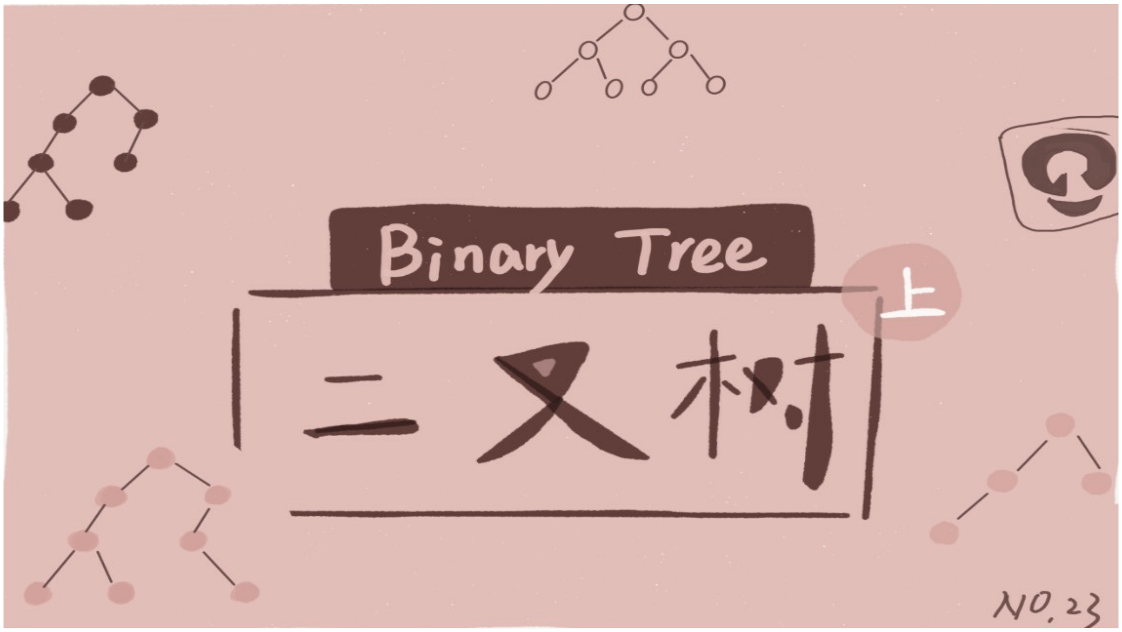
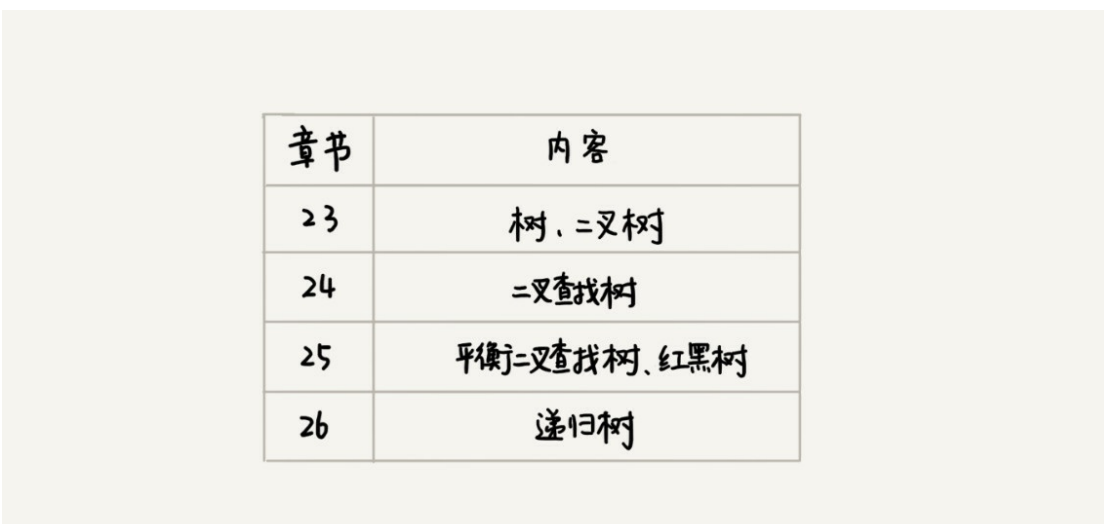
23 | 二叉树基础（上）：什么样的二叉树适合用数组来存储？

2018-11-12 王争



23 | 二叉树基础（上）：什么样的二叉树适合用数组来存储？
朗读人：修阳　10′11′′ | 4.67M

前面我们讲的都是线性表结构，栈、队列等等。今天我们讲一种非线性表结构，树。树这种数据结构比线性表的数据结构要复杂得多，内容也比较多，所以我会分四节来讲解。



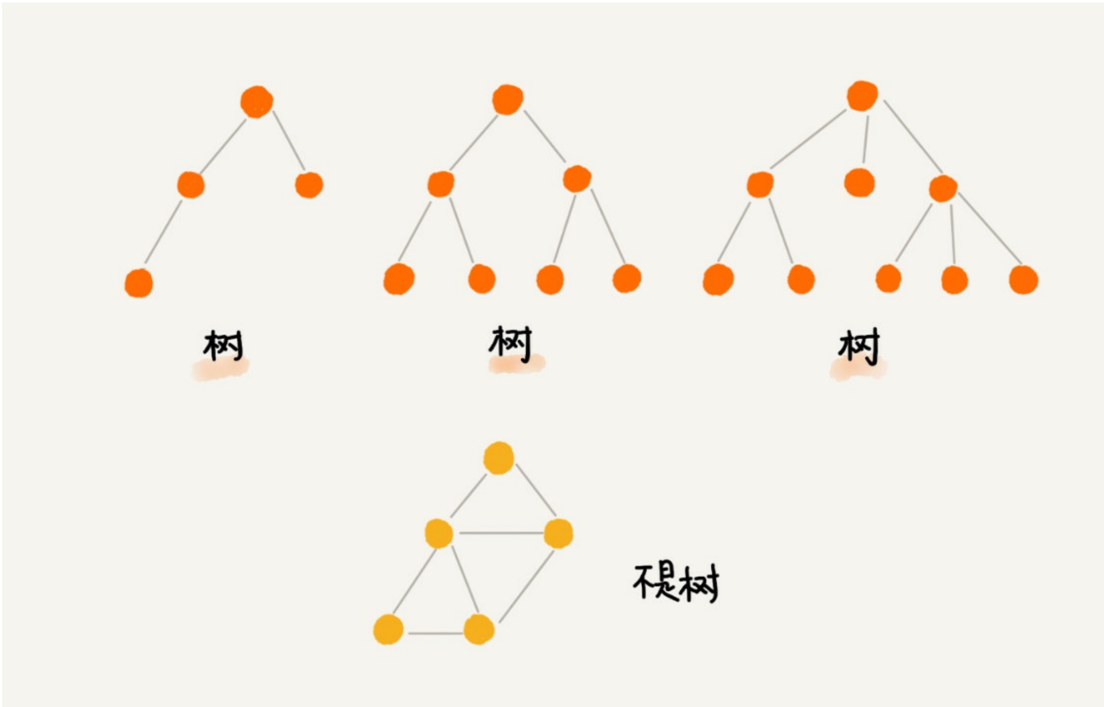
章节	内容
23	树、二叉树
24	二叉查找树
25	平衡二叉查找树、红黑树
26	递归树

我反复强调过，带着问题学习，是最有效的学习方式之一，所以在正式的内容开始之前，我还是给你出一道思考题：二叉树有哪几种存储方式？什么样的二叉树适合用数组来存储？

带着这些问题，我们就来学习今天的内容，树！

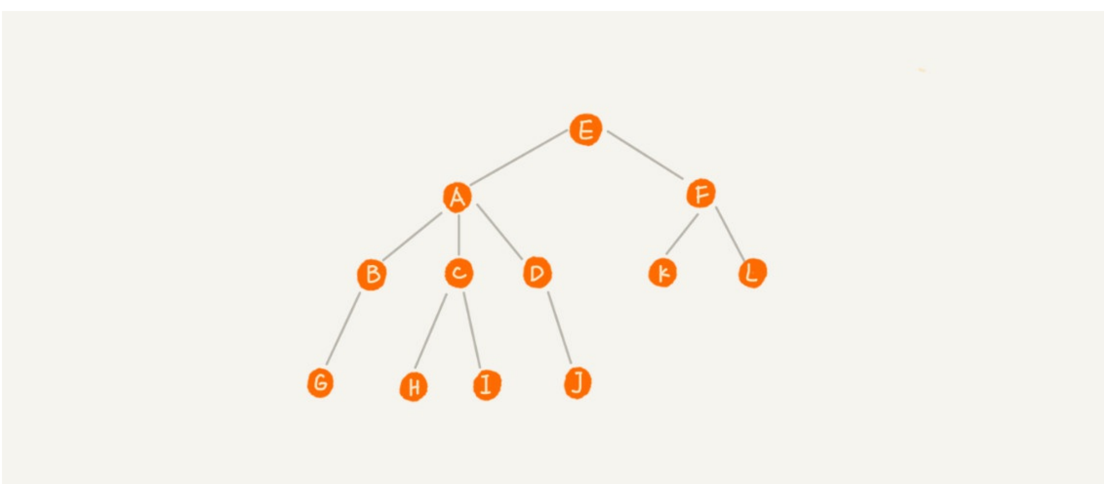
树（Tree）

我们首先来看，什么是“树”？再完备的定义，都没有图直观。所以我在图中画了几棵“树”。你来看看，这些“树”都有什么特征？

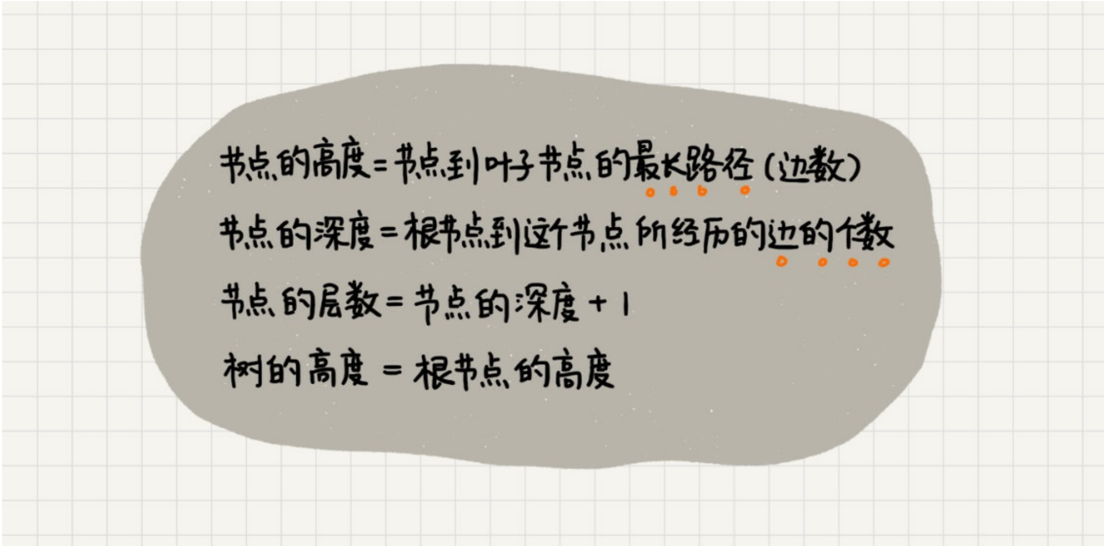


你有没有发现，“树”这种数据结构真的很像我们现实生活中的“树”，这里面每个元素我们叫作“节点”；用来连线相邻节点之间的关系，我们叫作“父子关系”。

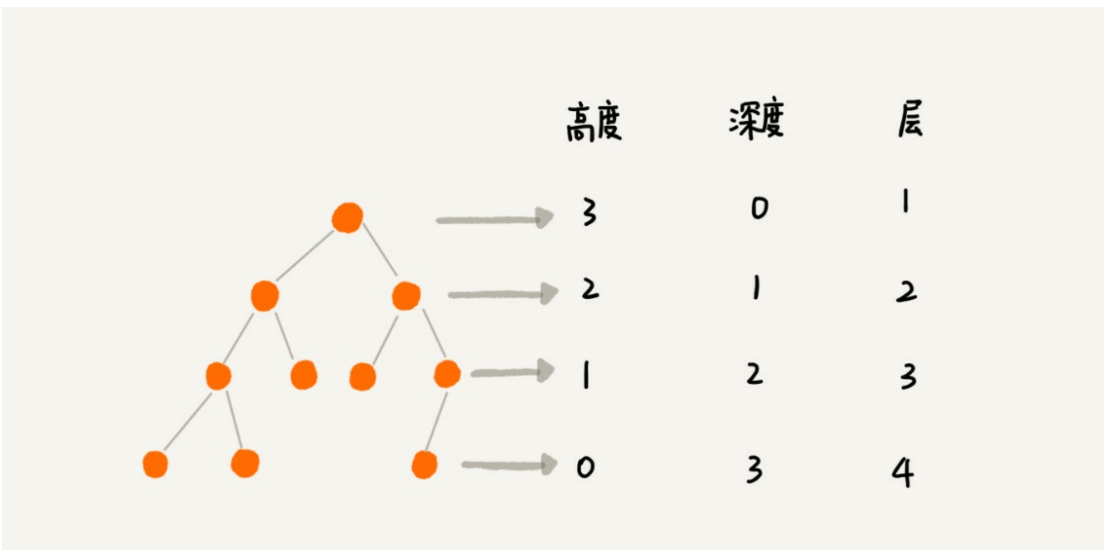
比如下面这幅图，A 节点就是 B 节点的父节点，B 节点是 A 节点的子节点。B、C、D 这三个节点的父节点是同一个节点，所以它们之间互称为兄弟节点。我们把没有父节点的节点叫作根节点，也就是图中的节点 E。我们把没有子节点的节点叫作叶子节点或者叶节点，比如图中的 G、H、I、J、K、L 都是叶子节点。



除此之外，关于“树”，还有三个比较相似的概念：高度（Height）、深度（Depth）、层（Level）。它们的定义是这样的：



这三个概念的定义比较容易混淆，描述起来也比较空洞。我举个例子说明一下，你一看应该就能明白。



记这几个概念，我还有一个小窍门，就是类比“高度”“深度”“层”这几个名词在生活中的含义。

在我们的生活中，“高度”这个概念，其实就是从下往上度量，比如我们要度量第 10 层楼的高度、第 13 层楼的高度，起点都是地面。所以，树这种数据结构的高度也是一样，从最底层开始计数，并且计数的起点是 0。

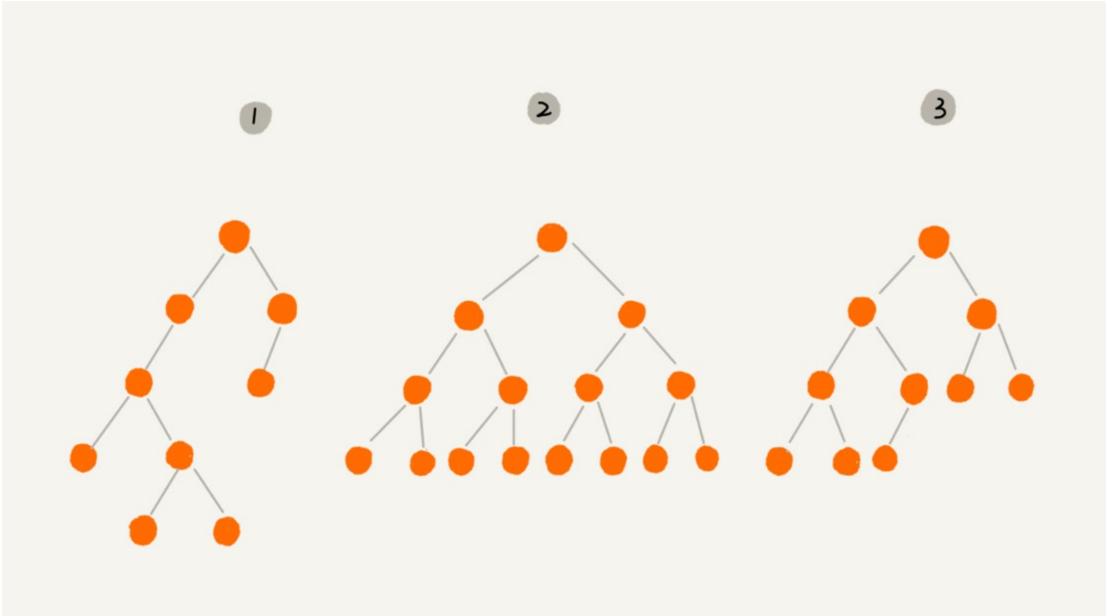
“深度”这个概念在生活中是从上往下度量的，比如水中鱼的深度，是从水平面开始度量的。所以，树这种数据结构的深度也是类似的，从根结点开始度量，并且计数起点也是 0。

“层数”跟深度的计算类似，不过，计数起点是 1，也就是说根节点的位于第 1 层。

二叉树（Binary Tree）

树结构多种多样，不过我们最常用还是二叉树。

二叉树，顾名思义，每个节点最多有两个“叉”，也就是两个子节点，分别是左子节点和右子节点。不过，二叉树并不要求每个节点都有两个子节点，有的节点只有左子节点，有的节点只有右子节点。我画的这几个都是二叉树。以此类推，你可以想象一下四叉树、八叉树长什么样子。

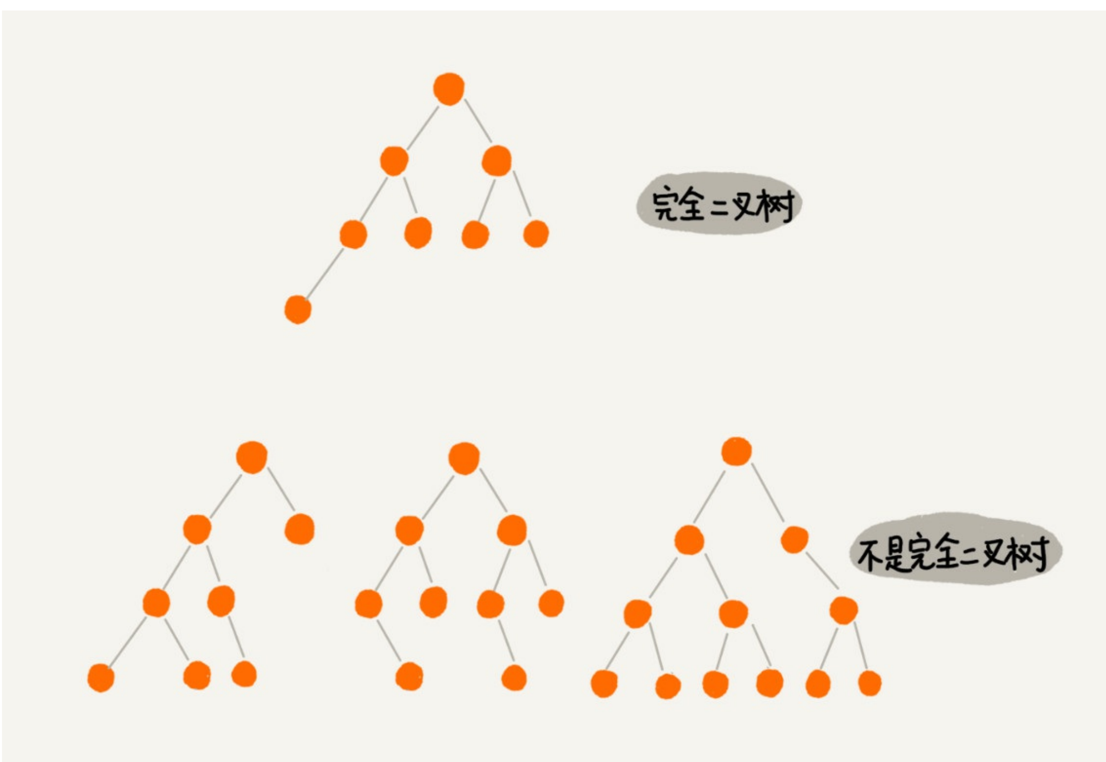


这个图里面，有两个比较特殊的二叉树，分别是编号 2 和编号 3 这两个。

其中，编号 2 的二叉树中，叶子节点全都在最底层，除了叶子节点之外，每个节点都有左右两个子节点，这种二叉树就叫作满二叉树。

编号 3 的二叉树中，叶子节点都在最底下两层，最后一层的叶子节点都靠左排列，并且除了最后一层，其他层的节点个数都要达到最大，这种二叉树叫作完全二叉树。

满二叉树很好理解，也很好识别，但是完全二叉树，有的人可能就分不清了。我画了几个完全二叉树和非完全二叉树的例子，你可以对比着看看。



你可能会说，满二叉树的特征非常明显，我们把它单独拎出来讲，这个可以理解。但是完全二叉树的特征不怎么明显啊，单从长相上来看，完全二叉树并没有特别特殊的地方啊，更像是“芸芸众

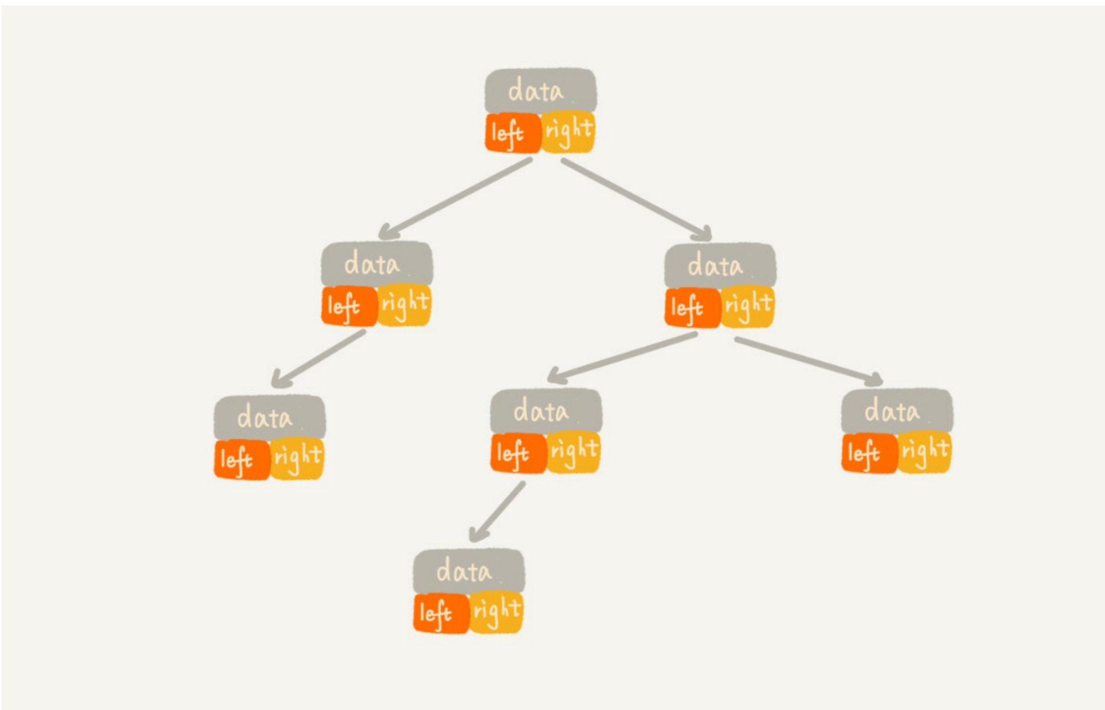
树”中的一种。

那我们为什么还要特意把它拎出来讲呢？为什么偏偏把最后一层的叶子节点靠左排列的叫完全二叉树？如果靠右排列就不能叫完全二叉树了吗？这个定义的由来或者说目的在哪里？

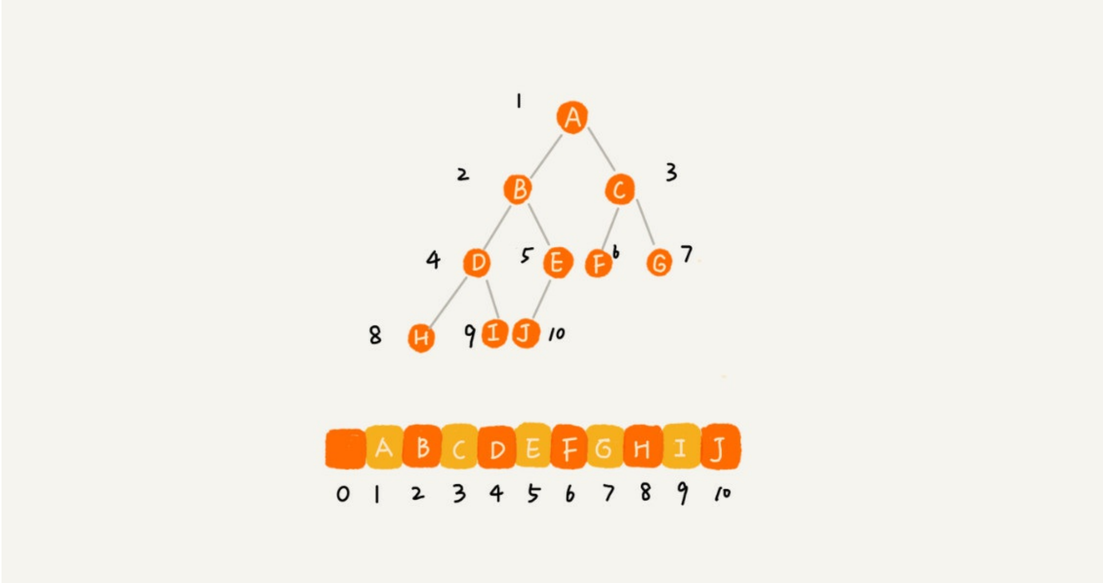
要理解完全二叉树定义的由来，我们需要先了解，如何表示（或者存储）一棵二叉树？

想要存储一棵二叉树，我们有两种方法，一种是基于指针或者引用的二叉链式存储法，一种是基于数组的顺序存储法。

我们先来看比较简单、直观的链式存储法。从图中你应该可以很清楚地看到，每个节点有三个字段，其中一个存储数据，另外两个是指向左右子节点的指针。我们只要拎住根节点，就可以通过左右子节点的指针，把整棵树都串起来。这种存储方式我们比较常用。大部分二叉树代码都是通过这种结构来实现的。

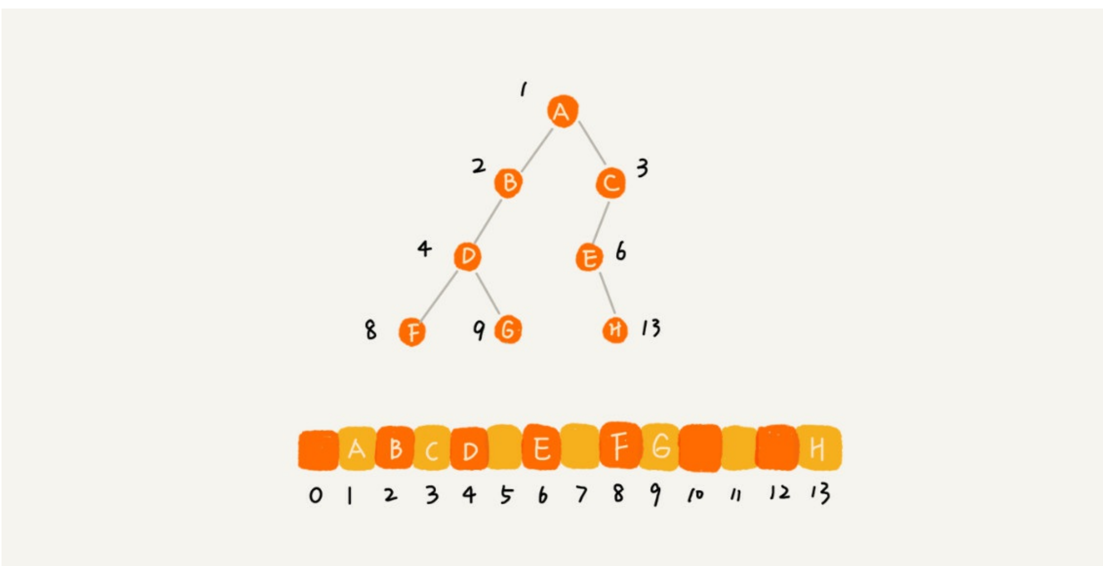


我们再来看，基于数组的顺序存储法。我们把根节点存储在下标 i = 1 的位置，那左子节点存储在下标 2 * i = 2 的位置，右子节点存储在 2 * i + 1 = 3 的位置。以此类推，B 节点的左子节点存储在 2 * i = 2 * 2 = 4 的位置，右子节点存储在 2 * i + 1 = 2 * 2 + 1 = 5 的位置。



我来总结一下，如果节点 X 存储在数组中下标为 i 的位置，下标为 2 * i 的位置存储的就是左子节点，下标为 2 * i + 1 的位置存储的就是右子节点。反过来，下标为 i/2 的位置存储就是它的父节点。通过这种方式，我们只要知道根节点存储的位置（一般情况下，为了方便计算子节点，根节点会存储在下标为 1 的位置），这样就可以通过下标计算，把整棵树都串起来。

不过，我刚刚举的例子是一棵完全二叉树，所以仅仅“浪费”了一个下标为 0 的存储位置。如果是非完全二叉树，其实会浪费比较多的数组存储空间。你可以看我举的下面这个例子。



所以，如果某棵二叉树是一棵完全二叉树，那用数组存储无疑是最节省内存的一种方式。因为数组的存储方式并不需要像链式存储法那样，要存储额外的左右子节点的指针。这也是为什么完全二叉树会单独拎出来的原因，也是为什么完全二叉树要求最后一层的子节点都靠左的原因。

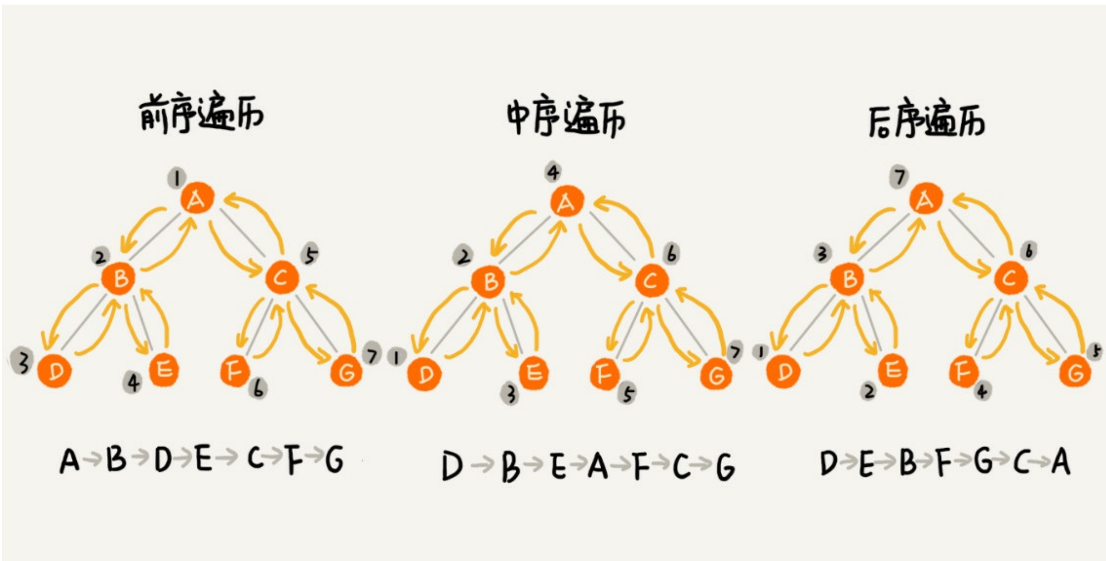
当我们讲到堆和堆排序的时候，你会发现，堆其实就是一种完全二叉树，最常用的存储方式就是数组。

二叉树的遍历

前面我讲了二叉树的基本定义和存储方法，现在我们来看二叉树中非常重要的操作，二叉树的遍历。这也是非常常见的面试题。

如何将所有节点都遍历打印出来呢？经典的方法有三种，前序遍历、中序遍历和后序遍历。其中，前、中、后序，表示的是节点与它的左右子树节点遍历打印的先后顺序。

- 前序遍历是指，对于树中的任意节点来说，先打印这个节点，然后再打印它的左子树，最后打印它的右子树。
- 中序遍历是指，对于树中的任意节点来说，先打印它的左子树，然后再打印它本身，最后打印它的右子树。
- 后序遍历是指，对于树中的任意节点来说，先打印它的左子树，然后再打印它的右子树，最后打印这个节点本身。



实际上，二叉树的前、中、后序遍历就是一个递归的过程。比如，前序遍历，其实就是先打印根节点，然后再递归地打印左子树，最后递归地打印右子树。

写递归代码的关键，就是看能不能写出递推公式，而写递推公式的关键就是，如果要解决问题 A，就假设子问题 B、C 已经解决，然后再来看如何利用 B、C 来解决 A。所以，我们可以把前、中、后序遍历的递推公式都写出来。


```
前序遍历的递推公式：
preOrder(r) = print r->preOrder(r->left)->preOrder(r->right)

中序遍历的递推公式：
inOrder(r) = inOrder(r->left)->print r->inOrder(r->right)

后序遍历的递推公式：
postOrder(r) = postOrder(r->left)->postOrder(r->right)->print r
```

有了递推公式，代码写起来就简单多了。这三种遍历方式的代码，我都写出来了，你可以看看。

复制代码

```
void preOrder(Node* root) {
  if (root == null) return;
  print root // 此处为伪代码，表示打印 root 节点
  preOrder(root->left);
  preOrder(root->right);
}

void inOrder(Node* root) {
  if (root == null) return;
  inOrder(root->left);
  print root // 此处为伪代码，表示打印 root 节点
  inOrder(root->right);
}

void postOrder(Node* root) {
  if (root == null) return;
  postOrder(root->left);
  postOrder(root->right);
  print root // 此处为伪代码，表示打印 root 节点
}
```

二叉树的前、中、后序遍历的递归实现是不是很简单？你知道二叉树遍历的时间复杂度是多少吗？我们一起来看看。

从我前面画的前、中、后序遍历的顺序图，可以看出来，每个节点最多会被访问两次，所以遍历操作的时间复杂度，跟节点的个数 n 成正比，也就是说二叉树遍历的时间复杂度是 O(n)。

解答开篇 & 内容小结

今天，我讲了一种非线性表数据结构，树。关于树，有几个比较常用的概念你需要掌握，那就是：根节点、叶子节点、父节点、子节点、兄弟节点，还有节点的高度、深度、层数，以及树的高度。

我们平时最常用的树就是二叉树。二叉树的每个节点最多有两个子节点，分别是左子节点和右子节点。二叉树中，有两种比较特殊的树，分别是满二叉树和完全二叉树。满二叉树又是完全二叉树的一种特殊情况。

二叉树既可以用链式存储，也可以用数组顺序存储。数组顺序存储的方式比较适合完全二叉树，其他类型的二叉树用数组存储会比较浪费存储空间。除此之外，二叉树里非常重要的操作就是前、中、后序遍历操作，遍历的时间复杂度是 O(n)，你需要理解并能用递归代码来实现。

课后思考

1. 给定一组数据，比如 1，3，5，6，9，10。你来算算，可以构建出多少种不同的二叉树？
2. 我们讲了三种二叉树的遍历方式，前、中、后序。实际上，还有另外一种遍历方式，也就是按层遍历，你知道如何实现吗？

欢迎留言和我分享，我会第一时间给你反馈。

上一篇 22 | 哈希算法（下）：哈希算法在分布式系统中有哪些应用？

下一篇 24 | 二叉树基础（下）：有了如此高效的散列表，为什么还需要二叉树？

写留言

精选留言

失火的夏天 20

1.是卡特兰数，是C[n,2n] / (n+1)种形状，c是组合数，节点的不同又是一个全排列，一共就是n!*C[n,2n] / (n+1)个二叉树。可以通过数学归纳法推导得出。
2.层次遍历需要借助队列这样一个辅助数据结构。（其实也可以不用，这样就要自己手动去处理节点的关系，代码不太好理解，好处就是空间复杂度是o(1)。不过用队列比较好理解，缺点就是空间复杂度是o(n)）。根节点先入队列，然后队列不空，取出对头元素，如果左孩子存在就入列队，否则什么也不做，右孩子同理。直到队列为空，则表示树层次遍历结束。树的层次遍历，其实也是一个广度优先的遍历算法。

2018-11-12

D→_→M 2

老师是否可以在您专栏的github上传一下二叉树这几节的相关代码，还有除了递归遍历二叉树，循环遍历是否也可以讲一下，或者在github上上传一下相关代码，自行研究学习。

2018-11-12

作者回复

非递归遍历比较复杂 不建议非得给自己制造学习难度 除非是为了面试。其他的二叉树的代码我会放到github上

2018-11-13

nothing 2

后序遍历节点不是最多被访问三次嘛， 还有那个深度我们学的深度和层次是一样的哇

2018-11-12

作者回复

1 从图上看是两次

2 从生活中的理解来说 应该没有第0层之说 但是是有深度为0的说法

2018-11-12

Monday

1

知识点都很好理解。但是两道思考题难到我了，得多查查资料。

2018-11-13

作者回复

带着问题查资料的过程是最好的学习方法。比单纯填鸭式的学习好多了

2018-11-13

传说中的成大大

1

刚刚思考了完全二叉树的定义 叶子结点必须要在最后两层 如果不在最后两层的话通过数组顺序存储也会浪费空间吧

2018-11-12

作者回复

是的

2018-11-13

Liam

1

1 递归地理解一下：按住根节点，如果有k个左节点，则有n-k-1个右节点，分步乘法，f(n) = f(k) * f(n - k - 1)， k可能性从0 到 n - 1 ,分步加法： f(n) = f(0)f(n-1) + ... + f(n-1)f(0)， 怎么计算该递推公式呢？参考Catalon数

2018-11-12

spark

1

写了下，测试了下，貌似没什么问题

1.定义树节点类

```
public class Tree<T> {
public Tree left;
public Tree right;
public T data;
public Tree(T data){
this.data = data;
}
}
```

2.测试类

```
public class TestTree {
public static void main(String[] args){
Tree<String> a = new Tree<>("A");
Tree<String> b = new Tree<>("B");
Tree<String> c = new Tree<>("C");
Tree<String> d = new Tree<>("D");
Tree<String> e= new Tree<>("E");
Tree<String> f = new Tree<>("F");
Tree<String> g = new Tree<>("G");
a.left = b;
a.right = c;
b.left = d;
b.right = e;
c.left = f;
c.right = g;
```

```
order(a);
System.out.println(a.data);
}
```

3.层遍历方法

```
public static void order(Tree r){
if(r==null){
return;
}
order(r.left);
order(r.right);
if(r.left != null) {
System.out.println(r.left.data);
}
if(r.right != null){
System.out.println(r.right.data);
}
}
}
```

2018-11-12

往事随风，顺其自然 1

按照蹭便利使用队列，广度优先搜索

2018-11-12

飞羽 0

迟到的习题

最近js写烦了，用Go实现了一遍。

具体做法跟楼上差不多，用一个数组存储每层的节点，并打印，然后将当前层级的下一层节点分别装进新的数组中，之后递归到下一层。

下面是代码实现，同时也做了先中后序遍历：

```
package main

import (
"fmt"
)

type Tree struct {
left *Tree
right *Tree
data int
}

func find(t *Tree, data int) (*Tree, int) {
if t.data == data {
return t, 0
} else if data < t.data {
if t.left != nil {
return find(t.left, data)
} else {
return t, 1
```

```
}
} else {
if t.right != nil {
return find(t.right, data)
} else {
return t, 2
}
}
}

func insert(t *Tree, data int) {
t, driction := find(t, data)
nt := Tree{nil, nil, data}
if driction == 1 {
t.left = &nt
} else if driction == 2 {
t.right = &nt
}
}

func preEach(t *Tree) {
fmt.Println(t.data)
if t.left != nil {
preEach(t.left)
}
if t.right != nil {
preEach(t.right)
}
}

func postEach(t *Tree) {
if t.left != nil {
postEach(t.left)
}
if t.right != nil {
postEach(t.right)
}
fmt.Println(t.data)
}

func midlEach(t *Tree) {
if t.left != nil {
midlEach(t.left)
}
fmt.Println(t.data)
if t.right != nil {
midlEach(t.right)
}
}

func order(ts []*Tree, l int) int {
```

```
nts := make([]*Tree, 0) // 此处有优化空间，考虑如何将切片length降到最低
if len(ts) == 0 {
return 0
}
for _, t := range ts {
fmt.Println(t.data)
if t.left != nil {
nts = append(nts, t.left)
}
if t.right != nil {
nts = append(nts, t.right)
}
}
return order(nts, l*2)
}

func main() {

t := Tree{&Tree{nil, nil, 8}, &Tree{nil, &Tree{nil, nil, 15}, 12}, 10}

insert(&t, 20)
insert(&t, 4)
insert(&t, 5)
insert(&t, 1)

fmt.Println(`先序遍历：`)
preEach(&t)
fmt.Println(`end ----------------------------`)

fmt.Println(`中序遍历：`)
midlEach(&t)
fmt.Println(`end ----------------------------`)

fmt.Println(`后序遍历：`)
postEach(&t)
fmt.Println(`end ----------------------------`)

fmt.Println(`按层遍历：`)
ts := make([]*Tree, 1)
ts[0] = &t
order(ts, 1)
fmt.Println(`end ----------------------------`)
}
```

2018-11-14

向科

0

老师，我有一个问题:

关于这三种遍历二叉树的算法，区别在于第几步访问父节点，在遍历的结果和时间复杂度上没有区别，在实际应用时，我们该如何选择？他们的这种区别和实际问题有什么关系呢？能举例说明吗？

2018-11-14

我是谁 0

第一个思考题，猜了一下应该是有132种不同的二叉树，但是公式没推出来。。。

2018-11-13

太空土豆 0

树的高度和深度的定义，现在没有统一的说法。讲义中从0开始，一般情况下没有问题。当某颗树是空的时候，它的高度是多少?当某棵树只有一个根结点时，它的高度又是多少?所以讲义中这种定义方法不能很好诠释空树的高度。对于这个问题，我比较倾向于叶结点的高度为1，跟节点的深度为1这种定义方式。这样，空树的高度为0，只有一个根结点的树高度为1。其实，这个问题我也纠结过。

2018-11-13

我的心里只有工作 0

我想知道老师什么时候出书，想买本

2018-11-13

liangjf 0

这个主要理解了遍历那里。我是这样想的。对于同一个结点，前，中，后序遍历其实就是针对当前遍历到的结点来说的。前序遍历，第一次遍历本结点，然后分别左右结点。中序遍历，第一次遍历左结点，然后分别是当前结点和右节点。后序遍历，第一次遍历右节点，然后分别是左结点和当前结点。

记住是以当前遍历到的结点为参考就OK

2018-11-13

along 0

按层遍历利用队列实现:

```
/**
* 按层遍历
* @param tree 根节点
*/
public static void levelOrder(BinaryTree tree) {
// 利用队列FIFO特点来实现按层遍历
LinkedList<BinaryTree> linkedList = new LinkedList<>();
// 记录当前遍历到哪个结点
BinaryTree currentNode = tree;
// 根节点入队
linkedList.add(currentNode);
// 从队列中弹出各结点数据，直到队列为空，遍历完毕
while (linkedList.size()>0){
// 弹出队首元素(当前结点)，打印其数据，并依次将其左右子节点入队
currentNode = linkedList.poll();
System.out.print(currentNode.data+" -> ");
if (currentNode.left!=null) {
linkedList.add(currentNode.left);
}
if (currentNode.right!=null) {
linkedList.add(currentNode.right);
}
}
}
```

2018-11-12

NeverMore 0

按层遍历使用队列吧，广度优先搜索。

2018-11-12

sky卤代烃 0

我当时记忆遍历顺序的时候默认每个节点访问三次，前序就是第一次访问，中序是第二次访问，后续就是第三次访问，我感觉按这种顺序更容易分清访问顺序

2018-11-12

城 0

1.我说下我的方法。分别安按照不同的层数来统计。层数最大6，如果不考虑节点的数值，只考虑节点的结构的话，那么层数为6时，有2种构造方式，根+左子树或根+右子树。层数为5时依次类推。

2018-11-12

Sharry 0

层序遍历使用队列

2018-11-12

朱月俊 0

感觉第一题题目没有描述清楚，比如给的数字是按照层序遍历还是插入二叉树的顺序

2018-11-12

作者回复

树的形态不一样 数字排布不一样 只要两者满足其一就表示不同的二叉树

2018-11-12