树形结构

前端精进（科班方向）

版权声明

本内容版权属杭州饥人谷教育科技有限公司（简称饥人谷）所有。

任何媒体、网站或个人未经本网协议授权不得转载、链接、转贴，
或以其他方式复制、发布和发表。

已获得饥人谷授权的媒体、网站或个人在使用时须注明「资料来源：饥人谷」。

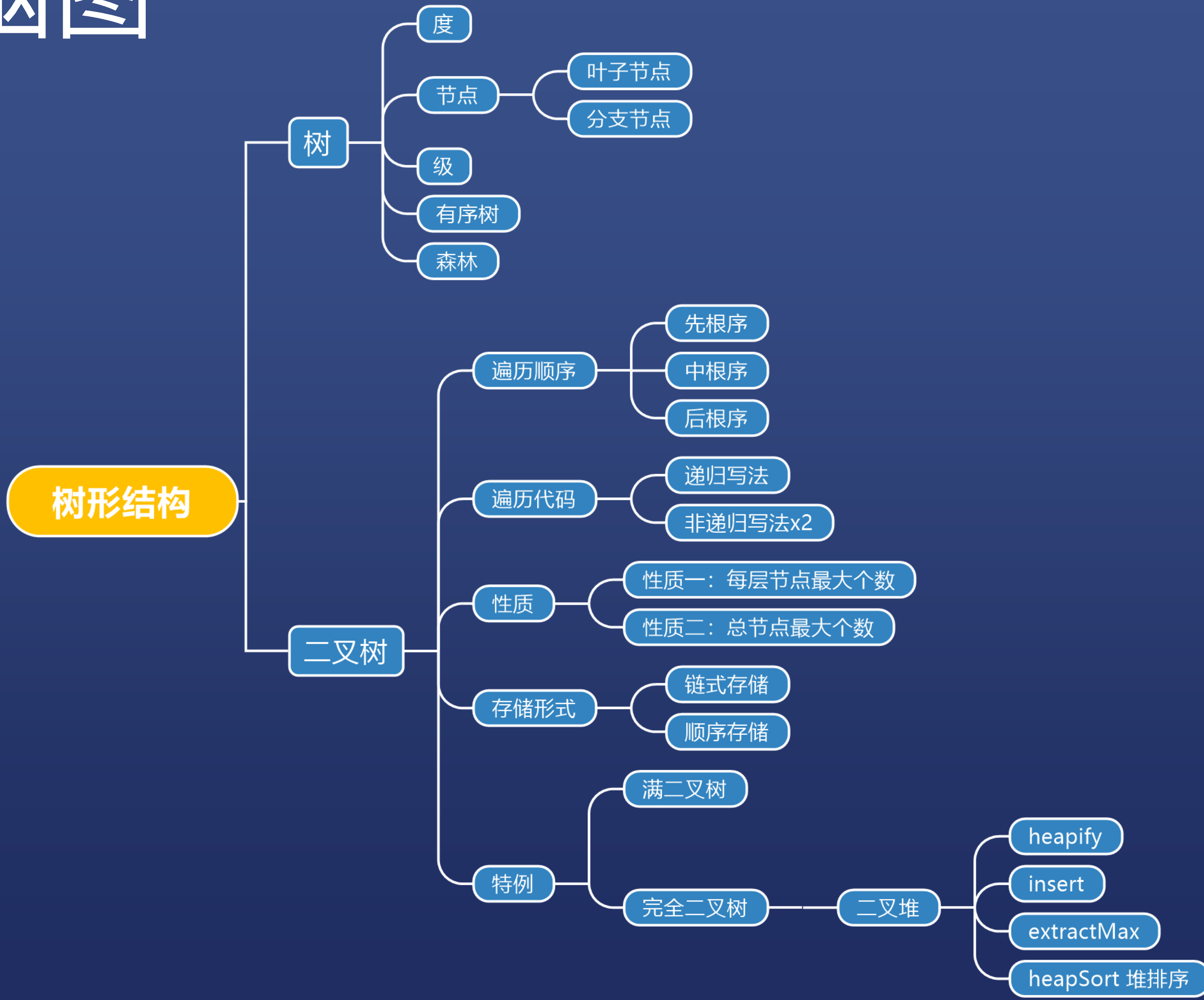
对于违反者，饥人谷将依法追究责任。

联系方式

如果你想要购买本课程
请微信联系 xiedaimala02 或 xiedaimala03

如果你发现有人盗用本课程
请微信联系 xiedaimala02 或 xiedaimala03

本课脑图


树形结构
树
度
节点
叶子节点
分支节点
级
有序树
森林
二叉树
遍历顺序
先根序
中根序
后根序
遍历代码
递归写法
非递归写法x2
性质
性质一：每层节点最大个数
性质二：总节点最大个数
存储形式
链式存储
顺序存储
特例
满二叉树
完全二叉树
二叉堆
heapify
insert
extractMax
heapSort 堆排序

树

- 树的形式化定义
  - ✓ 树是一个或多个节点的集合 T
  - ✓ T 必须有一个根节点 root
  - ✓ 除了 root 之外，T 剩余的节点被分划为 m≧0 个不相交的集合 T1、T2 …… Tm，这些集合也是树
- 分析
  - ✓ 树中最少有一个根节点
  - ✓ 树是由根和子树组成的，子树个数可以为 0

* 参考《计算机程序设计艺术（第一卷）》2.3 节

树相关概念

- **概念**
  - ✓ 度 - 节点的子树的个数
  - ✓ 叶 - 度为 0 的节点，也叫端节点
  - ✓ 分支节点 - 非叶的节点
  - ✓ 级 - 根节点的级为 0，任何其他节点的级比它爸爸大1
  - ✓ 级 - 另一种解释是节点到根节点的距离
  - ✓ 有序树 - 认为 A(B,C) 和 A(C,B) 是不同的树
  - ✓ 森林 - m≧0 个不相交的树的集合
- **推论**
  - ✓ 把一棵树的根删掉，就得到了一个森林

二叉树

另一种树形结构

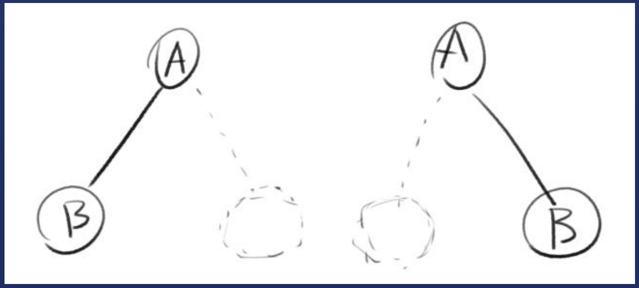
* 参考《计算机程序设计艺术（第一卷）》2.3.1 节

二叉树

- 形式化定义二叉树
  - ✓ 二叉树是 m≧0 个节点的集合 T
  - ✓ T 要么为空，要么满足如下规则
  - ✓ T 由一个根节点或两个二叉树组成
- 分析
  - ✓ 每个节点必有 2 个子节点，子节点可以是空节点

二叉树 V.S. 树

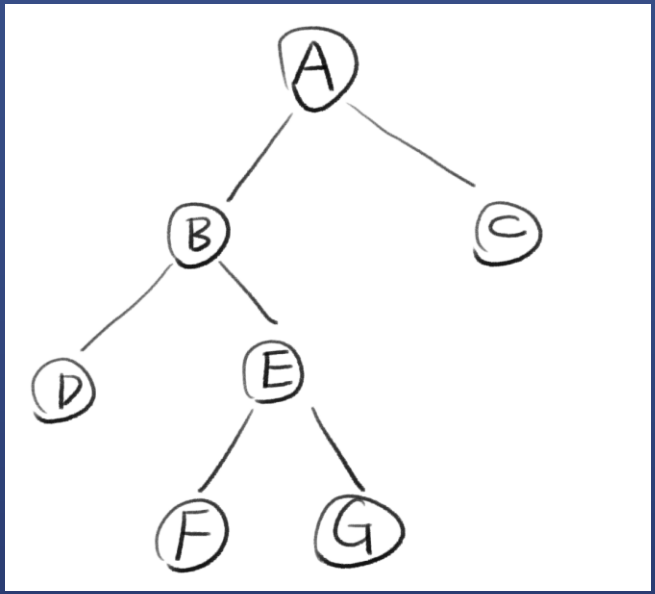
- 区别
  - ✓ 二叉树可以为空，树不能
  - ✓ 二叉树每个节点的度不大于 2，树则没有限制
  - ✓ 二叉树是有序的，树则分为有序和无序两种
- 二叉树 V.S 度最大为2的有序树
  - ✓ 两者也是不一样的
  - ✓ 在有序树中，下面两棵树是一样的
  - ✓ 但在二叉树中，下面两棵二叉树是不一样的，因为对于二叉树来说，空节点也是节点



二叉树、树是两种数据结构

都是树形的而已

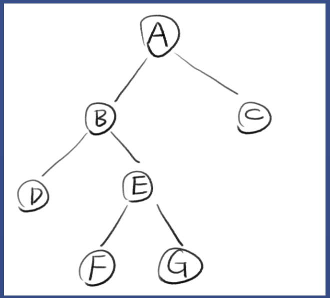
如何表示树结构

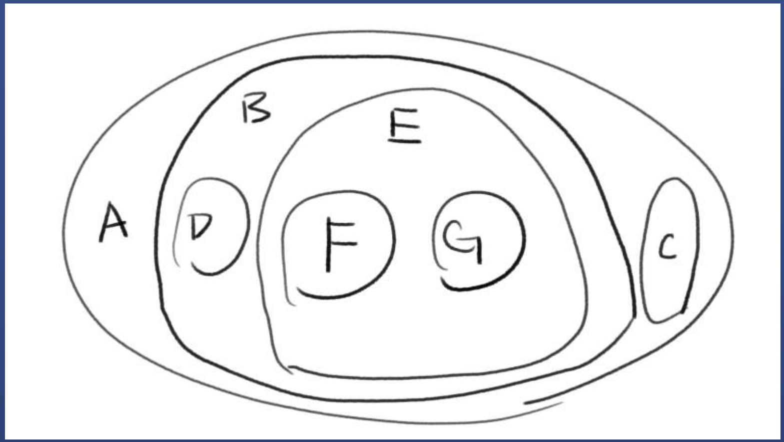


用图

用身体姿势

如何表示树结构 - 续





用另一种图

(A(B)(C))
(A(B(D)(E(F)(G)))(C))

用括号

1 A
1.1 B
1.1.1 D
1.1.2 E
1.1.2.1 F
1.1.2.2 G
1.2 C

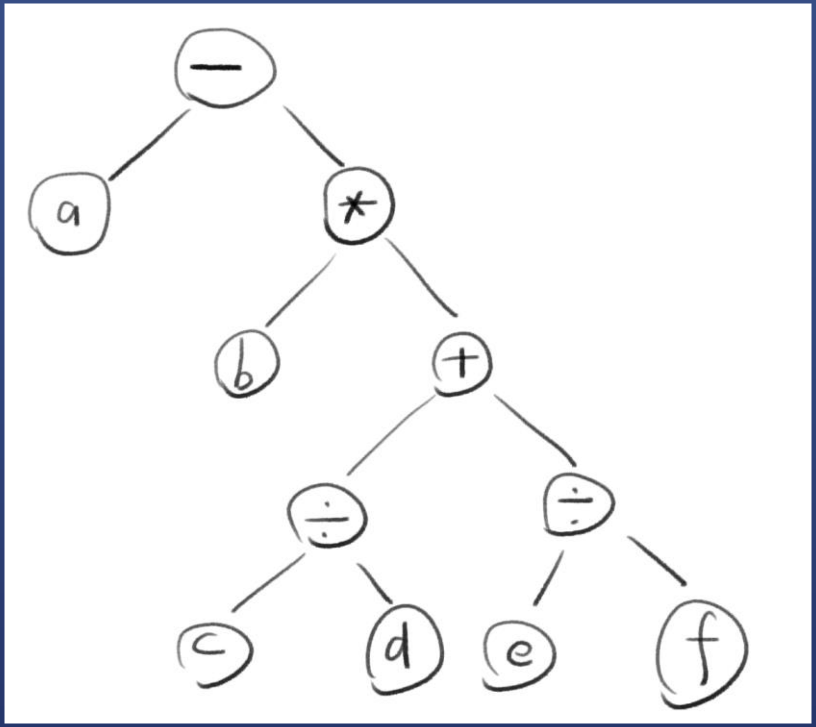
用图书目录

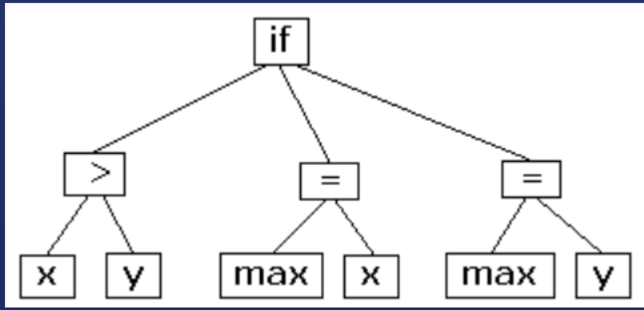
用代码表示二叉树

```
const node1 = {
  value: 12, left: node2, right: node3
}
const binaryTree1 = {
  value: 'A',
  left: {
    value: 'B',
    left: {value: 'C', left:null, right:null},
    right: null
  },
  right: {
    value: 'D',
    left: {value: 'E', left:null, right:null},
    right: {value: 'F', left:null, right:null},
  }
}
// 可以看出一个节点和一棵二叉树树的结构是一样的
// 注意：也有一些书籍为了编码方便，设置了一个多余的头节点
```

树能表示什么？

- 公司结构
- 公式
  - ✓ a - b * (c / d + e / f)
  - ✓ 见右图
- 代码
  - ✓ AST
  - ✓ 抽象语法树





面试题：扁平结构变成树

```
array = [
  {id: 1, name:'CEO', parent: null},
  {id: 2, name:'运营部', parent: 1},
  {id: 3, name:'财务部', parent: 1},
  {id: 4, name:'人事部', parent: 1},
  {id: 5, name:'技术部', parent: 1},
  {id: 6, name:'产品部', parent: 1},
  {id: 7, name:'后端开发部门', parent: 5},
  {id: 8, name:'前端开发部门', parent: 5},
  {id: 9, name:'前端基础设施组', parent: 8},
  {id: 10, name:'前端业务组', parent: 8},
]
// 请改成树形结构，子节点用 children 数组表示
// 注意即使打乱数组的顺序，你的程序也能正常工作
```

解答

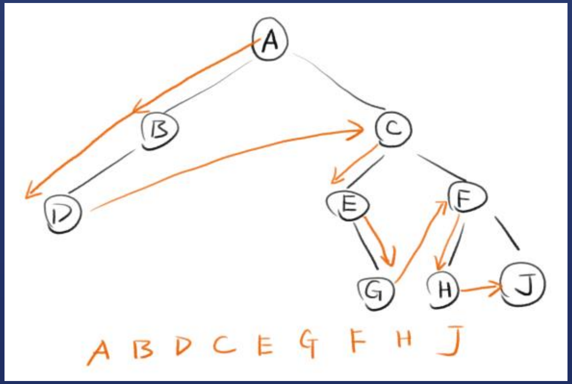
- 参考答案
  - ✓ https://codesandbox.io/s/jovial-lalande-vhmy7
  - ✓ 由于这题不是本课重点，而且比较简单，所以大家自行看答案
  - ✓ 你自己想出答案更好

遍历二叉树

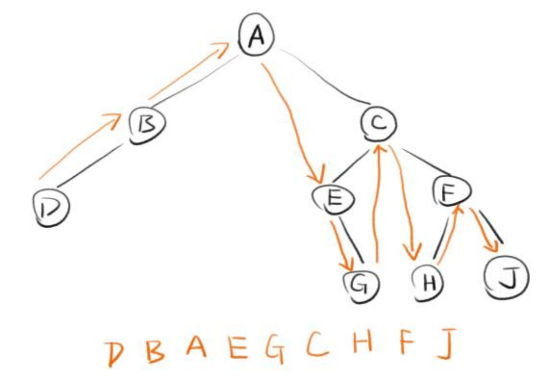
把每个节点按指定顺序打印出来

如何遍历二叉树

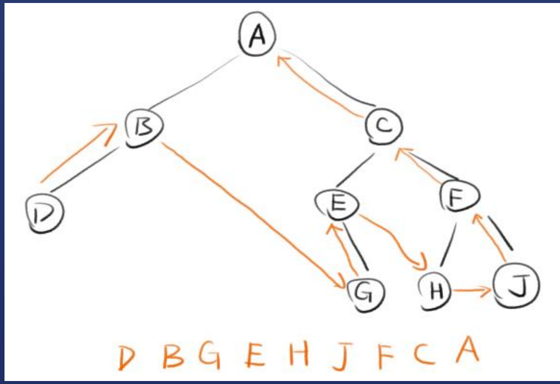
- 三种方法
  - ✓ 先根序：先根，再左，再右
  - ✓ 中根序：先左，再根，再右
  - ✓ 后根序：先左，再右，再根
  - ✓ 代码示例



先根序



中根序



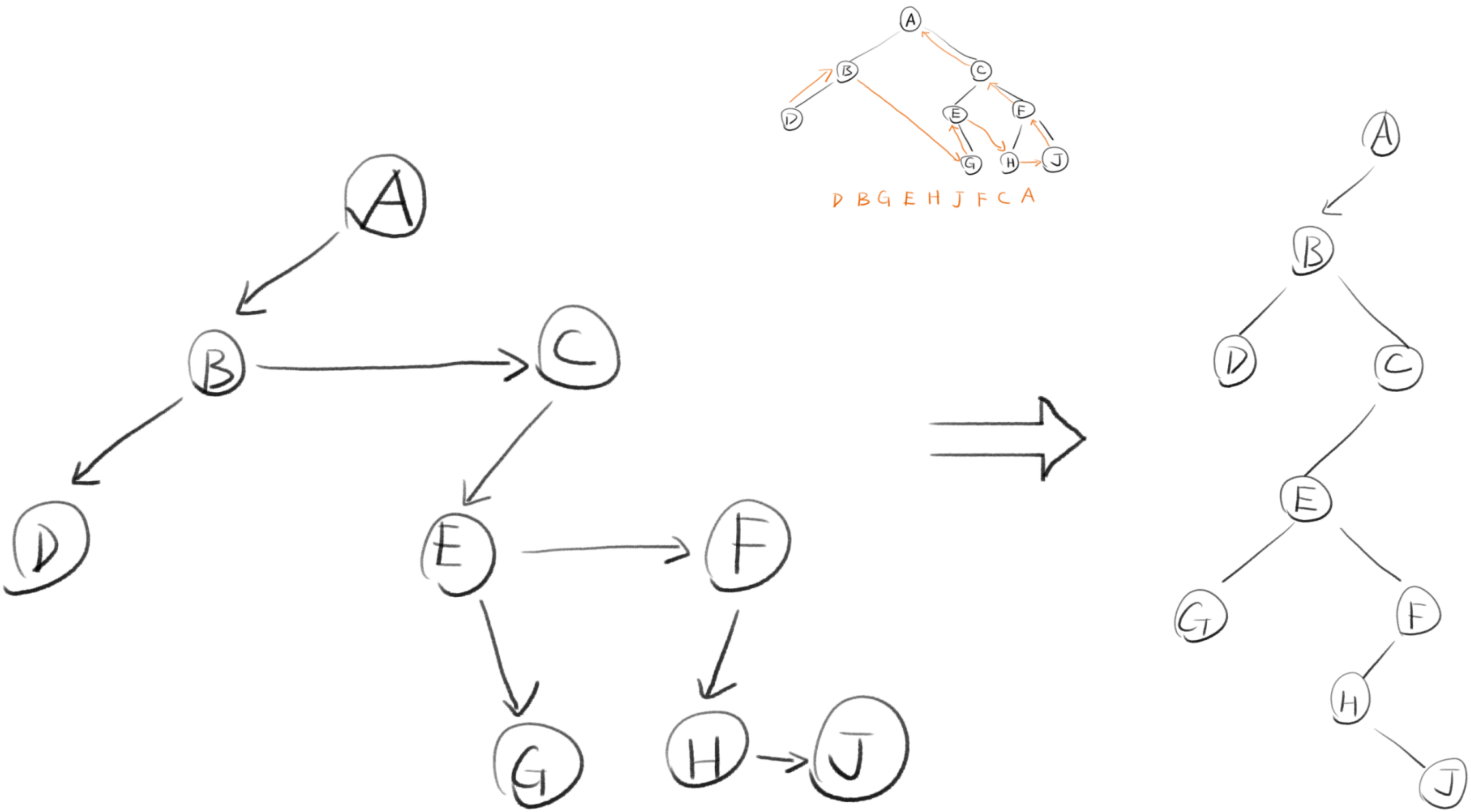
后根序

不用递归，怎么遍历

- 中根序遍历（头条二面题目）
  - ✓ 使用 goto 的写法，但 JS 没有 goto，就模拟了一个
  - ✓ 没有使用 goto 的写法
  - ✓ 思路就是先一路往左，然后打印最左一个节点并往右
- 先根序遍历
  - ✓ 使用了 Goto 的写法，只改动了一行
  - ✓ 没有使用 goto 的写法
- 后根序遍历
  - ✓ 比较复杂，可以用两个 stack，有兴趣可以挑战一下
  - ✓ 一种思路是把树进行转换（见下页），后根转成中根

树的转换

转换规则：每个节点指向第一个儿子和自己的弟弟



中根序：D B G E H J C A

二叉树的性质

有哪些特点

性质1：每层节点数

- 第i层（层数从0开始）

✓ 二叉树的第 i⩾0 层最多有 2^i 个节点

- 证明

✓ 可用数学归纳法证明

✓ 当 i=0 时，第 0 层最多有 2^0 = 1 个节点

✓ 假设 i=k 时结论成立，2^k 个节点每个节点最多有两个子节点，那么 k+1 层最多有 2^(k+1) 个节点，结论也成立

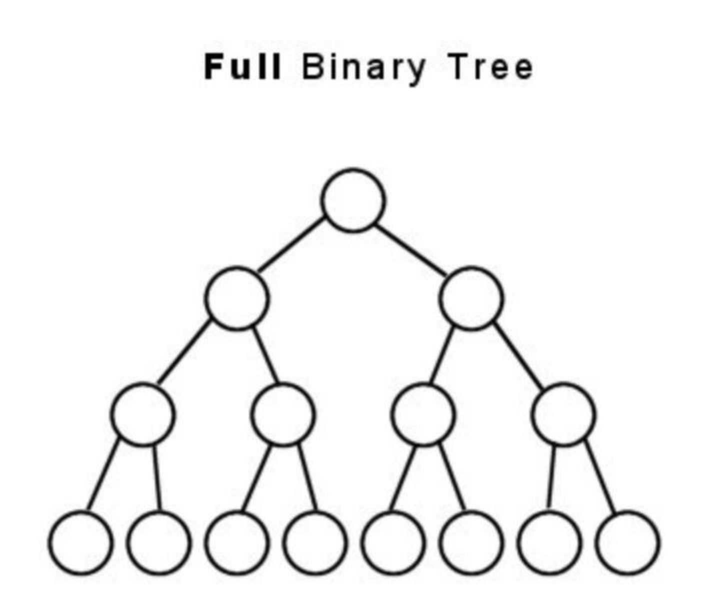
✓ 故，结论对 i⩾0 成立

性质2：总节点数

- 共 h≥1 层
  - ✓ 高度为 h 的二叉树至多有 2^h - 1 个节点
- 证明
  - ✓ 2^0 + 2^1 + 2^2 + ... + 2^(h-1) = 2^h - 1
  - ✓ 每层都是最多，总节点数就是最多了

满二叉树

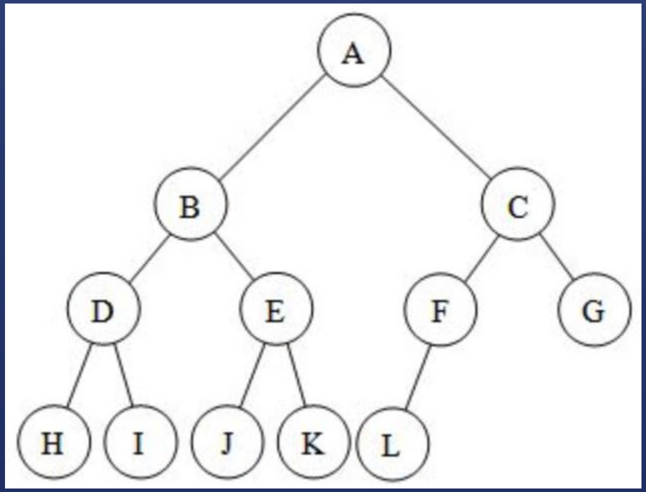
- 如果二叉树
  - ✓ 每层都是满的
  - ✓ 就叫做满二叉树



- 特点
  - ✓ 高度为 h = 4
  - ✓ 每层的节点数分别为 1、2、4、8……
  - ✓ 总节点数为 2^h - 1 = 16 - 1 = 15

完全二叉树（不满）

- 如果二叉树
  - ✓ 只有最后一层不满
  - ✓ 所有节点都尽量往左边靠
  - ✓ 那么就是完全二叉树
  - ✓ 完全二叉树是差一点点就满了（满二叉树）



完全二叉树的性质

- 高度
  - ✓ 如果完全二叉树的节点总数为 n
  - ✓ 则高度 h = log2(n)+1 取整
- 举例
  - ✓ 节点总数为1，则 h = 1
  - ✓ 节点总数为2，则 h = 2
  - ✓ 节点总数为3，则 h = 2
  - ✓ 节点总数为4~7，则 h = 3
  - ✓ 节点总数为8~15，则 h = 4
  - ✓ 当然我们可以用数学归纳法证明之

树形结构的存储形式

链式存储和顺序存储

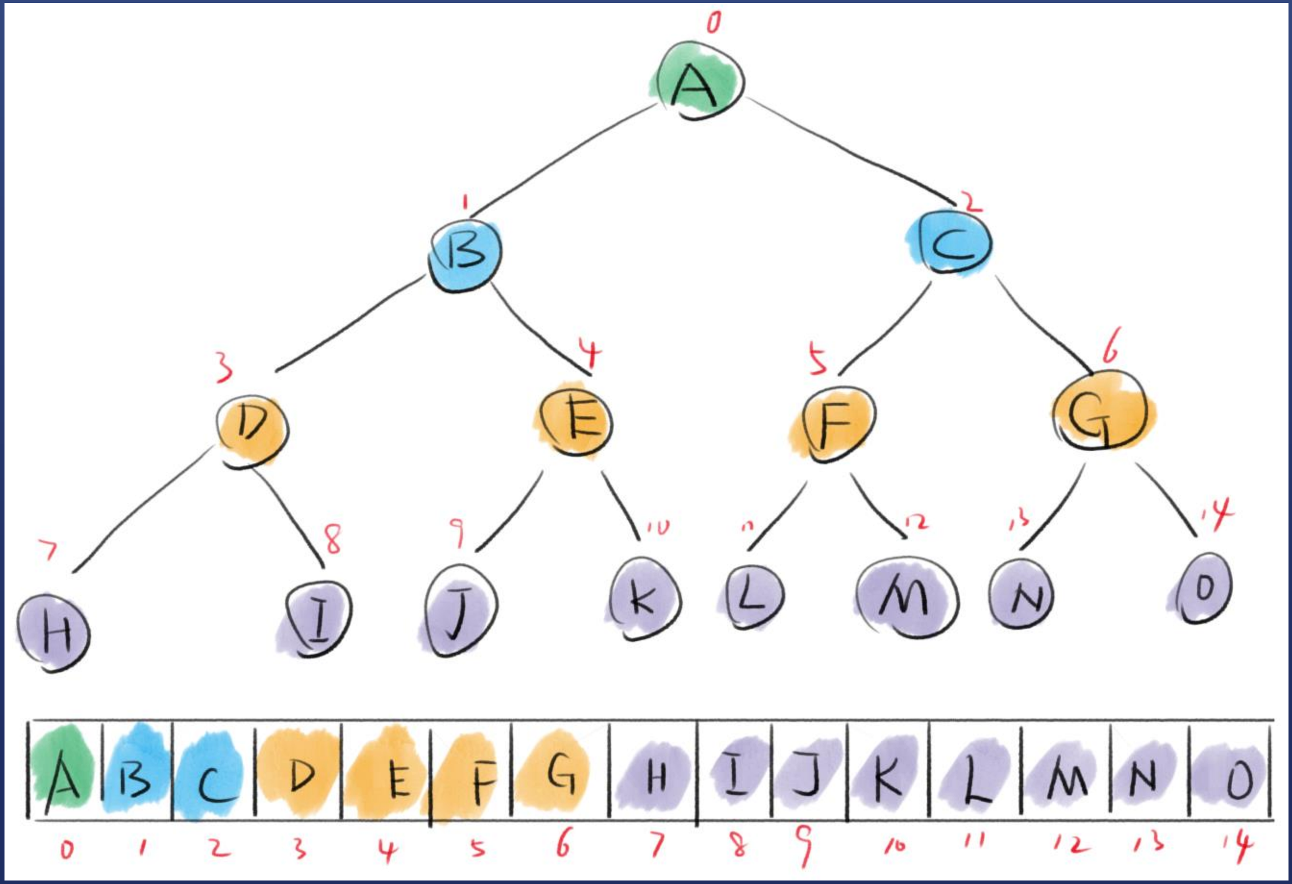
链式存储

- 二叉树如何存储
  - ✓ 每个节点保存三个数据：value, left(地址), right(地址)
  - ✓ 如果有需要，可以多保存一个 parent(父节点地址)
- 树如何存储
  - ✓ 也许你会想到每个节点保存 value 和 children
  - ✓ 但是 children 的长度确定呢?
  - ✓ 如果 children 的长度不确定，你就要写额外的代码
  - ✓ 比如 node.children = [ ]，这是一个动态数组
  - ✓ 当数组的长度不够时，你就要把数组在内存中挪动
  - ✓ 更经济的方法是把树转化为二叉树存储

顺序存储

- 满二叉树可以顺序存储

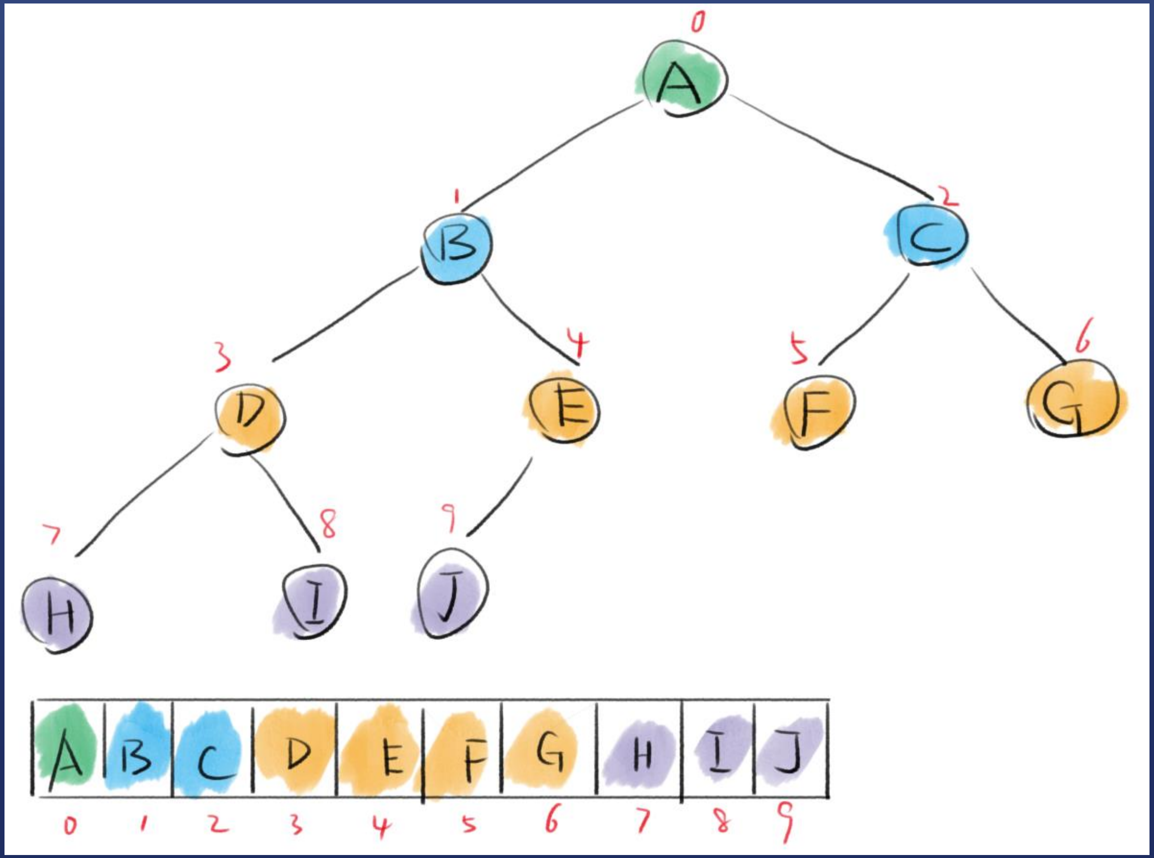
```
fullBinTree = ['A', 'B', 'C', 'D', 'E', 'F',
'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O']
```



顺序存储

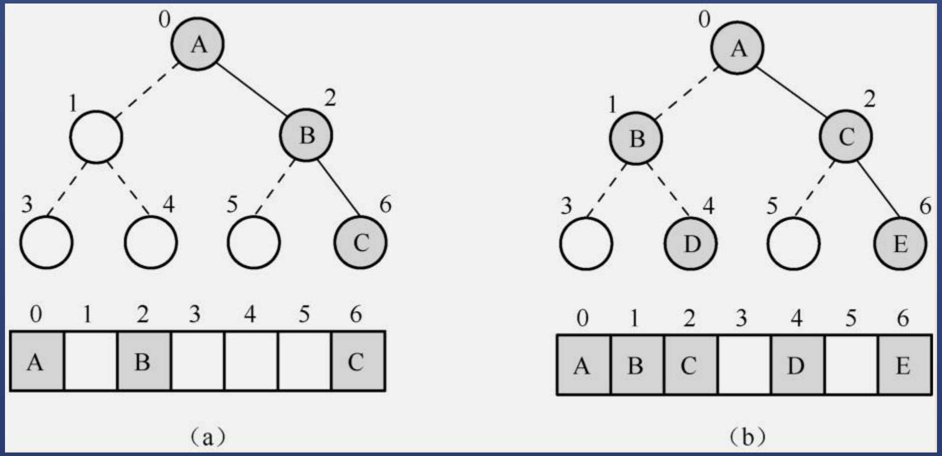
- 完全二叉树可以顺序存储

```
completeBinTree = ['A', 'B', 'C', 'D', 'E',
'F', 'G', 'H', 'I', 'J']
```



顺序存储

- 普通二叉树这样存很浪费



✓ 一般使用其他方式的顺序存储

顺序存储的完全二叉树

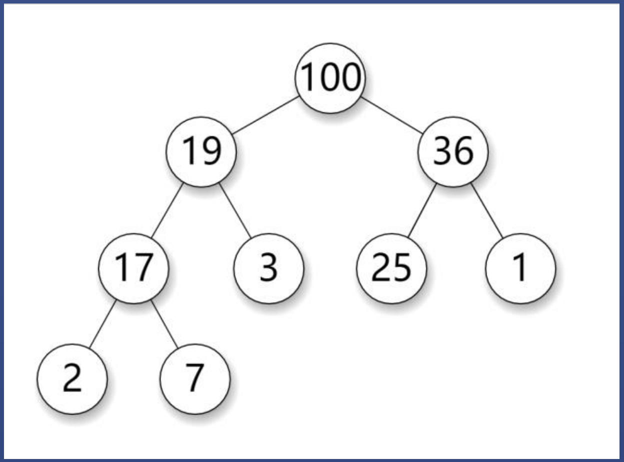
接下来我们主要研究这个玩意

顺序存储的完全二叉树

- 获取下标为i的节点的子节点
  - ✓ 看之前的图，观察规律可知
  - ✓ 子节点为 2i+1 和 2i+2
- 获取下标为i的节点的父节点下标
  - ✓ 看之前的图，观察规律可知
  - ✓ 结果为 (i-1)/2 取整（i=0除外）
- 获取i的兄弟节点
  - ✓ 若 i 为 0，则为根，没有兄弟
  - ✓ 若 i 为单数，则 i 为左节点，i + 1 是它弟弟
  - ✓ 若 i 为双数，则 i 为右节点，i - 1 是它哥哥

中根序遍历顺序存储的完全二叉树

综合一下目前的知识，代码链接



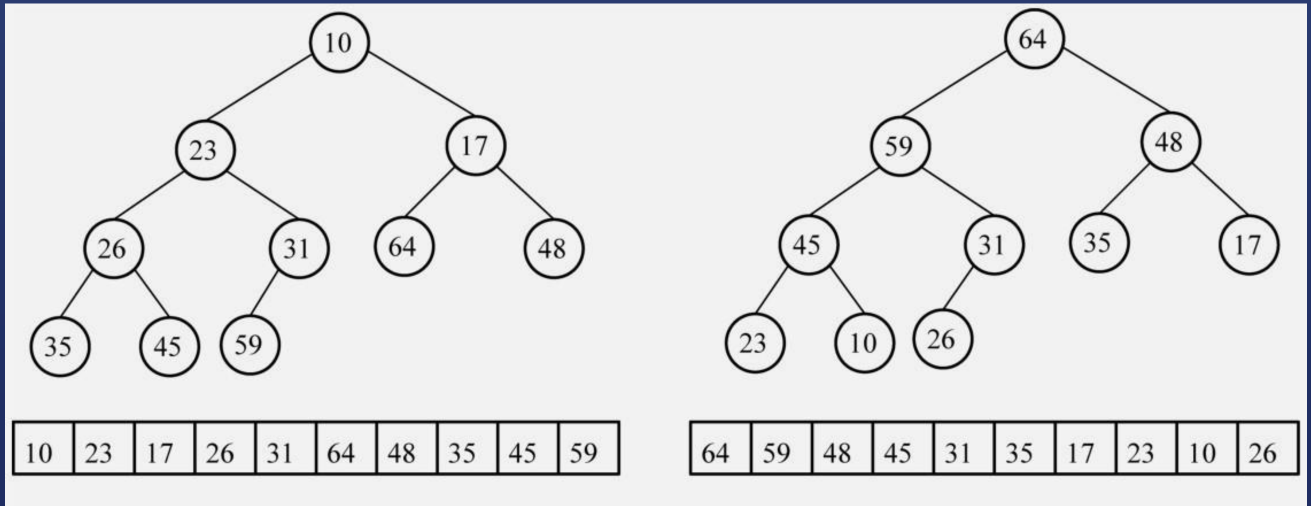
二叉堆，简称堆 Heap

亦yǎn的，或者尖的，完全二叉树

*也有三叉堆以及普通堆，但大部分时候堆就是指二叉堆

二叉堆的定义

- 二叉堆
  - ✓ 一颗完全二叉树
  - ✓ 父节点的值≥子节点的值，则称为最大堆
  - ✓ 父节点的值≤子节点的值，则称为最小堆
  - ✓ 注意：并没有要求左右节点的大小顺序
- 举例
  - ✓ 35,26,48,10,59,64,17,23,45,31



（a）最小堆 （b）最大堆

最大堆的性质

- 堆序性 heap order
  - ✓ 任意节点≥它的所有后代，最大值在堆的根上
- 完全树
  - ✓ 只有最底层不满，且节点尽可能地往左靠

堆的 API

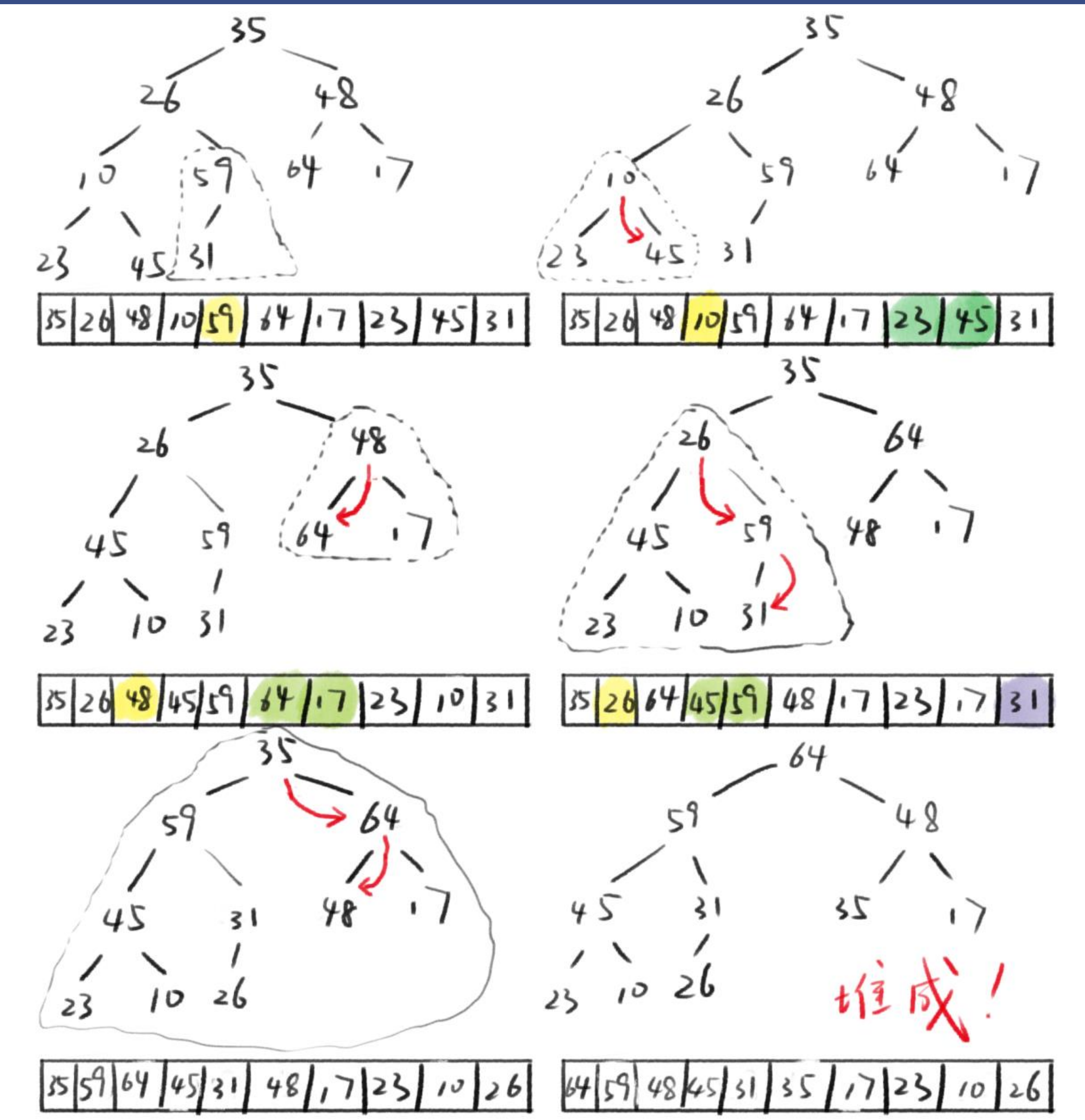
操作	描述	时间复杂度
堆化 heapify	让二叉树(数组)变成堆	O(n * log n)
插入节点 insert(heap, node)	向 heap 中插入新节点 node	O(log n)
弹出最大值 extract_max(heap)	删除并返回最大堆的堆顶节点	O(log n)

接下来研究这三个 API

API - heapify
如何把完全二叉树变成堆

完全二叉树可以用数组存储

思路(siftDown)



从最后一个节点开始
逐个向前
把每个节点与其后代比较
最大的放在上面

注意一个节点有可能需要调整多次（递归）

由于每次调整都是把数字放下降，所以叫做 siftDown

为何没有给出数学思维？

因为我们已经为了存储效率把存储结构变成了数组，现在用数学依然很繁琐，无法做到简洁

问答

- 为什么要从后往前
  - ✓ 为了从易到难
- 为什么从 59 开始
  - ✓ 因为所有叶子节点都可以跳过
- 什么时候递归
  - ✓ 调整父子之后，子节点所在的子树要再调整一次

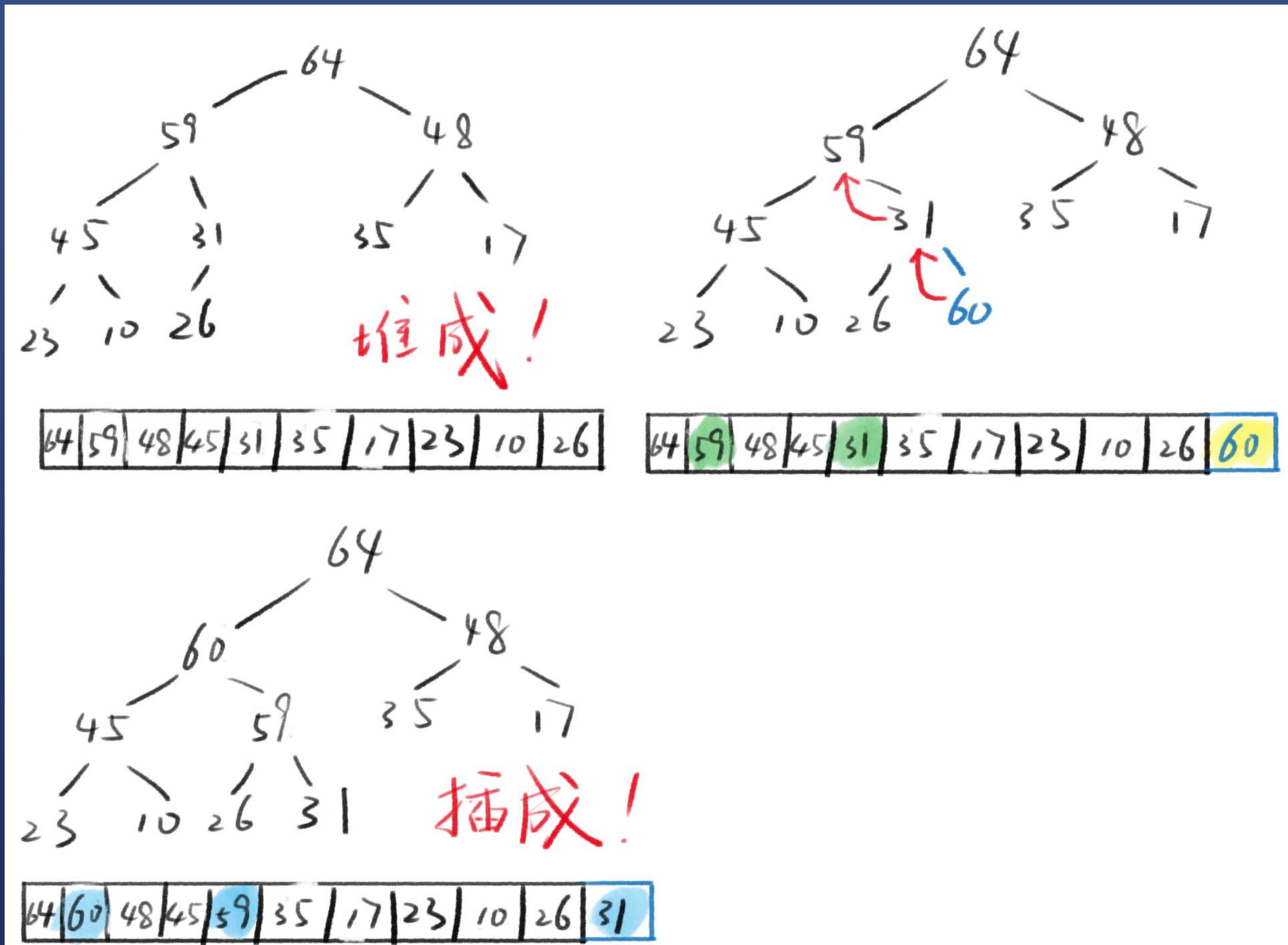
```
array = [35,26,48,10,59,64,17,23,45,31]
heapify = (array) => {
  for(let i=parseInt((array.length-1)/2); i>=0; i--){
    siftDown(array, i, array.length)
  }
  return array
}
siftDown = (heap, i, length) =>{
  const left = 2*i+1, right = 2*i+2
  let greater = left
  if(greater >= length){return}
  if(right < length && heap[right]>heap[greater]){
    greater = right
  }
  if(heap[greater]>heap[i]){
    console.log(`换 ${heap[greater]} ${heap[i]}`);
    [heap[greater],heap[i]] = [heap[i],heap[greater]]
    siftDown(heap, greater, length)
  }
}

heapify(array)
// [64, 59, 48, 45, 31, 35, 17, 23, 10, 26]
```

API - insert(heap, item)
如何向堆中插入一个值

要保证插入之后，依然得到一个堆

思路(siftUp)



堆成!
64 59 48 45 31 35 17 23 10 26
64 59 48 45 31 35 17 23 10 26 60
插成!
64 60 48 45 59 35 17 23 10 26 31

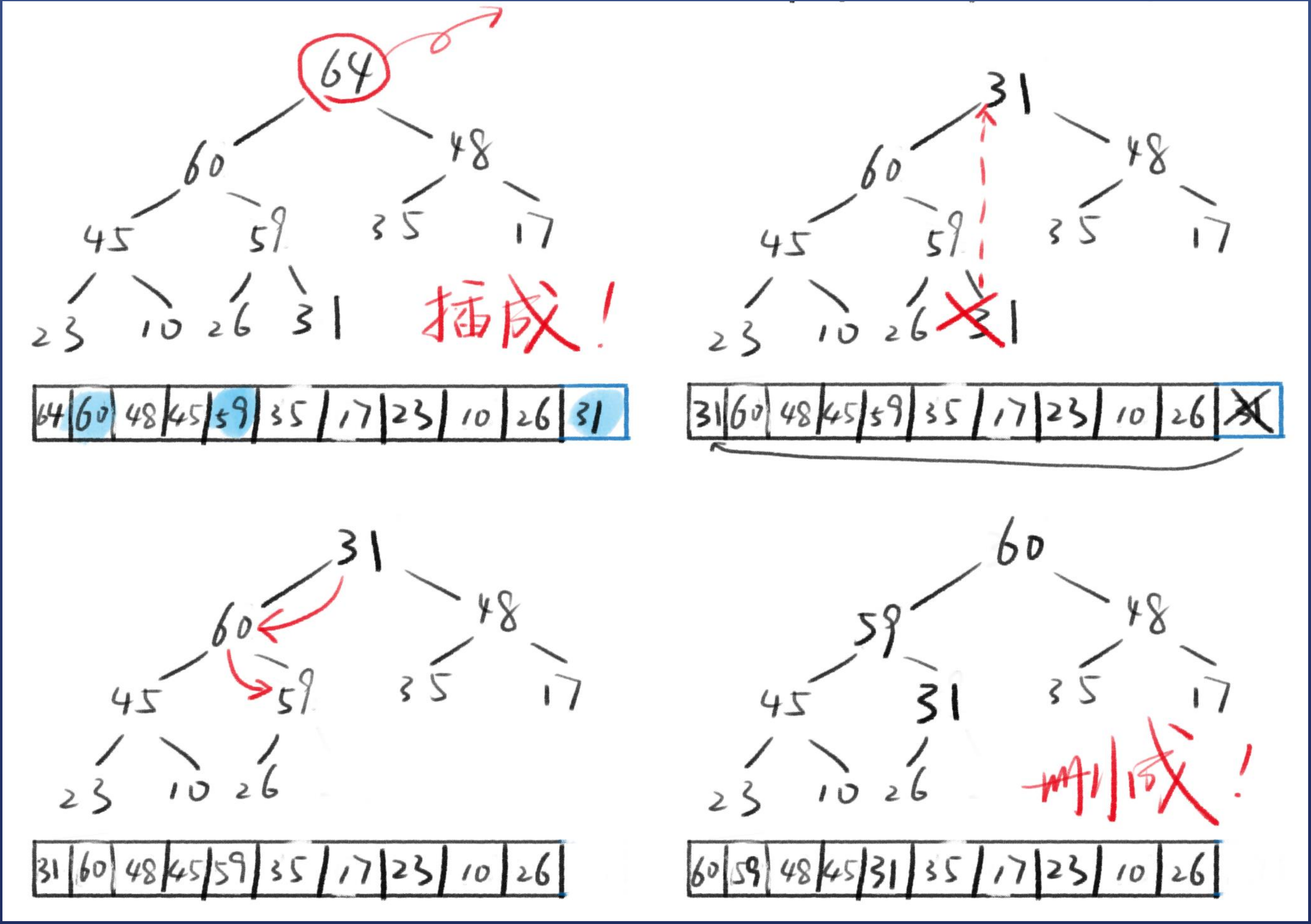
```
heap = [64,59,48,45,31,35,17,23,10,26]
insert = (heap, item) => {
  heap.push(item) // 把新值放到最后一个
  siftUp(heap, heap.length-1) // 开始上升
}
siftUp = (heap, i) => {
  if(i===0){return}
  const parent = parseInt((i-1)/2)
  if(heap[i]>heap[parent]){
    console.log(`换 ${heap[i]} ${heap[parent]}`);
    //注意上一行的分号
    [heap[i],heap[parent]]=[heap[parent],heap[i]]
    siftUp(heap, parent)
  }
}


insert(heap, 60)
console.log(heap)
```

API - extractMax(heap) 如何弹出堆顶的值

要保证弹出后，剩下的元素依然组成堆

思路 siftDown



```
heap = [64, 60, 48, 45, 59, 35, 17, 23, 10, 26, 31]
extractMax = (heap, start, end) => {
  const max = heap[start]
  heap[start] = heap[end - 1]
  heap[end - 1] = max // 把最大值放到最后，一会有用
  siftDown(heap, start, end-1) // 将 start 沉下去
  return max
}

max = extractMax(heap, 0, heap.length)
heap.pop() // 删掉最后一个多余的最大值
console.log(max, heap)
// 64, [60, 59, 48, 45, 31, 35, 17, 23, 10, 26]
```

堆排序，超简单

heap sort

```
array = [9,5,1,4,7,8,3,2,6]
heapSort = arr => {
  // 第一步：数组变成堆 O(N*logN)
  const heap = heapify(arr)
  // 第二步：不停把最大的放到最后 O(N*logN)
  for(let i=0; i<heap.length-1; i++){
    // 刚才留了一手，extractMax 自动把 max 放到最后
    extractMax(heap,0,heap.length-i)
  }
  // 第三步：没有第三步
  return heap
}
heapSort(array)
// [1, 2, 3, 4, 5, 6, 7, 8, 9]

// 复杂度：O(2*N*logN) 约等于 O(N*logN)
```