栈和队列

xbZhong

本页PDF

括号匹配

20. 有效的括号

简单 相关标签 相关企业 提示

给定一个只包括 `'('`，`')'`，`'{'`，`'}'`，`'['`，`']'` 的字符串 `s` ，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。
3. 每个右括号都有一个对应的相同类型的左括号。

示例 1：

> **输入：** s = "()"
> **输出：** true

示例 2：

> **输入：** s = "()[]{}"
> **输出：** true

示例 3：

> **输入：** s = "(]"
> **输出：** false

用栈来模拟括号匹配的过程 **用栈是因为栈可以解决具有完全包含关系的问题**

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
int n;
int flag = 0;
cin >> n;
vector<string> temp[n] , s;
string str;
string target;
for(int i = 0; i < n; i++)
{
    cin >> str;
        temp[i] = str;
        str = NULL ;
}
cin >> target;
for(int i = 0; i < n; i++)
{
    if(temp[i] == target)
    {
        s.push_back(temp[i]);
        flag = 1;
        break;
    }
    if(temp[i] == "return")
    s.pop_back(temp[i]);
    else
    s.push_back(temp[i]);
}
if(flag)
{
    for(int i = 0; i < s.size();i++)
    {
        if(i) cout << "->"
            cout << s[i];
    }
    cout << endl;
    else
        {
            cout << "NOT REFERENCED"
        }
}
return 0;
}
```

考研题 三元组最小距离

思路挺巧妙的

题目描述

定义三元组（a,b, c）（a,b,c 均为正数）的距离 D=|a-b|+|b-c|+|c-a|。给定 3 个非空整数集
并输出所有可能的三元组（a, b, c）（a∈S1,b∈S2,c∈S3）中的最小距离。例如 S1={-1, 0, 9}, S
元组为（9，10，9）。

程序中的主要部分已经帮你写好了，你只需要将如下代码拷贝到你的环境中，并且补充 func

输入参数

输入三个参数，分别为存储三个非空集合的队列

返回值说明

返回一个整形值，表示所有可能的三元组（a, b, c）（a∈S1,b∈S2,c∈S3）中的最小距离

* 栈里面的数字是从小到大排列的 * 因此我们只需要找到三个数字中最小的那个 * 对于他来说，其它两个数字再取栈后面的数字，其距离会越来越大 * 因此我们找到三个数字中的最小数字，就说明当前答案已经是对于这个数字来说的最小距离 * 我们便可以将它从栈中删除 * 同时要记录最小值 * 当有一个栈为空战时，就说明已经无数字可比，此时找到那个记录的最小值，输出即可

```
#include <iostream>
#include <cstdlib>
#include <queue>
using namespace std;

int min_num(int a, int b, int c) {
    if (a > b) swap(a, b);
    if (a > c) swap(a, c);
    return a;
}

int func(queue<int> que1, queue<int> que2, queue<int> que3) {
    int min = 0x3f3f3f;  //这个是16进制表示法，可以看成是正无穷
    while(!que1.empty() && !que2.empty() && !que3.empty())
    {
        int a = que1.front(),b = que2.front(),c = que3.front();
        int min_n = abs(a-b) + abs(a-c) + abs (b-c);
        if(min_n<min)  min = min_n;
        int d = min_num(a,b,c);
        if(a == d) que1.pop();
        if(b == d) que2.pop();
        if(c == d) que3.pop();
}
    return min;
}

int main() {
    int m, n, k, x;
    queue<int> que1, que2, que3;
    cin >> m >> n >> k;
    for (int i = 0; i < m; i++) {
        cin >> x;
        que1.push(x);
    }
    for (int i = 0; i < n; i++) {
        cin >> x;
        que2.push(x);
    }
    for (int i = 0; i < k; i++) {
        cin >> x;
        que3.push(x);
    }
    cout << func(que1, que2, que3) << endl;
    return 0;
}
```

比较含退格的字符串

844. 比较含退格的字符串

已解

简单 相关标签 相关企业 A文

给定 `s` 和 `t` 两个字符串，当它们分别被输入到空白的文本编辑器后，如果两者相等，返回 `true` 。`#` 代表
字符。

注意：如果对空文本输入退格字符，文本继续为空。

示例 1：

> **输入：** s = "ab#c", t = "ad#c"
> **输出：** true
> **解释：** s 和 t 都会变成 "ac"。

示例 2：

> **输入：** s = "ab##", t = "c#d#"
> **输出：** true
> **解释：** s 和 t 都会变成 ""。

示例 3：

> **输入：** s = "a#c", t = "b"
> **输出：** false
> **解释：** s 会变成 "c"，但 t 仍然是 "b"。

使用栈的思想 * 遇到#就出栈，否则就入栈 * 比较俩栈的字符

```
    class Solution {
    public:
        bool backspaceCompare(string s, string t) {
            stack<char> temp1,temp2;
            for(int i = 0;s[i];i++)
            {
                if(s[i] == '#')   //判断是否为退格字符
                {
                    if(!temp1.empty()) //判断栈是否非空 若无这个判断条
                temp1.pop(); 件 遇到a##b时便会把#压入栈，使得逻辑错误
                }
                else temp1.push(s[i]);
            }
            for(int i = 0;t[i];i++)
            {
                if(t[i] == '#')
                {
                    if(!temp2.empty())
                temp2.pop();
                }
                else temp2.push(t[i]);
            }
            if(temp1.size() != temp2.size()) return false;
            while(!temp1.empty())
            {
                if(temp1.top() != temp2.top() ) return false;
                temp1.pop();
                temp2.pop();
            }
            return true;
        }
    };
```

火车进栈

#263. 火车进

描述 提交 自定义测试 题解视频

题目描述

有 n 列火车按 1 到 n 的顺序从东方左转进站，这个车站是南北方向的，它虽然无限长，只可惜是一车必须进站，先进后出。

进站的火车编号顺序为 $1 \sim n$，现在请你按火车编号从小到大的顺序，输出前 20 种可能的出站方案

输入

输入一行一个整数 n。 （ $n \leq 20$ ）

输出

输出前 20 种答案，每行一种，不要空格。

样例输入1

```
3
```

样例输出1

```
123
132
213
231
321
```

* 全排列输出 * 重点是判断输出序列是否为合法序列 * 用栈模拟 1. 假设有一组输出序列a1，a2，a3，a4，栈顶元素为x 2. 现在轮到a2输出，用a2与x比较 3. a2 > x 说明a2还没进栈，此时应该继续进栈，直到a2进栈 4. a2 = x 栈顶元素即进站的火车编号为x，此时a2弹出栈 5. a2 < x 说明a2在栈顶元素下方，无法弹出，此时序列不合法

- 一种是使用next_permutation函数来进行全排列
- 一种是使用递归的排列型枚举来进行全排列
- **next_permutation函数做法**

  #include #include #include #include #include #include using namespace std; bool valid(int a[],int n) { stack s; int x = 1; for(int i = 0;i < n;i++) { if(s.empty() || s.top() < a[i]) //血的教训！！ 先判空再比较 { while (x<=a[i]) s.push(x),x+=1; } if(s.top() != a[i]) return false; s.pop(); } return true; } int main() { int a[25],n,cnt = 20; cin >> n; for(int i =0;i < n;i++) a[i] = i + 1; do { if(valid(a,n)) { for(int i = 0;i <n;i++) { cout << a[i]; } cout << endl; cnt-=1; } }while(next_permutation(a,a+n) && cnt); return 0; }

递归的排列型枚举做法

```
#include<iostream>
#include<cstdlib>
#include<cstdio>
#include<string>
#include<cstring>
#include<stack>
#include<algorithm>
#include<queue>
using namespace std;
int vis[25]={0};
int a[25],b[25];
int times = 0;
bool valid(int n)
{
    stack<int> s;
    int x = 1;
    for(int i = 0;i < n;i++)
    {
        if(s.empty() || s.top()<a[i])
        {
            while (x<=a[i])
            {
                s.push(x);
                x+=1;
            }
        }
        if(s.top()!=a[i]) return false;
        s.pop();
    }
    return true;
}
void print(int n)
{
    for(int i = 0;i < n;i++)
    {
            cout << b[i];
    }
    cout << endl;
}

void f(int i,int n)
{
    if(times == 20) return ;
    if(i == n && valid(n))
    {
        print(i);
        times ++;
        return;
    }
    for(int j = 1;j <= n;j++)
    {
        if(vis[j]) continue;
        b[i] = j;
        a[i] = j;
        vis[j] = 1;
        f(i+1,n);
        vis[j] = 0;
    }
}
int main()
{
    int n;
    cin >> n;
    f(0,n);
}
```

重点解释一下next_permutation函数 * 在头文件include< algorithm >中 * 接受两个迭代器作为参数，并返回一个bool值，表示是否成功生成下一个排列 * 迭代器：访问数据结构中的元素 * 函数是对这两个迭代器之间的数据进行字典序排序 * 当序列已经是字典序最大排列，返回false，循环结束 * 作用：生成给定序列的下一个较大排序，直到序列按降序排列为止 * 常与do while联用 * 若与while连用，则会丢失初始时的排序 * 若你想要得到所有的排序结果，初始时要按照升序来排序

验证栈序列

946. 验证栈序列

已解

中等 相关标签 相关企业

给定 `pushed` 和 `popped` 两个序列，每个序列中的 **值都不重复**，只有当它们可能是在最初空栈上进行的推入 push 和弹出 pop 操作序列的结果时，返回 `true`；否则，返回 `false` 。

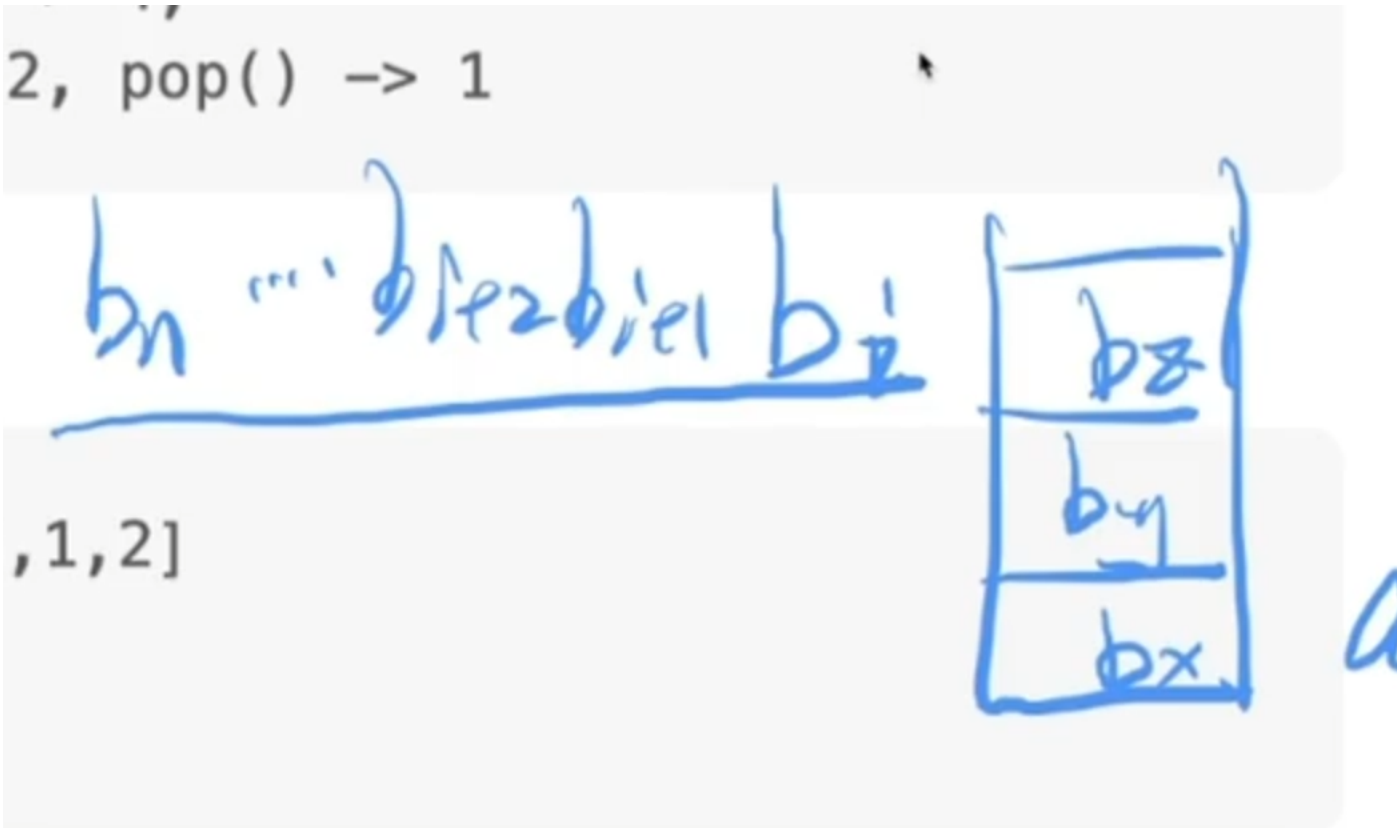
示例 1：

> **输入：** pushed = [1,2,3,4,5], popped = [4,5,3,2,1]
> **输出：** true
> **解释：** 我们可以按以下顺序执行：
> push(1), push(2), push(3), push(4), pop() -> 4,
> push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1

示例 2：

> **输入：** pushed = [1,2,3,4,5], popped = [4,3,5,1,2]
> **输出：** false
> **解释：** 1 不能在 2 之前弹出。

题目意思是按照所给顺序对空栈进行入栈和出栈操作 若执行到最后栈为空，则return true * 要对出栈序列进行分析 * 若出栈序列当前元素与栈顶元素相同，则pop * 不相同则一直入栈 * 若将入栈序列遍历完也没有找到当前元素，且栈顶元素也与其不同，return false * 因为该元素在栈顶元素下方 无法pop



> alt text

```
class Solution {
public:
    bool validateStackSequences(vector<int>& pushed, vector<int>& popped) {
        int x = 0,n = pushed.size();
        stack<int> s;
        for(int i = 0;i < n;i++)
        {
            if(s.empty() || s.top() != popped[i])
            {
        while(x < pushed.size() && pushed[x] != popped[i])
        {
            s.push(pushed[x]);
            x += 1;
        }
        if(x == pushed.size()) return false;
        s.push(pushed[x]);
        x +=1 ;
            }
        s.pop();
        }
        return true;
    }
};
```

括号画家

#265. 括号

描述　提交　自定义测试　题解视频

题目描述

Candela 是一名漫画家，她有一个奇特的爱好，就是在纸上画括号。这一天，刚刚起床的 *Candela*
。这排随意绘制的括号序列显得杂乱无章，于是 *Candela* 定义了什么样的括号序列是美观的：

1. 空的括号序列是美观的；
2. 若括号序列 `A` 是美观的，则括号序列 `(A)、[A]、{A}` 也是美观的；
3. 若括号序列 `A、B` 都是美观的，则括号序列 `AB` 也是美观的；

例如 `[(){}]()` 是美观的括号序列，而 `)({)[}](` 则不是。

现在 *Candela* 想在她绘制的括号序列中，找出其中连续的一段，满足这段子序列是美观的，并且长

要让匹配到的一对括号做上标记

```
#include<iostream>
#include<stack>
using namespace std;
stack<int> s;
char str[10005];
int match[10005];
int main()
{
    cin >> (str + 1);  //让字符数组下标从1开始
    for(int i = 1;str[i];i++)
    {
        switch(str[i])
        {
        case'(':
        case'[':
        case'{': s.push(i); break;
        case')':
            if(!s.empty() && str[s.top()] == '(')
            {
                match[s.top()] = i;  //表明s.top()和i位置是匹配的
                s.pop();
            }
            else  s.push(i); //表明i位置之前的括号序列非法，做一个信息阻隔
            break;
        case']':

            if(!s.empty() && str[s.top()] == '[')
            {
                match[s.top()] = i;
                s.pop();
            }
            else s.push(i);
            break;
        case'}':
            if(!s.empty() && str[s.top()] == '{')
            {
                match[s.top()] = i;
                s.pop();
            }
            else s.push(i);
            break;
            }
    }
    int temp_ans = 0,ans = 0,i = 1;
    while(str[i])
    {
        if(match[i])
        {
            temp_ans += (match[i] - i + 1); //临时的长度 +=是因为可能会出现([]){}的形式
            i = match[i] + 1;             //跳转到匹配位置的下一个位置，寻找下一个美观长度
        }
        else
        {
            temp_ans = 0;
            i++;
        }
        if(temp_ans > ans) ans = temp_ans; //取最大长度
    }
    cout << ans;
    return 0;
}
```

设计循环队列

622. 设计循环队列

中等 相关标签 相关企业

设计你的循环队列实现。 循环队列是一种线性数据结构，其操作表现基于 FIFO（先进先出）原则并且队尾被连接在队首之后以形成一个循环。它也被称为“环形缓冲器”。

循环队列的一个好处是我们可以利用这个队列之前用过的空间。在一个普通队列里，一旦一个队列满了，我们就不能插入下一个元素，即使在队列前面仍有空间。但是使用循环队列，我们能使用这些空间去存储新的值。

你的实现应该支持如下操作：

- `MyCircularQueue(k)`: 构造器，设置队列长度为 k 。
- `Front`: 从队首获取元素。如果队列为空，返回 -1 。
- `Rear`: 获取队尾元素。如果队列为空，返回 -1 。
- `enQueue(value)`: 向循环队列插入一个元素。如果成功插入则返回真。
- `deQueue()`: 从循环队列中删除一个元素。如果成功删除则返回真。
- `isEmpty()`: 检查循环队列是否为空。
- `isFull()`: 检查循环队列是否已满。

示例：

```
MyCircularQueue circularQueue = new MyCircularQueue(3); // 设置长度为 3
circularQueue.enQueue(1);  // 返回 true
circularQueue.enQueue(2);  // 返回 true
```

alt text

本题难点：需要确定初始时尾指针位置 * 与队列结构定义不一样，队列机构定义头尾指针采取左闭右开式 * 在本题头尾指针采取左闭右闭式 * 因此尾指针初始位置应该在头指针前一位才能表示队列为空

```
struct Node
{
    int data;
    Node *next;
};
class MyCircularQueue {
public:
    int count,size;
    Node *head,*tail;
    MyCircularQueue(int k) {
        head = new Node();
        tail = head;
        for(int i = 0;i < k;i++)
        {
            tail->next = new Node();
            tail = tail->next;
        }
        count = 0;
        size = k;
        tail->next = head;
    }

    bool enQueue(int value) {
        if(isFull())  return false;
        tail = tail->next;
        tail->data = value;
        count+=1;
        return true;
    }

    bool deQueue() {
        if(isEmpty()) return false;
        head = head->next;
        count-=1;
        return true;
    }

    int Front() {
        if(isEmpty()) return -1;
        return head->data;
    }

    int Rear() {
        if(isEmpty()) return -1;
        return tail->data;
    }

    bool isEmpty() {
        return count == 0;
    }

    bool isFull() {
        return count == size;
    }
    };
```

表达式求值

题目描述

给出一个表达式,其中运算符仅包含 `+,-,*,/,^` 要求求出表达式的最终值。

数据可能会出现括号情况，还有可能出现多余括号情况，忽略多余括号，正常计算即可；

数据保证不会出现大于 *max long int* 的数据；

数据可能会出现负数情况，幂次不可能为负数，除法采用向 0 取整。

注意：−9 和 +9 分别代表负数和正数的 9

输入

共一行，即为表达式。表达式长度不会超过1000.

输出

共一行，既为表达式算出的结果。

样例输入1

```
(2+2)^(1+1)
```

样例输出1

```
16
```

样例输入2

* 使用递归的思想 * 找到计算优先度最低的符号进行分割 * 对符号左右进行求值 * 最后返回计算结果 * 要找计算优先度最低的符号须计算权重比 * 对于括号内部的符号权重比加100 * 考虑负数，对于数字前的负号，权重比更大，加1000 * 因为负数与相减相比优先度更高 * 负数实现过程是用负号分割。用0减去当前数字 * 为了让分割位置指向符号，我们让数字优先度达到正无穷 * pos代表要分割的位置 * cur_pri统计当前位置的权重 * temp_pri统计遇到括号的权重 * 设定pri为正无穷-1 * 目的是为了找出分割位置 * 减去1是为了不让pos指向数字 * 找分割位置条件带等于号是因为当遇见权重比相同的符号时，靠后的符号权重比更小 例如**7-5-2**，先计算7-5，再计算后面的表达式

```
#include<bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f
string str;
bool is_operator(char c)
{
    switch(c)
    {
    case'+':
    case'-':
    case'*':
    case'/':
    case'^': return true;
    default : return false;
    }
    return false;
}

long long result(string &s, long long l,long long r)
{
    long long pos = -1 , pri =INF - 1 , cur_pri , temp_pri = 0;
    for(long long i = l;i < r;i++)
    {
        cur_pri = INF;
        switch(s[i])
        {
        case'(':
            temp_pri += 100;
            break;
        case')':
            temp_pri -=100;
            break;
        case'+':
        case'-':
            cur_pri = 1 + temp_pri;
            break;
        case'*':
        case'/':
            cur_pri = 2 + temp_pri;
            break;
        case'^':
            cur_pri = 3 + temp_pri;
            break;
        }
        if((s[i] == '-' || s[i] == '+') && (i -1 < 0 || is_operator(s[i - 1])))
        cur_pri += 1000;
        if(pri >= cur_pri)  //要找优先级最小的一个位置
        {
            pri = cur_pri;  // 更新最小值
            pos = i;
        }
    }
    if(pos == -1)
    {
    long long num = 0;
    for(long long i = l;i < r;i++)
    {
        if(s[i] < '0'|| s[i] > '9') continue;
        num = num * 10 + (s[i]-'0');
    }
    return num;
    }
    else
    {
        long long a = result(s,l,pos);
        long long b = result(s,pos+1,r);
        switch(s[pos])
        {
            case'+': return a+b;
            case'-': return a-b;
```

```
                    case'*': return a*b;
                    case'/': return a/b;
                    case'^': return pow(a,b);
                }
            }
        }
        int main()
        {
            cin >> str;
            cout << result(str,0,str.size());
            return 0;
        }
```