Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики
«Кафедра вычислительной математики и программирования»

Лабораторная работа №2
по предмету "Дискретный анализ"

Студент: Елистратова П.А.
Преподаватель: Макаров Н.К.
Группа: М8О-210Б-21
Дата:
Оценка:
Подпись:

Москва 2023 г.

Оглавление

Цель работы

Реализовать структуру данных: B-дерево для выполнения поставленной задачи на Си++.

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 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».

! 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).

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 $k_1,...,k_n$, имеет n+1 сына. i-й сын содержит ключи из отрезка $[k_i−1;k_i]$, $k_0=−\infty$, $k_{n+1}=\infty$.
- Ключи в каждом узле упорядочены по возрастанию.
- Все листья находятся на одном уровне.

В своей реализации я выбрала t = 5.

Реализация

Для решения поставленной задачи была реализована структура `InputData` для входных данных и два класса `BTreeNode` — узел нашего B-дерева и класс самого дерева — `Btree`.

Функции для работы с `BtreeNode`:

Название	Что делает	Временная сложность
BtreeNode(bool leaf);	Конструктор класса. Принимает bool, которое говорит является ли создаваемый узел листом. Выделяет память под пары «ключ-значение», выделяет память под детей и присваивает им значение нулевого указателя.	O(1)
BTreeNode* search(char* key, unsigned long long &number);	Рекурсивный метод поиска ключа в дереве.	$O(t*\log_t(n))$
void InsertNonFull(InputData &pair);	Вставляет переданное значение в незаполненный узел.	$O(t*\log_t(n))$
void SplitChild(int i, BtreeNode* full_node);	Разделяет заполненный узел.	O(t)
void deletion(InputData &pair);	Рекурсивный метод удаления ключа из дерева.	$O(t*\log_t(n))$

void RemoveFromLeaf(int idx);	Удаление элемента из листа дерева.	O(t)
void RemoveFromNonLeaf(int idx);	Удаление элемента из внутреннего узла дерева.	$O(t*\log_t(n))$
InputData GetPredecessor(int idx);	Получение предшественника элемента с индексом idx в текущем узле.	$O(\log_t(n))$
InputData GetSuccessor(int idx);	Получение следующего элемента по величине за элементом с индексом idx в текущем узле.	$O(\log_t(n))$
void FillNode(int idx);	Наполняет узел, содержащий t-1 ключ.	O(t)
void BorrowFromPrev(int idx);	Берёт элемент из левого брата.	O(t)
void BorrowFromNext(int idx);	Берёт элемент из правого брата.	O(t)
void merge(int idx);	Объединяет два ребёнка и разделяющий их ключ.	O(t)
void save(std::ofstream &file);	Рекурсивный метод сохранения дерева. Сначала сохраняет детей, потом родителей.	O(n)
void DeleteNode();	Рекурсивный метод удаления дерева.	O(n)

Функции для работы с `Btree`:

Название	Что делает	Временная сложность
BTree();	Конструктор класса.	O(1)
BTreeNode* BTree::search(char* key, unsigned long long &number);	Поиск ключа в дереве.	$O(t*\log_t(n))$
void BTree::insert(InputData &pair)	Добавляет пару «ключ-значение» в дерево.	$O(t*\log_t(n))$
void BTree::deletion(InputData &pair)	Удаляет из дерева переданную пару.	$O(t*\log_t(n))$
void SaveToFile(char* path);	Сохраняет дерево в бинарном представлении.	O(n)
void LoadFromFile(char* path);	Загружает дерево из бинарного представления.	$O(n*t*\log_t(n))$

~BTree();	Деструктор дерева.	O(n)

Код программы

lab2.cpp

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

const unsigned short TREE_DEGREE = 5;
const unsigned short MAX_KEY_LENGTH = 257;
const unsigned short MAX_ARRAY_LENGTH = 2 * TREE_DEGREE - 1;

void StringToLower(char* string_key)
{
    int length = strlen(string_key);
    for (int i = 0; i < length; ++i)
    {
        string_key[i] = tolower(string_key[i]);
    }
}

struct InputData
{
    char string_key[MAX_KEY_LENGTH];
    unsigned long long number;
};

class BTreeNode
{
    InputData* pairs;
    BTreeNode** children;
    int current_length;
    bool is_leaf;

public:
    BTreeNode(bool leaf);

    BTreeNode* search(char* key, unsigned long long &number);

    void InsertNonFull(InputData &pair);
    void SplitChild(int i, BTreeNode* full_node);

    void traverse();

    void deletion(InputData &pair);
    void RemoveFromLeaf(int idx);
    void RemoveFromNonLeaf(int idx);
    InputData GetPredecessor(int idx);
    InputData GetSuccessor(int idx);
    void FillNode(int idx);
    void BorrowFromPrev(int idx);
    void BorrowFromNext(int idx);
    void merge(int idx);

    void save(std::ofstream &file);
    void DeleteNode();

    friend class BTree;
};
```

```
BTreeNode::BTreeNode(bool leaf)
{
    is_leaf = leaf;
    pairs = new InputData[MAX_ARRAY_LENGTH];
    children = new BTreeNode *[MAX_ARRAY_LENGTH + 1];
    for (int i = 0; i < MAX_ARRAY_LENGTH + 1; ++i)
    {
        children[i] = nullptr;
    }
    current_length = 0;
}

BTreeNode* BTreeNode::search(char* key, unsigned long long &number)
{
    int i = 0;

    while ((i < current_length) && (strcmp(pairs[i].string_key, key) < 0))
    {
        ++i;
    }
    if (i < current_length && strcmp(pairs[i].string_key, key) == 0)
    {
        number = pairs[i].number;
        return this;
    }
    else if (is_leaf == true)
    {
        return nullptr;
    }

    return children[i]->search(key, number);
}

void BTreeNode::InsertNonFull(InputData &pair)
{
    int i = current_length - 1;
    if (is_leaf == true)
    {
        while (i >= 0 && strcmp(pair.string_key, pairs[i].string_key) < 0)
        {
            pairs[i + 1] = pairs[i];
            --i;
        }

        pairs[i + 1] = pair;
        ++current_length;
    }
    else
    {
        while (i >= 0 && strcmp(pair.string_key, pairs[i].string_key) < 0)
        {
            --i;
        }

        if (children[i + 1]->current_length == 2 * TREE_DEGREE - 1)
        {
            SplitChild(i + 1, children[i + 1]);

            if (strcmp(pairs[i + 1].string_key, pair.string_key) < 0)
            {
                ++i;
            }
        }
        children[i + 1]->InsertNonFull(pair);
```

```
    }
}

void BTreeNode::SplitChild(int i, BTreeNode* full_node)
{
    BTreeNode* new_child_node = new BTreeNode(full_node->is_leaf);
    new_child_node->current_length = TREE_DEGREE - 1;

    for (int j = 0; j < TREE_DEGREE - 1; ++j)
    {
        new_child_node->pairs[j] = full_node->pairs[j + TREE_DEGREE];
    }

    if (full_node->is_leaf == false)
    {
        for (int j = 0; j < TREE_DEGREE; ++j)
        {
            new_child_node->children[j] = full_node->children[j + TREE_DEGREE];
        }

    }

    full_node->current_length = TREE_DEGREE - 1;

    for (int j = current_length; j >= i + 1; --j)
    {
        children[j + 1] = children[j];
    }

    children[i + 1] = new_child_node;

    for (int j = current_length - 1; j >= i; --j)
    {
        pairs[j + 1] = pairs[j];
    }

    pairs[i] = full_node->pairs[TREE_DEGREE - 1];
    current_length = current_length + 1;
}

void BTreeNode::traverse()
{
    int i;
    std::cout << "[ ";
    for (i = 0; i < current_length; ++i)
    {
        if (is_leaf == false)
        {
            children[i]->traverse();
        }

        std::cout << " " << pairs[i].string_key;
    }
    std::cout << " ]";

    if (is_leaf == false)
    {
        children[i]->traverse();
        std::cout << "\n";
    }
}

void BTreeNode::deletion(InputData &pair)
{
    int idx = 0;
```

```
    while (idx < current_length && strcmp(pairs[idx].string_key, pair.string_key) <
0)
    {
        ++idx;
    }

    if (idx < current_length && strcmp(pairs[idx].string_key, pair.string_key) ==
0)
    {
        if (is_leaf)
        {
            RemoveFromLeaf(idx);
        }
        else
        {
            RemoveFromNonLeaf(idx);
        }
    }
    else
    {
        if (is_leaf)
        {
            return;
        }
        if (children[idx]->current_length < TREE_DEGREE)
        {
            FillNode(idx);
        }

        if (idx > current_length)
        {
            children[idx - 1]->deletion(pair);
        }
        else
        {
            children[idx]->deletion(pair);
        }

    }
}

void BTreeNode::RemoveFromLeaf(int idx)
{
    for (int i = idx + 1; i < current_length; ++i)
    {
        pairs[i - 1] = pairs[i];
    }

    --current_length;
}

void BTreeNode::RemoveFromNonLeaf(int idx)
{
    InputData pair;
    pair = pairs[idx];

    if (children[idx]->current_length >= TREE_DEGREE)
    {
        InputData predecessor = GetPredecessor(idx);
        pairs[idx] = predecessor;
        children[idx]->deletion(predecessor);
    }
    else if (children[idx + 1]->current_length >= TREE_DEGREE)
    {
        InputData succecessor = GetSuccessor(idx);
```

```
        pairs[idx] = succecessor;
        children[idx + 1]->deletion(succecessor);
    }
    else
    {
        merge(idx);
        children[idx]->deletion(pair);
    }
}

InputData BTreeNode::GetPredecessor(int idx)
{
    BTreeNode* current_node = children[idx];
    while (!current_node->is_leaf)
    {
        current_node = current_node->children[current_node->current_length];
    }

    return current_node->pairs[current_node->current_length - 1];
}

InputData BTreeNode::GetSuccessor(int idx)
{
    BTreeNode* current_node = children[idx + 1];
    while (!current_node->is_leaf)
    {
        current_node = current_node->children[0];
    }

    return current_node->pairs[0];
}

void BTreeNode::FillNode(int idx)
{
    if (idx != 0 && children[idx - 1]->current_length >= TREE_DEGREE)
    {
        BorrowFromPrev(idx);
    }
    else if (idx != current_length && children[idx + 1]->current_length >=
TREE_DEGREE)
    {
        BorrowFromNext(idx);
    }
    else
    {
        if (idx != current_length)
        {
            merge(idx);
        }
        else
        {
            merge(idx - 1);
        }
    }
}

void BTreeNode::BorrowFromPrev(int idx)
{
    BTreeNode* child = children[idx];
    BTreeNode* sibling = children[idx - 1];

    for (int i = child->current_length; i >= 1; --i)
    {
        child->pairs[i] = child->pairs[i - 1];
    }
```

```
    child->pairs[0] = pairs[idx - 1];

    if (!child->is_leaf)
    {
        for (int i = child->current_length + 1; i >= 1; --i)
        {
            child->children[i] = child->children[i - 1];
        }
        child->children[0] = sibling->children[sibling->current_length];
    }
    pairs[idx - 1] = sibling->pairs[sibling->current_length - 1];

    child->current_length += 1;
    sibling->current_length -= 1;
}

void BTreeNode::BorrowFromNext(int idx)
{
    BTreeNode* child = children[idx];
    BTreeNode* sibling = children[idx + 1];

    child->pairs[child->current_length] = pairs[idx];
    pairs[idx] = sibling->pairs[0];

    if (!(child->is_leaf))
    {
        child->children[child->current_length + 1] = sibling->children[0];
    }

    for (int i = 0; i < sibling->current_length - 1; ++i)
    {
        sibling->pairs[i] = sibling->pairs[i + 1];
    }

    if (!sibling->is_leaf)
    {
        for (int i = 0; i < sibling->current_length; ++i)
        {
            sibling->children[i] = sibling->children[i + 1];
        }
    }

    child->current_length += 1;
    sibling->current_length -= 1;
}

void BTreeNode::merge(int idx)
{
    BTreeNode* child = children[idx];
    BTreeNode* sibling = children[idx + 1];

    child->pairs[TREE_DEGREE - 1] = pairs[idx];

    for (int i = 0; i < TREE_DEGREE - 1; ++i)
    {
        child->pairs[i + TREE_DEGREE] = sibling->pairs[i];
    }

    if (!child->is_leaf)
    {
        for (int i = 0; i < TREE_DEGREE; ++i)
        {
            child->children[i + TREE_DEGREE] = sibling->children[i];
        }
    }
```

```
    for (int i = idx + 1; i < current_length; ++i)
    {
        pairs[i - 1] = pairs[i];
    }

    for (int i = idx + 2; i <= current_length; ++i)
    {
        children[i - 1] = children[i];
    }

    child->current_length = 2 * TREE_DEGREE - 1;
    --current_length;

    delete[] sibling->children;
    delete[] sibling->pairs;
    delete sibling;
}

void BTreeNode::save(std::ofstream &file)
{
    if (children[0] == nullptr)
    {
        for (int i = 0; i < current_length; ++i)
        {
            file.write(pairs[i].string_key, sizeof(char) *
(strlen(pairs[i].string_key) + 1));
            file.write((char*)&pairs[i].number, sizeof(unsigned long long));
        }
    }
    else
    {
        for (int i = 0; i < current_length; ++i)
        {
            children[i]->save(file);
            file.write(pairs[i].string_key, sizeof(char) *
(strlen(pairs[i].string_key) + 1));
            file.write((char*)&pairs[i].number, sizeof(unsigned long long));
        }
        children[current_length]->save(file);
    }
}

void BTreeNode::DeleteNode()
{
    delete[] pairs;
    if (is_leaf)
    {
        delete[] children;
        return;
    }
    for (int i = 0; i <= current_length; ++i)
    {
        if (!is_leaf)
        {
            children[i]->DeleteNode();
        }
        delete children[i];
    }
    delete[] children;
}


class BTree
{
```

```
    BTreeNode* root;

public:
    BTree();
    BTreeNode* search(char* key, unsigned long long &number);
    void insert(InputData &pair);
    void deletion(InputData &pair);
    void SaveToFile(char* path);
    void LoadFromFile(char* path);
    void traverse();
    ~BTree();
};

BTree::BTree()
{
    root = nullptr;
}

BTree::~BTree()
{
    if (root != nullptr)
    {
        root->DeleteNode();
    }
    delete root;
}

BTreeNode* BTree::search(char* key, unsigned long long &number)
{
    if (root == nullptr)
    {
        return nullptr;
    }
    else
    {
        return root->search(key, number);
    }
}

void BTree::insert(InputData &pair)
{
    if (root == nullptr)
    {
        root = new BTreeNode(true);
        root->pairs[0] = pair;
        root->current_length = 1;
    }
    else
    {
        if (root->current_length == MAX_ARRAY_LENGTH)
        {
            BTreeNode* new_root = new BTreeNode(false);
            new_root->children[0] = root;
            new_root->SplitChild(0, root);
            int i = 0;
            if (strcmp(new_root->pairs[0].string_key, pair.string_key) < 0)
            {
                ++i;
            }
            new_root->children[i]->InsertNonFull(pair);
            root = new_root;
        }
        else
            root->InsertNonFull(pair);
    }
```

```
}

void BTree::deletion(InputData &pair)
{
    root->deletion(pair);

    if (root->current_length == 0)
    {
        BTreeNode* old_root = root;
        if (root->is_leaf)
        {
            root = nullptr;
        }
        else
        {
            root = root->children[0];
        }

        delete[] old_root->children;
        delete[] old_root->pairs;
        delete old_root;
    }
}

void BTree::SaveToFile(char* path)
{
    std::ofstream file(path, std::ios::binary);
    short is_tree = 1;
    char end_of_tree = '$';

    if (root == nullptr)
    {
        is_tree = 0;
        file.write((char*)&is_tree, sizeof(short));
        return;
    }

    file.write((char*)&is_tree, sizeof(short));
    root->save(file);

    file.write((char*)&end_of_tree, sizeof(char));
    file.close();
}

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

    std::ifstream file(path, std::ios::binary);
    char symbol;
    short is_tree;
    file.read((char*)&is_tree, sizeof(short));

    if (is_tree)
    {
        while (true)
        {
            file.read((char*)&symbol, sizeof(char));
            if (symbol == '$') {
                break;
```

```
            }
            InputData input_pair;
            input_pair.string_key[0] = symbol;

            for (int i = 1; symbol != '\0'; ++i) {
                file.read((char*)&symbol, sizeof(char));
                input_pair.string_key[i] = symbol;
            }

            file.read((char*)&input_pair.number, sizeof(unsigned long long));
            this->insert(input_pair);
        }
    }
    file.close();
}

void BTree::traverse()
{
    if (root != nullptr)
        root->traverse();
}


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

    BTree t;
    char buff[MAX_KEY_LENGTH];
    InputData input_pair;
    while (std::cin >> buff)
    {
        if (strcmp(buff, "+") == 0)
        {
            std::cin >> input_pair.string_key >> input_pair.number;
            StringToLower(input_pair.string_key);
            if (t.search(input_pair.string_key, input_pair.number) != nullptr)
            {
                std::cout << "Exist\n";
            }
            else
            {
                t.insert(input_pair);
                std::cout << "OK\n";
            }
        }
        else if (strcmp(buff, "-") == 0)
        {
            std::cin >> input_pair.string_key;
            StringToLower(input_pair.string_key);
            if (t.search(input_pair.string_key, input_pair.number) != nullptr)
            {
                t.deletion(input_pair);
                std::cout << "OK\n";
            }
            else
            {
                std::cout << "NoSuchWord\n";
            }
        }
        else if (strcmp(buff, "!") == 0)
        {
            char action[5];
```

```
            std::cin >> action >> buff;
            if (strcmp(action, "Load") == 0)
            {
                t.LoadFromFile(buff);
                std::cout << "OK\n";
            }
            else if ((strcmp(action, "Save") == 0))
            {
                t.SaveToFile(buff);
                std::cout << "OK\n";
            }
        }
        else
        {
            unsigned long long number = 0;
            StringToLower(buff);
            if (t.search(buff, number) != nullptr)
            {
                std::cout << "OK: " << number << "\n";
            }
            else
            {
                std::cout << "NoSuchWord\n";
            }
        }
    }
    return 0;
}
```

Пример работы

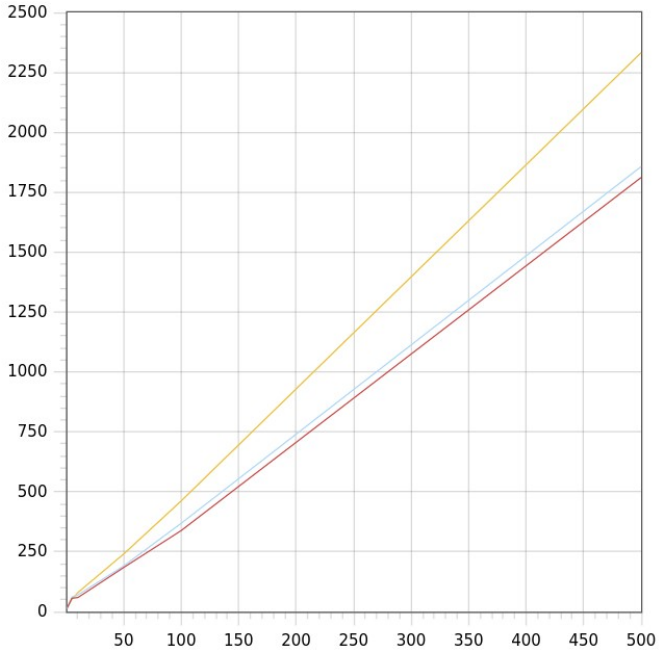
```
polina@pelis:~/diskran/lab2$ cat input.txt
+ a 1
+ A 2
+
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaa 18446744073709551615
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaa
A
- A
a
polina@pelis:~/diskran/lab2$ g++ lab2.cpp
polina@pelis:~/diskran/lab2$ ./a.out < input.txt
OK
Exist
OK
OK: 18446744073709551615
OK: 1
OK
NoSuchWord
```

Сравнения производительности

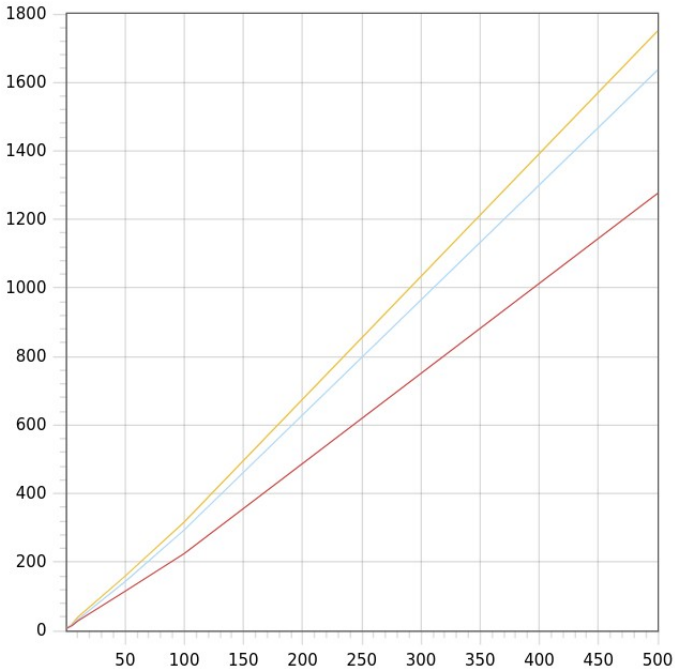
Построим графики зависимости времени выполнения(мс) от входных данных (*1000) и сравним операции вставки, поиска и удаления значений в нашем B-

дереве (красный график) с std::map (голубой график) и бинарным деревом поиска (жёлтый график).

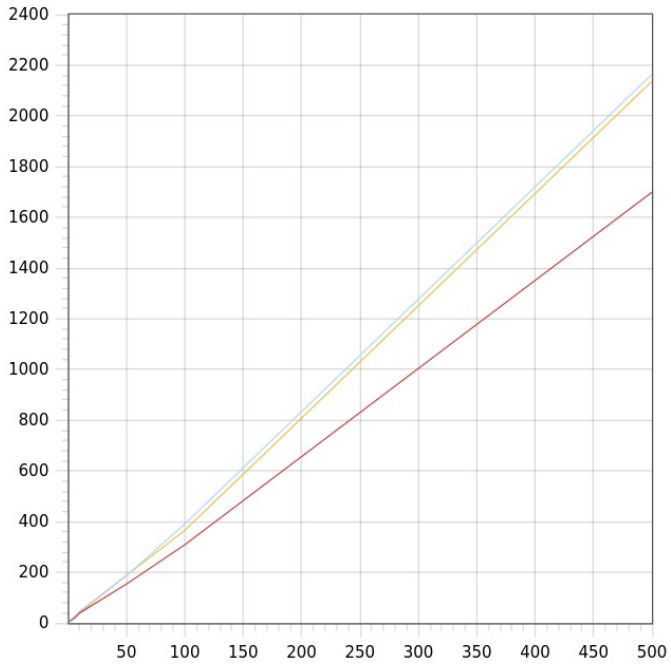
Вставка:



Поиск:



Удаление:



На графиках видно, что B-дерево работает быстрее всего. Бинарное дерево поиска показывает худший результат, но на операции удаления работает примерно одинаково по времени с std::map.

Вывод

Выполнив данную лабораторную работу, я научилась реализовывать на C++ такую полезную структуру как B-дерево. B-деревья представляют собой сбалансированные деревья поиска, созданные специально для эффективной работы с дисковой памятью (и другими типами вторичной памяти с непосредственным доступом). Многие СУБД используют для хранения информации именно B-деревья (или их разновидности).
Также, для того чтобы код работал правильно, приходилось следить за утечками памяти в процессе работы, с этим мне помогла утилита valgrind. Благодаря лабораторной работе, я научилась использовать её для дебага своего кода.
Также я провела тест производительности словаря на основе B-дерева, сравнив его с std::map и бинарным деревом поиска.