Московский авиационный институт (национальный исследовательский университет)

Факультет компьютерных наук и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу «Дискретный анализ»

Студент: С. Ю. Свиридов
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-22

Дата:
Оценка:
Подпись:

Москва 2024

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 264 - 1. Разным словам может быть поставлен в соответствие один и тот же номер.

Программа должна обрабатывать строки входного файла до его окончания. Каждая строка может иметь следующий формат:

+ **word 34** — добавить слово «word» с номером 34 в словарь. Программа должна вывести строку «OK», если операция прошла успешно, «Exist», если слово уже находится в словаре.

- word — удалить слово «word» из словаря. Программа должна вывести «OK», если слово существовало и было удалено, «NoSuchWord», если слово в словаре не было найдено.

word — найти в словаре слово «word». Программа должна вывести «OK: 34», если слово было найдено; число, которое следует за «OK:» — номер, присвоенный слову при добавлении. В случае, если слово в словаре не было обнаружено, нужно вывести строку «NoSuchWord».

! Save /path/to/file — сохранить словарь в бинарном компактном представлении на диск в файл, указанный парамером команды. В случае успеха, программа должна вывести «OK», в случае неудачи выполнения операции, программа должна вывести описание ошибки (см. ниже).

! Load /path/to/file — загрузить словарь из файла. Предполагается, что файл был ранее подготовлен при помощи команды Save. В случае успеха, программа должна вывести строку «OK», а загруженный словарь должен заменить текущий (с которым происходит работа); в случае неуспеха, должна быть выведена диагностика, а рабочий словарь должен остаться без изменений. Кроме системных ошибок, программа должна корректно обрабатывать случаи несовпадения формата указанного файла и представления данных словаря во внешнем файле.

Для всех операций, в случае возникновения системной ошибки (нехватка памяти, отсутсвие прав записи и т.п.), программа должна вывести строку, начинающуюся с «ERROR:» и описывающую на английском языке возникшую ошибку.

Вариант структуры: PATRICIA.

1 Описание

Дерево `PATRICIA` (Practical Algorithm To Retrieve Information Coded In Alphanumeric) - это дерево, использующееся для эффективного поиска больших строк разной длины. Эффективность достигается тем, что каждый узел дерева, помимо самой строки, хранит так же и `индекс` - номер первого бита строки, отличающегося от бита в предыдущей строке. Кроме того, в этом дереве присутствует строгое увеличение индекса по мере увеличения глубины узла, поэтому успешность поиска близка к 100 процентам. Каждая вершина `Patricia` хранит в себе два указателя - левый и правый.

Для реализации программы-словаря на основе указанной структуры данных, требуется описать саму структуру данных, описать функции добавления узла в дерево, удаления узла из дерева, поиска и возвращения значения узла, а так же функции загрузки словаря в файл и выгрузки его оттуда. Так же следует написать пользовательский интерфейс, позволяющий пользователю взаимодействовать с программой.

2 Исходный код

Перед написанием кода объявим константу `BIT_COUNT` и присвоим ей значение `пять`. Эта условность сделана потому, что все ключи узлов будут записаны в нижнем регистре и для каждого символа рассматриваются 5 последних битов, поскольку только они отличаются от битов верхнего регистра соответсвтующего символа. Создадим класс `Patricia` и в модификаторе доступа `private` опишем структуру `Node`, имеющую пять полей для работы с деревом `Patricia`, а так же конструктор и деструктор. Также в классе объявим некоторые поля для работы с деревом, реализацию которых представим позже.

```
1  const int BIT_COUNT = 5;
2
3  typedef class Patricia {
4  private:
5      struct Node {
6          std::string key;
7          unsigned long long value;
8          size_t index;
9          Node *right, *left;
10
11         Node(const std::string& key, const unsigned long long& value, const int& index)
12             : key(key), value(value), index(index), left(nullptr), right(nullptr) { }
13
14         ~Node(){}
15     };
16
17     Node *root = nullptr;
18     Node* search(const std::string& finding) const;
19     Node** Search_Parent(const std::string& finding) const;
20     void Insert(const std::string& key, const unsigned long long& value, const size_t&
           index);
21
22     void SaveData(const Patricia::Node *node, std::ofstream &stream);
23     void RecursiveSave(const Patricia::Node *node, std::ofstream &stream);
24     Node* LoadData(std::ifstream &stream);
25 public:
26
27     void Add(const std::string& key, unsigned long long value);
28     unsigned long long Get_Value(const std::string& finding);
29     void Remove(const std::string& key);
30     void DeleteNode(Node* node);
31     void DeletePatricia();
32
33     void SaveToFile(const std::string &path);
34     void LoadFromFile(const std::string& path);
35 } patr;
```

Напишем маленькую функцию, которая будет принимать на вход строку, перемещать все ее символы в нижний регистр и возвращать ее.

```
std::string ToLowercase(std::string str) {
    for (char& c : str) {
        c = std::tolower(c);
    }
    return str;
}
```

Опишем функцию поиска вершины по ключу. Эта функция всегда будет возвращать какую либо вершину типа `Node`, даже если искомой не существует (в таком случае функция вернет максимально похожую на искомую вершину). Поиск начинается с корня дерева. Запоминаются два указателя: на левую вершину от корня и на сам корень, затем происходит спуск по дереву. Запускается цикл `while`, которые будет выполнятся, пока индекс текущей вершины будет больше, чем индекс предыдущей, то есть пока мы не перейдем по обратной ссылке. В цикле вычисляется номер символа в строке, которая является ключом, затем при помощи оператора смещения нужный бит (по индексу) нужного символа сравнивается с единицей. Если бит - единица, то переходим по `правому указателю текущей вершины`, иначе - по `левому`.

```
Patricia::Node* Patricia::search(const std::string& finding) const {
    Node *currentNode = root->left, *prevNode = root;
    while (currentNode->index > prevNode->index) {
        size_t charIdx = (currentNode->index - 1) / BIT_COUNT;
        if (charIdx >= finding.size()) {
            prevNode = currentNode;
            currentNode = currentNode->left;
            continue;
        }
        char currentChar = finding[charIdx];
        int offset = (BIT_COUNT - 1 - ((currentNode->index - 1) % BIT_COUNT));
        bool currentBit = (currentChar >> offset) & 1;
        prevNode = currentNode;

        if (currentBit) {
            currentNode = currentNode->right;
        }
        else {
            currentNode = currentNode->left;
        }
    }
    return currentNode;
}
```

Опишем функцию добавления узла в дерево. В качестве аргумента функция принимает ключ и значение будущей вершины. Первым делом проверяем, есть ли в нашем дереве корень. Если нет, то добавляемая вершина станет им и функция вернет управ-

ление. Затем проверяется проверка на существование узла с таким ключом в дереве. Выполняется поиск вершины; в случае, если ключ найденной вершины совпадет с ключом искомой вершины, то функция выкинет ошибку с описанием "Exist". После чего запускается цикл `while`, служащий для получения индекса несовпадающего бита в строке, найденной функцией поиска и строке, являющейся ключом нового узла. После того, как различие было найдено, вычисляется индекс текущего бита (с учетом номера символа в строке) и с помощью функции вставки `Insert` вставляется в дерево. Цикл на этом моменте завершается

```
1  void Patricia::Add(const std::string& key, unsigned long long value){
2      if(!root){
3          root = new Node(key, value, 0);
4          root->left = root;
5          return;
6      }
7
8      Node *foundNode = search(key);
9      if(foundNode->key == key)
10         throw std::runtime_error("Exist\n");
11
12     bool run = true;
13     size_t charIndex = 0;
14     while(run){
15         char foundedKey = (foundNode->key.size() > charIndex ? foundNode->key[charIndex
               ] : '\0');
16         char inputKey = (key.size() > charIndex ? key[charIndex] : '\0');
17         for(size_t i = 0; i < BIT_COUNT; ++i){
18             bool foundedKeyBit = foundedKey >> (BIT_COUNT - 1 - i) & 1;
19             bool inputKeyBit = inputKey >> (BIT_COUNT - 1 - i) & 1;
20             if(foundedKeyBit != inputKeyBit){
21                 Insert(key, value, charIndex * BIT_COUNT + i + 1);
22                 run = false;
23                 break;
24             }
25         }
26         ++charIndex;
27     }
28 }
```

Опишем функцию вставки `Insert`. Она нужна для того, чтобы выделить память для новой вершины, а также правильно выбрать место для нее и связать ее указатели с существующими вершинами в дереве. Сначала выполняется поиск искомой вершины по ее ключу, но так как ее еще не существует в дереве, то мы получаем максимально похожую на нее вершину, на которую указывает указатель `currentNode`, причем индекс у этой вершины больше, чем у вставляемой. Второй указатель `prevNode` указывает на вершину с индексом, меньше, чем у вставляемой. Фактически, новая вершина встанет между вершинами `currentNode` и `prevNode`. Выделяем для нее память

и передаем значения из аргументов функции. Затем, нужно связать ее указатели с указателями уже имеющихся в дереве вершин. Сначала смотрим на то, каким ребенком была искомая вершина `currentNode`: если она была левым ребенком - то левым ребенком станет новая вершина и наоборот. После этого смотрим на бит в ключе новой вершины по ее же индексу: если бит - единица, то правая ссылка новой вершины будет указывать сама на себя, а левая - на потомка и наоборот в случае, если бит - нулевой. Потомком новой вершины будет являтся старый потомок ее родителя, то есть вершина `currentNode`.

```
1   void Patricia::Insert(const std::string& key, const unsigned long long& value,
        const size_t& index){
2   Node *currentNode = root->left, *prevNode = root;
3
4   while(currentNode->index > prevNode->index){
5       if(currentNode->index > index)
6           break;
7
8       size_t charIndex = (currentNode->index - 1) / BIT_COUNT;
9       if(charIndex >= key.size()){
10          prevNode = currentNode;
11          currentNode = currentNode->left;
12          continue;
13      }
14      char currentChar = key[charIndex];
15      int offset = (BIT_COUNT - 1 - ((currentNode->index - 1) % BIT_COUNT));
16      bool currentBit = (currentChar >> offset) & 1;
17
18      prevNode = currentNode;
19      currentBit ? currentNode = currentNode->right
20                 : currentNode = currentNode->left;
21  }
22  char getCharFromKey = key[(index - 1) / BIT_COUNT];
23  bool getBit = getCharFromKey >> (BIT_COUNT - 1 - (index - 1) % BIT_COUNT) & 1;
24  Node *newNode = new Node(key, value, index);
25
26  if(prevNode->left == currentNode)
27      prevNode->left = newNode;
28  else
29      prevNode->right = newNode;
30
31  getBit ? (newNode->right = newNode, newNode->left = currentNode)
32         : (newNode->left = newNode, newNode->right = currentNode);
33 }
```

Опишем функцию `Get_Value` по получению значения вершины по ее ключу. Функция очень простая: выполняется поиск вершины по ключу, указанному в аргументах фукции и возвращается ее значение в случае совпадения ключа найденной вершины и ключа из аргумента функции. Иначе, в случае, если была найдена другая вершина

или дерево пустое, выкидывается ошибка с описанием "NoSuchWord".

```
unsigned long long Patricia::Get_Value(const std::string& finding) {
    if (!root) {
        throw std::runtime_error("NoSuchWord\n");
    }

    Node* get = search(finding);
    if (get->key == finding) {
        return get->value;
    }
    throw std::runtime_error("NoSuchWord\n");
}
```

Итак, самое сложное - функция удаления вершины из дерева. Основная сложность заключается в том, чтобы сохранить увеличение индексов вершин по мере увеличения глубины дерева. Также нужно сохранить правильно все ссылки каждой вершины, чтобы корренктно работал поиск. Итак, при удалении возможны три случая: Первый - когда дерево состоит только из корня и он удаляется, второй - когда удаляемый элемент является листом дерева (одна из его ссылок указывает на себя, другая - на потомка) и третий - когда удаляемый элемент - вершина, обе ссылки которой указывают на ее потомком. Для этого требуется дополнительная функция, поскольку структура вершины дерева не содержит поля `parent`.

```
Patricia::Node** Patricia::Search_Parent(const std::string& finding) const {
    Node** arr = new Node*[3];
    Node *currentNode = root->left, *prevNode = root, *prevPrevNode = root;
    while(currentNode->index > prevNode->index) {
        size_t charIndex = (currentNode->index - 1) / BIT_COUNT;

        if(charIndex >= finding.size()){
            prevPrevNode = prevNode;
            prevNode = currentNode;
            currentNode = currentNode->left;
            continue;
        }

        char currentChar = finding[charIndex];
        int offset = (BIT_COUNT - 1 - ((currentNode->index - 1) % BIT_COUNT));
        bool currentBit = (currentChar >> offset) & 1;

        prevPrevNode = prevNode;
        prevNode = currentNode;
        currentBit ? currentNode = currentNode->right
                   : currentNode = currentNode->left;
    }

    arr[0] = currentNode;
    arr[1] = prevNode;
```

```
    arr[2] = prevPrevNode;

    return arr;
}
```

Функция возвращает указатель на массив указателей типа `Node`, первый элемент которого `arr[0]` содержит ссылку на удаляемую вершину (`X`), второй элемент содержит ссылку на владельца удаляемой вершины (`Q`), (Вершина A является владельцем вершины B, если один из ее указателей указывает на вершину B); третий элемент содержит ссылку на владельца владельца удаляемой вершины (`P`).

В первом случае все очевидно, просто удаляем корень и возвращаем управление. Во втором случае мы ищем владельца нашей удаляемой вершины и родителя этого владельца. `Arr[1]` содержит ссылку на владельца удаляемой вершины, а `arr[2]` содержит ссылку на родителя владельца. Для того, чтобы определить является ли наш удаляемый узел листом (попадаем ли мы во второй кейс), достаточно просто сравнить указатели из `arr[0]` и `arr[1]`. Если они равны, то перед нами второй кейс. Для начала надо понять, каким ребеном является удаляемый узел. Если правым, то смотрим на то, какая из ссылок удаляемого узла указывает на потомка, после чего делаем новым потомком родителя удаляемого узла потомка удаляемого узла. Если удаляемый узел является левым ребенком своего родителя, то левым ребенком родителя удаляемого узла становится потомок удаляемого узла. Если проще, то внук родителя удаляемого узла становится его сыном, а удаялемый узел, сын родителя, удаялется.

```
void Patricia::Remove(const std::string& key) {
    if (!root) {
        throw std::runtime_error("NoSuchWord\n");
    }

    Node** arr = Search_Parent(key);
    Node* deleteNode = arr[0];
    Node* ownerDeleteNode = arr[1];
    Node* parentOwnerDeleteNode = arr[2];

    if (deleteNode->key != key) {
        throw std::runtime_error("NoSuchWord\n");
    }

    if (deleteNode == root && root->left == root) {
        delete root;
        root = nullptr;
        std::cout << "OK" << std::endl;
        return;
    }

    if (ownerDeleteNode == deleteNode) {
```

```
        if (parentOwnerDeleteNode->right == deleteNode) {
            if (deleteNode->right == deleteNode) {
                parentOwnerDeleteNode->right = deleteNode->left;
            }
            else {
                parentOwnerDeleteNode->right = deleteNode->right;
            }
        }
        else {
            if (deleteNode->right == deleteNode) {
                parentOwnerDeleteNode->left = deleteNode->left;
            }
            else {
                parentOwnerDeleteNode->left = deleteNode->right;
            }
        }
        delete deleteNode;
        std::cout << "OK" << std::endl;
        return;
    }
}
```

В третьем случае, когда удаляемая вершина `X` имеет двух потомков `A` и `B`, нужно найти владельца `Q` вершины `X` и владельца `P` владельца `Q` вершины `X`. Чтобы получить вершину `P` нужно для вершины `Q` вызвать функцию `Search_Parent`, описанную ранее. Дальнейшие действия следующие: Сначала нужно скопировать данные вершины Q в вершину X. Затем ссылку вершины `P`, указывающую на вершину `Q` перенаправить на вершину `X`, после чего, согласно второму кейсу, произвести удаление вершины `Q`. Ниже представлен полный код для функции удаления

```
void Patricia::Remove(const std::string& key) {
    if (!root) {
        throw std::runtime_error("NoSuchWord\n");
    }

    Node** arr = Search_Parent(key);
    Node* deleteNode = arr[0];
    Node* ownerDeleteNode = arr[1];
    Node* parentOwnerDeleteNode = arr[2];

    if (deleteNode->key != key) {
        throw std::runtime_error("NoSuchWord\n");
    }

    if (deleteNode == root && root->left == root) {
        delete root;
        root = nullptr;
        std::cout << "OK" << std::endl;
        return;
```

```
    }

    if (ownerDeleteNode == deleteNode) {
        if (parentOwnerDeleteNode->right == deleteNode) {
            if (deleteNode->right == deleteNode) {
                parentOwnerDeleteNode->right = deleteNode->left;
            }
            else {
                parentOwnerDeleteNode->right = deleteNode->right;
            }
        }
        else {
            if (deleteNode->right == deleteNode) {
                parentOwnerDeleteNode->left = deleteNode->left;
            }
            else {
                parentOwnerDeleteNode->left = deleteNode->right;
            }
        }
        delete deleteNode;
        std::cout << "OK" << std::endl;
        return;
    }


    Node** arr1 = Search_Parent(ownerDeleteNode->key);
    Node* OwnerOwnerDeleteNode = arr1[1];

    deleteNode->key = ownerDeleteNode->key;
    deleteNode->value = ownerDeleteNode->value;

    if(OwnerOwnerDeleteNode == ownerDeleteNode){
        if (parentOwnerDeleteNode->right == ownerDeleteNode) {
            parentOwnerDeleteNode->right = deleteNode;
        }
        else {
            parentOwnerDeleteNode->left = deleteNode;
        }
    }
    else {
        if (parentOwnerDeleteNode->right == ownerDeleteNode) {
            if (ownerDeleteNode->right == deleteNode) {
                parentOwnerDeleteNode->right = ownerDeleteNode->left;
            }
            else {
                parentOwnerDeleteNode->right = ownerDeleteNode->right;
            }
        }
        else {
```

```
            if (ownerDeleteNode->right == deleteNode) {
                parentOwnerDeleteNode->left = ownerDeleteNode->left;
            }
            else {
                parentOwnerDeleteNode->left = ownerDeleteNode->right;
            }
        }
        if (OwnerOwnerDeleteNode->right == ownerDeleteNode) {
            OwnerOwnerDeleteNode->right = deleteNode;
        }
        else {
            OwnerOwnerDeleteNode->left = deleteNode;
        }
    }
    delete ownerDeleteNode;
    std::cout << "OK" << std::endl;
}
```

Далее опишем процесс сохранения текущего словаря в файл. Для этого потребуется описать три функции. Первая из них принимает на вход дерево (его узел) и файл. Эта функция сохраняет в файл данные текущей вершины, указанной в аргументах. В файле в одну строчку записываются три поля: длина ключа, ключ и значение.

```
void Patricia::SaveData(const Patricia::Node *node, std::ofstream &stream)
    std::string key = node->key;
    size_t keySize = key.size();
    unsigned long long value = node->value;
    stream.write((char *) (&keySize), sizeof(size_t));
    stream.write(key.c_str(), sizeof(char) * keySize);
    stream.write((char *) (&value), sizeof(unsigned long long));
}
```

Вторая функция из вышеназванных выполняет рекурсивный обход дерева и вызывает в каждом его узле функцию `SaveData`.

```
void Patricia::RecursiveSave(const Patricia::Node *node, std::ofstream &stream) {
    SaveData(node, stream);
    if (node->left->index > node->index) {
        RecursiveSave(node->left, stream);
    }
    if (node->right->index > node->index) {
        RecursiveSave(node->right, stream);
    }
}
```

Третья функция из этого списка нужна для работы с пользователем и в аргументах принимает лишь путь то файла, в который будет сохранено дерево. Сначала в функции открывается файл на запись бинарных данных. Далее, в случае, если дерево не пустое вызывается функция `SaveData` для сохранения данных корня и затем, в

случае, если дерево состоит не только из корня, но и имеет потомков, выполняется его рекурсивный обход, после чего файл закрывается.

```
void Patricia::SaveToFile(const std::string &path) {
    std::ofstream fout;
    fout.open(path, std::ios_base::out | std::ios_base::trunc | std::ios_base::binary);
    if (!root) {
        fout.close();
        return;
    }

    SaveData(root, fout); /
    if (root != root->left) {
        RecursiveSave(root->left, fout);
    }
    fout.close();
}
```

Далее на очереди загрузка словаря из файла. Перед загрузкой нового словаря в дерево, его нужно очистить. Для этого сущетсвует функция очистки дерева, а точнее две. Эти функции, подобно способу загрузки словаря в файл, выполняют рекурсивный обход дерева, и удаляют каждую из его вершин. Таким образом после их выполнения дерево становится пустым.

```
void::Patricia::DeleteNode(Node* node) {
    if (node->left->index > node->index) {
        DeleteNode(node->left);
    }
    if (node->right->index > node->index) {
        DeleteNode(node->right);
    }

    delete node;
}

void::Patricia::DeletePatricia() {
    if (!root) {
        return;
    }
    if (root != root->left) {
        DeleteNode(root->left);
    }
    delete root;
    root = nullptr;
}
```

Далее происходит работа функций для загрузки словаря. Для этого процесса разработаны две функции: Первая из них, функция `LoadData` проверяет корректность формата записи ключа. В этой функции считывается строка из файла и выполня-

ется попытка создания узла дерева (выделения для него памяти), используя считанные данные. После чего созданный узел возвращается. Далее основная функция, предназначенная для работы с пользователем, открывает файл на чтение данных в бинарном виде и производит анализ узла, который вернула функция `LoadData`. Если узел создался, то он добавляется в дерево, с помощью функции `Add`. Таким образом, перед загрузкой узлов в дерево, происходит проверка на корректность их записи в файле.

```
1  void Patricia::LoadFromFile(const std::string &path) {
2      std::ifstream fin;
3      fin.open(path, std::ios_base::in | std::ios_base::binary);
4      Node* curNode;
5      while ((curNode = LoadData(fin)) != nullptr) {
6          Add(curNode->key, curNode->value);
7          delete curNode;
8      }
9  }
10
11 Patricia::Node *Patricia::LoadData(std::ifstream &stream) {
12     if (stream.eof()) {
13     }
14     size_t keySize;
15     std::string key;
16     unsigned long long value;
17     stream.read((char *) &keySize, sizeof(size_t));
18     if (!stream) {
19         return nullptr;
20     }
21     key.resize(keySize);
22     stream.read(key.data(), sizeof(char) * keySize);
23     stream.read((char *) &value, sizeof(sizeof(unsigned long long)));
24     Node *node = new Node(key, value, 0);
25     return node;
26 }
```

3 Консоль

```
stepan@stepan-ASUS:~/Рабочий стол/учеба/prog4sem/discran/laba2$ g++ main.cpp
stepan@stepan-ASUS:~/Рабочий стол/учеба/prog4sem/discran/laba2$ ./a.out
+ a 1
OK
+ A 2
Exist
+ b 2
OK
```

```
+ aaaaaaaaaaaaaaaaaaaaaaaaaaaaa 13
OK
! Save ./temp.txt
OK
+ qq 14114
OK
qq
OK: 14114
! Load ./temp.txt
OK
a
OK: 1
b
OK: 2
aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
OK: 13
qq
NoSuchWord
```

4 Тест производительности

Тесты производительности представляют из себя следующее: Будут сравниваться операции вставки, поиска и удаления пар "ключ-значение"в дереве **Patricia** и в ассоциативном контейнере **map**, в котором все пары отсортированы по ключу. Вставляться будут 4 элемента, искаться и удаляться один.

```
stepan@stepan-ASUS:~/Рабочий стол/учеба/prog4sem/discran/laba2$ ./a.out
Insert in patricia: 13ms
Insert in map container: 7ms

Search in patricia: 1ms
Search in map container: 2ms

Remove in patricia: 2ms
Remove in map container: 2ms
```

Как видно из результатов теста, в структуре **Patricia** поиск проходит хоть и немного, но быстрее, чем в структуре **map**. Дерево **Patricia** эффективно при поиске в больших базах данных, поскольку в нем реализован поиск строк за сложность $\Theta(h)$, где h - высота дерева, в отличие от поиска в RB-деревьях, на которых основан поиск в контейнере **map**, имеющий сложность $\Theta(log_2 n)$, где n - количество элементов в дереве. Конечно, при малом количестве элементов в дереве (у меня их всего 4), поиск за логарифмическую сложность не будет сильно отсавать от поиска за m, даже может быть быстрее, но при большем количестве элементов поиск в дереве **Patricia** будет работать быстрее.

5 Выводы

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я узнал и разобрал новую для меня структуру данных **Patricia**. Я удивился тому, насколько быстро может работать поиск в ней в сравнении с алгоритмом бинарного поиска. Также я сравнил результаты работы некоторых алгоритмов дерева **Patricia** с алгоритмами структуры **map** и проанализировал. В целом, мне было интересно изучать устройство работы дерева и сравнивать его с другими контейнерами. Однако реализация этого дерево показалась мне достаточно сложной и трудноreализуемой.

Список литературы

[1] *Radix tree modifications*
URL: `https://en.wikipedia.org/wiki/Radix_tree/`

[2] *Patricia*
URL: `https://habr.com/ru/articles/575720/`