Trie, или нагруженное дерево

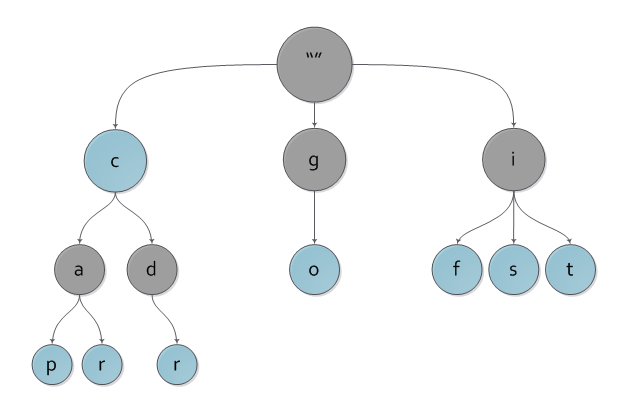
Рассмотрим структуру данных известную как **словарь на нагруженном дереве**, или **префиксное дерево**, или **trie**.

Что это?

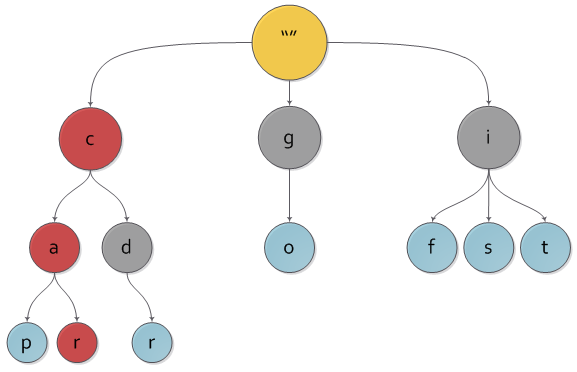
Нагруженное дерево — структура данных реализующая интерфейс ассоциативного массива, то есть позволяющая хранить пары «ключ-значение». Сразу следует оговорится, что в большинстве случаев ключами выступают строки, однако в качестве ключей можно использовать любые типы данных, представимые как последовательность байт (то есть вообще любые).

Как это работает?

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*.



И то же самое дерево с выделенным на нем ключем *car*.



Сразу видно, что наше 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 реально существующим либо промежуточным по дороге в какой-либо другой.

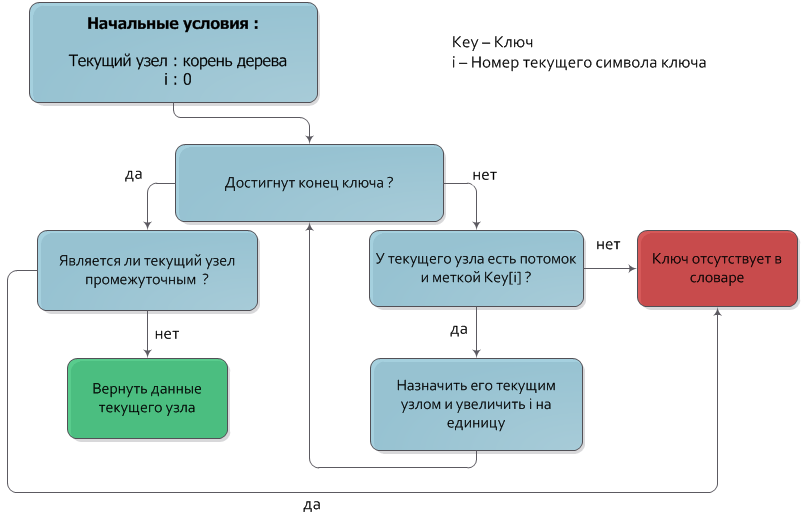
**Основные операции**

Так как нагруженное дерево реализует интерфейс ассоциативного массива, в нем можно выделить три основные операции, а именно вставку, удаление и поиск ключа. Как и многие деревья, нагруженное дерево обладает свойством самоподобия, то есть любое его поддерево также является полноценным нагруженным деревом. Легко заметить, что все ключи в таком поддереве имеют общий префикс, (откуда и пошло название «префиксное дерево») а значит можно выделить специфичную для этого дерева операцию — получение всех ключей дерева с заданным префиксом за время O(|Prefix|).

**Поиск ключа**

Как уже было сказано, ключ, соответствующий узлу — конкатенация меток узлов, содержащихся в пути от корня к данному узлу. Из этого свойства естественным образом следует алгоритм поиска ключа (как, впрочем, и алгоритмы добавления и удаления). 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, очевидно, равна О(|Key|).

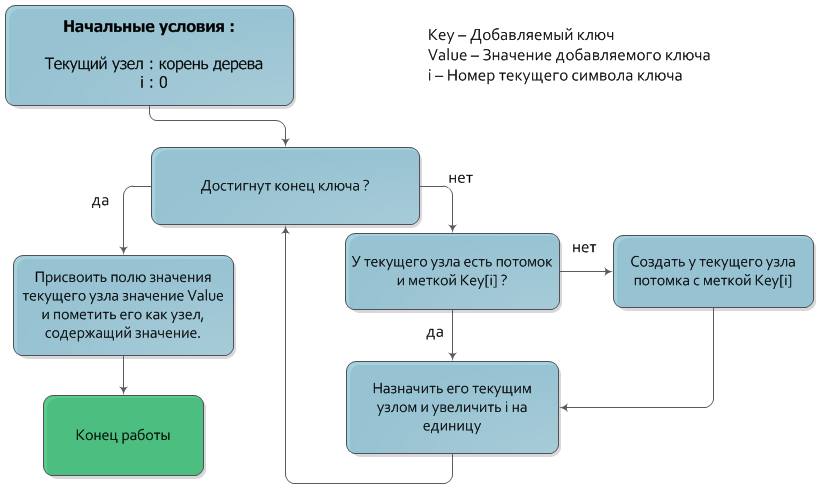
Более подробно алгоритм показан на блок-схеме:



**Вставка**

Алгоритм добавления ключа в дерево очень похож на алгоритм поиска.   
  
Пусть дана пара из ключа Key и значения Value, которую нужно добавить. Как и в алгоритме поиска ключа, 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 (также, разумеется, 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.  
  
Временная сложность добавления ключа — О(|Key|).

Иллюстрация алгоритма на блок-схеме:



**Удаление**

Удаление ключа также реализуется очень легко.  
  
Пусть дан ключ Key, который необходимо удалить из дерева. Проведем поиск этого ключа. 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 оценка алгоритма удаления — знакомое уже О(|Key|).

**Требовательность к ресурсам**

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

Процессорное время

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

Память

По потреблению памяти нагруженное 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.

**Оптимизации**

Существует 2 основных типа оптимизации нагруженного 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.

Зачем все это нужно?

Собственно, область применения нагруженных деревьев огромна — их можно применять везде где требуется реализация интерфейса ассоциативного массива. Особенно нагруженные деревья удобны для реализации словарей, спел-чекеров и прочих Т9 — то есть в задачах, где необходимо быстро получать наборы ключей с заданным префиксом. Также нагруженное дерево использует в своей работе небезызвестный алгоритм [Ахо — Корасик](http://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%90%D1%85%D0%BE_%E2%80%94_%D0%9A%D0%BE%D1%80%D0%B0%D1%81%D0%B8%D0%BA).

/\* ==========  ========== ========== ========= \*/

//          Trie Tree Data Structure           //

//                using C++ STL                //

//                                             //

//         Functions follow Pascal Case        //

//           Convention and Variables          //

//         follow Camel Case Convention        //

//                                             //

//            Author - Vamsi Sangam            //

//            Theory of Programming            //

/\* ========== ========== ========== ========== \*/

#include <cstdio>

#include <cstdlib>

#include <vector>

#define ALPHABETS 26

#define CASE 'a'

#define MAX\_WORD\_SIZE 25

using namespace std;

struct Node

{

    struct Node \* parent;

    struct Node \* children[ALPHABETS];

    vector<int> occurrences;

};

// Inserts a word 'text' into the Trie Tree

// 'trieTree' and marks it's occurence as 'index'.

void InsertWord(struct Node \* trieTree, char \* word, int index)

{

    struct Node \* traverse = trieTree;

    while (\*word != '\0') {     // Until there is something to process

        if (traverse->children[\*word - CASE] == NULL) {

            // There is no node in 'trieTree' corresponding to this alphabet

            // Allocate using calloc(), so that components are initialised

            traverse->children[\*word - CASE] = (struct Node \*) calloc(1, sizeof(struct Node));

            traverse->children[\*word - CASE]->parent = traverse;  // Assigning parent

        }

        traverse = traverse->children[\*word - CASE];

        ++word; // The next alphabet

    }

    traverse->occurrences.push\_back(index);      // Mark the occurence of the word

}

// Searches for the occurence of a word in 'trieTree',

// if not found, returns NULL,

// if found, returns poniter pointing to the

// last node of the word in the 'trieTree'

// Complexity -> O(length\_of\_word\_to\_be\_searched)

struct Node \* SearchWord(struct Node \* treeNode, char \* word)

{

    // Function is very similar to insert() function

    while (\*word != '\0') {

        if (treeNode->children[\*word - CASE] != NULL) {

            treeNode = treeNode->children[\*word - CASE];

            ++word;

        } else {

            break;

        }

    }

    if (\*word == '\0' && treeNode->occurrences.size() != 0) {

        // Word found

        return treeNode;

    } else {

        // Word not found

        return NULL;

    }

}

// Searches the word first, if not found, does nothing

// if found, deletes the nodes corresponding to the word

void RemoveWord(struct Node \* trieTree, char \* word)

{

    struct Node \* trieNode = SearchWord(trieTree, word);

    if (trieNode == NULL) {

        // Word not found

        return;

    }

    trieNode->occurrences.pop\_back();    // Deleting the occurence

    // 'noChild' indicates if the node is a leaf node

    bool noChild = true;

    int childCount = 0;

    // 'childCount' has the number of children the current node

    // has which actually tells us if the node is associated with

    // another word .This will happen if 'childCount' != 0.

    int i;

    // Checking children of current node

    for (i = 0; i < ALPHABETS; ++i) {

        if (trieNode->children[i] != NULL) {

            noChild = false;

            ++childCount;

        }

    }

    if (!noChild) {

        // The node has children, which means that the word whose

        // occurrence was just removed is a Suffix-Word

        // So, logically no more nodes have to be deleted

        return;

    }

    struct Node \* parentNode;     // variable to assist in traversal

    while (trieNode->occurrences.size() == 0 && trieNode->parent != NULL && childCount == 0) {

        // trieNode->occurrences.size() -> tells if the node is associated with another word

        //

        // trieNode->parent != NULL -> is the base case sort-of condition, we simply ran

        // out of nodes to be deleted, as we reached the root

        //

        // childCount -> does the same thing as explained in the beginning, to every node

        // we reach

        childCount = 0;

        parentNode = trieNode->parent;

        for (i = 0; i < ALPHABETS; ++i) {

            if (parentNode->children[i] != NULL) {

                if (trieNode == parentNode->children[i]) {

                    // the child node from which we reached

                    // the parent, this is to be deleted

                    parentNode->children[i] = NULL;

                    free(trieNode);

                    trieNode = parentNode;

                } else {

                    ++childCount;

                }

            }

        }

    }

}

// Prints the 'trieTree' in a Pre-Order or a DFS manner

// which automatically results in a Lexicographical Order

void LexicographicalPrint(struct Node \* trieTree, vector<char> word)

{

    int i;

    bool noChild = true;

    if (trieTree->occurrences.size() != 0) {

        // Condition trie\_tree->occurrences.size() != 0,

        // is a neccessary and sufficient condition to

        // tell if a node is associated with a word or not

        vector<char>::iterator charItr = word.begin();

        while (charItr != word.end()) {

            printf("%c", \*charItr);

            ++charItr;

        }

        printf(" -> @ index -> ");

        vector<int>::iterator counter = trieTree->occurrences.begin();

        // This is to print the occurences of the word

        while (counter != trieTree->occurrences.end()) {

            printf("%d, ", \*counter);

            ++counter;

        }

        printf("\n");

    }

    for (i = 0; i < ALPHABETS; ++i) {

        if (trieTree->children[i] != NULL) {

            noChild = false;

            word.push\_back(CASE + i);   // Select a child

            // and explore everything associated with the cild

            LexicographicalPrint(trieTree->children[i], word);

            word.pop\_back();

            // Remove the alphabet as we dealt

            // everything associated with it

        }

    }

    word.pop\_back();

}

int main()

{

    int n, i;

    vector<char> printUtil;       // Utility variable to print tree

    // Creating the Trie Tree using calloc

    // so that the components are initialised

    struct Node \* trieTree = (struct Node \*) calloc(1, sizeof(struct Node));

    char word[MAX\_WORD\_SIZE];

    printf("Enter the number of words-\n");

    scanf("%d", &n);

    for (i = 1; i <= n; ++i) {

        scanf("%s", word);

        InsertWord(trieTree, word, i);

    }

    printf("\n");   // Just to make the output more readable

    LexicographicalPrint(trieTree, printUtil);

    printf("\nEnter the Word to be removed - ");

    scanf("%s", word);

    RemoveWord(trieTree, word);

    printf("\n");   // Just to make the output more readable

    LexicographicalPrint(trieTree, printUtil);

    return 0;

}

<http://theoryofprogramming.com/2015/01/16/trie-tree-implementation/>

http://theoryofprogramming.com/trie-tree-using-cpp-class/