链表的基本概念

链表的简介

链表是程序设计中一种重要的动态数据结构，它是动态的进行存储分配的一种结构。

其动态性体现为：

1、链表中元素的个数可以根据需要增加和减少，不像数组，在声明之后就固定不变。

2、元素的位置可以变化，即可以从某个位置删除，然后再插入到一个新的地方。

我们通过几个例子，来深刻的了解一下链表。

例1：

张老师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，然而张老师并没有记录队伍是怎么排的。但好消息是，她还记得排在队首的是明明同学(序号为1)，此外，每个同学都记得排在自己身后的是谁。那么，我们能否使用这些信息，还原出队列的初始状态呢？

很明显是可以的，我们可以建立一个Next数组，索引是每个学生的序号，数组的内容则是排在该名学生之后的下一位同学的编号。如下所示：

Next				
1	2	3	4	5
4	3	5	2	0

根据上面数组的描述，我们很容易得到最初的队列顺序，整个完整顺序如下所示：

1 4 2 3 5

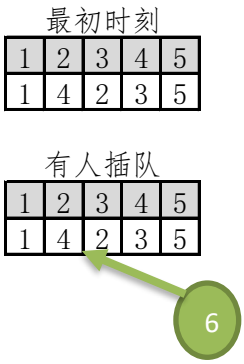
代码也很容易写，示例如下：

```
#include<bits/stdc++.h>
using namespace std;
//记录下一名同学编号的数组
int Next[5] = {4,3,5,2,0};

int main() {
    //从数组中取出下一名同学的索引，并打印
    for (int i = 0; i < 5; i = Next[i]) {
        printf("%d", i);
    }

    return 0;
}
```

像上文中Next数组这样的数据结构，即“**每个节点都能找到相邻节点**”的表，我们就称之为链表。

接下来考虑另外一个问题。本来n名同学在好好的排队，但是来了一名不守规矩的同学(6)号，插队到2号后面，而其余的同学顺序不变。插队之后，队伍是什么样的顺序呢？

我们可能很容易想到一个暴力的做法：使用数组记录排队顺序；发生变化的时候，把插队者放到相应的地方，其余的元素后移一位。如下所示：



代码实现过程如下：

```
#include<bits/stdc++.h>

using namespace std;

//全部同学的顺序
int Seq[10] = {1, 4, 2, 3, 5};

int main() {
    //整体后移
    for (int i = 2; i <= 6; ++i) {
        Seq[i + 1] = Seq[i];
    }
    //插入插队的同学
    Seq[2] = 6;

    return 0;
}
```

这种做法的效率无疑是很低下的，那么有没有更好的方法呢？参考第一问还原队列顺序的思路，我们发现使用链表来处理这个问题会非常的容易。代码如下：

```
//将x插入到head之后
void Insert(int head, int x) {
    Next[x] = Next[head];
    Next[head] = x;
}
```

这时，2号同学因故需要离开，那么整个队列该怎么变化呢？

Next				
1	2	3	4	5
4	3	5	2	0

根据最初的Next链表，我们知道2号同学是排在4号之后的，那么2号同学离开之后，排在2号的下一位，即3号同学就应该排在4号同学之后，因此，我们可以很容易的写出变化代码。

```
//移除x后的元素
void Remove(int x) {
    Next[x] = Next[Next[x]];
}
```

例

实现一个数据结构，维护一张表（最初只有一个元素 1）。需要支持下面的操作，其中 x 和 y 都是 int 范围内的正整数，且都不一样，操作数量不多于 2000：

ins_back(x,y) ：将元素 y 插入到 x 后面；
ins_front(x,y) ：将元素 y 插入到 x 前面；
ask_back(x) ：询问 x 后面的元素；
ask_front(x) ：询问 x 前面的元素；
del(x)：从表中删除元素 x，不改变其他元素的先后顺序。

解析

题目中要求我们能够将数据任意的插入到某个数据的前面或者后面，如果使用上面的单向链表则难以实现，因为使用单向链表，我们只能够知道某一个数据或者前或者后的内容，而无法同时知晓两侧的数据。

所以，本题应该使用双向链表。即我们在一个节点中同时记录每个节点的前驱和后继，这样就可以往两个方向遍历啦。

最初时刻的节点如下：

索引	0
键值	1
前驱	0
后继	0

基本结构代码如下：

```
struct node {
    // 分别记录前驱和后继结点在数组s中的下标
    int pre, nxt;
    // 结点的值
    int key;

    //结构体初始化
    node(int key = 0, int pre = 0, int nxt = 0) {
        this->pre = pre;
        this->nxt = nxt;
        this->key = key;
    }
};

// 一个池。以后想要新建结点，就从s数组里面分配给新结点。
node s[1005];

//最初时刻节点的数量是1
int tot = 1;
```

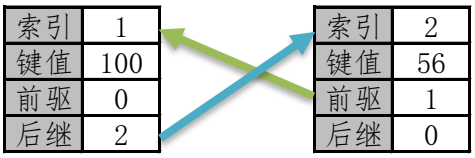
接下来我们需要实现插入函数：ins_back(x,y)，将元素 y 插入到 x 后面。

但是，我们发现，如果想实现插入，我们必须先找到键值为x的节点，因此，我们需要先来实现一个查找函数：find(int x)，即从链表数组中找到我们想要的那个节点。代码如下：

```
int find(int x) {
    int now = 1;
    //链表无法像数组一样索引，
    //因此只能从第一个节点不断的向后寻找
    while (now && s[now].key != x) {
        now = s[now].nxt;
    }
    return now;
}
```

找到目标节点后，我们就可以对齐进行操作了，我们需要把目标节点的后继标记成插入的节点键值，还需要把插入节点的前驱标记成为目标节点的键值。如下图所示，我们在1号节点后插入一个数字76：

原始结构

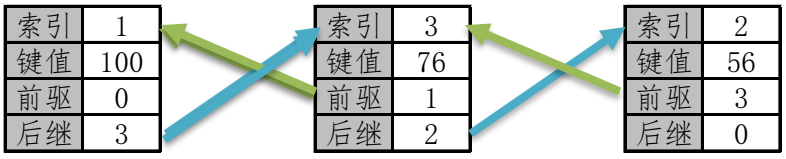


插入后结构

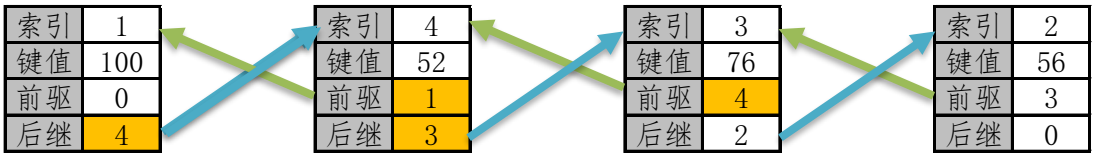

```
void ins_back(int x, int y) {
    int now = find(x);
    //写入新节点的键值、前驱和后继
    s[++tot] = node(y, now, s[now].nxt);
    //更新原后继节点的前驱信息
    s[s[now].nxt].pre = tot;
    //更新当前节点的后继信息
    s[now].nxt = tot;
}
```

向前插入也是类似，这一次，我们将52插入到3号节点之前

原始结构



插入后结构




```
void ins_front(int x, int y) {
    int now = find(x);
    //写入新节点的键值、前驱和后继
    s[++tot] = node(y, s[now].pre,now);
    //更新原前驱节点的后继信息
    s[s[now].pre].nxt = tot;
    //更新当前节点的前驱信息
    s[now].pre = tot;
}
```

接下来，我们需要实现ask_back(x)，询问x元素后的值，很明显，我们只需要找到x的nxt节点的key就好。

```
int ask_back(int x) {
    int now = find(x);
    return s[s[now].nxt].key;
}
```

那么ask_front也是类似：

```
int ask_front(int x) {
    int now = find(x);
    return s[s[now].pre].key;
}
```

最后，需要实现删除逻辑，删除一个元素时，只需让这个元素的前驱的后继变成它的后继。

```
void del(int x) {
    int now = find(x);
    //取出待删除节点的前驱和后继
    int le = s[now].pre, rt = s[now].nxt;
    //前驱的后继等于当前节点的后继
    s[le].nxt = rt;
    //后继的前驱等于当前节点的前驱
    s[rt].pre = le;
}
```

通过以上代码，我们可以得出：相比于数组，链表插入删除快，但是定位（找到第k个）慢。

完整代码

```
#include<bits/stdc++.h>

using namespace std;

struct node {
    // 分别记录前驱和后继结点在数组s中的下标
    int pre, nxt;
    // 结点的值
    int key;

    //结构体初始化
    node(int key = 0, int pre = 0, int nxt = 0) {
        this->pre = pre;
        this->nxt = nxt;
        this->key = key;
    }
};

// 一个池。以后想要新建结点，就从s数组里面分配给新结点。
node s[1005];

//最初时刻节点的数量是1
int tot = 1;

//使用双向链表。首先每次操作之前，都需要知道一个元素
//在表中的编号。
int find(int x) {
    int now = 1;
```

```
    //链表无法像数组一样索引，
    //因此只能从第一个节点不断的向后寻找
    while (now && s[now].key != x) {
        now = s[now].nxt;
    }
    return now;
}

//注意更新后继的前驱以及前驱的后继，这里 tot 代表使
//用了多少节点的位置。
void ins_back(int x, int y) {
    int now = find(x);
    //写入新节点的键值、前驱和后继
    s[++tot] = node(y, now, s[now].nxt);
    //更新原后继节点的前驱信息
    s[s[now].nxt].pre = tot;
    //更新当前节点的后继信息
    s[now].nxt = tot;
}

//注意更新后继的前驱以及前驱的后继，这里 tot 代表使
//用了多少节点的位置。
void ins_front(int x, int y) {
    int now = find(x);
    //写入新节点的键值、前驱和后继
    s[++tot] = node(y, s[now].pre, now);
    //更新原前驱节点的后继信息
    s[s[now].pre].nxt = tot;
    //更新当前节点的前驱信息
    s[now].pre = tot;
}

//只需根据编号获得其后继的值即可。
int ask_back(int x) {
    int now = find(x);
    return s[s[now].nxt].key;
}


int ask_front(int x) {
    int now = find(x);
    return s[s[now].pre].key;
}

//删除一个元素时，只需让这个元素的前驱的后继变成它的后继，
//它的后继的前驱变成它的前驱即可。
void del(int x) {
    int now = find(x);
    //取出待删除节点的前驱和后继
```

```
    int le = s[now].pre, rt = s[now].nxt;
     //前驱的后继等于当前节点的后继
    s[le].nxt = rt;
     //后继的前驱等于当前节点的前驱
    s[rt].pre = le;
}


int main() {

    return 0;
}
```

内置链表：list

C++提供了内置的list容器，它的功能与上文中的双向链表极其相似。我们先来看一下它的内置函数：

```
#include<bits/stdc++.h>

using namespace std;


int main() {
    //====================    多种声明方法 ======================//
    //方法1：
    //定义一个int型的链表
    list<int> nums1;

    //方法2：
    //将数组的前3个元素作为链表的初始值
    int arr[5] = {1, 2, 3};
     list<int> nums2(arr, arr + 3);

    //定义链表的迭代器（指针），即当前数据的位置
    list<int>::iterator it;

    //获取链表开头的迭代器
    list<int>::iterator begin = nums1.begin();
    //获取链表结尾的迭代器
    list<int>::iterator end = nums1.end();

    //迭代器向后移动，可以得到下一个数据
    begin++;
    //迭代器向后移动，可以得到上一个数据
    begin--;

    //在表的开头插入元素
    nums1.push_front(1);
```

```
    //在表的结尾插入元素
    nums1.push_back(1);


    nums1.insert(it, 1);

    //删除链表开头的元素
    nums1.pop_front();
    //删除链表结尾的元素
    nums1.pop_back();

    //删除指定迭代器位置的数据
    nums1.erase(it);

    //遍历链表
    for (it = nums1.begin(); it != nums1.end(); it++) {
        cout << *it;
    }

    return 0;
}
```