**Цель работы:** изучение принципов организации и построения древовидных структур данных, а также программирование операций включения элемента, удаления элемента и обхода дерева.

**Постановка задачи:** разработать программу формирования дерева двоичного поиска и выполнения заданной операции с ним – удалить элемент с ключом k. Предусмотреть получение значение элемента дерева.

**Описание переменных и методов**

В программе использовалось следующие переменные:

**Private** Node<T1, T2> root – переменная, хранящая экзмепляр дерева

**static** **class** Node<T1, T2> ) – класс реализующий структуру узла

**public** **void** add(T1 k, T2 v) – метод, реализующий добавление элементов

**public** T2 get(T1 k) – метод, реализует возврат значения узла

**public** **void** remove(T1 k) – метод, реализующий удаление элемента

**Текст программы.**

**package** com.worm2fed.sp\_lab6;

**public** **class** BSTree<T1 **extends** Comparable<T1>, T2> {

// T1 extends Comparable<T1> необходимо для возможности сравнивать ключи

// т.е. в левом поддереве хранятся ключи меньшие, чем в корне

// а в правом - большие

**static** **class** Node<T1, T2> {

T1 key;

T2 value;

Node<T1, T2> left, right;

Node(T1 key, T2 value) {

**this**.key = key;

**this**.value = value;

}

}

**private** Node<T1, T2> root = **null**;

**public** T2 get(T1 k) {

Node<T1, T2> x = root;

**while** (x != **null**) {

// = 0 - если значения равны;

// < 0 - если вызываемый объект меньше параметра;

// > 0 - если вызываемый объект больше параметра.

**int** cmp = k.compareTo(x.key);

**if** (cmp == 0)

**return** x.value;

**if** (cmp < 0)

x = x.left;

**else**

x = x.right;

}

**return** **null**;

}

**public** **void** add(T1 k, T2 v) {

Node<T1, T2> x = root, y = **null**;

**while** (x != **null**) {

**int** cmp = k.compareTo(x.key);

**if** (cmp == 0) {

x.value = v;

**return**;

} **else** {

y = x;

**if** (cmp < 0)

x = x.left;

**else**

x = x.right;

}

}

Node<T1, T2> newNode = **new** Node<T1, T2>(k, v);

**if** (y == **null**)

root = newNode;

**else** {

**if** (k.compareTo(y.key) < 0)

y.left = newNode;

**else**

y.right = newNode;

}

}

**public** **void** remove(T1 k) {

Node<T1, T2> x = root, y = **null**;

**while** (x != **null**) {

**int** cmp = k.compareTo(x.key);

**if** (cmp == 0)

**break**;

**else** {

y = x;

**if** (cmp < 0)

x = x.left;

**else**

x = x.right;

}

}

**if** (x == **null**)

**return**;

**if** (x.right == **null**) {

**if** (y == **null**)

root = x.left;

**else** {

**if** (x == y.left)

y.left = x.left;

**else**

y.right = x.left;

}

} **else** {

Node<T1, T2> leftMost = x.right;

y = **null**;

**while** (leftMost.left != **null**) {

y = leftMost;

leftMost = leftMost.left;

}

**if** (y != **null**)

y.left = leftMost.right;

**else**

x.right = leftMost.right;

x.key = leftMost.key;

x.value = leftMost.value;

}

}

**public** **static** **void** main(String[] args) {

BSTree<Integer, Integer> tree = **new** BSTree<Integer, Integer>();

tree.add(1, 3);

tree.add(2, 4);

tree.add(5, 10);

System.***out***.println(tree.get(5));

tree.remove(5);

System.***out***.println(tree.get(5));

}

}

**Пример работы:**

Ключ: 5 Значение: 10

Ключ: 1 Значение: 3

Ключ: 2 Значение: 4

Ключ: 5 Значение: null

**Вывод:** Изучили принципы работы с деревом двоичного поиска.