Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет «Компьютерные науки и прикладная математика»

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу

«Дискретный анализ»

Тема работы

“Сбалансированные деревья”

Студент : А.А. Дудовцев

Группа : М8О-208Б-22

Оценка : _____

Дата : _____

Подпись : _____

Москва, 2024

1. Постановка задачи

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. 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.

Вариант №4: B-дерево

2. Описание

B-дерево (англ. B-tree) — сильноветвящееся сбалансированное дерево поиска,

позволяющее проводить поиск, добавление и удаление элементов за O(logn).

B-дерево является идеально сбалансированным, то есть глубина всех его листьев

одинакова. B-дерево имеет следующие свойства (t— параметр дерева, называемый

минимальной степенью B-дерева, не меньший 2):

- Каждый узел, кроме корня, содержит не менее $t-1$ ключей, и каждый внутренний узел имеет по меньшей мере t дочерних узлов. Если дерево не является пустым, корень должен содержать как минимум один ключ.
- Каждый узел, кроме корня, содержит не более $2t-1$ ключей и не более чем 2t сыновей во внутренних узлах.
- Корень содержит от 1 до $2t-1$ ключей, если дерево не пусто и от 2 до 2t детей при высоте большей 0.
- Каждый узел дерева, кроме листьев, содержащий ключи k1,...,kn, имеет n+1 сына. i-й сын содержит ключи из отрезка $[ki-1;ki]$, $k0=-\infty$, $kn+1=\infty$.
- Ключи в каждом узле упорядочены по неубыванию.
- Все листья находятся на одном уровне.
- Для моей задачи я выбрал t = 3.

3. Разбор программы

Функции работы с BTree:

Function name	Что делает	Time complexity	Space complexity
BTree(int)	Конструктор класса. Принимает t.	O(1)	O(1)
void AddNode(Data);	Добавляет пару ключ-значение в дерево. При успешном добавлении напишет OK. Если ключ уже присутствует - напишет Exis	O(t*logt(N))	O(N)

void AddOnLoad(Data);	Добавляет пару ключ-значение в дерево в функции Load. Не выводит сообщения OK и NoSuchWord	O(t*logt(N))	O(N)
void DeleteNode(char*);	Удаляет переданный ключ из дерева. Если удаление пройдет успешно, напишет OK в консоль, иначе напишет NoSuchWord	O(t*logt(N))	O(1)
void Search(char*);	Ищет переданный ключ в дереве. Если ключ присутствует, то будет выведено OK: `значение по ключу`. Если ключа нет, то в консоль будет выведено NoSuchWord	O(t*logt(N))	O(1)
void SaveToFile(char*);	Сохраняет дерево в бинарном формате в файл	O(N)	O(1)
void LoadFromFile(char*);	Загружает дерево из бинарного формата, которое было подготовлено методом SaveToFile	O(N)	O(N)
~BTree();	Деструктор дерева. Удаляет дерево и всю выделенную память	O(N)	O(1)

Функции работы с Node:

Function name	Что делает	Time complexity	Space complexity
Node(int, bool);	Конструктор класс node. Выделяет память под ключи, под детей. Принемает	O(1)	O(N)

	bool, которое говорит, является ли созданная нода листом		
Node* Search(char*);	Рекурсивный метод поиска ключа в дереве. Вызывается из BTree::Search()	O(t*logt(N))	O(1)
void InsertNonFull(Data);	Метод вставки пары ключ-значение в незаполненный узел. Вызывается из BTree::AddNode	O(t*logt(N))	O(N)
void SplitChild(int, Node*);	Метод, разделяющий заполненный дочерний узел.	O(t)	O(t)
void Remove(char*);	Рекурсивный метод удаления ключа из дерева. Вызывается из BTree::DeleteNode	O(t*logt(N))	O(1)
void RemoveFromLeaf(int);	Метод удаления ключа из листа дерева. Принемает	O(t)	O(1)
void RemoveFromNonLeaf (int);	Метод удаления ключа из внутреннего узла дерева.	O(t*logt(N))	O(1)
void BorrowFromNext(int);	Метод, который берет один элемент из правого брата	O(t)	O(1)
void BorrowFromPrev(int);	Метод, который берет один элемент из левого брата	O(t)	O(1)
void FillNode(int);	Метод, который наполняет узел в t-1 ключей	O(t)	O(1)
void Merge(int);	Метод объединения узла. Он склеивает два ребенка и разделяющий их ключ	O(t)	O(1)
void Save(ofstream&);	Рекурсивный метод сохранения дерева.	O(N)	O(1)

	Сохраняет сначала детей, потом их родителей.		
void Delete();	Рекурсивный метод удаления дерева	O(N)	O(1)

4. Исходный код

```
#include <iostream>
#include <string.h>
#include <fstream>

using namespace std;

const short MAX_SIZE = 257;

struct Data {
    char key[MAX_SIZE] = "";
    unsigned long long value = 0;
};

struct Node {
    Data* data;
    Node** child; // массив указателей на детей
    int n; // количество элементов
    bool leaf; // является ли листом
    int t; // характеристическое число

    Node(int, bool);
    Node* Search(char*); // поиск ноды с нужным ключем
    void SplitChild(int, Node*); // если ребенок переполнен, разделяем его
    void InsertNonFull(Data); // вставка
    void Remove(char*);
    void RemoveFromLeaf(int);
    void RemoveFromNonLeaf(int);
    void BorrowFromNext(int); // если у следующей ноды >= t, то крадем у
неё один элемент
    void BorrowFromPrev(int); // если у предыдущей ноды >= t, то крадем у
неё один элемент
    void FillNode(int); // увеличивает количество элементов в ноде
    void Merge(int); // объединяет текущую (по переданному индексу) и
```

```
следующую ноду
    void Save(ofstream&);
    void Destroy();
};

Node::Node(int _t, bool is_leaf) {
    t = _t;
    leaf = is_leaf;
    data = new Data[2 * t - 1];
    child = new Node*[2 * t];
    for (int i = 0; i < 2 * t; ++i) {
        child[i] = nullptr;
    }
    n = 0;
}

Node* Node::Search(char* key) {
    int i = 0;
    while (i < n && strcmp(key, data[i].key) > 0) {
        i++;
    }

    if (i < n && strcmp(key, data[i].key) == 0) {
        return this;
    }

    if (leaf == true) {
        return nullptr;
    }
    return child[i]->Search(key);
}

void Node::SplitChild(int index, Node* full_node) {
    // index это индекс, в который попадет ключ из дочернего узла
    // full_node это дочерний узел, который мы разделяем

    // создаем новый узел
    Node* new_node = new Node(full_node->t, full_node->leaf);
    new_node->n = t - 1;
    // копируем вторую половину старого узла в новый
    for (int i = 0; i < t - 1; ++i) {
        new_node->data[i] = full_node->data[i + t];
    }
```

```
    // если есть дети, то их тоже перепривязываем
    if (full_node->leaf == false) {
        for (int i = 0; i < t; ++i) {
            new_node->child[i] = full_node->child[i+t];
        }
    }

    full_node->n = t - 1;

    for (int i = n; i >= index + 1; --i) {
        child[i + 1] = child[i];
    }
    // теперь new_node стал правым ребенком, а full_node стал левым
    child[index + 1] = new_node;
    // сдвигаю ключи вправо, чтобы поставить новый ключ
    for (int i = n - 1; i >= index; --i) {
        data[i + 1] = data[i];
    }
    // ставим в родительский узел середину дочернего узла
    data[index] = full_node->data[t - 1];
    ++n;
}

void Node::InsertNonFull(Data inserted_elem) {
    int i = n - 1;

    if (leaf == true) {
        while (i >= 0 && strcmp(data[i].key, inserted_elem.key) > 0) {
            data[i+1] = data[i];
            --i;
        }
        data[i + 1] = inserted_elem;
        ++n;
    } else {
        while (i >= 0 && strcmp(data[i].key, inserted_elem.key) > 0) {
            --i;
        }
        // если ребенок заполнен
        if (child[i+1]->n == 2*t-1) {
            this->SplitChild(i + 1, child[i+1]);

            if (strcmp(inserted_elem.key, data[i+1].key) > 0) {
                ++i;
```

```
            }
        }
        child[i+1]->InsertNonFull(inserted_elem);
    }
}

void Node::Remove(char* key) {
    int idx = 0;
    while (idx < n && strcmp(key, data[idx].key) > 0) {
        ++idx;
    }

    if (idx < n && strcmp(key, data[idx].key) == 0) {
        if (leaf) {
            RemoveFromLeaf(idx);
        } else {
            RemoveFromNonLeaf(idx);
        }
    } else {
        if (child[idx]->n < t) {
            FillNode(idx);
        }

        if (idx > n) {
            child[idx-1]->Remove(key);
        } else {
            child[idx]->Remove(key);
        }
    }
}

void Node::RemoveFromLeaf(int index) {
    for (int i = index+1; i < n; i++) {
        data[i-1] = data[i];
    }
    --n;
}

void Node::RemoveFromNonLeaf(int index) {
    char key[MAX_SIZE];
    strcpy(key, data[index].key);
    Data new_parent;
    // при удалении нельзя допустить t-2 элемента
```

```
    // поэтому удаляем только тогда, когда есть как минимум
    // t элементов
    if (child[index]->n >= t) {
        // нахожу максимальный ключ в левом поддереве
        Node* max_node = child[index];
        while (!max_node->leaf) {

            max_node = max_node->child[max_node->n];
        }
        new_parent = max_node->data[max_node->n-1];
        // найденный ключ ставлю на место того ключа, который удаляю
        data[index] = new_parent;
        child[index]->Remove(new_parent.key);
    } else if (child[index+1]->n >= t) {
        // нахожу минимальный ключ в правом поддереве
        Node* min_node = child[index+1];
        while(!min_node->leaf) {
            min_node = min_node->child[0];
        }
        new_parent = min_node->data[0];
        // найденный ключ ставлю на место того ключа, который удаляю
        data[index] = new_parent;
        child[index+1]->Remove(new_parent.key);
    } else {
        // если у обоих детей по t-1 элеметов, то их нужно объединить
        // и из объединенного узла удалить ключ
        Merge(index);
        child[index]->Remove(key);
    }
}

void Node::FillNode(int index) {
    if (index != 0 && child[index-1]->n >= t) {
        BorrowFromPrev(index);
    } else if (index != n && child[index+1]->n >= t) {
        BorrowFromNext(index);
    } else {
        if (index == n) {
            Merge(index-1);
        } else {
            Merge(index);
        }
    }
```

```
}

void Node::BorrowFromNext(int index) {
    Node* current = child[index];
    Node* next = child[index+1];

    // спустили разделяющий ключ
    current->data[current->n] = data[index];
    data[index] = next->data[0];
    // т.к. мы поставили на место родительского ключа, минимальный
    // из правого брата, то мы должны правильно перепривязать их детей
    // здесь мы говорим, что самый левый сын правого брата, становится
    // самым правым сыном левого брата
    if (!current->leaf) {
        current->child[(current->n) + 1] = next->child[0];
    }

    // затерли минимальный из правого брата
    for (int i = 0; i < next->n - 1; ++i) {
        next->data[i] = next->data[i+1];
    }
    // если у него были дети, то тоже сдвигаем
    if (!next->leaf) {
        for (int i = 0; i < next->n; ++i) {
            next->child[i] = next->child[i+1];
        }
    }
    // обновляем количество элементов
    current->n += 1;
    next->n -= 1;
}

void Node::BorrowFromPrev(int index) {
    Node* current = child[index];
    Node* prev = child[index-1];

    for (int i = current->n; i >= 1; --i) {
        current->data[i] = current->data[i-1];
    }
    // я нахожусь в правом ребенке, поэтому
    // разделяющий ключ лежит в data[index-1]
    current->data[0] = data[index-1];
    // передвигаю детей
```

```
    if (!current->leaf) {
        for (int i = current->n + 1; i >= 1; --i) {
            current->child[i] = current->child[i-1];
        }
        // самым левым сыном правого поддерева стал самый правый из левого
        current->child[0] = prev->child[prev->n];
    }
    // ставлю на место разделительного ключа максимум
    // из левого поддерева
    data[index-1] = prev->data[prev->n - 1];

    current->n += 1;
    prev->n -= 1;
}

// Объединяет текущего и следующего ребенка
void Node::Merge(int index) {
    Node* left_child = child[index];
    Node* right_child = child[index+1];

    // спускаем родителя вниз
    left_child->data[t-1] = data[index];
    // копируем значения
    for (int i = 0; i < t - 1; ++i) {
        left_child->data[i+t] = right_child->data[i];
    }
    // копируем детей
    if (!left_child->leaf) {
        for (int i = 0; i < t; ++i) {
            left_child->child[i + t] = right_child->child[i];
        }
    }
    // так как мы спустили родителя, то смещаем все ключи родительского узла
влево
    for (int i = index + 1; i < n; ++i) {
        data[i-1] = data[i];
    }

    for (int i = index + 2; i <= n; ++i) {
        child[i-1] = child[i];
    }

    left_child->n = 2*t - 1;
```

```
    --n;
    delete[] right_child->data;
    delete[] right_child->child;
    delete right_child;
}

void Node::Destroy() {
    delete[] data;
    if (child[0] == nullptr) {
        delete[] child;
        return;
    }
    for (int i = 0; i <= n; ++i) {
        if (child[i] != nullptr) {
            child[i]->Destroy();
        }
        delete child[i];
    }
    delete[] child;
}

class BTree {
    Node* root;
    int t;
public:
    BTree(int);
    void AddNode(Data);
    void AddOnLoad(Data);
    void DeleteNode(char*);
    void Search(char*);
    void SaveToFile(char*);
    void LoadFromFile(char*);
    ~BTree();
};

BTree::BTree(int _t = 2) {
    root = nullptr;
    t = _t;
}

BTree::~BTree() {
    if (root != nullptr) {
        root->Destroy();
```

```
    }
    delete root;
}

void BTree::AddNode(Data elem) {
    if (root == nullptr) {
        root = new Node(t, true);
        root->data[0] = elem;
        root->n = 1;
        cout << "OK\n";
        return;
    }
    // ключ уже есть
    if (root->Search(elem.key) != nullptr) {
        cout << "Exist\n";
        return;
    }
    // надо сплитануть корень
    if (root->n == 2*t-1) {
        Node* new_root = new Node(t, false);
        new_root->child[0] = root;
        new_root->SplitChild(0, root);
        // отвечает за то, в какое поддерево нужно вставлять ключ
        int i = 0;
        if (strcmp(elem.key, new_root->data[0].key) > 0) {
            ++i;
        }
        new_root->InsertNonFull(elem);
        root = new_root;
    } else {
        root->InsertNonFull(elem);
    }
    cout << "OK\n";
}

void BTree::Search(char* key) {
    if (root == nullptr) {
        cout << "NoSuchWord\n";
        return;
    }

    Node* result = root->Search(key);
    if (result == nullptr) {
```

```
        cout << "NoSuchWord\n";
    } else {
        int i = 0;
        while (i < result->n && strcmp(key, result->data[i].key) > 0) {
            i++;
        }
        cout << "OK: " << result->data[i].value << "\n";
    }
}

void BTree::DeleteNode(char* key) {
    if (root == nullptr) {
        cout << "NoSuchWord\n";
        return;
    }

    if (root->Search(key) == nullptr) {
        cout << "NoSuchWord\n";
        return;
    }

    root->Remove(key);
    // если после удаления корень пуст
    if (root->n == 0) {
        Node* old_root = root;
        if (root->leaf) {
            root = nullptr;
        } else {
            root = root->child[0];
        }
        delete[] old_root->data;
        delete[] old_root->child;
        delete old_root;
    }
    cout << "OK\n";

}

void Node::Save(ofstream &out) {
    if (child[0] == nullptr) {
        for (int i = 0; i < n; ++i) {
            out.write(data[i].key, sizeof(char) * (strlen(data[i].key) +
1));
```

```
            out.write((char*)&data[i].value, sizeof(long long int));
        }
    } else {
        for (int i = 0; i < n; ++i) {
            child[i]->Save(out);
            out.write(data[i].key, sizeof(char) * (strlen(data[i].key) +
1));
            out.write((char*)&data[i].value, sizeof(long long int));
        }
        child[n]->Save(out);
    }
}

void BTree::SaveToFile(char* path) {
    ofstream out(path, ios::binary);
    short is_tree = 1;
    if (root == nullptr) {
        is_tree = 0;
        out.write((char*)&is_tree, sizeof(short));
        return;
    }

    out.write((char*)&is_tree, sizeof(short));
    root->Save(out);
    char end_token = '$';
    out.write((char*)&end_token, sizeof(char));
    out.close();
}

void BTree::LoadFromFile(char* path) {
    if (root != nullptr) {
        root->Destroy();
        delete root;
        root = nullptr;
    }

    ifstream in(path, ios::binary);
    short is_tree;
    in.read((char*)&is_tree, sizeof(short));
    char c;
    if (is_tree) {
        while (true) {
            in.read((char*)&c, sizeof(char));
```

```
            if (c == '$') {
                break;
            }
            Data inserted_elem;
            inserted_elem.key[0] = c;
            for (int i = 1; c != '\0'; ++i) {
                in.read((char*)&c, sizeof(char));
                inserted_elem.key[i] = c;
            }
            in.read((char*)&inserted_elem.value, sizeof(unsigned long
long));
            this->AddOnLoad(inserted_elem);
        }
    }
    in.close();
}

void BTree::AddOnLoad(Data elem) {
    if (root == nullptr) {
        root = new Node(t, true);
        root->data[0] = elem;
        root->n = 1;
        return;
    }
    // надо сплитануть корень
    if (root->n == 2*t-1) {
        Node* new_root = new Node(t, false);
        new_root->child[0] = root;
        new_root->SplitChild(0, root);
        // отвечает за то, в какое поддерево нужно вставлять ключ
        int i = 0;
        if (strcmp(elem.key, new_root->data[0].key) > 0) {
            ++i;
        }
        new_root->InsertNonFull(elem);
        root = new_root;
    } else {
        root->InsertNonFull(elem);
    }
}

void TolowerString(char* str) {
    int len = strlen(str);
```

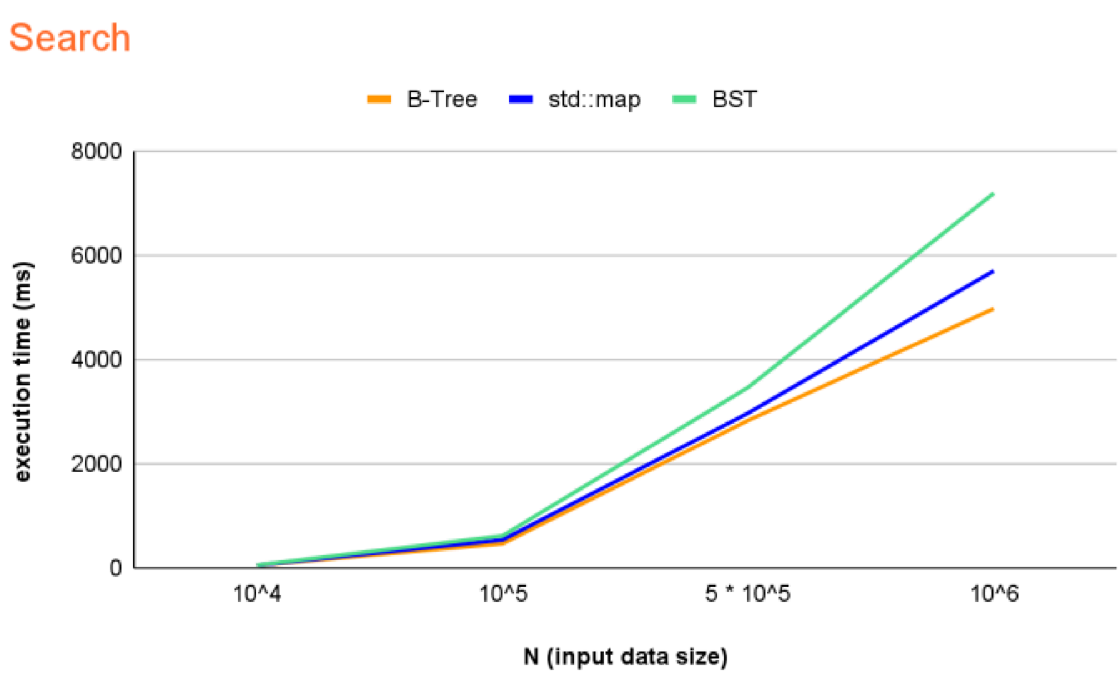
```
    for (int i = 0; i < len; ++i) {
        str[i] = tolower(str[i]);
    }
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);

    BTree Tree(3);
    Data data;
    char path[MAX_SIZE];
    char buffer[MAX_SIZE];
    while(cin >> buffer) {
        switch (buffer[0]) {
            case '+':
                cin >> data.key >> data.value;
                TolowerString(data.key);
                Tree.AddNode(data);
                break;
            case '-':
                cin >> buffer;
                TolowerString(buffer);
                Tree.DeleteNode(buffer);
                break;
            case '!':
                cin >> buffer >> path;
                if (!strcmp(buffer, "Save")) {
                    Tree.SaveToFile(path);
                } else {
                    Tree.LoadFromFile(path);
                }
                cout << "OK\n";
                break;
            default:
                TolowerString(buffer);
                Tree.Search(buffer);
                break;
        }
    }
    return 0;
}
```

5. Демонстрация работы программы

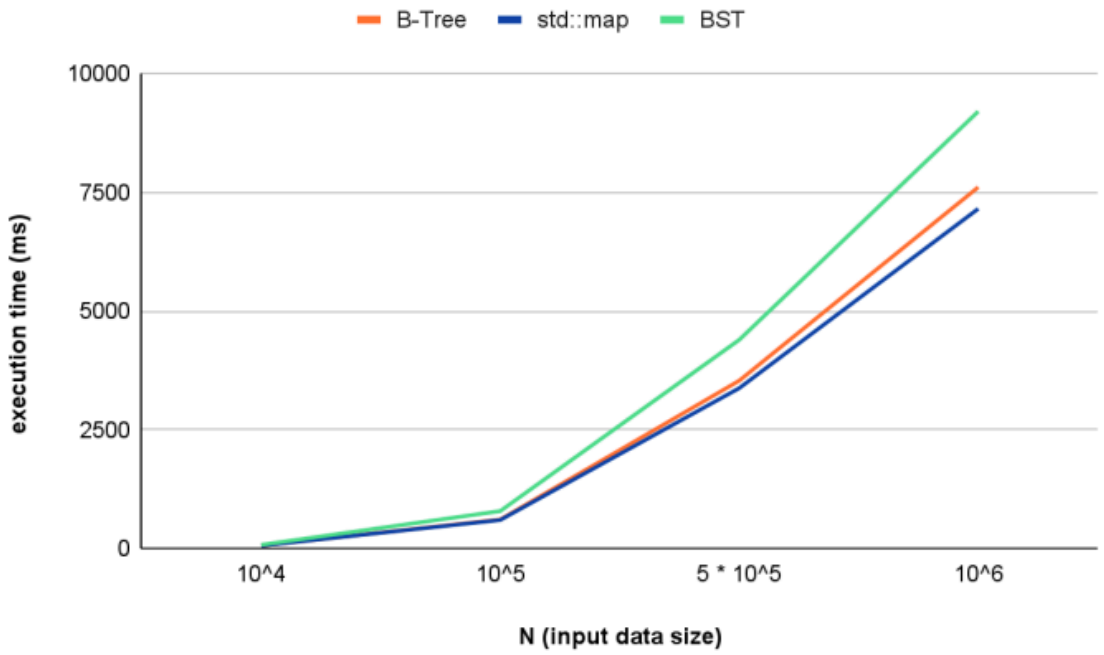
```
qwz@qwz-VirtualBox:~/DA2-3$ g++ b-tree.cpp
qwz@qwz-VirtualBox:~/DA2-3$ cat test.txt
+ a 1
+ A 2
+ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa 18446
744073709551615
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
A
- A
a
qwz@qwz-VirtualBox:~/DA2-3$ ./a.out<test.txt
OK
Exist
OK
OK: 18446744073709551615
OK: 1
OK
NoSuchWord
```

6. Тест производительности

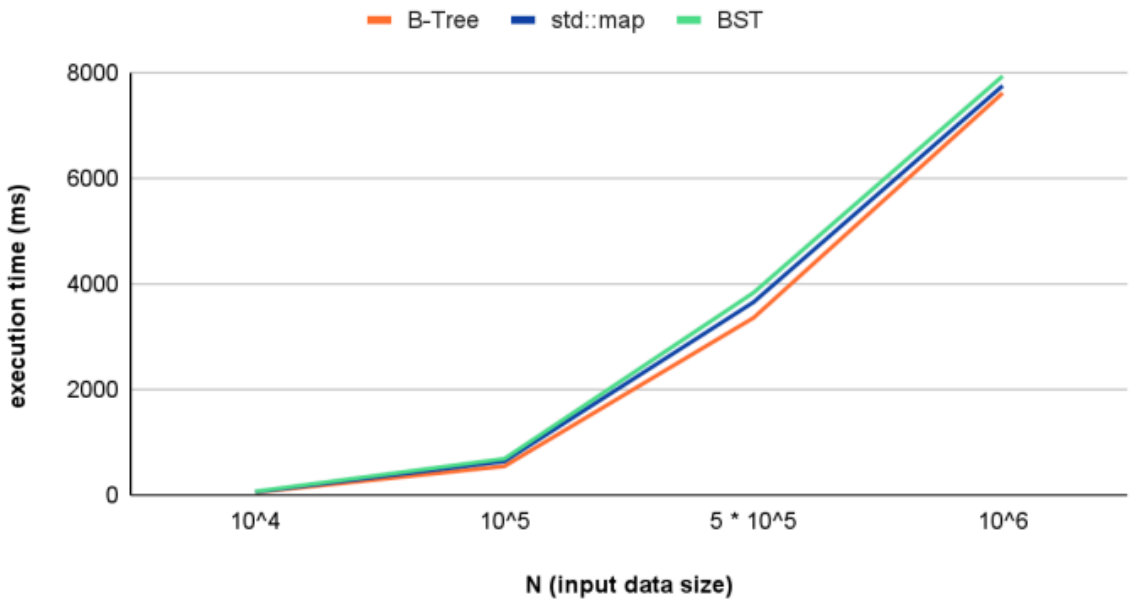
Тест производительности представляет из себя следующее: сравним реализацию словаря с помощью B-Tree с std::map и обычным бинарным деревом поиска. Будем сравнивать операции вставки, удаления и поиска.



Insert



Remove



На графиках видно, что B-Tree иногда обгоняет по скорости std::map. Std::map находится стабильно посерединке, а binary search tree показывает самый худший результат.

7. Выводы

Проделав лабораторную работу, я научился реализовывать такую структуру данных, как B-Tree. Эта структура очень полезна и используется на практике в областях, где нужно хранить большое количество данных, поэтому теперь я больше знаю об устройстве баз данных. Интересной особенностью B-Tree является его характеристическое число t. Очевидно, увеличивая t (минимальную степень), мы увеличиваем ветвление нашего дерева, а следовательно уменьшаем высоту. Обычно это число находится в пределах от 50 до 2000. Пусть у нас есть миллиард ключей, и t=1001. Тогда нам потребуется всего лишь 3 дисковые операции для поиска любого ключа! При этом учитываем, что корень мы можем хранить постоянно. Из-за специфики дерева, пришлось работать с массивом указателей на дочернии узлы. Поэтому возникали баги с неправильными сдвигами детей, выходами за границы массивов, а также ошибки с очищением памяти. Решить эти проблемы мне помогла утилита valgrind. Благодаря лабораторной работе, я научился её использовать для дебага своего кода. Также я провел тест производительности словаря на основе B-tree, сравнив его с std::map и binary search tree.