数据结构与算法
实验报告

第 2 次

姓名	丁昌灏
班级	软件 72 班
学号	2174213743
电话	15327962577
Email	dch0031@stu.xjtu.edu.cn
日期	2020-11-6

目录

实验 1

1、题目

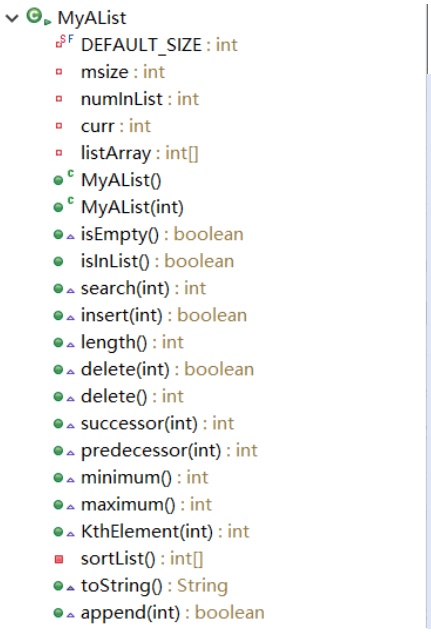
针对上面线性表的 ADT 定义，完成如下工作：

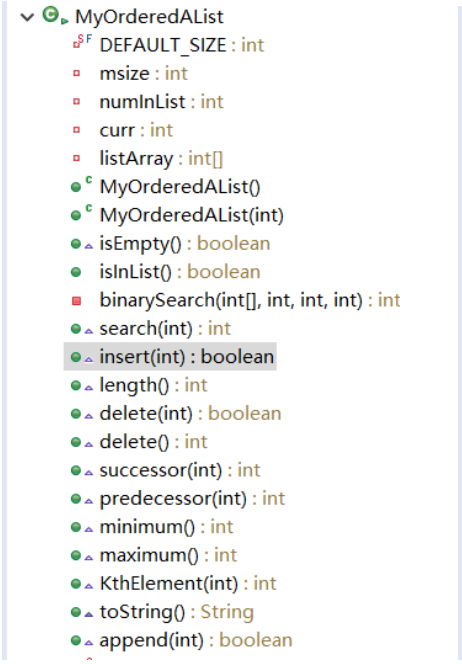
(1)按照如下要求设计和实现不同的数据结构:存储方式分别采用①未排序的顺序结构、②已排序的顺序结构、③未排序的单循环链表、④已排序的单循环链表。

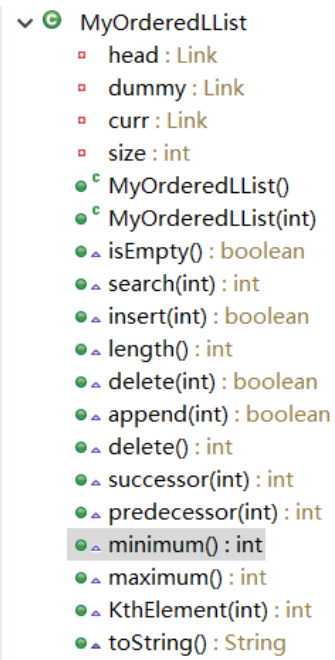
(2)设置一组测试,验证(1)中实现的四个数据结构都可以正确执行任务;(3)设计一个表格，将四个不同的数据结构在各个行为的时间复杂度列出来。

2、数据结构设计

为了便于操作，定义辅助变量 size，curr 分别表示线性表大小和当前结点；在顺序表中再定义 msize 表示最大长度，在链表中定义 dummy 结点。







3、算法设计

对于寻找线性表中是否存在某个元素，未排序的顺序结构线性表采用遍历搜索，已排序的顺序表采用二分查找，两种链表均采用遍历方法。

对于获得线性表中第 k 大元素，在未排序顺序结构线性表中采用先排序后寻找的策略，在已排序的顺序结构中直接根据下标寻找，对于两种链表采用遍历的方法寻找；

4、主干代码说明

由于方法很多，在这里展示 search()和 KthElement()方法，顺序依次为：

①未排序的顺序结构

```
@Override
public int search(int x) { // Return index of Object x
    for (int i = 0; i < numInList; i++) {
        if (listArray[i] == x) {
            return i;
        }
    }
    return -1;
}

@Override
public int KthElement(int k) {
    if (k > numInList || k < 1) {
        throw new IndexOutOfBoundsException("k is out of bounds");
    }
    int[] temp = sortList();
    return temp[temp.length - k];
}
```

---

②已排序的顺序结构

```
@Override
public int search(int x) { // Return index of Object x
    return binarySearch(listArray, x, 0, numInList - 1);
}


@Override
public int KthElement(int k) {
    if (k > numInList || k < 1) {
        throw new IndexOutOfBoundsException("k is out of bounds");
    }
    return listArray[numInList - k];
}
```

③未排序的单循环链表

```
@Override
public int search(int x) {
    if (!isEmpty()) {
        int index = 0;
        Link temp = head;
        while (temp != null) {
            if (temp.element() == x) {
                return index;
            }
            temp = temp.next();
            index++;
        }
    }
    return -1;
}
```

```
@Override
public int KthElement(int k) {
    if(k<=0) {
        throw new IndexOutOfBoundsException("index out of bounds");
    }
    if (isEmpty()) {
        throw new IndexOutOfBoundsException("list is empty");
    }
    int[] array = new int[size];
    Link temp = head;
    int i = 0;
    while (temp != null) {
        array[i] = temp.element();
        i++;
        temp = temp.next();
    }
    SortAlgorithm.qsort(array, 0, array.length - 1);
    return array[array.length - k];
}
```

---

④已排序的单循环链表。

```
@Override
public int search(int x) {
    if (!isEmpty()) {
        int index = 0;
        Link temp = head;
        while (temp != null) {
            if (temp.element() == x) {
                return index;
            }
            temp = temp.next();
            index++;
        }
    }
    return -1;
}
```

```java
@Override
public int KthElement(int k) {
    if (k < 1 || k >size) {
        throw new IndexOutOfBoundsException("index out of bounds");
    }
    if (isEmpty()) {
        throw new IndexOutOfBoundsException("list is empty");
    }
    int count = size - k;
    Link temp = head;
    while (count > 0) {
        temp = temp.next();
        count--;
    }
    return temp.element();
}
```

5、运行结果展示

测试代码类似，展示线性表对于的操作：插入、删除、最大最小值、第 k（测试中为 3）大的值、最大最小值的前驱和后继元素（如果是头节点和尾结点不存在设为-1）

```java
MyAList l1 = new MyAList();
l1.insert(2);
l1.insert(10);
l1.insert(7);
l1.insert(1);
l1.insert(5);
l1.insert(0);
l1.insert(4);
System.out.println("the elements of Alist is:" + l1);
l1.delete(7);
l1.delete(5);
System.out.println("the elements of Alist is:" + l1);
System.out.print(l1.minimum() + "\t");
System.out.print(l1.maximum() + "\t");
System.out.println(l1.KthElement(3));
System.out.print(l1.successor(10) + "\t");
System.out.println(l1.predecessor(10));
System.out.print(l1.successor(0) + "\t");
System.out.println(l1.predecessor(0));
```

依次测试：

①未排序的顺序结构

```
the elements of Alist is:[4, 0, 5, 1, 7, 10, 2]
the elements of Alist is:[4, 0, 1, 10, 2]
0       10      2
2       1
1       4
```

②已排序的顺序结构

```
the elements of OAlist:[0, 1, 2, 4, 5, 7, 10]
the elements of OAlist:[0, 1, 2, 4, 10]
0       10      2
-1      4
1       -1
```

③未排序的单循环链表

结果应该与未排序顺序结构相同

```
the elements of Llist:[4, 0, 5, 1, 7, 10, 2]
the elements of Llist:[4, 0, 1, 10, 2]
0       10      2
2       1
1       4
```

④已排序的单循环链表

结果应该与已排序的顺序结构相同

```
the elements of OLlist:[0, 1, 2, 4, 5, 7, 10]
the elements of OLlist:[0, 1, 2, 4, 10]
0       10      2
-1      4
1       -1
```

均通过测试

四种列表对应操作的时间复杂度如下表：

	search(int x)	insert(int x)	delete(int x)	successor(int x)	predecessor(int x)	minimum()	maximum()	KthElement(int k)
未排序的顺序结构	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)	O(nlogn)
已排序的顺序结构	O(logn)	O(n)	O(n)	O(logn)	O(logn)	O(1)	O(1)	O(1)
未排序的单循环链表	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)	O(n)	O(nlogn)
已排序的单循环链表	O(n)	O(n)	O(n)	O(n)	O(n)	O(1)	O(1)	O(1)

6、总结和收获

通过编程实现算法刚开始还是遇到了许多问题，反应了实践的重要性，在编程之前没有想到有这么多的细节需要处理。

实验 2

1、题目

本题是要计算类似如下的布尔表达式：(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(非)，&表示 and （与），|表示 or （或），^表示 xor (异或)，并允许使用括号。

2、算法设计

从左到右扫描字符串，遇到操作数压入操作数栈，遇到操作符压入操作符栈，在操作符压入栈前检查栈中是否存在更高优先级的操作符，若存在先将其弹出并进行计算，然后再将操作符压入栈。扫描完毕后若操作数栈存在操作符则弹出计算。

异常检测：括号匹配时，申明两个变量记录栈中左括号和右括号数，左括号压入时左括号数+1，而右括号压入栈时先进行判断，如果左括号数>右括号数则抛出异常，否则正常入栈并进行操作。每个操作符弹出时检查操作数栈是否有对应数量的操作数，如果不相同则抛

出异常。

3、主干代码说明

为为了方便，将 TF 与对应 boolean 变量转换封装成一个函数

```
private static boolean char2Boolean(char symbol) {
    if (symbol == 'T') {
        return true;
    } else {
        return false;
    }
}

private static char boolean2Char(boolean bool) {
    if (bool) {
        return 'T';
    } else {
        return 'F';
    }
}
```

从左到右扫描压栈逻辑：

```
public static boolean calculate(String expression) throws Exception {
    leftCount = 0;
    rightCount = 0;
    AStack<Character> operand = new AStack<>();
    AStack<Character> operator = new AStack<>();
    for (int i = 0; i < expression.length(); i++) {
        char symbol = expression.charAt(i);
        if (symbol == 'T' || symbol == 'F') {
            operand.push(symbol);
        } else {
            switch (symbol) {
            case '(':
                operator.push(symbol);
                leftCount++;
                break;
            case ')':
                if (leftCount <= rightCount)
                    throw new Exception("bracket mismatch");
                operator.push(symbol);
                rightCount++;
                calculateBracket(operand, operator);
                break;
            case '!':
                operator.push(symbol);
                break;
            case '&':
                if (operator.contains('!')) {
                    not(operand, operator);
                }
                operator.push(symbol);
                break;
            case '^':
                if (operator.contains('!')) {
                    not(operand, operator);
```

```
                if (operator.contains('!')) {
                    not(operand, operator);
                }
                if (operator.contains('&')) {
                    and(operand, operator);
                }
                operator.push(symbol);
                break;
            case '|':
                if (operator.contains('!')) {
                    not(operand, operator);
                }
                if (operator.contains('&')) {
                    and(operand, operator);
                }
                if (operator.contains('^')) {
                    xor(operand, operator);
                }
                operator.push(symbol);
                break;
            }
        }
    }
    if (leftCount != rightCount)
        throw new Exception("bracket mismatch");

    if (operator.isEmpty() && operand.size() > 1)
        throw new Exception("missing operator");

    while (!operator.isEmpty()) {
        char symbol = operator.topValue();
        choose2Execute(symbol, operand, operator);
    }
    if (operand.size() > 1) {
        throw new Exception("missing operator");
    }
    if (operand.pop() == 'T') {
        return true;
    } else {
        return false;
    }
}
```

各种操作符对应的运算逻辑：

```
private static void or(AStack<Character> operand, AStack<Character> operator) throws Exception {
    if (operand.size() < 2) {
        throw new Exception("missing operand");
    }
    boolean temp1 = char2Boolean(operand.pop());
    boolean temp2 = char2Boolean(operand.pop());
    char symbol = operator.pop();
    if (symbol == '|') {
        operand.push(boolean2Char(temp1 | temp2));
    } else {
        throw new Exception("operator mismatch");
    }
}

private static void xor(AStack<Character> operand, AStack<Character> operator) throws Exception {
    if (operand.size() < 2) {
        throw new Exception("missing operand");
    }
    boolean temp1 = char2Boolean(operand.pop());
    boolean temp2 = char2Boolean(operand.pop());
    char symbol = operator.pop();
    if (symbol == '^') {
        operand.push(boolean2Char(temp1 ^ temp2));
    } else {
        throw new Exception("operator mismatch");
    }
}
```

```
private static void and(AStack<Character> operand, AStack<Character> operator) throws Exception {
    if (operand.size() < 2) {
        throw new Exception("missing operand");
    }
    boolean temp1 = char2Boolean(operand.pop());
    boolean temp2 = char2Boolean(operand.pop());
    char symbol = operator.pop();
    if (symbol == '&') {
        operand.push(boolean2Char(temp1 & temp2));
    } else {
        throw new Exception("operator mismatch");
    }
}

private static void not(AStack<Character> operand, AStack<Character> operator) throws Exception {
    if (operand.size() < 1) {
        throw new Exception("missing operand");
    }
    boolean temp1 = char2Boolean(operand.pop());
    char symbol = operator.pop();
    if (symbol == '!') {
        operand.push(boolean2Char(!temp1));
    } else {
        throw new Exception("operator mismatch");
    }
}
```

括号的运算逻辑：

```
private static void calculateBracket(AStack<Character> operand, AStack<Character> operator) throws Exception {
    operator.pop(); // pop the right bracket
    rightCount--;
    while (operator.topValue() != '(') {
        choose2Execute(operator.topValue(), operand, operator);
    }
    operator.pop(); // pop the left bracket
    leftCount--;
}
```

4、运行结果展示

正常输入：

```
182     public static void main(String[] args) throws Exception {
183         System.out.println(calculate("T|(T&F)"));
184         System.out.println(true|(true&false));
185     }
```

```
Console  Debug Shell
<terminated> BooleanExpressionCalculate [Java Application] D:\Java\jdk-13\bin\javaw.
true
true
```

括号不匹配：

```
182     public static void main(String[] args) throws Exception {
183         System.out.println(calculate("T|(T&T)("));
184     }
```

```
Console  Debug Shell
<terminated> BooleanExpressionCalculate [Java Application] D:\Java\jdk-13\bin\javaw.exe (2020年11月6日 下午9:18:12)
Exception in thread "main" java.lang.Exception: bracket mismatch
        at com.pickupppp.task2.BooleanExpressionCalculate.calculate(BooleanExpressionCalculate.java:64)
        at com.pickupppp.task2.BooleanExpressionCalculate.main(BooleanExpressionCalculate.java:183)
```

```
182    public static void main(String[] args) throws Exception {
183        System.out.println(calculate("T|(T&T))"));
184    }
```

```
Console  Debug Shell
<terminated> BooleanExpressionCalculate [Java Application] D:\Java\jdk-13\bin\javaw.exe (2020年11月6日 下午9:18:50)
Exception in thread "main" java.lang.Exception: bracket mismatch
        at com.pickupppp.task2.BooleanExpressionCalculate.calculate(BooleanExpressionCalculate.java:25)
        at com.pickupppp.task2.BooleanExpressionCalculate.main(BooleanExpressionCalculate.java:183)
```

缺少运算符：

```
2    public static void main(String[] args) throws Exception {
3        System.out.println(calculate("T|(T&T)T"));
4    }
```

```
onsole  Debug Shell
minated> BooleanExpressionCalculate [Java Application] D:\Java\jdk-13\bin\javaw.exe (2020年11月6日 下午9:20:52)
eption in thread "main" java.lang.Exception: missing operator
    at com.pickupppp.task2.BooleanExpressionCalculate.calculate(BooleanExpressionCalculate.java:74)
    at com.pickupppp.task2.BooleanExpressionCalculate.main(BooleanExpressionCalculate.java:183)
```

缺少操作数：

```
    public static void main(String[] args) throws Exception {
        System.out.println(calculate("T|(T&T)&"));
    }
```

```
sole  Debug Shell
inated> BooleanExpressionCalculate [Java Application] D:\Java\jdk-13\bin\javaw.exe (2020年11月6日 下午9:21:26)
tion in thread "main" java.lang.Exception: missing operand
    at com.pickupppp.task2.BooleanExpressionCalculate.or(BooleanExpressionCalculate.java:101)
    at com.pickupppp.task2.BooleanExpressionCalculate.choose2Execute(BooleanExpressionCalculate.java:177)
    at com.pickupppp.task2.BooleanExpressionCalculate.calculate(BooleanExpressionCalculate.java:71)
    at com.pickupppp.task2.BooleanExpressionCalculate.main(BooleanExpressionCalculate.java:183)
```

5、总结和收获

关于括号的运算是该问题的一个难点，不仅在于要判断是否匹配，而且还要判断括号位置，否则会使运算结果错误。在各个操作符的行为中很多代码相似，可以按照操作符需要的操作数数量按照单目、双目、三目封装成相应的方法，以简化代码。

实验 3

1、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】

2、算法设计

问题 1 中先找出数值最大的数，然后计算位数，之后从个位开始入桶和回收；

问题 2 中字符串比较位数越低则影响权重越大，如 az<ba，按照这种思路从字符串末尾

的字符先进行入桶和回收；

问题 3 中 n 个队列映射到一个数组，则入队时判断(rear+n)%msize == front 来确定是否满，而出队时 front 坐标改为 front=(front+n)%msize;

3、主干代码说明

1）找最大值：

```
private static int getMaxValue(int[] array) { // Return the maximum number
    if (array.length == 0) {
        return -1;
    }
    int maxValue = array[0];
    for (int value : array) {
        if (value > maxValue) {
            maxValue = value;
        }
    }
    return maxValue;
}
```

计算位数：

```
private static int getNumLength(long num) {
    if (num == 0) {
        return 1;
    }
    int length = 0;
    for (long temp = num; temp != 0; temp = temp / 10) {
        length++;
    }
    return length;
}
```

基数排序：

```
public static int[] radixSort(int[] array) {
    int maxDigit = getNumLength(getMaxValue(array));
    int mod = 10;
    int dev = 1;
    AQueue<Integer>[] bucket = new AQueue[10];
    for (int i = 0; i < 10; i++) {
        bucket[i] = new AQueue<Integer>();
    }
    for (int i = 0; i < maxDigit; i++, dev *= 10, mod *= 10) {
        for (int j = 0; j < array.length; j++) {
            bucket[(array[j] % mod) / dev].enqueue(array[j]);
        }
        int pos = 0;
        for (AQueue<Integer> queue : bucket) {
            while (!queue.isEmpty()) {
                array[pos++] = queue.dequeue();
            }
        }
    }
    return array;
}
```

2)关于字符串的基数排序：

```
public static String[] radixSort(String[] array) {
    if (array.length == 0) {
        return null;
    }
    int times = array[0].length();
    AQueue<String>[] bucket = new AQueue[52];
    for (int i = 0; i < 52; i++) {
        bucket[i] = new AQueue<String>();
    }
    for (int i = times - 1; i >= 0; i--) {
        for (int j = 0; j < array.length; j++) {
            if (array[j].charAt(i) >= 65 && array[j].charAt(i) <= 90) {
                bucket[array[j].charAt(i) - 65].enqueue(array[j]);
            } else {
                bucket[array[j].charAt(i) - 71].enqueue(array[j]);
            }
        }
        int pos = 0;
        for (AQueue<String> queue : bucket) {
            while (!queue.isEmpty()) {
                array[pos++] = queue.dequeue();
            }
        }
    }
    return array;
}
```

4、运行结果展示

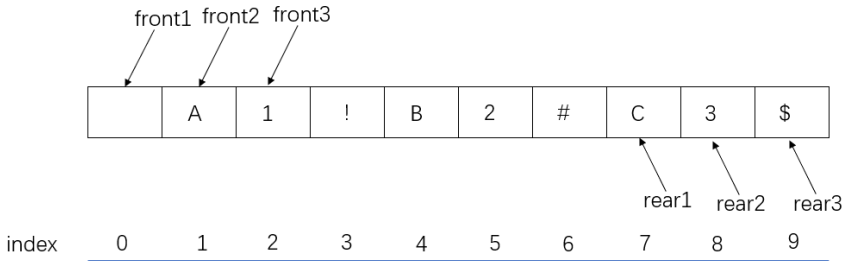
```
81    public static void main(String[] args) {
82        int[] arr = new int[] { 1, 21, 12, 322, 44, 123, 2312, 765, 56 };
83        radixSort(arr);
84        System.out.println(Arrays.toString(arr));
85
86        String[] arr1 = { "abc", "bde", "fad", "abd", "bef", "fdd", "abe" };
87        radixSort(arr1);
88        System.out.println(Arrays.toString(arr1));
89    }
90}
91
```

Console ⊠ Debug Shell

<terminated> RadixSort [Java Application] D:\Java\jdk-13\bin\javaw.exe (2020年11月6日 下午9:34:50)

```
[1, 12, 21, 44, 56, 123, 322, 765, 2312]
[abc, abd, abe, bde, bef, fad, fdd]
```

关于队列的映射如图：以 3 个为例说明



用三种不同字符表示三个队列的映射，每个队列单独维护自己的front和rear
front仍然表示队首元素的前趋位置
如果数组下标最大值正好能整除n则：
判断队列满：(rear + n) % msize == front
判断队列空：front == rear
若一个队列长度为m则数组长度应该为nm+1
如果存在数组不能平均分的情况，则应该先“补齐”缺少的格子
缺失的格子：nums = ((msize – 1) / n + 1) * n + 1 – msize
则判断队满：(rear + n) % (msize + nums) == front