链表

链表是个非常简单的数据结构。本文只考虑单链表，双链表等更加复杂的链表都是类似的。

链表节点定义如下（我们假设节点里只存一个int）：

```
struct ListNode {
  int x;
  ListNode *next;
  ListNode() = default;
  ListNode(int val) : x(val), next(nullptr) {}
};
```

`next`表示当前节点的下一个节点的地址。

静态链表

动态链表的所有操作都非常简单，这里就不讲了。但是，动态链表在竞赛代码中几乎不会用，它有一个严重的缺陷：在动态分配内存的时候，我们需要新用到一个节点，就new一次，而new一次的时间代价是非常昂贵的。更优的方法是利用“内存池”的思想，先将可能要用的节点一次性全new出来，然后每次需要的时候，就从池子里拿出来一个返回。这样效率更高的原因是，程序向操作系统申请堆空间的时候，申请n次每次申请1个int的速度，远远慢于一次性申请n个int。也就是说，一次new的时间代价，跟要申请的内存大小几乎无关。

下面给出一个代码案例来帮助理解。我们先不考虑内存泄漏的问题。

`head_file.h`:

```
#include <iostream>
using namespace std;
const int N = 1e7 + 10;
int n = 1e7;

struct ListNode {
  int x;
  ListNode *next;
  ListNode() : x(0), next(nullptr) {}
  ListNode(int val) : x(val), next(nullptr) {}
};
```

main1.cc

```cpp
#include "head_file.h"

ListNode &get(int x) {
  static ListNode *node = new ListNode[N];
  static int idx = 0;
  node[idx].x = x;
  return node[idx++];
}

int main() {
  ListNode *head = new ListNode(0), *tmp = head;
  for (int i = 1; i <= n; i++) {
    head->next = &get(i);
    head = head->next;
  }

  while (tmp) {
    cout << tmp->x << endl;
    tmp = tmp->next;
  }
}
```

main2.cc

```cpp
#include "head_file.h"

int main() {
  ListNode *head = new ListNode(0), *tmp = head;
  for (int i = 1; i <= n; i++) {
    head->next = new ListNode(i);
    head = head->next;
  }

  while (tmp) {
    cout << tmp->x << endl;
    tmp = tmp->next;
  }
}
```

第一个版本里，是一次性new出10^7个节点，然后每次需要的时候，就返回一个给调用方；第二个版本，是每次需要的时候就new一下。

下面给出一个脚本，这个脚本可以跑一个进程10次，然后求一下运行时间的平均值。

`run.sh`

```
#!/bin/bash

executable=$1
runs=10
total_time=0

for ((i=1; i<=runs; i++))
do
    echo "Run #$i"
    runtime=$( { /usr/bin/time -f "%e" ./$executable > /dev/null; } 2>&1 )
    runtime_ms=$(echo "$runtime * 1000" | bc)
    echo "Time: $runtime_ms ms"
    total_time=$(echo "$total_time + $runtime_ms" | bc)
done

average_time=$(echo "scale=3; $total_time / $runs" | bc)
echo "Average Time: $average_time ms"
```

实际跑下来，第一个版本的运行速度比第二个版本要快很多。读者可以自行验证。但是，即使是快，也没有快到10^7倍这么夸张。原因主要是：

1. 如上所述，new多次要比new一次慢。但是在实际操作中，有的现代的编译器是有能力对此做出优化的，所以很有可能两者差距并没有那么大。
2. 缓存友好。显然第一个版本的节点们内存地址都是相邻的，从而非常缓存友好。这是第二个版本所不具备的。其实“快”主要是快在这里。

数组模拟链表

数组模拟链表本质上就是静态链表，只不过写法上完全舍弃了指针。

需要开若干数组，示意代码如下：

```
const int N = 1e5 + 10;
int head, e[N], ne[N], idx;
```

其中`e`和`ne`的下标都代表着边，而`head`本身存储的是该单链表的第一条边。也就是说，`e[i]`和`ne[i]`两个数组其实都表示的是编号为`i`的那条边的信息。而`idx`表示接下来要使用的那条边的编号，其一开始值为`0`。

具体来讲：`e[i]`表示的是编号`i`的边指向的点的编号，而`ne[i]`表示的是编号`i`的边指向的点继续出发的边的编号。如果`ne[i] = -1`，表示的就是`e[i]`这个点的出边是`nullptr`，也就是`e[i]`没有出边。即，`-1`在这里有着表示空指针的特殊含义。

上述说法太抽象，我们可以先看一下这样表示的链表的具体操作。

1. 初始化：

```
void init() { head = -1; }
```

因为首先链表还没有任何节点，所以链表的“第一条边”的编号当然应该是`-1`，这也是我们对`-1`这个数字的特殊含义的约定，即`head = -1`。

2. 在链表头加入值为`x`的节点（这里“值”可以理解为之前`ListNode`里的那个int）：

```
void add(int x) {
  e[idx] = x, ne[idx] = head, head = idx++;
}
```

注意几点：

数组模拟链表通常情况下都只在链表头添加节点，实操过程中非常少见不在头部添加节点的。

头插法的操作和动态链表一模一样：首先`idx`是接下来要使用的边，`e[idx] = x`意思是让这条边指向的点赋值为`x`；接着`ne[idx] = head`意思是，新用的这条边，其“下一条边”赋值为`head`，这么做的原因是`head`是链表的最头上的边，既然我们采用头插法，那当然新开的边的“下一条边”应当赋值为`head`；最后让`head = idx++`，因为我们头部插了一个点，那个点的入边下标就是`idx`，而这条边是头插结束后的新链表的头边，所以当然要将`head`赋值为`idx`；而`idx++`含义是，下面要用的边的编号应该加`1`，也非常显然；

3. 将编号为`i`的边指向的节点删除

```
i == head ? head = ne[head] : ne[i] = ne[ne[i]];
```

这也很好理解，如果要删头边，那就将`head`向后挪一个；否则需要将`ne[i]`向后挪一个。

注意这里删除节点的时候，我们是不会“回收”边的，而仅仅是那些边再也用不到了而已。也就是说，原来的`ne[i]`这个编号的边，以后再也不会用了，即使新开节点也不会用它。这仅仅是为了代码方便和效率。

例题：https://blog.csdn.net/qq_46105170/article/details/113798838

双链表

例题：https://blog.csdn.net/qq_46105170/article/details/113801111

作业

自己实现动态链表及其所有基本操作

实现数组模拟链表和基本操作

所有例题