

Министерство цифрового развития, связи и массовых коммуникаций
Российской Федерации
ФГБОУ ВО «Сибирский государственный университет телекоммуникаций и
информатики» (СибГУТИ)

Кафедра ПМиК

КУРСОВАЯ РАБОТА

по дисциплине «Объектно-ориентированное программирование»

Тема: «Создание англо-русского словаря с применением алгоритма
построения двоичного Б-дерева»

Выполнил:

Студент группы ИП-311

Подкорытова Александра Валерьевна

Проверил:

Старший преподаватель кафедры ПМиК

Дементьева Кристина Игоревна

Новосибирск, 2024

Содержание

1. Постановка задачи.....	3
2. Технологии ООП.....	3
3. Структура классов.....	4
4. Результаты работы программы.....	6
5. Заключение.....	8
6. Используемые источники.....	9
7. Приложение. Листинг.....	9

1. Постановка задачи

Цель работы – разработать программу для построения англо-русского словаря с использованием алгоритма построения двоичного Б-дерева. Дерево должно быть описано как потомок объекта Двоичное дерево. В качестве методов должны быть описаны функция вычисления высоты, поиск заданного элемента, вставка и удаление узла.

2. Технологии ООП

В программе реализованы следующие принципы ООП:

- **Абстракция**

- Программа включает абстрактный класс BinaryTree, который определяет интерфейс (методы Insert, Search_DBD, Remove) для работы с деревом, не уточняя, как эти методы должны быть реализованы.
- Конкретная реализация методов происходит в классе-наследнике DBD, что позволяет отделить концепцию (интерфейс) дерева от деталей его реализации.

- **Наследование**

- Класс DBD наследует класс BinaryTree, что позволяет использовать общую логику, определённую в базовом классе (DeleteTree, GetHeight), а также дополнять или изменять поведение в производном классе.

- **Инкапсуляция**

- Данные дерева (корень Root, узлы Vertex) скрыты внутри классов и недоступны напрямую извне. Доступ к этим данным осуществляется через методы классов.
- Методы Insert, Search_DBD и Remove предоставляют строго определённый интерфейс для взаимодействия с деревом, скрывая сложную внутреннюю реализацию.

- Поля структуры Vertex (Data, Translation, Bal, Left, Right, Equal) также защищены от прямого доступа и управляются только через методы дерева.
- **Полиморфизм**
- Методы Insert, Search_DBD, Remove в базовом классе объявлены как виртуальные. Это позволяет объектам, представляемым типом базового класса BinaryTree, вызывать реализации этих методов из производного класса DBD.

3. Структура классов

1) Класс Vertex (структура узла дерева)

- Назначение: хранение данных и связей узла дерева.
- Поля:
 - string Data — ключ узла (слово).
 - string Translation — значение узла (перевод слова).
 - short int Bal — баланс фактора для реализации сбалансированности дерева.
 - Vertex* Left — указатель на левый дочерний узел.
 - Vertex* Right — указатель на правый дочерний узел.
 - Vertex* Equal — указатель на следующий узел с таким же ключом (цепочка дубликатов).
- Особенности:
 - Хранит данные одного элемента словаря.
 - Поддерживает механизм обработки дубликатов ключей через поле Equal.

2) Абстрактный класс BinaryTree – двоичное дерево

- Назначение: базовый класс, описывающий общую структуру бинарного дерева. Содержит общие методы для управления деревом.
- Поля:

- Vertex* Root — указатель на корень дерева.
- Методы:
 1. Виртуальные методы (должны быть реализованы в наследниках):
 - virtual void Insert(const string& key, const string& translation) = 0; — метод для добавления нового узла.
 - virtual string Search_DBD(const string& key) = 0; — метод для поиска узла по ключу.
 - virtual void Remove(const string& key) = 0; — метод для удаления узла по ключу (если встречаются одинаковые элементы, удалит первый).
 2. Общие методы:
 - void DeleteTree(Vertex* node) — рекурсивная очистка дерева.
 - int CalculateHeight(Vertex* node) — вычисление высоты дерева (рекурсивно).
 - int GetHeight() — вызов вычисления высоты дерева от корня.
 3. Конструктор и деструктор:
 - BinaryTree() — инициализация дерева с пустым корнем.
 - ~BinaryTree() — деструктор, удаление дерева, вызов DeleteTree (вызывается по завершению работы программы).
- Особенности:
 - Реализует общую логику дерева и предоставляет интерфейс для его управления.
 - Абстрактный класс (наличие чисто виртуальных методов) — нельзя создавать экземпляры.

3) Класс DBD (производный от BinaryTree) – двоичное Б-дерево

- Назначение: реализация конкретного варианта сбалансированного двоичного дерева (похожего на AVL-дерево).
- Методы:
 1. Реализация методов из BinaryTree:
 - void Insert(const string& key, const string& translation) override — вставка нового узла в дерево с учётом балансировки.

- `string Search_DBD(const string& key) override` — поиск узла по ключу.
 - `void Remove(const string& key) override` — удаление узла по ключу (учитывая цепочки дубликатов).
2. Внутренние вспомогательные методы:
- `void Insert(Vertex*& root, const string& key, const string& translation)` — рекурсивная вставка с балансировкой.
 - `string Search_DBD(Vertex* root, const string& key)` — рекурсивный поиск.
 - `void Remove(Vertex*& root, const string& key)` — рекурсивное удаление узла.
 - `void TreeRight(Vertex* node)` — обход дерева в симметричном порядке.
3. Другие методы:
- `void Print()` — печать дерева (через вызов `TreeRight`).
- Поля:
 - Наследует `Vertex* Root` из базового класса.
 - Особенности:
 - Использует дополнительные флаги `VR` и `HR` для управления балансировкой дерева при вставке.
 - Умеет обрабатывать цепочки дубликатов ключей через поле `Equal` в узле.

Отношения классов:

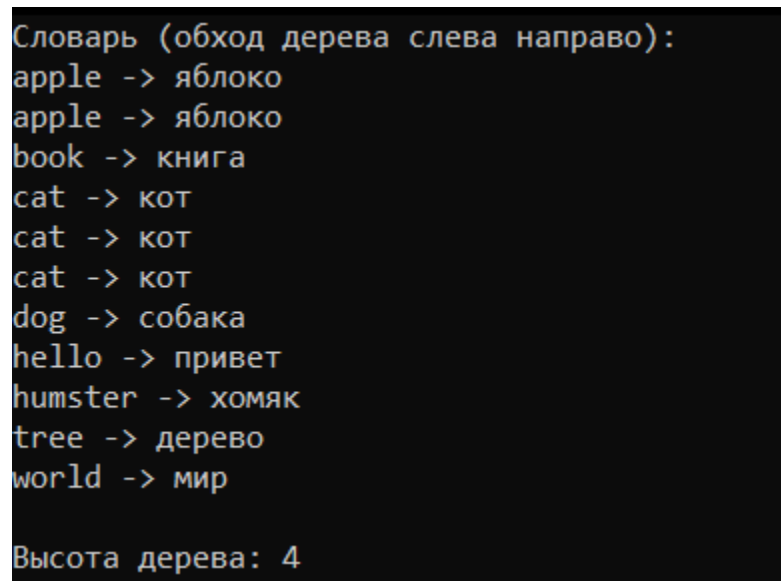
1. Наследование:
 - `DBD` наследует базовый класс `BinaryTree` и переопределяет его виртуальные методы.
2. Ассоциация:
 - Класс `DBD` использует объекты типа `Vertex` для создания структуры дерева.

4. Результаты работы программы

Программа позволяет:

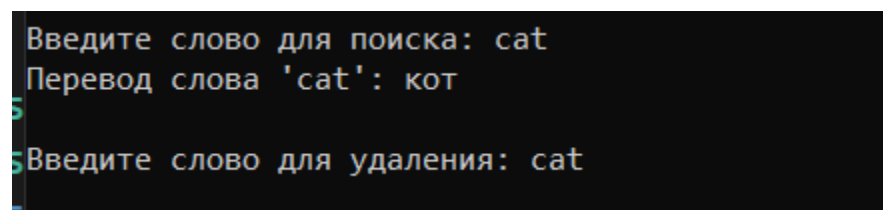
- Создавать англо-русский словарь, применяя алгоритм построения двоичного Б-дерева;
- Вычислять высоту получившегося дерева;
- Искать элемент, заданный с клавиатуры;
- Вставлять подаваемый на вход элемент в дерево;
- Удалять элемент из дерева.

Скриншоты работы программы:



```
Словарь (обход дерева слева направо):  
apple -> яблоко  
apple -> яблоко  
book -> книга  
cat -> кот  
cat -> кот  
cat -> кот  
dog -> собака  
hello -> привет  
hamster -> хомяк  
tree -> дерево  
world -> мир  
  
Высота дерева: 4
```

Рисунок 1 – получившийся словарь и высота дерева.



```
Введите слово для поиска: cat  
Перевод слова 'cat': кот  
5  
Введите слово для удаления: cat  
-
```

Рисунок 2 – поиск слова, заданного с клавиатуры, и вывод его перевода на экран.

```
Введите слово для удаления: cat

Словарь после удаления 'cat':
apple -> яблоко
apple -> яблоко
book -> книга
cat -> кот
cat -> кот
dog -> собака
hello -> привет
hamster -> хомяк
tree -> дерево
world -> мир

Высота дерева после удаления слова: 4
```

Рисунок 3 – удаление слова, заданного с клавиатуры; вывод высоты дерева после удаления.

5. Заключение

В разработанной программе я успешно смогла реализовать все поставленные задачи.

Получившаяся программа хорошо демонстрирует ключевые принципы объектно-ориентированного программирования (ООП).

Преимущества программы:

- Высокая производительность: балансировка дерева обеспечивает операции $O(\log n)$.
- Поддержка дубликатов и возможность их хранения.
- Логика добавления, поиска и удаления скрыта за интерфейсом.
- Симметричный обход дерева.
- Реализована обработка всех случаев удаления (лист, один или два потомка).

Недостатки программы и возможные улучшения:

- Добавить обработку ошибок (например, ввод некорректных данных).
- Добавить обработку ошибок (например, ввод некорректных данных).

В целом, программа является удобным инструментом для выполнения базовых операций для работы с двоичным Б-деревом.

6. Используемые источники

1. «Язык программирования C++» — Бьярн Страуструп.
2. Документация по C++: cplusplus.com.
3. Примеры использования ООП в C++ из открытых источников.
4. Лекции по объектно-ориентированному программированию (Автор: Ситняковская Е.И.).
5. «Структуры и алгоритмы обработки данных», методическое пособие – Е.В. Курапова, Е.П. Мачикина.

7. Приложение. Листинг

```
#include <iostream>
#include <string>
#include <iomanip>
#include <locale>

bool VR = true, HR = true;

using namespace std;

struct Vertex
```

```

{
    string Data;
    string Translation;
    short int Bal;
    Vertex* Left;
    Vertex* Right;
    Vertex* Equal;
};

// Класс абстрактного двоичного дерева
class BinaryTree
{
protected:
    Vertex* Root;

    void DeleteTree(Vertex* node)
    {
        if (node)
        {
            DeleteTree(node->Left);
            DeleteTree(node->Right);
            DeleteTree(node->Equal);
            delete node;
        }
    }

public:

```

```

BinaryTree() : Root(nullptr) {}
virtual ~BinaryTree() { DeleteTree(Root); }

virtual void Insert(const string& key, const string& translation) = 0;
virtual string Search_DBD(const string& key) = 0;
virtual void Remove(const string& key) = 0;
int CalculateHeight(Vertex* node)
{
    if (!node) return 0;
    return 1 + max(CalculateHeight(node->Left), CalculateHeight(node-
>Right));
}
int GetHeight()
{
    return CalculateHeight(Root);
}
};

class DBD : public BinaryTree
{
private:
    void Insert(Vertex*& root, const string& key, const string& translation)
    {
        if (!root)
        {
            root = new Vertex{ key, translation, 0, nullptr, nullptr };
            root->Data = key;

```

```

    root->Left = root->Right = nullptr;
    root->Bal = 0;
    VR = true;
}
else if (key < root->Data)
{
    Insert(root->Left, key, translation);
    if (VR)
    {
        if (!root->Bal)
        {
            Vertex* q = root->Left;
            root->Left = q->Right;
            q->Right = root;
            root = q;
            q->Bal = 1;
            VR = false;
            HR = true;
        }
        else
        {
            root->Bal = 0;
            VR = true;
            HR = false;
        }
    }
}
else

```

```

        HR = false;
    }
else if (key > root->Data)
{
    Insert(root->Right, key, translation);
    if (VR)
    {
        root->Bal = 1;
        HR = true;
        VR = false;
    }
else if (HR)
{
    if (root->Bal)
    {
        Vertex* q = root->Right;
        root->Bal = 0;
        q->Bal = 0;
        root->Right = q->Left;
        q->Left = root;
        root = q;
        VR = true;
        HR = false;
    }
else
    HR = false;
}
}

```

```

    }
    else if (key == root->Data)
    {
        Vertex* newEqual = new Vertex{ key, translation, 0, nullptr,
nullptr };
        newEqual->Data = key;
        newEqual->Equal = root->Equal;
        root->Equal = newEqual;
    }
}

```

```

string Search_DBD(Vertex* root, const string& key)

```

```

{
    if (!root) return "Перевод слова не найден.";

    if (key < root->Data)
        return Search_DBD(root->Left, key);
    else if (key > root->Data)
        return Search_DBD(root->Right, key);
    else
        return root->Translation; //вернём первый найденный перевод
}

```

```

void Remove(Vertex*& root, const string& key)

```

```

{
    if (!root) return;

```

```

if (key < root->Data)
    Remove(root->Left, key);
else if (key > root->Data)
    Remove(root->Right, key);
else
{
    if (root->Equal)
    {
        //если у элемента есть дубли, удаляем первый из цепочки
дубликатов
        Vertex* temp = root->Equal;
        root->Equal = root->Equal->Equal;
        delete temp;
    }
    else
    {
        if (!root->Left && !root->Right)
        {
            delete root;
            root = nullptr;
        }
        else if (!root->Left)
        {
            //один потомок справа
            Vertex* temp = root;
            root = root->Right;
            delete temp;
        }
    }
}

```

```

    }
    else if (!root->Right)
    {
        //один потомок слева
        Vertex* temp = root;
        root = root->Left;
        delete temp;
    }
    else
    {
        //2 потомка: замена минимальным элементом в правом
        поддереве
        Vertex* minRight = root->Right;
        while (minRight->Left)
        {
            minRight = minRight->Left;
        }
        root->Data = minRight->Data;
        root->Translation = minRight->Translation;
        Remove(root->Right, minRight->Data);
    }
}
}
}
}

```

public:

```
DBD() : BinaryTree() {}
```



```

void Insert(const string& key, const string& translation) override
{
    Insert(Root, key, translation);
}

```

```

string Search_DBD(const string& key) override
{
    return Search_DBD(Root, key);
}

```

```

void Remove(const string& key) override
{
    Remove(Root, key);
}

```

```

void TreeRight(Vertex* node)
{
    if (!node) return;
    TreeRight(node->Left);
    cout << node->Data << " -> " << node->Translation << endl;
    TreeRight(node->Equal);
    TreeRight(node->Right);
}

```

```

void Print()
{

```

```

    if (!Root)
        cout << "Словарь пустой." << endl;
    else
        TreeRight(Root);
}
};

int main()
{
    setlocale(LC_ALL, "ru");

    DBD derevo;
    derevo.Insert("hello", "привет");
    derevo.Insert("apple", "яблоко");
    derevo.Insert("cat", "кот");
    derevo.Insert("world", "мир");
    derevo.Insert("tree", "дерево");
    derevo.Insert("cat", "кот");
    derevo.Insert("apple", "яблоко");
    derevo.Insert("dog", "собака");
    derevo.Insert("hamster", "хозяк");
    derevo.Insert("book", "книга");
    derevo.Insert("cat", "кот");

    cout << "Словарь (обход дерева слева направо):" << endl;
    derevo.Print();
}

```

```

cout << "\nВысота дерева: " << derevo.GetHeight() << endl;

//поиск слова и его перевод
string search_word;
cout << "\nВведите слово для поиска: ";
cin >> search_word;

cout << "Перевод слова " << search_word << ": " <<
derevo.Search_DBD(search_word) << endl;

//удаление слова
string delete_word;
cout << "\nВведите слово для удаления: ";
cin >> delete_word;
derevo.Remove(delete_word);
cout << "\nСловарь после удаления " << delete_word << ":" << endl;
derevo.Print();

cout << "\nВысота дерева после удаления слова: " << derevo.GetHeight()
<< endl;

return 0;
}

```