Московский авиационный институт
(национальный исследовательский университет)

Институт № 8 «Информационные технологии и прикладная математика»

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу «Дискретный анализ»

Студент: Л. В. Короткевич
Преподаватель: Н. С. Капралов
Группа: М8О-208Б-19
Дата:
Оценка:
Подпись:

Москва 2020

Лабораторная работа №2

Задача: Необходимо создать программную библиотеку, реализующую указанную структуру данных, на основе которой разработать программу-словарь. В словаре каждому ключу, представляющему из себя регистронезависимую последовательность букв английского алфавита длиной не более 256 символов, поставлен в соответствие некоторый номер, от 0 до 2^{64} - 1. Разным словам может быть поставлен в соответствие один и тот же номер.

Вариант структуры данных: PATRICIA

1. **Описание**

Патриция - специальная структура данных, необходимая для хранения ключей (чаще всего - слов). Узлы дерева содержат в себе пару ссылок на узлы (или на себя, или на «младшие», а то и «старшие» узлы), ключ, значение и номер бита, который мы будем сравнивать при проходе через этот узел.

Как не трудно догадаться, все операции основаны на сравнении определённых битов, так что временная сложность операций поиска, добавления и удаления элемента из дерева оценивается как O(k), где k — длина обрабатываемого элемента. Очевидно, что время работы не зависит от количества элементов в дереве.

Данная реализация имеет одну закономерность - узлы дерева отсортированы (по мере прохождения вглубь) по возрастанию по номеру бита, который они хранят. Это позволит нам определять, является ли ссылка указателем “наверх” без использования специальных меток.

Что отличает Патрицию от ее предшественников (Trie, Prefix Trie) — в ней не происходит одностороннего ветвления, что обусловлено наличием того самого индекса — номера бита, необходимого для сравнения ключей.

Структуру узла, конструкторы и деструкторы я описал в Node.hpp, а структуру дерева со всеми необходимыми операции для работы с ним — в Trie.hpp. Дополнительные функции для работы со строками, битами и т.д. вынесены в Additional.hpp.

Отдельно стоит сказать о сохранении/загрузке дерева. Сериализовывал я следующим образом: прежде чем записать информацию об узлах в поток, я нумерую их. Сделано это за тем, чтобы упростить себе жизнь: вместо рекурсивного вызова функции сериализации для дочерних узлов при сериализации какго-то узла, я буду писать в поток номера его детей (нумерация может быть любой; например, я их нумеровал последовательно при прямом обходе). Десериализация происходит аналогично сериализации, узлы считываются в том же порядке.

2. Исходный код

Node.hpp:

```
#pragma once

const int MAXLEN = 256;
typedef unsigned long long TValue;
typedef char TKey;

struct TNode
{
   int id = -1;
   int bit;

   TKey *key;
   TValue value;

   TNode *left;
   TNode *right;

   void Initialize(int b, TKey *k, TValue v, TNode *l, TNode *r)
   {
      bit = b;
      if (k)
      {
         key = new char[strlen(k) + 1];
         strcpy(key, k);
      }
      else
         key = k;
      value = v;
      left = l;
      right = r;
   }

   TNode()
   {
      Initialize(-1, 0, 0, this, this);
   }

   TNode(int b, TKey *k, TValue v)
   {
```

```
        Initialize(b, k, v, this, this);
    }

    TNode(int b, TKey *k, TValue v, TNode *l, TNode *r)
    {
        Initialize(b, k, v, l, r);
    }

    ~TNode()
    {
        delete [] key;
    }
};
```

Trie.hpp:

```
#pragma once

#include "Node.hpp"

struct TTrie
{
    TNode *root;
    int size;

    TTrie()
    {
        root = new TNode();
        root->key = nullptr;
        size = 0;
    }

    void DestructR(TNode *node)
    {
        if (node->left->bit > node->bit)
            DestructR(node->left);
        if (node->right->bit > node->bit)
            DestructR(node->right);
        delete node;
    }

    ~TTrie()
    {
```

```
    DestructR(root);
}

TNode *Find(TKey *key)
{
    TNode *p = root;
    TNode *q = root->left;

    while (p->bit < q->bit)
    {
        p = q;
        q = (GetBit(key, q->bit) ? q->right : q->left);
    }

    if (!Equal(key, q->key))
        return 0;

    return q;
}

TNode *Insert(TKey *key, TValue value)
{
    TNode *p = root;
    TNode *q = root->left;
    while (p->bit < q->bit)
    {
        p = q;
        q = (GetBit(key, q->bit) ? q->right : q->left);
    }

    if (Equal(key, q->key))
        return 0;

    int lBitPos = FirstDifBit(key, q->key);

    p = root;
    TNode *x = root->left;

    while (p->bit < x->bit && x->bit < lBitPos)
    {
        p = x;
        x = (GetBit(key, x->bit) ? x->right : x->left);
```

```
    }

    try
    {
        q = new TNode();
    }
    catch (const std::bad_alloc &e)
    {
        std::cout << "ERROR: fail to allocate the requested storage space\n";
        return 0;
    }

    q->Initialize(lBitPos, key, value,
                  (GetBit(key, lBitPos) ? x : q),
                  (GetBit(key, lBitPos) ? q : x));

    if (GetBit(key, p->bit))
        p->right = q;
    else
        p->left = q;

    size++;
    return q;
}

void KVCopy(TNode *src, TNode *dest)
{
    if (strlen(dest->key) < strlen(src->key))
    {
        delete[] dest->key;
        dest->key = new char[strlen(src->key) + 1];
    }
    strcpy(dest->key, src->key);

    dest->value = src->value;
}

bool Delete(TKey *k)
{
    // прадед удаляемого узла pp, родитель p и сам сын t (сына предстоит удалить)
    TNode *p, *t, *pp = 0;
```

```
    p = root;
    t = (p->left);

    // найдем pp, p и t
    while (p->bit < t->bit)
    {
        pp = p;
        p = t;
        t = (GetBit(k, t->bit) ? t->right : t->left);
    }

    // если ключа искомого-то и нет -- выходим
    if (!Equal(k, t->key))
        return false;

    TNode *x, *r;
    char *key;

    // если p == t, то у t есть селфпоинтер. в таком случае достаточно лишь
    // переподвесить к родителю (pp) "реальный" указатель t, который не селфпоинтер
    if (p != t)
    {
        // иначе же, кладем ключ и знач. p в t, чтобы далее удалять именно p, а не t
        KVCopy(p, t);

        key = p->key;
        r = p;
        x = (GetBit(key, p->bit) ? p->right : p->left);

        // ищем того, кто на p бекпоинтерит (будет лежать в r; а x, по сути, будет в точности равняться p)
        while (r->bit < x->bit)
        {
            r = x;
            x = (GetBit(key, x->bit) ? x->right : x->left);
        }

        // и вместо бекпоинтера на p, будем бекпоинтерить на t
        if (GetBit(key, r->bit))
            r->right = t;
        else
            r->left = t;
```

```
        }

        // остается подвесить к родителю p (pp) "реальный" указатель p, который не
селфпоинтер
        TNode *ch = (GetBit(k, p->bit) ? p->left : p->right);
        if (GetBit(k, pp->bit))
            pp->right = ch;
        else
            pp->left = ch;

        // и беззаботно удалить p: больше на него никто не указывает, ведь мы избавились
        // и от бекпоинтера на него, и от родительского (pp) указателей сверху
        delete p;

        size--;

        return true;
    }

    void Save(std::ofstream &file)
    {
        // подаем размер дерева
        file.write((const char *)&(size), sizeof(int));

        // пронумеровка узлов, инициализация массива указателей
        int index = 0;
        TNode **nodes;
        try
        {
            nodes = new TNode *[size + 1];
        }
        catch (const std::bad_alloc &e)
        {
            std::cout << "ERROR: fail to allocate the requested storage space\n";
            return;
        }
        enumerate(root, nodes, index);

        // теперь просто последовательно (как при обходе в enumerate)
        // подаем всю инфу об узлах, но вместо указателей left/right подаем
        // айди узлов (каковы они были при обходе в enumerate) left/right
        TNode *node;
```

```
    for (int i = 0; i < (size + 1); ++i)
    {
        node = nodes[i];
        file.write((const char *)&(node->value), sizeof(TValue));
        file.write((const char *)&(node->bit), sizeof(int));
        int len = node->key ? strlen(node->key) : 0;
        file.write((const char *)&(len), sizeof(int));
        file.write(node->key, sizeof(char) * len);
        file.write((const char *)&(node->left->id), sizeof(int));
        file.write((const char *)&(node->right->id), sizeof(int));
    }
    delete[] nodes;
}

void enumerate(TNode *node, TNode **nodes, int &index)
{
    // важно, что index передается по ссылке: айди узлов не будут повторяться
    node->id = index;
    nodes[index] = node;
    ++index;
    if (node->left->bit > node->bit)
    {
        enumerate(node->left, nodes, index);
    }
    if (node->right->bit > node->bit)
    {
        enumerate(node->right, nodes, index);
    }
}

void Load(std::ifstream &file)
{
    // считываем размер
    int n;
    file.read((char *)&n, sizeof(int));
    size = n;
    // если он нуль - выходим
    if (!size)
        return;

    TNode **nodes = new TNode *[size + 1];
    // рут уже инициализировался, когда мы пишем создали new Trie()
```

```
        // незачем этого делать повторно
        nodes[0] = root;
        for (int i = 1; i < (size + 1); ++i)
            // а вот прочие узлы надо инитнуть
            nodes[i] = new TNode();

        // поля узлов, которые нам предстоит считывать
        int bit;
        int len;
        TKey *key = 0;
        TValue value;
        int idLeft, idRight;

        for (int i = 0; i < (size + 1); ++i)
        {
            file.read((char *)&(value), sizeof(TValue));
            file.read((char *)&(bit), sizeof(int));
            file.read((char *)&(len), sizeof(int));
            if (len)
            {
                key = new char[len + 1];
                key[len] = 0;
            }
            file.read(key, len);
            // поскольку считываем в том же порядке, что и писали в Load-е
            // айди узлов-сыновей будут сохранять свой порядок, и дерево соберется таким же
            file.read((char *)&(idLeft), sizeof(int));
            file.read((char *)&(idRight), sizeof(int));
            nodes[i]->Initialize(bit, key, value, nodes[idLeft], nodes[idRight]);
            delete[] key;
        }

        delete[] nodes;

        return;
    }

#define safeKey(node) (node->key ? node->key : "root")

    void PrintDefinitions(TNode *node, std::ofstream &out)
    {
        out << ' ' << safeKey(node) << "[label=\"" << safeKey(node) << ", " << node->bit << "\"];\
```

```
n";
        if (node->left->bit > node->bit)
            PrintDefinitions(node->left, out);
        if (node->right->bit > node->bit)
            PrintDefinitions(node->right, out);
    }

    void PrintRelations(TNode *node, std::ofstream &out)
    {
        if (node->left->bit > node->bit)
        {
            out << ' ' << safeKey(node) << "->" << safeKey(node->left) << "[label=\"l\"];\n";
            PrintRelations(node->left, out);
        }
        else
        {
            out << ' ' << safeKey(node) << "->" << safeKey(node->left) << "[label=\"l\"];\n";
        }
        if (node->right->bit > node->bit)
        {
            out << ' ' << safeKey(node) << "->" << safeKey(node->right) << "[label=\"r\"];\n";
            PrintRelations(node->right, out);
        }
        else
        {
            out << ' ' << safeKey(node) << "->" << safeKey(node->right) << "[label=\"r\"];\n";
        }
    }
};
```

Additional.hpp:

```
#pragma once

// проверка на равенство строк
static inline bool Equal(char *a, char *b)
{
    if (a == 0 || b == 0)
        return 0;
    return (strcmp(a, b) == 0);
}

// получить bit-ый по счету бит слева
```

```
static inline int GetBit(char *key, int bit)
{
    if (bit < 0)
        bit = 0;
    return ((key[bit / 8] - 'a') >> (7 - (bit % 8))) & 1;
}

static inline int FirstDifBit(char *a, char *b)
{
    if (a == 0 || b == 0)
        return 0;

    int i = 0;

    // продвижение по байтам
    while (a[i] == b[i])
        i++;
    i *= 8;

    // добивка: продвижание по биту в найденном байте
    while (GetBit(a, i) == GetBit(b, i))
        i++;

    return i;
}

static inline void Lowercase(char *str) {
    for (int i = 0; i < strlen(str); ++i) {
        str[i] = str[i] >= 'A' && str[i] <= 'Z' ? str[i] - 'A' + 'a' : str[i];
    }
}
```

main.cpp:

```
#include <iostream>
#include <fstream>
#include <cstring>

#include "Additional.hpp"
#include "Trie.hpp"

int main()
```

```
{
    // оптимизация
    std::ios::sync_with_stdio(false);
    std::cin.tie(0);
    std::cout.tie(0);

    // потоки i/o -- файлы для сериализации/десериализации, хранения/считывания конструкции патриции
    std::ofstream fout;
    std::ofstream dotout;
    std::ifstream fin;

    char input[MAXLEN];
    TValue value;

    // основное дерево trie
    TTrie *trie;
    try
    {
        trie = new TTrie();
    }
    catch (const std::bad_alloc &e)
    {
        std::cout << "ERROR: fail to allocate the requested storage space\n";
        exit(0);
    }

    TNode *node;

    while ((std::cin >> input))
    {
        if (!std::strcmp(input, "+"))
        {
            std::cin >> input;
            Lowercase(input);
            std::cin >> value;

            std::cout << (trie->Insert(input, value) ? "OK" : "Exist");
            std::cout << '\n';
        }
        else if (!std::strcmp(input, "-"))
        {
```

```
    std::cin >> input;
    Lowercase(input);

    std::cout << (trie->Delete(input) ? "OK" : "NoSuchWord");
    std::cout << '\n';
}
else if (!std::strcmp(input, "!"))
{
    std::cin >> input;
    if (!std::strcmp(input, "Save"))
    {
        std::cin >> input;
        fout.open(input, std::ios::out | std::ios::binary | std::ios::trunc);
        if (!fout.is_open()) {
            std::cout << "ERROR: can't create file\n";
            continue;
        }

        trie->Save(fout);
        std::cout << "OK\n";

        fout.close();
    }
    else if (!std::strcmp(input, "Load"))
    {
        std::cin >> input;
        fin.open(input, std::ios::in | std::ios::binary);
        if (!fin.is_open()) {
            std::cout << "ERROR: can't open file\n";
            continue;
        }

        delete trie;
        trie = new TTrie();
        trie->Load(fin);

        std::cout << "OK\n";

        fin.close();
    }
}
else if (!std::strcmp(input, "1p"))
```

```
        {
            std::ofstream dot;
            dot.open("source.dot", std::ios::out | std::ios::trunc);

            dot << "digraph {\n";
            trie->PrintDefinitions(trie->root, dot);

            trie->PrintRelations(trie->root, dot);

            dot << "}\n";

            dot.flush(), dot.close();

            if (system("dot source.dot -Tpng -o res.png") == -1) {
                std::cout << "ERROR: fail compiling source.dot file into png\n";
                continue;
            }

            if (system("xdg-open res.png") == -1) {
                std::cout << "ERROR: fail trying to open res.png\n";
                continue;
            }
        }
        else
        {
            Lowercase(input);
            node = trie->Find(input);
            if (!node)
                std::cout << "NoSuchWord";
            else
                std::cout << "OK: " << node->value;
            std::cout << '\n';
        }
    }

    delete trie;

    return 0;
}
```

3. Консоль

```
[leo@pc Pat]$ cat inprog
! Save db0
- bb
! Load db0
ca
- bb
[leo@pc Pat]$ ./prog <inprog
OK
NoSuchWord
OK
NoSuchWord
NoSuchWord
[leo@pc Pat]$ cat inprog
- ba
c
- ccc
+ a 1114405545
+ ccb 510116504
- abb
+ a 1628686999
c
bc
bcb
[leo@pc Pat]$ ./prog <inprog
NoSuchWord
NoSuchWord
NoSuchWord
OK
OK
NoSuchWord
Exist
NoSuchWord
NoSuchWord
NoSuchWord
```

```
[leo@pc Pat]$ cat inprog
- ba
c
bca
- bb
! Save db1
+ caa 1827191341
+ bca 1875584253
c
+ abb 765090149
- ba
- c
ca
! Save db0
- bca
+ a 1294963798
[leo@pc Pat]$ ./prog <inprog
NoSuchWord
NoSuchWord
NoSuchWord
NoSuchWord
OK
OK
OK
NoSuchWord
OK
NoSuchWord
NoSuchWord
NoSuchWord
OK
OK
OK
```

4. Тест производительности

Сравнение производительности будет производится с красно-черным деревом, представленном в стандартной библиотеке шаблонов C++ – контейнером map.

Тест №1: добавление 1000 элементов.

Тест №2: добавление 1000000 элементов.

Тест №3: добавление 1000 элементов, удаление каждого 4-го.

Тест №4: добавление 1000000 элементов, удаление каждого 4-го.

Тест №5: сначала добавление 1000 элементов, потом их удаление.

Тест №6: сначала добавление 1000000 элементов, потом их удаление.

Время исполнения представлено в микросекундах.

Test	map	PATRICIA	Times faster
1	368	179	2,055865922
2	336192	178322	1,885308599
3	2238	725	3,086896552
4	512560	154529	3,31691786
5	2289	1195	1,915481172
6	431921	285703	1,511783215

Как оказалось, для хранения слов PATRICIA Trie оказалась много эффективней красно-черного дерева. Не удивительно, ведь сложность операций поиска/добавления/удаления в КЧД — O(log(n)) , где n — число элементов в дереве; в случае же Патриции, сложность этих же операций — O(k), где k — длина слова. Притом Патриция сравнивает лишь определенные биты слов во время их поиска, а КЧД же – все слово целиком. Нет так же затрат на балансировку Патриции в отличии от КЧД. Наблюдательно, что Патриция в результате вставки случайных слов сама по себе является почти идеально-сбалансированным деревом, и тем более она таковой является, чем длиннее слова в нее вставляемые.

Ниже приведено несколько изображений Патриции со случайным словами.

Рис. 1. Никнеймы Машнова Вячеслава, православного российского рэпера.

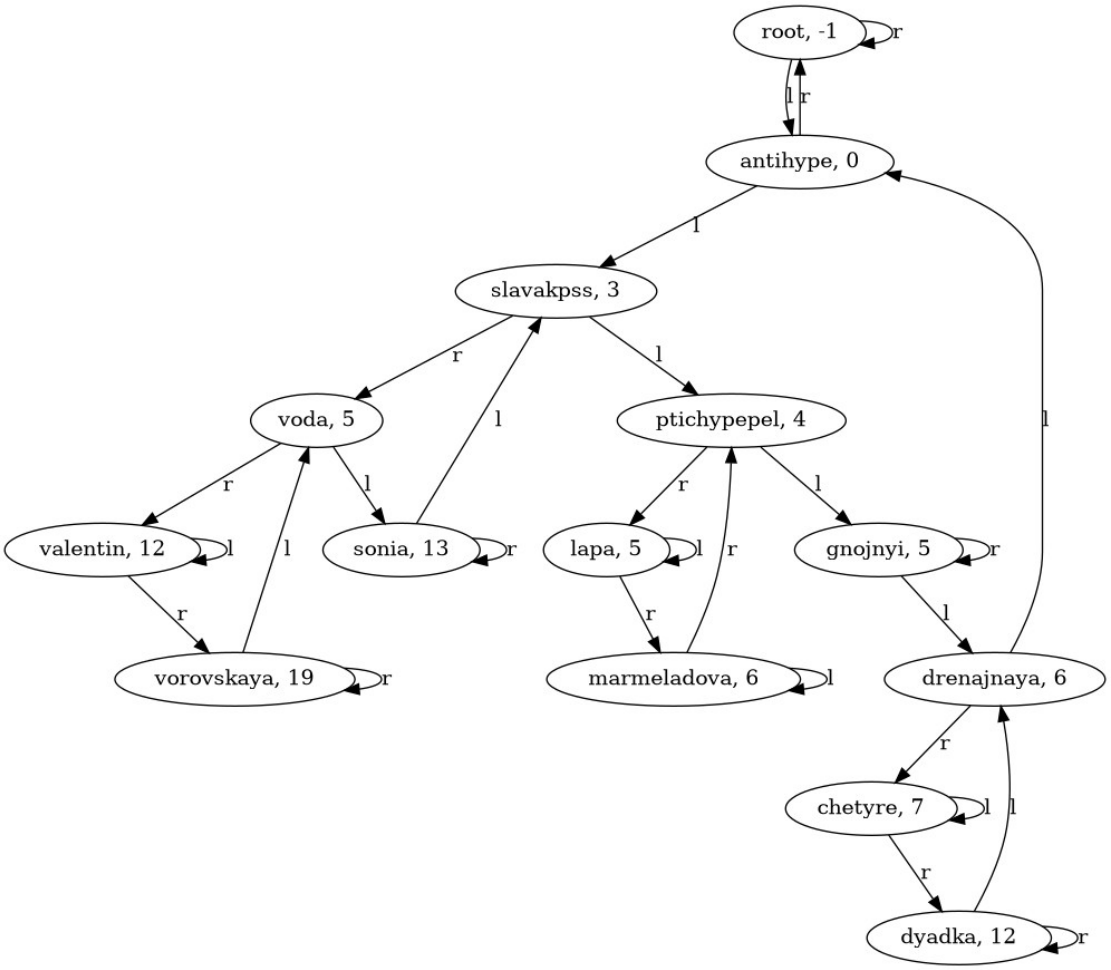
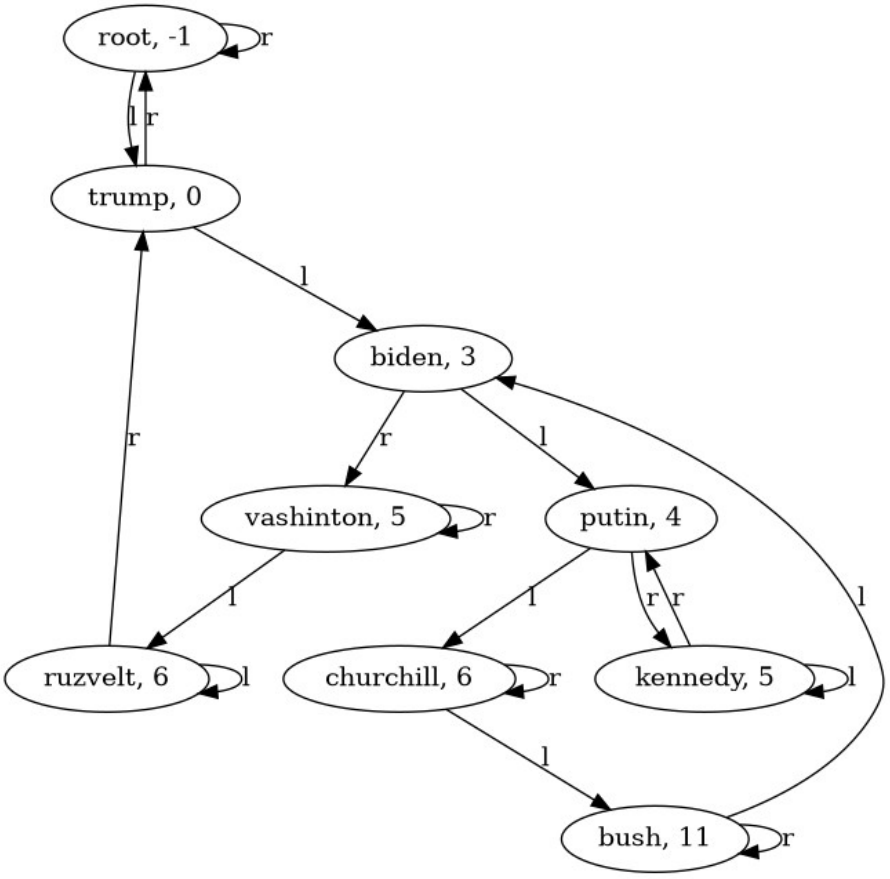


Рис. 2. Мой личный топ 8 президентов мира.



5. Выводы

Структура данных патриция — чудесная вещь, используется чаще всего для построения ассоциативных массивов с ключами, которые могут быть представлены в виде строк. Патриция находит особое применение в сфере IP маршрутизации, там, где возможность содержать большие диапазоны значений за редким исключением особенно подходит для иерархической организации IP-адресов. Патриция также используется для инвертированных индексов текстовых документов в информационном поиске.

В отличие от сбалансированных деревьев, деревья Патриции допускают поиск, вставку и удаление за O(k), а не O(log n). Это не кажется преимуществом, так как обычно $k \geq \log n$, но в сбалансированном дереве каждое сравнение представляет собой сравнение строк, требующее O(k) времени, что на практике крайне медленно из-за длинных общих префиксов слов. В Trie все сравнения требуют константного времени, но для поиска строки длиной m требуется m сравнений. Деревья Патриции же могут выполнять эти операции с меньшим количеством сравнений и требуют гораздо меньшего количества узлов.

Однако деревья Патриции также имеют общие недостатки Trie-деревьев: поскольку они могут быть применены только к строкам элементов или элементам с эффективно обратимым отображением на строки, им не хватает «универсальности» сбалансированных деревьев поиска, которые применимы к любому типу данных с полным упорядочением. Обратимое сопоставление со строками может быть использовано для получения требуемого общего порядка для сбалансированных деревьев поиска, но не наоборот. Это также может быть проблематично, если тип данных предоставляет только операцию сравнения, но не операцию (де)сериализации.