**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:

Оглавление

[Цель работы 3](#__RefHeading___Toc3871_94863195)

[Постановка задачи 3](#__RefHeading___Toc3873_94863195)

[Описание 3](#__RefHeading___Toc1694_1904022149)

[Реализация 4](#__RefHeading___Toc1694_19040221493)

[Код программы 6](#__RefHeading___Toc1694_190402214931)

[Вывод 18](#__RefHeading___Toc422_3869341128)

# **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 k1,...,kn, имеет n+1 сына. i-й сын содержит ключи из отрезка [ki−1;ki], k0=−∞, kn+1=∞.
* 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\*logt(n)) |
| void InsertNonFull(InputData &pair); | Вставляет переданное значение в незаполненный узел. | O(t\*logt(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\*logt(n)) |
| void RemoveFromLeaf(int idx); | Удаление элемента из листа дерева. | O(t) |
| void RemoveFromNonLeaf(int idx); | Удаление элемента из внутреннего узла дерева. | O(t\*logt(n)) |
| InputData GetPredecessor(int idx); | Получение предшественника элемента с индексом idx в текущем узле. | O(logt(n)) |
| InputData GetSuccessor(int idx); | Получение следующего элемента по величине за элементом с индексом idx в текущем узле. | O(logt(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. | О(1) |
| BTreeNode\* BTree::search(char\* key, unsigned long long &number); | Поиск ключа в дереве. | O(t\*logt(n)) |
| void BTree::insert(InputData &pair) | Добавляет пару «ключ-значение» в дерево. | O(t\*logt(n)) |
| void BTree::deletion(InputData &pair) | Удаляет из дерева переданную пару. | O(t\*logt(n)) |
| void SaveToFile(char\* path); | Сохраняет дерево в бинарном представлении. | O(n) |
| void LoadFromFile(char\* path); | Загружает дерево из бинарного представления. | O(n\*t\*logt(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

+ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa 18446744073709551615

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

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

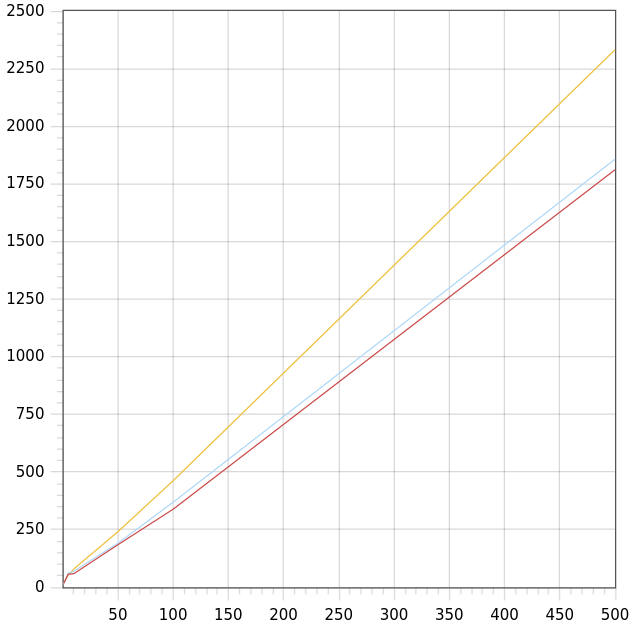
OK

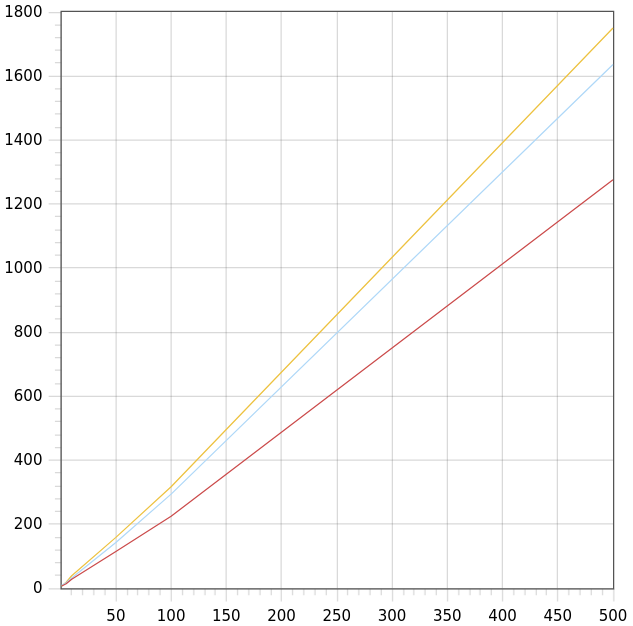
NoSuchWord

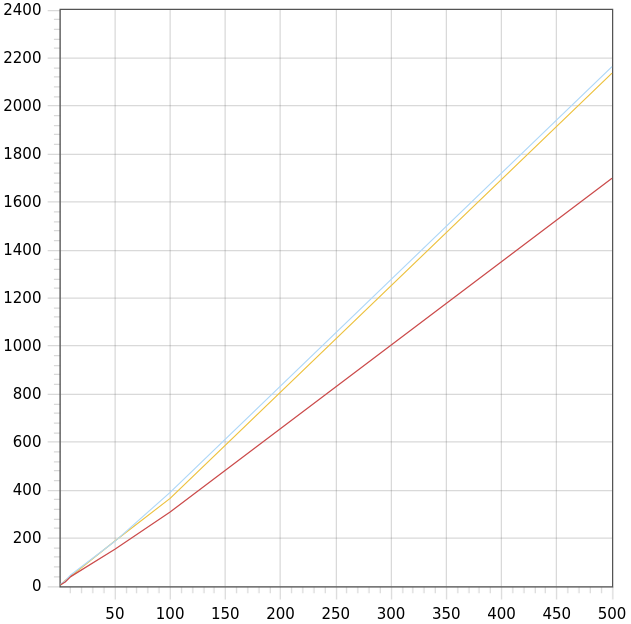
**Сравнения производительности**

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 лаборатоной 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.