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：202300130183	姓名： 宋浩宇	班级：23 级人工智能班
实验题目：2024 数据结构--数据/智能 实验 4 链式描述线性表		
实验学时：2	实验日期：2024/10/9	

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。

软件开发工具：

1. visual studio code 2022（使用 C/C++、C/C++ Extension Pack、C/C++ Themes 插件）
2. mingw64 工具包

1. 实验内容

完成 2024 数据结构--数据/智能 实验 4 链式描述线性表 A ： 链表实现，B ： 链表合并两道题目。

（附加要求：

1. 要求封装链表类，链表迭代器类；
2. 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；
3. 不得使用与链表实现相关的 STL。）

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）

本实验全程使用了双向链表数据结构，另外还使用了快速排序算法。

首先是第一题，链表实现，就本题的题目来看并不难，主要在于链表增删查改实现的思路。链表的插入操作实现为：从链表的头开始遍历到要插入的索引处，然后将该索引处前一个元素存储的下一个元素的地址改为插入元素的地址，插入元素的上一个元素的地址改为该索引处的前一个元素的地址，插入元素的下一个元素的地址改为该索引处原本元素的地址，该索引处原本元素的前一个元素的地址改为插入元素的地址。删除操作，首先遍历到需要删除的元素的索引位置，然后将该索引的前一个元素的下一个元素的地址改为要删除的元素的下一个元素的地址，再将要删除的元素的下一个元素的前一个元素的地址改为该索引处前一个元素的地址。查找操作，从链表头开始遍历，每次更新节点指针使其指向下一个节点，比较该节点处存储的数据，并记录该节点的索引，找到需要查找的元素后返回该元素所在节点的索引，如果遍历到链表的最后一个元素还没有找到需要的元素，则返回-1。更改操作，基于查找操作，先查找到要修改的元素的索引，在查找时可以选择返回对应元素的指针，也可以选择返回对应元素的引用，在对返回值进行修改。以上为增删查改操作的实现思路，考虑到题目要求，再基于基础功能添加新的功能，首先是原地反转，实现思路为：从链表的头节点开始遍历，交换每个节点的前一个节点的地址和后一个节点的地址，再将链表数据结构的头节点的地址和尾节点的地址交换。另外我们还需要书写链表的迭代器，迭代器的设计主要为，一个指针类型的数据成员，为迭代器指向的元素，索引值，为该迭代器所指向的元素在对应容器中的索引，自增自减大于等于小于的运算符重载，用于地址的改变。然后我们可以通过迭代器自增的方式来完成对于链表数据结构的遍历（这个自增的实现为：将指针类型的数据成员改为这个指针所指向的节点的下一个节点的地址）。

然后是第二题，首先我们可以套用前一个问题所写的链表类和迭代器类，然后，对这个链表类进行拓展，首先是进行两个链表类之间加法的运算符重载，让两个链表的链接由加法的形式实现，实际实现起来也很简单，只要把两个链表进行一次深复制，再将前一个链表的尾节

点的下一个节点的地址改为后一个链表的头节点的地址，再依次更新地址即可。
本题还需要完成链表的排序，并要求使用迭代器来进行排序，经过实验，使用冒泡排序会出现超时的情况，因此我们以迭代器的方式来实现快速排序，快速排序的描述与本节无关，故省略，详细内容请见附件中的 quick_sort()函数的代码。通过迭代器的方式，我们可以用通用的方法操作线性容器，即可以将所有的线性容器都视为数组，只要做好运算符的重载即可。

3. 测试结果（测试输入，测试输出）

第一题测试输入：

```
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：

```
0
398665
-1
410141
5
410141
0
```

第二题测试输入：

```
3 0
3 1 2
```

输出：

```
5
0
5
```

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

从结果来看，我们成功解决了这两道算法题，且自行书写出了链表的类模板和迭代器的类模板，并且通过迭代器的方式将排序算法封装成了线性容器中的通用方法。成功掌握了线性表结构、链式描述方法（链式存储结构）、链表的实现、链表迭代器的实现与应用。
也进一步深刻理解了面向对象编程的多态的特点。

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）

```cpp
/*2024 数据结构--数据智能 实验 4 链式描述线性表 A 链表实现.cpp*/
#include<iostream>
using namespace std;

template<class T>
```

```
7.    class list_node
8.    {
9.    private:
10.       T data;
11.
12.   public:
13.       list_node<T>* front;
14.       list_node<T>* next;
15.       bool operator==(const list_node<T>& list_node) const { return this->data == l
    ist_node.data; };
16.       T& getData() { return data; };
17.       list_node<T>* getFront() { return front; };
18.       list_node<T>* getNext() { return next; };
19.       list_node(){};
20.       list_node(const T& data, list_node<T>* front = nullptr, list_node<T>* next =
    nullptr);
21.       list_node(list_node<T>* const list_node) { this->data = list_node->data;this-
    >front = list_node->front;this->next = list_node->next; };
22.   };
23.
24.   template<class T>
25.   list_node<T>::list_node(const T& data,list_node<T>* front,list_node<T>* next)
26.   :data(data),front(front),next(next)
27.   {
28.
29.   }
30.
31.
32.
33.   template<class T>
34.   class chainlist
35.   {
36.   private:
37.       list_node<T>* head;
38.       list_node<T>* back;
39.       unsigned int size;
40.   public:
41.       void push_back(list_node<T>* operated_list_node);
42.       void push_back(const T& data);
43.       void push_front(list_node<T>* operated_list_node);
44.       void push_front(const T& data);
45.       void pop_back();
46.       void pop_front();
47.       void insert(T data, int index);
48.       void insert(list_node<T>* operated_list_node, int index);
49.       void erase(const unsigned int index);
```

```
50.       unsigned int getSize() { return size; };
51.       unsigned int find_first(list_node<T>* operated_list_node);
52.       unsigned int find_first(const T& data);
53.       void inverse();
54.       chainlist();
55.       chainlist(const chainlist<T>& chainlist);
56.       void print();
57.       list_node<T>* getHead() { return head; };
58.       list_node<T>* getBack() { return back; };
59.
60.   };
61.
62.   template<class T>
63.   void chainlist<T>::push_back(list_node<T>* operated_list_node)
64.   {
65.       list_node<T>* new_list_node = new list_node<T>(operated_list_node);
66.
67.
68.       if (size == 0)
69.       {
70.           this->head = new_list_node;
71.           this->back = new_list_node;
72.           size++;
73.           return;
74.       }
75.
76.       this->back->next = new_list_node;
77.       new_list_node->front = this->back;
78.       new_list_node->next = nullptr;
79.       this->back = new_list_node;
80.
81.       size++;
82.   }
83.
84.   template<class T>
85.   void chainlist<T>::push_back(const T& data)
86.   {
87.       T new_data = data;
88.       list_node<T>* new_list_node = new list_node<T>(new_data);
89.       this->push_back(new_list_node);
90.       delete new_list_node;
91.   }
92.
93.   template<class T>
94.   void chainlist<T>::push_front(list_node<T>* operated_list_node)
95.   {
```

```
96.       list_node<T>* new_list_node = new list_node<T>(operated_list_node);
97.
98.       if (size == 0)
99.       {
100.          this->back = new_list_node;
101.          this->head = new_list_node;
102.          size++;
103.          return;
104.      }
105.      new_list_node->next = this->head;
106.      new_list_node->front = nullptr;
107.      this->head->front = new_list_node;
108.      this->head = new_list_node;
109.
110.      size++;
111.  }
112.
113.  template<class T>
114.  void chainlist<T>::push_front(const T& data)
115.  {
116.      T new_data = data;
117.      list_node<T>* new_list_node = new list_node<T>(new_data);
118.      this->push_front(new_list_node);
119.      delete new_list_node;
120.  }
121.
122.  template<class T>
123.  void chainlist<T>::pop_back()
124.  {
125.      if (this->back == nullptr) return;
126.      list_node<T>* temp = this->back;
127.      this->back = this->back->front;
128.      delete temp;
129.      if (size == 1)
130.      {
131.          this->head = nullptr;
132.          this->back = nullptr;
133.      }
134.
135.      size--;
136.  }
137.
138.  template<class T>
139.  void chainlist<T>::pop_front()
140.  {
141.      if (this->head == nullptr) return;
```

```
142.     list_node<T>* temp = this->head;
143.     this->head = this->head->next;
144.     delete temp;
145.     if (size == 1)
146.     {
147.         this->head = nullptr;
148.         this->back = nullptr;
149.     }
150.     size--;
151. }
152. 
153. 
154. template<class T>
155. void chainlist<T>::insert(list_node<T>* operated_list_node, int index)
156. {
157.     if (index == 0)
158.     {
159.         push_front(operated_list_node);
160.         return;
161.     }
162.     if (index == size)
163.     {
164.         push_back(operated_list_node);
165.         return;
166.     }
167. 
168.     list_node<T>* temp = this->head;
169.     for (int i = 0; i < index - 1; i++)
170.     {
171.         temp = temp->next;
172.     }
173.     list_node<T>* new_list_node = new list_node<T>(operated_list_node);
174.     new_list_node->next = temp->next;
175.     temp->next->front = new_list_node;
176.     temp->next = new_list_node;
177.     new_list_node->front = temp;
178.     size++;
179. }
180. 
181. template<class T>
182. void chainlist<T>::insert(T data, int index)
183. {
184.     list_node<T>* new_list_node = new list_node<T>(data);
185.     this->insert(new_list_node, index);
186.     delete new_list_node;
187. 
```

```
188.  }
189.
190.  template<class T>
191.  void chainlist<T>::erase(const unsigned int index)
192.  {
193.      if (index == 0)
194.      {
195.          pop_front();
196.          return;
197.      }
198.      if (index == size)
199.      {
200.          pop_back();
201.          return;
202.      }
203.
204.      list_node<T>* temp = this->head;
205.      for (int i = 0; i < index - 1; i++)
206.      {
207.          temp = temp->next;
208.      }
209.      list_node<T>* temp2 = temp->next;
210.      temp->next = temp2->next;
211.      temp->next->front = temp;
212.      delete temp2;
213.      size--;
214.  }
215.
216.  template<class T>
217.  unsigned int chainlist<T>::find_first(list_node<T>* operated_list_node)
218.  {
219.      list_node<T>* temp = this->head;
220.      unsigned int index = 0;
221.      while (temp != nullptr)
222.      {
223.          if ((*temp) == (*operated_list_node)) return index;
224.          temp = temp->next;
225.          index++;
226.      }
227.      return -1;
228.  }
229.
230.  template<class T>
231.  unsigned int chainlist<T>::find_first(const T& data)
232.  {
233.      list_node<T>* temp = this->head;
```

```
234.     unsigned int index = 0;
235.     while (temp != nullptr)
236.     {
237.         if (temp->getData() == data) return index;
238.         temp = temp->next;
239.         index++;
240.     }
241.     return -1;
242. }
243. 
244. template<class T>
245. void chainlist<T>::inverse()
246. {
247.     list_node<T>* temp1 = this->head;
248.     list_node<T>* temp2 = nullptr;
249.     list_node<T>* temp3 = nullptr;
250.     for (int i = 0; i < size; i++)
251.     {
252.         temp2 = temp1->next;
253.         temp3 = temp1->front;
254.         temp1->front = temp1->next;
255.         temp1->next = temp3;
256.         temp1 = temp2;
257.     }
258.     temp1 = this->head;
259.     this->head = this->back;
260.     this->back = temp1;
261. }
262. 
263. template<class T>
264. chainlist<T>::chainlist()
265. {
266.     this->head = nullptr;
267.     this->back = nullptr;
268.     this->size = 0;
269. }
270. 
271. template<class T>
272. chainlist<T>::chainlist(const chainlist<T>& chainlist)
273. {
274.     this->head = nullptr;
275.     this->back = nullptr;
276.     this->size = 0;
277.     list_node<T>* temp = chainlist.head;
278.     while (temp != nullptr)
279.     {
```

```
            push_back(temp);
            temp = temp->next;
        }
    }

    template<class T>
    void chainlist<T>::print()
    {
        list_node<T>* temp = this->head;
        while (temp != nullptr)
        {
            cout << temp->getData() << " ";
            temp = temp->next;
        }
        cout << endl;
    }


    template<class T>
    class my_iterator
    {
    private:
        list_node<T>* current;
        unsigned int index;
    public:
        my_iterator& operator++() { this->current = this->current->next; this->index++; return *this; };
        my_iterator& operator++(int) { my_iterator temp = *this; this->current = this->current->next;this->index++; return temp; };
        my_iterator(list_node<T>* current) { this->current = current; this->index = 0; };
        list_node<T>* getCurrent() { return this->current; };
        unsigned int getIndex() { return this->index; };
    };


    class Solution
    {
    public:
        void solote();
    };

    void Solution::solote()
    {
        int n, m;
        cin >> n >> m;
```

```
    chainlist<int> list;
    for (int i = 0; i < n; i++)
    {
        int x;
        cin >> x;
        list.push_back(x);
    }
    for (int i = 0; i < m; i++)
    {
        int operation;
        cin >> operation;
        if (operation == 1)
        {
            int index, value;
            cin >> index >> value;
            list.insert(value, index);
        }
        else if (operation == 2)
        {
            int value;
            cin >> value;
            if (list.find_first(value) != -1)
            {
                list.erase(list.find_first(value));
            }
            else
            {
                cout << -1 << endl;
            }

        }
        else if (operation == 3)
        {
            list.inverse();
        }
        else if (operation == 4)
        {
            int value;
            cin >> value;
            cout << (int)list.find_first(value)<<endl;
        }
        else if (operation == 5)
        {
            my_iterator<int> it(list.getHead());
            int ans = 0;
            for (int j = 0;j < list.getSize();j++)
```

```
            {
                ans += it.getCurrent()->getData() ^ it.getIndex();
                it++;
            }
            cout << ans << endl;
        }

    }
}


int main()
{
    Solution solution;
    solution.solote();
    // system("pause");
    return 0;
}
```

```
/*2024 数据结构--数据智能 实验 4 链式描述线性表 B 链表合并.cpp*/
#include<iostream>
using namespace std;


template<class T>
void my_swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}

template<class T>
class list_node
{
private:
    T data;

public:
    list_node<T>* front;
    list_node<T>* next;
    bool operator==(const list_node<T>& list_node) const { return this->data == list_node.data; };
    T& getData() { return data; };
    list_node<T>* getFront() { return front; };
    list_node<T>* getNext() { return next; };
```

```
27.       list_node(){};
28.       list_node(const T& data, list_node<T>* front = nullptr, list_node<T>* next =
    nullptr);
29.       list_node(list_node<T>* const list_node) { this->data = list_node->data;this-
    >front = list_node->front;this->next = list_node->next; };
30.   };
31.
32.   template<class T>
33.   list_node<T>::list_node(const T& data,list_node<T>* front,list_node<T>* next)
34.   :data(data),front(front),next(next)
35.   {
36.
37.   }
38.
39.
40.   template<class T>
41.   class my_iterator
42.   {
43.   private:
44.       list_node<T>* current;
45.       unsigned int index;
46.   public:
47.       my_iterator& operator++() { this->current = this->current->next; this->index+
    +; return *this; };
48.       my_iterator& operator++(int) { my_iterator temp = *this; this->current = this
    ->current->next;this->index++; return temp; };
49.       my_iterator& operator--() { this->current = this->current->front; this->index
    --; return *this; };
50.       my_iterator& operator--(int) { my_iterator temp = *this; this->current = this
    ->current->front;this->index--; return temp; };
51.       bool operator==(const my_iterator& my_iterator) const { return this->current
    == my_iterator.current; };
52.       bool operator!=(const my_iterator& my_iterator) const { return this->current
    != my_iterator.current; };
53.       bool operator<(const my_iterator& my_iterator) const { return this->index < m
    y_iterator.index; };
54.       my_iterator(list_node<T>* current, unsigned int index = 0) { this->current =
    current; this->index = index; };
55.       list_node<T>* getCurrent() { return this->current; };
56.       unsigned int getIndex() { return this->index; };
57.       void reset(list_node<T>* current) { this->current = current; this->index = 0;
     };
58.   };
59.
60.   template<class T>
61.   class chainlist
```

```
62.  {
63.  private:
64.      list_node<T>* head;
65.      list_node<T>* back;
66.      unsigned int size;
67.  public:
68.      void push_back(list_node<T>* operated_list_node);
69.      void push_back(const T& data);
70.      void push_front(list_node<T>* operated_list_node);
71.      void push_front(const T& data);
72.      void pop_back();
73.      void pop_front();
74.      void insert(T data, int index);
75.      void insert(list_node<T>* operated_list_node, int index);
76.      void erase(const unsigned int index);
77.      unsigned int getSize() { return size; };
78.      unsigned int find_first(list_node<T>* operated_list_node);
79.      unsigned int find_first(const T& data);
80.      void inverse();
81.      chainlist();
82.      chainlist(const chainlist<T>& chainlist);
83.      void self_bubble_sort();
84.      void self_quick_sort();
85.      void print();
86.      list_node<T>* getNode(int index);
87.      list_node<T>* operator[](int index) { list_node<T>* temp = this->head; for (s
   ize_t i = 0; i < index; i++) { temp = temp->next; } return temp; };
88.      list_node<T>* getHead() { return head; };
89.      list_node<T>* getBack() { return back; };
90.
91.  };
92.
93.  template<class T>
94.  void chainlist<T>::push_back(list_node<T>* operated_list_node)
95.  {
96.      list_node<T>* new_list_node = new list_node<T>(operated_list_node);
97.
98.
99.      if (size == 0)
100.     {
101.         this->head = new_list_node;
102.         this->back = new_list_node;
103.         size++;
104.         return;
105.     }
106.
```

```
107.     this->back->next = new_list_node;
108.     new_list_node->front = this->back;
109.     new_list_node->next = nullptr;
110.     this->back = new_list_node;
111.
112.     size++;
113. }
114.
115. template<class T>
116. void chainlist<T>::push_back(const T& data)
117. {
118.     T new_data = data;
119.     list_node<T>* new_list_node = new list_node<T>(new_data);
120.     this->push_back(new_list_node);
121.     delete new_list_node;
122. }
123.
124. template<class T>
125. void chainlist<T>::push_front(list_node<T>* operated_list_node)
126. {
127.     list_node<T>* new_list_node = new list_node<T>(operated_list_node);
128.
129.     if (size == 0)
130.     {
131.         this->back = new_list_node;
132.         this->head = new_list_node;
133.         size++;
134.         return;
135.     }
136.     new_list_node->next = this->head;
137.     new_list_node->front = nullptr;
138.     this->head->front = new_list_node;
139.     this->head = new_list_node;
140.
141.     size++;
142. }
143.
144. template<class T>
145. void chainlist<T>::push_front(const T& data)
146. {
147.     T new_data = data;
148.     list_node<T>* new_list_node = new list_node<T>(new_data);
149.     this->push_front(new_list_node);
150.     delete new_list_node;
151. }
152.
```

```
153. template<class T>
154. void chainlist<T>::pop_back()
155. {
156.     if (this->back == nullptr) return;
157.     list_node<T>* temp = this->back;
158.     this->back = this->back->front;
159.     delete temp;
160.     if (size == 1)
161.     {
162.         this->head = nullptr;
163.         this->back = nullptr;
164.     }
165. 
166.     size--;
167. }
168. 
169. template<class T>
170. void chainlist<T>::pop_front()
171. {
172.     if (this->head == nullptr) return;
173.     list_node<T>* temp = this->head;
174.     this->head = this->head->next;
175.     delete temp;
176.     if (size == 1)
177.     {
178.         this->head = nullptr;
179.         this->back = nullptr;
180.     }
181.     size--;
182. }
183. 
184. template<class T>
185. void chainlist<T>::insert(list_node<T>* operated_list_node, int index)
186. {
187.     if (index == 0)
188.     {
189.         push_front(operated_list_node);
190.         return;
191.     }
192.     if (index == size)
193.     {
194.         push_back(operated_list_node);
195.         return;
196.     }
197. 
198.     list_node<T>* temp = this->head;
```

```
199.      for (int i = 0; i < index - 1; i++)
200.      {
201.          temp = temp->next;
202.      }
203.      list_node<T>* new_list_node = new list_node<T>(operated_list_node);
204.      new_list_node->next = temp->next;
205.      temp->next->front = new_list_node;
206.      temp->next = new_list_node;
207.      new_list_node->front = temp;
208.      size++;
209.  }
210.
211.  template<class T>
212.  void chainlist<T>::insert(T data, int index)
213.  {
214.      list_node<T>* new_list_node = new list_node<T>(data);
215.      this->insert(new_list_node, index);
216.      delete new_list_node;
217.
218.  }
219.
220.  template<class T>
221.  void chainlist<T>::erase(const unsigned int index)
222.  {
223.      if (index == 0)
224.      {
225.          pop_front();
226.          return;
227.      }
228.      if (index == size)
229.      {
230.          pop_back();
231.          return;
232.      }
233.
234.      list_node<T>* temp = this->head;
235.      for (int i = 0; i < index - 1; i++)
236.      {
237.          temp = temp->next;
238.      }
239.      list_node<T>* temp2 = temp->next;
240.      temp->next = temp2->next;
241.      temp->next->front = temp;
242.      delete temp2;
243.      size--;
244.  }
```

```
245.
246. template<class T>
247. unsigned int chainlist<T>::find_first(list_node<T>* operated_list_node)
248. {
249.     list_node<T>* temp = this->head;
250.     unsigned int index = 0;
251.     while (temp != nullptr)
252.     {
253.         if ((*temp) == (*operated_list_node)) return index;
254.         temp = temp->next;
255.         index++;
256.     }
257.     return -1;
258. }
259.
260. template<class T>
261. unsigned int chainlist<T>::find_first(const T& data)
262. {
263.     list_node<T>* temp = this->head;
264.     unsigned int index = 0;
265.     while (temp != nullptr)
266.     {
267.         if (temp->getData() == data) return index;
268.         temp = temp->next;
269.         index++;
270.     }
271.     return -1;
272. }
273.
274. template<class T>
275. void chainlist<T>::inverse()
276. {
277.     list_node<T>* temp1 = this->head;
278.     list_node<T>* temp2 = nullptr;
279.     list_node<T>* temp3 = nullptr;
280.     for (int i = 0; i < size; i++)
281.     {
282.         temp2 = temp1->next;
283.         temp3 = temp1->front;
284.         temp1->front = temp1->next;
285.         temp1->next = temp3;
286.         temp1 = temp2;
287.     }
288.     temp1 = this->head;
289.     this->head = this->back;
290.     this->back = temp1;
```

```
}

template<class T>
chainlist<T>::chainlist()
{
    this->head = nullptr;
    this->back = nullptr;
    this->size = 0;
}

template<class T>
chainlist<T>::chainlist(const chainlist<T>& chainlist)
{
    this->head = nullptr;
    this->back = nullptr;
    this->size = 0;
    list_node<T>* temp = chainlist.head;
    while (temp != nullptr)
    {
        push_back(temp);
        temp = temp->next;
    }
}

template<class T>
void chainlist<T>::print()
{
    list_node<T>* temp = this->head;
    while (temp != nullptr)
    {
        cout << temp->getData() << " ";
        temp = temp->next;
    }
    cout << endl;
}

template<class T>
list_node<T>* chainlist<T>::getNode(int index)
{
    list_node<T>* temp = this->head;
    for (int i = 0; i < index; i++)
    {
        temp = temp->next;
    }
    return temp;
}
```

```
337.
338.
339.  template<class T>
340.  void chainlist<T>::self_bubble_sort()
341.  {
342.      for (size_t i = 0; i < size; i++)
343.      {
344.          for (size_t j = 0; j < size - i - 1; j++)
345.          {
346.              if (getNode(j)->getData() > getNode(j + 1)->getData())
347.              {
348.                  my_swap(getNode(j)->getData(), getNode(j + 1)->getData());
349.              }
350.          }
351.
352.      }
353.
354.  }
355.
356.  template<class T>
357.  void quick_sort(my_iterator<T> head, my_iterator<T> tail)
358.  {
359.      if (head == tail||tail<head) { return; }
360.      T pivot = head.getCurrent()->getData();
361.      my_iterator<T> i = head;
362.      my_iterator<T> j = tail;
363.      while (i < j)
364.      {
365.          while (j.getCurrent()->getData() >= pivot && i < j)
366.          {
367.              j--;
368.          }
369.          if (i<j)
370.          {
371.              i.getCurrent()->getData() = j.getCurrent()->getData();
372.          }
373.
374.          while (i.getCurrent()->getData() <= pivot && i < j)
375.          {
376.              i++;
377.          }
378.          if (i<j)
379.          {
380.              j.getCurrent()->getData() = i.getCurrent()->getData();
381.          }
382.          if (i==j)
```

```
383.         {
384.             i.getCurrent()->getData() = pivot;
385.         }
386.     }
387.     my_iterator<T> j2 = j;
388.     j2++;
389.     quick_sort(head, j);
390.     quick_sort(j2, tail);
391. }
392. 
393. template<class T>
394. void chainlist<T>::self_quick_sort()
395. {
396.     my_iterator<T> head(this->head);
397.     my_iterator<T> tail(this->back, this->size - 1);
398.     quick_sort(head, tail);
399. }
400. 
401. class Solution {
402. 
403. public:
404.     void solute();
405.     void test();
406. };
407. 
408. void Solution::solute()
409. {
410.     chainlist<int> list1;
411.     chainlist<int> list2;
412.     int a, b;
413.     cin >> a >> b;
414.     for (size_t i = 0; i < a; i++)
415.     {
416.         int value;
417.         cin >> value;
418.         list1.push_back(value);
419.     }
420.     for (size_t i = 0; i < b; i++)
421.     {
422.         int value;
423.         cin >> value;
424.         list2.push_back(value);
425.     }
426.     chainlist<int> list3;
427.     my_iterator<int> it2(list2.getHead());
428.     my_iterator<int> it1(list1.getHead());
```

```
429.     my_iterator<int> it3(list3.getHead());
430.     while (it1.getCurrent() != nullptr)
431.     {
432.         list3.push_back(it1.getCurrent()->getData());
433.         it1++;
434.     }
435.     while (it2.getCurrent() != nullptr)
436.     {
437.         list3.push_back(it2.getCurrent()->getData());
438.         it2++;
439.     }
440.     list1.self_quick_sort();
441.     list2.self_quick_sort();
442.     list3.self_quick_sort();
443.     it1.reset(list1.getHead());
444.     it2.reset(list2.getHead());
445.     it3.reset(list3.getHead());
446.     int ans1 = 0, ans2 = 0, ans3 = 0;
447.     while (it1.getCurrent() != nullptr)
448.     {
449.         ans1 += it1.getCurrent()->getData() ^ it1.getIndex();
450.         it1++;
451.     }
452.     while (it2.getCurrent() != nullptr)
453.     {
454.         ans2 += it2.getCurrent()->getData() ^ it2.getIndex();
455.         it2++;
456.     }
457.     while (it3.getCurrent() != nullptr)
458.     {
459.         ans3 += it3.getCurrent()->getData() ^ it3.getIndex();
460.         it3++;
461.     }
462.     cout << ans1 << endl << ans2 << endl << ans3;
463. }
464.
465. void Solution::test()
466. {
467.     chainlist<int> list1;
468.     int n;
469.     cin >> n;
470.     for (size_t i = 0; i < n; i++)
471.     {
472.         int value;
473.         cin >> value;
474.         list1.push_back(value);
```

```
475.     }
476.     list1.print();
477.     my_iterator<int> it1(list1.getHead());
478.     my_iterator<int> it2(list1.getBack(), list1.getSize() - 1);
479. 
480.     quick_sort(it1, it2);
481.     list1.print();
482. }
483. 
484. int main()
485. {
486. 
487.     Solution solution;
488. 
489.     solution.solute();
490. 
491.     return 0;
492. 
493. }
```