数据结构： 判别回文字符串 实验报告

毛子恒 李臻 张梓靖

2020 年 10 月 17 日

小组成员

班级：2019211309	姓名：毛子恒	学号：2019211397	分工：代码 文档
班级：2019211310	姓名：李臻	学号：2019211458	分工：测试 文档
班级：2019211308	姓名：张梓靖	学号：2019211379	分工：文档

目录

1 需求分析

1.1 题目描述

回文串：正序表示和逆序表示相同的字符串。
输入一个字符串，判断其是否是回文串。

1.2 输入描述

程序从标准输入中读入数据。输入一行一个字符串，以“#”结束。
输入满足字符串串的长度 $LEN \leq 10^6$

1.3 输出描述

程序向标准输出中输出结果。
输出分为两种情况：

1. 输入合法，程序正常运行结束。此时输出一行一个字符串“YES”或“NO”（不带引号），分别表示该字符串是/不是回文串。
2. 程序发生运行时错误，比如内存分配失败。此时程序没有输出。

1.4 样例输入输出

1.4.1 样例输入输出 1

【输入】

```
11121#
```

【输出】

```
NO
```

1.4.2 样例输入输出 2

【输入】

```
121#
```

【输出】

```
YES
```

1.4.3 样例输入输出 3

【输入】

```
kndwtgsynmvlrcqctfdcwwscfrdjdrvtyojjoytvrdjdrfcswwcdftcqcrlvmnysgtwdnk#
```

【输出】

```
YES
```

1.4.4 样例输入输出 4

【输入】

```
adllaspkbqoytsfxpxwnxvljdmzyyfnlbjnomgxcnsnadllaspkbqoytsfxpxwnxvljdmzyyfnlbjnomgxcnsn#
```

【输出】

```
NO
```

1.5 程序功能

程序判别输入的字符串是否是回文串。

2 概要设计

2.1 问题解决的思路

根据栈先进后出和队列先进先出的性质，输入时将元素同时入队、入栈，之后同时出队、出栈，由于出队顺序是正序，出栈顺序是逆序，只需要比较每次出队和出栈元素是否相同即可判断字符串是否是回文串。

此题中栈实现了初始化、判空、入栈、出栈、释放空间这五种操作；队列分别采用顺序和链式两种方式实现了初始化、判空、入队、出队、释放空间这五种操作。

2.2 栈的定义

```
//数据对象
typedef char ElemType;

typedef struct stack
{
   ElemType * top;
   ElemType * base;
   int stacksize;
} Stack;

/*
 * 操作：初始化栈，分配空间
 * 后件：s指向一个空栈
 */
void initStack(Stack * s);

/*
 * 操作：判断栈是否为空
 * 前件：s是一个栈
 * 后件：如果该栈为空，返回true；否则返回false
 */
bool isStackEmpty(Stack s);

/*
```

```
 * 操作：将数据元素入栈
 * 前件：s指向一个栈
 * 后件：如果入栈成功，item成为栈顶元素；如果入栈之前该栈已满，则重新分配空间
 */
void pushStack(Stack * s, ElemType item);

/*
 * 操作：获取栈顶元素
 * 前件：s是一个栈
 * 后件：如果该栈不为空，返回栈顶元素
 */
ElemType getStackTop(Stack s);

/*
 * 操作：栈顶元素出栈
 * 前件：s指向一个栈
 * 后件：如果该栈不为空，栈顶元素出栈，返回这个出栈的元素
 */
ElemType popStack(Stack * s);

/*
 * 操作：释放栈空间
 * 前件：s指向一个栈
 * 后件：释放该栈的空间
 */
void destroyStack(Stack * s);
```

2.3 队列的定义（顺序）

```
//数据对象
typedef char ElemType;

typedef struct queue
{
    ElemType * front;
    ElemType * rear;
    ElemType * base;
    int queuesize;
} Queue;

/*
 * 操作：初始化队列，分配空间
 * 后件：q指向一个空队列
 */
void initQueue(Queue * q);

/*
 * 操作：判断队列是否为空
```

```
 * 前件：q是一个队列
 * 后件：如果该队列为空，返回true；否则返回false
 */
bool isQueueEmpty(Queue q);

/*
 * 操作：将数据元素入队
 * 前件：q指向一个队列
 * 后件：如果入队成功，item成为队尾元素；如果入队之前该队列已满，则重新分配空间
 */
void pushQueue(Queue * q, ElemType item);

/*
 * 操作：队首元素出队
 * 前件：q指向一个队列
 * 后件：如果该队列不为空，队首元素出栈，返回这个出队的元素
 */
ElemType popQueue(Queue * q);

/*
 * 操作：释放队列空间
 * 前件：q指向一个队列
 * 后件：释放该队列的空间
 */
void destroyQueue(Queue * q);
```

2.4 队列的定义（链式）

```
//数据对象
typedef char ElemType;

typedef struct node
{
   ElemType item;
   struct Node * next;
} Node;

typedef struct queue
{
   Node * front, * rear;
} Queue;

/*
 * 操作：初始化队列
 * 后件：q指向一个空队列
 */
void initQueue(Queue * q);

```

```
/*
 * 操作：判断队列是否为空
 * 前件：q是一个队列
 * 后件：如果该队列为空，返回true；否则返回false
 */
bool isQueueEmpty(Queue q);

/*
 * 操作：将数据元素入队
 * 前件：q指向一个队列
 * 后件：如果入队成功，item成为队尾元素
 */
void pushQueue(Queue * q, ElemType item);

/*
 * 操作：队首元素出队
 * 前件：q指向一个队列
 * 后件：如果该队列不为空，队首元素出队，并且释放其占用的空间，返回这个出队的元素
 */
ElemType popQueue(Queue * q);

/*
 * 操作：释放队列空间
 * 前件：q指向一个队列
 * 后件：释放该队列的空间
 */
void destroyQueue(Queue * q);
```

2.5 主程序的流程

1. 初始化栈和队列
2. 输入，元素入栈和队列
3. 每次从栈顶和队首取出元素，并且比较是否相同，重复执行直到队列或栈为空
4. 输出
5. 释放空间

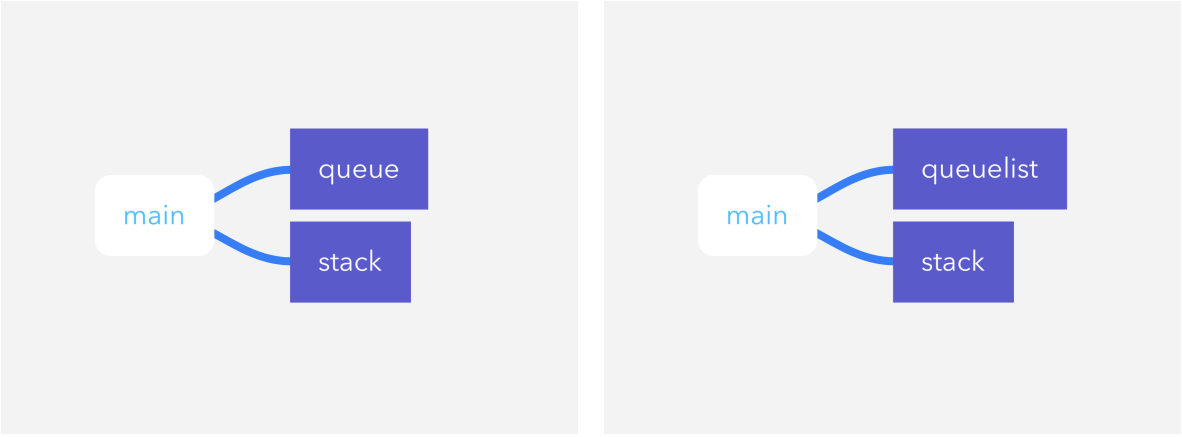
2.6 各程序模块之间的层次关系

程序模块层次关系图如图 1。

3 详细设计

3.1 栈的实现

栈设计中基本操作的伪代码算法如下：



(a) 栈和队列的顺序实现 (b) 栈和队列的链表实现

图 1: 程序模块层次关系

```
void Stack(Stack * s) // 初始化栈
{
    给*s分配内存
    if (*s内存分配失败) // 空间分配失败
        异常退出
    s->top <- s->base
    s->stacksize <- STACKINCREASESIZE // 初始栈空间为STACKINCREASESIZE
}

bool isStackEmpty(Stack s)// 判断栈是否为空
{
    if (s栈为空) 返回1
    else 返回0
}

void pushStack(Stack * s, ElemType item)// 将数据元素入栈
{
    if (s栈满)
    {
        分配给s更多的空间
        if (s空间分配失败)
            异常退出
        s->top <- s->base + s->stacksize
        s->stacksize <- s->stacksize + STACKINCREASESIZE
    }
    item入栈
}

ElemType popStack(Stack * s)// 栈顶元素出栈
{
    if (栈为空)
        返回异常
```

```
    返回 *(--s->top) // 返回栈顶元素，并且top减1
}

void destroyStack(Stack * s)// 释放栈空间
{
    释放s->base
    s->base <- NULL
    s->top <- NULL
    s->stacksize <- 0
}
```

3.2 队列的实现（顺序）

```
void initQueue(Queue * q) // 初始化队列，分配空间
{
    给q->base分配内存
    if (q->base 内存分配失败)
        异常退出
    q->front <- q->base //q指向一个空队列
    q->rear <- q->base
    q->queuesize <- QUEUEINCREASESIZE
}

bool isQueueEmpty(Queue q) // 判断队列是否为空
{
    if (q.front 和 q.rear 相等) 返回1
    else 返回0
}

void pushQueue(Queue * q, ElemType item) // 将数据元素入队
{
    if (队列满)
    {
        temp1 <- q->front - q->base
        分配给q->base更多的内存
        if (内存分配失败)
            异常退出
        q->front <- q->base + temp1
        q->rear <- q->base + q->queuesize
        q->queuesize <- q->queuesize + QUEUEINCREASESIZE
    }
    item入队
}

ElemType popQueue(Queue * q) // 队首元素出队
{
    if (*q队列为空)
        异常退出
```

```
    返回 *(q->front++)
}

void destroyQueue(Queue * q)// 释放队列空间
{
    释放q->base
    q->base <- NULL
    q->front <- NULL
    q->rear <- NULL
    q->queuesize <- 0
}
```

3.3 队列的实现（链式）

```
void initQueue(Queue * q)// 初始化队列
{
    定义newNode并为其分配内存
    if (newNode内存分配失败)
        异常退出
    q->front = q->rear <- newNode
    newNode->item <- 0 // 创建空的头结点
    newNode->next <- NULL
}

bool isQueueEmpty(Queue q) // 判断队列是否为空
{
    if (q.front 和 q.rear 相等) 返回1
    else 返回0
}

void pushQueue(Queue * q, ElemType item) // 将数据元素入队
{
    定义newNode并为其分配内存
    if (newNode内存分配失败)
        异常退出
    newNode->item <- item
    newNode->next <- NULL // 将newNode作为链表的尾元素
    q->rear->next <- newNode
    q->rear <- newNode // 将队列尾指向newNode
}

ElemType popQueue(Queue * q) // 队首元素出队
{
    nextNode 指向 q->front->next
    item <- nextNode->item // 出队的是队首指针的下一个元素
    释放q->front
    q->front <- nextNode
    返回 item
```

```
}

void destroyQueue(Queue * q)// 释放队列空间
{
    while (队列不为空)
        删除队列中剩余的节点
    释放q->front
    q->front <- NULL
    q->rear <- NULL
}
```

3.4 函数的调用关系图

函数调用关系图如图 2。

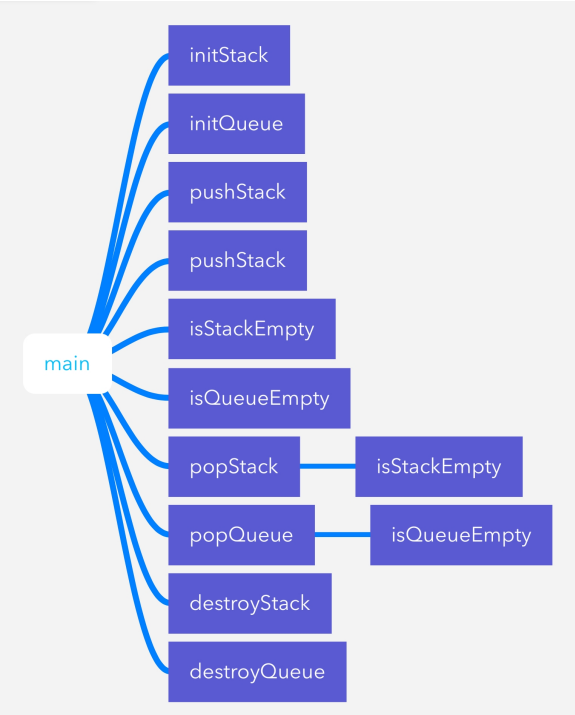


图 2: 函数调用关系图

4 调试分析报告

4.1 调试过程中遇到的问题和思考

调试过程中发现对于数据规模较大的情况，队列的顺序实现中重新分配内存后各指针的值需要更新，于是增加几个语句改正。

4.2 设计实现的回顾讨论

队列的链表实现中，$front$ 指针指向队首元素的前一个位置。

期望输入合法，所以对于输入末尾没有 # 的情况没有作处理。

由于不清楚测试用数据的范围大小，$STACKINCREASESIZE$,$QUEUEINCREASESIZE$ 常量的大小取到 100，也就是每次增加 100 个元素的空间。当输入很大时复杂度可能会比较大。

由于主函数对函数的调用足够严密，所以栈、队列的实现没有考虑不符合前件的情况。

4.3 算法复杂度分析

队列的链表实现中，destroyList 函数的时间复杂度为 $O(n)$。

其他函数的时间复杂度均为 $O(1)$。

主程序复杂度为 $O(n)$，整体时间复杂度为 $O(n)$。

整体空间复杂度为 $O(n)$。

4.4 改进设想的经验和体会

4.4.1 改进 1

使用数组实现的回文串判断 (testing/test.c)：

```c
#include <stdio.h>
#include <string.h>

const int MAXN = 1e6 + 10;
int n;
char a[MAXN];

int main()
{
    scanf("%s", a + 1);
    n = strlen(a + 1) - 1;
    for (int i = 1; i <= n / 2; ++i)
        if (a[i] != a[n - i + 1])
        {
            puts("NO");
            return 0;
        }
    puts("YES");
    return 0;
}
```

此程序用于测试第三部分。

5 用户使用说明

5.1 栈和队列的顺序实现

使用 gcc 编译生成可执行文件。

```
gcc -o main -std=c11 main.c stack.c queue.c
```

执行可执行文件：

```
./main
```

在 Windows cmd 下：

```
main
```

之后通过标准输入输入数据，输入格式参考 1.2 节的输入描述，结果通过标准输出返回。如果输入合法并且程序正常运行结束，主函数返回值为 0。

5.2 栈和队列的链表实现

使用 gcc 编译生成可执行文件。

```
gcc -o main -std=c11 main1.c stack.c queuelist.c
```

执行可执行文件：

```
./main
```

在 Windows cmd 下：

```
main
```

之后通过标准输入输入数据，输入格式参考 1.2 节的输入描述，结果通过标准输出返回。如果输入合法并且程序正常运行结束，主函数返回值为 0。

6 测试结果

本题的两种实现方式均通过以下测试。
测试环节分为三个步骤。

6.1 测试第一部分

对 1.4 节给出的样例进行测试。

6.2 测试第二部分

测试边界条件。
【输入】

```
#
```

（认为空串是回文串）
【输出】

```
YES
```

【输入】

```
1#
```

【输出】

```
YES
```

【输入】

```
00#
```

【输出】

```
YES
```

6.3 测试第三部分

将原解法与 4.4.1 节改进解法比对。

测试在 macOS Catalina 10.15.6 下进行。

在 $LEN <= 10$，$LEN <= 1000$，$LEN <= 1000000$ 的范围下分别随机生成 1000 组测试数据，分别传入 main 和 test，并且比对两程序的输出。

3000 组数据中两程序的输出均相同。

数据生成程序 (testing/data.cpp) 如下：

```cpp
#include <bits/stdc++.h>

using namespace std;

const int LEN = 1e6;
char a[LEN + 10];

int main()
{
    srand(time(0));
    int n = rand() % LEN + 1;
    for (int i = 1; i <= n / 2; ++i)
        printf("%c", a[i] = rand() % 26 + 'a');
    if (rand() % 2 == 1)
    {
        for (int i = 1; i <= n / 2; ++i)
            printf("%c", a[i]);
    }
    else
    {
        for (int i = n / 2; i >= 1; --i)
            printf("%c", a[i]);
    }
    puts("#");
    return 0;
}
```

比对脚本 (testing/chk.sh) 如下：

```
for i in {1..100}
do
   sleep 1
   ./data >in.in
   ./main <in.in >out.out
   ./test <in.in >out1.out
   if ! diff out.out out1.out
   then
      break
   fi
   echo "Correct"
done
```