链表结构（LinkedList）

王红元 coderwhy

目录
content

链表以及数组的缺点

- **链表和数组一样，可以用于存储一系列的元素，但是链表和数组的实现机制完全不同。**
- **这一章中，我们就来学习一下另外一种非常常见的用于存储数据的线性结构：链表。**

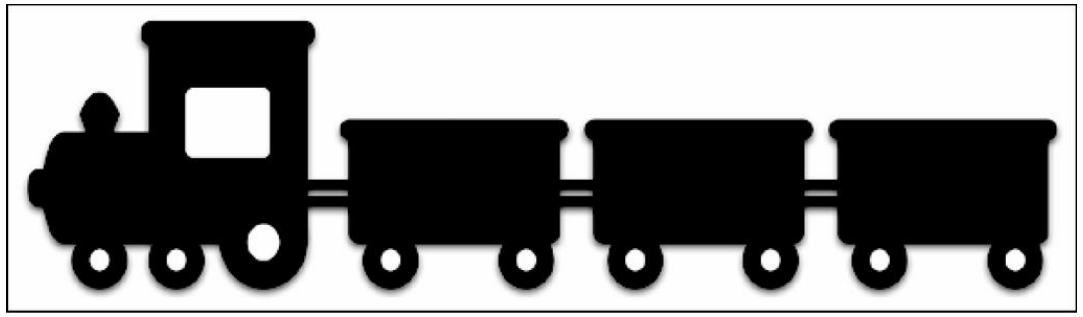
- **数组**:
  - 要存储多个元素，数组（或选择链表）可能是最常用的数据结构。
  - 我们之前说过，几乎每一种编程语言都有默认实现数组结构。
- **但是数组也有很多缺点：**
  - 数组的创建通常需要申请一段连续的内存空间(一整块的内存)，并且大小是固定的(大多数编程语言数组都是固定的)，所以当当前数组不能满足容量需求时，需要扩容。 (一般情况下是申请一个更大的数组，比如2倍。 然后将原数组中的元素复制过去)
  - 而且在数组开头或中间位置插入数据的成本很高，需要进行大量元素的位移。
  - 尽管JavaScript的Array底层可以帮我们做这些事，但背后的原理依然是这样。

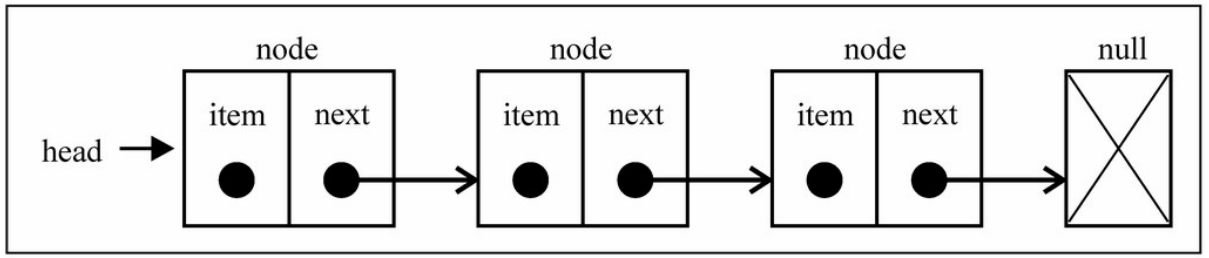
链表的优势

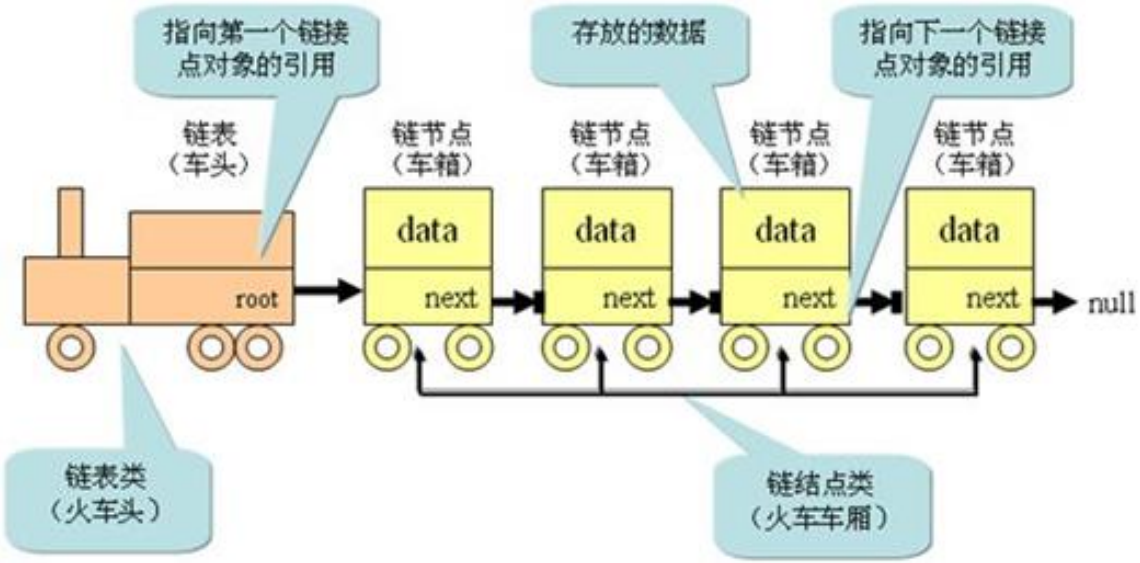
- **要存储多个元素，另外一个选择就是链表。**
- **但不同于数组，链表中的元素在内存中不必是连续的空间。**
  - 链表的每个元素由一个存储元素本身的节点和一个指向下一个元素的引用(有些语言称为指针或者链接)组成。
- **相对于数组，链表有一些优点：**
  - 内存空间不是必须连续的。
    - ✓ 可以充分利用计算机的内存，实现灵活的内存动态管理。
  - 链表不必在创建时就确定大小，并且大小可以无限的延伸下去。
  - 链表在插入和删除数据时，时间复杂度可以达到O(1)。
    - ✓ 相对数组效率高很多。
- **相对于数组，链表有一些缺点：**
  - 链表访问任何一个位置的元素时，都需要从头开始访问。(无法跳过第一个元素访问任何一个元素)。
  - 无法通过下标直接访问元素，需要从头一个个访问，直到找到对应的元素。

链表到底是什么?

- 什么是链表呢?
  - 其实上面我们已经简单的提过了链表的结构，我们这里更加详细的分析一下。
  - 链表类似于火车：有一个火车头，火车头会连接一个节点，节点上有乘客(类似于数据)，并且这个节点会连接下一个节点，以此类推。
- 链表的火车结构:







链表结构的封装

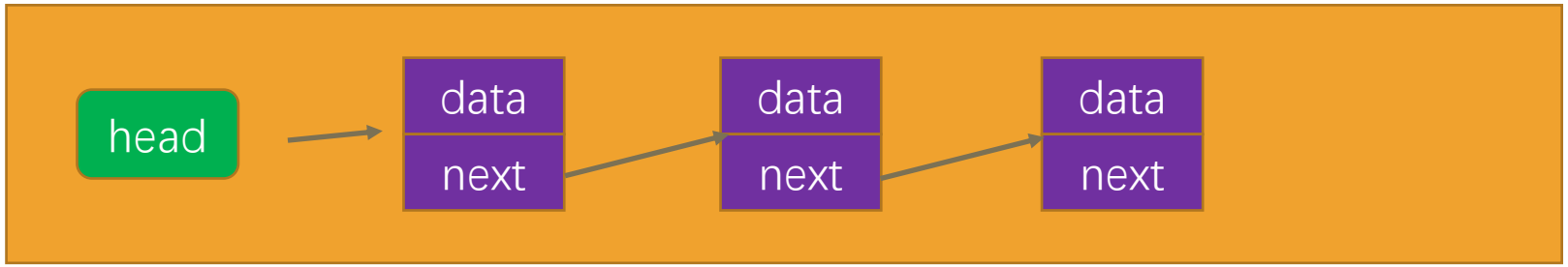
- 我们先来创建一个链表类

```
class Node<T> {
  value: T
  next: Node<T> | null

  constructor(value: T) {
    this.value = value
    this.next = null
  }
}
```

```
class LinkedList<T> {
  head: Node<T> | null = null
  size: number = 0

  get length() {
    return this.size
  }
}
```

- 代码解析：
  - 封装一个Node类，用于封装每一个节点上的信息（包括值和指向下一个节点的引用），它是一个泛型类。
  - 封装一个LinkedList类，用于表示我们的链表结构。 (和Java中的链表同名，不同Java中的这个类是一个双向链表，在第二阶段中我们也会实现双向链表结构)。
  - 链表中我们保存两个属性，一个是链表的长度，一个是链表中第一个节点。
  - 当然，还有很多链表的操作方法。 我们放在下一节中学习。

链表图片准备



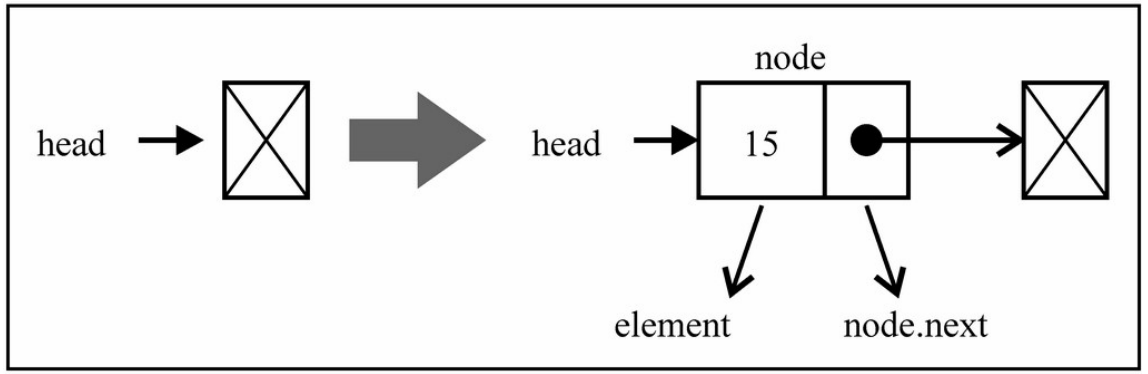
链表常见操作

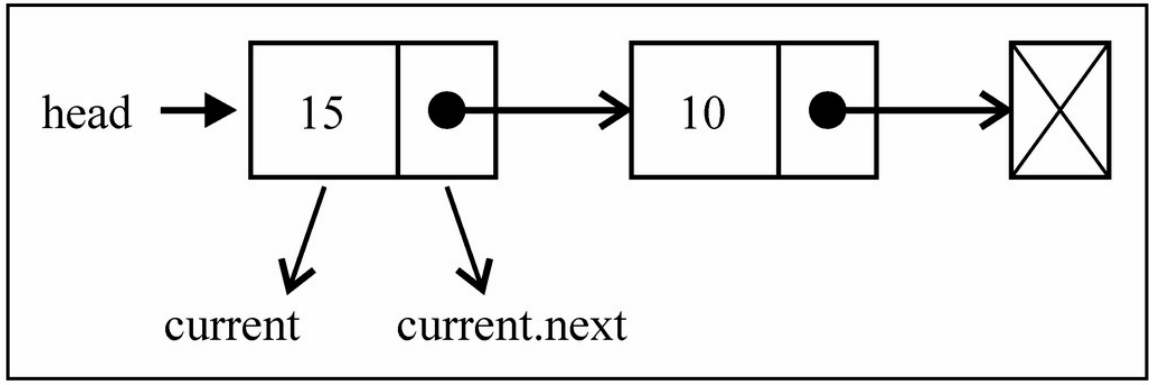
- **我们先来认识一下，链表中应该有哪些常见的操作**
  - append(element)：向链表尾部添加一个新的项
  - insert(position，element)：向链表的特定位置插入一个新的项。
  - get(position)：获取对应位置的元素
  - indexOf(element)：返回元素在链表中的索引。如果链表中没有该元素则返回-1。
  - update(position，element)：修改某个位置的元素
  - removeAt(position)：从链表的特定位置移除一项。
  - remove(element)：从链表中移除一项。
  - isEmpty()：如果链表中不包含任何元素，返回true，如果链表长度大于0则返回false。
  - size()：返回链表包含的元素个数。与数组的length属性类似。

- **整体你会发现操作方法和数组非常类似，因为链表本身就是一种可以代替数组的结构。**

append方法

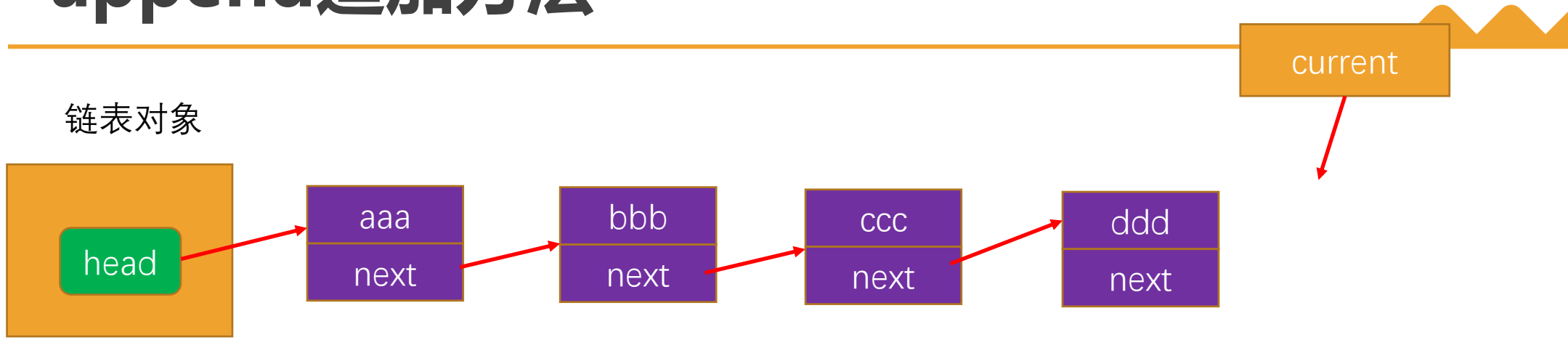
- 向链表尾部追加数据可能有两种情况：
  - 链表本身为空，新添加的数据时唯一的节点。
  - 链表不为空，需要向其他节点后面追加节点。






```
append(element: T) {
  const newNode = new Node(element)
  if (!this.head) { // head为null
    this.head = newNode
  } else { // head不为null
    // 创建一个current指针
    let current = this.head
    while (current.next) {
      current = current.next
    }
    current.next = newNode
  }

  this.size++
}
```

append追加方法



链表的遍历方法（traverse）

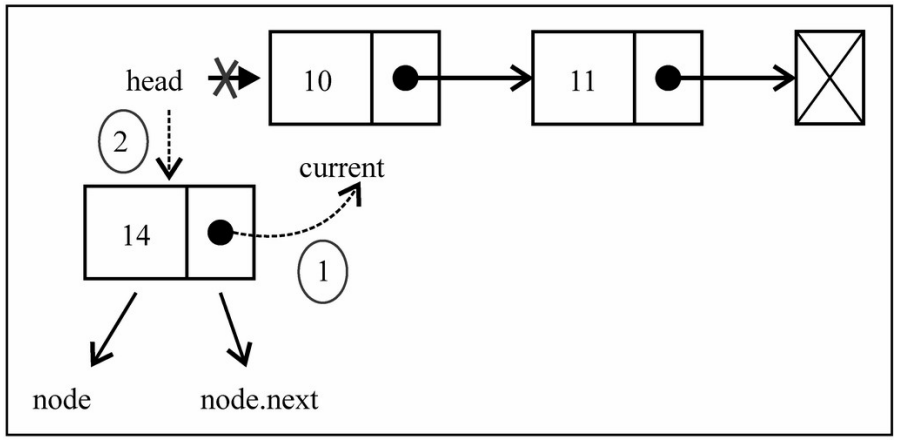
- **为了可以方便的看到链表上的每一个元素，我们实现一个遍历链表每一个元素的方法：**
  - 这个方法首先将当前结点设置为链表的头结点。
  - 然后，在while循环中，我们遍历链表并打印当前结点的数据。
  - 在每次迭代中，我们将当前结点设置为其下一个结点，直到遍历完整个链表。

```
traverse() {
  const values: T[] = []
  let current = this.head
  while (current) {
    values.push(current.value)
    current = current.next
  }
  console.log(values.join('->'))
}
```

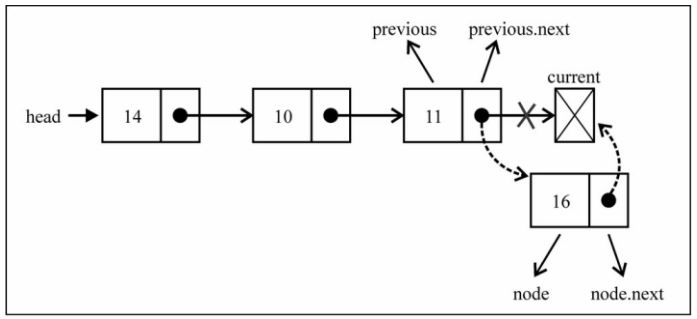
insert方法

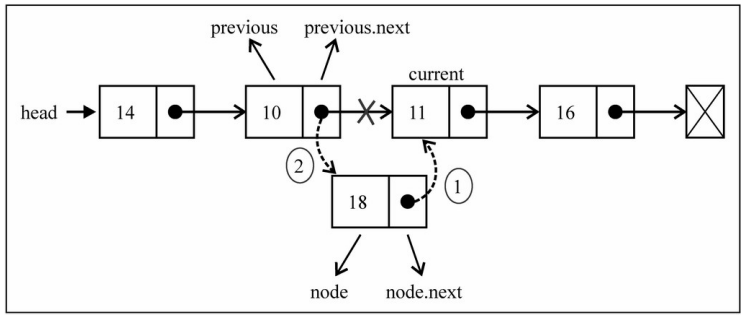
- 接下来实现另外一个添加数据的方法：在任意位置插入数据。

- 添加到第一个位置：
  - 添加到第一个位置，表示新添加的节点是头，就需要将原来的头节点，作为新节点的next
  - 另外这个时候的head应该指向新节点。

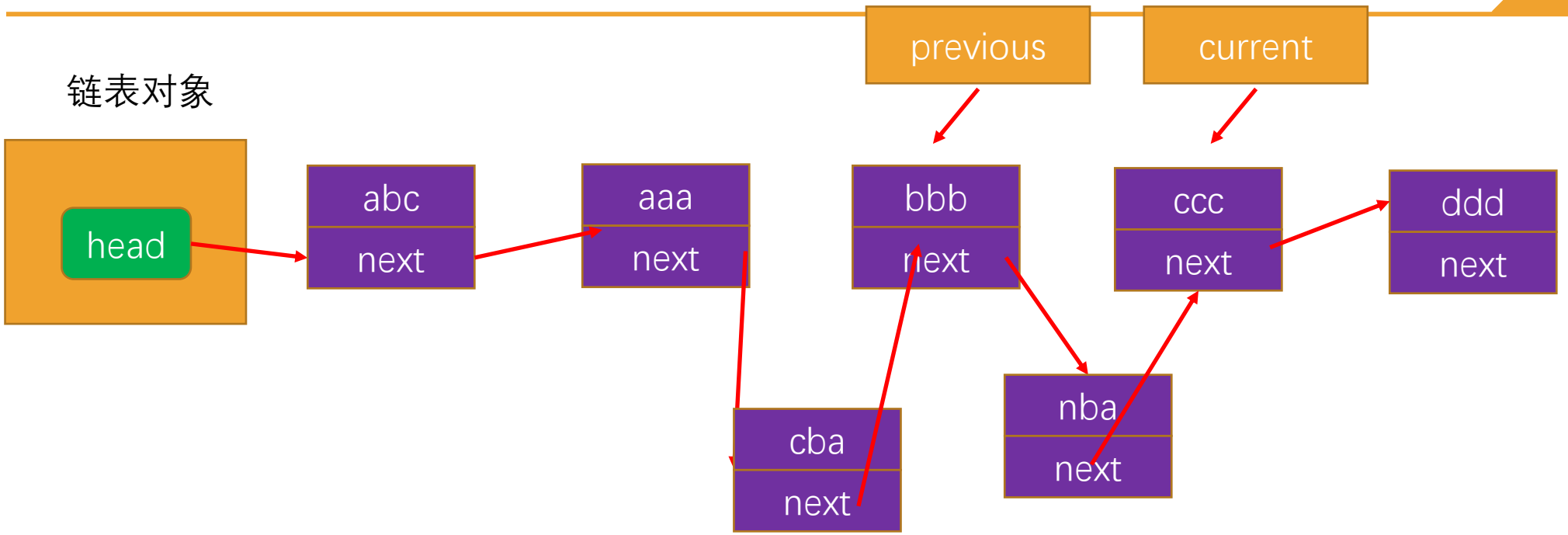


- 添加到其他位置：
  - 如果是添加到其他位置，就需要先找到这个节点位置了。
  - 我们通过while循环，一点点向下找。 并且在这个过程中保存上一个节点和下一个节点。
  - 找到正确的位置后，将新节点的next指向下一个节点，将上一个节点的next指向新的节点。





insert方法的过程



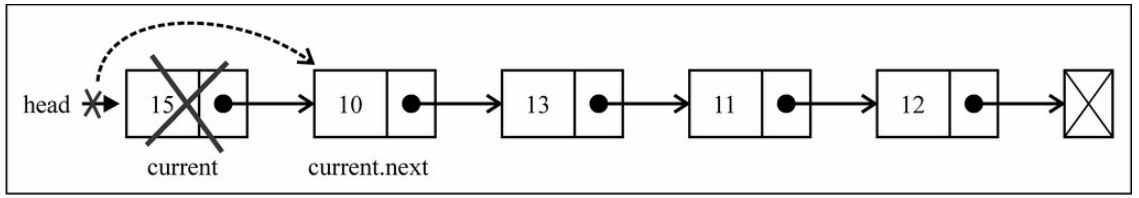
coderwhy

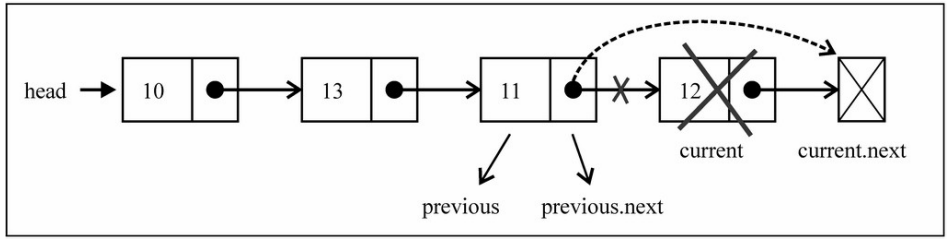
insert方法图解实现

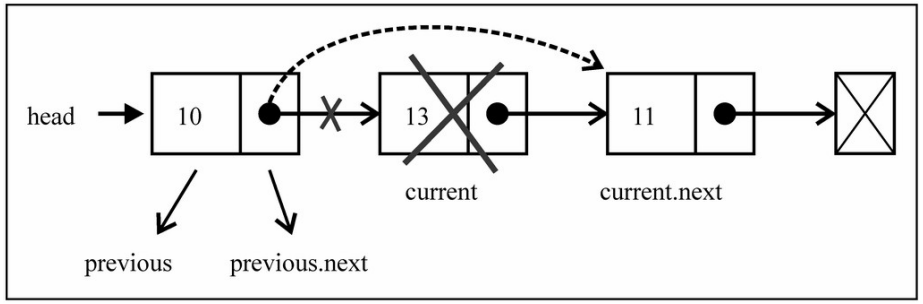
```
insert(element: T, position: number): boolean {
    // 1.判断是否position越界
    if (position < 0 || position > this.size) return false

    // 2.创建新的节点
    const newNode = new Node(element)
    let current = this.head

    // 3.是否是第一个位置
    if (position === 0) {
        newNode.next = current
        this.head = newNode
    } else {
        let index = 0
        let previous: Node<T> | null = null
        while (index++ < position && current) {
            previous = current
            current = current.next
        }

        newNode.next = current
        previous!.next = newNode
    }
    this.size++

    return true
}
```

removeAt方法

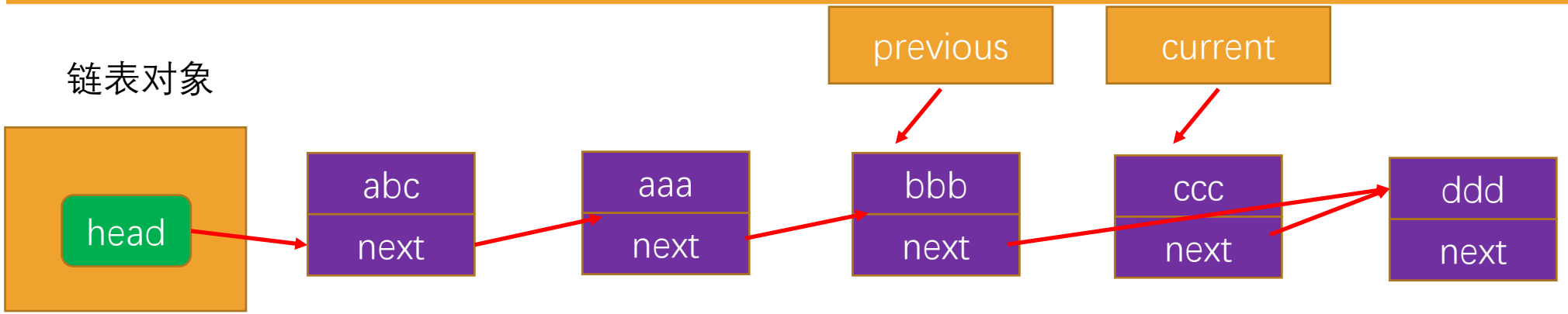
- **移除数据有两种常见的方式：**
  - 根据位置移除对应的数据
  - 根据数据，先找到对应的位置，再移除数据
- **移除第一项的信息：**
  - 移除第一项时，直接让head指向第二项信息就可以啦。
  - 那么第一项信息没有引用指向，就在链表中不再有效，后面会被回收掉。
- **移除其他项的信息：**
  - 移除其他项的信息操作方式是相同的。
  - 首先，我们需要通过while循环，找到正确的位置。
  - 找到正确位置后，就可以直接将上一项的next指向current项的next，这样中间的项就没有引用指向它，也就不再存在于链表后，会面会被回收掉。







removeAt方法的过程



removeAt方法图解实现

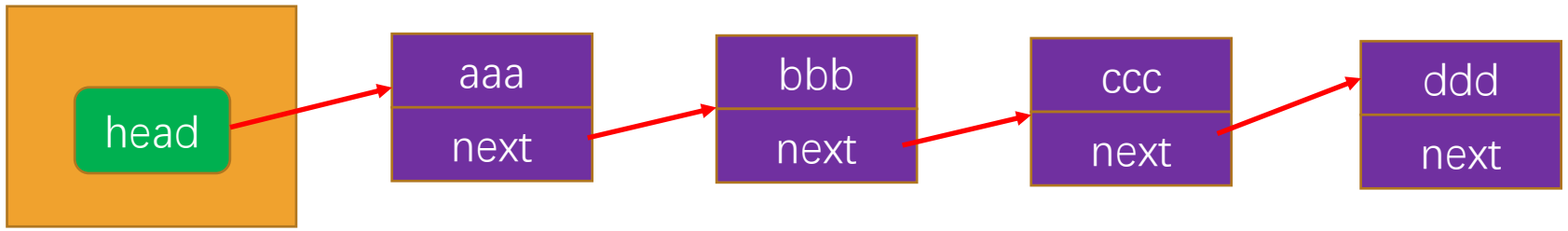
```
removeAt(position: number): T | null {
  // 1.检测越界问题
  if (position < 0 || position >= this.length) return null

  // 2.定义变量
  let current = this.head
  let previous: Node<T>|null = null

  if (position === 0) {
    this.head = current?.next ?? null
  } else {
    let index = 0
    while (index++ < position && current) {
      previous = current
      current = current.next
    }
    previous!.next = current?.next ?? null
  }

  this.size--

  return current!.value
}
```

get方法

- get(position)：获取对应位置的元素

```
get(position: number): T | null {
  // 1.判断是否position越界
  if (position < 0 || position > this.size) return null

  // 2.查找元素
  let current = this.head
  let index = 0
  while (index++ < position && current) {
    current = current.next
  }

  return current?.value ?? null
}
```

链表对象



遍历结点的操作重构

- 因为遍历结点的操作我们需要经常来做，所以可以进行如下的重构：

```
private getNode(position: number): Node<T> | null {
  let index = 0
  let current = this.head
  while (index++ < position && current) {
    current = current.next
  }
  return current
}
```

- 其他需要获取节点的，可以调用上面的方法，比如操作操作：

```
// 3.是否是第一个位置
if (position === 0) {
  newNode.next = this.head
  this.head = newNode
} else {
  const previous = this.getNode(position - 1)
  newNode.next = previous!.next
  previous!.next = newNode
}
this.size++
```

update方法

- update(position，element)：修改某个位置的元素

```
update(element: T, position: number) {
  // 1.判断是否position越界
  if (position < 0 || position >= this.size) return

  // 2.获取结点
  const current = this.getNode(position)
  if (current) {
    current.value = element
  }
}
```

indexOf方法

- 我们来完成另一个功能：根据元素获取它在链表中的位置

```
// 其他方法
indexOf(element: T): number {
  let current = this.head
  let index = 0
  while (current) {
    if (current.value === element) {
      return index
    }
    index++
    current = current.next
  }
  return -1
}
```

remove方法以及其他方法

- 有了上面的indexOf方法，我们可以非常方便实现根据元素来删除信息

```
remove(element: T) {
  const index = this.indexOf(element)
  return this.removeAt(index)
}
```

- isEmpty方法

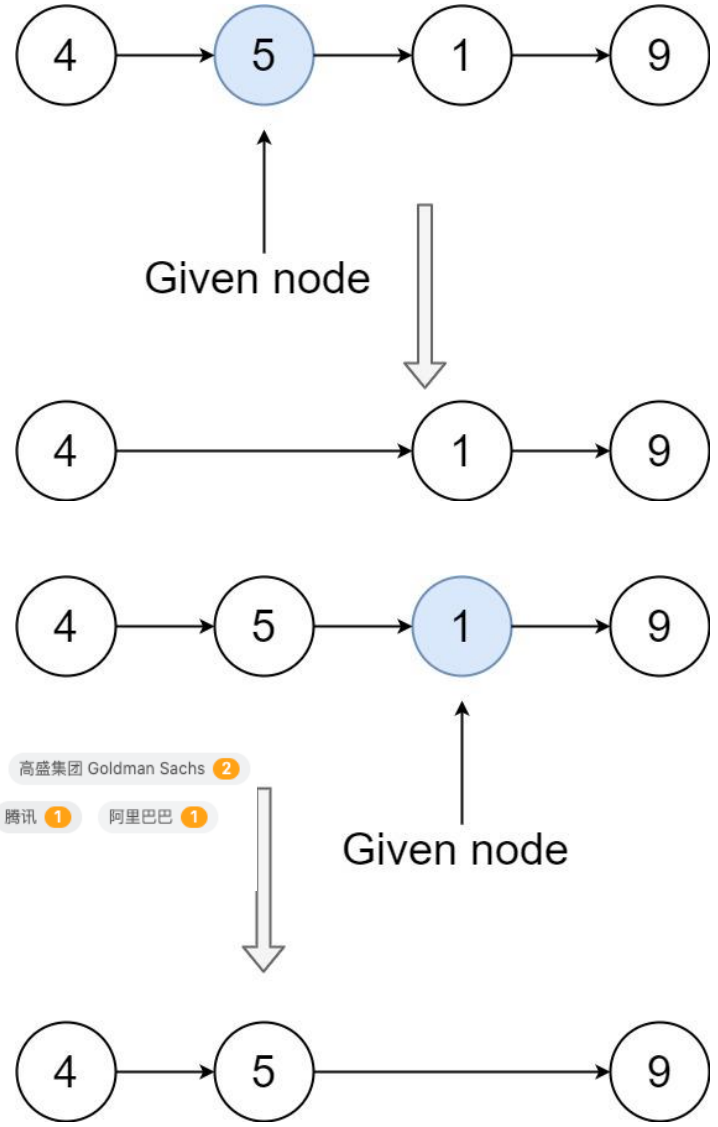
```
isEmpty() {
  return this.size === 0
}
```

707. 设计链表 -字节、腾讯等公司面试题

- **707. 设计链表**
  - https://leetcode.cn/problems/design-linked-list/
- **设计链表的实现。**
  - 您可以选择使用单链表或双链表。
  - 单链表中的节点应该具有两个属性：val 和 next。val 是当前节点的值，next 是指向下一个节点的指针/引用。
  - 如果要使用双向链表，则还需要一个属性 prev 以指示链表中的上一个节点。假设链表中的所有节点都是 0-index 的。
- **在链表类中实现这些功能：**
  - get(index)：获取链表中第 index 个节点的值。如果索引无效，则返回-1。
  - addAtHead(val)：在链表的第一个元素之前添加一个值为 val 的节点。插入后，新节点将成为链表的第一个节点。
  - addAtTail(val)：将值为 val 的节点追加到链表的最后一个元素。
  - addAtIndex(index,val)：在链表中的第 index 个节点之前添加值为 val 的节点。如果 index 等于链表的长度，则该节点将附加到链表的末尾。如果 index 大于链表长度，则不会插入节点。如果index小于0，则在头部插入节点。
  - deleteAtIndex(index)：如果索引 index 有效，则删除链表中的第 index 个节点。

237. 删除链表中的节点 – 字节、阿里等公司面试题

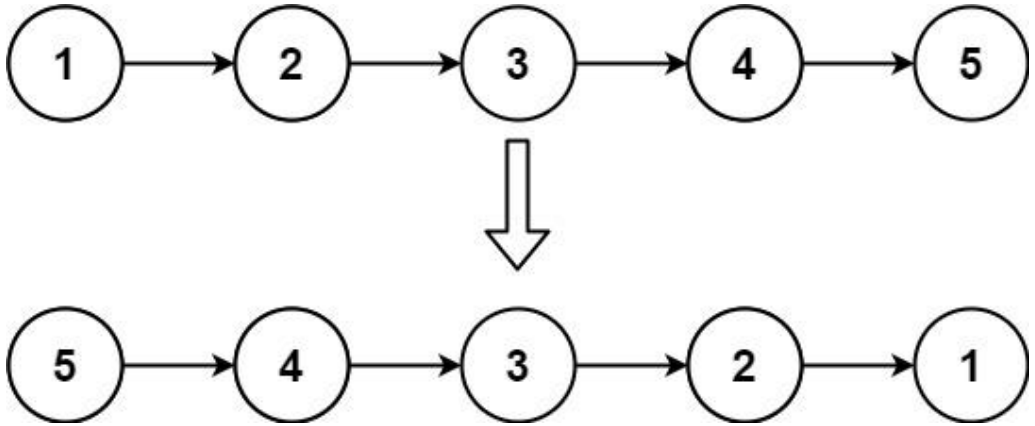
- **237. 删除链表中的节点**
  - https://leetcode.cn/problems/delete-node-in-a-linked-list/description/
- **有一个单链表的 head，我们想删除它其中的一个节点 node。**
  - 给你一个需要删除的节点 node 。
  - 你将 **无法访问** 第一个节点 head。
- **链表的所有值都是 唯一的，并且保证给定的节点 node 不是链表中的最后一个节点。**
- **删除给定的节点。注意，删除节点并不是指从内存中删除它。这里的意思是：**
  - 给定节点的值不应该存在于链表中。
  - 链表中的节点数应该减少 1。
  - node 前面的所有值顺序相同。
  - node 后面的所有值顺序相同。




```
function deleteNode(node: ListNode | null): void {
    node.val = node.next.val
    node.next = node.next.next
};
```

206. 反转链表 – 字节、谷歌等面试题

- 206. 反转链表
  - https://leetcode.cn/problems/reverse-linked-list/
- 给你单链表的头节点 head ，请你反转链表，并返回反转后的链表。



0 - 6 个月　6 个月 - 1 年　1 年 - 2 年

亚马逊 10　字节跳动 7　谷歌 Google 2　微软 Microsoft 2

相关企业

0 - 6 个月　6 个月 - 1 年　1 年 - 2 年

苹果 Apple 14　彭博 Bloomberg 6　Facebook 5　美团 3

甲骨文 Oracle 3　优步 Uber 2　英伟达 NVIDIA 2　三星 2

J.P Morgan 摩根大通 2

0 - 6 个月　6 个月 - 1 年　1 年 - 2 年

腾讯 16　百度 8　滴滴 8　VMware 6　思科 Cisco 5　eBay 5

IBM 4　高通 4　servicenow 4　快手 3

- 进阶：链表可以选用迭代或递归方式完成反转。你能否用两种方法解决这道题?

206. 反转链表（非递归）

```ts
function reverseList2(head: ListNode | null): ListNode | null {
  if (!head || !head.next) return head

  let newHead: ListNode | null = null
  while (head) {
    const current = head.next
    head.next = newHead
    newHead = head
    head = current
  }
  return newHead
};
```

206. 反转链表（递归）

```ts
function reverseList(head: ListNode | null): ListNode | null {
  // // 如果节点本身为null
  // if (!head) return null
  // // 如果只有一个节点
  // if (!head.next) return head
  // 可以转换成一行代码
  if (!head || !head.next) return head

  // 获取递归的结果
  const newHead = reverseList(head.next)

  // 让剩下的head节点的next节点的next指向head
  head.next.next = head
  // head本身的next指向null
  head.next = null

  return newHead
};
```

什么是算法复杂度（现实案例）？

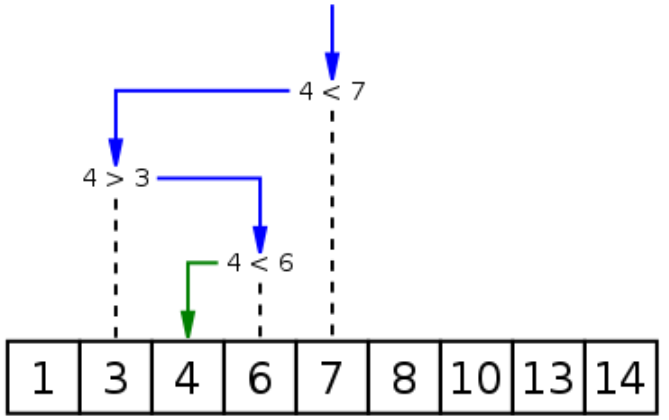
- **前面我们已经解释了什么是算法？其实就是解决问题的一系列步骤操作、逻辑。**
- **对于同一个问题，我们往往其实有多种解决它的思路和方法，也就是可以采用不同的算法。**
  - 但是不同的算法，其实效率是不一样的。
- **举个例子（现实的例子）：在一个庞大的图书馆中，我们需要找一本书。**
  - 在图书已经按照某种方式摆好的情况下（数据结构是固定的）
- **方式一：顺序查找**
  - 一本本找，直到找到想要的书；（累死）
- **方式二：先找分类，分类中找这本书**
  - 先找到分类，在分类中再顺序或者某种方式查找；
- **方式三：找到一台电脑，查找书的位置，直接找到；**
  - 图书馆通常有自己的图书管理系统；
  - 利用图书管理系统先找到书的位置，再直接过去找到；

什么是算法复杂度（程序案例）？

- 我们再具一个程序中的案例：让我们来比较两种不同算法在查找数组中（数组有序）给定元素的时间复杂度。
- 方式一: 顺序查找
  - 这种算法从头到尾遍历整个数组，依次比较每个元素和给定元素的值。
  - 如果找到相等的元素，则返回下标；如果遍历完整个数组都没找到，则返回-1。

```
function sequentialSearch(arr: number[], target: number) {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) {
      return i;
    }
  }
  return -1;
}
```

- 方式二：二分查找
  - 这种算法假设数组是有序的，每次选择数组中间的元素与给定元素进行比较。
  - 如果相等，则返回下标；如果给定元素比中间元素小，则在数组的左半部分继续查找；
  - 如果给定元素比中间元素大，则在数组的右半部分继续查找；
  - 这样每次查找都会将查找范围减半，直到找到相等的元素或者查找范围为空；




```
function binarySearch(arr: number[], target: number) {
  let left = 0;
  let right = arr.length - 1;
  while (left <= right) {
    let mid = Math.floor((left + right) / 2);
    if (arr[mid] === target) {
      return mid;
    } else if (arr[mid] < target) {
      left = mid + 1;
    } else {
      right = mid - 1;
    }
  }
  return -1;
}
```

顺序查找和二分查找的测试代码

- **顺序查找：**

```
// 顺序查找算法
let arr = new Array(10000000).fill(0).map((x,i)=>i);
let target = 5000000;
let start = performance.now();
let index = sequentialSearch(arr, target);
let end = performance.now();
console.log(`目标元素的索引是: ${index}`);
console.log(`顺序查找消耗的时间: ${end - start}ms`);
```

```
目标元素的索引是： 5000000
顺序查找消耗的时间： 6.526858091354 37ms
```

顺序查找算法的时间复杂度是O(n)

- **二分查找：**

```
// 二分查找算法
let arr = new Array(10000000).fill(0).map((x,i)=>i);
let target = 5000000;
let start = performance.now();
let index = binarySearch(arr, target);
let end = performance.now();
console.log(`目标元素的索引是: ${index}`);
console.log(`二分查找消耗的时间: ${end - start}ms`);
```

```
目标元素的索引是： 5000000
二分查找消耗的时间： 0.12210702896118164ms
```

二分查找算法的时间复杂度是O(log n)

大O表示法（Big O notation）

- **大O表示法（Big O notation）英文翻译为大O符号（维基百科翻译），中文通常翻译为大O表示法（标记法）。**
  - 这个记号则是在德国数论学家爱德蒙·兰道的著作中才推广的，因此它有时又称为兰道符号（Landau symbols）。
  - 代表“order of ...”（……阶）的大O，最初是一个大写希腊字母“O”（omicron），现今用的是大写拉丁字母“O”。
- **大O符号在分析算法效率的时候非常有用。**
  - 举个例子，解决一个规模为n的问题所花费的时间（或者所需步骤的数目）可以表示为：$T(n) = 4n^2 - 2n + 2$
    - ✓ 当n增大时，n^2项开始占据主导地位，其他各项可以被忽略；
  - 举例说明：当n=500
    - ✓ $4n^2$项是2n项的1000倍大，因此在大多数场合下，省略后者对表达式的值的影响将是可以忽略不计的。
    - ✓ 进一步看，如果我们与任一其他级的表达式比较，n^2的系数也是无关紧要的。

这样，针对第一个例子$T(n) = 4n^2 - 2n + 2$，大O符号就记下剩余的部分，写作：

$$T(n) \in \mathrm{O}(n^2)$$

或

$$T(n) = \mathrm{O}(n^2)$$

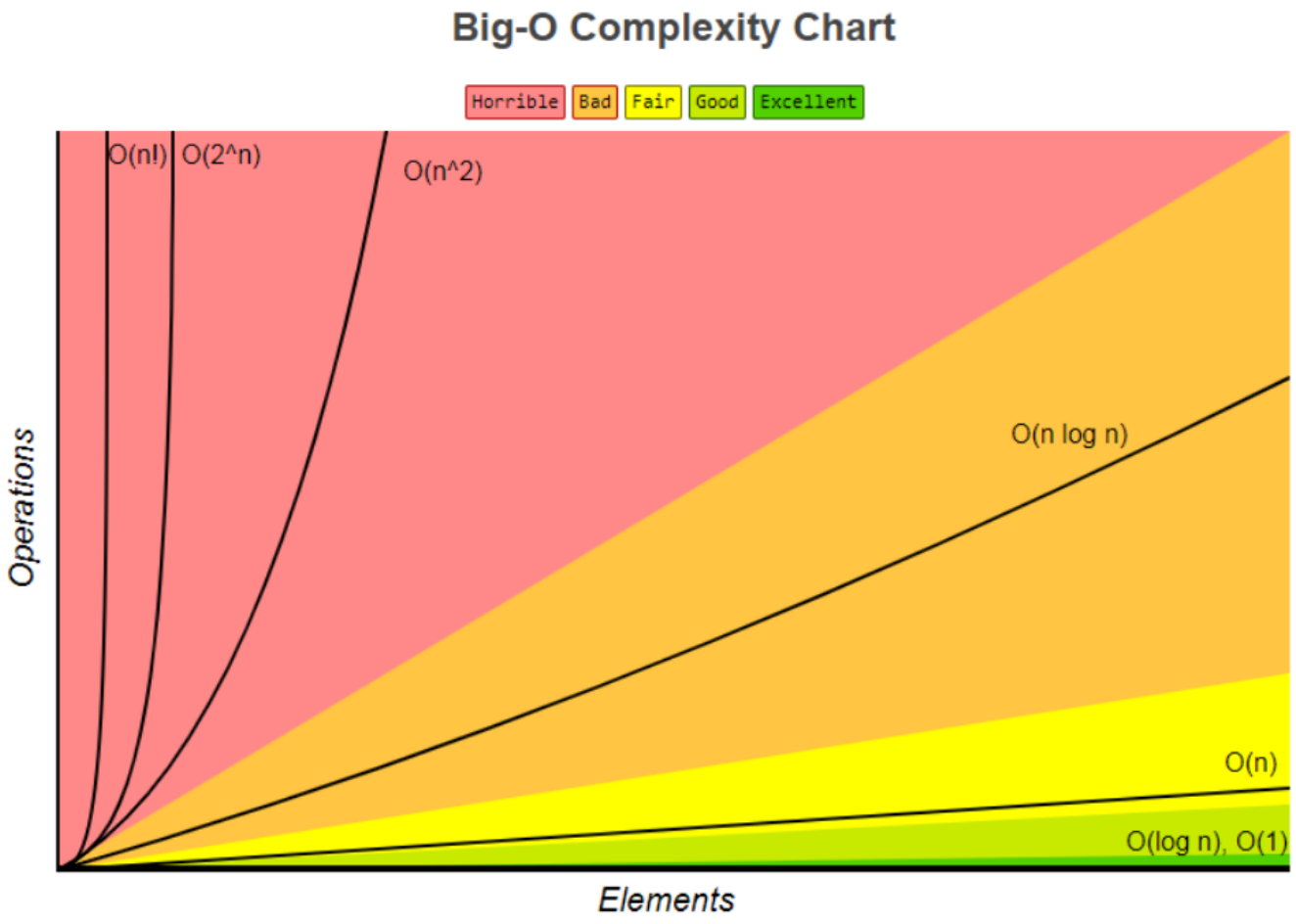
- **我们就说该算法具有n^2阶（平方阶）的时间复杂度，表示为$O(n^2)$。**

大O表示法 - 常见的对数结

■ 常用的函数阶

符号	名称
$O(1)$	常数（阶，下同）
$O(\log n)$	对数
$O(n)$	线性，次线性
$O(n \log n)$	线性对数，或对数线性、拟线性、超线性
$O(n^2)$	平方
$O(n^c), \text{Integer}(c > 1)$	多项式，有时叫作“代数”（阶）
$O(c^n)$	指数，有时叫作“几何”（阶）

案例	时间复杂度	术语
1234567890	O(1)	常数阶
2n+1	O(n)	线性阶
$6n^2+3n+2$	$O(n^2)$	平方阶
$6n^3+3n+2$	$O(n^3)$	立方阶
$3\text{Log}(2^n)+3n+2$	O(logn)	对数阶
$3n\text{Log}(2^n)+3n+2$	O(nlogn)	nlogn对数阶
2^n	$O(2^n)$	指数阶



空间复杂度

- **空间复杂度指的是程序运行过程中所需要的额外存储空间。**
  - 空间复杂度也可以用大O表示法来表示；
  - 空间复杂度的计算方法与时间复杂度类似，通常需要分析程序中需要额外分配的内存空间，如数组、变量、对象、递归调用等。
- **举个栗子：**
  - 对于一个简单的递归算法来说，每次调用都会在内存中分配新的栈帧，这些栈帧占用了额外的空间。
    - ✓ 因此，该算法的空间复杂度是O(n)，其中n是递归深度。
  - 而对于迭代算法来说，在每次迭代中不需要分配额外的空间，因此其空间复杂度为O(1)。
- **当空间复杂度很大时，可能会导致内存不足，程序崩溃。**

- **在平时进行算法优化时，我们通常会进行如下的考虑：**
  - 使用尽量少的空间（优化空间复杂度）；
  - 使用尽量少的时间（优化时间复杂度）；
  - 特定情况下：使用空间换时间或使用时间换空间；

数组和链表的复杂度对比

- **接下来，我们使用大O表示法来对比一下数组和链表的时间复杂度：**

Data Structure	Access	Search	Insertion	Deletion
Array	O(1)	O(N)	O(N)	O(N)
Linked list	O(N)	O(N)	O(1)	O(1)

- **数组是一种连续的存储结构，通过下标可以直接访问数组中的任意元素。**
  - 时间复杂度：对于数组，随机访问时间复杂度为O(1)，插入和删除操作时间复杂度为O(n)。
  - 空间复杂度：数组需要连续的存储空间，空间复杂度为O(n)。
- **链表是一种链式存储结构，通过指针链接起来的节点组成，访问链表中元素需要从头结点开始遍历。**
  - 时间复杂度：对于链表，随机访问时间复杂度为O(n)，插入和删除操作时间复杂度为O(1)。
  - 空间复杂度：链表需要为每个节点分配存储空间，空间复杂度为O(n)。

- **在实际开发中，选择使用数组还是链表需要根据具体应用场景来决定。**
  - 如果数据量不大，且需要频繁随机访问元素，使用数组可能会更好。
  - 如果数据量大，或者需要频繁插入和删除元素，使用链表可能会更好。