《初级算法与数据结构》

链表

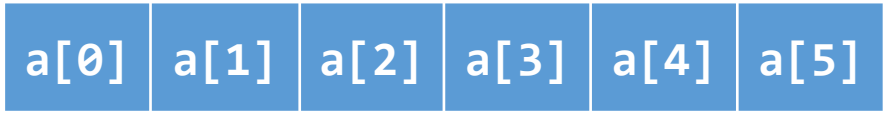


链表

一排人手牵手站在一起，其中有一个人离队，剩下人保持原来顺序牵着手使队伍相连。那么每个人所牵着手的人有没有变化呢?

链表

数组示例图：



今天来学习一个非常重要，也非常有意思的数据结构—**链表**

在学习链表之前，先回忆一下数组。如果老师让大家按身高排成一队，然后记录下每个人的名字，用数组来存是自然是最方便的，比如 `string names[50];`

用数组来记录队伍有什么特点?

每个元素它们都是**肩并着肩，位置是连续的、固定的**。这就对应着数据在计算机内存里一样，数组里的元素一个挨着一个，地址是连续的。

对于数组而言，如果想找某个元素，那么可以直接找到对应元素地址，然后进行访问，一步到位。这也是数组的优点：**随机访问速度快**。

但是对于数组而言，如果需要删除或者添加一个元素，那么添加一个元素后，它后面所有的元素都需要往后挪一位，而删除也是同样道理。这就是数组的缺点：**删除、插入速度慢**。

但是如果刚才那队同学解散了，他们没有按顺序站在一起了，数据散落在内存中各个位置，那么我们还有可能可以还原出刚才排队的顺序吗？

不可能还原出来，因为已经**丢失了相对位置信息**了，那么能通过**添加什么信息**使得我们可以还原出刚刚的排队顺序吗？可以想一下体育课的时候你们是如何找到方阵中自己的位置的。

1. 在体育课你们肯定不太记得自己是第几排第几个，但肯定都记得一件事：**自己的后面/前面的那个人是谁。**（对于一维数组只需要考虑前后关系）

2. 并且还需要知道**排在最前面的**同学是谁。 比如，知道队伍的第一个人是小止，他是这个队伍起点，通过这个起点就可以 **1号推2号，2号推3号...直到把整个队伍给推演出来。**

链表

注意到了吗，虽然这些同学在大厅里站的位置是乱七八糟、不连续的，但通过一个信息（你后面是谁）找到了下一个，再通过下一个的信息找到了下下个……最终把整个队伍给串起来了。

这种**数据在内存中可以不连续存放**，元素之间依靠一个信息来维持先后关系的数据结构，我们就把它叫做—**链表(List)**。

而那个起点（1号同学），在链表里我们叫它**头节点（Head）**。每个同学（节点）都包含两部分信息：他自己的**数据（Data）**（比如他的编号）和指向下一个同学的**信息（Next）**。

例15-10 排队记录(B0222)

n 名同学在排队进入大厦后解散自由活动。现在希望知道当初排队的顺序,然而并没有记录队伍是怎么排的。但好消息是,已经知道止止(编号是1)排在最前面；除此之外,每人都记得自己后面是谁。如何利用这些信息，还原出那天晚上的排队顺序呢?

链表

为了在程序里表示 **某人后面是谁** 这个信息，可以用一个数组，就叫它 `Next` 数组。

而`Next` 数组的规则很简单：**`Next[i] = j` 的意思就是：编号为 `i` 的同学，他后面站的是编号为 `j` 的同学(桶数组思想)。**

举一个具体的例子：有1到4号同学，原排列顺序为1,4,3,2，已经知道的信息是：

- 1 号后面是 4 号并且是 1 号是起点。
- 4 号后面是 3 号。
- 3 号后面是 2 号。
- 2 号后面没人了。

那么，根据刚才的信息，我们的 `Next` 数组就是：

- `Next[1] = 4;`
- `Next[4] = 3;`
- `Next[3] = 2;`

3 号后面没人了怎么办？可以用一个特殊值来表示结束。

- `Next[2] = -1;`

Next 数组

下标(编号):	0	1	2	3	4
数组值(后继):	-1	4	-1	2	3

Next[i] = j 的意义为编号为 i 的同学，后面站的是编号为 j 的同学。

现在 Next 数组和 起点编号(1号)都有了。就可以写代码把**队伍的原本顺序还原出来了**。

1. 设置一个当前位置的变量，让它从起点出发。 `int i = 1;`
2. 开始循环。 只要我们没走到终点，就要一直走下去。而终点的标记是是 -1。所以循环的条件是 当前位置不等于0。 `while (i != 0)`
3. 在循环里，每到一个位置做两件事：
   - 把当前位置的人的编号打印出来。`cout << i;`
   - 移动到下一个位置。怎么移动？问当前这个人，你后面是谁？也就是 `Next[i]`。所以我们让 `i` 更新为它的下一个位置。`i = Next[i];`

链表

Next 数组

下标(编号):	0	1	2	3	4
数组值(后继):	-1	4	-1	2	3

Next[i] = j 的意义为编号为 i 的同学，后面站的是编号为 j 的同学。

把这几步合起来，就是完整的遍历代码：

```
// 假设Next数组已经像上面那样填好了
int i = 1; // 从头节点出发
while (i != 0) { // 只要还没到终点
    cout << i << " "; // 打印当前位置
    i = Next[i];      // 查询后面一个人的位置
}
```

遍历顺序为：

1 -> 4 -> 3 -> 2

而这个循环对应了三个步骤：

`int i = 1` (从哪开始)

`i != 0` (何时停止)

`i = Next[i]` (如何去下一站)。

这种从头走到尾，访问链表中每一个元素的操作，就叫做**链表的遍历**。

链表

除了通过 while 循环遍历链表，for 循环也能实现相同的效果

```
//  (出发点，初始化为起点)  (终点判断)          (前进)
for (int current = head; current != 0; current = Next[current]) {
    // ... 在这里处理 current 节点 ...
}
```

通过 Next 这个关系网，就可以建立起了 i 指向 j 的逻辑顺序。

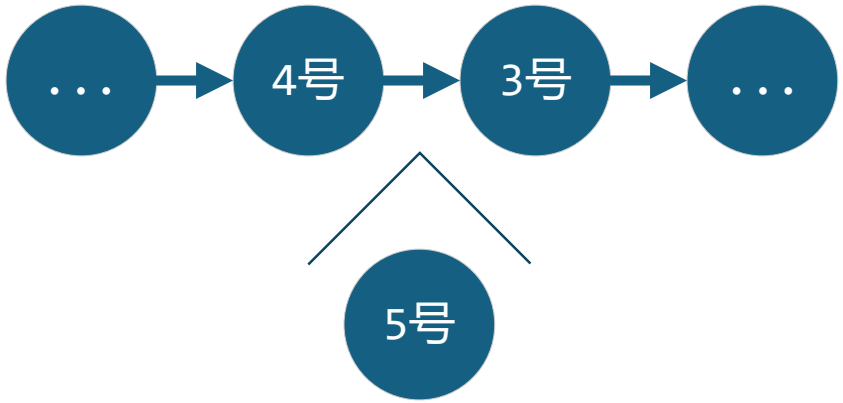
同时我们也会发现链表的一个缺点，**如果要查询某一个位置的元素，就必须要遍历整个链表**，也就是时间复杂度为 O(n)，远高于数组的O(1)，所以链表要尽量避免随机查询元素。

但是前面说过对于数组而言，如果需要删除或者添加一个元素的速度是比较慢的，**那么对于链表是否也会有这个缺点呢?**

链表

在链表中**插入新元素**，还需要像数组那样大面积移动数据吗？

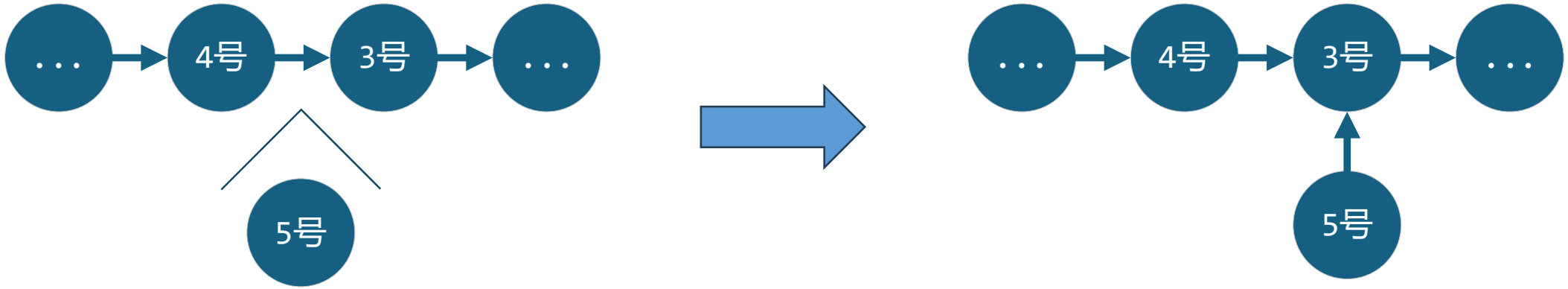
假设当前的链表顺序为 `...` -> 4号 -> 3号 -> `...`，现在有一个新的节点 **5号（y）**，需要插队到 **4号（x）** 的后面。



在链表中插入新元素，无需大面积移动数据，只需要**修改前后元素的指向**即可。

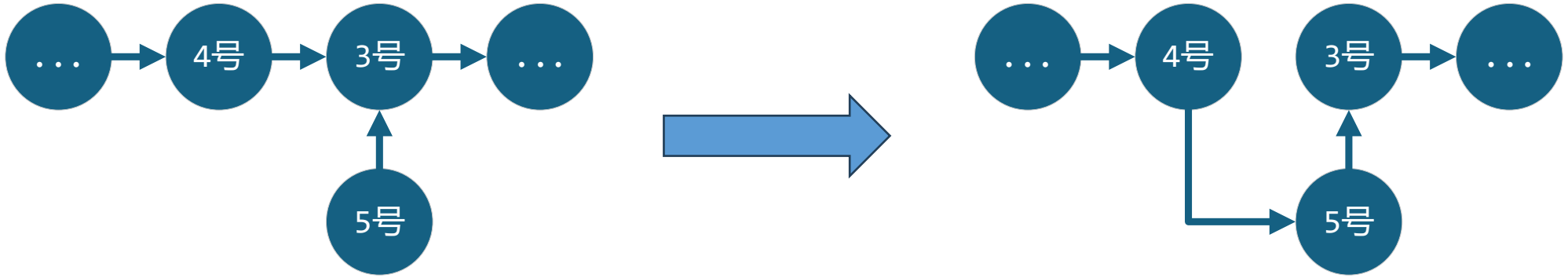
插入元素第一步：新节点连后(连接后节点)

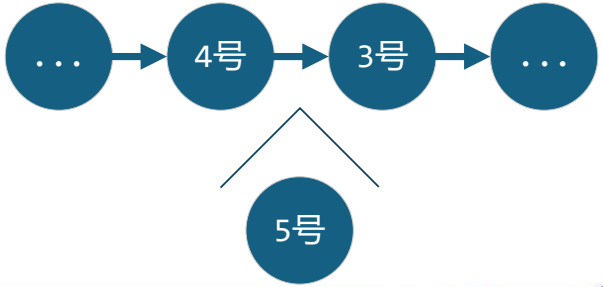
5号节点首先需要知道**自己该连在谁前面**。它查询4号节点原本的后继是谁（结果是3号）。于是，5号节点将自己的后继设定为3号。此时，5号和3号建立了联系。



第二步：前节点断后连新(连接前节点)

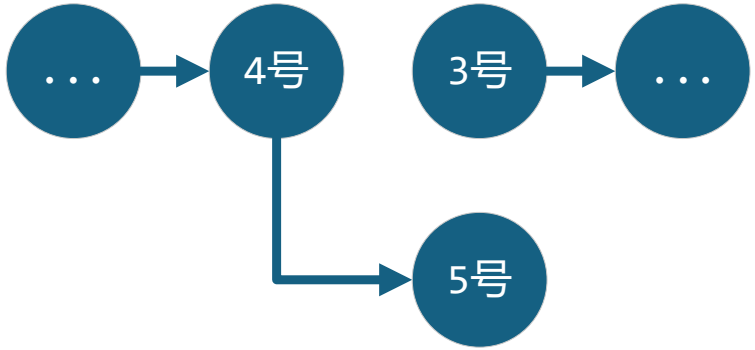
4号节点随后更新自己的信息，将它的后继从原本的3号，修改为新来的5号。





问：新节点连后 和 **前节点断后连新** 可以调换顺序吗?

不能颠倒顺序。如果先**前节点断后连新**，那么 4号节点会立刻将自己的后继更新为5号。此时，**4号原本的后继是3号 这一关键信息就彻底丢失了。**

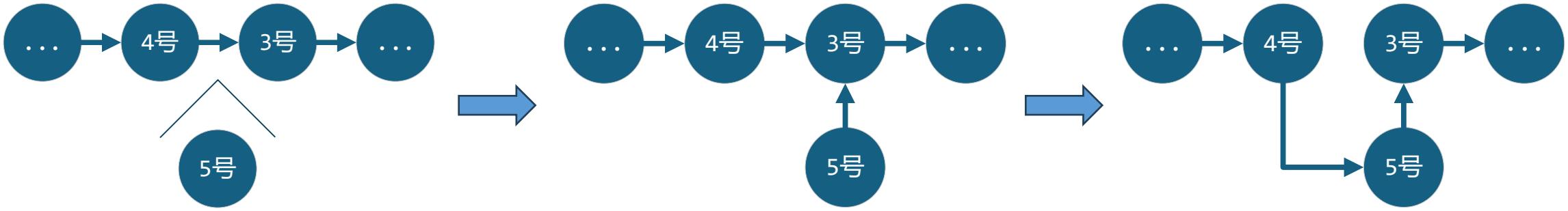


如左图所示，如果先 **前节点断后连新** 那么就无法找到 3号节点去更新它的前驱节点了。

当程序接着**新节点连后**时，5号节点去查询4号的后继，得到的结果变成了5号自己。

因此，链表插入的一个重要原则是：**先连接新节点的后继，再修改前驱节点的后继(先连后，再连前)。**必须先让新节点牵住后面节点的手，前一个节点才能去牵新节点。

将插入新节点过程翻译为代码，可以书写一个insert()函数：

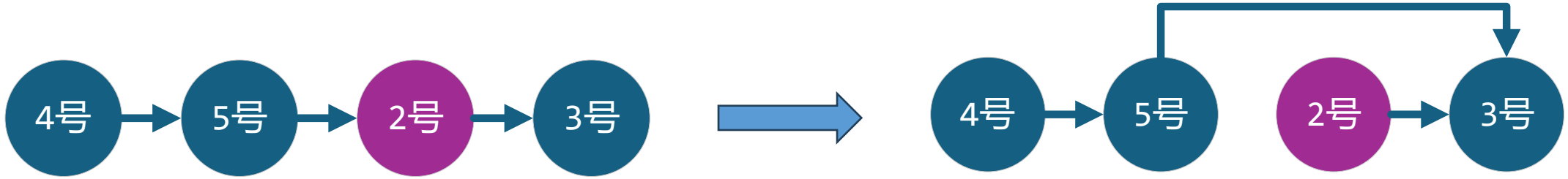



```
void insert(int x, int y) { // 将 y 插入到 x 的后面
    Next[y] = Next[x];      // y 的后继变成 x 原本的后继（5号指向3号）
    Next[x] = y;            // x 的新后继变成 y（4号指向5号）
}
```

而**链表的节点删除操作**同样不需要挪动其余节点，其本质是**绕过需要删除的节点。**

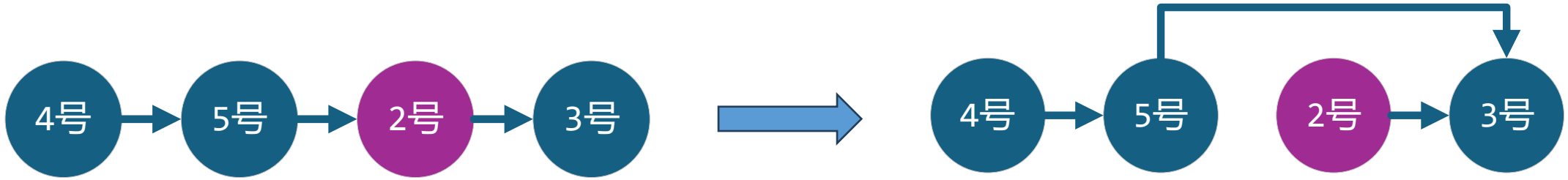
假设队伍当前的顺序是：4号 -> 5号 -> 2号 -> 3号

现在，5号节点后面的 **2号节点** 需要退出队伍。



5号节点的后继是2号。既然2号要离开，5号只需要越过2号，直接指向2号后面的节点即可。

换程序的语言来说：**5号节点（x）的新后继，应该变成它原本后继（`Next[x]`，即2号）的后继（`Next[Next[x]]`，即3号）。所以删除一个元素其实操作的对象是该元素的前驱节点。**



还是用一个函数来执行删除操作（注意参数意义），而代码仅需一行：

```
void remove(int x) {                    // 删除 x 后面的那一个节点
    Next[x] = Next[Next[x]];            // 越过中间的节点，直接连接到下下个节点
}
```

链表

分析上述的插入和删除过程可以发现，无论是加入新节点还是移除旧节点，整个过程仅仅是修改了一到两个 `Next` 数组的指向值。因此，**链表的插入和删除操作的时间复杂度都是极快的 $O(1)$**，这正是链表相比于普通数组最大的优势所在。

然而，这种单向的链表结构也存在一个问题。在 `remove` 函数中，必须提供要删除节点的前驱节点。

假设现在仅仅是要把2号节点删掉，但在单向结构中，2号节点只通过 `Next[2]` 知道自己的后继是3号，却**无法回头**知道自己的前驱是5号。找不到5号，就无法将5号和3号连接起来，删除操作便陷入僵局。

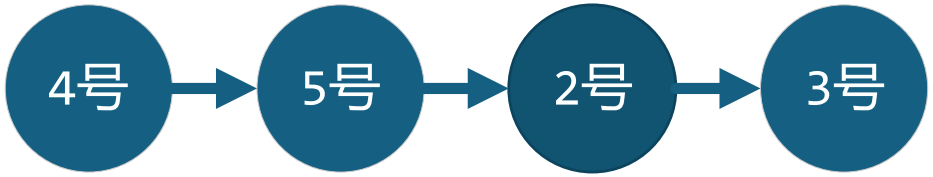
为了解决该问题，链表引入了更完善的设计：**让每个节点不仅记录自己的后继是谁，同时再用一个变量记录自己的前驱是谁，无法回头这个问题便解决了。**这种结构叫做**双向链表。**

链表

为了解决单向链表无法回头的局限性，数据结构中自然而然地衍生出了**双向链表**。

在之前基于数组的模拟中，只有一个 `Next` 数组来记录后继。在双向链表中，需引入一个新的数组，通常命名为 `Pre`）。

- `Next[i] = j`：依然表示编号为 `i` 的元素，后面是 `j`。
- `Pre[i] = k`：表示编号为 `i` 的元素，前面是 `k`。

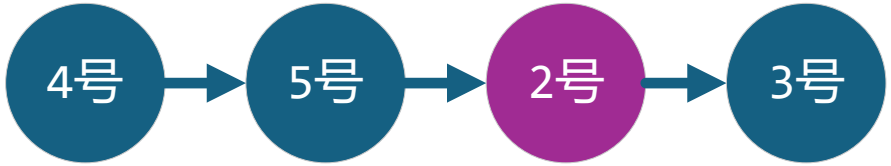


对于左图，双向链表的 `Pre` 数组的表示为：

Pre 数组

下标(编号):	0	1	2	3	4	5
数组值(前驱):	-1	-1	5	2	-1	4

链表



那现在如果**把 2号节点 删掉**，在双向链表中该如何操作呢?

现在的 2号节点 拥有了向前看也可以向后看的能力：可以通过 `Pre[2]` 知道自己前面是 5号。也可以通过 `Next[2]` 知道自己后面是 3号。

因此程序可以直接**提取出前驱和后继节点 5号 和 3号**，并让它们相互连接在一起：让 5号的后面变成 3号。让 3号 的前面变成 5号。

这样，2号 节点就被删除掉了。但是要注意**除了更新 Next 数组的值，也需要更新 Pre 数组的值**，对于删除 2号 节点而言，Pre 数组需要将 3号 的前驱节点设置为 5号(原前驱的前驱)，代码也就是 `Pre[3] = Pre[Pre[3]];`

链表

双向链表使得**双向遍历**（从头到尾，或者从尾到头）成为可能，同时也让**任意已知节点的删除和插入**变得轻而易举，时间复杂度依然保持在 $O(1)$。

但是在进行插入和删除时，而双向链表需要同时维护两个数组，逻辑上需要更加细心。

总而言之，双向链表是单向链表的升级，多了向前遍历的能力，在实际的代码编写与软件开发中，为了应对复杂的增删查改需求，底层大多采用的都是这种双向链表结构。

而不管是什么链表，核心始终都是 `Next` 数组。它能赋予了数据指向的能力。

通过 `Next` 数组的这个关系网，就可以建立起 `i` 指向 `j` 的逻辑顺序。整个数据的顺序就是靠这样一条条关系给串联起来的。

我们通过 `Next` 数组来模拟表达一个链表的方式叫做**静态链表(因为数组的大小通常固定)。**

例15-11 排队模拟(B0223)

你需要实现一个类似于排队的数据结构，维护一个最初只包含元素 1 的表（其中元素 1 没有前驱，也没有后继），支持以下五种操作：

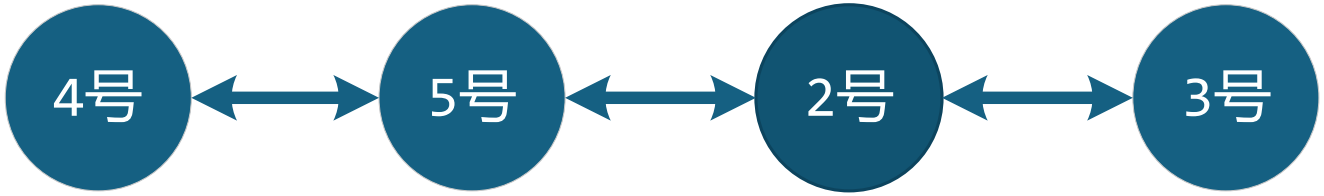
1. $ins_back(x,y)$： 将元素 y 插入到 x 后面
2. $ins_front(x,y)$：将元素 y 插入到 x 前面
3. $ask_back(x)$：询问 x 后面的元素
4. $ask_front(x)$：询问 x 前面的元素
5. $del(x)$：从表中删除元素 x（不改变其他元素的先后顺序）

对于每个 ask_back 和 ask_front 操作，输出询问结果。如果询问的元素不存在或没有前驱/后继，输出$NULL$。

链表

首先分析这个题目。我们需要维护一个动态的队伍，支持五种操作：**在任意位置的前/后插入，查询前/后的元素，以及删除任意元素。**

其中 **在任意位置的前/后插入** 和 **删除任意元素** 是链表的强项，时间复杂度为O(1)，可是对于 **查询前/后的元素** 链表就需要 O(n) 的时间复杂度了有些慢了。有什么优化的办法呢?



对于链表而言，我们一般保存的都是起点位置，通过起点位置就可以把整个数据串起来，但是代价就是查询一个元素需要从起点开始遍历，直到找到对应元素为止

链表

分析一下本题的约束：`1 <= x, y <= 100`：也就是所有同学的编号最大不会超过100。

同学的编号最大只有100。这意味着我们可以开一个大小为 101 的数组，直接用**同学的编号（值）来当做数组的下标**，通过 **数组[同学的编号]** 直接查询结果(也就是桶数组的思想)！

- `int R[101]; // R` 代表 Right，`R[i]` 存储 i 右边的元素是谁。
- `int L[101]; // L` 代表 Left，`L[i]` 存储 i 左边的元素是谁。

比如想知道5号的前驱后继信息，就不再需要从头遍历，而是直接去看数组的第5个位置就行了。通过这种方式，将查找操作的时间复杂度从$O(n)$降到了 $O(1)$，这其实就是也就是前面讲解的 Pre、Next 静态链表数组。

代码实现：

$ask_back(x)$：询问 x 后面的元素

$ask_front(x)$：询问 x 前面的元素

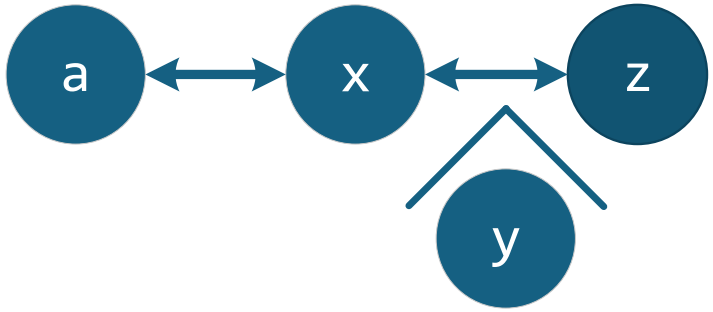
```
void ask_back(int x) {
    if (R[x] == 0) {
        cout << "NULL" << endl;
    } else {
        cout << R[x] << endl;
    }
}

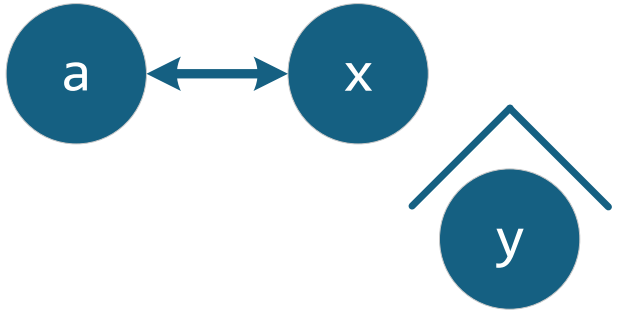
void ask_front(int x) {
    if (L[x] == 0) {
        cout << "NULL" << endl;
    } else {
        cout << L[x] << endl;
    }
}
```

链表

代码实现：$ins_back(x,y)$：　将元素 y 插入到 x 后面

链表的插入原则是：先连后，再连前。






```
void ins_back(int x, int y) {
    int z = R[x]; // 先暂存x右边的人z

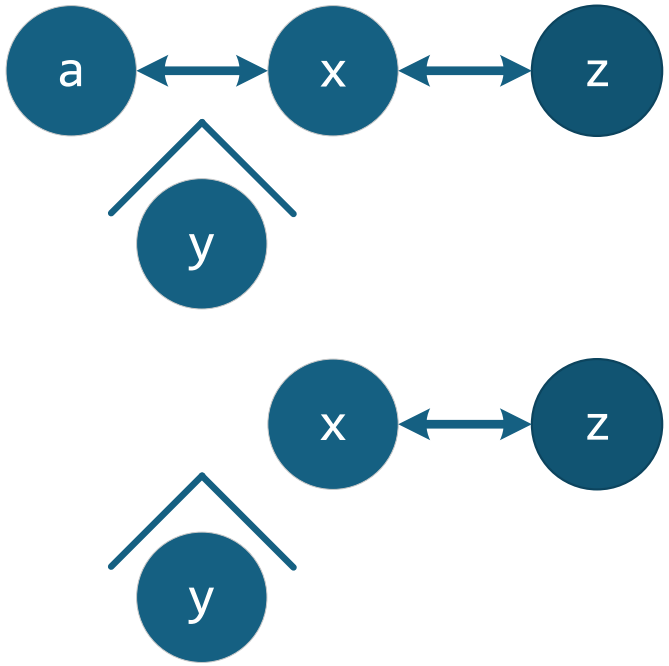
    R[y] = z;     // y的后继变成z
    R[x] = y;     // x的后继变成y

    L[y] = x;     // y的前驱变成x
    if (z != 0) { // 如果z确实存在
        L[z] = y; // z的前驱变成y
    }
}
```

链表

代码实现：$ins_front(x, y)$：将元素 y 插入到 x 前面

链表的插入原则是：先连后，再连前。



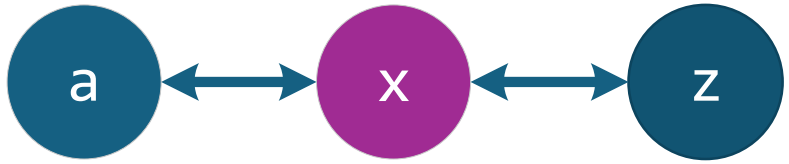

```
void ins_front(int x, int y) {
    int a = L[x]; // 先暂存x左边的人a

    R[y] = x;     // y的后继变成x
    if (a != 0) { // 如果a确实存在
        R[a] = y; // a的后继变成y
        L[y] = a; // y的前驱变成a
    }

    L[x] = y;     // x的前驱变成y
}
```

链表

代码实现：$del(x)$：从表中删除元素 x（不改变其他元素的先后顺序）



x 离开后，a 和 b 需要直接互指。

1. a 的后继需要越过 x，直接指向 b。
2. b 的前驱也需要越过 x，直接指向 a。

```
void del(int x) {
    int a = L[x]; // x左边的人是a
    int b = R[x]; // x右边的人是b

    // 如果a确实存在
    if (a != 0) {
        R[a] = b; // a的后继变为b。
    }
    // 如果b确实存在
    if (b != 0) {
        L[b] = a; // b的前驱变为a。
    }
    // 可选,可以把x的连接清空
    L[x] = 0;  R[x] = 0;
}
```

链表

之前刚刚用数组来模拟链表，例如开了一个 `L[1005]` 和 `R[1005]`。这叫**静态链表**。

但是静态链表有个问题：**你必须要提前知道链表中的元素数量，并且如果利用元素值作为下标则必须要求下标的最大值不能太大。**

如果为了安全，直接开一个 1个亿 大小的数组，结果只有 2 个元素，那剩下的 9999 万个位置全浪费了，这在真实的软件开发中是绝对不允许的，而且如果元素值非常大的话，也无法使用传统的桶数组来将元素值作为数组下标了。

那么有没有一种办法，**来一个人，就找计算机要一个位置；走一个人，就把位置还给计算机**？完全按需分配，一点不浪费？

是可以做到的，这就要用到 C++ 的高级语法—**动态内存分配。**

链表

在数组里，我们用**数组下标** 来表示元素的位置。 但在计算机的内存中，如果要访问一个元素的时候不是通过数组下标，而是通过**内存地址**。

我们也可以**通过记录地址的办法来创建链表**，创立一个节点的结构体，之前在静态数组中记的是前驱和后继的**数组下标**，而现在分别去记下 **前驱和后继节点的内存地址（指针）**

而具体的地址值是多少，我们则通过**动态内存分配**的办法，去为前驱节点和后继节点手动创立储存空间并分配给对应节点。

```
struct Node {
    int value;      // 节点的编号/值
    Node* prev;     // 指向前驱节点的指针
    Node* next;     // 指向后继节点的指针
};
```

注意这里的 `Node*`。带个星号 `*`，它代表着是**指向节点地址的指针，内部**存放的是内存地址。

如果指向得节点是空的，C++中统一会将这个指针就指向一个特殊的空地址：`nullptr`。

链表

现在假如有一个元素 y 需要加入链表了。目前要给他一块空间储存数据，应该怎么办？

那么向操作系统的内存中申请空间，在 C++ 里，手动申请内存空间的关键字是：**new**。

例如：

```
Node* p = new Node;
```

```
int* p = new int;
```

```
char* p = new char;
```

而这句代码，对于计算机而言会发生三件事：

1. `new Node`：计算机立刻在内存中，找到了一块能放下 `Node` 的空间，计算机会把这块空间分配出来并标记归属的数据类型是 `Node`。
2. `new Node` 就会把分配好的这块内存地址返回。
3. `Node* p = ...`：把这个分配好的内存地址，保存在了一个叫 `p` 的指针变量里。

总结：`Node* p = new Node;` 的意思是我手动请操作系统新建一个 `Node` 变量，并把这个新变量的内存地址存入名为 `p` 的指针变量中。从此以后，我们就可以通过 `p` 这个地址去访问那个新创建的 `Node` 变量了，这块内存地址就永远不会被程序被动回收，随时都可访问。

但是，因为 `p` 存的是地址，我们不能用之前普通变量的点 `.` 来访问结构体里面的内容，必须用一个**箭头符号 `->`**（代表顺着地址走过去）。

```
p->value = x;       // 顺着 p 找到新节点，把它的值设为 x
p->prev = new Node; // 创立新节点的前驱节点
p->prev->value = y; // 将新节点的前驱节点的值位置为 y
```

那么如何在动态链表中实现 ins_back 和 ins_front 这两个操作呢?

1. 首先需要找到值为 x 的节点

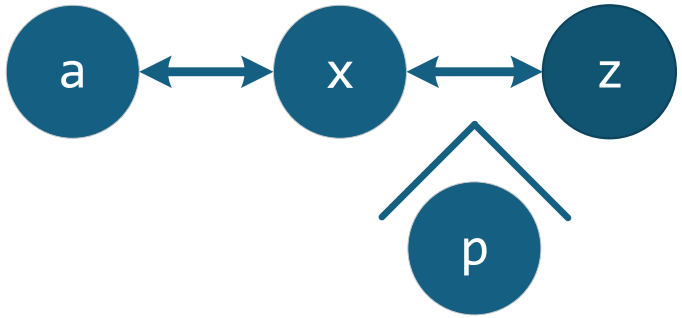
在动态链表中，我们不像数组那样可以直接用下标来定位。链表只有一个开始 / 结束节点——head/tail 指针(双向链表)。所以必须写一个查询函数，从头开始挨个查询直到找到目标。

```
// 从头开始，寻找值为val的节点，如果找到了，就返回那个节点的指针，反之就返回nullptr
Node* findNode(Node* head, int val) {
    Node* current = head; // 从起点开始查询
    while (current != nullptr) {
        if (current->data == val)    return current; // 找到了，返回它的地址
        current = current->next; // 没找到，去下一个节点继续查询
    }
    return nullptr; // 整个队伍都问完了，没找到
}
```

链表

已经找到了要插入那个位置节点 x 。现在要把新节点 p 插入到 x 的后面。插入的逻辑和之前一模一样，只不过写法变了，要将前驱与后继的内存地址改为对应的新的地址。

场景： 编写 ins_back 函数，例如现在需要把 p 插入到 x 和 z 中间(插入到 x 后面)。



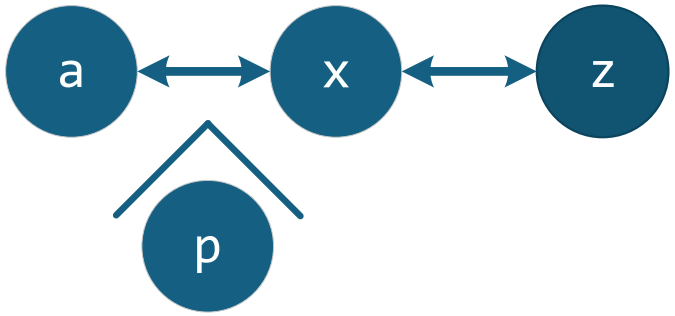

```
// 指针 x 指向我们要操作的节点，指针 p 指向新创建的节点
void ins_back(Node* x, Node* p) {
    Node* z = x->next; // 暂存x的后继z

    p->next = z;       // 先连后，p 的后继变成 z
    x->next = p;       // 再连前，x 的后继变成 p

    p->prev = x;       // p 的前驱变成 x
    if (z != nullptr) { // 先判断 z 存不存在
        z->prev = p;    // z 前驱连向新来的 p
    }
}
```

链表

而 ins_front 的实现与刚刚的代码是对称的，例如现在需要把 p 插入到 x 前面。



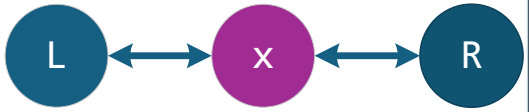

```
// 指针 x 指向我们要操作的节点，指针 p 指向新创建的节点
void ins_front(Node* x, Node* p) {
    Node* a = x->prev; // 暂存x的前驱a

    p->next = x;          // 先连后，p 的后继变成 x
    if (a != nullptr) { // 先判断 z 存不存在
        a->next = p;        // 再连前，p 的后继变成 a
        p->prev = a;        // p 的前驱变成 a
    }
    x->prev = p;     // x 前驱连向新来的 p
}
```

链表

del() 函数的逻辑实现与之前的删除逻辑仍然一致，但是要一定注意动态内存分配需要**手动释放内存空间**，例如现在需要删除节点 x 。

```
void del(Node* x) {
    Node* L = x->prev;    Node* R = x->next; // 获取前后节点
    if (L != nullptr) {
        L->next = R; // 前驱节点越过 x 直接指向后继
    } else {
        head = R; // x没有前驱，则 x 是起始节点，删除后更新起始节点为 x 的后继
    }
    if (R != nullptr) {
        R->prev = L; // 后继节点越过 x 直接指向前驱
    } else {
        // 因为是双向链表，为了方便反向遍历，所以有一个指向末尾节点的指针 tail
        tail = L; // 代表删除的是末尾节点，更新末尾节点的指针为x的前驱
    }
    delete target; // 手动释放内存，语法为: delete 指针变量名
}
```



而对于查询前驱后继元素，目前我们学习的动态链表就只能进行挨个查询了

```
// 询问 x 后面的元素
void ask_back(int x) {
    Node* target = findNode(head, x); // 挨个查询O(n)

    // 如果没这个元素或其不存在后继
    if (target == nullptr || target->next == nullptr) {
        cout << "NULL" << endl;
    } else {
        cout << target->next->value << endl; // 输出后继
    }
}
// 询问 x 前面的元素
void ask_front(int x) {
    // 与 ask_back基本一致，只不过变为 target->prev>value
}
```

动态链表总结（指针法）

核心： `struct` 定义节点，包含 数据 和 指向前后节点的**指针**（`Node*`）。

原理： 使用 C++ 的 `new` 和 `delete` 关键字，在程序的堆（Heap）内存中按需动态地创建和销毁节点。

优点：

- 灵活：内存使用完全按需分配，来一个节点 `new` 一个，走一个节点 `delete` 一个，空间利用率极高。
- 无值域限制：节点的 `value` 可以是任意类型，任意大小。
- **在以后的二叉树、图数据结构中，还会多次用到通过指针创建节点的情况**，所以更推荐大家使用动态链表（而关于查询效率 O(n) 的问题在学习了哈希表章节后可以优化为平均O(1)）

例15-11 队列安排(B0224)

一个学校里老师要将班上 N 个同学排成一列，同学被编号为 $1 \sim N$，他采取如下的方法：

1. 先将 1 号同学安排进队列，这时队列中只有他一个人；
2. 2 ～ N 号同学依次入列，编号为 i 的同学入列方式为：老师指定编号为 i 的同学站在编号为 $1 \sim (i-1)$ 中某位同学（即之前已经入列的同学）的左边或右边；
3. 从队列中去掉 M 个同学，其他同学位置顺序不变。

在所有同学按照上述方法队列排列完毕后，老师想知道从左到右所有同学的编号。

对于这道题目，其通过链表的解决方法基本与上题保持一致，就留做课后习题了

链表

C++ 标准库中的双向链表：list

在 C++ 标准模板库（STL）中，其实也为我们提供了**双向链表这一数据结构的实现，它叫做：list**。它将复杂的指针操作和内存管理细节隐藏起来，为程序员提供了一套高效、安全且易于使用的类。要使用 list，必须首先包含其头文件 #include <list>。

1. 创建与初始化

创建空链表： list<int> a;

- 其定义了一个名为 a 的空链表，链表中的元素类型为 int。
- 语法为 list<数据类型> 变量名;

使用列表初始化将链表初始化为指定值：

- list<int> l = {1, 2, 3};

从现有的数组中进行初始化

```
vector<int> arr = {1, 2, 3};
list<int> l(arr.begin(), arr.end());
```

`list` 可以接受两个迭代器作为参数，用于指定一个区间 `[first, last)`。

上述代码将数组 `arr` 的全部元素作为链表 `l` 的初始值。因此，`l` 被初始化为 `{1, 2, 3}`。

获取链表中元素数量：

```
long long count = l.size();
```

```
list<int> l = {1, 2, 3};
```

获取边界迭代器： `list` 提供了两个成员函数来获取代表其边界的迭代器：

- `auto b_it = l.begin()`：返回一个指向链表第一个元素(1)的迭代器。
- `auto e_it = l.end()`：返回一个指向链表末尾元素之后的位置的迭代器。它本身不指向任何有效元素，仅作为循环结束的标志。

迭代器的移动： 由于 `list` 是双向链表，其迭代器支持双向移动：

- `it++`：使迭代器向前移动到下一个元素。例如 `b_it++` 就指向了元素2。
- `it--`：使迭代器向后移动到前一个元素。

链表

```
list<int> l = {1, 2, 3};
```

`list` 以其高效的插入和删除操作著称，这些操作的时间复杂度均为 $O(1)$。

在首尾添加/删除元素：

- `a.push_front(x)` / `a.push_back(x)`：分别在链表的头部或尾部插入新元素 `x`。
- `a.pop_front()` / `a.pop_back()`：分别删除链表的头部或尾部元素。

在指定位置插入/删除元素：

- `a.insert(it, x)`：在迭代器 `it` 所指向的位置插入一个新元素 `x`。
- `a.erase(it)`：删除迭代器 `it` 所指向的元素。注意：该操作会使**被删除元素的迭代器失效**，但其他**已经存在的迭代器不会失效**，且其返回值为被删除迭代器 `it` 的下一个元素。例如 `l.erase(l.begin());` 会返回指向元素 `2` 的迭代器。

遍历

访问 `list` 中的所有元素通常通过迭代器和循环结构来完成。

- **使用基础 `for` 循环遍历：**

```
for (auto it = a.begin(); it != a.end(); it++) {
        // 使用 *it 解引用来获取迭代器指向的元素值
        cout << *it << " ";
}
```

遍历

访问 `list` 中的所有元素通常通过迭代器和循环结构来完成。

- **使用基于范围的 `for` 循环遍历：**

```
for (auto val : a) {
    // val 就是链表中的每个元素，遍历顺序为从 l.begin() 到 l.end()
    cout << val << " ";
}
```

现在就可以通过 STL 中的 list 来方便的使用链表了，但链表的内部实现方法也必须掌握

例15-13 约瑟夫问题(B0225)

n 个人围成一圈，从第一个人开始报数,数到 m 的人出列，再由下一个人重新从 1 开始报数，数到 m 的人再出圈，依次类推，直到淘汰 $n-1$ 名小朋友，请输出依次出圈人的编号。**请使用链表解决本问题。**

本题目也较为简单，创建一个链表去模拟报数淘汰的过程即可，故也留做课后习题

链表

链表总结

链表的诞生，是为了解决**数组物理位置的连续性**所导致的特性。因为它会使得数组在插入和删除操作上效率低下 $(O(n))$。

链表的核心思想是：

- **物理储存离散，但逻辑连续：**数据元素（节点）可以散落在内存的任何角落。
- **指针串联：**每个节点内部都包含一个指针，指向下一个（或上一个）节点的内存地址，从而在逻辑上形成一条线性的序列。

在使用过程中，推荐使用动态链表 或 STL中的 list 来实现链表

动态链表实现主要是靠：`struct Node` + `Node*` 指针 + `new/delete`。

- 原理：使用内存地址（指针）连接节点，按需在堆上动态分配和释放内存。