|  |  |
| --- | --- |
| **数据结构与算法 作业报告** | |
| 第2次 | |
| 图片包含 标牌  已生成极高可信度的说明 | |
|  | |
|  | |
|  | |
| **姓名** |  |
| **班级** |  |
| **学号** |  |
| **电话** |  |
| **Email** |  |
| **日期** |  |

# 目录

### 任务一 指定的List ADT实现 2

1. 题目 2
2. 数据设计 2
3. 算法设计 3
4. 主干代码说明 4
5. 运行结果展示 6
6. 总结和收获 8

### 任务二 栈和递归之间的关系 8

1. 题目 8
2. 数据设计 8
3. 算法设计 9
4. 主干代码说明 9
5. 运行结果展示 12
6. 总结和收获 12

### 任务三 创建一个可自动调整空间大小的Queue 数据结构 13

1. 题目 13
2. 数据设计 14
3. 算法设计 14
4. 主干代码说明 15
5. 运行结果展示 17
6. 总结和收获 17

### 任务四 基数排序 17

1. 题目 17
2. 数据设计 17
3. 算法设计 18
4. 主干代码说明 18
5. 运行结果展示 19
6. 总结和收获 19

**附录：源代码 20**

# 任务一 指定的List ADT实现

### 题目

### 数据设计

1. 顺序数组表示需要存储的数据：一个顺序数组、数组大小、实际元素个数、光标位置；
2. 单向链表表示需要存储的数据：三个Link类型的指示符，分别指示头、光标、尾；
3. Link类需要存储的数据：数据本身、Link类型的指示符指示下一位；
4. 双向链表表示需要存储的数据：三个DoubleLink类型的指示符，分别指示头、光标、尾；
5. DoubleLink类需要存储的数据：数据本身、DoubleLink类型的指示符指示上一位和下一位；

### 算法设计

1. 顺序数组表示：
2. 插入：将光标以后的元素向后移动一位，在空出的位置插入元素；元素个数加一；
3. 删除：将光标以后的元素向前移动一位；元素个数减一；
4. 替换：将光标位置元素值替换为新值；
5. 清除：光标和元素个数重置为0；
6. 是否为空、是否为满：元素个数是否为0或等于数组大小；
7. 到头部、到尾部、向前、向后：对光标进行相应操作即可；
8. 输出：遍历数据表即可；
9. 单向链表表示：
10. 插入：判断表是否为空或在表尾；将当前元素的后继改为新元素，其中新元素的后继是之前元素的后继；
11. 删除：判断表是否删除为空、在表首或在表尾；将当前元素的前一个元素的后继改为当前元素的后继；移动光标到后继元素；
12. 替换：将光标位置元素值替换为新值；
13. 清除：重置光标、头和尾；
14. 是否为空：头元素是否与尾元素相等；
15. 是否为满：链表不会为满；
16. 到头部、到尾部、向后：对光标进行相应操作即可；
17. 向前：利用遍历找到当前光标的前一个元素；
18. 输出：遍历数据表即可；再次遍历以获取光标在表中的位置；
19. 双向链表表示与单向链表表示基本同理，但在插入、删除时需要考虑前趋，向前时无需进行遍历；
20. ListTest类采取Scanner和增强的switch来得到需要完成的操作；

### 主干代码说明

while(test.hasNextLine()){

    String[] operates=test.nextLine().split(" ");

    for (String operate : operates) {

        switch (operate) {

            case "-"-> aList.remove();

            case "#"-> aList.gotoBeginning();

            case "\*"-> aList.gotoEnd();

            case ">"-> aList.gotoNext();

            case "<"-> aList.gotoPrev();

            case "~"-> aList.clear();

            default-> {

                switch (operate.charAt(0)) {

                    case '+' -> aList.insert(operate.charAt(1));

                    case '=' -> aList.replace(operate.charAt(1));

                }

            }

        }

    }

    aList.showStructure();

    System.out.println(result.nextLine());

}

以上是在ListTest中实现测试的代码，利用了hasNextLine()和两个增强的switch，以实现每行一次输出、按照字符执行操作。

    public void insert(T newElement) {

        if(newElement!=null && !isFull()){*//元素不为空且数组未满*

            if(isEmpty()){arrayList[cursor]=newElement;}

            else {

                cursor++;

                for (int i = length; i > cursor; i--) {

                    arrayList[i] = arrayList[i - 1];

                }

                arrayList[cursor] = newElement;

            }

            length++;

        }

    }

    public void remove() {

        if(!isEmpty()){

            length--;

            for(int i=cursor;i<length;i++) {

                arrayList[i] = arrayList[i + 1];

            }

            if(cursor==length)cursor=0;

        }

    }

以上代码实现了顺序数组表示的插入和删除。

public class Link<T> {

    private T element;

    private Link<T> next;

    public Link(T element,Link<T> next){

        this.element=element;

        this.next=next;

    }

    public Link(Link<T> next){

        this.next=next;

    }

    public Link<T> next(){return next;}

    public void setNext(Link<T> next){this.next=next;}

    public T element(){return element;}

    public void setElement(T element){this.element=element;}

}

以上代码实现了一个Link类，DoubleLink类与之基本同理。

    public void insert(T newElement) {

        if(newElement!=null){

            if(isEmpty())cursor.setElement(newElement);*//表空*

            else if(cursor==tail){*//在表尾*

                cursor.setNext(new Link<>(newElement,null));

                gotoNext();

                tail=cursor;

            }

            else{

                cursor.setNext(new Link<>(newElement,cursor.next()));

                gotoNext();

            }

        }

    }

    public void remove() {

        if(!isEmpty()){

            if(!gotoPrev()){*//在表首*

                if(!gotoNext())clear();*//删除后表空*

                else head=cursor;

            }

            else if(cursor.next()==tail){*//删除后光标在尾*

                tail=cursor;

            }

            else{

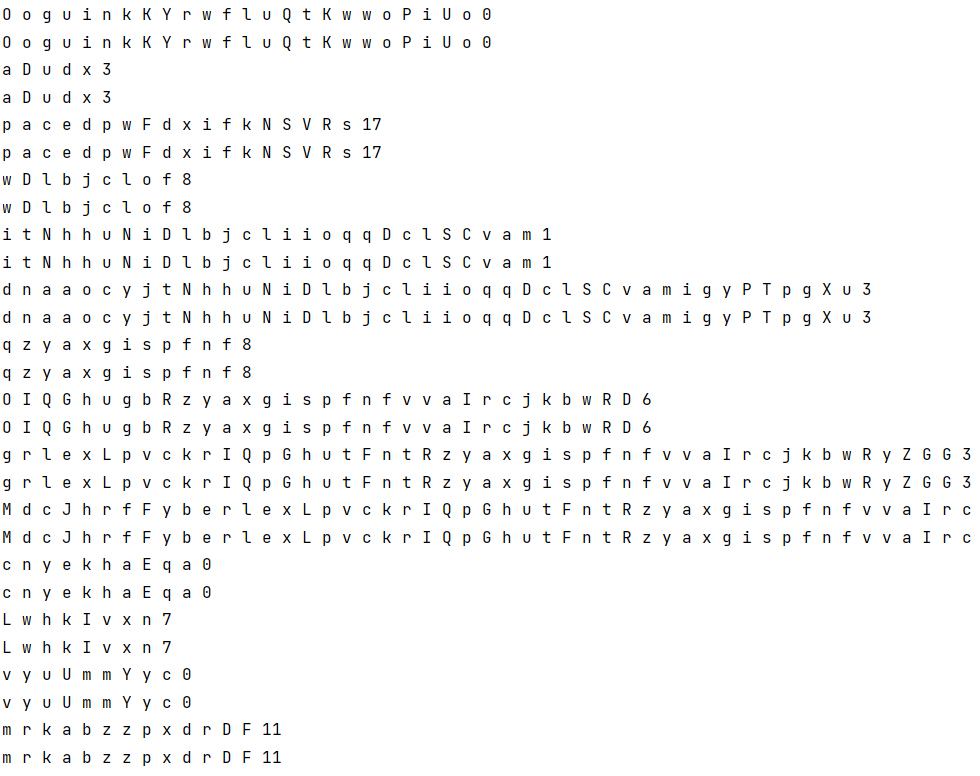
                cursor.setNext(cursor.next().next());

                gotoNext();

            }

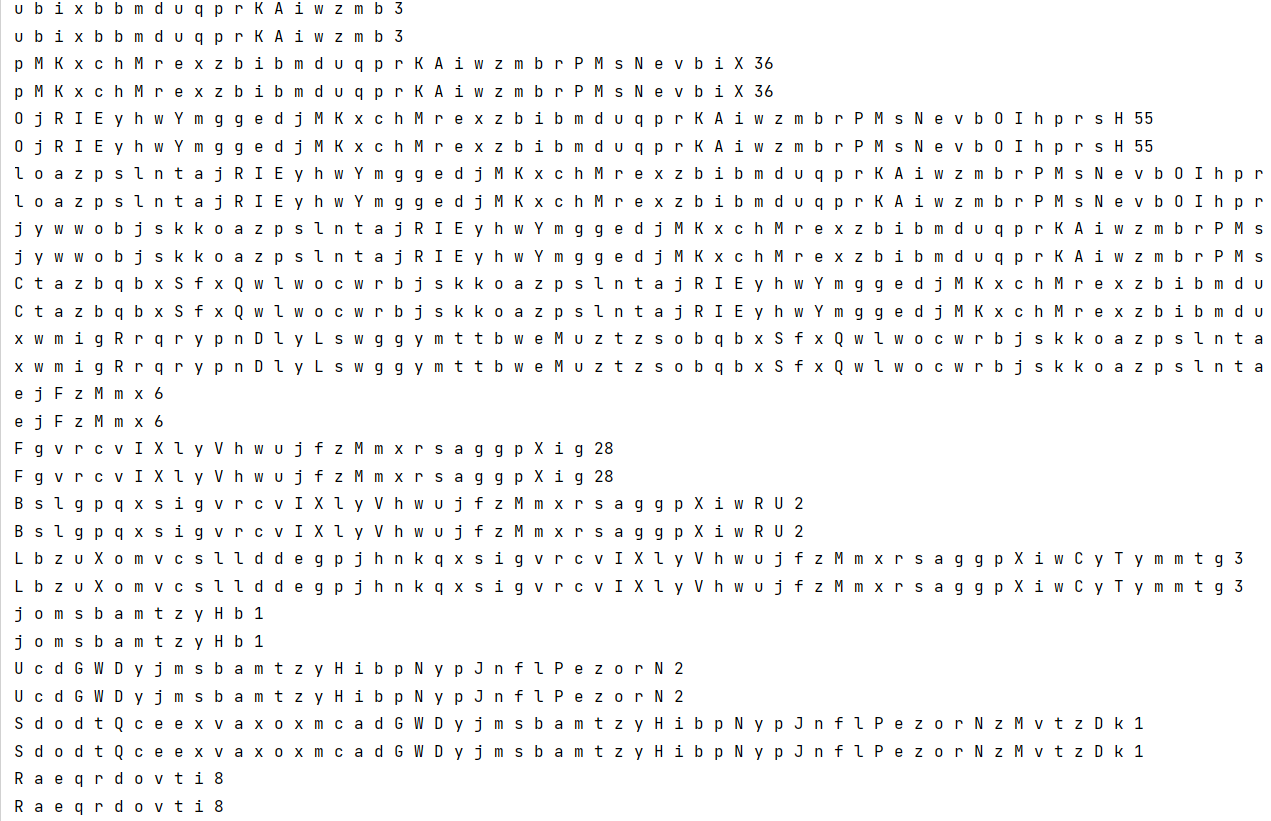
        }

    }

以上代码实现了单向链表表示的插入和删除，双向链表表示与之基本同理。

### 运行结果展示

运行得到的结果与list\_result.txt中的结果一致，说明该线性表可以正常运行。

每种数据结构下的所有方法的时间复杂度如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 数据结构 | 顺序数组 | 单向链表 | 双向链表 |
| Insert | O(n) | O(1) | O(1) |
| remove | O(n) | O(n)\* | O(1) |
| replace | O(1) | O(1) | O(1) |
| clear | O(1) | O(1) | O(1) |
| isEmpty,isFull | O(1) | O(1) | O(1) |
| gotoBeginning,gotoEnd,gotoNext | O(1) | O(1) | O(1) |
| gotoPrev | O(1) | O(n) | O(1) |
| getCursor | O(1) | O(1) | O(1) |
| showStructure | O(n) | O(n) | O(n) |

\*注：单向链表的remove方法的时间复杂度为O(n)的原因是因为这里的remove移除cursor指向的元素而不是cursor之后的元素，这导致remove方法必须调用gotoPrev来更改前一个元素的next值。

### 总结和收获

本题分别通过顺序数组、单向链表、双向链表三种方式实现了线性表，并比较了三种方式的方法的时间复杂度。三种方式的方法的时间复杂度依次递减，但使用的空间更多，在使用时需统筹考虑时间和空间。本题加深了对线性表的方法实现和不同实现方式的理解。

# 任务二 栈和递归之间的关系

### 题目

### 数据设计

数据输入后利用toCharArray转化为一个char数组输入给方法；递归调用时需要一个参数k控制当前位，初始值为0；非递归调用时，方法中需定义一个栈和数个操作，其中：

栈类包含的元素：栈的大小、栈顶位置、存储栈的数组；

操作类包含的元素：操作类的类型（交换或递归）、操作类的参数（交换类型2个，递归类型1个）

### 算法设计

将输出全排列的过程分解为三个过程：

1. 遍历当前位之后的位，将之与当前位交换；
2. 对当前位之后的位递归调用输出全排列；
3. 将当前位交换回原位；

即交换-递归-交换。其中，递归调用到最后时，当前位是数组的最后一位，此时将整个数组输出就是输出了一种排列。

根据该算法修改为非递归算法，可以通过定义一个操作类来实现；初始时将交换、递归、交换三种操作压入栈，之后开始弹栈，弹出交换操作时执行交换，弹出递归操作时，判断递归是否能够结束，若不能结束则将递归分解并再次压入栈；若递归能结束则输出整个数组。

题目要求的变形1只需要在递归调用前，检查当前位和遍历位是否相等，只有在当前位与遍历位重合或当前位和遍历位的值不相等时才执行；

题目要求的变形2需要给递归函数增加一个参数，并在每次递归时参数减一；递归结束的条件修改为该参数为零时，输出数组的前几位。

### 主干代码说明

递归算法如下：

    private static void permutationByRecursion(char[] array,int k){

        if(array.length-1==k){

            for(int i=0;i< array.length;i++){

                System.out.print(array[i]);

            }

            System.out.print(" ");

        }

        for(int i=k;i<array.length;i++){

            swap(array,i,k);

            permutationByRecursion(array,k+1);

            swap(array,i,k);

        }

    }

前半段控制递归结束，后半段是“交换-递归-交换”的算法。

非递归方法如下：

    private static void permutationByNoRecursion(char[] array){

        AStack<Operation> aStack=new AStack<>(100);

        for(int i=array.length-1;i>=0;i--){

            aStack.push(new Operation(MOVE,0,i));

            aStack.push(new Operation(OPERATION,1,0));

            aStack.push(new Operation(MOVE,0,i));

        }

以上是执行方法后先压入栈的操作；

        while (!aStack.isEmpty()){

            Operation operation = aStack.pop();

            if(operation.getType()==MOVE){

                swap(array, operation.getIndex1(), operation.getIndex2());

            }

            else if(operation.getType()==OPERATION){

                if(operation.getIndex1()==array.length-1){*//输出*

                    for(int i=0;i< array.length;i++){

                        System.out.print(array[i]);

                    }

                    System.out.print(" ");

                }

                else {

                    for(int i=array.length-1;i>=operation.getIndex1();i--){

                        aStack.push(new Operation(MOVE,operation.getIndex1(),i));

                        aStack.push(new Operation(OPERATION, operation.getIndex1()+1, 0));

                        aStack.push(new Operation(MOVE, operation.getIndex1(), i));

                    }

                }

            }

        }

    }

通过!isEmpty()控制不断循环，并执行交换操作和递归操作；递归达到结束条件时输出。其中栈的定义：

public class AStack<T> {

    private int size;

    private int top;

    private T[] arrayList;

    public AStack(int size){

        this.size=size;

        top=0;

        arrayList=(T[])new Object[size];

    }

    public void push(T newElement){

        arrayList[top++]=newElement;

    }

    public T pop(){

        return arrayList[--top];

    }

    public boolean isEmpty(){

        return top==0;

    }

}

仅实现了基本的pop、push、isEmpty操作供本题使用；

操作类的定义：

    int type;

    int index1;

    int index2;

    public Operation(int type,int index1,int index2){

        this.type=type;

        this.index1=index1;

        this.index2=index2;

    }

并实现了获取类型、获取参数的方法以供读取。

递归变种1添加的条件：

 for(int i=k;i<array.length;i++){

            if(array[i]!=array[k]||i==k) {*//只有二者不相等或二者下标相同时调用*

                swap(array, i, k);

                permutationVariant1(array, k + 1);

递归变种2的修改：

if(m==0){

            for(int i=0;i<k;i++){

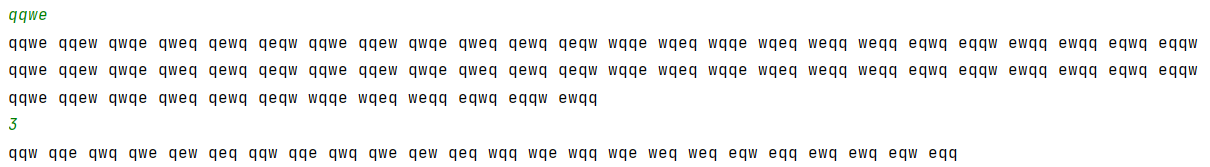
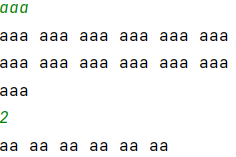
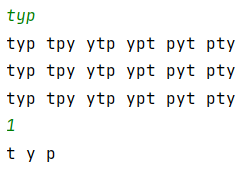
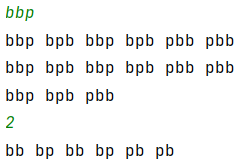
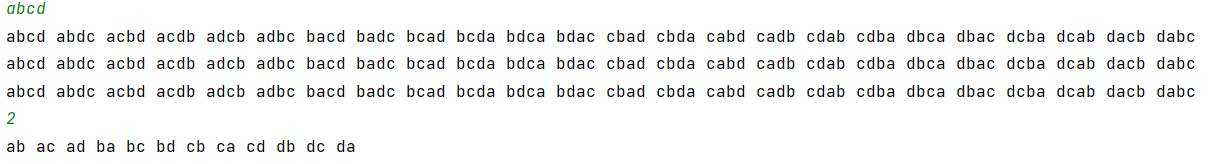
                System.out.print(array[i]);

            }

...

permutationVariant2(array, k + 1,m-1);

### 运行结果展示



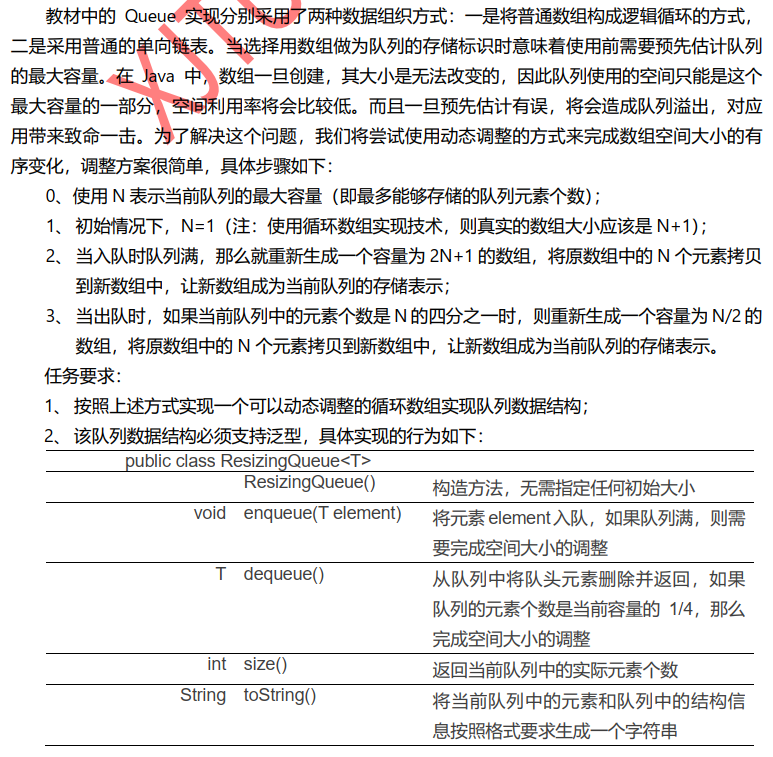
由上至下四个输出分别为递归、非递归、变形1、变形2；

### 总结和收获

该任务中将全排列分解为交换-递归-交换来实现，是仿照汉诺塔问题的分解进行的，使我进一步理解了汉诺塔问题的分解；通过定义操作类和栈，模拟递归的过程，从而实现了将算法转换为非递归方法，增进了对栈的定义、栈的使用的认识；实质上任何递归算法在系统内执行的本质都是栈，都可以通过定义栈转化为非递归方法；最后，通过控制递归进入的条件实现了不重复的全排列和指定位数的全排列，体会到了递归算法中约束条件的作用。

# 任务三 创建一个可自动调整空间大小的Queue 数据结构

### 题目



### 数据设计

本题需要实现队列数据结构，则ResizingQueue<T>类中需要存储的数据有：

1. 当前队列的最大容量N；
2. 当前队列的头和尾front和rear，分别存储队首元素的前趋位置和队尾元素的位置；
3. 用于存储数据的数组T[] arrayList；
4. 数组的长度length，length等于N+1；

### 算法设计

ResizingQueue<T>类中需要实现的方法有：

1. 构造方法：无需参数，初始N为1，front=rear=0；
2. 入队：若数组满，创建新数组、复制数据并重置front，rear，N，length的值；将队尾rear后移并将数据放入；
3. 出队：将队首front后移并返回出队元素；若数组内实际元素个数等于容量的四分之一，创建新数组、复制数据并重置front，rear，N，length的值；
4. 复制数据：遍历原数组的数据并将其复制至新数组的头部；
5. 获得元素个数：(rear-front+length)%length；
6. toString：定义函数getValues返回start到end的值以空格隔开；若数组大小小于等于20，获得整个数组的值并生成字符串；若数组大小大于20，获得前5和后5位的值并生成字符串；
7. getValues：遍历所给参数区间的值并生成字符串；

定义ResizingQueueTest类对ResizingQueue<T>进行测试。将test文件和result文件分别定义为两个Scanner，读取test文件中的内容并实现相应的操作。当读到”?”时，读取两行result文件中的内容与输出结果比较，判断输出的正确性。

### 主干代码说明

if(size()==N){//若数组满，创建新数组

                T[] arrayListNew = (T[])new Object[2\*N+1];

                copy(arrayList,arrayListNew);

                arrayList=arrayListNew;

                front=0;

                rear=N;

                N\*=2;

                length=N+1;

            }

            rear=(rear+1)%length;

            arrayList[rear]=element;//入队

}

入队主要操作如上，出队同理。创建新数组后利用copy方法复制数据并重置队首、队尾和大小。copy方法如下：

 private void copy(T[] arrayList,T[] arrayListNew){*//将原数组中的值复制进新数组*

        for(int i=1;i<size()+1;i++){

            arrayListNew[i]=arrayList[(front+i)%length];

        }

    }

以下实现题目所要求的对toString的重写：

public String toString(){

        String str="[";

        if(size()<=20){str+=getValues(1,size());}

        else {str=str+getValues(1,5)+"  ... "+getValues(size()-4,size());}

        str=str+"]\nelements: "+size()+" size:"+N;

        return str;

    }

private String getValues(int start,int end){*//获得从start位到end位的值（以字符串形式）*

        String valueStr="";

        for(int i=start;i<end;i++){

            valueStr=valueStr+arrayList[(front+i)%length]+" ";

        }

        valueStr+=arrayList[(front+end)%length];

        return valueStr;

    }

ResizingQueueTest类实现测试的代码如下：

 while (test.hasNext()){

     if(test.hasNextInt()){queue.enqueue(test.nextInt());}

     else{

         switch (test.next()) {

             case "-" -> queue.dequeue();

             case "?" -> {

                 String results = result.nextLine() + "\n" + result.nextLine();

                 System.out.println(results.equals(queue.toString()));

             }

         }

     }

 }

其中通过hasNextInt识别数字，通过switch识别不同操作。

### 运行结果展示

分别对test1000.txt和test5000.txt进行测试，结果均为true，即结果与result相符。

### 总结和收获

本题通过数组大小的自动调整实现了一个能够自动调整大小的队列，避免了因为大小限制造成的内存浪费和队列溢出。经测试，实现的队列能够完美完成不同数据规模的存储和相关操作。通过编写程序，本题加深了我对队列数据结构的实现的认识。

# 任务四 基数排序

### 题目

### 数据设计

利用Scanner读入文件中的数据并将int和String类型的数据存储在testInt数组和testString数组中，并记录下他们的大小以便排序使用。在排序方法中需要记录数据的最大位数，可能需要记录int数组中的最大值。排序方法中还需要一个队列的数组ResizingQueue<Integer>[] temp，以实现基数排序算法中的相关内容。

### 算法设计

main函数中输入数据、存储数据、调用排序方法并输出；

由于具体实现差异，分别实现对int的基数排序和对String的基数排序：

1. radixSortInt中，需要先找到数组中的最大值并存储它的位数；此后初始化队列的数组；最后对于从后之前的每一位，将每个数录入该位数字对应的队列，并按顺序取出至原数组；
2. radixSortString中，只需将位数赋值，之后与int的排序同理，但对于字母需要采取转换的方式，将“A-Z”“a-z”转换为0-51，从而录入队列。

### 主干代码说明

以下代码实现了对String的基数排序：

public static void radixSortString(String[] list, int length) {

    int maxLength = 8;

    ResizingQueue<String>[] temp = new ResizingQueue[52];

    for (int i = 0; i < 52; i++) {*//队列的初始化*

        temp[i] = new ResizingQueue<>();

}

以上是一些定义和初始化过程；

for (int i = maxLength - 1; i >= 0; i--) {*//每一次循环对第i位进行排序*

    for (int j = 0; j < length; j++) {*//将字符串放入队列*

        char letter = list[j].charAt(i);

        int position = 0;*//根据第i位的字母判断放入哪个队列*

        if (letter >= 'A' && letter <= 'Z') position = letter - 'A';

      if (letter >= 'a' && letter <= 'z') position = letter - 'a' + 26;

        temp[position].enqueue(list[j]);

}

以上是将每个字符串按当前位放入队列；

        for (int k = 0, index = 0; k < 52; k++) {*//将队列中的数取出，第一个for循环遍历52个队列*

            while (temp[k].size() != 0) {*//第二个for循环遍历每个队列里的数*

                list[index] =  temp[k].dequeue();

                index++;

            }

        }

    }

}

以上是队列中的数取出放回数组中；对int的基数排序同理，但需要在开始时寻找最大值以获得最大位数。

### 运行结果展示

0 6 10 16 18 18 21 23 27 28 33 37 37 37 41 42 43 47 48 51 56 68 69 70 85 87 91 96 97 102 105 111 112 124 126 128 133 140 142 142 142 146 147 148 149 154 165 166 169 175 175 178 188 200 200 202 …… 1222540 1222578 1223702 1224240 1224891 1225525 1225955 1226036 1227416 1228735 1228800 1230213 1230231 1231992 1233870 1234264

ACgnsNit ADBJMszB ADRKabPw ADRKsblB ADjSFRsM AEppMxnC AGIWmdlX AKpTKsKE ALPQrDwG APOeBcfB AQdwNrjL ARzprbYP AUNkreFd AWtDOqOs AXQvcSbs AYOGHDOX AYbuimYA …… zbjNVRKU zbvBOAXn zcjHhskB zdROHpXm zdzRKwXO zeEAzNEa zelAQmtc zfrVOagj ziERtCDH zjDxuOGn zjhZTuac zjwiyEQc zthWBwXw zvrAVQls zwLTMxHN zyCbelhX zyknRWGz

排序结果过长，因此只能部分展示。

### 总结和收获

本题利用了任务三实现的队列数据结构，实现了对int和String的基数排序，加深了对队列数据结构的应用的理解。但实现的排序仍有缺陷，即排序方法未能实现统一，可以考虑使用一个Comparable接口类等方法进行改进，以实现方法的统一，扩充方法的适用范围。

# 附录 源代码

### 任务一 指定的List ADT实现

public interface List<T> {

    void insert(T newElement);

    void remove();

    void replace(T newElement);

    void clear();

    boolean isEmpty();

    boolean isFull();

    boolean gotoBeginning();

    boolean gotoEnd();

    boolean gotoNext();

    boolean gotoPrev();

    T getCursor();

    void showStructure();

}

public class ArrayList<T> implements List<T> {

    private int size;*//数组大小*

    private int length;*//实际元素个数*

    private int cursor;

    private T[] arrayList;

    public ArrayList(int size){

        this.size = size;

        length = 0;

        cursor = 0;

        arrayList = (T[]) new Object[this.size];

    }

    public void insert(T newElement) {

        if(newElement!=null && !isFull()){*//元素不为空且数组未满*

            if(isEmpty()){arrayList[cursor]=newElement;}

            else {

                cursor++;

                for (int i = length; i > cursor; i--) {

                    arrayList[i] = arrayList[i - 1];

                }

                arrayList[cursor] = newElement;

            }

            length++;

        }

    }

    public void remove() {

        if(!isEmpty()){

            length--;

            for(int i=cursor;i<length;i++) {

                arrayList[i] = arrayList[i + 1];

            }

            if(cursor==length)cursor=0;

        }

    }

    public void replace(T newElement) {

        if(!isEmpty()&&newElement!=null){

            arrayList[cursor]=newElement;

        }

    }

    public void clear() {

        length = cursor = 0;

    }

    public boolean isEmpty() {

        return length == 0;

    }

    public boolean isFull() {

        return length == size;

    }

    public boolean gotoBeginning() {

        if(!isEmpty()){

            cursor = 0;

            return true;

        }

        return false;

    }

    public boolean gotoEnd() {

        if(!isEmpty()){

            cursor = length-1;

            return true;

        }

        return false;

    }

    public boolean gotoNext() {

        if(cursor<length-1){

            cursor++;

            return true;

        }

        return false;

    }

    public boolean gotoPrev() {

        if(cursor>0){

            cursor--;

            return true;

        }

        return false;

    }

    public T getCursor() {

        return arrayList[cursor];

    }

    public void showStructure() {

        if(isEmpty())System.out.println("Empty list");

        else{

            for(int i=0;i<length;i++){

                System.out.print(arrayList[i]+" ");

            }

        }

        System.out.println(cursor);

    }

}

public class Link<T> {

    private T element;

    private Link<T> next;

    public Link(T element,Link<T> next){

        this.element=element;

        this.next=next;

    }

    public Link(Link<T> next){

        this.next=next;

    }

    public Link<T> next(){return next;}

    public void setNext(Link<T> next){this.next=next;}

    public T element(){return element;}

    public void setElement(T element){this.element=element;}

}

public class DoubleLink<T> {

    private T element;

    private DoubleLink<T> next;

    private DoubleLink<T> prev;

    public DoubleLink(T element,DoubleLink<T> next,DoubleLink<T> prev){

        this.element=element;

        this.next=next;

        this.prev=prev;

    }

    public DoubleLink(DoubleLink<T> next,DoubleLink<T> prev){

        this.next=next;

        this.prev=prev;

    }

    public DoubleLink<T> next(){return next;}

    public void setNext(DoubleLink<T> next){this.next=next;}

    public DoubleLink<T> prev(){return prev;}

    public void setPrev(DoubleLink<T> prev){this.prev=prev;}

    public T element(){return element;}

    public void setElement(T element){this.element=element;}

}

public class LinkList<T> implements List<T> {

    private Link<T> head;

    private Link<T> tail;

    private Link<T> cursor;

    public LinkList(){

        head=tail=cursor=new Link<>(null);

    }

    public void insert(T newElement) {

        if(newElement!=null){

            if(isEmpty())cursor.setElement(newElement);*//表空*

            else if(cursor==tail){*//在表尾*

                cursor.setNext(new Link<>(newElement,null));

                gotoNext();

                tail=cursor;

            }

            else{

                cursor.setNext(new Link<>(newElement,cursor.next()));

                gotoNext();

            }

        }

    }

    public void remove() {

        if(!isEmpty()){

            if(!gotoPrev()){*//在表首*

                if(!gotoNext())clear();*//删除后表空*

                else head=cursor;

            }

            else if(cursor.next()==tail){*//删除后光标在尾*

                tail=cursor;

            }

            else{

                cursor.setNext(cursor.next().next());

                gotoNext();

            }

        }

    }

    public void replace(T newElement) {

        if(!isEmpty()&&newElement!=null){

            cursor.setElement(newElement);

        }

    }

    public void clear() {

        head=tail=cursor=new Link<>(null);

    }

    public boolean isEmpty() {

        return head==tail;

    }

    public boolean isFull() {

        return false;

    }

    public boolean gotoBeginning() {

        if(!isEmpty()){

            cursor = head;

            return true;

        }

        return false;

    }

    public boolean gotoEnd() {

        if(!isEmpty()){

            cursor = tail;

            return true;

        }

        return false;

    }

    public boolean gotoNext() {

        if(cursor!=tail&&cursor!=null){

            cursor=cursor.next();

            return true;

        }

        return false;

    }

    public boolean gotoPrev() {

        if(cursor!=head&&cursor!=null){

            Link<T> temp=head;

            while (temp.next()!=cursor){

                temp=temp.next();

            }

            cursor=temp;

            return true;

        }

        return false;

    }

    public T getCursor() {

        return cursor.element();

    }

    public void showStructure() {

        if(isEmpty())System.out.println("Empty list");

        else{

            for(Link temp=head;temp!=tail;temp=temp.next()){

                System.out.print(temp.element()+" ");

            }

            Link temp=head;int i=0;

            while (temp!=cursor){

                temp=temp.next();

                i++;

            }

            System.out.println(i);

        }

    }

}

public class DoubleLinkList<T> implements List<T> {

    private DoubleLink<T> head;

    private DoubleLink<T> tail;

    private DoubleLink<T> cursor;

    public DoubleLinkList(){

        head=tail=cursor=new DoubleLink<T>(null,null);

    }

    public void insert(T newElement) {

        if(newElement!=null){

            if(isEmpty())cursor.setElement(newElement);*//表空*

            else if(cursor==tail){*//在表尾*

                cursor.setNext(new DoubleLink<>(newElement,null,cursor));

                gotoNext();

                tail=cursor;

            }

            else{

                cursor.setNext(new DoubleLink<>(newElement,cursor.next(),cursor));

                gotoNext();

            }

        }

    }

    public void remove() {

        if(!isEmpty()){

            if(!gotoPrev()){*//在表首*

                if(!gotoNext())clear();*//删除后表空*

                else head=cursor;cursor.setPrev(null);

            }

            else if(cursor.next()==tail){*//删除后光标在尾*

                tail=cursor;

                cursor.setNext(null);

            }

            else{

                cursor.setNext(cursor.next().next());

                cursor.next().setPrev(cursor);

                gotoNext();

            }

        }

    }

    public void replace(T newElement) {

        if(!isEmpty()&&newElement!=null){

            cursor.setElement(newElement);

        }

    }

    public void clear() {

        head=tail=cursor=new DoubleLink<>(null,null);

    }

    public boolean isEmpty() {

        return head.element()==null;

    }

    public boolean isFull() {

        return false;

    }

    public boolean gotoBeginning() {

        if(!isEmpty()){

            cursor = head;

            return true;

        }

        return false;

    }

    public boolean gotoEnd() {

        if(!isEmpty()){

            cursor = tail;

            return true;

        }

        return false;

    }

    public boolean gotoNext() {

        if(cursor!=tail&&cursor!=null){

            cursor=cursor.next();

            return true;

        }

        return false;

    }

    public boolean gotoPrev() {

        if(cursor!=head&&cursor!=null){

            cursor=cursor.prev();

            return true;

        }

        return false;

    }

    public T getCursor() {

        return cursor.element();

    }

    public void showStructure() {

        if(isEmpty())System.out.println("Empty list");

        else{

            for(DoubleLink temp=head;temp!=tail;temp=temp.next()){

                System.out.print(temp.element()+" ");

            }

            DoubleLink temp=head;int i=0;

            while (temp!=cursor){

                temp=temp.next();

                i++;

            }

            System.out.println(i);

        }

    }

}

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class ListTest {

    public static void main(String[] args) throws FileNotFoundException {

        LinkList aList = new LinkList();

        Scanner test=new Scanner(new File("list\_testcase.txt"));

        Scanner result=new Scanner(new File("list\_result.txt"));

        while(test.hasNextLine()){

            String[] operates=test.nextLine().split(" ");

            for (String operate : operates) {

                switch (operate) {

                    case "-"-> aList.remove();

                    case "#"-> aList.gotoBeginning();

                    case "\*"-> aList.gotoEnd();

                    case ">"-> aList.gotoNext();

                    case "<"-> aList.gotoPrev();

                    case "~"-> aList.clear();

                    default-> {

                        switch (operate.charAt(0)) {

                            case '+' -> aList.insert(operate.charAt(1));

                            case '=' -> aList.replace(operate.charAt(1));

                        }

                    }

                }

            }

            aList.showStructure();

            System.out.println(result.nextLine());

        }

    }

}

### 任务二 栈和递归之间的关系

import java.util.Scanner;

public class Permutation {

    private static final int MOVE = 1;

    private static final int OPERATION = 2;

    public static void main(String[] args){

        Scanner in = new Scanner(System.in);

        String input = in.next();

        permutationByRecursion(input.toCharArray(),0);

        System.out.println();

        permutationByNoRecursion(input.toCharArray());

        System.out.println();

        permutationVariant1(input.toCharArray(),0);

        System.out.println();

        permutationVariant2(input.toCharArray(),0,in.nextInt());

        System.out.println();

    }

    private static void swap(char[] array,int a,int b){

        char temp=array[a];

        array[a]=array[b];

        array[b]=temp;

    }

    private static void permutationByRecursion(char[] array,int k){

        if(array.length-1==k){

            for(int i=0;i< array.length;i++){

                System.out.print(array[i]);

            }

            System.out.print(" ");

        }

        for(int i=k;i<array.length;i++){

            swap(array,i,k);

            permutationByRecursion(array,k+1);

            swap(array,i,k);

        }

    }

    private static void permutationByNoRecursion(char[] array){

        AStack<Operation> aStack=new AStack<>(100);

        for(int i=array.length-1;i>=0;i--){

            aStack.push(new Operation(MOVE,0,i));

            aStack.push(new Operation(OPERATION,1,0));

            aStack.push(new Operation(MOVE,0,i));

        }

        while (!aStack.isEmpty()){

            Operation operation = aStack.pop();

            if(operation.getType()==MOVE){

                swap(array, operation.getIndex1(), operation.getIndex2());

            }

            else if(operation.getType()==OPERATION){

                if(operation.getIndex1()==array.length-1){*//输出*

                    for(int i=0;i< array.length;i++){

                        System.out.print(array[i]);

                    }

                    System.out.print(" ");

                }

                else {

                    for(int i=array.length-1;i>=operation.getIndex1();i--){

                        aStack.push(new Operation(MOVE,operation.getIndex1(),i));

                        aStack.push(new Operation(OPERATION, operation.getIndex1()+1, 0));

                        aStack.push(new Operation(MOVE, operation.getIndex1(), i));

                    }

                }

            }

        }

    }

    private static void permutationVariant1(char[] array,int k){

        if(array.length-1==k){

            for(int i=0;i< array.length;i++){

                System.out.print(array[i]);

            }

            System.out.print(" ");

            return;

        }

        for(int i=k;i<array.length;i++){

            if(array[i]!=array[k]||i==k) {*//只有二者不相等或二者下标相同时调用*

                swap(array, i, k);

                permutationVariant1(array, k + 1);

                swap(array, i, k);

            }

        }

    }

    private static void permutationVariant2(char[] array,int k,int m){

        if(m==0){

            for(int i=0;i<k;i++){

                System.out.print(array[i]);

            }

            System.out.print(" ");

            return;

        }

        for(int i=k;i<array.length;i++){

                swap(array, i, k);

                permutationVariant2(array, k + 1,m-1);

                swap(array, i, k);

        }

    }

}

public class AStack<T> {

    private int size;

    private int top;

    private T[] arrayList;

    public AStack(int size){

        this.size=size;

        top=0;

        arrayList=(T[])new Object[size];

    }

    public void push(T newElement){

        arrayList[top++]=newElement;

    }

    public T pop(){

        return arrayList[--top];

    }

    public boolean isEmpty(){

        return top==0;

    }

}

public class Operation {

    int type;

    int index1;

    int index2;

    public Operation(int type,int index1,int index2){

        this.type=type;

        this.index1=index1;

        this.index2=index2;

    }

    public int getType() {

        return type;

    }

    public int getIndex1() {

        return index1;

    }

    public int getIndex2() {

        return index2;

    }

}

### 任务三 创建一个可自动调整空间大小的Queue 数据结构

import java.util.NoSuchElementException;

public class ResizingQueue<T> {

    private int N;

    private int length;

    private int front;

    private int rear;

    private T[] arrayList;

    public ResizingQueue(){

        N = 1;

        length = N+1;

        front = rear = 0;

        arrayList = (T[])new Object[length];

    }

    public void enqueue(T element){

        if(element==null)throw new IllegalArgumentException("Element is null.");*//输入元素为空*

        else{

            if(size()==N){*//若数组满，创建新数组*

                T[] arrayListNew = (T[])new Object[2\*N+1];

                copy(arrayList,arrayListNew);

                arrayList=arrayListNew;

                front=0;

                rear=N;

                N\*=2;

                length=N+1;

            }

            rear=(rear+1)%length;

            arrayList[rear]=element;*//入队*

        }

    }

    public T dequeue(){

        if(size()==0)throw new NoSuchElementException("Queue is empty.");*//队列空*

        else{

            front=(front+1)%length;

            T dequeueObject=arrayList[front];*//出队*

            arrayList[front]=null;

            if(size()==N/4&&N>1){*//若数组内元素等于大小的四分之一，创建新数组*

                T[] arrayListNew = (T[])new Object[N/2+1];

                copy(arrayList,arrayListNew);

                arrayList=arrayListNew;

                front=0;

                rear=N/4;

                N/=2;

                length=N+1;

            }

            return dequeueObject;

        }

    }

    public int size(){ return (rear-front+length)%length; }

    public String toString(){

        String str="[";

        if(size()<=20){str+=getValues(1,size());}

        else {str=str+getValues(1,5)+"  ... "+getValues(size()-4,size());}

        str=str+"]\nelements: "+size()+" size:"+N;

        return str;

    }

    private void copy(T[] arrayList,T[] arrayListNew){*//将原数组中的值复制进新数组*

        for(int i=1;i<size()+1;i++){

            arrayListNew[i]=arrayList[(front+i)%length];

        }

    }

    private String getValues(int start,int end){*//获得从start位到end位的值（以字符串形式）*

        String valueStr="";

        for(int i=start;i<end;i++){

            valueStr=valueStr+arrayList[(front+i)%length]+" ";

        }

        valueStr+=arrayList[(front+end)%length];

        return valueStr;

    }

}

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class ResizingQueueTest {

    public static void main(String[] args) throws FileNotFoundException {

        ResizingQueue<Integer> queue = new ResizingQueue<>();

        Scanner test=new Scanner(new File("test1000.txt"));

        Scanner result=new Scanner(new File("result1000.txt"));

        while (test.hasNext()){

            if(test.hasNextInt()){queue.enqueue(test.nextInt());}

            else{

                switch (test.next()) {

                    case "-" -> queue.dequeue();

                    case "?" -> {

                        String results = result.nextLine() + "\n" + result.nextLine();

                        System.out.println(results.equals(queue.toString()));

                    }

                }

            }

        }

    }

}

### 任务四 基数排序

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class RadixSort {

    public static void main(String[] args) throws FileNotFoundException {

        Scanner test1 = new Scanner(new File("radixSort1.txt"));

        int[] testInt = new int[10000];

        int length1 = 0;

        for (; test1.hasNext(); length1++) {*//将数据读入数组*

            testInt[length1] = test1.nextInt();

        }

        radixSortInt(testInt, length1);*//排序*

        for (int i = 0; i < length1; i++) {*//输出*

            System.out.print(testInt[i] + " ");

        }

        System.out.println();

        Scanner test2 = new Scanner(new File("radixSort2.txt"));

        String[] testString = new String[10000];

        int length2 = 0;

        for (; test2.hasNext(); length2++) {*//将数据读入数组*

            testString[length2] = test2.next();

        }

        radixSortString(testString, length2);*//排序*

        for (int i = 0; i < length2; i++) {*//输出*

            System.out.print(testString[i] + " ");

        }

    }

    public static void radixSortInt(int[] list, int length) {

        int max = 0;

        for (int i = 0; i < length; i++) {*//寻找数组的最大值*

            if (list[i] > max) max = list[i];

        }

        int maxLength = (int) Math.log10(max) + 1;*//最大位数*

        ResizingQueue<Integer>[] temp = new ResizingQueue[10];

        for (int i = 0; i < 10; i++) {*//队列的初始化*

            temp[i] = new ResizingQueue<>();

        }

        for (int i = 0; i < maxLength; i++) {*//每一次循环对第i位进行排序*

            for (int j = 0; j < length; j++) {*//将数组中的数放入队列*

                temp[(int) (list[j] / Math.pow(10, i)) % 10].enqueue(list[j]);

            }

            for (int k = 0, index = 0; k < 10; k++) {*//将队列中的数取出，第一个for循环遍历10个队列*

                while (temp[k].size() != 0) {*//第二个for循环遍历每个队列里的数*

                    list[index] = temp[k].dequeue();

                    index++;

                }

            }

        }

    }

    public static void radixSortString(String[] list, int length) {

        int maxLength = 8;

        ResizingQueue<String>[] temp = new ResizingQueue[52];

        for (int i = 0; i < 52; i++) {*//队列的初始化*

            temp[i] = new ResizingQueue<>();

        }

        for (int i = maxLength - 1; i >= 0; i--) {*//每一次循环对第i位进行排序*

            for (int j = 0; j < length; j++) {*//将数组中的数放入队列*

                char letter = list[j].charAt(i);

                int position = 0;*//根据第i位的字母判断放入哪个队列*

                if (letter >= 'A' && letter <= 'Z') position = letter - 'A';

                if (letter >= 'a' && letter <= 'z') position = letter - 'a' + 26;

                temp[position].enqueue(list[j]);

            }

            for (int k = 0, index = 0; k < 52; k++) {*//将队列中的数取出，第一个for循环遍历52个队列*

                while (temp[k].size() != 0) {*//第二个for循环遍历每个队列里的数*

                    list[index] =  temp[k].dequeue();

                    index++;

                }

            }

        }

    }

}