

# Лекция 3.

## Косые деревья. Списки с пропусками.

## Префиксные деревья



**Даниил Михайлович Берлиз**

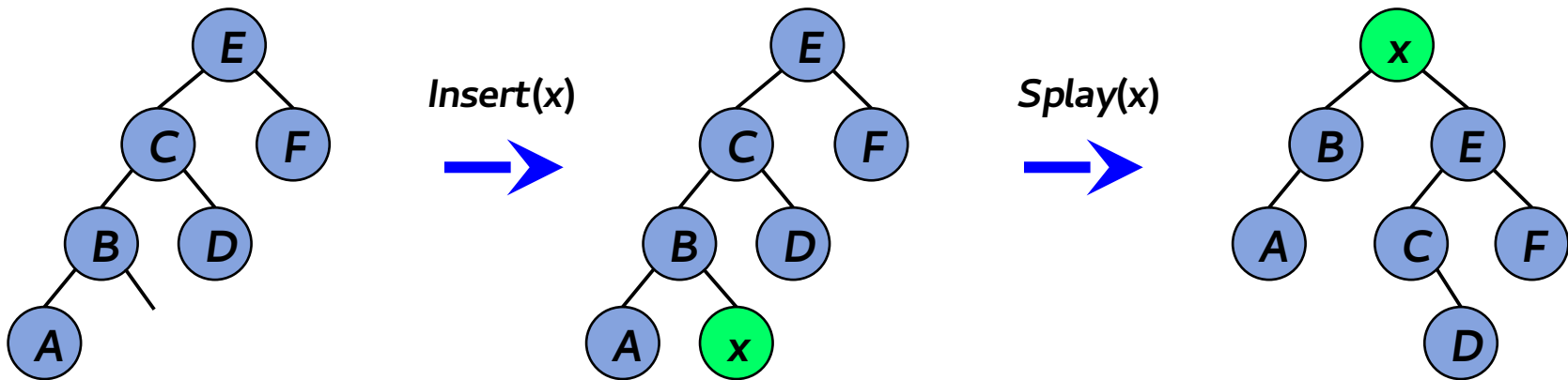
Старший преподаватель Кафедры вычислительных систем СибГУТИ

**E-mail:** [sillyhat34@gmail.com](mailto:sillyhat34@gmail.com)

Курс «Структуры и алгоритмы обработки данных»  
Осенний семестр, 2021 г.

# Косые деревья (*splay trees*)

- **Косое дерево**<sup>\*</sup>, расширяющееся дерево, скошенное дерево (*splay tree*) — это дерево поиска, обеспечивающее быстрый доступ к часто используемым узлам
- Добавление элемента  $x$  в дерево:
  1. Находим лист для вставки элемента  $x$  и создаём его (как в традиционных BST)
  2. Применяем к узлу  $x$  процедуру **Splay**, которая поднимает его в корень дерева



[\*] Sleator D., Tarjan R. Self-Adjusting Binary Search Trees // Journal of the ACM. 1985. Vol. 32 (3). P. 652–686

# Косые деревья (*splay trees*)

## Удаление элемента $x$ из дерева:

1. Отыскиваем узел  $x$  и удаляем его (как в традиционных BST)
2. Применяем к родителю узла  $x$  процедуру ***Splay***

или

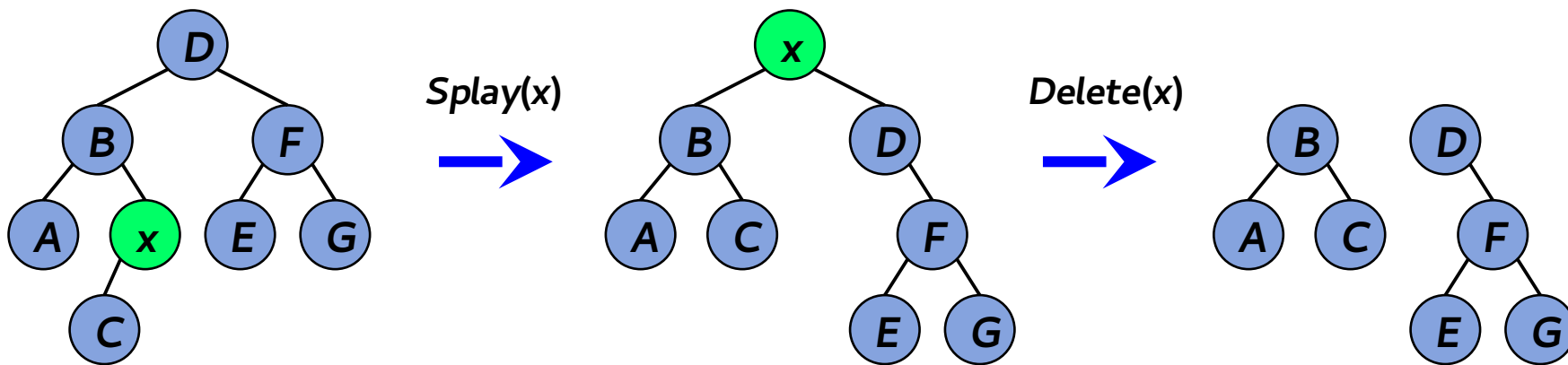
1. Применяем к узлу  $x$  процедуру ***Splay*** и удаляем его — образуются 2 поддерева,  $L$  и  $R$
2. Реализуется один из методов:
  - применить процедуру ***Splay*** к максимальному элементу поддерева  $L$
  - применить процедуру ***Splay*** к минимальному элементу поддерева  $R$

# Косые деревья (*splay trees*)

1. Применяем к узлу  $x$  процедуру **Splay** и удаляем его — образуются 2 поддерева,  $L$  и  $R$

2. Реализуется один из методов:

- применить процедуру **Splay** к максимальному элементу поддерева  $L$
- применить процедуру **Splay** к минимальному элементу поддерева  $R$

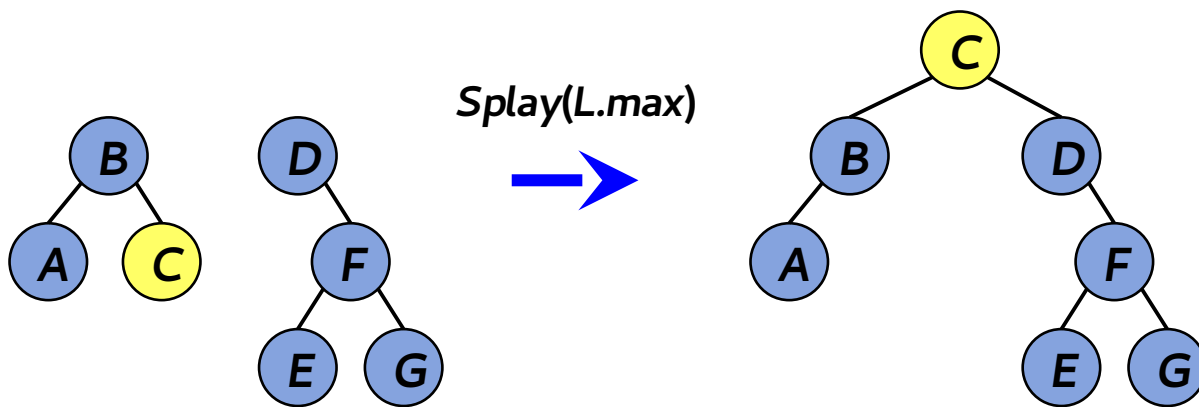


# Косые деревья (*splay trees*)

1. Применяем к узлу  $x$  процедуру **Splay** и удаляем его — образуются 2 поддерева,  $L$  и  $R$

2. Реализуется один из методов:

- применить процедуру **Splay** к максимальному элементу поддерева  $L$
- применить процедуру **Splay** к минимальному элементу поддерева  $R$



# Косые деревья (*splay trees*)

## Поиск элемента $x$ :

- Отыскиваем узел  $x$  (как в традиционных BST)
- При нахождении элемента запускаем ***Splay*** для него

## Процедура ***Splay***( $x$ ):

***Splay*** перемещает узел  $x$  в корень при помощи трёх операций: *Zig*, *Zig-Zig* и *Zig-Zag* (пока  $x$  не станет корнем)

Пусть  $p$  — родитель узла  $x$ ,  $g$  — родитель  $p$  (дед узла  $x$ )

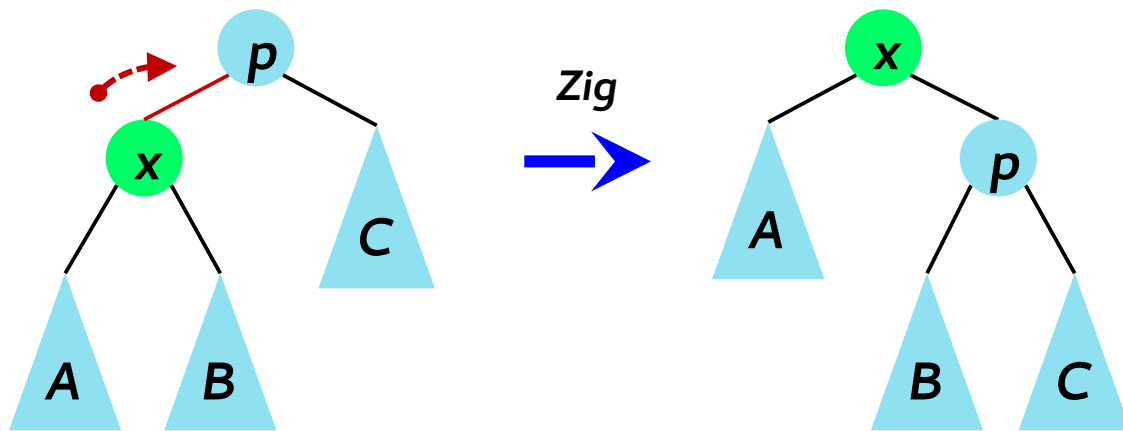
# Косые деревья (*splay trees*)

## Процедура *Splay(x)*:

**Splay** перемещает узел  $x$  в корень при помощи трёх операций: *Zig*, *Zig-Zig* и *Zig-Zag* (пока  $x$  не станет корнем)

Пусть  $p$  — родитель узла  $x$ ,  $g$  — родитель  $p$  (дед узла  $x$ )

- **Zig** — выполняется, если  $p$  — корень дерева. Дерево поворачивается по ребру между  $x$  и  $p$

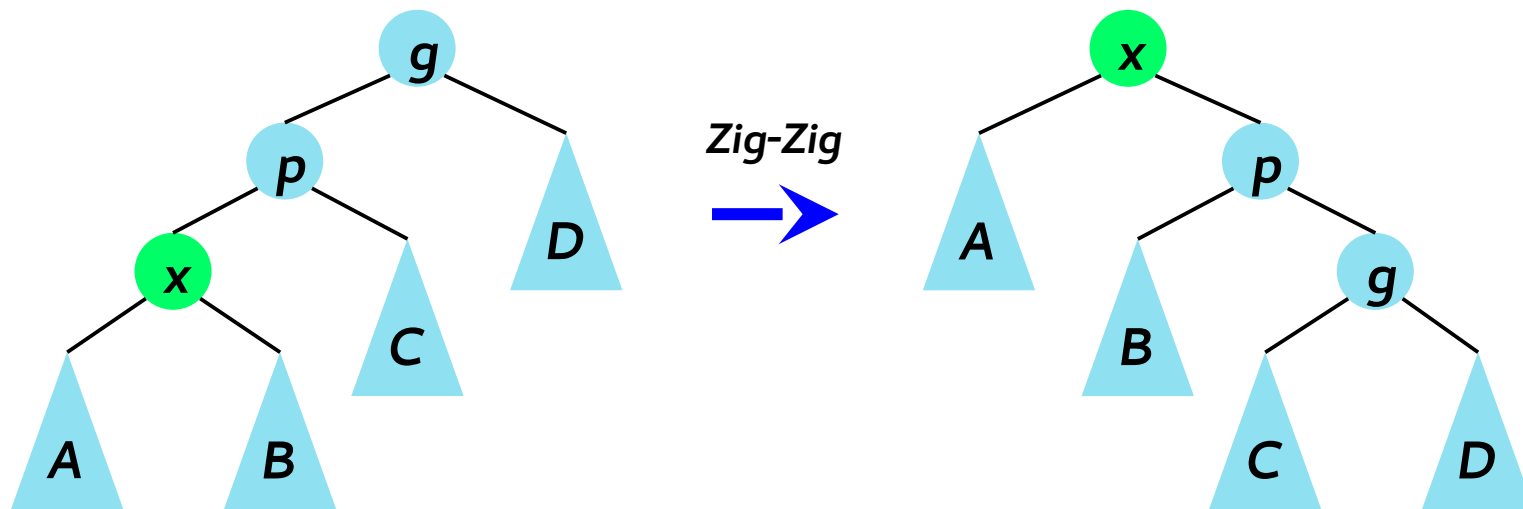


# Косые деревья (*splay trees*)

## Процедура *Splay(x)*:

Пусть  $p$  — родитель узла  $x$ ,  $g$  — родитель  $p$  (дед узла  $x$ )

- **Zig-Zig** — выполняется, когда и  $x$ , и  $p$  являются левыми (или правыми) дочерними элементами. Дерево поворачивается по ребру между  $g$  и  $p$ , затем — по ребру между  $p$  и  $x$



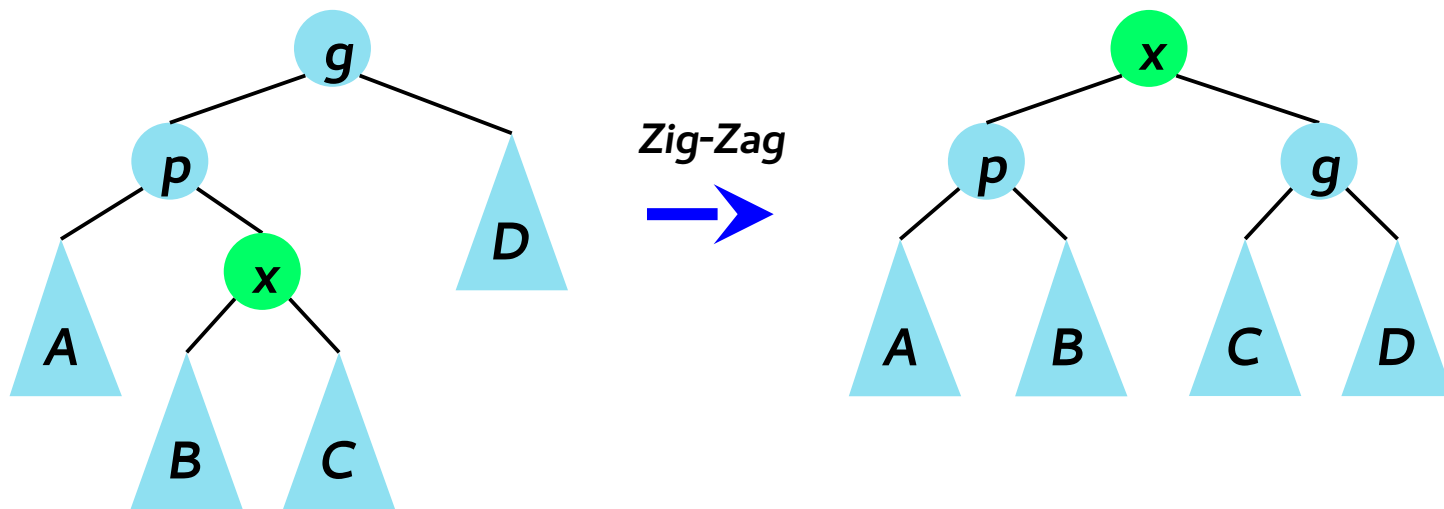


# Косые деревья (*splay trees*)

## Процедура *Splay(x)*:

Пусть  $p$  — родитель узла  $x$ ,  $g$  — родитель  $p$  (дед узла  $x$ )

- **Zig-Zag** — выполняется, когда  $x$  является правым потомком, а  $p$  — левым (или наоборот). Дерево поворачивается по ребру между  $p$  и  $x$ , затем — по ребру между  $x$  и  $g$



# Косые деревья (*splay trees*)

```
void splaytree_splay(struct splaytree_node *x)
{
    while (x->parent) {
        if (!x->parent->parent) {           // x->parent is a root node
            if (x->parent->left == x)       // x is in a left subtree of its parent
                splaytree_right_rotate(x->parent);
            else                           // x is in a right subtree of its parent
                splaytree_left_rotate(x->parent);
        } else if (x->parent->left == x && x->parent->parent->left == x->parent) {
            splaytree_right_rotate(x->parent->parent);
            splaytree_right_rotate(x->parent);
        }
        /* ... */
    }
}
```

# Косые деревья (*splay trees*)

```
/* ... */
} else if (x->parent->right == x && x->parent->parent->right == x->parent) {
    splaytree_left_rotate(x->parent->parent);
    splaytree_left_rotate(x->parent);
} else if (x->parent->left == x && x->parent->parent->right == x->parent) {
    splaytree_right_rotate(x->parent);
    splaytree_left_rotate(x->parent );
} else {
    splaytree_left_rotate(x->parent);
    splaytree_right_rotate(x->parent);
}
}
}
```

# Косые деревья (*splay trees*)

Операция	Средний случай (average case)	Худший случай (worst case)
<b>Add</b> (key, value)	$O(\log n)$	
<b>Lookup</b> (key)	$O(\log n)$	<b>Amortized</b> $O(\log n)$
<b>Remove</b> (key)	$O(\log n)$	

- Сложность по памяти (space complexity):  $O(n)$
- В худшем случае дерево может иметь высоту  $O(n)$

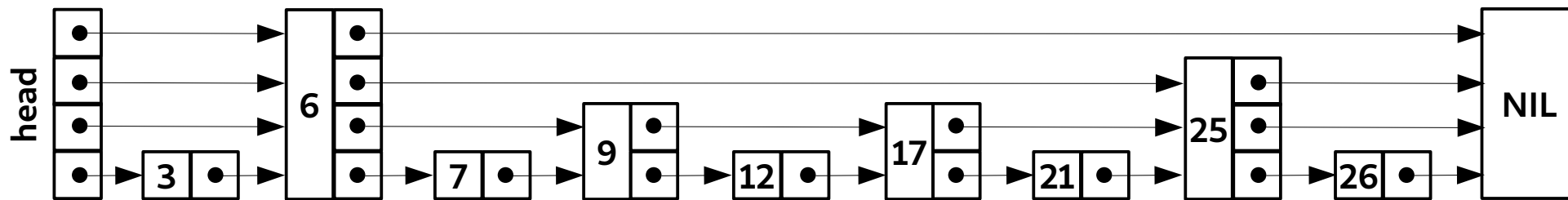
**Амортизированная сложность алгоритма — ?**

## СПИСОК ИСТОЧНИКОВ

- **Sleator D., Tarjan R.** Self-Adjusting Binary Search Trees // Journal of the ACM. 1985. Vol. 32 (3). P. 652-686
- **Thareja R.** Data Structures Using C. — 2nd edition. — Oxford University Press, 2014. — 557 p. (Раздел 10.6)
- **Binstock A., Rex J.** Practical Algorithms for Programmers. — Addison Wesley, 1995. — 585 p. (стр. 287-293)
- **Круз Р. Л.** Структуры данных и проектирование программ. — 2-е изд. — М. : БИНОМ. Лаборатория знаний, 2014. — 765 с. (Раздел 10.5)

# Списки с пропусками (*skip lists*)

- ♦ **Список с пропусками\*** (*skip list*) — это структура данных для реализации словаря, основанная на нескольких параллельных отсортированных связанных списках, пропускающих узлы
- ♦ **Применение на практике:**
  - ♦ Cyrus IMAP server
  - ♦ QMap (Qt 4.8)
  - ♦ Redis persistent key-value store
  - ♦ ...

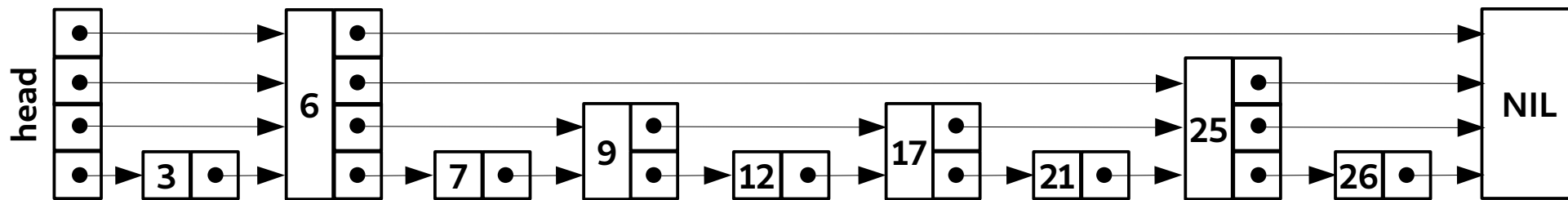


[\*] Pugh W. Skip lists: a probabilistic alternative to balanced trees // Communications of the ACM. 1990. Vol. 33 (6), P. 668–676

# Списки с пропусками (*skip lists*)

- **Список с пропусками\*** (*skip list*) — это структура данных для реализации словаря, основанная на нескольких параллельных отсортированных связанных списках, пропускающих узлы

Основная идея — реализация  
бинарного поиска для связанных списков  
(выполнять поиск быстрее, чем тривиальный  
проход по списку за линейное время)



[\*] Pugh W. Skip lists: a probabilistic alternative to balanced trees // Communications of the ACM. 1990. Vol. 33 (6), P. 668–676

## Списки с пропусками (*skip lists*)

- Каждый элемент в списке с пропусками представлен узлом
- У каждого узла есть **высота** или **уровень**, который соответствует количеству указателей на следующие уровни
- $i$ -ый указатель узла указывает на следующий узел, находящийся на уровне  $i$  или выше
- При вставке нового элемента в список, узел вставляется на уровень **со случайным номером**
- Уровни со случайными номерами генерируются по шаблону. Например: 50% — уровень 1, 25% — уровень 2, 12.5% — уровень 3 и т.д.



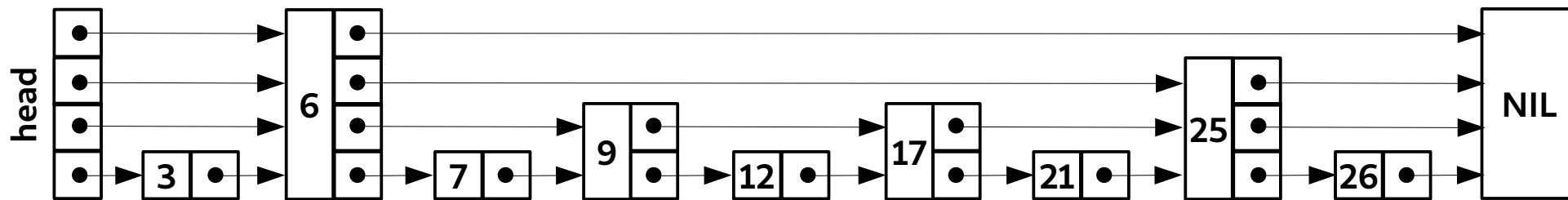
# Списки с пропусками (*skip lists*)

- Каждый элемент в списке с пропусками представлен узлом
- У каждого узла есть **высота** или **уровень**, который соответствует количеству указателей на следующие уровни
- $i$ -ый указатель узла указывает на следующий узел, находящийся на уровне  $i$  или выше
- При вставке нового элемента в список, узел вставляется на уровень **со случайным номером**
- Уровни со случайными номерами генерируются по шаблону. Например: 50% — уровень 1, 25% — уровень 2, 12.5% — уровень 3 и т.д.

**Skip list** — это связный список, в котором каждый узел содержит различное количество связей, причём  $i$ -ые связи в узлах реализуют односвязные списки, пропускающие узлы, содержащие менее чем  $i$  связей

# Списки с пропусками (*skip lists*)

- Каждый элемент в списке с пропусками представлен узлом
- У каждого узла есть **высота** или **уровень**, который соответствует количеству указателей на следующие уровни
- $i$ -ый указатель узла указывает на следующий узел, находящийся на уровне  $i$  или выше
- При вставке нового элемента в список, узел вставляется на уровень **со случайным номером**
- Уровни со случайными номерами генерируются по шаблону. Например: 50% — уровень 1, 25% — уровень 2, 12.5% — уровень 3 и т.д.



# Списки с пропусками (*skip lists*)

Операция	Средний случай (average case)	Худший случай (worst case)
<b>Add</b> (key, value)	$O(\log n)$	$O(n)$
<b>Lookup</b> (key)	$O(\log n)$	$O(n)$
<b>Remove</b> (key)	$O(\log n)$	$O(n)$

- Сложность по памяти (space complexity):  $O(n \log n)$

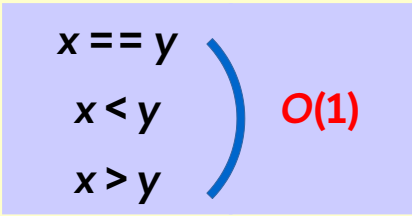
## Список источников

- **Седжвик Р.** Фундаментальные алгоритмы на С++. Анализ/Структуры данных/Сортировка/Поиск.  
– К.: ДиаСофт, 2001. — 688 с. (С. 555)
- **Pugh W.** A Skip List Cookbook // <http://cg.scs.carleton.ca/~morin/teaching/5408/refs/p90b.pdf>

# Словари со строковым ключом

- При анализе вычислительной сложности операций бинарных деревьев поиска (АВЛ-деревьев, красно-чёрных деревьев и др.), списков с пропусками (skip lists) предполагается, что время выполнения операции сравнения двух ключей ( $=$ ,  $<$ ,  $>$ ) константное
- Если ключи — строки, то время выполнения операции сравнения становится значимым и его следует учитывать

```
struct rbtree *rbtree_lookup(struct rbtree *tree, int key)
{
    while (tree != NULL) {
        if (key == tree->key)
            return tree;
        else if (key < tree->key)
            tree = tree->left;
        else
            tree = tree->right;
    }
    return tree;
}
```



$x == y$   
 $x < y$   
 $x > y$   $O(1)$

# Префиксные деревья

- **Префиксное дерево**\* (*trie, prefix tree, digital tree, radix tree*) — структура данных для реализации ассоциативного массива (словаря), ключами в котором являются строки
- **Авторы:** R. de la Brandais, 1959; E. Fredkin, 1960
- Происхождение слова «trie» — **retrieval** (поиск, извлечение, выборка, возврат)

Альтернативные названия:

- **бор** — Д. Кнут, Т. 3, 1978, «вы**бор**ка»
- **луч** — Д. Кнут, Т. 3, 2000, «пол**учение**»
- нагруженное дерево — А. Ахо и др., 2000

[\*] **Fredkin E.** Trie Memory // Communications of the ACM Vol.3, Issue 9. — 1960. — pp. 490–499.

# Префиксные деревья

- **Префиксное дерево\*** (*trie, prefix tree, digital tree, radix tree*) — структура данных для реализации ассоциативного массива (словаря), ключами в котором являются строки

## Практические применения:

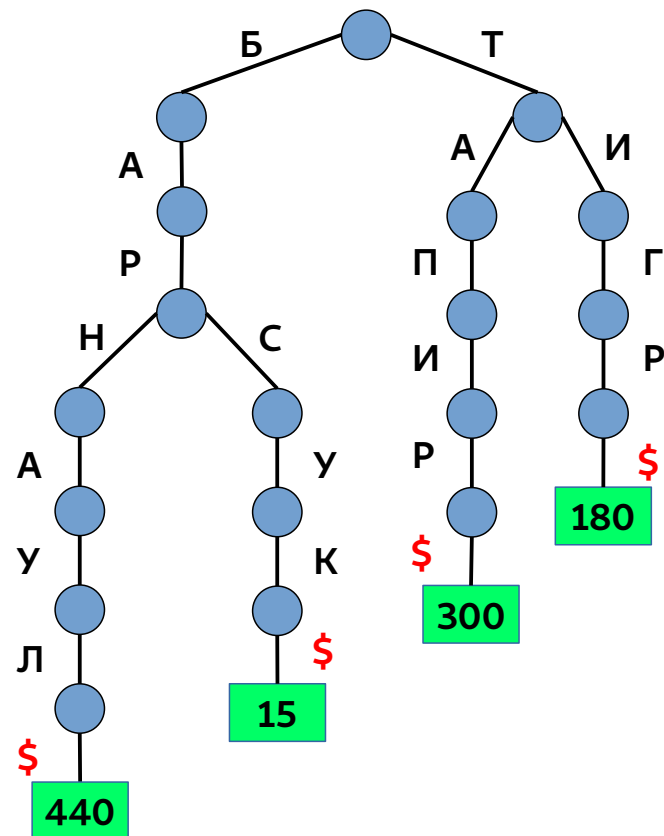
- Предиктивный ввод текста (*predictive text*) — поиск возможных завершений слов
- Автозавершение (*autocomplete*) в текстовых редакторах и IDE
- Проверка правописания (*spellcheck*)
- Автоматическая расстановка переносов слов (*hyphenation*)
- Squid Caching Proxy Server

[\*] **Fredkin E.** Trie Memory // Communications of the ACM Vol.3, Issue 9. — 1960. — pp. 490–499.

# Префиксные деревья

Словарь

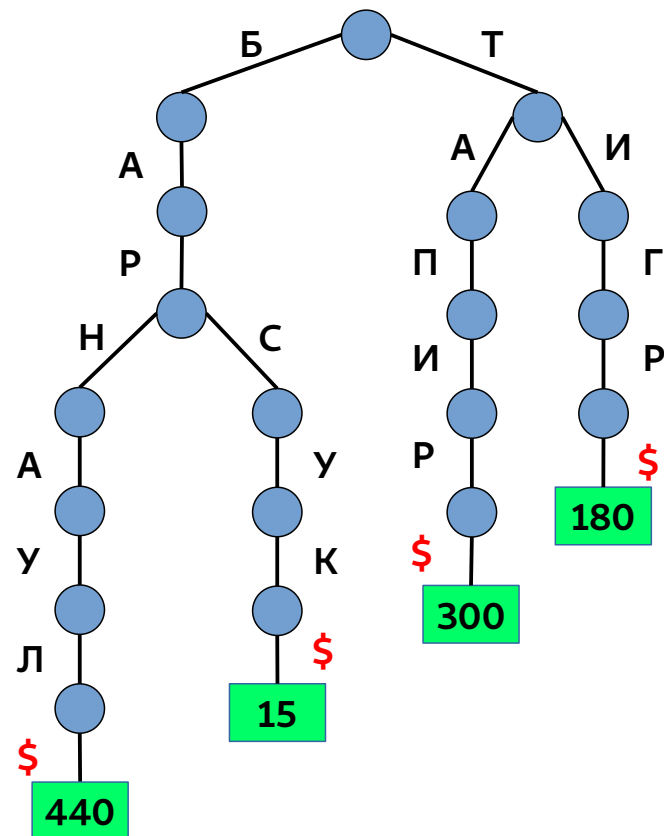
Ключ (key)	Значение (value)
Тигр	180
Тапир	300
Барсук	15
Барнаул	440





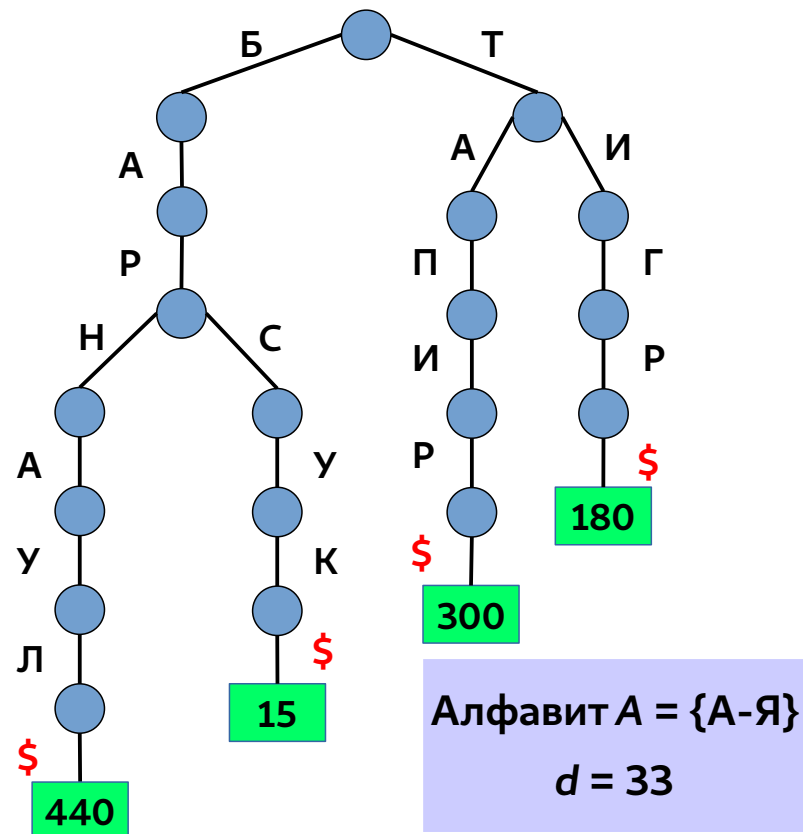
# Префиксные деревья

- **Префиксное дерево** (*trie*) содержит  $n$  строковых ключей и ассоциированные с ними значения (*values*)
- **Ключ** (*key*) — это набор символов  $\{c_1, c_2, \dots, c_m\}$  из алфавита  $A = \{a_1, a_2, \dots, a_d\}$
- Каждый узел содержит от 1 до  $d$  дочерних узлов
- За каждым ребром закреплен символ алфавита
- Символ **\$** — это маркер конца строки (ключа)



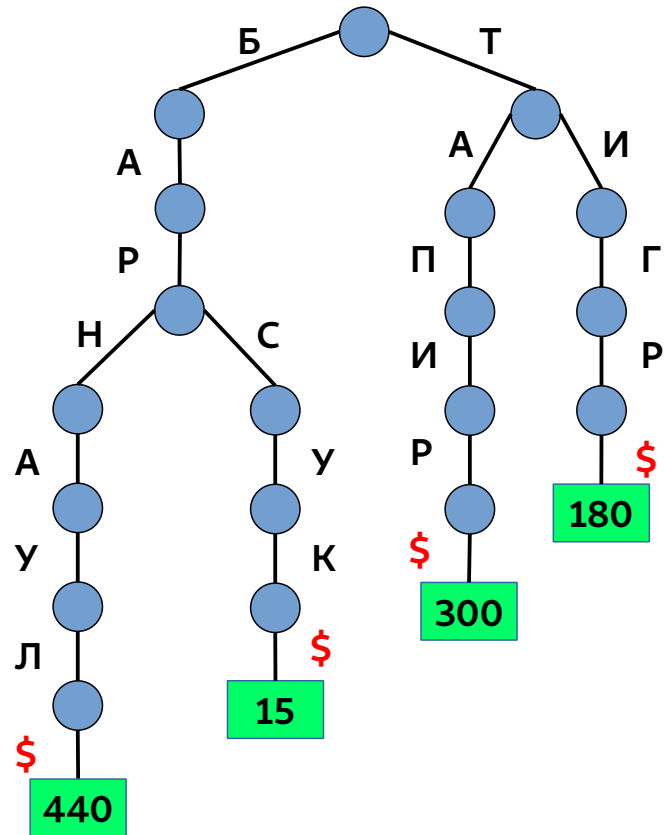
# Префиксные деревья

- **Префиксное дерево** (*trie*) содержит  $n$  строковых ключей и ассоциированные с ними значения (*values*)
- **Ключ** (*key*) — это набор символов  $\{c_1, c_2, \dots, c_m\}$  из алфавита  $A = \{a_1, a_2, \dots, a_d\}$
- Каждый узел содержит от 1 до  $d$  дочерних узлов
- За каждым ребром закреплен символ алфавита
- Символ **\$** — это маркер конца строки (ключа)



# Префиксные деревья

- Ключи не хранятся в узлах дерева
- Позиция листа в дереве определяется значением его ключа
- Значения хранятся в листьях

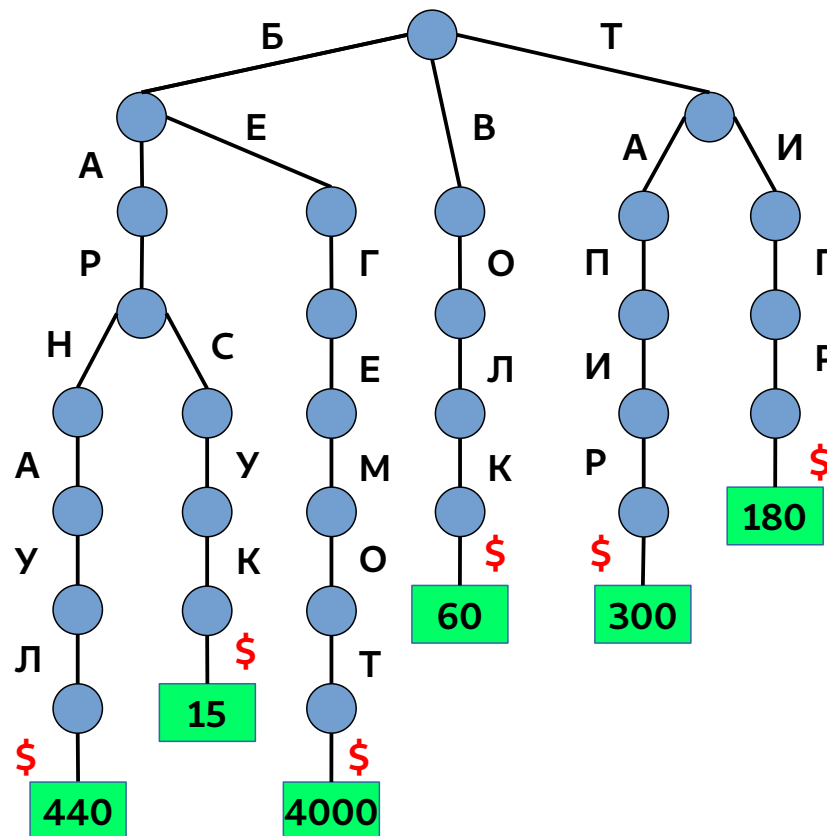


# Префиксные деревья

# Словарь

Ключ (key)	Значение (value)
Тигр	180
Тапир	300
Барсук	15
Барнаул	440
Волк	60
Бегемот	4000

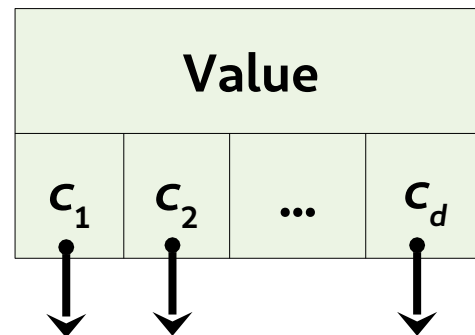
- Символ **\$** — это маркер конца строки
- Высота дерева  $h = O(\max(key_i)), i = 1, 2, \dots, n$



# Узел префиксного дерева

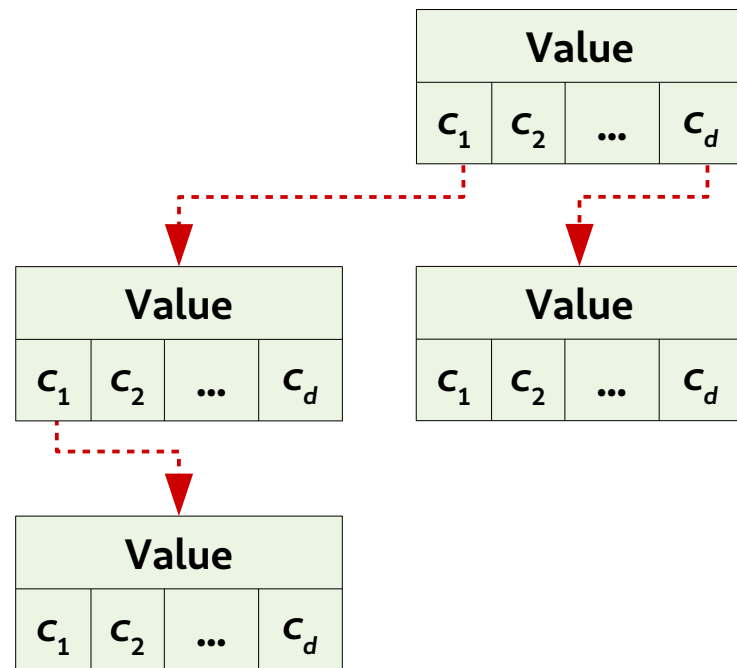
- Ключ — это набор символов  $\{c_1, c_2, \dots, c_m\}$  из алфавита  $A = \{a_1, a_2, \dots, a_d\}$
- Каждый узел содержит от 1 до  $d$  указателей на дочерние узлы
- Значения хранятся в **листьях**

Как хранить  $c_1, c_2, \dots, c_m$ ?



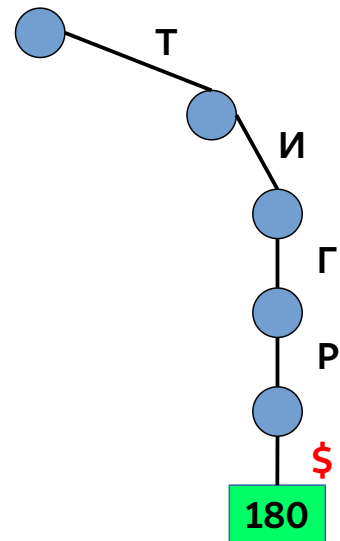
# Вставка элемента в префиксное дерево

1. Инициализируем  $k = 1$
2. В текущем узле (начиная с корня) отыскиваем символ  $c_i$ , равный  $k$ -му символу ключа  $key[k]$
3. Если  $c_i \neq \text{NULL}$ , то
  - Делаем текущим узел, на который указывает  $c_i$
  - Переходим к следующему символу ключа ( $k = k + 1$ ) и пункту 2
4. Если  $c_i = \text{NULL}$ , создаём новый узел, делаем его текущим, переходим к следующему символу ключа и пункту 2
5. Если достигли конца строки ( $\$$ ), вставляем значение в текущий узел (проверить, что ключ уникальный)



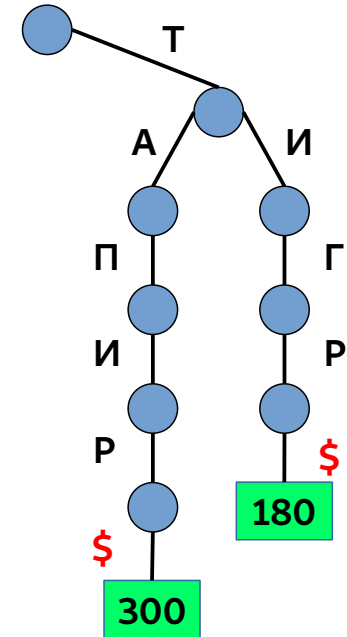
## Вставка элемента в префиксное дерево

- Добавление элемента ("Тигр", 180)



# Вставка элемента в префиксное дерево

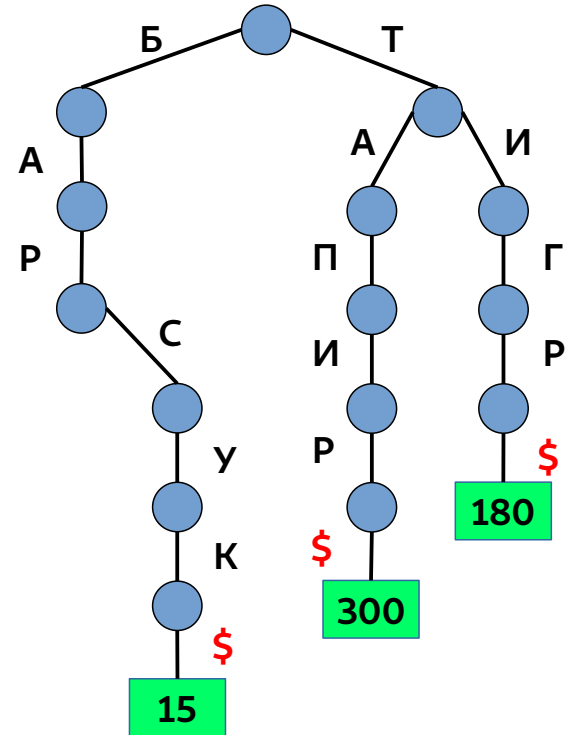
- Добавление элемента ("Тигр", 180)
- Добавление элемента ("Тапир", 300)





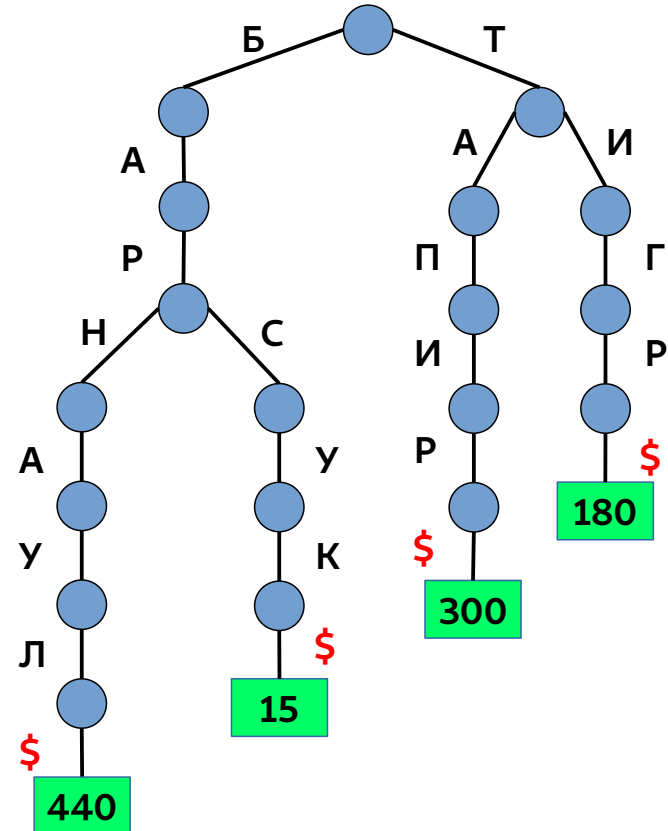
# Вставка элемента в префиксное дерево

- Добавление элемента ("Тигр", 180)
- Добавление элемента ("Тапир", 300)
- Добавление элемента ("Барсук", 15)



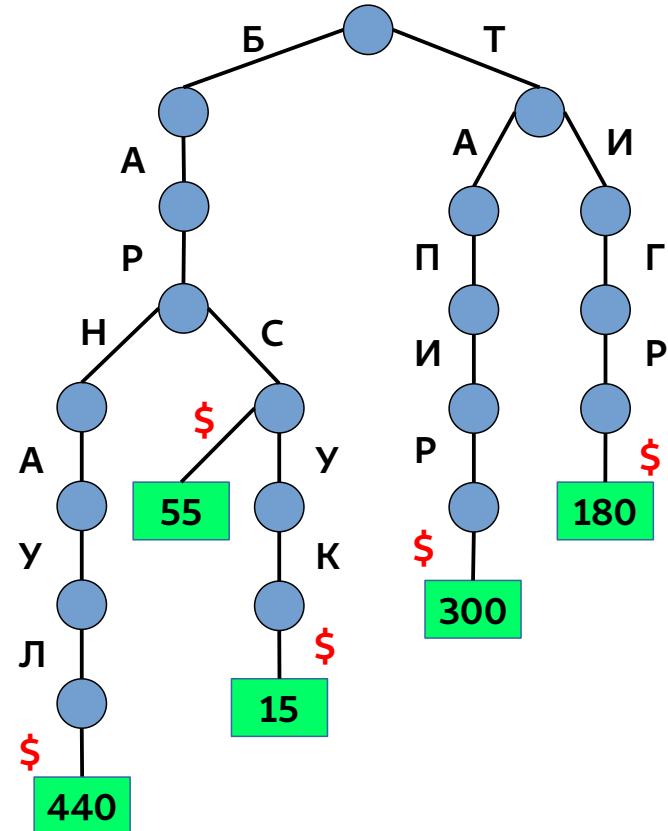
# Вставка элемента в префиксное дерево

- Добавление элемента ("Тигр", 180)
- Добавление элемента ("Тапир", 300)
- Добавление элемента ("Барсук", 15)
- Добавление элемента ("Барнаул", 440)



# Вставка элемента в префиксное дерево

- Добавление элемента ("Тигр", 180)
- Добавление элемента ("Тапир", 300)
- Добавление элемента ("Барсук", 15)
- Добавление элемента ("Барнаул", 440)
- Добавление элемента ("Барс", 55)



## Вставка элемента в префиксное дерево

```
function TrieInsert(root, key, value)
  node = root
  for i = 1 to Length(key) do
    child = GetChild(node, key[i])
    if child = NULL then
      child = TrieCreateNode()
      SetChild(node, key[i], child)
    end if
    node = child
  end for
  node.value = value
end function
```

- **GetChild**(node, c) — возвращает указатель на дочерний узел, соответствующий символу c
- **SetChild**(node, c, child) — устанавливает указатель, соответствующий символу c, в значение *child*

## Вставка элемента в префиксное дерево

```
function TrieInsert(root, key, value)
  node = root
  for i = 1 to Length(key) do
    child = GetChild(node, key[i])
    if child = NULL then
      child = TrieCreateNode()
      SetChild(node, key[i], child)
    end if
    node = child
  end for
  node.value = value
end function
```

$$T_{\text{Insert}} = O(m * (T_{\text{SetChild}} + T_{\text{GetChild}}))$$

- **GetChild**(node, c) — возвращает указатель на дочерний узел, соответствующий символу c
- **SetChild**(node, c, child) — устанавливает указатель, соответствующий символу c, в значение *child*

# Поиск элемента в префиксном дереве

```
function TrieLookup(root, key)
  node = root
  for i = 1 to Length(key) do
    child = GetChild(node, key[i])
    if child = NULL then
      return NULL
    end if
    node = child
  end for
  if node.value = NULL then // Найденный ключ не имеет значения
    return NULL
  return node
end function
```

- **GetChild**(node, c) — возвращает указатель на дочерний узел, соответствующий символу c

# Поиск элемента в префиксном дереве

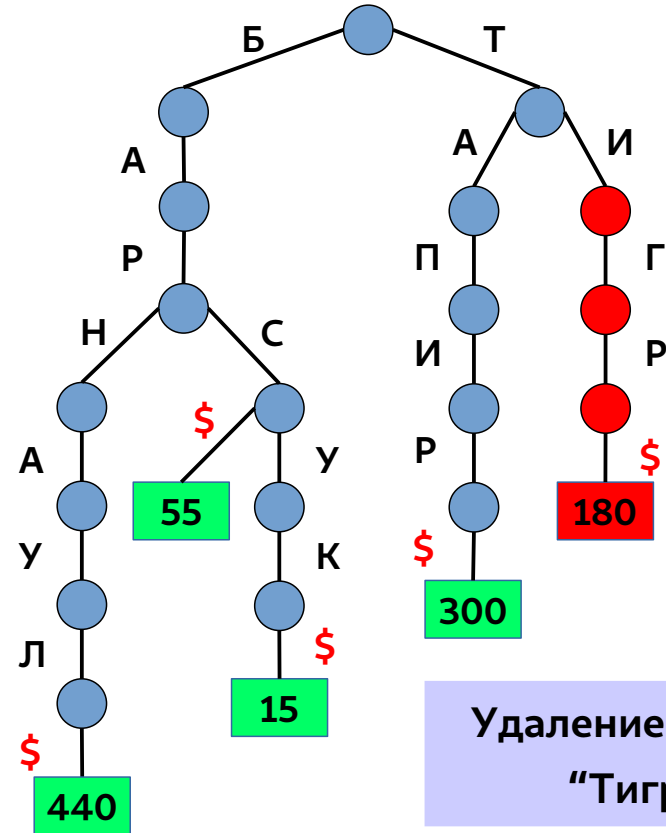
```
function TrieLookup(root, key)
  node = root
  for i = 1 to Length(key) do
    child = GetChild(node, key[i])
    if child = NULL then
      return NULL
    end if
    node = child
  end for
  if node.value = NULL then // Найденный ключ не имеет значения
    return NULL
  return node
end function
```

$$T_{Lookup} = O(m * T_{GetChild})$$

- **GetChild**(node, c) — возвращает указатель на дочерний узел, соответствующий символу c

# Удаление элемента из префиксного дерева

1. Отыскиваем лист, содержащий искомый ключ *key*
2. Если текущий узел не имеет дочерних узлов, удаляем его из памяти (в противном случае заканчиваем подъем по дереву)
3. Делаем текущим родительский узел и переходим к пункту 2





# Представление внутренних узлов префиксного дерева

Как хранить указатели  $c_1, c_2, \dots, c_d$  на дочерние узлы дерева?

Value			
$c_1$	$c_2$	...	$c_d$

- Массив указателей (индекс — номер символа)

```
struct trie *child[33];  
node->child[char_to_index('Б')];
```

- Сложность **GetChild/SetChild** —  $O(1)$

# Представление внутренних узлов префиксного дерева

Как хранить указатели  $c_1, c_2, \dots, c_d$  на дочерние узлы дерева?

Value			
$c_1$	$c_2$	...	$c_d$

- **Связный список** указателей на дочерние узлы

```
struct trie *child;  
linked_list_lookup(child, 'Б');
```

- Сложность **GetChild/SetChild** —  $O(d)$

# Представление внутренних узлов префиксного дерева

Как хранить указатели  $c_1, c_2, \dots, c_d$  на дочерние узлы дерева?

Value			
$c_1$	$c_2$	...	$c_d$

- Сбалансированное дерево поиска (Red-black tree / AVL-tree)

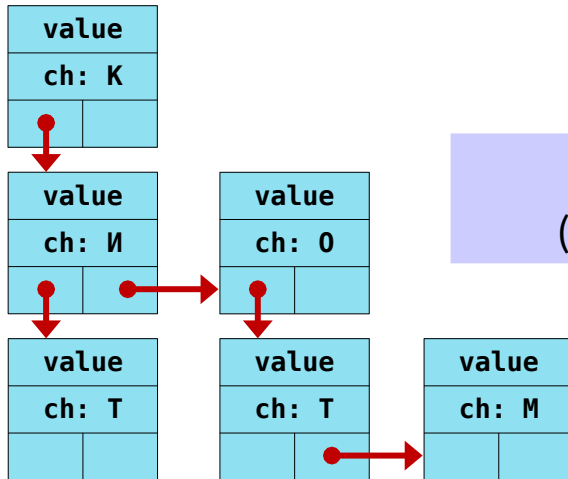
```
struct rbtree *child;  
rbtree_lookup(child, 'Б')
```

- Сложность **GetChild/SetChild** —  $O(\log d)$

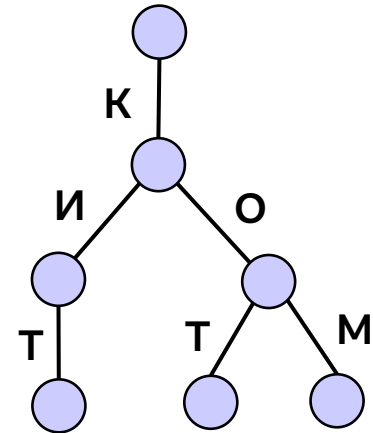
# Представление узла префиксного дерева

```
struct trie {  
    char *value;  
    char ch;  
    struct trie *sibling;    /* Sibling node */  
    struct trie *child;     /* First child node */  
};
```

$$T_{\text{GetChild}} = T_{\text{SetChild}} = O(d)$$



Неупорядоченное дерево поиска  
(unordered search tree, unordered set)



## Создание пустого узла

```
struct trie *trie_create()
{
    struct trie *node;

    if ((node = malloc(sizeof(*node))) == NULL)
        return NULL;
    node->ch = '\0';
    node->value = NULL;
    node->sibling = NULL;
    node->child = NULL;

    return node;
}
```

$$T_{\text{Create}} = O(1)$$

# Поиск узла по ключу

```
char *trie_lookup(struct trie *root, char *key)
{
    struct trie *node, *list;

    for (list = root; *key != '\0'; key++) {
        for (node = list; node != NULL; node = node->sibling) {
            if (node->ch == *key)
                break;
        }
        if (node != NULL)
            list = node->child;
        else
            return NULL;
    }
    return node->value; /* Node must be a leaf */
}
```

$$T_{\text{Lookup}} = O(md)$$

## Вставка узла в префиксное дерево

```
struct trie *trie_insert(struct trie *root, char *key, char *value)
{
    struct trie *node, *parent, *list;

    parent = NULL;
    list = root;
    for (; *key != '\0'; key++) {
        /* Lookup sibling node */
        for (node = list; node != NULL; node = node->sibling)
        {
            if (node->ch == *key)
                break;
        }
    }
}
```

## Вставка узла в префиксное дерево (продолжение)

```
if (node == NULL) {                                /* Node not found. Add new node */
    node = trie_create();
    node->ch = *key;
    node->sibling = list;
    if (parent != NULL)
        parent->child = node;
    else
        root = node;
    list = NULL;
} else {                                            /* Node found. Move to next level */
    list = node->child;
}
parent = node;
}
```



## Вставка узла в префиксное дерево (окончание)

```
/* Update value in leaf */  
if (node->value != NULL)  
    free(node->value);  
node->value = strdup(value);  
return root;  
}
```

$$T_{\text{Insert}} = O(md)$$

## Удаление узла из префиксного дерева

```
struct trie *trie_delete(struct trie *root, char *key)
{
    int found;

    return trie_delete_dfs(root, NULL, key, &found);
}
```

## Удаление узла из префиксного дерева (продолжение)

```
struct trie *trie_delete_dfs(struct trie *root, struct trie *parent,
                             char *key, int *found)
{
    struct trie *node, *prev = NULL;

    *found = (*key == '\0' && root == NULL) ? 1 : 0;
    if (root == NULL || *key == '\0')
        return root;

    for (node = root; node != NULL; node = node->sibling) {
        if (node->ch == *key)
            break;
        prev = node;
    }
    if (node == NULL)
        return root;
```

## Удаление узла из префиксного дерева (окончание)

```
trie_delete_dfs(node->child, node, key + 1, found);
if (*found > 0 && node->child == NULL) {
    /* Delete node */
    if (prev != NULL)
        prev->sibling = node->sibling;
    else {
        if (parent != NULL)
            parent->child = node->sibling;
        else
            root = node->sibling;
    }
    free(node->value);
    free(node);
}
return root;
}
```

$$T_{Delete} = T_{Lookup} + O(m) = O(md + m) = O(md)$$

# Вычислительная сложность операций

Операция	Способ работы с указателями $c_1, c_2, \dots, c_d$		
	Связный список	Массив	Self-balancing search tree (Red-black / AVL tree)
Lookup	$O(md)$	$O(m)$	$O(m \log d)$
Insert	$O(md)$	$O(m)$	$O(m \log d)$
Delete	$O(md)$	$O(m)$	$O(m \log d)$
Min	$O(hd)$	$O(hd)$	$O(h \log d)$
Max	$O(hd)$	$O(hd)$	$O(h \log d)$

- $h$  — высота дерева (количество символов в самом длинном ключе)
- В случае упорядоченного префиксного дерева (список  $c_1, c_2, \dots, c_d$  упорядочен в лексикографическом порядке) операции **min** и **max** реализуются за время  $O(h)$

# Преимущества префиксных деревьев

- Время поиска не зависит от количества элементов в словаре (зависит от длины ключа и мощности алфавита)
- Для хранения ключей не требуется дополнительной памяти (ключи не хранятся в узлах)
- В отличие от хеш-таблиц имеется возможность обхода в упорядоченной последовательности (от меньших ключей к большим и наоборот, реализация *ordered map/set*) — зависит от реализации *SetChild/GetChild*
- В отличие от хеш-таблиц не возникает коллизий

# Производительность строковых словарей, худший случай

Операция	Trie (linked list)	Self-balanced search tree (Red-black/AVL tree)	Hash table (chaining)
Lookup	$O(md)$	$O(m \log n)$	$O(m + nm)$
Insert	$O(md)$	$O(m \log n)$	$O(m + n)$
Delete	$O(md)$	$O(m \log n)$	$O(m + nm)$
Min	$O(hd)$	$O(\log n)$	$O(H + nm)$
Max	$O(hd)$	$O(\log n)$	$O(H + nm)$

- $m$  — длина ключа
- $d$  — количество символов в алфавите (константа)
- $n$  — количество элементов в словаре
- $H$  — размер хеш-таблицы

# Производительность строковых словарей, средний случай

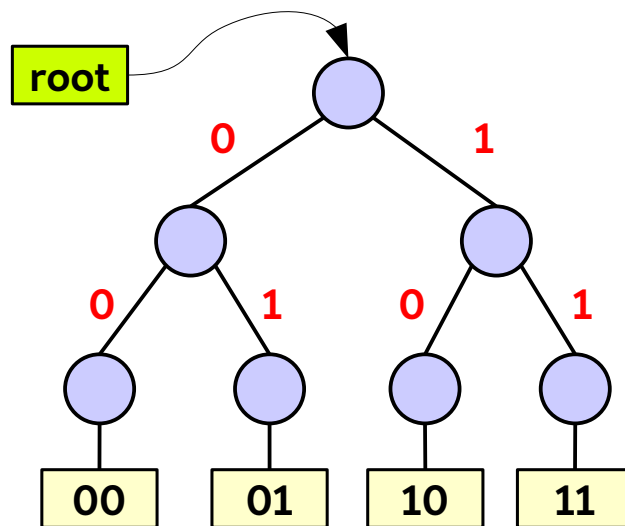
Операция	Trie (linked list)	Self-balanced search tree (Red-black/AVL tree)	Hash table (chaining)
Lookup	$O(md^*)$	$O(m \log n)$	$O(m + (n / H)m) = O(m)$
Insert	$O(md^*)$	$O(m \log n)$	$O(m + (n / H)m) = O(m)$
Delete	$O(md^*)$	$O(m \log n)$	$O(m + (n / H)m) = O(m)$
Min	$O(hd^*)$	$O(\log n)$	$O(H + nm)$
Max	$O(hd^*)$	$O(\log n)$	$O(H + nm)$

- $m$  — длина ключа
- $d$  — количество символов в алфавите (константа)
- $n$  — количество элементов в словаре
- $H$  — размер хеш-таблицы



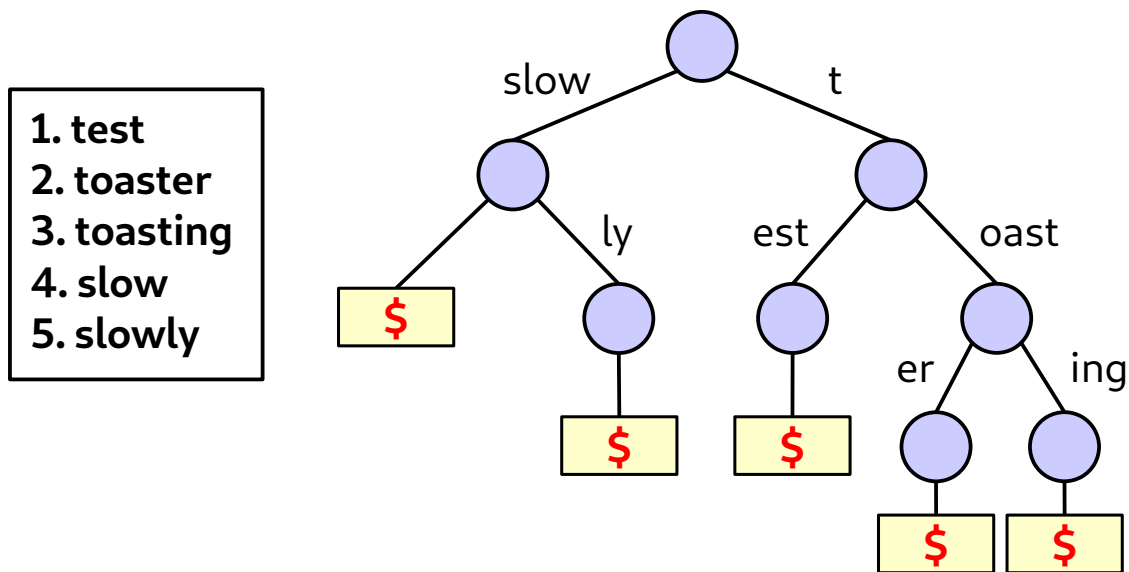
# Bitwise tree

- **Bitwise tree** — префиксное дерево, в котором ключи представлены как последовательность битов
- Bitwise tree позволяет хранить ключи **произвольного типа данных**
- Bitwise tree — это бинарное дерево; алфавит  $A = \{0, 1\}$



# Radix tree

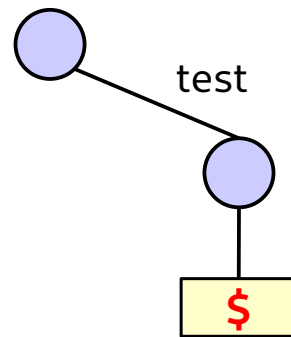
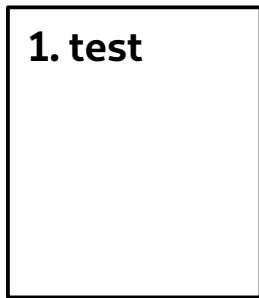
- **Базисное дерево**<sup>1,2</sup> (*radix tree, patricia trie, compact prefix tree*) — префиксное дерево, в котором узел, содержащий один дочерний элемент, объединяется с ним для экономии памяти



[1] **D. R. Morrison.** PATRICIA — Practical Algorithm to Retrieve Information Coded in Alphanumeric. Jnl. of the ACM, 15(4). pp. 514-534, Oct 1968.

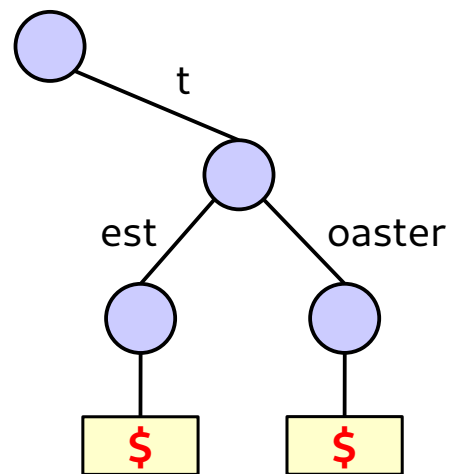
[2] **Gwehenberger G.** Anwendung einer binären Verweiskettenmethode beim Aufbau von Listen. Elektronische Rechenanlagen 10 (1968), pp. 223-226

# Radix tree, добавление элементов



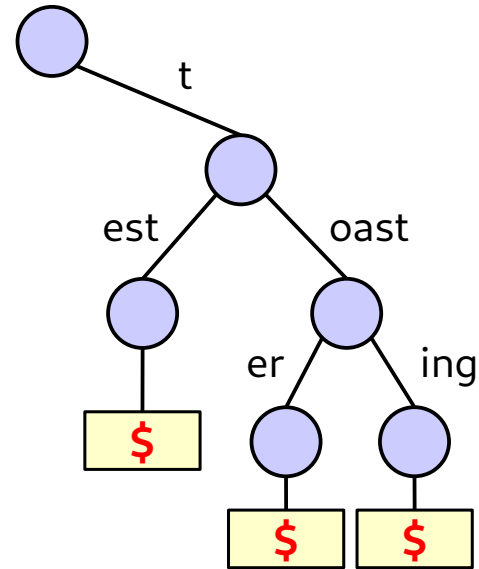
# Radix tree, добавление элементов

1. test  
2. toaster



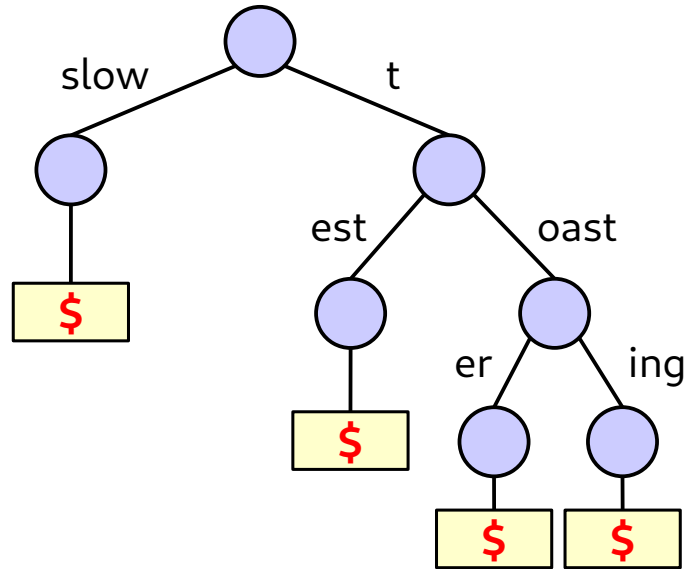
# Radix tree, добавление элементов

1. test  
2. toaster  
3. toasting



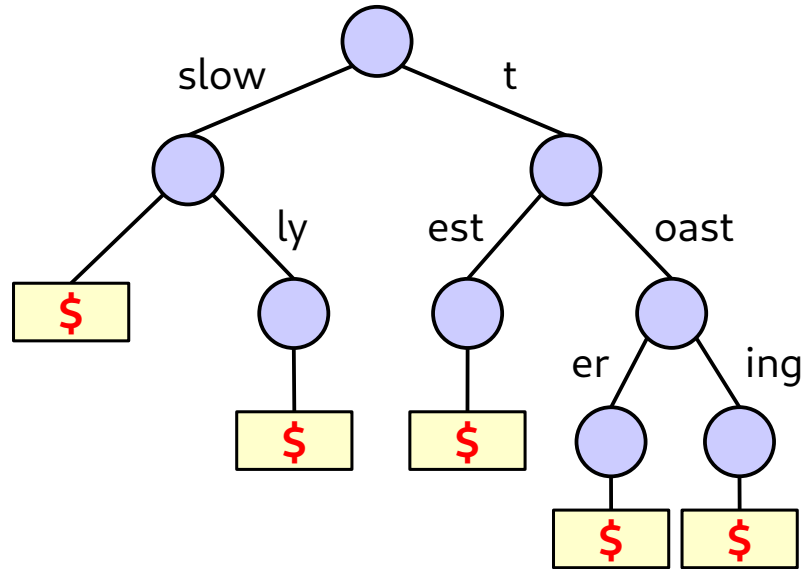
# Radix tree, добавление элементов

1. test
2. toaster
3. toasting
4. slow



# Radix tree, добавление элементов

1. test
2. toaster
3. toasting
4. slow
5. slowly



# Суффиксное дерево

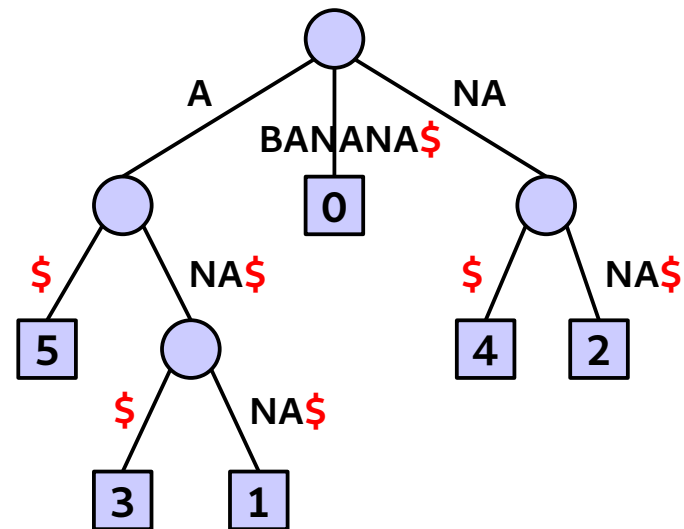
- **Суффиксное дерево**\* (*suffix tree, PAT tree, position tree*) — вариация префиксного дерева, содержащая все суффиксы заданного текста (ключи) и их начальные позиции в тексте (значения)

## Применение:

- Поиск подстроки в строке за время  $O(m)$
- Поиск наибольшей повторяющейся подстроки
- Поиск наибольшей общей подстроки
- Поиск наибольшей подстроки-палиндрома
- Алгоритм LZW сжатия информации

Суффиксы текста  
"BANANA":

BANANA\$ [0]  
ANANA\$ [1]  
NANA\$ [2]  
ANA\$ [3]  
NA\$ [4]  
A\$ [5]



[\*] **Weiner P.** Linear pattern matching algorithms // 14th Annual IEEE Symposium on Switching and Automata Theory, 1973, p. 1-11



## Дальнейшее чтение

1. Подробнее ознакомиться с устройством префиксных деревьев:

- **Ахо А. В., Хопкрофт Д., Ульман Д. Д.** Структуры данных и алгоритмы. — М.: Вильямс, 2001. — 384 с.
- **Гасфилд Д.** Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология. — Санкт-Петербург: Невский Диалект, БХВ-Петербург, 2003. — 656 с.
- **Билл Смит.** Методы и алгоритмы вычислений на строках. Теоретические основы регулярных вычислений. — М.: Вильямс, 2006. — 496 с.

2. Изучить алгоритмы поиска и удаления элементов в базисном дереве

# ご清聴ありがとうございました!



**Даниил Михайлович Берлизов**

Старший преподаватель Кафедры вычислительных систем СибГУТИ

**E-mail:** [sillyhat34@gmail.com](mailto:sillyhat34@gmail.com)

Курс «Структуры и алгоритмы обработки данных»

Осенний семестр, 2021 г.