8         while  $j < m$  and  $x.\text{arcs}[j] = \text{NULL}$ :
9              $j \leftarrow j + 1$ 
10        if  $j < m$ : break
11         $y \leftarrow x.\text{parent}$ 
12         $i \leftarrow i - 1$ 
13         $y.\text{arcs}[k[i]] \leftarrow \text{NULL}$ 
14        освободить память, занимаемую вершиной  $x$ 
15         $x \leftarrow y$ 
```

Без строчек 6..15 алгоритм тоже работает, но не освобождается память, и могут появиться служебные листовые вершины, что «поломает» реализацию операций Minimum и Maximum.

Рекурсивное формирование лексикографически отсортированной последовательности словарных пар бора:

```
1 Sequence(in  $t$ , in/out  $queue$ )
2     if  $t \neq \text{NULL}$ :
3         if  $t.v \neq \text{NULL}$ :
4             Enqueue( $queue$ ,  $t$ )
5          $i \leftarrow 0$ 
6         while  $i < m$ :
7             Sequence( $t.arcs[i]$ ,  $queue$ )
8              $i \leftarrow i + 1$ 
```

Проверка в строчке 3 нужна для того, чтобы в последовательность не попадали служебные вершины.

§14. Сжатый бор

Реализация бора требует $O(qm)$ памяти, где q – суммарная длина всех ключей, а m – размер алфавита. Действительно, в худшем случае, когда все ключи начинаются с разных букв, количество вершин в боре будет равно $q + 1$.

Сжатый бор (compact trie) – это дерево, каждая дуга которого помечена строкой. При этом первые буквы строк, которыми помечены дуги, исходящие из одной вершины, должны различаться. Кроме того, из нелистовой вершины сжатого бора исходит не менее двух дуг.

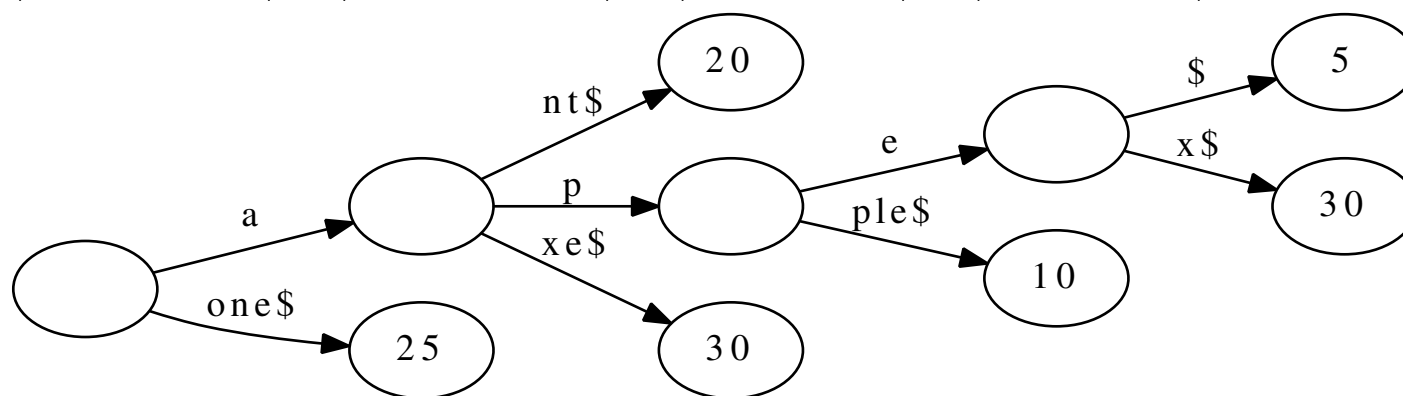
Фактически, сжатый бор – это бор, в котором вершины, из которых исходит только одна дуга, объединены со своими дочерними вершинами.

Сжатый бор позволяет сократить размер требуемой памяти до $O(nm + q)$, где n – количество ключей.

Ключи, содержащиеся в сжатом боре, не могут быть префиксами друг друга. Действительно, если ключ заканчивается в нелистовой вершине, и из этой вершины исходит только одна дуга, вершину придётся объединить с её дочерней вершиной, что лишит словарную пару места для хранения её значения.

Поэтому мы будем добавлять в конец каждого ключа специальный символ \$, который не может содержаться в другом месте ключа. Для обеспечения лексикографического порядка на множестве ключей будем считать, что символ \$ предшествует всем символам алфавита (имеет нулевой код).

Пример. (Сжатый бор, содержащий словарные пары $\langle ant$, 20 \rangle$, $\langle ape$, 5 \rangle$, $\langle apex$, 30 \rangle$, $\langle apple$, 10 \rangle$, $\langle axe$, 30 \rangle$, $\langle one$, 25 \rangle$.)

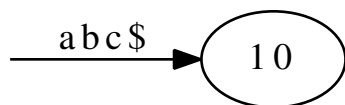


Из того, что ключи в сжатом боре не могут быть префиксами друг друга, следует, что ключи оканчиваются в листовых вершинах, а служебные вершины всегда промежуточные.

Пустой сжатый бор не может быть представлен единственной вершиной, которая одновременно является и корнем, и листом, потому что листовая вершина не может быть служебной. Поэтому пусть пустой сжатый бор вообще не содержит ни одной вершины.

Сжатый бор с единственной словарной парой не содержит «развилок», а значит должен быть представлен единственной вершиной. Спрашивается, откуда в этом случае взять дугу, помеченную ключом этой словарной пары? Для решения этого вопроса разрешим существование дуги, входящей в корень бора.

Пример. (Сжатый бор, содержащий единственную пару $\langle abc$, 10 \rangle$.)



Вершину сжатого бора мы будем представлять структурой

$\langle v, label, parent, arcs \rangle$, в которой:

- v – указатель на значение, привязанное к ключу, соответствующему вершине (указатель $v = \text{NULL}$, если вершина является служебной);
- $label$ – метка дуги, входящей в вершину;
- $parent$ – указатель на родительскую вершину (NULL в случае корня);
- $arcs$ – массив указателей на дочерние вершины (если из вершины исходит дуга, метка которой начинается с символа x , то $arcs[x]$ содержит указатель на вершину, в которую эта дуга входит; в противном случае $arcs[x] = \text{NULL}$).

Так как пустой сжатый бор не содержит вершин, операции инициализации и проверки сжатого бора на пустоту тривиальны:

```
1 InitCompactTrie(out t)
2     t ← NULL

4 MapEmpty(in t): empty
5     empty ← t = NULL
```

Для обозначения позиции, до которой мы доходим в сжатом боре в процессе спуска от корня, мы будем использовать структуру $\langle x, i \rangle$, в которой x – указатель на вершину сжатого бора, i – расстояние (в символах), которое мы прошли по дуге, входящей в вершину x .

Вспомогательная операция Descend в случае сжатого бора приобретает вид

```
1 Descend ( in  $t$  , in  $k$  ) :  $pos$  ,  $i$ 
2      $pos \leftarrow \langle t, 0 \rangle$ 
3      $i \leftarrow 0$ 
4     loop :
5          $l \leftarrow pos.x.label$ 
6         while  $i < len(k)$  and  $pos.i < len(l)$  and  $k[i] = l[pos.i]$  :
7              $i \leftarrow i + 1$  ,  $pos.i \leftarrow pos.i + 1$ 
8         if  $i = len(k)$  or  $pos.i \neq len(l)$  : break
9          $y \leftarrow pos.x.arcs[k[i]]$ 
10        if  $y = \text{NULL}$  : break
11         $pos \leftarrow \langle y, 0 \rangle$ 
```

```

1 DescendToLeaf(in  $t$ , in  $k$ ):  $x$ 
2     if  $t = \text{NULL}$ : panic
3      $\text{pos}, i \leftarrow \text{Descend}(t, k)$ 
4     if  $i \neq \text{len}(k)$  or  $\text{pos}.i \neq \text{len}(\text{pos}.x.\text{label})$  or  $\text{pos}.x.v = \text{NULL}$ :
5         panic
6      $x \leftarrow \text{pos}.x$ 

8 MapSearch(in  $t$ , in  $k$ ):  $\text{verdict}$ 
9     if  $t = \text{NULL}$ :
10         return false
11      $\text{pos}, i \leftarrow \text{Descend}(t, k)$ 
12      $\text{verdict} \leftarrow i = \text{len}(k)$  and  $\text{pos}.i = \text{len}(\text{pos}.x.\text{label})$  and
13          $\text{pos}.x.v \neq \text{NULL}$ 

15 Lookup(in  $t$ , in  $k$ ):  $v$ 
16      $x \leftarrow \text{DescendToLeaf}(t, k)$ 
17      $v \leftarrow x.v$ 

19 Reassign(in  $t$ , in  $k$ , in  $v$ )
20      $x \leftarrow \text{DescendToLeaf}(t, k)$ 
21      $x.v \leftarrow v$ 

```

```

1 Minimum(in  $t$ ):  $x$ 
2      $x \leftarrow t$ 
3     if  $x \neq \text{NULL}$ :
4         while  $x.v = \text{NULL}$ :
5              $i \leftarrow 0$ 
6             while  $x.\text{arcs}[i] = \text{NULL}$ :
7                  $i \leftarrow i + 1$ 
8              $x \leftarrow x.\text{arcs}[i]$ 

```

Maximum – аналогично.

```

1  Link(in/out  $t$ , in  $parent$ , in  $child$ )
2       $child.parent \leftarrow parent$ 
3      if  $parent = \text{NULL}$ :  $t \leftarrow child$ 
4      else:  $parent.arcs[child.label[0]] \leftarrow child$ 

6  Split(in/out  $t$ , in/out  $pos$ )
7       $l \leftarrow pos.x.label$ 
8      if  $pos.i < len(l)$ :
9           $y \leftarrow \text{новая вершина}$ 
10          $y.label \leftarrow l[0 : pos.i - 1]$ 
11          $pos.x.label \leftarrow l[pos.i : len(l) - 1]$ 
12         Link(in/out  $t$ ,  $pos.x.parent$ ,  $y$ )
13         Link(in/out  $t$ ,  $y$ ,  $pos.x$ )
14          $pos.x \leftarrow y$ 

```



```

1 Insert(in/out  $t$ , in  $k$ , in  $v$ ):  $x$ 
2      $x \leftarrow$  новая вершина
3      $x.v \leftarrow v$ 
4     if  $t = \text{NULL}$ :
5          $x.\text{label} \leftarrow k$ 
6          $t \leftarrow x$ 
7     else :
8          $\text{pos}, i \leftarrow \text{Descend}(t, k)$ 
9         if  $i = \text{len}(k)$ : panic

11         Split(in/out  $t$ , in/out  $\text{pos}$ )
12          $x.\text{label} \leftarrow k[i : \text{len}(k) - 1]$ 
13         Link(in/out  $t$ ,  $\text{pos}.x$ ,  $x$ )

```

```

1 Delete(in/out  $t$ , in  $k$ )
2      $x \leftarrow \text{DescendToLeaf}(t, k)$ 
3      $l \leftarrow x.\text{label}$ 
4      $y \leftarrow x.\text{parent}$ 
5     освободить память, занимаемую вершиной  $x$ 
6     if  $y = \text{NULL}$ :  $t \leftarrow \text{NULL}$ 
7     else :
8          $y.\text{arcs}[l[0]] \leftarrow \text{NULL}$ 
9          $z \leftarrow \text{NULL}$ ,  $i \leftarrow 0$ 
10        while  $i < m$  :
11            if  $y.\text{arcs}[i] \neq \text{NULL}$  :
12                if  $z \neq \text{NULL}$ : return
13                 $z \leftarrow y.\text{arcs}[i]$ 
14             $i \leftarrow i + 1$ 
15
16         $z.\text{label} \leftarrow y.\text{label} + z.\text{label}$ 
17        Link(in/out  $t$ ,  $y.\text{parent}$ ,  $z$ )
18        освободить память, занимаемую вершиной  $y$ 

```

§15. Система непересекающихся множеств

Система непересекающихся множеств – это абстрактная структура данных, представляющая множество $\{\sigma_i\}_0^{n-1}$, элементами которого являются непустые конечные непересекающиеся множества.

Чаще всего эта структура данных применяется для решения задачи разбиения некоторого конечного множества на классы эквивалентности.

Представитель множества σ_i – это один из элементов σ_i , выбранный для обозначения всего множества. Мы будем считать, что представитель множества выбирается произвольно и обозначать его как $\text{repr}(\sigma_i)$.

Пусть G – универсальное множество, из элементов которого будут составлены наши непересекающиеся множества.

Пусть S – множество систем непересекающихся множеств.

Будем рассматривать систему непересекающихся множеств $\{\sigma_i\}_0^{n-1}$ как частичную функцию $s : G \rightarrow G$ такую, что $(\forall x \in \sigma_i) : s(x) = \text{repr}(\sigma_i)$.

Определим три операции для системы непересекающихся множеств:

1. $\text{MakeSet} : S \times G \rightarrow S$ – добавление в систему нового множества, состоящего из одного элемента.

$$\text{MakeSet}(s, x) = s \cup \langle x, x \rangle.$$

(Ограничение: $x \notin \text{dom } s$.)

2. $\text{Union} : S \times G \times G \rightarrow S$ – объединение двух множеств, каждое из которых идентифицируется одним из своих элементов.

$$\text{Union}(s, x, y) = (s \setminus (w \times s(y))) \cup (w \times s(x)), \text{ где } w = \{z \mid s(z) = s(y)\}.$$

(Ограничение: $x \in \text{dom } s, y \in \text{dom } s$.)

3. $\text{Find} : S \times G \rightarrow G$ – поиск представителя множества, которому принадлежит указанный элемент.

$$\text{Find}(s, x) = s(x).$$

(Ограничение: $x \in \text{dom } s$.)

§16. Лес непересекающихся множеств

Лес непересекающихся множеств является одной из реализаций системы непересекающихся множеств.

Основная идея данной реализации состоит в том, что каждое множество σ_i из системы $\{\sigma_i\}_0^{n-1}$ представлено в виде дерева, вершинами которого являются элементы σ_i .

Вершина дерева σ_i может иметь любое количество дочерних вершин. Более того, элементы множества σ_i могут как угодно располагаться в дереве. Единственное ограничение – представитель σ_i должен быть корнем.

Каждая вершина дерева σ_i в памяти компьютера представляется в виде структуры $\langle x, parent \rangle$, в которой x – элемент σ_i , а $parent$ – указатель на родительскую вершину. Родительской вершиной для корня дерева является она сама.

Наивная реализация операций:

```
1 MakeSet(in  $x$ ):  $t$ 
2      $t \leftarrow$  новая вершина
3      $t.x \leftarrow x$ 
4      $t.parent \leftarrow t$ 

6 Union(in  $x$ , in  $y$ )
7      $root_x \leftarrow \text{Find}(x)$ 
8      $root_y \leftarrow \text{Find}(y)$ 
9      $root_x.parent \leftarrow root_y$ 

11 Find(in  $x$ ):  $root$ 
12     if  $x.parent = x$ :
13          $root \leftarrow x$ 
14     else :
15          $root \leftarrow \text{Find}(x.parent)$ 
```

Замечание. Union работает, даже если x и y принадлежат одному дереву.

В наивной реализации MakeSet работает за $O(1)$, а Union и Find — за $O(n)$, где n — общее количество вершин в лесу. Действительно, в худшем случае все вершины дерева могут образовывать однонаправленный список (максимально несбалансированное дерево).

Наивную реализацию операций можно оптимизировать, если при объединении двух деревьев делать корень менее глубокого дерева дочерней вершиной корня более глубокого дерева. Эта оптимизация называется *эвристикой объединения по глубине*.

Для реализации этой эвристики понадобится добавить в каждую вершину поле `depth`, задающее глубину её поддеревя.

Эвристика объединения по глубине требует переписывания кода операций MakeSet и Union:

```
1 MakeSet(in  $x$ ):  $t$ 
2      $t \leftarrow$  новая вершина
3      $t.x \leftarrow x$ ,  $t.depth \leftarrow 0$ ,  $t.parent \leftarrow t$ 

5 Union(in  $x$ , in  $y$ )
6      $root_x \leftarrow \text{Find}(x)$ 
7      $root_y \leftarrow \text{Find}(y)$ 
8     if  $root_x.depth < root_y.depth$ :
9          $root_x.parent \leftarrow root_y$ 
10    else :
11         $root_y.parent \leftarrow root_x$ 
12        if  $root_x.depth = root_y.depth$  and  $root_x \neq root_y$ :
13             $root_x.depth \leftarrow root_x.depth + 1$ 
```

Можно показать, что операции Union и Find теперь работают за время $O(\lg n)$, где n – общее количество вершин в лесу.

Время работы операций можно существенно улучшить, если в операции Find делать все пройденные вершины дочерними вершинами корня дерева (*эвристика сжатия пути*).

```
1 Find(in  $x$ ):  $root$ 
2     if  $x.parent = x$ :
3          $root \leftarrow x$ 
4     else :
5          $root \leftarrow x.parent \leftarrow \text{Find}(x.parent)$ 
```

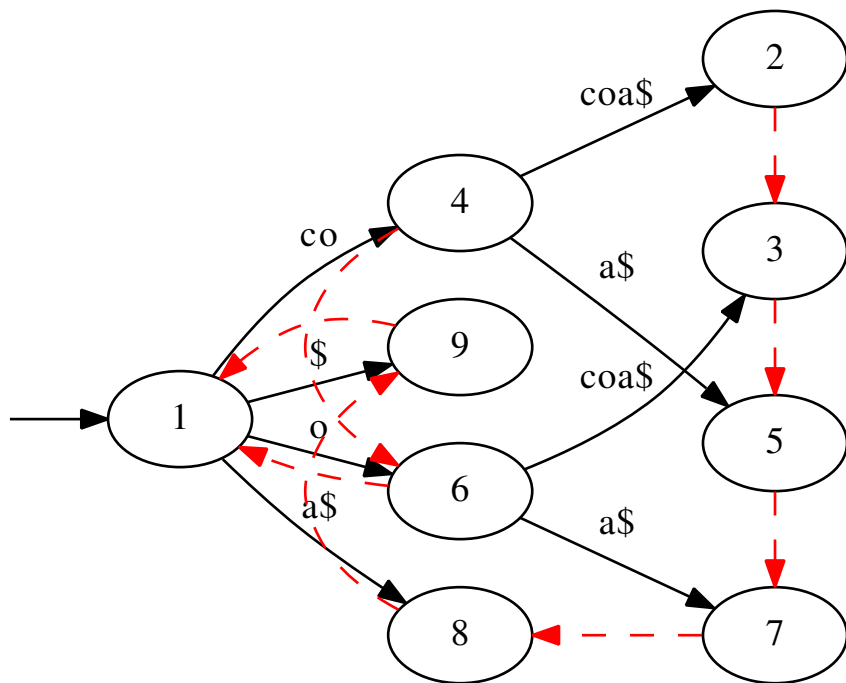
Замечание. Применение эвристики сжатия пути приводит к тому, что значение поля `depth` в вершине дерева может превышать реальную глубину растущего из этой вершины поддерева.

Существует достаточно сложное доказательство того, что совместное применение эвристики объединения по глубине и эвристики сжатия пути приводит к тому, что операции Union и Find начинают работать за амортизированное время $O(\alpha(n))$, где α — чрезвычайно медленно растущая функция такая, что $\alpha(n) < 5$ для всех разумных значений n .

§17. Суффиксное дерево (черновик)

Суффиксное дерево – это модификация сжатого бора, предназначенная для хранения всех суффиксов некоторой строки.

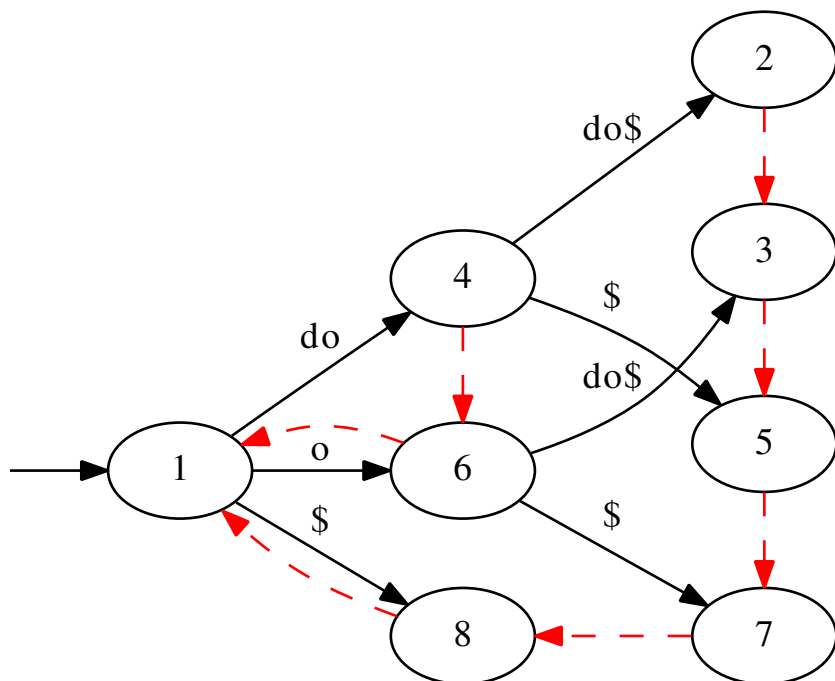
Пример. (Суффиксное дерево для строки «сосоа».)



В вершинах – просто их порядковые номера. Красные стрелки – суффиксные связи.

Суффиксная связь – дополнительная дуга, соединяющая вершину, соответствующую строке aX , с вершиной, соответствующей строке X . (Здесь a – буква, а X – строка.)

Пример. (Суффиксное дерево для строки «dodo».)



Суффиксные связи играют вспомогательную роль при построении дерева.

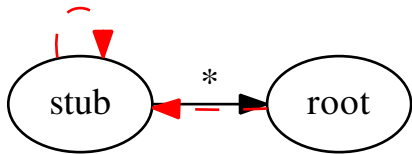
Вершину суффиксного дерева мы будем представлять структурой $\langle label, parent, suf, arcs \rangle$, в которой:

- *label* – метка дуги, входящей в вершину;
- *parent* – указатель на родительскую вершину;
- *suf* – указатель на вершину, соответствующую самому длинному собственному суффиксу данной вершины (суффиксная связь);
- *arcs* – массив указателей на дочерние вершины (если из вершины исходит дуга, метка которой начинается с символа x , то $arcs[x]$ содержит указатель на вершину, в которую эта дуга входит; в противном случае $arcs[x] = \text{NULL}$).

Пустое суффиксное дерево будет представлено корневой вершиной и специальной вершиной-ограничителем, упрощающей запись алгоритмов.

Все элементы массива *arcs* ограничителя должны содержать указатели на корень. При этом ограничитель является родителем корня, и в корень входит «псевдодуга» длиной в один символ, значение которого не важно.

Кроме того, ограничитель будет являться суффиксом корня, а также суффиксом самого себя.



Позиции в дереве, как и в случае сжатого бора, будут представляться парами $\langle x, i \rangle$, в которых x — указатель на вершину дерева, i — расстояние (в символах), которое мы прошли по дуге, входящей в вершину x .

Инициализация суффиксного дерева в соответствии со схемой, приведённой на предыдущем слайде, выглядит следующим образом:

```
1 InitSuffixTree(out root)
2     stub ← новая вершина
3     root ← новая вершина
4     stub.label ← пустая строка
5     stub.parent ← NULL
6     stub.suf ← stub
7     for each c ∈ алфавит:
8         stub.arcs[c] ← root
9     root.label ← строка из одного символа
10    root.parent ← stub
11    root.suf ← stub
```

Алгоритм InitSuffixTree возвращает указатель на корень пустого дерева.

Вспомогательная операция Descend для суффиксного дерева выполняет спуск из произвольной позиции $start$ в соответствии со строкой k и возвращает самую глубокую позицию pos , до которой можно спуститься, а также количество i пройденных символов строки:

```

1 Descend ( in   $start$  ,  in   $k$  ) :   $pos$  ,   $i$ 
2       $pos \leftarrow start$  ,   $i \leftarrow 0$ 
3      loop :
4           $l \leftarrow pos.x.label$ 
5          while   $i < len(k)$   and   $pos.i < len(l)$   and   $k[i] = l[pos.i]$  :
6               $i \leftarrow i + 1$  ,   $pos.i \leftarrow pos.i + 1$ 
7              if   $i = len(k)$   or   $pos.i \neq len(l)$  :  break
8               $y \leftarrow pos.x.arcs[k[i]]$ 
9              if   $y = \text{NULL}$  :  break
10              $pos \leftarrow \langle y, 1 \rangle$ 
11              $i \leftarrow i + 1$ 

```

Операцию Descend можно запускать в том числе и из вершины-ограничителя

Основной тип запросов к суффиксному дереву заключается в поиске некоторой подстроки в строке, на основании которой дерево построено.

Очевидно, что если подстрока S входит в строку T , то какой-то суффикс строки T начинается с S .

Поэтому, если для строки T построено суффиксное дерево с корнем $root$, то с позиции $\langle root, 1 \rangle$ в дереве начинается любая её подстрока. Тем самым, проверку вхождения подстроки S в T можно записать как

```
1 Contains(in  $root$ , in  $s$ ):  $verdict$   
2      $pos, i \leftarrow \text{Descend}(\langle root, 1 \rangle, s)$   
3      $verdict \leftarrow i = \text{len}(s)$ 
```


Вспомогательная операция Link делает вершину *child* дочерней по отношению к вершине *parent*:

```
1 Link ( in parent , in child )  
2     child.parent  $\leftarrow$  parent  
3     parent.arcs[child.label[0]]  $\leftarrow$  child
```

Операция Split вставляет новую вершину в позиции *pos*, если эта позиция находится в середине дуги. Она возвращает булевское значение, показывающее, понадобилось ли создание новой вершины:

```
1 Split ( in / out pos ) : splitted  
2     splitted  $\leftarrow$  pos.i < len(pos.x.label)  
3     if splitted :  
4         y  $\leftarrow$  новая вершина  
5         y.label  $\leftarrow$  pos.x.label[0 : pos.i - 1]  
6         pos.x.label  $\leftarrow$  pos.x.label[pos.i : len(l) - 1]  
7         Link ( pos.x.parent , y )  
8         Link ( y , pos.x )  
9         pos.x  $\leftarrow$  y
```

```

1  ImSuf( in  pos ):  sufPos
2       $l \leftarrow pos.x.label$  ,   $x \leftarrow pos.x.parent.suf$ 
3      if   $x = pos.x.parent$  :           ; если pos показывает на корень
4           $sufPos \leftarrow \langle x, 0 \rangle$            ; вернуть ограничитель
5      else :
6          ; иначе быстро спустаться по дереву,
7          ; руководствуясь только первыми символами дуг
8           $i \leftarrow 0$ 
9          loop :
10              $x \leftarrow x.arcs[l[i]]$ 
11             if   $pos.i - i \leq len(x.label)$  :  break
12              $i \leftarrow i + len(x.label)$ 
13          $sufPos \leftarrow \langle x, pos.i - i \rangle$ 

```

```

1 Append(in root , in s)
2     pos  $\leftarrow \langle root, 1 \rangle$  , top  $\leftarrow$  новая вершина , prev  $\leftarrow$  NULL , i  $\leftarrow$  0
3     loop :
4         splitted  $\leftarrow$  Split (in / out pos)
5         if prev  $\neq$  NULL : prev.suf  $\leftarrow$  pos.x , prev  $\leftarrow$  NULL
6         j  $\leftarrow$  0
7         if not splitted :
8             pos, j  $\leftarrow$  Descend (pos , s[i : len(s) - 1])
9             i  $\leftarrow$  i + j
10            if i = len(s) : top.suf  $\leftarrow$  pos.x , break
11        if j = 0 :
12            top.suf  $\leftarrow$  новая вершина
13            top  $\leftarrow$  top.suf
14            top.label  $\leftarrow$  s[i : len(s) - 1]
15            Link (pos.x , top)
16            prev  $\leftarrow$  pos.x
17            pos  $\leftarrow$  ImSuf (pos)

```

§18. Дерево критических битов

Размер сжатого бора в памяти можно уменьшить до $O(n \cdot \lg m + q)$, где n – количество словарных пар, q – суммарная длина всех ключей, а m – размер алфавита.

Для этого достаточно представить каждый символ алфавита в двоичной системе счисления. Тем самым мы сократим алфавит до двух символов, в результате чего ключи превратятся в последовательности нулей и единиц, а сжатый бор станет бинарным (из нелистовых вершин будут исходить ровно две дуги).

Существует большое количество реализацией бинарного сжатого бора. Первая реализация – PATRICIA trees (Дональд Моррисон, 1968).

Наиболее компактной реализацией является так называемое дерево критических битов (Дэниэл Дж. Бернстайн, 2004).

Листовые и промежуточные вершины дерева критических бит представлены в памяти по-разному: листовая вершина содержит пару $\langle k, v \rangle$, а промежуточная – тройку $\langle pos, left, right \rangle$.

Здесь:

- k и v – ключ и значение словарной пары;
- pos – длина в битах строки, соответствующей промежуточной вершине;
- $left$ и $right$ – указатели на вершины, в которые входят дуги, метки которых начинаются на 0 и 1, соответственно.

Для работы дерева критических бит важно по указателю на вершину уметь понять, листовая она или промежуточная. При записи алгоритмов мы будем использовать «волшебный» предикат $is_leaf(ptr)$, отвечающий на вопрос, указывает ли ptr на листовую вершину.

Бернстайн рекомендует размещать вершины по чётным адресам, чтобы иметь возможность использовать младший бит указателя для индикации типа вершины, на которую этот указатель указывает. Тогда is_leaf должен проверять, установлен ли младший бит указателя.