数组扩展操作算法

/\*用一个子串（字符数不小于1）替换掉给定字符串中的某“个”指定字符\*/

/\*算法描述：

给定字符串的长度是len;

1.统计出给定字符串中含有的指定字符的数量count；

2.替换掉所有指定字符后字符串的长度是len+count\*more;

3.指针q指向原字符串尾，从后往前扫描原字符串；指针r指向替换后的字符串尾，

从后往前接收q扫描的字符；

4.q扫描到非指定字符，直接移到r；q扫描到指定字符，r及r前相应单元存储子串；

q前移一位；

5.直到q扫描完原字符串的首位后结束；

\*/

#include<stdio.h>

#include<string.h>

void Replace(char \*str) {

int len = strlen(str);

int i = 0;

printf("请输入被替换的字符:\n");

char ch = getchar();

printf("请输入替换的子串，且子串长度不大于14:\n");

char str1[15];

fflush(stdin);

gets(str1);

/\*替换一次字符增加的字符长度\*/

int more = strlen(str1) - 1;

/\*统计目标字符串中待替换字符的数量\*/

int count = 0;

while (str[i] != '\0') {

if (ch == str[i])

++count;

++i;

}

char \*q = &str[len];//取尾地址

char \*r = &str[len + count\*more];//取尾地址

while (q >= str) {

if (\*q != ch) {//非指定字符

\*r-- = \*q--;

}

else {//指定字符

i = strlen(str1) - 1;

while (i >= 0) {

\*r = str1[i];

--r; --i;

}

--q;

}

}

}

int main() {

printf("注释:(这里我用 hello world i am Joe !作为测试字符串,");

printf("用户也可以自行输入其他字符串)\n");

printf("请输入要处理的字符串:\n");

char p[200];

gets(p);

Replace(p);

printf("处理后的字符串:\n");

puts(p);

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*在字符数组中删除所有的elem\*/

/\*设置变量k = 0;

从前往后扫描字符串，碰到elem，++k;

碰到非elem，把该元素往前移动k位;

\*/

/\*k个字符把字符串分成k+1段，每段包含的字符，往前移动的

位数是不同的，段内所有字符移动的位数相同。

\*/

#include<stdio.h>

#include<string.h>

void DeleteAllElem(char \*str, char elem) {

int k = 0;

char \*p = str;

while (\*p != '\0') {

if (elem == \*p) {

++k;

}

else {

\*(p - k) = \*p;

}

++p;

}

\*(p - k) = \*p;

}

int main() {

printf("请输入待处理的字符串：\n");

char str[100];

gets(str);

puts(str);

printf("请输入要删除的元素：\n");

fflush(stdin);

char elem = getchar();

DeleteAllElem(str, elem);

printf("处理后的字符串是：\n");

puts(str);

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*一个随机数组，负数放前面，0和正数放前面\*/

/\*特别区分该程序和快速排序的区别；快速排序和分离组员两特征随机数组

都是把元素剥离，满足基准左的成员都小于基准，基准右的成员都不小于基准；

但是，分离组员两特征随机数组while（i<j）终止循环时；i == j；

i的终态不一定指向基准！！！

如 【1,2,6,4,3,3】，基准是3，但是i指向的是6,但是i一定是特征分界线.

当然有可能【2,1,3,6,7】，基准是3，终态i也指向3，但是这发生的概率不大。

快速排序既满足，基准左的成员都小于基准，基准右的成员都不小于基准，

又满足i一定指向基准。

如【1，3,6,7,6,8】，基准是第一个6，终态i一定指向第一个6.

\*/

void Divide(int \*a, int left, int right) {

while (left < right) {//当left和right相等时终止

while (left < right&&a[right] >= 0) {

--right;

}//从后往前寻找负数；结果是找到，或者索引重合

while (left < right&&a[left] < 0) {

++left;

}//从前往后寻找非负数；结果找到；或者索引重合

swap(a[left],a[right];

/\*两次寻找有三种情况：

1. i == j,同一单元交换

2. i != j

i和j必然逐渐靠拢,因为交换后i和j的指向必不再满足条件,必然执行++i和—j;这点和二分查找不同.

\*/

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

算法1

输入一个字符串1(长度位m) ,在字符串2(长度为n)中去除所有字符串1中出现的字符

算法分析:

1. 去除字符串1中重复出现的字符

设置一个数组,存放[a-z,A-Z,0-9]等所有可能出现的字符,标志位全部初始化成0,扫描字符串1,扫描到每一个字符,查看数组的对应单元,如果标志位是0,置1,如果是1,扫描指针后移.空间复杂度O(1),时间复杂度O(m).

1. 统计去重后字符串1的长度,假设是k.对于字符串2来说,字符串1中的字符没有差异,都是要删除的元素,方法与删除一个无序数组的某个字符,算法一样.时间复杂度是O(n\*k).

patition,部分排序,快速排序的联系

## patition

功能:选取数组中某一位置或值为某一值的元素,调整该元素在数组中的相对位置,使得该元素位置左侧的所有元素都<=它,右侧的元素都>=它.

返回值:目标元素被调整后的位置

参数列表:数组基地址,起始,终点

流程:

1,临时变量tmp保存第一个位置的元素,第一个位置相当于被挖空.

2,从后往前扫描,找到第一个小于tmp的元素,值注入空位.

3,从前往后扫描,找到第一个大于tmp的元素,值注入空位

4,重复1,2,3 直到两个扫描指针重合

注意:另设变量i,j代替起点和终点,形参值不变,可以在调整后,轻松得到起点-目的点-终点的索引.

unsigned partition(int \*a, unsigned low, unsigned high) {

/\*不直接用low和high扫描,是因为patition函数一般只是调用函数,low和high还有用\*/

unsigned i = low, j = high;

int temp;

/\*在这里可以拿其他元素与第一个位置的元素swap,从而可以调整任意元素\*/

temp = a[i];//挖空数组第一个位置,临时保存第一个元素的值

while (i < j) {//直到两个扫描指针重合

/\*先往后从前扫描\*/

while (i < j && a[j] >= temp) {

--j;

}

a[i] = a[j];

/\*再从前往后扫描\*/

while (i < j && a[i] <= temp) {

++i;

}

a[j] = a[i];

}

a[i] = temp;//放入被调整的元素

return i;//返回放入的位置

}

## patition与排序联系

一个无序数组,如果某个位置的元素,满足patition关系,那么它所处位置也是整个数组完全排序后的最终位置.

关于数组的问题,很多情况,要是数组有序的话,问题很好解决,所以一般先对数组排序,再进行最终目的算法,排序造成解决问题的复杂度最低是O(logn).但其实有时候,不需要整个数组有序,只需要获得局部有序的些许信息,就能辅助很容易解决问题,这时候就要用快速排序算法的

patition.

/\*

1. 排序时,优先填好第i个位置
2. 获知数组完全有序时第i个位置元素是谁

(3)获知数组完全有序时value在数组中的位置

\*/

功能: 排序时,优先填好第i个位置

返回值:void

形参列表:数组基地址,起点,终点,目标位置

流程:选取第一个元素,使它patition,如果它patition后的位置是i,返回

<i,对右边片段patition. >i,对左边片段patition.

void i\_pos\_partition(int \*a, unsigned low, unsigned high, unsigned pos) {

if (pos >= low && pos <= high && low <= high) {

unsigned index;

index = partition(a, low, high);

while (index != pos) {

if (index > pos) {

high = index - 1;

index = partition(a, low, high);

}

if (index < pos) {

low = index + 1;

index = partition(a, low, high);

}

}

}

}

## 实战

剑指offer - 面试题29 数组中出现次数超过一半的数字

- 面试题30 数组中最小的k个数

二分查找

## 二分查找

功能:在一个有序数组中查找关键字key.

返回值:找到返回key的标号，没找到，返回-1，发生错误返回-2

参数列表:数组基地址,起点,终点,数组长度,关键字

流程:

1，比较头标号i和尾标号j，等于key返回 i 或 j。

2，算出中间标号mid

3，如果中间元素等于key，返回中间标号；如果大于key，j = mid - 1;

如果小于key，i = mid + 1;

4，直到 i等于j

[注意]两个连续标号a，b除2求中间值mid，由于会舍去小数部分，会发生，mid等于i或j，

如果程序不等于key时，采用i = mid或j = mid，循环变量不变，发生死循环！

循环结束状态 i 一定等于 j.

若数组中有多个key,该方法查找到的是哪个key是不确定的.

int BinSearch (int \*a, int start, int end, int n, int key) {

if (!(start <= end && start <= n - 1 && end <= n - 1))

return -2;

int i = start, j = end;

while (i < j) {

if (a[i] == key)

return i;

if (a[j] == key)

return j;

int mid = i + (j - i) / 2;

if (a[mid] == key)

return mid;

else if (a[mid] > key) {

j = mid - 1;

}

else {

i = mid + 1;

}

}

return -1;

}

时间复杂度分析，

mid = n/2, (n/2)/2, ... n/2^k,

设n/2^k = 1，解得k = log2(n);

时间复杂度是log2(n).

## 二分查找拓展

功能:在数组中查找第一个等于3的元素的位置.

返回值:找到返回在数组中的下标, 没有返回-1，发生错误返回-2.

参数列表:数组基地址,起点,终点,数组长度,关键字

int SearchFristKey(int \*a, int start, int end, int length, int key) {

if (!(start <= end && start <= length - 1 && end <= length - 1))

return -2;

int i = start, j = end;

while (i < j) {

if (a[i] == key)

return i;

if (a[i] > key)

return -1;

int mid;

mid = i + (j - i) / 2;

if (a[mid] < key)

i = mid + 1;

if (a[mid] >= key)

j = mid - 1;

}

if (a[i] == key)

return i;

if (a[i + 1] == key)

return i + 1;

else

return -1;

}

查找最后一个等于3的元素的位置，没有返回-1，发生错误返回-2

int SearchLastKey(int \*a, int start, int end, int length, int key) {

if (!(start <= end && start <= length - 1 && end <= length - 1))

return -2;

int i = start, j = end;

while (i < j) {

if (a[j] == key)

return j;

if (a[j] < key)

return -1;

int mid;

mid = i + (j - i) / 2;

if (a[mid] <= key)

i = mid + 1;

if (a[mid] > key)

j = mid - 1;

}

if (a[i] == key)

return i;

if (a[i - 1] == key)

return i - 1;

else

return -1;

}

统计有序数组中key的个数

int NumOfKey(int \*a, int start, int end, int length, int key) {

int i = SearchFristKey(a, start, end, length, key);

int j = SearchLastKey(a, start, end, length, key);

if (i == -2 || j == -2)

return -2;

if (i == -1 || j == -1)

return -1;

return j - i + 1;

}

## 二分查找感悟

查找关键字key，整个过程就是两边往中间挤，循环结束时，i == j；a[i]左边的全部满足<key；a[i]右边的全部满足大于key，那么a[i]可能大于小于等于key。

查找第一个等于key的关键字，结束循环时，a[i]左边的都小于key，右边的都不小于key，

那么a[i]值不确定，但是key一定不会出现在i的左边，如果i本身等于key，则i是第一个，

如果i不等于key，i右边第一个等于key则i+1是第一个等于key的，且右边可能还有key。

如果不等于key，那么i右边的必然全部大于key，即，数组中无key。

Q：

思考如何求有序数组第一个大于key的？

队列基础算法

构造队列的初始套路是：定义队列参数结构体，参数赋值成空队列特征

链队列前言：

链队列参数：front，rear

front指向头节点，rear指向队尾

1. 定义一个队列（参数结构体）
2. 分配头节点，front和rear指向头节点，即空队列

空队列是可描述的：front == rear ==头节点

思考：为什么要引入头节点，而不是front = rear = NULL?

答:如果没有头节点，第一次入队需要修改front，最后一个节点出队，需要修改rear,麻烦！

顺序队列前言：

顺序队列参数：base,front，rear

base 初始队列容量的基址，用于操作各个队列元素

front指向队头（有效元素），rear指向队尾元素的下一个空位

1. 定义一个队列（参数结构体）
2. 分配一定空间的连续存储空间
3. front = rear = 0；

重要概念：

顺序队列要是循环队列，不然会随着出队入队造成空间的浪费。

如何实现空间的循环利用？

答：N个连续的存储单元，第一个单元编号是0，最后一个单元是N-1;

i代表任意一个单元的编号，++i % N能实现i的值从0走到N-1再自动回到0，依次循环往复。

空队列：front = rear

队满：（rear + 1）% N = front

难点：怎样通过rear和front计算队列元素的个数。

链队列C代码：

typedef struct Node

{

ElemType data;

struct Node\* next;

}QNode;//结点包括指针域和数据域

typedef struct

{

QNode\* front;

Qnode\* rear;

}Queue;//队列参数

Status InitQueue(Queue\*Q)

{

(\*Q).front = (\*Q).rear = (QNode\*)malloc(sizeof(QNode));

if(!(\*Q).front)

exit(-1);

(\*Q).front->next = NULL;

return ok;

}//初始化，空队列

Status DestoryQueue(Queue\*Q)

{

while((\*Q).front)

{

(\*Q).rear = (\*Q).front->next;

free((\*Q).front);

(\*Q).front = (\*Q).rear;

}

return ok;

}

Status ClearQueue(Queue\*Q)

{

QNode\*q,\*p;

(\*Q).rear = (\*Q).front;//都指向头节点

p = (\*Q).front->next;

(\*Q).front->next = NULL;//头节点指针域为空

while(p)

{

q = (\*p).next;

free(p);

p = q;

}

return ok;

}

Status QueueEmpty(Queue Q)

{

return ((Q).front==(Q).rear)?true:false;

}

int QueueLength(Queue Q)

{

int i = 0;//计数器

while(Q.front!=Q.rear)

{

++i;

Q.front = Q.front->next;

}

return i;

}

Status EnQueue(Queue\*Q,ElemType e)

{

QNode\* p = (QNode\*)malloc(sizeof(QNode));

if(!p)

exit(-1);

(\*p).data = e;

(\*p).next = NULL;

(\*Q).rear->next = p;

(\*Q).rear = p;

return ok;

}

Status DeQueue(Queue\*Q,ElemType\* e)

{

QNode\* p;

if(QueueEmpty(\*Q))

return error;//非空队列才能出队列

p = (\*Q).front->next;

\*e = p->data;

(\*Q).front->next = p->next;

if(p==(\*Q).rear)//出队可能改变谁是队尾，要更新参数

(\*Q).rear = (\*Q).front;

free(p);

p = NULL;

return ok;

}

void TraverseQueue(Queue Q)

{

if(QueueEmpty(Q))

return error;

QNode\* p = Q.front->next;

while(p!=Q.rear)

{

printf("%d ",p->data);

p = p->next;

}

printf("%d\n",Q.rear->data);

}

/\*顺序队列C代码

顺序队列也叫循环队列，之所以循环，是为了避免由于入队和出队

过程造成的空间浪费，使空间循环利用\*/

typedef struct

{

ElemType\* base;//操控每个元素

int front;//队头

int rear;//队尾

}SqQueue;//顺序队列参数

Status InitSqQue(SqQueue\*Q)

{

(\*Q).base = (ElemType\*)malloc(sizeof(ElemType)\*MAXSIZE);

if(!(\*Q).base)//分配队列容量

exit(-1);

(\*Q).front = (\*Q).rear = 0;//空队列

return ok;

}

Status ClearSqQue(SqQueue\*Q)

{

(\*Q).front = (\*Q).rear = 0;

return ok;//恢复初始化状态即为空队列

}

Status SqQueEmpty(SqQueue Q)

{

return Q.front==Q.rear?true:false;

}

Status SqQueFull(SqQueue Q)

{

return (Q.rear+1)%MAXSIZE == Q.front?true:false;

}

int SqQueLength(SqQueue Q)

{

return (Q.rear-Q.front+MAXSIZE)%MAXSIZE;

}

Status EnSqQue(SqQueue\*Q,ElemType e)

{

if(SqQueFull(\*Q))

return error;//队列不能满才能入队

(\*Q).base[(\*Q).rear] = e;

(\*Q).rear = ((\*Q).rear+1)%MAXSIZE;

return ok;

}

Status DeSqQue(SqQueue\*Q,ElemType\*e)

{

if(SqQueEmpty(\*Q))

return error;//队列不空才能出队

\*e = (\*Q).base[(\*Q).front];

(\*Q).front = ((\*Q).front+1)%MAXSIZE;

return ok;

}

Status TraverseSqQue(SqQueue Q)

{

if(SqQueEmpty(Q))

return error;

while(Q.front!=Q.rear)

{

printf("%d ",Q.base[Q.front]);

Q.front = (Q.front+1)%MAXSIZE;

}

printf("\n");

return ok;

}

栈基础算法

顺序栈前言：

顺序栈参数：base 首基址，操作各个连续的存储单元，top 栈顶元素的下一个空位，maxsize 栈的最大容量

空栈：top = base 栈满：top – base == maxsize

注意点：栈满时，top数组越界指向maxsize，而不是top指向maxsize-1

链栈前言：

链栈参数：top 指向栈顶元素，count 记录栈元素个数

思考链栈为什么引入头节点？

重点：链栈的节点的next存放的是上一个节点的地址，而不是下一个节点的地址！！！

目的是出栈时便于找到当前栈顶的上一个单元的地址！

顺序栈C代码：

#define INIT\_STACK\_SIZE 100

#define STACK\_INCREMENT 10

typedef int Status;

typedef int ElemType;

typedef struct

{

ElemType\* base;//基址

ElemType\* top;//栈顶元素的下一个空位

int maxsize;//栈的最大容量

}SqStack;

Status InitStack(SqStack\* S)

{

(\*S).base = (ElemType\*)malloc(sizeof(ElemType)\*INIT\_STACK\_SIZE);

if(!(\*S).base)

exit(-1);

(\*S).top = (\*S).base;//空栈

(\*S).maxsize = INIT\_STACK\_SIZE;//栈容量

return ok;

}

Status DestroyStack(SqStack\* S)

{

free((\*S).base);

(\*S).base = NULL;

(\*S).top = NULL;

(\*S).maxsize = 0;

return ok;

}

Status ClearStack(SqStack\* S)

{

(\*S).top = (\*S).base;

return ok;

}

Status StackEmpty(SqStack S)

{

return S.top == S.base?true:false;

}

int StackLength(SqStack S)

{

return S.top - S.base;

}

Status Push(SqStack\* S,ElemType e)

{

if((\*S).top - (\*S).base ==(\*S).maxsize)

{

ElemType\* p = (ElemType\*)realloc((\*S).base,sizeof(ElemType)\*((\*S).maxsize+STACK\_INCREMENT));

if(!p)

exit(-1);

(\*S).base = p;

(\*S).top=(\*S).base+(\*S).maxsize;//C语言的realloc用法细节，易忽视处

(\*S).maxsize += STACK\_INCREMENT;

}

\*(\*S).top++ = e;

return ok;

}

Status Pop(SqStack\* S,ElemType\* e)

{

if(StackEmpty(\*S))

return error;

\*e = \*--(\*S).top;

return ok;

}

Status TraverseStack(SqStack S)

{

if(StackEmpty(S))

return error;

while(S.base!=S.top)

{

printf("%d\t",\*S.base);

++S.base;

}

printf("\n");

return ok;

}

链栈C代码：

typedef int Status;

typedef int ElemType;

typedef struct Node

{

ElemType data;//数据域

struct Node\* next;//上一个节点的地址，而非下一个节点的地址

}SNode;

typedef struct

{

SNode\* top;//栈顶元素

int count;//计数器

}LinkStack;

Status InitLStack(LinkStack\* S)

{//空栈，头节点

(\*S).top = (SNode\*)malloc(sizeof(SNode));

if(!(\*S).top)

exit(-1);

(\*S).top->next = NULL;

(\*S).count = 0;

return ok;

}

Status PUSH(LinkStack\*S,ElemType e)

{

SNode\* p = (SNode\*)malloc(sizeof(SNode));

if(!p) exit(-1);

(\*p).data = e;

(\*p).next = (\*S).top;//新节点保存当前栈顶元素的地址

(\*S).top = p;//更新栈顶top参数，使其指向新节点

(\*S).count++;

return ok;

}

Status PoP(LinkStack\*S,ElemType\*e)

{

SNode\* p;//临时保存待删除节点的地址，释放之

if((\*S).count == 0)

return error;

p = (\*S).top;

\*e = (\*p).data;

(\*S).top = p->next;

free(p);//C语言中删除意味着free

p = NULL;

(\*S).count--;

return ok;

}

中/后缀表达式

/\*功能：中缀表达式转换成后缀表达式\*/

/\*中缀表达式是人类易于识别的算术表达式，如a\*(b-c)+d/(e+f),

但是计算机计算中缀表达式过程太复杂，先把中缀表达式转换成后缀表达式，

计算机借助一个栈就可轻松得到后缀表达式的结果。

中缀表达式转换成后缀表达式的算法：

1.扫描中缀表达式

2.碰到操作数直接输出

3.碰到操作符，分情况处理：

① 碰到 ( 直接入栈

② 碰到 + - \* /，若栈空，直接入栈；栈不空，若当前扫描到的操作符的

优先级高于栈顶操作符的优先级，直接入栈；否则，逐渐退栈并把退栈的元素

输出，直到栈顶操作符的优先级小于当前操作符的优先级或栈顶操作符是 (,

③ 碰到 ），退栈并输出，直到栈顶元素是 (,最后退出 (,但是不输出 ( 。

④ 扫完中缀表达式后，退出栈中所有操作符并输出。

\*/

/\*补充：中缀表达式有括号，后缀表达式没有括号，其实转换过程就是一个去括号的过程。

后缀表达式的计算算法：

1，借助栈

2，扫描后缀表达式，碰到操作数进栈，碰到运算符取栈顶两个元素计算，计算结果进栈

3，直到扫描完毕，这时候栈也刚好变空。\*/

#include<stdio.h>

#include<stdlib.h>

#define ARRSIZE 4

/\*比较( , +，- , \*，/，任意两个操作符的优先级

优先级相同返回0，前者大于后者返回1，前者小于后者返回-1

\*/

typedef struct {

char ch;//算术运算符

int prior;//优先级

}arith;

/\*储存算术运算符的及其优先级，以待检索\*/

arith arr[ARRSIZE] = { { '+', 1 }, { '-', 1 }, { '\*', 2 }, { '/', 2 } };

/\*比较两个算术运算符的优先级的函数\*/

int priority(char a, char b) {

int i = 0, j = 0;

while (i<ARRSIZE&&a != arr[i].ch) {

++i;

}//在优先级表中找到a

if (ARRSIZE == i) {

printf("运算符有误，操作失败！\n");

return -2;

}

while (j<ARRSIZE&&b != arr[j].ch) {

++j;//在优先级表中找到b

}

if (ARRSIZE == j) {

return -2;

}

if (arr[i].prior == arr[j].prior)

return 0;

if (arr[i].prior > arr[j].prior)

return 1;

if (arr[i].prior < arr[j].prior)

return -1;

}

/\*中缀表达式转换成后缀表达式\*/

void fun(char \*str1, char\* str2) {

char stack[30];

int top = -1;//栈

char \*p1 = str1;//扫描中缀表达式

char \*p2 = str2;//传出后缀表达式

while (\*p1 != '\0') {

if ('(' == \*p1) {

stack[++top] = \*p1++;

}//扫描到算术操作符

else if ('+' == \*p1 || '-' == \*p1 || '\*' == \*p1 || '/' == \*p1) {

if (-1 == top)

stack[++top] = \*p1++;

else {

int i;

for (i = top; i>-1; --i) {

if ('(' == stack[top] || 1 == priority(\*p1, stack[i])) {

stack[++top] = \*p1++;

break;

}

else {

\*p2++ = stack[top--];

}

}

stack[++top] = \*p1++;

}

}//扫描到 )

else if (')' == \*p1) {

while (stack[top] != '(') {

\*p2++ = stack[top--];

}

--top;

p1++;

}

else//扫描操作数

\*p2++ = \*p1++;

}

while (-1 != top) {//扫描完毕别忘了清栈

\*p2++ = stack[top--];

}

}

检测括号是否匹配

/\*利用栈检测括号是否匹配算法：

扫描表达式，碰到左括号进栈，碰到右括号，若栈顶是与右括号相匹配的

括号，则出栈，若不是，返回不匹配。

扫描完毕，刚好栈空，返回匹配，栈不空，返回不匹配。

\*/

#define STACKSIZE 20

#include<stdio.h>

int Check(char \*str) {

/\*top指向栈顶，初始栈内含有一个元素\*/

char stack[STACKSIZE + 1];

int top = 0;

stack[0] = '#';

char \*p = str;

while (\*p != '\0') {

switch (\*p) {

case '(':stack[++top] = '('; break;

case '[':stack[++top] = '['; break;

case '{':stack[++top] = '{'; break;

case ')': if ('(' == stack[top]) --top;

else return 0; break;

case ']': if ('[' == stack[top]) --top;

else return 0; break;

case '}': if ('{' == stack[top]) --top;

else return 0; break;

default: break;

}

++p;//【易错点】

}

return 0 == top ? 1 : 0;

}

两栈模拟队列

#define STACKSIZE 10

#define QUEUESIZE 10

/\*利用两个栈模拟一个队列的算法：

1，设置两个栈S1，S2；

2，入队就是入栈S1

3，【出队】：S2不空，出栈；

S2是空，把S1出栈【干净】，出栈元素再依次进栈S2，最后出栈 。

重点，S2不出栈干净是决不允许从S1取元素，而且，若从S1取元素，务必一次性取干净

\*/

//先编写出栈入栈基本操作，再模拟队列

//两栈模拟队列

//模拟出队

Status DeleteHead(Stack \*S1, Stack \*S2, char \*Qch)

{

if (!IsEmpty(S2))

{

Pop(S2, Qch);

return ok;

}

else

{

while (!IsEmpty(S1))

{

Pop(S1, Qch);

Push(S2, \*Qch);

}

if (!IsEmpty(S2))

{

Pop(S2, Qch);

return ok;

}

else

return error;

}

}

//模拟入队

Status AppendTail(Stack \*S1, char Qch)

{

if (IsFull(S1))

return error;

Push(S1, Qch);

return ok;

}

两队列模拟栈

/\*两队列模拟栈\*/

/\*从宏观来看，本质就是保证后来的元素在某个队列队头，而要保证是队头，

肯定是第一个入队列的，所以肯定保证有一个队列是空的\*/

/\*1，设置两个队列A，B;A，B初始状态都是空;

2，入栈：元素入空的那个队列；不空的队列的所有元素出队进入另一队列；

3，出栈：哪个队列是非空的，出队。

\*/

/\*该算法过程，始终保证了一个队列是空的，用来入栈。

每次入栈，两个队列空的不空，不空的边空；而入栈就找空的，出栈就找不空的，

所以，可设置一个标志位flag；

flag = 0时，表示A空，自然B非空

flag = 1时，表示B空，自然A非空

\*/

#include<stdio.h>

#include<stdlib.h>

/\*先编写好循环队列的函数，再来模拟栈\*/

#define QUEUESIZE 100/\*由于为了判断队列是否已满，

舍弃一个单元不用所以栈最多容纳99个元素 \*/

typedef struct {

int \*base;//基地址

int front, rear;//队头和队尾的索引

}Queue;

/\*初始化队列，front指向队头，rear指向队尾元素的下一个空位\*/

int flag = 0;//flag指示哪个队列空;0，A空，1，B空

void InitQueue(Queue &Q) {

Q.base = (int\*)malloc(sizeof(int)\*(QUEUESIZE));

if (NULL == Q.base) exit(-1);

Q.front = 0; Q.rear = 0;

}

/\*入队\*/

void EnQueue(Queue &Q, int elem) {

if ((Q.rear + 1) % (QUEUESIZE) == Q.front)

return;//队满

Q.base[Q.rear++] = elem;

}

/\*出队\*/

void OutQueue(Queue &Q,int &elem) {

if (Q.front == Q.rear)

return;//队空

elem = Q.base[Q.front];

Q.front = (Q.front + 1 + QUEUESIZE) % QUEUESIZE;

}

/\*模拟入栈\*/

void EnStack(Queue &A, Queue &B,int elem,int &flag) {

int temp;

if (flag == 0) {

EnQueue(A, elem);

while (B.front != B.rear) {

OutQueue(B, temp);

EnQueue(A, temp);

}

flag = 1;

}

else {

EnQueue(B, elem);

while (A.front != A.rear) {

OutQueue(A, temp);

EnQueue(B, temp);

}

flag = 0;

}

printf("%d已入栈！\n", elem);

}

/\*模拟出栈\*/

void OutStack(Queue &A, Queue &B) {

int temp;

if (0 == flag) {

if (B.front == B.rear)

return;

else {

OutQueue(B, temp);

printf("%d已出栈\n", temp);

}

}

else {

if (A.front == A.rear)

return;

else {

OutQueue(A, temp);

printf("%d已出栈\n", temp);

}

}

}

KMP算法

/\*改良后的KMP算法\*/

/\* http://www.cnblogs.com/tangzhengyue/p/4315393.html

在一段正文中查找特定的字符串，如果采用暴力匹配算法，

出现不相等时，正文要回退到上次开始匹配的下一个位置，

子串要回退到1号字符位置，时间效率是 m\*n

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

KMP算法有“记忆功能”，能利用上次匹配的信息，减少下一次匹配的回退长度.

出现不相等时，分为两种情况：

1.子串的第一个字符不相等：正文索引i前进一个位置，继续与子串开头匹配.

2.子串的非第一个字符不相等：正文索引不变，子串索引跳到其next值，再

继续匹配。

3.直到正文结束或子串结束

\*\*\*\*子串的next至于子串自身有关，与其他皆无关\*\*\*\*

引入概念：

next(x) = y的含义：

模式串【1，2，…, y-1】与【x-y+1，x-y+2，…，x-1】完全匹配，那很显然，

next(x+1) = y + 1

一个字符key[i]的next值不止一个！只是不同的next值[1 -- next[j]](j<i),

长度不一，我们要找的是最长的next位。

求next值过程：

根本思想是，from 1 to last -1 根据本位的next值求下一位的next的值.

1，分配一个与子串长度相等的数组

2，预定义0号字符next值是 -1

3，如果本位与next位相等，就得出下一字符的next值【注释：此步可优化】

4，不相等，就找本位的下下一个next值，直到找到与本位相等的next值或

next值超出数组，即=-1时结束，并填写下一个字符的next值。

\*/

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

void Next(char \*strkey, int\* &next) {

int keylen = strlen(strkey);//获取待匹配串的长度

next = (int\*)malloc(sizeof(int)\*keylen);

if (!next) exit(-1);//分配等长的next数组

next[0] = -1;//预定义0号字符的next值是-1；

/\*根据本位，推出下一位的next值。从0号字符开始推\*/

int k = -1;//k始终保存当前扫描位的next值，这点十分重要

int i = 0;

while (i < keylen - 1) {//遍历 1--last - 1

if (-1 == k || strkey[i] == strkey[k]) {

/\*满足字符和它的next字符相等就填写下一个字符的next值

不满足，继续找直到找到相等的next值，或者next == -1\*/

++i; ++k;

if (strkey[i] != strkey[k] || k == 0)//\*\*

next[i] = k;//\*\*

else //\*\* 这4行是优化改进的核心代码

next[i] = next[k];//\*\*

}

else

k = next[k];//找更短的"回跳比肩next值"

}

}

int KMP(char \*txt, char \*strkey, int \*next) {

int txtlen = strlen(txt);//正文长度

int keylen = strlen(strkey);//子串长度

int i = 0, j = 0;//扫描，不能使用for循环

while (i < txtlen && j < keylen) {

if (txt[i] == strkey[j]) {

++i;

++j;

}

else {/\*难点就是：不相等时，有两种情况要分别处理！！\*/

if (j == 0) {

++i;

}

else

j = next[j];

}

}

if (j == keylen && i <= txtlen) {

return i - keylen + 1;

}

else {

return 0;

}

}

int main() {

int \*next;//用于接收Next函数传出的next数组的首地址

char txt[200];

printf("请输入你的正文：\n");

gets(txt);

printf("请输入你要查找的子串:\n");

char strkey[20];

// fflush(stdin);

gets(strkey);

Next(strkey, next);

printf("子串的next求值结果：\n");

for (unsigned int i = 0; i <= strlen(strkey) - 1; ++i) {

printf("%5c", strkey[i]);

}

printf("\n");

for (unsigned int i = 0; i <= strlen(strkey) - 1; ++i) {

printf("%5d", next[i]);

}

printf("\n");

int pos = KMP(txt, strkey, next);

if (pos)

printf("字符匹配，匹配点是%d\n", pos);

else

printf("字符失配！\n");

system("pause");

return 0;

}