

3.1 数据结构和算法

- 基本概念
- 常见数据结构
- 常见算法
- 其他数据结构与算法



目录

1

基本概念

2

常见数据结构

3

常见算法

4

其他数据结构与
算法

数据结构

- **数据结构**[データ構造]是计算机中存储、组织和管理数据的形式。
- 同样的数据，我们可以用不同方式进行存储。选择数据结构时，最重要的是考虑以下各方面要素：
 1. 数据的**存储方式**是什么？比如，它是有序的还是无序的？各个不同的数据之间是没有关联的还是有关联的？
 2. 我们可以使用哪些**操作（功能）**？比如，我们能检查某个数据是否已被存储吗？我们能删除某个数据吗？我们能对数据排序吗？
 3. 以上这些操作**效率**如何？即，它们分别要花费多少**时间**和**空间**资源？

接次页 

Example

存储一个班级里的学生档案，考虑以下两种方法：**第一种**是直接把所有文件随意摆在桌子上；**另一种**是把文件按学号顺序放在文件夹里。

1. 存储方式：**第一种**方法各个文件之间没有关系；**第二种**方法文件按顺序存储。
2. 操作：都可以存入文件、取出文件、检索文件等。
3. 效率：**第一种**方法存入文件的速度较快，但检索较慢；**第二种**方法存入的速度较慢，但检索速度较快。

抽象数据类型

- **抽象数据类型**[抽象データ型]只考虑数据的**存储方式**和**操作类型**。
不需要确定具体实现方法和操作的效率。
- 当我们决定了抽象类型之后，它可能通过多种**数据结构**进行实现。我们可以通过比较这些数据结构实现操作的**效率**来决定选择哪一个数据结构。
- 决定数据结构的一般顺序：
 1. 根据需求的功能，确定使用哪种**抽象数据类型**。
 2. 确认哪些**数据结构**可以实现该抽象类型。
 3. 根据对实际操作效率的需求，确定使用哪一种**数据结构**。

接次页 

Example

我们想要保存博客网站上一个用户的所有博客。

1. 抽象数据类型：我们需要博客按时间顺序存储。我们需要存入最新博客、浏览任意博客、删除博客等功能。我们发现“列表”这一类型可以实现这些功能。
2. 数据结构候选：我们发现数组和链表可以实现列表。
3. 决定数据结构：我们发现用户经常使用浏览博客的操作。数组在这一操作上比链表的效率更高，因此我们最终决定使用数组。

算法

- **算法**[アルゴリズム]是用于解决某个问题的一系列操作步骤的描述。
- 从广义上说，只要是对解决问题的具体步骤的描述都可以称之为算法，比如菜谱、电器的安装说明书等。我们一般说的算法特指计算机程序解决问题的步骤。
- **算法**和**程序**的区别：算法只需要**抽象**地描述解决问题的**步骤**，而程序需要**具体**将这些步骤用对应语言的**代码**实现。
- 因此，算法可以**无关具体的语言**；同一个算法可以由多种不同语言实现。
- 算法和数据结构息息相关：选择合适的数据结构可以提高算法的效率；数据结构本身的操作又需要设计一些算法来实现。

算法的评价标准

- 解决同一个问题，可以有多种不同的方法。我们如何比较不同算法的优劣呢？
- 在计算机科学中，一般作为算法评价标准的是它需要消耗多少**资源**。通常主要考虑两种资源：**时间资源**和**空间资源**。
- 时间资源的评价指标是**时间复杂度**[時間計算量]：简单来说，时间复杂度越高，算法运行就越**慢**。
- 空间资源的评价指标是**空间复杂度**[空間計算量]：空间复杂度越高，算法运行使用的存储空间（一般就是内存）就越**多**。
- 以上的指标也可用于衡量数据结构的**操作**，因为它们也是根据某个算法实现的。

复杂度的常见级别

- 下表列出了一些复杂度的常见级别（由低到高）。为了便于查询资料选择数据结构或算法，只要大体知道这些级别的优劣关系，不需要定量理解它们。

名称	记法	常见写法
常数级别	$O(1)$	$O(1)$
对数级别	$O(\log n)$	$O(\log n)$ 、 $O(\log(n))$
线性级别	$O(n)$	$O(n)$
线性对数级别	$O(n \log n)$	$O(n \log n)$ 、 $O(n \log(n))$
平方级别	$O(n^2)$	$O(n^2)$
多项式级别	$O(n^c)$	$O(n^c)$ 、 $O(n^k)$
指数级别	$O(2^n)$	$O(2^n)$ 、 $O(c^n)$

Q & A

Question and answer

Coffee ☕ Break

复杂度与大 O 符号

为什么要用复杂度来评价算法？算法不依存于某种特定的语言和机器，这意味着我们很难直接用实际运行时间（空间）或者代码运行量来评定算法。同时，即使算法在某些简单的测试情况表现良好，也不能保证当情况变得更复杂时还能保持优越性。

复杂度实际评价的不是单纯的资源消耗量，而是消耗量随问题规模变大时**变化**的速率。换言之，复杂度低的算法，即使问题规模变得很大，花费的资源也不会增加太多。

算法具体的复杂度通常非常难以计算，而且结果十分复杂，不便比较。因此，我们一般计算的都是对复杂度大概估计，使用统计数学中的渐进符号（**大 O 符号**）来表示。

目录

1

基本概念

2

常见数据结构

3

常见算法

4

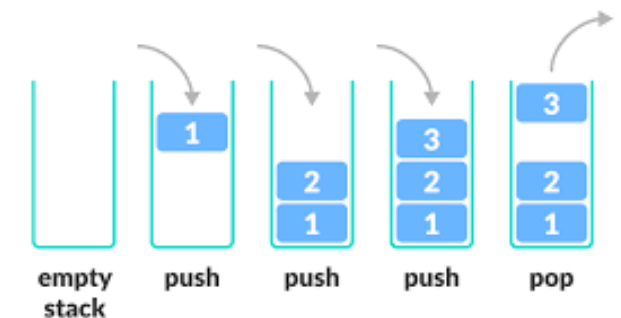
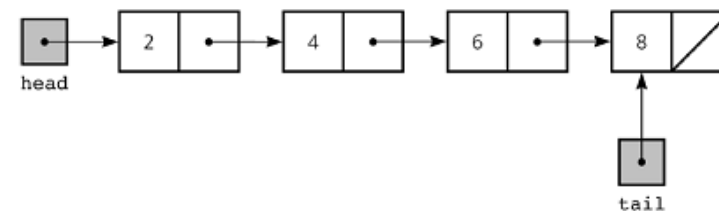
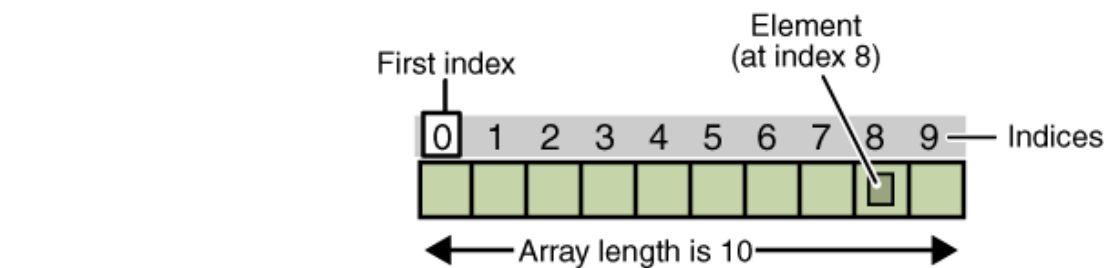
其他数据结构与
算法

常见数据结构与抽象类型

- 在实际的开发过程中，我们用到的大部分数据结构都有现成的实现。因此，我们只需要理解它们的功能和大体的效率，在设计系统时知道如何**选择**即可。

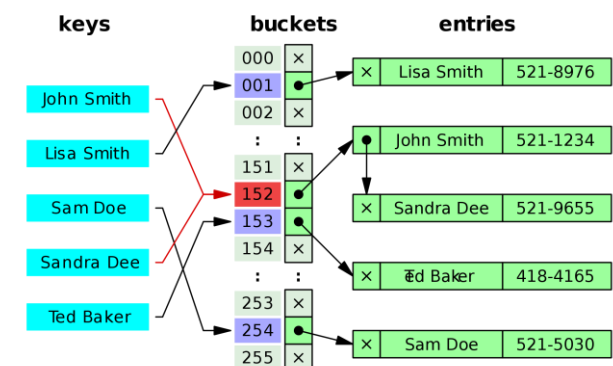
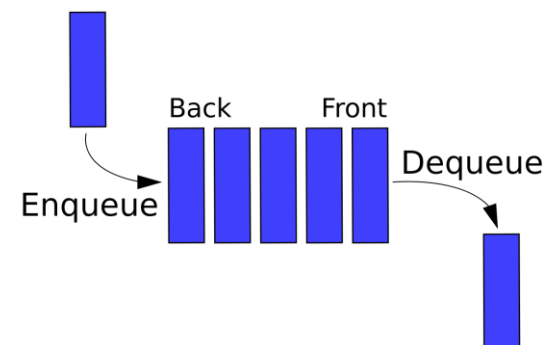
- 常见的数据结构有：

- 数组；
- 链表；
- 图、树。



- 常见的抽象类型有：

- 列表、栈、队列；
- 集合；
- 关联数组。

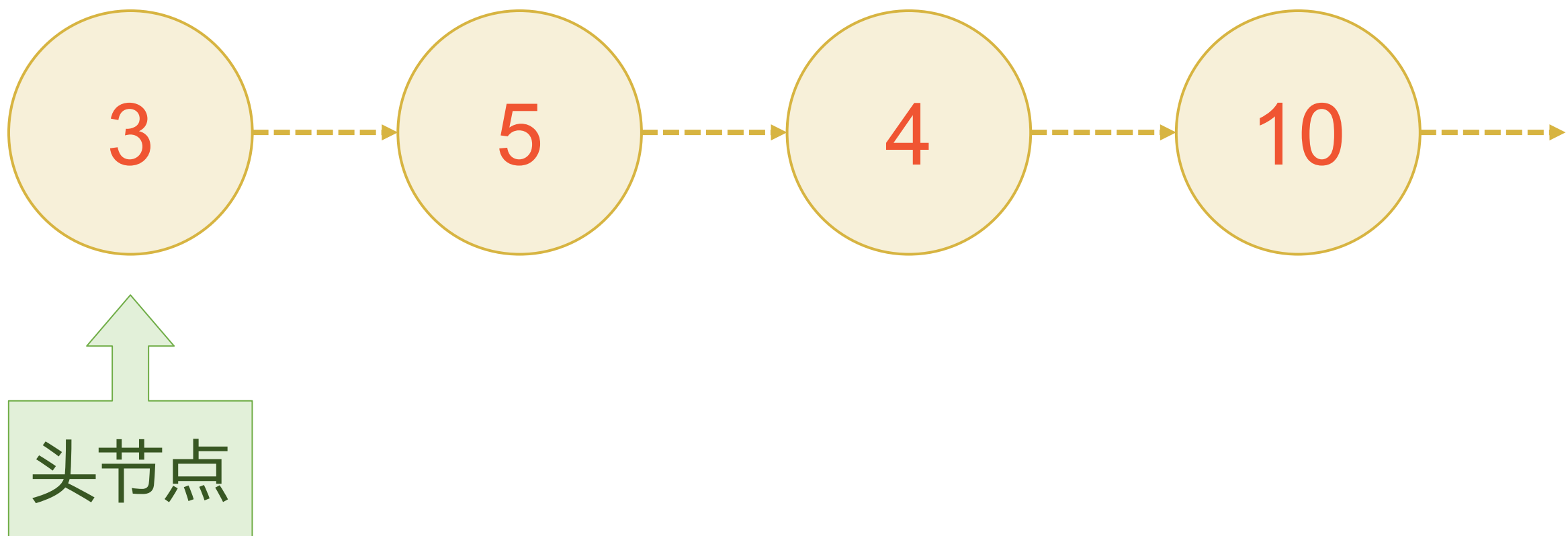


数组

- **数组**[配列] (Array) 是最为基本的数据结构之一。我们已经在 Java 中熟悉了它的存储方式和功能。让我们一起做一个简单概括：
 - 存储方式：在数组中，所有数据按顺序存储。每一个数据有一个**整数**索引。
 - 获取操作：根据索引 i 获得第 i 个的数据的值。
 - 设置操作：根据索引 i 改变第 i 个的数据的值。
- 对计算机语言来说，数组的实现方式较为简单：把所有数据按顺序连续地存储进内存即可。所以大部分语言中数组都是基本类型，语法简便。

链表

- **链表**[連結リスト] (Linked List) 是另一种比较基本的数据结构。和数组一样，它也是用来按顺序存储数据的，但存储方式略有不同：链表里的数据存在一个个**节点**[ノード]里，每个节点只能够访问下一个数据的节点：



链表的存储方式和操作

- 存储方式：在链表中，所有数据有序存储。但和数组不同，数据没有索引，但每一个数据都知道下一个数据存在哪里。
- 获取开头操作：获得链表的第一个节点。
- 迭代操作：获取某个节点的下一个节点。
- 获取、设置操作：获取某个节点中存的数据，或修改数据的值。
- 插入操作：在某个节点的后面新增一个节点，插入某个数据。
- 删除操作：把某个节点从链表中删除。

链表的实现

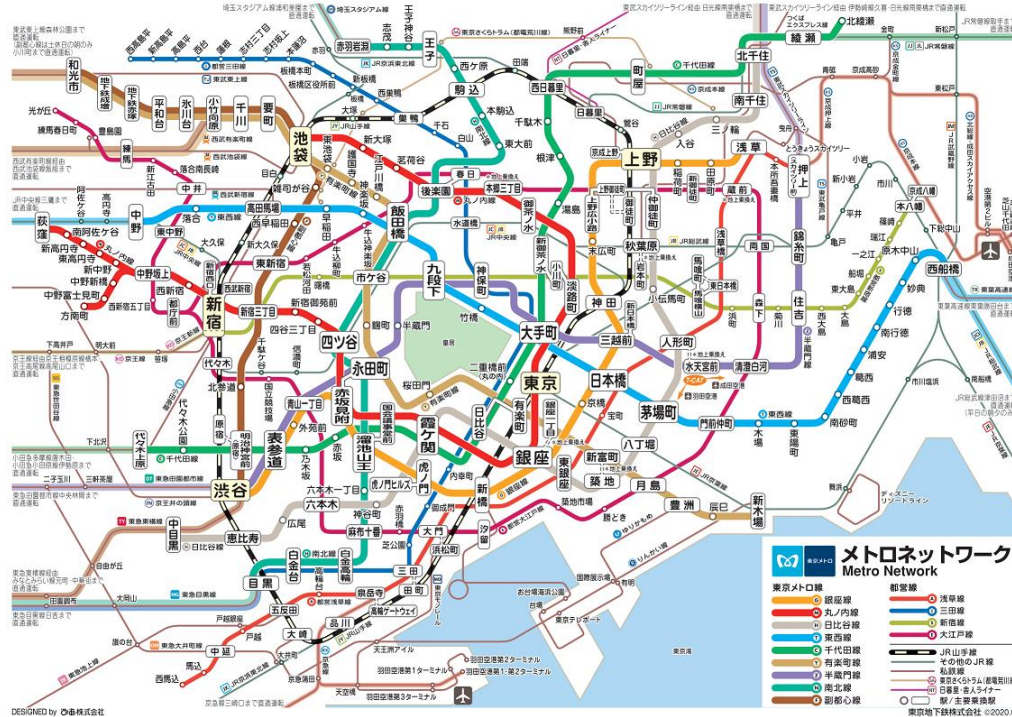
- 很多语言不直接支持链表，但可以非常简单地用基础语法实现链表。比如 C/C++ 中可以用 *指针* 记录下一个节点的地址来实现。
- Java 中没有指针，但有类似的 *引用*。我们可以把数据和下一个节点的对象（引用）存在每个节点中。



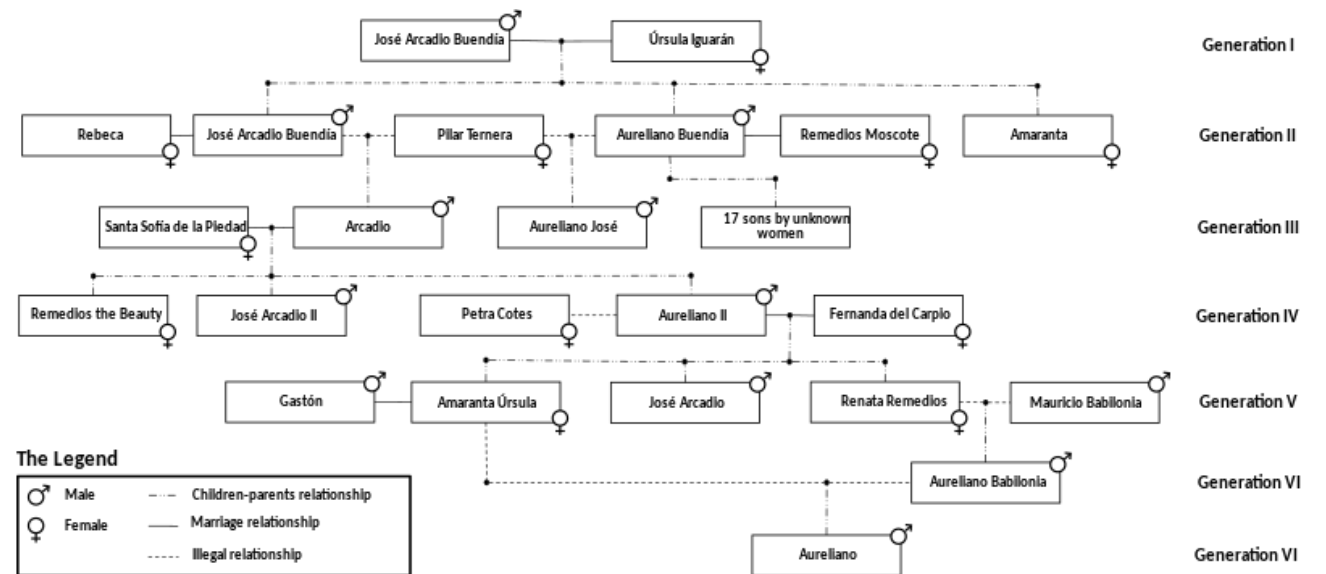
- 实际使用到的链表常常是原始版链表的变种，比如双向链表、循环链表等。

图

- **图[グラフ] (Graph)** 是一种很常用的数据结构。图中的数据也存储在一个个**节点[ノード]**里，但每个节点可以通过**边[エッジ]**和多个其它节点连接。生活中很多数据都可以用图的形式表达：



地铁路线图



人物关系图

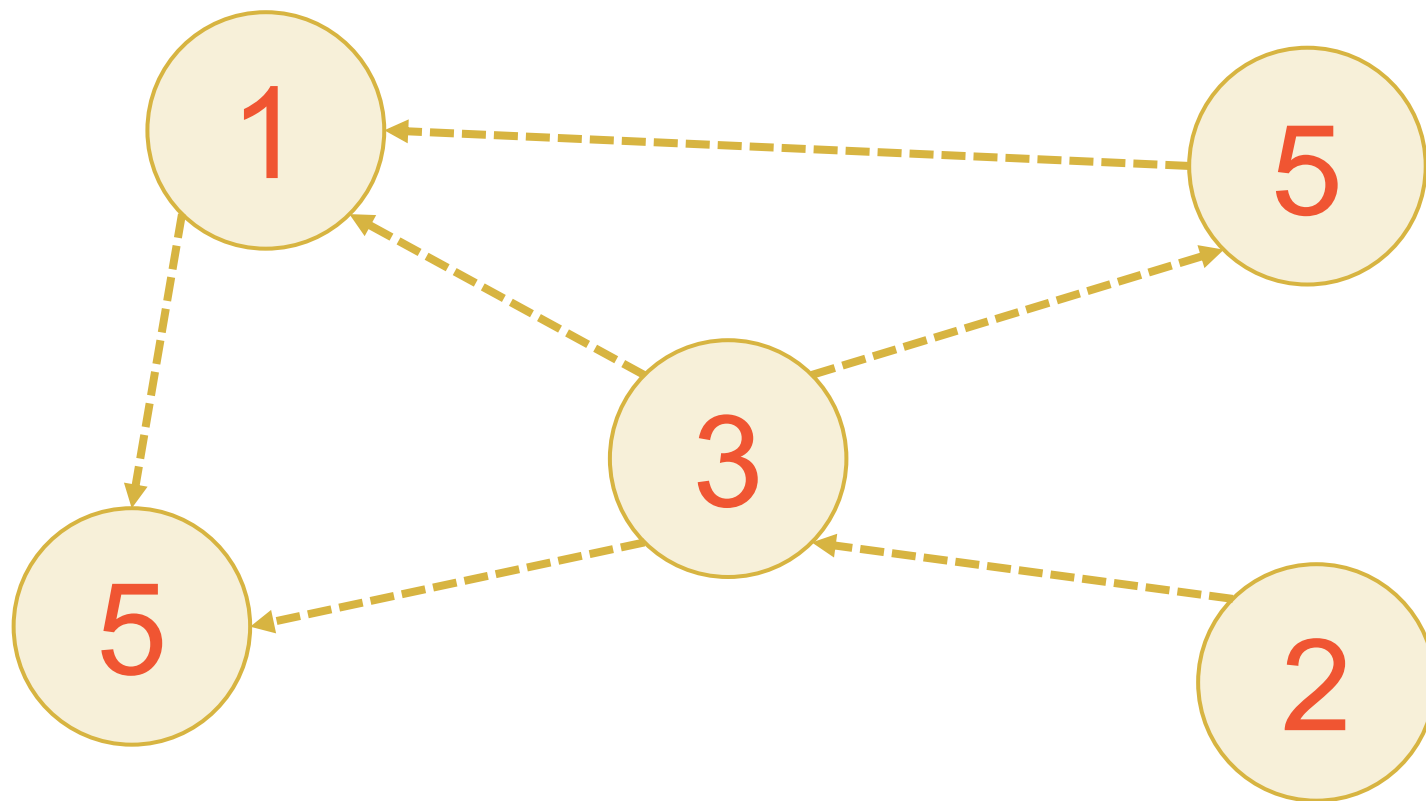
- 注意区分**图 (Graph)** 和**图片 (Picture、Image)**。

图的存储方式和操作

- 存储方式：在图中，数据之间没有顺序关系。数据存在一个个节点里，每个节点可以访问与自己相邻的节点（“邻居”）。
- 获取相邻节点操作：获得某个节点的“邻居”。
- 获取、设置操作：获取某个节点中存的数据，或修改数据的值。
- 增加、删除节点操作：增加或删除某个节点。
- 增加、删除边操作：增加或删除边，改变节点间的关系。

图的实现

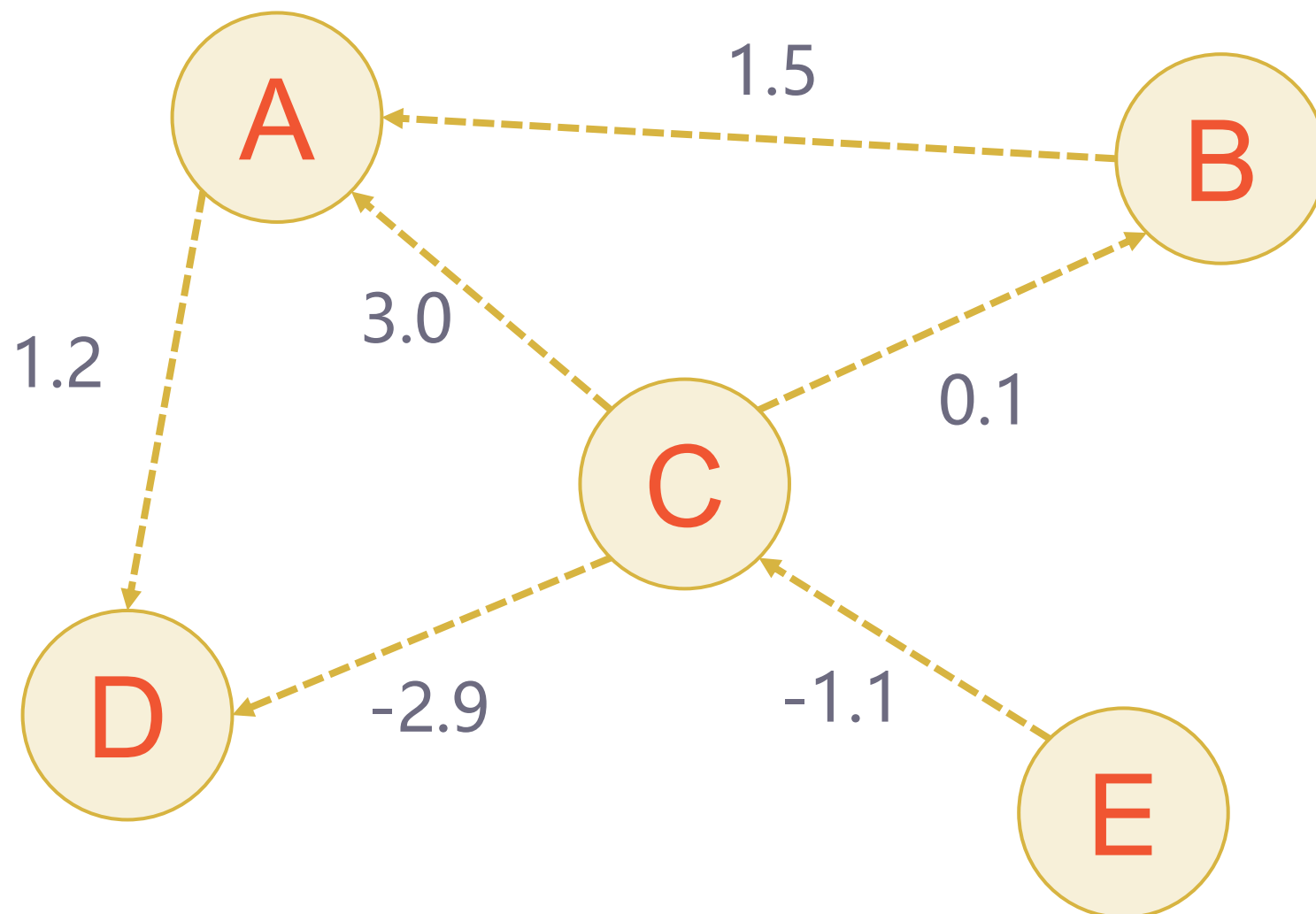
- 在 Java 中，同样可以通过对象来实现图的结构：每一个节点记录数据和与它相邻的节点。



- 从底层数据结构的角度说，这种方法类似邻接表的形式。除此之外，还有邻接矩阵、关联矩阵等形式，适合存储边较为密集的图。

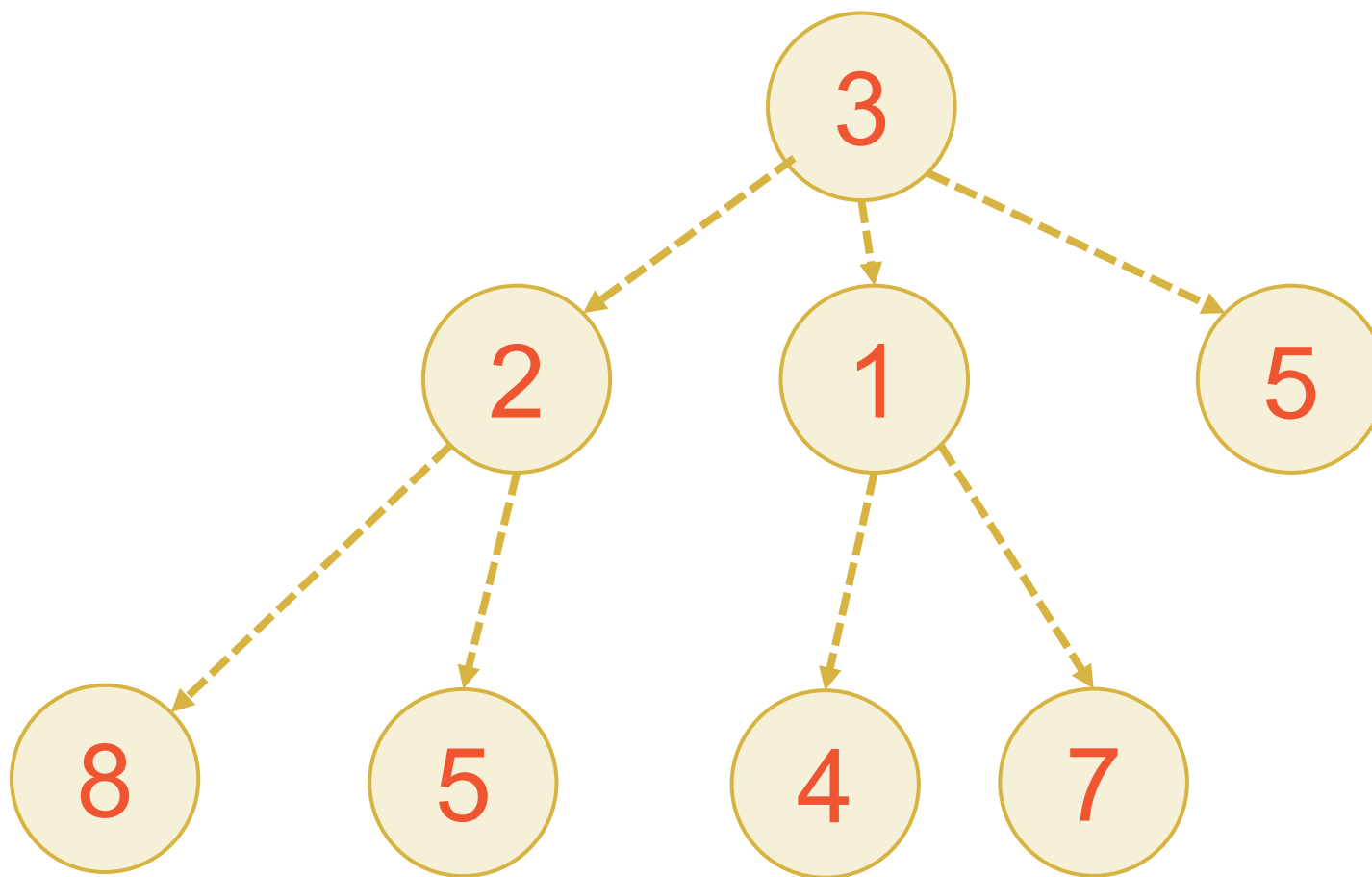
网络

- 还可以在图中每条边上记录一个数据，这种图被称为**加权图**或**网络**[ネットワーク]：



树

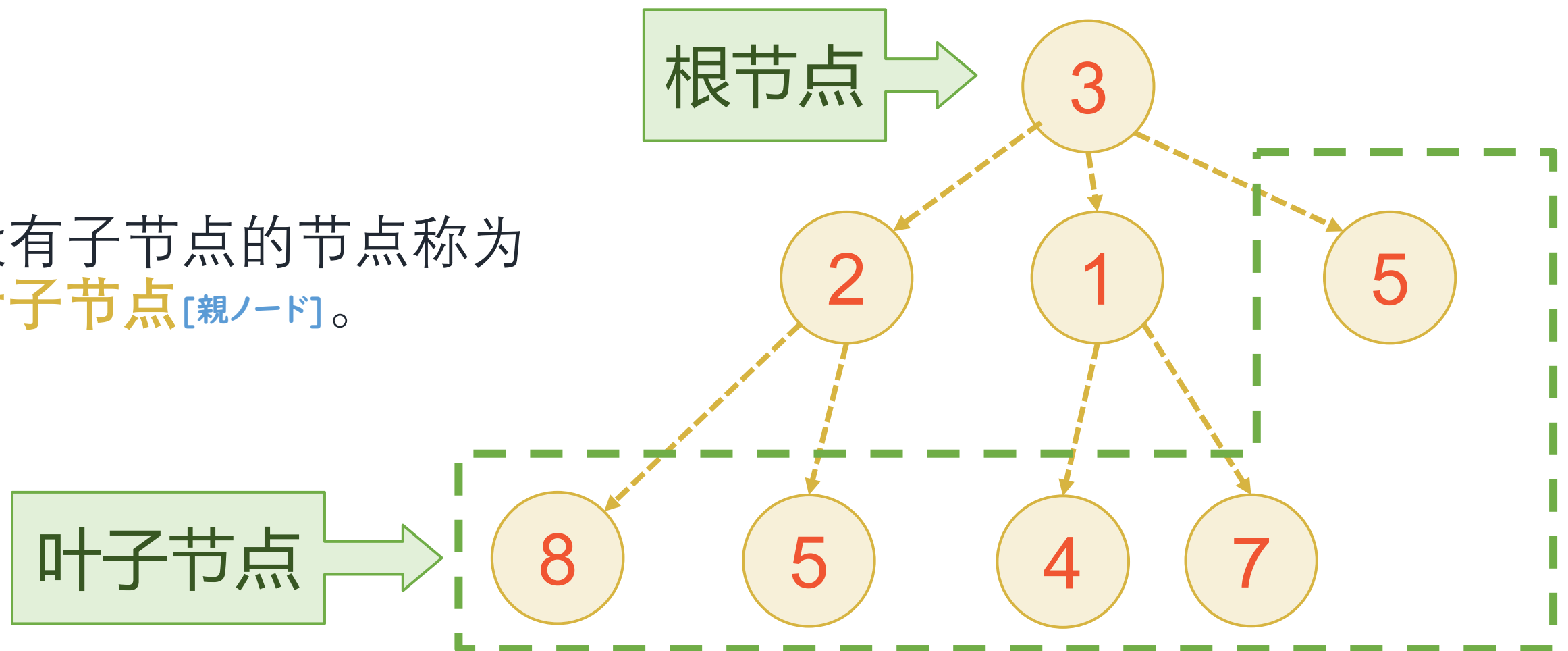
- **树[木] (Tree)** 是一种特殊的图。树中不会有回路。在实际的数据结构中，一般将所有节点分成几个层次，每个节点都连向下一层的一些节点：



树的相关概念

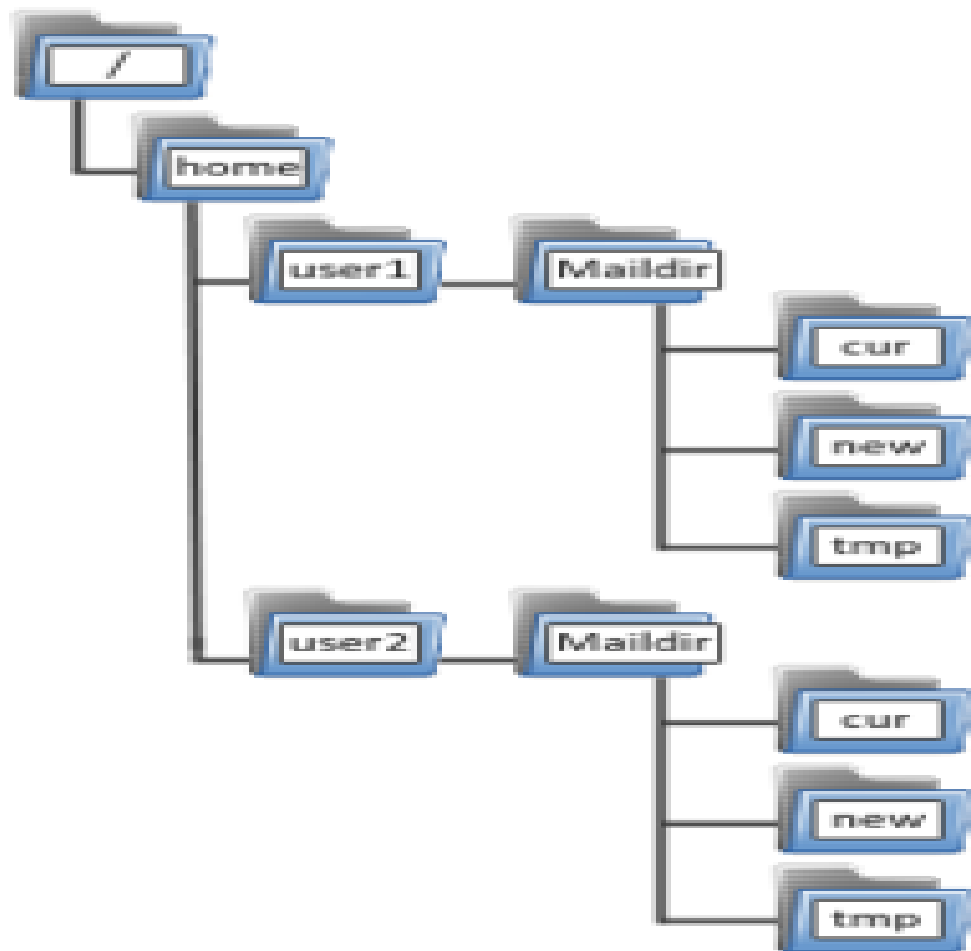
- 最上面一层的节点被称为**根节点** [根ノード]，一棵树只有一个。
- 最每个节点连向的节点被称为它的**子节点** [子ノード]。而它就是这些节点的**父节点** [親ノード]。每个节点只有一个父节点。

- 没有子节点的节点称为**叶子节点** [親ノード]。



树的应用

- 虽然我们暂时无需考虑如何实装树，但其实生活中很多地方我们已经用到了类似的结构，比如文件夹系统：



Tips 💡

你有听说过“根目录”、“子文件夹”等说法吗？现在你理解它们名字的含义了吗？

- 想想还有什么地方用到了树的结构？

Q & A

Question and answer

列表

- **列表**[リスト] (List) 是最常用到的抽象数据类型之一，表示简单的有序数据。
 - 存储方式：列表中的数据应**有序**存储。类似数组，每个数据有应一个**整数索引**对应它是第几个数据。
 - 获取、设置操作：根据索引获取或设置某个位置上的数据。
 - 任意插入、删除操作：往列表的任意位置（根据索引）添加某个数据或删除某个索引上的数据。
 - 特殊插入、删除操作：比如在表头或表尾处添加、删除数据。
 - 查找操作：找到某个数据在列表中的索引；或者确认其不在表中。

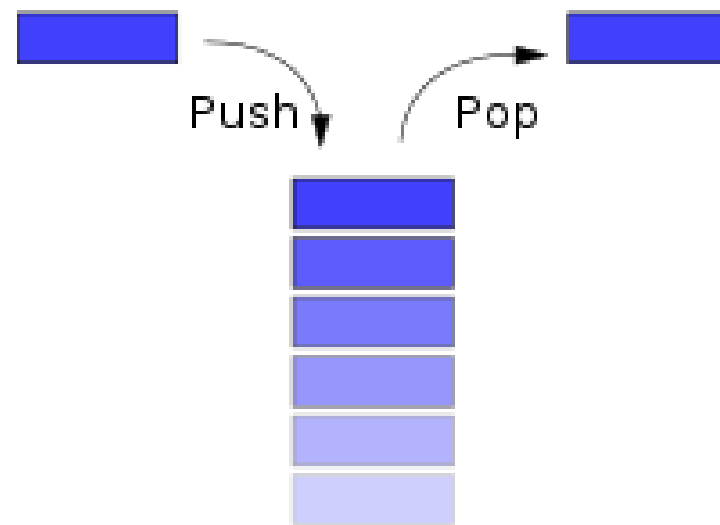
列表的实现

- 列表最常见的实现有两种：使用**数组**或**链表**。建议记住这两种方法分别在什么操作上效率更高，以便选择：

操作	用数组实现	用链表实现
获取、设置	$O(1)$	$O(n)$
任意插入、删除	$O(n)$	$O(n)$
特殊插入、删除	$O(n)$	$O(1)$
查找	$O(n)$	$O(n)$

栈

- **栈**[スタック] (Stack) 也是按顺序保存数据的一种结构。但和列表不同，栈只能从一端 (**栈顶**[トップ]) 访问、插入或删除数据：



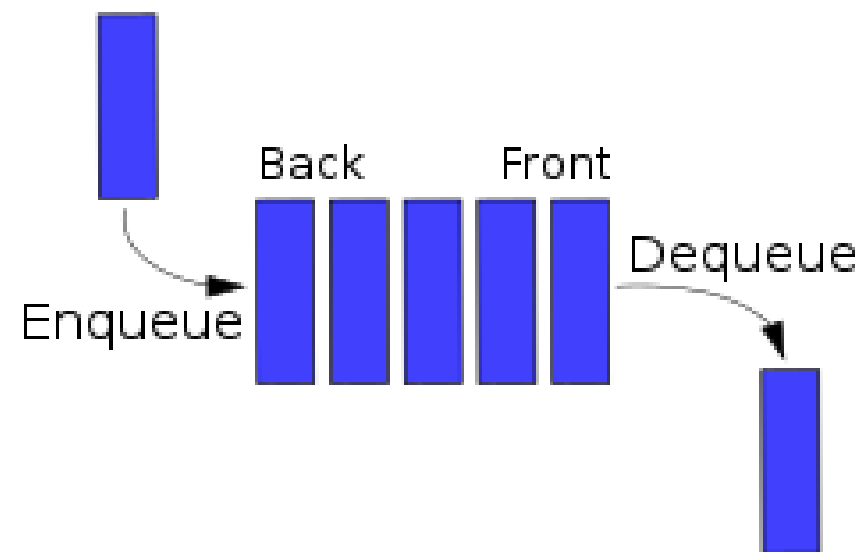
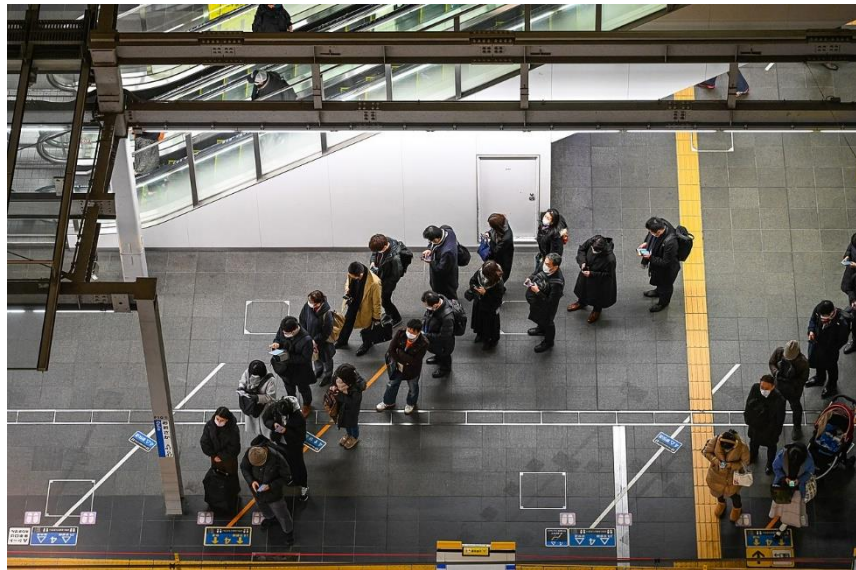
- 这使得栈中的数据永远是**先入后出**（或后入先出，英文称作 Last In First Out, **LIFO**）的。
- 中文里有时也将栈译作**堆栈**，注意它和另一个数据结构**堆**没有关系。

栈的操作和实现

- 栈的操作：
 - 存储方式：列表中的数据应**有序**存储。
 - 压栈操作（**push**）：向栈顶插入一个新的数据。
 - 弹栈操作（**pop**）：取出栈顶数据使用（该数据会从栈中被删除）。
 - 预览操作（**peek**）：获得栈顶数据（但不删除）。
- 栈一般可以用数组或链表简单地实现。
- 一般的实装中，以上 3 种操作的复杂度都是 $O(1)$ 。

队列

- **队列**_[キュー] (Queue) 也是按顺序保存数据的一种结构。队列只能从一端 (**队尾**_[末尾]) 插入数据, 另一端 (**队头**_[先頭]) 删除或查看数据:



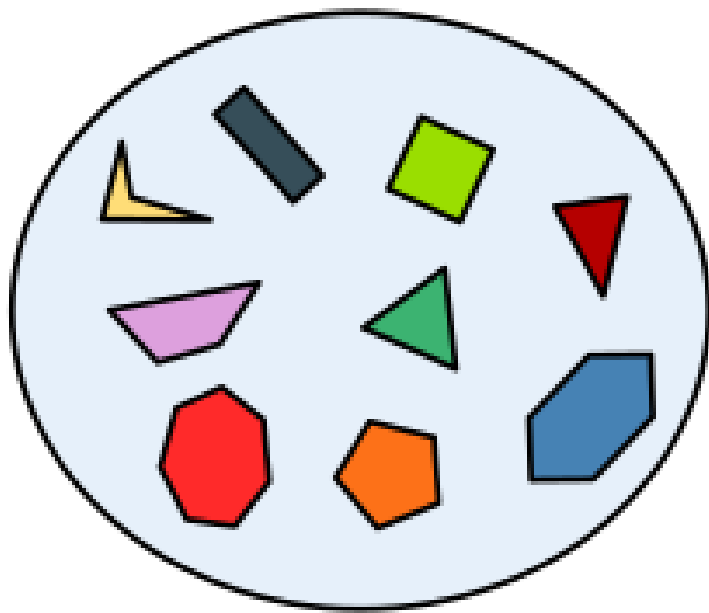
- 和栈相反, 队列的数据永远是**先入先出** (或后入后出, 英文称作First In First Out, **FIFO**) 的。

队列的操作和实现

- 队列的操作：
 - 存储方式：队列中的数据应**有序**存储。
 - 入队操作（**enqueue**）：向队尾插入一个新的数据。
 - 出队操作（**dequeue**）：取出队头数据使用（该数据会从队列中被删除）。
 - 预览操作（**peek**）：获得队头数据（但不删除）。
- 队列也可以用数组或链表简单地实现。
- 一般的实装中，以上 3 种操作的复杂度都是 $O(1)$ 。

集合

- **集合** (Set) 是一种**不按顺序**保存数据的抽象类型。和数学上的集合一样，集合中**不会有重复的元素**：



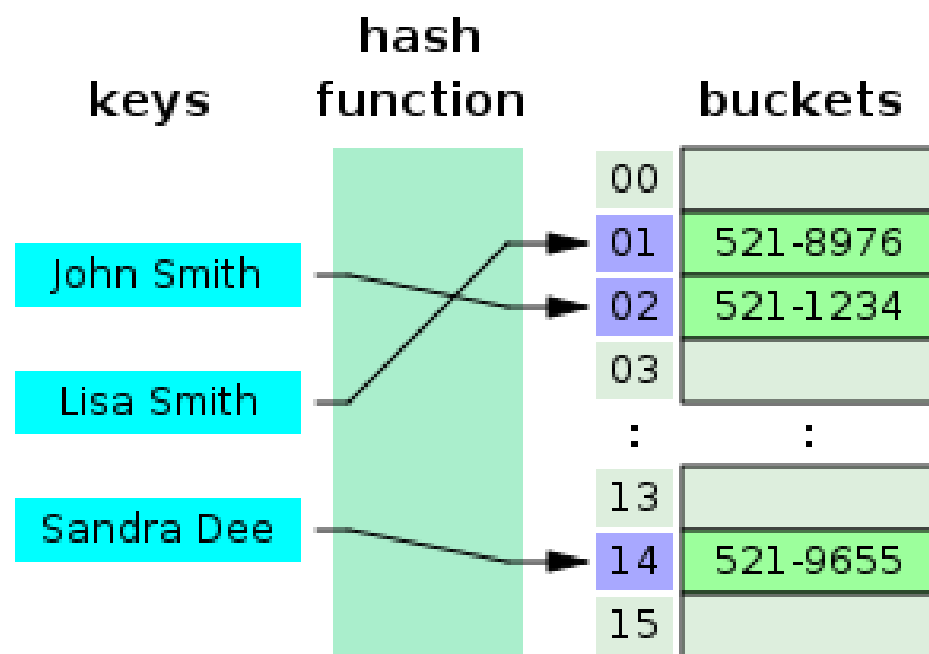
$\{1, 3, 2, 7\}$

集合的操作和实现

- 集合的操作：
 - 存储方式：集合中的数据可以**无序**存储，并且不应有重复的数据。
 - 添加操作：向集合中放入一个新的数据。
 - 删除操作：删除一个集合中的数据。
 - 查找操作：确认某个数据是否被包含在集合中。
- 集合的常见实装是**散列集**[ハッシュ集合]（Hash Set）。以上 3 个操作的时间复杂度均为 $O(1)$ 。

关联数组

- **关联数组**^[連想配列]，又称**映射**^[マップ]（Map）或**字典**^[辞書]（Dictionary），是一种按照**索引**保存数据的抽象类型。和数组不同的是，关联数组可以用**数字以外的类型**作为索引。比如，存储电话簿的信息，可以使用联系人的名字（字符串类型）当作索引：



- 其中，用于索引的数据被称为**键**^[キー]，实际被存储的数据被称为**值**。一个键对应一个值，**不会有重复的键**。

关联数组的操作和实现

- 关联数组的操作：
 - 存储方式：关联数组中的数据应按**键值对**的方式进行存储。
 - 获取操作：根据**键（索引）**获得对应的**值（数据）**。
 - 设置操作：修改某个**键**所对应的**值**。
 - 查找操作：确认某个**键**是否在关联数组中。
 - 删除操作：从关联数组中删除某个**键**（的键值对）。
- 关联数组的常见实装是**散列表**[ハッシュテーブル]（Hash Table）。以上操作的时间复杂度均为 $O(1)$ 。

Q & A

Question and answer

Coffee ☕ Break

散列表与散列函数 (1)

散列表的底层实现其实仍然是使用数组：我们只要有办法把任意类型的**键**转换成一个整数，以这个数字作为数组的索引保存对应的**值**即可。

这里的核心问题是如何把一个可能的取值范围很大的数据（比如字符串）转换为指定范围内的整数。散列表的实现重点就是选取一个合适从大型数据到整数的映射，也就是**散列函数**[ハッシュ関数]。

通过散列函数得到的整数就被称为**散列值**[ハッシュ]。散列值就像数据的“指纹”：不同的数据（极大概率）有不同的散列值。

Coffee ☕ Break

散列表与散列函数 (2)

除此之外，散列函数还有一个很重要的特征：它是**不可逆**的，即我们无法通过散列值求出原本的数据。

这种特征使得散列函数在通信安全领域也非常有用。比如，如果直接将网站用户的密码保存在服务器上，被黑客攻击就可能导致所有用户密码的泄露。但如果保存的是用户密码的散列值，那么在用户登录时仍然可以判断密码是否输入正确；同时，即使服务器数据泄露，黑客也无法还原出原本的密码。

除此之外，散列函数还在数据加密、数据检查、加密货币等领域有所运用。

目录

1

基本概念

2

常见数据结构

3

常见算法

4

其他数据结构与
算法

常见算法

- 在实际开发过程中，大部分算法都可以有现成的实现（比如各个数据结构的**操作**）。我们可以先**使用**并熟悉已经实现的常用算法，再尝试**实现**已有的其他算法，最后**设计**自己的算法。
- 接下来介绍一些常见问题及算法：
 1. **数组搜索问题**：遍历查找、二分查找。
 2. **排序问题**：冒泡排序、插入排序、快速排序、归并排序。
 3. **图的搜索问题**：深度优先搜索、广度优先搜索。

列表搜索问题

- **搜索**_[检索] (Search) 问题是指在某个数据结构中查找某个数据或是具有某个特征的数据的问题。
- 同时也能判断数据是否在被查找的数据结构里。

Example ✓

- 在所有学生的数据中找到“张三”的数据。
- 在所有 20 世纪的年份中找到第一个闰年。

- 我们先来考虑比较简单的数据结构，比如在**数组**里面的搜索。

直接查找

- 最简单的方法就是直接**遍历**[走查]整个数组，判断每一个元素是否符合条件。
- 请思考一下，这种算法最好的是什么情况？最差的又是什么情况？分别要花费多长时间？
- 最好情况：第一个元素就是想要的数据。时间复杂度为 $O(1)$ 。
- 最差情况：最后一个元素是想找的数据；或者数组里没有想找的数据。时间复杂度为 $O(n)$ 。

搜索排好序的数组

- 假如数组里的数据是排好序的，我们能否利用这一性质加速搜索过程呢？

Example ✓

- 所有同学都已按身高从矮到高排好序。找出第一个身高超过 1.7 米的同学。
- 电话簿中所有电话号码都已从小到大排好序。判断 012-3456-7890 是否在电话簿里。

二分查找法

- **二分查找法**[二分探索] (Binary Search) 是对有序数组的一种快速搜索算法。
- 方便起见，以下假定数据是从小到大排序的。
- 思路：比较数组正中间的数据和要搜索的数据。如果一样大，那么我们就找到了想搜索的结果；如果数组中间的数据较大，我们就只需要搜索数组的左半边；如果数组中见的数据较小，我们就只需要搜索数组的右半边。在搜索左、右半边的过程中，我们可以再次利用二分查找减少搜索时间。
- 通过这种方法，我们可以减少大量无用的探索时间。时间复杂度可以被优化至 $O(\log n)$ 。

二分查找法的例子

Example ✓

有一个有序数组 {1, 3, 6, 8, 15, 18, 20}。

- 判断 6 在不在数组中。
- 判断 19 在不在数组中。

Try 

BinarySearch.java

Q & A

Question and answer


排序问题

- **排序**_[整列] (Sort) 问题是指将列表中的数据按一定顺序排列的方法。
- 如果是数字，我们可以直接按从小到大或从大到小排列。但其他数据类型也可以排列，只要决定好比较标准。

Example ✓

- 把所有学生按身高从矮到高排列。
 - 把所有公司按平均薪水从高到低排列。
 - 把所有用户名按**字典序**_[辞書順]排列。
- 为了方便起见，以下只考虑把数字从小到大排列的问题。

冒泡排序

- **冒泡排序**[バブルソート] (Bubble Sort) 是一种简单的排序算法。
- 思路：如果数组排好序了，两两相邻的数字应该总是左边比右边小。所以如果我们看到两个相邻数字，左边比右边大的，就将它们交换。重复直到数组被排好序。
- 观看算法演示动画：
 <https://visualgo.net/en/sorting>。

插入排序

- **插入排序**[挿入ソート] (Insertion Sort) 是另一种简单的排序算法。
- 思路：和我们玩扑克游戏时的理牌方式类似。先把所有牌不按顺序地放在右手边。把一张牌放到左手边，再从右手边把牌一张一张插入左手边正确的位置，始终保持左手边的牌是按顺序排列的。
- 观看算法演示动画。

快速排序

- **快速排序**[クイックソート] (Quicksort) 是一种实现较为复杂的排序算法。
- 思路：快速排序本身的思路很简单：
 1. 随便取一个数 k ，把所有数分成比 k 大的和比 k 小的。
 2. 把比 k 小的数放到 k 左边，用快速排序把它们排好序。
 3. 把比 k 大的数放到 k 右边，用快速排序把它们排好序。
- 注意到了吗？这就是之前提到的**递归**的思想。它的**终止条件**是什么？
- 观看算法演示动画。

归并排序

- **归并排序**[マージソート] (Merge Sort) 是另一种较为复杂的排序算法。
- 思路：同样使用递归思想：
 1. 将数组从正中间分为两半。
 2. 把左半边用归并排序排好序。
 3. 把右半边用归并排序排好序。
 4. 把两个排好的数组合并成一个完整的、排好序的数组。
- 归并排序效率高的关键是第四步的合并操作（即归并）只需要 $O(n)$ 的时间复杂度。
- 观看算法演示动画。

排序算法的复杂度比较

- 你能想象出前面介绍的各种排序算法的时间复杂度是多少吗？
- 下表列出了这些算法的时间和空间复杂度：

算法	最差时间复杂度	平均时间复杂度	空间复杂度
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$
插入排序	$O(n^2)$	$O(n^2)$	$O(1)$
快速排序	$O(n^2)$	$O(n \log n)$	$O(\log n)$
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n)$

- 事实上，基于比较的排序算法最快也只能是 $O(n \log n)$ 复杂度。

排序算法的使用

- 觉得太复杂了记不住？别担心，实际的开发过程中我们通常只需要用到高效的**快速排序**和**归并排序**算法。
- 我们也不需要记住它们的实现方式：现代编程语言中基本都提供了标准的排序算法，我们只需直接使用（会在下一章讲解）。
- 比如，Java 中提供了标准的列表排序算法，是一种优化过的归并排序。

Q & A

Question and answer

图的搜索问题

- 有时，我们需要在复杂的数据结构中搜索数据，比如图。

Example ✓

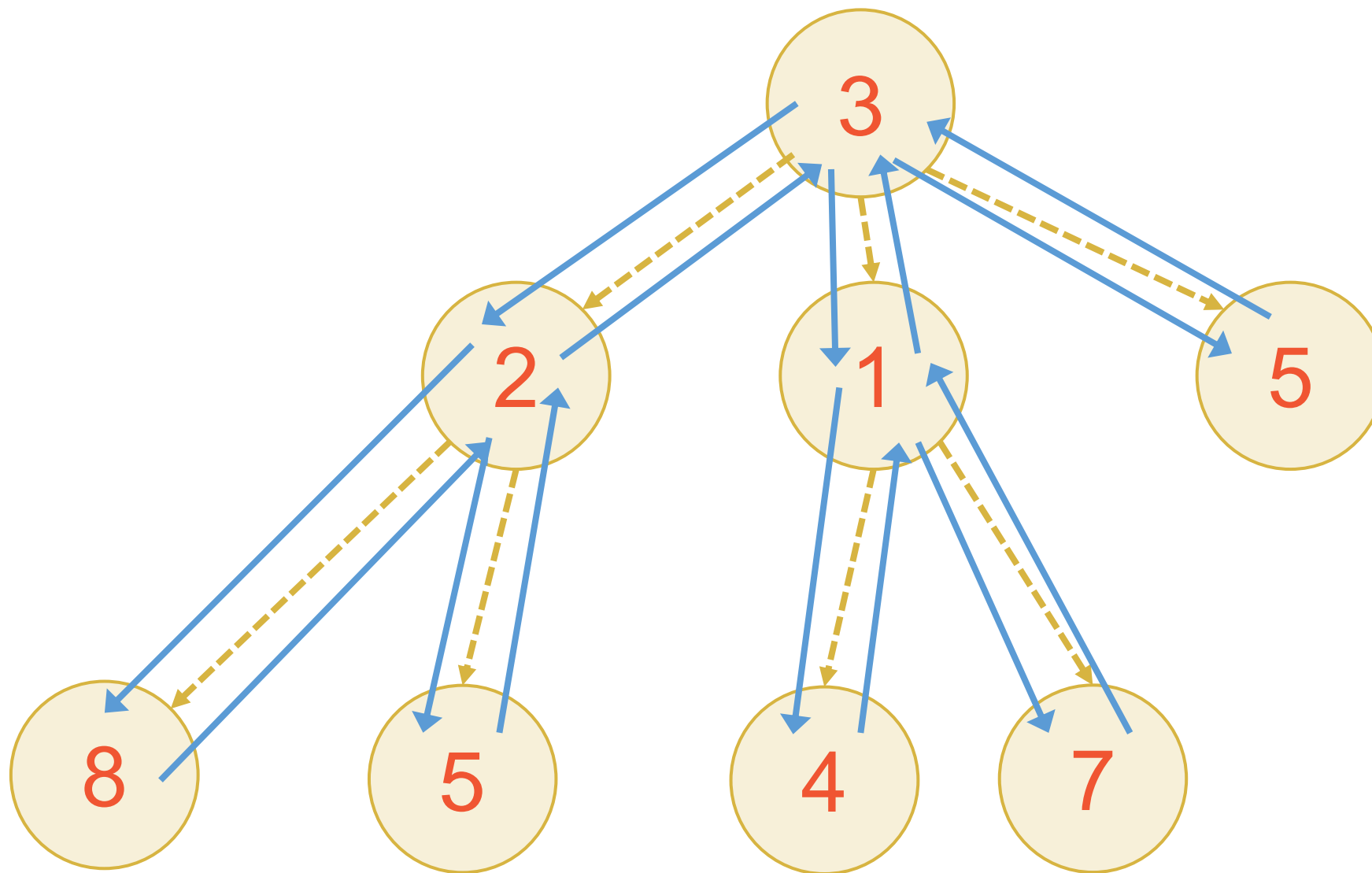
- 从东京站出发，在地铁路线中搜索涩谷站。
- 从用户的计算机出发，在互联网中搜索服务器的计算机。
- 从根目录出发，找到具有指定文件名的文件。

图的搜索算法

- 有两种最基本和常用的图的搜索算法：
 1. **深度优先搜索**[深さ優先探索] (Depth First Search, **DFS**)。
 2. **广度优先搜索**[幅優先探索] (Breadth First Search, **BFS**)。
- 这两种搜索方式都能尽量不重复，不遗漏地遍历每一个图中的节点。
- 我们接下来就以比较简单的图——**树**为例看一下它们的实行过程。其他图的情况与之类似。

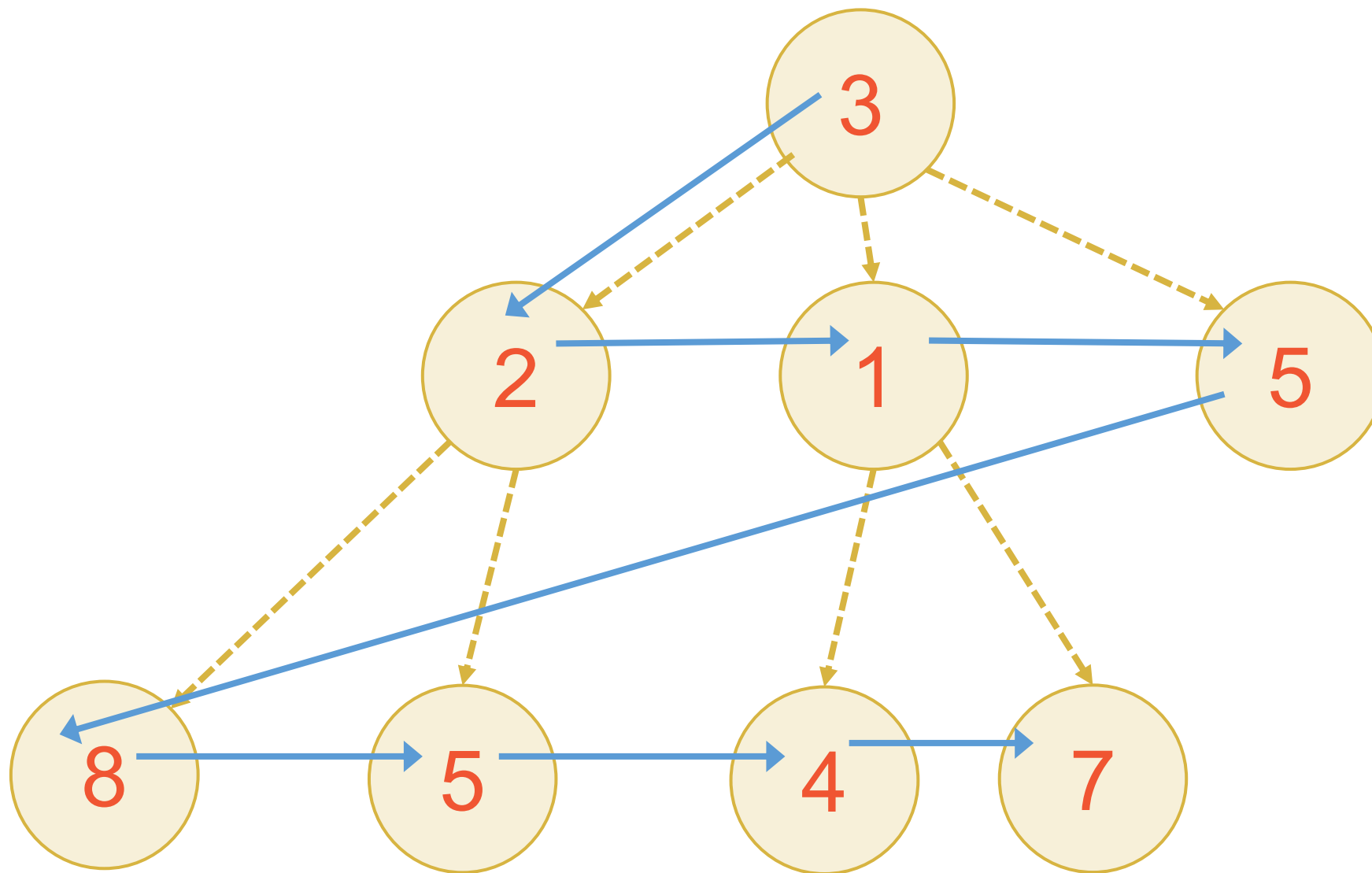
深度优先搜索

- **深度优先搜索**从根出发，尽可能搜索没被搜索过的子节点，如果所有子节点都搜索过了，就回到父节点：



广度优先搜索

- **广度优先搜索**也从根出发，尽可能搜索当前节点的所有子节点，再搜索子节点的所有子节点，再搜索这些节点的所有子节点.....



Q & A

Question and answer

目录

1

基本概念

2

常见数据结构

3

常见算法

4

其他数据结构与
算法

其他数据结构与算法

- 在这一节中，我们介绍一些相对进阶一些的实用数据结构和算法。
- 对于这些数据结构和算法，目前只需对其实现的功能有印象，具体的实现方法和复杂度可以待用到时再查。

二叉查找树

- **二叉树**_[二分木] (Binary Tree) 是一种特殊的树结构：每个节点只有不超过两个子节点。
- **二叉查找树**_[二分探查木] (Binary Search Tree, **BST**) 是一种特殊的二叉树，基于二分搜索的思想。它可以将数据**排好序存储**，并且搜索数据、删除数据及插入新数据的**平均时间复杂度**均为 $O(\log n)$ 。
- **平衡二叉查找树**_[平衡二分探查木] (Balanced BST) 是一种二叉树查找树的优化。普通的二叉搜索树可能因为数据出现**不平衡**的最差情况，导致操作花费 $O(n)$ 的复杂度。通过保证二叉树平衡，可以确保 $O(\log n)$ 的**最差时间复杂度**。有很多不同的优化方法，如 AVL 树、红黑树、伸展树等。
- 二叉查找树多被用于存储需要频繁变化的有序数据。

堆和优先队列

- **堆**[ヒープ] (Heap) 是一种特殊的二叉树：每个节点的数据比它的所有子节点都大（最大堆） / 小（最小堆）。一般用到的堆都是二叉堆。
- **优先队列**[優先度付きキュー] (Priority Queue) 是一种抽象数据类型，一般由**堆**实现。优先队列中的数据**排好序存储**，支持以下操作：
 1. 插入一个数据至优先队列。用堆实现的复杂度为 $O(\log n)$ 。
 2. 取出最大的数据（并删除）。用堆实现的复杂度为 $O(\log n)$ 。
 3. 查看最大的数据（不删除）。用堆实现的复杂度为 $O(1)$ 。
- 优先队列可用于存储一些需要按优先级顺序使用的数据。

特殊的集合

- **位集合**[ビット集合] (Bitset) 是一种特殊的集合。虽然它只可用于存储整数，但可大大节省存储空间。
- **并查集**[素集合] (Disjoint-set) 是一种特殊的集合。除了集合的基本操作外，主要增加对以下**集合间**操作的支持：
 1. 查询一个元素属于几个集合中的哪一个。
 2. 将两个集合合并为一个。

并查集要求处理的几个集合**没有交集**。

并查集有几种不同的实现方法，其中最优的实现，以上操作的时间复杂度均接近 **$O(1)$** 。

并查集主要用于辅助一些**图**的算法。

其他排序算法

- 除了刚刚介绍的排序算法以外，还有很多其他排序算法：
 - **选择排序**。另一种简单的排序算法。时间复杂度为 $O(n^2)$ 。
 - **堆排序**。基于堆的排序。时间复杂度为 $O(n \log n)$ ，有时会替代归并排序或快速排序。
 - **希尔排序、计数排序、桶排序**等。这些排序不基于比较进行排序，时间复杂度可达 $O(n)$ ，但只能应对有限情况，比如数据只有一定范围内的正整数。

图的算法

- **Dijkstra 算法**用于计算从某个节点到其他所有节点的**最短距离**。
- **贝尔曼 - 福特算法** (Bellman–Ford Algorithm) 同样用于计算从某个节点开始的**最短距离**，但允许图中边上的权值为负数。
- **Floyd-Warshall 算法**用于求所有节点间的**最短距离**。
- **A* 算法**同样用于求某两个节点间的**最短距离**，被广泛运用于寻路或导航。
- **福特 - 富尔克森算法** (Ford–Fulkerson algorithm) 用于解决**最大流问题**，比如从某个地点另外一个地点的最大交通量。


字符串的算法

- **KMP 算法** 用于查找某个词在一个很长的字符串中出现的位置。
- **博耶 - 穆尔算法** (Boyer-Moore Algorithm) 同样用于查找子串出现的位置，但一般情况下会比其他算法更加高效。此算法被广泛用于**搜索引擎**。
- **前缀树** (Trie) 是一种专用于储存很多字符串的**集合** (或**字典**) 的数据结构，被广泛用于**单词补全**。

算法设计

- 前面介绍的都是一些已经存在的算法，如果我们该如何自己设计算法呢？这里介绍一些比较常见的算法设计思路，如果要自己学习，可以从它们开始着手：
 1. **暴力穷举**[しらみつぶし] (Brute-force) 。
 2. **贪心法**[貪欲法] (Greedy) 。
 3. **分治法**[分割統治法] (Divide-and-conquer) 。
 4. **回溯法**[バックトラック法] (Backtracking) 。
 5. **动态规划法**[動的計画法] (Dynamic Programming, DP) 。

进一步的学习

- 要熟悉各种数据结构和算法，提升自己实现甚至设计数据结构和算法的能力，最好的方式莫过于**实际编程解决问题**。
- 如果没有实际问题需要解决，也可以通过某些算法练习网站在线练习，比如力扣（LeetCode）：
 <https://leetcode.com/>。

Q & A

Question and answer

总结

Sum Up

1. 数据结构和算法的基本概念：
 - ① 选择数据结构的流程。
 - ② 评价数据结构和算法的标准。
2. 一些基本数据结构：
主要记一些抽象结构：列表、集合和字典。
3. 一些基本算法：搜索问题、排序问题。
二叉搜索的思想。
4. 进一步的学习方法：实践。

THANK YOU!