**Префиксные деревья**

1. **Введение понятия «*префиксное дерево*»**

(2 слайд)

***Префиксное дерево*** (также бор (вы*бор*ка по Кнуту), луч (по*луч*ение по Кнуту), нагруженное дерево, trie (re*trie*val – поиск, извлечение)) — структура данных, позволяющая хранить ассоциативный массив, ключами которого являются строки. Сразу следует оговорится, что в большинстве случаев ключами выступают строки, однако в качестве ключей можно использовать любые типы данных, представимые как последовательность байт (то есть любые).

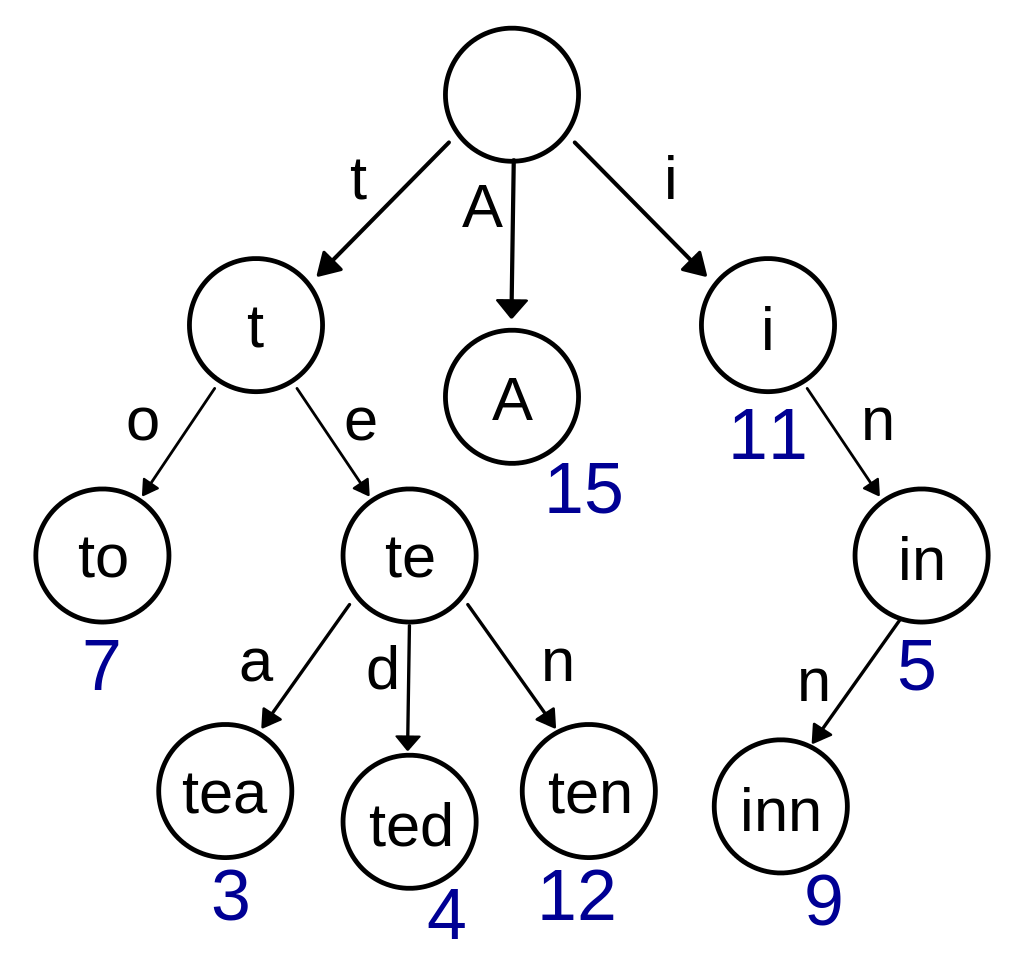
***Ассоциативный массив*** — абстрактный тип данных, позволяющий хранить пары вида «(ключ, значение)» и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу.

Попытки были впервые описаны Рене де ла Брианье в 1959 году. Термин «три» был придуман двумя годами позже Эдвардом Фредкиным, который произносит его / ˈtriː / (как «дерево»). Однако другие авторы произносят его / ittraɪ / (как «попробуй»), пытаясь в устной форме отличить его от «дерева».

(3 слайд)

Представляет собой корневое дерево, каждое ребро которого помечено каким-то символом. Некоторые узлы префиксного дерева ***выделены*** (на рисунке они подписаны числами) и считается, что префиксное дерево содержит данную строку-ключ тогда и только тогда, когда эту строку можно прочитать на пути из корня до некоторого (единственного для этой строки) выделенного узла.

Префиксное дерево отличается от обычных N-арных деревьев тем, что в его узлах не хранятся ключи. Вместо них в узлах хранятся односимвольные метки, а ключом, который соответствует некоторому 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.



В боре, изображённом выше, хранятся следующие данные:

* “to” - 7 экземпляров
* “tea” - 3 экземпляра
* “ted” - 4 экземпляра
* “ten” - 12 экземпляров
* “A” - 15 экземпляров
* “i” - 11 экземпляров
* “in” - 5 экземпляров
* “inn” - 9 экземпляров

Сразу видно, что данное дерево содержит «лишние» ключи, ведь любому узлу дерева соответствует единственный путь до него от корня, а значит и некоторый ключ. Чтобы избежать проблемы с «лишними» ключами, каждому узлу дерева добавляется булева характеристика, указывающая, является ли узел действительно существующим либо промежуточным по дороге в какой-либо другой.

(4 слайд)

Таким образом, в отличие от бинарных деревьев поиска, ключ, идентифицирующий конкретный узел дерева, не явно хранится в данном узле, а задаётся положением данного узла в дереве. Получить ключ можно выписыванием подряд символов, помечающих рёбра на пути от корня до узла. Часто в выделенных узлах хранят дополнительную информацию, связанную с ключом, и обычно выделенными являются только листья и, возможно, некоторые внутренние узлы (на рисунке – помеченные числами (весами)).

Преимущества:

* Время поиска в словаре не зависит от количества элементов (зависит от длины ключа)
* Для хранения ключей не используется дополнительной памяти
* В отличии от хеш-таблиц не возникает коллизий (равенство значений хеш-функции на двух различных блоках данных)

(5 слайд)

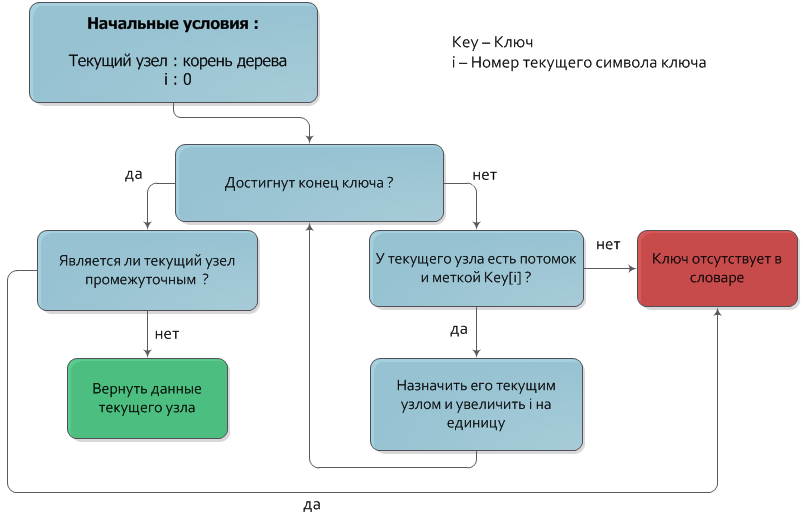
1. **Основные операции**

А) *Поиск выделенного ключа*

Ключ, соответствующий узлу — конкатенация меток узлов, содержащихся в пути от корня к данному узлу. Из этого свойства естественным образом следует алгоритм поиска ключа.  
 Пусть дан ключ Key, который необходимо найти в дереве. Будем спускаться из корня дерева на нижние уровни, каждый раз переходя в узел, чья метка совпадает с очередным символом ключа. После того как обработаны все символы ключа, узел, в котором остановился спуск и будет искомым узлом. Если в процессе спуска не нашлось узла с меткой, соответствующей очередному символу ключа, или спуск остановился на промежуточной вершине (вершине, не имеющей значения), то искомый ключ отсутствует в дереве.

Временная сложность этого алгоритма равна О(|Key|).

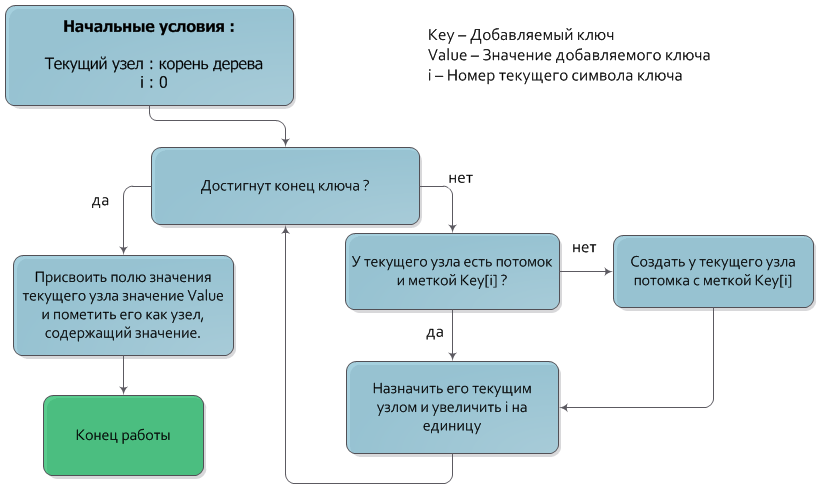
Алгоритм:



(8-15 слайды)

Б) *Вставка выделенного ключа*

Пусть дана пара из ключа Key и значения Value, которую нужно добавить. Будем спускаться из корня дерева на нижние уровни, каждый раз переходя в узел, чья метка совпадает с очередным символом ключа. После того как обработаны все символы ключа, узел, в котором остановился спуск и будет узлом, которому должно быть присвоено значение Value (узел должен быть помечен как имеющий значение). Если в процессе спуска отсутствует узел с меткой, соответствующей очередному символу ключа, то следует создать новый промежуточный узел с нужной меткой и назначить его потомком текущего узла.  
 Временная сложность этого алгоритма равна О(|Key|). Алгоритм:



(16 слайд)

В) *Удаление выделенного ключа*

Пусть дан ключ Key, который необходимо удалить из дерева. Если ключ существует в словаре, то зная узел, которому он соответствует, можно просто пометить его как промежуточный, сделав его «невидимым» для последующих поисков.  
 После этого можно подняться от «отключенного» узла к корню, попутно удаляя все узлы, которые являются листьями, однако экономия памяти в данном случае не существенна, а для эффективного определения того, является ли узел листом потребуется вводить дополнительную характеристику узла.

Временная сложность этого алгоритма равна О(|Key|).

(19 слайд)

1. **Характеристики**

Нагруженное дерево по показателям потребления памяти/процессорного времени не уступает хэш-таблицам и сбалансированным деревьям, а иногда и превосходит их по этим параметрам.

Сложность операций вставки, удаления и поиска — O(|Key|). Хотя сбалансированные деревья и выполняют эти операции за O(ln(n)), но здесь скрыто время, необходимое для сравнения ключей, составляющее O(|Key|). С хэш-таблицами ситуация аналогична — хоть время доступа и составляет О(1+a), но взятие хэша (если он не вычислен заранее) занимает O(|Key|).

(20 слайд)

По потреблению памяти префиксное дерево часто выигрывает у хэш-таблиц и сбалансированных деревьев. Это связано с тем, что у множества ключей в нагруженном дереве совпадают префиксы, и вместе с ними память, которую они используют. Также, в отличии от сбалансированных деревьев, в нагруженном дереве нет необходимости хранить ключ в каждом узле.

(21-26 слайды)

1. **Результаты тестирования**

Практические исследования показали, что дерево Бор работает значительно лучше хеш-таблицы, дерева AVL и связного списка лишь на больших объемах данных. На маленьких выборках дерево Бор уступает вышеперечисленным ассоциативным структурам данных.

(27 слайд)

Существует 2 основных типа оптимизации префиксного дерева:

* *Сжатие* - сжатое префиксное 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* - такое нагруженное дерево получается из сжатого (или обычного) удалением промежуточных узлов, которые имеют одного ребенка.

**5. Применение**

(28 слайд)

Область применения нагруженных деревьев огромна — их можно применять везде где требуется реализация интерфейса ассоциативного массива. Особенно нагруженные деревья удобны для реализации словарей, спеллчекеров и прочих Т9 — то есть в задачах, где необходимо быстро получать наборы ключей с заданным префиксом. Также нагруженное дерево использует в своей работе не безызвестный алгоритм Ахо — Корасик.

Алгоритм Ахо-Корасик реализует эффективный поиск всех вхождений всех строк-образцов в заданную строку. Был разработан в 1975 году Альфредом Ахо и Маргарет Корасик.  Решает следующую задачу: на вход поступают несколько строк pattern[i] и строка s. Необходимо найти все возможные вхождения строк pattern[i] в s. Суть алгоритма заключена в использование структуры данных — бора и построения по нему конечного детерминированного автомата.

Также может быть использовано в:

* Предиктивный ввод текста – поиск возможных завершений слов
* Автозавершение в текстовых редакторах и IDE
* Проверка правописания
* Автоматическая расстановка переносов слов