### Постановка задачи.

- Реализовать двоичное дерево на Java.  
- Добавить поддержу операций добавления, удаления, поиска.

### Введение.

Двоичные деревья - одна из самых важных структур данных в программировании. Они широко используются в различных задачах, таких как поиск, сортировка, обработка и многое другое. Сегодня мы проведем обзор реализации двоичных деревьев на языке программирования Java, а также обсудим поддержку операций добавления, удаления и поиска.

### Решение.

Двоичное дерево - это иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым потомками.

Реализация двоичного дерева на java, будет осуществлена с помощью структур (экземпляров класса с только им присущими полями). Так класс дерево обладает ссылкой на первый узел, а узлы – ссылками на левый и правый дочерние узлы.

Операции построены рекурсивными способами.

**package** tree;

//Данный класс символизирует узел.

**class** **Node** {

**int** value;

Node left;

Node right;

//Данный конструктор позволяет создавать узел (экземпляр этого класса), с переданным в него значением.

**public** **Node**(**int** value) {

**this**.value = value;

right = **null**;

left = **null**;

}

}

**package** tree;

//Класс, символизирует собой двоичное дерево.

**public** **class** **BinaryTree** {

//Логика как у связного списка, дерево хранит ссылку на первый элемент.

Node root;

//Данный конструктор позволяет создавать экземпляры этого класса.

**public** **BinaryTree**() {

root = **null**;

}

//Вставка будет производится сначала в левый, потом в правый элемент.

//Данный метод запустится для определнного экземпляра дерева. Публичный класс запускает приватный, который работает с узлами.

**public** **void** **insert**(**int** value) {

//Вставка всегда должна начинаться с вершины.

root = insert(root, value);

}

//Методы с одинаковыми именами, но разным количеством аргументов не перезапишут друг-друга и не вызовут ошибку.

**private** Node **insert**(Node node, **int** value) {

//Если узла нет(="его значени нуль"). Вставка новго узла с новым значением

**if** (node == **null**) {

node = **new** Node(value);

//Вставка влево или вправо, в зависимости от наличия узлов

} **else** {

**if** (node.left == **null**) {

node.left = insert(node.left, value);

} **else** {

node.right = insert(node.right, value);

}

}

**return** node;

}

**public** **boolean** **search**(**int** value) {

//Поиск с вершины, конкретного дерева

**return** **search**(root, value);

}

**private** **boolean** **search**(Node current, **int** value) {

//Узла не существует в конце, а значит элемент так и не был найден.

**if** (current == **null**) {

**return** **false**;

}

//Если значени данного узла

**if** (value == current.value) {

**return** **true**;

}

//После нужно сделать проверку для узлов, находящихся ниже.

**return** (search(current.left, value)==**true**) ||( search(current.right, value));

}

**public** **void** **remove**(**int** value) {

root = removeRecursive(root, value);

}

**private** Node **removeRecursive**(Node current, **int** value) {

// Если текущий узел пустой, возвращаем null.

**if** (current == **null**) {

**return** **null**;

}

// Если значение текущего узла равно заданному значению,

// то удаляем текущий узел.

**if** (value == current.value) {

// Если текущий узел не имеет дочерних узлов,

// возвращаем null.

**if** (current.left == **null** && current.right == **null**) {

**return** **null**;

}

// Если текущий узел имеет только один дочерний узел,

// возвращаем этот дочерний узел.

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

**return** current.left;

}

**if** (current.left == **null**) {

**return** current.right;

}

//Если текущий элемент имеет два дочерних узла. Связываем отцовский узел

//с правым дочерним, а левый дочерний первому свободному из правого поддерева.

current.right.left=current.left;

**return** current.right;

}

current.left = removeRecursive(current.left, value);

current.right = removeRecursive(current.right, value);

**return** current;

}

}

**package** tree;

**public** **class** **App** {

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

BinaryTree tree = **new** BinaryTree();

tree.insert(**5**);

tree.insert(**2**);

tree.insert(**3**);

tree.insert(**7**);

tree.insert(**10**);

tree.insert(**12**);

tree.insert(**16**);

tree.insert(**18**);

tree.insert(**20**);

System.out.println("Наличие 2 "+tree.search(**2**));

System.out.println("Наличие 5 "+tree.search(**5**));

System.out.println("Наличие 20 "+tree.search(**20**));

System.out.println("Наличие 3 "+tree.search(**3**));

System.out.println("Удание 3");

tree.remove(**3**);

System.out.println("Наличие 3 "+tree.search(**3**));

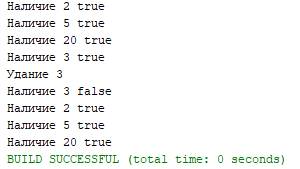
System.out.println("Наличие 2 "+tree.search(**2**));

System.out.println("Наличие 5 "+tree.search(**5**));

System.out.println("Наличие 20 "+tree.search(**20**));

}

}



### Заключение.

В заключении можно сказать, что реализация двоичных деревьев на Java - это незаменимый инструмент в программировании, который помогает эффективно работать с большим объемом данных. Он позволяет нам быстро находить необходимые элементы в массиве, сравнивать значения и производить множество других операций. Благодаря добавлению поддержки операций добавления, удаления и поиска мы получаем еще больше гибкости в работе с этой структурой данных.