数据结构实验（2）

链表、队列

目录

- 线性表的几种物理实现（2）
  - C/C++ 数组
  - C++ std::vector
  - C++ std::list
  - C/C++ 自定义线性表
  - C/C++ 自定义链表
  - …

回顾上节课，我们了解了：

1. **不同的内存空间分配区域（栈空间&堆空间）**
2. **不同的内存空间管理方式（预分配&动态分配）**
3. **不同的时间复杂度**

- # 线性表的几种物理实现（2）

  - 现象：基本操作的时间复杂度不同

	插入	删除	随机访问	查找	扩容
std::vector	O(N)	O(N)	O(1)	O(N) / O(logN)	O(N)
std::list	O(1)	O(1)	O(N)	O(N) / O(logN)	O(1)

  - 本质：内存中的存储方式不同

    - std::vector

      - 在内存中连续存储，且物理地址与线性表中元素的逻辑序列有序对应

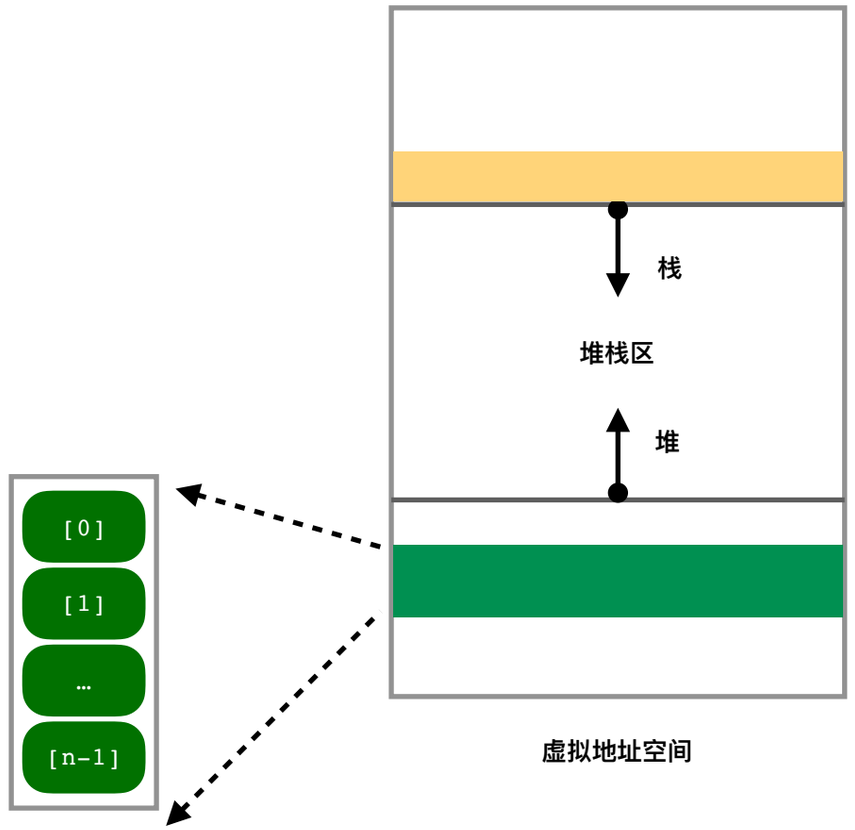
    - std::list

      - 每个元素分别记录线性表中逻辑序列中的前一个、后一个元素的物理地址

- 线性表的几种物理实现（2）
  - 现象：基本操作的时间复杂度不同

	插入	删除	随机访问	查找	扩容
std::vector	O(N)	O(N)	O(1)	O(N) / O(logN)	O(N)
std::list	O(1)	O(1)	O(N)	O(N) / O(logN)	O(1)

  - 本质：内存中的存储方式不同
    - std::vector
      - 在内存中连续存储，且物理地址与线性表中元素的逻辑序列有序对应
    - std::list
      - 每个元素分别记录线性表中逻辑序列中的前一个、后一个元素的物理地址



- # 线性表的几种物理实现（2）

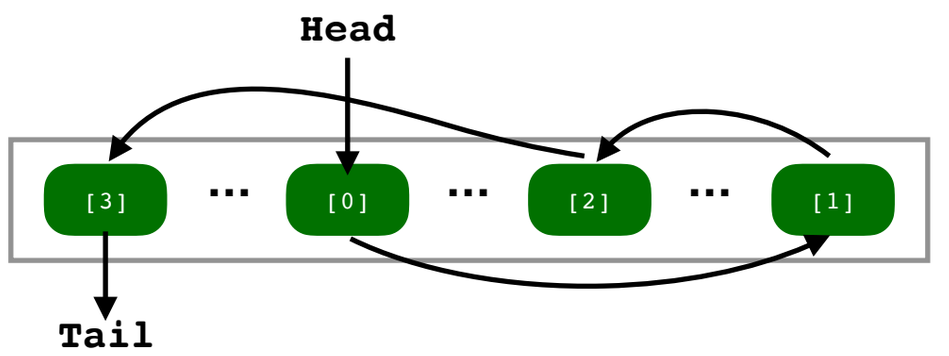
  - 现象：基本操作的时间复杂度不同

	插入	删除	随机访问	查找	扩容
std::vector	O(N)	O(N)	O(1)	O(N) / O(logN)	O(N)
std::list	O(1)	O(1)	O(N)	O(N) / O(logN)	O(1)

  - 本质：内存中的存储方式不同

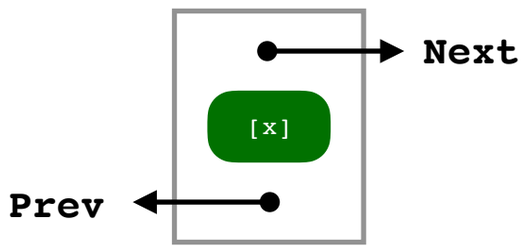
    - std::vector

      - 在内存中连续存储，且物理地址与线性表中元素的逻辑序列有序对应



    - std::list

      - 每个元素分别记录线性表中逻辑序列中的前一个、后一个元素的物理地址



1. 我们能在 std::list 中，通过 [x] 方式高效访问第 x 个元素么?
2. 我们能在 std::vector 中，高效实现插入 N 个或删除 N 个元素么?

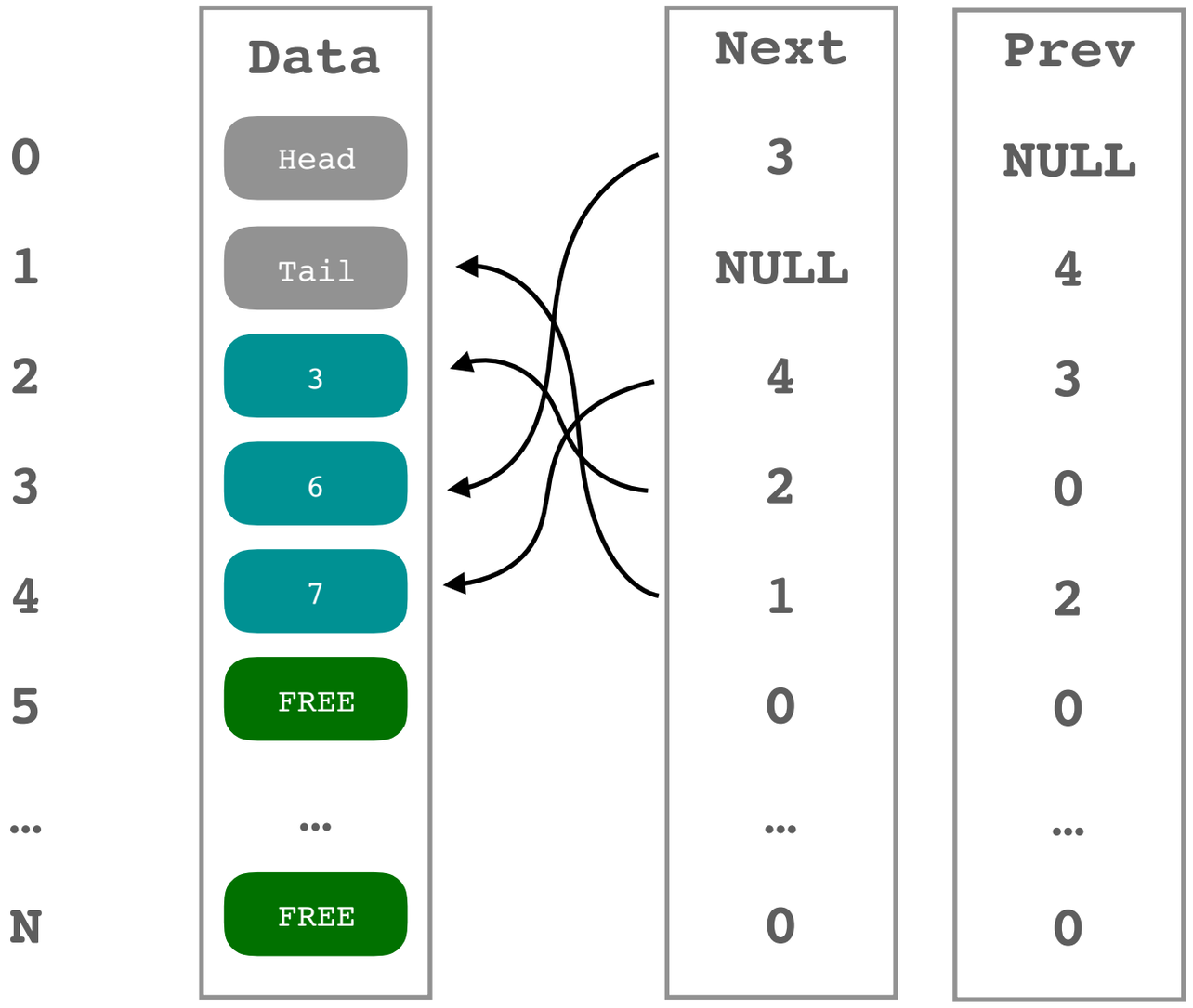
- 线性表的几种物理实现（2）
  - 总结

	内存分配位置	内存分配方式*	物理存储方式
数组	栈空间	预分配（一次性分配）	连续存储、有序记录
std::vector	堆空间	预分配（一次性分配）	连续存储、有序记录
std::list	堆空间	动态分配	离散存储、记录前序后序

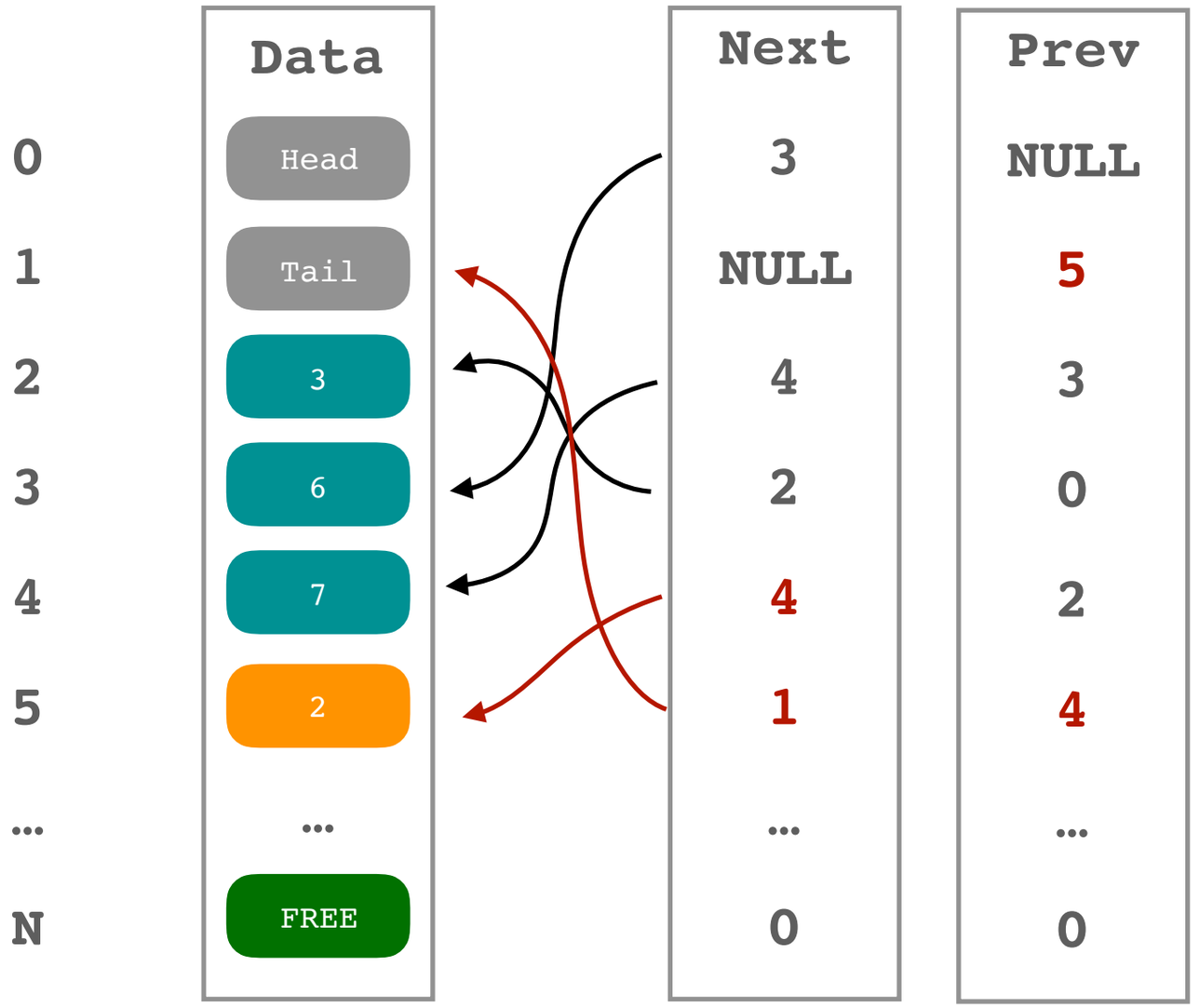
目录

- 理解链表基本操作
  - List 与 Vector 的本质区别：在空间中离散存储，通过“地址”获取实际值。
    - 用数组实现链表，通过数组记录某节点的前序、后序节点
    - 用 C/C++ 的指针实现链表，通过内存地址记录某节点的前序、后序节点

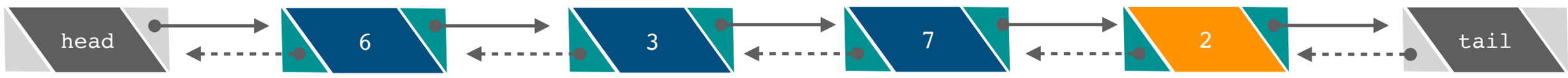
- 理解链表基本操作（1）
  - 链表的数组实现



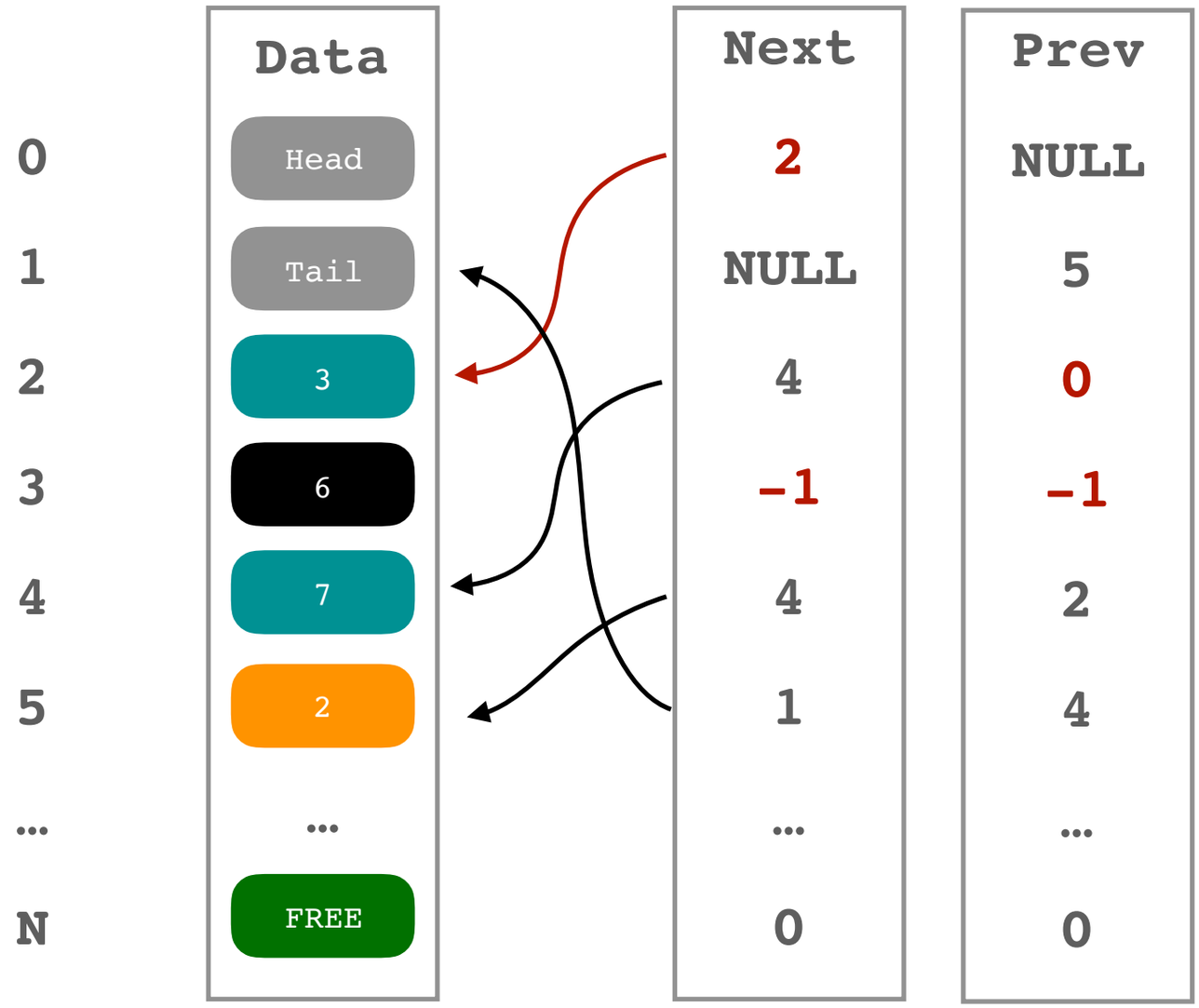
- 理解链表基本操作（1）
  - 链表的数组实现——插入操作



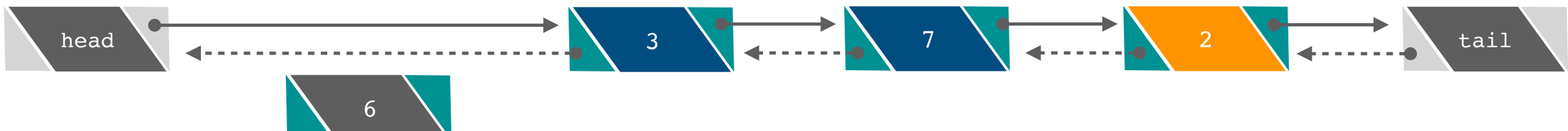

```
def insertBefore(K):
    index = Data.find(K)

    Data.append(2)

    Next[Data.size] = Next[index]
    Next[index] = Data.size

    Prev[Data.size] = Prev[index]
    Prev[Next[Data.size]] = index
```



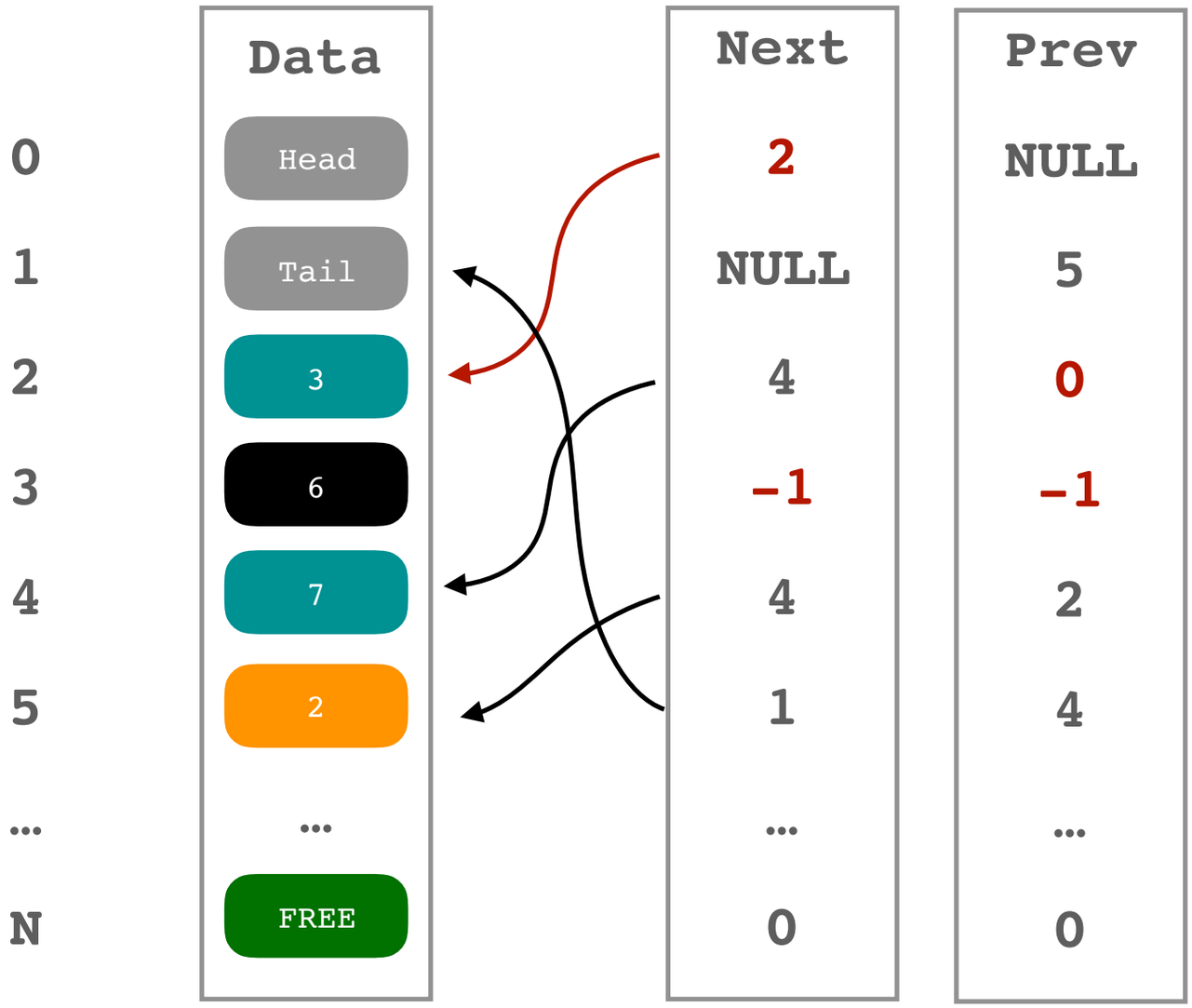
- 理解链表基本操作（1）
  - 链表的数组实现——删除操作



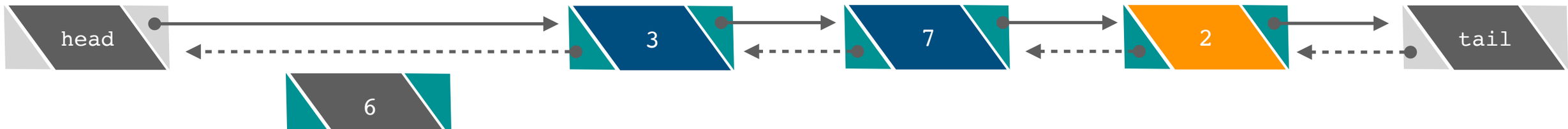

```
def insertBefore(K):
    index = Data.find(K)

    Prev[Next[index]] = Prev[index]
    Next[Prev[index]] = Next[index]

    Prev[index] = -1
    Next[index] = -1
```



- 理解链表基本操作（1）
  - 链表的数组实现



1.我们可以用数组实现链表的逻辑操作
2.如何处理删除元素后留下的空洞?



- 穿插话题1：理解 C/C++ 的指针，及指针与数组的区别

```cpp
#include <iostream>

int main(int, char**) {

    int a[10] = {1, 2, 3, 4, 5, 5, 4, 3, 2, 1};
    int *b = a;

    for (int i = 0; i < 10; ++i) {
        std::cout << a[i] << " ";
    }  // output 1 2 3 4 5 5 4 3 2 1
    std::cout << std::endl;

    for (int i = 0; i < 10; ++i) {
        std::cout << b[i] << " ";
    }  // output 1 2 3 4 5 5 4 3 2 1
    std::cout << std::endl;

    return 0;
}
```

输出 a 和 b 的结果是一样的吗?
int a[] 与 int *b 是一样的吗?

- 理解链表基本操作（2）

  - 数组的指针实现——初始化和析构

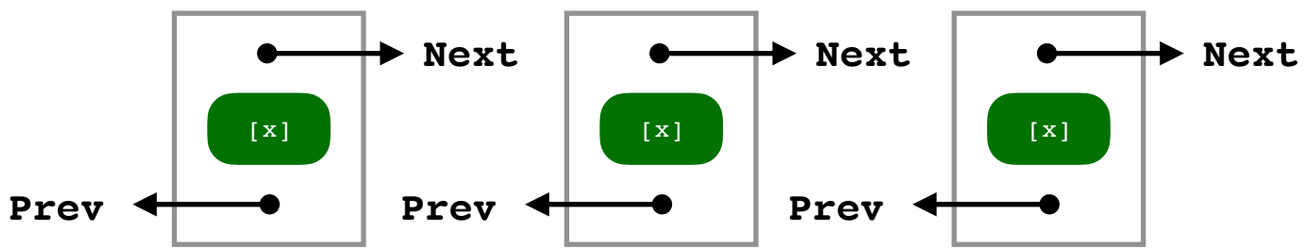
```
template<typename T>
class Node {
public:
    Node() : next(nullptr), prev(NULL) {
    }
    Node(const T& data) : data(data), next(nullptr), prev(NULL) {
    }
    ~Node() {
    }

    std::unique_ptr<Node<T> > next;  // unique_ptr ?
    Node<T> *prev;

    T data;
};

template <typename T>
class List {
public:
    List() {
        head = std::unique_ptr<Node<T> >(new Node<T>());
        head->next = std::unique_ptr<Node<T> >(new Node<T>());
        tail = head->next.get();
        tail->prev = head.get();
    }

    ~List() {
        while (head) {
            head = std::move(head->next);  // std::move ?
        }
    }

    std::unique_ptr<Node<T> > head;
    Node<T> *tail;
};
```

- # 穿插话题2：RAII 与智能指针

  - 回顾 RAII：
    - 目的：在栈空间定义变量，自动管理堆空间的内存
    - 方法：
      - 定义一个类来封装资源（内存、连接、文件操作符等）的分配与释放；
      - 类的构造函数中完成资源的分配及初始化；
      - 类的析构函数中完成资源的清理，可以保证资源的正确初始化和释放；
    - 问题：
      - **当对象需要被拷贝时，只能复制资源（深拷贝），而无法直接引用（浅拷贝）；**

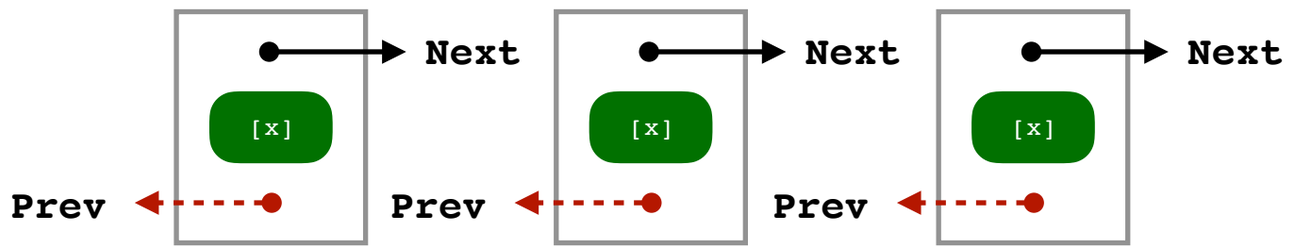
```cpp
template<typename T>
struct Array {

private:
    T* elem;

public:

    Array(int n) {
        this->elem = new T[n];
    }
    ~Array() {
        delete[] this->elem;
    }

    T* get() {
        return this->elem;
    }

};

int main(int, char**) {
    // 通过 Array 对象，在堆空间创建线性表
    Array<int> arr(10);

    // 如果需要 Array<int> arr_another = arr;
    // Do something with arr_another …

    return 0;
}   // arr 会在离开作用域后自动调用析构函数，释放资源
```

- 穿插话题2：RAII 与智能指针
  - 智能指针：
    - `shared_ptr`：通过引用计数管理资源
      - 问题：



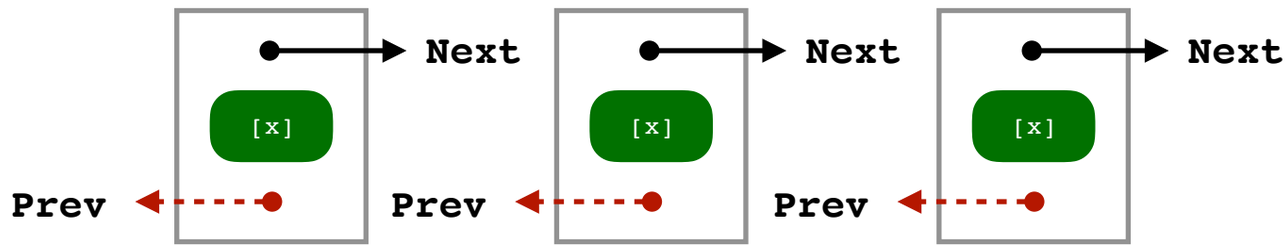
        - **双向链表中存在 `x`，`x.next`，`x.next.prev` 的循环引用，导致引用计数失效。**

    - `unique_ptr`：明确资源唯一属主，不提供 copy 操作防止多个unique_ptr指向同一对象。
      - 实现链表：

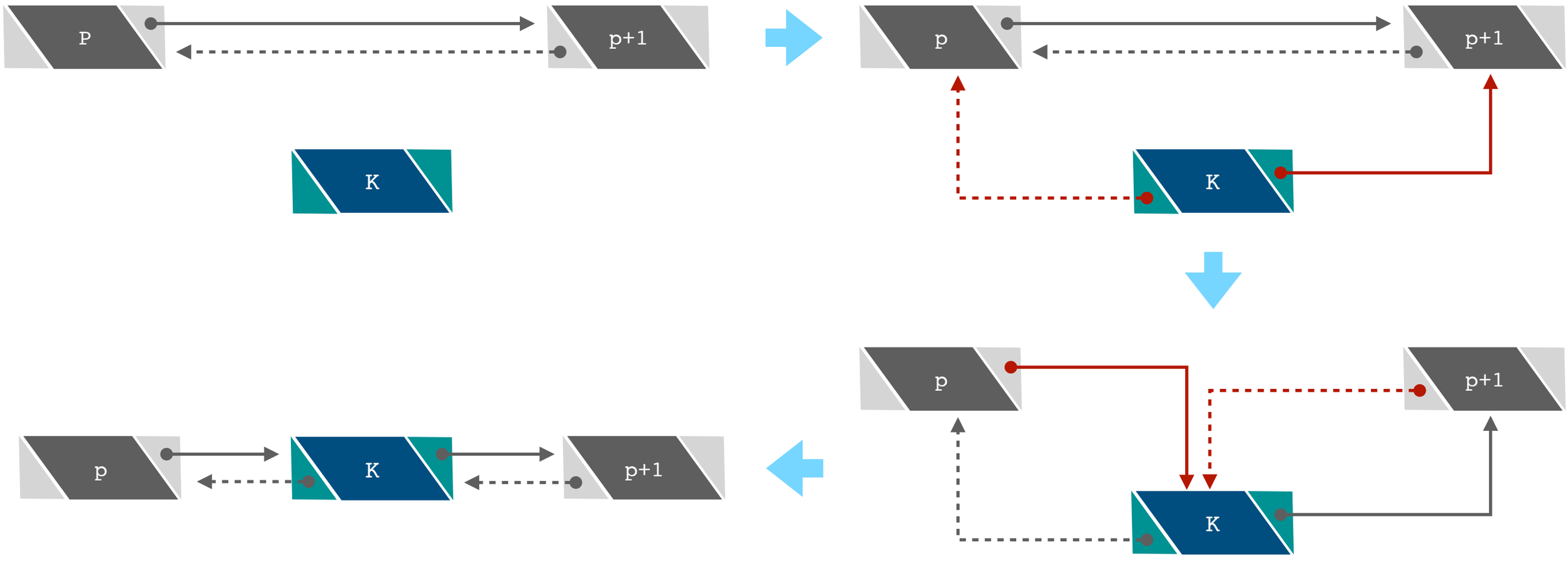


        - 每个链表节点，Next 对象负责管理资源，Prev 只作为弱引用；
        - `unique_ptr` 提供了 `move` 操作，因此我们可以用 `std::move()` 来显式转移 `unique_ptr` 的资源所有权。

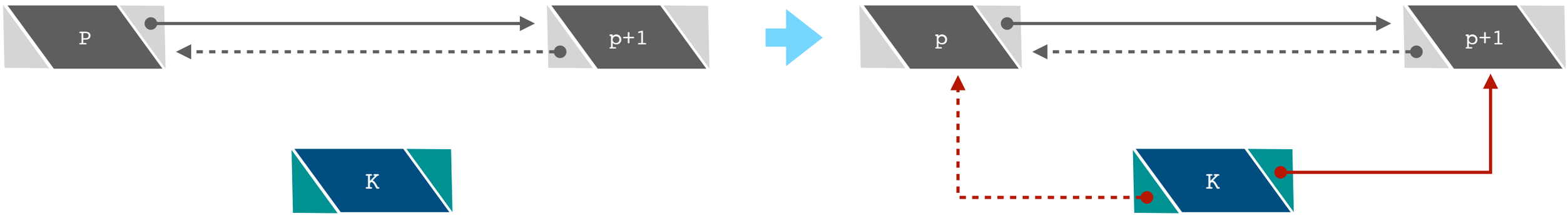
- 穿插话题2：RAII 与智能指针
  - 智能指针：
    -



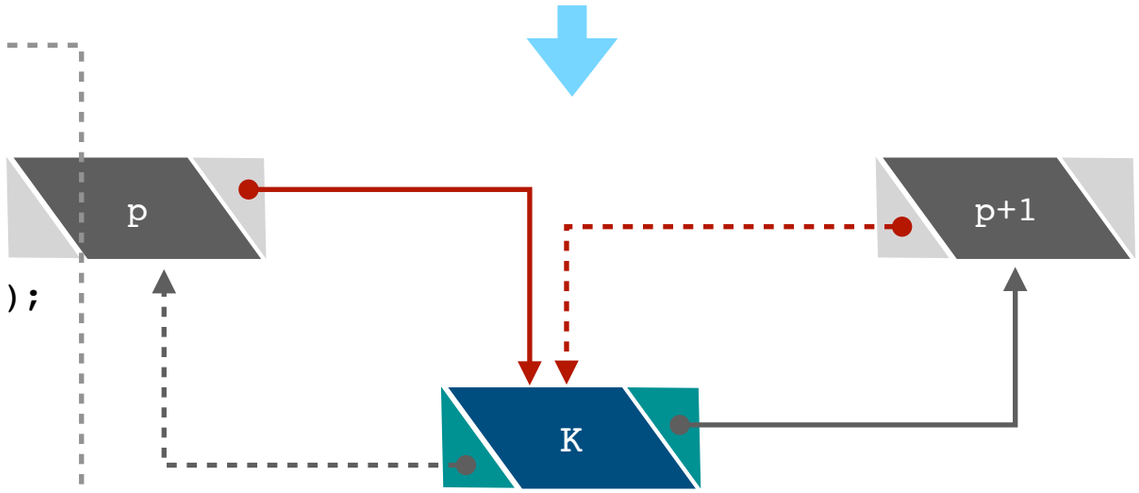
- 理解链表基本操作（2）
  - 插入 list<T>::insertBefore(p, K) & list<T>::insertAfter(p, K)



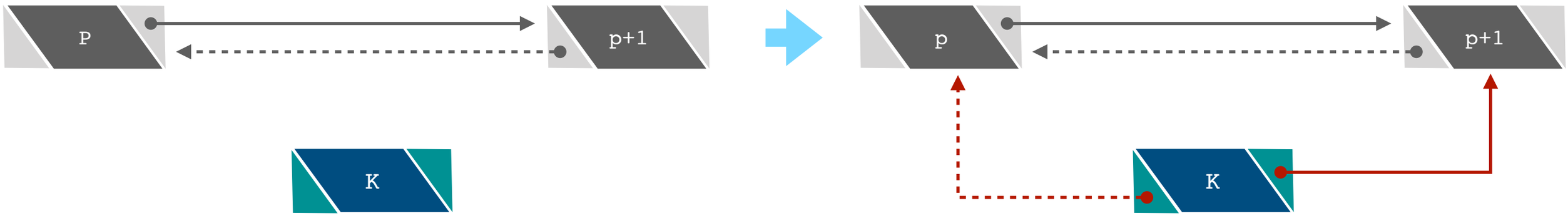
- 理解链表基本操作（2）
  - 插入 list<T>::insertBefore(p, K) & list<T>::insertAfter(p, K)



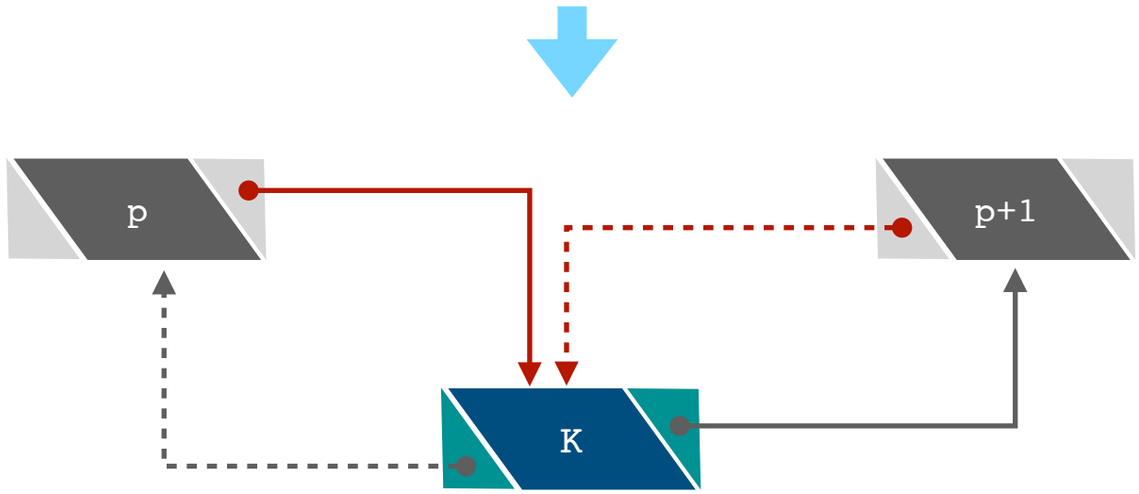

```
template <typename T>
class List {
    // ...

    Node<T> * insertAfter(Node<T> *p, T data) {
    (1) auto node = std::unique_ptr<Node<T> >(new Node<T>(data));

    (2) node->prev = p;
        node->next = std::move(p->next);

    (3) node->next->prev = node.get();
        node->prev->next = std::move(node);

        return node.get();
    }

    // ...
};
```



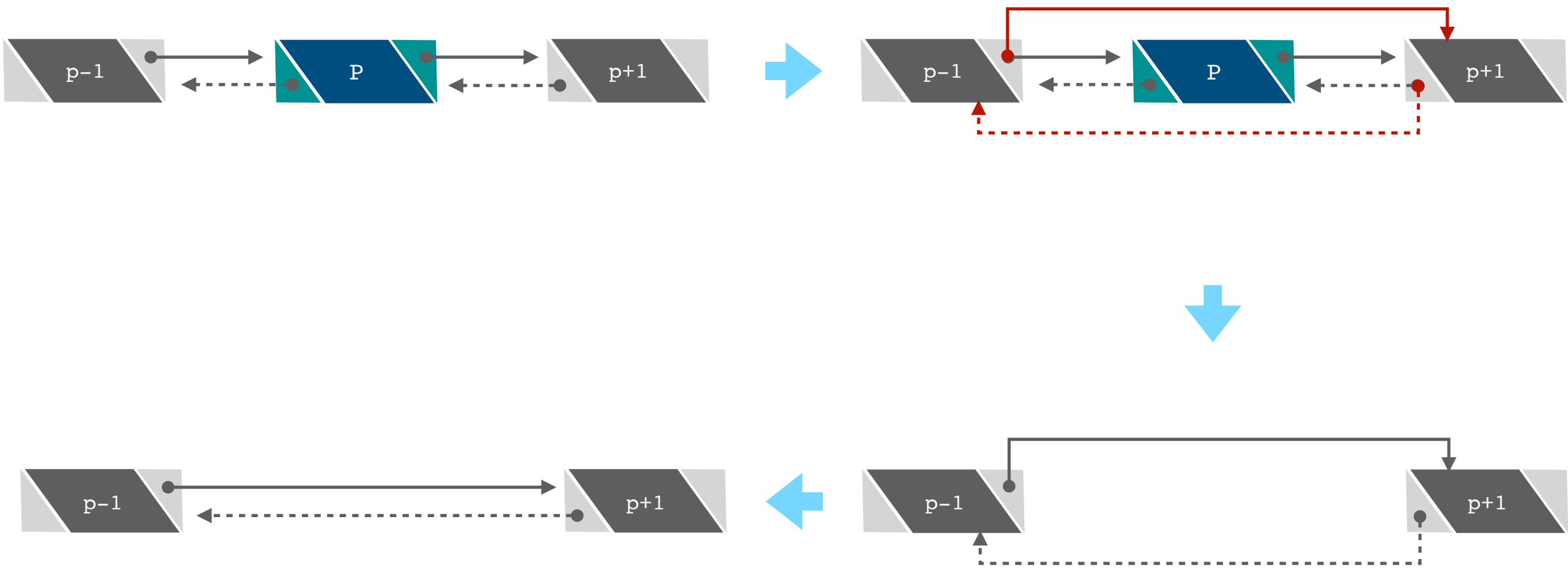
- 理解链表基本操作（2）
  - 插入 list<T>::insertBefore(p, K) & list<T>::insertAfter(p, K)



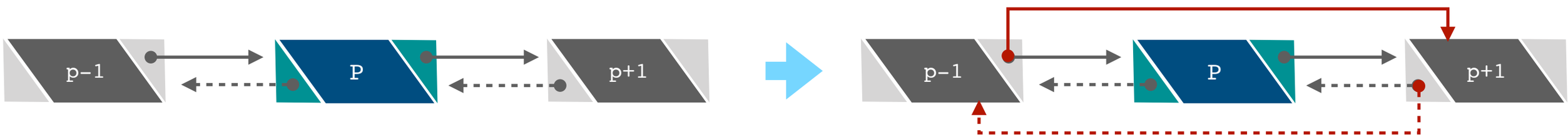

```
template <typename T>
class List {
    // ...

    Node<T> * insertBefore(Node<T> *p, T data) {
        return insertAfter(p->prev, data);
    }

    // ...
};
```



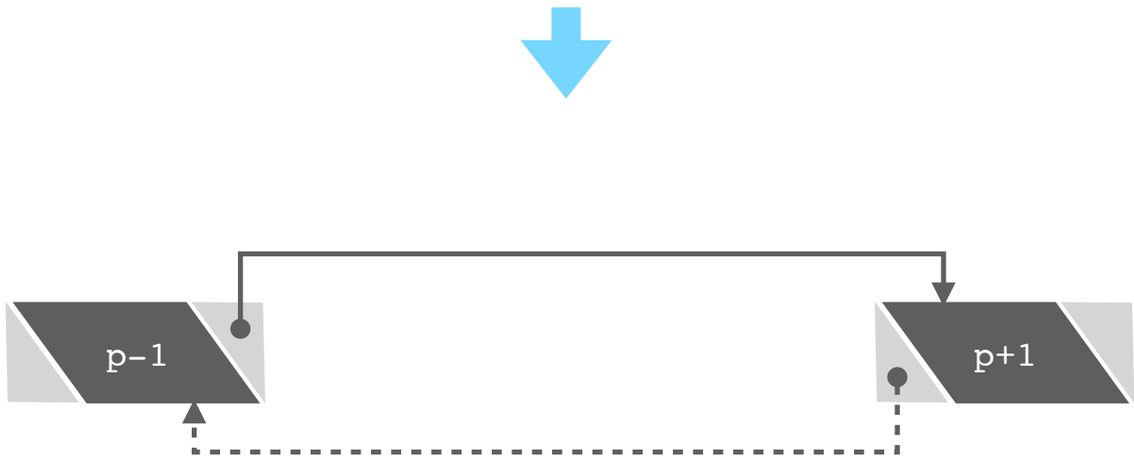
- 理解链表基本操作（2）
  - 删除 list<T>::remove(p)



- 理解链表基本操作（2）
  - 删除 list<T>::remove(p)




```
template <typename T>
class List {
    // ...

    void remove(Node<T> *p) {
        p->next->prev = p->prev;
        p->prev->next = std::move(p->next);
    }

    // ...

};
```

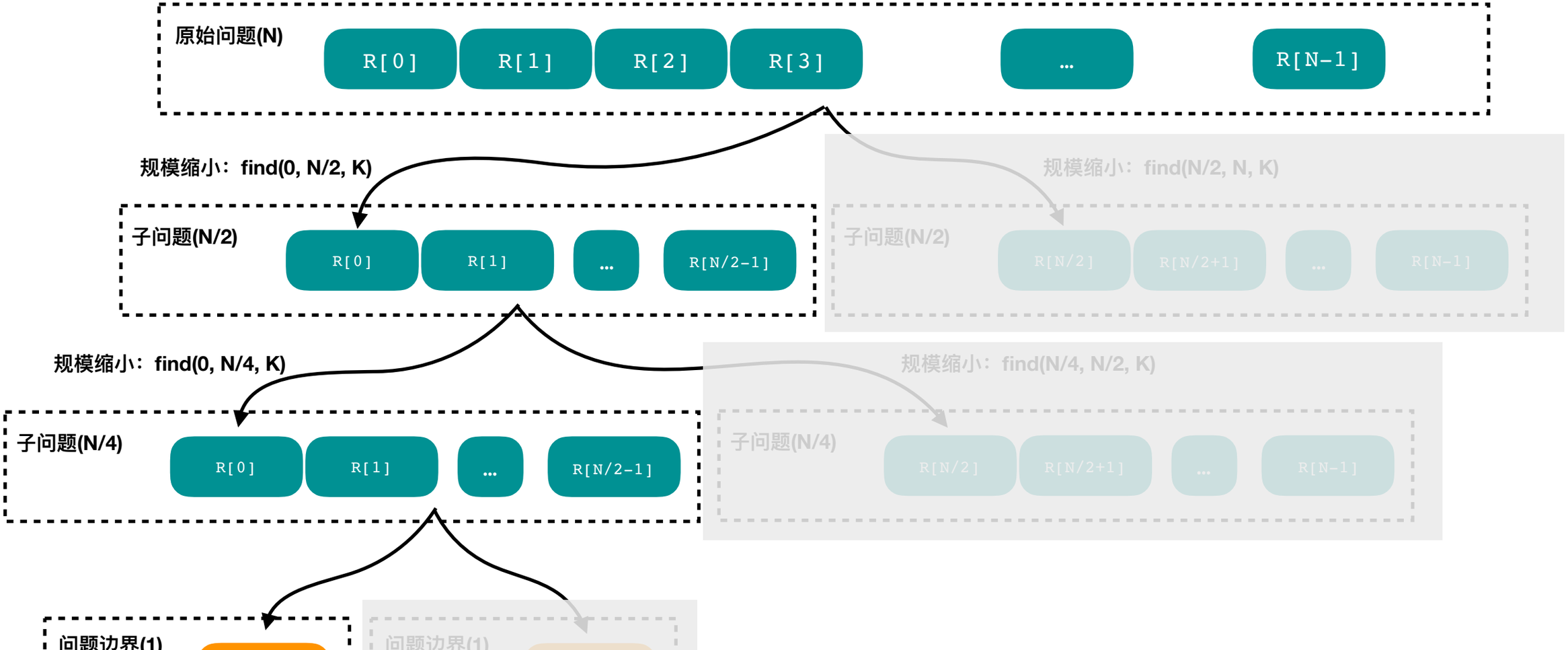


- 理解链表基本操作（2）
  - 查找 list<T>::find(K)


```
template <typename T>
class List {
    // ...

    Node<T> * find(T data) {
        auto p = head->next.get();
        while (p->next.get() != tail) {
            if (p->data == data) {
                return p;
            }
            p = p->next.get();
        }
        return p;
    }

    Node<T> * get(int index) {
        auto p = head->next.get();
        while (index-- > 0) {
            p = p->next.get();
        }
        return p;
    }

    // ...
};
```

- 理解链表基本操作（2）
  - 有序链表二分查找 list<T>::binary_find(K)

有序 std::list 能否像有序 std::vector 那样在 O(logN) 时间复杂度实现二分查找?

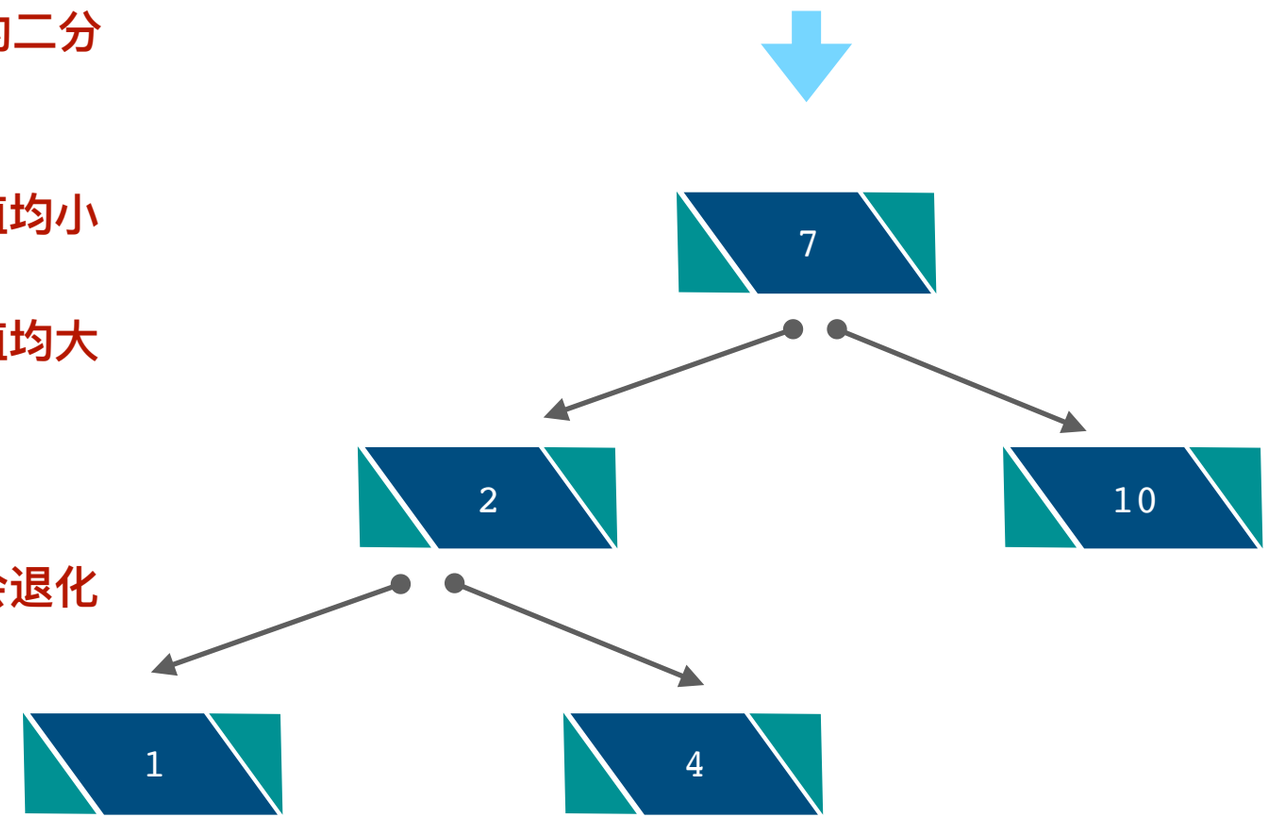


- 穿插话题3：链表与二叉排序树
  - 存在有序链表：

将链表转为二叉排序树，实现 O(logN) 的二分查找：

- 若左子树不空，则左子树上所有结点的值均小于或等于它的根结点的值；
- 若右子树不空，则右子树上所有结点的值均大于或等于它的根结点的值；
- 左、右子树也分别为二叉排序树；

注意！构造方法有技巧，否则二叉排序树会退化为“链表”



LeetCode 109. <Convert Sorted List to Binary Search Tree>
https://leetcode.com/problems/convert-sorted-list-to-binary-search-tree/

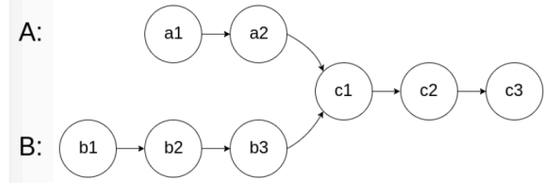
目录

- 链表相关经典（面试）问题
  - 实现链表反转操作

    LeetCode 206. <Reverse Linked List>
    https://leetcode.com/problems/reverse-linked-list/

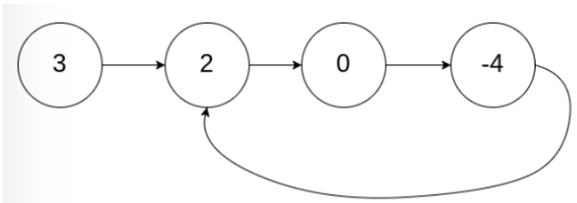
  - 判断两个单链表是否相交，并返回相交节点

    LeetCode 160. <Intersection of Two Linked Lists>
    https://leetcode.com/problems/intersection-of-two-linked-lists/



  - 判断单向链表是否存在“环”

    LeetCode 141. <Linked List Cycle>
    https://leetcode.com/problems/linked-list-cycle/



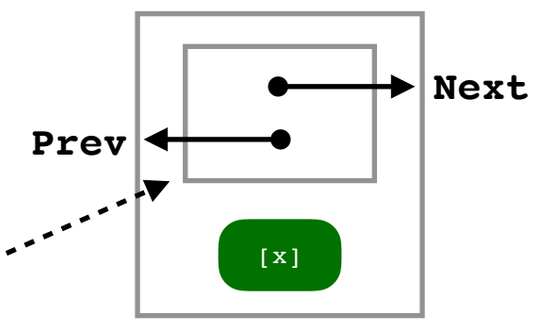
  - 判断链表是否为回文链表

    LeetCode 234. <Palindrome Linked List>
    https://leetcode.com/problems/palindrome-linked-list/

- 穿插话题4：Linux 内核源代码中提供了一种 C 语言的通用链表
  - 《深入分析 Linux 内核链表》 by IBM Developer

```c
struct NodeOld {
    struct NodeOld *next, *prev;
    int x;
    int y;
};
```

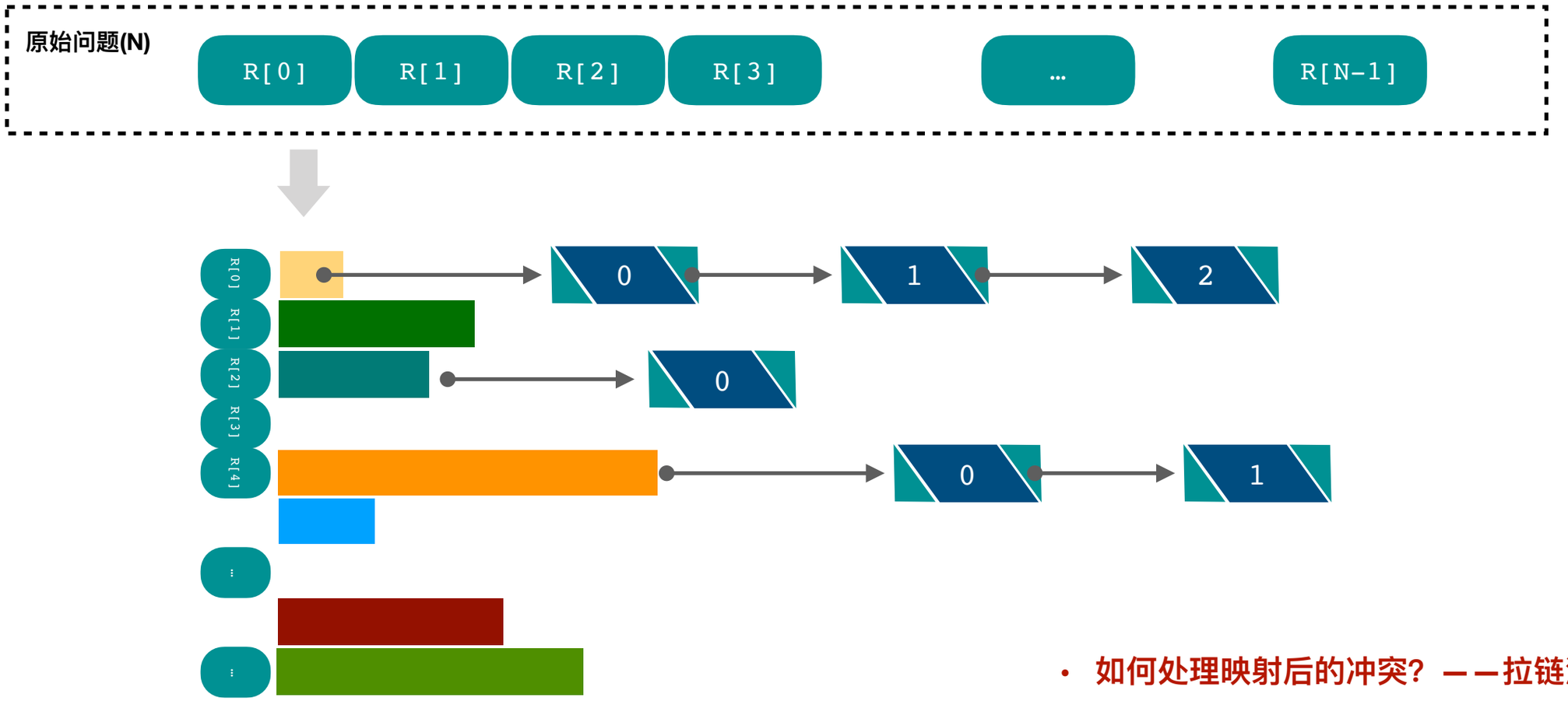
```c
struct ListHead {
    struct ListHead *next, *prev;
};

struct MyNode {
    struct ListHead list_head;
    int x;
    int y;
};
```



sizeof(ListHead) 是固定的
通过 *next 指向的地址 + sizeof(ListHead) 得到真实链表节点的地址

目录

- 线性表的扩展：Bitmap/Hashmap 冲突处理
  - 老问题——去重 vector<T>::deduplicate (0, N)
  - 新场景：
    - 如果 N 很大，线性表中元素取值范围 [0, m)，且 m 也不小
    - 如果 N 很大，且线性表中元素是英文单词



- 如何处理映射后的冲突？——拉链法

扩展练习

LeetCode 206. <Reverse Linked List>
https://leetcode.com/problems/reverse-linked-list/

LeetCode 160. <Intersection of Two Linked Lists>
https://leetcode.com/problems/intersection-of-two-linked-lists/

LeetCode 141. <Linked List Cycle>
https://leetcode.com/problems/linked-list-cycle/

LeetCode 234. <Palindrome Linked List>
https://leetcode.com/problems/palindrome-linked-list/

POJ 2503. <Babelfish >
http://poj.org/problem?id=2503