尚硅谷大数据之算法和数据结构

(作者：尚硅谷大数据研发部)

版本：V1.0

# 第 1 章 数据结构概述

软件公司 = 软件 + 商业模式

软件 = 程序 + 软件工程

程序 = 算法 + 数据结构

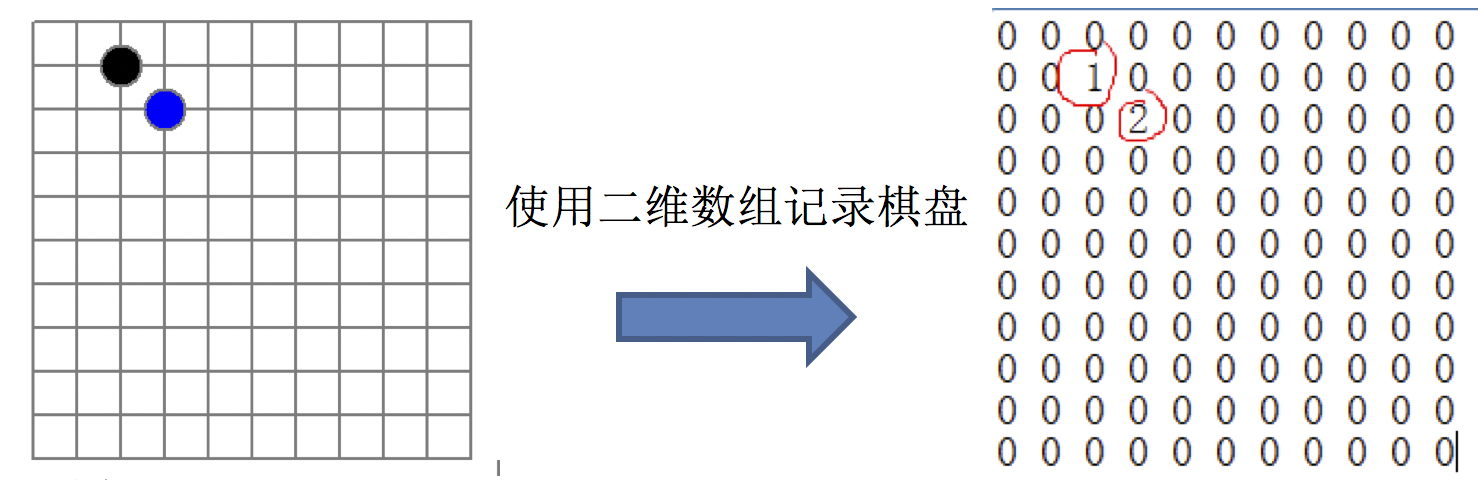
* 数据结构是一门研究算法的学科，自从有了编程语言也就有了数据结构.学好数据结构可以编写出更加漂亮,更加有效率的代码。
* 算法是程序的灵魂，为什么有些网站能够在高并发，和海量吞吐情况下依然坚如磐石
* 数据结构是算法的基石

# 第 2 章 基本数据结构

## 稀疏数组

### 什么是稀疏数组

假设编写一个五子棋游戏， 就需要使用一个二维数组来表示整个棋盘和棋盘中黑子与白子的位置。



* 存在的问题：

该二维数组中很多默认值都是0， 因此会记录很多没意义的数据。可以使用稀疏数组来解决这个问题： 只存储有效棋子的位置， 默认值可以不用存储。

### 实现稀疏数组

* 目的：

1. 使用稀疏数组，来保留类似前面的二维数组(棋盘、地图等等)
2. 把稀疏数组存盘，并且可以重新恢复原来的二维数组数

* 具体实现：

**package** com.atguigu.datastruc

**import** scala.collection.mutable.ArrayBuffer

object SparseArrayDemo {

    // 五子棋盘大小

    val rowsNum = 15

    val colsNum = 15

    def main(args: Array[String]): Unit = {

        // 1. 初始化五子棋盘

        val chess: Array[Array[Int]] = initChessBoard()

        // 2. 五子棋盘转成稀疏数组

        val sparseArr: SparseArray = toSparseArray(chess)

        println(sparseArr)

        // 3. 稀疏数字转成屋子棋盘

        val arr2: Array[Array[Int]] = sparse2Chess(sparseArr)

        printChess(arr2)

    }

    // 4. 打印棋盘

    def  printChess(chess: Array[Array[Int]]) = {

**for**(row <- chess.indices; col <- chess(row).indices){

            print(chess(row)(col) + " ")

**if**(col == chess(row).length - 1) println()

        }

    }

    // 3. 把稀疏数组转换成二维棋盘

    def sparse2Chess(sparseArr: SparseArray) = {

        // 五子棋盘

        val arr: Array[Array[Int]] = Array.ofDim[Int](rowsNum, colsNum)

        sparseArr.buf.foreach(node => {

            arr(node.row)(node.col) = node.value

        })

        arr

    }

    // 2. 五子棋棋盘转成稀疏数组

    def toSparseArray(chess: Array[Array[Int]]): SparseArray = {

        val sparseArr = **new** SparseArray

        // 把 值不是 0 的存入到稀疏数组中

**for** (row <- chess.indices; col <- chess(row).indices **if** chess(row)(col) != 0) {

            sparseArr.add(row, col, chess(row)(col))

        }

        sparseArr

    }

    // 1. 初始化五子棋棋盘

    def initChessBoard(): Array[Array[Int]] = {

        // 棋盘的每个位置自动会置为 0

        val arr: Array[Array[Int]] = Array.ofDim[Int](rowsNum, colsNum)

        // 放置几个棋子: 白=1 黑=2

        val white = 1

        val black = 2

        arr(1)(2) = white

        arr(2)(3) = black

        arr

    }

}

 // 定义稀疏数组

**class** SparseArray {

    // 使用可变数组来存储 Node (行列值信息)

    val buf: ArrayBuffer[Node] = ArrayBuffer[Node]()

    // 判断稀疏数组是否为 Empty

    def isEmpty: Boolean = buf.isEmpty

    // 向稀疏数组中添加数据

    def add(row: Int, col: Int, value: Int): ArrayBuffer[Node] = {

        buf += Node(row, col, value)

    }

    override def toString: String = buf.toString

    // 用来存储二维数组中非默认值的具体: 行, 列, 值 信息

**case** **class** Node(row: Int, col: Int, value: Int)

}

* 自己实现： 把棋盘通过稀疏数组存盘， 并从磁盘复原棋盘

## 队列

队列也是一种常用的数据结构。比如要管理银行的排队客户就需要用到队列。

### 什么是队列

1. 队列是一个有序列表，可以用数组或是链表来实现。
2. 遵循先入先出的原则（FIFO）。即：先存入队列的数据，要先取出。后存入的要后取出

### 使用数组自定义循环队列

循环队列需要3个参数，head,tail,count

1. 队列初始化时，head和tail值都为零。
2. count 用来记录队列中元素的个数

package com.atguigu.datastruc

import scala.reflect.ClassTag

object QueueDemo {

    def main(args: Array[String]): Unit = {

        val queue = **new** ArrayQueue[Int](4)

        queue.enqueue(1)

        queue.enqueue(2)

        queue.enqueue(3)

        queue.enqueue(4)

        queue.enqueue(4)

        println(queue.dequeue())

        println(queue.dequeue())

        println(queue.dequeue())

        println(queue.dequeue())

        println(queue.dequeue())

        println(queue.dequeue())

    }

}

/\*\*

  \* 使用数组来模拟队列

  \* 使用数组模拟环绕队列

  \*

  \* @param initSize 队列的初始容量

  \* @tparam T 队列中存储的元素的类型

  \*/

**class** ArrayQueue[T: ClassTag](val initSize: Int) {

    // 内置数组: 存储数据

    val arr = **new** Array[T](initSize)

    // 队列头部:

    var head: Int = 0

    // 队列尾部: 指向队列中下一个要添加的元素的位置

    var tail: Int = 0

    // 队列中元素的个数: 用来判断队列是满或是空

    var count: Int = 0

    // 判断队列是否为空

    def isEmpty: Boolean = count == 0

    // 循环队列是否已满:

    // 方法 1: 增加一个元素个数参数(===)

    // 方法 2: 少用一个存储空间.(最后一个位置不用)

    def isFull: Boolean = count == initSize

    // 入队

    def enqueue(ele: T): Unit = {

**if** (isFull) **throw** **new** UnsupportedOperationException("队列已满, 不能再添加元素....")

        // 把入队的元素添加到下一个位置

        arr(tail) = ele

        // 队列中元素的个数 +1

        count += 1

        // 更新下一个要添加元素的位置

        tail += 1

        // tail 到了数组的尾部, 则从头开始

**if** (tail == initSize) tail = 0

    }

    // 出队

    def dequeue(): Option[T] = {

        // 如果是空队列则出队元素为: None

**if** (isEmpty) None

**else** { // 不为空, 则返回队头元素, 并更新队头位置.  队列元素计数 -1

            val res: T = arr(head)

            count -= 1

            head += 1

**if** (head == initSize) head = 0

            Some(res)

        }

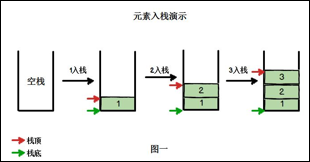
    }

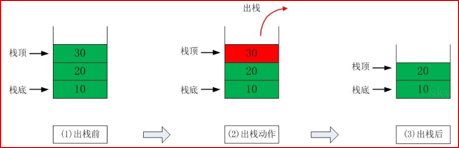
}

## 栈

### 什么是栈

* 栈(stack)是一个先入后出(FILO-First In Last Out)的有序列表。
* 栈(stack)是限制线性表中元素的插入和删除只能在线性表的同一端进行的一种特殊线性表。允许插入和删除的一端，为变化的一端，称为栈顶(Top)，另一端为固定的一端，称为栈底(Bottom)。
* 根据堆栈的定义可知，最先放入栈中元素在栈底，最后放入的元素在栈顶，而删除元素刚好相反，最后放入的元素最先删除，最先放入的元素最后删除
* 栈主要两个操作: 入栈(push)和出栈(pop)





### 栈应用场景

* 子程序的调用：在跳往子程序前，会先将下个指令的地址存到堆栈中，直到子程序执行完后再将地址取出，以回到原来的程序中。
* 处理递归调用：和子程序的调用类似，只是除了储存下一个指令的地址外，也将参数、区域变量等数据存入堆栈中。
* 表达式的转换与求值(实际解决), 计算器原理。
* 二叉树的遍历。
* 图形的深度优先(depth一first)搜索法。

### 使用数组自定义栈

package com.atguigu.datastruc

import scala.reflect.ClassTag

/\*\*

  \* Author lzc

  \* Date 2019-11-26 16:45

  \*/

object StackDemo {

    def main(args: Array[String]): Unit = {

        val stack = **new** ArrayStack[Int](4)

        stack.push(1)

        stack.push(2)

        stack.push(3)

        stack.push(4)

        println(stack.pop())

        println(stack.pop())

        println(stack.pop())

        println(stack.pop())

        println(stack.pop())

    }

}

/\*\*

  \* 使用数组模拟栈

  \*

  \* @param maxSize 栈的容量

  \* @param classTag$T

  \* @tparam T 入栈的元素的类型

  \*/

**class** ArrayStack[T: ClassTag](val maxSize: Int) {

**private** val arr = **new** Array[T](maxSize)

    // 栈顶()

**private** var top: Int = -1

    // 栈是否空

    def isEmpty: Boolean = top == -1

    // 栈是否满

    def isFull: Boolean = top == maxSize - 1

    // 入栈

    def push(ele: T)= {

**if** (isFull) **throw** **new** UnsupportedOperationException("栈已经满员...")

        top += 1

        arr(top) = ele

    }

    // 出栈

    def pop() = {

**if** (isEmpty) None

**else** {

            val res: T = arr(top)

            top -= 1

            Some(res)

        }

    }

    // 获取栈顶元素, 但不弹出栈顶元素

    def peak(): Option[T] = {

**if** (isEmpty) None

**else** {

            Some(arr(top))

        }

    }

}

### 实现计算器(课堂不讲,自己完成)

**package** com.atguigu.datastruc

object CalculatorDemo {

    def main(args: Array[String]): Unit = {

        val calculator = **new** Calculator

        println(calculator.start("2\*12-2/2-6/2-1"))

    }

}

/\*\*

  \* 表示一个计算器

  \* 只考虑 +-\*- 不考虑()等其他运算

  \* 23 + 10 \* 5 - 10

  \*/

**class** Calculator {

    /\*

        计算给出的表达式的值: 利用栈的工作原理

        两个栈: 一个用来存放数字 另一个用来存放符号

     \*/

    def start(expression: String): Double = {

        // 数栈

        val numStack = **new** ArrayStack[Double](1000)

        // 符号栈

        val notationStack = **new** ArrayStack[Char](1000)

        // 1. 去掉字符串中可能的空白字符

        val expr: String = expression.replaceAll("\\s+", "");

        // 2. 遍历字符串, 存入到对应的栈中

        // 3. 进入栈的同时计算高优先级或者同等优先级的

        // 4. 把结果也压入栈中

        val sb = **new** StringBuilder // 用来缓存完整的数字(12)

        expr.foreach {

            // 如果是数字或者.(不是运算符), 则表示数字, 存入到 sb 中, 等将来碰到一个完整的数再压入到数栈中

**case** c **if** c.toString.matches("[\\d\\.]") => sb.append(c)

**case** c => { // 剩下情况就是运算符则: 1. sb 中的数字压入到数栈 2. 运算符压入到符号栈(先把优先级高的计算出来)

                numStack.push(sb.toDouble)

                sb.clear() // 清空缓存

                var isOver = **false**

**while** (!isOver) { // 考虑到优先级高的算完之后, 出现优先级相同的连在一起, 所以要一直算下去

                    // 比较新的运算符和栈顶的运算符的优先级: 如果低于或等于栈顶优先级, 则需要从把优先级高的计算出来, 然后把结果入栈

                    val topNotation: Option[Char] = notationStack.peak()

**if** (topNotation.isDefined && lteq(c, topNotation.get)) { // 栈顶有值, 并且 c 的运算符低于栈顶运算符的优先级, 则需要计算高优先级的运算

                        val tmpRes: Double = calc(notationStack.pop().get, numStack.pop().get, numStack.pop().get)

                        // 计算结果重新压入数栈

                        numStack.push(tmpRes)

                    } **else** {

                        isOver = **true**

                    }

                }

                notationStack.push(c) // c 运算符压入符号栈

            }

        }

        numStack.push(sb.toDouble) // 最后一个数字也要压如到栈中

        // 5. 计算栈中剩余优先级的运算

**while** (!notationStack.isEmpty) {

            val tmpRes: Double = calc(notationStack.pop().get, numStack.pop().get, numStack.pop().get)

            numStack.push(tmpRes)

        }

        // 6. 返回结果

        numStack.pop().get

    }

    /\*\*

      \* 计算:  num2 op num1

      \*

      \* @param op

      \* @param num1

      \* @param num2

      \*/

    def calc(op: Char, num1: Double, num2: Double) = {

        op match {

**case** '+' => num2 + num1

**case** '-' => num2 - num1

**case** '\*' => num2 \* num1

**case** '/' => num2 / num1

        }

    }

    /\*\*

      \* 判断第一个运算符的优先级是否低于或等于第二个运算符的优先级

      \*

      \* @param notation1 第一个运算符

      \* @param notation2 第二个运算符

      \*/

    def lteq(notation1: Char, notation2: Char): Boolean = getPriority(notation1) <= getPriority(notation2)

    /\*\*

      \* 定义运算符的优先级

      \*

      \* @param notation 优先级

      \* @return

      \*/

    def getPriority(notation: Char): Int = {

        notation match {

**case** '+' => 1

**case** '-' => 1

**case** '\*' => 2

**case** '/' => 2

        }

    }

}

## 链表

### 什么是链表

链表也是一种有序列表, 其中的各对象按线性顺序排列.

数组的线性顺序是由数组下标决定的, 而链表的顺序是由各个对象里的指针决定的.

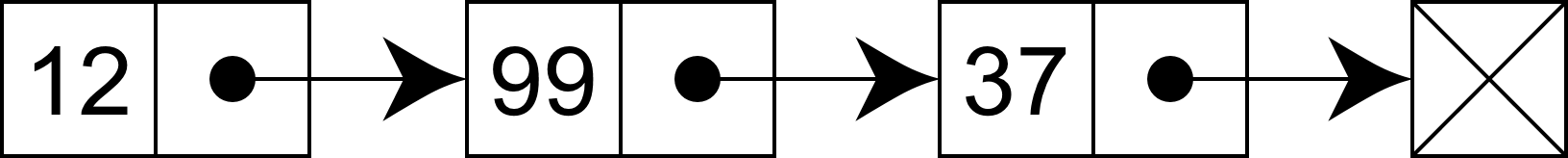
使用链表结构可以克服数组链表需要预先知道数据大小的缺点，链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。

链表作为一种基础的数据结构可以用来生成其它类型的数据结构

根据链表结构不同, 链表分为: 单向链表, 双向链表和循环链表.

### 实现单向链表

单向链表是一种最简单的链表，它包含两个域，一个信息域和一个指针域。这个链接指向列表中的下一个节点，而最后一个节点则指向一个空值。



**package** com.atguigu.datastruc.linked\_list

/\*\*

  \* Author lzc

  \* Date 2019-11-27 08:40

  \*/

object SinglyLinkedListDemo {

    def main(args: Array[String]): Unit = {

        val list = **new** SinglyLinkedList[Int]()

        list.add(10)

        list.add(20)

        list.add(30)

        list.printInfo()

        list.delete(20)

        list.add(40)

        list.printInfo()

    }

}

/\*\*

  \* 单向列表

  \*

  \* @tparam T 列表中存储的元素的类型

  \*/

**class** SinglyLinkedList[T] {

    // 头结点

**private** var head: Node = \_

    // 为了提高添加的效率, 设置一个尾结点

**private** var tail: Node = \_

    /\*\*

      \* 添加元素

      \*

      \* @param ele

      \* @return 表示添加成功

      \*/

    def add(ele: T): Boolean = {

**if** (head == **null**) { //如果是第一次添加元素, 则第一个节点置为 head 节点. 尾结点和头节点一致

            head = Node(ele, **null**) // 头节点一旦确定, 后续不再变化(除非删除)

            tail = head //

        } **else** { // 不是第一次添加, 则让尾节点的 next 指向新节点, 然后再更新尾节点到新节点

            tail.next = Node(ele, **null**)

            tail = tail.next // tail 指向新节点

        }

**true**

    }

    /\*\*

      \* 从链表中删除指定的元素

      \*

      \* @param ele

      \* @return 删除是否成功

      \*/

    def delete(ele: T): Boolean = {

        // 如果头节点为 null, 表示没有元素, 所以删除失败

**if** (head == **null**) **return** **false**

        // 如果头节点就是要删除, 则删除当前的头节点, 并把下一个节点设置为头节点. 如果删除的节点也是尾节点,则需要更新尾节点

**if** (head.vale == ele) {

**if** (head.eq(tail)) { // 必须比较是否为同一个对象. 等价于 java 的比较地址值是否相等

                tail = head.next // 更新尾节点

            }

            head = head.next // 更新头节点

**return** **true**

        } **else** { // 如果头节点不是要删除的节点, 则遍历后面的节点

            var currentNode: Node = head // 当前节点

            var nextNode: Node = currentNode.next // 下一个节点

**while** (nextNode != **null**) {

**if** (nextNode.vale == ele) { // 删除

                    currentNode.next = nextNode.next // 让当前节点指向下一个节点的下一个节点

**if** (nextNode.eq(tail)) { // 如果要删除的节点是尾节点, 尾节点需要指向当前节点

                        tail = currentNode

                    }

**return** **true**

                }

                currentNode = nextNode

                nextNode = currentNode.next

            }

        }

**false**

    }

    /\*\*

      \* 测试指定的元素是否存在

      \*

      \* @param ele

      \*/

    def contains(ele: T): Boolean = {

**if** (head == **null**) **return** **false**

        var tmp: Node = head

**do** {

**if** (tmp.vale == ele) **return** **true**

            tmp = tmp.next

        } **while** (tmp != **null**)

**false**

    }

    /\*\*

      \* 打印链表的元素

      \*/

    def printInfo(): Unit = {

**if** (head == **null**) **return**

        var tmp: Node = head

**do** {

            print(tmp.vale + "->")

            tmp = tmp.next

        } **while** (tmp != **null**)

        println()

    }

    /\*\*

      \* 链表中的节点

      \*

      \* @param vale

      \* @param next

      \*/

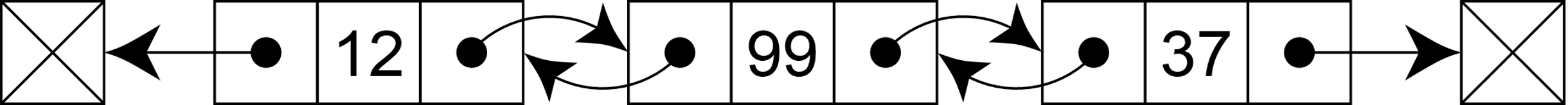
**case** **class** Node(vale: T, var next: Node)

}

### 实现双向链表(课堂不讲,学生自学)

双向链表也叫双链表。双向链表中不仅有指向后一个节点的指针，还有指向前一个节点的指针。这样可以从任何一个节点访问前一个节点，当然也可以访问后一个节点，以至整个链表。

一般是在需要大批量的另外储存数据在链表中的位置的时候用。双向链表也可以



**package** com.atguigu.datastruc.linked\_list

/\*\*

  \* Author lzc

  \* Date 2019-11-27 11:11

  \*/

object DoublyLinkedListDemo {

    def main(args: Array[String]): Unit = {

        val list = **new** DoublyLinkedList[Int]()

        list.add(10)

        list.add(20)

        list.add(30)

        list.printInfo()

        list.delete(30)

        list.add(40)

        list.printInfo()

    }

}

**class** DoublyLinkedList[T] {

**private** var head: Node = \_

**private** var tail: Node = \_

    /\*\*

      \* 删除节点

      \* 双向节点删除比较方便: 支持自删除

      \*

      \* @param ele

      \*/

    def delete(ele: T): Boolean = {

        // 找到要删除的节点

        val targetNode: Node = find(ele)

**if** (targetNode == **null**) { // 如果要删除的节点不存在

**false**

        }

**else** { // 删除的节点存在

            val preNode: Node = targetNode.pre // 上一个节点点

            val nextNode: Node = targetNode.next // 下一个节点

**if** (targetNode == head) { // 如果是头节点

                nextNode.pre = **null** // 下一个节点的 pre 指向 null

                head = nextNode // 更新头节点

            } **else** **if** (targetNode == tail) { // 如果是尾节点

                preNode.next = **null** // 上一个节点的 next 指向 null

                tail = preNode // 更新尾节点

            } **else** {

                preNode.next = nextNode

                nextNode.pre = preNode

            }

**true**

        }

    }

    /\*\*

      \* 找到要删除的元素所在的节点

      \*

      \* @param ele

      \*/

**private** def find(ele: T): Node = {

        var tmp: Node = head // 从头节点开始查找

**while** (tmp != **null**) {

**if** (tmp.value == ele) **return** tmp

            tmp = tmp.next

        }

**null**

    }

    /\*\*

      \* 新增节点

      \*

      \* @param ele

      \*/

    def add(ele: T): Boolean = {

        val newNode: Node = Node(ele, **null**, **null**)

**if** (head == **null**) { // 第一次添加

            head = newNode // 第一个元素, 上一个和下一个节点都应该是 null

        } **else** { // 不是第一次添加, tail 的next节点指向新节点, 新节点的pre节点指向 tail

            tail.next = newNode

            newNode.pre = tail

        }

        // 更新 tail 的指向

        tail = newNode

**true**

    }

    /\*\*

      \* 打印链表的元素

      \*/

    def printInfo(): Unit = {

**if** (head == **null**) **return**

        var tmp: Node = head

**do** {

            print(tmp.value + "->")

            tmp = tmp.next

        } **while** (tmp != **null**)

        println()

    }

**case** **class** Node(value: T, var pre: Node, var next: Node)

}

### 实现循环链表(课堂不讲,学生自学)

在一个循环链表中, 首节点和末节点被连接在一起。这种方式在单向和双向链表中皆可实现。

**package** com.atguigu.datastruc.linked\_list

/\*\*

  \* Author lzc

  \* Date 2019-11-27 12:29

  \*/

object CircularLinkedListDemo {

    def main(args: Array[String]): Unit = {

        val list = **new** CircularLinkedList[Int]()

        list.add(10)

        list.add(20)

        list.add(30)

        list.add(40)

        list.printInfo()

        println(list.delete(40))

        println(list.delete(0))

        list.printInfo()

    }

}

/\*\*

  \* 做一个双向的循环链表

  \*

  \* @tparam T

  \*/

**class** CircularLinkedList[T] **extends** DoublyLinkedList[T] {

**private** var \_len = 0

    // 返回循环列表中元素的个数

    def length: Int = \_len

    /\*\*

      \* 元素添加成功之后需要重新更新环

      \*

      \* @param ele

      \* @return

      \*/

    override def add(ele: T): Boolean = {

        // 添加到双向列表中

**super**.add(ele)

        head.pre = tail

        tail.next = head

        \_len += 1

**true**

    }

    /\*\*

      \* 删除环形链表中的节点

      \*

      \* @param ele

      \* @return

      \*/

    override def delete(ele: T): Boolean = {

**if** (**super**.delete(ele)) {

            head.pre = tail

            tail.next = head

\_len -= 1

**true**

        } **else** {

**false**

        }

    }

    override def printInfo(): Unit = {

**if** (head == **null**) **return**

        var tmp: Node = head

**while** (**true**) {

**if** (tmp.eq(tail)) {

                println(tmp.value)

**return**

            }

**else** {

                print(tmp.value + " -> ")

            }

            tmp = tmp.next

        }

    }

    // 当元素不存在的时候, 防止循环查找

    override def find(ele: T): Node = {

        var tmp: Node = head // 从头节点开始查找

**while** (tmp != **null**) {

**if** (tmp.value == ele) **return** tmp

**if**(tmp.eq(tail)) **return** **null** // 如果已经找到了尾节点, 则不再继续寻找

            tmp = tmp.next

        }

**null**

    }

}

#### 使用循环链表求解约瑟夫(Josephu )问题

* 问题描述:

设编号为1，2，… n的n个人围坐一圈，约定编号为k（1<=k<=n）的人从1开始报数，数到m 的那个人出列，它的下一位又从1开始报数，数到m的那个人又出列，依次类推，直至剩下一个人为止。

**package** com.atguigu.datastruc.linked\_list

/\*\*

  \* Author lzc

  \* Date 2019-11-27 14:05

  \*/

object Josephu {

    def main(args: Array[String]): Unit = {

        println("\n留下来的是: " + start(5, 1, 2) + " 号")

    }

    /\*\*

      \* n 个人围一圈

      \* 编号 k 的开始从 1 数

      \* 数到 m 的那个人出圈

      \*

      \* @param n

      \* @param k

      \* @param m

      \*/

    def start(n: Int, k: Int, m: Int) = {

        val list = **new** CircularLinkedList[Int]

        // 先在环形链表中初始化 n 个人, 每个人的编号依次为 1,2,3,...n

**for** (no <- 1 to n) {

            list.add(no)

        }

        // 找到开始数的节点的上一个节点 (然后是从下一个节点开始数 1)

        var startNode: list.Node = list.find(k).pre

        // 开始出圈, 当只剩下一个人的时候游戏结束

**while** (list.length != 1) {

**for** (num <- 1 to m) {

                startNode = startNode.next

            }

            // 删除 startNode 节点

            list.delete(startNode.value)

            print(startNode.value + " -> ")

            // startNode 再置为上一个节点

            startNode = startNode.pre

        }

        list.head.value

    }

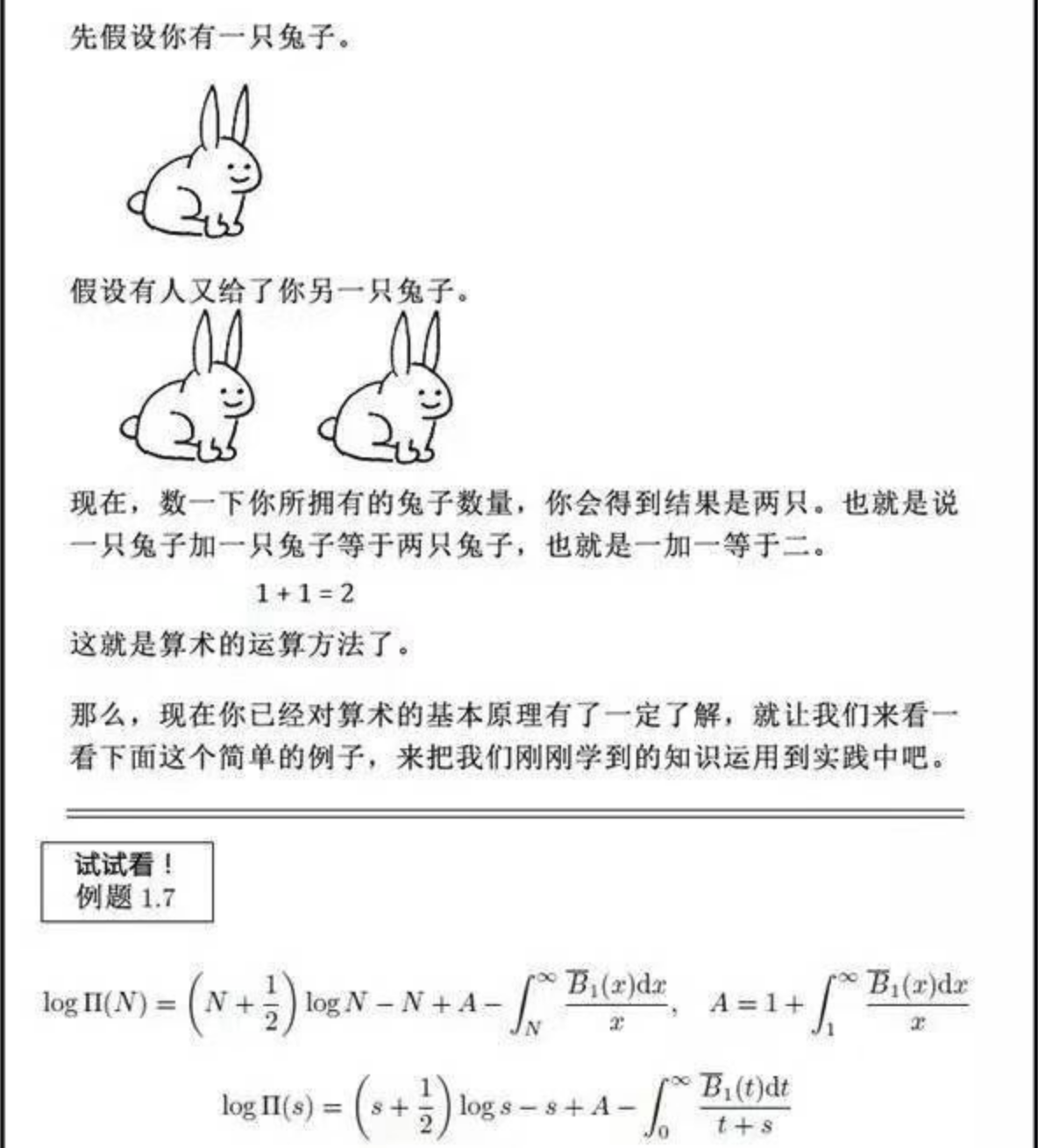
}

# 第 3 章 常见排序算法

参考动画: <http://www.webhek.com/post/comparison-sort.html>

## 什么是算法

维基定义: 算法（algorithm），在数学（算学）和计算机科学之中，为任何一系列良定义的具体计算步骤，常用于计算、数据处理和自动推理。作为一个有效方法，算法被用于计算函数，它包含了一系列定义清晰的指令，并可于有限的时间及空间内清楚的表述出来。



## 冒泡排序(课堂不讲)

**package** com.atguigu.datastruc.sort

/\*\*

  \* Author lzc

  \* Date 2019-11-27 14:49

  \*/

object Sort {

    def main(args: Array[String]): Unit = {

        val arr = Array(10, 2, 100, 2, 30, 5, 9, -10, -30)

        BubbleSort(arr)

        println(arr.toList)

    }

    /\*\*

      \* 交换给定的数组中的两个索引的位置的元素

      \*

      \* @param arr

      \* @param index1

      \* @param index2

      \*/

    def swap(arr: Array[Int], index1: Int, index2: Int): Unit = {

        val tmp: Int = arr(index1)

        arr(index1) = arr(index2)

        arr(index2) = tmp

    }

    /\*\*

      \* 冒泡排序

      \* 本质:

      \* 相邻的元素进行比较, 然后小的在前, 大的再后.  一轮下能成功排序 1 个

      \* 如何有 n 个元素, 则需要排序 n-1 轮

      \*

      \* @param arr

      \*/

    def BubbleSort(arr: Array[Int]): Unit = {

**for** (i <- 0 until arr.length - 1) { // 外层循环表示有多少元素(len - 1)需要排序

**for** (j <- 0 until arr.length - 1 - i) { // 每个元素需要比较的次数(第 1 个比较 len-1, 第 2 个比较 len-1-1, ...)

**if** (arr(j) > arr(j + 1)) { // 如果前面的大于后面的则交互二者的位置

                    swap(arr, j, j + 1)

                }

            }

        }

    }

}

## 选择排序

/\*\*

  \* 选择排序:

  \* 本质:

  \*     1. 首先选择第 1 个数为最小的数, 然后让这个数和后面所有的数比较,

  \* 一轮下来找到最小的数, 然后把最小的放在第 1 个位置

  \*     2. 再选择第 2 个数为第 2 小的数, 再后面逐次比较,.......

  \*

  \* @param arr

  \*/

def selectionSort(arr: Array[Int]): Unit = {

**for** (i <- 0 until arr.length - 1) { // 一共 len 个元素, 只需要找到 len-1 个就可以了, 剩下一个位置自动正确

        var minIndex: Int = i // 选中第 i 个元素为第 i 小(只记录索引即可, 成功之后交换一次元素即可)

**for** (j <- i + 1 until arr.length) { // 让第 i 个元素, 逐次与 i + 1 位置元素比较

**if** (arr(j) < arr(minIndex)) minIndex = j // 如果有比 minIndex 的位置更小的元素, 就记录下新的索引

        }

        //把 i 位置和 minIndex 位置的元素交换

**if** (i != minIndex) {

            swap(arr, i, minIndex)

        }

    }

}

## 插入排序

/\*\*

  \* 插入排序:

  \* 本质:

  \*     1. 类似于玩扑克牌, 手里的牌总是有序的.

  \* 每取一张牌, 只需要从后向前比较即可,一旦碰到比新牌小的, 新牌只需要插入到这张小牌的后面即可

  \*

  \*     2. 具体排序:  第 1 个元素是有序的, 第 2 个元素以前与前面的进行比较, 如果碰到大的就交互, 碰到小的停下来. 然后前 2 个元素排序完成

  \* 然后第 3 个元素....

  \*

  \*     3. n 个元素, 只需要排 n-1 次即可

  \*

  \* @param arr

  \*/

def insertionSort(arr: Array[Int]): Unit = {

**for** (i <- 0 until arr.length - 1) { // len-1 个元素需要排序

**for** (j <- i + 1 until(0, -1)) { // 从 i+1 的位置开始向前比较

**if** (arr(j) < arr(j - 1)) { // 后边的位置小于前面的位置, 就交互元素

                swap(arr, j, j - 1)

            }

        }

    }

}

## 快速排序

### 分治思想

许多有用的算法在结构上是递归的. 这些算法遵循分治法的思想, 将原问题分解为几个规模较小但类似于原问题的子问题, 然后再合并这些子问题的解来建立原问题的解.

分治模式在每层递归时都有三个步骤:

1. 分解. 分解原问题为若干子问题, 这些子问题是原问题的规模较小的实例
2. 解决. 解决这些子问题, 递归的求解各个子问题. 然而, 若子问题的规模足够小, 则直接求解.
3. 合并. 合并这些子问题的解成原问题的解.

### 快速排序描述

快速排序典型的使用的分治思想.

假设有数组 A, 长度为 len, 则元素为: A[0, …, m,.. len-1]

1. 分解:

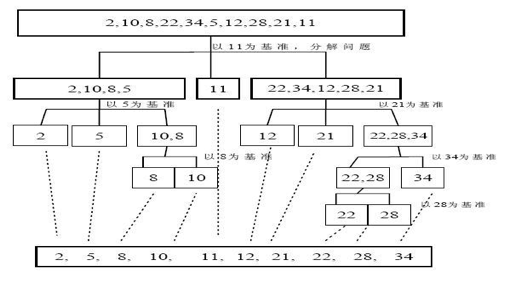
把数组A分成 2 个子数组, 使得左边数组的元素都小于一个参考值A[m], 右边的子数组的元素都大于参考值A[m]

1. 解决:

通过递归解决数组A[0,…m-1]和数组A[m+1, …, len]

1. 合并:

因为子数组都是原址排序的, 所以不需要合并, 原数组已经有序



### 快速排序具体实现

/\*\*

  \* 快速排序

  \*

  \* @param arr   需要排序的数组

  \* @param left  左扫描指针

  \* @param right 右扫描指针

  \*/

def quickSort(arr: Array[Int], left: Int, right: Int): Unit = {

    // 如果左指针超过或等于右指针, 则排序已经完成, 无序再排序

**if** (left >= right) **return**

    // 把数组分区, 保证左边的都小于等于某一个参数值, 右边的都大于等于某一个参考值.

    // 返回参考值所在的位置, 作为划分数组的边界.

    val m: Int = quickPartition(arr, left, right)

    // 再分别对左数组和右数组进行排序

    quickSort(arr, left, m - 1)

    quickSort(arr, m + 1, right)

}

/\*\*

  \* 快速排序分区

  \*  6, 5, 4, 1, 10, 8

  \* @param arr

  \* @param left

  \* @param right

  \*/

def quickPartition(arr: Array[Int], left: Int, right: Int): Int = {

    // 左扫描指针向右扫, 右扫描指针向左扫. 需要更改指针位置

    var l: Int = left

    var r: Int = right

    // 参考值: 保证左边的都小于 p, 右边的都大于等于 p

    val p: Int = arr(l)

    // 如果左扫描指针在左边就一直扫描

**while**(l < r){

        // 左指针找到一个大于 p 的, 右指针找到一个小于等于p的, 然后交换

**while**(l < right && arr(l) <= p){

            l += 1

        }

**while** (r > left && arr(r) > p ){

            r -= 1

        }

        // 小于等于p 的去左边, 大于 p 的去右边

**if**(l <= r){

            swap(arr, l, r)

        }

    }

    // 让参考元素去到正确的位置, 右指针已经到了小于 p 的区域, 所以可以和右指针的元素进行交互

    swap(arr, left, r)

    // 现在右指针就是分割位置

    r

}

## 归并排序

### 归并排序描述

归并排序算法也完全遵循分治模式.

1. 分解

待排序的数组分成2个子数组: A[0, …. len/2] 和 A[len/2 + 1, len - 1]

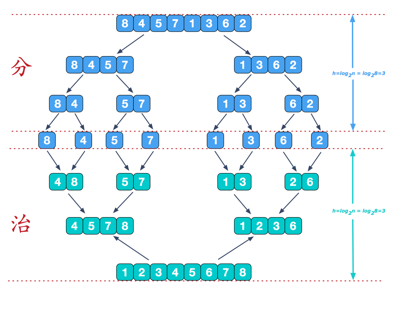
1. 解决

使用归并排序递归的地排序两个子序列

1. 合并

合并两个已排序的子序列

当待排序的序列长度为 1 时, 递归”开始回升”, 在这种情况下不需要做任何操作, 因为长度为1的序列都已经排好序了



### 归并排序具体实现

/\*\*

  \* 归并排序:

  \* 合并阶段

  \* 1 2 3 4

  \* 5 6 7 8 9 0

  \*

  \* @param arr

  \* @param low

  \* @param mid

  \* @param high

  \*/

@deprecated

def merge1(arr: Array[Int], low: Int, mid: Int, high: Int): Unit = {

    // 我们需要把  [low, mid] 的数据缓存到一个数组中, [mid+1, high]的数据缓存到另一个数组中,

    // 最后把这两部分归并到原数组中

    val left: Array[Int] = arr.slice(low, mid + 1) // 切割到的元素[low, mid+1) 前闭后开

    val right: Array[Int] = arr.slice(mid + 1, high + 1)

    // 扫描两个数组, 归并到原数组 arr 中

    var i = 0

    var j = 0

**for** (k <- low to high) {

**if** (i > left.length - 1) {

            arr(k) = right(j)

            j += 1

        } **else** **if** (j > right.length - 1) {

            arr(k) = left(i)

            i += 1

        } **else** **if** (left(i) <= right(j)) {

            arr(k) = left(i)

            i += 1

        } **else** {

            arr(k) = right(j)

            j += 1

        }

    }

}

/\*\*

  \* 改进版合并: 增加"哨兵", 简化比较的时候越界检测

  \*  使用最大值作为哨兵, 当有一个哨兵漏出来的时候, 其他的值肯定比不过哨兵, 当两个都漏出来的时候

正好也排好序了

  \* @param arr

  \* @param low

  \* @param mid

  \* @param high

  \*/

def merge(arr: Array[Int], low: Int, mid: Int, high: Int): Unit = {

    // 我们需要把  [low, mid] 的数据缓存到一个数组中, [mid+1, high]的数据缓存到另一个数组中,

    // 最后把这两部分归并到原数组中  给左右数组分别增加一个哨兵: Int 中的最大值

    val left: Array[Int] = arr.slice(low, mid + 1) :+ Int.MaxValue

    val right: Array[Int] = arr.slice(mid + 1, high + 1) :+ Int.MaxValue

    // 扫描两个数组, 归并到原数组 arr 中

    var i = 0

    var j = 0

**for** (k <- low to high) {

**if** (left(i) < right(j)) {

            arr(k) = left(i)

            i += 1

        } **else** {

            arr(k) = right(j)

            j += 1

        }

    }

}

/\*\*

  \* 归并排序入口

  \*

  \* @param arr

  \*/

def mergeSort(arr: Array[Int], low: Int, high: Int): Unit = {

**if** (low >= high) **return**

    // 计算出中间坐标

    val mid: Int = (low + high) / 2

    // 对左边排序

    mergeSort(arr, low, mid)

    // 对右边排序

    mergeSort(arr, mid + 1, high)

    // 合并

    merge(arr, low, mid, high)

}

## 希尔排序

### 希尔排序描述

希尔排序（Shellsort），也称递减增量排序算法，是插入排序的一种更高效的改进版本。希尔排序是非稳定排序算法。希尔排序因DL．Shell于1959年提出而得名。

希尔排序是基于插入排序的以下两点性质而提出改进方法的：

1. 在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率
2. 但插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位

**原理:**  希尔排序是将待排序的数组元素 按下标的一定增量分组 ，分成多个子序列，然后对各个子序列进行直接插入排序算法排序；然后依次缩减增量再进行排序，直到增量为1时，进行最后一次直接插入排序，排序结束。

假设数组: [ 13 14 94 33 82 25 59 94 65 23 45 27 73 25 39 10 ] 如果我们以步长为5开始进行排序

分5组(5列):

13 14 94 33 82

25 59 94 65 23

45 27 73 25 39

10

然后对每列进行排序(每列内进行排序)

10 14 73 25 23

13 27 94 33 39

25 59 94 65 82

45

分3组:

10 14 73

25 23 13

27 94 33

39 25 59

94 65 82

45

然后每组进行排序:

10 14 13

25 23 33

27 25 59

39 65 73

45 94 82

94

最后步长1进行排序, 就退化成了插入排序

### 希尔排序具体实现

/\*\*

  \* 希尔排序

  \* 本质:

  \*     1. 分组进行插入排序

  \*     2. 最后一步步长必须是 1

  \*

  \* @param arr

  \*/

def shellSort(arr: Array[Int]) = {

    var gap = arr.length / 2 //初始化步长: 长度的一半

**while** (gap > 0) { // 步长

**for** (i <- gap until arr.length) { //从每组的第二行开始, (插入排序)

            var j: Int = i - gap

**while**(j >= 0){

**if**(arr(j) > arr(j + gap)){  // 上一行和当前行进行比较

                    swap(arr, j, j + gap)

                }

                j -= gap

            }

        }

        gap /= 2 //降低步长

    }

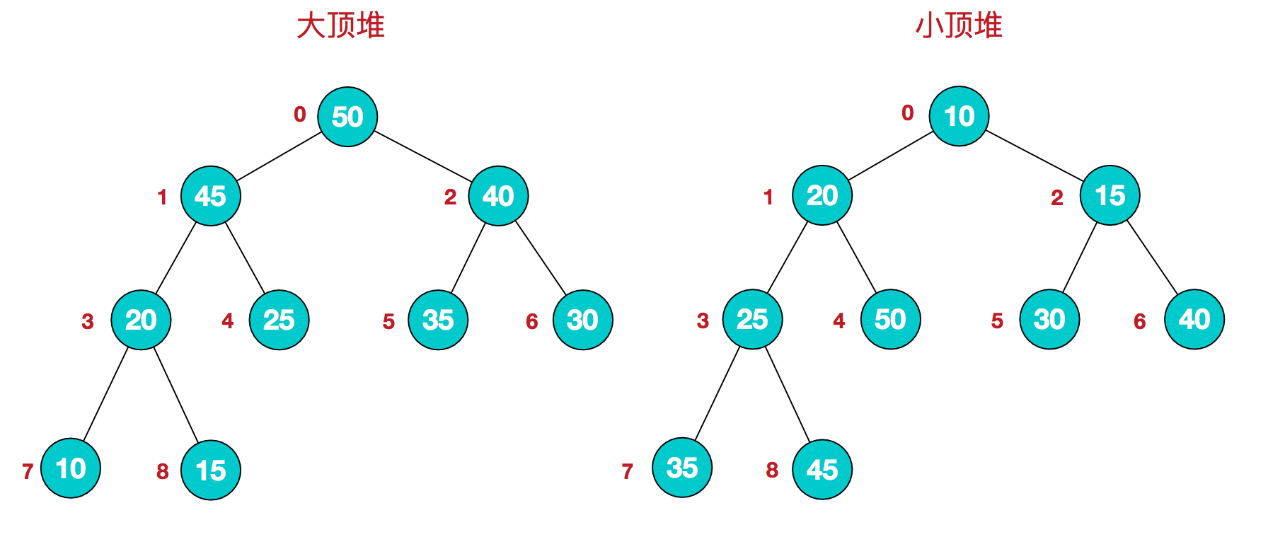
}

## 堆排序(学完二叉树再学)

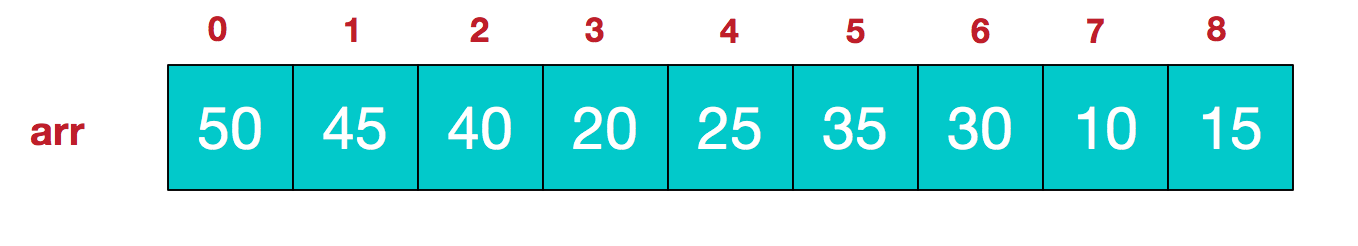
### 堆相关知识

堆是具有以下性质的完全二叉树(完全二叉树的一个优秀的性质就是，除了最底层之外，每一层都是满的)：

1. 每个节点的值都大于或等于其左右孩子节点的值，称为大顶堆；
2. 或者每个节点的值都小于或等于其左右孩子节点的值，称为小顶堆。



同时，我们对堆中的节点按层进行编号，将这种逻辑结构映射到数组中就是下面这个样子.



该数组从逻辑上讲就是一个堆结构，我们用简单的公式来描述一下堆的定义就是：

1. 大顶堆：

arr[i] >= arr[2i+1] && arr[i] >= arr[2i+2]

1. 小顶堆：

arr[i] <= arr[2i+1] && arr[i] <= arr[2i+2]

### 堆排序原理

堆排序是利用堆这种数据结构而设计的一种排序算法，堆排序是一种改进的选择排序.

#### 基本思想

1. 输入

一系列的无序元素（比如说，数字）组成的输入数组A

1. 处理

堆排序的过程可以具体分为三步:

1. 创建堆

以数组的形式将堆中所有的数据重新排序，使其成为最大堆/最小堆。

1. 调整堆

调整过程需要保证堆序性质：在一个二叉堆中任意父节点大于其子节点。

1. 堆排序

取出位于堆顶的第一个数据（最大堆则为最大数，最小堆则为最小数），放入输出数组B 中，再将剩下的对作调整堆的迭代/重复运算直至输入数组 A中只剩下最后一个元素。

1. 输出

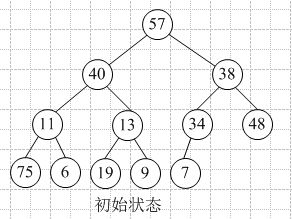
输出数组B，里面包含的元素都是A 中的但是已经按照要求排好了顺序

#### 实例分析

假设我们要对目标数组 A {57, 40, 38, 11, 13, 34, 48, 75, 6, 19, 9, 7}进行堆排序

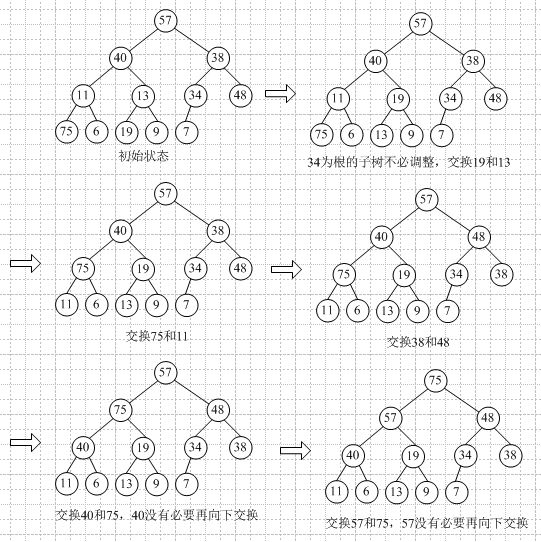
##### 创建堆

升序的情况下使用大顶推, 用给出的序列构造堆的初始状态如下：



##### 调整堆

从最后一个分支节点开始调整，目标是得到大顶堆

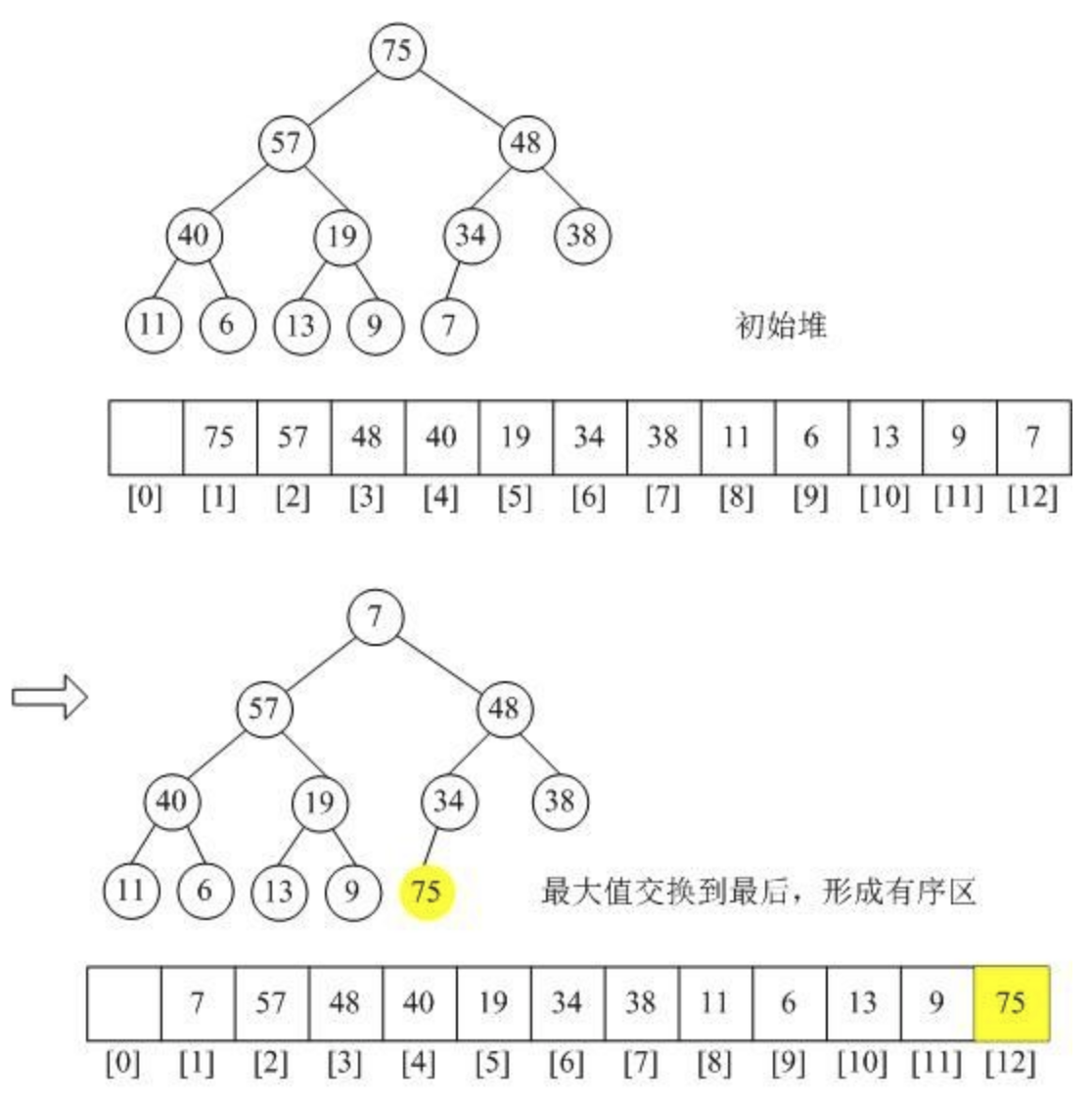


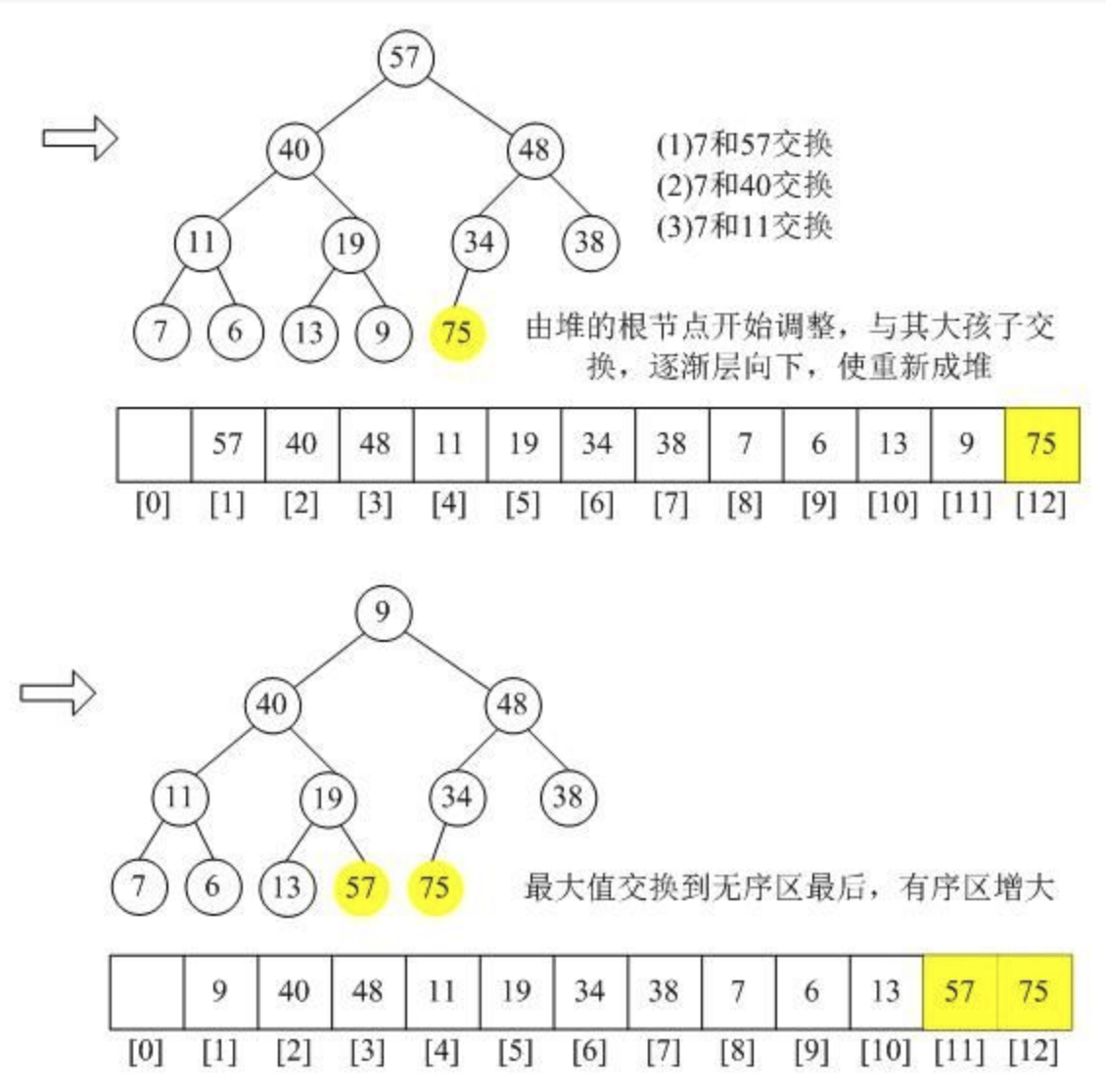
得到的大顶堆的存储结构如下:



##### 排序堆

最大值交换到最后, 形成有序区





直至所有数据排序完成

#### 一个结论

基于0的数组中

1. 位置i的元素, 他的左子节点是: 2i + 1, 右子节点是2i + 2
2. 他的父节点是 (i - 1) / 2

### 堆排序具体实现

/\*\*

  \*

  \* 堆排序 入口

  \* 第1步: 堆初始化

  \* 第2步: 堆调整(大顶堆)

  \* 第3步: 排序堆

  \*

  \* @param arr

  \*/

def heapSort(arr: Array[Int]): Unit = {

    // 建堆

    buildHeap(arr)

    // 调整堆

    // 排序   交换-> 调整-> 交互-> 调整

    val len: Int = arr.length

**for** (i <- len until(0, -1)) {

        swap(arr, 0, i - 1)

        maxHeapify(arr, 0, i - 1)

    }

}

/\*\*

  \* 建大顶堆

  \* 1. 从最右非子节点的开始构建(叶子节点可以看成已经符合堆要求的节点)

  \* 2. 调整符合堆的性质

  \*

  \* @param arr

  \* @param len

  \*/

def buildHeap(arr: Array[Int]): Unit = {

    val len: Int = arr.length // 未排序部分的长度

    val beginIndex: Int = (len - 1 - 1) / 2 // 开始索引(最后一个页节点的父节点)

**for** (i <- beginIndex to(0, -1)) { // 堆化

        maxHeapify(arr, i, len)

    }

}

/\*\*

  \* 将指定的数组堆化

  \*

  \* @param arr   需要堆化的数组

  \* @param index 需要堆化处理的数据的索引

  \* @param len   未排序的堆（数组）的长度

  \*/

@scala.annotation.tailrec

def maxHeapify(arr: Array[Int], index: Int, len: Int): Unit = {

    val li: Int = index \* 2 + 1 // 左子节点索引

    val ri: Int = li + 1 // 右子节点索引

**if** (li >= len) **return** // 如果左子节点超出计算范围, 直接返回

    // 找到最大的子节点

    var maxChild: Int = li

**if** (ri < len && arr(ri) > arr(li)) maxChild = ri

    // 父节点是否需要与子节点交互

**if** (arr(index) < arr(maxChild)) {

        swap(arr, index, maxChild) // 父子节点交换

        // 交换之后, 需要判断子节点是否还满足堆的特性(父>子)

        maxHeapify(arr, maxChild, len)

    }

}

## 算法的时间复杂度

### 如何评估算法的性能

数据结构和算法，本质上是解决现实存在的问题，即如何让解决指定问题的代码运行得更快？一个算法如果能在所要求的资源限制(resource constraint)内将问题解决好，则称这个算法是有效率的(effient)。

限制可分为空间限制和时间限制。

如何衡量算法的执行时间? 可以使用如下的代码测试:

void calcDuration(){

long begin = System.currentTimeMillis();

function(n); // 测试function的执行时间

long end = System.currentTimeMillis();

}

我们可以通过end-begin来计算出function()的执行时间。这类“事后统计”的方法用来评估算法执行效率是正确的，但是它也有它的局限性：

1. 受不同机器配置的影响，结果差异不同，甚至同一段算法，在不同机器上跑出来的结果差异很大。
2. 算法的测试结果受输入的规模影响很大，也就是说，输入不同规模的数据，结果差别非常大。

能不能事先“算”出算法的执行效率呢？这就引出了以下两个概念：时间复杂度，空间复杂度。

### 大O表示法

看下面的简单代码:

void foo***(){***

**for**(int i =0;i<n;i++)***{****//执行了n次*

System.*out*.println***(*"Hello world"*)***;*//执行了n次*

***}***

System.*out*.println***(*"function over"*)***;*//执行了1次*

***}***

在上述例子中，第1行的代码总共执行了n次，第2行的代码总共执行n次，第4行代码会执行1次。(这里假设计算机处理语句代价都为c)

假设算法总执行时间为，其中n为问题的规模, 那么上述代码的总执行时间就是:

可以看出，与代码的总执行次数总是成正比的，即与成正比. 我们将代码执行次数用来表示, 即:

有了执行次数后, 用大 (英语：Big O notation)来表示算法的复杂度, 即:

针对上述代码的时间复杂度:

在表示时间复杂度的时候, 大并不是表示代码真正的执行时间, 而是表示代码执行时间随着数据规模增长的变化趋势. 一般具有以下特点:

1. 不保留系数
2. 只保留最高阶的项(随着数据规模n的增长, 低阶项对结果的影响越来越小, 可以省去. 常数可以看成0阶)

可以把大O符号想象成一个过滤性的漏斗，过滤那些系数以及低阶的项。

比如: , 则时间复杂度记为:

时间复杂度又分: 最优时间复杂度, 平均时间复杂度, 最坏时间复杂度. 一般平均复杂度更有指导意义.

### 排序算法的复杂度

#### 冒泡算法时间复杂度计算

以冒泡算法为例, 我们算下冒泡算法的复杂度:

**def** bubbleSort***(***arr: Array***[***Int***])***: Unit = ***{***

**for *(***i <- 0 until arr.length - 1***) { //***

**for *(***j <- 0 until arr.length - 1 - i***) //***

**if *(***arr***(***j***)*** > arr***(***j + 1***)) {***

*swap****(***arr, j, j + 1***)***

***}***

***}***

***}***

***}***

最优(最初有序)情况: 每次都不需要交换, 则只需次数

所以时间复杂度是:

最差(最初逆序)情况: 每次都需要交换, 则每次增加三次交换:

所以时间复杂度是:

平均情况:

#### 其他排序算法时间复杂度



常见复杂度比较:

# 第 4 章 高级数据结构和算法

## 二分查找法 (课堂不讲)

在集合中查找元素不一般有两种方式:

1. 顺序查找

适用在无序集合中查找元素. 时间复杂度:

1. 二分查找(折半查找)

适用在有序集合中查找元素. 时间复杂度:

### 二分查找示例代码

**package** com.atguigu.datastruc.high

*/\*\**

*\* Author lzc*

*\* Date 2019-11-28 19:14*

*\*/*

**object** BinarySearchDemo ***{***

**def** main***(***args: Array***[***String***])***: Unit = ***{***

**val** arr = *Array****(***10, 20, 30, 50, 60, 100, 200, 1000***)***

*println****(****binarySearch****(***arr, 50, 0, arr.length - 1***))***

*println****(****binarySearch****(***arr, 0, 0, arr.length - 1***))***

***}***

*/\*\**

*\* 在有些数组 arr 中查找 ele 元素出现的索引*

*\**

*\** **@param arr** *有序数组(假设升序)*

*\** **@param ele** *要查找的元素*

*\** **@param low** *开始索引*

*\** **@param high** *结束索引*

*\** **@return** *出现的索引. 没有找到返回 -1*

*\*/*

@scala.annotation.tailrec

**def** binarySearch***(***arr: Array***[***Int***]***, ele: Int, low: Int, high: Int***)***: Int = ***{***

**if *(***low > high***)*** -1 *// 没有找到*

**else *{***

**val** midIndex: Int = ***(***low + high***)*** / 2 *// 折半查找*

**val** midValue: Int = arr***(***midIndex***)*** *// 折半位置的值*

**if *(***ele == midValue***)*** midIndex *// 如果相等, 则直接返回折半位置*

**else if *(***ele < midIndex***) {*** *// 元素小于折半位置的值, 则去左边查找*

*binarySearch****(***arr, ele, low, midIndex - 1***)***

***}* else *{*** *// 如果元素大于折半位置的值, 则去右边查找*

*binarySearch****(***arr, ele, midIndex + 1, high***)***

***}***

***}***

***}***

***}***

## 递归和回溯(迷宫问题)

### 什么是回溯

也称为试探法，它并不考虑问题规模的大小，而是从问题的最明显的最小规模开始逐步求解出可能的答案，并以此慢慢地扩大问题规模，迭代地逼近最终问题的解。这种迭代类似于穷举并且是试探性的，因为当目前的可能答案被测试出不可能可以获得最终解时，则撤销当前的这一步求解过程，回溯到上一步寻找其他求解路径。

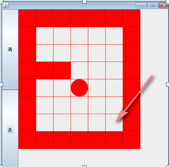
为了能够撤销当前的求解过程，必须保存上一步以来的求解路径，这一点相当重要。

### 回溯和递归的区别

1. 回溯算法是一种算法思想，是我们解决问题的策略；而递归是一种算法结构。
2. 递归就是函数调用本身，一般回溯法多用递归来实现。

### 回溯实例: 迷宫问题

给定一个迷宫，指明起点和终点，找出从起点出发到终点的有效可行路径，就是迷宫问题（maze problem）



**package** com.atguigu.datastruc.high

*/\**

*使用二维数组存储迷宫*

*在迷宫中元素约定:*

*0: 可以走, 还没有走*

*1: 墙, 不能走*

*2: 走过, 是否可以走通未知*

*3: 走过, 是死路*

*设置策略: 右->下->左->上*

*策略不同, 得到的路径是不同的*

*\*/*

**object** MazeProblem ***{***

*// 迷宫*

**var** *mazeMap*: Array***[***Array***[***Int***]]*** = \_

**val** *endI* = 8 *// 目的地坐标*

**val** *endJ* = 8

**def** main***(***args: Array***[***String***])***: Unit = ***{***

*// 1. 初始化迷宫*

*initMaze****(***10, 10***)***

*// 2. 查找路径*

*printMaze****()***

*setPath****(***1, 1***)***

*printMaze****()***

***}***

*/\*\**

*\* 测试路径*

*\* 策略: 左->上->右->下*

*\**

*\** **@param i** *出发地坐标*

*\** **@param j** *出发地坐标*

*\** **@return** *是否可以走通*

*\*/*

**def** setPath***(***i: Int, j: Int***)***: Boolean = ***{***

*// 表示路径已经找到, 查找路径结束*

**if *(****mazeMap****(****endI****)(****endJ****)*** == 2***)* return true**

**if *(****mazeMap****(***i***)(***j***)*** == 0***) {*** *// 表示此路第一次到达*

*// 把此路设置为 2, 表示走过*

*mazeMap****(***i***)(***j***)*** = 2

*// 开始递归回溯*

**if *(****setPath****(***i, j + 1***))* true** *// 先右走*

**else if *(****setPath****(***i + 1, j***))* true** *// 先下走*

**else if *(****setPath****(***i, j - 1***))* true** *// 向左走*

**else if *(****setPath****(***i - 1, j***))* true** *// 先上走*

**else *{***

*// 此路不通*

*mazeMap****(***i***)(***j***)*** = 3

**false**

***}***

***}* else *{*** *// 是 1 2 3*

**false**

***}***

***}***

*/\*\**

*\* 初始化迷宫地图*

*\**

*\** **@param rows** *迷宫的尺寸*

*\** **@param cols** *迷宫的尺寸*

*\*/*

**def** initMaze***(***rows: Int, cols: Int***)***: Unit = ***{***

*mazeMap* = Array.*ofDim****[***Int***](***rows, cols***)***

*// 设置地图: 四周设置为墙*

**for *(***i <- *mazeMap*.indices; j <- *mazeMap****(***i***)***.indices***) {***

**if *(***i == 0 || j == 0 || i == rows - 1 || j == cols - 1***)*** *mazeMap****(***i***)(***j***)*** = 1

***}***

*// 内部设置几堵墙*

*mazeMap****(***1***)(***3***)*** = 1

*mazeMap****(***2***)(***4***)*** = 1

*mazeMap****(***1***)(***6***)*** = 1

*mazeMap****(***4***)(***3***)*** = 1

*mazeMap****(***4***)(***8***)*** = 1

*mazeMap****(***5***)(***5***)*** = 1

*mazeMap****(***5***)(***7***)*** = 1

*mazeMap****(***5***)(***3***)*** = 1

*mazeMap****(***2***)(***7***)*** = 1

*mazeMap****(***3***)(***8***)*** = 1

*mazeMap****(***6***)(***6***)*** = 1

*mazeMap****(***3***)(***6***)*** = 1

***}***

*/\*\**

*\* 打印迷宫*

*\*/*

**def** printMaze***()*** = ***{***

*println****(*"------------------------------"*)***

**for *(***i <- *mazeMap*.indices; j <- *mazeMap****(***i***)***.indices***) {***

*print****(****mazeMap****(***i***)(***j***)*** + **" "*)***

**if *(***j == *mazeMap****(***i***)***.length - 1***)*** *println****()***

***}***

***}***

***}***

## 散列表

### 散列表概述

散列表（Hash table，也叫哈希表），是根据键（Key）而直接访问在内存存储位置的数据结构。也就是说，它通过计算一个关于键值的函数，将所需查询的数据映射到表中一个位置来访问记录，这加快了查找速度。这个映射函数称做散列函数，存放记录的数组称做散列表。

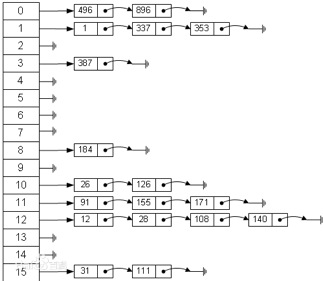
这里的对应关系称为散列函数, 又称为hash函数, 采用散列技术将记录存储在一块连续的存储空间中，这块连续存储空间称为散列表或哈希表（Hash table）

哈希表就是把key通过一个固定的算法函数(就是哈希函数)转换成一个整形数字, 然后就将该数字与数组长度取余, 取余结果就当做数组的下标, 然后将value存储在以该数字为下标的数组空间里.

而当使用哈希表进行查询的时候, 就会再次使用哈希函数将key转换为对应的数组下标, 并定位到该空间获取value, 如此一来, 就可以充分利用到数组的定位性能进行数据定位.

数组的特点是：寻址容易，插入和删除困难; 而链表的特点是：寻址困难，插入和删除容易. 我们可以综合两者的特性, 做出一种寻址容易, 插入删除也容易的数据结构, 这就是hash表.

哈希表有多重不同的实现方法, 我们要解释最常用的一种方法: 拉链法, 也可以理解为”链表的数组”. 如图:



### 实现散列表

需求: 有一个公司,当有新的员工来报道时,要求将该员工的信息加入(id,性别,年龄,住址..),当输入该员工的id时,要求查找到该员工的 所有信息.

要求: 不使用数据库,尽量节省内存,速度越快越好=>哈希表(散列)

**package** com.atguigu.datastruc.high

**import** com.atguigu.datastruc.linked\_list.DoublyLinkedList

*/\*\**

*\* Author lzc*

*\* Date 2019-11-29 17:44*

*\*/*

**object** HashTableDemo ***{***

**def** main***(***args: Array***[***String***])***: Unit = ***{***

**val** table = **new** HashTable***[***Employee***](***2***)***

table.add***(****Employee****(*"lisi"**, 20***))***

table.add***(****Employee****(*"zs"**, 10***))***

table.add***(****Employee****(*"zs"**, 30***))***

table.add***(****Employee****(*"zs"**, 4***))***

table.add***(****Employee****(*"ww"**, 2***))***

table.add***(****Employee****(*"ww"**, 2***))***

table.add***(****Employee****(*"ww"**, 2***))***

table.add***(****Employee****(*"w1w"**, 2***))***

table.printInfo***()***

***}***

***}***

**case class** Employee***(***name: String, age: Int***)***

*/\*\**

*\* 哈希表*

*\**

*\** **@param size** *哈希表中数组的容量*

*\** **@tparam T** *哈希表中存储的数据的类型*

*\*/*

**class** HashTable***[***T***](*val** size: Int***) {***

*// 创建一个存储双向链表的数组,*

**private val** *arr* = **new** Array***[***DoublyLinkedList***[***T***]](***size***)***

*// 数组长度*

**private val** *len* = *arr*.length

*/\*\**

*\* 向 hash 表中添加元素*

*\**

*\** **@param ele**

*\*/*

**def** add***(***ele: T***)*** = ***{***

**val** index = getIndex***(***ele***)***

*// 首先判断次数的链表是否已经创建, 如果没有需要先创建链表对象*

**if *(****arr****(***index***)*** == **null*)*** *arr****(***index***)*** = **new** DoublyLinkedList***[***T***]()***

*// 从数组中取出该位置的链表*

**val** list: DoublyLinkedList***[***T***]*** = *arr****(***index***)***

list.add***(***ele***)*** *// 数据添加此链表中*

**true**

***}***

*/\*\**

*\* 删除指定的元素*

*\**

*\** **@param ele**

*\*/*

**def** delete***(***ele: T***)***: Boolean = ***{***

**val** index: Int = getIndex***(***ele***)***

*arr****(***index***)*** != **null** && *arr****(***index***)***.delete***(***ele***)***

***}***

**def** getIndex***(***ele: T***)***: Int = ***{***

*// 通过对象的 hash 算法, 找到这个元素应该存储在数组中的哪个列表中*

ele.hashCode***()***.abs % *len*

***}***

**def** printInfo***() {***

**for *(***list <- *arr****) {***

list.printInfo***()***

***}***

***}***

***}***

## 树形数据结构介绍

### 为什么需要树形数据结构

* **数组的优点和缺点**

1. 优点：

通过下标方式访问元素，速度快。对于有序数组，还可使用二分查找提高检索速度。

1. 缺点：

如果要检索具体某个值，或者插入值(按一定顺序)会整体移动，效率较低

* **链式存储的优点和缺点**

1. 优点：

在一定程度上对数组存储方式有优化(比如：插入一个数值节点，只需要将插入节点，链接到链表中即可)

1. 缺点：

在进行检索时，效率仍然较低，比如(检索某个值，需要从头节点开始遍历)

* **树存储方式分析**

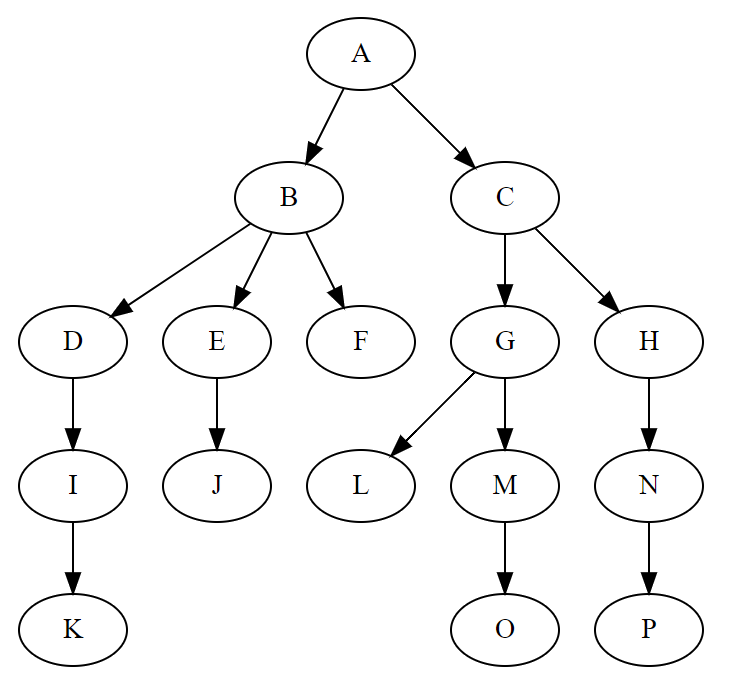
能提高数据存储，读取的效率, 比如利用 二叉排序树(Binary Sort Tree)，既可以保证数据的检索速度，同时也可以保证数据的插入，删除，修改的速度。

### 什么是树形结构

#### 树的概念

们更容易理解线性的数据结构(数组, 链表等)而不是树和图这种数据结构。树是非线性数据结构。它们不以线性方式存储数据。他们按层次组织数据.

树（英语：tree）是一种抽象数据类型（ADT: Abstract Data Type）或是实现这种抽象数据类型的数据结构，用来模拟具有树状结构性质的数据集合。它是由n（n>0）个有限节点组成一个具有层次关系的集合。把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。

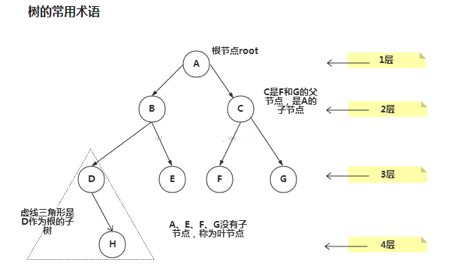


#### 树的特点

1. 每个节点都只有有限个子节点或无子节点；
2. 没有父节点的节点称为根节点；
3. 每一个非根节点有且只有一个父节点；
4. 除了根节点外，每个子节点可以分为多个不相交的子树；
5. 树里面没有环路(cycle)

#### 树的一些术语

1. 节点的度：一个节点含有的子树的个数称为该节点的度；
2. 树的度：一棵树中，最大的节点度称为树的度；
3. 叶节点：度为零的节点；
4. 非终端节点或分支节点：度不为零的节点；
5. 父亲节点或父节点：若一个节点含有子节点，则这个节点称为其子节点的父节点；
6. 孩子节点或子节点：一个节点含有的子树的根节点称为该节点的子节点；
7. 兄弟节点：具有相同父节点的节点互称为兄弟节点；
8. 节点的层次：从根开始定义起，根为第1层，根的子节点为第2层，以此类推；
9. 深度：对于任意节点n,n的深度为从根到n的唯一路径长，根的深度为0；
10. 高度：对于任意节点n,n的高度为从n到一片树叶的最长路径长，所有树叶的高度为0；
11. 堂兄弟节点：父节点在同一层的节点互为堂兄弟；
12. 节点的祖先：从根到该节点所经分支上的所有节点；
13. 子孙：以某节点为根的子树中任一节点都称为该节点的子孙。
14. 森林：由m（m>=0）棵互不相交的树的集合称为森林；



### 树的种类

* 无序树：树中任意节点的子节点之间没有顺序关系，这种树称为无序树，也称为自由树；
* 有序树：树中任意节点的子节点之间有顺序关系，这种树称为有序树；

1. 二叉树：每个节点最多含有两个子树的树称为二叉树；

* 完全二叉树
* 满二叉树
* 排序二叉树(二叉查找树（英语：Binary Search Tree))：
* 平衡二叉树（AVL树）

1. 霍夫曼树
2. B树

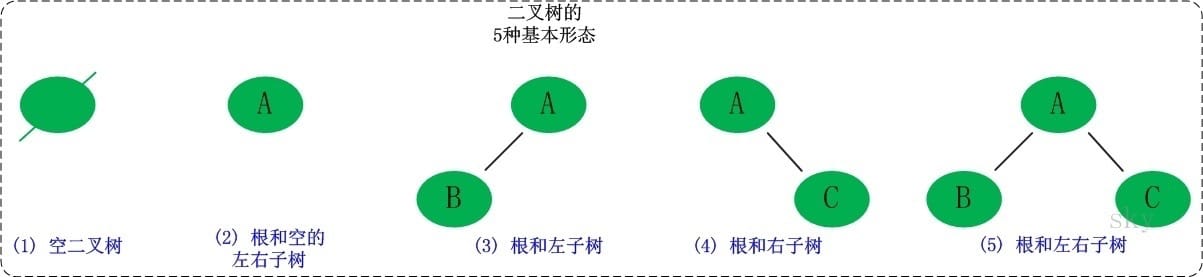
## 二叉树

### 二叉树的定义

二叉树是每个节点最多有两个子树（不存在度大于2的结点）的树结构。二叉树的子树有左右之分，次序不能颠倒。

二叉树有5种基本形态：

1. 二叉树可以是空集
2. 可以有空的左子树
3. 可以有空右子树；
4. 左右子树皆为空。
5. 左右子树都有



### 二叉树的3个主要性质

#### 性质1

第层最多有个节点

#### 性质2

深度为的二叉树的至多有个节点 (根节点的深度是 0 )

或者层数为的二叉树的至多有个节点

#### 性质3

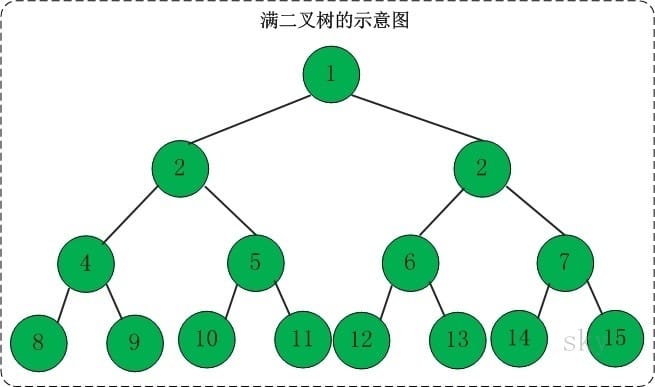
非空的二叉树，若树叶总数为，分支度为2的总数为 ，则。

### 二叉树的种类

#### 满二叉树(full binary tree)

层数为, 有个节点的二叉树就是满二叉树

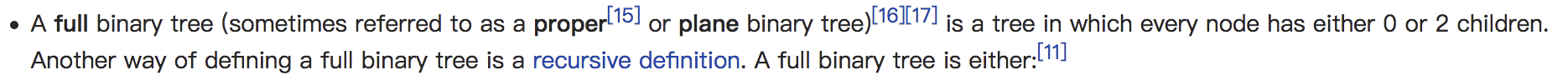
1. 第层的节点数量是:
2. 总节点数:



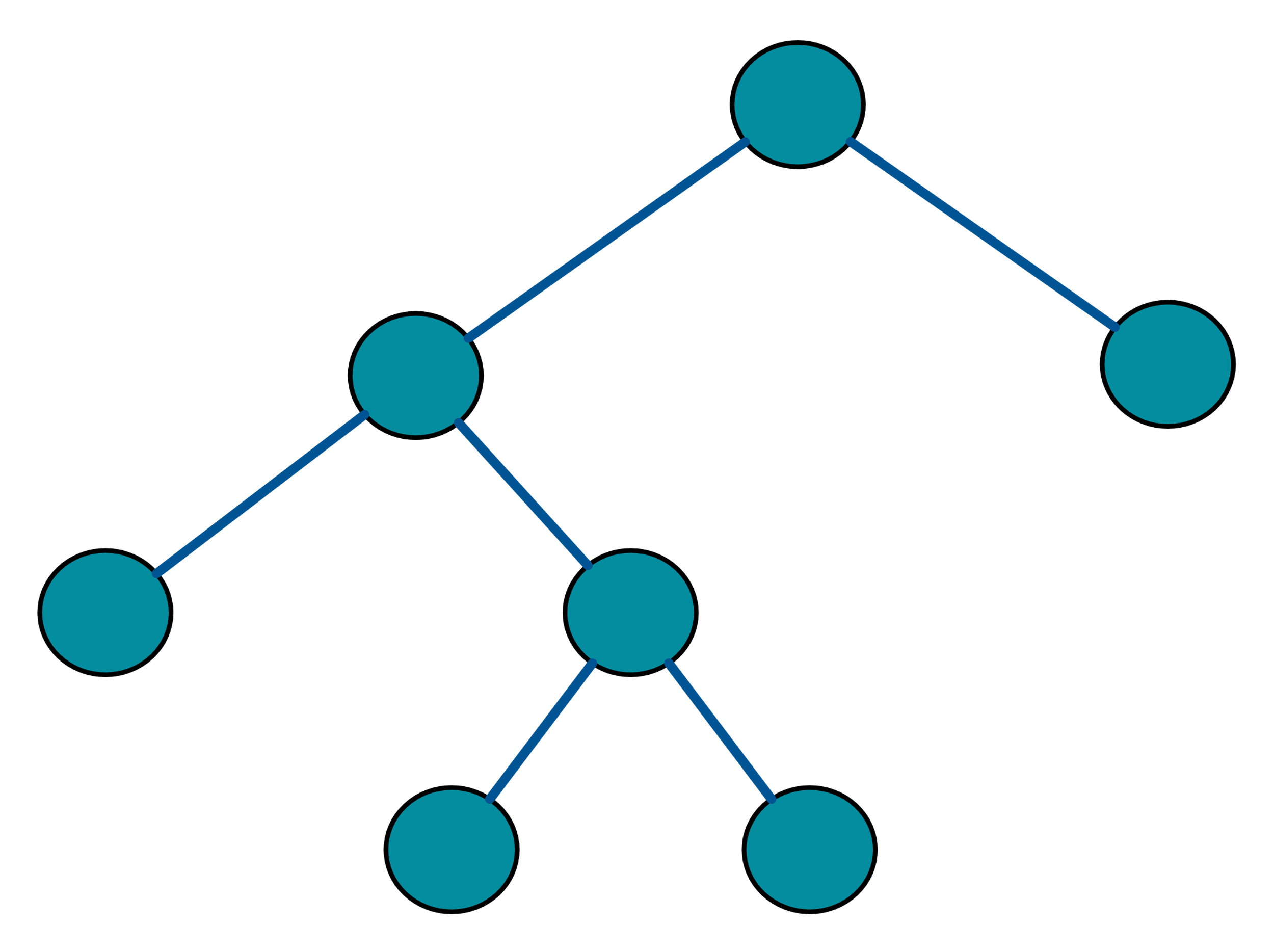
##### 关于满二叉树和完美二叉数

国内和国外这块是由歧义的

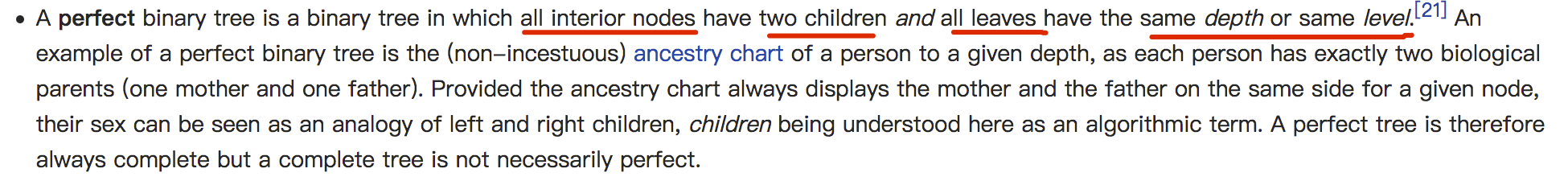
1. 维基百科对满二叉树的定义



一个节点要么没有子节点, 要么有两个子节点, 这样的数叫满二叉树.



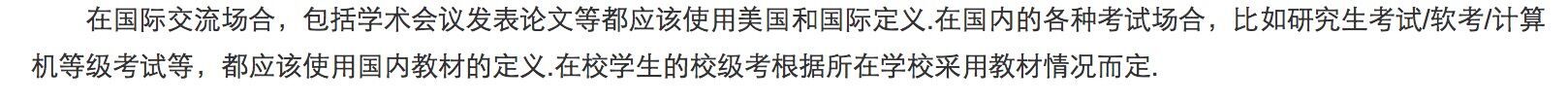
1. 维基百科对完美二叉树的定义



所有的非叶子节点都有两个孩子, 所有的叶子节点都在同一层.

1. 歧义解释





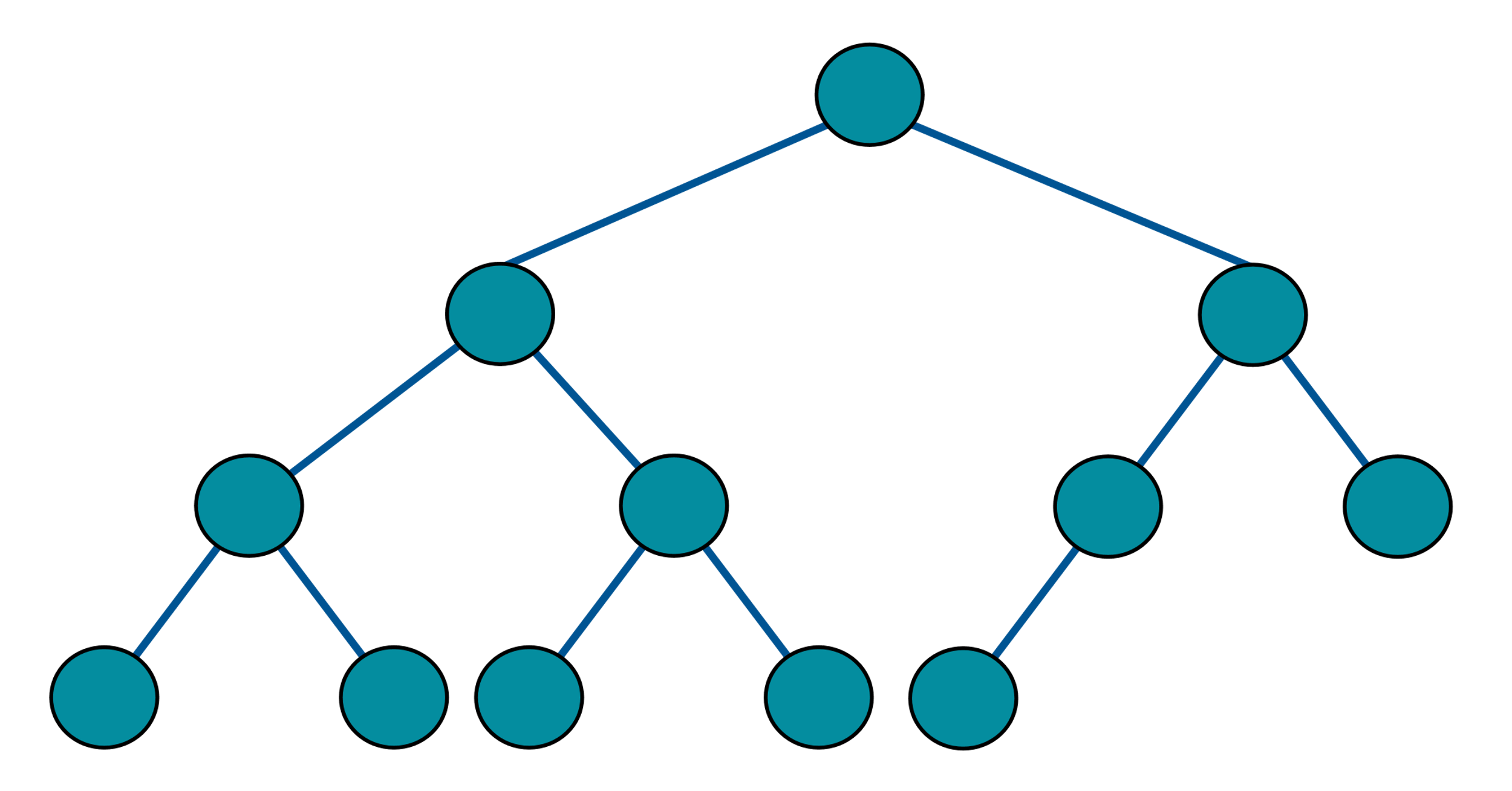
1. 我认为

我觉着是国内的大佬们应该是从字面上理解错了, 把人家的完美二叉树理解成了满二叉树.

#### 完全二叉树(Complete Binary Tree)

如果该二叉树的所有叶子节点都在最后一层或者倒数第二层，而且最后一层的叶子节点在左边连续，倒数第二层的叶子节点在右边连续，我们称为完全二叉树。

(除最后一层外，若其余层都是满的，并且最后一层或者是满的，或者是在右边缺少连续若干节点)



完全二叉树是效率很高的数据结构，堆是一种完全二叉树，所以效率极高. 在做堆排序的时候, 就是把数组看成一个完全二叉树来对待.

#### 平衡二叉树(AVL数)

AVL树得名于它的发明者G. M. Adelson-Velsky和Evgenii Landis，他们在1962年的论文《An algorithm for the organization of information》中公开了这一数据结构。

定义: 这种左右子树的高度相差不超过 1 的树为平衡二叉树。

平衡之意，如天平，即两边的分量大约相同。【对二叉搜索树做平衡】

#### 二叉搜索树

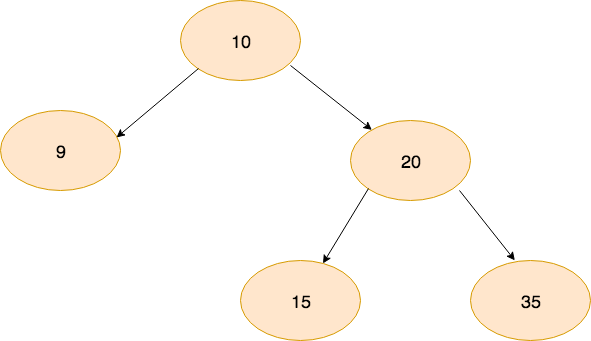
二叉查找树（英语：Binary Search Tree），也称为二叉搜索树、有序二叉树（ordered binary tree）或排序二叉树（sorted binary tree），是指一棵空树或者具有下列性质的二叉树：

1. 若任意节点的左子树不空，则左子树上所有节点的值均小于它的根节点的值；
2. 若任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值；
3. 任意节点的左、右子树也分别为二叉查找树；
4. 没有键值相等的节点。

### 二叉树操作相关代码

#### 静态初始化一个二叉树

如下图的二叉树



**package** com.atguigu.datastruc.high

*/\*\**

*\* Author lzc*

*\* Date 2019-12-01 19:35*

*\*/*

**object** BinaryTree ***{***

**def** main***(***args: Array***[***String***])***: Unit = ***{***

***// 初始化一个二叉树***

**val** tree: BinaryTree***[***Int***]*** = *initOneBinaryTree****()***

***}***

*// 初始化一个二叉树*

**def** initOneBinaryTree***()***: BinaryTree***[***Int***]*** = ***{***

*// 创建 5 个节点*

**val** n10 = **new** TreeNode***[***Int***](***10***)***

**val** n9 = **new** TreeNode***[***Int***](***9***)***

**val** n20 = **new** TreeNode***[***Int***](***20***)***

**val** n15 = **new** TreeNode***[***Int***](***15***)***

**val** n35 = **new** TreeNode***[***Int***](***35***)***

*// 设置 n10 的左右子节点*

n10.*left* = n9

n10.*right* = n20

*// 设置 n9 的父节点*

n9.*p* = n10

*// 设置 n20 的左右子节点和父节点*

n20.*p* = n10

n20.*left* = n15

n20.*right* = n35

*// 设置 n15 的父节点*

n15.*p* = n20

*// 设置 n35 的父节点*

n35.*p* = n20

*// 创建一个二叉树对象*

**val** tree = **new** BinaryTree***[***Int***]***

*// 把根节点设置为 n10*

tree.*root* = n10

*// 返回二叉树*

tree

***}***

***}***

**class** BinaryTree***[***T***] {***

**var** *root*: TreeNode***[***T***]*** = \_

***}***

*/\*\**

*\* 定义二叉树中的节点*

*\**

*\** **@param value** *每个几点的值域的值*

*\** **@tparam T** *值域的避类型*

*\*/*

**class** TreeNode***[***T***](*val** value: T***) {***

*// 左子节点*

**var** *left*: TreeNode***[***T***]*** = \_

*// 右子节点*

**var** *right*: TreeNode***[***T***]*** = \_

*// 双亲节点(父节点)*

**var** *p*: TreeNode***[***T***]*** = \_

***}***

#### 遍历二叉树

二叉树有3中遍历方式: 先序遍历, 中序遍历,后序遍历

##### 先序遍历

先访问根节点，然后访问左节点，最后访问右节点(根->左->右)

**package** com.atguigu.datastruc.high

*/\*\**

*\* Author lzc*

*\* Date 2019-12-01 19:35*

*\*/*

**object** BinaryTree ***{***

**def** main***(***args: Array***[***String***])***: Unit = ***{***

**val** tree: BinaryTree***[***Int***]*** = *initOneBinaryTree****()***

tree.preForeach***(****println****)***

***}***

**class** BinaryTree***[***T***] {***

***//….省略部分***

*/\*\**

*\* 先序遍历二叉树*

*\**

*\** **@param op**

*\*/*

**def** preForeach***(***op: T => Unit***)***: Unit = ***{***

**if *(****root* != **null*) {***

*root*.preForeach***(***op***)***

***}***

***}***

***}***

*/\*\**

*\* 定义二叉树中的节点*

*\**

*\** **@param value** *每个几点的值域的值*

*\** **@tparam T** *值域的避类型*

*\*/*

**class** TreeNode***[***T***](*val** value: T***) {***

*// ….省略部分*

*/\*\**

*\* 前序遍历*

*\**

*\** **@param op**

*\*/*

**def** preForeach***(***op: T => Unit***)***: Unit = ***{***

*// 遍历当前节点*

op***(***value***)***

*// 遍历左子节点 一定进行非空判断*

**if *(****left* != **null*)*** *left*.preForeach***(***op***)***

*// 遍历右子节点*

**if *(****right* != **null*)*** *right*.preForeach***(***op***)***

***}***

***}***



##### 中序遍历

先访问左节点，然后访问根节点，最后访问右节点(左->根->右)

**package** com.atguigu.datastruc.high

*/\*\**

*\* Author lzc*

*\* Date 2019-12-01 19:35*

*\*/*

**object** BinaryTree ***{***

**def** main***(***args: Array***[***String***])***: Unit = ***{***

**val** tree: BinaryTree***[***Int***]*** = *initOneBinaryTree****()***

*// tree.preForeach(x => println(x))*

tree.infixForeach***(****println****)***

***}***

***}***

**class** BinaryTree***[***T***] {***

***// ...省略***

*/\*\**

*\* 中序遍历*

*\**

*\** **@param op**

*\*/*

**def** infixForeach***(***op: T => Unit***)***: Unit = ***{***

**if *(****root* != **null*)*** *root*.infixForeach***(***op***)***

***}***

***}***

*/\*\**

*\* 定义二叉树中的节点*

*\**

*\** **@param value** *每个几点的值域的值*

*\** **@tparam T** *值域的避类型*

*\*/*

**class** TreeNode***[***T***](*val** value: T***) {***

***// ...省略***

*/\*\**

*\* 中序遍历*

*\**

*\** **@param op**

*\** **@return**

*\*/*

**def** infixForeach***(***op: T => Unit***)***: Unit = ***{***

*// 先遍历子左节点*

**if *(****left* != **null*)*** *left*.infixForeach***(***op***)***

*// 再遍历当前节点*

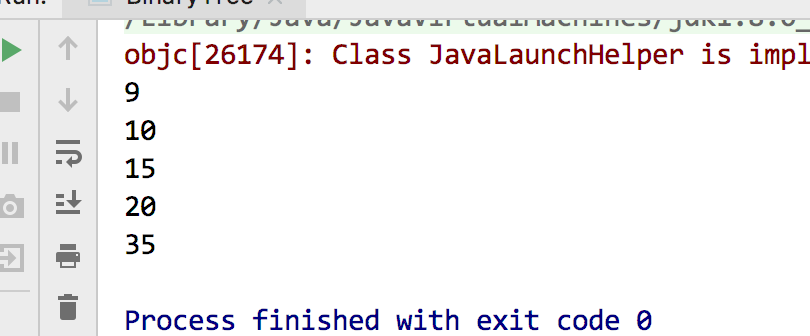
op***(***value***)***

*// 最后遍历有子节点*

**if *(****right* != **null*)*** *right*.infixForeach***(***op***)***

***}***

***}***



##### 后序遍历

先访问左节点，然后访问右节点，最后访问根节点(左->右->根)

**package** com.atguigu.datastruc.high

*/\*\**

*\* Author lzc*

*\* Date 2019-12-01 19:35*

*\*/*

**object** BinaryTree ***{***

**def** main***(***args: Array***[***String***])***: Unit = ***{***

**val** tree: BinaryTree***[***Int***]*** = *initOneBinaryTree****()***

*// tree.preForeach(x => println(x))*

*// tree.infixForeach(println)*

tree.postForeach***(****println****)***

***}***

***}***

**class** BinaryTree***[***T***] {***

**var** *root*: TreeNode***[***T***]*** = \_

// ...省略部分

**def** postForeach***(***op: T => Unit***)*** = ***{***

**if *(****root* != **null*)*** *root*.postForeach***(***op***)***

***}***

***}***

*/\*\**

*\* 定义二叉树中的节点*

*\**

*\** **@param value** *每个几点的值域的值*

*\** **@tparam T** *值域的避类型*

*\*/*

**class** TreeNode***[***T***](*val** value: T***) {***

*// ...省略部分*

*/\*\**

*\* 后序遍历*

*\**

*\** **@param op**

*\** **@return**

*\*/*

**def** postForeach***(***op: T => Unit***)***: Unit = ***{***

*// 先遍历左子节点*

**if *(****left* != **null*)*** *left*.postForeach***(***op***)***

*// 先遍历右子节点*

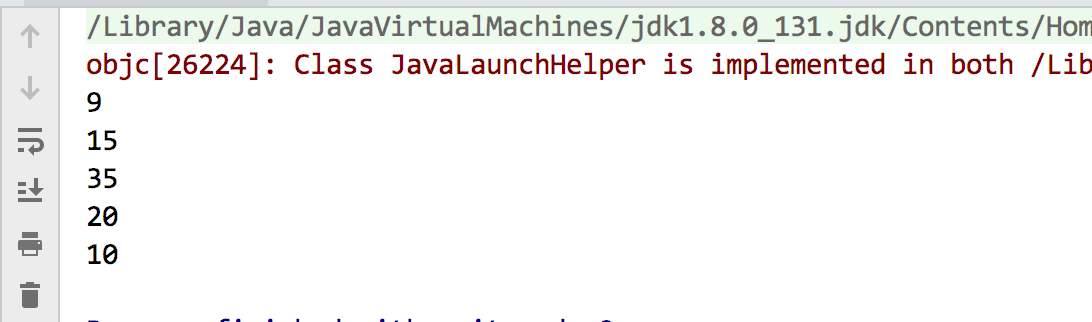
**if *(****right* != **null*)*** *right*.postForeach***(***op***)***

*// 最后遍历当前*

op***(***value***)***

***}***

***}***



#### 在二叉树中查找数据

使用遍历的方式查找. 不同的遍历方式, 需要比较的次数是不同的!

代码省略…

#### 在二叉树中删除数据

删除节点的处理方式:

1. 叶子节点直接删除
2. 非叶子节点则删除以这个节点为根节点的子树.

**package** com.atguigu.datastruc.high

*/\*\**

*\* Author lzc*

*\* Date 2019-12-01 19:35*

*\*/*

**object** BinaryTree ***{***

**def** main***(***args: Array***[***String***])***: Unit = ***{***

**val** tree: BinaryTree***[***Int***]*** = *initOneBinaryTree****()***

*// tree.preForeach(x => println(x))*

*// tree.infixForeach(println)*

tree.postForeach***(****println****)***

*println****(*"-----"*)***

*println****(***tree.deleteNode***(***15***))***

tree.postForeach***(****println****)***

***}***

***}***

**class** BinaryTree***[***T***] {***

*/\*\**

*\* 先序遍历的方式删除指定的元素*

*\**

*\** **@param ele**

*\** **@return**

*\*/*

**def** deleteNode***(***ele: T***)***: Boolean = ***{***

**if *(****root* == **null*)* false** *// 如果是空, 则删除失败*

**else *{***

**if *(****root*.value == ele***) { // 如果根节点是要删除的对象, 则置为null***

*root* = **null**

**true**

***}* else *{ // 否则递归删除***

*root*.deleteNode***(***ele***)***

***}***

***}***

***}***

***}***

*/\*\**

*\* 定义二叉树中的节点*

*\**

*\** **@param value** *每个几点的值域的值*

*\** **@tparam T** *值域的避类型*

*\*/*

**class** TreeNode***[***T***](*val** value: T***) {***

*/\*\**

*\* 先序删除*

*\**

*\** **@param ele**

*\** **@return**

*\*/*

**def** deleteNode***(***ele: T***)***: Boolean = ***{***

*// 先判断左子节点是否要删除的节点*

**if *(*this**.*left* != **null** && **this**.*left*.value == ele***) {***

*left* = **null**

**return true**

***}***

*// 再判断右子节点是否要删除的节点*

**if *(*this**.*right* != **null** && **this**.*right*.value == ele***) {***

*right* = **null**

**return true**

***}***

*// 递归删除左子节点*

**if *(*this**.*left* != **null** && **this**.*left*.deleteNode***(***ele***)) {***

**return true**

***}***

*// // 递归删除右子节点*

**if *(*this**.*right* != **null** && **this**.*right*.deleteNode***(***ele***)) {***

**return true**

***}***

*// 如果前面都没有删除成功, 则删除失败: 没有要删除的元素*

**false**

***}***

***}***

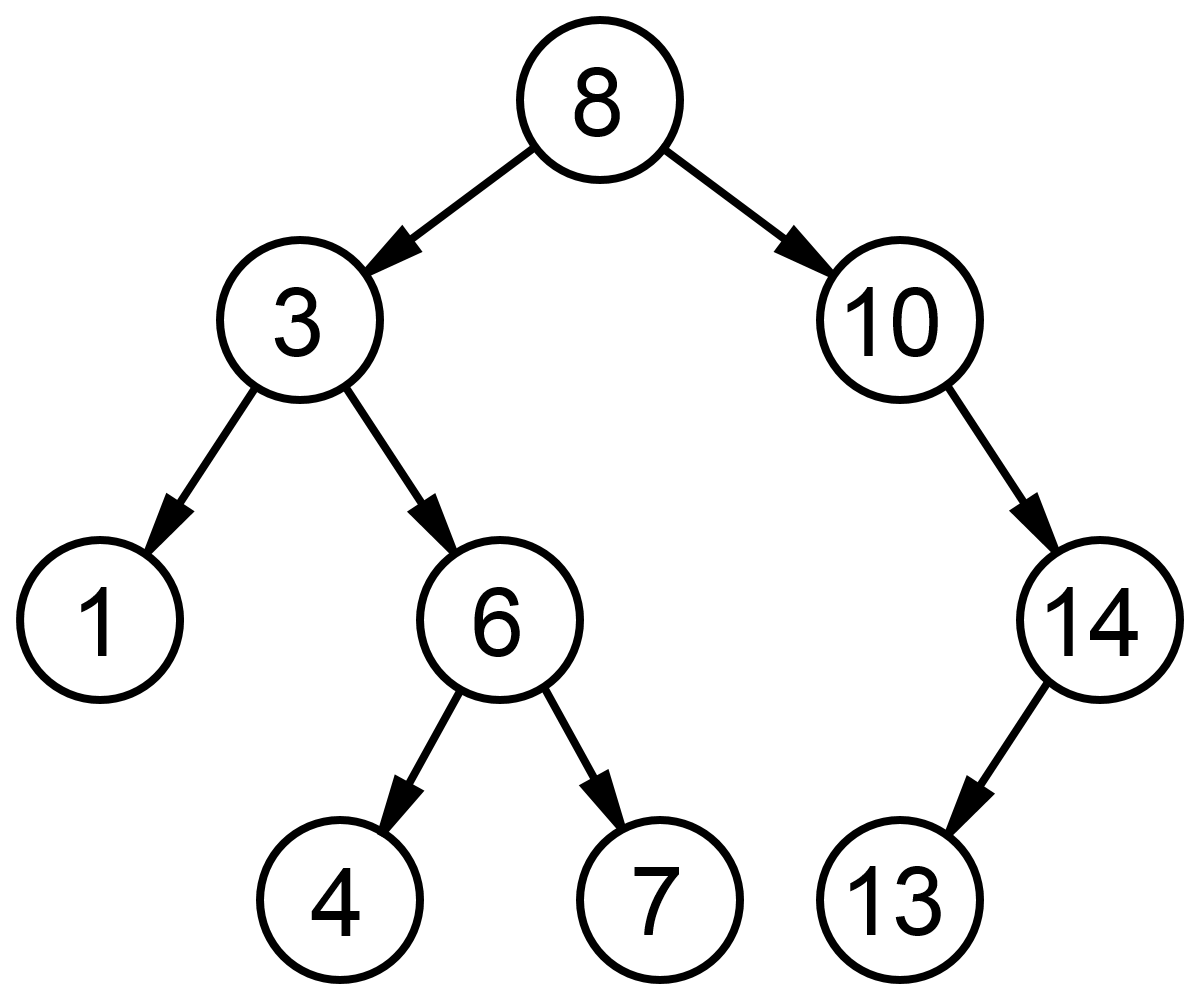
## 排序二叉树

### 排序二叉树定义

排序二叉树, 又称二叉查找树、二叉搜索树、有序二叉树。可以是一棵空树或者具有下列性质的二叉树：

1. 对任意节点来说，若左子树不为空， 则左子树上所有节点的值均小于等于他的根节点的值。
2. 对任意节点来说，若右子树不为空， 则右子树上所有节点的值均大于等于他的根节点的值。
3. 任意节点的左、右子树也分别为二叉查找树。

* **下图是一个3层的二叉搜索树**



二叉查找树相比于其他数据结构的优势在于查找、插入的时间复杂度较低。为。二叉查找树是基础性数据结构，用于构建更为抽象的数据结构，如集合、多重集、关联数组等。

### 构建一个排序二叉树

我们可以从数组构建一个排序二叉树

**package** com.atguigu.datastruc.high.search\_binary\_tree

*/\*\**

*\* Author lzc*

*\* Date 2019-12-02 18:42*

*\*/*

**object** SearchBinaryTreeDemo ***{***

**def** main***(***args: Array***[***String***])***: Unit = ***{***

**val** arr = *Array****(***8, 4, 9, 10, 1, 6, 7, 12***)***

**val** sbt = *initSBT****(***arr***)***

sbt.infixForeach***(****println****)***

***}***

*// 初始化一个排序二叉树*

**def** initSBT***(***arr: Array***[***Int***])*** = ***{***

**val** sbt = **new** SearchBinaryTree***[***Int***]()***

arr.foreach***(***x => ***{***

sbt.add***(***x***)***

***})***

sbt

***}***

***}***

**class** SearchBinaryTree***[***T: Ordering***] {***

*// 二叉查找树的根节点*

**var** *root*: TreeNode***[***T***]*** = \_

*// 向排序二叉树中添加节点*

**def** add***(***ele: T***)***: Unit = ***{***

*// 如果 root 节点为 null, 则把元素置为 root 位置*

**if *(****root* == **null*) {***

*root* = **new** TreeNode***[***T***](***ele***)***

***}* else *{*** *// 如果 root 节点不为空则调用 root 的 add 方法来添加元素*

*root*.add***(***ele***)***

***}***

***}***

*/\*\**

*\* 中序遍历*

*\**

*\** **@param op**

*\*/*

**def** infixForeach***(***op: T => Unit***)***: Unit = ***{***

**if *(****root* != **null*)*** *root*.infixForeach***(***op***)***

***}***

***}***

**class** TreeNode***[***T: Ordering***](*vars** value: T***) {***

*// 左子节点*

**var** *left*: TreeNode***[***T***]*** = \_

*// 右子节点*

**var** *right*: TreeNode***[***T***]*** = \_

*// 双亲节点(父节点)*

**var** *p*: TreeNode***[***T***]*** = \_

*/\*\**

*\* 中序遍历*

*\**

*\** **@param op**

*\*/*

**def** infixForeach***(***op: T => Unit***)***: Unit = ***{***

*// 先遍历子左子节点*

**if *(****left* != **null*)*** *left*.infixForeach***(***op***)***

*// 再遍历当前节点*

op***(***value***)***

*// 最后遍历有子节点*

**if *(****right* != **null*)*** *right*.infixForeach***(***op***)***

***}***

*/\*\**

*\* 插入数据*

*\**

*\** **@param ele**

*\*/*

**def** add***(***ele: T***)***: Unit = ***{***

*// 从冥界召唤可以比较 T 类型的比较器(Ordering)*

**val** ord: Ordering***[***T***]*** = *implicitly****[***Ordering***[***T***]]***

**if *(***ord.lteq***(***ele, value***)) {*** *// 如果小于等于当前值, 则在左边添加*

**if *(****left* == **null*) {*** *// 如果左节点是 null, 则左节点置为新节点*

*left* = **new** TreeNode***[***T***](***ele***)***

***left.p = this***

***}* else *{*** *// 否则递归的在左节点添加节点*

*left*.add***(***ele***)***

***}***

***}* else *{***

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

*right* = **new** TreeNode***[***T***](***ele***)***

***right.p = this***

***}* else *{***

*right*.add***(***ele***)***

***}***

***}***

***}***

***}***

### 查找节点

**package** com.atguigu.datastruc.high.search\_binary\_tree

*/\*\**

*\* Author lzc*

*\* Date 2019-12-02 18:42*

*\*/*

**object** SearchBinaryTreeDemo ***{***

**def** main***(***args: Array***[***String***])***: Unit = ***{***

**val** arr = *Array****(***8, 4, 9, 10, 1, 6, 7, 12***)***

**val** sbt: SearchBinaryTree***[***Int***]*** = *initSBT****(***arr***)***

sbt.infixForeach***(****println****)***

*println****(***sbt.search***(***80***))***

***}***

***//…省略部分***

***}***

**class** SearchBinaryTree***[***T: Ordering***] {***

***//...省略部分***

*/\*\**

*\* 查找指定的元素是否存在*

*\**

*\** **@param ele**

*\** **@return**

*\*/*

**def** search***(***ele: T***)***: Boolean = ***{***

**if *(****root* == **null*)* false** *// 如果 root 是空, 直接返回 false*

**else** *root*.search***(***ele***)*** *// 从 root 节点开始查找*

***}***

***}***

**class** TreeNode***[***T: Ordering***](*var** value: T***) {***

***// ...省略部分***

*/\*\**

*\* 查找元素*

*\**

*\** **@param ele**

*\** **@return**

*\*/*

**def** search***(***ele: T***)***: Boolean = ***{***

**val** ord: Ordering***[***T***]*** = *implicitly****[***Ordering***[***T***]]***

*// 如果和当前节点的值相等, 则直接返回 true*

**if *(***ord.equiv***(***ele, value***))* true**

**else if *(***ord.lt***(***ele, value***)) {*** *// 如果小于当前节点, 则去左边查找*

**if *(****left* == **null*)* false** *// 如果左节点是 false, 则直接返回 false. 否则就递归的去左节点查找*

**else** *left*.search***(***ele***)***

***}* else *{****// 如果大于当前节点, 则去右边查找*

**if *(****right* == **null*)* false** *// 如果右节点是 false, 则直接返回 false. 否则就递归的去右节点查找*

**else** *right*.search***(***ele***)***

***}***

***}***

***}***

### 删除节点

排序二叉树的删除节点比较复杂, 需要考虑3种情况:

1. 删除叶子节点
2. 删除只有一棵子树的节点
3. 删除有两棵子树的节点

**package** com.atguigu.datastruc.high.search\_binary\_tree

*/\*\**

*\* Author lzc*

*\* Date 2019-12-02 18:42*

*\*/*

**object** SearchBinaryTreeDemo ***{***

**def** main***(***args: Array***[***String***])***: Unit = ***{***

**val** arr = *Array****(***8, 4 , 9, 10, 1, 6, 7, 12***)***

**val** sbt: SearchBinaryTree***[***Int***]*** = *initSBT****(***arr***)***

*println****(***sbt.delete***(***4***))***

sbt.infixForeach***(****println****)***

*// println(sbt.search(80))*

***}***

***}***

**class** SearchBinaryTree***[***T: Ordering***] {***

*/\*\**

*\* 删除节点*

*\**

*\** **@param ele** *删除是否成功*

*\*/*

**def** delete***(***ele: T***)***: Boolean = ***{***

**if *(****root* == **null*)* false** *// 如果root 是空, 直接返回 false, 表示删除失败*

**else if *(****root*.value == ele***) {*** *// 如果要删除的是根节点*

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

*root* = **null**

***}* else if *(****root*.*left* != **null** && *root*.*right* != **null*) {***

*root*.value = *root*.*right*.deleteMin

***}* else *{***

**val** notNullNode: TreeNode***[***T***]*** = **if *(****root*.*left* != **null*)*** *root*.*left* **else** *root*.*right*

*root* = notNullNode

***}***

**true**

***}***

**else** *root*.delete***(***ele***) // 则删除root的子节点去***

***}***

***}***

**class** TreeNode***[***T: Ordering***](*var** value: T***) {***

*/\*\**

*\* 删除指定的元素*

*\**

*\** **@param ele**

*\** **@return**

*\*/*

**def** delete***(***ele: T***)***: Boolean = ***{***

**val** ord: Ordering***[***T***]*** = *implicitly****[***Ordering***[***T***]]***

*// 如果当前节点是否要删除的元素, 分3 种情况处理*

**if *(***ord.equiv***(***ele, value***)) {***

*// 判断当前节点是父节点的左节点还是右节点, 后面需要让父节点的左节点或者右节点指向新的节点*

**var** isLeft: Boolean = **true** *// 假设该节点是父节点的左节点*

**if *(****p* != **null** && *p*.*right* != **null** && ord.equiv***(****p*.*right*.value, value***)) {*** *// 判断是否为右节点*

isLeft = **false**

***}***

**if *(****left* == **null** && *right* == **null*) {*** *// 当前节点是叶子节点*

**if *(***isLeft***) {*** *// 如果当前节点是父节点的左节点, 则让父节点的左节点指向 null*

*p*.*left* = **null**

***}* else *{***

*p*.*right* = **null**

***}***

*// 左右节点都在. 则把右节点最小值删掉, 然后放在 value 替换掉当前节点的值(或者左节点最大值)*

***}* else if *(****left* != **null** && *right* != **null*) {***

value = *right*.deleteMin

***}* else *{*** *// 只有一个子节点*

*// 找到唯一的一个子节点*

**val** onlyChild: TreeNode***[***T***]*** = **if *(****left* != **null*)*** *left* **else** *right*

**if *(***isLeft***) {***

*p*.*left* = onlyChild *// 父的左子节点指向这些当前节点的唯一节点*

***}* else *{***

*p*.*right* = onlyChild

***}***

***}***

**true**

***}* else if *(***ord.lt***(***ele, value***)) {***

**if *(****left* == **null*)* false**

**else** *left*.delete***(***ele***)***

***}* else *{***

**if *(****right* == **null*)* false**

**else** *right*.delete***(***ele***)***

***}***

***}***

*/\*\**

*\* 删除当前节点的最小子节点*

*\**

*\** **@param node**

*\** **@return**

*\*/*

**def** deleteMin: T = ***{***

**var** minNode: TreeNode***[***T***]*** = **this**

**while *(*this**.*left* != **null*) {*** *//遍历左节点, 找到最小的子节点*

minNode = **this**.*left*

***}***

*// 删除最小子节点*

minNode.delete***(***minNode.value***)***

*// 返回最小子节点的值域*

minNode.value

***}***

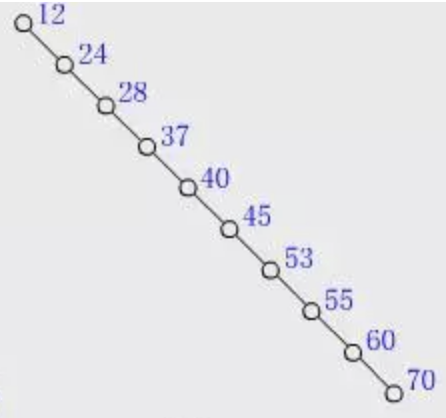
***}***

## 平衡二叉树

### 为什么需要平衡二叉树

在最好的情况下，二叉排序树的查找效率比较高，是，其访问性能近似于折半查找(二分查找)；

但最差时候会是 ，比如插入的元素是有序的，生成的二叉排序树就是一个链表，这种情况下，需要遍历全部元素才行。



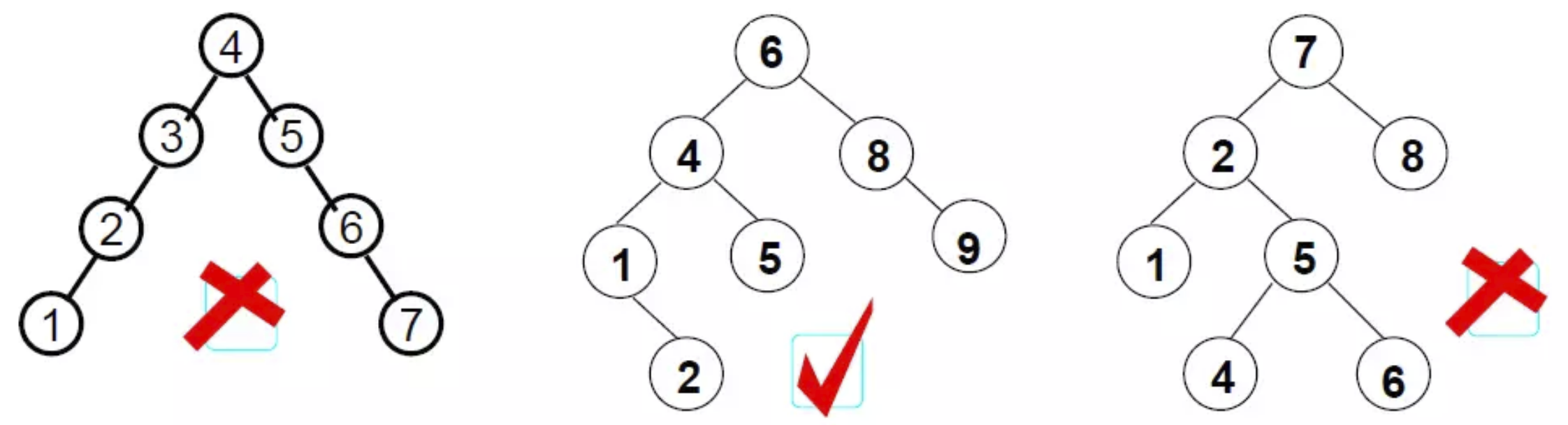
如果我们可以保证二叉排序树不出现上面提到的极端情况（插入的元素是有序的，导致变成一个链表），就可以保证很高的效率了。但这在插入有序的元素时不太好控制，按二叉排序树的定义，我们无法判断当前的树是否需要调整。

因此就要用到平衡二叉树。

### 什么是平衡二叉树

它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树.

平衡二叉树就是让这棵树变的”矮胖”, 而是变的”高瘦”



### 各种平衡树

1. AVL树.

是最早被发明的自平衡二叉查找树。在AVL树中，任一节点对应的两棵子树的最大高度差为1，因此它也被称为高度平衡树。查找、插入和删除在平均和最坏情况下的时间复杂度都是

1. 树堆

是有一个随机附加域满足堆的性质的二叉搜索树，其结构相当于以随机数据插入的二叉搜索树。其基本操作的期望时间复杂度为。相对于其他的平衡二叉搜索树，Treap的特点是实现简单，且能基本实现随机平衡的结构。

1. 伸展树
2. 红黑树

在1972年由鲁道夫·贝尔发明，被称为"对称二叉B树"，它现代的名字源于Leo J. Guibas和Robert Sedgewick于1978年写的一篇论文。红黑树的结构复杂，但它的操作有着良好的最坏情况运行时间，并且在实践中高效：它可以在时间内完成查找，插入和删除，这里的是树中元素的数目。

1. 加权平衡树

加权平衡树的每个结点储存这个结点下子树的大小，可以用来实现顺序统计树操作。优势在于占用空间相对较小

1. 2-3树

其内部节点（存在子节点的节点）要么有2个孩子和1个数据元素，要么有3个孩子和2个数据元素，叶子节点没有孩子，并且有1个或2个数据元素。2–3树和AA树是等距同构的，换句话说，对于每个2–3树，都至少有1个AA树和它的元素排列是相同的。

1. AA树

AA树是红黑树的一种变种，是Arne Andersson教授在1993年年在他的论文"Balanced search trees made simple"中介绍，设计的目的是减少红黑树考虑的不同情况，区别于红黑树的是，AA树的红节点只能作为右叶子。换句话说，没有红节点可以是一个左子儿。这导致代替2-3-4树，从而大大简化了维护2-3树的模拟。

1. 替罪羊树

其平衡基于部分重建，在非平衡的二叉搜索树中，每次操作以后检查操作路径，找到最高的满足左右子树大小大于平衡因子（alpha）乘以自身大小的结点，重建整个子树。这样就得到了替罪羊树，而被重建的子树的原来的根就被称为替罪羊节点。

## 平衡二叉树之AVL树

### AVL树概念

在计算机科学中，AVL树是最早被发明的自平衡二叉查找树。在AVL树中，任一节点对应的两棵子树的最大高度差为1，因此它也被称为高度平衡树。查找、插入和删除在平均和最坏情况下的时间复杂度都是。增加和删除元素的操作则可能需要借由一次或多次树旋转，以实现树的重新平衡。AVL树得名于它的发明者G. M. Adelson-Velsky和Evgenii Landis，他们在1962年的论文《An algorithm for the organization of information》中公开了这一数据结构。

节点的平衡因子是它的左子树的高度减去它的右子树的高度（有时相反）。带有平衡因子1、0或 -1的节点被认为是平衡的。带有平衡因子 -2或2的节点被认为是不平衡的，并需要重新平衡这个树。平衡因子可以直接存储在每个节点中，或从可能存储在节点中的子树高度计算出来。

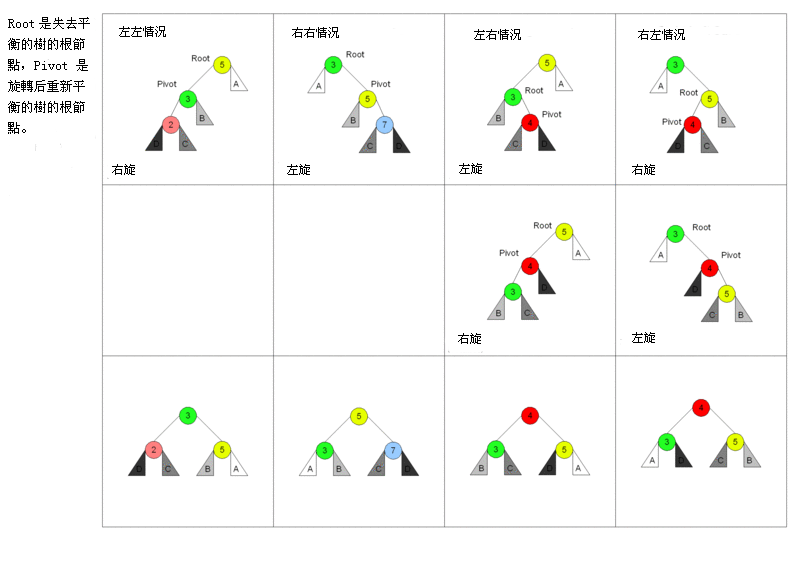
### AVL树操作

AVL二叉树相比前面的二叉树操作基本相同. 但是要进行预先或随后做一次或多次所谓的"AVL旋转"。把失衡的二叉树变成平衡的二叉树

若平衡二叉树种某个结点的左子树和右子树的高度相差大于1，该树就是失衡了，该结点称为失衡点，就要通过旋转来保持二叉树的平衡。

以下图表以四列表示四种情况，每行表示在该种情况下要进行的操作。在左左和右右的情况下，只需要进行一次旋转操作；在左右和右左的情况下，需要进行两次旋转操作。**【高度和深度从0开始，层级从1开始】【左左1-（-1）=2】**

**【左左：失衡节点的左节点的左节点】**



### AVL树相关操作代码

为了保证二叉树的平衡， AVL 树引入了所谓监督机制，就是在树的某一部分的不平衡度超过一个阈值后触发相应的平衡操作。保证树的平衡度在可以接受的范围内。

既然引入了监督机制，我们必然需要一个监督指标，以此来判断是否需要进行平衡操作。这个监督指标被称为“平衡因子（Balance Factor）”。定义如下：

平衡因子： 某个结点的左子树的高度减去右子树的高度得到的差值。

对AVL平衡树来说, 平衡因子的值为1(左高), 0(平衡), -1(左低)

**package** com.atguigu.datastruc.high.avl\_tree

*/\*\**

*\* Author lzc*

*\* Date 2019-12-02 18:42*

*\*/*

**object** AVLTreeDemo ***{***

**def** main***(***args: Array***[***String***])***: Unit = ***{***

*// val arr = Array(4, 3, 6, 5, 7, 8) // (右右)需要左旋*

*// val arr = Array(10, 12, 8, 9, 7, 6) // (左左)需要右旋*

*// val arr: Array[Int] = Array(10, 11, 7, 6, 8, 9) // (左右) 先左旋再右旋*

**val** arr: Array***[***Int***]*** = *Array****(***10, 8, 16, 18, 14, 12***)*** *// (右左) 先右旋再左旋*

**val** avl: AVLTree***[***Int***]*** = *initAVL****(***arr***)***

*println****(*"数的高度:"** + avl.heigth***())***

*println****(*"左子树的高度:"** + avl.*root*.*left*.height***())***

*println****(*"右子树的高度:"** + avl.*root*.*right*.height***())***

*println****(*"根节点:"** + avl.*root*.value***)***

avl.infixForeach***(****println****)***

***}***

*// 初始化一个排序二叉树*

**def** initAVL***(***arr: Array***[***Int***])***: AVLTree***[***Int***]*** = ***{***

**val** avl = **new** AVLTree***[***Int***]()***

arr.foreach***(***x => ***{***

avl.add***(***x***)***

***})***

avl

***}***

***}***

**class** AVLTree***[***T: Ordering***] {***

*// AVL树的根节点*

**var** *root*: AVLTreeNode***[***T***]*** = \_

*// 向AVL树中添加节点*

**def** add***(***ele: T***)***: Unit = ***{***

*// 如果 root 节点为 null, 则把元素置为 root 位置*

**if *(****root* == **null*) {***

*root* = **new** AVLTreeNode***[***T***](***ele***)***

***}* else *{*** *// 如果 root 节点不为空则调用 root 的 add 方法来添加元素*

*root*.add***(***ele***)***

**if *(****root*.*p* != **null*)*** *root* = *root*.*p // 如果数平衡之后, 根节点发生了变化, 则需要把根节点更新*

***}***

***}***

*/\*\**

*\* 查找指定的元素是否存在*

*\**

*\** **@param ele**

*\** **@return**

*\*/*

**def** search***(***ele: T***)***: Boolean = ***{***

**if *(****root* == **null*)* false** *// 如果 root 是空, 直接返回 false*

**else** *root*.search***(***ele***)*** *// 从 root 节点开始查找*

***}***

*/\*\**

*\* 删除节点*

*\**

*\** **@param ele** *删除是否成功*

*\*/*

**def** delete***(***ele: T***)***: Boolean = ***{***

**if *(****root* == **null*)* false** *// 如果root 是空, 直接返回 false, 表示删除失败*

**else if *(****root*.value == ele***) {*** *// 如果要删除的是根节点*

**if *(****root*.*left* == **null** && *root*.*right* == **null*) {*** *// 如果根节点没有左右子节点*

*root* = **null** *// 根节点直接置为空*

***}* else if *(****root*.*left* != **null** && *root*.*right* != **null*) {*** *// 如果左右子节点均在*

*root*.value = *root*.*right*.deleteMin *// 删除右节点的最小子节点, 并把值设置给根节点*

***}* else *{*** *// 如果只有左节点右节点, 则把左或者右设置为 root*

**val** notNullNode: AVLTreeNode***[***T***]*** = **if *(****root*.*left* != **null*)*** *root*.*left* **else** *root*.*right*

*root* = notNullNode

***}***

**true**

***}***

**else** *root*.delete***(***ele***)*** *// 递归的删除查找删除指定的值的节点*

***}***

*/\*\**

*\* 中序遍历*

*\**

*\** **@param op**

*\*/*

**def** infixForeach***(***op: T => Unit***)***: Unit = ***{***

**if *(****root* != **null*)*** *root*.infixForeach***(***op***)***

***}***

*/\*\**

*\* 完整树的高度*

*\**

*\** **@return**

*\*/*

**def** heigth***()***: Int = ***{***

**if *(****root* == **null*)*** -1

**else** *root*.height***()***

***}***

***}***

**class** AVLTreeNode***[***T: Ordering***](*var** value: T***) {***

*// 左子节点*

**var** *left*: AVLTreeNode***[***T***]*** = \_

*// 右子节点*

**var** *right*: AVLTreeNode***[***T***]*** = \_

*// 双亲节点(父节点)*

**var** *p*: AVLTreeNode***[***T***]*** = \_

*/\*\**

*\* 中序遍历*

*\**

*\** **@param op**

*\*/*

**def** infixForeach***(***op: T => Unit***)***: Unit = ***{***

*// 先遍历子左子节点*

**if *(****left* != **null*)*** *left*.infixForeach***(***op***)***

*// 再遍历当前节点*

op***(***value***)***

*// 最后遍历右子节点*

**if *(****right* != **null*)*** *right*.infixForeach***(***op***)***

***}***

*/\*\**

*\* 插入数据*

*\**

*\** **@param ele**

*\*/*

**def** add***(***ele: T***)***: Unit = ***{***

*// 从冥界召唤可以比较 T 类型的比较器(Ordering)*

**val** ord: Ordering***[***T***]*** = *implicitly****[***Ordering***[***T***]]***

**if *(***ord.lteq***(***ele, value***)) {*** *// 如果小于等于当前值, 则在左边添加*

**if *(****left* == **null*) {*** *// 如果左节点是 null, 则左节点置为新节点*

*left* = **new** AVLTreeNode***[***T***](***ele***)***

*left*.*p* = **this**

***}* else *{*** *// 否则递归的在左节点添加节点*

*left*.add***(***ele***)***

***}***

***}* else *{***

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

*right* = **new** AVLTreeNode***[***T***](***ele***)***

*right*.*p* = **this**

*// 对树进行再平衡*

***}* else *{***

*right*.add***(***ele***)***

***}***

***}***

*// 对树进行旋转*

rotate***()***

***}***

*/\*\**

*\* 查找元素*

*\**

*\** **@param ele**

*\** **@return**

*\*/*

**def** search***(***ele: T***)***: Boolean = ***{***

**val** ord: Ordering***[***T***]*** = *implicitly****[***Ordering***[***T***]]***

*// 如果和当前节点的值相等, 则直接返回 true*

**if *(***ord.equiv***(***ele, value***))* true**

**else if *(***ord.lt***(***ele, value***)) {*** *// 如果小于当前节点, 则去左边查找*

**if *(****left* == **null*)* false** *// 如果左节点是 false, 则直接返回 false. 否则就递归的去左节点查找*

**else** *left*.search***(***ele***)***

***}* else *{***

*// 如果大于当前节点, 则去右边查找*

**if *(****right* == **null*)* false** *// 如果右节点是 false, 则直接返回 false. 否则就递归的去右节点查找*

**else** *right*.search***(***ele***)***

***}***

***}***

*/\*\**

*\* 删除指定的元素*

*\**

*\** **@param ele**

*\** **@return** *删除是否成功*

*\*/*

**def** delete***(***ele: T***)***: Boolean = ***{***

**val** ord: Ordering***[***T***]*** = *implicitly****[***Ordering***[***T***]]***

*// 如果当前节点是否要删除的元素, 分3 种情况处理*

**if *(***ord.equiv***(***ele, value***)) {***

*// 判断当前节点是父节点的左节点还是右节点, 后面需要让父节点的左节点或者右节点指向新的节点*

**var** isLeft: Boolean = **true** *// 假设该节点是父节点的左节点*

**if *(****p* != **null** && *p*.*right* != **null** && ord.equiv***(****p*.*right*.value, value***)) {*** *// 判断是否为右节点*

isLeft = **false**

***}***

**if *(****left* == **null** && *right* == **null*) {*** *// 1. 当前节点是叶子节点*

**if *(***isLeft***) {*** *// 如果当前节点是父节点的左节点, 则让父节点的左节点指向 null*

*p*.*left* = **null**

***}* else *{***

*p*.*right* = **null**

***}***

*// 2. 左右节点都在. 则把右节点最小值删掉, 然后放在 value 替换掉当前节点的值(或者左节点最大值)*

***}* else if *(****left* != **null** && *right* != **null*) {***

value = *right*.deleteMin

***}* else *{*** *// 3. 只有一个子节点 找到唯一的一个子节点*

**val** onlyChild: AVLTreeNode***[***T***]*** = **if *(****left* != **null*)*** *left* **else** *right*

**if *(***isLeft***) {***

*p*.*left* = onlyChild *// 父的左子节点指向这些当前节点的唯一节点*

***}* else *{***

*p*.*right* = onlyChild

***}***

***}***

**true**

***}* else if *(***ord.lt***(***ele, value***)) {***

**if *(****left* == **null*)* false**

**else** *left*.delete***(***ele***)***

***}* else *{***

**if *(****right* == **null*)* false**

**else** *right*.delete***(***ele***)***

***}***

***}***

*/\*\**

*\* 删除当前节点的最小子节点*

*\**

*\** **@param node**

*\** **@return**

*\*/*

**def** deleteMin: T = ***{***

**var** minNode: AVLTreeNode***[***T***]*** = **this**

**while *(*this**.*left* != **null*) {*** *//遍历左节点, 找到最小的子节点*

minNode = **this**.*left*

***}***

*// 删除最小子节点*

minNode.delete***(***minNode.value***)***

*// 返回最小子节点的值域*

minNode.value

***}***

*/\*\**

*\* 返回数的高度*

*\**

*\** **@return**

*\*/*

**def** height***()***: Int = ***{***

*// 使用递归的方式计算左树的高度*

**val** leftHeight: Int = **if *(****left* != **null*)*** *left*.height***()* else** -1

*// 使用递归的方式计算右树的高度*

**val** rightHeight: Int = **if *(****right* != **null*)*** *right*.height***()* else** -1

leftHeight.max***(***rightHeight***)*** + 1

***}***

*/\*\**

*\* 计算左子树的高度*

*\**

*\** **@return**

*\*/*

**def** leftHeight***()***: Int = ***{***

**if *(****left* == **null*)*** -1

**else** *left*.height***()***

***}***

*/\*\**

*\* 计算右子树的高度*

*\**

*\** **@return**

*\*/*

**def** rightHeight***()***: Int = ***{***

**if *(****right* == **null*)*** -1

**else** *right*.height***()***

***}***

*/\*\**

*\* 对树进行旋转*

*\*/*

**def** rotate***() {***

*// 右右*

**if *(*this**.leftHeight***()*** - **this**.rightHeight***()*** < -1 && **this**.*right*.leftHeight***()*** - **this**.*right*.rightHeight***()*** < 0***) {*** *// 需要左旋(右边高)*

*println****(*"右右"*)***

leftRotate***()***

**return**

***}***

*// 左左*

**if *(*this**.leftHeight***()*** - **this**.rightHeight***()*** > 1 && **this**.*left*.leftHeight***()*** - **this**.*left*.rightHeight***()*** > 0***) {***

*println****(*"左左"*)***

rightRotate***()***

**return**

***}***

*// 左右*

**if *(*this**.leftHeight***()*** - **this**.rightHeight***()*** > 1 && **this**.*left*.leftHeight***()*** - **this**.*left*.rightHeight***()*** < 0***) {***

*println****(*"左右"*)***

leftRightRotate***()***

**return**

***}***

*// 右左*

**if *(*this**.leftHeight***()*** - **this**.rightHeight***()*** < -1 && **this**.*right*.leftHeight***()*** - **this**.*right*.rightHeight***()*** > 0***) {***

*println****(*"右左"*)***

rightLeftRotate***()***

**return**

***}***

***}***

*// 1. 左旋*

**def** leftRotate***() {***

**val** tmpRight: AVLTreeNode***[***T***]*** = *right // 最初的右节点*

**val** tmpRithtLeft: AVLTreeNode***[***T***]*** = *right*.*left // 最初的右节点的左节点*

**val** tmpP: AVLTreeNode***[***T***]*** = *p // 最初的父节点*

*// 1. 当前节点的右子节点成为当前节点的父节点的(左/右)子节点*

**if *(****p* != **null** && *p*.*right* == **this*) {***

*p*.*right* = *right*

***}* else if *(****p* != **null*) {***

*p*.*left* = *right*

***}***

*// 2. 当前节点的右子节点指向当前节点右节点的左节点*

*right* = *right*.*left*

*// 3. 当前节点右节点的左节点指向当前节点*

tmpRight.*left* = **this**

*// 4. 更新父节点*

tmpRight.*p* = tmpP *// 当前节点的最初左节点指向当前节点的父节点*

*p* = tmpRight *// 当前节点的父节点指向最初的右节点*

**if *(***tmpRithtLeft != **null*)*** tmpRithtLeft.*p* = **this** *// 原来节点的左节点的右节点的父节点指向当前节点*

***}***

*// 2. 右旋*

**def** rightRotate***() {***

**val** tmpLeft: AVLTreeNode***[***T***]*** = *left*

**val** tmpLeftRight: AVLTreeNode***[***T***]*** = *left*.*right*

**val** tmpP: AVLTreeNode***[***T***]*** = *p*

*// 1.*

**if *(****p* != **null** && *p*.*right* == **this*) {***

*p*.*right* = *left*

***}* else if *(****p* != **null*) {***

*p*.*left* = *left*

***}***

*// 2.*

*left* = *left*.*right*

*// 3.*

tmpLeft.*right* = **this**

*// 4.*

tmpLeft.*p* = tmpP

*p* = tmpLeft

**if *(***tmpLeftRight != **null*)*** tmpLeftRight.*p* = **this**

***}***

*// 3. 先左旋再右旋*

**def** leftRightRotate***() {***

**this**.*left*.leftRotate***() // 当前节点的左子节点左旋***

rightRotate***() // 当前节点右旋***

***}***

*// 4. 先右旋再左旋*

**def** rightLeftRotate***() {***

**this**.*right*.rightRotate***() // 当前节点的右子节点右旋***

leftRotate***() // 当前节点左旋***

***}***

***}***