山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

学号：202000130198	姓名： 隋春雨	班级： 20.4
实验题目：排序算法		
实验学时：2	实验日期： 2021-10-21	

实验目的：

1、 掌握线性表结构、链式描述方法（链式存储结构）、链表的实现。

2、 掌握链表迭代器的实现与应用。

软件开发环境：

CLION2020

1. 实验内容

1、题目描述：

要求封装链表类，链表迭代器类；

链表类需提供操作：在指定位置插入元素，删除指定元素，搜索链表中是否有指定元素，原地逆置链表，输出链表；

不得使用与链表实现相关的 STL。

输入输出格式：

输入：第一行两个整数 N 和 Q。

第二行 N 个整数，作为节点的元素值，创建链表。

接下来 Q 行，执行各个操作，具体格式如下：

插入操作 ：1 idx val，在链表的 idx 位置插入元素 val;

删除操作 ：2 val，删除链表中的 val 元素。若链表中存在多个该元素，仅删除第一个。若该元素不存在，输出 -1；

逆置操作 ：3，原地逆置链表；

查询操作 ：4 val，查询链表中的 val 元素，并输出其索引。若链表中存在多个该元素，仅输出第一个的索引。若不存在该元素，输出 -1；

输出操作 ：5，使用链表迭代器，输出当前链表索引与元素的异或和。

2、题目描述：

要求使用题目一中实现的链表类，迭代器类完成本题；

不得使用与题目实现相关的 STL；

给定两组整数序列，你需要分别创建两个有序链表，使用链表迭代器实现链表的合并，并分别输出这三个有序链表的索引与元素的异或和。

注：给定序列是无序的，你需要首先得到一个有序的链表。

输入输出格式：

输入：

第一行两个整数 N 和 M；

第二行 N 个整数，代表第一组整数序列；

第三行 M 个整数，代表第二组整数序列。

输出：

三行整数。分别代表第一组数、第二组数对应的有序链表与合并后有序链表的索引与元素的异或和。

2. 数据结构与算法描述　　（整体思路描述，所需要的数据结构与算法）

1. 首先对于每一个链表结点我们需要封装一个结构体，这个类需要有保存的值和指向下一个结点的指针，即为

```
template <class T>
struct chainNode
{
    T element;
    chainNode<T>* next;
```

然后我们需要站在链表的角度对结点进行统一的管理，链表需要查询索引、删除某个值、在某个特定位置插入特定的值、输出题目要求的异或值等功能，因此我们建立链表类如下：

```
template<class T>
class chain
{

public:
    // constructor, copy constructor and destructor
    chain(int initialCapacity = 10);
    chain(const chain<T>&);
    ~chain();


    void indexOf(const T& val) const;
    void erase(T val);
    void insert(int theIndex, const T& theElement);
    void output() const;
```

2. 对于迭代器，我们考虑到它是一个智能指针，需要重载++ ，*，->等运算符，我们平时用*iter 的意思是取出它所指的元素的值，iter->意思是取出它所指的结点，因此构造如下：**值得注意的是**，书上的重载->可能是错的，它返回的是&node->element，这个在 clion 上只能取出来 element，对于 next 指针就不行了。

```
class iterator
{
public:
    iterator(chainNode<T>* theNode = nullptr)
    {
        node = theNode;
    }

    T& operator*() const { return node->element; }
    chainNode<T>* operator->() const { return node; }

    bool operator!=(const iterator right) const
    {
        return node != right.node;
    }
    bool operator==(const iterator right) const
    {
        return node == right.node;
    }
```

3. 我们平时还经常使用到 begin(),end()等函数，这两个函数不应该是 iterator 里边的函数，而应该是 chain 的函数，因为我们使用的时候，都是站在容器的角度来使用该容器的 begin()或者 end()，而不是迭代器
4. A 题的插入操作：我们首先要找到需要插入的位置的前一个结点，然后更改 next 的值，同时考虑到，它可能没有前一个结点（就是说，我们要插入的位置是 fisrtNode），那么这种情况需要特殊处理一下，让其 next 直接指向头结点，然后更新头结点的值，最终算法如下：

```cpp
template<class T>
void chain<T>::insert(int theIndex, const T& theElement)
{
    if (theIndex == 0)//插入到头结点
        firstNode = new chainNode<T>(theElement, firstNode);
    else
    {
        chainNode<T>* p = firstNode;
        for (int i = 0; i < theIndex - 1; i++)
            p = p->next;//找到前一个结点

        p->next = new chainNode<T>(theElement, p->next);
    }
    listSize++;
}
```

5. 删除操作： 删除操作与插入操作类似，都是需要找到被删除结点的前一个结点，然后更新 next 值同样的，如果它没有前一个结点（firstNode），那么需要更改 fisrtNode 的值，最终算法如下：

```cpp
template<class T>
void chain<T>::erase(T val)
{
    iterator iter = begin();
    iterator pre( theNode: nullptr);
    while (iter != end()&& *iter != val  )
    {
        pre = iter;
        iter++;
    }
    if (iter == end())//如果没找到
    {
        cout << -1 << endl;
    }
    else
    {
        //找到
        if (iter == begin())
        {
            firstNode = firstNode->next;
            listSize--;
        }
```

```
        else
        {
            pre->next = iter->next;
            delete iter.ptr();
            listSize--;
        }
    }
}
```

6. 对于 rerverse 操作：我们需要用三个指针记录，本算法使用的是迭代器来进行操作，p1 是 p2 的上一个结点，p3 是 p2 的下一个结点，每次都让 p2->next 指向 p1 指向的 chainNode，然后 p2 与 p1 均往后移动，因为它们原来的 next 已经改了，所以用 p3 记录，再往后移动。最终代码如下：

```
template <class T>
void chain<T>::reverse()
{
    //构造函数规定了至少要有一个结点
    chain<T>::iterator p1(firstNode);//p1为p2的上一个结点
    chain<T>::iterator p2(p1->next);
    p1->next = nullptr;//p1是firstNode，故reverse之后一定是最后一个结点
    while (p2 != nullptr)
    {
        chain<T>::iterator p3(p2->next);
        p2->next = p1.ptr();
        p1 = p2;//记录一下p2
        firstNode = p2.ptr();//每次都更新一下firstNode
        p2 = p3;//移动p2
    }
}
```

7. 对于查询操作：使用迭代器遍历搜寻，当没有到 end()并且没有找到就++，最后判断一下是否找到即可，代码如下：

```cpp
template<class T>
void chain<T>::indexOf(const T& val) const
{
    int pos = 0;//记录索引
    iterator iter(firstNode);
    while (iter != end() && *iter != val)//如果没到end并且没找到就++
    {
        iter++;
        pos++;
    }
    if (iter == end())//没找到
    {
        cout << -1 << endl;
    }
    else//找到了
    {
        cout << pos << endl;
    }
}
```

8. 输出异或和：思路与查询类似，也是遍历搜寻，最后记录一下即可

```cpp
template<class T>
void chain<T>::output() const
{
    int pos = 0;
    int ans = 0;
    for (iterator iter = begin(); iter != end(); iter++, pos++)//没到end()就++
    {
        ans += *iter ^ pos;
    }
    cout << ans << endl;
}
```

9. 对于 B 题使用的数据结构，与 A 题大体类似，都是链表与迭代器。唯一有变动的是排序算法那里，需要增加 merge 函数与 sort 函数，我们使用的 sort 函数是基数排序

10. Merge 函数其实就是归并排序中对左右两个区间整理有序之后，放回去的过程，时间复杂度 O(n)，只要没到 end，就可以继续比较，代码如下：

```cpp
template<class T>
void chain<T>::merge(chain<T>& a, chain<T>& b)
{

    iterator a_iter(a.firstNode);//a的迭代器
    iterator b_iter(b.firstNode);//b的迭代器
    iterator end( theNode: nullptr);

    while (a_iter != end && b_iter != end)//只要没到最后，就可以继续
    {
        push_back( &: *a_iter <= *b_iter ? *a_iter++ : *b_iter++);
    }
    while (a_iter != end)//把a剩下的元素都Push_back进去
    {
        push_back( &: *a_iter++);
    }
    while (b_iter != end)
    {
        push_back( &: *b_iter++);
    }

    listSize = a.listSize + b.listSize;//更新私有变量的值
}
```

11. 对于基数排序，由于它的精髓就是稳定排序，因此我们增加了 push_back 函数，使得对于当前这一轮中的所有箱子里，它们的相对顺序是不变的（稳定），对于每一轮操作，我们都需要提取出有效的数字，放到相应的箱子里去。然后收集的时候是从前往后收集，保持稳定。代码如下：

```cpp
template<class T>
void chain<T>::radixSort(int r, int d)//r=range, d=the number of loop,
{
    //指针数组
    //从后往前搜，保证了稳定性
    chain<T>* bin = new chain<T>[r];
    for (int i = 0; i < d; i++)//循环次数
    {
        int l = listSize;//防止listSize被破坏
        for (int j = 0; j < l; j++)
        {
            //将每一位都变成箱子排序
            int index;//放到哪个箱子里边
            index = (this->get(0) / (int (pow(r, i)))) % r;
            bin[index].push_back( this->get(0));
            this->erase(0);
        }
        //从前往后收集
        for (int j = 1; j < r; j++)
        {
            while (!bin[j].empty())
            {
                this->push_back( bin[j].get(0));
                bin[j].erase(0);
            }
        }
    }
    delete[]bin;
    bin = nullptr;
}
```

3. 测试结果（测试输入，测试输出）

A 题输入：

Input

```
10 10
6863 35084 11427 53377 34937 14116 5000 49692 70281 73704
4 6863
1 2 44199
5
4 21466
1 6 11483
5
4 34937
5
4 6863
1 10 18635
```

输出：

```
5
4 34937
5
4 6863
1 10 186350
398665
-1
410141
5
410141
0
```

B 题输出

输入

```
3 0
3 1 2
```

输出：

```
3 0
3 1 2
5
0
5

进程已结束，退出代码为 0
```

1	♡ 202000130198 隋春雨	2	200	100 632	100 658

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

(1) 测试数据的时候发现死循环了怎么办？
解决：经过 debug 发现，是因为短路问题

```
while (*iter != val && iter != end() )
{
    pre = iter;
    iter++;
}
```

这么写会造成死循环，因为如果这个时候 iter 的值是 nullptr，而*nullptr 是没有定义的，正确的应该是先判断 iter 是否为 end()，即为：

```
while (iter != end()&& *iter != val  )
{
    pre = iter;
    iter++;
}
```

(2) 对于迭代器，我们是应该将其单独作为一个类合适还是放到了 chain 类里边合适？
解决：应该放到类里，如果不放到类里，那么我们在使用的时候就会很麻烦，对于类型的传递就要传两次。如果放到了 chain 类里，那么我们使用的时候会很方便，同时对于一些 chain 类的操作，也可以借助 iterator 来实现。

(3) 对于迭代器，我们是将其作为一个成员放到 chain 类私有成员或者共有成员里好还是作为定义放到 public 里好？

解决：应该作为定义放到 public 里好，①首先，如果作为一个私有成员，那么用户在使用的时候就无法使用了（除非调用 public 函数），②其次，如果作为一个成员放到 Public 里，那么我们用户自行定义的时候，就必须使用这个成员，会非常令人疑惑，使用成本很大，如果作为一个定义放到了 chain 类里就不会有什么问题。

(4) 对于边界条件的判定，我们在插入与删除函数的时候，都要找到上一个结点的位置，而如果被插入和删除的结点如果是 firstNode，那么它就没有上一个结点，这个时候需要特判一下

(5) 在 reverse 函数中，第一次调用的时候跟预期结果不一样怎么办？
解决：debug 发现，是因为原来的 next 值被更改了，而使用的时候没注意，就发生了错误。以后在写程序的时候，一定先想好逻辑在开始。同时对于每一次的更新，都需更新一下 firstNode

```
void chain<T>::reverse()
{
    //构造函数规定了至少要有一个结点
    chain<T>::iterator p1(firstNode);//p1为p2的上一个结点
    chain<T>::iterator p2(p1->next);
    p1->next = nullptr;//p1是firstNode，故reverse之后一定是最后一个结点
    while (p2 != nullptr)
    {
        chain<T>::iterator p3(p2->next);
        p2->next = p1.ptr();
        p1 = p2;//记录一下p2
        firstNode = p2.ptr();//每次都更新一下firstNode
        p2 = p3;//移动p2
    }
}
```

(6) 对于迭代器，我们考虑到它是一个智能指针，需要重载++ ，*，->等运算符，我们平时用*iter 的意思是取出它所指的元素的值，iter->意思是取出它所指的结点，因此构造如下：**值得注意的是**，书上的重载->可能是错的，它返回的是&node->element，这个在 clion 上只能取出来 element，对于 next 指针就不行。正确的写法应该是：

```
chainNode<T>* operator->() const { return node; }
```

(7) 对于基数排序，需要在末尾插入，如果是一个一个 Insert 会很慢，因此我们更新一个变量 lastNode，可以帮助我们更快的插入。

(8) 自己写的时候测的样例都是对的，交到 oj 平台上就 RE 了，怎么办？
解决：RE 常见情况的是数组下标越界，但是经过自己 debug 发现，实际情况是 switch case 条件没有 break 语句，才 RE，在平时，能用 switch case 尽量用 switch case 而不是 If else ，因为 switch case 执行的次数少。

(9) 在测试样例的时候发现自己的输出值跟预期不同，怎么办？

解决：经过 debug 发现，在删除操作的时候，对于数组的 size 变量没有更新，从而导致错误。以后在写函数的时候，一定需要注意的一点就是调用更新私有变量成员。

(10)一个一个写操作很麻烦怎么办？
解决：运用面向对象的思想，将函数封装为类内函数，以后只需要调用类内函数即可进行操作。

5. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）

① A 题

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


template <class T>
struct chainNode
{
    T element;
    chainNode<T>* next;//指向下一个结点的指针

    chainNode() {}
    chainNode(const T& element)
    {
        this->element = element;
    }
    chainNode(const T& element, chainNode<T>* next)
    {
        this->element = element;
        this->next = next;
    }
};


template<class T>
class chain
{

public:
    // constructor, copy constructor and destructor
    chain(int initialCapacity = 10);
    chain(const chain<T>&);
    ~chain();


    void indexOf(const T& val) const;//查询索引
```

```
37.     void erase(T val);//删除
38.     void insert(int theIndex, const T& theElement);//插入
39.     void output() const;//输出元素异或和
40.     void reverse();//反转操作
41.
42.     class iterator;//迭代器
43.     iterator begin() const { return iterator(firstNode); }
44.     iterator end() const { return iterator(nullptr); }
45.
46.
47.
48.     class iterator
49.     {
50.     public:
51.         iterator(chainNode<T>* theNode = nullptr)
52.         {
53.             node = theNode;
54.         }
55.
56.         T& operator*() const { return node->element; }//重载*
57.         chainNode<T>* operator->() const { return node; }//重载->
58.
59.         bool operator!=(const iterator right) const
60.         {
61.             return node != right.node;
62.         }
63.         bool operator==(const iterator right) const
64.         {
65.             return node == right.node;
66.         }
67.         iterator& operator++()  //前++
68.         {
69.             node = node->next; return *this;
70.         }
71.         iterator operator++(int) // 后++
72.         {
73.             iterator old = *this;
74.             node = node->next;
75.             return old;
76.         }
77.
78.
79.         iterator operator =(const chainNode<T>& c_ptr)
80.         {
81.             node = c_ptr;
82.             return iterator(node);
```

```
        }

        chainNode<T>* ptr()
        {
            return node;//返回指针
        }

    protected:
        chainNode<T>* node;
    };

protected:

    chainNode<T>* firstNode;
    int listSize;
};


template<class T>
void chain<T>::indexOf(const T& val) const
{
    int pos = 0;//记录索引
    iterator iter(firstNode);
    while (iter != end() && *iter != val)//如果没到 end 并且没找到就++
    {
        iter++;
        pos++;
    }
    if (iter == end())//没找到
    {
        cout << -1 << endl;
    }
    else//找到了
    {
        cout << pos << endl;
    }
}

template<class T>
void chain<T>::erase(T val)
{
    iterator iter = begin();
    iterator pre(nullptr);//前一个结点
    while (iter != end()&& *iter != val  )
    {
```

```
129.        pre = iter;
130.        iter++;
131.    }
132.    if (iter == end())//如果没找到
133.    {
134.        cout << -1 << endl;
135.    }
136.    else
137.    {
138.        //找到
139.        if (iter == begin())
140.        {
141.            firstNode = firstNode->next;
142.            listSize--;
143.        }
144.        else
145.        {
146.            pre->next = iter->next;
147.            delete iter.ptr();
148.            listSize--;
149.        }
150.    }
151. }
152.
153. template<class T>
154. void chain<T>::insert(int theIndex, const T& theElement)
155. {
156.    if (theIndex == 0)//插入到头结点
157.        firstNode = new chainNode<T>(theElement, firstNode);
158.    else
159.    {
160.        chainNode<T>* p = firstNode;
161.        for (int i = 0; i < theIndex - 1; i++)
162.            p = p->next;//找到前一个结点
163.
164.        p->next = new chainNode<T>(theElement, p->next);
165.    }
166.    listSize++;
167. }
168.
169. template<class T>
170. void chain<T>::output() const
171. {
172.    int pos = 0;
173.    int ans = 0;
174.    for (iterator iter = begin(); iter != end(); iter++, pos++)//没到 end()就++
```

```
    {
        ans += *iter ^ pos;
    }
    cout << ans << endl;
}



template <class T>
void chain<T>::reverse()
{
    //构造函数规定了至少要有一个结点
    chain<T>::iterator p1(firstNode);//p1 为 p2 的上一个结点
    chain<T>::iterator p2(p1->next);
    p1->next = nullptr;//p1 是 firstNode，故 reverse 之后一定是最后一个结点
    while (p2 != nullptr)
    {
        chain<T>::iterator p3(p2->next);
        p2->next = p1.ptr();
        p1 = p2;//记录一下 p2
        firstNode = p2.ptr();//每次都更新一下 firstNode
        p2 = p3;//移动 p2
    }
}

template<class T>
chain<T>::chain(int initialCapacity)
{
    firstNode = nullptr;
    listSize = 0;
}


template<class T>
chain<T>::~chain()
{// Chain destructor. Delete all nodes in chain.
    chainNode<T>* nextNode;
    while (firstNode != NULL)
    {// delete firstNode
        nextNode = firstNode->next;
        delete firstNode;
        firstNode = nextNode;
    }
}

```

```
221.
222.
223. int main()
224. {
225.     int n, q;
226.     cin >> n >> q;
227.     chain<int>my_chain;
228.     for (int i = 0; i < n; i++)
229.     {
230.         int val;
231.         cin >> val;
232.         my_chain.insert(i, val);//插入到相应的位置
233.     }
234.     int flag;
235.     int idx, val;
236.     for (int i = 0; i < q; i++)
237.     {
238.         cin >> flag;//标记
239.         switch (flag)
240.         {
241.             case 1:
242.                 cin >> idx >> val;
243.                 my_chain.insert(idx, val);
244.                 break;
245.             case 2:
246.                 cin >> val;
247.                 my_chain.erase(val);
248.                 break;
249.             case 3:
250.                 my_chain.reverse();
251.                 break;
252.             case 4:
253.                 cin >> val;
254.                 my_chain.indexOf(val);
255.                 break;
256.             case 5:
257.                 my_chain.output();
258.                 break;
259.         }
260.     }
261.     return 0;
262. }
```

(2)

② B 题

```
#include <iostream>
#include <cmath>
#include <ctime>

using namespace std;

template <class T>
struct chainNode
{
   T element;
   chainNode<T>* next;

   chainNode() {}

   chainNode(const T& element, chainNode<T>* next)
   {
      this->element = element;
      this->next = next;
   }
};


template<class T>
class chain
{

public:
   // constructor, copy constructor and destructor
   chain(int initialCapacity = 10);
   chain(const chain<T>&);
   ~chain();

   // ADT methods
   bool empty() const { return listSize == 0; }
   int size() const { return listSize; }

   void insert(int theIndex, const T& theElement);
   void output() const;

   void push_back(T& val);
   T& get(int theIndex) const;

   class iterator;
   iterator begin() const { return iterator(firstNode); }
   iterator end() const { return iterator(NULL); }
   //作业
```

```
47.     void erase(int theIndex);
48.     void merge(chain<T>& c1, chain<T>& c2);
49.
50.     void insertSort();
51.     void test()
52.     {
53.        iterator iter(firstNode);
54.        while (iter != end())
55.        {
56.           cout << *iter++<<" ";
57.        }
58.        cout << endl;
59.     }
60.     void radixSort(int r, int d);
61.     void sort_by_radix10();
62.
63.     class iterator
64.     {
65.     public:
66.
67.        iterator(chainNode<T>* theNode = nullptr)
68.        {
69.           node = theNode;
70.        }
71.
72.        iterator(const T& val, chainNode<T>* next)
73.        {
74.           node = new chainNode<T>(val, next);
75.
76.        }
77.        T& operator*() const { return node->element; }
78.        chainNode<T>* operator->() const { return node; }
79.
80.        iterator& operator++()   // preincrement
81.        {
82.           node = node->next; return *this;
83.        }
84.        iterator operator++(int) // postincrement
85.        {
86.           iterator old = *this;
87.           node = node->next;
88.           return old;
89.        }
90.
91.        // equality testing
92.        bool operator!=(const iterator right) const
```

```
93.         {
94.            return node != right.node;
95.         }
96.         bool operator==(const iterator right) const
97.         {
98.            return node == right.node;
99.         }
100.
101.        iterator operator =(const chainNode<T>& c_ptr)
102.        {
103.           node = c_ptr;
104.           return iterator(node);
105.        }
106.
107.        chainNode<T>* ptr()
108.        {
109.           return node;
110.        }
111.
112.     protected:
113.        chainNode<T>* node;
114.     };  // end of iterator class
115.
116. protected:
117.
118.     chainNode<T>* firstNode;  // pointer to first node in chain
119.     chainNode<T>* lastNode;
120.     int listSize;            // number of elements in list
121. };
122.
123. template<class T>
124. chain<T>::chain(int initialCapacity)
125. {// Constructor.
126.
127.     firstNode = nullptr;
128.     lastNode = nullptr;
129.     listSize = 0;
130. }
131.
132.
133. template<class T>
134. chain<T>::~chain()
135. {// Chain destructor. Delete all nodes in chain.
136.     chainNode<T>* nextNode;
137.     while (firstNode != NULL)
138.     {// delete firstNode
```

```
139.       nextNode = firstNode->next;
140.       delete firstNode;
141.       firstNode = nextNode;
142.    }
143. }
144.
145. template<class T>
146. void chain<T>::insert(int theIndex, const T& theElement)
147. {// Insert theElement so that its index is theIndex.
148.
149.
150.    if (theIndex == 0)
151.    {
152.       firstNode = new chainNode<T>(theElement, firstNode);
153.       lastNode = firstNode;
154.    }
155.    else
156.    {  // find predecessor of new element
157.       chainNode<T>* p = firstNode;
158.       for (int i = 0; i < theIndex - 1; i++)
159.          p = p->next;
160.
161.       p->next = new chainNode<T>(theElement, p->next);
162.       if (theIndex == listSize)
163.       {
164.          lastNode = p->next;
165.       }
166.    }
167.    listSize++;
168. }
169.
170.
171. template<class T>
172. void chain<T>::output() const
173. {
174.
175.    int ans = 0;
176.    int index = 0;
177.    for (iterator iter = begin(); iter != end(); iter++, index++)
178.    {
179.       ans += index ^ *iter;
180.    }
181.    cout << ans << endl;
182. }
183.
184.
```

```
185.
186. template<class T>
187. void chain<T>::merge(chain<T>& a, chain<T>& b)
188. {
189.
190.    iterator a_iter(a.firstNode);//a 的迭代器
191.    iterator b_iter(b.firstNode);//b 的迭代器
192.    iterator end(nullptr);
193.
194.    while (a_iter != end && b_iter != end)//只要没到最后，就可以继续
195.    {
196.       push_back(*a_iter <= *b_iter ? *a_iter++ : *b_iter++);
197.    }
198.    while (a_iter != end)//把 a 剩下的元素都 Push_back 进去
199.    {
200.       push_back(*a_iter++);
201.    }
202.    while (b_iter != end)
203.    {
204.       push_back(*b_iter++);
205.    }
206.
207.    listSize = a.listSize + b.listSize;//更新私有变量的值
208. }
209.
210. template<class T>
211. void chain<T>::erase(int theIndex)
212. {
213.    chainNode<T>* deleteNode;
214.    if (theIndex == 0)
215.    {
216.       deleteNode = firstNode;
217.       firstNode = firstNode->next;
218.
219.    }
220.    else
221.    {
222.       chainNode<T>* p = firstNode;
223.       for (int i = 0; i < theIndex - 1; i++)
224.          p = p->next;
225.
226.       deleteNode = p->next;
227.       p->next = p->next->next; // remove deleteNode from chain
228.    }
229.    if (theIndex == listSize)
230.    {
```

```
231.         lastNode = nullptr;
232.     }
233.     listSize--;
234.     delete deleteNode;
235. }
236. 
237. template<class T>
238. T& chain<T>::get(int theIndex) const
239. {
240. 
241.     chainNode<T>* currentNode = firstNode;
242.     for (int i = 0; i < theIndex; i++)
243.         currentNode = currentNode->next;
244. 
245.     return currentNode->element;
246. }
247. 
248. 
249. template<class T>
250. void chain<T>::insertSort()
251. {
252.     if (listSize == 0 || listSize == 1)
253.     {
254.         return;
255.     }
256. 
257.     iterator pre(firstNode);
258.     iterator iter(firstNode->next);
259.     while (iter != nullptr)
260.     {
261.         iterator pos(firstNode);
262. 
263. 
264.         if (*iter <= *pos)
265.         {
266.             pre->next = iter->next;
267.             iter->next = firstNode;
268.             firstNode = iter.ptr();
269.             iter = pre->next;//因为 iter 已经被更新了，所以我们要借助 pre 指针进行更新
270.         }
271.         else
272.         {
273.             while (pos->next->element < *iter)//iter 的存在确保了不会越界，类似于放置一个哨兵
274.             {
275.                 pos++;
276.             }
```

```
            if (pos == pre)
            {
                pre++;
                iter++;
                continue;
            }
            else
            {
                pre->next = iter->next;
                iter->next = pos->next;
                pos->next = iter.ptr();
                iter = pre->next;//因为 iter 已经指向了头节点，所以我们要借助 pre 指针进行更新
            }
        }
    }
}

template<class T>
void chain<T>::radixSort(int r, int d)//r=range, d=the number of loop,
{
    //指针数组
    //从后往前搜，保证了稳定性
    chain<T>* bin = new chain<T>[r];
    for (int i = 0; i < d; i++)//循环次数
    {
        int l = listSize;//防止 listSize 被破坏
        for (int j = 0; j < l; j++)
        {
            //将每一位都变成箱子排序
            int index;//放到哪个箱子里边
            index = (this->get(0) / (int (pow(r, i)))) % r;
            bin[index].push_back( this->get(0));
            this->erase(0);
        }
        //从前往后收集
        for (int j = 1; j < r; j++)
        {
            while (!bin[j].empty())
            {
                this->push_back( bin[j].get(0));
                bin[j].erase(0);
            }
        }
    }
    delete[]bin;
    bin = nullptr;
```

```
}


template<class T>
void chain<T>::sort_by_radix10()
{
    if (listSize == 0 || listSize == 1)
    {
        return;
    }
    iterator iter = begin();
    int _max = -1;
    while (iter != end())
    {
        _max = max(*iter, _max);
        iter++;
    }
    int loop_num = 0;
    while (_max)
    {
        loop_num++;
        _max /= 10;
    }
    radixSort(10, loop_num);
}

template<class T>
void chain<T>::push_back(T& val)
{
    if (listSize == 0)
    {
        //iterator temp(val, nullptr);
        chainNode<T>* temp = new chainNode<T>(val, nullptr);
        lastNode = firstNode = temp;
    }
    else
    {
        chainNode<T>* temp = new chainNode<T>(val, nullptr);
        lastNode->next = temp;
        lastNode = temp;
    }
    listSize++;
}

int main()
{
```

```
369.
370.    int n, m;
371.    cin >> n >> m;
372.
373.
374.    chain<int>my_chain1;
375.    chain<int>my_chain2;
376.    chain<int>result;//结果链表
377.    int val;
378.
379.    for (int i = 0; i < n; i++)
380.    {
381.        cin >> val;
382.        my_chain1.insert(0, val);
383.    }
384.
385.
386.    for (int i = 0; i < m; i++)
387.    {
388.        cin >> val;
389.        my_chain2.insert(0, val);
390.    }
391.
392.
393.    my_chain1.sort_by_radix10();//基数排序
394.    my_chain2.sort_by_radix10();//基数排序
395.
396.    result.merge(my_chain1, my_chain2);//合并操作
397.
398.    my_chain1.output();//输出异或和
399.    my_chain2.output();
400.    result.output();
401.
402.
403.    return 0;
404. }
```