数据结构：2/8 进制转换器 实验报告

毛子恒 李臻 张梓靖

2020 年 10 月 16 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

Contents

1 需求分析

1.1 题目描述

输入一串二进制数，输出其八进制表示。

1.2 输入描述

程序从标准输入中读入数据。输入一个 0/1 串表示二进制数，以“#”表示输入结束。
输入满足二进制串的长度 $LEN \leq 10^6$

1.3 输出描述

程序向标准输出中输出结果。
输出分为三种情况：

1. 输入合法，程序正常运行结束。此时输出一行一个八进制数。
2. 输入不合法，即输入中包含除了 0/1、空白字符、# 以外的其他字符，此时输出一行一个字符串”Please check your input.”（不带引号）。
3. 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】

```
110#
```

【输出】

```
6
```

1.4.2 样例输入输出 2

【输入】

```
010111111#
```

【输出】

```
277
```

1.4.3 样例输入输出 3

【输入】

```
0110001010#
```

【输出】

```
612
```

1.4.4 样例输入输出 4

【输入】

```
1101000110010000000101000101100100100000101110011011111111101000000101100010111110001010001110 0#
```

【输出】

```
150620050544413467772013057424 34
```

1.4.5 样例输入输出 5

【输入】

```
112000#
```

【输出】

```
Please check your input.
```

1.5 程序功能

程序将输入的二进制串转化为八进制并且输出。

2 概要设计

2.1 问题解决的思路

使用栈模拟操作过程。首先将所有二进制位依次入栈 $s1$。之后每次从 $s1$ 栈顶取出三个二进制位，将其转化为一个八进制位后入栈 $s2$，最后把 $s2$ 中元素依次出栈并输出。

此题中栈实现了初始化、判空、入栈、出栈、获取栈顶元素、释放空间这六种操作。

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：指向一个空栈s
 */
void initStack(Stack * s);

```

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

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

/*
 * 操作：获取栈顶元素
 * 前件：是一个栈s
 * 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：指向一个栈s
 * 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：指向一个栈s
 * 后件：释放该栈的空间
 */
void destroyStack(Stack * s);
```

2.3 主程序的流程

1. 初始化栈
2. 输入，元素入 $s1$ 栈
3. 每次从 $s1$ 中出栈三个元素，计算对应的八进制位并入 $s2$ 栈，重复执行直到 $s1$ 为空
4. 元素出 $s2$ 栈并输出
5. 释放空间

2.4 各程序模块之间的层次关系

模块调用关系图如图 1。

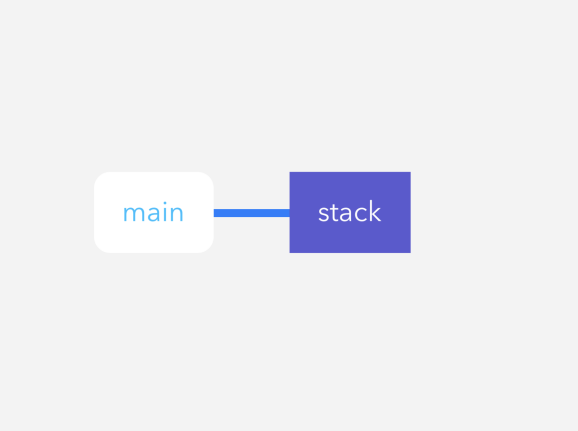

Figure 1: 模块的调用关系

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：

```
void Stack(Stack * s) // 初始化栈
{给
    *分配内存s
    if (*内存分配失败s) // 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 (栈为空s) 返回1
    else 返回0
}

void pushStack(Stack * s, ElemType item)// 将数据元素入栈
{
    if (栈满s)
    {分配给更多的空间
        s
        if (空间分配失败s)异常退出

        s->top <- s->base + s->stacksize
        s->stacksize <- s->stacksize + STACKINCREASESIZE
    }入栈
    item
}

ElemType getStackTop(Stack s)// 获取栈顶元素
```

```
{
    if 栈为空()返回异常返回

     *(s.top – 1) // 返回栈顶元素
}

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，并且减top1
}

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

3.2 函数的调用关系图

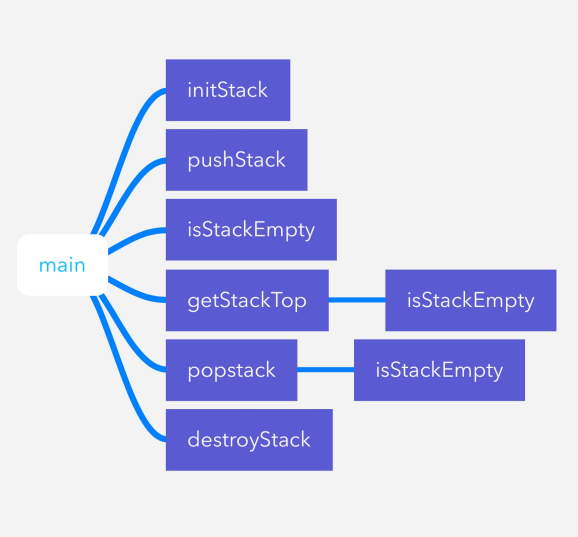

Figure 2: 函数的调用关系

4 调试分析报告

4.1 调试过程中遇到的问题和思考

初步实现后，测试样例时发现对于前导零的情况没有处理，遂增加查询栈顶操作，并且在主程序中增加弹出栈顶 0 元素的循环。

之后对串全为 0 的情况增加特判。

对于规模较大的数据测试时发现指针会访问到无效位置，发现是 realloc 操作之后没有更新 *top* 指针的位置所致。遂增加更新 *top* 指针的语句。

4.2 设计实现的回顾讨论

Stack 类型的 *top* 指针指向栈顶元素的下一个位置，实现时有几次没有注意到这个问题而导致错误。

期望输入合法，所以对于更多的不合法输入（比如输入末尾没有 #）没有作处理。

由于不清楚测试用数据的范围大小，$STACKINCREASESIZE$ 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，所以栈的实现没有考虑不符合前件的情况。

4.3 算法复杂度分析

initStack, isStackEmpty, pushStack, getStackTop, popStack, destroyStack 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 $O(n)$。

4.4 改进设想的经验和体会

4.4.1 改进 1

程序中的各个栈操作都可以直接改成一个数组的正序/逆序的访问。但是空间浪费很严重。并且不符合题目要求。

4.4.2 改进 2

此题中可以将改为根据输入规模一次性分配足够规模的栈空间，但是由于主程序中读入方式的限制而没有实现。

5 用户使用说明

使用 gcc 编译生成可执行文件。

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

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

测试非法输入和边界条件。

【输入】

```
1$21#
```

【输出】

```
Please check your input.
```

【输入】

```
0#
```

【输出】

```
0
```

【输入】

```
00000000000000000000#
```

【输出】

```
0
```

【输入】

```
0000000010#
```

【输出】

```
2
```

6.3 测试第三部分

使用 Python 实现 2/8 进制转换器 (test.py) 如下：

```
a = int(input()[:-1], 2)
print(oct(a)[2:])
```

将原解法与此解法比对。

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.py，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 (data.cpp) 如下：

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

using namespace std;

const int LEN = 1e4;

```

```cpp
int main()
{
    srand(time(0));
    int n = rand() % LEN + 1;
    for (int i = 1; i <= n; ++i)
        printf("%d", rand() % 2);
    puts("#");
    return 0;
}
```

比对脚本 (chk.sh) 如下：

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