Урок 6

Деревья

Рассмотрим работу с двоичными деревьями

Введение

Деревья — это структура данных, которая часто используется в программировании и сочетает в себе преимущества массивов и списков: быструю вставку, поиск и удаление элементов.

Чтобы вставить элемент в упорядоченный массив, ему сначала необходимо определить позицию вставки. Затем следует передвинуть все элементы, которые больше вставляемого, на одну позицию вправо. Все эти перемещения занимают много времени, в среднем **O(N/2)**.

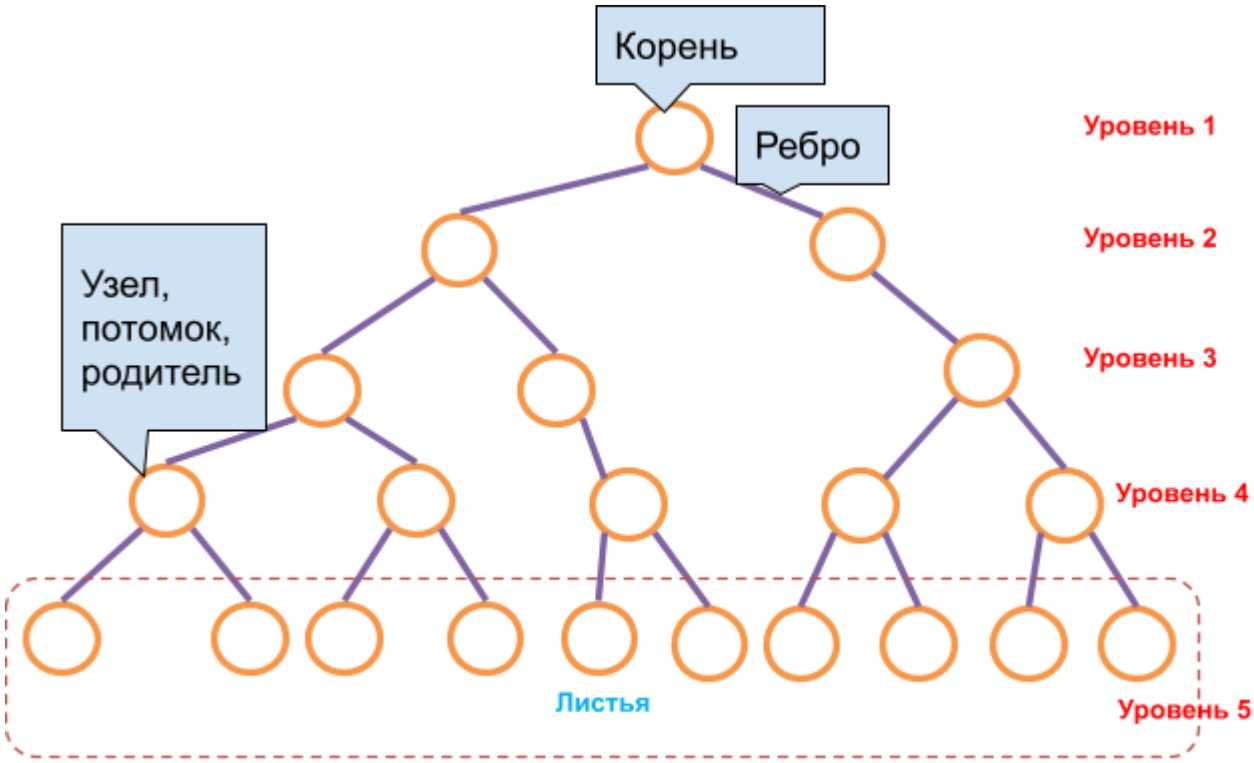
Если вставку и удаление элементов придется использовать часто, лучше воспользоваться связанным списком. Эти операции в списке выполняются за **O(1)**. Но вот поиск элемента может занять много времени, так как приходится перебирать все элементы по очереди, что может занять время **O(N)**.

В деревьях операции поиска, вставки и удаления элементов выполняются быстро. У этой структуры несколько типов: красно-черные деревья, деревья 2-3-4 или двоичные, которые рассмотрим на этом уроке.

Немного теории

Разберемся, из чего состоит двоичное дерево.

Основными компонентами дерева являются вершины и связи между ними. Связи еще называют ребрами, а вершины — узлами. У каждого узла может быть два дочерних элемента. Корень — это узел, который расположен на вершине дерева. Элементы в двоичном дереве разделяются на родителей и потомков. Каждый узел, кроме корневого, имеет одно ребро, уходящее вверх к другому узлу — родителю. Через ребра родители связываются с потомками. Если в дереве есть узел потомков, его называют листом. Каждое поколение находится на своем уровне: как правило, корневой элемент расположен на нулевом, его потомки — на первом, а потомки потомков — на втором, и так далее.



Создание двоичного дерева

Какую структуру данных выбрать за основу для реализации двоичного дерева? Самый распространенный способ — хранение в памяти компьютера через несмежные блоки памяти, то есть списки. Также можно хранить узлы дерева в массиве, располагая их в определенных позициях. Рассмотрим способ, в котором узлы в памяти компьютера представлены в виде списка.

Создание узлов

Чтобы создать узлы дерева, добавим класс **Node**, который будет содержать данные о сотруднике предприятия: порядковый номер, ФИО и возраст. Назовем такой класс **Person**. В узле дерева будет храниться объект типа **Person**. Так как дерево двоичное, необходимо завести два поля для хранения левого и правого потомка. Также в классе надо реализовать метод **display**, который выводит информацию об узле. Класс будет иметь следующий вид:

```
class Node {
    public Person person;
    public Node leftChild;
    public Node rightChild;

    public void display(){
        System.out.println("Name: "+person.name+", age: "+person.age);
    }
}
```

Также создадим класс **Person** с конструктором, в который передадим имя и возраст человека:

```
class Person {
    public String name;
    public int id;
    public int age;

    public Person(){

    }

    public Person(String name, int id, int age) {
        this.name = name;
        this.id = id;
        this.age = age;
    }
}
```

Создание дерева

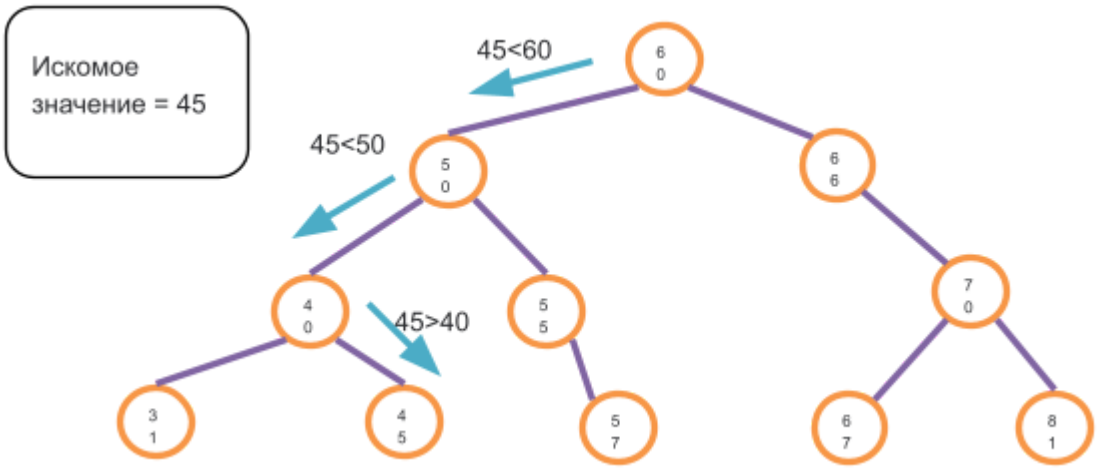
Создадим класс **Tree**, в котором реализуем методы для удаления, добавления, поиска, обхода узлов и вывода содержимого дерева.

```
class Tree {
    private Node root;

    public Node find(int key){
        // Тело метода
    }

    public void insert(Person person){
        // Тело метода
    }

    public boolean delete(int id){
        // Тело метода
    }

    public void displayTree(){
        // Тело метода
    }
}
```

В поле **root** будет храниться корневой элемент дерева, у которого есть ссылки на его потомков. Метод **find** возвращает найденный по идентификатору человека узел. Метод **insert** осуществляет вставку нового узла в дерево. Один из самых сложных методов в деревьях — **delete**, который удаляет узел из дерева по указанному идентификатору. Метод **displayTree** выводит на экран все узлы дерева. Начнем с простой задачи — поиска узла.

Поиск узла

Алгоритм поиска узла с заданным ключом — тривиальная задача. Ключом может быть поле **id** класса **Person**. При поиске узла алгоритм будет обходить дерево, обращаясь к потомкам узла. Обход начинается с корня дерева. Если ключ в корневом узле больше ключа в искомом элементе, поиск будет выполняться в левой половине дерева, в ином случае — в правой. Операция будет продолжаться, пока не найдется искомый элемент. Если такого элемента не существует, будет возвращено значение null.



Представим, что искомое значение равно 45. Начиная с корня дерева, сравниваем число 45 с 60. 45 меньше 60, значит идем в левую часть дерева. На следующем шаге сравниваем наше число 45 с 50. 45 меньше 50 — идем в левую часть. На следующем шаге сравниваем 45 с 40. 45 больше 40 — идем в правую часть. И на следующем шаге, сравнивая числа 45 и 45, находим искомый узел.

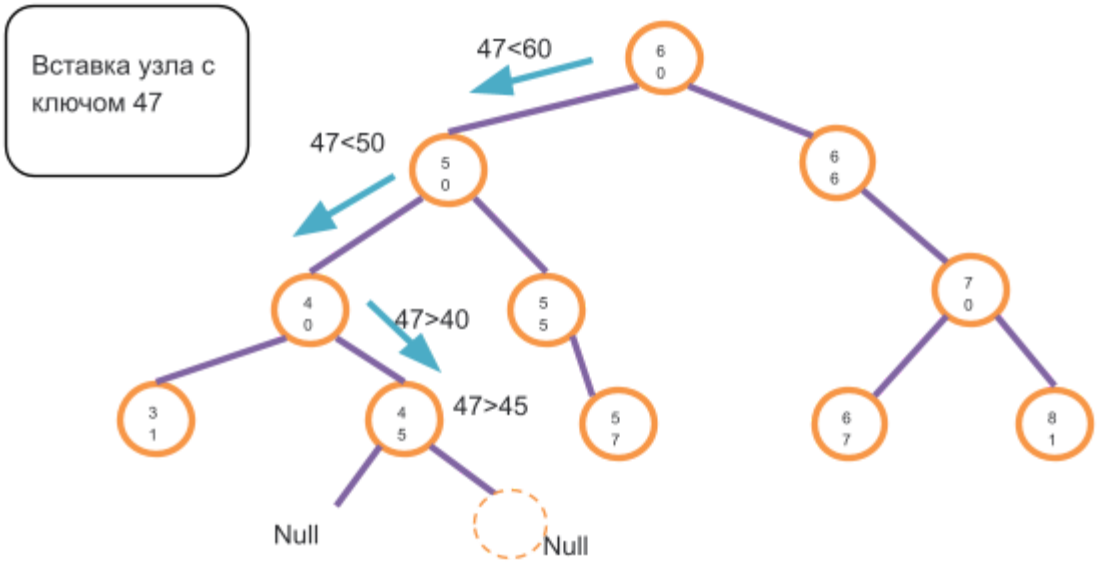
```
public Node find(int key){
      Node current = root;
      while (current.person.id != key) {
            if (key < current.person.id){
                  current = current.leftChild;
            } else {
                  current = current.rightChild;
            }

            if (current == null){
                  return null;
            }
      }
      return current;
}
```

В методе создается переменная **current**, соответствующая узлу дерева, в котором в данный момент идет поиск. В начале метода ей передается ссылка на корневой узел. Пока не будет найден нужный узел, алгоритм будет спускаться вниз по дереву. Если узел так и не обнаружится, метод вернет **null**.

Вставка узла

Вставку узла можно разделить на две части. Первая — это поиск места в дереве, куда будет вставлен новый элемент. Эта часть очень похожа на алгоритм поиска, когда искомый узел не был найден. Метод выполняет поиск родителя для нового узла.

Вторая часть — это вставка. Когда родитель будет найден, метод сравнит его ключ со значением ключа вставляемого узла, и если ключ родителя окажется меньше, новый узел будет добавлен как правый потомок, в противном случае — как левый.



Для вставки узла в дерево есть метод **insert**, в который передается вставляемый узел. Если он является первым в дереве, его ссылка передается в поле **root**. Если не первым, дальнейшие

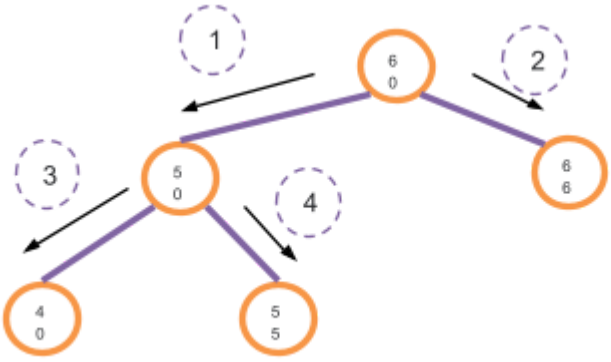
действия похожи на алгоритм поиска узла. Но надо учитывать, что в поиске при нахождении **null** считалось, что искомый узел не обнаружен, а в алгоритме вставки это будет означать, что найдено место, куда будет вставлен новый узел.

```
public void insert(Person person){
      Node node = new Node();
      node.person = person;
      if (root == null){
            root = node;
      } else {
            Node current = root;
            Node parent;
            while (true) {
                  parent = current;
                  if (person.id < current.person.id){
                        current = current.leftChild;
                        if (current == null){
                              parent.leftChild = node;
                              return;
                        }
                  } else {
                        current = current.rightChild;
                        if (current == null){
                              parent.rightChild = node;
                              return;
                        }
                  }
            }
      }

}
```

Обход дерева

Обход дерева — это посещение всех узлов в определенном порядке. На практике он используется редко, но представляет интерес с теоретической точки зрения. Алгоритмы обхода деревьев работают медленнее, чем вставка, удаление и поиск узлов. Выделяют три способа обхода дерева: прямой, симметричный и обратный.

При симметричном обходе узлы дерева перебираются в порядке возрастания ключей. Одно из возможных применений — создание отсортированного списка данных двоичного дерева. В нашей реализации симметричного обхода будем использовать рекурсию.



Разберем алгоритм симметричного обхода по действиям: ему необходимо вызвать самого себя для обхода левой стороны дерева, посетить узел и вызвать самого себя для обхода правой стороны.

```
private void inOrder(Node rootNode){
      if(rootNode != null){
            inOrder(rootNode.leftChild);
            rootNode.display();
            inOrder(rootNode.rightChild);
      }

}
```

В качестве второго действия (посещения узла) выполняется вывод информации о нем в консоль.

Алгоритмы обратного и прямого обхода дерева отличаются только последовательностью действий. Для прямого обхода выполняется:

1. Посещение узла.
2. Вызов самого себя для обхода левой стороны.
3. Вызов самого себя для обхода правой стороны.

Для обратного обхода:

1. Вызов самого себя для обхода левой стороны.
2. Вызов самого себя для обхода правой стороны.
3. Посещение узла.

Поиск минимума и максимума

Поиск минимума и максимума — самый простой алгоритм работы с двоичными деревьями. Из теории мы знаем, что самый левый элемент дерева является минимальным, а самый правый — максимальным.

Для поиска минимума создадим метод **min()**.

```
public Node min() {

      Node current, last = null;
      current = root;
      while (current != null) {
            last = current;
            current = current.leftChild;
      }

      return last;

}
```

Для поиска максимума — метод **max()**.

```
public Node max() {
```

```
        Node current, last = null;
        current = root;
        while (current != null) {
                last = current;
                current = current.rightChild;
        }

        return last;

}
```

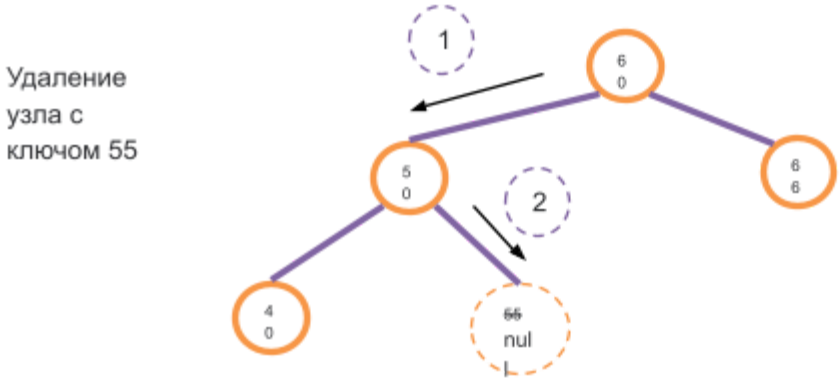
Удаление узла

Один из самых сложных алгоритмов, касающихся деревьев, — это удаление заданного узла. Для его реализации необходимо найти узел и удалить его, то есть изменить его значение на **null**. Алгоритм будет работать правильно, только если у удаляемого узла нет потомков.

Рассмотрим ситуации, которые могут возникать при удалении узла.

1. Удаляемый узел является листом (не имеет потомков).
2. Удаляемый узел имеет одного потомка.
3. Удаляемый узел имеет двух потомков.

Покажем реализацию самого простого случая, когда узел не имеет потомков. Так как в Java используется автоматический сборщик мусора, для удаления такого узла достаточно изменить ссылку на **null**.



Реализуем метод **delete** для удаления узла без потомков. Для начала выполним поиск удаляемого узла.

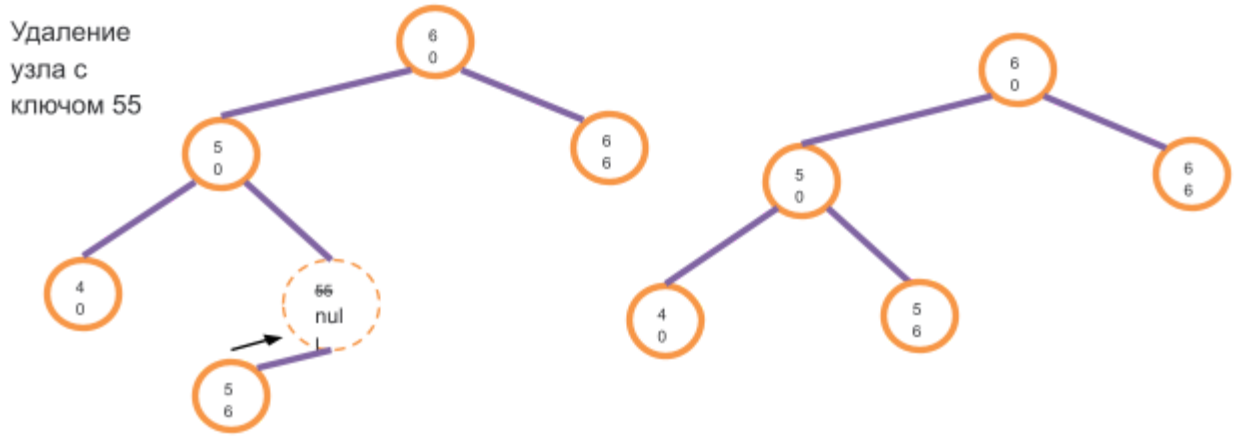
```
public boolean delete(int id) {
        Node current = root;
        Node parent = root;
        boolean isLeftChild = true;

        while (current.person.id != id) {
            parent = current;
            if (id < current.person.id){
                isLeftChild = true;
                current = current.leftChild;
            } else {
                isLeftChild = false;
```

```
                current = current.rightChild;
            }
            if (current == null){
                return false;
            }

        }
```

Когда удаляемый узел найден, проверяем, что он действительно не имеет потомков, и удаляем его. Если он является корневым, изменяем значение поля **root** на **null**.

```
if (current.leftChild == null && current.rightChild == null) {
      if (current == root) {
            root = null;
      } else if(isLeftChild) {
            parent.leftChild = null;
      } else {
            parent.rightChild = null;
      }

}
```

Если удаляемый узел имеет одного потомка, необходимо соединить родителя удаляемого узла с его потомком.



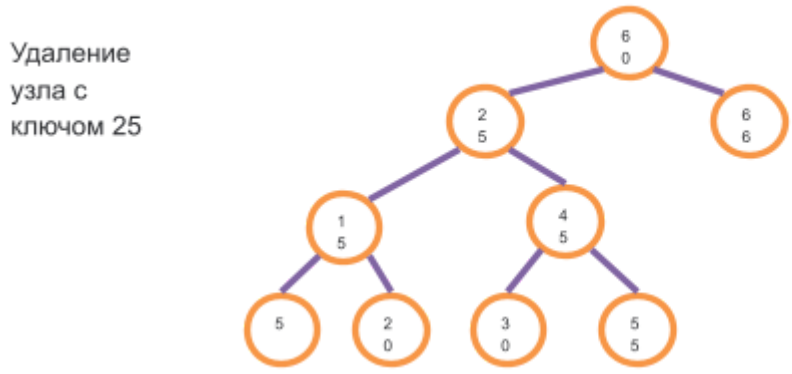
Если удаляемый узел является корневым, его ссылка заменяется ссылкой потомка.

Удаление узла, который имеет одного потомка:

```
// Если нет правого потомка
else if(current.rightChild == null) {
      if (current == root) {
            root = current.leftChild;
      } else if(isLeftChild) {
            parent.leftChild = current.leftChild;
      }else{
            parent.rightChild = current.leftChild;
      }
```

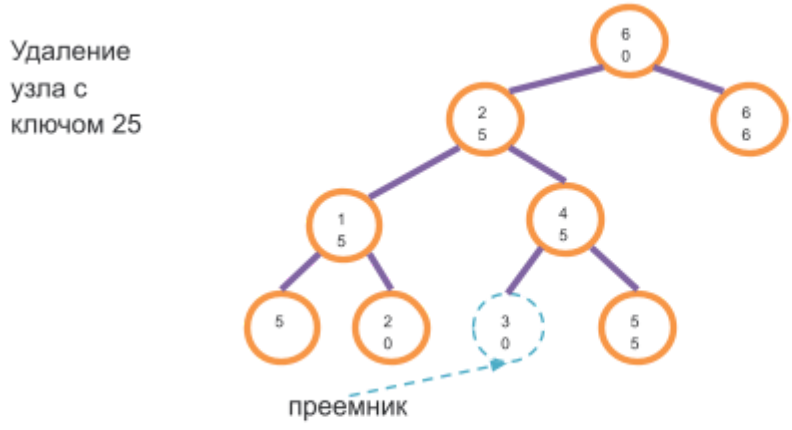
```
}
// Если нет левого потомка
else if(current.leftChild == null) {
      if (current == root) {
            root = current.rightChild;
      } else if(isLeftChild) {
            parent.leftChild = current.rightChild;
      } else {
            parent.rightChild = current.rightChild;
      }

}
```

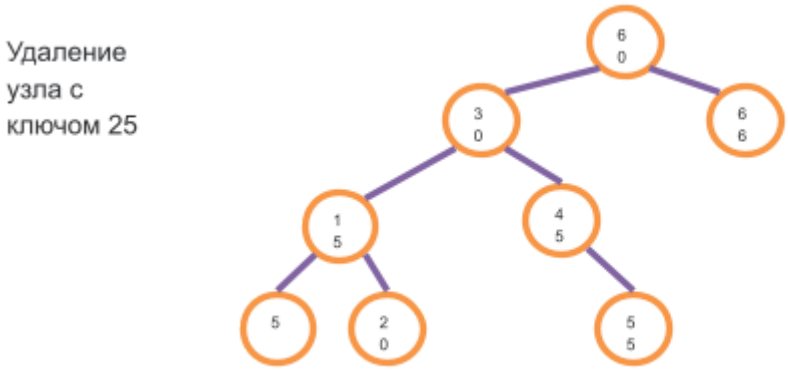
Если удаляемый элемент имеет двух потомков, нельзя просто выбрать одного из них и поставить на место родителя. Рассмотрим пример, в котором вместо удаляемого узла устанавливается правый потомок. Представим, что работаем с таким деревом:



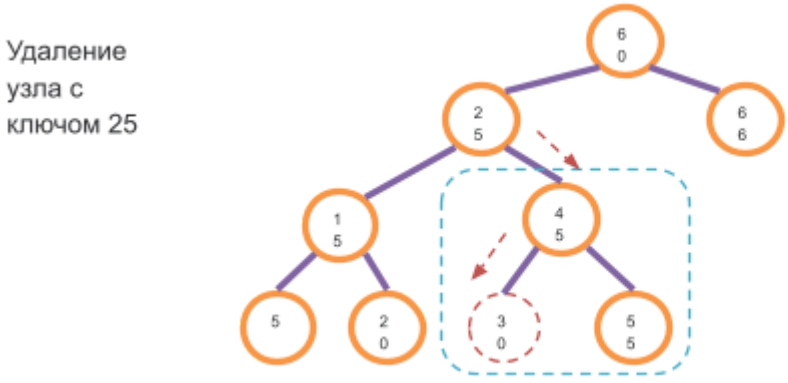
Удаляемый элемент имеет ключ 25. Какой из потомков должен встать на его место? Если это будет правый потомок со значением 45, то кто будет левым потомком нового узла 45? Потомок с ключом 15 или 30? Структура двоичного дерева будет нарушена. Надо найти преемника, который встанет на место удаляемого элемента.

В двоичном дереве все элементы располагаются в порядке возрастания. Чтобы найти приемника, необходимо выбрать элемент, ключ которого является следующим по возрастанию после удаляемого узла.





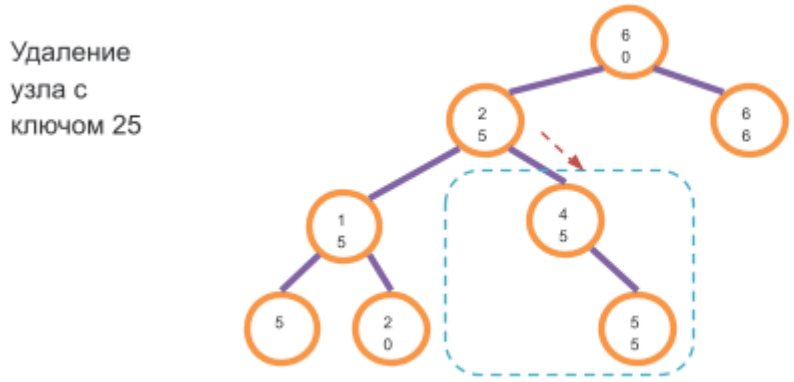
Так как ключ преемника должен быть следующим по возрастанию после удаляемого элемента, его поиск происходит в правом поддереве.



Поиск в правом поддереве

Сначала выбирается правый потомок, ключ которого больше удаляемого узла. Потом — левый потомок правого узла, далее — к левому потомку левого потомка правого узла, и так вниз по цепочке левых потомков, пока не будет найден последним. Он и будет преемником. Фактически мы применяем алгоритм нахождения минимума для правого поддерева удаляемого элемента.

У правого потомка удаляемого элемента может не быть левого потомка. Тогда первый правый потомок удаляемого элемента будет его преемником.



Преемник

Для реализации алгоритма поиска преемника создадим метод **getSuccessor**, который будет возвращать узел, являющийся приемником.

```
public Node getSuccessor(Node node) {
      Node successorParent = node;
      Node successor = node;
      Node current = node.rightChild;

      while (current != null) {
            successorParent = successor;
            successor = current;
            current = current.leftChild;
      }
      if (successor != node.rightChild) {
            successorParent.leftChild = successor.rightChild;
            successor.rightChild = node.rightChild;
      }

      return successor;

}
```

Если преемник является правым потомком удаляемого элемента, переносим все поддерево на один уровень вверх, где преемник становится на освободившееся место.

Если преемник является левым потомком правого потомка удаляемого элемента, удаление производится следующим образом. Сначала сохраняется ссылка на правого потомка преемника в поле **leftChild** родителя преемника. Далее сохраняется ссылка на правого потомка удаляемого элемента в поле **rightChild** преемника. Убирается **current** из поля **rightChild** его родителя и в нем же сохраняется ссылка на преемника **successor**. Так же убирается ссылка на левого потомка **current** из объекта **current** и сохраняется в поле **leftChild** объекта **Successor**.

Листинг программы

```
class Stack{
    private int maxSize;
    private Node[] stack;
    private int top;

    public Stack(int size){
        this.maxSize = size;
        this.stack = new Node[this.maxSize];
        this.top = -1;
    }

    public void push(Node n){
        this.stack[++this.top] = n;
    }

    public Node pop(){
        return this.stack[this.top--];
    }

    public Node peek(){
        return this.stack[this.top];
    }

    public boolean isEmpty(){
        return (this.top == -1);
```

```
    }

    public boolean isFull(){
        return (this.top == this.maxSize-1);
    }
}

class Person{
    public String name;
    public int id;
    public int age;

    public Person(){

    }

    public Person(String name, int id, int age) {
        this.name = name;
        this.id = id;
        this.age = age;
    }
}

class Node{
    public Person person;
    public Node leftChild;
    public Node rightChild;

    public void display(){
        System.out.println("Name: "+person.name+", age: "+person.age);
    }
}

class Tree{

    private Node root;

    public Node find(int key){
        Node current = root;
        while (current.person.id != key) {
            if (key < current.person.id){
                current = current.leftChild;
            } else {
                current = current.rightChild;
            }

            if (current == null){
                return null;
            }
        }
        return current;
    }

    public void insert(Person person){
        Node node = new Node();
        node.person = person;
        if (root == null){
            root = node;
        } else {
            Node current = root;
```

```
            Node parent;
            while (true) {
                parent = current;
                if (person.id < current.person.id){
                    current = current.leftChild;
                    if (current == null){
                        parent.leftChild = node;
                        return;
                    }
                } else {
                    current = current.rightChild;
                    if (current == null){
                        parent.rightChild = node;
                        return;
                    }
                }
            }
        }
    }

    public boolean delete(int id){
        Node current = root;
        Node parent = root;
        boolean isLeftChild = true;

        while (current.person.id != id) {
            parent = current;
            if (id < current.person.id){
                isLeftChild = true;
                current = current.leftChild;
            } else {
                isLeftChild = false;
                current = current.rightChild;
            }
            if (current == null){
                return false;
            }
        }

        // Если узел не имеет потомков

        if (current.leftChild == null && current.rightChild == null){
            if (current == null){
                root = null;
            } else if(isLeftChild){
                parent.leftChild = null;
            }else{
                parent.rightChild = null;
            }
        }
        // Если нет правого потомка
        else if(current.rightChild == null){
            if (current == null){
                root = current.leftChild;
            } else if(isLeftChild){
                parent.leftChild = current.leftChild;
            }else{
                parent.rightChild = current.leftChild;
            }
        }
```

```
        // Если нет левого потомка
        else if(current.leftChild == null){
            if (current == null){
                root = current.rightChild;
            } else if(isLeftChild){
                parent.leftChild = current.rightChild;
            } else {
                parent.rightChild = current.rightChild;
            }
        } else {
            Node successor = getSuccesor(current);
            if (current == root){
                root = successor;
            }else if(isLeftChild){
                parent.leftChild = successor;
            }else {
                parent.rightChild = successor;
            }
            successor.leftChild = current.leftChild;
        }
      return true;
    }

    public Node getSuccesor(Node node){
        Node successorParent = node;
        Node successor = node;
        Node current = node.rightChild;

        while (current != null) {
            successorParent = successor;
            successor = current;
            current = current.leftChild;
        }
        if (successor != node.rightChild){
            successorParent.leftChild = successor.rightChild;
            successor.rightChild = node.rightChild;
        }

        return successor;
    }

    public void traverse(int traverseType){
        switch(traverseType){
            case 1:System.out.println("Preorder traversal");
            preOrder(root);
            break;
        }
    }

    private void preOrder(Node rootNode){
        if(rootNode != null){
            rootNode.display();
            preOrder(rootNode.leftChild);
            preOrder(rootNode.rightChild);
        }
    }

    private void postOrder(Node rootNode){
         if(rootNode != null){
             postOrder(rootNode.leftChild);
```

```
            postOrder(rootNode.rightChild);
            rootNode.display();
        }
    }

    private void inOrder(Node rootNode){
        if(rootNode != null){
            inOrder(rootNode.leftChild);
            rootNode.display();
            inOrder(rootNode.rightChild);
        }
    }

    public void displayTree(){
        Stack stack = new Stack(100);
        stack.push(root);
        int nBlanks = 32;
        boolean isRowEmpty = false;

        while (!isRowEmpty) {
            Stack localStack = new Stack(100);
            isRowEmpty = true;
            for(int i=0;i<nBlanks;i++){
                System.out.print(" ");
            }
            while (!stack.isEmpty()) {
                Node temp = stack.pop();
                if (temp != null){
                    temp.display();
                    localStack.push(temp.leftChild);
                    localStack.push(temp.rightChild);
                    if(temp.leftChild != null || temp.rightChild != null){
                        isRowEmpty = false;
                    }
                }else{
                    System.out.print("--");
                    localStack.push(null);
                    localStack.push(null);
                }
                 for(int j=0; j < nBlanks * 2 - 2; j++)
                     System.out.print(' ');
            }
            System.out.println(" ");
            nBlanks = nBlanks / 2;
            while (!localStack.isEmpty()) {
                stack.push(localStack.pop());
            }
            System.out.println("..........................................");
        }
    }

}

public class TreeApp {
    public static void main(String[] args) throws IOException{
        int value;
        Tree theTree = new Tree();
        theTree.insert(new Person());
        theTree.insert(new Person());
        theTree.insert(new Person());
```

```
        theTree.insert(new Person());
        theTree.insert(new Person());
        theTree.insert(new Person());
        theTree.insert(new Person());

         while(true){
            System.out.print("Enter first letter of show, ");
            System.out.print("insert, find, delete, or traverse: ");
            int choice = getChar();
            switch(choice){
                case 's':
                    theTree.displayTree();
                    break;
                case 'i':
                    System.out.print("Enter value to insert: ");
                    value = getInt();
                    theTree.insert(new Person());
                        break;
                case 'f':
                    System.out.print("Enter value to find: ");
                    value = getInt();
                    Node found = theTree.find(value);
                    if(found != null){
                        System.out.print("Found: ");
                    }
                    found.display();
                    System.out.print("\n");
                    break;
            }
        }
    }

     public static String getString() throws IOException {
         InputStreamReader isr = new InputStreamReader(System.in);
         BufferedReader br = new BufferedReader(isr);
         String s = br.readLine();
         return s;
     }

      public static char getChar() throws IOException {
          String s = getString();
          return s.charAt(0);
      }

       public static int getInt() throws IOException {
           String s = getString();
           return Integer.parseInt(s);
       }
}
```

Эффективность двоичных деревьев

Для поиска элемента алгоритм спускается вниз от уровня к уровню, поэтому можно сказать, что сложность алгоритма поиска зависит от количества уровней дерева. Так как дерево является двоичным, количество уровней напрямую связано с количеством узлов. Например, дерево из трех

узлов будет иметь два уровня, из семи — три. Если количество узлов обозначить переменной **N**, а уровней — **L**, то можно составить равенство, где **N** на единицу меньше двух в степени **L**.

$$N = 2^L - 1$$

Прибавив по единице к двум сторонам, получим:

$$N + 1 = 2^L$$

или

$$L = log_2(N + 1)$$

Если представить данную формулу в O-синтаксисе, сложность выполнения алгоритма поиска можно обозначить как **$O(log_2N)$**.

Сравним поиски по дереву, в неупорядоченном массиве и связанном списке. Если есть 1000000 элементов, поиск в неупорядоченном массиве или связанном списке в среднем будет занимать 500000 сравнений, а в дереве всего 20. В упорядоченном массиве поиск осуществляется быстро, но вставка потребует перемещения 500000 элементов. А вставка элемента в дерево займет 20 операций сравнения и незначительное время на связывание. Аналогичным образом можно сравнить операции удаления.

Практическое задание

1. Создать и запустить программу для построения двоичного дерева. В цикле построить двадцать деревьев с глубиной в 6 уровней. Данные, которыми необходимо заполнить узлы деревьев, представляются в виде чисел типа **int**. Число, которое попадает в узел, должно генерироваться случайным образом в диапазоне от -100 до 100.

2. Проанализировать, какой процент созданных деревьев являются несбалансированными.

Дополнительные материалы

1. *Про красно-черные деревья:* Лафоре Р. Структуры данных и алгоритмы в Java. Классика Computers Science. 2-е изд. — СПб.: Питер, 2013. — 403–435 сс.

2. *Про деревья 2-3-4*: Лафоре Р. Структуры данных и алгоритмы в Java. Классика Computers Science. 2-е изд. — СПб.: Питер, 2013. — 436–485 сс.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Лафоре Р. Структуры данных и алгоритмы в Java. Классика Computers Science. 2-е изд. — СПб.: Питер, 2013. — 346–401 сс.