Лекция 4: Префиксные деревья (Prefix tree, trie)

Курносов Михаил Георгиевич

к.т.н. доцент Кафедры вычислительных систем
Сибирский государственный университет
телекоммуникаций и информатики

http://www.mkurnosov.net

Словарь со строковыми ключами

- При анализе вычислительной сложности операций бинарных деревьев поиска, АВЛ-деревьев, красно-черных деревьев, Splay trees, списков с пропусками (Skip lists) мы полагали, что время выполнения операции сравнения двух ключей (=, <, >) выполняется за время $O(1)$

- Если ключи – это длинные строки (char []), 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;
}
```

$key_1 = key_2$
$key_1 < key_2$
$key_1 > key_2$
} $O(1)$

Префиксные деревья (Trie)

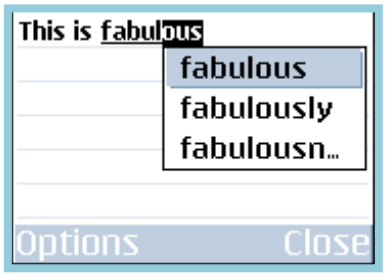
- **Префиксное дерево** (Trie, prefix tree, digital tree, radix tree) – это структура данных для реализации словаря (ассоциативного массива), ключами в котором являются строки
- Авторы: Briandais, 1959; Fredkin, 1960
- Происхождение слова “trie” – re*trie*val (поиск, извлечение, выборка, возврат)
- Альтернативные названия:
  - **бор** (Д. Кнут, Т. 3, 1978, вы*бор*ка)
  - **луч** (Д. Кнут, Т. 3, 2000, по*луч*ение)
  - **нагруженное дерево** (А. Ахо и др., 2000)

Префиксные деревья (Trie)

- **Префиксное дерево** (Trie, prefix tree, digital tree, radix tree) – это структура данных для реализации словаря (ассоциативного массива), ключами в котором являются строки

- Практические применения:

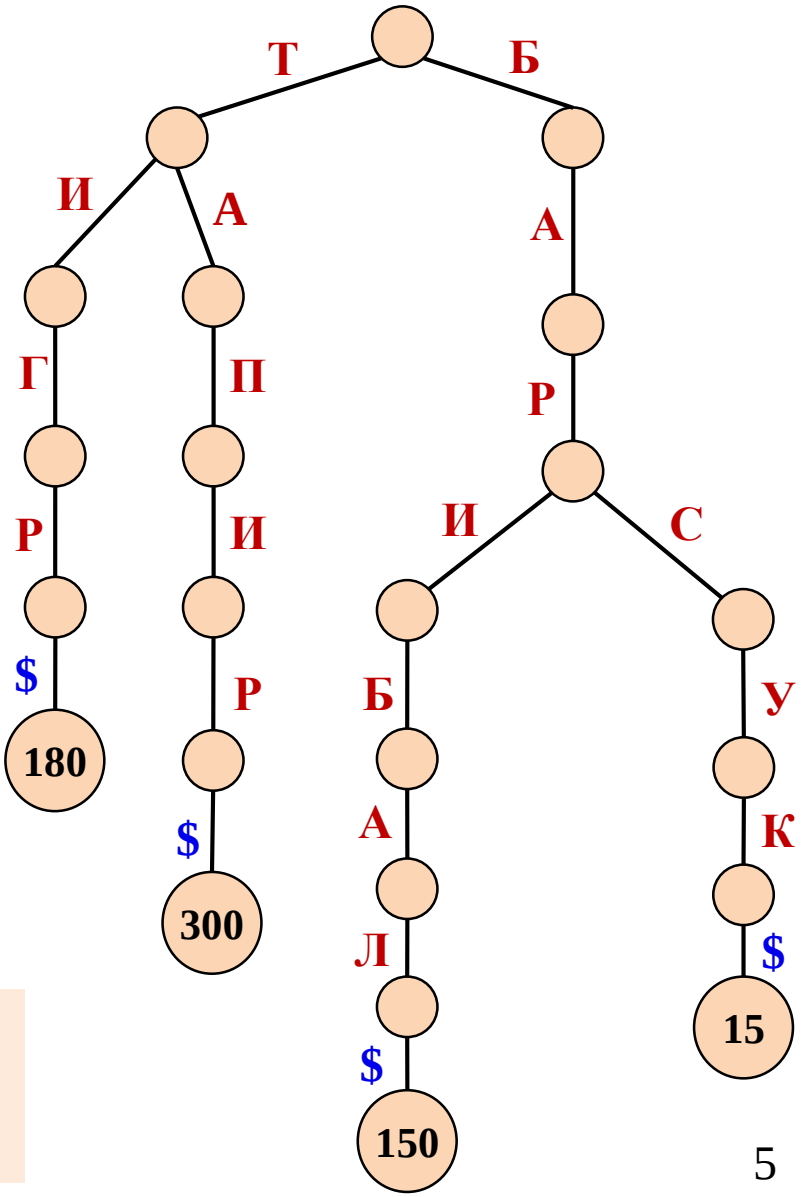
  - Предиктивный ввод текста (predictive text) – поиск возможных завершений слов
  - Автозавершение (Autocomplete) в текстовых редакторах и IDE
  - Проверка правописания (spellcheck)
  - Автоматическая расстановка переносов слов (hyphenation)
  - Squid Caching Proxy Server



Префиксные деревья (Trie)

Словарь

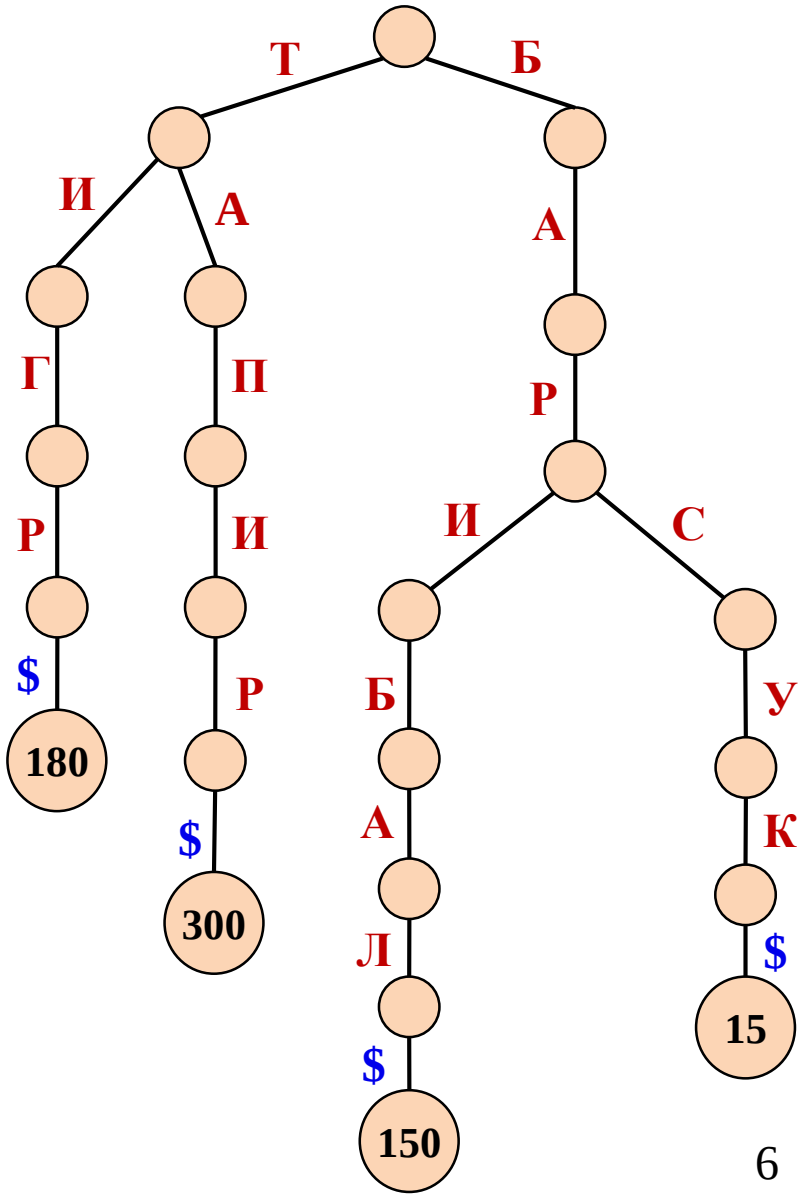
Ключ (Key)	Значение (Value)
ТИГР	180
ТАПИР	300
БАРИБАЛ	150
БАРСУК	15



Неупорядоченное дерево поиска
(ребра одного узла не упорядочены по алфавиту)

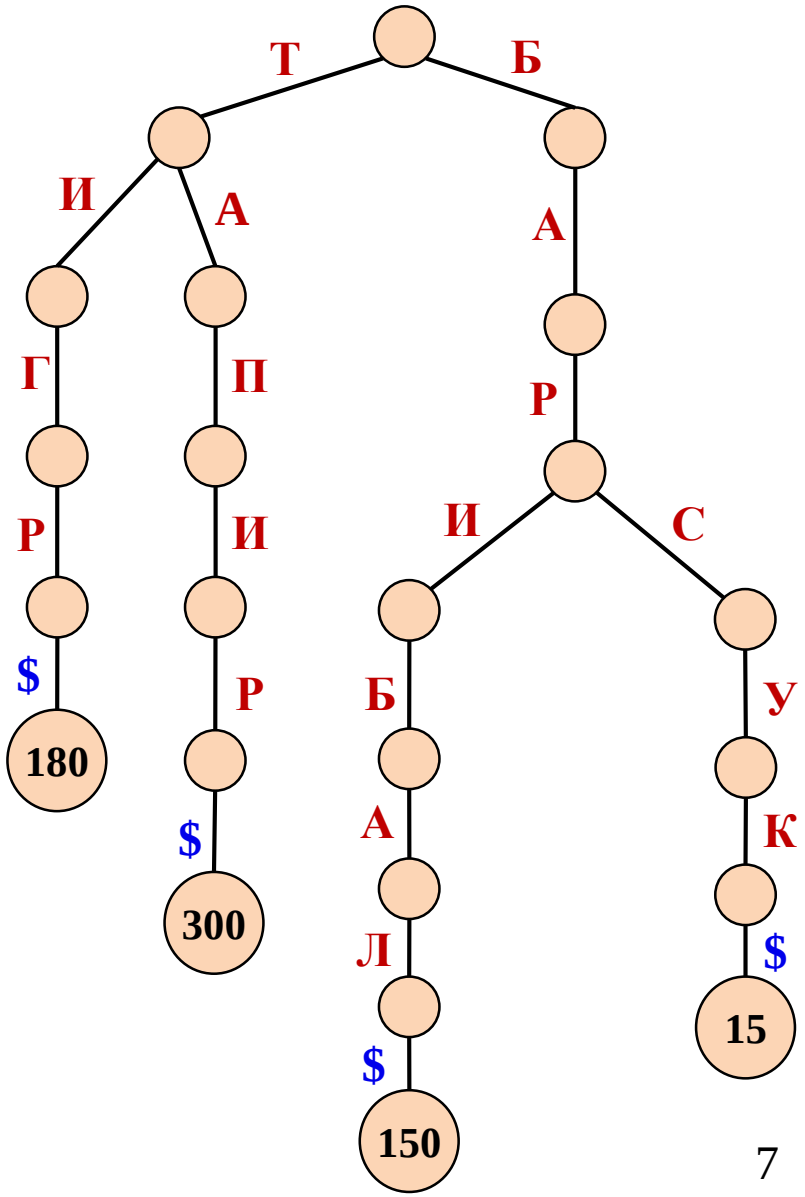
Префиксные деревья (Trie)

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



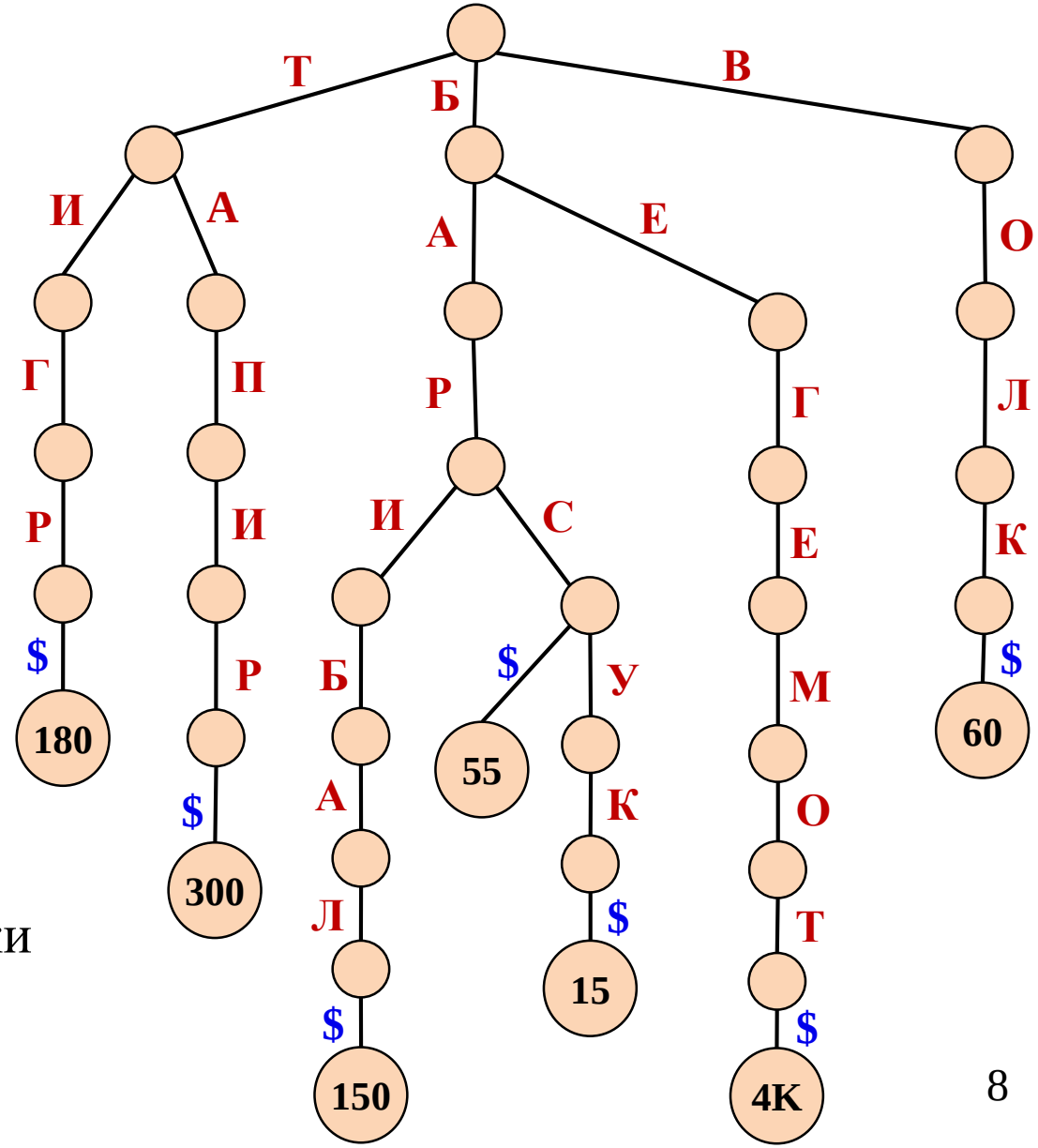
Префиксные деревья (Trie)

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



Префиксные деревья (Trie)

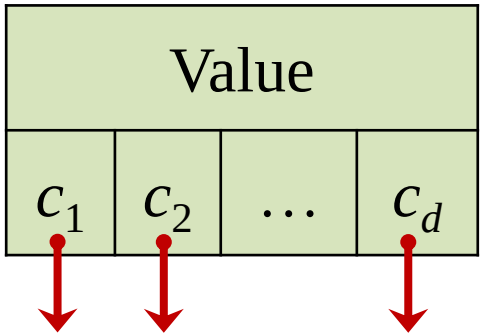
Ключ	Значение
Тигр	180
Барибал	150
Тапир	300
Волк	60
Барсук	15
Бегемот	4000
Барс	55

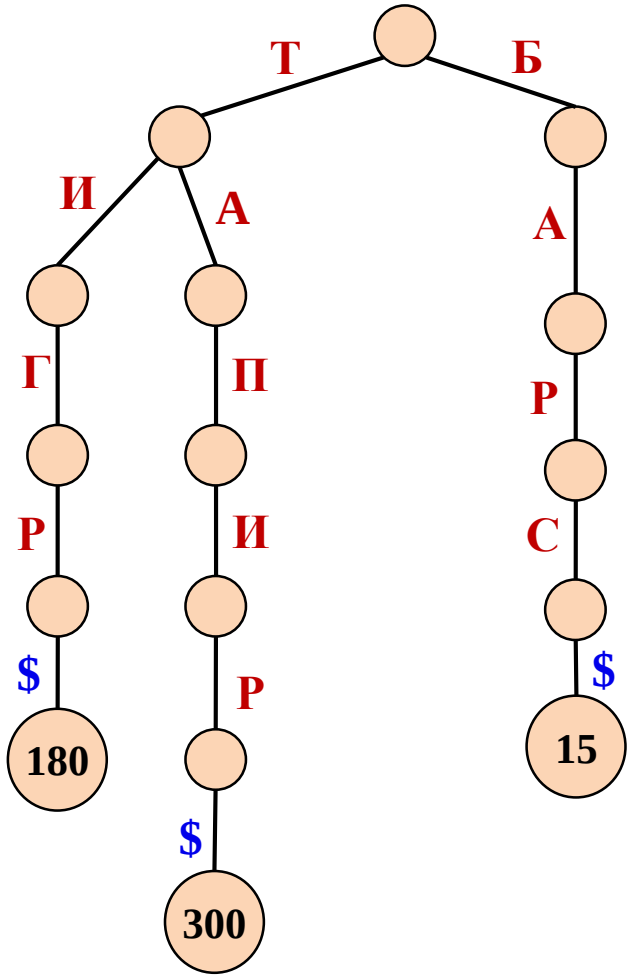


- Символ **$** –
  это маркер конца строки
- Высота $h = \max(key_i)$,
  $i = 1, 2, \ldots, n$

Узел префиксного дерева (Trie)

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

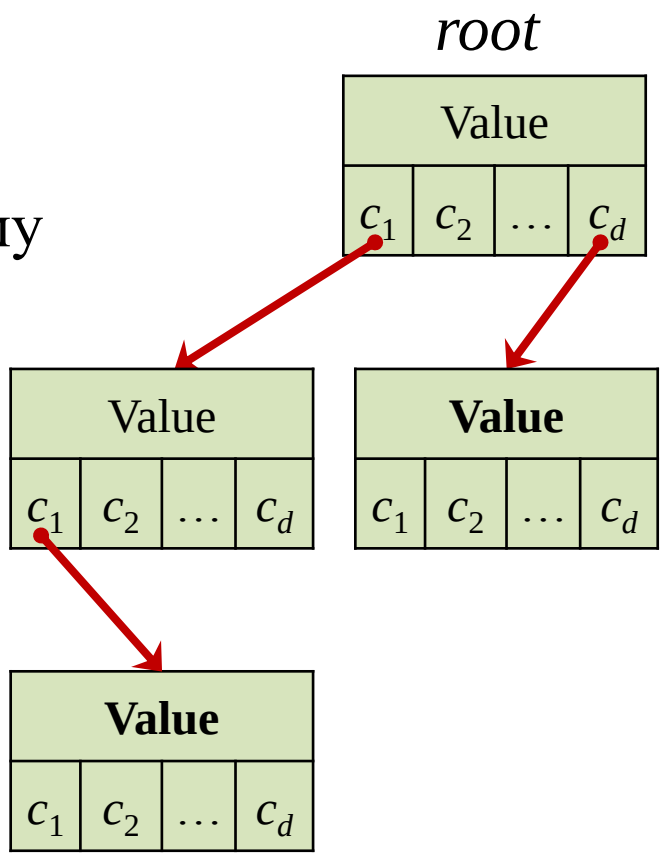




- Как хранить $c_1, c_2, \ldots, c_d$ (массив, список, BST, hash table)?

Вставка элемента в префиксное дерево (Trie)

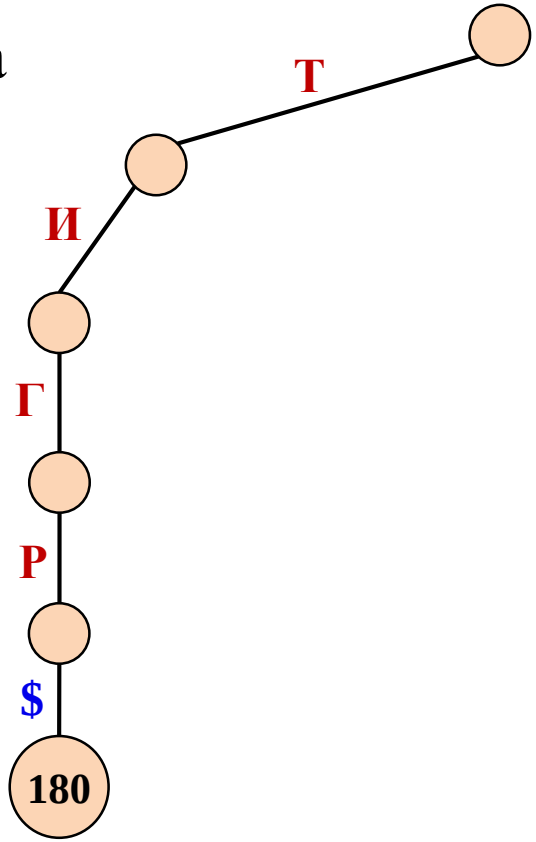
1. Инициализируем $k = 1$
2. В текущем узле (начиная с корня) отыскиваем символ c_i, равный k-ому символу ключа $key[k]$
3. Если $c_i \neq$ NULL, то делаем текущим узел, на который указывает c_i; переходим к следующему символу ключа ($k = k + 1$) и пункту 2



4. Если $c_i =$ NULL, создаем новый узел, делаем его текущим, переходим к следующему символу ключа и пункту 2
5. Если достигли конца строки ($) вставляем значение в текущий в узел

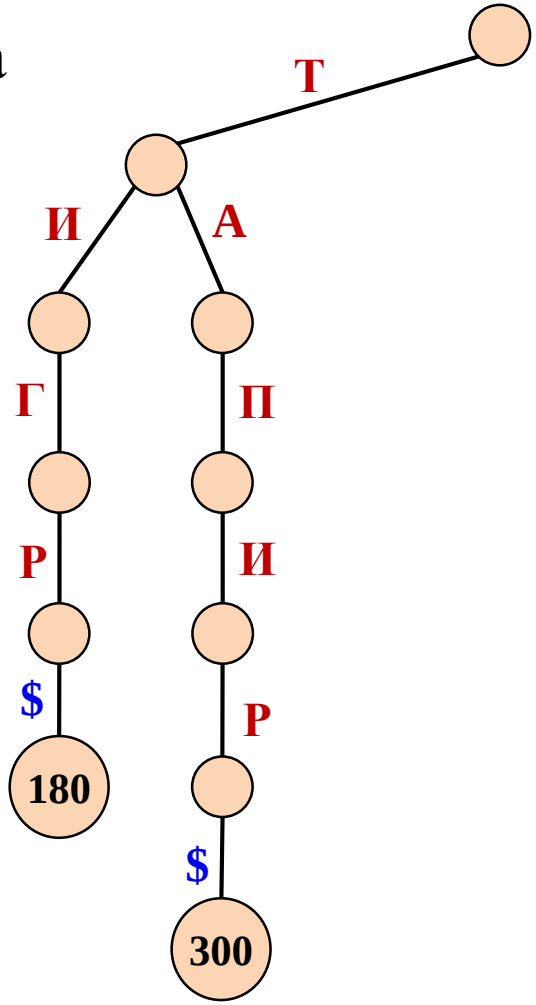
Вставка элемента в префиксное дерево (Trie)

- Добавление элемента (Тигр, 180)



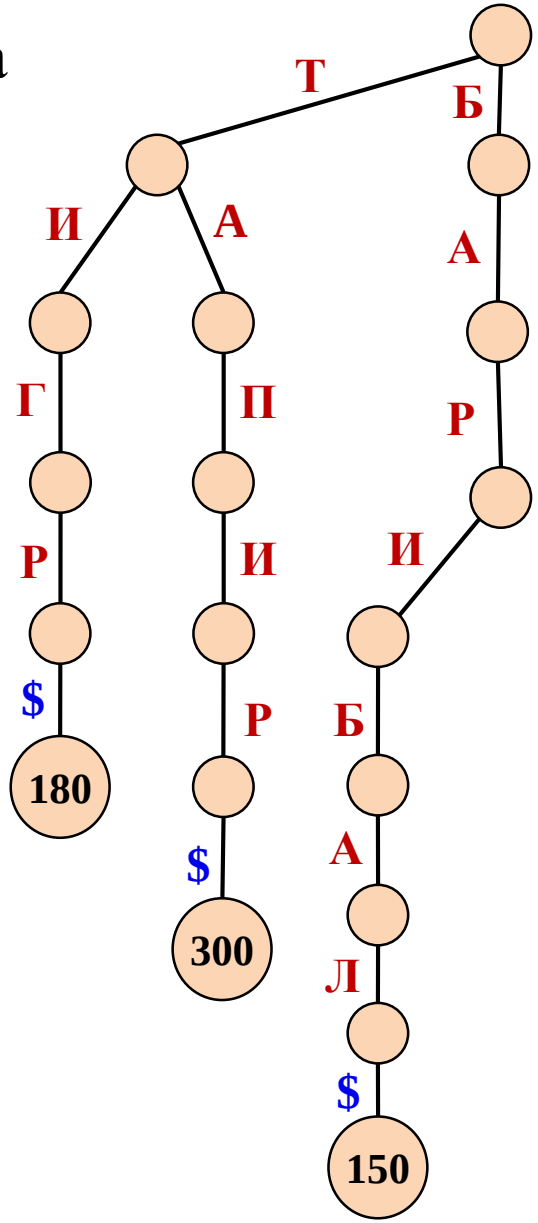
Вставка элемента в префиксное дерево (Trie)

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



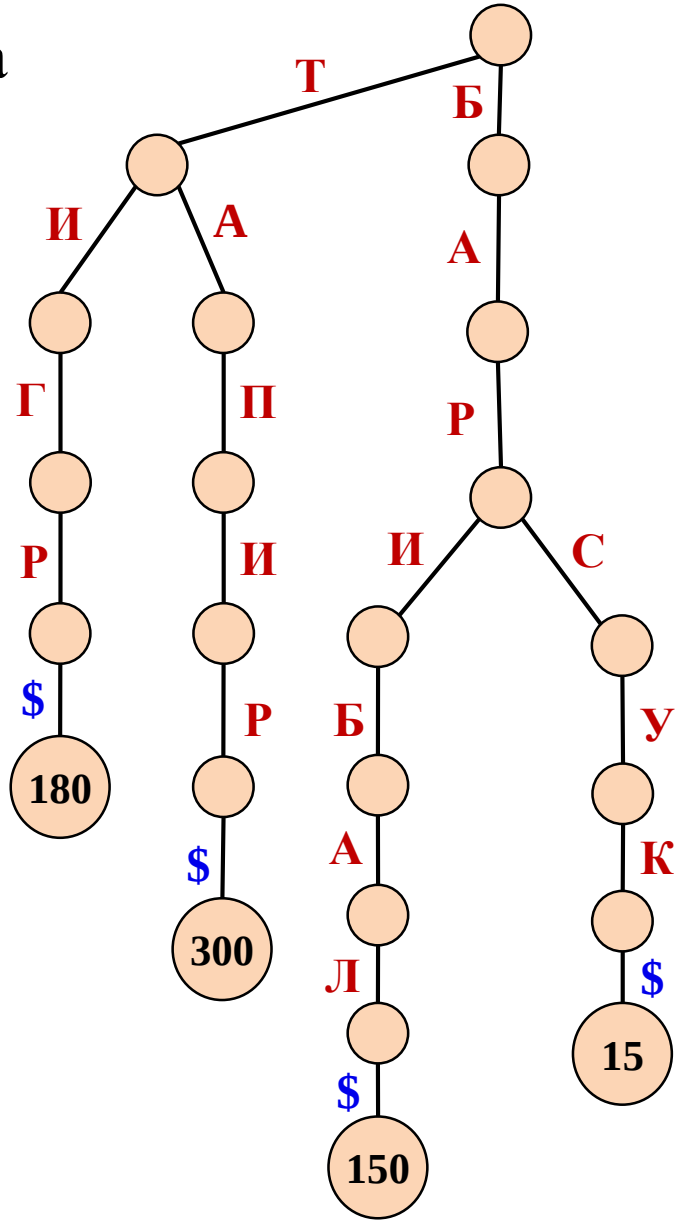
Вставка элемента в префиксное дерево (Trie)

- Добавление элемента (Тигр, 180)
- Добавление (Тапир, 300)
- Добавление (Барибал, 150)



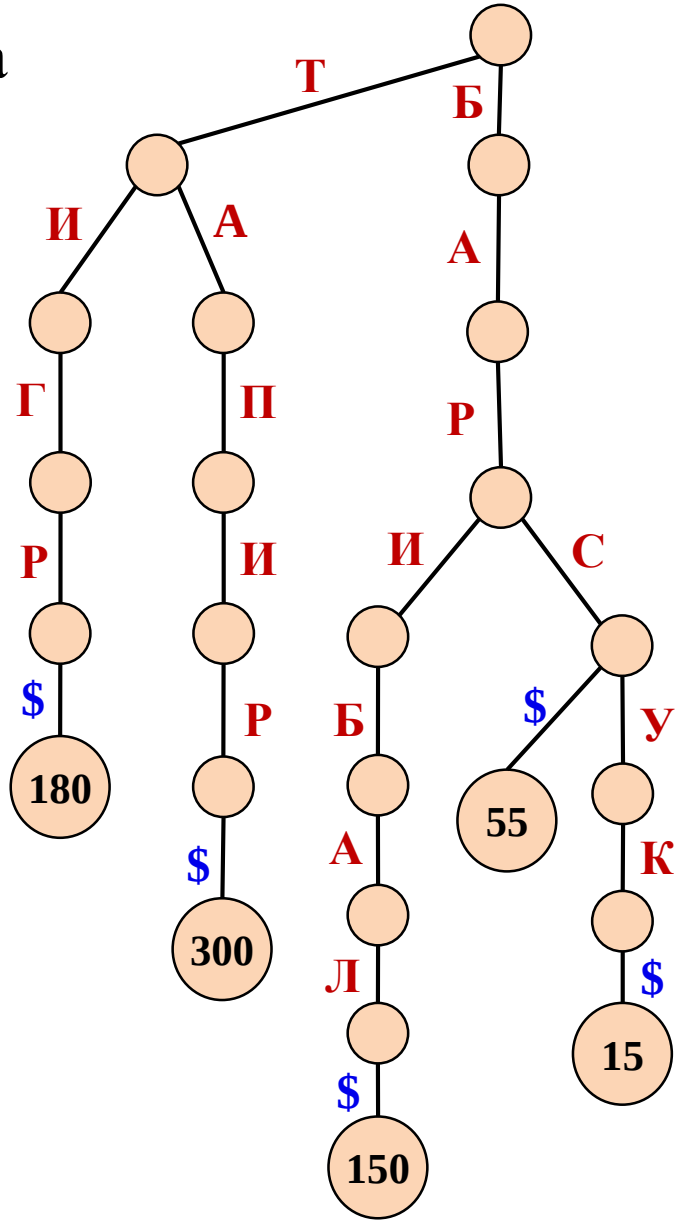
Вставка элемента в префиксное дерево (Trie)

- Добавление элемента (Тигр, 180)
- Добавление (Тапир, 300)
- Добавление (Барибал, 150)
- Добавление (Барсук, 15)



Вставка элемента в префиксное дерево (Trie)

- Добавление элемента (Тигр, 180)
- Добавление (Тапир, 300)
- Добавление (Барибал, 150)
- Добавление (Барсук, 15)
- Добавление (Барс, 55)



Вставка элемента в префиксное дерево (Trie)

```
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 (Trie)

```
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
```

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

- **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 (Trie)

```
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
    return node
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*

Поиск элемента в префиксном дереве (Trie)

```
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
    return node
end function
```

$$T_{Lookup} = O(mT_{GetChild})$$

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

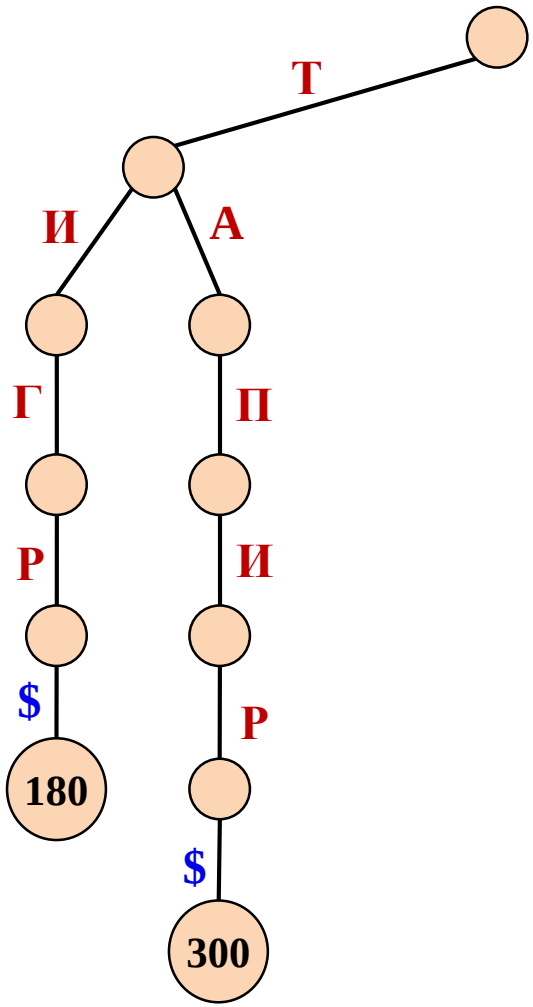
Удаление элемента из префиксного дерева

I. Отыскиваем лист, содержащий искомый ключ *key*

II. Делаем текущим родительский узел найденного листа

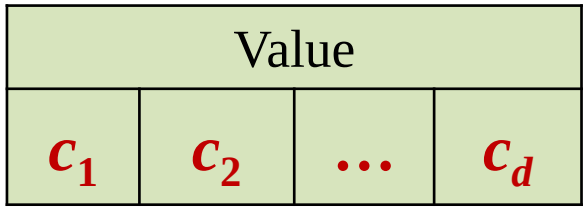
III. Поднимаемся вверх по дереву и удаляем узлы

1. Если текущий узел не имеет дочерних узлов, удаляем его
2. Делаем текущим родительский узел и переходим к пункту 2



Узел префиксного дерева (Trie)

Как хранить указатели c_1, c_2, …, c_d на дочерние узлы?



- **Массив указателей** (индекс – номер символа)

```
struct trie *child[33];
node->child[char_index('Г')]
```

Сложность GetChild/SetChild ***O*(1)**

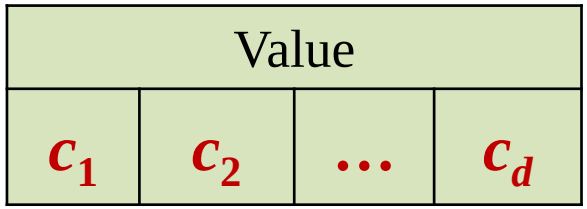
- **Связный список** указателей на дочерние узлы

```
struct trie *child;
linkedlist_lookup(child, 'Г')
```

Сложность GetChild/SetChild ***O*(*d*)**

Узел префиксного дерева (Trie)

Как хранить указатели c_1, c_2, …, c_d на дочерние узлы?



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

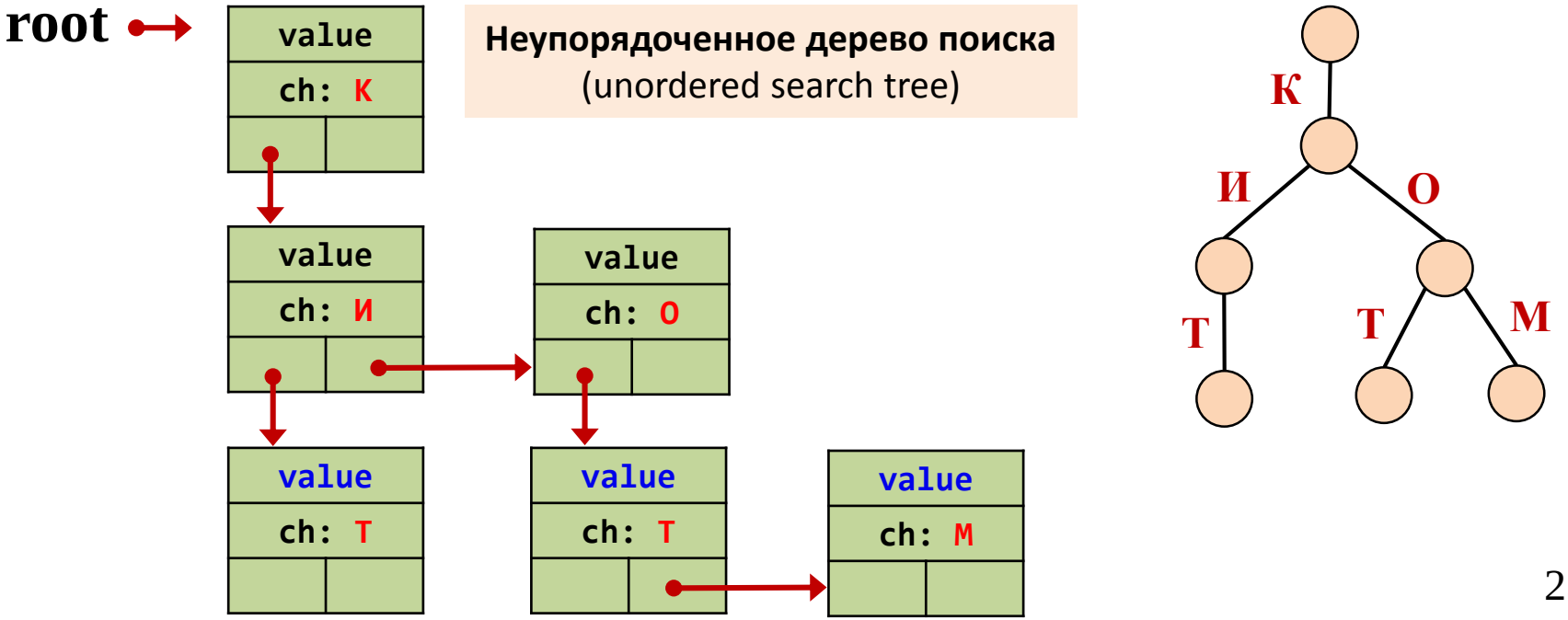
```
struct rbtree *child;
rbtree_lookup(child, 'Г')
```

Сложность GetChild/SetChild ***O*(log*d*)**

Представление узла Trie

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

Сложность GetChild/SetChild: ***O(d)***



Неупорядоченное дерево поиска
(unordered search tree)

Создание пустого узла Trie

```
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;
}
```

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

Поиск узла в Trie 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;
    }
    /* Check: Node must be a leaf node! */
    return node->value;
}
```

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

Вставка узла в Trie

```
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 Trie

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

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

Удаление узла из Trie

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

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

Удаление узла из Trie

```
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 Trie

```
    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;
}
```

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

Вывод на экран элементов Trie

```
void trie_print(struct trie *root, int level)
{
    struct trie *node;
    int i;

    for (node = root; node != NULL;
         node = node->sibling)
    {
        for (i = 0; i < level; i++)
            printf("  ");
        if (node->value != NULL)
            printf("%c (%s)\n", node->ch, node->value);
        else
            printf("%c \n", node->ch);

        if (node->child != NULL)
            trie_print(node->child, level + 1);
    }
}
```

Пример работы с Trie

```
int main()
{
    struct trie *root;

    root = trie_insert(NULL, "bars", "60");
    root = trie_insert(root, "baribal", "100");
    root = trie_insert(root, "kit", "3000");
    root = trie_insert(root, "lev", "500");
    root = trie_insert(root, "bars", "70");
    trie_print(root, 0);

    printf("Lookup 'bars': %s\n",
            trie_lookup(root, "bars"));

    root = trie_delete(root, "bars");
    trie_print(root, 0);

    return 0;
}
```

Вычислительная сложность операций Trie

Операция	Способ работы с указателями c_1, c_2, …, c_d		
	Связный список	Массив	Self-balanced 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)

Преимущества префиксных деревьев

- Время поиска не зависит от количества *n* элементов в словаре (зависит от длины ключа)
- Для хранения ключей на используется дополнительной памяти (ключи не хранятся в узлах)
- В отличии от хеш-таблиц поддерживает **обход в упорядоченном порядке** (от меньших к большим и наоборот, реализует ordered map) – зависит от реализации SetChild/GetChild
- В отличии от хеш-таблиц не возникает коллизий

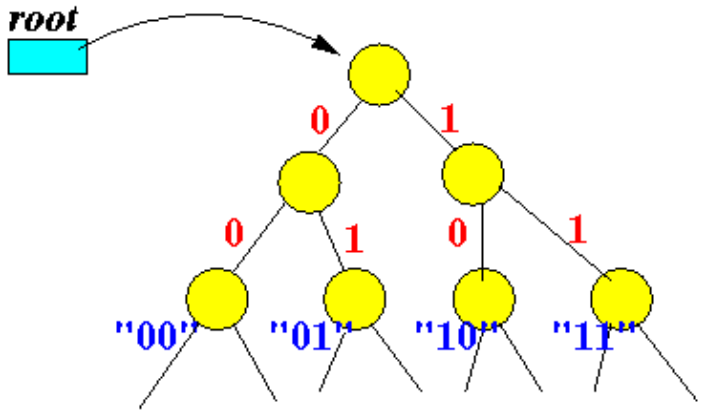
Производительность строковых словарей

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

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

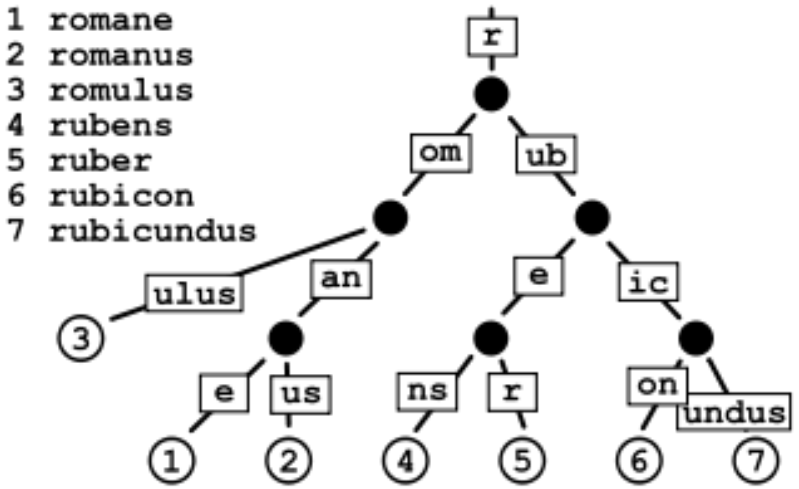
Bitwise tree

- **Bitwise trie –** префиксное дерево (trie), в котором ключи рассматривается как последовательность битов
- Bitwise trie – это бинарное дерево



Radix tree (patricia trie)

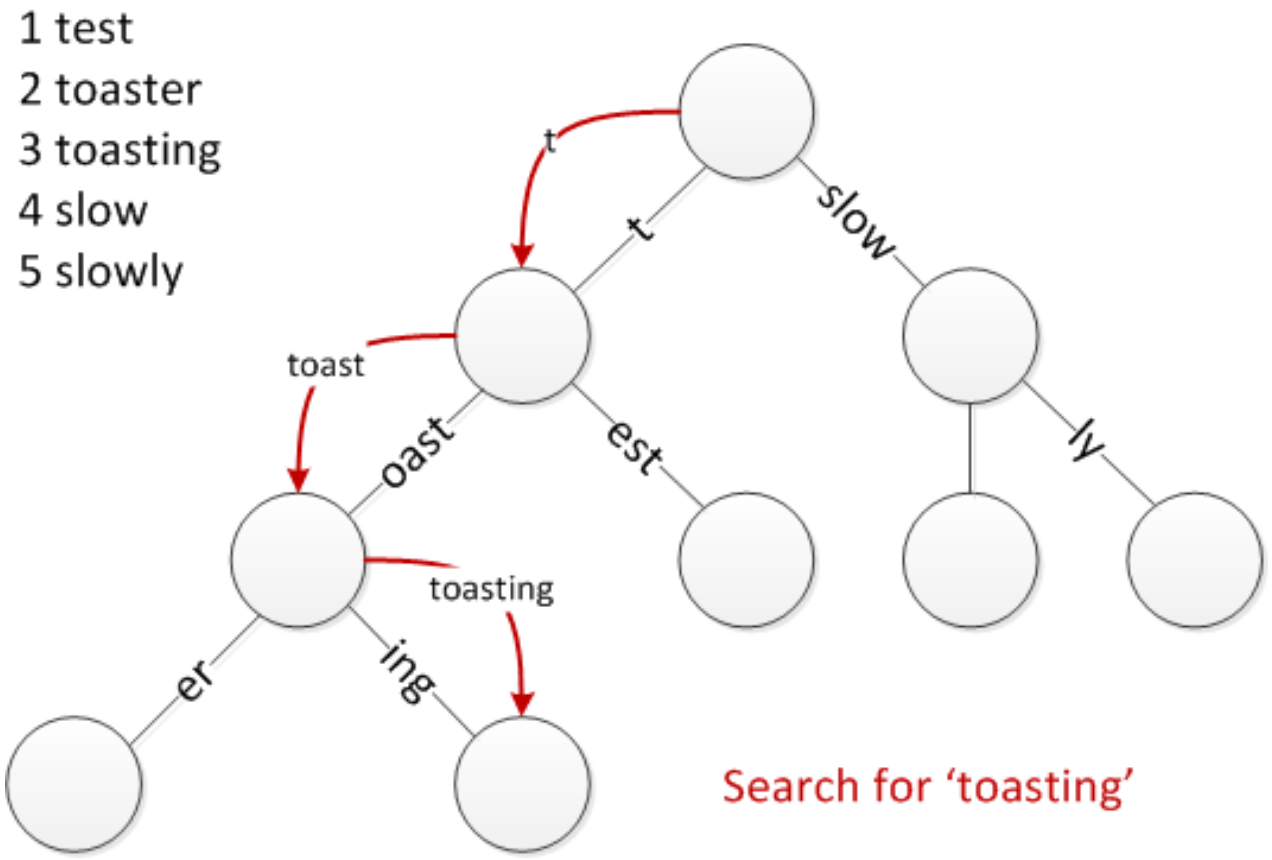
- **Radix tree (radix trie, patricia trie, compact prefix tree)** – префиксное дерево (trie), в котором узел содержащий один дочерний элемент объединяется с ним для экономии памяти



PATRICIA trie:

- D. R. Morrison. **PATRICIA – Practical Algorithm to Retrieve Information Coded in Alphanumeric**. Jrnl. of the ACM, 15(4). pp. 514-534, Oct 1968.
- Gwehenberger G. **Anwendung einer binären Verweiskettenmethode beim Aufbau von Listen**. Elektronische Rechenanlagen 10 (1968), pp. 223–226

Radix tree (patricia trie)



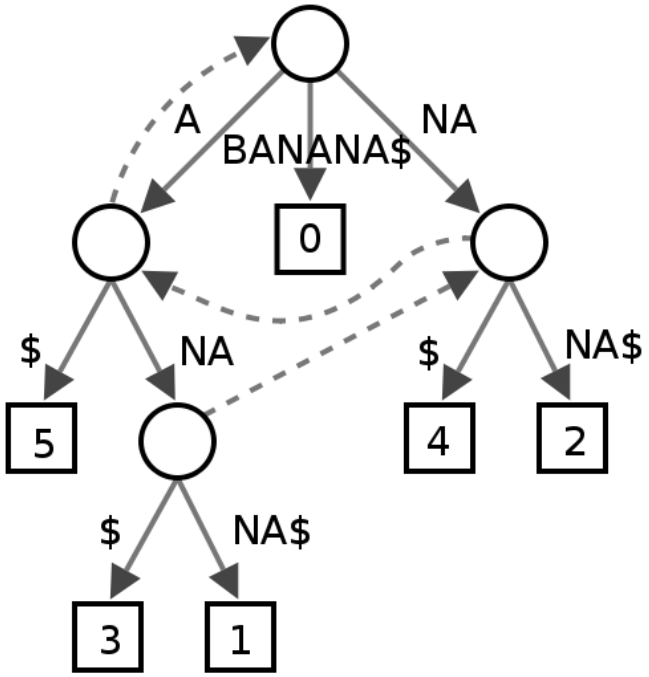
http://en.wikipedia.org/wiki/Radix_tree

Suffix tree

- **Suffix tree (PAT tree, position tree) –** префиксное дерево (trie), содержащее все суффиксы заданного текста (ключи) и их начальные позиции в тексте (values)

- **Суффиксное дерево для текста BANANA**

  Суффиксы: A$, NA$, ANA$, NANA$, ANANA$, BANANA$



- **Применение:**
  - Поиск подстроки в строке $O(m)$
  - Поиск наибольшей повторяющейся подстроки
  - Поиск наибольшей общей подстроки
  - Поиск наибольшей подстроки-палиндрома
  - Алгоритм LZW сжатия информации

- Автор: Weiner P**. Linear pattern matching algorithms** // 14th Annual IEEE Symposium on Switching and Automata Theory, 1973, pp. 1-11

Литература

- 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.