Модуль 4. Алгоритмы и структуры данных

Тема 4.4. Деревья

2 часа

Оглавление

4.4. Деревья

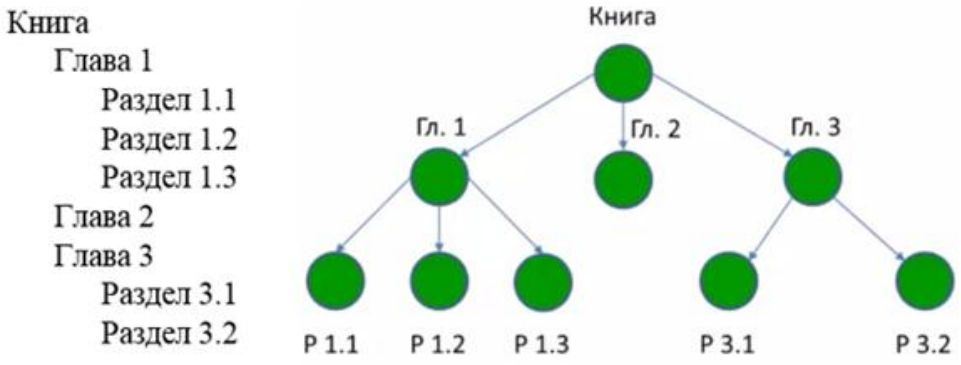
4.4.1 Дерево. Разновидности деревьев

Древовидная структура является одним из способов представления иерархической структуры в графическом виде. Такое название она получила потому, что граф выглядит как перевернутое дерево. Корень дерева (корневой узел) находится на самом верху, а листья (потомки) – внизу.

Деревья широко применяются в компьютерных технологиях, и самым близким примером является файловая система, представляющая собой иерархическую структуру из файлов и каталогов.

Уже изученный нами язык XML также имеет древовидную структуру.

В качестве примера древовидной структуры представим оглавление книги.



Узел “Книга” является корнем дерева, т.к. находится в вершине дерева, книга содержит определенные главы, а главы в свою очередь состоят из разделов, каждый раздел может содержать подразделы и т.д.

На рисунке стрелками изображены родительские отношения (ребра, ветви дерева) между узлами (вершинами) дерева. На верхнем уровне каждый “родитель” указывает стрелкой на своих “потомков”. Т.е. в этой иерархической структуре вершина всегда “знает” своих потомков.

Для того чтобы более точно оперировать структурой Дерево, нужно дать определение некоторым ключевым понятиям:

- **корневой узел** - самый верхний узел дерева, он не имеет предков (на рисунке узел “Книга”);
- **лист, листовой или терминальный узел** - конечный узел, т.е. не имеющий потомков (на рисунке узлы “Гл. 2”, “Р 1.1”, “Р 1.2”, “Р 1.3”, “Р 3.1”, “Р 3.2”);
- **внутренний узел** - любой узел дерева, имеющий потомков, т.е не лист (на рисунке узлы “Гл. 1”, “Гл. 3”);

С корневого узла начинается выполнение большинства операций над деревом, потому что, как и в списках, чтобы получить доступ к любому элементу структуры, необходимо, переходя по ветвям, перебирать элементы, начиная с головы - корневого узла. Корневой узел - это своеобразный вход в дерево.

Большинство алгоритмов работы с деревом строятся на том, что каждый узел дерева рассматриваются как корневой узел поддерева, «растущего» из этого узла. Такой подход дает возможность зацикливать выполнение операций при прохождении по элементам дерева. Но в связи с тем, что при прохождении по дереву (в отличие от массива) неизвестно сколько шагов будет в этом

	цикле используется другой инструмент - рекурсивный вызов. Понятие рекусии в программировании рассматривается в этом же модуле, но в рамках другой темы.

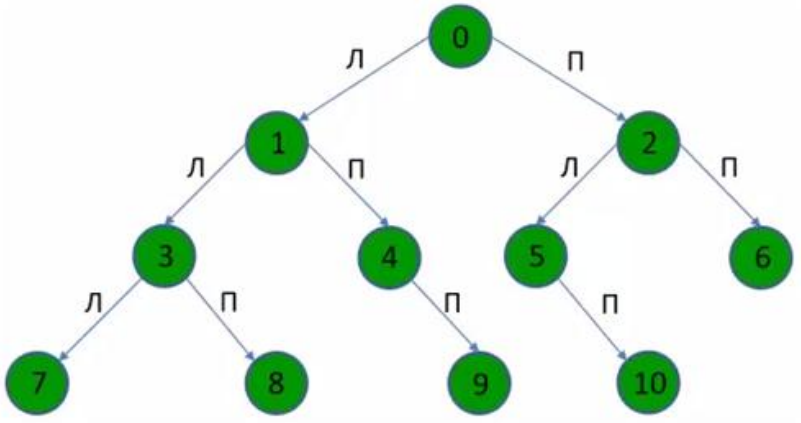
Как и у списков существует огромное разнообразие древовидных структур, например несколько несвязанных между собой деревьев образуют структуру данных, которая называется "лес".

В данной теме мы рассмотрим только основные классификации:

1. По числу ветвей, отходящих от узла, деревья делят на
   - двоичные (или бинарные): каждый узел может иметь не более двух потомков;
   - с числом ветвей больше 2: часто такие деревья называют K-деревьями, их разновидностей более полусотни.
2. По признаку сбалансированности дерева: сбалансированные и несбалансированные.

4.4.2 Понятие бинарного дерева

Двоичное (бинарное) дерево - это древовидная структура данных, где каждый узел имеет не более двух потомков. Этих потомков называют левым (Л) и правым (П) потомком или “сыном”. Ниже приведен рисунок, представляющий двоичное дерево.



- узел-родитель потомка называют **предком**. Для узла 3 предком является узел 1, левым сыном - 7, правым - 8.
- левый потомок является корневым узлом **левого поддерева**, а правый потомок - корневым узлом **правого поддерева.**
- **высотой дерева** - является максимальное количество уровней, на которых располагаются его узлы. На рисунке от узла 6 до корня узлы располагаются на трех уровнях, а от узлов 7, 8, 9, 10 - на четырех, следовательно высота дерева равна 4.

Упражнение 4.4.1

Реализуем простой класс бинарного дерева на языке Java:

- Поля: значение в узле дерева (value), ссылка левого потомка (lchild) и ссылка на правого потомка (rchild).
- Методы: заполнение дерева и печать в консоль содержимого дерева (обхода дерева).

```
public class BinaryTree {

	int value;
	BinaryTree lchild; // левый потомок
	BinaryTree rchild; // правый потомок

	public BinaryTree(int value) {
		this.value = value;
		this.lchild = null;
		this.rchild = null;
	}

	public BinaryTree() {
		this.value = -1; // значение для пустого узла
		this.lchild = null;
		this.rchild = null;
	}

	/* метод вставки элементов в дерево
	   root - ссылка на текущий узел дерева
	   valueNode - значение, которое добавляем в дерево
	 */
	public void insertNode(BinaryTree root, int valueNode) {
		// если дерево пустое
		if (root.value == -1) {
			root.value = valueNode; // записываем значение в узел
			return;
		}
		/*если значение в текущем узле больше valueNode, то переходим в
		  левое поддерево*/
		if (root.value > valueNode) {
			//если левого потомка нет, то создаем его с значением valueNode
			if (root.lchild == null)
				root.lchild = new BinaryTree(valueNode);
			// если левый потомок есть, то переходим ниже в левое поддерево
			else
				insertNode(root.lchild, valueNode);
		}
		/*если значение в текущем узле меньше вставляемого,
		  то переходим в правое поддерево*/
		 else if (root.value < valueNode) {
			//если правого потомка нет, то создаем его с значением valueNode
			if (root.rchild == null)
				root.rchild = new BinaryTree(valueNode);
			// если правый потомок есть, то переходим ниже в правое
поддерево
			else
				insertNode(root.rchild, valueNode);
		}
	}

	// метод вывода дерева на экран
	public void printBinaryTree(BinaryTree root, int level) {
		if (root != null) {
			printBinaryTree(root.lchild, level + 1);
			for (int i = 0; i < level; i++)
			   System.out.print("    ");//чем ниже уровень, тем отступ
больше
			System.out.println(root.value);
			printBinaryTree(root.rchild, level + 1);
```

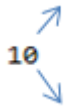
```
            }
      }

      public static void main(String[] args) {
            // массив значений для добавления в дерево
            int b[] = { 10, 25, 20, 6, 4, 8, 50, 30, 6 };
            BinaryTree tree = new BinaryTree();
            // добавление элементов массива в дерево
            for (int i = 0; i < b.length; i++)
                  tree.insertNode(tree, b[i]);
            // вывод содержимого дерева
            tree.printBinaryTree(tree, 0);
      }
}
```

Запустите программу и на экране получите результат - дерево ветвями вправо:

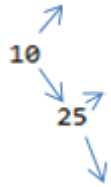
```
<terminated> BinaryTree
      4
   6
      8
10
      20
   25
         30
      50
```

Чтобы понять результат, разберем алгоритм вставки значения в дерево на нашем примере. Мы поочередно вставляли элементы массива { 10, 25, 20, 6, 4, 8, 50, 30, 6 }.

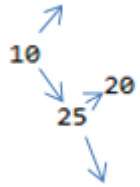
Вначале дерево было пустым, поэтому 10 был сохранен в корне дерева



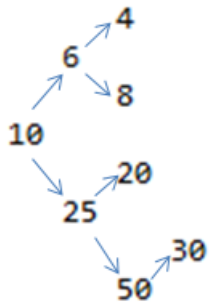
Следующее значение 25 сравнили с 10, так как оно больше, перешли вправо, правый потомок был пустым, поэтому он был создан конструктором с параметром 25:



Третье значение 20 сравнили с 10 - ушли вправо (потому что 20>10), сравнили с 25 - ушли влево (потому что 20<25) и создали новый узел:



Аналогично добавляются все значения, кроме последнего. Последнее значение 6 не подпадает ни под одно из условий при сравнении с уже существующим узлом 6. Значение не больше и не меньше, а равно значению в узле дерева! В программе для такого условия никаких действий не производится. Т.е. повторяющиеся значения не вставляются, и в результате мы получили дерево:



Методы вставки значений и вывода дерева используют рекурсивные вызовы функции. Этот механизм заслуживает отдельного глубокого рассмотрения и представлен в следующей теме (тема 4.5.).

Здесь же достаточно понять, что функция вывода работает по так называемому **левостороннему обходу**:

1. начинает с корня;
2. спускается по левым ветвям до самого левого листа (4);
3. выводит значение листа (4), затем его родителя (6) и правого потомка (8);
4. подымается через уровень к родителю родителя и выводит его (10), спускается к правому потомку
5. повторяет все с шага 2 для текущего узла (25) - т.е. начинает новый левосторонний обход с выводом значений: 20, 25, 30, 50.

Для имитации структуры дерева в методе вывода дерева перед значением узла выводится количество пробелов, пропорциональное уровню узла. Если закоментировать цикл печати пробелов, то будет выведена отсортированная последовательность !

4 6 8 10 20 25 30 50

Думаю, Вы догадались, что разобранный нами алгоритм включения значений в дерево позволяет получить упорядоченное дерево.

4.4.3 Понятие сбалансированного дерева

В предыдущем упражнении мы создали такое бинарное дерево, в котором для каждого узла выполняются следующие правила:

- все значения в узлах его левого поддерева меньше значения этого узла;
- все значения в узлах его правого поддерева больше значения этого узла;
- одинаковые значения в дереве не допускаются.

Очевидно, что в таком дереве легко найти элемент, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения (ключа) в каждом узле (этот алгоритм поиска назвается двоичным - он будет разобран в следующей теме).

Дерево позволяет легко реализовать поиск, а также позволяет быстро выполнять операции вставки и удаления элементов. Другие структуры данных — массивы, сортированные массивы, связанные списки — по крайней мере в одной из этих областей работают недостаточно быстро.

Однако вспомним, что двоичный поиск преполагает, что диапазон поиска каждый раз делится попалам, а дерево вовсе не обязательно может обеспечивать такое условие. Например, запустим нашу программу из Упражнения 4.4.1 для уже упорядоченного массива {10, 15, 28, 30, 45, 50, 70}, получим дерево:

```
<terminated> BinaryTree [Jav
10
   15
      28
         30
            45
               50
                  70
```

Узлы выстраиваются в линию без ветвления. Поскольку каждый узел больше узла, вставленного перед ним, каждый узел является правым потомком, поэтому все узлы располагаются по одну сторону от корня. Дерево получается предельно несбалансированным. Если вставить элементы, упорядоченные по убыванию, то каждый узел будет левым потомком своего родителя, а дерево окажется несбалансированным с другой стороны

Сбалансированное дерево - это дерево, в котором разница между листьями составляет не более чем 1 уровень.

Наше первое дерево было сбалансированным, последнее дерево - крайне несбалансированное.

В несбалансированном дереве теряется возможность быстрого поиска, а также вставки или удаления. Трудоемкость поиска в последнем дереве ничем не отличается от поиска в списке (см. предыдущую тему).

Чтобы обеспечить быстрое время поиска, на которое способно дерево, необходимо позаботиться о том, чтобы дерево всегда оставалось сбалансированным (или по крайней мере почти сбалансированным). Это означает, что каждый узел дерева должен иметь справа и слева приблизительно одинаковое количество потомков (во всех поколениях).

Методы получения сбалансированного дерева разработано множество приемов, но мы не будем рассматривать их в данной теме.

4.4.4 Библиотечный класс TreeSet

Как и для списков в Java реализован библитечный класс работы с деревьями TreeSet. Он представляет древовидную структуру данных, в которой все объекты хранятся в отсортированном виде по возрастанию. Класс TreeSet является наследником класса AbstractSet и реализует интерфейс NavigableSet.

В классе TreeSet определены конструкторы:

- *TreeSet() - создает пустое дерево;*
- *TreeSet(Collection <? extends E> c) - создает дерево, в которое добавляет все элементы коллекции c;*
- *TreeSet(SortedSet <E> Set) - создает дерево, в которое добавляет все элементы сортированного набора Set;*
- *TreeSet(Comparator <? super E> comparator) - создает пустое дерево, где все добавляемые элементы впоследствии будут отсортированны компаратором.*

Класс TreeSet обладает следующими методами:

`E ceiling(E объект)`	ищет в наборе наименьший элементе, для которого истинно `e>=объект`. Если такой элемент найден, он возвращается. В противном случае возвращается значение `null`
`E floor(Eобъект)`	ищет в наборе наибольший элементе, для которого истинно

	e<=объект. Если такой элемент найден, он возвращается. В противном случае возвращается значение null
add(E объект)	добавляет элемент e в коллекцию, если такого еще там нет. Возвращает true, если элемент добавлен
remove(E объект)	удаляет указанный элемент из коллекции
E contains(E объект)	возвращает true, если указанный элемент имеется в коллекции
E subSet(E нижнГраница, E верхнГраница)	возвращает объект интерфейса NavigableSet, включающий все элементы вызывающего набора, которые больше нижнГраница и меньше верхнГраница.
size()	возвращает количество элементов коллекции
clear()	удаляет все элементы коллекции

Упражнение 4.4.2

Создадим Java приложение, демонстрирующее работу класса TreeSet.

```
import java.util.SortedSet;
import java.util.TreeSet;

/* Примериспользованиядеревьев */
public class Main {
  public static void main(String[] args) {
    //Создание пустого дерева
    TreeSet<String> tree = new TreeSet<String>();

    //Добавление элементов в дерево
    tree.add("abc");
    tree.add("aba");

    //Элементы выводятся в сортированном (лексикографическом) порядке
    //Классы должны быть Comparable
    System.out.println("===== Tree =====");
    for (String s : tree){
      System.out.println(s);
    }
    System.out.println();

    //Удаление элементов
    System.out.println(tree.remove("aaa"));
    //удаление несуществующего элемента возвращает False
    System.out.println(tree.remove("abc"));

    System.out.println("===== Tree =====");
    for (String s : tree) {
      System.out.println(s);
    }
    System.out.println();

    tree.add("zzz");
```

```
    tree.add("xyz");
    tree.add("zca");

    //Быстрая проверка наличия элемента в дереве
    System.out.println(tree.contains("xyz") + " " + tree.contains("aab"));

    // Выводит наименьший элемент больший или равный указанного
    System.out.println("ceiling = " + tree.ceiling("zyb"));
    System.out.println(tree.ceiling("zzzz")); //null если нет такого
элемента

    //Выводит наибольший элемент меньший или равный указанного
    System.out.println("floor = " + tree.floor("zyb"));
    System.out.println(tree.floor("a")); //null если нет такого элемента

    System.out.println("===== Subtree =====");
    //Взятие подмножества элементов >= zaa и < zzzz
    SortedSet<String>subtree = tree.subSet("zaa", "zzzz");
    for (String s : subtree){
      System.out.println(s);
    }
    System.out.println();
  }
}
```

Результат работы программы выглядит так:

```
===== Tree =====
aba
abc

false
true
===== Tree =====
aba

true false
ceiling = zzz
null
floor = zca
null
===== Subtree =====
zca
zzz
```

Задание 4.4.1

Реализовать класс, который с использованием TreeSet находит список учеников, сдавших экзамен на оценку выше заданного числа. Для решение задачи нужно реализовать собственный класс Pupil, реализующий интерфейс Comparable.

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю IT ШКОЛЫ SAMSUNG Глухову Денису Александровичу.