Белорусский государственный технологический университет

Факультет информационных технологий

Кафедра программной инженерии

 Лабораторная работа № 11

По дисциплине «Основы алгоритмизации и программирования»

На тему «Бинарные деревья»

Выполнил:

Студент 1 курса 10 группы

Жамойдо Артём Игоревич

Преподаватель: асс. Андронова М.В.

Минск, 2024

Разработать программу работы с ***бинарным деревом поиска***, в которую включить основные функции манипуляции данными и функцию в соответствии со своим вариантом из таблицы, представленной ниже.

**Вариант 4**

Вершина бинарного дерева содержит ключ, строку и два указателя на потомков. Написать функцию, которая подсчитывает число ветвей от корня до ближайшей вершины с заданным ключом, и выводит их.

#include <iostream>

#include <string.h>

using namespace std;

// Cоздаем шаблонный класс Tree

template<typename T>

class Tree

{

private:

template<typename T>

// Cоздаем шаблонный класс Node - элемент дерева

class Node

{

public:

// Указатель на правый элемент

Node<T>\* right;

// Указатель на левый элемент

Node<T>\* left;

// Данные

T data;

// Ключ

int key;

// Конструктор для узла дерева

Node(T data, int key, Node<T>\* right = nullptr, Node<T>\* left = nullptr)

{

// Записываем данные в экземпляр объекта Node

this->data = data;

this->key = key;

this->right = right;

this->left = left;

}

};

// Указатель на корень дерева

Node<T>\* Root;

public:

// Конструктор основного класса

Tree();

// Методы класса для работы с деревом

Node<T>\* getRoot()

{

return Root;

}

int getSize()

{

return Size;

}

void print(Node<T>\* t, int level);

void insert(T data, int key);

void findMax(Node<T>\* current, int key, int level);

int Size;

T sum;

};

template<typename T>

// Конструктор главного класса

Tree<T>::Tree()

{

// Задаем начальные значения

Root = nullptr;

Size = 0;

}

template<typename T>

// Метод поиска максимального значения

void Tree<T>::findMax(Node<T>\* current, int key, int level)

{

// Проверка на пустоту дерева

if (current != nullptr)

{

findMax(current->left, key, level + 1);

if (current->key == key)

{

cout << "Количество ветвей: " << level;

}

findMax(current->right, key, level + 1);

}

}

template<typename T>

// Метод вывода элементов дерева

void Tree<T>::print(Node<T>\* t, int level)

{

// Проверка на пустоту

if (t != nullptr)

{

// Рекунрсивно выводим элементы дерева с отступами, в зависимости от занчения level

print(t->left, level + 1);

for (int i = 0; i < level; i++)

cout << " ";

cout << t->key << " " << t->data << endl;

print(t->right, level + 1);

}

}

template<typename T>

// Метод добавления нового элемента в дерево

void Tree<T>::insert(T data, int key)

{

// Создаем временные переменные на текущий и предыдущий элементы

Node<T>\* current = Root;

Node<T>\* prev = nullptr;

// Проверка на пустоту дерева

if (Root == nullptr)

{

// Записываем в корень первый элемент

Root = new Node<T>(data, key);

}

else

{

// Перебираем дерево, пока не дойдем до конца

while (current != nullptr)

{

// Записываем в prev текущее значение и в зависимости от значения ключа смещаем current

prev = current;

if (key > current->key)

{

current = current->right;

}

else

{

current = current->left;

}

}

// Сравнения значения ключа последнего элемента с текущим и создаение нового

if (key > prev->key)

{

prev->right = new Node<T>(data, key);

}

else

{

prev->left = new Node<T>(data, key);

}

}

Size++;

}

// Подсчет числа ветвей от корня до ближайшей вершины

void countBranches()

{

Tree<string> tree;

tree.insert("Artiom", 19);

tree.insert("Ivan", 17);

tree.insert("Stepan", 21);

tree.insert("Gleb", 15);

tree.insert("Alex", 23);

tree.insert("Egor", 13);

tree.insert("Anton", 25);

tree.insert("Petr",11);

tree.insert("Stas", 27);

tree.print(tree.getRoot(), 0);

cout << endl << endl;

int N;

cout << "Введите ключ для подсчета ветвей до элемента: ";

cin >> N;

tree.findMax(tree.getRoot(), N, 1);

}

int main()

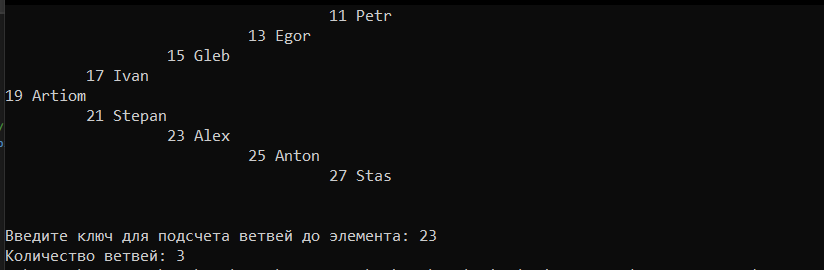
{

setlocale(0, "ru");

countBranches();

return 0;

}



Доп. задачи

**Вариант 14**

Вершина бинарного дерева содержит ключ и два указателя на потомков. Написать функцию вычисления среднего арифметического всех элементов дерева.

#include <iostream>

#include <string.h>

using namespace std;

// Cоздаем шаблонный класс Tree

template<typename T>

class Tree

{

private:

template<typename T>

// Cоздаем шаблонный класс Node - элемент дерева

class Node

{

public:

// Указатель на правый элемент

Node<T>\* right;

// Указатель на левый элемент

Node<T>\* left;

// Данные

T data;

// Ключ

int key;

// Конструктор для узла дерева

Node(T data, int key, Node<T>\* right = nullptr, Node<T>\* left = nullptr)

{

// Записываем данные в экземпляр объекта Node

this->data = data;

this->key = key;

this->right = right;

this->left = left;

}

};

// Указатель на корень дерева

Node<T>\* Root;

public:

// Конструктор основного класса

Tree();

// Методы класса для работы с деревом

Node<T>\* getRoot()

{

return Root;

}

int getSize()

{

return Size;

}

void print(Node<T>\* t, int level);

void insert(T data, int key);

void findAverage(Node<T>\* current);

void findMax(Node<T>\* current, int key, int level);

int Size;

T sum;

};

template<typename T>

// Конструктор главного класса

Tree<T>::Tree()

{

// Задаем начальные значения

Root = nullptr;

Size = 0;

}

template<typename T>

// Метод нахождения среднего арифметического всех элементов дерева

void Tree<T>::findAverage(Node<T>\* current)

{

if (current != nullptr)

{

findAverage(current->left);

sum += current->data;

findAverage(current->right);

}

}

template<typename T>

// Метод поиска максимального значения

void Tree<T>::findMax(Node<T>\* current, int key, int level)

{

// Проверка на пустоту дерева

if (current != nullptr)

{

findMax(current->left, key, level + 1);

if (current->key == key)

{

cout << level;

}

findMax(current->right, key, level + 1);

}

}

template<typename T>

// Метод вывода элементов дерева

void Tree<T>::print(Node<T>\* t, int level)

{

// Проверка на пустоту

if (t != nullptr)

{

// Рекунрсивно выводим элементы дерева с отступами, в зависимости от занчения level

print(t->left, level + 1);

for (int i = 0; i < level; i++)

cout << " ";

cout << t->key << " " << t->data << endl;

print(t->right, level + 1);

}

}

template<typename T>

// Метод добавления нового элемента в дерево

void Tree<T>::insert(T data, int key)

{

// Создаем временные переменные на текущий и предыдущий элементы

Node<T>\* current = Root;

Node<T>\* prev = nullptr;

// Проверка на пустоту дерева

if (Root == nullptr)

{

// Записываем в корень первый элемент

Root = new Node<T>(data, key);

}

else

{

// Перебираем дерево, пока не дойдем до конца

while (current != nullptr)

{

// Записываем в prev текущее значение и в зависимости от значения ключа смещаем current

prev = current;

if (key > current->key)

{

current = current->right;

}

else

{

current = current->left;

}

}

// Сравнения значения ключа последнего элемента с текущим и создаение нового

if (key > prev->key)

{

prev->right = new Node<T>(data, key);

}

else

{

prev->left = new Node<T>(data, key);

}

}

Size++;

}

// Среднее арифметическое всех элементов дерева

void average()

{

Tree<int> tree;

tree.insert(5, 19);

tree.insert(10, 21);

tree.insert(15, 17);

tree.insert(20, 23);

tree.insert(25, 15);

tree.insert(30, 25);

tree.insert(35, 13);

tree.insert(40, 27);

tree.insert(45, 11);

tree.print(tree.getRoot(), 0);

tree.findAverage(tree.getRoot());

cout << "Среднее арифметическое: " << tree.sum / tree.Size;

}

int main()

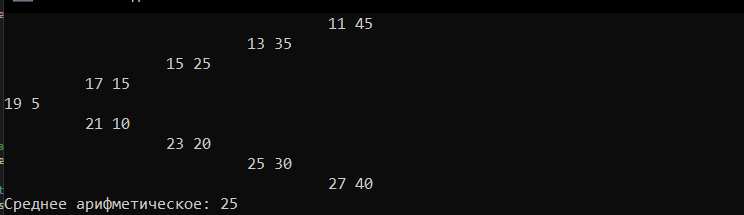
{

setlocale(0, "ru");

average();

return 0;

}

****

**Вариант 8**

Вершина бинарного дерева содержит ключ, 2 целых числа и два указателя на потомков. Написать функцию удаления вершины с максимальной суммой 2 целых значений узла.

#include <iostream>

#include <string.h>

using namespace std;

template<typename T>

// Создаем шаблонный класс Tree

class Tree

{

private:

template<typename T>

// Создаем шаблонный класс Node - элемент дерева

class Node

{

public:

Node<T>\* right; // Указатель на правый элемент

Node<T>\* left; // Указатель на левый элемент

// Данные

T dataF;

T dataS;

int key;

int sum;

// Конструктор для узла дерева

Node(T dataF, T dataS, int key, Node<T>\* right = nullptr, Node<T>\* left = nullptr)

{

// Записываем данные в экземпляр объекта Node

this->dataF = dataF;

this->dataS = dataS;

this->key = key;

this->right = right;

this->left = left;

this->sum = dataS + dataF;

}

};

int Size;

Node<T>\* Root; // Указатель на корень дерева

public:

Tree(); // Конструктор основного класса

// Методы класса для работы с деревом

Node<T>\* getRoot()

{

return Root;

}

int getSize()

{

return Size;

}

void print(Node<T>\* t, int level);

void insert(T dataF, T dataS, int key);

void del(int key);

void findMax(Node<T>\* current);

int MAX;

int index;

};

template<typename T>

// Конструктор главного класса

Tree<T>::Tree()

{

// Задаем начальные значения

Root = nullptr;

Size = 0;

MAX = 0;

index = 0;

}

template<typename T>

// Метод поиска максимального значения

void Tree<T>::findMax(Node<T>\* current)

{

// Проверка на пустоту дерева

if (current != nullptr)

{

// Рекурсивно перебираем элементы справа и слева и с помощью условия находим максимальную сумму

findMax(current->left);

if (MAX < current->sum)

{

MAX = current->sum;

index = current->key;

}

findMax(current->right);

}

}

template<typename T>

// Метод удаления вершины

void Tree<T>::del(int key)

{

Node<T>\* Del, \* Prev\_Del, \* R, \* Prev\_R;

Del = Root;

Prev\_Del = NULL;

while (Del != NULL && Del->key != key) // Поиск элемента и его родителя

{

Prev\_Del = Del;

if (Del->key > key)

Del = Del->left;

else

Del = Del->right;

}

if (Del == NULL) // Элемент не найден

{

puts("\nНет такого ключа");

return;

}

if (Del->right == NULL) // Поиск элемента R для замены

R = Del->left;

else

{

if (Del->left == NULL)

R = Del->right;

else

{

// Поиск самого правого элемента в левом поддереве

Prev\_R = Del;

R = Del->left;

while (R->right != NULL)

{

Prev\_R = R;

R = R->right;

}

// Найден элемент для замены R и его родителя Prev\_R

if (Prev\_R == Del)

R->right = Del->right;

else

{

R->right = Del->right;

Prev\_R->right = R->left;

R->left = Prev\_R;

}

}

}

if (Del == Root)

Root = R; // Удаление корня и замена его на R

else

{

// Поддерево R присоединяется к родителю удаляемого узла

if (Del->key < Prev\_Del->key)

{

Prev\_Del->left = R; // На левую ветвь

}

else

Prev\_Del->right = R; // На правую ветвь

}

int tmp = Del->key;

cout << "\nУдален элемент с ключом " << tmp << endl;

delete Del;

Size--;

}

template<typename T>

// Метод вывода элементов дерева

void Tree<T>::print(Node<T>\* t, int level)

{

// Проверка на пустоту

if (t != nullptr)

{

// Рекунрсивно выводим элементы дерева с отступами, в зависимости от занчения level

print(t->left, level + 1);

for (int i = 0; i < level; i++)

cout << " ";

cout << t->key << " (" << t->dataF << " " << t->dataS << ")" << endl;

print(t->right, level + 1);

}

}

template<typename T>

// Метод добавления нового элемента в дерево

void Tree<T>::insert(T dataF, T dataS, int key)

{

// Создаем временные переменные на текунщий и на предыдущий элементы

Node<T>\* current = Root;

Node<T>\* prev = nullptr;

// Проверка на пустоту дерева

if (Root == nullptr)

{

// Записываем в корень первый элемент

Root = new Node<T>(dataF, dataS, key);

}

else

{

// Перебираем дерево, пока не дойдем до конца

while (current != nullptr)

{

// Записываем в prev текущее значение и в зависимости от значения ключа смещаем current

prev = current;

if (key > current->key)

{

current = current->right;

}

else

{

current = current->left;

}

}

// Сравнения значения ключа последнего элемента с текущим и создаение нового

if (key > prev->key)

{

prev->right = new Node<T>(dataF, dataS, key);

}

else

{

prev->left = new Node<T>(dataF, dataS, key);

}

}

Size++;

}

// Удаление вершины с максимальной суммой 2 целых значений узла

void deleteMax()

{

Tree<int> tree;

tree.insert(10, 11, 19);

tree.insert(16, 15, 21);

tree.insert(13, 5, 17);

tree.insert(14, 16, 23);

tree.insert(11, 6, 15);

tree.insert(12, 18, 25);

tree.insert(17, 7, 13);

tree.insert(17, 7, 27);

tree.insert(17, 7, 11);

tree.print(tree.getRoot(), 0);

cout << endl << endl;

tree.findMax(tree.getRoot());

tree.del(tree.index);

cout << endl;

tree.print(tree.getRoot(), 0);

}

int main()

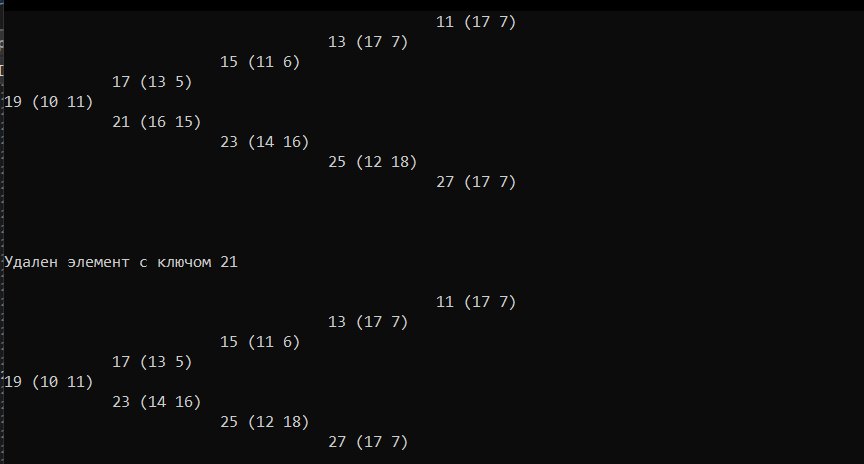
{

setlocale(0, "ru");

deleteMax();

return 0;

}

****

**Вариант 3**

Вершина бинарного дерева содержит ключ, строку и два указателя на потомков. Написать функцию вывода всех элементов дерева по уровням: корень дерева, вершины 1-го уровня, вершины 2-го уровня, ... и т. д.

#include <iostream>

#include <string.h>

using namespace std;

// Cоздаем шаблонный класс Tree

template<typename T>

class Tree

{

private:

template<typename T>

// Cоздаем шаблонный класс Node - элемент дерева

class Node

{

public:

// Указатель на правый элемент

Node<T>\* right;

// Указатель на левый элемент

Node<T>\* left;

// Данные

T data;

// Ключ

int key;

// Конструктор для узла дерева

Node(T data, int key, Node<T>\* right = nullptr, Node<T>\* left = nullptr)

{

// Записываем данные в экземпляр объекта Node

this->data = data;

this->key = key;

this->right = right;

this->left = left;

}

};

// Указатель на корень дерева

Node<T>\* Root;

public:

// Конструктор основного класса

Tree();

// Методы класса для работы с деревом

Node<T>\* getRoot()

{

return Root;

}

int getSize()

{

return Size;

}

void print(Node<T>\* t, int level);

void insert(T data, int key);

void findAverage(Node<T>\* current);

void findMax(Node<T>\* current, int key, int level);

int Size;

T sum;

};

template<typename T>

// Конструктор главного класса

Tree<T>::Tree()

{

// Задаем начальные значения

Root = nullptr;

Size = 0;

}

template<typename T>

// Метод нахождения среднего арифметического всех элементов дерева

void Tree<T>::findAverage(Node<T>\* current)

{

if (current != nullptr)

{

findAverage(current->left);

sum += current->data;

findAverage(current->right);

}

}

template<typename T>

// Метод поиска максимального значения

void Tree<T>::findMax(Node<T>\* current, int key, int level)

{

// Проверка на пустоту дерева

if (current != nullptr)

{

findMax(current->left, key, level + 1);

if (current->key == key)

{

cout << level;

}

findMax(current->right, key, level + 1);

}

}

template<typename T>

// Метод вывода элементов дерева

void Tree<T>::print(Node<T>\* t, int level)

{

// Проверка на пустоту

if (t != nullptr)

{

// Рекунрсивно выводим элементы дерева с отступами, в зависимости от занчения level

print(t->left, level + 1);

for (int i = 0; i < level; i++)

cout << " ";

if (level == 0)

{

cout << "Корень: " << t->key << " " << t->data << endl;

}

else

{

cout << "Уровень " << level << ": " << t->key << " " << t->data << endl;

}

print(t->right, level + 1);

}

}

template<typename T>

// Метод добавления нового элемента в дерево

void Tree<T>::insert(T data, int key)

{

// Создаем временные переменные на текущий и предыдущий элементы

Node<T>\* current = Root;

Node<T>\* prev = nullptr;

// Проверка на пустоту дерева

if (Root == nullptr)

{

// Записываем в корень первый элемент

Root = new Node<T>(data, key);

}

else

{

// Перебираем дерево, пока не дойдем до конца

while (current != nullptr)

{

// Записываем в prev текущее значение и в зависимости от значения ключа смещаем current

prev = current;

if (key > current->key)

{

current = current->right;

}

else

{

current = current->left;

}

}

// Сравнения значения ключа последнего элемента с текущим и создаение нового

if (key > prev->key)

{

prev->right = new Node<T>(data, key);

}

else

{

prev->left = new Node<T>(data, key);

}

}

Size++;

}

// Вывод всех элементов дерева по уровням

void getLevel()

{

Tree<int> tree;

tree.insert(5, 19);

tree.insert(10, 21);

tree.insert(15, 17);

tree.insert(20, 23);

tree.insert(25, 15);

tree.insert(30, 25);

tree.insert(35, 13);

tree.insert(40, 27);

tree.insert(45, 11);

tree.print(tree.getRoot(), 0);

}

int main()

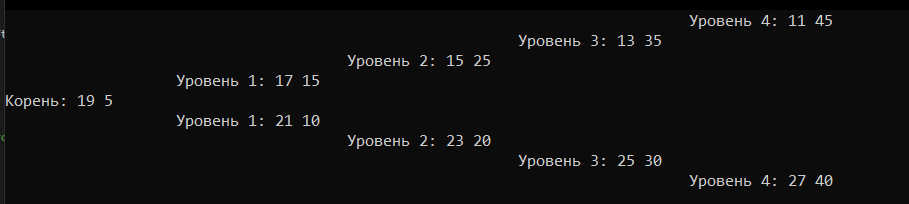
{

setlocale(0, "ru");

getLevel();

return 0;

}

****