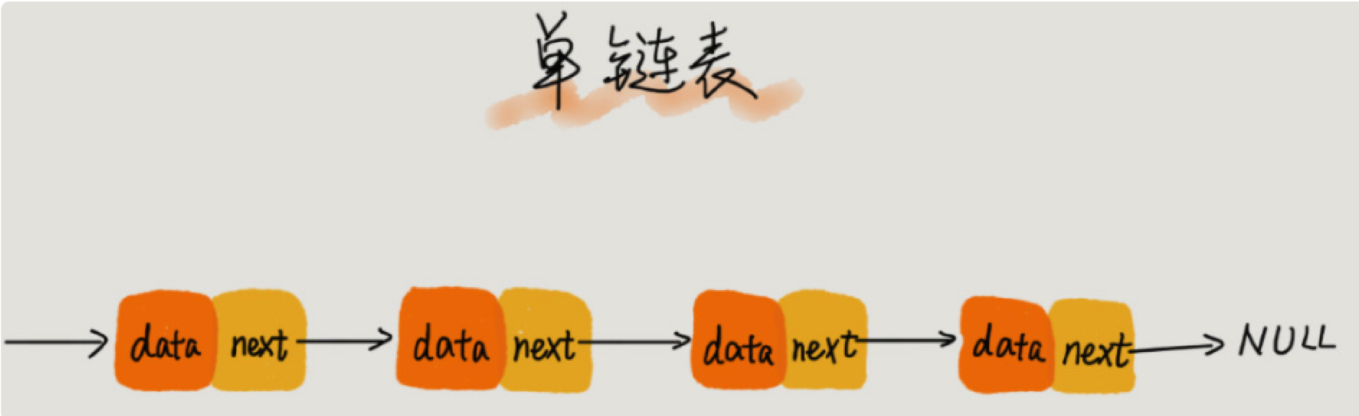
每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域



节点用代码表示，则如下：

JavaScript 复制代码

```javascript
class Node {
  constructor(val) {
    this.val = val;
    this.next = null;
  }
}
```

- data 表示节点存放的数据
- next 表示下一个节点指向的内存空间

相比于线性表顺序结构，操作复杂。由于不必须按顺序存储，链表在插入的时候可以达到 `O(1)` 的复杂度，比另一种线性表顺序表快得多，但是查找一个节点或者访问特定编号的节点则需要O(n)的时间，而线性表和顺序表相应的时间复杂度分别是 `O(logn)` 和 `O(1)`

链表的结构也十分多，常见的有四种形式：

- 单链表：除了头节点和尾节点，其他节点只包含一个后继指针
- 循环链表：跟单链表唯一的区别就在于它的尾结点又指回了链表的头结点，首尾相连，形成了一个环
- 双向链表：每个结点具有两个方向指针，后继指针(next)指向后面的结点，前驱指针(prev)指向前面的结点，其中节点的前驱指针和尾结点的后继指针均指向空地址NULL
- 双向循环链表：跟双向链表基本一致，不过头节点前驱指针指向尾迹诶单和尾节点的后继指针指向头节点

7.2. 操作

关于链表的操作可以主要分成如下：

- 遍历
- 插入
- 删除

7.2.1. 遍历

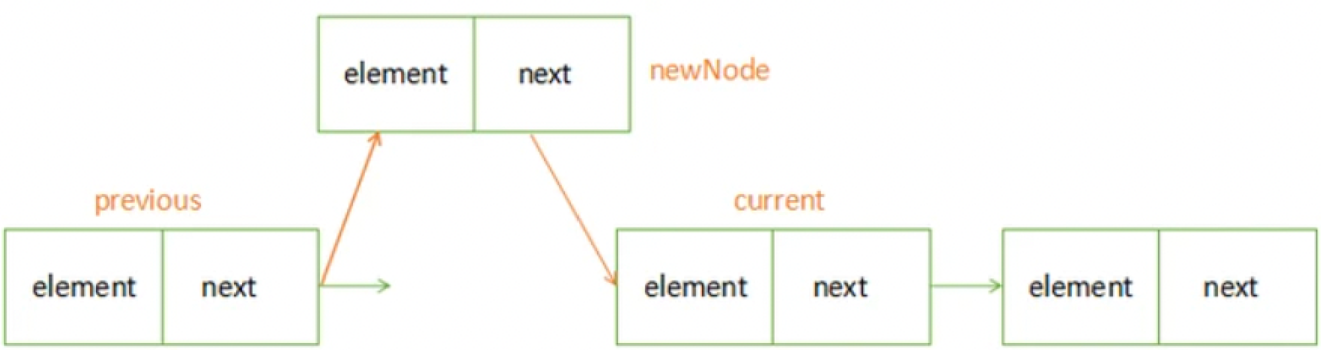
遍历很好理解，就是根据 `next` 指针遍历下去，直到为 `null` ，如下：

JavaScript 复制代码

```javascript
let current = head
while(current){
 console.log(current.val)
  current = current.next
}
```

7.2.2. 插入

向链表中间插入一个元素，可以如下图所示：



可以看到，插入节点可以分成如下步骤：

- 存储插入位置的前一个节点
- 存储插入位置的后一个节点
- 将插入位置的前一个节点的 next 指向插入节点
- 将插入节点的 next 指向前面存储的 next 节点

相关代码如下所示：

JavaScript | 复制代码

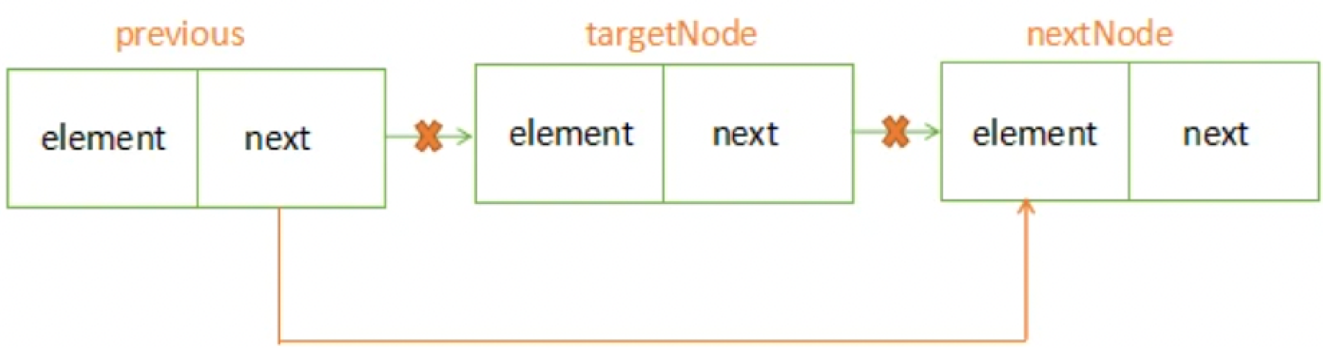
```javascript
let current = head
while (current < position){
  pervious = current;
  current = current.next;
}
pervious.next = node;
node.next = current;
```

如果在头节点进行插入操作的时候，会实现 `previousNode` 节点为 `undefined`，不适合上述方式

解放方式可以是在头节点前面添加一个虚拟头节点，保证插入行为一致

7.2.3. 删除

向链表任意位置删除节点，如下图操作：



从上图可以看到删除节点的步骤为如下：

- 获取删除节点的前一个节点
- 获取删除节点的后一个节点
- 将前一个节点的 next 指向后一个节点
- 向删除节点的 next 指向为null

如果想要删除制定的节点，示意代码如下：

JavaScript | 复制代码

```javascript
while (current != node){
  pervious = current;
  current = current.next;
  nextNode = current.next;
}
pervious.next = nextNode
```

同样如何希望删除节点处理行为一致，可以在头节点前面添加一个虚拟头节点

7.3. 应用场景

缓存是一种提高数据读取性能的技术，在硬件设计、软件开发中都有着非常广泛的应用，比如常见的 `CPU` 缓存、数据库缓存、浏览器缓存等等

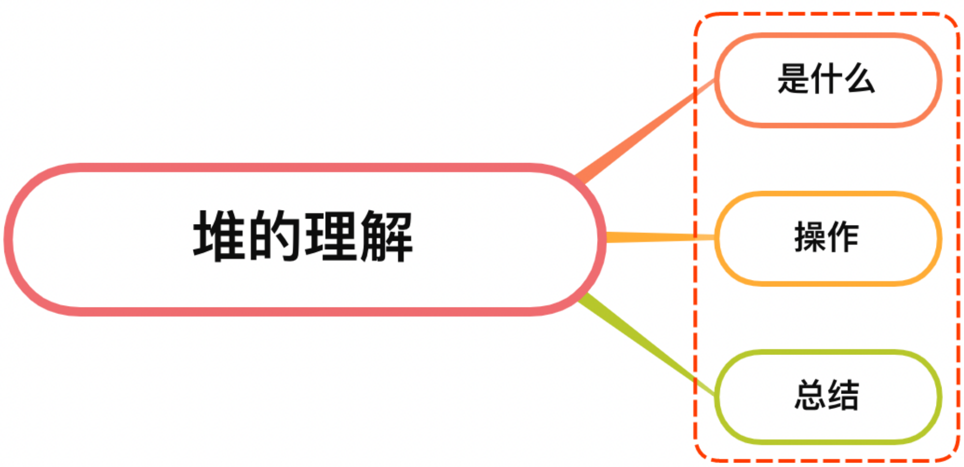
当缓存空间被用满时，我们可能会使用 `LRU` 最近最好使用策略去清楚，而实现 `LRU` 算法的数据结构是链表，思路如下：

维护一个有序单链表，越靠近链表尾部的结点是越早之前访问的。当有一个新的数据被访问时，我们从链表头部开始顺序遍历链表

- 如果此数据之前已经被缓存在链表中了，我们遍历得到这个数据的对应结点，并将其从原来的位置删除，并插入到链表头部
- 如果此数据没在缓存链表中
    - 如果此时缓存未满，可直接在链表头部插入新节点存储此数据
    - 如果此时缓存已满，则删除链表尾部节点，再在链表头部插入新节点

由于链表插入删除效率极高，达到O(1)。对于不需要搜索但变动频繁且无法预知数量上限的数据的情况的时候，都可以使用链表

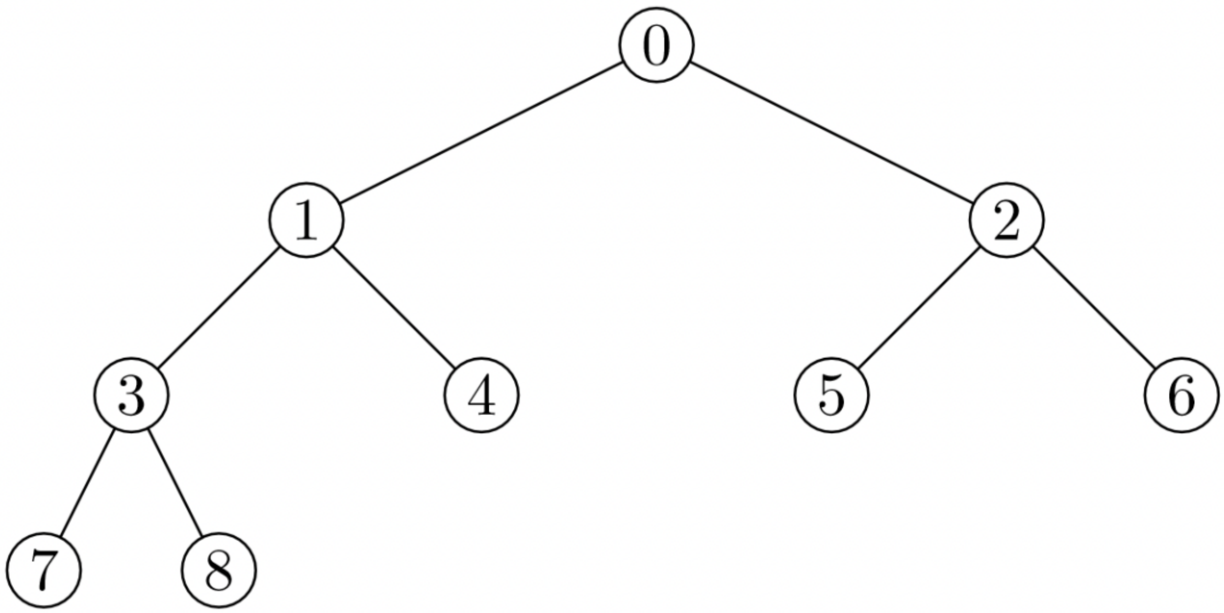
8. 说说你对堆的理解？如何实现？应用场景？



8.1. 是什么

堆(Heap)是计算机科学中一类特殊的数据结构的统称

堆通常是一个可以被看做一棵完全二叉树的数组对象，如下图：



总是满足下列性质：

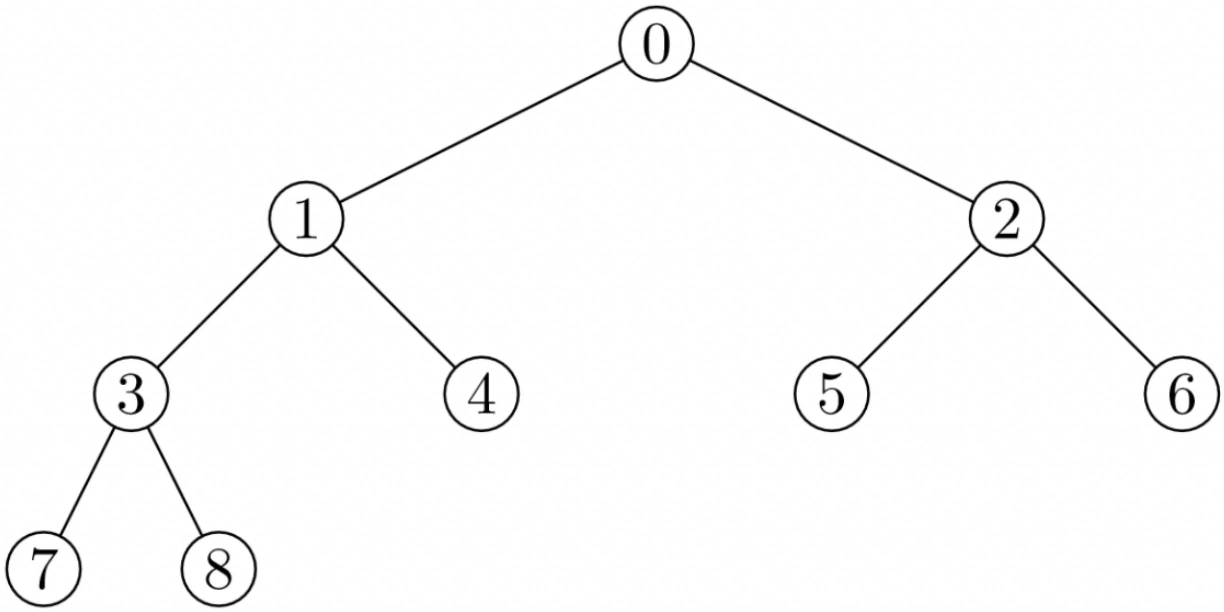
- 堆中某个结点的值总是不大于或不小于其父结点的值
- 堆总是一棵完全二叉树

堆又可以分成最大堆和最小堆：

- 最大堆：每个根结点，都有根结点的值大于两个孩子结点的值
- 最小堆：每个根结点，都有根结点的值小于孩子结点的值

8.2. 操作

堆的元素存储方式，按照完全二叉树的顺序存储方式存储在一个一维数组中，如下图：



用一维数组存储则如下：

JavaScript 复制代码

```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

根据完全二叉树的特性，可以得到如下特性：

- 数组零坐标代码的是堆顶元素
- 一个节点的父亲节点的坐标等于其坐标除以2整数部分
- 一个节点的左节点等于其本身节点坐标 * 2 + 1
- 一个节点的右节点等于其本身节点坐标 * 2 + 2

根据上述堆的特性，下面构建最小堆的构造函数和对应的属性方法：

JavaScript 复制代码

```javascript
class MinHeap {
  constructor() {
    // 存储堆元素
    this.heap = []
  }
  // 获取父元素坐标
  getParentIndex(i) {
    return (i - 1) >> 1
  }

  // 获取左节点元素坐标
  getLeftIndex(i) {
    return i * 2 + 1
  }

 // 获取右节点元素坐标
  getRightIndex(i) {
    return i * 2 + 2
  }

  // 交换元素
  swap(i1, i2) {
    const temp = this.heap[i1]
    this.heap[i1] = this.heap[i2]
    this.heap[i2] = temp
  }

  // 查看堆顶元素
  peek() {
    return this.heap[0]
  }

  // 获取堆元素的大小
  size() {
    return this.heap.length
  }
}
```

涉及到堆的操作有：

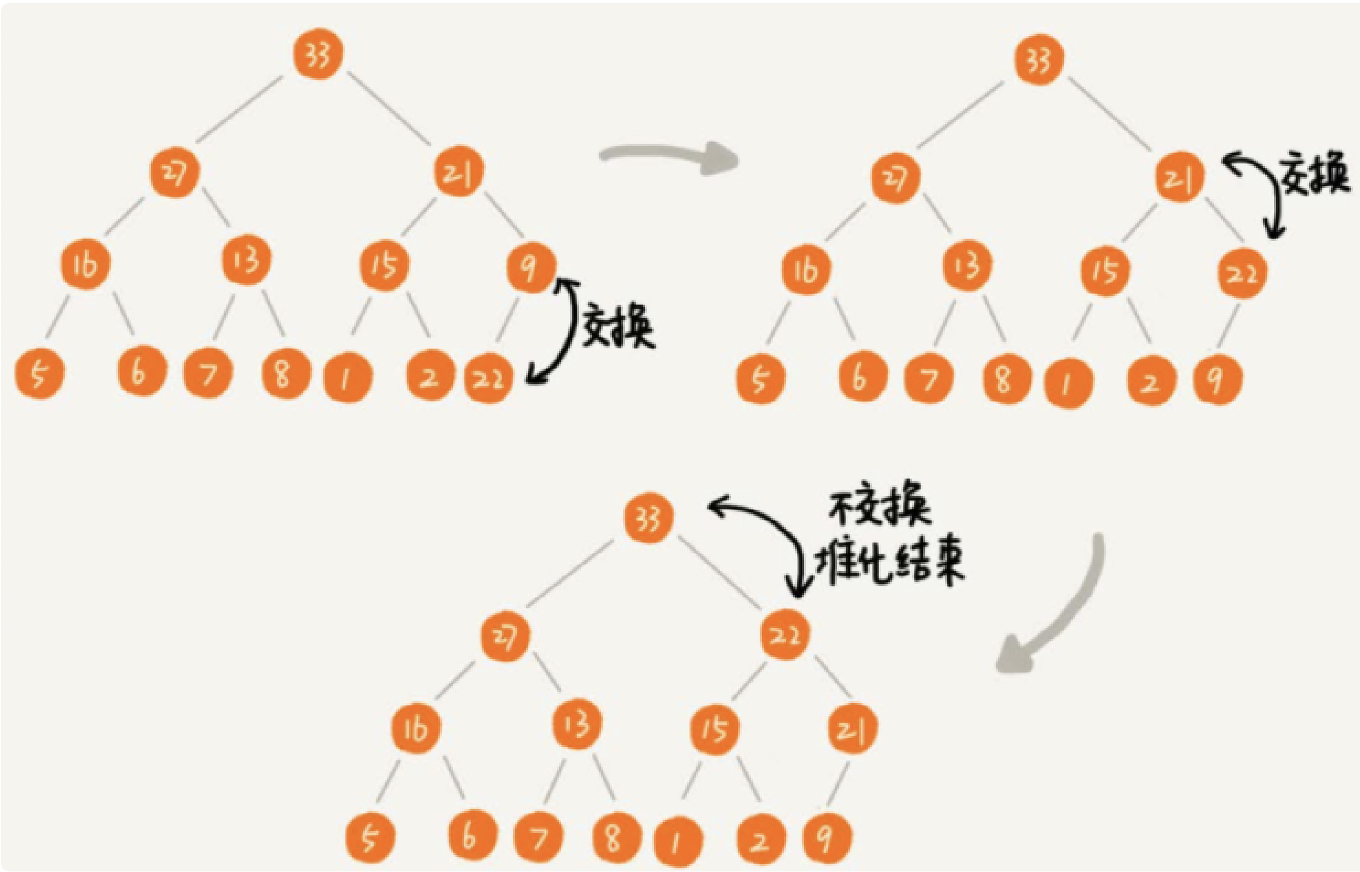
- 插入
- 删除

8.2.1. 插入

将值插入堆的底部，即数组的尾部，当插入一个新的元素之后，堆的结构就会被破坏，因此需要堆中一个元素做上移操作

将这个值和它父节点进行交换，直到父节点小于等于这个插入的值，大小为 `k` 的堆中插入元素的时间复杂度为 `O(logk)`

如下图所示，22节点是新插入的元素，然后进行上移操作：



相关代码如下：

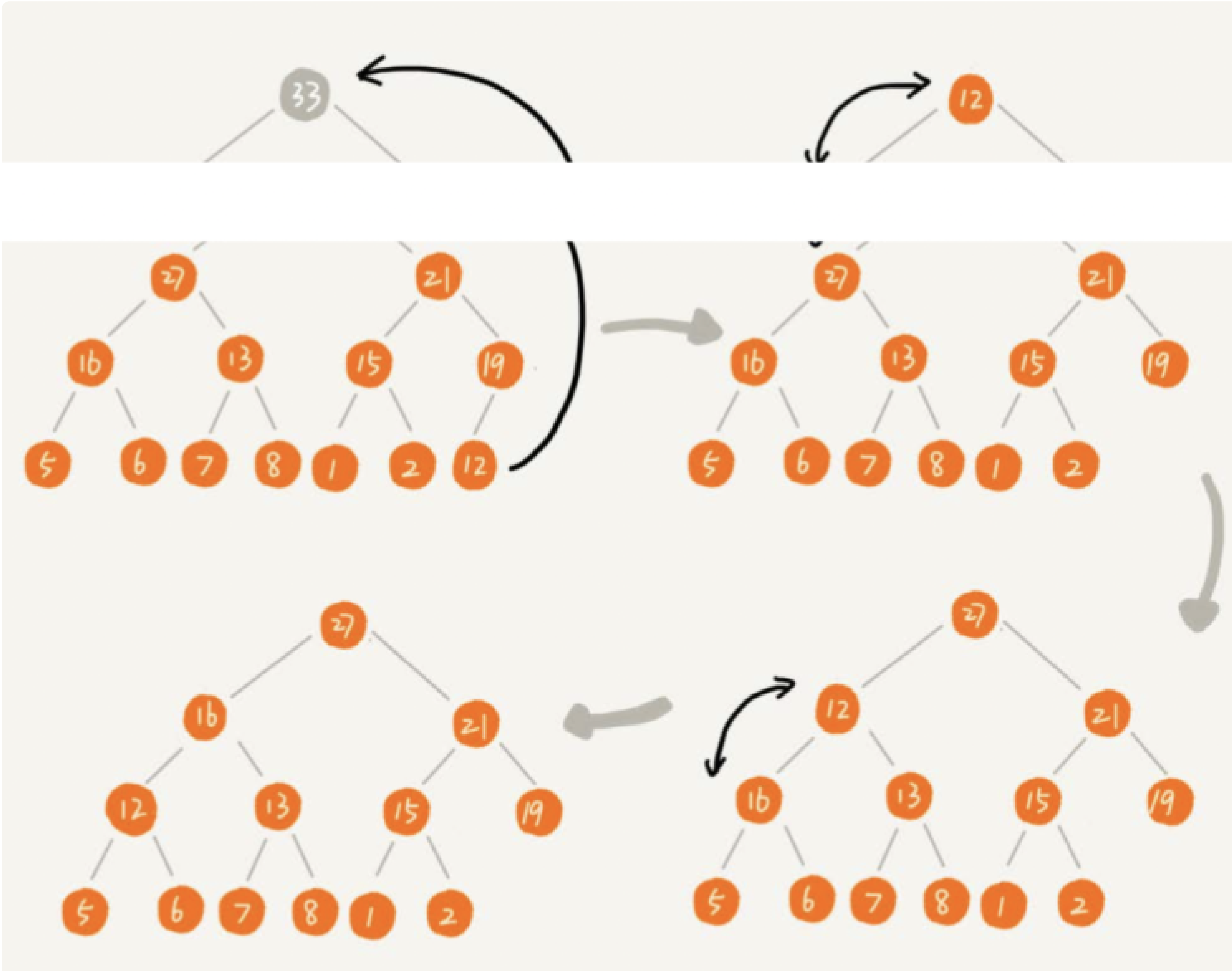
```JavaScript
// 插入元素
insert(value) {
  this.heap.push(value)
  this.shifUp(this.heap.length - 1)
}

// 上移操作
shiftUp(index) {
  if (index === 0) { return }
  const parentIndex = this.getParentIndex(index)
  if(this.heap[parentIndex] > this.heap[index]){
    this.swap(parentIndex, index)
    this.shiftUp(parentIndex)
  }
}
```

8.2.2. 删除

常见操作是用数组尾部元素替换堆顶，这里不直接删除堆顶，因为所有的元素会向前移动一位，会破坏了堆的结构

然后进行下移操作，将新的堆顶和它的子节点进行交换，直到子节点大于等于这个新的堆顶，删除堆顶的时间复杂度为 `O(logk)`

整体如下图操作：



相关代码如下：

JavaScript | 复制代码

```javascript
// 删除元素
pop() {
  this.heap[0] = this.heap.pop()
  this.shiftDown(0)
}

// 下移操作
shiftDown(index) {
  const leftIndex = this.getLeftIndex(index)
  const rightIndex = this.getRightIndex(index)
  if (this.heap[leftIndex] < this.heap[index]){
    this.swap(leftIndex, index)
    this.shiftDown(leftIndex)
  }
  if (this.heap[rightIndex] < this.heap[index]){
    this.swap(rightIndex, index)
    this.shiftDown(rightIndex)
  }
}
```

8.2.3. 时间复杂度

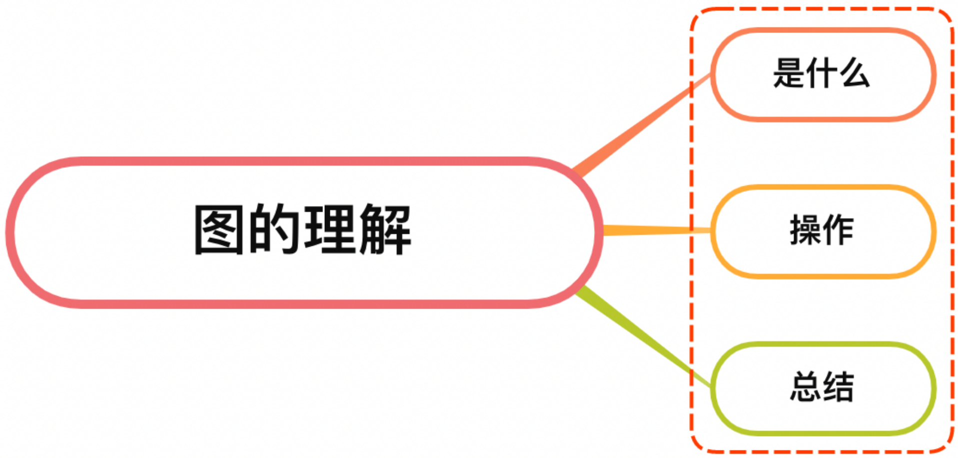
关于堆的插入和删除时间复杂度都是 `Olog(n)` ，原因在于包含n个节点的完全二叉树，树的高度不会超过 `log2n`

堆化的过程是顺着节点所在路径比较交换的，所以堆化的时间复杂度跟树的高度成正比，也就是 `Olog(n)` ，插入数据和删除堆顶元素的主要逻辑就是堆化

8.3. 总结

- 堆是一个完全二叉树
- 堆中每一个节点的值都必须大于等于(或小于等于)其子树中每个节点的值
- 对于每个节点的值都大于等于子树中每个节点值的堆，叫作“大顶堆”
- 对于每个节点的值都小于等于子树中每个节点值的堆，叫作“小顶堆”
- 根据堆的特性，我们可以使用堆来进行排序操作，也可以使用其来求第几大或者第几小的值

9. 说说你对图的理解？相关操作有哪些?



9.1. 是什么

在计算机科学中，图是一种抽象的数据类型，在图中的数据元素通常称为结点， `V` 是所有顶点的集合， `E` 是所有边的集合

如果两个顶点 `v` , `w` ，只能由 `v` 向 `w` ，而不能由 `w` 向 `v` ，那么我们就把这种情况叫做一个从 `v` 到 `w` 的有向边。 `v` 也被称做初始点， `w` 也被称为终点。这种图就被称做有向图

如果 `v` 和 `w` 是没有顺序的，从 `v` 到达 `w` 和从 `w` 到达 `v` 是完全相同的，这种图就被称为无向图

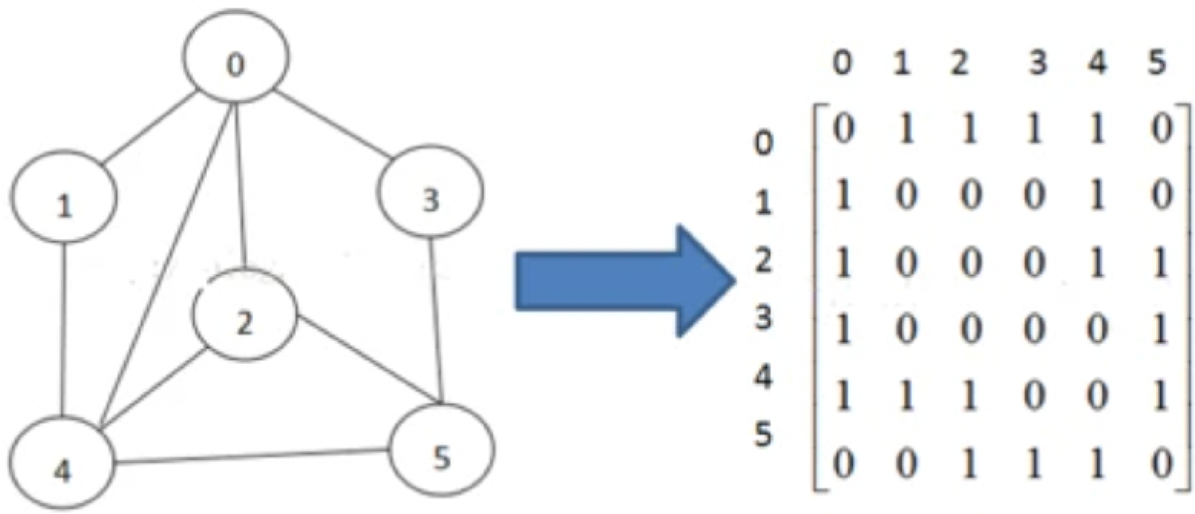
图的结构比较复杂，任意两个顶点之间都可能存在联系，因此无法以数据元素在存储区中的物理位置来表示元素之间的关系

常见表达图的方式有如下：

- 邻接矩阵
- 邻接表

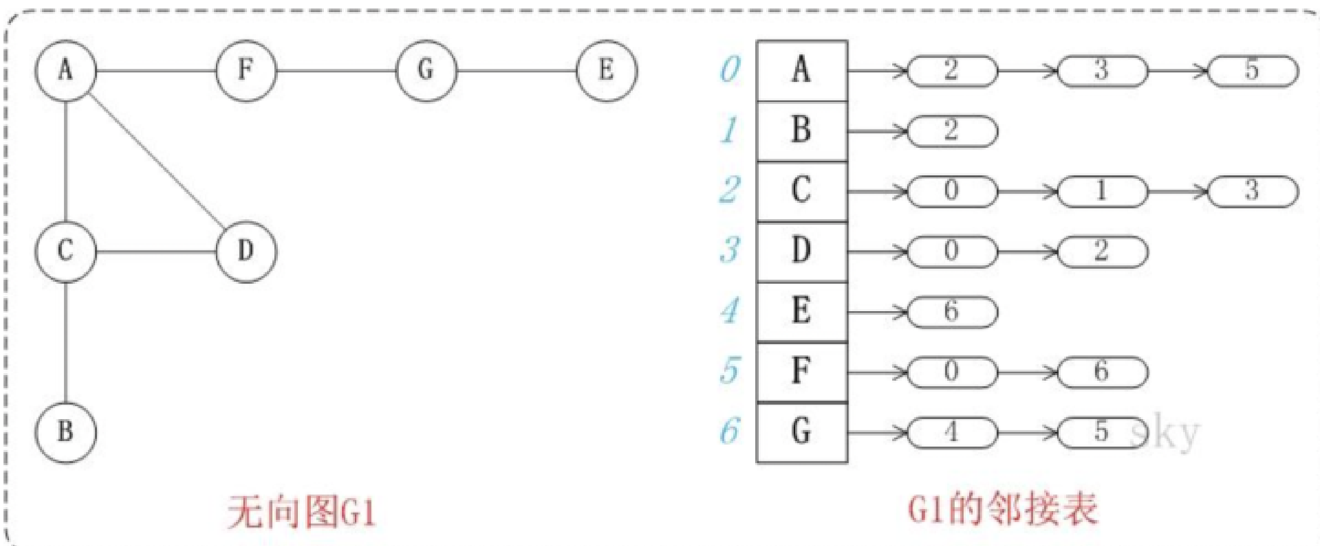
9.1.1. 邻接矩阵

通过使用一个二维数组 `G[N][N]` 进行表示 `N` 个点到 `N-1` 编号，通过邻接矩阵可以立刻看出两顶点之间是否存在一条边，只需要检查邻接矩阵行 `i` 和列 `j` 是否是非零值，对于无向图，邻接矩阵是对称的



9.1.2. 邻接表

存储方式如下图所示：



在 `javascript` 中，可以使用 `Object` 进行表示，如下：

```javascript
const graph = {
  A: [2, 3, 5],
  B: [2],
  C: [0, 1, 3],
  D: [0, 2],
  E: [6],
  F: [0, 6],
  G: [4, 5]
}
```

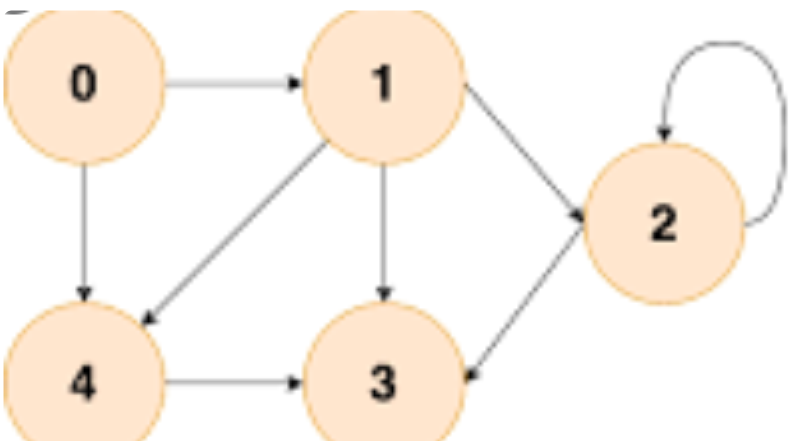
图的数据结构还可能包含和每条边相关联的数值（edge value），例如一个标号或一个数值（即权重，weight；表示花费、容量、长度等）

9.2. 操作

关于的图的操作常见的有：

- 深度优先遍历
- 广度优先遍历

首先构建一个图的邻接矩阵表示，如下面的图：



用代码表示则如下：

```JavaScript
const graph = {
  0: [1, 4],
  1: [2, 4],
  2: [2, 3],
  3: [],
  4: [3],
}
```

9.2.1. 深度优先遍历

也就是尽可能的往深处的搜索图的分支

实现思路是，首先应该确定一个根节点，然后对根节点的没访问过的相邻节点进行深度优先遍历

确定以 0 为根节点，然后进行深度遍历，然后遍历1，接着遍历 2，然后3，此时完成一条分支 `0 - 1 - 2- 3` 的遍历，换一条分支，也就是4，4后面因为3已经遍历过了，所以就不访问了

用代码表示则如下：

```JavaScript
const visited = new Set()
const dfs = (n) => {
  console.log(n)
  visited.add(n) // 访问过添加记录
  graph[n].forEach(c => {
    if(!visited.has(c)){ // 判断是否访问呢过
      dfs(c)
    }
  })
}
```

9.2.2. 广度优先遍历

先访问离根节点最近的节点，然后进行入队操作，解决思路如下：

- 新建一个队列，把根节点入队
- 把队头出队并访问
- 把队头的没访问过的相邻节点入队
- 重复二、三步骤，知道队列为空

用代码标识则如下：

JavaScript | 复制代码

```javascript
const visited = new Set()
const dfs = (n) => {
  visited.add(n)
  const q = [n]
  while(q.length){
    const n = q.shift()
    console.log(n)
    graph[n].forEach(c => {
      if(!visited.has(c)){
        q.push(c)
        visited.add(c)
      }
    })
  }
}
```

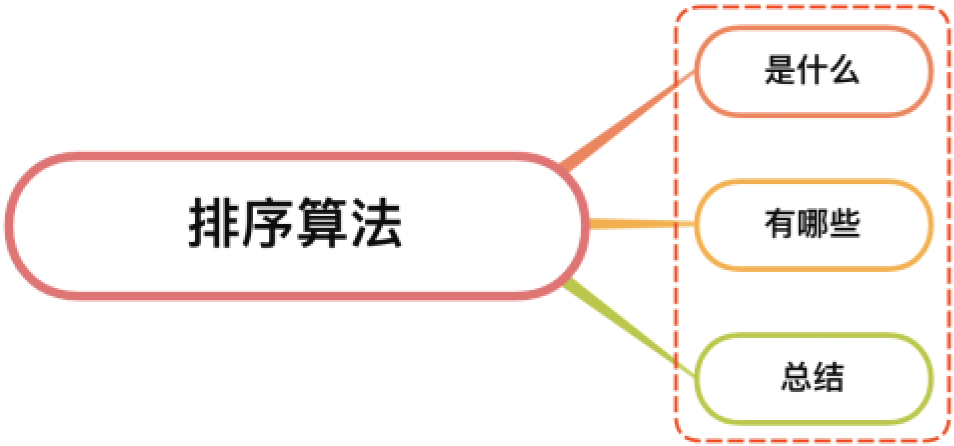
9.3. 总结

通过上面的初步了解，可以看到图就是由顶点的有穷非空集合和顶点之间的边组成的集合，分成了无向图与有向图

图的表达形式可以分成邻接矩阵和邻接表两种形式，在 `javascript` 中，则可以通过二维数组和对象的形式进行表达

图实际是很复杂的，后续还可以延伸出无向图和带权图，对应如下图所示：

10. 说说常见的排序算法有哪些？区别？



10.1. 是什么

排序是程序开发中非常常见的操作，对一组任意的数据元素经过排序操作后，就可以把他们变成一组一定规则排序的有序序列

排序算法属于算法中的一种，而且是覆盖范围极小的一种，彻底掌握排序算法对程序开发是有很大的帮助的

对与排序算法的好坏衡量，主要是从时间复杂度、空间复杂度、稳定性

时间复杂度、空间复杂度前面已经讲过，这里主要看看稳定性的定义

稳定性指的是假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变

即在原序列中，r[i] = r[j]，且 r[i] 在 r[j] 之前，而在排序后的序列中，r[i] 仍在 r[j] 之前，则称这种排序算法是稳定的；否则称为不稳定的

10.2. 有哪些

常见的算法排序算法有：

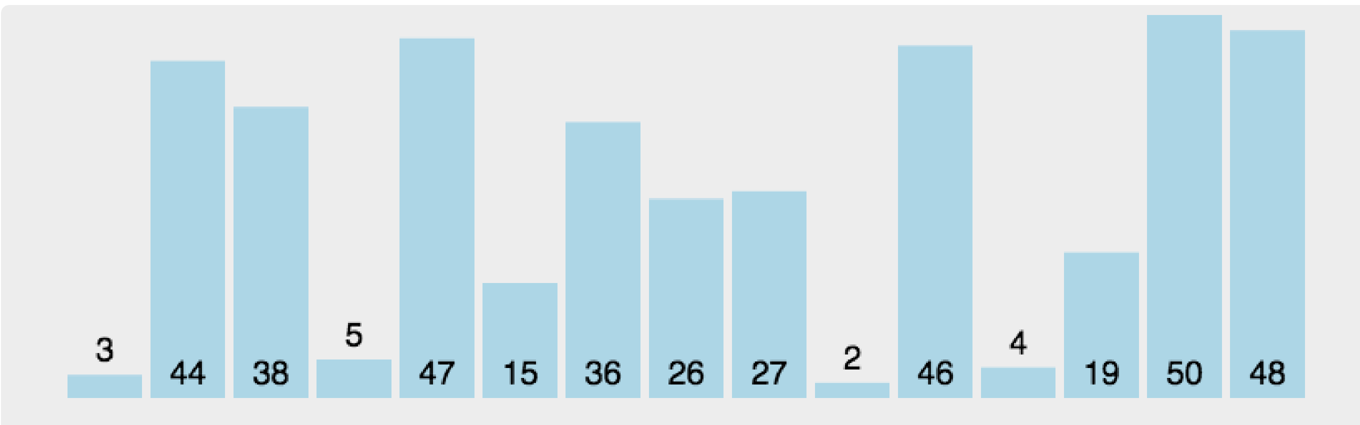
- 冒泡排序
- 选择排序
- 插入排序
- 归并排序
- 快速排序

10.2.1. 冒泡排序

一种简单直观的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来

思路如下：

- 比较相邻的元素，如果第一个比第二个大，就交换它们两个
- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数
- 针对所有的元素重复以上的步骤，除了最后一个
- 重复上述步骤，直到没有任何一堆数字需要比较



10.2.2. 选择排序

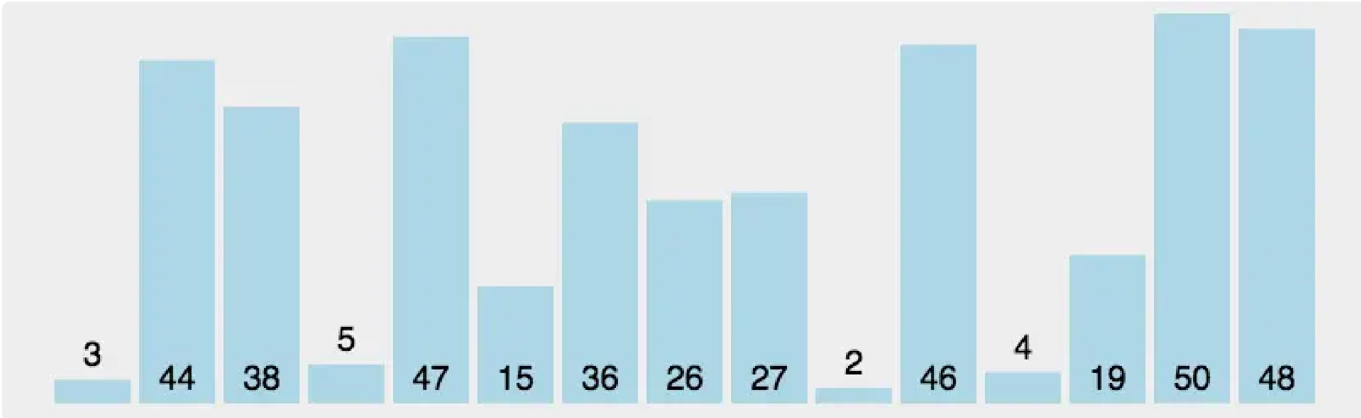
选择排序是一种简单直观的排序算法，它也是一种交换排序算法

无论什么数据进去都是 `O(n²)` 的时间复杂度。所以用到它的时候，数据规模越小越好

唯一的好处是不占用额外的内存存储空间

思路如下：

- 在未排序序列中找到最小（大）元素，存放到排序序列的起始位置
- 从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。
- 重复第二步，直到所有元素均排序完毕



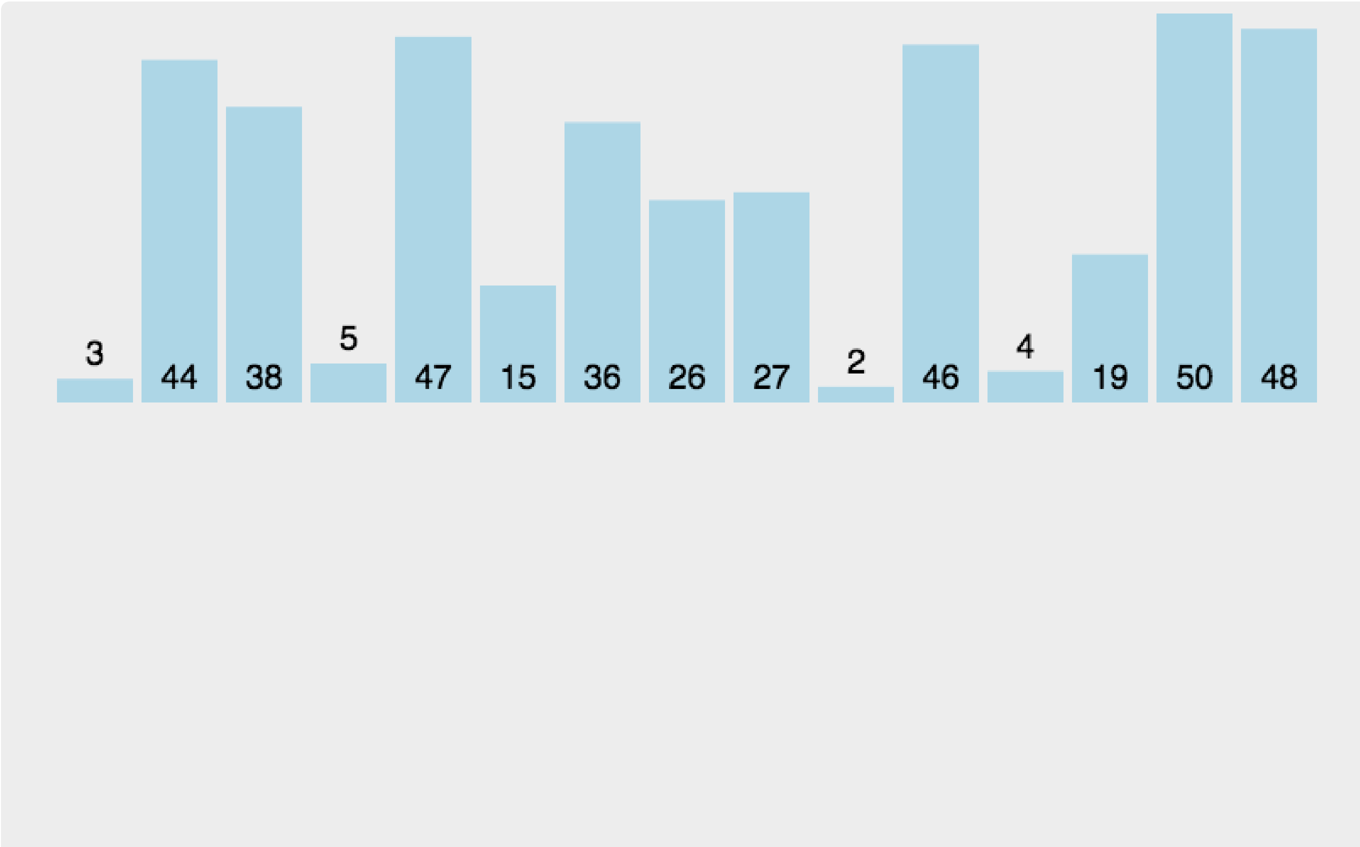
10.2.3. 插入排序

插入排序是一种简单直观的排序算法

它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入

解决思路如下：

- 把待排序的数组分成已排序和未排序两部分，初始的时候把第一个元素认为是已排好序的
- 从第二个元素开始，在已排好序的子数组中寻找到该元素合适的位置并插入该位置（如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面。）
- 重复上述过程直到最后一个元素被插入有序子数组中



10.2.4. 归并排序

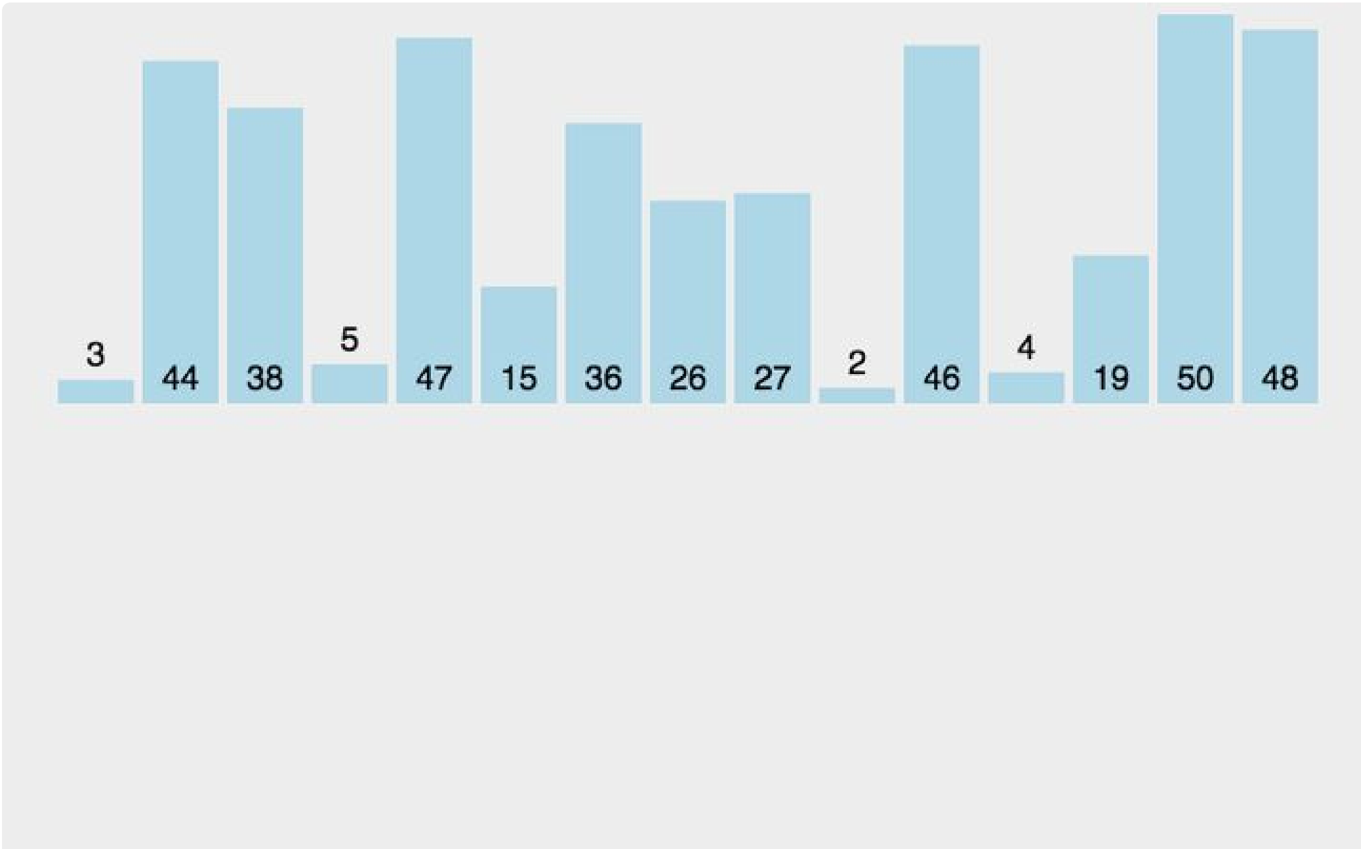
归并排序是建立在归并操作上的一种有效的排序算法

该算法是采用分治法的一个非常典型的应用

将已有序的子序列合并，得到完全有序的序列，即先使每个子序列有序，再使子序列段间有序

解决思路如下：

- 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列
- 设定两个指针，最初位置分别为两个已经排序序列的起始位置
- 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置
- 重复步骤3直到某一指针到达序列尾
- 将另一序列剩下的所有元素直接复制到合并序列尾



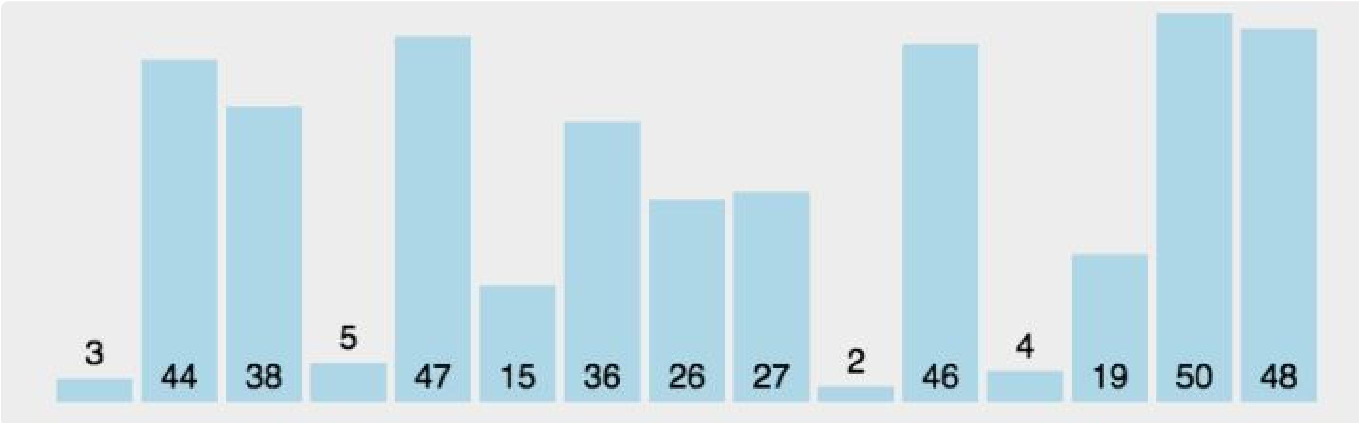
10.2.5. 快速排序

快速排序是对冒泡排序算法的一种改进，基本思想是通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据比另一部分的所有数据要小

再按这种方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，使整个数据变成有序序列

解决思路如下：

- 从数列中挑出一个元素，称为"基准"（pivot）
- 重新排序数列，所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准后面（相同的数可以到任何一边）。在这个分区结束之后，该基准就处于数列的中间位置。这个称为分区（partition）操作
- 递归地（recursively）把小于基准值元素的子数列和大于基准值元素的子数列排序



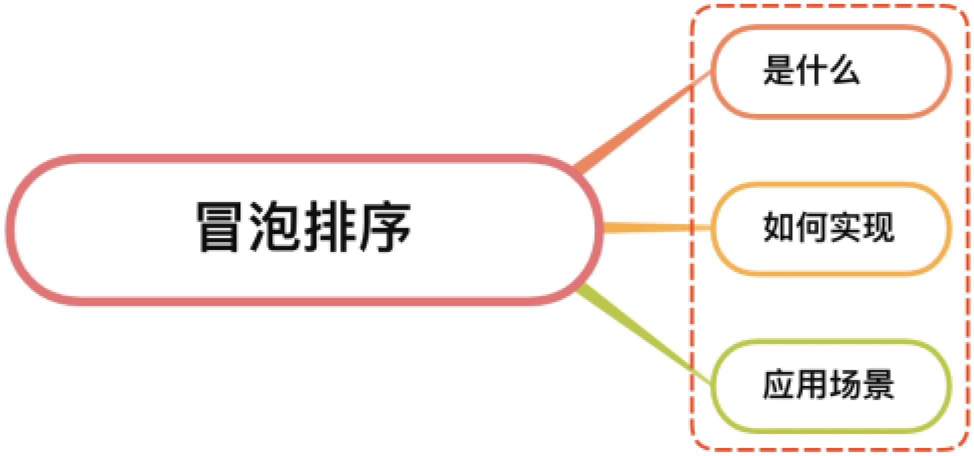
10.3. 区别

除了上述的排序算法之外，还存在其他的排序算法，例如希尔排序、堆排序等等......

区别如下图所示：

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

11. 说说你对冒泡排序的理解？如何实现？应用场景？



11.1. 是什么

冒泡排序（Bubble Sort），是一种计算机科学领域的较简单的排序算法

冒泡排序的思想就是在每次遍历一遍未排序的数列之后，将一个数据元素浮上去（也就是排好了一个数据）

如同碳酸饮料中二氧化碳的气泡最终会上浮到顶端一样，故名“冒泡排序”

假如我们要把 12、35、99、18、76 这 5 个数从大到小进行排序，那么数越大，越需要把它放在前面

思路如下：

- 从后开始遍历，首先比较 18 和 76，发现 76 比 18 大，就把两个数交换顺序，得到 12、35、99、76、18
- 接着比较 76 和 99，发现 76 比 99 小，所以不用交换顺序
- 接着比较 99 和 35，发现 99 比 35 大，交换顺序
- 接着比较 99 和 12，发现 99 比 12 大，交换顺序

最终第 1 趟排序的结果变成了 99、12、35、76、18，如下图所示：