C++算法将以 链表（二）

一、 引言

1.1 问题引入：静态链表的局限性

前面我们学习了用数组来模拟链表（静态链表），比如开一个 `Next[1005]` 和 `Prev[1005]` 数组。这种方法在算法竞赛中很常见，因为它写起来快。但是它有两个缺点：

1. **空间浪费或溢出**：你必须提前估计好元素的最大数量（比如开 1000 万个大小的数组）。如果最终只有 10 个人排队，剩下的 999 万多个位置就白白浪费了；如果来了 1001 万个人，程序就会因为数组越界而崩溃。
2. **值与下标的绑定**：如果我们要存的数据本身非常大（比如学生的学号是 `202300001`），我们显然不能开一个大小为 2亿 的数组来把学号当作下标。

1.2 价值阐述

有没有一种办法：来一个人，我就找计算机要一个刚好的位置；走一个人，我就把这个位置还给计算机？完全**按需分配，一点都不浪费**？

这就是我们今天要学习的——**动态内存分配**与**指针**。基于此构建的**动态链表**，突破了数组容量的限制，是计算机底层系统中极为常见的数据结构。同时，我们还将学习 C++ 已经为我们造好的双向链表工具——`list`。

二、 动态链表的核心：指针与 `new/delete`

2.1 定义节点（`struct Node`）

在静态链表中，用数组的**下标**来表示位置（`Next[i] = j`）。
在动态链表中，直接用内存的**真实地址**来表示位置。存放地址的变量，叫做**指针**。

```
struct Node {
    int value;      // 数据域：节点存储的值
    Node* prev;     // 指针域：指向前驱节点的地址
    Node* next;     // 指针域：指向后继节点的地址
};
```

注意：`Node*` 中的星号 `*`，代表这是一个**指针类型**。`prev` 和 `next` 里存的不是一个数字，而是一个内存地址。如果一个节点没有后继（比如它是尾节点），我们让指针指向一个特殊的空地址：`nullptr`。

2.2 向操作系统 要 和 还 内存

- **要内存（`new`）**：
  当一个新元素加入时，我们需要在内存中为他申请一块空间。

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

  这句代码做了三件事：

  1. `new Node`：计算机在堆区找到一块刚好能放下 `Node` 结构体的内存空间。
  2. 返回这块内存空间的**内存地址**。

3. `Node* p = ...`：把这个地址保存在指针变量 `p` 中。以后我们就可以通过 `p` 找到这个新元素了。

- **访问指针指向的内容 (`->`)**：
  因为 `p` 存的是地址，我们不能用普通的 `.` 来访问数据，必须用**箭头符号** `->`（代表顺着地址走过去看）。

```
p->value = 10;   // 把这块内存里的 value 设置为 10
p->next = nullptr; // 把这块内存里的 next 设置为空
```

- **还内存 (`delete`)**：
  当再也不需要使用该元素时，就必须把这块空间还给操作系统，否则会导致**内存泄漏**。

```
delete p; // 释放 p 指向的那块内存空间
```

三、 动态双向链表的实战操作

掌握了 `new` 和 `->`，那么动态链表的操作逻辑和静态链表**完全一模一样**，只是写法换成了指针！

3.1 查找节点 (O(n))

在动态链表中，我们只有头节点 `head` 的地址，要找某个值，必须从头挨个顺着 `next` 指针往下问。

```
Node* findNode(Node* head, int val) {
    Node* current = head; // 从起点出发
    while (current != nullptr) { // 只要还没走到空（尽头）
        if (current->value == val) {
            return current; // 找到了，返回这个节点的地址
        }
        current = current->next; // 没找到，顺着指针去下一个节点
    }
    return nullptr; // 找遍了也没找到，返回空地址
}
```

3.2 插入节点：`ins_back(x, p)`

场景：找到节点 `x`，把新创建的节点 `p` 插在 `x` 和它的后继 `z` 之间。

```
void ins_back(Node* x, Node* p) {
    Node* z = x->next; // 提前找到 x 的后继 z

    // 设置新节点 p 的前后驱
    p->next = z;       // p 的后继连向 z
    p->prev = x;       // p 的前驱连向 x

    // 更新原有的 x 和 z
    x->next = p;       // x 的后继改为 p
    if (z != nullptr) { // 如果 z 存在（x 不是尾节点）
        z->prev = p;   // z 的前驱改为 p
    }
}
```

3.3 删除节点：`del(x)`

场景：将节点 `x` 从链表中删除，并释放内存。

```
// 全局变量有 Node* head 指向开始元素的头指针; 和指向结束位置的尾指针 Node* tail;
void del(Node* x) {
    Node* L = x->prev; // x 的前驱
    Node* R = x->next; // x 的后继

    // 处理前驱的连接
    if (L != nullptr) {
        L->next = R;    // 前驱直接连向后继
    } else {
        head = R;       // 如果没有前驱，说明 x 是头节点，删除后头节点变更为 R
    }

    // 处理后继的连接
    if (R != nullptr) {
        R->prev = L;    // 后继直接连向前驱
    } else {
        tail = L;       // 如果没有后继，说明 x 是尾节点，更新尾节点为 L
    }

    delete x; // 手动释放 x 占用的内存空间
}
```

四、 C++ 标准库：`list`

在日常开发和算法竞赛中，C++ 标准库（STL）已经为我们写好了一个方便、极其强大的双向链表工具：`list`。而使用前需要包含头文件：`#include <list>`

4.1 创建与初始化

```
#include <list>
using namespace std;

list<int> a;                    // 创建一个空的存放 int 的双向链表
list<int> l = {1, 2, 3};        // 创建并初始化为 1, 2, 3

// 获取元素数量
long long count = l.size();
```

4.2 特定增删操作 (O(1))

- **首尾操作**：
  - **在头部/尾部插入元素**：`l.push_front(x);` / `l.push_back(x);`
  - **删除头部/尾部元素**：`l.pop_front();` / `l.pop_back();`
- **如何表示每个元素的位置?** ：
  STL 提供了**迭代器**，用于表示链表中的位置。
  - **指向第一个有效元素**：`auto b_it = l.begin();`

    - **指向最后一个元素的下一个位置（没有实际元素，仅作为结束标志）**：`auto e_it = l.end();`
    - **移动**：`it++` (向后移)，`it--` (向前移)。
    - **访问迭代器指向的元素(解引用)**：`cout << *b_it;`，其中`b_it`是指向第一个有效元素的迭代器，那么本代码会输出第一个元素的值。
- **指定位置增删 (O(1))**：
    - **在迭代器 `it` 指向的位置前插入 `x`**：`l.insert(it, x);`
    - **删除 `it` 指向的元素**：`l.erase(it);`

4.3 遍历链表

遍历链表更加推荐使用基于范围的 `for` 循环：

```
list<int> a = {10, 20, 30};

// val 依次代表链表 a 中的每一个元素
for (int val : a) {
    cout << val << " ";
}
// 输出: 10 20 30

// 也可以使用传统的 for 循环方式
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 << " ";
}
```