Урок 9 > Подсистемы для поиска

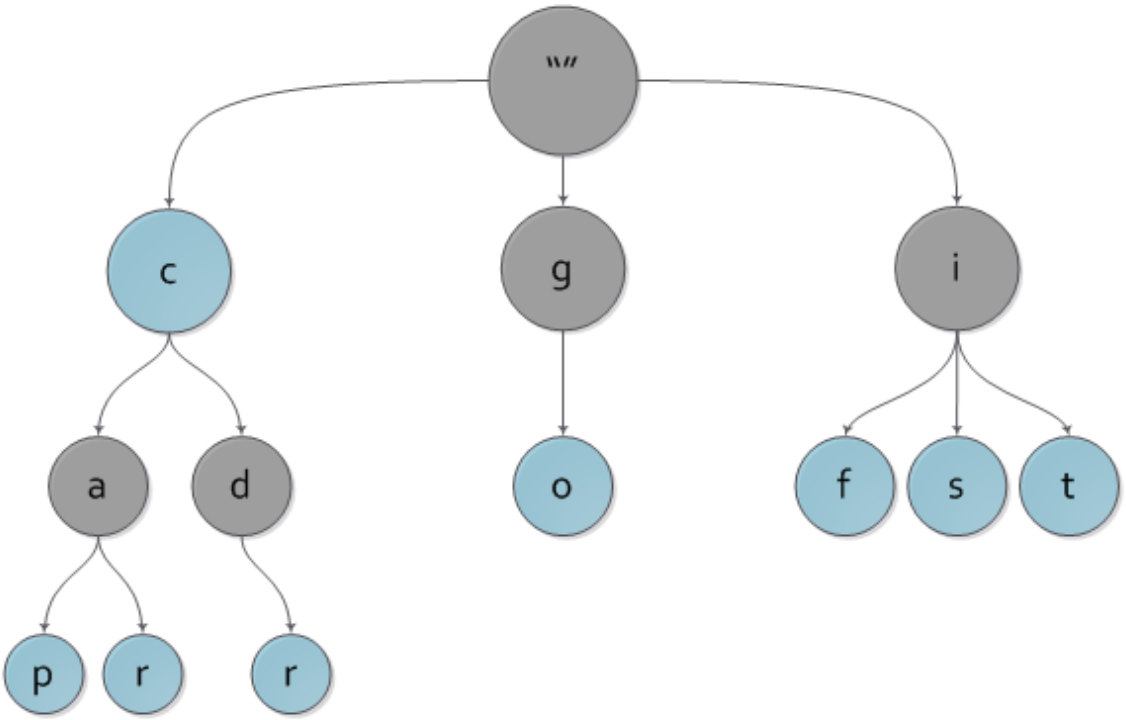
> Оглавление

> Префиксное дерево

Префиксное дерево (**бор** или **trie**) — структура данных для компактного хранения строк, устроенная в виде дерева, где на рёбрах между вершинами записаны символы, а некоторые вершины помечены терминальными. Говорят, что префиксное дерево принимает строку s, если существует такая терминальная вершина v, что, если выписать подряд все буквы на путях от корня до v, то получится строка s.

Нагруженное дерево отличается от обычных n-арных деревьев тем, что в его узлах не хранятся ключи. Вместо них в узлах хранятся односимвольные метки, а ключем, который соответствует некоему узлу является путь от корня дерева до

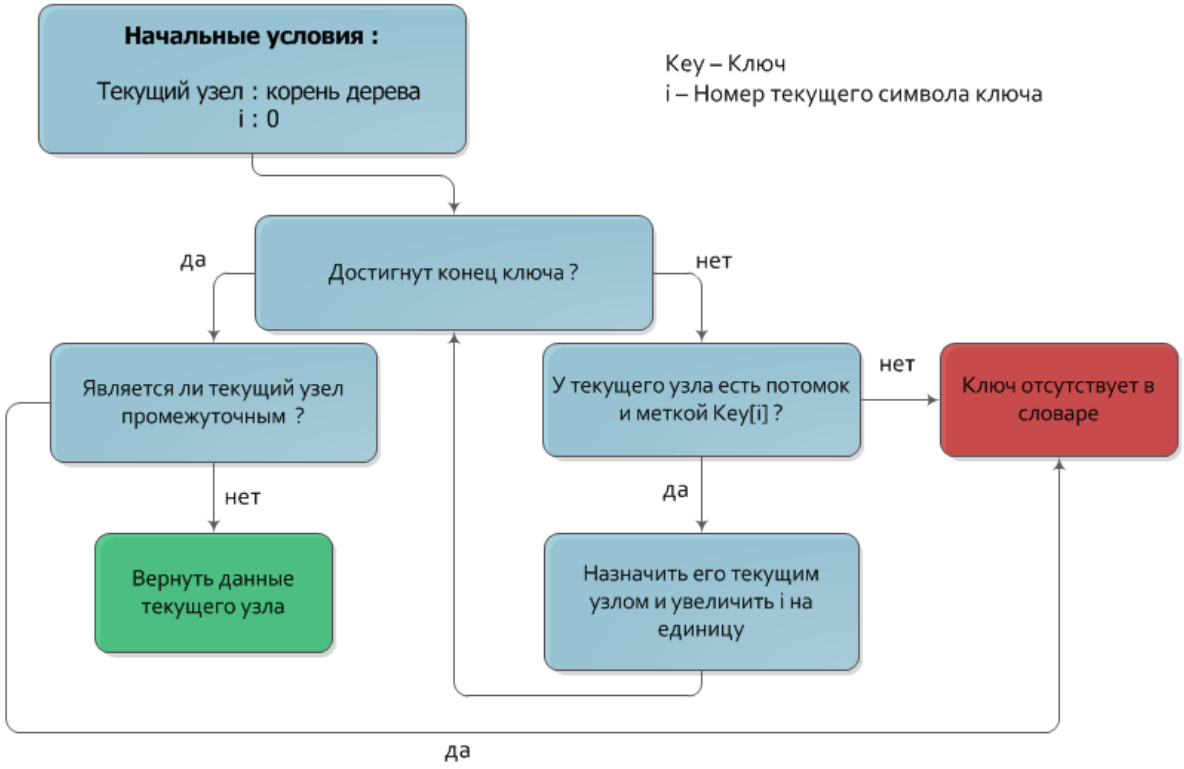
этого узла, а точнее строка составленная из меток узлов, повстречавшихся на этом пути. В таком случае корень дерева, очевидно, соответствует пустому ключу.



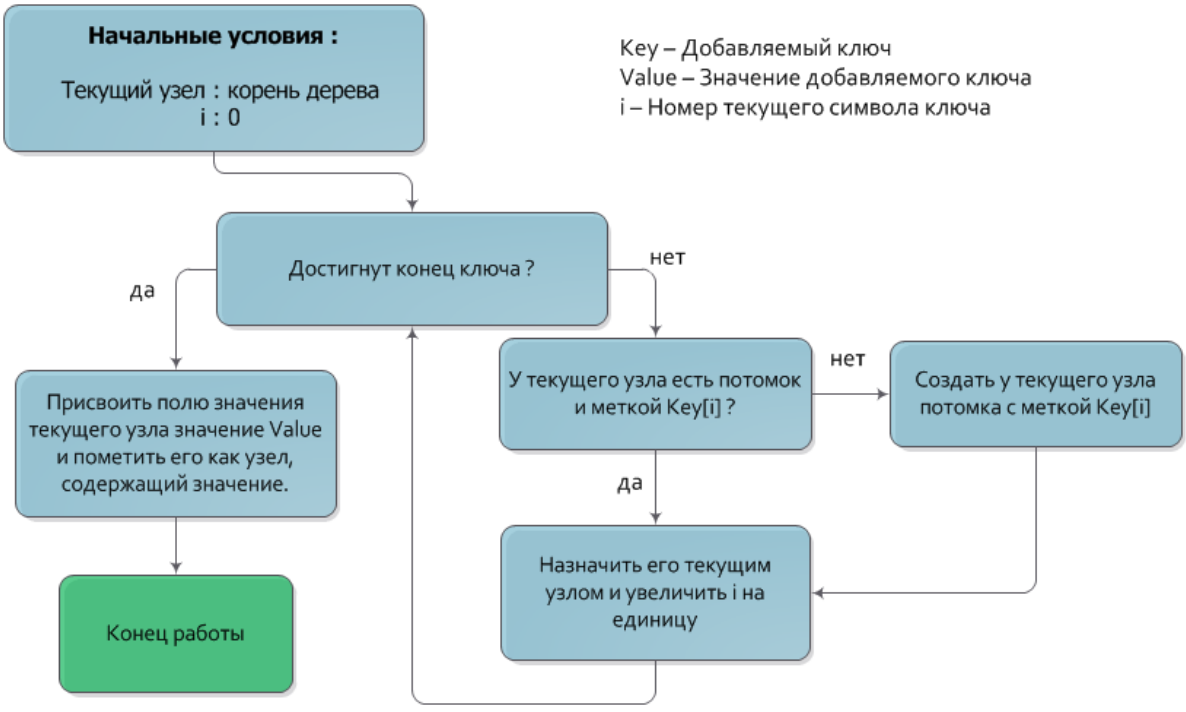
На рисунке вы можете наблюдать пример нагруженного дерева с ключами c, cap, car, cdr, go, if, is, it

Основные операции над префиксным деревом

1. Поиск по ключу. Временная сложность этого алгоритма равна $\mathrm{O}(|Key|)$



2. Вставка. Временная сложность добавления ключа — $O(|Key|)$



3. Удаление. Временная оценка алгоритма удаления — $O(|Key|)$

Оптимизация

Существует два основных типа оптимизации префиксного дерева:

- **Сжатие**. Сжатое нагруженное дерево получается из обычного удалением промежуточных узлов, которые ведут к единственному не промежуточному узлу. Например, цепочка промежуточных узлов с метками a, b, c заменяется на один узел с меткой abc.
- **Patricia**. Patricia нагруженное дерево получается из сжатого (или обычного) удалением промежуточных узлов, которые имеют одного ребенка.

> Префикс-функция

Префикс-функцией от строки S называется массив p, где $p(i)$ равно длине самого большого префикса строки $s(0)s(1)s(2)\dots s(i)$, который также является и суффиксом i-того префика (не считая весь i-й префикс).
Для строки '$abcdabcabcdabcdab$' вычисление будет таким:

```
1  S[1]='a', k=π=0;
2  S[2]='b'!=S[k+1] => k=π=0;
3  S[3]='c'!=S[1] => k=π=0;
4  S[4]='d'!=S[1] => k=π=0;
5  S[5]='a'==S[1] => k=π=1;
6  S[6]='b'==S[2] => k=π=2;
7  S[7]='c'==S[3] => k=π=3;
8  S[8]='a'!=S[4] => k:=π(S, 3)=0, S[8]==S[1] => k=π=1;
9  S[9]='b'==S[2] => k=π=2;
10 S[10]='c'==S[3] => k=π=3;
11 S[11]='d'==S[4] => k=π=4;
12 S[12]='a'==S[5] => k=π=5;
13 S[13]='b'==S[6] => k=π=6;
14 S[14]='c'==S[7] => k=π=7;
15 S[15]='d'!=S[8] => k:=π(S, 7)=3, S[15]==S[4] => k=π=4;
16 S[16]='a'==S[5] => k=π=5;
17 S[17]='b'==S[6] => k=π=6;
```

И результат таков: `[0,0,0,0,1,2,3,1,2,3,4,5,6,7,4,5,6]`.

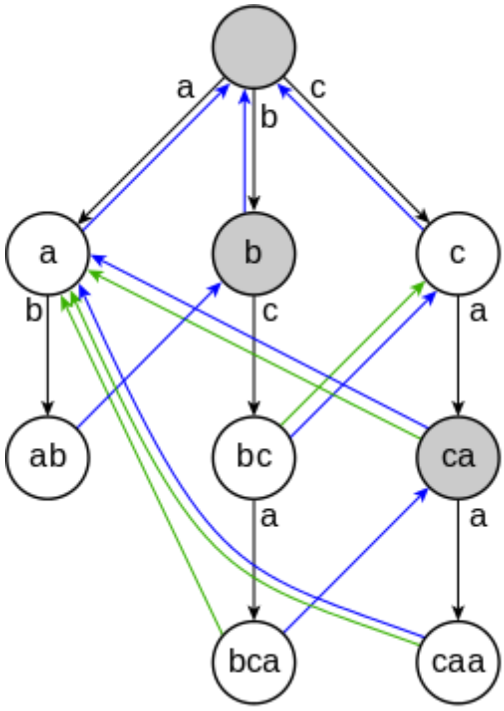
Реализация алгоритма на python:

```
def prefix(s):
    p = [0] * len(s)
    for i in range(1, len(s)):
        k = p[i - 1]
        while k > 0 and s[k] != s[i]:
            k = p[k - 1]
        if s[k] == s[i]:
            k += 1
        p[i] = k
    return p
```

> Алгоритм Ахо-Корасик

Алгоритм Ахо-Корасик — алгоритм поиска подстроки, разработанный Альфредом Ахо и Маргарет Корасик в 1975 году, реализует поиск множества подстрок из словаря в данной строке.

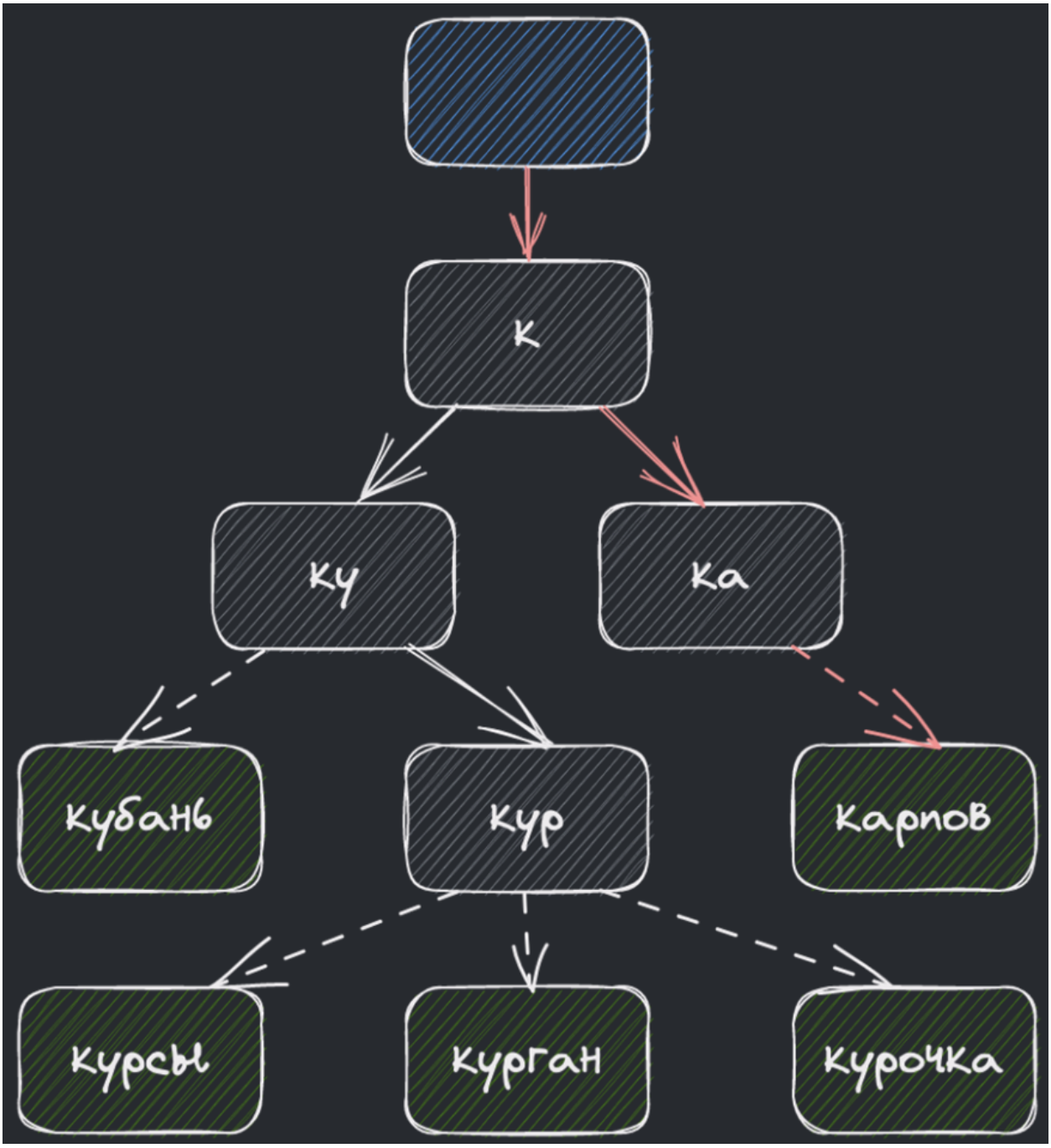
Алгоритм строит конечный автомат, которому затем передаёт строку поиска. Автомат получает по очереди все символы строки и переходит по соответствующим рёбрам. Если автомат пришёл в конечное состояние, соответствующая строка словаря присутствует в строке поиска.



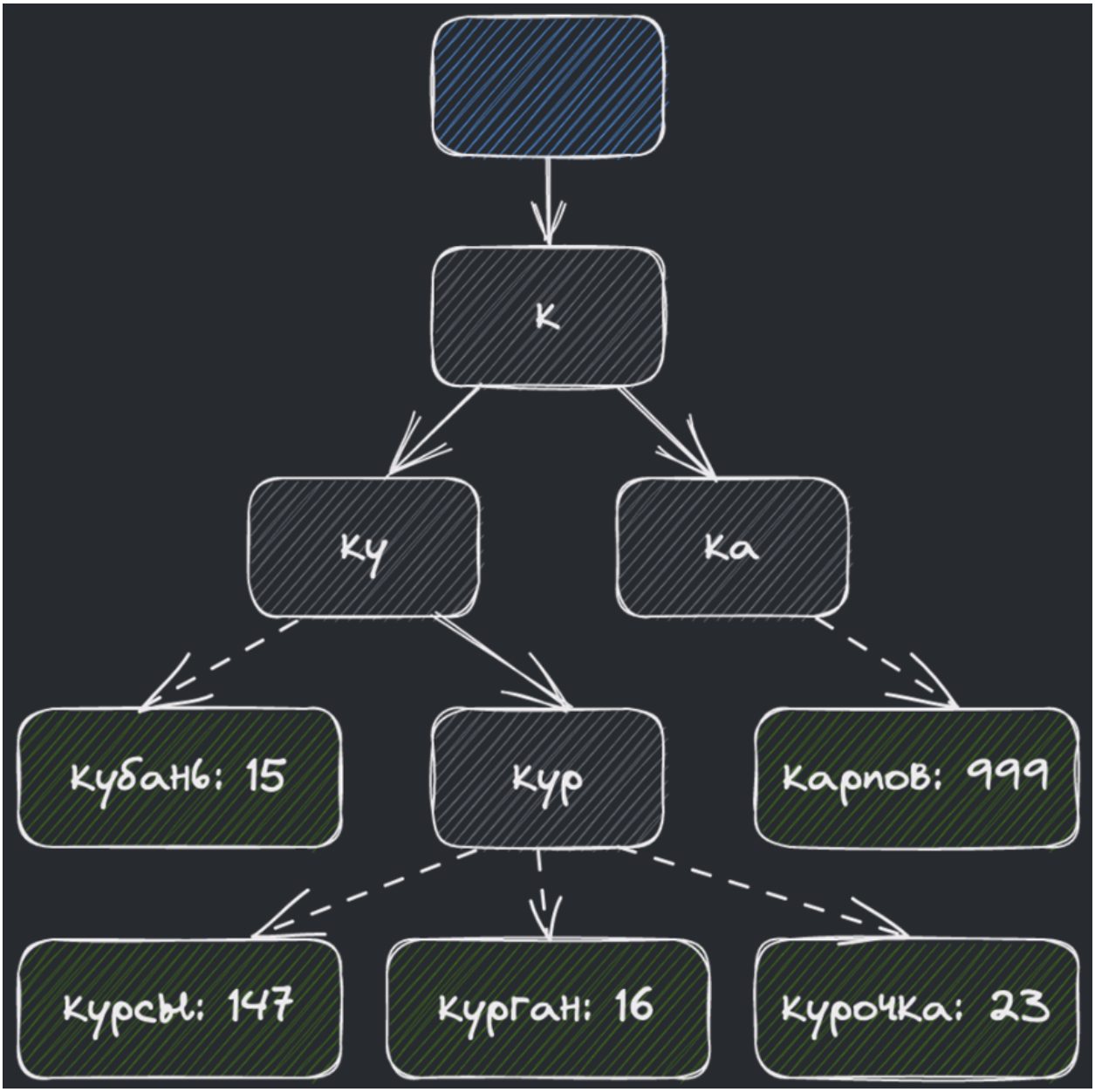
Недетерминированный автомат для словаря {a, ab, bc, bca, c, caa}. Серые вершины промежуточные, белые конечные. Синие стрелки - суффиксные ссылки, зелёные - конечные.

> Автодополнение строк

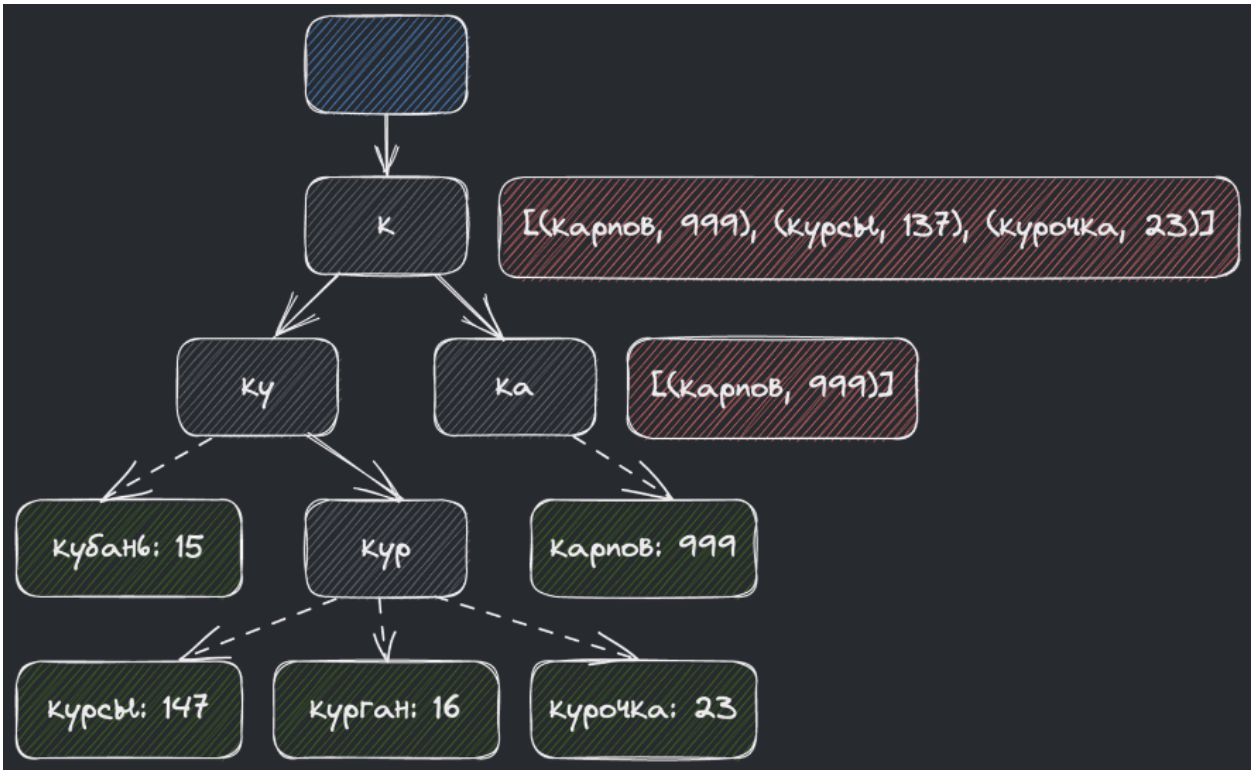
Для оптимального хранения статистики со строками и поиска рассмотрим структуру данных «префиксное дерево». Каждому слову из истории вызовов соответствует путь от корня к листьевой вершине дерева.



Статистику по запросам можно хранить в самих вершинах, передвигая счетчик каждого слова, по соответствующему запросу.



Для большей оптимизации можно класть массив статистики запросов в каждой вершине. Это даст возможность получить весь список за константное время.



> Текстовый поиск

Для поиска подстроки в строке отлично подходит префикс-функция.

Для начала соединим нашу подстроку, которую нужно найти в строке, со строкой через разделяющий спецсимвол, который не входит в алфавит. Например можно взять #.

Подстрока найдена, если в значениях функции есть длина подстроки.

> Поиск вхождений по словарю

Суть задачи:

Необходимо в заданном тексте найти все вхождения любого слова из заранее подготовленного словаря.

Алгоритм решения:

Для оптимального решения этой задачи подойдет алгоритм Ахо-Корасик.

По словарю строится конечный автомат, а затем через этот автомат прогоняется заданный текст.

> Неточный поиск (wildcard matching)

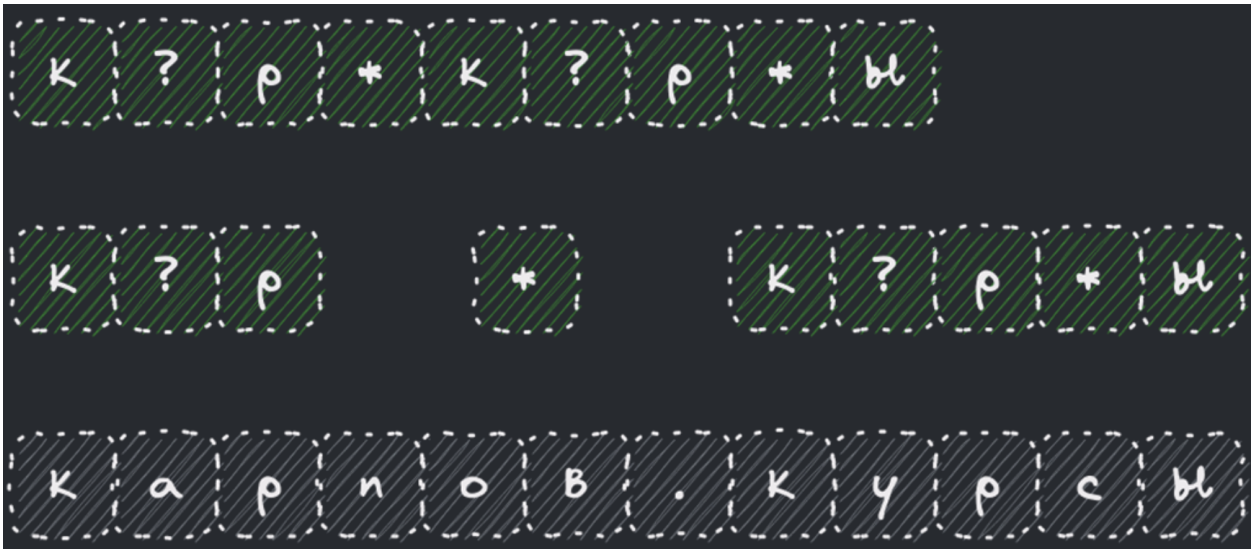
Суть задачи:

Допустим, мы хотим найти шаблон p в строке s, где в шаблоне могут быть символы $?$ и $*$.

Символ $?$ соответствует одной любой букве, тогда как $*$ соответствует любому количеству букв.

Можно считать, что строка s полностью определяется шаблоном p (иначе берем шаблон $*p*$).

Вид такого поиска часто встречается в текстовых редакторах или в сервисах бронирования билетов.



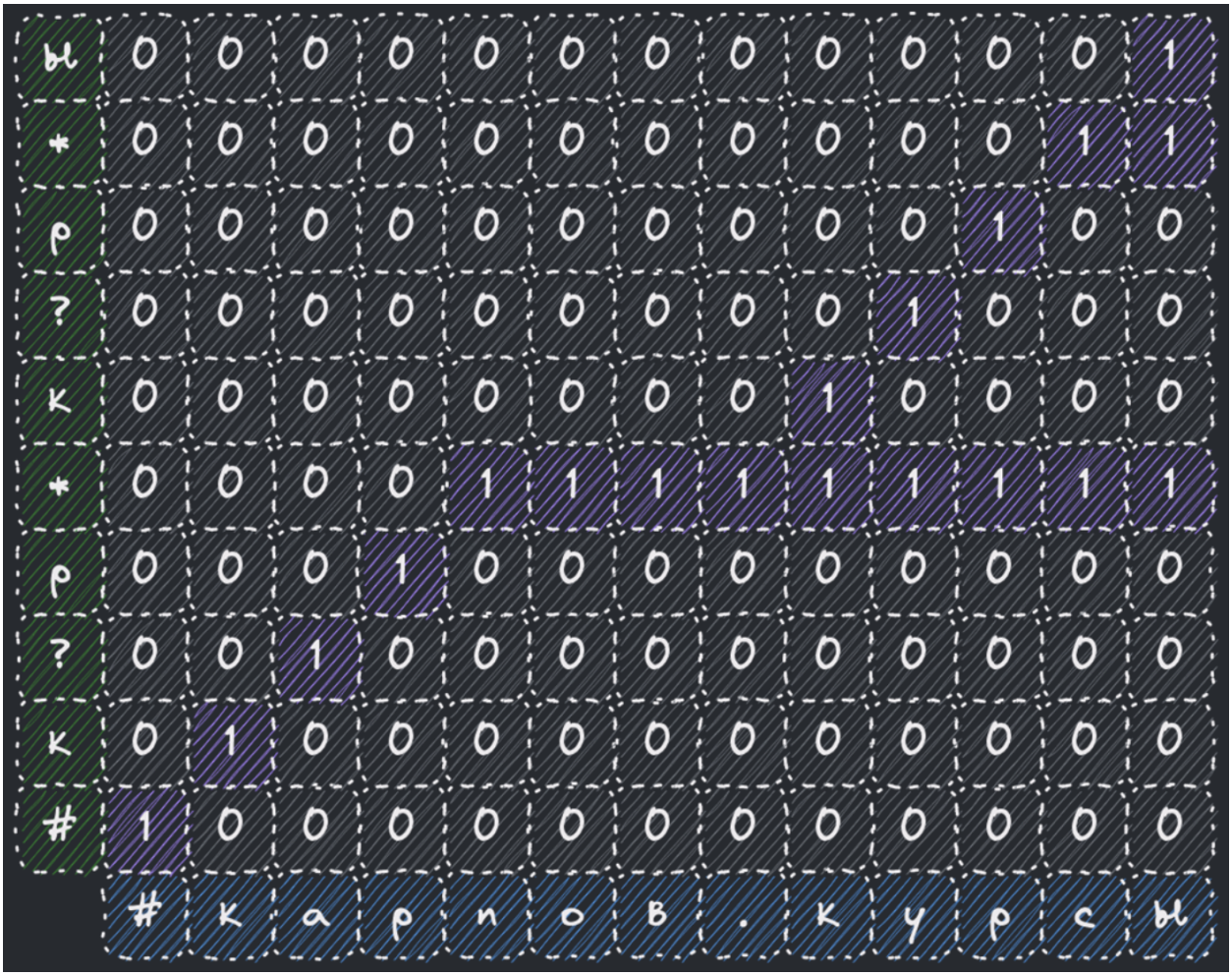
Для оптимального решения задачи воспользуемся динамическим программированием. Построим матрицу

D размера $S \times P$, где $D[m][n]$ – матч между $S[1..m]$ и $P[1..n]$.

Значение $D[m][n]$ легко рассчитать на основе $D[m-1][n-1]$:

— $D[m][n] = D[m-1][n-1]$, если $S[m]$ совпадает с $P[n]$ или $P[n] =$ “?”.

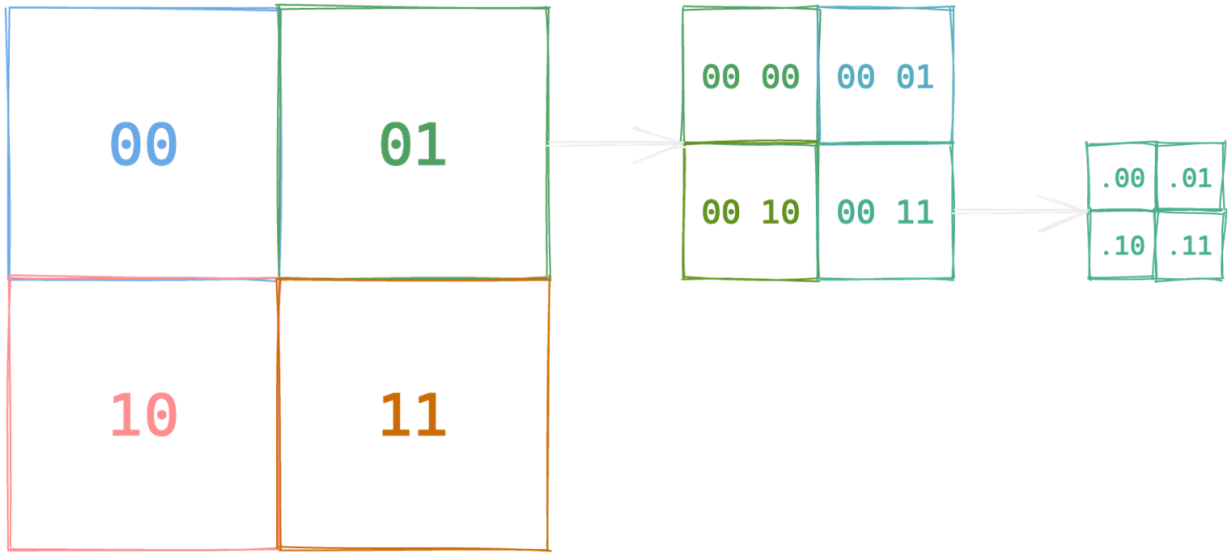
— Если $P[n] =$ “$*$”, то опять же $D[m][n]$ будет матчем в след за $D[m-1][n-1]$, при этом и все $D[m+i][n]$ тоже.



> Поиск по геолокации

GeoHash

Для поиска точки на карте, или набора каких-то точек, оптимально разбить карту на отдельные квадраты, каждому из которых задавать адрес. При этом большие квадраты разбиваются на более мелкие с постепенным увеличением адреса последних.



Такой подход позволит записать в 12 бит квадрат со стороной в 1 км, а в 20 бит - около метра

Итоговый адрес ячейки записывается в виде строки. Чтобы найти точки поблизости, достаточно изучить ячейки с общим префиксом.

Особенности использования GeoHash для поиска ближайших локаций:

- Если префикс у двух точек совпадает до какой-то степени, это гарантирует их близость.
- Обратное не верно – достаточно близкие точки могут не иметь общего префикса вообще.
- Найти ближайшие места можно простым `SELECT * FROM places WHERE geohash LIKE 'prefix%'`.
- Также два заведения совсем рядом могут иметь разный хеш, поэтому стоит включать соседние клетки.
- При необходимости расширить зону поиска достаточно немного уменьшить префикс.
- Можно создавать локальный геохэш для областей присутствия

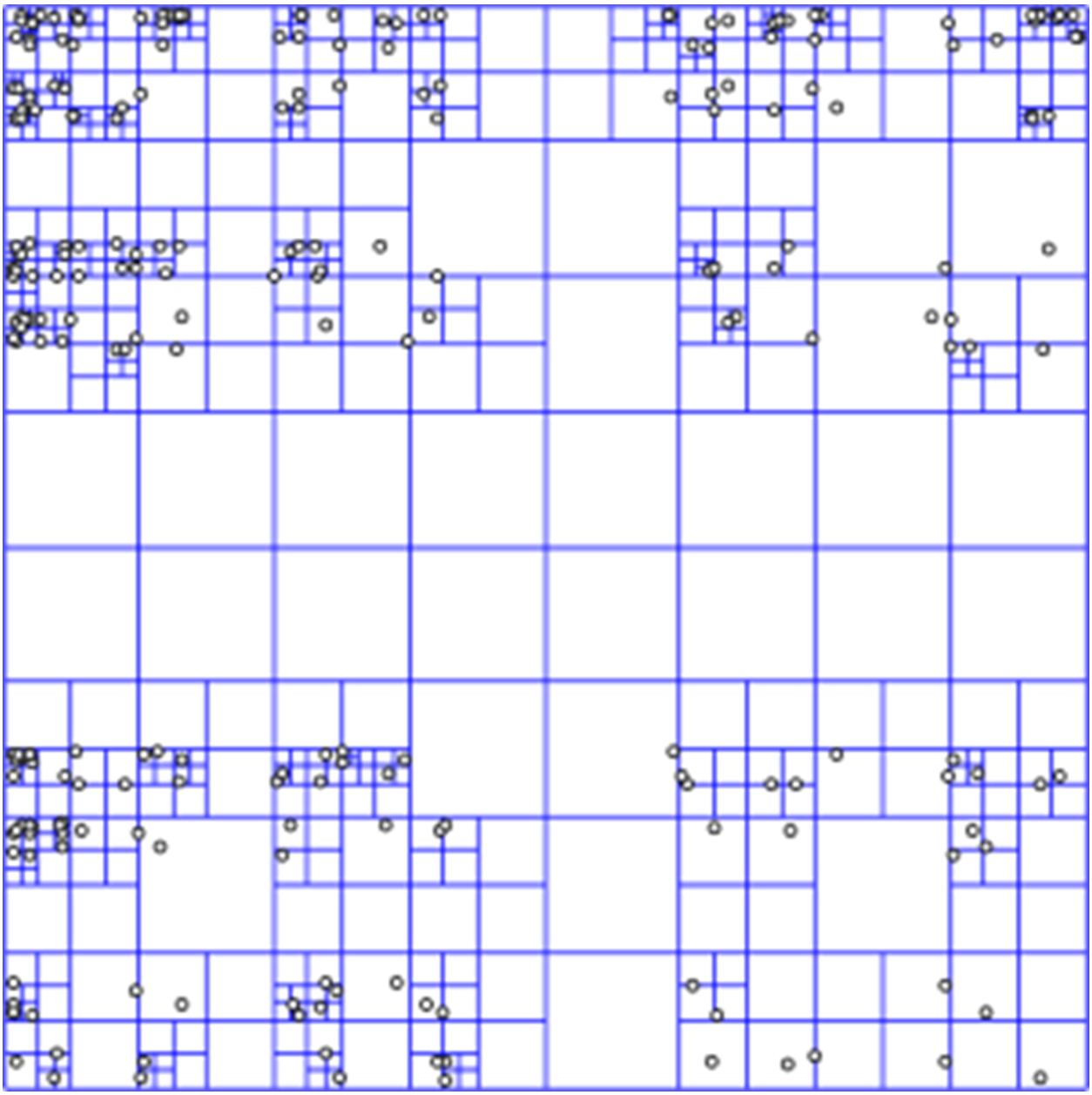
QuadTree

Дерево квадрантов (также **квадродерево**, **4-дерево, quadtree**) — дерево, в котором у каждого внутреннего узла ровно 4 потомка. Деревья квадрантов часто используются для рекурсивного разбиения двухмерного пространства по 4 квадранта (области). Области представляют собой квадраты, прямоугольники или имеют произвольную форму. Англоязычный термин **quadtree** был придуман Рафаэлем Финкелем и Джоном Бентли в 1974 году. Аналогичное разбиение пространства известно как Q-дерево.

Общие черты разных видов деревьев квадрантов:

- разбиение пространства на адаптирующиеся ячейки (adaptable cells)
- максимально возможный объём каждой ячейки
- соответствие направления дерева пространственному разбиению

Предлагаемый подход также разбивает карту на квадраты, но делает это более оптимально — динамически увеличивает глубину разбиения в зависимости от количества точек



Особенности использования QuadTree для поиска ближайших локаций:

- Построение дерева линейно по количеству точек интереса
- Соответствие между искомой позицией и листом в дереве находится за считанные шаги
- Для поиска ближайших точек интереса либо останавливаемся в листе, либо поднимаемся выше

- Два заведения совсем рядом могут попасть в разные ячейки, поэтому можно взять соседние
- При необходимости расширить зону поиска достаточно подняться на один уровень выше
- Расширение зоны поиска происходит в неравной степени по сторонам света и по диагонали
- Все решение легко умещается в оперативную память и поиск проходи стремительно