数据结构与算法分析：第二次作业

线性表\栈\队列应用训练

课程	数据结构与算法分析
姓名	白靖
专业班级	软件 52
学号	2151601024
邮箱	516267116@qq.com
提交日期	2016 年 11 月 1 日

目录

数据结构与算法分析作业报告：第二次作业

线性表\栈\队列应用训练

此报告作为数据结构与算法课程的第一次作业报告，内容包括作业题的题目分析、程序实现、结果展示与算法分析。作业中的程序代码采用 Java 语言实现并在 Windows 环境下测试，运行环境参数如下：

处理器及内存	CPU：Intel(R) Core(TM) i7-4710HQ @2.50GHz RAM：16.00GB (DDR3L 1600MHz)
运行操作系统	MS Windows 10 Home x64 (Version: 10.0.14393)
Java 运行环境	Java(TM) SE Runtime Environment (build 1.8.0_102-b14)

1 四种数据结构的实现

1.1 作业题目

> *实现如下四种数据结构：*
> *未排序的顺序数组、已排序的顺序数组、未排序的单循环链表和已排序的单循环链表。为了简化问题，这些数据结构都仅仅存储 int 类型，并且实现了如下抽象对象类型*
> *List：*
> *interface List{*
> *boolean search(int x);*
> *boolean insert(int x);*
> *int delete(int x);*
> *int successor(int x);//获得该线性表中 x 元素的直接后继元素*
> *int predecessor(int x);//获得该线性表中 x 元素的直接前驱元素*
> *int minimum();//获得该线性表的最小元素*
> *int maximum();//获得该线性表的最大元素*
> *int KthElement(int k);//获得线性表中第 k 大元素，参数为指定的 k 值的大小*
> *。。。//此处根据需要自己添加行为定义*
> *}*
>
> *本题目标：*
> *1）具体实现所要求的四种数据结构*
> *2）用一个表将每种数据结构中的每个具体行为的时间渐进性列出来。*

所有测试代码详见百度云：http://pan.baidu.com/s/1c2sp01E
密码：38a9

1.2 程序实现

```
//UnsortedArray.java 未排序数组

package list_test;

import java.util.Arrays;
import java.util.Random;

public class UnsortedArray implements List {

    //默认数组大小200

    private int DEFAULT_SIZE = 200;
    private static int size = 0;
    public int[] elementData;
    private static int index = 0;

    //错误标记（int类型下界）

    private int ERROR = -2147483648;


    //构造函数

    public UnsortedArray() {
        elementData = new int[DEFAULT_SIZE];
        Random ra = new Random();
        for(; size < DEFAULT_SIZE - 20; size++) {
            elementData[size] = ra.nextInt(200);
        }
    }

    @Override
    public boolean search(int x) {
        for(int i = 0; i < size; i++) {
            if(elementData[i] == x) {
                index = i;
                return true;
            }
        }
        return false;
    }

    @Override
    public boolean insert(int x) {
        if(size == DEFAULT_SIZE) {
```

```
            return false;
        }
        elementData[size++] = x;
        return true;
    }

    @Override
    public int delete(int x) {

        //仅当此元素存在表中时执行删除

        while(search(x) && size != 0) {
            int temp = x;
            for(int i = index; i < size; i++) {
                elementData[i - 1] = elementData[i];
            }
            size--;
            return temp;
        }
        if(!search(x)) {

            throw new Error("要删除的变量不存在！");

        } else{

            throw new Error("数组中已无元素！");

        }
    }

    @Override
    public int successor(int x) {

        //寻找x元素

        search(x);
        if(index == size - 1) {

            throw new Error("该元素无直接后继!");

        } else{
            return elementData[index + 1];
        }
    }

    @Override
    public int predecessor(int x) {

        //寻找x元素
```

```
        search(x);
        if(index == size - 1) {
            throw new Error("该元素无直接前驱！");
        } else{
            return elementData[index - 1];
        }
    }

    @Override
    public int minimum() {
        int min = -ERROR;
        for(int i = 0 ; i < size; i++)
            min = min < elementData[i] ? min : elementData[i];
        return min;
    }

    @Override
    public int maximum() {
        int max = ERROR;
        for(int i = 0 ; i < size; i++)
            max = max > elementData[i] ? max : elementData[i];
        return max;
    }

    @Override
    public int KthElement(int k) {
        if(k <=0 || k > size)
            throw new Error("无第" + k + "大元素！");

        //保存至临时数组
        int[] temp = elementData.clone();
        //对临时数组排序
        Arrays.sort(temp, 0, size);
        return elementData[size - k];
    }

}
```

```java
//SequenceArray.java 已排序数组

package list_test;

public class SequenceArray implements List {

    //默认数组大小200

    private int DEFAULT_SIZE = 200;
    private static int size = 0;
    public int[] elementData;
    private static int index = 0;

    //以@param firstElement 指定顺序线性表中第一个元素创建一个默认大小的顺序

数组

    SequenceArray(int firstElement){
        elementData = new int[DEFAULT_SIZE];
            for(; size < DEFAULT_SIZE - 20; size++) {
                elementData[size] = firstElement;
                firstElement++;
            }
    }

    @Override

    //二分法查找

    public boolean search(int x) {
        boolean flag = true;
        int start = 0;
        int end = size;
        while(x < elementData[size - 1] && x > elementData[0] && start <

end) {
            int mid = (start + end) / 2;
            if(x < elementData[mid]) {
                end = mid - 1;
            } else if(x > elementData[mid]){
                start  = mid + 1;
            } else{
                flag = true;

                //记录下标

                index = mid;
                return flag;
            }
```

```
        }
        if(x == elementData[0]){
            flag = true;
            index = 0;
            return flag;
        } else{
            flag = false;

            System.out.println(x + "元素不存在");

            return flag;
        }
    }

    @Override
    public boolean insert(int x) {
        if(size == DEFAULT_SIZE) {
            return false;
        }
        if(size == 0) {
            elementData[size++] = x;
            return true;
        }

        //将x插入到已排序数组的合适位置

        for(int i = size-1; i >= 0; i--) {
            if(elementData[i] > x) {
                elementData[i + 1] = elementData[i];
            } else{
                elementData[i + 1] = x;
                break;
            }
        }
        size++;
        return true;
    }

    @Override
    public int delete(int x) {

        //仅当此元素存在表中时执行删除

        while(search(x) && size != 0) {
            int temp = x;
            for(int i = index; i < size; i++) {
                elementData[i - 1] = elementData[i];
```

```
        }
        size--;
        return temp;
    }
    if(!search(x)) {
        throw new Error("要删除的变量不存在！");
    } else{
        throw new Error("数组中已无元素！");
    }

}

@Override
public int successor(int x) {
    //寻找x元素
    search(x);
    if(index == size - 1) {
        throw new Error("该元素无直接后继!");
    } else{
        return elementData[index + 1];
    }
}

@Override
public int predecessor(int x) {
    //寻找x元素
    search(x);
    if(index == size - 1) {
        throw new Error("该元素无直接前驱！");
    } else{
        return elementData[index - 1];
    }
}


@Override
public int minimum() {
    while(size != 0) {
```

```
            System.out.println("最小元素是" +  this.elementData[0]);
            return  this.elementData[0];
        }
        throw new Error("数组为空！");
    }

    @Override
    public int maximum() {
        while(size != 0) {
            System.out.println("最大元素是" +  this.elementData[size -
1]);
            return this.elementData[size - 1];
        }
        throw new Error("数组为空！");
    }

    @Override
    public int KthElement(int k) {
        if(k > size || k <= 0) {
            throw new Error("无第" + k + "大元素！");
        } else{
            return elementData[size - k];
        }
    }
}
```

```java
//UnsortedLinkList.java 未排序链表

package list_test;

import java.util.Arrays;

public class UnsortedLinkList implements List {

    public Node first; // 定义一个头结点

    public Node current = null;
    public int size = 0;

    //错误标记（int类型下界）

    private int ERROR = -2147483648;
    public class Node {

         protected Node next; //指针域

         protected int data;//数据域


         public Node( int data) {
               this. data = data;
         }
    }

    public UnsortedLinkList() {
        first = new Node(0);

        //指向自身

        first.next = first;
    }

    @Override
    public boolean search(int x) {
        current = first;
        while (current.next != first) {
            if (current.next.data == x)
                return false;
           current = current. next;
        }
        return true;
    }

    @Override
```

```
public boolean insert(int x) {
    Node node = new Node(x);
    node.next = first;
    current.next = node;
    size++;
    return true;
}

@Override
public int delete(int x) {
    search(x);
     if(current.next == first)
           throw new Error("无元素可以删除！");
     current.next = current.next.next;
     size --;
     return x;

}

@Override
public int successor(int x) {
    search(x);
     if(current.next == first || current.next.next == first)
         throw new Error("无直接后继！");
     return current.next.next.data;
}

@Override
public int predecessor(int x) {
    search(x);
     if(current.next == first || current.next.data == x)
         throw new Error("无直接前驱！");
     return current.data;
}

@Override
public int minimum() {
    int min = -ERROR;
    current = first;
    while (current.next != first) {
        if(current.next.data < min) {
```

```
            //最小值变更及current指针后移
            min = current.next.data;
            current = current.next;
        }
    }
    return min;
}

@Override
public int maximum() {
    int max = ERROR;
    current = first;
    while (current.next != first) {
        if(current.next.data > max) {
            //最大值变更及current指针后移
            max = current.next.data;
            current = current.next;
        }
    }
    return max;
}

@Override
public int KthElement(int k) {
    if(k <=0 || k > size)
        throw new Error("无第" + k + "大元素！");
    int[] temp = new int[size];
    int i = 0;
    current = first;
    //拷贝到数组
    while(current.next != first)
    {
        temp[i++] = current.next.data;
        current = current.next;
    }
    //排序并查找
    Arrays.sort(temp);
    return temp[size - k];
}
```

```
}

//SortedLinkList.java 已排序链表
package list_test;

public class SortedLinkList extends UnsortedLinkList{
    public boolean insert(int x)
    {
        size++;
        current = first;
        while(current.next != first && current.next.data < x)
            current = current.next;
        Node ins = new Node(0);
        ins.data = x;
        ins.next = current.next;
        current.next = ins;
        return true;
    }
    public int KthNode(int k)
    {
        if(k <=0 || k > size)
            throw new Error("无第" + k + "大元素！");
        Node current = first;
        for(int pos = 0;pos <= size - k;pos++)
            current = current.next;
        return current.data;
    }

}
```

1.3 算法分析

操作	时间复杂度
二分法搜索	O(nlogn)
Arrays.sort()排序	O(nlogn)
遍历元素	O(n)
取直接后继（链表&数组）	O(1)
取直接前驱（链表&数组）	O(1)
求最大/最小值（无序数组&链表）	O(n)
求最大/最小值（有序数组）	O(1)
求第 k 大元素（链表&无序数组） 由于其中包括 Arrays.sort()操作	O(nlogn)

求第 k 大元素（有序数组）	O(1)
插入操作（无序数组&无序链表）	O(1)
插入操作（有序数组）	$O(n^2)$
删除操作(数组)	O(n)
删除操作(链表)	O(1)

2　布尔表达式计算器

2.1 作业题目

> *本题是要计算如下的布尔表达式：(T \|T)& F &(F\|T)，其中 T 表示 True，F 表示 False。表达式可以包含如下运算符：！表示 not，&表示 and，\|表示 or，允许使用括号。*
>
> *为了执行表达式的运算，要考虑运算符的优先级：not 的优先级最高，or 的优先级最低。计算器要产生 V 或 F，表达最终表达式计算的结果。*
>
> *1）一个表达式不超过 100 个符号，符号间可以用任意个空格分开，或者根本没有空格，所以表达式总的长度也就是字符的个数，它是未知的。*
>
> *2）要能处理表达式中出现括号不匹配、运算符缺少运算操作数等常见的输入错误。*

2.2 程序实现（给出测试代码）

```
//Calc.java 布尔计算器

//引入输入类
import java.util.Scanner;
//引入栈类
import java.util.Stack;

public class Calc {
    //提示函数
    static void alert(String message) {
        System.out.println(message);
    }
```

```
//比较优先级
static boolean compare(char op1,char op2) {
    String ops = "!&|(";
    return ops.indexOf(op1) <= ops.indexOf(op2);
}

//计算函数
static void calc(char op,Stack<Boolean> stack_num) {
    if(op == '&' || op == '|') {
        if(stack_num.size() < 2) {
            alert("nums exception");
        } else{
            boolean num1,num2;
            num1 = stack_num.pop();
            num2 = stack_num.pop();
            if(op == '&') {
                stack_num.push(num1 && num2);
            } else{
                stack_num.push(num1 || num2);
            }
        }
    }
    else if(op == '!') {
        if(stack_num.size() < 1) {

            alert("无运算数！");

        }
        else {
            stack_num.push(!stack_num.pop());
        }
    }
}
static boolean valid(String input) {

    //运算数栈

    Stack<Boolean> stack_num = new Stack<Boolean>();

    //运算符栈

    Stack<Character> stack_op = new Stack<Character>();

    //获取表达式长度

    int len = input.length();
```

```
for(int i = 0;i < len;i++) {
    //根据索引号获得表达式中相应位置的字符
    char c = input.charAt(i);
    switch(c) {
        case 'T':
            //压栈
            stack_num.push(true);
        break;
        case 'F':
            //压栈
            stack_num.push(false);
        break;
        case ')':
            //遇到右括号，弹栈
            assert(stack_op.isEmpty() == false);
            char preop = stack_op.pop();
            while(preop != '(') {
                calc(preop,stack_num);
                assert(stack_op.isEmpty() == false);
                preop = stack_op.pop();
            }
        break;
        case '(':
            stack_op.push('(');
        break;
        case '&':
        case '|':
        case '!':
            if(stack_op.isEmpty()) {
                stack_op.push(c);
                break;
            }
            preop = stack_op.pop();
            if(compare(c,preop)) {
                stack_op.push(preop);
                stack_op.push(c);
            } else {
                calc(preop,stack_num);
                stack_op.push(c);
            }
```

```
                break;
                default:
                    alert("非法字符！");
                break;
            }
        }
        //运算符栈非空
        while(!stack_op.isEmpty()) {
            char preop = stack_op.pop();
            calc(preop,stack_num);
        }
        return stack_num.pop();
    }
    //测试代码（输入）
    public static void main(String[] args) {
        //scan未关闭，可以持续输入

        System.out.println("输入要计算的式子：");
        Scanner scan = new Scanner(System.in);
        while(true) {
            String input = scan.next();
            if(valid(input)) {
                alert("V");
            } else{
                alert("F");
            }
        }
    }
}
```

2.3 算法分析

布尔计算器类设计的核心是将输入的表达式转换为后綴达式，并借助两个栈——操作符栈操作数栈实现的。进行关键的弹栈与压栈操作的时间复杂度都是 O(1)，因此此算法是有时间保证的。

3　利用队列实现基数排序

3.1 作业题目

> *利用队列实现对某一个数据序列的排序（采用基数排序），其中对数据序列的数据（第 1 和第 2 条进行说明）和队列的存储方式（第 3 条进行说明）有如下的要求：*
>
> *1） 当数据序列是整数类型的数据的时候，数据序列中每个数据的位数不要求等宽，比如：*
>
> *1、21、12、322、44、123、2312、765、56*
>
> *2） 当数据序列是字符串类型的数据的时候，数据序列中每个字符串都是等宽的，比如：*
>
> *"abc","bde","fad","abd","bef","fdd","abe"*
>
> *3）要求重新构建队列的存储表示方法：使其能够将 n 个队列顺序映射到一个数组 listArray 中，每个队列都表示成内存中的一个循环队列【这一项是可选项】*

3.2 程序实现

```
//Element.java Element接口类
public interface Element {
    //返回以@pos 为位置参数的元素值
    public abstract int numAt(int pos);
    //元素长度
    public abstract int length();
}

//MyInt.java 整型数类
public class MyInt implements Element{

    //构造函数
    int num;
    MyInt(int num) {
        this.num = num;
    }

    @Override
    public int numAt(int pos) {
        //临时变量
```

```
        int temp = num;
        while(pos > 0) {
            temp /= 10;
            pos--;
        }
        return temp % 10;
    }

    @Override
    public int length() {
        int count = 0;
        int tem = num;
        while(tem > 0) {
            count ++;
            tem /= 10;
        }
        return count;
    }
}

//MyInt.java 字符串类
public class MyStr implements Element {

    //构造函数
    String str;
    MyStr(String str) {
        this.str = str;
    }

    @Override
    public int numAt(int pos) {
        int len = str.length();
        if(pos >= len)
            return 0;
        else
            return str.charAt(len-1-pos) - 'a';
    }

    @Override
    public int length() {
        return str.length();
    }
```

```
}

//Queue.java 队列类
public class Queue {
    int DEFAULT_SIZE;
    Element[] instance;
    int head = 0;
    int tail = 1;

    //构造函数
    Queue()
    {
        DEFAULT_SIZE = 300;
        instance = new Element[DEFAULT_SIZE];
    }

    //判断是否空队
    boolean isEmpty() {
        return (head+1) % DEFAULT_SIZE == tail;
    }

    //判断是否满队
    boolean isFull() {
        return head == tail;
    }

    //入队
    boolean enqueue(Element e) {
        if(isFull()) {
            return false;
        } else{
            instance[tail] = e;
            tail = (tail + 1) % DEFAULT_SIZE;
        }
        return true;
    }

    //出队
```

```
    Element dequeue() {
        if(isEmpty()) {
            return null;
        }
        else {
            head = (head + 1) % DEFAULT_SIZE;
            return instance[head];
        }
    }


    //获得第一个元素
    Element head() {
        if(isEmpty())
            return null;
        else
            return instance[(head + 1) % DEFAULT_SIZE];
    }

}
```

3.3 算法分析

本算法主要借助了队列这种数据结构对整数型和字符串型数据进行了基数排序，但对要求 3 未给予实现，其时间复杂度为 O (nlog(r)m)，其中 r 为所采取的基数，而 m 为堆数。且排序主要用于外排序（即大量数据储存在外存中时），该算法是有时间保证的。