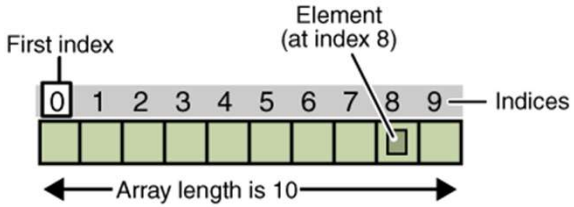
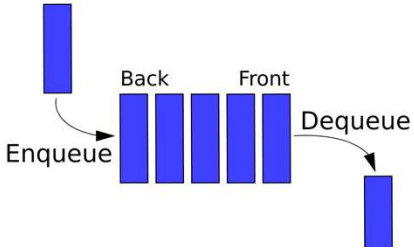
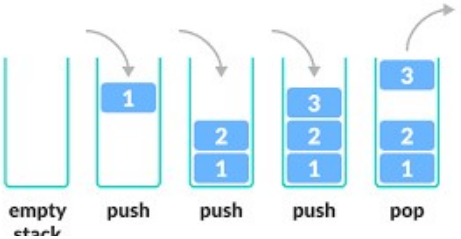
第6章
アルゴリズムとデータ構造

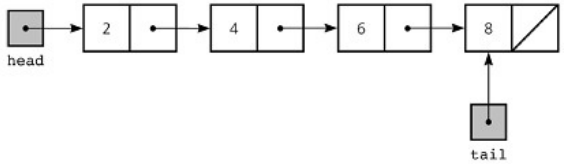
数据结构[データ構造] (Data Structure)

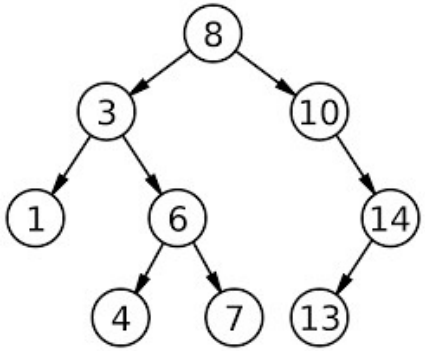
- 在计算机科学中，数据结构[データ構造]是计算机中存储、组织数据的方式。
- 数据结构[データ構造]往往与算法是分不开的，因为算法所要处理的对象[オブジェクト]就是数据结构[データ構造]中的数据。
- 合适的数据结构[データ構造]选择可以提高算法的效率。





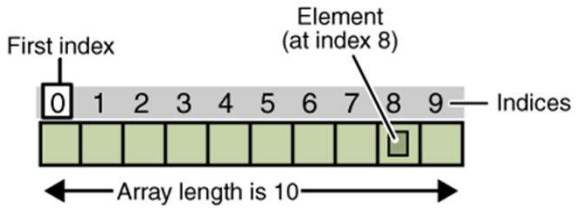


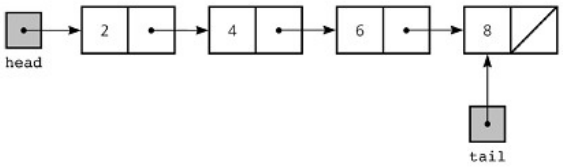


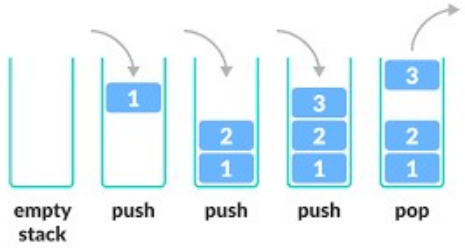


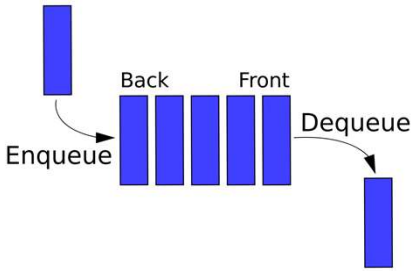
常用的数据结构[データ構造]

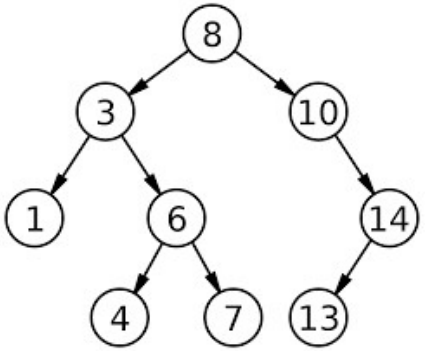
- Array 数组[配列]
- Linked List 链表[連結リスト]
- Stack 栈[スタック]
- Queue 队列[キュー]
- Tree 树[木]









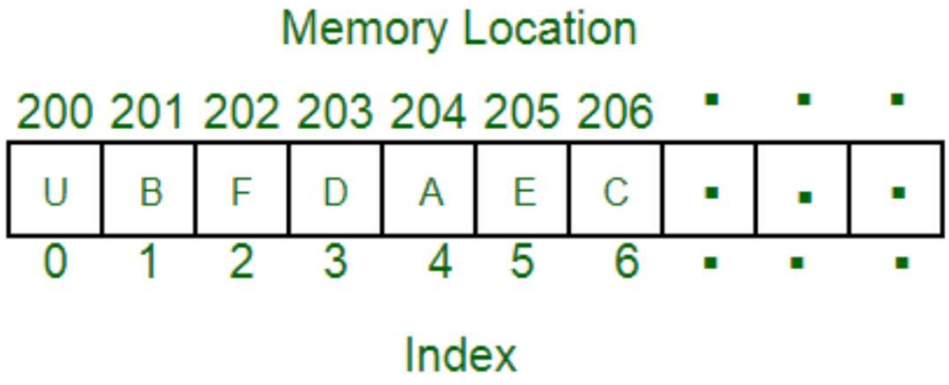


数组[配列]（Array）

- 数组[配列]是存储在一段连续的内存空间的容器。
- 存储的变量拥有相同的类型
- 通过索引（index）找到想要的数据
- 索引值从0开始
- 比如java里定义如下数组[配列]：

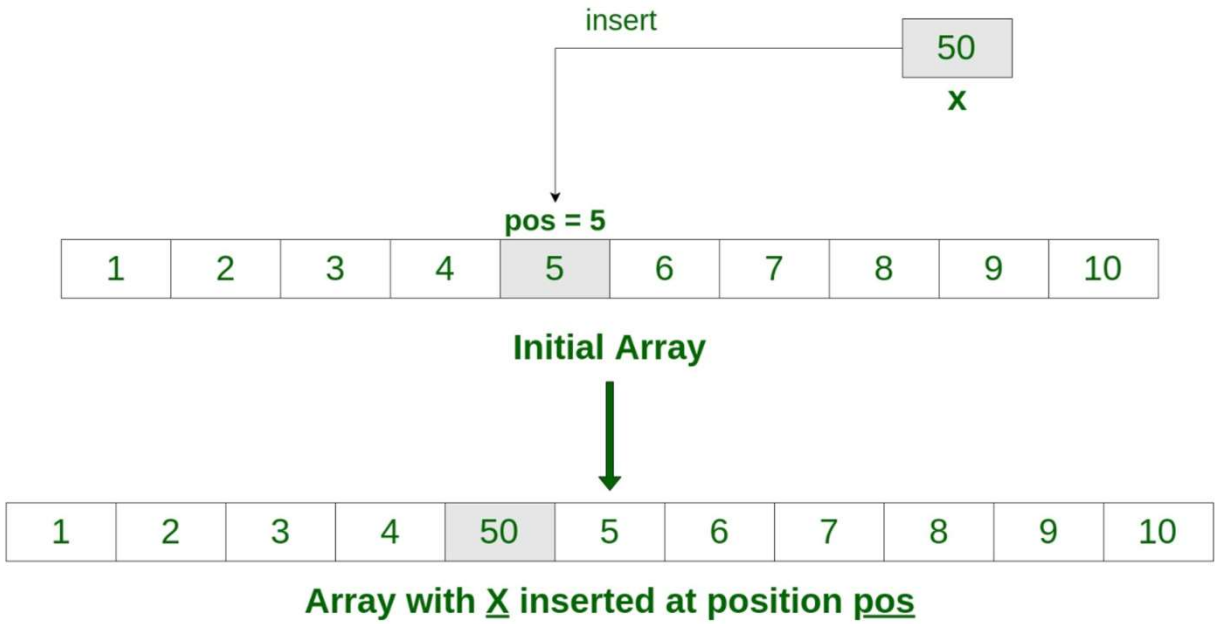
  int arr[] = {10, 20, 30, 40, 50};

  那么arr[0]存储的就是10，arr[4]存储的就是50。



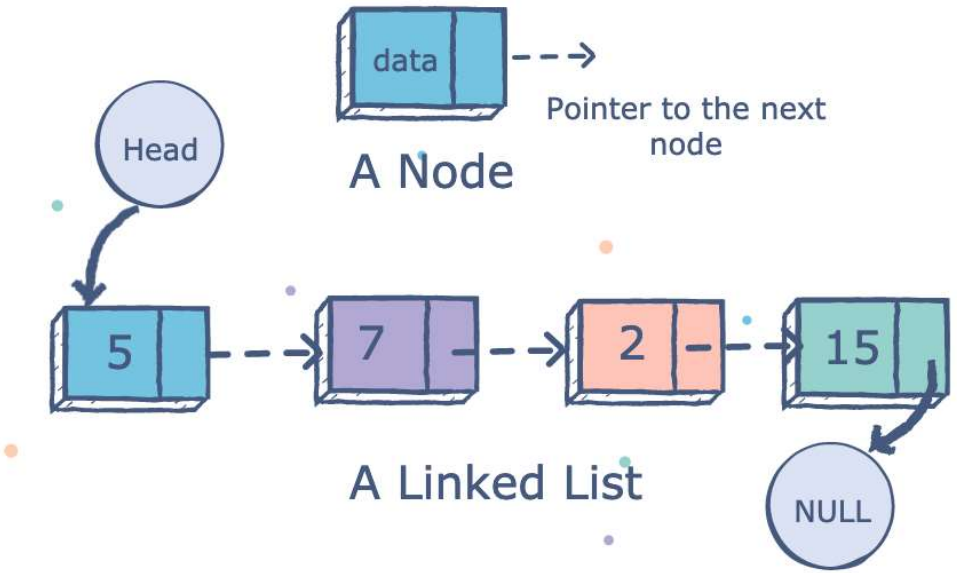
数组[配列] （Array）

- 中部插入数据：
  - 由于数组[配列]数据在内存上是连续的，插入会导致后面的数据都右移改变位置。
  - 下图插入50前5，6，7，8，9，10五个数字的位置都向右移1位
- 思考一下如何删除数据



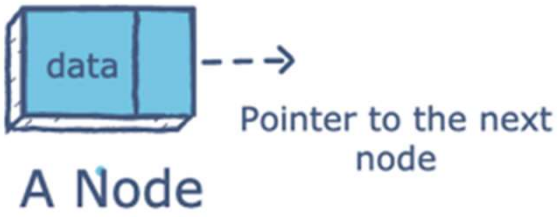
链表[連結リスト]（Linked List）

- 链表是线性的数据结构[データ構造]，但与数组[配列]不同，链表的元素并不存储在连续的内存空间里。
- 链表有很多种，这里提到的链表指单向链表链表由多个节点[ノード]（node）与指针[ポインタ]（pointer）构成。如下图所示：



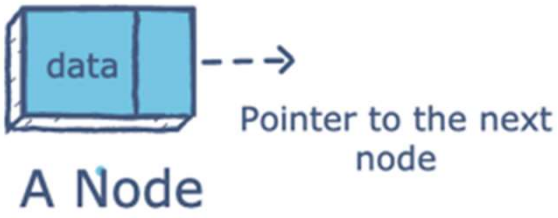
链表[連結リスト]里的节点

- 在最基本的单向链表[連結リスト]里，一个节点里会存两个东西，一个是数据，一个是指向下一个节点的指针。
- 这里的数据可以是任何类型的数据，比如int，char，String，甚至是MyClass。
- 指针可以理解为下一个节点的内存地址。



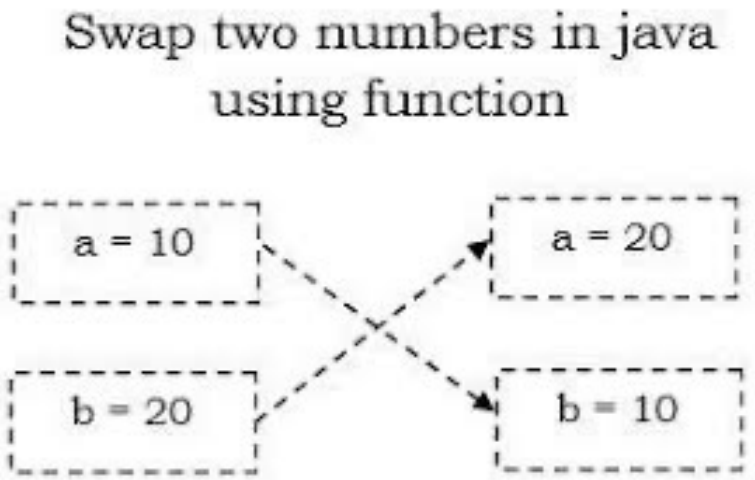
指针[ポインタ]（Pointer）

- 指针是编程里的一个非常重要的底层概念。
- 每个变量都存在内存的一个地方，每个地方都会有一个地址与其对应。像C语言这种稍微底层一点的高级语言，存在指针这个东西，虽然叫做指针，但其实是变量的内存地址（address）。
- Java不存在指针（pointer），但存在引用[参照]（reference），表示对象[オブジェクト]的存储位置。
- 所以java中，一个节点是一个对象[オブジェクト]，节点里存的指针其实就是下一个节点的引用[参照]（reference）。



指针[ポインタ]（Pointer）

- 尝试代码：Swap.java
- 思考这个代码为什么会输出这样的结果?
- 要怎么改才能输出期待的结果?



链表[連結リスト]的遍历[走查]

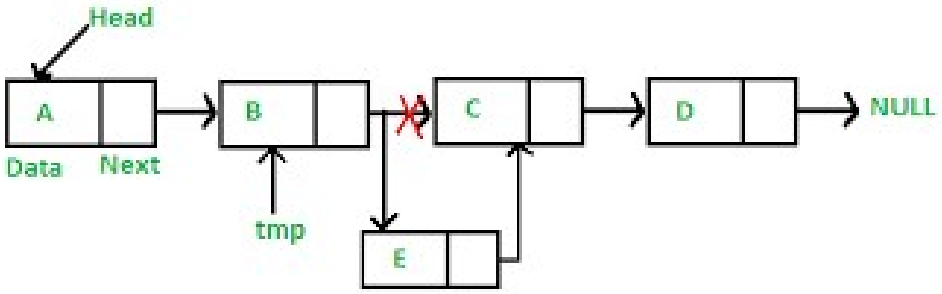
- 可以通过当前节点的指针找到下一个节点。
- 因此只要拥有头节点（head node），就相当于掌握了整个链表[連結リスト]。
- 链表[連結リスト]的最后一个节点的指针存储null。

```
//想象有一个链表，头节点是head，以下是伪代码

node = head;
while(node!=null){
    print(node.data); //输出当前节点存的内容
    node = node.next;  //将当前节点赋值为下一个节点
}
```

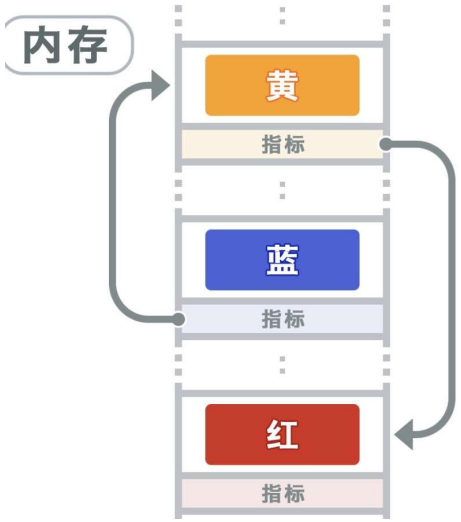
链表[連結リスト]的插入

- 如何向链表[連結リスト]中插入新数据?
- 以下图为例，将E插入到B和C之间的步骤:
  1. 将B指向的节点赋给E， E.next = B.next
  2. 将B的指针改成指向E， B.next = E
- 思考如何删除某个节点



思考时间

- 比较链表[連結リスト]与数组[配列]的区别：
  - 访问数据的速度
  - 插入数据，删除数据的速度
  - 内存上的存储方式

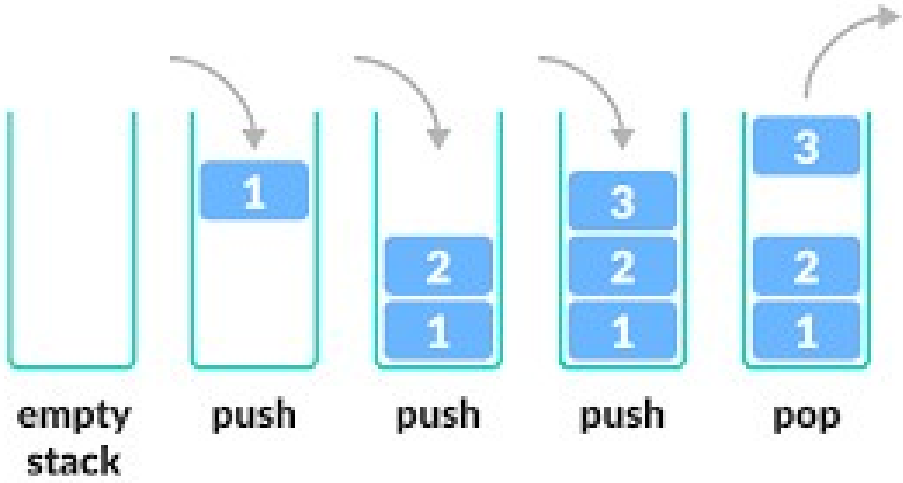


栈[スタック]（Stack）

- LIFO, Last In First Out，最后一个放进去的数据最先出来
- 可以想象成一个羽毛球桶，只有一个口提供放进和取出动作

栈[スタック]（Stack）

- 放进叫做push，取出叫做pop
- Java里面的Stack相关类[クラス]：Stack

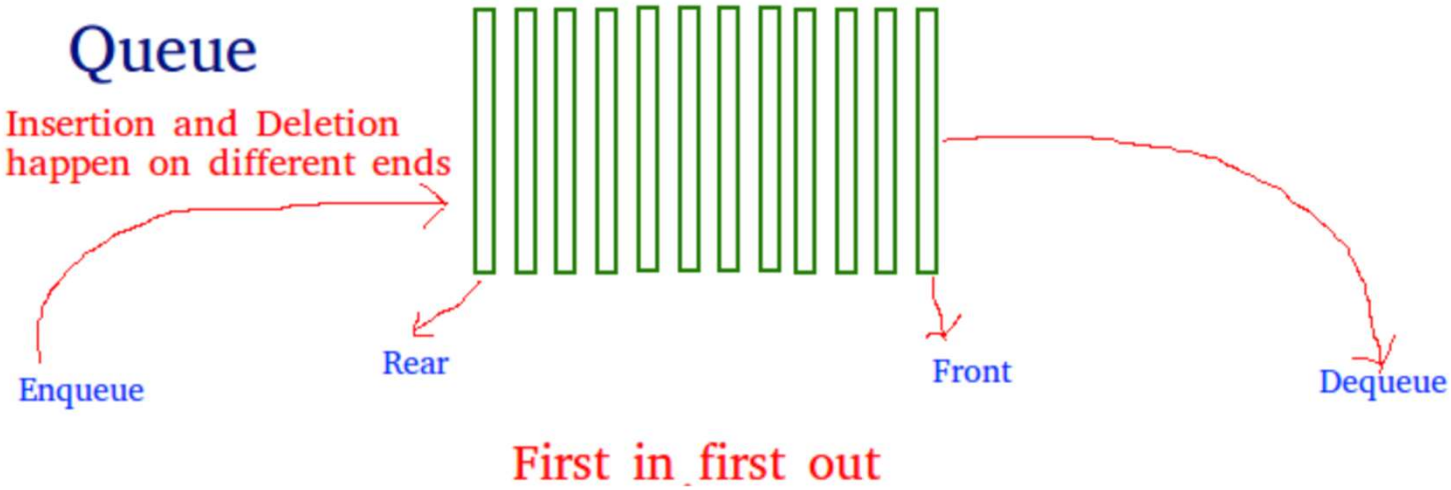


队列[キュー]（Queue）

- FIFO，First In First Out，先进来的数据，最先出去
- 生活中常见的排队就是一种队列[キュー]数据结构[データ構造]，先到者先得

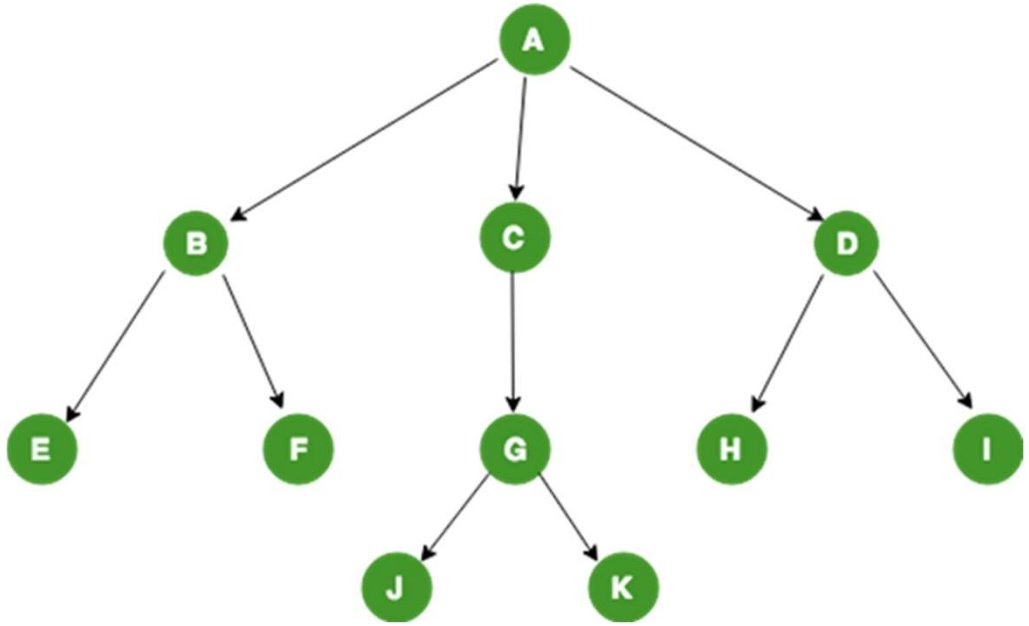
队列[キュー]（Queue）

- 进入队列[キュー]叫enqueue，出队列[キュー]叫dequeue
- Java里面的Queue类[クラス]：Queue



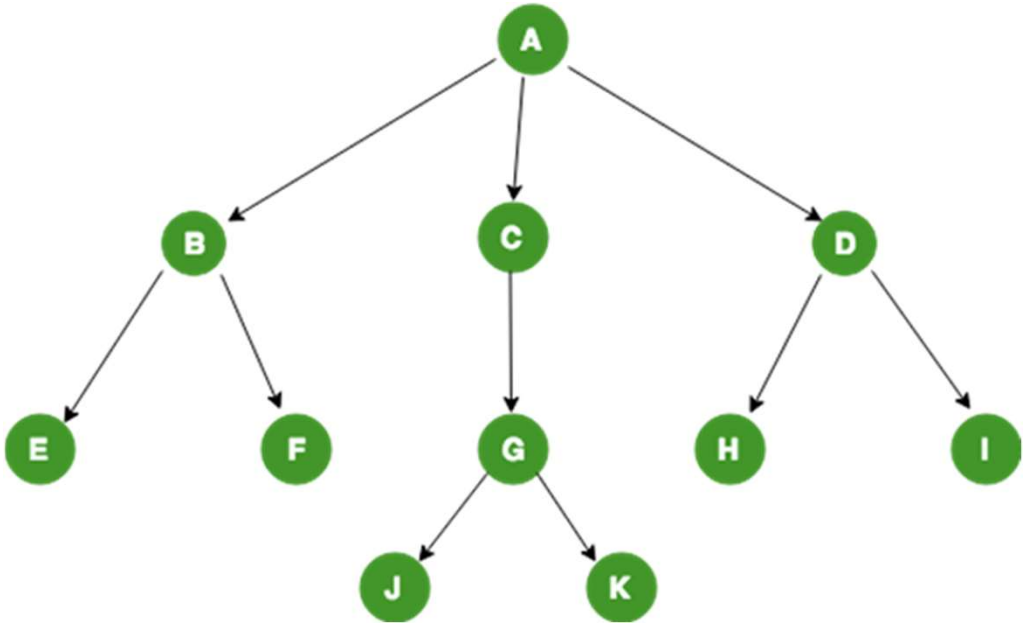
树[木]（Tree）

- 根节点，子节点，边组成的一种数据结构[データ構造]
- 不存在环状结构（即节点和边组成的闭合结构）
- 我们经常讨论二叉树[二分木]



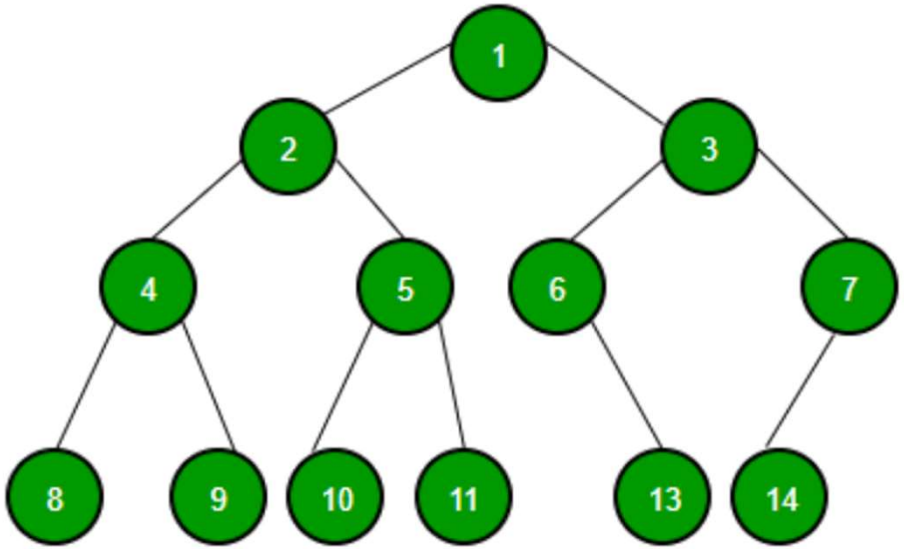
树（Tree）

- 以下图的树为例，A为根节点（root），BCD都是A的子节点（child），A是BCD的父节点（parent），EF是B的子节点，B是EF的父节点，以此类推。
- 没有子节点的节点也被称为叶（leaf）。
- 高度（height）= 层数-1，这棵树h就是4-1=3



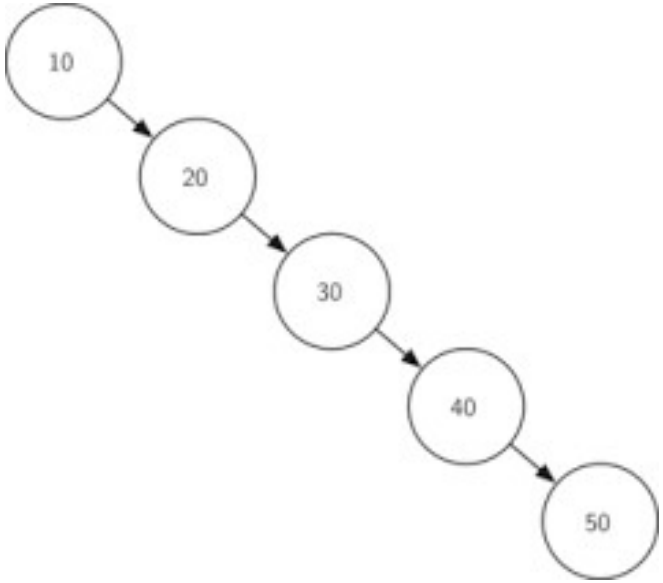
二叉树[二分木]（Binary Tree）

- 定义：每个节点最多只有两个子节点的树结构。（left child，right child）
- 左边子树：以左子节点为根节点构成的树
- 右边子树：以右子节点为根节点构成的树
- 我们经常讨论二叉搜索树[二分探索木]



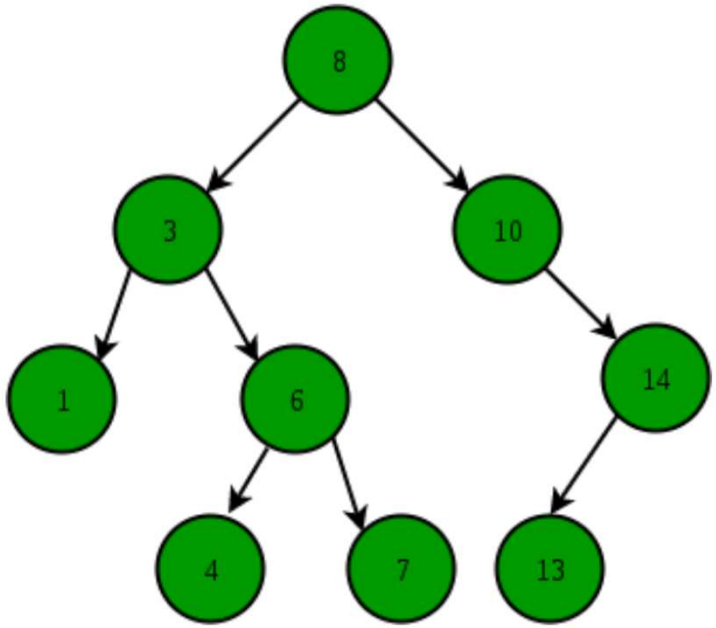
练习时间

- 下面这个链表[連結リスト]结构是不是树?
- 是二叉树么?



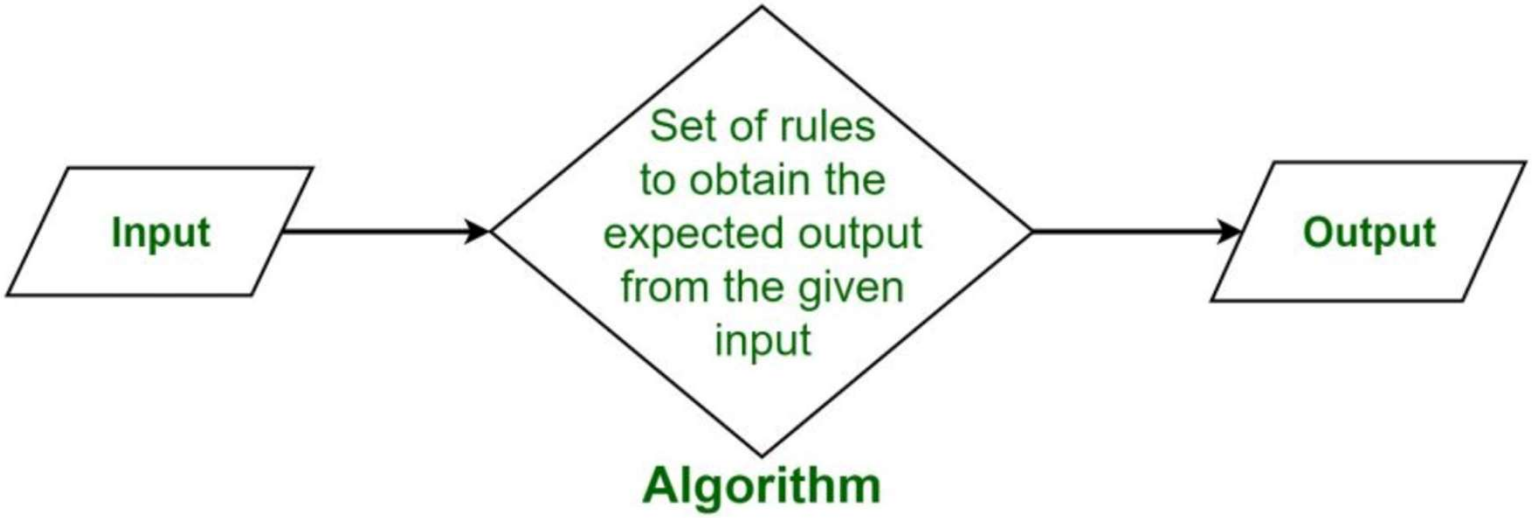
二叉搜索树[二分探索木] （Binary Search Tree, BST）

- 基本上用来存储可以比较大小的数据
- 定义：
  - 一个父节点的左边子树里所有节点的值都比父节点的值小
  - 一个父节点的右边子树里所有节点的值都比父节点的值大
  - 一个父节点的左边子树和右边子树均必须为二叉搜索树



什么是算法[アルゴリズム]（Algorithm）？

- 算法是一些计算或者解决问题的操作所用遵循的过程或者一些规则。
- 以烹饪新食谱为例，要烹饪新食谱，需要先阅读说明和步骤，然后按照给定的顺序逐一执行。这样才能完美的烹饪出新菜。同样，算法有助于完成编程任务以获取预期的输出。
- 设计的算法与编程语言无关，即算法是可以用任何一种语言实现的简单指令。



关于解决问题

- 一个问题，多种算法
- 比如将手里的扑克牌进行排序，可以有很多种方法：
  - 抽一张牌，插入到合适的位置
  - 抽牌随便放到手里，全部抽完之后再进行顺序调整
  - …
- 但不同的算法，具有不同性质：
  - 解决问题的时间长短
  - 解决问题占用的空间的大小
  - …
- 因此，在某个环境下为了解决某个问题，找到那个最合适现有环境的算法尤为重要。

如何找到算法

- 一般不会想不出“算法”，只可能会想不出“最合适的算法”。
- 可以先想出最直接的算法，然后逐步优化它。
- 深入理解并编程实现一些基本算法：
  - 分治统治
  - 排序算法
  - 搜索算法
  - 动态规划
  - …
- 培养自己的计算机思维（多敲代码，多做算法题leetcode）
- 是一种可以后天培养的能力。

今天会介绍的算法

- 排序（Sort）：
  - 冒泡排序[バブルソート] 必须掌握
  - 插入排序[挿入ソート]
  - 快速排序[クイックソート]
  - 归并排序[マージソート]
- 哈希[ハッシュ]（Hash）

冒泡排序[バブルソート](Bubble Sort)　必须掌握

- 想法：从小到大排序，相邻的两个数字如果顺序不对就将它们调换，直到所有相邻数字的顺序都正确。
- 例：将下图数列用冒泡排序[バブルソート]从大到小排序

- 从左往右先比较14和33，顺序正确，不用交换

- 然后比较33和27，顺序不对进行调换

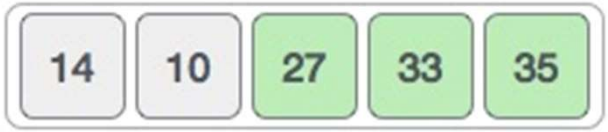
- 再比较33和35，顺序正确，不用交换

冒泡排序[バブルソート](Bubble Sort)

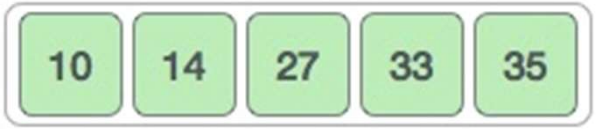
- 然后比较35和10，顺序错误，需要调换。

- 到这一步，就能保证35是数列里最大的数字了
- （想想为什么），因此不用再管它。
- 再一次从左往右重复上面的步骤，将33也确定好

- 再次重复，将27确定好



- 最后一次，将10和14的相对位置确定好，排序结束。



冒泡排序[バブルソート](Bubble Sort)

- 从前面的例子可以感觉到，大的数字像气泡冒到水面一样，移动到右边，这就是名字的由来。
- 尝试代码： BubbleSort.java

插入排序[挿入ソート](Insertion Sort)

- 打扑克抽牌放到手里所用到的排序
- 想法：
  1. 将第一张牌放入手中（将第一个数字放到排好序的一边）。
  2. 抽到第n张牌，则将第n张牌插入到手里的牌的合适位置（使得手里的牌再次排好顺序）。
  3. 最终手里的牌将是排好顺序的牌。

- 至于如何将第n张牌放到合适的位置，有很多方法，比如一个一个比较或者使用二分法（二分插入排序）。

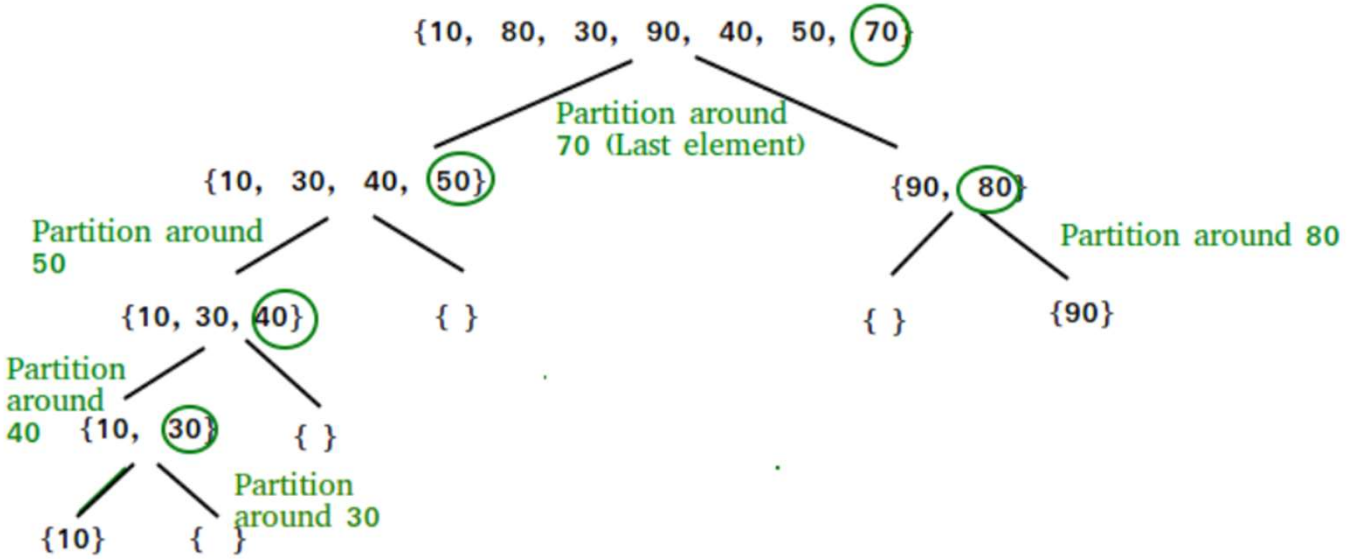
尝试代码： InsertionSort.java

快速排序[クイックソート]（Quick Sort）

- 算法：
  1. 在要排序的数列中随便选一个数字（第一个，最后一个，随机位置一个，中间的数字都可以）作中枢[ピボット]（pivot）。
  2. 将其他数字和pivot进行比较，比它小的放在他左边，比它大的放在他右边。
  3. 如果左边的数字不止一个的话，则对左边数列进行1，2，3的步骤；否则左边数字不动。
  4. 如果右边的数字不止一个的话，则对右边数列进行1，2，3的步骤；否则右边数字不动。
  5. 排序结束。

快速排序[クイックソート]（Quick Sort）

- 如下图所示，圈起来的数字为当前数列的pivot。



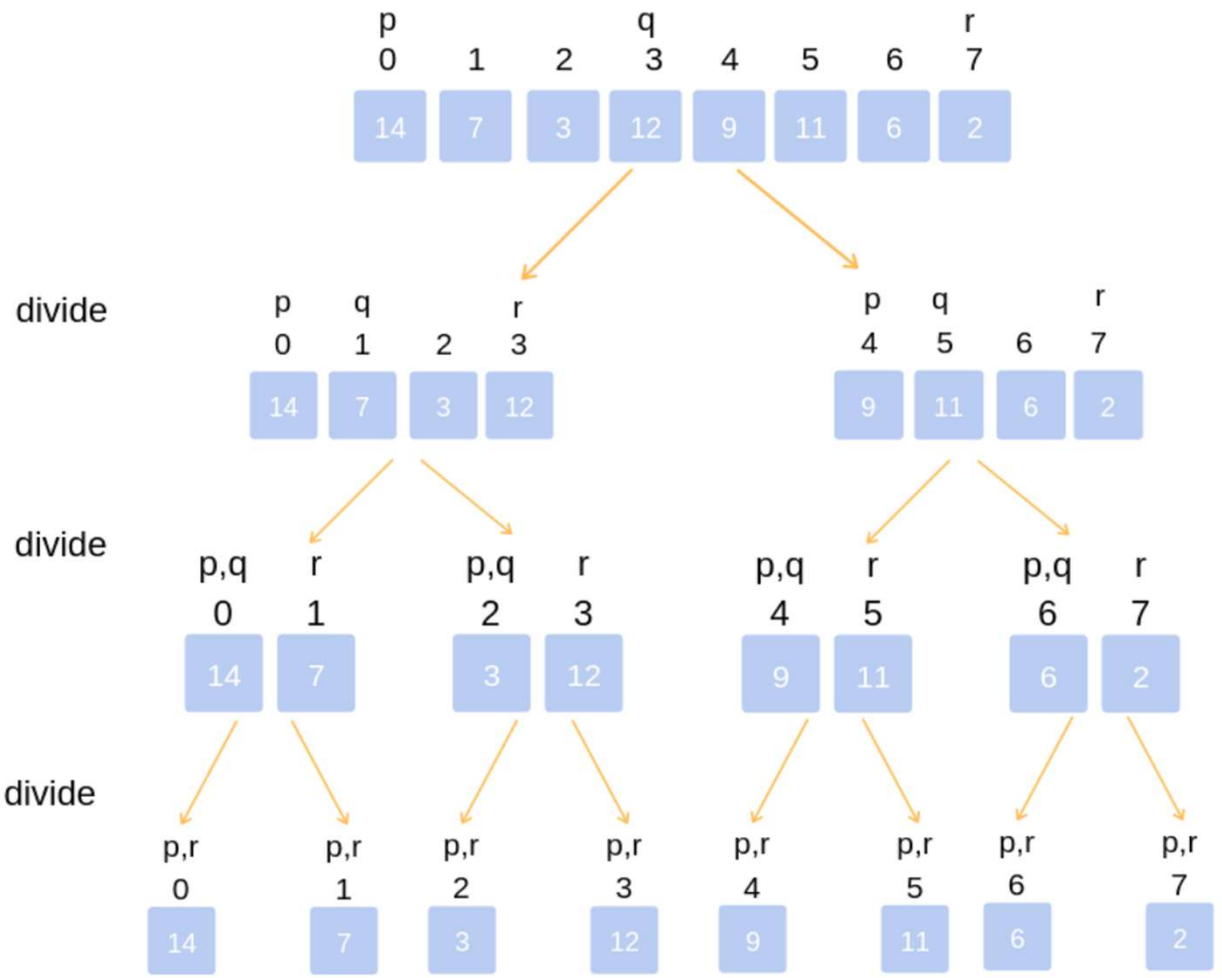
- 尝试代码：QuickSort.java

归并排序[マージソート]（Merge Sort）

- 利用了分治统治的思想[分割統治法]（Divide and Conquer）
- 想法：先把大数组[配列]分成的最小数组[配列]（长度为1），按顺序合并两个小数组[配列]，最后得到排好顺序的大数组[配列]。
- 算法：
  - 先定义一个操作叫mergeSort（array），array是我们想要排序的数组[配列]
  - mergeSort(array)：
    1. 将array从中间分开，分成left, right两个数组[配列]。（divide）
    2. 如果left长度不为1，则执行mergeSort(left)
    3. 如果right长度不为1，则执行mergeSort(right)
    4. 将left和right进行merge（）操作
  - 下一页有图

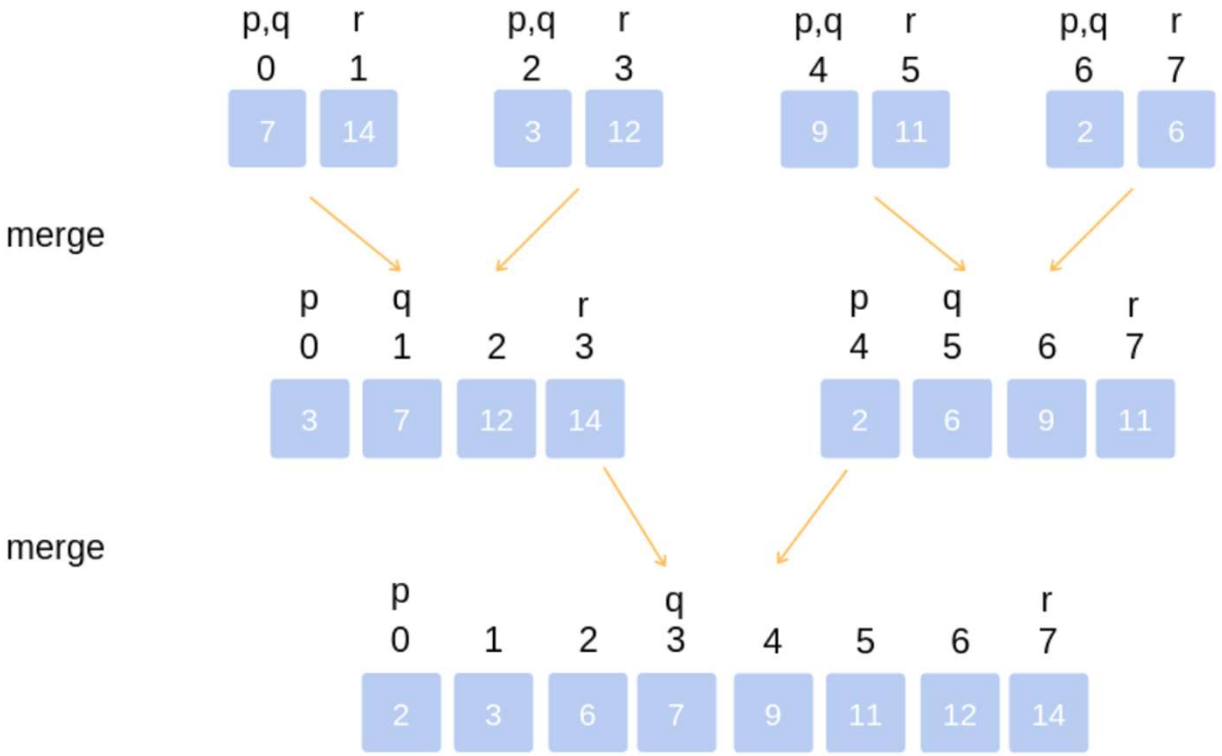
归并排序[マージソート]（Merge Sort）

- 分解：



归并排序[マージソート]（Merge Sort）

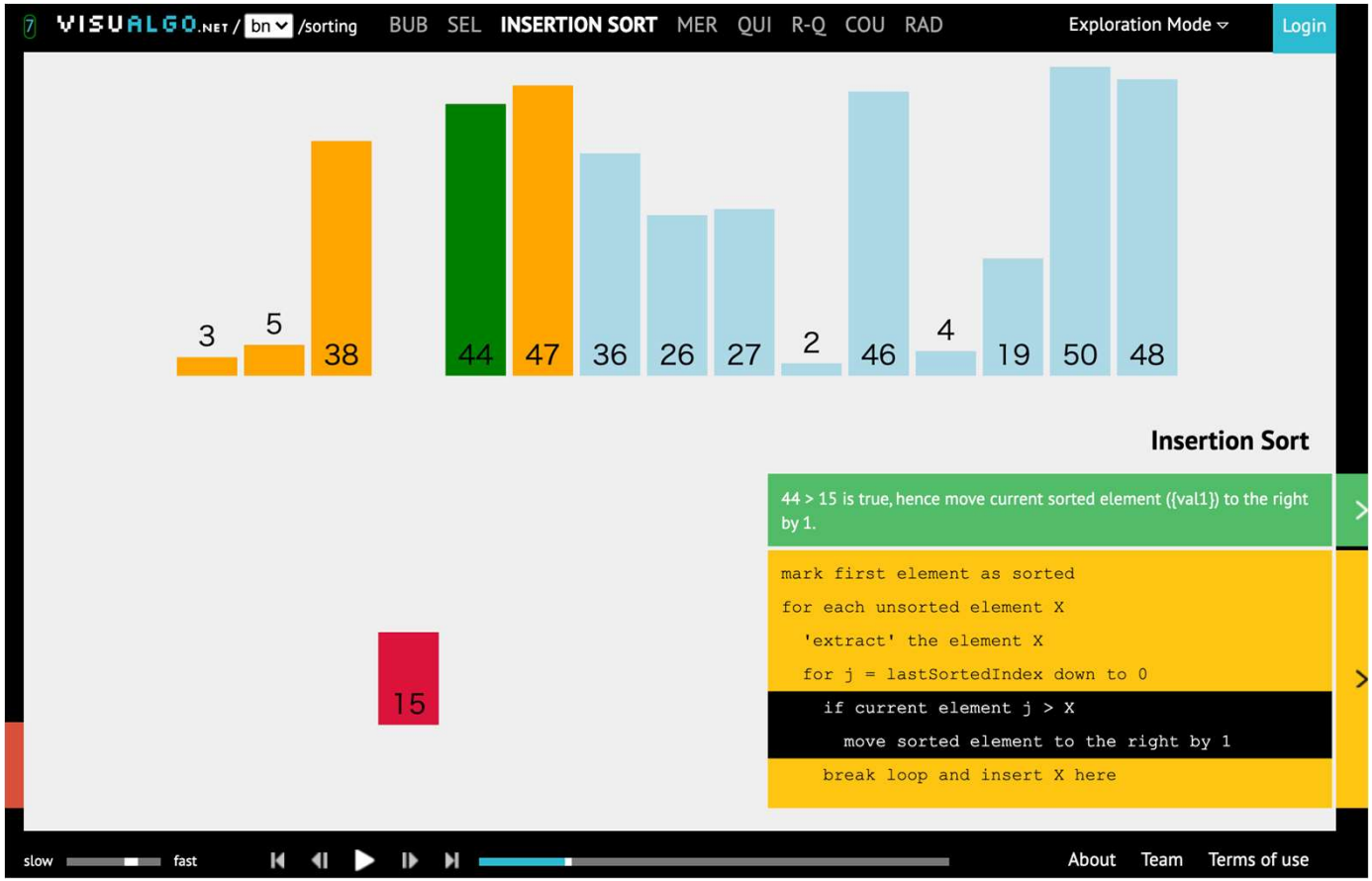
- 合并：



尝试代码： MergeSort.java

排序算法的动画

- https://visualgo.net/bn/sorting

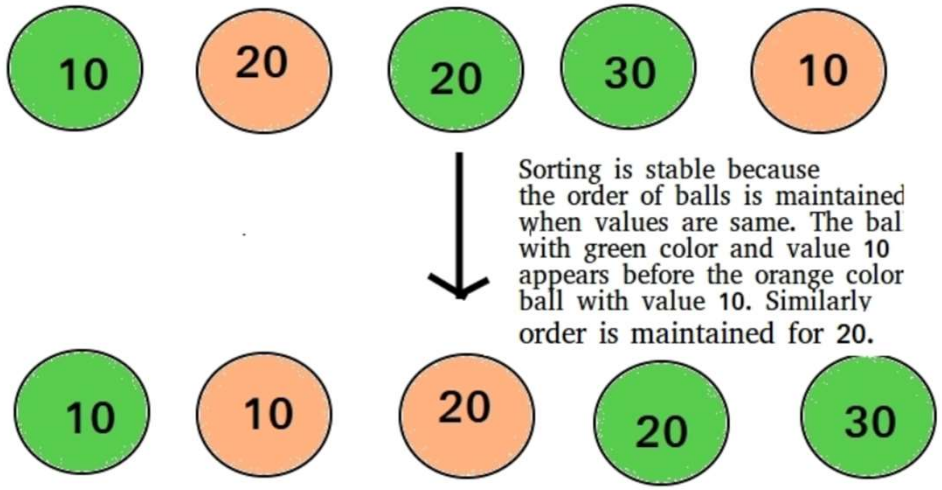


比较这四种排序算法

- 最坏情况的速度（时间复杂度）：快排 = 归并 > 插入 = 冒泡
- 如果有n个数据进行排序，快排和归并循环的次数是nlogn级别的，而插入和冒泡则是n^2级别的。（思考原因）
- 最坏情况的空间占有大小：归并>快排=插入=冒泡（归并不是在原数组[配列]中改变顺序，需要额外的空间）

排序算法的稳定性[安定性]（Stability）

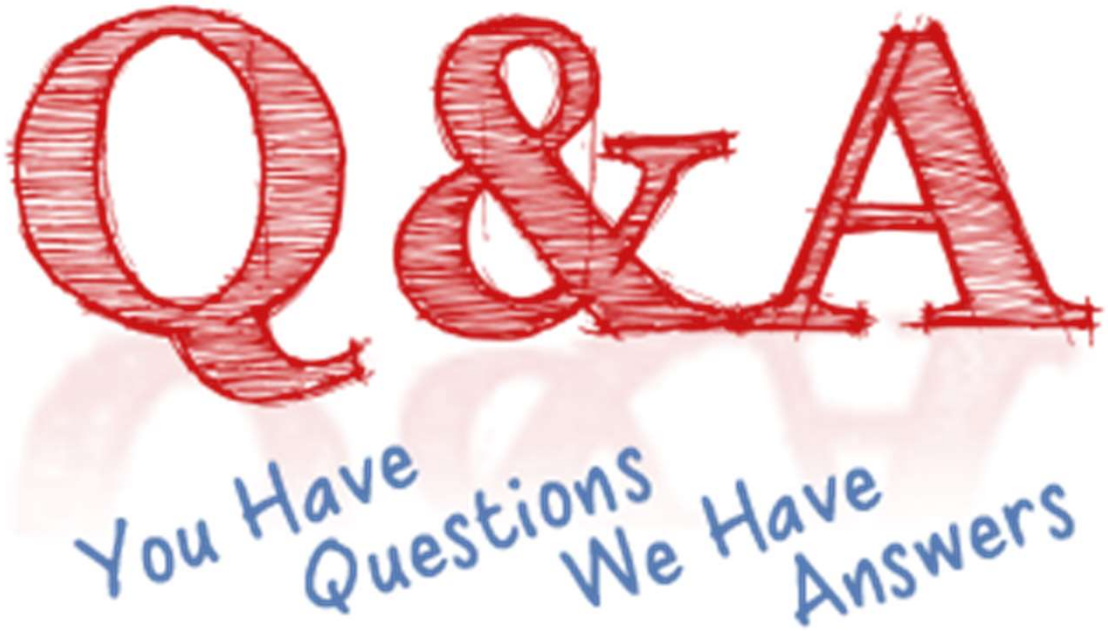
- 如果元素具有多个属性，比如下图的小球具有数字和颜色两种属性。



- 那么将小球以数字大小进行排序之后，能保证同样数字的小球的颜色的相对顺序和原来一样，那么这个排序算法就是稳定的，否则不稳定。
- 思考一下这四种哪些是稳定的?

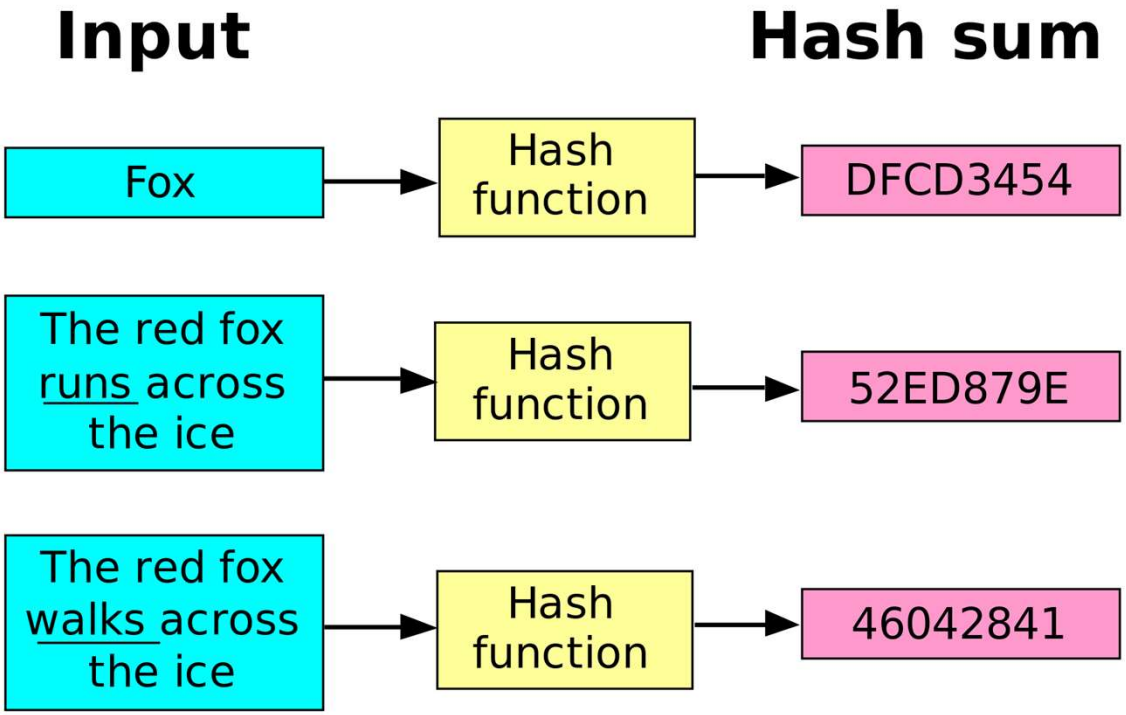
还有哪些排序?

- 堆排序
- 桶排序
- 希尔排序
- 选择排序
- 随机排序



哈希[ハッシュ]函数（Hash function）

- 是一种从任何一种数据中创建小的数字“指纹”的方法，简单来说，就是将任意长度的输入，压缩为某一固定长度的输出的方法。



哈希[ハッシュ]函数

- 不是指某种特定的函数，而是某一类函数
- 一个简单的例子：

- mod是求余的意思
- 输入为598，输出为22
- 输入为23235.输出为3
- 无论输入为多大的数字，输出都会分布在0～63之间

哈希[ハッシュ]函数的用途

- 加密算法是一种哈希[ハッシュ]，eg：sha256
- 哈希表（hash table）
- java语言里所有对象[オブジェクト]都有一个独一无二的哈希码（hashcode），可以用来比较对象[オブジェクト]是否相同。

HashMap

- Java里面的通过key索引得到value的数据结构[データ構造]，索引速度与哈希表存储量无关，是索引速度最快的数据结构[データ構造]。
- 键值对：哈希表里存的单位元素，一个键（key）对应一个值（value）
- 一个哈希表里每个key不可重复，value可以有重复。