数据结构与算法
实验报告

第 二 次

姓名	DBQDSS
班级	数学强基 22
学号	*****
电话	*****
Email	*****
日期	2023-12-22

目录

任务 1：实现线性表的基本操作

一、题目

在本次实验中， 主要完成的任务是：
1、基于线性表的两种存储结构（即顺序存储和链式存储）编程实现以下功能：
InitList(&L):初始化表。构造一个空的线性表。
DestroyList(&L):销毁表。销毁线性表L并释放线性表所占存储空间。
Length(L):求表长。返回线性表L的长度，即线性表L中元素的个数。
Empty(L):判空。若L为空表则返回True,否则返回Flase。
PrintList(L):输出表。将线性表的元素依次输出。
LocateList(L, e):按值查找操作，查找元素e在表L中的位置。
GetList(L, i):按位查找操作，查找第i个位置的元素的值。
ListInsert(&L, i, e):在表L的第i个位置插入元素e。
ListDelete(&L, i, &e):删除表L的第i个位置的元素，并用e返回删除元素的值。
2、要求对于每一种操作进行算法设计描述、实现描述及测试说明（须有程序运行截图）。
3、比较并说明上述操作在顺序存储和链式存储下的异同。

二、算法设计

算法设计细节部分与主干代码说明一起说明。

由于 Java 不直接支持引用参数，所以我将函数的参数进行了改进。

顺序存储：以数组作为数据存储的方式，此代码包含了错误处理，比如插入或删除操作越界，和查找为空的情况。

链式存储：以链表作为数据存储的方式。链表的操作主要涉及对节点的引用和解引用，以及遍历链表。插入和删除操作需要找到目标位置的前驱节点和当前节点，然后插入新的节点或断开对当前节点的引用。

三、主干代码说明

（一）顺序存储实现线性表

1、InitList

```java
package task1;

import java.util.ArrayList;
import java.util.List;

2 个用法
public class AList<E> {
    1 个用法
    private int defaultSize = 10; // 默认容量
    13 个用法
    private E[] list; // 数组，用于存放元素
    15 个用法
    private int size = 0; // 当前表长度

    // 初始化线性表
    0 个用法
    public void InitList() {
        this.list = (E[]) new Object[defaultSize];
        this.size = 0;
    }

    1 个用法
    public void InitList(int len) {
        this.list = (E[]) new Object[len];
        this.size = 0;
    }
```

初始化操作，构造数组变量 list，然后创建给定大小（此处为 20）的数组（此处设置默认大小 defaultSize 为 10）来存储线性表的元素，同时用 size 记录当前线性表的长度。

2、DestroyList

```
15      // 销毁线性表
        1 个用法
16      public void DestroyList() {
17          list = null;
18          size = 0;
19      }
```

销毁操作，我们只需将我们的数组变量设为 null，并将线性表长度重置为 0 即可。

3、Length

```
21      // 获取线性表长度
        2 个用法
22      public int Length() {
23          return size;
24      }
```

返回线性表的当前长度，直接返回我们记录的当前长度数值 size。

4、Empty

```
26      // 判断线性表是否为空
        2 个用法
27      public boolean Empty() {
28          return size == 0;
29      }
```

判断线性表是否为空，只需看线性表长度是否为 0，于是比较 size 与 0 的大小即可。

5、PrintList

```
31      // 打印表中元素
        3 个用法
32      public void PrintList() {
33          for(int i = 0; i < size; i++) {
34              System.out.print(list[i]);
35              System.out.print(" ");
36          }
37          System.out.println();
38      }
```

通过 for 循环遍历并打印每个元素。

6、LocateList

```
47      // 按值查找元素位置
        3 个用法
48      public List<Integer> LocateList(E e) {
49          List<Integer> positions = new ArrayList<>();
50          for (int i = 0; i < size; i++) {
51              if (list[i].equals(e)) {
52                  positions.add(i);
53              }
54          }
55          return positions;
56      }
```

①创建一个空的数组 positions 用于存放找到的索引。

②遍历数组 list 以找到所有值等于指定元素的元素。我们将数组的长度用作循环的结束条件，并在循环的每次迭代中检查当前元素是否等于我们正在查找的元素。若当前元素等于我们正在查找的元素，我们将其索引添加到我们在第一步中创建的索引列表中。

③在数组遍历结束后，我们返回索引数组。如果该数组为空（即我们在数组中没有找到指定的元素），则返回的列表中不会有任何元素。

7、GetList

```
50      // 按位查找元素
        2 个用法
51      public E GetList(int i) {
52          if(i >= 0 && i < size) {
53              return list[i];
54          }else {
55              System.out.println("查找位置无效");
56              return null;
57          }
58      }
```

按位查找，返回索引 i 处的元素。先检查 i 是否在合理范围内（0 到 size −1），如果在范围内则返回该处的元素，否则提示错误并返回 null

8、ListInsert

```
60      // 在指定位置插入元素
        2 个用法
61      public void ListInsert(int i, E e) {
62          if(i < 0 || i > size) {
63              System.out.println("插入位置无效");
64              return;
65          }
66          if(size == list.length) {
67              System.out.println("表已满，无法插入");
68              return;
69          }
70          for(int j = size; j > i; j--) {
71              list[j] = list[j-1];
72          }
73          list[i] = e;
74          size++;
75      }
```

插入元素，首先检查索引 i 是否在范围内（0 到 size），如果超出范围提示错误并返回。然后从 size 开始，将每个元素向后移一位，直到索引 i 的位置，将 e 插入到 i 的位置，然后增加 size。

9、ListDelete

```
77      // 删除指定位置元素
        1 个用法
78      public E listDelete(int i) {
79          if(i < 0 || i >= size) {
80              System.out.println("删除位置无效");
81              return null;
82          }
83          E deletedElement = list[i];
84          for(int j = i; j < size - 1; j++) {
85              list[j] = list[j+1];
86          }
87          size--;
88          return deletedElement;
89      }
90  }
```

删除元素，首先检查索引 i 是否在范围内（0 到 size−1），如果超出范围提示错误并返回 null。然后保存索引 i 的元素 e，从 i 开始，将每个元素向前移一位，直到最后，最后减少 size，然后返回 e。

10、测试代码（TestAList）

```java
package task1;

public class TestAList {
    public static void main(String[] args) {
        AList<Integer> myList = new AList<~>();
        myList.InitList( len: 20);

        // 检测初始化是否成功
        System.out.println("初始的线性表状态：");
        // 测试线性表是否为空
        System.out.println("线性表表是否为空？ " + myList.Empty());
        // 检测线性表长度
        System.out.println("线性表长为： " + myList.Length());

        // 测试插入函数（线性表从0开始）
        for(int i = 0; i < 9; i++) {
            myList.ListInsert(i,  e: i+1);
        }
        myList.ListInsert( i: 9,  e: 1);
        myList.ListInsert( i: 10,  e: 1);

        // 打印表的长度
        System.out.println("插入元素之后，线性表长为：" + myList.Length());

        // 打印表中元素
        System.out.println("线性表中的元素为： ");
        myList.PrintList();

        // 测试按值查找函数，查找值为5的元素位置
        System.out.println("值为5的元素的位置为： " + myList.LocateList( e: 5));
        System.out.println("值为1的元素的位置为： " + myList.LocateList( e: 1));
        System.out.println("值为15的元素的位置为： " + myList.LocateList( e: 15));

        // 测试按位查找函数，查找位置为7的元素值
        System.out.println("处于7号位的元素值为： " + myList.GetList( i: 7));
        System.out.println("处于15号位的元素值为： " + myList.GetList( i: 15));

        // 测试插入函数，位置为5插入值为10的元素
        myList.ListInsert( i: 5,  e: 10);
        System.out.println("在5号位后插入一个值为10的元素，线性表变为： ");
        myList.PrintList();

        // 测试删除函数，删除位置为8的元素
        System.out.println("8号位元素的值为： " + myList.listDelete( i: 8));
        System.out.println("删去8号位之后，线性表变为： ");
        myList.PrintList();

        // 销毁表
        myList.DestroyList();
        System.out.println("线性表在销毁后是否为空： " + myList.Empty());
    }
}
```

（二）链式存储实现线性表

1、InitList

```java
package task1;

import java.util.*;
2 个用法
public class LList<E> {
    12 个用法
    Node<E> head;
    9 个用法
    int size;

    // 初始化链表
    1 个用法
    public LList() {
        head = null;
        size = 0;
    }
```

初始化操作，我们创建一个空节点来作为链表的头节点，同时设置一个 size 变量记录当前链表的长度（即元素数）

2、DestroyList

```java
    // 销毁链表
    1 个用法
    public void DestroyList() {
        head = null;
        size = 0;
    }
```

销毁操作，我们将头节点设为 null，并将链表长度重置为 0 即可。

3、Length

```java
    // 返回链表长度
    1 个用法
    public int Length() {
        return size;
    }
```

返回链表的当前长度，直接返回我们记录的当前长度数值

4、Empty

```java
    // 判断链表是否为空
    2 个用法
    public boolean Empty() {
        return size == 0;
    }
```

判断链表是否为空，只需看链表长度是否为 0

5、PrintList

```java
    // 输出链表元素
    2 个用法
    public void PrintList() {
        Node<E> temp = head;
        while (temp != null) {
            System.out.print(temp.data);
            System.out.print(" ");
            temp = temp.next;
        }
        System.out.println();
    }
```

通过循环链表并打印每个节点的数据。

6、LocateList

```
41      // 按值查找元素位置
        4 个用法
42      public List<Integer> LocateList(E e) {
43          List<Integer> result = new ArrayList<>();
44          Node<E> temp = head;
45          int index = 0;
46          while (temp != null) {
47              if (temp.data.equals(e)) {
48                  result.add(index);
49              }
50              temp = temp.next;
51              index++;
52          }
53          return result;
54      }
```

（此函数规避了多个元素的值相同只输出第一个位置的情况）

初始化：

①创建一个空列表 result 来存储找到的索引。

②设定一个存储当前索引的变量 index。

③设定一个节点 temp 用来遍历，初始指向链表头部。

开始遍历链表：

①检查当前 temp 节点的元素值是否等于目标值，如果相等，则将当前的 index 加入到结果列表 result 中。

②temp 移动到下一个节点，同时将 index 增加 1。

遍历结束后，返回结果列表 result，它包含了所有等于目标值的元素位置。

7、GetList

```
56      // 按位查找元素
        1 个用法
57      public E GetList(int i) {
58          if (i < 0 || i >= size) {
59              throw new IndexOutOfBoundsException("索引超出范围!");
60          }
61          Node<E> temp = head;
62          for (int j = 0; j < i; j++) {
63              temp = temp.next;
64          }
65          return temp.data;
66      }
```

按位查找，返回索引 i 处的元素。先检查 i 是否在合理范围内（0 到 size−1），如果在范围内则循环到该位置并返回该处的元素，否则提示错误。

8、ListInsert

```
// 在第i个位置插入元素e
5 个用法
public void ListInsert(int i, E e) {
    if (i < 0 || i > size) {
        throw new IndexOutOfBoundsException("索引超出范围!");
    }
    Node<E> newNode = new Node<>(e);
    if (i == 0) {
        newNode.next = head;
        head = newNode;
    } else {
        Node<E> prev = head;
        for (int j = 0; j < i - 1; j++) {
            prev = prev.next;
        }
        newNode.next = prev.next;
        prev.next = newNode;
    }
    size++;
}
```

插入元素，首先检查索引 i 是否在范围内（0 到 size），如果超出范围提示错误并返回。然后创建一个新的节点和 e，并找到索引 i 的前一个位置的节点，将新节点插入到前一个节点和当前节点之间，然后增加 size。

9、ListDelete

```
// 删除第i个位置的元素，并返回其值
1 个用法
public E ListDelete(int i) {
    if (i < 0 || i >= size) {
        throw new IndexOutOfBoundsException("索引超出范围!");
    }
    Node<E> removedNode;
    if (i == 0) {
        removedNode = head;
        head = head.next;
    } else {
        Node<E> prev = head;
        for (int j = 0; j < i - 1; j++) {
            prev = prev.next;
        }
        removedNode = prev.next;
        prev.next = removedNode.next;
    }
    size--;
    return removedNode.data;
}
```

删除元素，首先检查索引 i 是否在范围内（0 到 size-1），如果超出范围提示错误并返回 null。然后找到索引 i 的前一个位置的节点和当前节点，将前一个节点的 next 指向当前节点的 next，从而断开对当前节点的引用，然后减少 size，然后返回当前节点的数据。

10、测试代码（TestLList）

```java
package task1;

import java.util.List;

public class TestLList {
    public static void main(String[] args) {
        LList<Integer> list = new LList<>();

        // 测试初始化
        System.out.println("初始化之后表是否为空? " + list.Empty());

        // 测试插入
        list.ListInsert( i: 0,  e: 1);
        list.ListInsert( i: 1,  e: 1);
        list.ListInsert( i: 2,  e: 2);
        list.ListInsert( i: 3,  e: 3);
        list.ListInsert( i: 4,  e: 2);

        System.out.println("在插入之后，线性表为：");
        list.PrintList();

        // 测试长度
        System.out.println("线性表长度为：" + list.Length());

        // 测试按值查找
        List<Integer> loc = list.LocateList( e: 2);
        System.out.println("线性表中值为2的元素是:" + loc);
        List<Integer> loc1 = list.LocateList( e: 1);
        System.out.println("线性表中值为1的元素是:" + loc1);
        List<Integer> loc2 = list.LocateList( e: 3);
        System.out.println("线性表中值为3的元素是:" + loc2);
        List<Integer> loc3 = list.LocateList( e: 4);
        System.out.println("线性表中值为4的元素是:" + loc3);

        // 测试按位查找
        int val = list.GetList( i: 1);
        System.out.println("线性表中位于1号位的元素为:" + val);

        // 测试删除
        int delVal = list.ListDelete( i: 1);
        System.out.println("将要从1号位删除的元素的值为:" + delVal);
        System.out.println("删除此元素之后，线性表变为:");
        list.PrintList();

        // 测试销毁
        list.DestroyList();
        System.out.println("销毁表之后，是否为空? " + list.Empty());
    }
}
```

四、运行结果展示

（一）顺序存储实现线性表

```
初始的线性表状态：
线性表表是否为空？ true
线性表长为： 0
插入元素之后，线性表长为：11
线性表中的元素为：
1 2 3 4 5 6 7 8 9 1 1
值为5的元素的位置为： [4]
值为1的元素的位置为： [0, 9, 10]
值为15的元素的位置为： []
处于7号位的元素值为： 8
查找位置无效
处于15号位的元素值为： null
在5号位后插入一个值为10的元素，线性表变为：
1 2 3 4 5 10 6 7 8 9 1 1
8号位元素的值为： 8
删去8号位之后，线性表变为：
1 2 3 4 5 10 6 7 9 1 1
线性表在销毁后是否为空： true
```

顺序存储实现成功。

（二）链式存储实现线性表

```
初始化之后表是否为空？ true
此时线性表长度为：0
在插入之后，线性表为：
1 1 2 3 2
线性表长度为：5
线性表中值为2的元素是:[2, 4]
线性表中值为1的元素是:[0, 1]
线性表中值为3的元素是:[3]
线性表中值为4的元素是:[]
线性表中位于1号位的元素为:1
将要从1号位删除的元素的值为:1
删除此元素之后，线性表变为：
1 2 3 2
销毁表之后，是否为空？ true
```

链式存储实现成功。

五、总结与收获

InitList(&L)：初始化线性表。在顺序存储和链式存储中都需要这个操作，但实现差异很大。在**顺序存储**中，需要分配一定大小的**连续内存空间**。而在**链式存储**中，只需创建一个指向 null 的头节点，使用**离散内存空间**即可。

DestroyList(&L)：销毁线性表。在**顺序存储**中，需要**彻底删除数组**，释放分配给它的所有内存。在**链式存储**中，需要**遍历链表并逐个删除节点**。

Length(L)：返回线性表长度。在顺序和链式存储中，这通常涉及到在实现数据结构时维护一个 size 或 length 标记。对于数组，可以直接返回 size；对于链表，需要遍历列表以计算它的长度。

Empty(L)：判断是否为空表。在两者均可以通过检查长度是否为 0 来实现。

PrintList(L)：输出表内容。在顺序和链式存储中，都可以通过循环来访问和打印所有元素，但在链式存储中，我们需要遵循链表的链接来访问每个元素。

LocateList(L, e)：在链表中查找元素 e 的位置。在**顺序存储**中，我们可以直接**遍历**数组来查找元素；在**链式存储**中，我们需要从头节点开始，**遍历**每个节点。

GetList(L, i)：查找特定位置的元素。使用**顺序存储**，可以**直接使用索引**获得元素。在**链式存储**中，需要从头节点开始，**遍历**到第 i 个节点。

ListInsert(&L, i, e)：在特定位置插入元素。在**顺序存储**中，需要**移动位置 i 之后的所有元素**以为新元素腾出空间；在**链式存储**中，我们**只需更改一些引用**即可，但需要遍历到第 i - 1 个节点。

ListDelete(&L, i, &e)：删除特定位置的元素。在**顺序存储**中，需要移**动位置 i 之后的所有元素**来填补空出的空位；在**链式存储**中，我们**只需更改一些引用**即可，但需要遍历到第 i-1 个节点。

结论：

顺序存储（数组）通常具有快速访问元素的优势，但在中间插入或删除元素时需要移动大量元素。而链式存储（链表）在插入和删除操作上更高效，但访问特定位置的元素需要更多时间，因为需要从头节点开始遍历。

任务 2：栈

子任务一：

一、题目：两种解决 Hanoi 塔问题的方法

众所周知， 栈虽然是一个操作受限的线性表，但是其用途却很广泛，栈的数据结构实现非常简单， 因此我们只从应用层面熟悉栈。请完成下面三个子任务。

①递归是一种解决很多复杂问题最简单的思想方法，而任何编程语言对递归程序的支持都是通过栈实现的。 请自行查阅资料理解 “Hanoi 塔”问题，通过递归编程的方式解决该问题，并尝试使用非递归的编程方法解决该问题。

二、算法设计

（一）递归方法

可以将 n 个盘子看作只有两个盘子：最底下的一个盘子（最大的）和上面的 n-1 个盘子（看作一个整体）。对于这两个“盘子”，我们完全可以按照汉诺塔的解决方式：

1、将上面的盘子（即 n-1 个盘子看作的那一个）从 A 柱移动到 B 柱，过程中使用 C 柱作为缓冲。

2、将剩下的那个盘子（也就是编号为 n 的盘子）直接从 A 柱移动到 C 柱。

3、将 B 柱的 n-1 个盘子移动到 C 柱，过程中使用 A 柱作为缓冲。

然后对于步骤 1 和步骤 3，递归地执行以上步骤，直到每一步的盘子数量为 1，也就是我们的递归结束条件。

在编写程序中，我们用三个栈 A,B,C 来模拟三个柱子，编写对应盘子的移动函数 move，使用递归的方式，在每一次调用递归函数 move 的时候都输出盘子移动的信息。

（二）非递归方法

n 个盘子的汉诺塔的非递归算法：

1. 将三根柱子按顺序排成品字型，若 n 为偶数，按顺时针方向依次摆放 A、B、C；若 n 为奇数，按顺时针方向依次摆放 A、C、B。
2. 把圆盘 1 从现在的柱子移动到顺时针方向的下一根柱。
3. 接着，把另外两根柱上可以移动的圆盘移动到新的柱上（事实上只有唯一的选择）。
4. 如果没有达到目标要求，则返回步骤 2。

三、主干代码说明

（一）递归方法 HanoiTower

```
1   package task2;                                                    ⚠4 ^ v
2
3   import java.util.Stack;
4
5   public class HanoiTower {
6
7       // 定义三个栈，分别表示三个塔
        3 个用法
8       Stack<Integer> A = new Stack<Integer>();
        2 个用法
9       Stack<Integer> B = new Stack<Integer>();
        2 个用法
10      Stack<Integer> C = new Stack<Integer>();
11
```

```java
    // 移动盘子的方法
    3 个用法
    public void move(int n, String from, String buffer, String to) {
        if (n <= 0) // 如果没有盘子需要移动，则直接返回
            return;
        move( n: n-1, from, to, buffer); // 把上面的 n-1 个盘子，借助 to，移动到 buffer 上
        moveTop(from, to); // 把最后一个盘子从 from 移动到 to
        move( n: n-1, buffer, from, to); // 最后把 buffer 上的 n-1 个盘子，借助 from，移动到 to 上
    }

    // 移动栈顶盘子的方法
    1 个用法
    private void moveTop(String from, String to) {
        int disc = 0;
        if (from.equals("A")) // 如果应该从A塔取盘子
            disc = A.pop(); // 从A塔的顶部取出盘子
        else if (from.equals("B")) // 如果应该从B塔取盘子
            disc = B.pop();  // 从B塔的顶部取出盘子
        else if (from.equals("C")) // 如果应该从C塔取盘子
            disc = C.pop();  // 从C塔的顶部取出盘子

        if (to.equals("A")) // 如果盘子应该放到A塔
            A.push(disc); // 把盘子放到A塔的顶部
        else if (to.equals("B")) // 如果盘子应该放到B塔
            B.push(disc);  // 把盘子放到B塔的顶部
        else if (to.equals("C")) // 如果盘子应该放到C塔
            C.push(disc);  // 把盘子放到C塔的顶部

        System.out.println(disc + "号盘：" + from + "-->" + to); // 打印盘子移动信息
    }

    // 汉诺塔的主方法
    1 个用法
    public void hanoi(int n) {
        for(int i = n; i >= 1; i--) {
            // 把所有的盘子按大小顺序（从小到大）放入A塔
            A.push(i);
        }

        move(n, from: "A", buffer: "B", to: "C"); // 通过递归移动把A塔的盘子移动到C塔
    }

    // 测试方法
    public static void main(String[] args) {
        HanoiTower h = new HanoiTower(); // 实例化HanoiTower类
        h.hanoi( n: 4); // 调用hanoi方法解决3个盘子的汉诺塔问题
    }
}
```

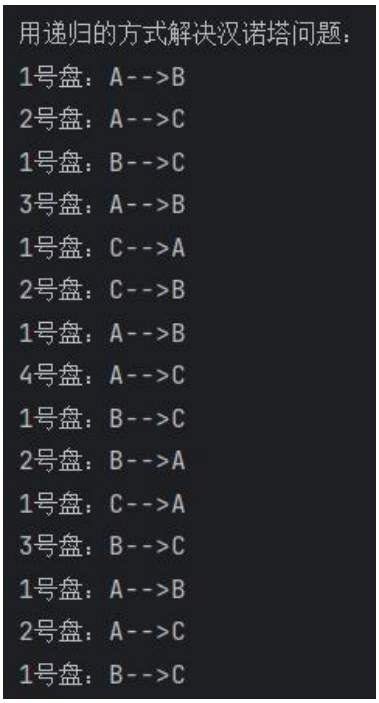
（二）非递归方法 HanoiTowerNonRecursive

```java
package task2;

import java.util.Stack;

public class HanoiTowerNonRecursive {
    // 定义从 '0','A','B','C' 对应的字符数组
    4 个用法
    private static char[] s = {'0', 'A', 'B', 'C'};
    // 创建一个堆栈数组来存放盘子
    14 个用法
    private static Stack<Integer>[] a = new Stack[4];
```

```
    // 汉诺塔的解决方法
    1 个用法
    private static void solveHanoiTower(int n) {
        // 依次将n个盘子从大到小压入第一个塔
        for (int i = n; i > 0; i--)
            a[1].push(i);
        // 当盘子数为奇数时，调整初始的塔的顺序
        if (n % 2 == 1) {
            s[2] = 'C';
            s[3] = 'B';
        }

        // 当所有的盘子都移动到目标塔时，结束循环
        while (true) {
            int next = 0;
            // 找到最小盘子并移到下一个塔
            for (int i = 1; i <= 3; i++)
                if (!a[i].isEmpty() && a[i].peek() == 1) {
                    next = i == 3 ? 1 : i + 1;
                    move(i, next);
                    break;
                }
            // 当所有盘子都移到了目标塔，则跳出循环
            if (a[2].size() == n || a[3].size() == n)
                break;

            // 记录非当前的两个塔
            int other1 = 0, other2 = 0;
            switch (next) {
                case 1: {
                    other1 = 2;
                    other2 = 3;
                    break;
                }
                case 2: {
                    other1 = 3;
                    other2 = 1;
                    break;
                }
                case 3: {
                    other1 = 1;
                    other2 = 2;
                    break;
                }
            }

            // 移动非当前塔中较小的盘子到较大的塔
            if (a[other1].isEmpty())
                move(other2, other1);
            else if (a[other2].isEmpty())
                move(other1, other2);
            else {
                if (a[other1].peek() < a[other2].peek())
                    move(other1, other2);
                else move(other2, other1);
            }
        }
    }
}
```

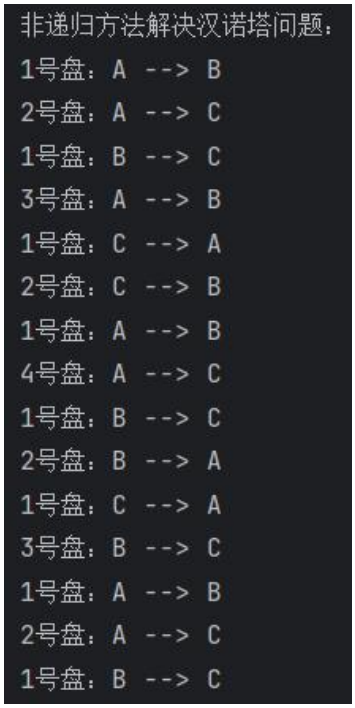
四、运行结果展示

（一）递归方法



以 4 为例，说明递归方法成功

（二）非递归方法



以 4 为例，说明递归方法成功

五、总结与收获

递归方法是最直观的解决办法，它直接模拟了汉诺塔问题的规则；非递归的方法则是用显式的数据结构栈来保存状态信息。用非递归的方法来实现会使程序变得复杂，但是，非递归的方法对于程序的调用栈没有依赖，因此在解决大规模问题时会有更好的性能。

在实现这个任务后，我认识到，在处理可递归问题时，我们可以尝试将问题的状态转化为递归程序，然后用非递归的方式去改编，因为非递归的方法更适合解决大规模问题，这种思想在未来的学习、工作中面对更大规模的问题时都会非常实用。

子任务二：

一、题目：非递归快速排序

②请在①的基础上进一步学习掌握递归问题的非递归转化方法，并实现对递归快速排序的非递归转化。

二、数据结构设计

数据结构是一个可比较元素的数组 Comparable[] objs。这种结构允许算法处理各种类型的数据，只要这些数据可以进行比较操作。

本题程序用一个内部类 Range 来记录当前待排序的序列的起始和终止位置。在每一次迭代中，从堆栈中 pop 出一个 Range 对象，这个 Range 对象表示当前待处理和待排序的序列。

三、算法设计

本实验的核心是将递归快速排序算法转化为非递归形式。

每经过一次遍历，就将 Rang 中的序列分为两个部分，其中一部分的所有元素都不大于另一部分的所有元素，并得到分区点 pivot。[pivot - 1, low] 与 [pivot + 1, high] 这两个子区间将会被创建为新的 Range 对象，然后被 push 到堆栈中。堆栈保证了最小的子区间首先被处理，这样可以尽快地释放空间。

这就说明可以通过使用栈（Stack<Range>）来模拟递归调用栈的方式实现。栈用于存储待排序的数组范围，模拟了递归过程中的调用和返回。

四、主干代码说明

（一）QuickSortOfRecursion 以及测试程序（主程序）

```java
package task2;

import java.util.Stack;
public class QuickSortOfRecursion extends SortAlgorithm {

    // 覆盖父类的sort方法，输入一个可比较对象的数组
    3 个用法
    @Override
    public void sort(Comparable[] objs) {
        // 创建一个范围的栈
        Stack<Range> stack = new Stack<>();
        // 将数组的索引范围入栈，索引开始为0，结束为数组长度-1
        stack.push(new Range( low: 0,  high: objs.length - 1));

        // 当栈不为空时执行循环
        while (!stack.isEmpty()) {
            // 弹出栈顶的范围项
            Range range = stack.pop();

            // 在索引起始值小于终止值时
            if(range.low < range.high){
                // 对数组范围内的元素进行分区，并返回新的基准点索引
                int pivot = partition(objs, range.low, range.high);
                // 将基准点左侧的范围入栈
                stack.push(new Range(range.low, high: pivot - 1));
                // 将基准点右侧的范围入栈
                stack.push(new Range( low: pivot + 1, range.high));
            }
        }
    }
```

```java
    // 将数组的一部分以对象高位为划分点分为两部分，返回分区后的基准点索引
    1 个用法
    private int partition(Comparable[] objs,int low,int high){
        // 将最后一个对象作为基准点
        Comparable pivot=objs[high];
        int i=low-1;
        // 从索引 low 开始至 high-1结束，针对数组每个元素
        for (int j = low; j < high; j++){
            // 若当前元素小于等于基准点值
            if(less(objs[j], pivot) || equals(objs[j], pivot)){
                i++;
                // 交换 i 和 j 位置的元素
                exchange(objs, i, j);
            }
        }
        // 交换 i+1 和 high 位置的元素
        exchange(objs, i + 1, high);
        return i + 1;
    }

    // 定义Range静态内部类，保存索引范围
    5 个用法
    static class Range{
        // low, high, 分别代表底部和顶部
        4 个用法
        int low,high;
        // 构造函数，用于新建一个范围
        3 个用法
        public Range(int low,int high){
            this.high = high;
            this.low = low;
        }
    }

    // 测试
    public static void main(String[] args) {

        QuickSortOfRecursion sorter = new QuickSortOfRecursion();
        Comparable[] arr = {12, -7, 26, 5, -14, 30, 21, -9, 10, -28, 15, 23, -6, 19, -4};

        // 打印原始数组
        System.out.println("原始数组为："+"[12, -7, 26, 5, -14, 30, 21, -9, 10, -28, 15, 23, -6, 19, -4]");
        // 使用sorter对象对数组进行排序
        sorter.sort(arr);
        // 打印排序后的数组
        System.out.println("排序后的数组为：");
        sorter.show(arr);
    }
}
```

（二）SortAlgorithm 类

```java
package task2;                                                    14  1

3 个用法  1 个继承者
public abstract class SortAlgorithm {
    3 个用法  1 个实现
    public abstract void sort(Comparable[] objs);
    2 个用法
    protected void exchange(Comparable[] numbers, int i, int j){
        Comparable temp;
        temp = numbers[i];
        numbers[i] = numbers[j];
        numbers[j] = temp;
    }
    1 个用法
    protected boolean less(Comparable one, Comparable other){
        return one.compareTo(other) < 0;
    }
    1 个用法
    protected  boolean equals(Comparable one,Comparable other){
        return one.compareTo(other) == 0;
    }
```

```
    2 个用法
17 @    protected void show(Comparable[] numbers){
18          int N = numbers.length;
19          int line = 0;
20          for(int i = 0; i < N; i++){
21              System.out.printf("%s ", numbers[i]);
22              line++;
23              if(line % 20 == 0) System.out.println();
24          }
25          System.out.println();
26      }
        0 个用法
27 @    protected boolean isSorted(Comparable[] numbers){
28          int N = numbers.length;
29          for(int i = 0; i < N-1; i++)
30              if(numbers[i].compareTo(numbers[i+1]) > 0) return false;
31          return true;
32      }
33
34  }
```

五、运行结果展示

```
原始数组为：[12, -7, 26, 5, -14, 30, 21, -9, 10, -28, 15, 23, -6, 19, -4]
排序后的数组为：
-28 -14 -9 -7 -6 -4 5 10 12 15 19 21 23 26 30
```

说明非递归的快速排序算法完全正确

六、总结与收获

通过对子任务二的学习，我更深入地理解了快速排序的工作原理和其非递归版本的实现，同时也学习到了如何使用栈数据结构解决问题的技巧。

这个例子也证明了，通常，递归算法总是可以转换为使用栈的非递归算法，虽然在大多数情况下，递归方法更简洁，但是当递归深度非常大时，使用非递归的方法可以避免栈溢出。对于快速排序算法，非递归形式在整个排序过程中，通过交换元素的方式在原地进行排序，这也节省了额外的存储空间。

在这个过程中，我也加深了对快速排序算法理论的理解，也锻炼了自己的编程能力。

子任务三：

一、题目:双端顺序栈

③当我们在使用很多软件时都有类似“undo”功能，比如 Web 浏览器的回退功能、文本编辑器的撤销编辑功能。这些功能都可以使用 Stack 简单实现， 但是在现实中浏览器的回退功能也好，编辑器的撤销功能也好， 都有一定的数量限制。因此我们需要的不是一个普通的 Stack 数据结构，而是一个空间有限制的 Stack，虽然空间有限，但这样的 Stack 在入栈时从不会溢出，因为它会采用将最久远的记录丢掉的方式让新元素入栈， 也就是说总是按照规定的数量要求保持最近的历史操作。比如栈的空间是 5，当 a\b\c\d\e 入栈之后，如果继续让元素 f 入栈，那么栈中的元素将是 b\c\d\e\f。请设计一个满足上面要求的 LeakyStack 数据结构， 要求该数据结构的每一个操作的时间复杂度在最坏情形下都必须满足 O(1)。

二、数据结构设计

本实验的数据设计涉及到的主要是对有限大小的栈（LeakyStack）的实现。

LeakyStack 可以简单地用一个定长环形数组来实现。

通过这种设计，我们可以在常数时间内完成插入新元素和删除旧元素的操作，满足了题目的 O(1)时间复杂度的要求。此外，我们用构造函数中的 capacity 参数定义栈的固定容量。

三、算法设计

LeakyStack 的关键算法设计在于它的入栈（push）和出栈（pop）操作。

入栈（push）操作：在添加新元素时，首先更新的栈顶位置 top（通过模运算实现环形结构），然后在这个位置放置新元素；为了实现自堆底向上输出，还使用了 front 来标记。

出栈（pop）操作：返回并移除栈顶元素，同时更新栈顶位置。

四、主干代码说明

（一）LeakyStack 类

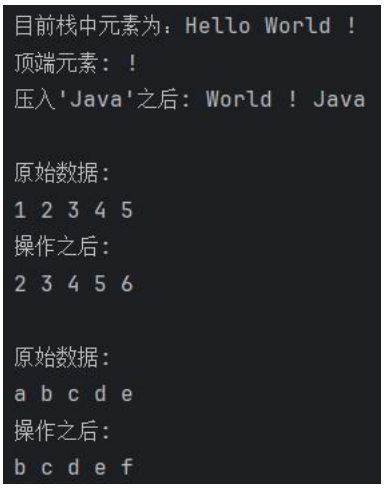
```
package task2;

6 个用法
public class LeakyStack <T>{
    7 个用法
    private T[] stack;
    16 个用法
    private int top;
    8 个用法
    private int capacity;
    9 个用法
    private int front;

    //构造函数
    4 个用法
    public LeakyStack(int capacity){
        this.capacity = capacity;
        stack = (T[]) new Object[capacity];
        this.top = -1;
        this.front = -1;
    }
    0 个用法
    public LeakyStack(){
        this( capacity: 5);
    }
```

```java
    //入栈
    16 个用法
    public void push(T element){
        if ((top + 1) % capacity == front)
            front = (front + 1) % capacity;

        top = (top + 1) % capacity;
        stack[top] = element;

        if (front == -1)
            front++;
    }

    //出栈
    0 个用法
    public T pop(){
        if(top == -1){
            System.out.println("Stack is Empty");
            return null;
        }
        T temp = stack[top];
        top = (top - 1 + capacity) % capacity;
        return temp;
    }

    // 查看栈顶元素
    1 个用法
    public T peek() {
        if (top == -1) {
            throw new IllegalStateException("Stack is empty");
        }

        return stack[top];
    }

    //判断栈空
    0 个用法
    public boolean isEmpty(){
        return top == -1;
    }

    //判断栈满
    0 个用法
    public boolean isFull(){
        return (top + 1) % capacity == 0;
    }

    //显示栈内元素
    6 个用法
    public void showStack(){
        if (front > top) {
            for (int i = front; i < capacity; i++)
                System.out.print(stack[i] + " ");
            for(int i = 0; i <= top; i++){
                System.out.print(stack[i]+ " ");
            }
            System.out.println();
        } else {
            for (int i = front; i <= top; i++)
                System.out.print(stack[i] + " ");
            System.out.println();
        }
    }

}
```

（二）测试类

```java
package task2;

public class Test {
    public static void main(String[] args) {
        // 创建一个字符串类型的 LeakyStack
        LeakyStack<String> stringStack = new LeakyStack<>( capacity: 3);
        stringStack.push( element: "Hello");
        stringStack.push( element: "World");
        stringStack.push( element: "!");
        System.out.print("目前栈中元素为：");
        stringStack.showStack();

        System.out.println("顶端元素: " + stringStack.peek());
        // 应该显示 "!" 75. 76. // 入栈新元素，这将导致最早的元素被移除
        stringStack.push( element: "Java");
        System.out.print("压入'Java'之后: " );
        stringStack.showStack();
        System.out.println();
        // 应该移除并返回 "World" 81. 82. System.out.println();

        // 创建一个整数类型的 LeakyStack
        LeakyStack<Integer> integerStack = new LeakyStack<>( capacity: 5);
        integerStack.push( element: 1);
        integerStack.push( element: 2);
        integerStack.push( element: 3);
        integerStack.push( element: 4);
        integerStack.push( element: 5);
        System.out.println("原始数据:");
        integerStack.showStack();
        integerStack.push( element: 6);
        System.out.println("操作之后:");
        integerStack.showStack();
        System.out.println();

        // 创建一个整数类型的 LeakyStack
        LeakyStack<Character> charStack = new LeakyStack<>( capacity: 5);
        charStack.push( element: 'a');
        charStack.push( element: 'b');
        charStack.push( element: 'c');
        charStack.push( element: 'd');
        charStack.push( element: 'e');
        System.out.println("原始数据:");
        charStack.showStack();
        charStack.push( element: 'f');
        System.out.println("操作之后:");
        charStack.showStack();
    }
}
```

五、运行结果展示



Leakystack 类实现成功

六、时间复杂度分析

由于本题的栈是使用数组实现的，因此对任意位置的读取操作时间复杂度都是 O(1)，我们开始对每个函数都进行检验：

（一）入栈

此操作首先更新 top、front 的值，这是通过取模运算实现的，是常数时间操作。

然后，将新元素放入计算出的位置。由于数组索引是直接访问的，这也是 O(1)操作。

（二）出栈

检查栈是否为空是常数时间操作。

访问栈顶元素并更新 top 指针也是 O(1)操作。

（三）查看栈顶元素

仅查看栈顶元素，不涉及任何循环或复杂操作，是 O(1)操作。

（四）判断栈空

此操作仅涉及检查 top 的值，是常数时间操作，也是 O(1)操作。

（五）判断栈满

这个方法检查栈是否已满，同样只涉及常数时间的计算，因此是 O(1)操作。

七、总结与收获

在实现 LeakyStack 类的过程中，我学会了如何设计并实现一个具有特定限制的数据结构，而且也加深了对环形数组与栈数据结构的理解。这种 LeakyStack 的设计在许多实际的应用中都非常有用。比如浏览器的回退功能和编辑器的撤销功能，在这些情境下常见的需求就是保持最近的操作，而旧的记录会被丢弃。

此外，这个实验还强调了在设计数据结构时考虑时间复杂度的重要性，确保每个操作都能在 O(1) 时间内完成。通过这个任务，我学习和理解了如何设计和实现一个满足特定需求的数据结构，这对于提高我的算法设计和编程技能非常有帮助。

任务 3：基数排序

一、题目

使用自定义的队列数据结构实现对某一个数据序列的排序(采用基数排序)，其中对待排序数据有如下的要求：

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"

注： radixsort1.txt 和 radixsort2.txt 是为上面两个数据序列提供的测试数据。

二、数据结构设计

主要数据结构：

自定义队列（Queue）：使用链表实现的队列，支持基本操作如入队（enqueue）和出队（dequeue）。

基数排序数组：整数数组和字符串数组，用于应用基数排序。

辅助数据结构：

桶（buckets）：在基数排序中，使用一个队列数组作为桶来存放特定位的数字或字符。

三、算法设计

整数排序：

根据每个数字的位数（从个位到最大位数），将数字放入相应的 0-9 桶中，然后按顺序收集这些桶中的数字。

字符串排序：

类似地，根据字符串的每个字符（从最后一个字符到第一个字符），将字符串放入相应的字母桶中，然后按顺序收集。

文件读取和写入：

读取：从文本文件读取整数或字符串序列。

写入：将排序后的数据写入指定文件。

四、主干代码说明

（一）链表节点类 Node

```
package task3;

1 个用法
public class Node<T> {
        1 个用法
        public T value;
        1 个用法
        public Node next;

        0 个用法
        public Node(T value) {
            this.value = value;
            this.next = null;
        }
}
```

（二）用链表实现队列操作

```java
package task3;

8 个用法
public class Queue <T>{
    7 个用法
    Node head,tail;
    2 个用法
    public Queue() {
        this.head = this.tail = null;
    }

    3 个用法
    public boolean isEmpty() {
        return head == null;
    }

    2 个用法
    public void enqueue(T value){
        Node temp = new Node<>(value);
        if(this.tail == null){
            this.head = this.tail = temp;
            return;
        }
        this.tail.next = temp;
        this.tail = temp;
    }


    2 个用法
    public T dequeue(){
        if(isEmpty()){
            System.out.println("This Queue is Empty");
            return null;
        }
        T temp= (T) head.value;
        this.head = this.head.next;
        if(head == null){
            this.tail = null;
        }
        return  temp;
    }
}
```

（三）RadixSort

```java
package task3;

import java.io.*;
import java.util.Scanner;

1 个用法
public class RadixSort {

    //数字数据基数排序
    1 个用法
    public static void radixSortForInteger(int[] arr) {

        //判断最大数字是几位数字
        int max = Integer.MIN_VALUE;
        for (int value : arr) {
            if (value >= max) {
                max = value;
            }
        }
```

```java
        int maxlength = Integer.toString(max).length();


        //0~9 一个数字一个桶
        Queue<Integer>[] buckets = new Queue[10];
        for(int i = 0; i < buckets.length; i++)
            buckets[i] = new Queue<Integer>();


        //根据最大位数决定比较次数，从个位数开始比较，依次将其放入对应桶中
        for (int i = 0, n = 1; i < maxlength; i++, n = n * 10) {
            for (int value : arr) {
                //求对应位数应放入的桶
                int num = (value / n) % 10;
                buckets[num].enqueue(value);
            }
            int index = 0;
            for (Queue queue : buckets) {
                while (!queue.isEmpty()) {
                    arr[index] = (int) queue.dequeue();
                    index++;
                }
            }
        }

    }

    //将结果输出到指定文档中
    1 个用法
    public static void writeToFile1(int[] array, String filePath) throws IOException {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath))) {
            int index=0;
            for (int value : array) {
                writer.write( str: value +"  ");
                index++;
                if(index==10){
                    writer.write( str: "\n");
                    index=0;
                }
            }
        }
    }

    1 个用法
    public static void writeToFile2(String[] array, String filePath) throws IOException {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath))) {
            int index=0;
            for (String value : array) {
                writer.write( str: value +"  ");
                index++;
                if(index==10){
                    writer.write( str: "\n");
                    index=0;
                }
            }
        }
    }
```

```java
    //字符串数据基数排序
    1 个用法
    public static void radixSortForString(String[] strings ){

        //求取最大长度
        int maxlength=strings[0].length();

        //为大小写共 52 个字母创建桶
        Queue[] temp = new Queue[52];
        for(int i = 0; i < temp.length; i++)
            temp[i] = new Queue<String>();

        for (int i = 0; i < maxlength; i++) {
            for (String string : strings) {
                char c = string.charAt(maxlength - 1 - i);
                int num = c >= 'a' ? c - 'a' + 26 : c - 'A';
                temp[num].enqueue(string);
            }
            int index = 0;
            for (Queue queue : temp) {
                while (!queue.isEmpty()) {
                    strings[index] = (String) queue.dequeue();
                    index++;
                }
            }
        }

    }
```

```java
    //两个测试函数用来测试是否已经排序成功
    1 个用法
    public static boolean isSortedOfNums(int[] nums){
        for(int i = 0; i < nums.length - 1; i++){
            if(nums[i] > nums[i+1]){
                return false;
            }
        }
        return true;
    }
    1 个用法
    public static boolean isSortedOfStrings(String[] strings){
        for(int i = 0; i < strings.length - 1; i++){
            if(strings[i].compareTo(strings[i+1]) > 0){
                return false;
            }
        }
        return true;
    }

    //补上两个读取数据的函数
    1 个用法
    public static String[] getStrings(String filename) throws FileNotFoundException {
        Scanner in= new Scanner(new File(filename));
        StringBuffer sb=new StringBuffer();
        String temp=in.next();
        sb.append(temp);
        temp=in.next();
        while(!temp.equals("stop")){
            sb.append('\n');
            sb.append(temp);
            temp=in.next();
        }
        return sb.toString().split( regex: "\n");
    }


    1 个用法
    public static int[] getNums(String filename) throws FileNotFoundException {
        Scanner in=new Scanner(new File(filename));
        StringBuffer sb=new StringBuffer();
        String temp=in.next();
        sb.append(temp);
        temp=in.next();
        while(!temp.equals("stop")){
            sb.append('\n');
            sb.append(temp);
            temp=in.next();
        }
        int[] nums= new int[sb.toString().split( regex: "\n").length];
        for(int i = 0; i < nums.length; i++){
            nums[i]=Integer.parseInt(sb.toString().split( regex: "\n")[i]);
        }
        return  nums;
    }

}
```

（四）读取文件

```java
package task3;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Arrays;

import static task3.RadixSort.*;

public class Test {

    //文件地址
    public static void main(String[] rags) throws IOException {
        int[] nums = getNums( filename: "C:\\Users\\17519\\Desktop\\数据结构与算法\\李垚辰\\Lab2\\radixSort1.txt");
        String[] strings = getStrings( filename: "C:\\Users\\17519\\Desktop\\数据结构与算法\\李垚辰\\Lab2\\radixSort2.txt");
        radixSortForInteger(nums);
        writeToFile1(nums, filePath: "C:\\Users\\17519\\Desktop\\outputIntegers.txt");
        System.out.println(isSortedOfNums(nums) ?
                "The numbers have been sorted!" : "The numbers are not sorted!");
        radixSortForString(strings);
        writeToFile2(strings, filePath: "C:\\Users\\17519\\Desktop\\outputStrings.txt");
        System.out.println(isSortedOfStrings(strings) ?
                "The strings have been sorted!" : "The strings are not sorted!");
    }
}
```

五、运行结果展示

```
The numbers have been sorted!
The strings have been sorted!
```

整数排序后：

```
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  207  215  217  221
229  236  236  237  255  263  264  264  265  270
271  277  279  280  282  286  289  296  298  299
308  313  314  317  319  319  322  324  326  335
339  343  343  345  354  359  360  361  362  363
363  363  368  368  377  379  384  384  386  392
395  404  413  414  414  418  419  435  436  438
439  442  444  445  447  449  457  457  463  466
468  473  476  478  483  492  497  500  514  520
528  533  535  550  552  553  554  560  563  567
574  575  575  576  577  588  590  600  603  614
615  617  618  637  638  643  643  653  654  657
657  660  664  668  671  692  695  714  720  724
731  735  736  737  743  744  746  753  753  759
764  771  783  786  792  798  798  800  802  803
805  813  814  824  829  838  843  862  865  869
874  875  877  881  882  882  887  891  891  895
900  902  908  909  909  912  913  916  936  936
937  939  944  946  947  951  953  961  964  967
967  968  972  974  975  975  977  985  985  985
990  997  998  1000  1002  1003  1017  1018  1019  1020
1031  1031  1033  1037  1041  1050  1054  1058  1063  1070
1077  1079  1082  1084  1084  1086  1087  1087  1089  1096
1097  1098  1100  1103  1103  1109  1114  1127  1130  1132
1138  1148  1155  1159  1161  1162  1164  1164  1174  1181
1185  1186  1186  1187  1189  1212  1213  1216  1219  1222
1224  1227  1228  1230  1231  1236  1239  1240  1241  1244
1274  1282  1284  1293  1295  1296  1297  1307  1309  1313
1321  1330  1332  1332  1339  1339  1342  1343  1345  1347
1349  1350  1359  1361  1363  1367  1380  1381  1381  1385
1395  1398  1404  1406  1407  1409  1412  1424  1426  1427
```

……

```
1084509  1084546  1086065  1086809  1087383  1087683  1088488  1088631  1089003  1089956
1090201  1090972  1092024  1092189  1092692  1092927  1093078  1094687  1095673  1097685
1100058  1100687  1101311  1102232  1103190  1103453  1103650  1104039  1105720  1108402
1110509  1111005  1111639  1113346  1113425  1114002  1114106  1115015  1118243  1118535
1118981  1122375  1122440  1124109  1124240  1124401  1124560  1124960  1125876  1128908
1128976  1129114  1129244  1129577  1129869  1130717  1132250  1133015  1134215  1134542
1136467  1137253  1137502  1138252  1138498  1138808  1139237  1140000  1140217  1141194
1142686  1145847  1146680  1147051  1147200  1147608  1148092  1148364  1148910  1149127
1149481  1149689  1150093  1150142  1150798  1152640  1152970  1153389  1153483  1153878
1153943  1154110  1156861  1161139  1161462  1162076  1162342  1163524  1164479  1164499
1164561  1164718  1165166  1165534  1167366  1168040  1169896  1170951  1171260  1173378
1173408  1174101  1176164  1177414  1178092  1178337  1179383  1179499  1180574  1181343
1181623  1183479  1183684  1184024  1184391  1184917  1188641  1188971  1189185  1189582
1189753  1190630  1191226  1191454  1191564  1192946  1194653  1195465  1195705  1196882
1198179  1199314  1199819  1200850  1201068  1201257  1202775  1203380  1203540  1203580
1203599  1204609  1207368  1207500  1207957  1210087  1210693  1211032  1213413  1214466
1214714  1214914  1216408  1217347  1218801  1219176  1219773  1220457  1221077  1221546
1221558  1221831  1222154  1222410  1222540  1222578  1223702  1224240  1224891  1225525
1225955  1226036  1227416  1228735  1228800  1230213  1230231  1231992  1233870  1234264
```

字符串排序后：

ACgnsNit	ADBJMszB	ADRKabPw	ADRKsblB	ADjSFRsM	AEppMxnC	AGIWmdlX	AKpTKsKE	ALPQrDwG	APOeBcfB
AQdwNrjL	ARzprbYP	AUNkreFd	AWtDOqOs	AXQvcSbs	AYOGHDOX	AYbuimYA	AYvhbcNk	AanVzwts	AgYNuVcd
AlRpOQFn	AlocZkRl	AmpCpEzH	ApyjWZtA	AqcsZvdQ	AsasKDXt	AuAkCOZp	AyeMQadX	AzbxOQjz	AztMmxSg
BBoJEHQO	BCKKRHgK	BCVEDsin	BEeMSeCF	BFPClXne	BFsZzJdJ	BGbPBCFy	BHXvhnwU	BIaGgssZ	BIpcyQtc
BKthFwAh	BLSapdFF	BLpdDeyE	BLrxGkDU	BMSgeojg	BNDToivF	BNoQwkio	BPEewCuy	BQLpANqw	BRdegocn
BRvuBbJk	BSvePbmL	BVowzoaV	BXsNOPsc	BYExhMvV	BZROZEhw	BZWGzKsB	BZatOZIY	BcfSsGlI	BdldiavX
BeESGied	BeNiNNeP	BeYlGzDR	BebCMCFG	BfQXQrSq	BhdMwoqp	BjIpWzKO	BlMgPzth	BldNMuoT	BluZbTNF
BmAOdcVg	BmWJuUBp	BnJDKKnP	BoWLAtbX	BoqxeRfF	BpLKtynI	BqUXvyKK	BrOwZIQJ	BrljOwPU	BrwMILAt
BsMzNwgB	BsOLVCxt	BtIZCAIO	BtPvaoGP	BwKtXaWi	BzwqTzbP	CBRRnnMj	CHYPClDu	CKFFxeLy	CKzDQoKU
CNTvDfNQ	COfaEqsa	CQXrSMyS	CSNLVWRY	CWjPhsZI	CZGygfyD	CbjEjRiS	CbkthTGB	ChVrQQNP	ChYqdePX
CpUMDOkm	CsOL1PXE	CsReToVT	CtyoeBif	CuYqXlxc	CvVFDJGp	CxwtNrcj	CzUcMLbQ	CzYvaoji	DAieCqpw
DAwHxUII	DECHGGpr	DErGCsXW	DGEJHqLJ	DJAlTpBp	DJVEqGKh	DNEHJNNH	DOIzzAIY	DQpJSjfZ	DSDJEFTl
DTGEtGol	DUmmGmkV	DVIUADcP	DVrdTWAG	DYYJZWot	DZpvQiND	DcuwFuDM	DdXkQqWb	DdbKZMDY	DeQbsPHB
DfiAuKcs	DhBCTDdg	DhJkthfw	DhxNmoMR	DibxlsnC	DlnBLKaO	DmRlOlwm	DnQENUKB	DoMuQsvu	DoPLqppt
DoqJhoSA	DotyEsrg	DqMuGCER	DqfgvmYH	DrOcfQgB	DsAUykiF	DvTjTKqs	DxoBFntt	DyFEruoL	EBfyRheE
EBuAWzjK	EBzwtuMw	EDbqMzSi	EDdYGnPy	EDqtfbOB	EFMhqVlW	EHPZsHQd	EIfVfFvB	EJEphBCl	EKaMXHEY
EKcuHzqc	EOjntupu	EPVRgPsz	EQLOjpPF	ERPmqnsu	ETLeOJFX	ETLkqTdN	EUwmjxbc	EVJfyFIL	EVsWYzAH
EVzZBZzk	EYYjGsyl	EYgWhHAW	EYzBBkry	EaQGvbgP	EavcnujP	EbFGvKAK	EbhwbxKm	EdMuaVYD	EeVKdyDj
EgJSSaAA	EhWUEJTY	EiWwMEPB	ElqPuFvK	EmNnwxNS	EmWIwMDL	EmfZTcBY	EnIZuTza	EpJzRNVk	ErAfCDdQ
ErMskxET	EsCgaPrL	EuiVkpSF	EvUmGRDK	ExQSCFfk	ExTgEnDT	EyVBNyXU	EyXXtZNf	FCcrHYuC	FCwTVUAF
FDgoZVjN	FExMDbUb	FFDsrSzo	FFnKjQdk	FGIovVOW	FHtidLbg	FHtlpsYc	FHwTfrlZ	FIsIkULW	FKOgQcPo
FKyEBPkm	FMGzZAay	FMzPHWAd	FOlvRYDO	FOvLcbrt	FPahkMUe	FQXBjkpq	FXGzDnhz	FYbtPAtY	FZWZHTZd
FZYPLNzB	FZeedXhF	FZjCpirC	FarfTEoH	FcBidqcH	FdyOAIWu	FkQlOfbd	FlIabauB	FlUlMWKu	Flzxeyxy
FpKTiymA	FqRPxuNQ	FuiyaYQf	FvLybtNi	FvUMRdJb	FvzQlmFY	FxSEupcd	FzPLBzSG	FzYBLtbD	GCAEFLTk
GCDKlEdF	GCSBZHTy	GCaZWpgj	GFBsWBPD	GIQTmzyf	GJDFwmVn	GJJnbhEN	GKCrVZLk	GLJmJhMi	GMfKdNth
GOdXKHzs	GOtSKGMU	GRJDeiZB	GRhiGRFe	GSFReRbe	GWUyyJLR	GYcECmFf	GYskJRfk	GZoUZcdp	GbDmWrQS
GbTAtKSK	GcGcWuhr	GdOdMtZE	GeQlSTiv	GjLPfZHC	GkBrkYom	GkTaNKjv	GknhihhS	GmYBVDNc	GmiOEHXJ
GoNjvkPw	GocDeEWt	GqQzoPeT	GsLZqbvg	GsQvOpmv	GsXxYZng	GsqYNBiS	GvCchkeM	GvDeIxPA	GveBXjaT
GwGMPTAV	GwOvKxwx	GxAeJGwe	GxDrHrrw	GxlKpiuo	GxtFTxoz	GyjtDFzN	HArnkdEU	HDLDYAFB	HDSfOQOo
HIIKjvWQ	HISnaERR	HLSojusj	HMGRFufh	HPBUhuxS	HPdRBtMe	HQGuqCop	HRdwZnvS	HSuLGxxR	HTTfskeL
HVRgSQaT	HYCxLykW	HYGCVEPI	HYarFmDN	HZKTnVPH	HZUGIhGR	HaAHNCEq	HaxQWgji	HbHesScN	HbWKhIqv
HdSremou	HdTbOYhD	HfCZgMae	HfLvPYeo	HgJnRIuX	HhUHmOmy	HhkxyXgc	HijkUkta	HioUtMXo	HlaeevsN
HqOFVOuy	HraqLdVX	HsBcxRwD	HwCvDotb	HwSJCmrl	HyRiZEIz	IBwtUPPi	ICdJqTLw	IDOOVsnx	IJDRaglH
IJWvKNTy	ILQSxWMW	ILpRfpkF	INosAxVq	IPeWnUrX	IPhoYJHQ	IQhAVhRZ	ISfNgsgo	ITDwMXng	IUWzXXFI
IUsMJNmV	IWYDockX	IYDuHJTv	IYDyQiiu	IYsojrwE	IZDpTHfE	IbfAmXhx	IcgDWCoP	IeIJGRsv	IeQYwSxn
IfVwYlEG	IibBuPXg	IkFtfrdJ	ImOiEfnJ	IoaXfjBQ	Iozlztvx	IrTwtgYw	ItllJfXQ	IwQYmbJt	IxDHJHJm
IxSCgfqf	IxmGKrHW	IyCJOhSn	IyJqUcuQ	Iygmdxuj	JAhoYuro	JFXmNqxa	JGErhkkB	JGXBkRbF	JImqPIJC
JInBShBN	JJYOpAcz	JKEyjknM	JKsxhcDj	JNWLldtF	JNoiUpib	JQiqnxAm	JQmrgLxR	JRwFyqCG	JSuCIYBw
JVCzQbYy	JXQSwHqT	JYKibvSO	JYRcRTkT	JdugyOEc	JeDuPysx	JePJJcsB	JgttvGjy	JjIgbYQV	JkASbZOq
JnsnmVyu	JqIYoouT	JqchfsDZ	JrZmDQlI	JrpbqqRq	JuQJPScH	KABdlLvD	KBTZudej	KBfionGw	KCYVMLWH
KCiAPHIb	KCrCRFJv	KCuveSUF	KCwIQzZn	KDCLMZVm	KDIftNul	KDfdtRbK	KEgiBXDm	KFrJzBDe	KIQhGtMh
KJTeefJQ	KKCEzLpg	KKaXCtAX	KLFznhfw	KNAUOpnA	KRUEapns	KSCyQJnd	KTtGDeEK	KUQQZpDs	KVsUWRKL
KZBBbvUP	KZKqdiLz	KZgZrDMt	KaekRHwM	KbWIWBlI	KcLsAowQ	KdIYgrNk	KgaBokeb	KhiLIKlk	KiGrcjon
Kiwmnxqh	KkdztITw	KlGrcKDy	KoYzmNCh	KptuyFbk	KsIDAqcT	KuvxNKdC	KvZmrWEO	KxwWujTk	KyNcTDbb
KyPuyQGW	KzJQiHZu	LAaCKkJA	LCxFoNXR	LELMxJeP	LERApwij	LFeZAQCa	LHdjBxKG	LIfVkBcc	LIgKsqzC
LLAbZdjD	LLPhVQii	LOQWguBi	LQtmFDmf	LRNFSSph	LSuZTcdD	LVVzAQtE	LWbdDuzi	LWpNYDmQ	LZREDklK
LcvnjAaN	LdSLGYuc	LeGQlVsY	LgfMnLwx	LgssCyEj	LiZKtGwt	LjzJCWCN	LmZJZmRO	LmlpqFbd	LqJPSzWA
LrPhCkqt	LriOKwWM	LtLAVSTu	LvouSyGU	LvttCIcL	LyQHCJvl	LzjJfYmk	MAMORPiq	MEwolreb	MFGSUTKr
MFOErsyW	MHcWvkCu	MHvJiFsL	MLChYbry	MLRErnmE	MLaOubIB	MMISfRwv	MNicBQtn	MOyTcIQJ	MPgAQRlS
MTZePcsc	MTZgILeu	MUGoEwDZ	MUMCIUTf	MUMonznM	MXMqANRY	MahpYhZQ	MdTDmJEh	MeTltDlT	Mewlyncp
MfDcfEue	MgnScSIa	MhZYVDAe	MhlOPeyo	MiAFYFYt	MiUrCRwl	MkhnRFLl	MkndsLPC	MpIuSxUP	MphjxhFy
MqkzYvWV	MsmobpTm	MtXKibOE	MtjQXzUu	MuqISGbJ	MvbiGdaQ	MvgYAHVy	MxHMbtsQ	MyCfEGyX	MygtEtot
NCEycQhm	NDFbvreI	NDceLvlz	NFzQzzvw	NGaYRtgC	NHiAiYUA	NHsoNMMf	NIHxHivM	NJzYSmEL	NKUQhhWi

……

ybzBiXoT	yedDKlXK	yhhhUmFZ	yizYwiMC	ykEgUsRb	ympNuhHq	ynumxUjF	yqBvFgfO	yqNbJJvp	yqRzxmzw
ysbTEeic	ysualdEJ	ysycEUPZ	ytVxNvTu	yvAohEKA	yvFAczZZ	ywubSoqX	yxlkilHT	yyQtFXvm	yzthZHXm
yzuRrjUp	zAPlZGHC	zAhjQveW	zAlDCdbN	zBXpCARs	zDvJOoRI	zEizxSXB	zFKxUWQM	zFzCSckv	zIxEOCBQ
zJGuBHZt	zKgtGunP	zKjoVYWz	zKohVqQR	zLWHYmeQ	zLXFseML	zLcALmIx	zMMTWdhx	zOWHMzJD	zOnAJYqj
zQCRFyWz	zQPClTUb	zRLIouoj	zSNcqCDk	zTDZIquz	zUwIWSML	zVFWviLi	zWSvWMTz	zXBRmOfJ	zXWmZNrR
zYNPcfoS	zZfRiyBW	zZgkXqgt	zbjNVRKU	zbvBOAXn	zcjHhskB	zdROHpXm	zdzRKwXO	zeEAzNEa	zelAQmtc
zfrVOagj	ziERtCDH	zjDxuOGn	zjhZTuac	zjwiyEQc	zthWBwXw	zvrAVQls	zwLTMxHN	zyCbelhX	zyknRWGz

六、总结与收获

这个实验提供了实践自定义数据结构（如队列）和复杂排序算法（如基数排序）的机会. 通过这个题目，我们不仅学习了基数排序算法的原理和实现，还加深了对文件操作、数据结构和算法的综合应用的理解。此外，实验还提高了对不同数据类型处理的灵活性，以及如何根据数据特性选择合适的排序算法。通过这个实验，我们提升了编程技巧和解决实际问题的能力。