Лабораторная работа № 2 по курсу дискретного анализа: сбалансированные деревья

Выполнил студент группы М8О-208Б-21 МАИ *Белоносов Кирилл.*

Условие

1. Общая 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.

2. Вариант задания(4):

   Дерево: B-дерево.

Метод решения

B-дерево - это сбалансированное дерево поиска, созданное специально для эффективной работы с дисковой памяти. Каждый узел такого дерева содержит множество ключей, а так имеет более двух потомков. B-дерево с параметром t, где t - минимальная степень B-дерева, обладает следующими свойствами:

1. Все листья расположены на одной и той же глубине
2. Каждый узел кроме корневого должен содержать как минимум t - 1 ключей
3. Каждый узел кроме корневого должен содержать минимум t дочерних узлов
4. Каждый узел содержит не более 2t - 1 ключей
5. Каждый узел имеет не более 2t дочерних узлов
6. Узел заполнен если он содержит ровно 2t - 1

Рассмотрим 3 основных операции связанных с B-деревом: Поиск, удаление и вставка
Поиск:
Для поиска ключа в дереве будем выполнять рекурсивные вызовы поиска, начиная с корня. Будем выполнять линейный поиск по всем ключам, принадлежащем данному узлу, так чтобы найти наименьший индекс i, такой что $k \leq key_i[x]$. Если мы нашли нужный ключ, в данном узле, то мы его возравщаем. Иначе, если же мы находимся

в листе, то завершаем с неудачей. Иначе же рекурсивно вызывем поиск от дочернего узла.

Вставка:

Вставлять новый элемент мы можем только в незаполненный лист, для этого, используя поиск, находим узел в который нам нужно вставить новый элемент. Возникает 2 ситуации: узел заполнен и не заполнен. В первом случае мы применяем процедуру разбиения узла на 2, причем среднее значение узла переходит в родительский. Если же родительский оказался тоже заполненым, что продолжаем разбивать узлы по рекурсии. Во втором же случае нам достаточно вставить наш элемент в нужную позицию.

Удаление:

Удаление является более сложной операцией, потому что удаление может производиться не только из листа, но и из внутренних узлов, что требует соблюдения инварианта нашего дерева и определенной его перестройки. У нас может возникнуть 2 кейса:

1. Если мы удаляем из листа и количество ключей в нем больше, чем t - 1, то просто удаляем ключ. Если же ключей t - 1, то заимствуем у соседа, и в сам узел спускаем разделитель нашего узла и узла соседа. Если нет соседей с количеством ключей больше чем t - 1, то просто сливаем 2 узла и их разделитель

2. Если же мы удаляем из внутреннего узла, то мы ищем дочерний узел, который предшествует нашему ключу, если дочерний узел содержит t - 1 переменную, то свапаем местами наш ключ и предшественника. В противном случаем мерджим наших потомков и удаляем ключ.

Описание программы

Для более удобного выполнения данного задания, было принятно решение реализовать свою имплементацию вектора и строки.

```
template<class T>
class TVector {
public:
    TVector();
    TVector(size_t size);
    TVector(size_t size, T value);
    TVector(const TVector& arg);
    TVector<T>& operator=(const TVector<T>& arg);
    T& operator [](size_t  index);
    const T& operator [](size_t index) const;
    inline size_t Size() const;
    void Reserve(size_t newmem);
    void Resize(size_t newsize);
    void PushBack(const T& arg);
    ~TVector();
```

```
private:
    T* arr;
    size_t size;
    size_t mem;

};
```

Реализация строки потребовала сделать перегрузку основных операторов сравнения, что было необходимо для создания B-дерева

```
class TString {
private:
    char* str;
    size_t size;
    friend TString operator+(const TString& lhs, const TString& rhs);
    friend std::ostream& operator<<(std::ostream& os, const TString&
       obj);
    friend std::istream& operator>>(std::istream& in, TString& obj);
    friend bool operator<(const TString& lhs, const TString& rhs);
    friend bool operator>(const TString& lhs, const TString& rhs);
    friend bool operator==(const TString& lhs, const TString& rhs);
    friend bool operator!=(const TString& lhs, const TString& rhs);
public:
    TString();
    TString(const char* cStr);
    TString(const TString& rhs);
    TString(TString&& rhs);
    ~TString();
    char& operator[](size_t index);
    const char& operator[](size_t index) const;
    TString& operator=(const TString& rhs);
    size_t Size() const;
    TString ToLower();
    const char* CStr() const;
};
```

Далее я реализовал само B-дерево. Сначала я определил его структуру узлов, использовал объявленные мною ранее типы данных. Также создам класс TBTree для более комфортной работы с самим деревом.

```
struct TBTreeNode {
private:
    TVector<TString> keys;
    TVector<unsigned long long> dataNode;
    TVector<TBTreeNode*> childs;
    int t;
    int keyCount;
```

```
    bool leaf;
public:
    TBTreeNode(int t, bool leaf);
    void Print();
    void Insert(TString& key, unsigned long long data);
    void Split(int i, TBTreeNode* node);
    bool IsFull();
    int FindKey(TString& key);
    void Delete(TString& key);
    void DeleteFromLeaf(int i);
    void DeleteNotLeaf(int i);
    TString GetPrev(int i);
    TString GetNext(int i);
    unsigned long long GetPrevData(int i);
    unsigned long long GetNextData(int i);
    void Fill(int i);
    void FromPrev(int i);
    void FromNext(int i);
    void Merge(int i);
    TBTreeNode* Search(TString& key);
    void WriteToFile(ofstream& File);
    friend class TBTree;
};

class TBTree {
private:
    TBTreeNode* root;
    int t;
public:
    TBTree(int t);
    void Print();
    bool Insert(TString& key, unsigned long long data);
    bool Delete(TString& key);
    TBTreeNode* Search(TString& key);
    bool Find(TString& key);
    unsigned long long operator [](TString& key);
    void ReadFromFile(ifstream& File);
    void WriteToFile(ofstream& File);
    void DeleteTree();
};
```

Теперь реализуем основные операции для работы с B-деревом.

```
void TBTreeNode::Split(int i, TBTreeNode* y) {
```

```
    TBTreeNode* z = new TBTreeNode(y->t, y->leaf);
    z->keyCount = t - 1;

    for(int j = 0; j < t - 1; ++j) {
        z->keys[j] = y->keys[j + t];
        z->dataNode[j] = y->dataNode[j + t];
    }

    if(y->leaf == false) {
        for(int j = 0; j < t; ++j) {
            z->childs[j] = y->childs[j + t];
        }
    }

    y->keyCount = t - 1;
    for(int j = keyCount; j >= i + 1; j--)
        childs[j + 1] = childs[j];
    childs[i + 1] = z;

    for(int j = keyCount - 1; j >= i; --j) {
        keys[j + 1] = keys[j];
        dataNode[j + 1] = dataNode[j];
    }

    keys[i] = y->keys[t - 1];
    dataNode[i] = y->dataNode[t - 1];
    keyCount = keyCount + 1;
}

void TBTreeNode::Insert(TString& key, unsigned long long data) {
    int i = keyCount - 1;

    if(leaf == true) {
        while(i >= 0 && keys[i] > key) {
            keys[i + 1] = keys[i];
            dataNode[i + 1] = dataNode[i];
            --i;
        }
        keys[i + 1] = key;
        dataNode[i + 1] = data;
        ++keyCount;
    } else {
        while(i >= 0 && keys[i] > key) {
            --i;
        }
```

```
        ++i;
        if(childs[i]->IsFull()) {
            Split(i, childs[i]);
            if(keys[i] < key) {
                ++i;
            }
        }
        childs[i]->Insert(key, data);
    }

void TBTreeNode::Delete(TString& key) {
    int i = FindKey(key);

    if(i < keyCount && keys[i] == key) {
        if(leaf) {
            DeleteFromLeaf(i);
        } else {
            DeleteNotLeaf(i);
        }
    } else {
        if(leaf) {
            return;
        }
        bool flag = ((i == keyCount) ? true : false);

        if(childs[i]->keyCount < t) {
            Fill(i);
        }

        if(flag && i > keyCount) {
            childs[i - 1]->Delete(key);
        } else {
            childs[i]->Delete(key);
        }
    }
}

void TBTreeNode::DeleteFromLeaf(int i) {
    for(int j = i + 1; j < keyCount; ++j) {
        keys[j - 1] = keys[j];
        dataNode[j - 1] = dataNode[j];
    }

    --keyCount;
}
```

```
void TBTreeNode::DeleteNotLeaf(int i) {
    TString key = keys[i];
    if(childs[i]->keyCount >= t) {
        TString prev = GetPrev(i);
        unsigned long long prevData = GetPrevData(i);
        keys[i] = prev;
        dataNode[i] = prevData;
        childs[i]->Delete(prev);
    } else if(childs[i + 1]->keyCount >= t) {
        TString next = GetNext(i);
        unsigned long long nextData = GetNextData(i);
        keys[i] = next;
        dataNode[i] = nextData;
        childs[i + 1]->Delete(next);
    } else {
        Merge(i);
        childs[i]->Delete(key);
    }
}

TBTreeNode* TBTreeNode::Search(TString& key) {
    int i = 0;
    while(i < keyCount && key > keys[i]) {
        ++i;
    }
    if(i < keyCount && keys[i] == key)
        return this;
    if(leaf == true)
        return nullptr;
    return childs[i]->Search(key);
}
```

Дневник отладки

Попытка 1-9 (ID 86496256)

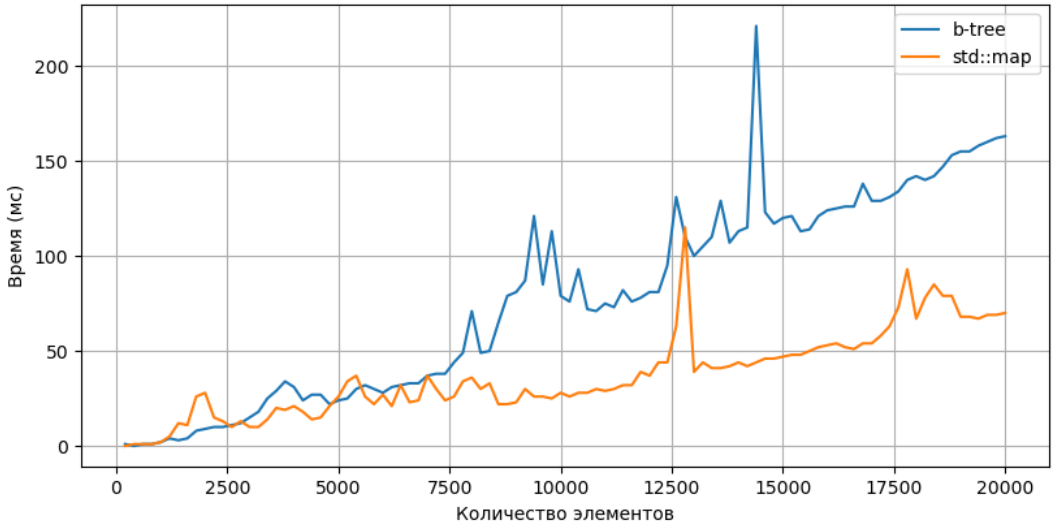
Все попытки были связанны с некорректной B-дерева, что приводило к выходу за границу вектора. Более детальное изучение дерева и написание тестов помогли локализовать проблему. В итоге с 10 попытки программа прошла тестрирование

Тест производительности

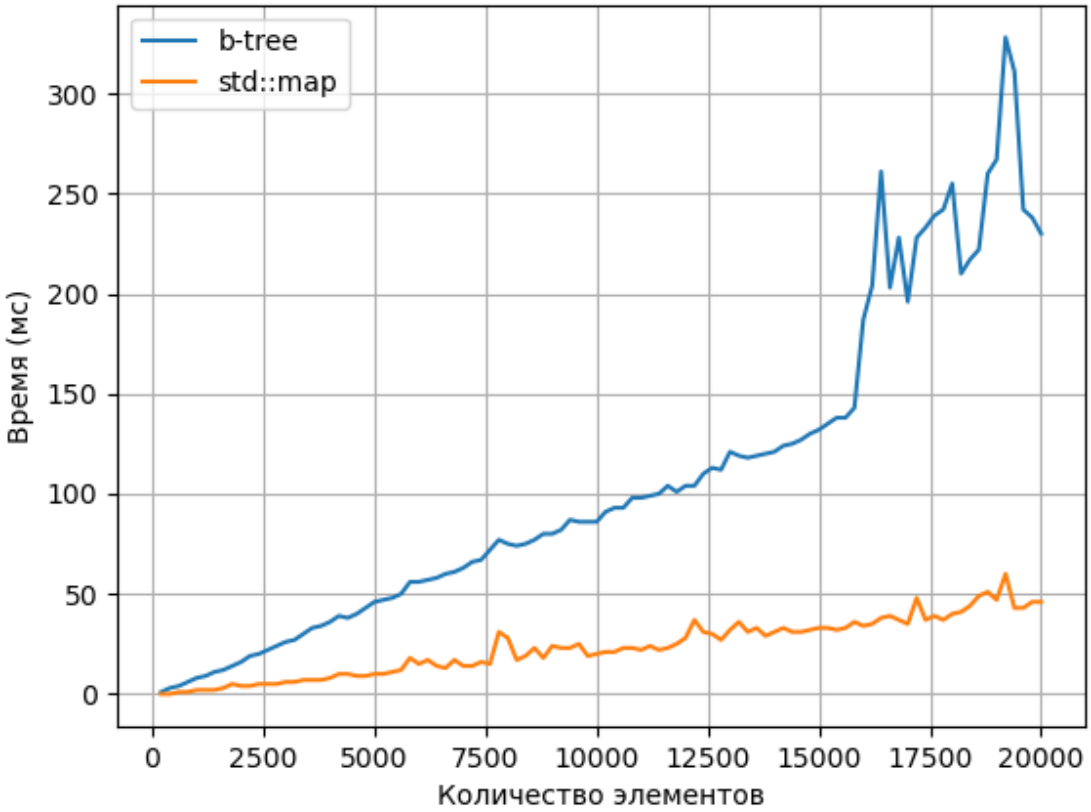
Тестирование программы производилось на 100 сгенерированных тестах с шагом в 200 элементов. Значения числа в диапазоне $[0; 18446744073709551615]$, а строка имела длин-

ну $[1; 256]$. Для сравнения было выбрана структура данных std::map из стандартной библиотеки шаблонов, так как данная структура является красно-черным деревом, что идеально подходит для сравнения сбалансированных деревьев поиска:

Вставка:

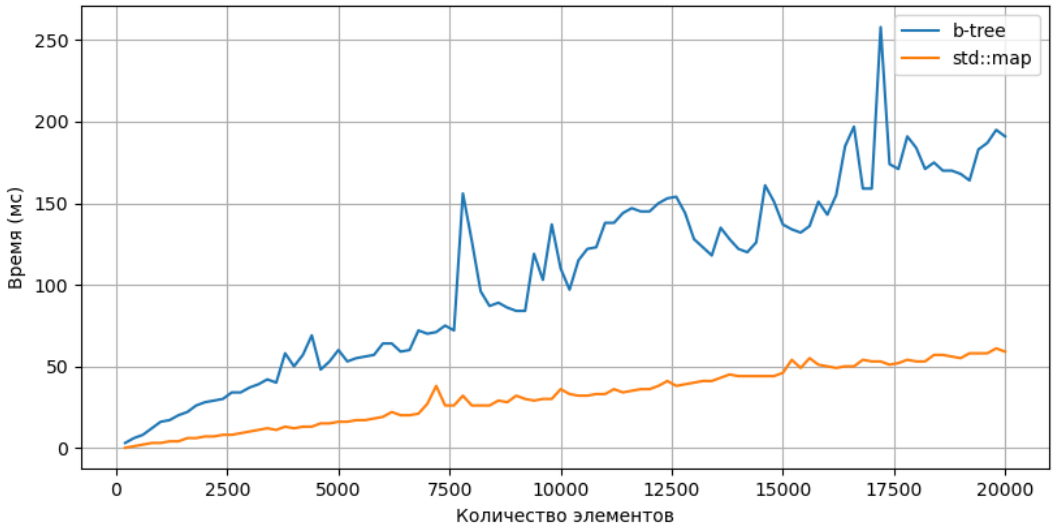


Учитывая сложность B-дерева, не удивительно, что std::map более лучшие результаты.
Поиск:



Опять же, результаты не кажутся удивительными, учитывая, что для нахождения элемента в B-дереве, нужно идти по узлам и использовать линейный поиск для нахождения ключа.

Удаление:



Удаление из B-дерева часто приводит к серьезной перестройке структуры самого дерева, в связи с чем данное удаление работает дольше.

Выводы

При выполнении данной лабораторной работы я познакомился с такой структурой данных как B-дерево. Данное дерево чаще всего применяется в том случае, если нам необходимо хранить большое количество данных и записывать их в долговременную память.

Но в то же время данное дерево обладает рядом недостатков. Ключевым недостатком является сложность его реализации. Пришлось написать кучу тестов, для того, чтобы локализовать ошибки. Также нельзя не отметить то, что дерево часто перестраивается во время удаления. При этом во время поиска нужно пробегать по всем ключам узла (если подразумевается, что t может быть большим, то не лишним будет использовать бинарный поиск)

Да, возможно B-дерево показало не высокую эффективность по сравнению с красно-черным деревом, но чаще всего, B-дерево используют для баз данных, когда весь объем информации не может храниться в быстрой памяти, для чего необходимо сохранять данные на жестком диске, и подгружать данные сразу страницами.