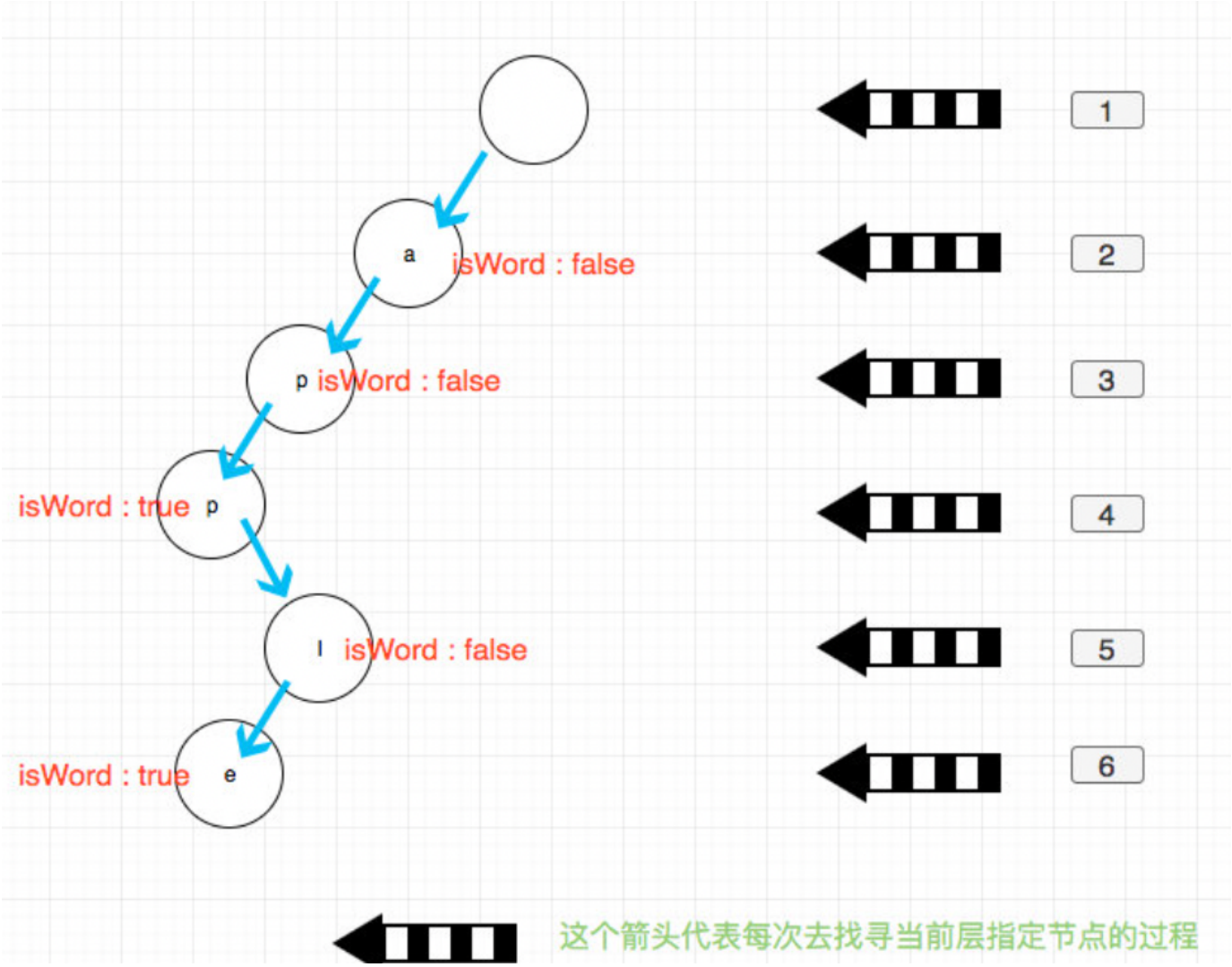
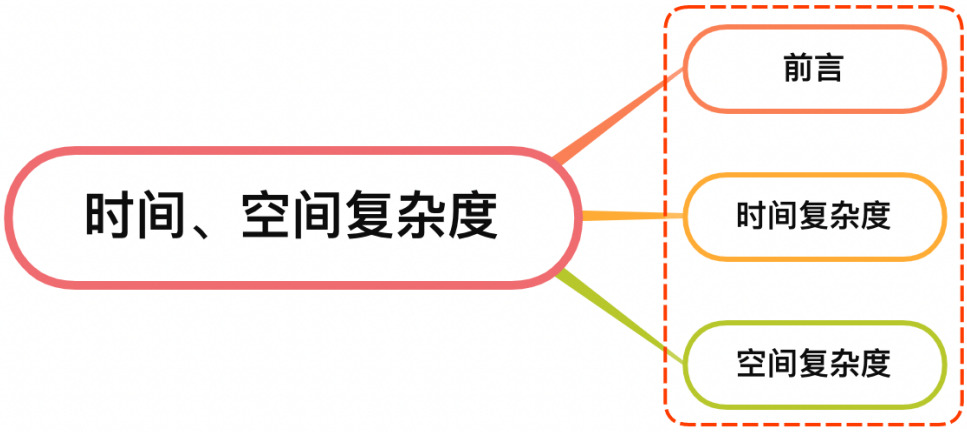


包括前端可能会做一些对字符串进行相似度检测，例如"每日一题"和"js每日一题"两个字符串进行相似度对比，这种情况可以通过“最小编辑距离”算法，如果 `a` 和 `b` 的编辑距离越小，我们认为越相似

日常在编写任何代码的都需要一个良好的算法思维，选择好的算法或者数据结构，能让整个程序效率更高

3. 说说你对算法中时间复杂度，空间复杂度的理解？如何计算？



3.1. 前言

算法（Algorithm）是指用来操作数据、解决程序问题的一组方法。对于同一个问题，使用不同的算法，也许最终得到的结果是一样的，但在过程中消耗的资源和时间却会有很大的区别

衡量不同算法之间的优劣主要是通过**时间**和**空间**两个维度去考量：

- 时间维度：是指执行当前算法所消耗的时间，我们通常用「时间复杂度」来描述。
- 空间维度：是指执行当前算法需要占用多少内存空间，我们通常用「空间复杂度」来描述

通常会遇到一种情况，时间和空间维度不能够兼顾，需要在两者之间取得一个平衡点是我们需要考虑的

一个算法通常存在最好、平均、最坏三种情况，我们一般关注的是最坏情况

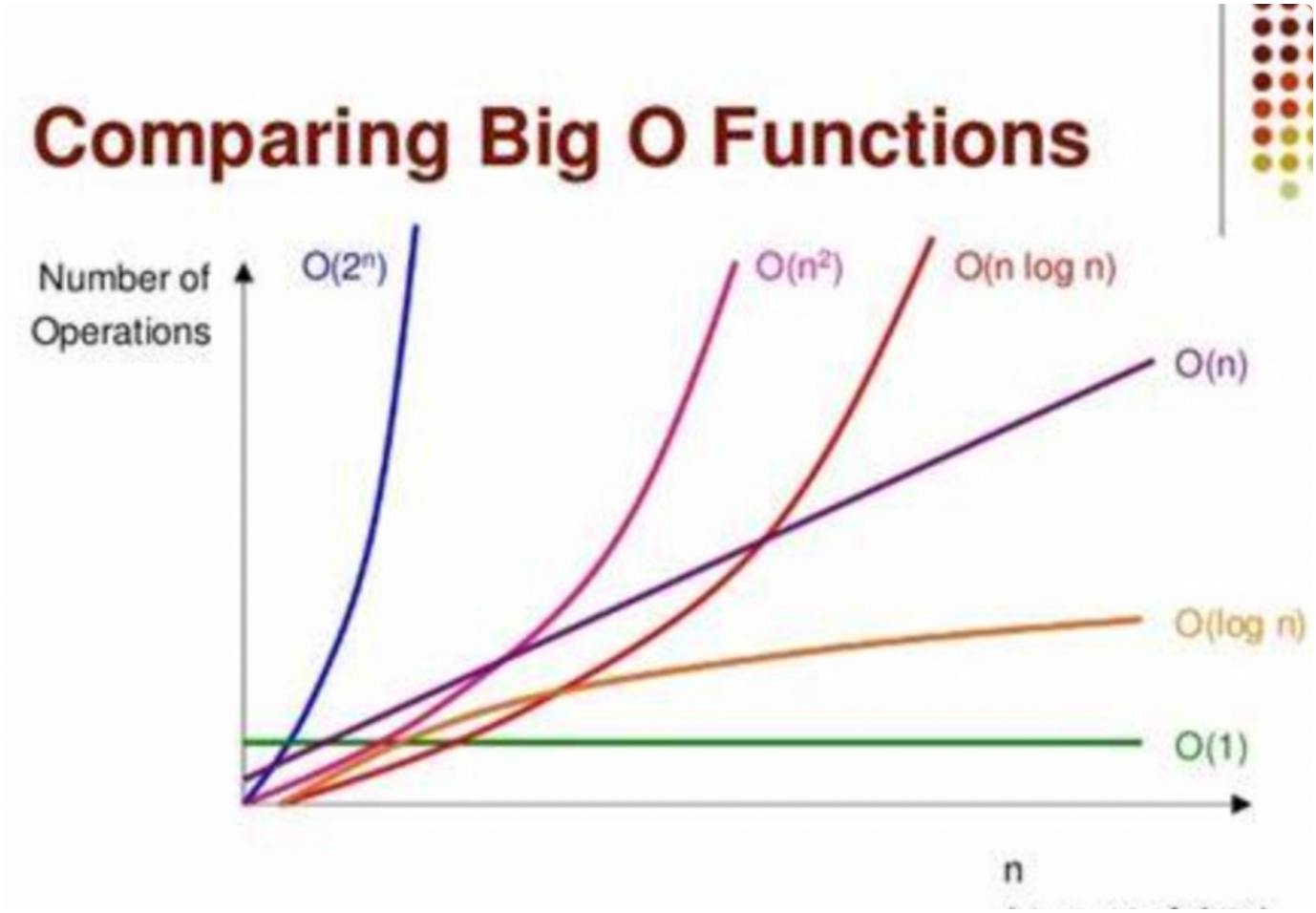
最坏情况是算法运行时间的上界，对于某些算法来说，最坏情况出现的比较频繁，也意味着平均情况和最坏情况一样差

3.2. 时间复杂度

时间复杂度是指执行这个算法所需要的计算工作量，其复杂度反映了程序执行时间「随输入规模增长而增长的量级」，在很大程度上能很好地反映出算法的优劣与否

一个算法花费的时间与算法中语句的「执行次数成正比」，执行次数越多，花费的时间就越多

算法的复杂度通常用大O符号表述，定义为 `T(n) = O(f(n))` ，常见的时间复杂度有：O(1)常数型、O(log n)对数型、O(n)线性型、O(nlogn)线性对数型、O(n$^{2)平方型、O(n}$3)立方型、O(n$^{k)k次方型、O(2}$n)指数型，如下图所示：



从上述可以看到，随着问题规模 `n` 的不断增大，上述时间复杂度不断增大，算法的执行效率越低，由小到大排序如下：

JavaScript | 复制代码

```
O(1)<O(log n)<O(n)<O(nlog n)<O(n2)<O(n3)<…<O(2^n)<O(n!)
```

注意的是，算法复杂度只是描述算法的增长趋势，并不能说一个算法一定比另外一个算法高效，如果常数项过大的时候也会导致算法的执行时间变长

关于如何计算时间复杂度，可以看看如下简单例子：

```javascript
function process(n) {
  let a = 1
  let b = 2
  let sum = a + b
  for(let i = 0; i < n; i++) {
    sum += i
  }
  return sum
}
```

该函数算法需要执行的运算次数用输入大小 `n` 的函数表示，即 `T(n) = 2 + n + 1`，那么时间复杂度为 `O(n + 3)`，又因为时间复杂度只关注最高数量级，且与之系数也没有关系，因此上述的时间复杂度为 `O(n)`

又比如下面的例子：

```javascript
function process(n) {
 let count = 0
  for(let i = 0; i < n; i++){
    for(let i = 0; i < n; i++){
      count += 1
    }
  }
}
```

循环里面嵌套循环，外面的循环执行一次，里面的循环执行 `n` 次，因此时间复杂度为 `O(n*n*1 + 2) = O(n^2)`

对于顺序执行的语句，总的时间复杂度等于其中最大的时间复杂度，如下：

```javascript
function process(n) {
  let sum = 0
  for(let i = 0; i < n; i++) {
    sum += i
  }
  for(let i = 0; i < n; i++){
    for(let i = 0; i < n; i++){
      sum += 1
    }
  }
  return sum
}
```

上述第一部分复杂度为 `O(n)`，第二部分复杂度为 `O(n^2)`，总复杂度为 `max(O(n^2), O(n)) = O(n^2)`

又如下一个例子：

```javascript
function process(n) {
  let i = 1; // ①
  while (i <= n) {
    i = i * 2; // ②
  }
}
```

循环语句中以2的倍数来逼近 `n`，每次都乘以2。如果用公式表示就是1 *2* 2 *2* … 2 <=n，也就是说2的 `x` 次方小于等于 `n` 时会执行循环体，记作 `2^x <= n`，于是得出 `x<=logn`

因此循环在执行 `logn` 次之后，便结束，因此时间复杂度为 `O(logn)`

同理，如果一个 `O(n)` 循环里面嵌套 `O(logn)` 的循环，则时间复杂度为 `O(nlogn)`，
像 `O(n^3)` 无非也就是嵌套了三层 `O(n)` 循环

3.3. 空间复杂度

空间复杂度主要指执行算法所需内存的大小，用于对程序运行过程中所需要的临时存储空间的度量

除了需要存储空间、指令、常数、变量和输入数据外，还包括对数据进行操作的工作单元和存储计算所需信息的辅助空间

下面给出空间复杂度为 `O(1)` 的示例，如下

```JavaScript
let a = 1
let b = 2
let c = 3
```

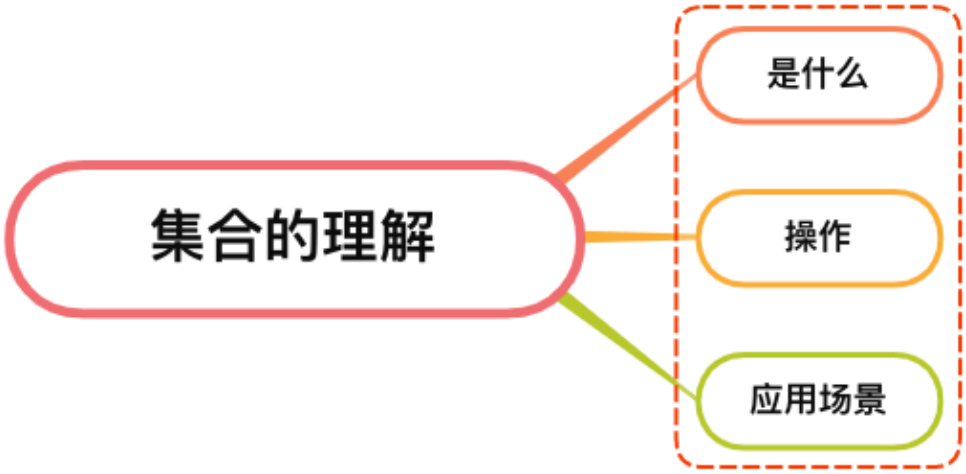
上述代码的临时空间不会随着 `n` 的变化而变化，因此空间复杂度为 `O(1)`

```JavaScript
let arr []
for(i=1; i<=n; ++i){
  arr.push(i)
}
```

上述可以看到，随着 `n` 的增加，数组的占用的内存空间越大

通常来说，只要算法不涉及到动态分配的空间，以及递归、栈所需的空间，空间复杂度通常为 `O(1)`，一个一维数组 `a[n]`，空间复杂度 `O(n)`，二维数组为 `O(n^2)`

4. 说说你对集合的理解？常见的操作有哪些？



4.1. 是什么

集合（Set），指具有某种特定性质的事物的总体，里面的每一项内容称作元素

在数学中，我们经常会遇到集合的概念：

- 有限集合：例如一个班集所有的同学构成的集合

- 无限集合：例如全体自然数集合

在计算机中集合道理也基本一致，具有三大特性：

- 确定性：于一个给定的集合，集合中的元素是确定的。即一个元素，或者属于该集合，或者不属于该集合，两者必居其一
- 无序性：在一个集合中，不考虑元素之间的顺序，只要元素完全相同，就认为是同一个集合
- 互异性：集合中任意两个元素都是不同的

4.2. 操作

在 `ES6` 中，集合本身是一个构建函数 `Set` ，用来生成 `Set` 数据结构，如下：

```JavaScript
const s = new Set();
```

关于集合常见的方法有：

- add()：增
- delete()：删
- has()：改
- clear()：查

4.2.1. add()

添加某个值，返回 `Set` 结构本身

当添加实例中已经存在的元素， `set` 不会进行处理添加

```JavaScript
s.add(1).add(2).add(2); // 2只被添加了一次
```

体现了集合的互异性特性

4.2.2. delete()

删除某个值，返回一个布尔值，表示删除是否成功

```
1  s.delete(1)
```

4.2.3. has()

返回一个布尔值，判断该值是否为 `Set` 的成员

```
1  s.has(2)
```

4.2.4. clear()

清除所有成员，没有返回值

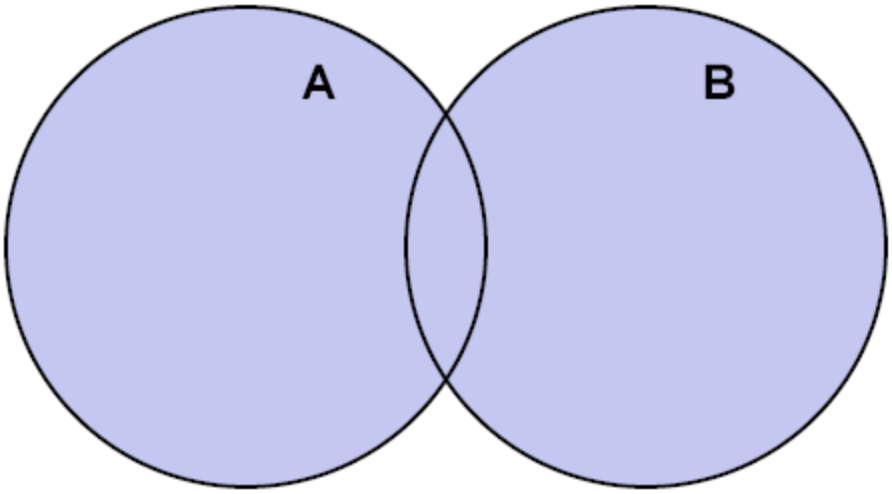
```
1  s.clear()
```

关于多个集合常见的操作有：

- 并集
- 交集
- 差集

4.2.5. 并集

两个集合的共同元素，如下图所示：



代码实现方式如下：

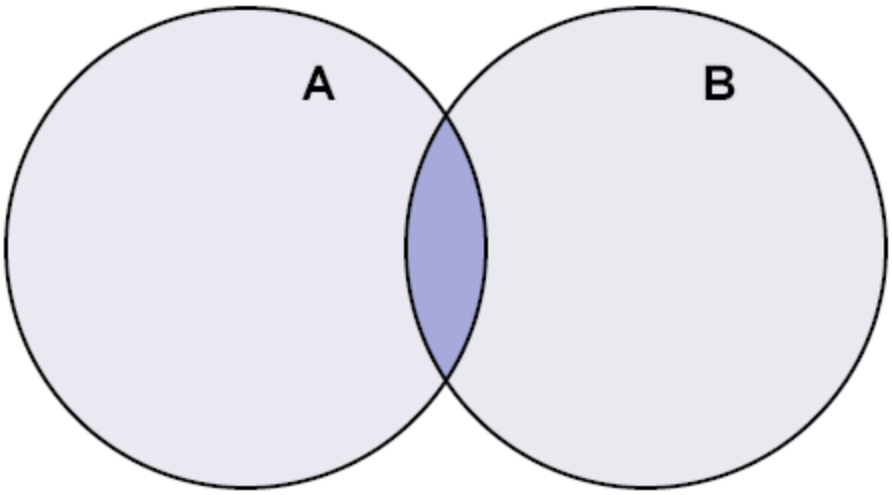
JavaScript | 复制代码

```
let a = new Set([1, 2, 3]);
let b = new Set([4, 3, 2]);

// 并集
let union = new Set([...a, ...b]);
// Set {1, 2, 3, 4}
```

4.2.6. 交集

两个集合 `A` 和 `B`，即属于 `A` 又属于 `B` 的元素，如下图所示：



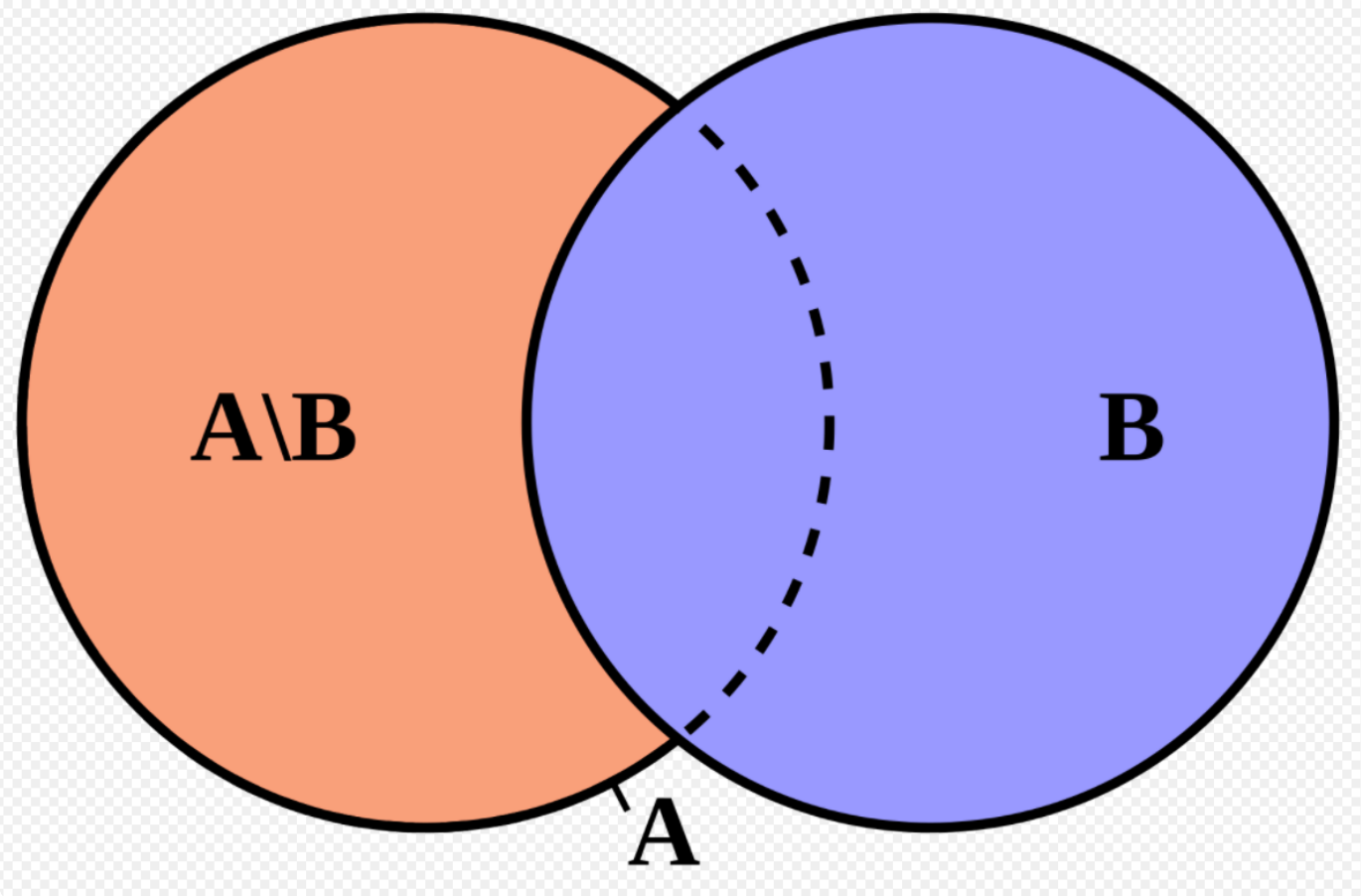
用代码标识则如下：

```javascript
let a = new Set([1, 2, 3]);
let b = new Set([4, 3, 2]);

// 交集
let intersect = new Set([...a].filter(x => b.has(x)));
// set {2, 3}
```

4.2.7. 差集

两个集合 `A` 和 `B`，属于 `A` 的元素但不属于 `B` 的元素称为 `A` 相对于 `B` 的差集，如下图所示：



代码标识则如下：

```javascript
let a = new Set([1, 2, 3]);
let b = new Set([4, 3, 2]);

// （a 相对于 b 的）差集
let difference = new Set([...a].filter(x => !b.has(x)));
// Set {1}
```

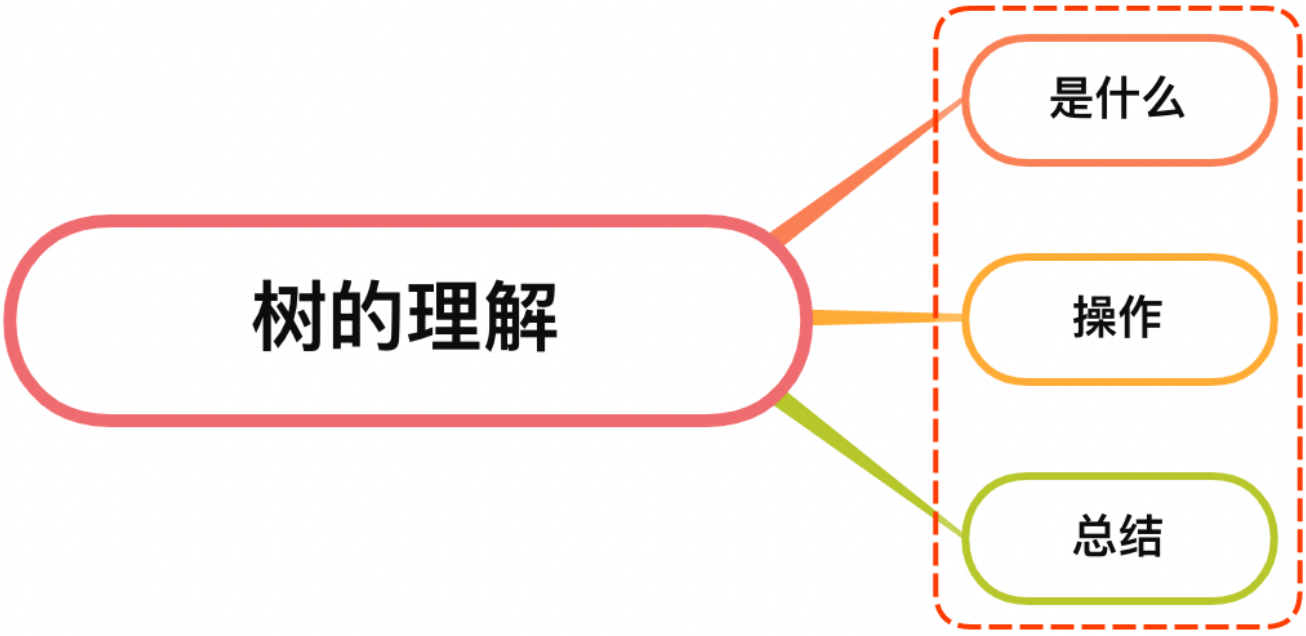
4.3. 应用场景

一般情况下，使用数组的概率会比集合概率高很多

使用 `set` 集合的场景一般是借助其确定性，其本身只包含不同的元素

所以，可以利用 `Set` 的一些原生方法轻松的完成数组去重，查找数组公共元素及不同元素等操作

5. 说说你对树的理解？相关的操作有哪些？



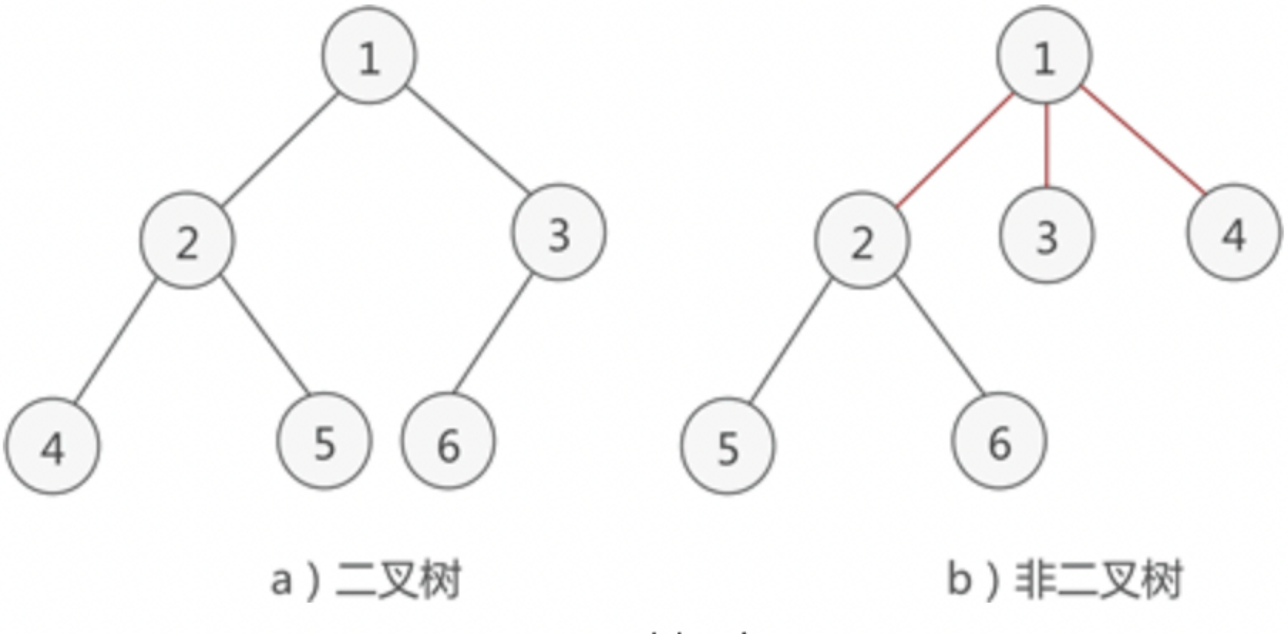
5.1. 是什么

在计算机领域，树形数据结构是一类重要的非线性数据结构，可以表示数据之间一对多的关系。以树与二叉树最为常用，直观看来，树是以分支关系定义的层次结构

二叉树满足以下两个条件：

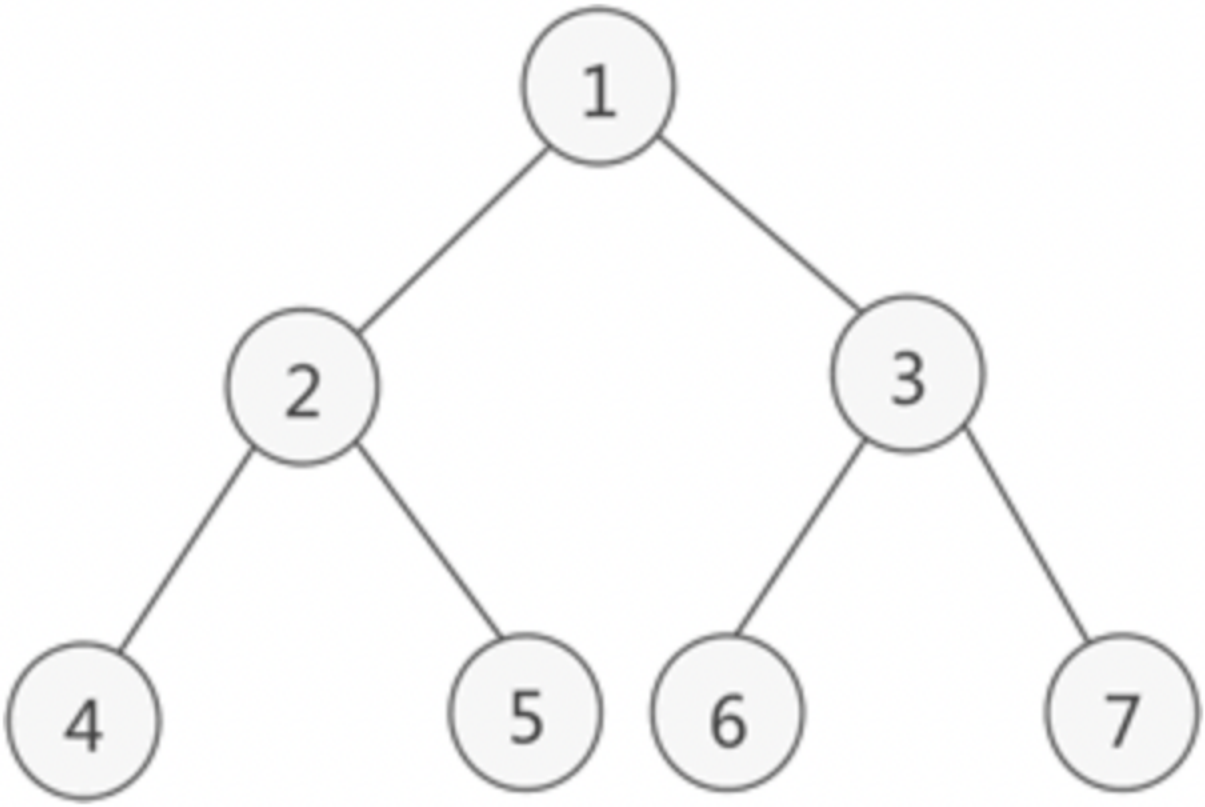
- 本身是有序树
- 树中包含的各个结点的不能超过 2，即只能是 0、1 或者 2

如下图，左侧的为二叉树，而右侧的因为头结点的子结点超过2，因此不属于二叉树：

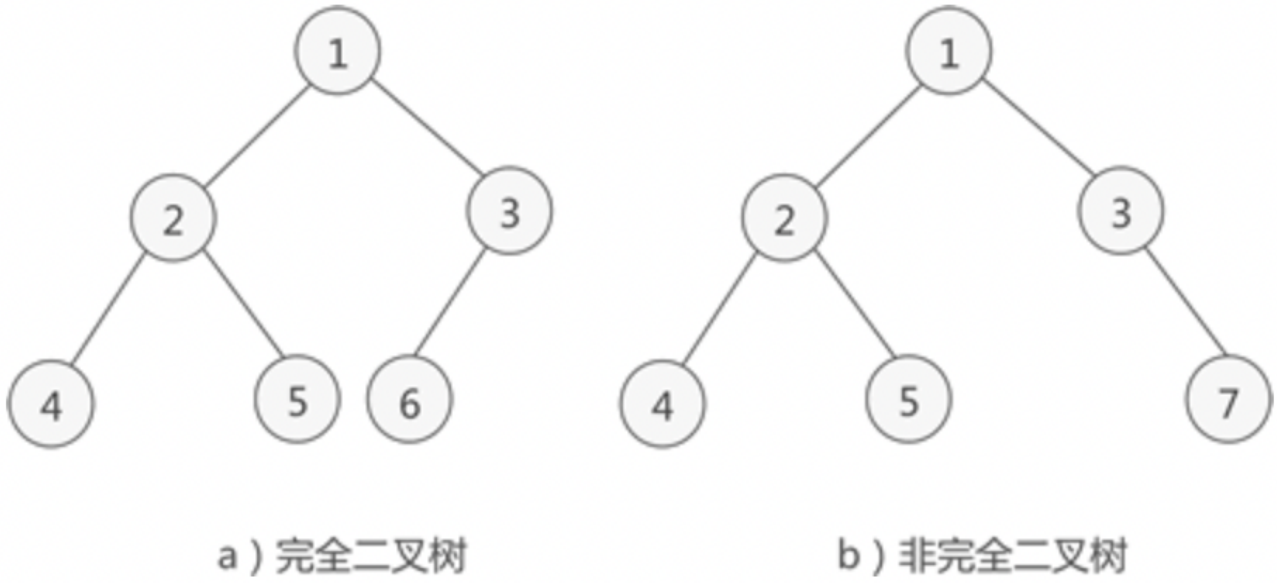


同时，二叉树可以继续进行分类，分成了满二叉树和完成二叉树：

- 满二叉树：如果二叉树中除了叶子结点，每个结点的度都为 2



- 完成二叉树：如果二叉树中除去最后一层节点为满二叉树，且最后一层的结点依次从左到右分布



5.2. 操作

关于二叉树的遍历，常见的有：

- 前序遍历
- 中序遍历
- 后序遍历
- 层序遍历

5.2.1. 前序遍历

前序遍历的实现思想是：

- 访问根节点
- 访问当前节点的左子树
- 若当前节点无左子树，则访问当前节点的右子

根据遍历特性，递归版本用代码表示则如下：

```javascript
const preOrder = (root) => {
  if(!root){ return }
  console.log(root)
  preOrder(root.left)
  preOrder(root.right)
}
```

如果不使用递归版本，可以借助栈先进后出的特性实现，先将根节点压入栈，再分别压入右节点和左节点，直到栈中没有元素，如下：

```javascript
const preOrder = (root) => {
  if(!root){ return }
  const stack = [root]
  while (stack.length) {
    const n = stack.pop()
    console.log(n.val)
    if (n.right) {
      stack.push(n.right)
    }
    if (n.left) {
      stack.push(n.left)
    }
  }
}
```

5.2.2. 中序遍历

前序遍历的实现思想是：

- 访问当前节点的左子树
- 访问根节点
- 访问当前节点的右子

递归版本很好理解，用代码表示则如下：

```javascript
const inOrder = (root) => {
  if (!root) { return }
  inOrder(root.left)
  console.log(root.val)
  inOrder(root.right)
}
```

非递归版本也是借助栈先进后出的特性，可以一直首先一直压入节点的左元素，当左节点没有后，才开始进行出栈操作，压入右节点，然后有依次压入左节点，如下：

```javascript
const inOrder = (root) => {
  if (!root) { return }
  const stack = [root]
  let p = root
  while(stack.length || p){
    while (p) {
      stack.push(p)
      p = p.left
    }
    const n = stack.pop()
    console.log(n.val)
    p = n.right
  }
}
```

5.2.3. 后序遍历

前序遍历的实现思想是：

- 访问当前节点的左子树
- 访问当前节点的右子
- 访问根节点

递归版本，用代码表示则如下：

```javascript
const postOrder = (root) => {
  if (!root) { return }
  postOrder(root.left)
  postOrder(root.right)
  console.log(n.val)
 }
```

后序遍历非递归版本实际根全序遍历是逆序关系，可以再多创建一个栈用来进行输出，如下：

```javascript
const preOrder = (root) => {
  if(!root){ return }
  const stack = [root]
  const outPut = []
  while (stack.length) {
    const n = stack.pop()
    outPut.push(n.val)
    if (n.right) {
      stack.push(n.right)
    }
    if (n.left) {
      stack.push(n.left)
    }
  }
  while (outPut.length) {
    const n = outPut.pop()
    console.log(n.val)
  }
}
```

5.2.4. 层序遍历

按照二叉树中的层次从左到右依次遍历每层中的结点

借助队列先进先出的特性，从树的根结点开始，依次将其左孩子和右孩子入队。而后每次队列中一个结点出队，都将其左孩子和右孩子入队，直到树中所有结点都出队，出队结点的先后顺序就是层次遍历的最终结果

用代码表示则如下：

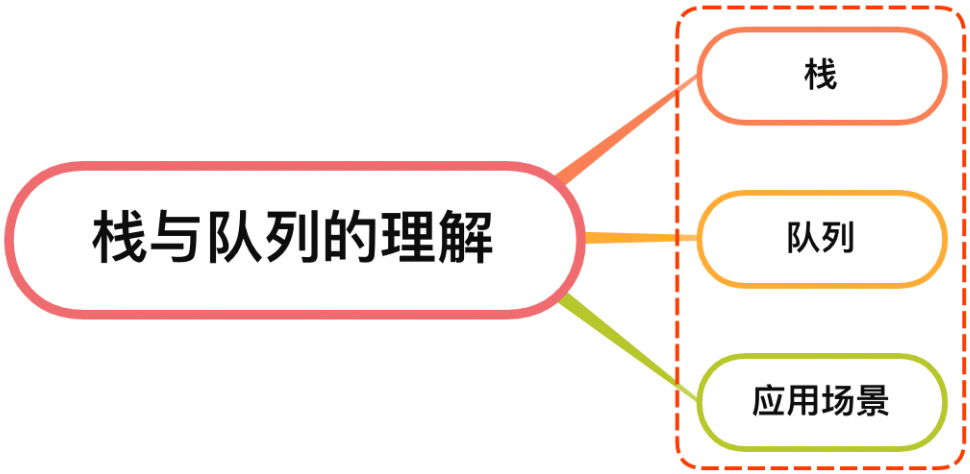
```javascript
const levelOrder = (root) => {
    if (!root) { return [] }
    const queue = [[root, 0]]
    const res = []
    while (queue.length) {
        const n = queue.shift()
        const [node, leval] = n
        if (!res[leval]) {
            res[leval] = [node.val]
        } else {
            res[leval].push(node.val)
        }
        if (node.left) { queue.push([node.left, leval + 1]) }
        if (node.right) { queue.push([node.right, leval + 1]) }
    }
    return res
};
```

5.3. 总结

树是一个非常重要的非线性结构，其中二叉树以二叉树最常见，二叉树的遍历方式可以分成前序遍历、中序遍历、后序遍历

同时，二叉树又分成了完成二叉树和满二叉树

6. 说说你对栈、队列的理解？应用场景？



6.1. 栈

栈（stack）又名堆栈，它是一种运算受限的线性表，限定仅在表尾进行插入和删除操作的线性表

表尾这一端被称为栈顶，相反地另一端被称为栈底，向栈顶插入元素被称为进栈、入栈、压栈，从栈顶删除元素又称作出栈

所以其按照先进后出的原则存储数据，先进入的数据被压入栈底，最后的数据在栈顶，需要读数据的时候从栈顶开始弹出数据，具有记忆作用

关于栈的简单实现，如下：

JavaScript | 复制代码

```javascript
class Stack {
  constructor() {
    this.items = [];
  }

  /**
   * 添加一个（或几个）新元素到栈顶
   * @param {*} element 新元素
   */
  push(element) {
    this.items.push(element)
  }

  /**
   * 移除栈顶的元素，同时返回被移除的元素
   */
  pop() {
    return this.items.pop()
  }

  /**
   * 返回栈顶的元素，不对栈做任何修改（这个方法不会移除栈顶的元素，仅仅返回它）
   */
  peek() {
    return this.items[this.items.length - 1]
  }

  /**
   * 如果栈里没有任何元素就返回true,否则返回false
   */
  isEmpty() {
    return this.items.length === 0
  }

  /**
   * 移除栈里的所有元素
   */
  clear() {
    this.items = []
  }

  /**
   * 返回栈里的元素个数。这个方法和数组的length属性很类似
   */
  size() {
```

```
        return this.items.length
    }
}
```

关于栈的操作主要的方法如下：

- push：入栈操作
- pop：出栈操作

6.2. 队列

跟栈十分相似，队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作

进行插入操作的端称为队尾，进行删除操作的端称为队头，当队列中没有元素时，称为空队列

在队列中插入一个队列元素称为入队，从队列中删除一个队列元素称为出队。因为队列只允许在一端插入，在另一端删除，所以只有最早进入队列的元素才能最先从队列中删除，故队列又称为先进先出

简单实现一个队列的方式，如下：

JavaScript | 复制代码

```javascript
class Queue {
    constructor() {
        this.list = []
        this.frontIndex = 0
        this.tailIndex = 0
    }
    enqueue(item) {
        this.list[this.tailIndex++] = item
    }
    unqueue() {
        const item  = this.list[this.frontIndex]
        this.frontIndex++
        return item
    }
}
```

上述这种入队和出队操作中，头尾指针只增加不减小，致使被删元素的空间永远无法重新利用

当队列中实际的元素个数远远小于向量空间的规模时，也可能由于尾指针已超越向量空间的上界而不能做入队操作，出该现象称为"假溢"

在实际使用队列时，为了使队列空间能重复使用，往往对队列的使用方法稍加改进：

无论插入或删除，一旦 `rear` 指针增1或 `front` 指针增1 时超出了所分配的队列空间，就让它指向这片连续空间的起始位置，这种队列也就是循环队列

下面实现一个循环队列，如下：

JavaScript | 复制代码

```javascript
class Queue {
    constructor(size) {
        this.size = size; // 长度需要限制，来达到空间的利用，代表空间的长度
        this.list = [];
        this.font = 0; // 指向首元素
        this.rear = 0;  // 指向准备插入元素的位置
    }
    enQueue() {
        if (this.isFull() == true) {
            return false
        }
        this.rear = this.rear % this.k;
        this._data[this.rear++] = value;
        return true
    }
    deQueue() {
        if(this.isEmpty()){
            return false;
        }
        this.font++;
        this.font = this.font % this.k;
        return true;
    }
    isEmpty() {
        return this.font == this.rear - 1;
    }
    isFull() {
        return this.rear % this.k == this.font;
    }
}
```

上述通过求余的形式代表首尾指针增1 时超出了所分配的队列空间

6.3. 应用场景

6.3.1. 栈

借助栈的先进后出的特性，可以简单实现一个逆序数处的功能，首先把所有元素依次入栈，然后把所有元素出栈并输出

包括编译器的在对输入的语法进行分析的时候，例如 `"()"` 、 `"{}"` 、 `"[]"` 这些成对出现的符号，借助栈的特性，凡是遇到括号的前半部分，即把这个元素入栈，凡是遇到括号的后半部分就比对栈顶元素是否该元素相匹配，如果匹配，则前半部分出栈，否则就是匹配出错

包括函数调用和递归的时候，每调用一个函数，底层都会进行入栈操作，出栈则返回函数的返回值

生活中的例子，可以把乒乓球盒比喻成一个堆栈，球一个一个放进去（入栈），最先放进去的要等其后面的全部拿出来后才能出来（出栈），这种就是典型的先进后出模型

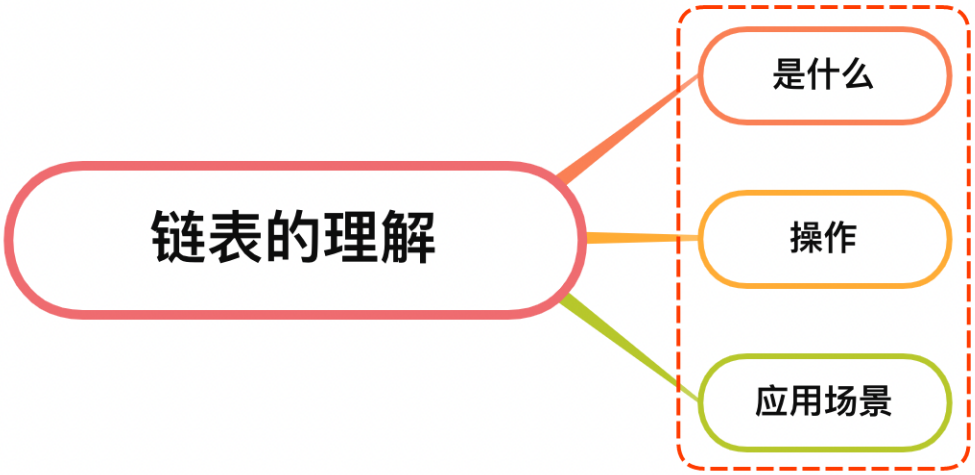
6.3.2. 队列

当我们需要按照一定的顺序来处理数据，而该数据的数据量在不断地变化的时候，则需要队列来帮助解题

队列的使用广泛应用在广度优先搜索种，例如层次遍历一个二叉树的节点值（后续将到）

生活中的例子，排队买票，排在队头的永远先处理，后面的必须等到前面的全部处理完毕再进行处理，这也是典型的先进先出模型

7. 说说你对链表的理解？常见的操作有哪些？



7.1. 是什么

链表（Linked List）是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的，由一系列结点（链表中每一个元素称为结点）组成