

Лекция 4

Префиксные деревья (trie, prefix tree)

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com

WWW: www.mkurnosov.net

Курс «Структуры и алгоритмы обработки данных»

Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)

Осенний семестр, 2015

Словарь со строковыми ключами

- При анализе вычислительной сложности операций бинарных деревьев поиска (АВЛ-деревьев, красно-черных деревьев и др.), списков с пропусками (skip lists) мы полагали, 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;
}
```

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

Префиксные деревья (trie)

- **Префиксное дерево** (trie, prefix tree, digital tree, radix tree) – структура данных для реализации множества/словаря, ключами в котором являются строки
- **Авторы:** Briandais, 1959; Fredkin, 1960
- Происхождение слова «trie» – *retrieval* (поиск, извлечение, выборка, возврат)
- Альтернативные названия:
 - ***бор*** – Д. Кнут, Т. 3, 1978, выборка
 - ***луч*** – Д. Кнут, Т. 3, 2000, получение
 - ***нагруженное дерево*** – А. Ахо и др., 2000

Префиксные деревья (trie)

Techniques

Trie Memory*

EDWARD FREDKIN, Bolt Beranek and Newman, Inc., Cambridge, Mass.

Introduction

Trie memory is a way of storing and retrieving information.¹ It is applicable to information that consists of function-argument (or item-term) pairs—information conventionally stored in unordered lists, ordered lists, or pigeonholes.

The main advantages of trie memory over the other memory plans just mentioned are shorter access time, greater ease of addition or up-dating, greater convenience in handling arguments of diverse lengths, and the ability to take advantage of redundancies in the information stored. The main disadvantage is relative inefficiency in using storage space, but this inefficiency is not great when the store is large.

In this paper several paradigms of trie memory are described and compared with other memory paradigms, their advantages and disadvantages are examined in detail, and applications are discussed.

Many essential features of trie memory were mentioned by de la Briandais [1] in a paper presented to the Western Joint Computer Conference in 1950. The present development is essentially independent of his, having been described in memorandum form in January 1959 [2], and it is fuller in that it considers additional paradigms (finite-dimensional trie memories) and includes experimental results bearing on the efficiency of utilization of storage space.

Basic Paradigm of Trie Memory

Let us consider first a simple abstract form of trie memory. Suppose that we need to keep track of a set of words, a set of sequences of alphabetic characters. The words are of various lengths. From time to time, additions to the set and deletions from it must be made. What we have to remember is just which ones, of all the possible finite sequences of alphabetic characters, are currently in the set. That is, given a word, we must be able to determine whether or not it is at present a member. In this example, each word is an argument, and the corresponding function

is simply the binary variable of which the admissible values are *member* and *nonmember*.

At the outset, before a storage is begun, the trie is merely a collection of registers. Except for two special registers, which we may call α and δ , every register has a cell for each member (type) of the ensemble of alphabetic characters. If we let that ensemble include a "space" to indicate the end of a word (argument), each register must have 27 cells.

Each cell has space for the address of any register in the memory. Cells in the trie that are not yet being used to represent stored information always contain the address of the special α register. A cell thus represents stored information if it contains the address of some register other than α . The information it represents is its own name, "A" for the A cell, "B" for the B cell, etc., and the address of the next register in the sequence.

Storage of words of alphabetic characters is illustrated in Fig. 1. For the sake of simplicity, the ensemble of characters has been restricted to the first five letters of the alphabet and ∇ for "space." Suppose that we want to store DAB, BAD, BADE, BE, BED, BEAD, CAB, CAD, and A. We may follow the procedure illustrated in Fig. 1, in which the rows represent registers, each one

	A	B	C	D	E	∇
17						1
16						1
15						1
14		15		16		1
13	14					1
12						1
11			12			1
10						1
9	11		10			1
8						1
7				8		1
6			7			1
5	6				9	1
4						1
3		4				1
2	3					1
1	17	5	13	2		1

FIG. 1. Schematic representation of storage of words in trie memory.

* The work reported here was begun at the MIT Lincoln Laboratory and completed at Bolt Beranek and Newman, Inc., with contractual support from the IBM Federal Systems Division. The author wishes to acknowledge his indebtedness to the many people—colleagues, friends—who have helped on this project. Especially, gratitude is expressed to Dr. J. C. R. Licklider for the most pervasive assistance.

¹ Ed. NOTE: "Trie" is apparently derived from *reTRIEval*.

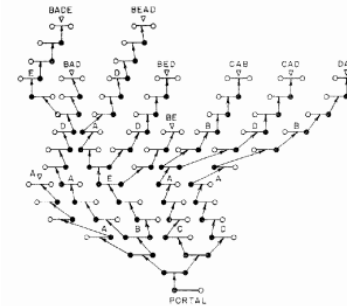


FIG. 5. Storage of words in a character-coded binary tree.

representation of alphanumeric characters that is given in Table 1, we may store the information of Fig. 1 in the binary trie of Fig. 5. For convenience, we may represent the memory in the form of a *tree*. Note, however, that the branches are constructed as needed. In this respect, a trie memory tree is quite different from a system of pigeonholes accessible through a predetermined decision tree.

Each memory register has only two cells, 0 and 1. Three levels are therefore required to store a character from the 6-character ensemble. Reading and writing are accomplished by exactly the same general procedure described in connection with multi-cell memories. It is important to note that the branching, tree-like structure is a result of an attempt to simplify the diagram; in principle, directly connected registers need not be physically near each other.

N-Dimensional Binary Trie's

As a memory is made larger, the number of bits required to specify one register (location) out of the entire set of registers increases, and we would like to have a way of minimizing the effect of that increase.

One approach is to impose a constraint on the set of registers at level $i + 1$ that are accessible from a register at level i . Such a constraint would let us move from the present register, not to any register selected from the entire set, but only to a neighboring register. The number of "neighbors" must be fairly large, of course, or we shall be too likely to be trapped, to find that all the neighbors have already been pre-empted by other storage paths previously laid down. What we need is a simple way of defining neighbors that will yield a fairly large number of neighbors and yet facilitate coding their locations.

We may think of a memory facility as being organized in N dimensions. For the sake of simplicity, we may restrict ourselves to move in only the positive direction along only

one dimension at a time. Then, if the present register has the location i, j, k, \dots , its neighbors are the registers with locations $i + 1, j, k, \dots$; $i, j + 1, k, \dots$; $i, j, k + 1, \dots$; etc. This is illustrated for a three-dimensional structure in Fig. 6. The number of neighbors is equal to the dimensionality of the structure, and the number of bits required to specify a relative location is only $\log_2 N$.

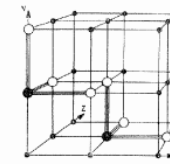


FIG. 6. Three-dimensional memory structure.

In these terms, the unconstrained trie's discussed in previous sections were $(n - 1)$ -dimensional trie's, n being the number of registers. When we refer to N -dimensional trie's, however, we shall imply that N is considerably smaller than $n - 1$.

Registers are represented by nodes. If one of the large solid nodes is the present register, the next register must be one of the large open nodes connected to it by heavy lines. When a new register is needed during the storage phase, we may examine those nodes, one at a time, in ordered sequence or in random sequence. As soon as we find an empty one, we use it, recording its relative location (dimension) at the large solid node. If all three (or, in general, all N) nodes have been pre-empted, we are blocked, and we must use some procedure outside the system.

In one sense, since we can make it any number we choose, N is independent of memory size. However, to make a useful memory, we must choose N in such a way as to keep low the probability that all N of the registers that are neighbors of a given register will have been pre-empted before that register is used. N must therefore be allowed to increase as some function of memory size.

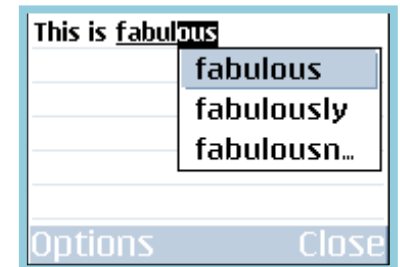
Analysis of the Space Utilization Problem

The problem of space utilization can be approached in two steps. First, we can develop a formula that will structure the problem. Second, we can conduct a Monte Carlo simulation to test the formulas and to provide an empirical measure of the efficiency of utilization.

After many sequences have been stored in a trie, at what level during storage of a new sequence will we cease to find the characters already represented in the trie and have to store the location of a register hitherto not employed? Suppose, to make the problem specific, that we want to store S sequences of b binary digits each, drawn at random without replacement from the 2^b such sequences. What is the probability of having to shift, at

Префиксные деревья (Trie)

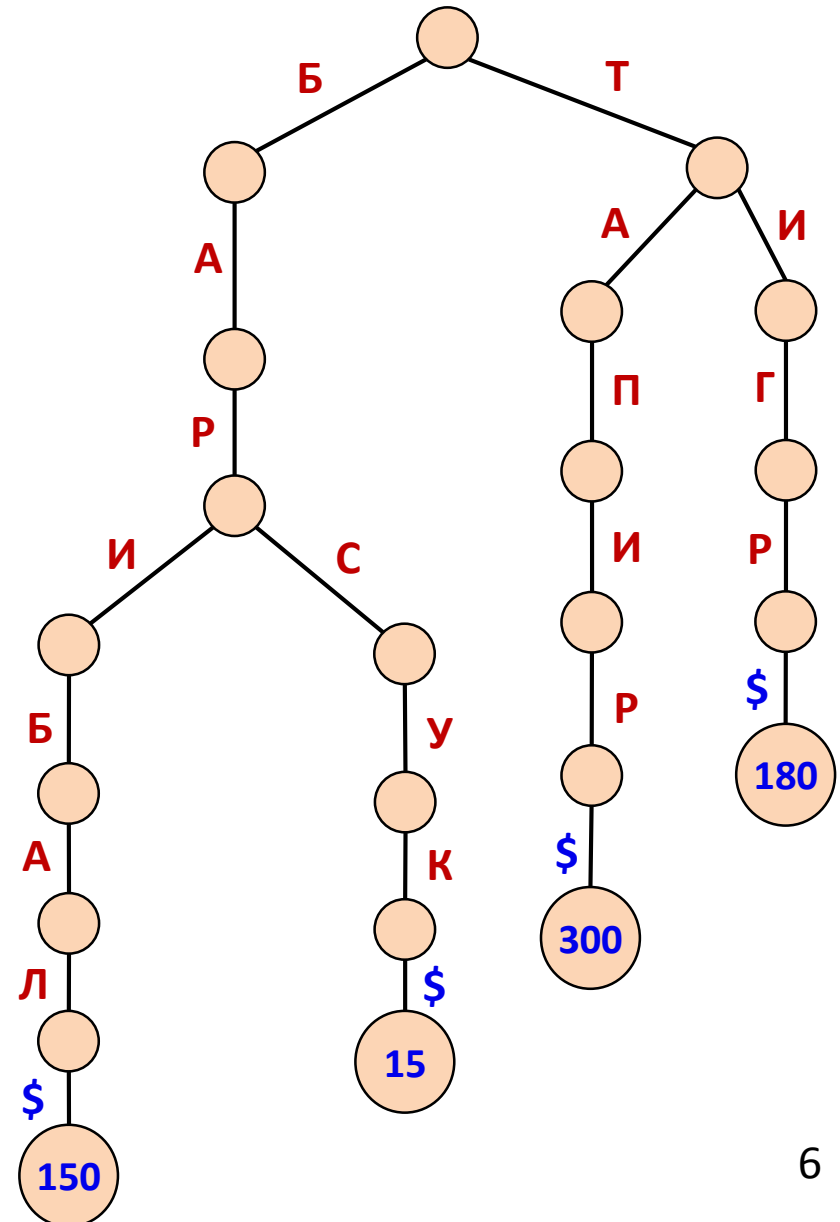
- **Префиксное дерево** (trie, prefix tree, digital tree, radix tree) – структура данных для реализации словаря (ассоциативного массива), ключами в котором являются строки
- **Практические применения:**
 - Предиктивный ввод текста (predictive text) – поиск возможных завершений слов
 - Автозавершение (autocomplete) в текстовых редакторах и IDE
 - Проверка правописания (spellcheck)
 - Автоматическая расстановка переносов слов (hyphenation)
 - Squid Caching Proxy Server



Префиксные деревья (trie)

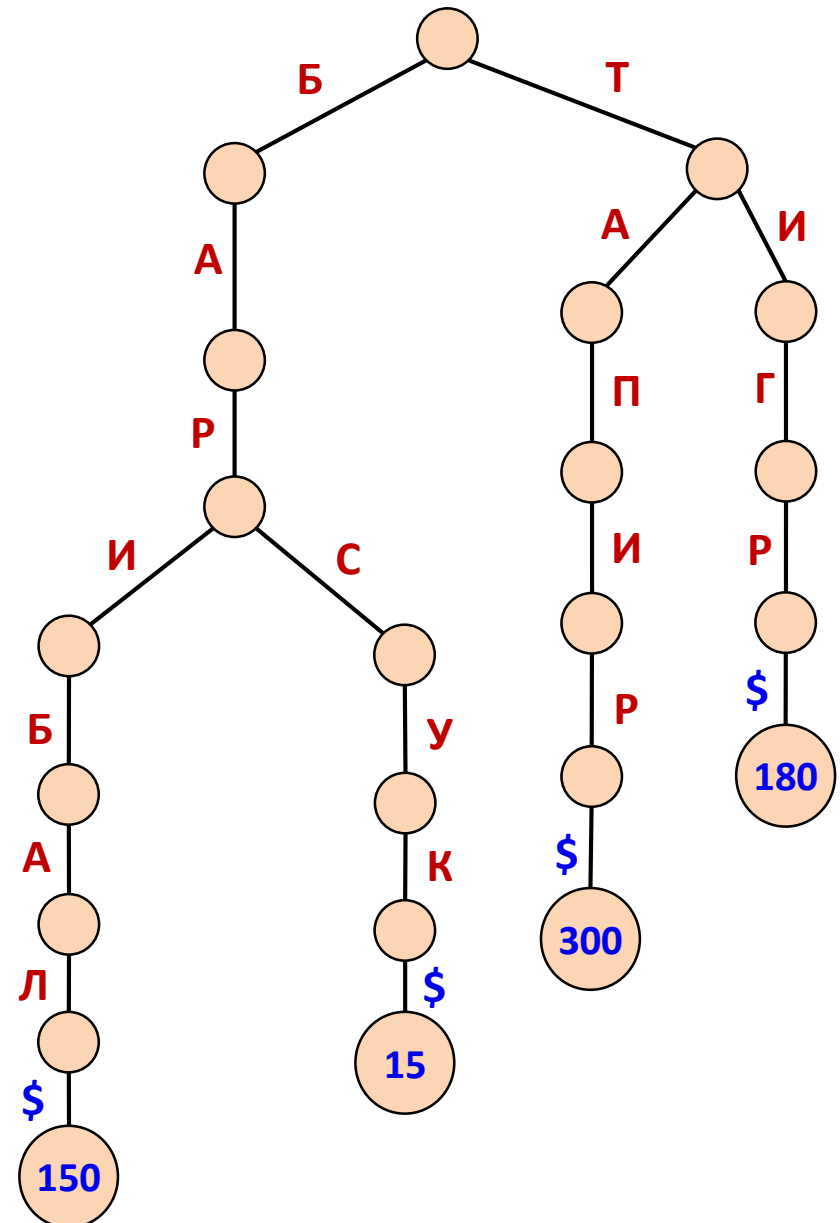
Словарь

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



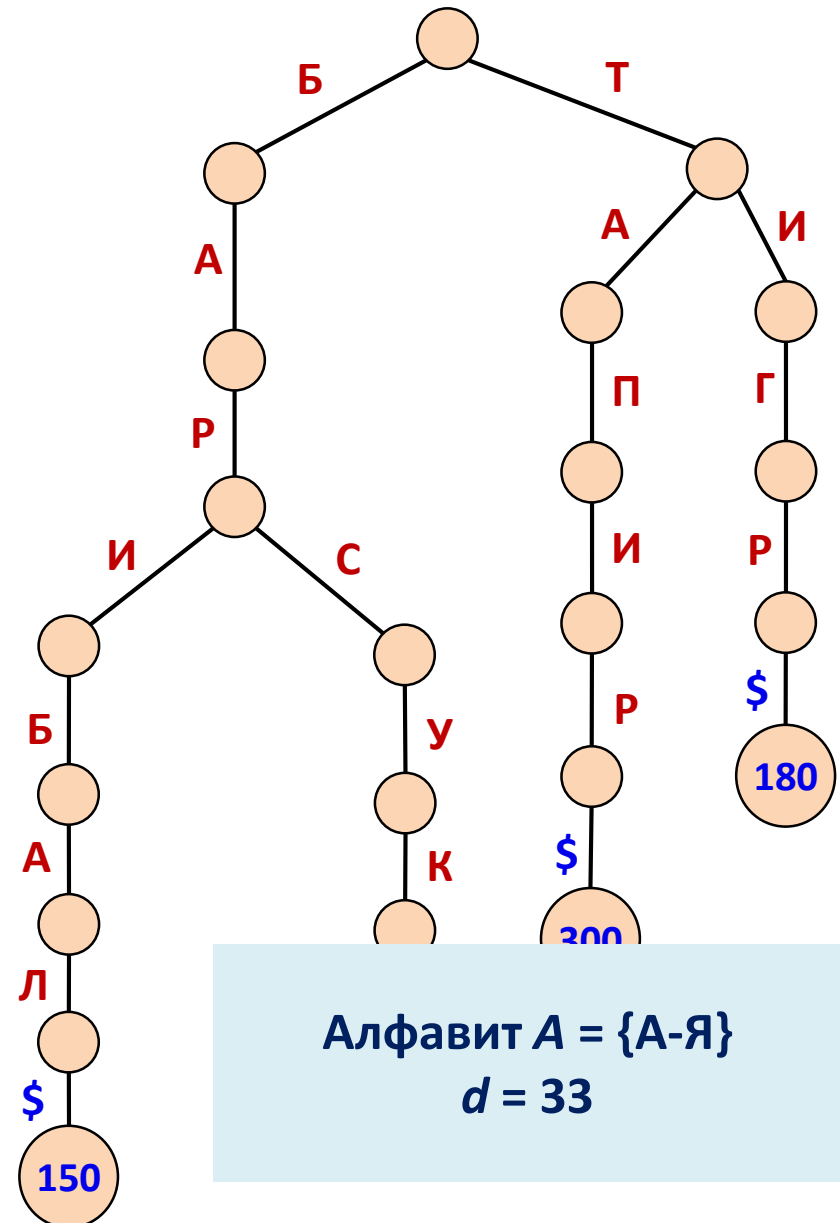
Префиксные деревья (trie)

- **Префиксное дерево** (trie) содержит n ключей (строк) и ассоциированные с ними значения (values)
- **Ключ** (key) – это набор символов (c_1, c_2, \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)



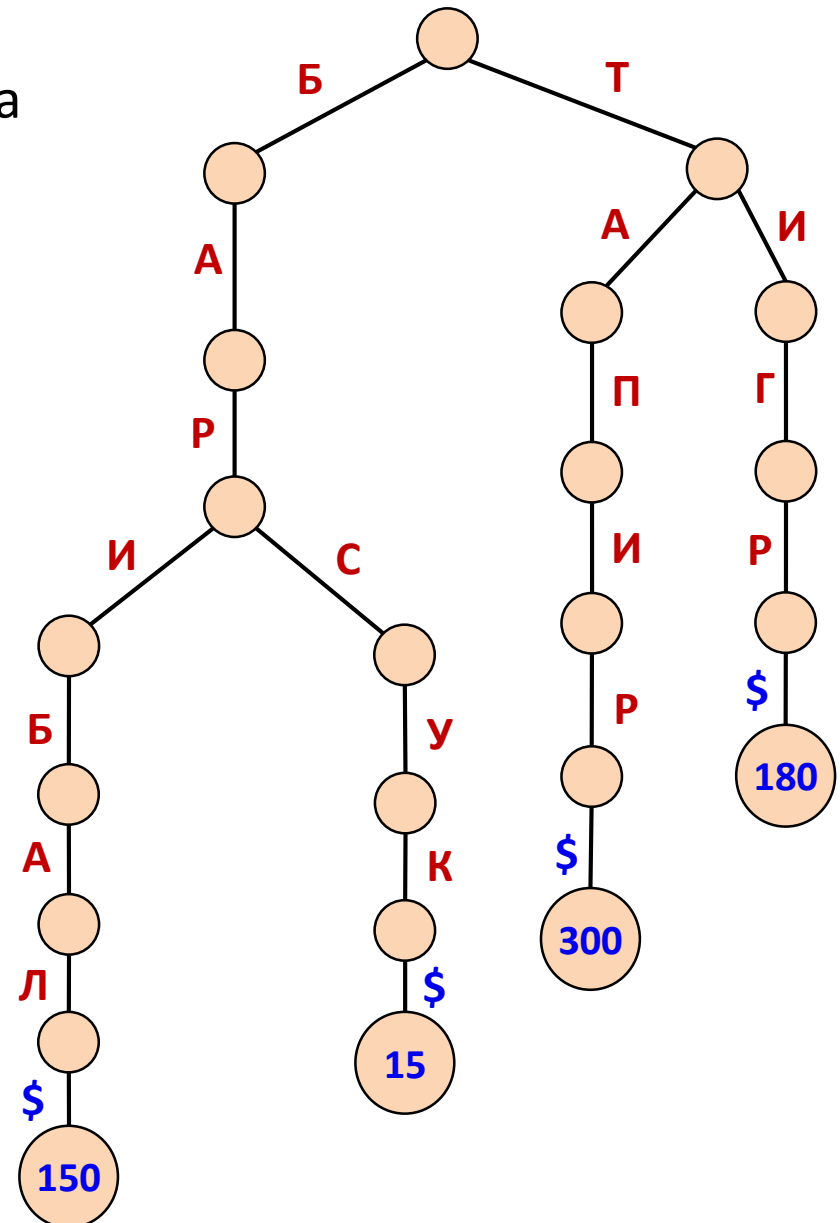
Префиксные деревья (trie)

- **Префиксное дерево** (trie) содержит n ключей (строк) 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 (trie)

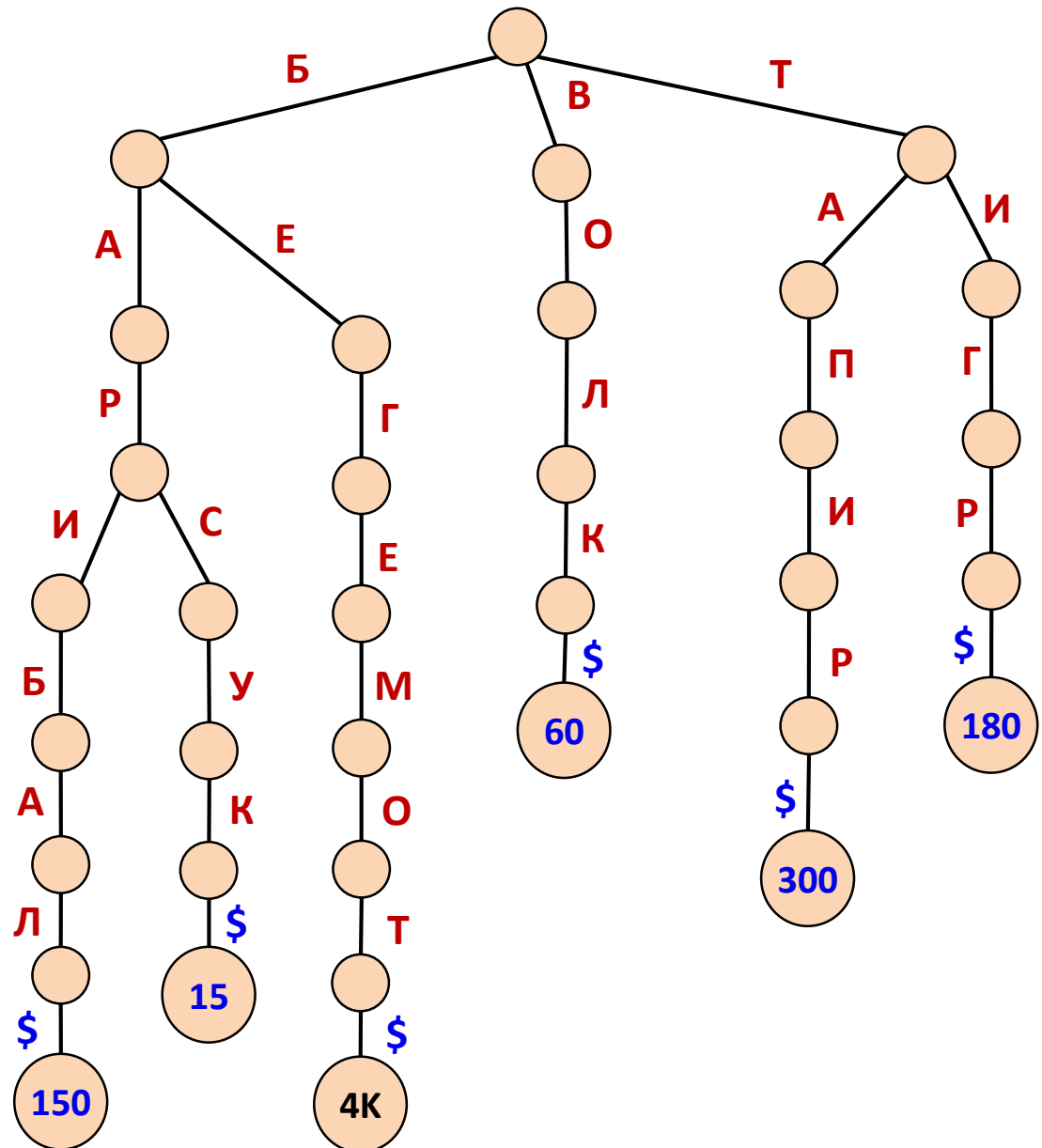
- Ключи не хранятся в узлах дерева
- Позиция листа в дереве определяется значением его ключа
- Значения хранятся в листьях



Префиксные деревья (trie)

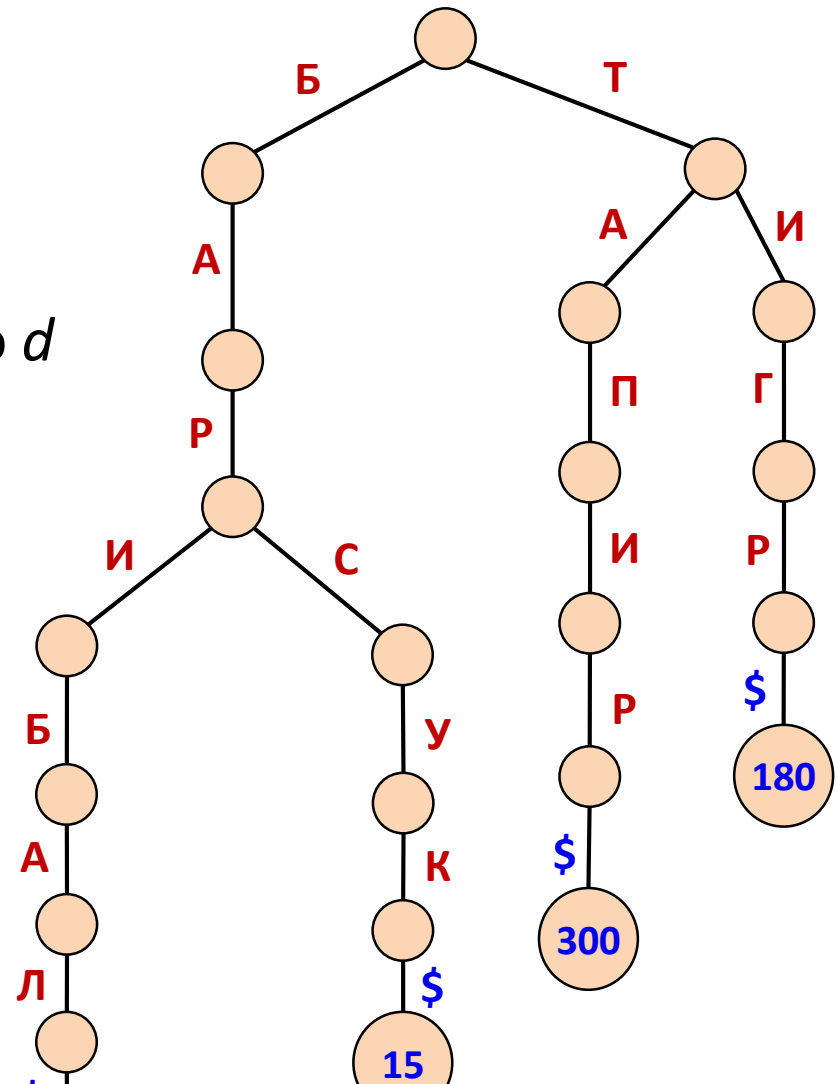
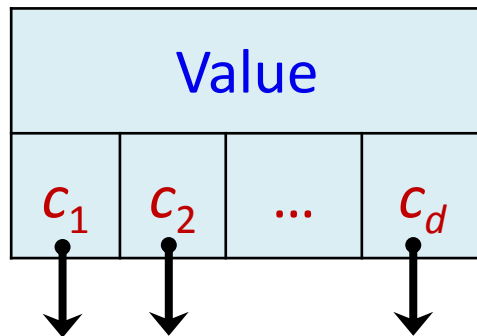
Ключ	Значение
Тигр	180
Барibal	150
Тапир	300
Волк	60
Барсук	15
Бегемот	4000
Барс	55

- Символ \$ — это маркер конца строки
- Высота дерева $h = O(\max(\text{key}_i))$, $i = 1, 2, \dots, n$



Узел префиксного дерева (trie)

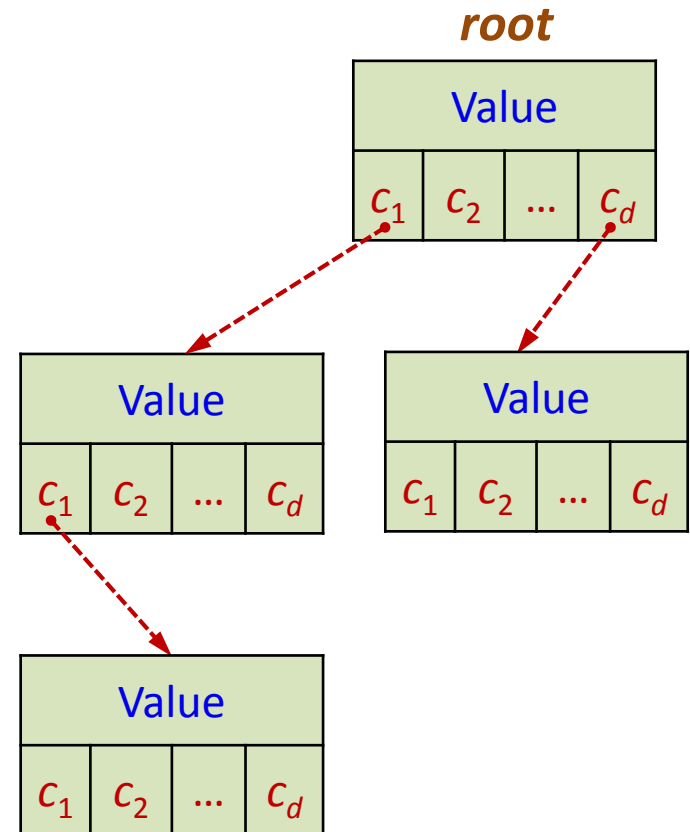
- Ключ – это набор символов (c_1, c_2, \dots, c_m) из алфавита $A = \{a_1, a_2, \dots, a_d\}$
- Каждый узел содержит от 1 до d указателей на дочерние узлы
- Значения хранятся в листьях



Как хранить c_1, c_2, \dots, c_d (массив, список, BST, hash table)?

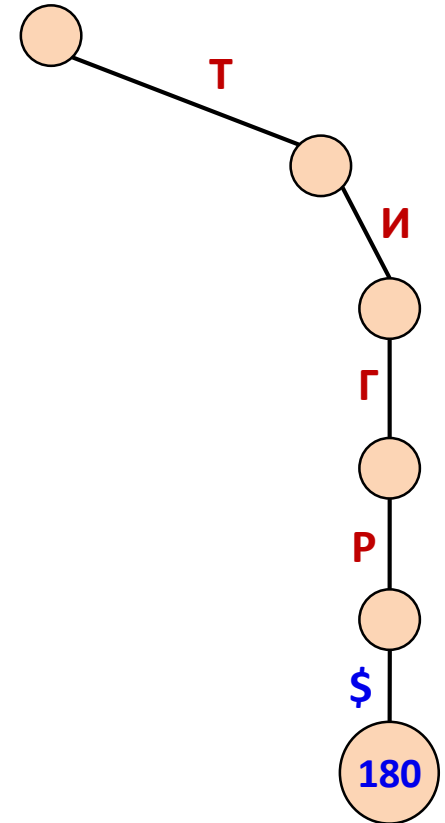
Вставка элемента в префиксное дерево (trie)

1. Инициализируем $k = 1$
2. В текущем узле (начиная с корня) отыскиваем символ c_i , равный k -у символу ключа $key[k]$
3. Если $c_i \neq \text{NULL}$, то
 - a) Делаем текущим узел, на который указывает c_i
 - b) Переходим к следующему символу ключа ($k = k + 1$) и пункту 2
4. Если $c_i = \text{NULL}$, создаем 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)



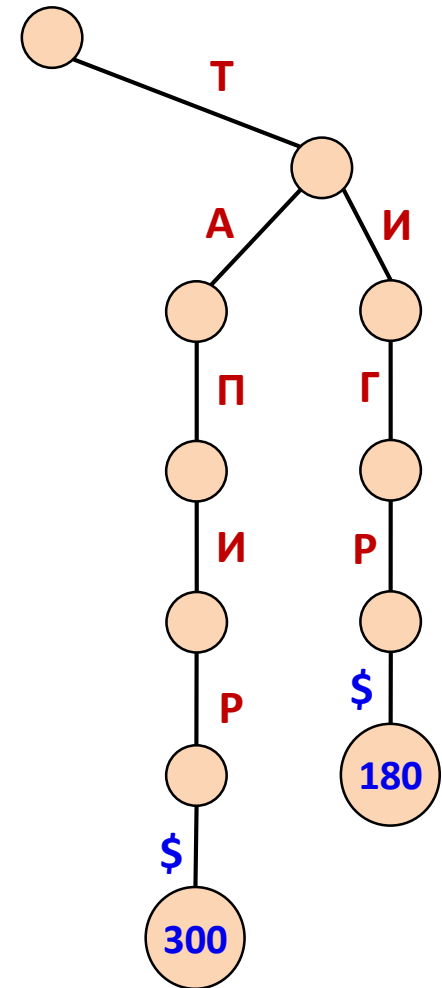
Вставка элемента в префиксное дерево (trie)

- Добавление элемента
(Тигр, 180)



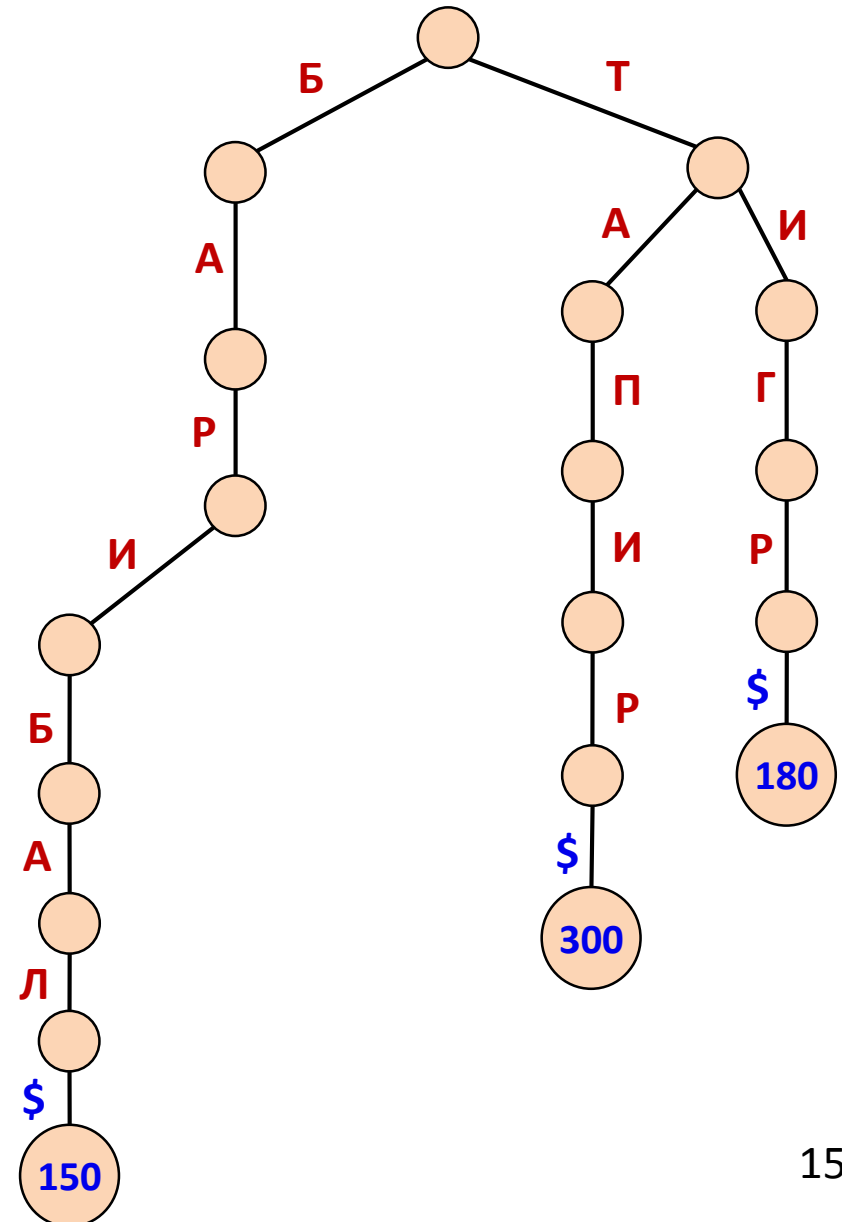
Вставка элемента в префиксное дерево (trie)

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



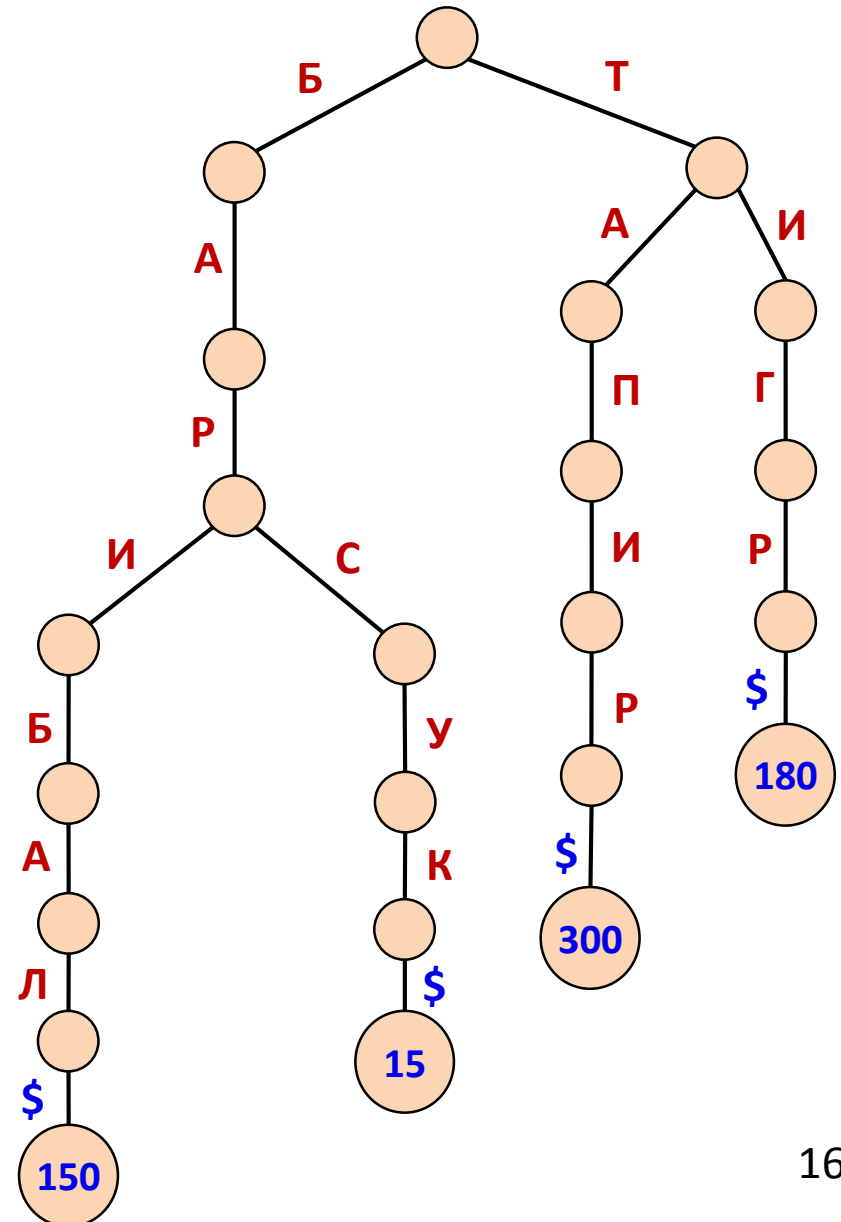
Вставка элемента в префиксное дерево (trie)

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



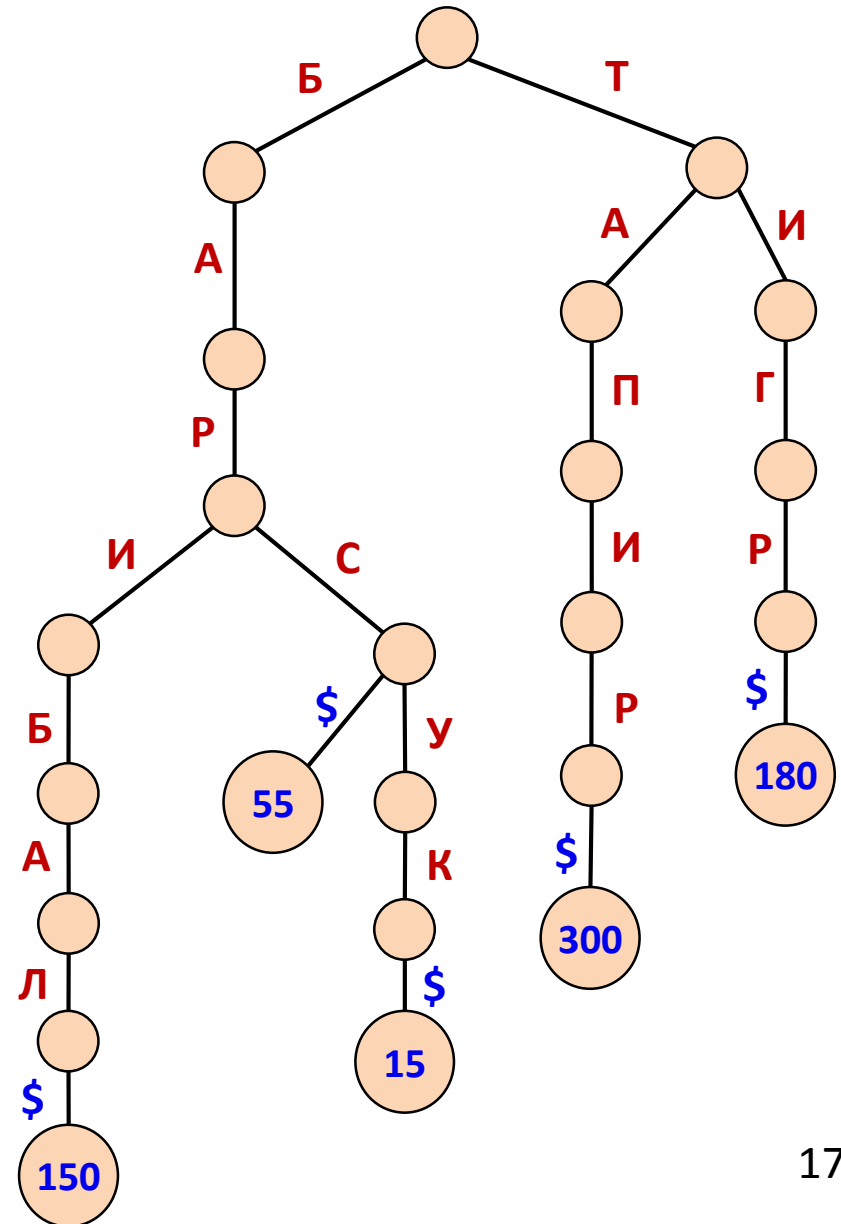
Вставка элемента в префиксное дерево (trie)

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



Вставка элемента в префиксное дерево (trie)

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

  // Ключ присутствует в дереве (не префикс другого)?
  if node.value = NULL then
    return NULL
  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

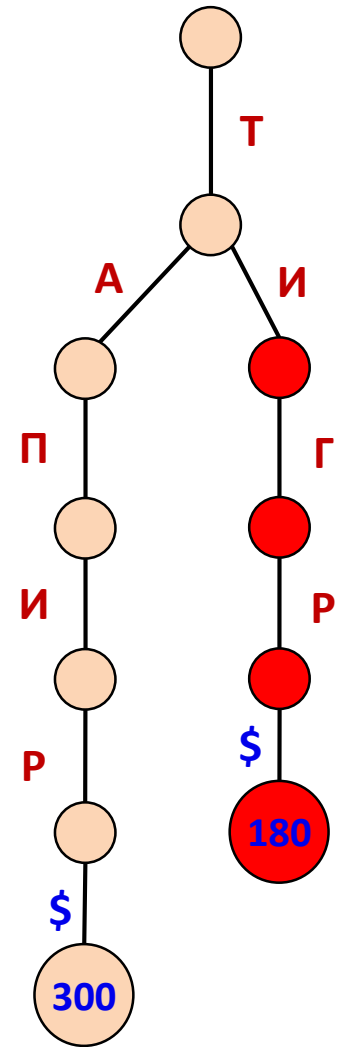
  // Ключ присутствует в дереве (не префикс другого)?
  if node.value = NULL then
    return NULL
  end if
  return node
end function
```

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

- **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

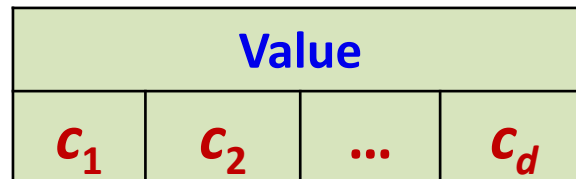
1. Отыскиваем лист, содержащий искомый ключ *key*
2. Если текущий узел не имеет дочерних узлов, удаляем его из памяти (в противном случае заканчиваем подъем по дереву)
3. Делаем текущим родительский узел и переходим к пункту 2



Удаление ключа
ТИГР

Узел префиксного дерева (trie)

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



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

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

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

- **Связный список** указателей на дочерние узлы

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

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

Узел префиксного дерева (trie)

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

Value			
c_1	c_2	...	c_d

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

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

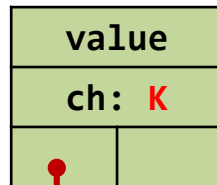
Сложность GetChild/SetChild $O(\log d)$

Представление узла trie

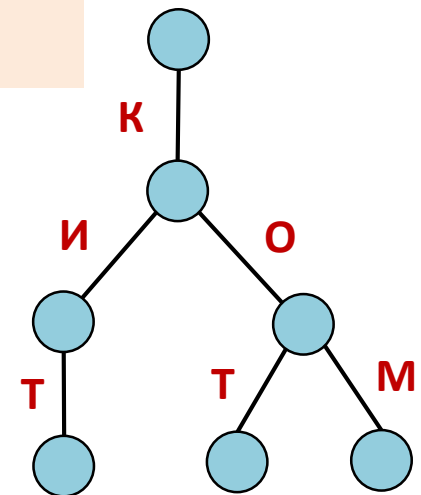
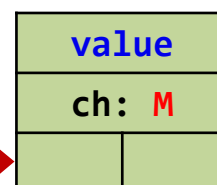
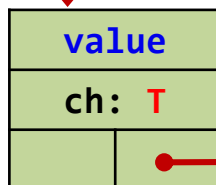
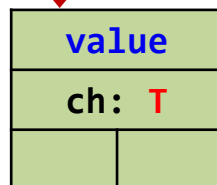
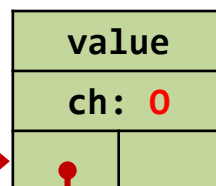
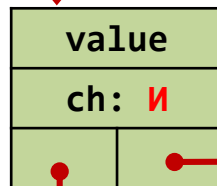
```
struct trie {  
    char *value;  
    char ch;  
    struct trie *sibling;    /* Sibling node */  
    struct trie *child;     /* First child node */  
};
```

Сложность GetChild/SetChild: $O(d)$

root →



Неупорядоченное дерево поиска
(unordered search tree, unordered set)



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;
}
```

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

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, \dots, 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)
- 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 n элементов в словаре (зависит от длины ключа, мощности алфавита)
- Для хранения ключей на используется дополнительной памяти (ключи не хранятся в узлах)
- В отличие от хеш-таблиц поддерживает **обход в упорядоченной последовательности** (от меньших ключей к большим и наоборот, реализует ordered map/set) – зависит от реализации SetChild/GetChild
- В отличие от хеш-таблиц не возникает коллизий

Производительность строковых словарей

Худший случай (worst case)

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

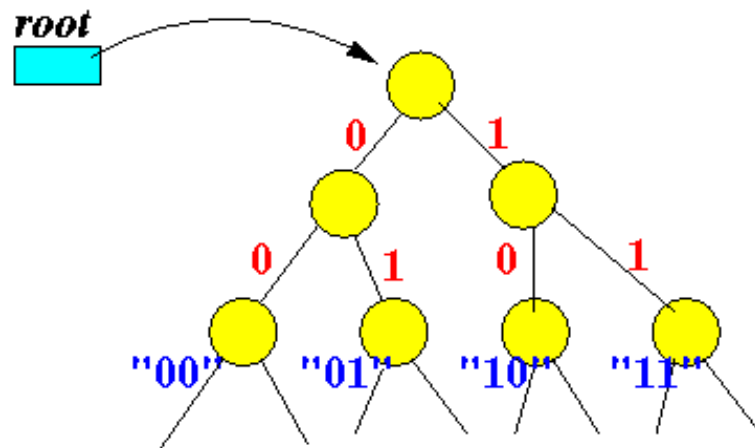
Средний случай (average case)

Операция	Trie (linked list)	Self-balanced search tree (Red black/AVL tree)	Hash table (Chaining)
Lookup	$O(md^*)$	$O(m \log n)$	$O(m + mn/h) = O(m)$
Insert	$O(md^*)$	$O(m \log n)$	$O(m + mn/h) = O(m)$
Delete	$O(md^*)$	$O(m \log n)$	$O(m + mn/h) = O(m)$
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

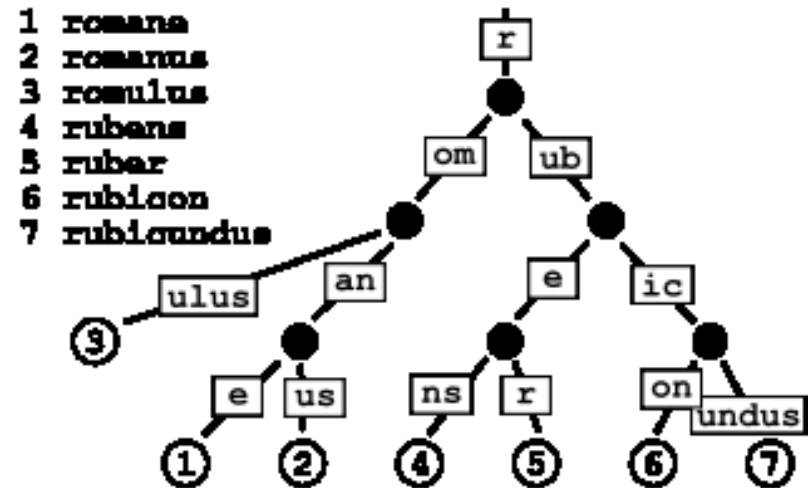
Bitwise tree

- **Bitwise trie** – префиксное дерево (trie), в котором ключи рассматриваются как последовательность битов
- Bitwise trie позволяет хранить ключи произвольного типа данных
- Bitwise trie – это бинарное дерево; алфавит $A = \{0, 1\}$



Radix tree (patricia trie)

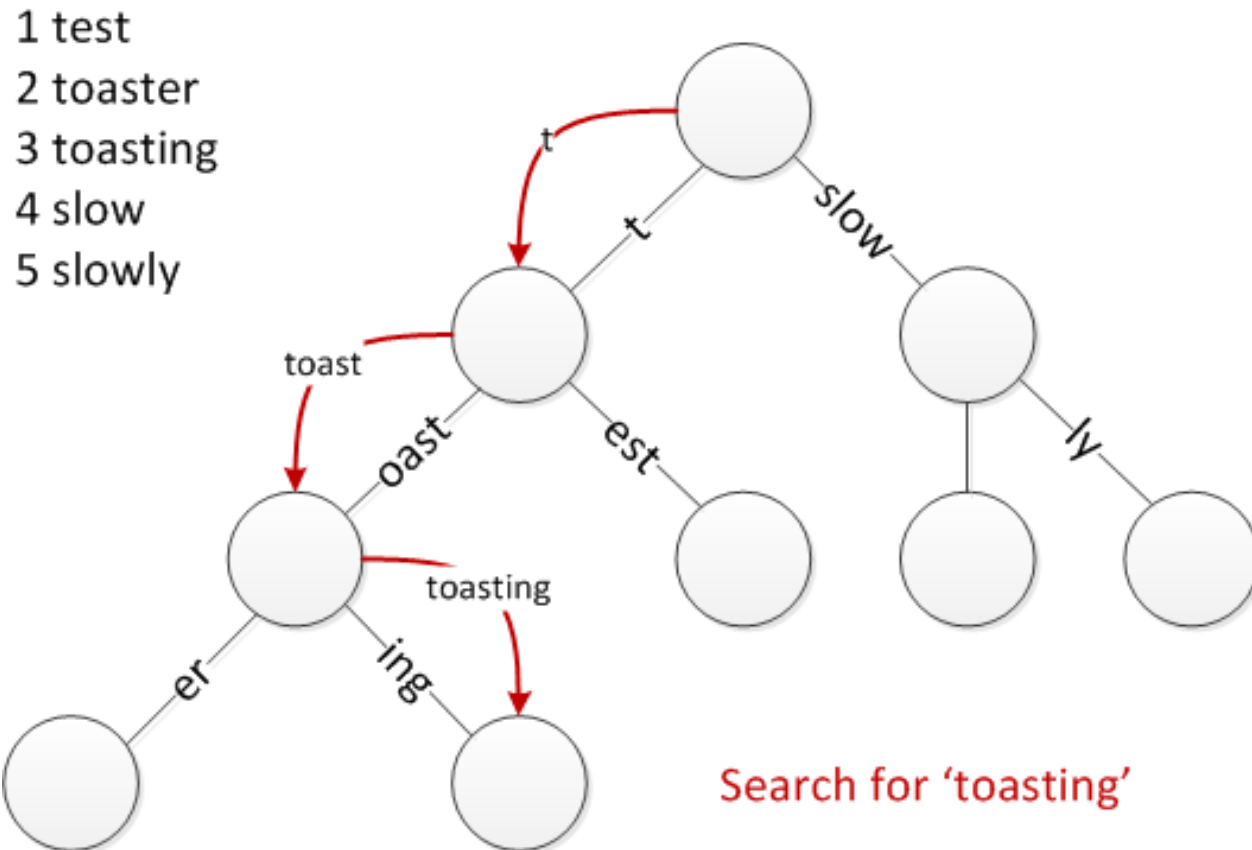
- **Radix tree (radix trie, patricia trie, compact prefix tree)** – префиксное дерево (trie), в котором узел содержащий один дочерний элемент объединяется с ним для экономии памяти



PATRICIA trie:

- ❑ D. R. Morrison. **PATRICIA – Practical Algorithm to Retrieve Information Coded in Alphanumeric**. Jnl. 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)



Suffix tree

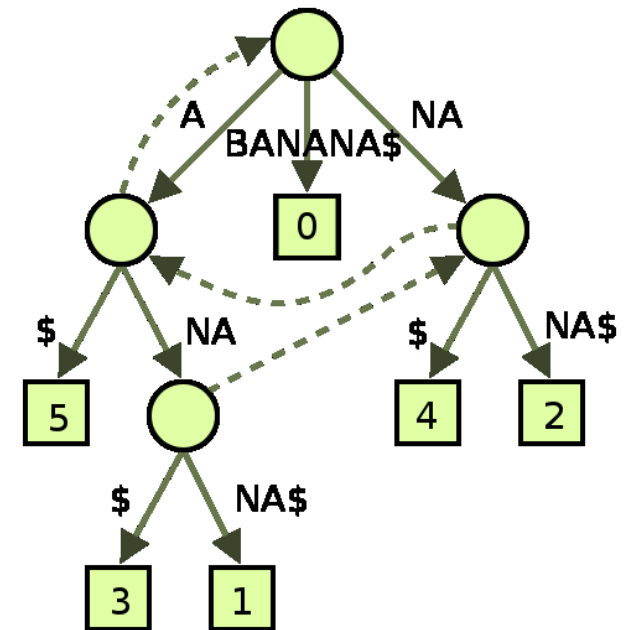
- **Suffix tree (PAT tree, position tree)** – префиксное дерево (trie), содержащее все суффиксы заданного текста (ключи) и их начальные позиции в тексте (values)

- **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.