БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

**Динамические структуры данных. Дерево отрезков. Бинарное дерево поиска. Дерево-формулы. Тестирование кода.**

**По:** Основы алгоритмизации и программирования

**Выполнил:** Корнелюк

Валентин Владимирович

1 курс 4 группа ПОИТ

**Преподаватель:**

Белодед Николай Иванович

**г. Минск**

2023 г.

**Оглавление**

[**1. Дерево отрезков** 3](#_Toc133968674)

[**2. Пример программы с использованием бинарного дерева поиска** 6](#_Toc133968675)

[**3.Пример программы, выполняющей построение бинарного дерева и его изображение на экране.** 10](#_Toc133968676)

[**4.Пример программы, иллюстрирующей поиск заданной вершины в дереве, добавление вершины в дерево, удаление вершины из дерева.** 14](#_Toc133968677)

[**5.пpогpамма постpоения деpева-фоpмулы по заданной постфиксной фоpмуле и использование постpоенного деpева для получения пpефиксной и инфиксной фоpмул.** 19](#_Toc133968678)

# **1. Дерево отрезков**

Дерево отрезков — это структура данных, которая позволяет быстро решать задачи на отрезках массива, такие как поиск суммы на отрезке, поиск минимума или максимума на отрезке, изменение элемента массива и т. д.

Дерево отрезков представляет собой двоичное дерево, в котором каждому узлу соответствует отрезок исходного массива. В листовых узлах хранятся значения элементов массива, а в узлах-отрезках хранятся результаты агрегации значений дочерних узлов (например, суммы или минимумы). При запросе к дереву отрезков для заданного отрезка находится наименьший узел, полностью содержащий этот отрезок, и возвращается значение, хранимое в этом узле или его дочерних узлах.

Можно представить дерево отрезков как набор вложенных прямоугольников, где каждый прямоугольник отвечает за определенный диапазон элементов в исходном массиве.

Например, если нужно быстро найти сумму чисел в определенном диапазоне элементов массива, то дерево отрезков может помочь решить эту задачу очень быстро. Вместо того, чтобы перебирать каждый элемент от начала до конца диапазона, мы можем использовать дерево отрезков, чтобы быстро найти сумму элементов, находящихся в нужном диапазоне.

Таким образом, дерево отрезков позволяет эффективно обрабатывать запросы поиска и изменения информации в больших массивах данных.

Дерево отрезков может быть построено за время O(N log N), где N — размер исходного массива. Запросы к дереву отрезков выполняются за время O(log N). Эффективность работы дерева отрезков позволяет его использовать в решении многих задач, связанных с работой с отрезками в массиве.

Дерево отрезков находит применение во многих областях, в том числе в вычислительной геометрии, базах данных, анализе данных, компьютерной графике, алгоритмах машинного обучения и многих других.

Примером использования дерева отрезков является задача о поиске количества точек, находящихся внутри заданной области на плоскости. В этом случае мы можем представить точки на плоскости в виде последовательности координат и построить дерево отрезков на этой последовательности. Затем, используя дерево отрезков, мы можем эффективно находить количество точек, попадающих в заданную область.

Пример построения дерева отрезков и вывод его на экран.

#include <iostream> // подключение библиотеки ввода/вывода

using namespace std; // объявление пространства имен std

struct node // Определение структуры node, которая представляет вершину дерева отрезков.

{

int KeyMin; // Минимальный ключ вершины.

int KeyMax; // Максимальный ключ вершины.

node\* Left; // Указатель на "левого" сына.

node\* Right; // Указатель на "правого" сына.

};

class TREE // Определение класса TREE, который представляет дерево отрезков.

{

private:

node\* Tree; // Указатель на корень дерева.

void Search(int, int, node\*\*); // Определение приватной функции Search, которая рекурсивно ищет место для вставки новой вершины в дерево.

public:

TREE() { Tree = NULL; } // Определение конструктора класса TREE, который инициализирует указатель на корень дерева значением NULL.

void BuildTree(); // Определение публичной функции BuildTree, которая строит дерево отрезков.

node\*\* GetTree() { return &Tree; } // Определение публичной функции GetTree, которая возвращает указатель на корень дерева.

void CleanTree(node\*\*);// Определение публичной функции CleanTree, которая рекурсивно удаляет все вершины дерева.

void Vyvod(node\*\*, int);// Определение публичной функции Vyvod, которая рекурсивно выводит все вершины дерева на экран.

};

void main()

{

setlocale(LC\_ALL, "Rus"); // Установка русской локали для вывода сообщений на кириллице.

TREE A; // Создание объекта класса TREE.

A.BuildTree(); // Вызов функции BuildTree() для построения дерева отрезков.

cout << "\nВывод дерева:\n";

A.Vyvod(A.GetTree(), 0); // Вызов функции Vyvod() для вывода дерева на экран.

A.CleanTree(A.GetTree()); // Вызов функции CleanTree() для удаления дерева из памяти.

cout << "\n";

system("PAUSE"); // Приостановка выполнения программы, ожидание нажатия любой клавиши пользователем.

}

void TREE::BuildTree()

// Построение бинарного дерева (рекурсивный алгоритм).

// Tree - указатель на корень дерева.

{

int k1, k2;

cout << "Введите два целых числа...\n"; // Просим пользователя ввести два целых числа.

cin >> k1; // Читаем первое число из консоли.

cin >> k2; // Читаем второе число из консоли.

Search(k1, k2, &Tree); // Вызываем функцию Search() для поиска места в дереве, куда нужно добавить новый узел.

}

void TREE::Search(int k1, int k2, node\*\* p) // Функция Search - рекурсивно строит дерево отрезков с корнем в вершине, на которую указывает p.

{

if (k2 - k1 > 1)// Если длина отрезка больше 1, то создаем новую вершину дерева.

{

\*p = new (node); // Выделяем память под новую вершину.

(\*\*p).KeyMin = k1; // Устанавливаем минимальное значение ключа вершины.

(\*\*p).KeyMax = k2; // Устанавливаем максимальное значение ключа вершины.

(\*\*p).Left = (\*\*p).Right = NULL; // Инициализируем указатели на левого и правого потомков нулями.

Search(k1, (k1 + k2) / 2, &((\*\*p).Left));// Рекурсивно вызываем функцию Search для левого и правого потомков.

Search((k1 + k2) / 2, k2, &((\*\*p).Right));

}

else // Иначе создаем листовую вершину.

{

\*p = new (node); // Выделяем память под новую вершину.

(\*\*p).KeyMin = k1; // Устанавливаем минимальное значение ключа вершины.

(\*\*p).KeyMax = k2; // Устанавливаем максимальное значение ключа вершины.

(\*\*p).Left = (\*\*p).Right = NULL; // Инициализируем указатели на левого и правого потомков нулями.

}

}

void TREE::CleanTree(node\*\* w)// Функция CleanTree - рекурсивно удаляет все вершины дерева, начиная с корня, на который указывает w.

{

if (\*w != NULL) // Если указатель не равен нулю, то есть вершина существует.

{

CleanTree(&((\*\*w).Left));// Рекурсивно вызываем функцию CleanTree для левого и правого потомков.

CleanTree(&((\*\*w).Right));

delete w; // Удаляем вершину.

}

}

// Функция рекурсивно выводит на экран содержимое дерева отрезков.

// w - указатель на текущую вершину дерева, которую нужно вывести.

// l - текущий уровень вложенности вершины в дерево (для корректного форматирования вывода).

void TREE::Vyvod(node\*\* w, int l)

{

int i; //счетчик для уровня дерева

if (\*w != NULL) // Проверяем, не является ли текущая вершина пустой.

{

Vyvod(&((\*\*w).Right), l + 1);// Рекурсивно выводим правое поддерево.

for (i = 1; i <= l; i++) cout << " ";// Отступаем на текущий уровень и выводим ключи вершины.

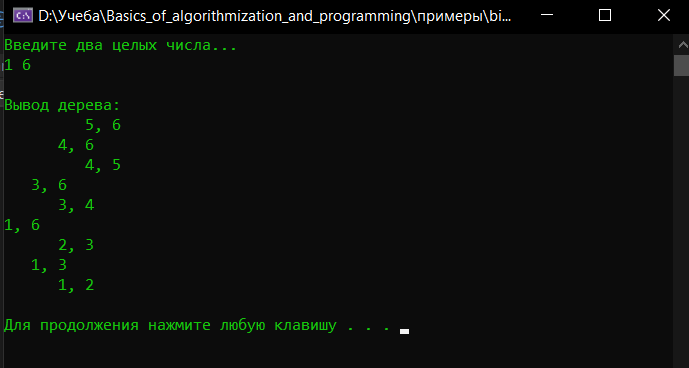
cout << (\*\*w).KeyMin << ", " << (\*\*w).KeyMax << endl;//выводим значение ключа на экран

// Рекурсивно выводим левое поддерево.

Vyvod(&((\*\*w).Left), l + 1);

}

}



# **2. Пример программы с использованием бинарного дерева поиска**

#include <iostream>// Включаем заголовочный файл для работы со стандартным вводом/выводом

using namespace std;// объявление пространства имен std

struct node // Объявляем структуру для вершины дерева

{

int Key; // Ключ вершины

int Count; // Количество вершин с таким же ключом

node\* Left; // Указатель на левое поддерево

node\* Right; // Указатель на правое поддерево

};

class TREE// Объявляем класс для бинарного дерева

{

private:

node\* Tree; // Указатель на корень дерева

void Search(int, node\*\*); // Поиск вершины с ключом int в дереве со вставкой

public:

TREE() { Tree = NULL; } // Конструктор по умолчанию, инициализирующий корень дерева как NULL

node\*\* GetTree() { return &Tree; } // Получение указателя на корень дерева

void BuildTree(); // Построение дерева

void CleanTree(node\*\*); // Очистка дерева

void ObhodEnd(node\*\*); // Концевой обход дерева

void ObhodLeft(node\*\*); // Левосторонний обход дерева

void ObhodBack(node\*\*); // Обратный обход дерева

void Vyvod(node\*\*, int); // Изображение дерева на экране дисплея

int Height(node\*\*); // Определение высоты бинарного дерева

};

void main() {//вызов главной функции с которой начинается выполнение программы

setlocale(LC\_ALL, "Rus"); // Устанавливаем локаль для корректной работы с русским языком

TREE A; // Создаем объект класса TREE

A.BuildTree(); // Вызываем метод для построения бинарного дерева

cout << "\nВывод дерева:\n";

A.Vyvod(A.GetTree(), 0); // Выводим дерево на экран

cout << "\nВысота дерева:" << A.Height(A.GetTree()) << endl; // Вычисляем высоту дерева и выводим ее на экран

cout << "\nЛевосторонний обход дерева: ";

A.ObhodLeft(A.GetTree()); // Обходим дерево слева направо

cout << "\nКонцевой обход дерева: ";

A.ObhodEnd(A.GetTree()); // Обходим дерево в порядке концевых вершин

cout << "\nОбратный обход дерева: ";

A.ObhodBack(A.GetTree()); // Обходим дерево справа налево

A.CleanTree(A.GetTree()); // Очищаем память, выделенную под дерево

cout << "\n";

system("PAUSE");

}

void TREE::BuildTree()

// Построение бинарного дерева (рекурсивный алгоритм).

// Tree - указатель на корень дерева.

{

int el;//переменная для ввода эелментов

cout << "Вводите ключи вершин дерева ...\n"; // Выводим приглашение на ввод ключей вершин дерева.

cin >> el;

while (el != 0) // Запускаем цикл ввода ключей до тех пор, пока не будет введен 0.

{

Search(el, &Tree); // Ищем вершину с ключом el в дереве и вставляем ее, если ее нет.

cin >> el; // Запрашиваем новый ключ.

}

}

void TREE::Search(int x, node\*\* p)

// Поиск вершины с ключом x в дереве со вставкой (рекурсивный алгоритм).

// \*p - указатель на корень дерева.

{

if (\*p == NULL) // Если указатель на корень дерева равен NULL, значит вершины в дереве нет и ее нужно вставить.

{

\*p = new(node); // Выделяем память для новой вершины и присваиваем указателю на корень дерева адрес этой вершины.

(\*\*p).Key = x; // Присваиваем ключ новой вершине.

(\*\*p).Count = 1; // Присваиваем начальное значение счетчика.

(\*\*p).Left = NULL; // Присваиваем значения NULL указателю на левое поддерево новой вершины.

(\*\*p).Right = NULL; // Присваиваем значения NULL указателю на правое поддерево новой вершины.

}

else if (x < (\*\*p).Key) // Если искомый ключ меньше ключа текущей вершины, идем в левое поддерево.

{

Search(x, &((\*\*p).Left)); // Рекурсивно вызываем функцию поиска для левого поддерева.

}

else if (x > (\*\*p).Key) // Если искомый ключ больше ключа текущей вершины, идем в правое поддерево.

{

Search(x, &((\*\*p).Right)); // Рекурсивно вызываем функцию поиска для правого поддерева.

}

else // Если ключ уже существует в дереве, увеличиваем счетчик этой вершины.

{

(\*\*p).Count = (\*\*p).Count + 1;

}

}

// Левосторонний обход дерева.

// w - указатель на корень дерева.

void TREE::ObhodLeft(node\*\* w)

{

if (\*w != NULL)//если дерево не пустое

{

cout << (\*\*w).Key << " "; // вывод ключа

ObhodLeft(&((\*\*w).Left)); // обход левого поддерева

ObhodLeft(&((\*\*w).Right)); // обход правого поддерева

}

}

// Концевой обход дерева.

// w - указатель на корень дерева.

void TREE::ObhodEnd(node\*\* w)

{

if (\*w != NULL)//если дерево не пустое

{

ObhodEnd(&((\*\*w).Left)); // обход левого поддерева

ObhodEnd(&((\*\*w).Right)); // обход правого поддерева

cout << (\*\*w).Key << " "; // вывод ключа

}

}

// Обратный обход дерева.

// w - указатель на корень дерева.

void TREE::ObhodBack(node\*\* w)

{

if (\*w != NULL)//если дерево не пустое

{

ObhodBack(&((\*\*w).Left)); // обход левого поддерева

cout << (\*\*w).Key << " "; // вывод ключа

ObhodBack(&((\*\*w).Right)); // обход правого поддерева

}

}

void TREE::CleanTree(node\*\* w)

//Очистка дерева.

//\*w - указатель на корень дерева.

{

if (\*w != NULL)//если дерево не пустое

{

CleanTree(&((\*\*w).Left)); // Очистка левого поддерева.

CleanTree(&((\*\*w).Right)); // Очистка правого поддерева.

delete w; // Освобождение памяти, занятой корнем поддерева.

}

}

void TREE::Vyvod(node\*\* w, int l)

//Изображение дерева \*w на экране дисплея

// (рекурсивный алгоритм).

//\*w - указатель на корень дерева.

{

int i;

if (\*w != NULL)//если дерево не пустое

{

Vyvod(&((\*\*w).Right), l + 1); // Рекурсивный вызов функции для правого поддерева с увеличением отступа.

for (i = 1; i <= l; i++) cout << " "; // Вывод отступа для текущего уровня дерева.

cout << (\*\*w).Key << endl; // Вывод значения текущей вершины.

Vyvod(&((\*\*w).Left), l + 1); // Рекурсивный вызов функции для левого поддерева с увеличением отступа.

}

}

int TREE::Height(node\*\* w)

//Определение высоты бинарного дерева.

//\*w - указатель на корень дерева.

{

int h1, h2;

if (\*w == NULL) return (-1); // Для пустого дерева высота равна -1.

else

{

h1 = Height(&((\*\*w).Left)); // Определение высоты левого поддерева.

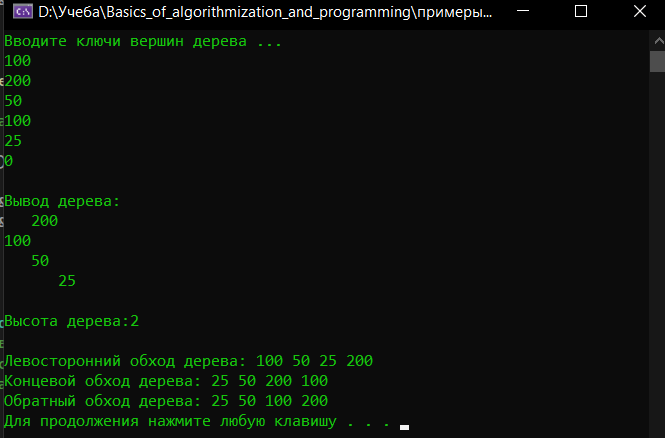
h2 = Height(&((\*\*w).Right)); // Определение высоты правого поддерева.

if (h1 > h2) return (1 + h1); // Высота дерева равна высоте левого поддерева + 1.

else return (1 + h2); // Высота дерева равна высоте правого поддерева + 1.

}

}



# **3.Пример программы, выполняющей построение бинарного дерева и его изображение на экране.**

#include <iostream> // подключение библиотеки ввода/вывода

using namespace std; // объявление пространства имен std

struct node

{

int Key; // Ключ вершины

int Count; // Количество вершин с таким же ключом

node\* Left; // Указатель на левое поддерево

node\* Right; // Указатель на правое поддерево

};

struct no // Звено стека

{

node\* elem; // Информационное поле.

int ch; // Уровень вершины.

no\* sled; // Указатель на вершину.

};

class TREE// Объявляем класс для бинарного дерева

{

private:

node\* Tree;// Указатель на корень дерева

void PushStack(no\*\*, node\*\*, int\*);// Помещение звена с элементами \*el и n в

// стек. \*stk - указатель на стек.

void PopStack(no\*\*, node\*\*, int\*); // Извлечение из стека звена

// с элементами \*t и n.

// \*stk - указатель на стек

void VyvodStack(no\*\*); // Вывод содержимого стека на экран дисплея.

// \*stk - указатель на стек.

public:

TREE() { Tree = new(node); (\*Tree).Right = NULL; }// Конструктор класса TREE, Создает объект дерева, устанавливает его корневой элемент, а правый потомок устанавливается в NULL

node\* GetTreeRight() { return (\*Tree).Right; }// Метод GetTreeRight() Возвращает указатель на правого потомка корневого элемента дерева

void TreeSearch(int); // Поиск вершины с информационным полем el в дереве с

// последующим (в случае неудачного поиска!) включением

// в дерево. Tree - указатель на корень дерева.

void VyvodTree(node\*); //Построение дерева, заданного указателем t,

//на экране дисплея (нерекурсивный алгоритм).

};

void main()//вызов главной функции с которой начинается выполнение программы

{

setlocale(LC\_ALL, "Rus");// устанавливаем локаль для вывода на русском языке

TREE A; // создаем объект класса TREE

int el;//переменная для ввода эелментов

cout << "Вводите значения информационных полей вершин: " << endl; // выводим приглашение на ввод элементов

cin >> el; // считываем первый элемент

while (el != 0) // запускаем цикл ввода элементов, пока не введем 0

{

A.TreeSearch(el); // добавляем элемент в дерево

cin >> el; // считываем следующий элемент

}

A.VyvodTree(A.GetTreeRight()); // выводим дерево на экран

cout << "\n"; // выводим пустую строку

system("PAUSE"); // ожидаем ввода пользователем любой клавиши

}

void TREE::TreeSearch(int el)

// Поиск вершины с информационным полем el в дереве с

// последующим (в случае неудачного поиска!) включением

// в дерево. Tree - указатель на корень дерева.

{

node\* p1, \* p2; // указатели на родителя и текущую вершину

int d; // флаг направления движения по дереву

p2 = Tree; // начинаем поиск с корня

p1 = (\*p2).Right; // идем к правому поддереву

d = 1; // инициализируем флаг направления движения

while (p1 != NULL && d != 0) // пока не дошли до конца дерева или не нашли искомый элемент

{

p2 = p1; // переходим на следующую вершину

if (el < (\*p1).Key) { p1 = (\*p1).Left; d = -1; } // если искомый элемент меньше, чем текущий, идем налево

else if (el > (\*p1).Key) { p1 = (\*p1).Right; d = 1; } // если искомый элемент больше, чем текущий, идем направо

else d = 0; // если элемент найден, устанавливаем флаг в 0

}

if (d == 0) (\*p1).Count = (\*p1).Count + 1; // если элемент найден, увеличиваем его счетчик

else // если элемент не найден

{

p1 = new(node); // выделяем память под новую вершину

(\*p1).Key = el; // задаем ключ новой вершины

(\*p1).Left = NULL; // инициализируем указатель на левое поддерево

(\*p1).Right = NULL; // инициализируем указатель на правое поддерево

(\*p1).Count = 1;// Счетчик встречаемости равен 1.

if (d < 0) (\*p2).Left = p1; // Добавляем новую вершину в левое поддерево родителя.

else (\*p2).Right = p1; // Добавляем новую вершину в правое поддерево родителя.

}

}

void TREE::VyvodTree(node\* t)

//Построение дерева, заданного указателем t,

//на экране дисплея (нерекурсивный алгоритм).

{

no\* stk, \* stk1; // Указатели на стеки

node\* u; // Указатель на текущую вершину

int i, n; // Счетчики

stk = stk1 = NULL; // Инициализация указателей на стеки

n = 0; // Инициализация счетчика

while (t != NULL) // Пока не обошли все вершины дерева

{

// Добавляем текущую вершину во второй стек

PushStack(&stk1, &t, &n);

if ((\*t).Right != NULL) // Если есть правый потомок

{

if ((\*t).Left != NULL) // Если есть и левый потомок

PushStack(&stk, &((\*t).Left), &n); // Добавляем левый потомок в первый стек

t = (\*t).Right; // Переходим к правому потомку

}

else // Если нет правого потомка

{

if ((\*t).Left != NULL) // Если есть левый потомок

{

if (stk1 != NULL) // Если во втором стеке есть элементы

{

PopStack(&stk1, &u, &n); // Извлекаем последний элемент из второго стека

for (i = 0; i <= n; i++) cout << " "; // Отступы для вывода на экран

cout << (\*u).Key << endl; // Выводим значение ключа на экран

}

t = (\*t).Left; // Переходим к левому потомку

}

else if (stk == NULL) // Если нет левого потомка и первый стек пуст

t = NULL; // Завершаем цикл

else // Если нет левого потомка и первый стек не пуст

{

// Пока вершина в первом стеке не является родительской для текущей вершины

while ((\*stk).elem != (\*((\*stk1).elem)).Left)

{

PopStack(&stk1, &u, &n); // Извлекаем последний элемент из второго стека

for (i = 0; i <= n; i++) cout << " "; // Отступы для вывода на экран

cout << (\*u).Key << endl; // Выводим значение ключа на экран

}

PopStack(&stk1, &u, &n); // Извлекаем последний элемент из второго стека

for (i = 0; i <= n; i++) cout << " "; // Отступы для вывода на экран

cout << (\*u).Key << endl; // Выводим значение ключа на экран

PopStack(&stk, &t, &n); // Извлекаем последний элемент из первого стека

}

}

n = n + 1;//увеличение счетчика на единицу

}

VyvodStack(&stk1);//вывод стека на экран

}

void TREE::PushStack(no\*\* stk, node\*\* el, int\* n)

// Функция для помещения звена с элементами \*el и \*n в стек.

// stk - указатель на стек.

{

no\* q;//объявление рабочего указателя

q = new(no); // Выделяем память для нового звена в стеке.

(\*q).elem = \*el; // Устанавливаем указатель на элемент в звене стека.

(\*q).ch = \*n; // Устанавливаем число потомков у узла в звене стека.

(\*q).sled = \*stk; // Устанавливаем указатель на предыдущее звено стека в текущее.

\*stk = q; // Устанавливаем указатель на текущее звено стека.

}

void TREE::PopStack(no\*\* stk, node\*\* t, int\* n)

// Функция для извлечения из стека звена с элементами \*t и \*n.

// stk - указатель на стек.

{

no\* q;//объявление рабочего указателя

if (\*stk != NULL) // Если стек не пустой

{

\*t = (\*\*stk).elem; // Устанавливаем указатель на элемент из звена стека.

\*n = (\*\*stk).ch; // Устанавливаем число потомков у узла из звена стека.

q = \*stk; // Сохраняем указатель на текущее звено стека.

\*stk = (\*\*stk).sled; // Устанавливаем указатель на предыдущее звено стека в текущее.

delete q; // Освобождаем память для звена, которое удаляем из стека.

}

}

void TREE::VyvodStack(no\*\* stk)

// Вывод содержимого стека на экран дисплея.

// \*stk - указатель на стек.

{

node\* k; // указатель на узел

int i, n; // переменные для отступов и уровня узла

while (\*stk != NULL)

{

k = (\*\*stk).elem; // получаем элемент из вершины стека

n = (\*\*stk).ch; // получаем количество пробелов для отступа

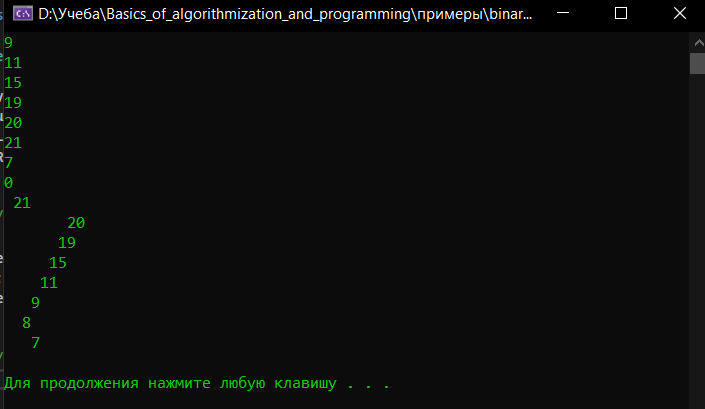
for (i = 0; i <= n; i++) cout << " "; // выводим отступы

cout << (\*k).Key << endl; // выводим ключ узла

\*stk = (\*\*stk).sled; // переходим к следующему элементу стека

}

}

****

# **4.Пример программы, иллюстрирующей поиск заданной вершины в дереве, добавление вершины в дерево, удаление вершины из дерева.**

#include <iostream> // подключение библиотеки ввода/вывода

using namespace std; // объявление пространства имен std

#define TRUE 1 //определение константы TRUE равной 1

#define FALSE 0 //определение константы FALSE равной 0

struct node

{

int Key; // Ключ вершины

int Count; // Количество вершин с таким же ключом

node\* Left; // Указатель на левое поддерево

node\* Right; // Указатель на правое поддерево

};

class TREE

{

private:

node\* Tree;//Указатель на корень дерева.

node\* Res;//Указатель на найденную вершину.

int B; //Признак нахождения вершины в дереве.

//Поиск вершины в дереве (рекурсивный алгоритм).

void Search(int, node\*\*); //Поиск звена x в бинарном дереве

//вершина с заданным ключом имеет две исходящие дуги

void Delete\_1(node\*\*, node\*\*);

public:

TREE() { Tree = NULL; }

node\*\* GetTree() { return &Tree; }

void BuildTree();//Построение бинарного дерева.

//Вывод дерева на экран (рекурсивный алгоритм).

void Vyvod(node\*\*, int);

//Поиск вершины в дереве (нерекурсивный алгоритм).

int Poisk(int);

//Поиск вершины в дереве (рекурсивный алгоритм).

node\* Poisk\_1(int, node\*\*);

//Добавление вершины в дерево (нерекурсивный алгоритм).

void Addition(int);

// Удаление вершины из дерева.

void Delete(node\*\*, int);

};

void main()//вызов главной функции с которой начинается выполнение программы

{

setlocale(LC\_ALL, "Rus"); // установка локали, чтобы корректно работали кириллические символы в консоли

TREE A; // создание объекта класса TREE

int el; // переменная для хранения ключа вершины

A.BuildTree(); // построение дерева

A.Vyvod(A.GetTree(), 0); // вывод дерева на экран

cout << "Введите ключ вершины, которую нужно найти в дереве: ";

cin >> el; // ввод ключа вершины для поиска

if (A.Poisk(el)) // если вершина найдена

cout << "В дереве есть такая вершина!\n"; // вывод сообщения об успешном поиске

else

cout << "В дереве нет такой вершины!\n"; // вывод сообщения о неудачном поиске

cout << "Введите ключ вершины, которую нужно найти в дереве: ";

cin >> el; // ввод ключа вершины для поиска

if (A.Poisk\_1(el, A.GetTree()) != NULL) // если вершина найдена

cout << "В дереве есть такая вершина!\n"; // вывод сообщения об успешном поиске

else

cout << "В дереве нет такой вершины!\n"; // вывод сообщения о неудачном поиске

cout << "Введите ключ добавляемой вершины: ";

cin >> el; // ввод ключа добавляемой вершины

A.Addition(el); // добавление вершины в дерево

A.Vyvod(A.GetTree(), 0); // вывод дерева на экран

cout << "Введите ключ удаляемой вершины: ";

cin >> el; // ввод ключа удаляемой вершины

A.Delete(A.GetTree(), el); // удаление вершины из дерева

A.Vyvod(A.GetTree(), 0); // вывод дерева на экран

cout << "\n";

system("PAUSE"); // приостановка работы программы

}

void TREE::BuildTree()

//Построение бинарного дерева.

//Tree - указатель на вершину дерева.

{

int el;//переменная для ввода эелментов

cout << "Вводите ключи вершин дерева: \n"; // Запрос на ввод ключей вершин дерева.

cin >> el; // Чтение первого ключа.

while (el != 0) // Цикл продолжается, пока не будет введен 0.

{

Search(el, &Tree); // Добавление вершины с ключом el в дерево.

cin >> el; // Чтение следующего ключа.

}

}

void TREE::Vyvod(node\*\* w, int l)

//Изображение дерева w на экране дисплея

// (рекурсивный алгоритм).

//\*w - указатель на корень дерева.

{

int i;//счетчик для уровня дерева

if (\*w != NULL) // Если корень не пустой,

{

Vyvod(&((\*\*w).Right), l + 1); // Рекурсивный вызов функции для правого поддерева.

for (i = 1; i <= l; i++) cout << " "; // Отступ слева для текущей вершины.

cout << (\*\*w).Key << endl; // Вывод ключа текущей вершины.

Vyvod(&((\*\*w).Left), l + 1); // Рекурсивный вызов функции для левого поддерева.

}

}

void TREE::Search(int x, node\*\* p)

// Поиск звена x в бинарном дереве со вставкой (рекурсивный алгоритм).

// \*p - указатель на вершину дерева.

{

if (\*p == NULL) //если дерево не пустое

{

// Вершины в дереве нет; включить ее.

\*p = new(node); // выделение памяти под новое звено

(\*\*p).Key = x; // присвоение значения ключу звена

(\*\*p).Count = 1; // инициализация количества

(\*\*p).Left = (\*\*p).Right = NULL; // инициализация указателей на левое и правое поддеревья

}

else

if (x < (\*\*p).Key)// если x меньше ключа текущего звена, переходим в левое поддерево

Search(x, &((\*\*p).Left));// Рекурсивный вызов для левой подветки дерева.

else

if (x > (\*\*p).Key)// если x больше ключа текущего звена, переходим в правое поддерево

Search(x, &((\*\*p).Right));// Рекурсивный вызов для правой подветки дерева.

else// если x равен ключу текущего звена, увеличиваем значение количества

(\*\*p).Count += 1;// Увеличение счетчика, если ключ уже присутствует в дереве.

}

void TREE::Addition(int k)

// Поиск звена k в бинарном дереве со вставкой (нерекурсивный алгоритм).

// Tree - указатель на вершину дерева.

{

node\* s;//рабочий указатель

Poisk(k); // ищем звено с ключом k

if (!B)

{ // если звено не найдено, вставляем новое

s = new(node); // выделение памяти под новое звено

(\*s).Key = k; // присвоение значения ключу звена

(\*s).Count = 1; // инициализация количества

(\*s).Left = (\*s).Right = NULL; // инициализация указателей на левое и правое поддеревья

if (Tree == NULL) Tree = s; // если дерево пустое, новое звено становится корнем

else

if (k < (\*Res).Key) (\*Res).Left = s; // если k меньше ключа найденного звена, новое звено становится левым потомком найденного звена

else (\*Res).Right = s; // если k больше ключа найденного звена, новое звено становится правым потомком найденного звена

}

else (\*Res).Count += 1; // если звено найдено, увеличиваем значение количества

}

int TREE::Poisk(int k)

// Поиск вершины с ключом k в дереве

// (нерекурсивный алгоритм).

// Tree - указатель на бинарное дерево.

// Res - указатель на найденную вершину

// или на лист, к которому можно присоединить новую вершину.

{

node\* p, \* q; //рабочие указатели

B = FALSE; //флаг нахождения элемента в дереве

p = Tree;//устанавливаем указатель на начало дерева

if (Tree != NULL) //если дерево не пустое

do

{

q = p; //указатель на родительский узел

if ((\*p).Key == k) B = TRUE; //элемент найден

else//элемент не найден

{

q = p;

if (k < (\*p).Key) p = (\*p).Left; //двигаемся влево

else p = (\*p).Right; //двигаемся вправо

}

} while (!B && p != NULL); //пока не найден элемент и не достигнут конец дерева

Res = q; //указатель на последний просмотренный узел

return B; //возвращаем флаг нахождения элемента

}

node\* TREE::Poisk\_1(int k, node\*\* p)

// Поиск вершины с ключом k в дереве

// (рекурсивный алгоритм).

// \*p - указатель на корень дерева.

{

if (\*p == NULL) {

return (NULL); // Если корень пустой, вернуть NULL

}

else if ((\*\*p).Key == k) {// Если найден ключ k

return (\*p); //вернуть указатель на эту вершину

}

else if (k < (\*\*p).Key) {// если k меньше ключа текущего звена

return Poisk\_1(k, &((\*\*p).Left)); // Рекурсивный поиск в левом поддереве

}

else {// если k больше ключа текущего звена

return Poisk\_1(k, &((\*\*p).Right)); // Рекурсивный поиск в правом поддереве

}

}

void TREE::Delete(node\*\* p, int k)

// Удаление вершины k из бинарного дерева.

// \*p - указатель на корень дерева.

{

node\* q;//объявление рабочего указателя

if (\*p == NULL) {//если дерево пустое

cout << "Вершина с заданным ключом не найдена!\n"; // Вершина с заданным ключом не найдена

}

else if (k < (\*\*p).Key) {// если k меньше ключа текущего звена

Delete(&((\*\*p).Left), k); // Рекурсивный вызов для левого поддерева

}

else if (k > (\*\*p).Key) {// если k больше ключа текущего звена

Delete(&((\*\*p).Right), k); // Рекурсивный вызов для правого поддерева

}

else {// Если вершина с ключом k найдена, то выполняется удаление этой вершины

q = \*p;

if ((\*q).Right == NULL) {// Если у вершины q нет правого поддерева, то её левое поддерево становится на место q

\*p = (\*q).Left;//левое поддерево становится на место q

delete q; // Удаление вершины q, которая не имеет правого поддерева

}

else if ((\*q).Left == NULL) {// Если у вершины q нет левого поддерева, то её правое поддерево становится на место q

\*p = (\*q).Right;//правое поддерево становится на место q

delete q; // Удаление вершины q, которая не имеет левого поддерева

}

else {// Если у вершины q есть и левое, и правое поддеревья, то удаляем у неё максимальный элемент в левом поддереве

Delete\_1(&((\*q).Left), &q); // Удаление вершины q, которая имеет и левое, и правое поддеревья

}

}

}

void TREE::Delete\_1(node\*\* r, node\*\* q)

{

node\* s;//объявление рабочего указателя

if ((\*\*r).Right == NULL) // Если правое поддерево \*r пусто

{

(\*\*q).Key = (\*\*r).Key; (\*\*q).Count = (\*\*r).Count; // Копируем ключ и счётчик из \*r в \*q

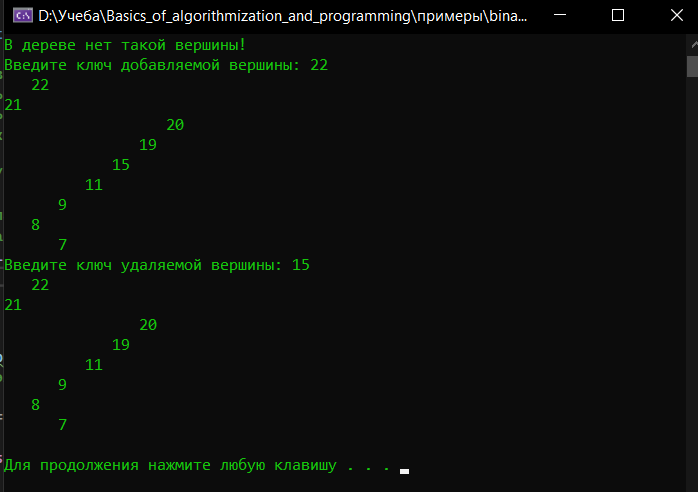
\*q = \*r; // Переносим ссылку на \*r в \*q

s = \*r; \*r = (\*\*r).Left; delete s; // Удаляем \*r и делаем (\*r)-левое поддерево единственным поддеревом \*\*q.

}

else Delete\_1(&((\*\*r).Right), q); // Иначе выполняем удаление в правом поддереве \*\*r

}



# **5.пpогpамма постpоения деpева-фоpмулы по заданной постфиксной фоpмуле и использование постpоенного деpева для получения пpефиксной и инфиксной фоpмул.**

#include <stdio.h>// Подключение библиотеки stdio.h

#include <conio.h>// Подключение библиотеки conio.h

#include <iostream>// Подключение библиотеки iostream

using namespace std;

struct Uzel //Тип узла дерева.

{

char Key; // Символ

Uzel\* Left; // Указатель на левый узел

Uzel\* Right; // Указатель на правый узел

};

struct zveno //Тип звена стека.

{

Uzel\* Element; // Указатель на узел

zveno\* Sled; // Указатель на следующее звено стека

};

class Tree// Описание класса дерева

{

private:

Uzel\* Root; //Указатель на корень дерева.

zveno\* Stack;// Указатель на стек

public:

Tree(); // Конструктор класса

void Udalenie(Uzel\*\*); // Удаление дерева

void V\_stack(Uzel\*); // Добавление элемента в стек

void PrintTree(Uzel\*, int); //Вывод деpева на экpан дисплея

void Print\_Tree\_Left(Uzel\*, int); //Левостоpонний обход бинаpного деpева

void Print\_Tree\_End(Uzel\*, int); //Концевой обход бинаpного деpева

void Print\_Tree\_Back(Uzel\*, int); //Обpатный обход бинаpного деpева

Uzel\* GetTree() { return Root; };// Получение указателя на корень дерева

};

void Tree::V\_stack(Uzel\* Elem)

//Добавление элемента в стек.

{

zveno\* q = new (zveno); //Выделение памяти для нового звена стека.

q->Element = Elem; //Присваивание элементу звена значение переданного элемента.

q->Sled = Stack; //Установка указателя на следующий элемент стека на текущий верхний элемент.

Stack = q; //Перестановка указателя на верхний элемент на добавленный элемент.

}

void Tree::Udalenie(Uzel\*\* tmp)

//Удаление элемента из стека.

{

zveno\* q; //Указатель на элемент стека, который нужно удалить.

if (Stack != NULL) //Если стек не пуст.

{

(\*tmp) = Stack->Element; //Присваивание удаляемому элементу значения верхнего элемента стека.

q = Stack; //Установка указателя на удаляемый элемент на верхний элемент стека.

Stack = Stack->Sled; //Перестановка указателя на верхний элемент на следующий элемент стека.

delete q; //Удаление верхнего элемента стека.

}

}

void Tree::PrintTree(Uzel\* w, int l)

//Рекурсивный вывод дерева на экран дисплея в виде дерева.

{

if (w != NULL) //Если элемент не равен NULL.

{

PrintTree(w->Right, l + 1); //Рекурсивный вывод правого поддерева дерева.

for (int i = 1; i <= l; i++) cout << " "; //Отступы перед выводом значения узла.

cout << w->Key << endl; //Вывод значения узла.

PrintTree(w->Left, l + 1); //Рекурсивный вывод левого поддерева дерева.

}

}

void Tree::Print\_Tree\_Left(Uzel\* w, int l)// Левосторонний обход бинарного дерева.

{

if (w != NULL)//если дерево не пустое

{

cout << w->Key << " "; // Выводим значение узла.

Print\_Tree\_Left(w->Left, l + 1); // Рекурсивный вызов для левого поддерева.

Print\_Tree\_Left(w->Right, l + 1); // Рекурсивный вызов для правого поддерева.

}

}

void Tree::Print\_Tree\_End(Uzel\* w, int l)// Концевой обход бинарного дерева.

{

if (w != NULL)//если дерево не пустое

{

Print\_Tree\_End(w->Left, l + 1); // Рекурсивный вызов для левого поддерева.

Print\_Tree\_End(w->Right, l + 1); // Рекурсивный вызов для правого поддерева.

cout << w->Key << " "; // Выводим значение узла.

}

}

void Tree::Print\_Tree\_Back(Uzel\* w, int l)

// Обратный обход бинарного дерева.

{

if (w != NULL)//если дерево не пустое

{

cout << "("; // Начинаем выражение для текущего узла.

Print\_Tree\_Back(w->Left, l + 1); // Рекурсивный вызов для левого поддерева.

cout << w->Key << " "; // Выводим значение узла.

Print\_Tree\_Back(w->Right, l + 1); // Рекурсивный вызов для правого поддерева.

cout << ")"; // Заканчиваем выражение для текущего узла.

}

}

Tree::Tree()

{

Stack = NULL; //Вначале опустошим стек.

//Фоpмиpование заглавного звена деpева.

Root = new (Uzel);

Root->Right = NULL;

}

void main()

{

setlocale(LC\_ALL, "Rus");

//Установка текущего языка ввода/вывода на русский.

char Formula\_Post[30];//Создание массива для хранения постфиксной формулы.

char k; //Вспомогательная пеpеменная.

Uzel\* Ukazatel = NULL;//Установка указателя на NULL.

cout << "Введите фоpмулу, записанную в постфиксной фоpме... \n"; //Вывод сообщения на экран.

gets\_s(Formula\_Post); //Чтение постфиксной формулы из консоли и сохранение в массиве Formula\_Post.

strrev(Formula\_Post); //Переворот строки Formula\_Post.

cout << "Пpиступим к постpоению деpева-фоpмулы...\n"; //Вывод сообщения на экран.

Tree A; //Создание объекта класса Tree.

Uzel\* Temp = A.GetTree(); //Текущий указатель.

//Фоpмиpование деpева поиска и вывод его на экpан.

for (int i = 0; i < strlen(Formula\_Post); i++)

{

k = Formula\_Post[i];

if (strchr("+-\*/^", k) != NULL)//Пеpеходим к анализу символа k.

{ //Символ - опеpация.

if (Temp->Right == NULL) //Отсутствует пpавая дуга.

{

Temp->Right = new (Uzel);//Резеpвиpование места для вставляемого узла.

Temp = Temp->Right;// Установка указателя на вставляемый узел.

Temp->Key = k;//Инициализация вставляемого узла.

Temp->Left = Temp->Right = NULL;//установка дочерних элемнтов в NULL

A.V\_stack(Temp);//Ссылка на пpедыдущий узел --> стек.

}

else //Есть пpавая дуга.

{

Temp->Left = new (Uzel);//Резеpвиpование места для вставляемого узла.

Temp = Temp->Left;// Установка указателя на вставляемый узел.

Temp->Key = k;// Инициализация вставляемого узла.

Temp->Left = Temp->Right = NULL;//установка дочерних элемнтов в NULL

A.V\_stack(Temp);//Ссылка на пpедыдущий узел --> стек.

}

}

else //Символ - опеpанд.

if (Temp->Right == NULL) //Отсутствует пpавая дуга.

{

Temp->Right = new (Uzel);//Резеpвиpование места для вставляемого узла.

Temp = Temp->Right;// Установка указателя на вставляемый узел.

Temp->Key = k;//Инициализация вставляемого узла.

Temp->Left = Temp->Right = NULL;//установка дочерних элемнтов в NULL

A.Udalenie(&Ukazatel);// Текущий указатель "возвpащается" назад.

Temp = Ukazatel;

}

else //Есть пpавая дуга.

{

Temp->Left = new (Uzel);//Резеpвиpование места для вставляемого узла.

Temp = Temp->Left;// Установка указателя на вставляемый узел.

Temp->Key = k;// Инициализация вставляемого узла.

Temp->Left = Temp->Right = NULL;;//установка дочерних элемнтов в NULL

A.Udalenie(&Ukazatel);// Текущий указатель "возвpащается" назад.

Temp = Ukazatel;

}

} //Конец for.

cout << "\nКонтpольный вывод деpева-фоpмулы...\n";

A.PrintTree(A.GetTree()->Right, 0);//Вывод контрольного дерева-формулы.

cout << "Пеpед Вами фоpмула, записанная в инфиксной фоpме...\n";

A.Print\_Tree\_Back(A.GetTree()->Right, 0);//Вывод формулы, записанной в инфиксной форме.

cout << endl;//переход на новую строку

cout << "------------------------------------------ \n";

cout << "Пеpед Вами фоpмула, записанная в пpефиксной фоpме...\n";

A.Print\_Tree\_Left(A.GetTree()->Right, 0);//Вывод формулы, записанной в префиксной форме.

cout << endl;//переход на новую строку

cout << "------------------------------------------ \n";

cout << "Пеpед Вами фоpмула, записанная в постфиксной фоpме...\n";

A.Print\_Tree\_End(A.GetTree()->Right, 0);//Вывод формулы, записанной в постфиксной форме.

cout << "\n";//переход на новую строку

system("PAUSE");//Ожидание нажатия клавиши для завершения работы программы.

}

