基本数据结构

Algorithms

+ Data Structures

= Programs

引言

- 计算机储存和组织数据的方式是非常复杂的，幸运的是，我们往往可以在更高的抽象层次上去考虑这一问题
- 数据结构（data structure）是由数据元素的集合和该集合中元素之间的关系所组成的，它并不依赖于某种特定的实现方法
- 通常，我们还希望数据结构能够支持某些特定的操作

引言

- 其实我们在搜索和排序章节中一直提到的“序列”也是一种数据结构
- 这也是最常用的数据结构之一，正式的名称为线性表（linear list）
- 它拥有最为普遍和简单的线性结构，各个元素是相继排列的，相邻元素是直接前驱和直接后继的关系
- 具体在计算机中可以使用数组或链表来实现，但这并不是我们所关心的内容

引言

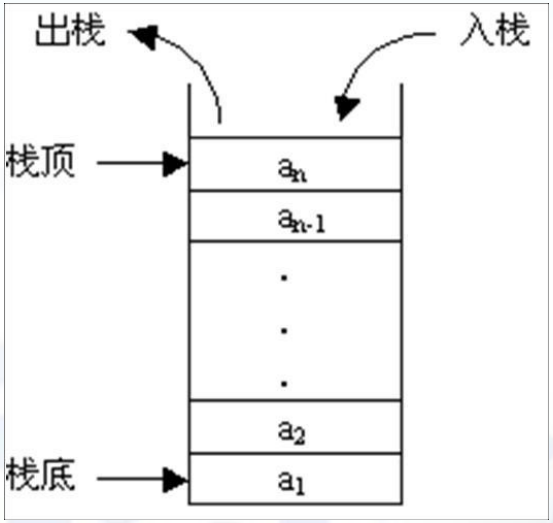
- 常见的数据结构操作
  - 数据结构的构建与销毁
  - 查找、插入、删除某个元素
  - 遍历整个数据结构
- 并非每种数据结构都要支持所有的操作，这与数据结构本身的特性以及具体的应用场景有关
- 线性表能够很容易地实现以上操作

概要

- 这一节我们将介绍以下几个除普通线性表之外最为经典与常见的数据结构
  - 栈
  - 队列
  - 树
- 通过对它们的学习，我们会初步了解如何根据实际的需求来组织和管理数据

栈

- 栈其实是一种特殊的线性表，只是我们限定了对它的访问形式
- 通常，栈（stack）可以定义为只允许在一端进行插入和删除的线性表



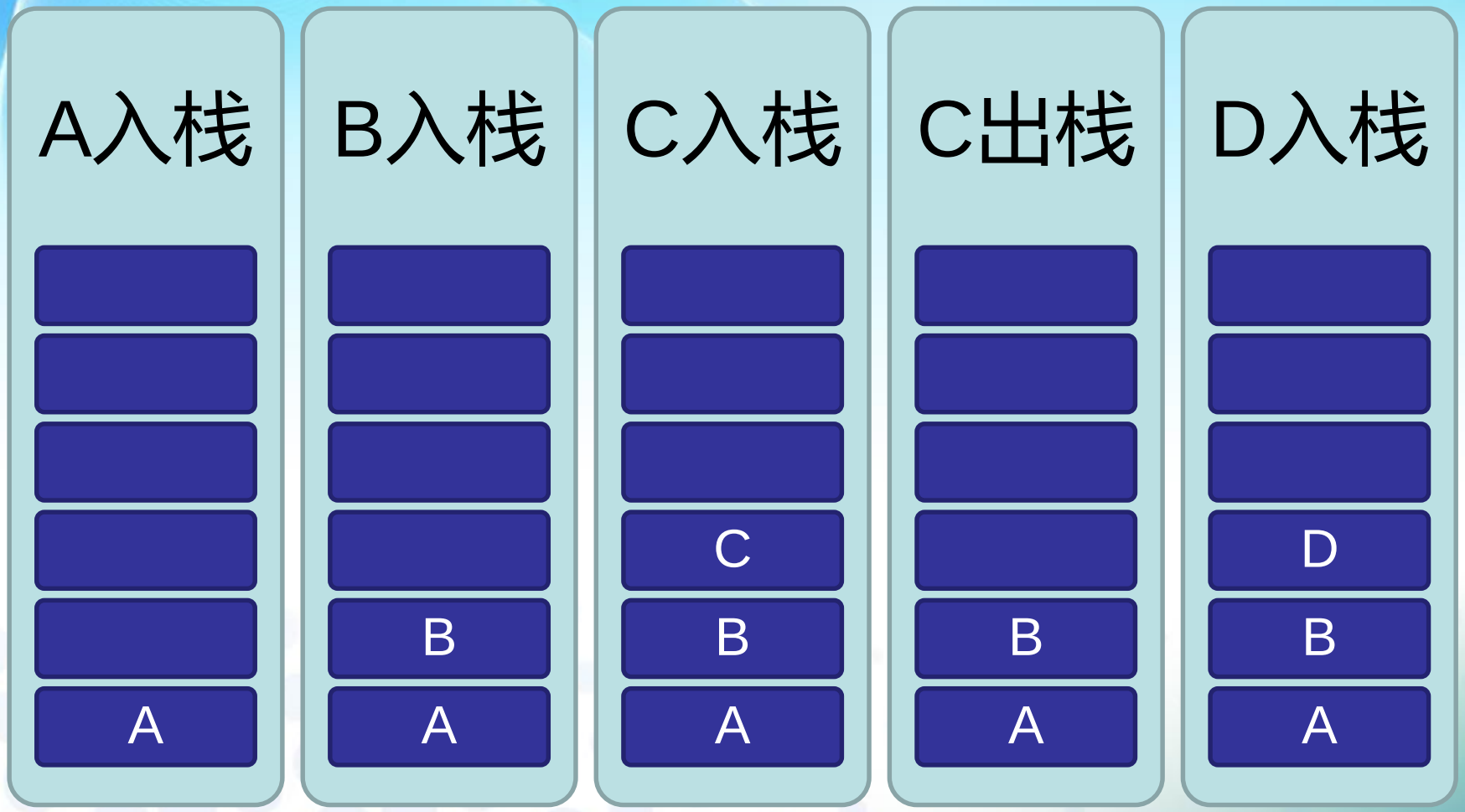
栈

- 我们把栈的头部，即允许插入删除的一端称为栈顶（top）；而栈的尾部，即不允许插入和删除的另一端叫做栈底（bottom）
- 在栈顶插入一个元素的过程叫做入栈（push），删除一个元素的过程叫做出栈（pop）
- 很容易发现，按这样的规则，后入栈的元素会先出栈，称为后进先出（LIFO, Last in first out）

栈

- 想一想，栈的这种后进先出的过程和生活中的哪些场景类似？

栈操作的例子



栈操作的伪代码

Push(Stack, x)
top ← top + 1
Stack[top] ← x

Pop(Stack)
if Stack is empty **then** error
top ← top – 1
return Stack[top+1]

栈的作用

- 栈所支持的操作看似非常简单，但却能完成你意想不到的重要工作
- 计算机程序在管理局部内存时，就是按照栈的模式组织的
- 在我们前面提到过的“递归”过程中，同样会用到递归工作栈，每递归调用一次，栈中就会分配新的工作单元入栈，防止冲突，而在过程结束后出栈返回调用处

栈的应用举例

- 给定一个括号的序列，如果每对左括号与右括号都正确匹配，那么称之为合法的括号序列，否则即为非法的括号序列。如“(())()”即是合法的括号序列，而“()())(”则是非法的括号序列
- 我们的目的是设计一个算法，能够检查括号序列的合法性，进一步的，能够输出没有正确匹配的括号

栈的应用举例

- 提示：可以观察到，如果从左到右扫描，那么每一个右括号将与最近遇到的未匹配的左括号相匹配
- 这个观察的结果使我们联想到可以在从左到右的扫描过程中把所遇到的左括号入栈，每遇到一个右括号时，就把它与栈顶的左括号（如果存在）相匹配，同时该左括号出栈

队列

- 队列（queue）是另一种限定存取位置的线性表，它只允许在表的一端插入，在另一端删除

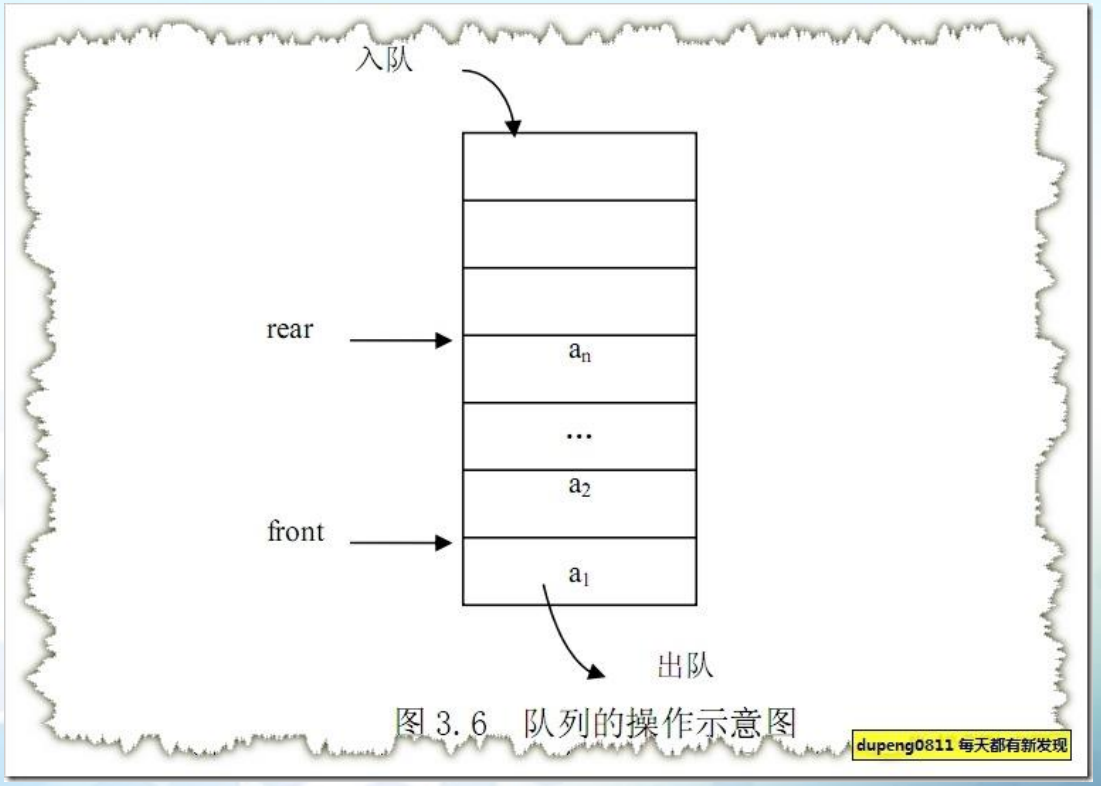


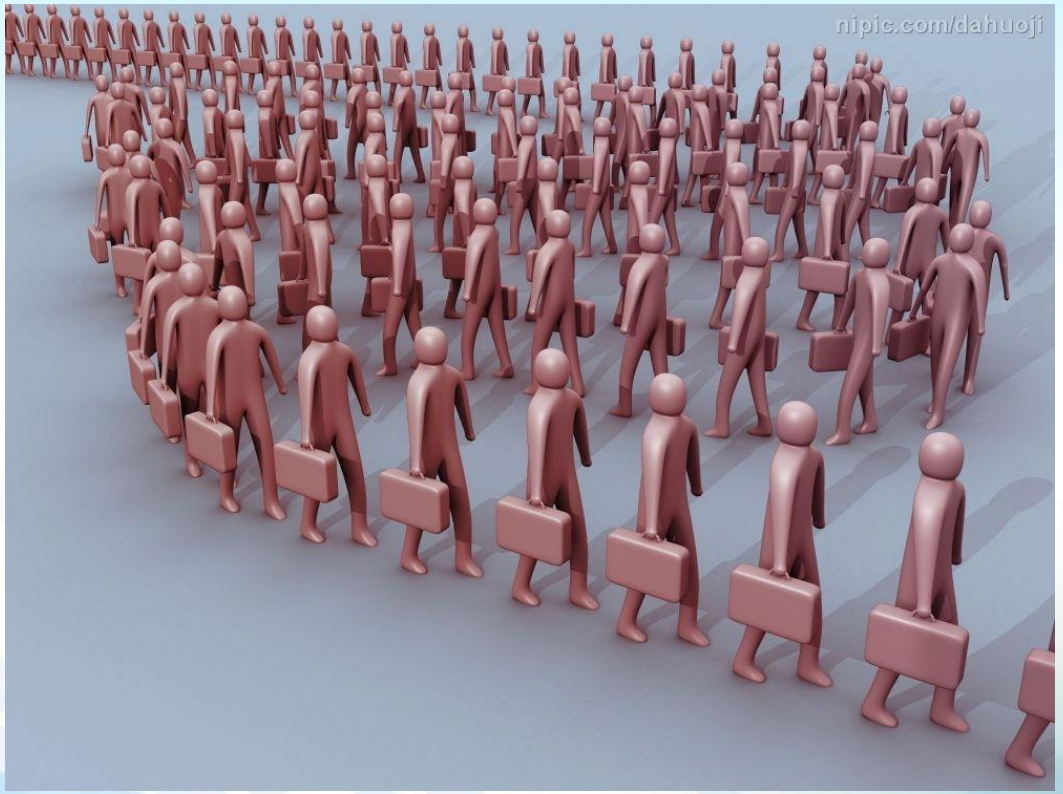
图 3.6　队列的操作示意图

队列

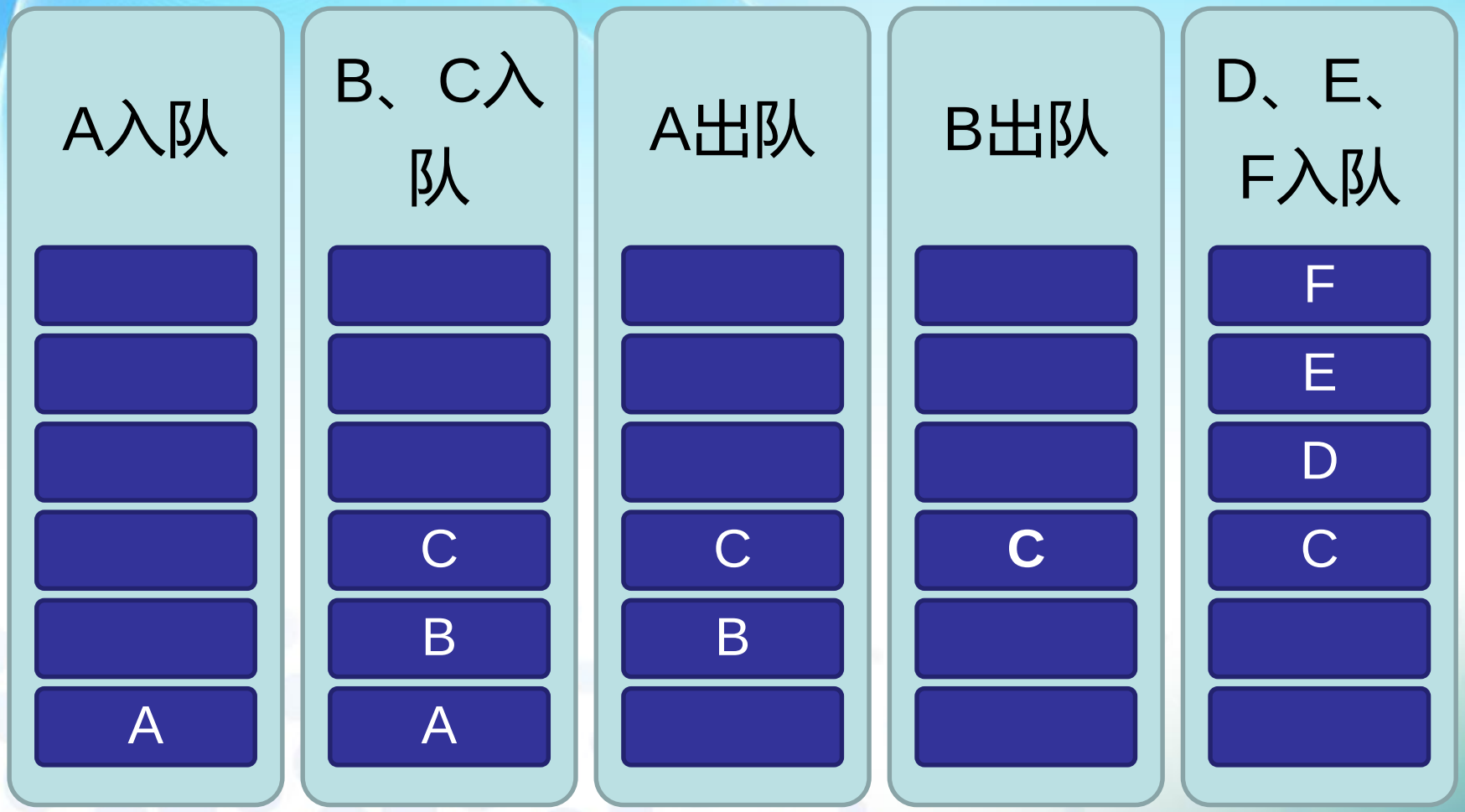
- 我们把允许插入的一端叫做队尾（rear），允许删除的一端叫做队头（front）
- 在队列中插入一个元素的过程称为入队（EnQueue），在队列中删除一个元素的过程称为出队（DeQueue）
- 与栈不同的是，在这样的设定下，先进队列的元素会先从队列中删除，称为先进先出（FIFO, First in first out）

队列

- 队列，顾名思义就是描绘了人们排队的过程



队列操作的例子



队列操作的伪代码

```
EnQueue(Queue, x)
Queue[rear] ← x
rear ← rear + 1

DeQueue(Queue)
x ← Queue[front]
front ← front + 1
return x
```

队列的作用

- 在操作系统中，作业调度和输入输出管理都有一个排队问题
- 作业调度策略中很常用的一个策略就是“先来先服务”，这通常都用队列来实现
- 等待输出的作业排成队，先提出请求的作业排在前面，一个作业完成并输出后出队，在队列的下一个作业再利用资源进行操作，依此类推

双端队列

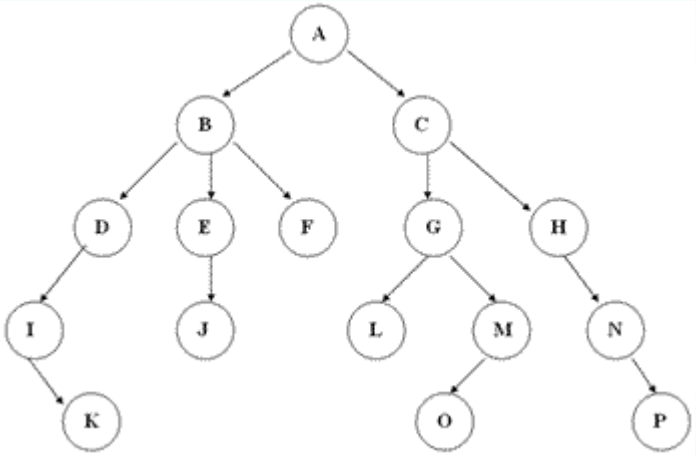
- 双端队列（Double-ended queue, Deque）是对队列概念的一种扩展
- 普通队列只允许在一端删除而在另一端插入，而双端队列可以在队列的两端进行插入和删除
- 双端队列的结构与队列类似，同学们可以尝试自己写出其入队和出队（注意现在有两种情况）的伪代码

树

- 我们前面讲过的栈和队列等数据结构，从本质上说是线性的
- 它们从时间和空间上来说都是依次紧密排列的，如果我们希望表达更复杂的关系，比如层次化的组织形式，就需要更为复杂的数据结构了
- 这种能够表达层次关系的分支数据结构被称为树（tree）

树

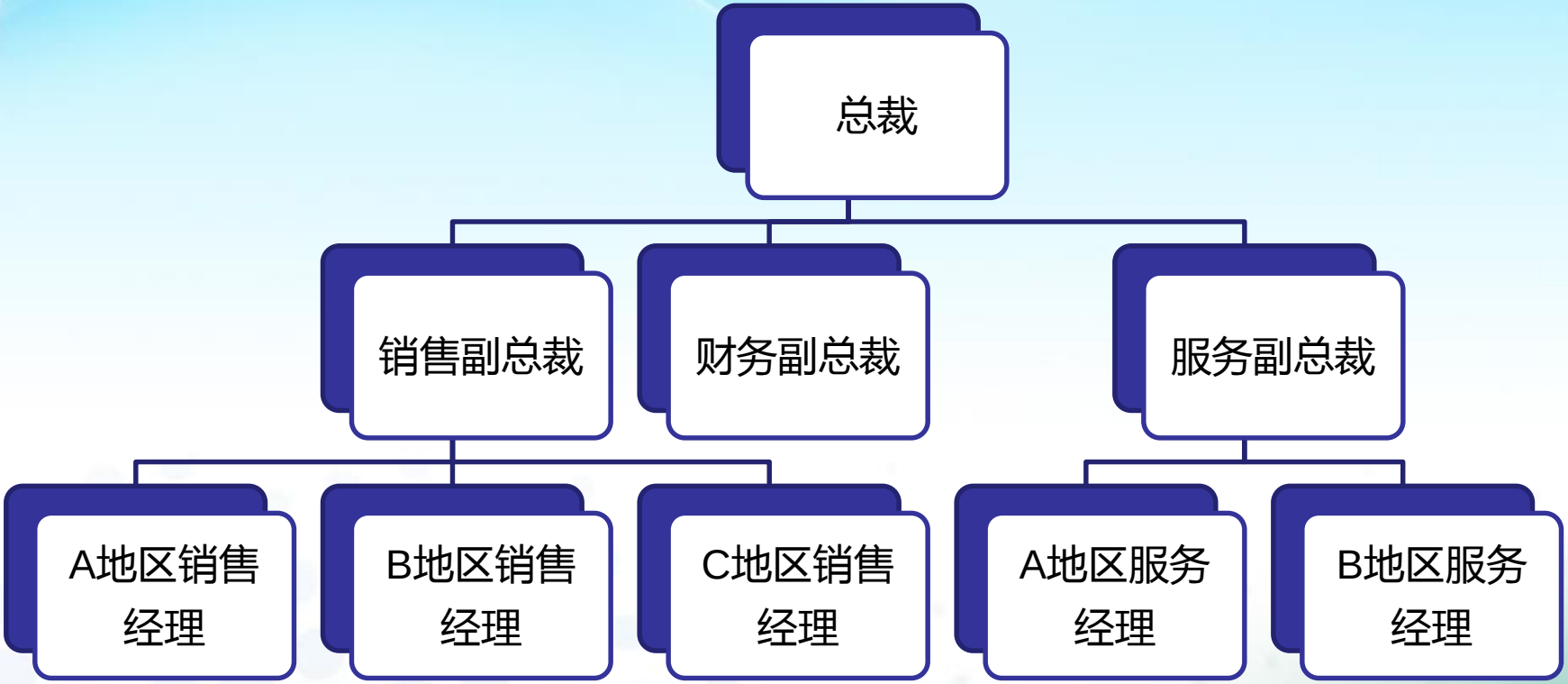
- 从图论的角度来说，树指没有回路的图，这我们会在后面的章节中再做阐述
- 这里所说的树是指具有层次结构的有根树（rooted tree）



树

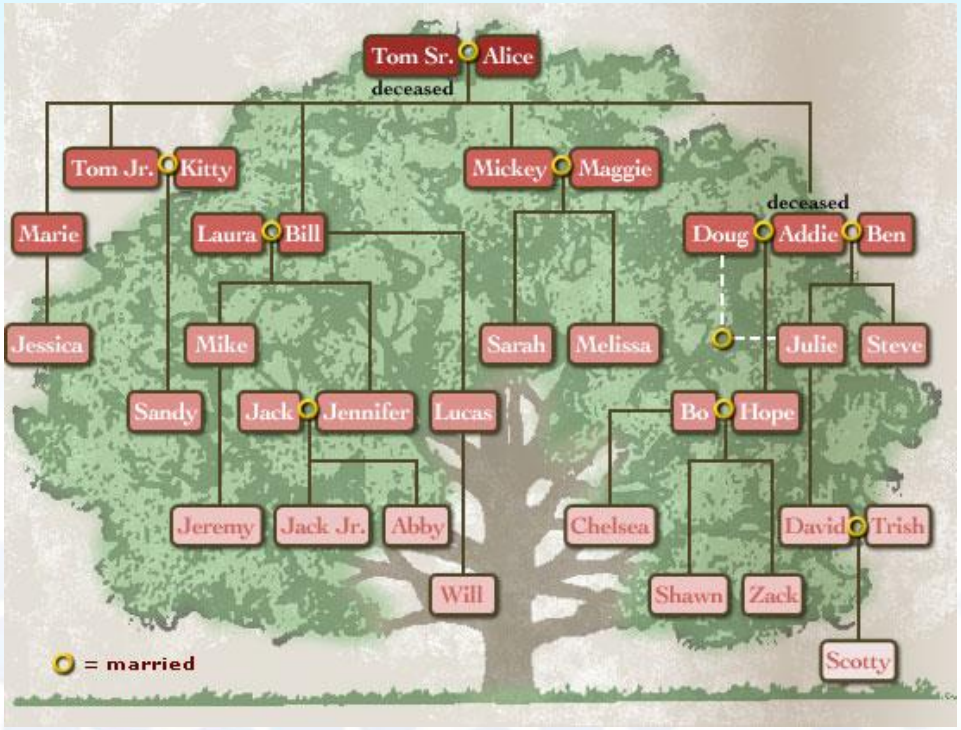
- 树中的每一个位置称为一个结点（node）
- 树根部的结点称为根结点（root node）
- 底下端点处的结点称为终端结点（terminal node），也叫做叶子结点（leaf node）
- 我们通常把从根结点到叶子结点的最长路径上的结点数称为树的深度（depth），比如上图中的树深度为5

树的例子


总裁
销售副总裁
财务副总裁
服务副总裁
A地区销售经理
B地区销售经理
C地区销售经理
A地区服务经理
B地区服务经理

树

- 有时候，我们会把树看作这样的结构：是由每个节点派生出其直接下层的节点，就像我们的家谱（英语中的family tree）



树

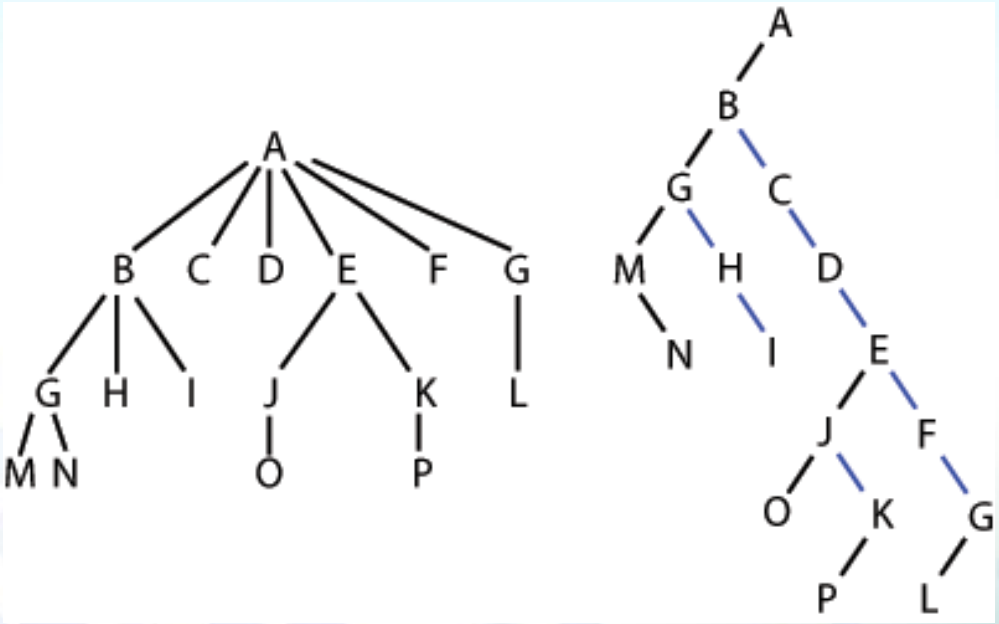
- 此时，我们称一个结点的直接后代为子结点（child），称其直接祖先为父结点（parent），而将有同一个父节点的那些结点称之为兄弟节点（sibling）
- 对于树中任意一个结点，该结点与其下层的结点也构成树结构，称为子树（subtree）
- 每个子节点就是父节点下子树的根节点，这样的子树称为父节点的一个分支（branch）

二叉树

- 二叉树（binary tree）是一种特殊的有根树，我们要求二叉树的每个结点最多只能有两个子节点，分别称为左儿子和右儿子
- 这样的命名也就意味着对于二叉树来说，其左、右子树的地位是需要加以区别的，因此其子树的次序不能颠倒
- 二叉树由于其结构定义良好，具有不错的时间与空间性能，因此使用最为频繁

二叉树的转换

- 由于二叉树拥有这样的优势，通常我们希望能将一般的树转换为二叉树，同时能保持原树的性质，如下图所示

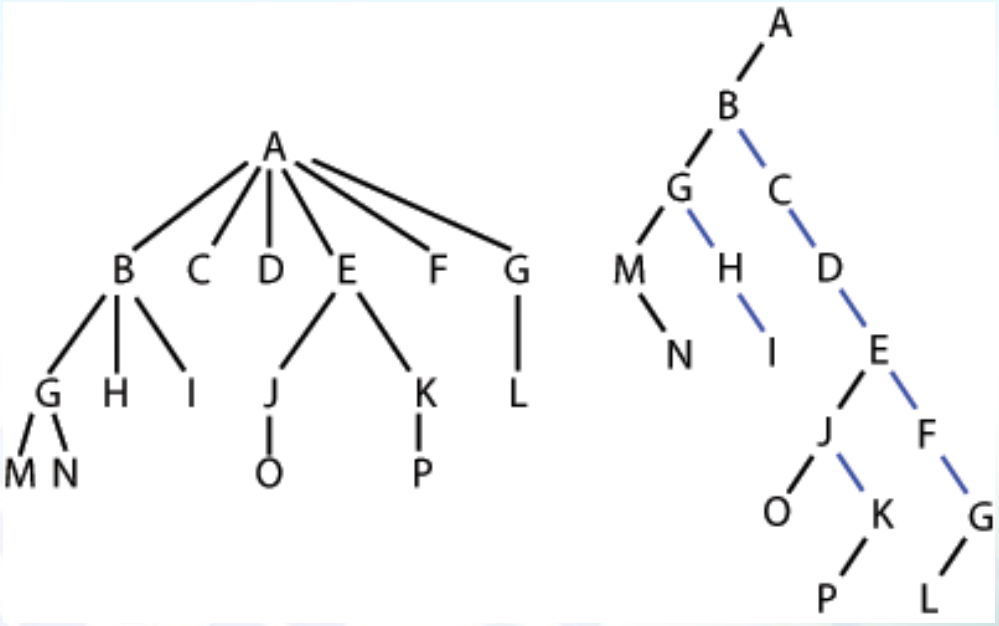


二叉树的转换

- 转换是根据“左儿子，右兄弟”这样的原则完成的
- 具体可以简述为以下的过程
  - 父节点只保留第一个儿子为左儿子结点，去除与其它儿子节点之间的联系
  - 每个结点将在原来树中与自己最靠近的右兄弟结点作为自己的右儿子
- 大家可以自行验证这一算法的正确性

二叉树的转换

- 这样的转化算法是非常高效的，而且原树与转换后的二叉树是一一映射的（想想怎样在二叉树中找到父结点所有的儿子）



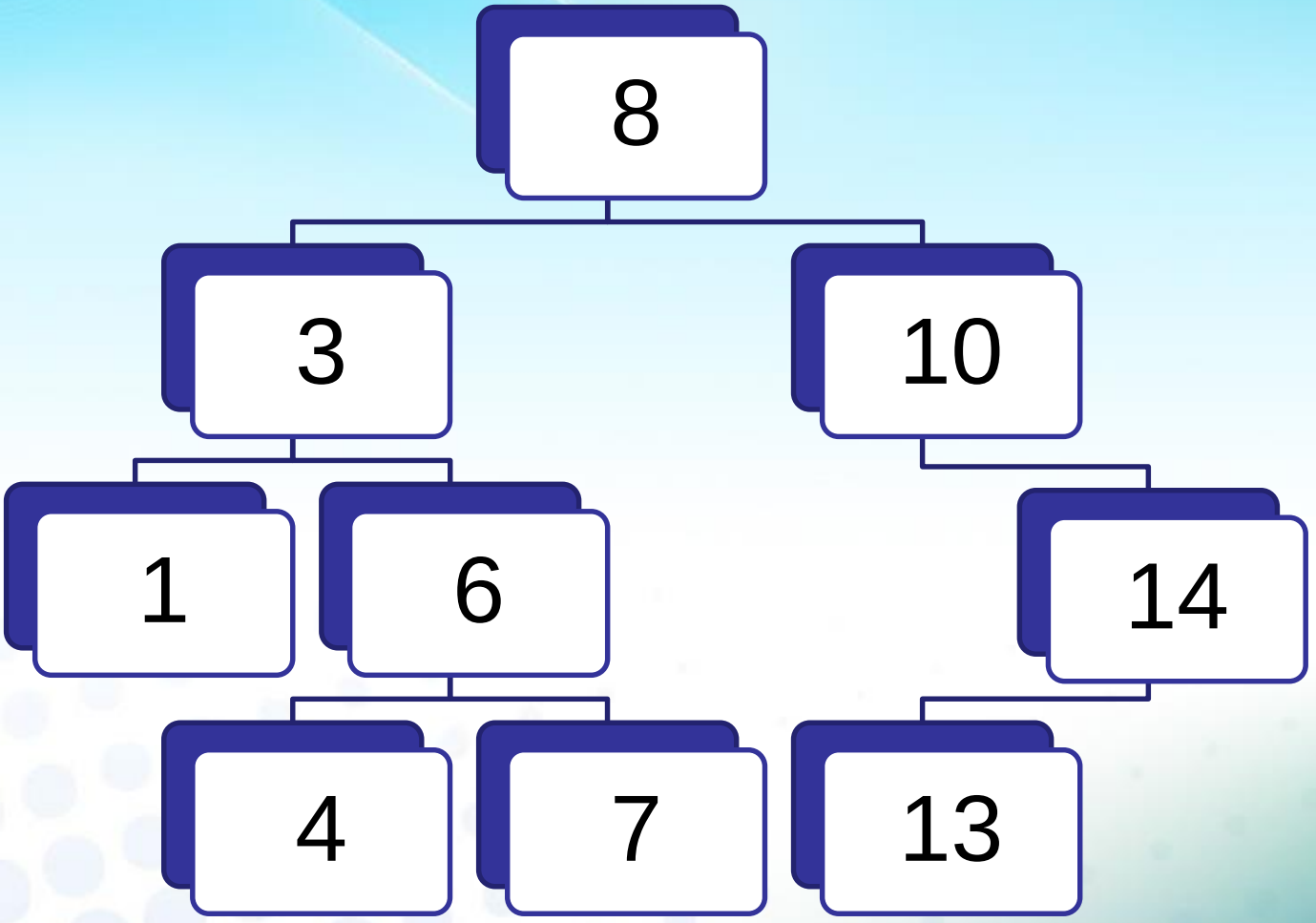
二叉搜索树

- 在前面介绍搜索时，我们提到过，二分搜索无法高效地处理有实时数据插入或是删除的情况
- 对于这样的情形，二叉搜索树（binary search tree）就是一种能满足这种需求的更好的选择了
- 二叉搜索树在形式上就是一棵二叉树，但同时又保持了特殊的性质

二叉搜索树

- 二叉搜索树的每个结点都附有一个关键词（key），一般要求关键词互不相同
- 左子树（如果存在）上所有结点的关键词都小于根节点的关键词
- 右子树（如果存在）上所有结点的关键词都大于根节点的关键词
- 左子树和右子树同时也满足以上二叉搜索树的性质

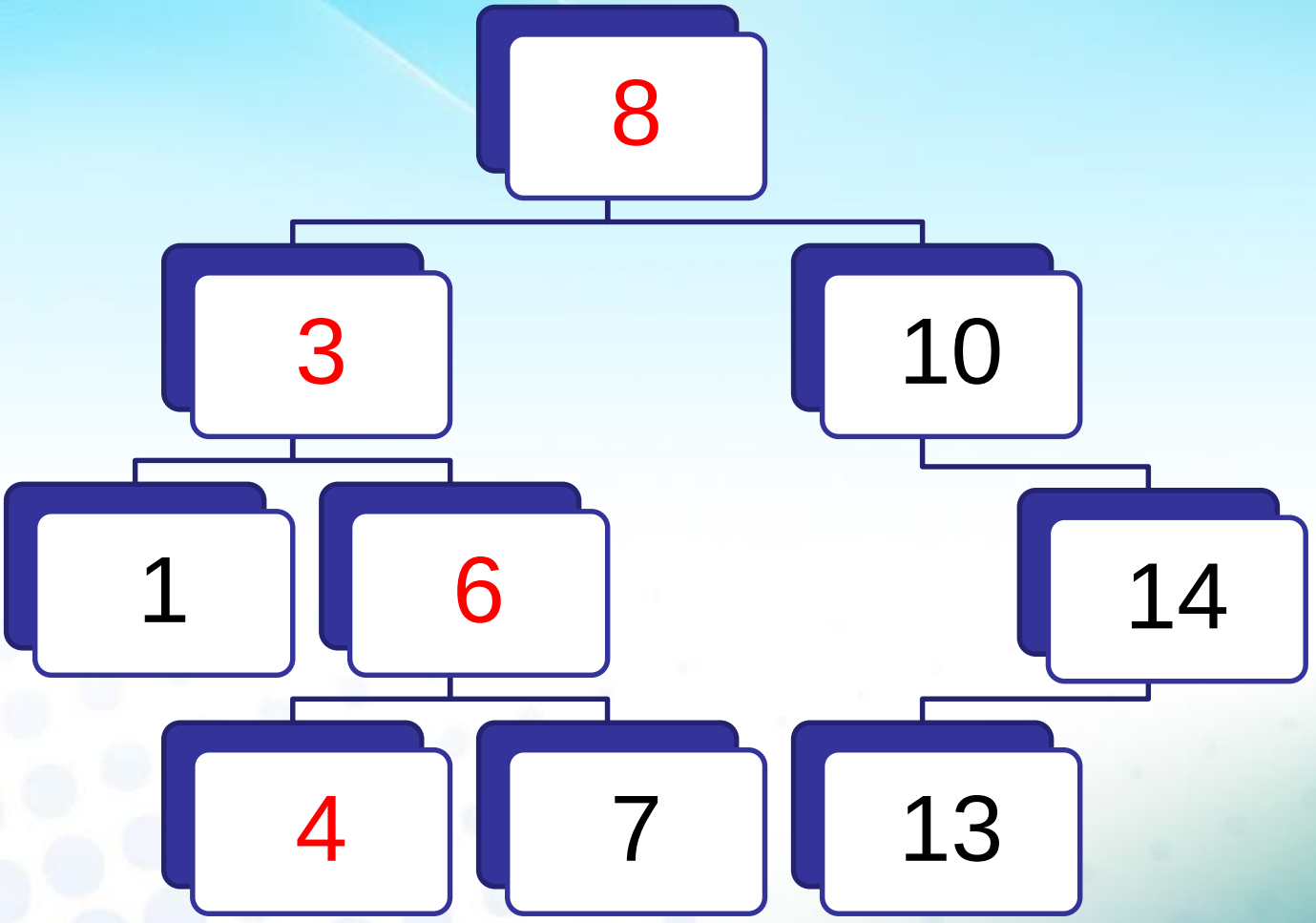
二叉搜索树的例子



二叉搜索树的查询

- 在二叉搜索树中查询的过程可以概括为
  - 如果查询关键词等于当前结点的关键词，则宣布查找成功
  - 如果查询关键词小于当前结点的关键词，则查找其左子树
  - 如果查询关键词大于当前结点的关键词，则查找其右子树
  - 如果已没有儿子节点，则宣布查找失败
- 这样查找次数最大不会超过树的深度

查询的例子



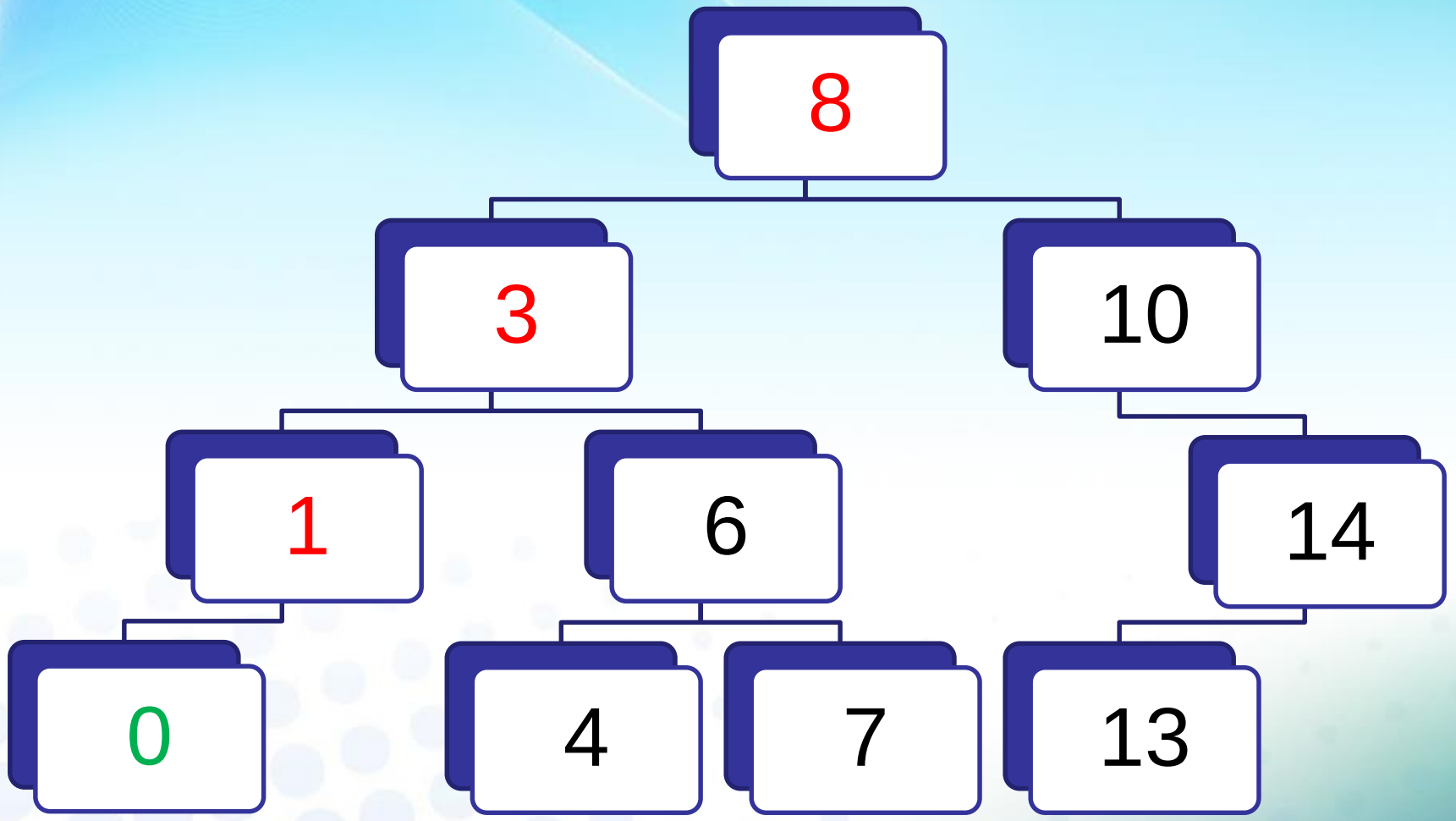
查询的伪代码

```
Tree-Search(x, k)
if x = NIL
        then return “not found”
if k = key[x]
        then return x
if k < key[x]
        then return Tree-Search(left[x], k)
        else return Tree-Search(right[x], k)
```

二叉搜索树的插入

- 能够动态添加数据是二叉搜索树的主要特性，由于二叉搜索树的结构有一定的要求，插入的过程应保持二叉搜索树的性质
- 插入的过程与查询类似，可以理解为对待插入的关键词首先做一次查询，然后再将这个元素插入到搜索操作停止的地方，因为查询在此中断意味着这里就是元素“应该”所在的位置

插入的例子



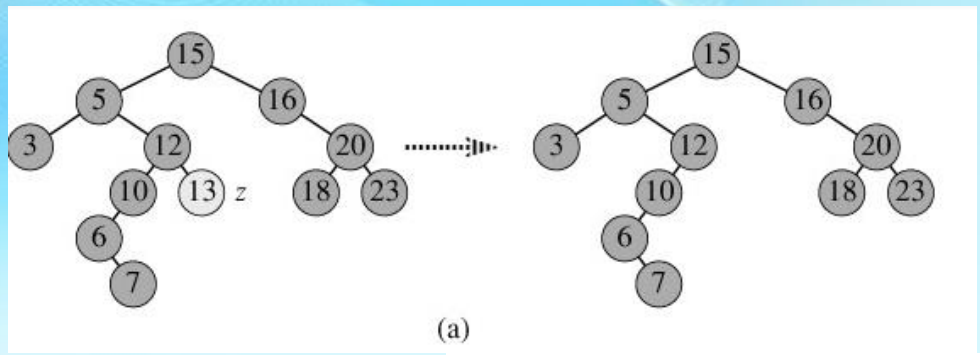
插入的伪代码

```
Tree-Insert(T, z)
y ← NIL
x ← root[T]
while (x ≠ NIL) do
        y ← x
        if (key[z]<key[x])
                then x ← left[x]
                else x ← right[x]
parent[z] ← y
if (key[z]<key[y])
        then left[y] ← z
        else right[y] ← z
```

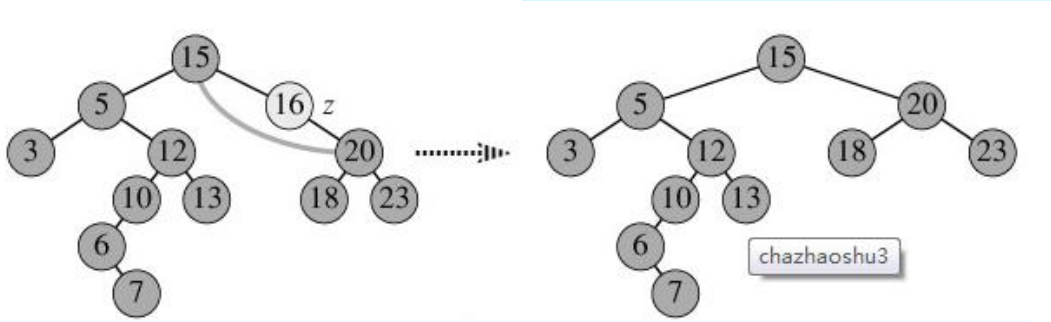
二叉搜索树的删除

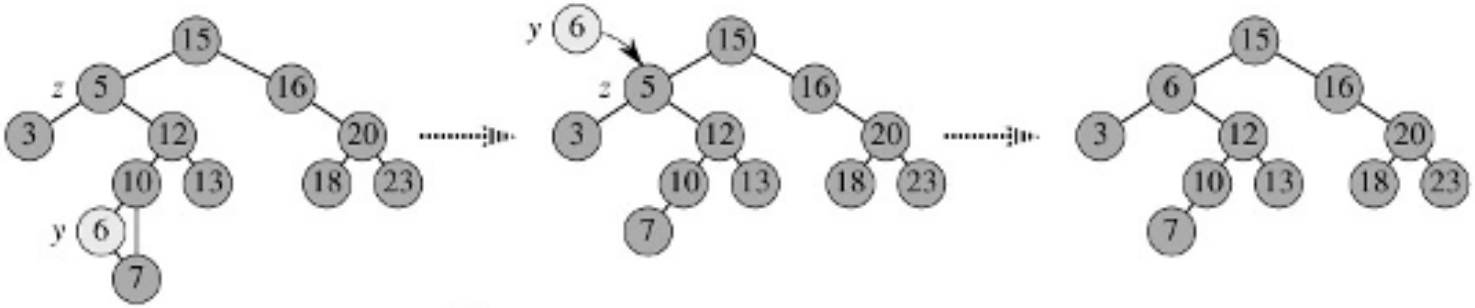
- 二叉搜索树的删除较为复杂，因为往往需要在树中找到一个能够“代替”被删除结点的元素
- 涉及到二叉树中结点的前驱（successor）和后继（predecessor）的概念
- 根据拟删除结点分情况讨论
  - 叶子结点
  - 只有单个分支
  - 同时含有两个子结点

二叉搜索树的删除


15
5
16
3
12
20
10
13
z
18
23
6
7

(a)


15
5
16
z
3
12
20
10
13
18
23
6
7
chazhaoshu3


15
z
5
16
3
12
20
10
13
18
23
y
6
7

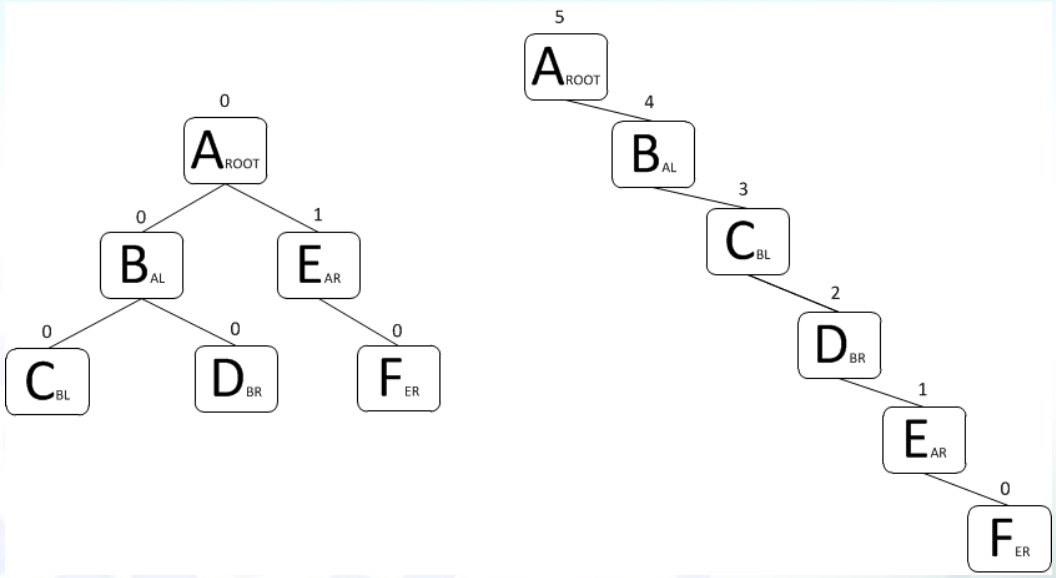
(c)

进阶内容

- 一般来说，由于二叉搜索树会在每一步都“忽略”两棵子树中的一棵，与二分搜索的情况类似，因此执行效率是很高的
- 但也存在着这样的情况：整棵二叉搜索树可能会退化为一条长长的线性链。此时，树的深度即为结点的总数，这样搜索比较的次数就会从$\log_2 n$的级别退化为n的级别，其中n为结点总数

进阶内容

- 比如在插入的序列完全有序时，就可能会造成下图右侧的退化情况，我们希望能够将其转化为左侧的平衡情形



进阶内容

- 一些能够保证平衡性的数据结构
  - 红黑树
  - AVL树
  - 伸展树（splay tree）
  - Treap
- 它们共同的思想是除了二叉树性质之外，同时使数据结构保持另外的某种性质，从而维护其平衡性

参考文献

- 本章介绍的内容以及其它更为详细的说明和讨论可以参考以下文献
  - 《数据结构——用面向对象方法与C++语言描述》
  - 《算法导论》第10、12、13章

Thank you

Q&A