实验6

练习1

题目

要求用带头结点的单链表（引入存储线性表长度的整型变量 curLen）实现线性表。对线性表进行若干次插入和删除，对每一次插入或删除，若成功，则输出线性表，否则输出出错信息。

执行示例：（加下划线部分为输入内容）
是否要对线性表进行插入和删除？（Y/N）
Y
进行插入还是删除？（1--插入，2--删除）
1
输入插入位置：
0
输入插入元素：
8
线性表为（8）
是否要对线性表进行插入和删除？（Y/N）
Y
进行插入还是删除？（1--插入，2--删除）
1
输入插入位置：
0
输入插入元素：
2
线性表为（2,8）
是否要对线性表进行插入和删除？（Y/N）
Y
进行插入还是删除？（1--插入，2--删除）
1
输入插入位置：
3
输入插入元素：
5
插入位置有误
是否要对线性表进行插入和删除？（Y/N）
Y
进行插入还是删除？（1--插入，2--删除）
1
输入插入位置：
2
输入插入元素：
5
线性表为（2,8,5）

是否要对线性表进行插入和删除？（Y/N）
Y
进行插入还是删除？（1--插入，2--删除）
2
输入删除位置：
2
线性表为（2,8）
是否要对线性表进行插入和删除？（Y/N）
Y
进行插入还是删除？（1--插入，2--删除）
2
输入删除位置：
8
删除位置有误
是否要对线性表进行插入和删除？（Y/N）
Y
进行插入还是删除？（1--插入，2--删除）
2
输入删除位置：
0
线性表为（8）
……
是否要对线性表进行插入和删除？（Y/N）
N
对线性表处理完毕

解析

与书上的示例代码不同，这里使用Java泛型实现一个高泛用的单链表List。（**这里定义的List将在之后的实验代码中多次使用**）

单链表支持以下特性：

1. 迭代。单链表将通过继承Iterable接口并定义自身的迭代器来支持foreach循环。
2. isEmpty、length、get、append、extend、insert、remove、delete、index、copy、count等基本方法。
3. 重写toString，使String隐式转换能返回正确的值。
4. reversed方法，将返回逆序的单链表；reverse方法，就地逆序。
5. sort方法，利用归并算法就地排序（显然，不支持随机读取的链表并不适合快速排序）。
6. toArray方法，将返回数组形式的列表；toArrayInt方法，返回数组形式的列表并将元素全部转换为int类型。

代码

List.java

```
import java.util.Iterator;

// 使用Java泛型实现的链表，支持迭代、输出数组、排序、逆序等方法
```

```java
public class List<Item extends Comparable<Item>> implements Iterable<Item> {
    private Node head;
    private int curLen;

    // Node类型
    private class Node {
        Item data;
        Node next;

        public Node(Item data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    // 构造函数
    public List() {
        this.head = new Node(null, null);
        this.curLen = 0;
    }

    public List(Item[] arr) {
        this();
        Node p = this.head;
        for (Item i : arr) {
            p.next = new Node(i, null);
            p = p.next;
        }
    }

    public List(List<Item> lst) {
        this();
        Node p = this.head;
        for (Item i : lst) {
            p.next = new Node(i, null);
            p = p.next;
        }
    }

    // 获取头节点
    public Node head() {
        return this.head;
    }

    // 是否为空
    public boolean isEmpty() {
        return this.curLen == 0;
    }

    // 获取长度
```

```
    public int length() {
        return this.curLen;
    }

    // 获取对应下标的元素
    public Item get(int index) throws Exception {
        if (index < 0 || index >= this.curLen) {
            throw new Exception("Invalid index " + index);
        }
        Node p = this.head.next;
        for (int i = 0; i < index; i++) {
            p = p.next;
        }
        return p.data;
    }

    // 增加元素
    public void append(Item x) {
        Node p = this.head;
        for (int i = 0; i < this.curLen; i++) {
            p = p.next;
        }
        p.next = new Node(x, null);
        this.curLen++;
    }

    // 扩展列表
    public void extend(Item[] arr) {
        Node p = this.head;
        for (int i = 0; i < this.curLen; i++) {
            p = p.next;
        }
        for (int i = 0; i < arr.length; i++) {
            p.next = new Node(arr[i], null);
            p = p.next;
        }
        this.curLen += arr.length;
    }

    public void extend(List<Item> lst) {
        Node p = this.head;
        for (int i = 0; i < this.curLen; i++) {
            p = p.next;
        }
        for (Item i : lst) {
            p.next = new Node(i, null);
            p = p.next;
        }
        this.curLen += lst.length();
    }
```

```
    // 插入元素
    public void insert(int index, Item x) throws Exception {
        if (index < 0 || index > this.curLen) {
            throw new Exception("Invalid index " + index);
        }
        Node p = this.head;
        for (int i = 0; i < index; i++) {
            p = p.next;
        }
        p.next = new Node(x, p.next);
        this.curLen++;
    }

    // 删除指定元素
    public void remove(Item x) throws Exception {
        Node p = this.head;
        while (p.next != null && !p.next.data.equals(x)) {
            p = p.next;
        }
        if (p.next == null) {
            throw new Exception(x + "is not in list");
        } else {
            p.next = p.next.next;
            this.curLen--;
        }
    }

    // 按下标删除元素
    public void delete(int index) throws Exception {
        if (index < 0 || index >= this.curLen) {
            throw new Exception("Invalid index " + index);
        }
        Node p = this.head;
        for (int i = 0; i < index; i++) {
            p = p.next;
        }
        p.next = p.next.next;
        this.curLen--;
    }

    // 查找元素对应下标
    public int index(Item x) throws Exception {
        Node p = this.head.next;
        int i;
        for (i = 0; p != null && !p.data.equals(x); i++) {
            p = p.next;
        }
        if (p != null) {
            return i;
```

```
            } else {
                throw new Exception(i + "is not in list");
            }
        }

        // 返回复制后的新列表
        public List<Item> copy() {
            List<Item> result = new List<Item>();
            Node p = this.head.next;
            Node q = result.head;
            while (p != null) {
                q.next = new Node(p.data, null);
                q = q.next;
                result.curLen++;
                p = p.next;
            }
            return result;
        }

        // 统计列表中指定元素的个数
        public int count(Item x) {
            int result = 0;
            Node p = this.head.next;
            while (p != null) {
                if (p.data.equals(x)) {
                    result++;
                }
                p = p.next;
            }
            return result;
        }

        // 就地排序
        public void sort() {
            // 由于链表不支持随机访问，因此不适用快速排序，这里采用归并排序
            mergeSort(this.head.next);
        }

        private Node mergeSort(Node head) {
            if (head == null || head.next == null) {
                return head;
            }
            Node middle = getMiddle(head);
            Node sHalf = middle.next;
            middle.next = null;
            return merge(mergeSort(head), mergeSort(sHalf));
        }

        private Node merge(Node a, Node b) {
            Node dummyHead, curr;
```

```
        dummyHead = new Node(null, null);
        curr = dummyHead;
        while (a != null && b != null) {
            if (a.data.compareTo(b.data) <= 0) {
                curr.next = a;
                a = a.next;
            } else {
                curr.next = b;
                b = b.next;
            }
            curr = curr.next;
        }
        curr.next = (a == null) ? b : a;
        return dummyHead.next;
    }

    private Node getMiddle(Node head) {
        if (head == null) {
            return head;
        }
        Node slow, fast;
        slow = fast = head;
        while (fast.next != null && fast.next.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }

    // 就地逆序
    public void reverse() {
        Node newHead = new Node(null, null);
        Node p = this.head.next;
        while (p != null) {
            newHead.next = new Node(p.data, newHead.next);
            p = p.next;
        }
        this.head = newHead;
    }

    // 返回逆序链表
    public List<Item> reversed() {
        List<Item> result = new List<Item>();
        result.head = new Node(null, null);
        Node p = this.head.next;
        while (p != null) {
            result.head.next = new Node(p.data, result.head.next);
            result.curLen++;
            p = p.next;
        }
```

```
        return result;
    }

    // 返回数组形式的列表
    public Item[] toArray() {
        Item[] result = (Item[]) new Object[this.curLen];
        // 由于类型擦除的问题，Java不支持泛型数组的创建，只能强制转换
        Node p = this.head.next;
        for (int i = 0; i < this.curLen; i++) {
            result[i] = p.data;
            p = p.next;
        }
        return result;
    }

    // 返回数组形式的列表并将全部元素强制转换为int
    public int[] toArrayInt() {
        int[] result = new int[this.curLen];
        Node p = this.head.next;
        for (int i = 0; i < this.curLen; i++) {
            result[i] = (int) p.data;
            p = p.next;
        }
        return result;
    }

    // 重写toString方法
    public String toString() {
        Node p = this.head.next;
        StringBuilder builder = new StringBuilder();
        builder.append("[");
        while (p != null) {
            builder.append(p.data + ", ");
            p = p.next;
        }
        if (this.curLen > 0) {
            builder.delete(builder.length() - 2, builder.length());
        }
        builder.append("]");
        return builder.toString();
    }

    // 返回迭代器，以支持迭代
    public Iterator<Item> iterator() {
        return new LinkListIterator();
    }

    // 迭代器
    private class LinkListIterator implements Iterator<Item> {
        private Node current = head.next;
```

```
        public boolean hasNext() {
            return current != null;
        }

        public void remove() {}

        public Item next() {
            Item data = current.data;
            current = current.next;
            return data;
        }
    }
}
```

ListTest.java

```
import java.util.Scanner;

public class ListTest {
    public static void main(String[] args) throws Exception{
        List<Integer> lst = new List<Integer>();
        Scanner in = new Scanner(System.in);
        System.out.println("是否对单链表进行插入或删除? (Y/N)");
        boolean yesOrNo = in.nextLine().equals("Y");
        while (yesOrNo) {
            System.out.println("进行插入还是删除? (1--插入, 2--删除)");
            if (in.nextLine().equals("1")) {
                System.out.println("输入插入位置:");
                int index = Integer.parseInt(in.nextLine());
                System.out.println("输入插入元素:");
                int x = Integer.parseInt(in.nextLine());
                try {
                    lst.insert(index, (Integer) x);
                    System.out.println("当前链表为: " + lst);
                } catch (Exception e) {
                    System.out.println("Error: " + e.getMessage());
                }
            } else {
                System.out.println("输入删除位置:");
                int index = Integer.parseInt(in.nextLine());
                try {
                    lst.remove(index);
                    System.out.println("当前链表为: " + lst);
                } catch (Exception e) {
                    System.out.println("Error: " + e.getMessage());
                }
            }
```

```
                System.out.println("是否对单链表进行插入或删除？（Y/N）");
                yesOrNo = in.nextLine().equals("Y");
            }
            in.close();
            System.out.println("单链表处理完毕");
        }
    }
```

测试（运行ListTest.java）（输入输出交替）

```
是否对单链表进行插入或删除？（Y/N）
Y
进行插入还是删除？（1--插入，2--删除）
1
输入插入位置：
0
输入插入元素：
8
当前链表为：[8]
是否对单链表进行插入或删除？（Y/N）
Y
进行插入还是删除？（1--插入，2--删除）
1
输入插入位置：
0
输入插入元素：
2
当前链表为：[2，8]
是否对单链表进行插入或删除？（Y/N）
Y
进行插入还是删除？（1--插入，2--删除）
1
输入插入位置：
3
输入插入元素：
5
Error: Invalid index 3
是否对单链表进行插入或删除？（Y/N）
Y
进行插入还是删除？（1--插入，2--删除）
1
输入插入位置：
2
输入插入元素：
5
当前链表为：[2，8，5]
是否对单链表进行插入或删除？（Y/N）
Y
```

```
进行插入还是删除？（1--插入，2--删除）
2
输入删除位置：
2
当前链表为：[2，8]
是否对单链表进行插入或删除？（Y/N）
Y
进行插入还是删除？（1--插入，2--删除）
2
输入删除位置：
8
Error: Invalid index 8
是否对单链表进行插入或删除？（Y/N）
Y
进行插入还是删除？（1--插入，2--删除）
2
输入删除位置：
0
当前链表为：[8]
是否对单链表进行插入或删除？（Y/N）
N
单链表处理完毕
```

练习2

题目

有2个元素都是整数的非递减有序表A和B。求由A和B的所有元素组成的线性表C中的第k小元素。说明：数据自定。

解析

定义指针i和j，初始值都为0；定义int类型变量count，初始值为1；定义变量result，存储输出结果。

重复以下步骤，直到count等于k：

1. i不断递增，直至lst1[i] >= lst2[j]，result更新为该过程中最大的lst1[i]，每更新一次count加1
2. j不断递增，直至lst1[i] < lst2[j]，result更新为该过程中最大的lst2[j]，每更新一次count加1

result的终值即为结果。

代码

Exercise_2.java

```
public class Exercise_2 {
    public static void main(String[] args) throws Exception {
        int k = ...; // k的具体值将在下面的“输入”中给出
        int[] lst1 = { ... }; // lst1的具体值将在下面的“输入”中给出
        int[] lst2 = { ... }; // lst2的具体值将在下面的“输入”中给出
        int i = 0, j = 0;
        int count = 1;
        int result = lst1[0];
        while (count < k) {
            while (lst1[i] < lst2[j] && count < k) {
                if (lst1[i] > result) {
                    result = lst1[i];
                    count++;
                }
                i++;
            }
            while (lst2[j] <= lst1[i] && count < k) {
                if (lst2[j] > result) {
                    result = lst2[j];
                    count++;
                }
                j++;
            }
        }
        System.out.println(result);
    }
}
```

输入

```
k = 17
lst1 = {2, 3, 4, 5, 9, 10, 17, 19, 22, 23, 25, 26, 27, 30, 30, 33, 39, 40, 42, 44,
46, 46, 47, 48, 48, 50, 54, 55, 56, 57, 63, 67, 68, 70, 72, 73, 76}
lst2 = {8, 13, 15, 16, 16, 20, 21, 22, 24, 28, 29, 31, 32, 34, 36, 37, 38, 40, 40,
41, 43, 44, 45, 48, 49, 50, 50, 52, 53, 59, 61, 62, 64, 64, 67, 68, 75, 77, 77}
```

输出

```
24
```

附

实际上，还根据书上的示例实现了普通的单链表LinkList，但由于设计比较烂，使用时需要经常考虑数据类型的转换，因此所有实验的代码均未使用这一数据类型。

Node.java

```
public class Node {

    private Object data;
    private Node next;

    public Object getData() {
        return data;
    }

    public void setData(Object data) {
        this.data = data;
    }

    public Node getNext() {
        return next;
    }

    public void setNext(Node next) {
        this.next = next;
    }

    public Node() {
        this(null, null);
    }

    public Node(Object data) {
        this(data, null);
    }

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }

}
```

LinkList.java

```
import java.util.Scanner;

public class LinkList {

    private Node head;
    private int length;

    public LinkList() {
        length = 0;
        head = new Node();
    }

    public LinkList(int n, boolean order) {
        this();
        if (order) {
            create1(n);
        } else {
            create2(n);
        }
    }

    // 尾插法
    public void create1(int n) {
        Scanner sc = new Scanner(System.in);
        for (int i = 0; i < n; i++) {
            insert(0, sc.next());
        }
        sc.close();
    }

    // 头插法
    public void create2(int n) {
        Scanner sc = new Scanner(System.in);
        for (int i = 0; i < n; i++) {
            append(sc.next());
        }
        sc.close();
    }

    public void clear() {
        length = 0;
        head.setNext(null);
    }

    public boolean isEmpty() {
        return length == 0;
    }
```

```
    public Node head() {
        return head;
    }

    public int length() {
        return length;
    }

    public Object get(int index) throws IndexOutOfBoundsException {
        if (index < 0 || index >= length) {
            throw new IndexOutOfBoundsException("Invalid index " + index);
        }
        Node p = head.getNext();
        for (int i = 0; i < index; i++) {
            p = p.getNext();
        }
        return p.getNext();
    }

    public void append(Object x) {
        Node p = head;
        for (int i = 0; i < length; i++) {
            p = p.getNext();
        }
        p.setNext(new Node(x));
        length++;
    }

    public void insert(int index, Object x) throws IndexOutOfBoundsException {
        if (index < 0 || index > length) {
            throw new IndexOutOfBoundsException("Invalid index " + index);
        }
        Node p = head;
        for (int i = 0; i < index; i++) {
            p = p.getNext();
        }
        p.setNext(new Node(x, p.getNext()));
        length++;
    }

    public void remove(int index) throws IndexOutOfBoundsException {
        if (index < 0 || index >= length) {
            throw new IndexOutOfBoundsException("Invalid index " + index);
        }
        Node p = head;
        for (int i = 0; i < index; i++) {
            p = p.getNext();
        }
        p.setNext(p.getNext().getNext());
```

```
            length--;
        }

        public int indexOf(Object x) {
            Node p = head.getNext();
            int i;
            for (i = 0; p != null && !p.getData().equals(x); i++) {
                p = p.getNext();
            }
            if (p != null) {
                return i;
            } else {
                return -1;
            }
        }

    }
```

心得体会

1. 为单链表引入头结点能很大程度上简化代码。
2. 链表能够灵活利用空间，插入删除操作很快，但查找操作的时间复杂度是线性的，且空间利用率不高。