数据结构课程设计

北京郵電大學

姓　　名　tobeApe6eHok

学　　院　人工智能学院

专　　业　智能科学与技术

班　　级　1111111111

学　　号　1111111111

班内序号　1

指导教师　周延泉

2022 年　4 月

目录

数据结构课程设计实验二

一、实验题目

假设一个算术表达式中包含圆括号、方括号和花括号 3 种类型的括号，设计一个算法判断算术表达式的括号是否正确配对，以字符“\0”作为算术表达式的结束符。

考虑设定括号匹配规则：

1、括号前后匹配。
2、小括号中不可以有中括号或者大括号。
3、中括号中必须要有小括号。
4、大括号中必须要有中括号。
5、但是同级括号之间可以并包，如(())[[()]]等。

二、题目分析与算法设计

1、设计思路：

基本的要求是判断算术表达式中的括号是否匹配，这个很好实现，所以要加上相应的括号匹配规则。可以设置括号的等级，将小括号、中括号、大括号的等级设置为依次增加，这样只需要在检查括号前后匹配时同时检查等级是否合法即可。

在引入等级这个概念后，相应的匹配要求就可以等价于：

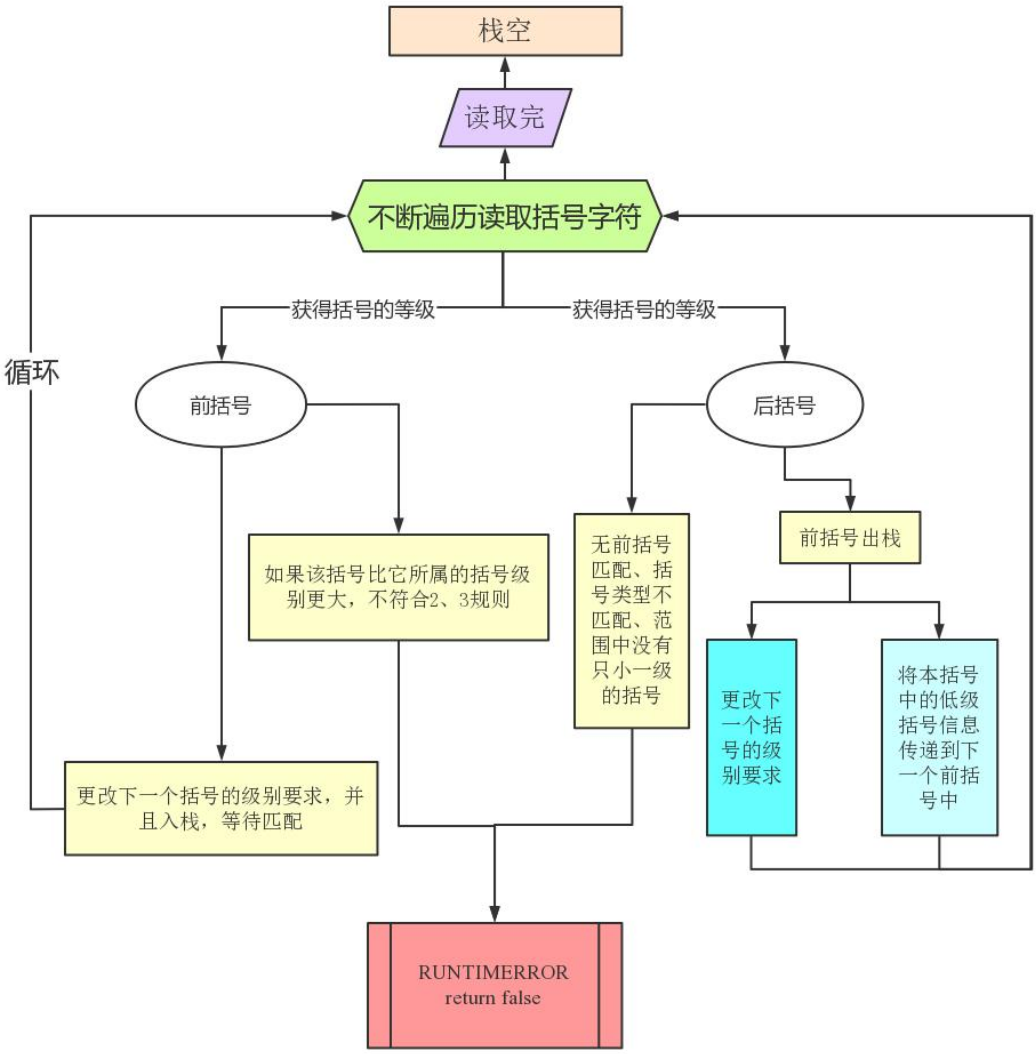
（1）大括号中必须要有中括号，中括号里必须要有小括号：等价于除了小括号外的括号范围内都必须要有等级只差一级的括号。这也意味着，在{[()]}中{}在[]判断为合法后不必纠结[]中是否有()。

（2）小括号中不可以有中括号或者大括号、中括号中不可以有大括号：等价于如果括号之间重叠，则必须要级别递减，而不是单调递减。

此外，我还设计了一个 RUNTIMERROR 的函数，来获得为什么匹配成功的原因，这是程序对修正括号算术表达式的一种建议。

2、算法描述：

由 1 的分析可以得到相应的算法，只需要在匹配前后括号的同时注意等级的变化即可，算法可以表示为如下图：



其中，“将本括号中的低级括号信息传到下一个前括号中”步骤也可以通过每次遇到前括号时完成，这里我在每次完成后括号匹配时完成，两者原理相同。

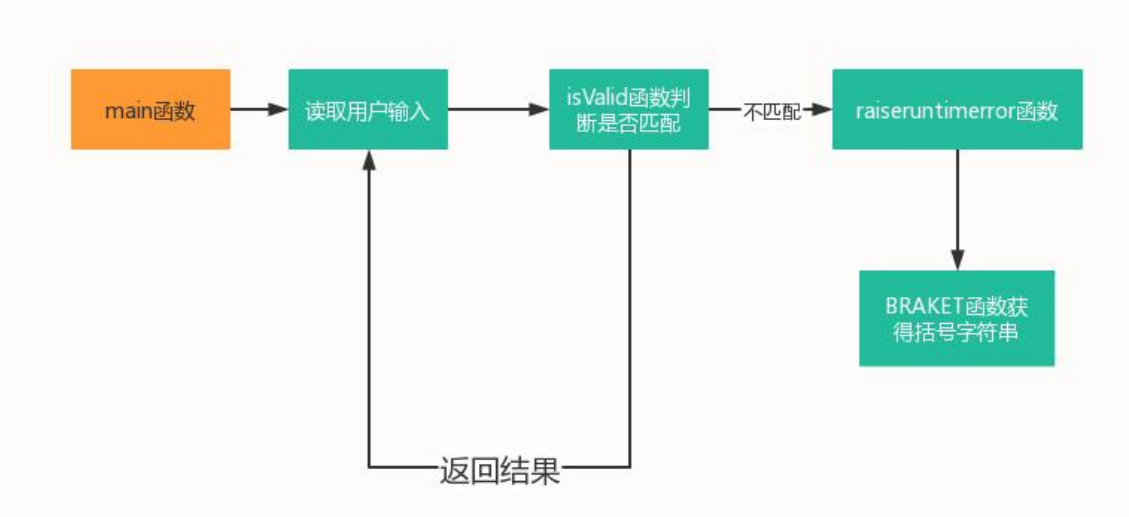
以下便是核心的算法（未带注释和错误识别的代码，但是在程序清单中含有）：

```
#define NONE           0
#define LITTLE         1
#define MIDDEL         2
#define LARGE          3
#define NeXTHAVENT     0
#define NeXTHAVED      1
bool isValid(char* s, int len)
{
    stack<pair<int, int>>unpaired;
    int NeXTsMALLER = LARGE, NOW = NONE;
    for (int i = 0; i < len; i++)
    {
        switch (s[i]) {
        case '(':
```

```
            NOW = (NOW == NONE ? LITTLE : NOW);
        case '[':
            NOW = (NOW == NONE ? MIDDEL : NOW);
        case '{':
            NOW = (NOW == NONE ? LARGE : NOW);
            if (NOW > NeXTsMALLER)
            {
                return false;
            }

            NeXTsMALLER = NOW;
            unpaired.push({ NOW,(NOW == LITTLE ? NeXTHAVED : NeXTHAVENT) });
            break;
        case ')':
            NOW = (NOW == NONE ? LITTLE : NOW);
        case ']':
            NOW = (NOW == NONE ? MIDDEL : NOW);
        case '}':
            NOW = (NOW == NONE ? LARGE : NOW);
            if
(unpaired.empty()||unpaired.top().first!=NOW||unpaired.top().second==Ne
XTHAVENT)
            {
                return false;
            }
            unpaired.pop();
            if (!unpaired.empty())
            {
                NeXTsMALLER = unpaired.top().first;
                if (NeXTsMALLER == NOW || NeXTsMALLER == NOW + 1)
                {
                    unpaired.top().second = NeXTHAVED;
                }
            }
            else
            {
                NeXTsMALLER = LARGE;
            }
            break;
        default:
            break;
        }
        NOW = NONE;
    }
    return unpaired.empty();
}
```

3、函数调用：



三、数据结构描述

1、数据结构的选择：

我使用的数据结构是 STL 中的 stack 栈，栈中的每个元素是一个 pair 键值对，实际中是用来存储每个遇到的前括号的信息，具体定义如下：

```
stack<pair<int, int>>unpaired;
```

2、数据结构的正确性：

题目要求判断相应的括号是否配对，首先括号分为前括号和后括号，如果后括号出现，那么前面必定要出现对应的前括号；其次，分析可得，无论括号之间的重叠有多深，最后必定有一个位置是()/{}/[],然后这样的结构 便可直接删除，删除后，又会出现这样的结构，这样就可以不断的迭代。但是得到这样的结构的标志是出现后括号，所以使用栈结构可以在遇到反括号后直接得到剩余的前括号，这样便证明了栈的正确性。

3、数据结构的意义：

键值对的意义如下：

（1）key：第一个元素存储前括号对应的级别

（2）Value：存储本括号范围内中是否出现了比本括号等级只小一级的括号，因为中括号中必须由小括号、大括号中必须要有中括号。

四、程序清单

```
#include<iostream>
#include<stack>
using namespace std;
#define LENGTH          150   //假设要判断的字符串长度最长是150,可以改变LENGTH的值
//将不同的括号设置为不同的整型
#define NONE            0     //未输入时为NONE
#define LITTLE          1     //表示小括号
#define MIDDEL          2     //表示中括号
#define LARGE           3     //表示大括号
//表示下一级对应的括号是否已经出现,stack<pair<int,int>>中的第二个值使用这两个宏
#define NeXTHAVENT      0     //表示下一级对应的括号未出现
#define NeXTHAVED       1     //表示下一级对应的括号出现了
//表示为什么括号匹配不成功,或者括号的使用有什么不正确的错误类型
#define BIGinSMALL      1     //表示在低级括号中使用了高级括号
#define NOLITTLE        2     //表示在一个高级括号中没有使用小括号
#define CANTMATCH       3     //表示前后两个括号不可以匹配,或者丢失了一个反括号
#define CHAREXTRA       4     //表示输入的字符串中有冗余括号
//根据宏定义的值获得代表括号的字符串
string BRAKET(int NOW)
{
    if (NOW == LITTLE) return "()";
    else if (NOW == MIDDEL) return "[]";
    else return "{}";
}
//判断括号是否匹配成功的函数
//s是用户的输入，len是字符串的长度
bool isValid(char* s, int len)
{
    //一个打印为什么匹配不同，或者括号使用不正确的原因的函数
    //type表示错误的类型，与上面的宏定义一一对应
    auto raiseruntimerror = [](int type, int FIRST = -1, int SECOND = -1)->void {
        cout << endl;
        if (type == BIGinSMALL)
        {
            cout << "从" << BRAKET(FIRST) << "中发现" << BRAKET(SECOND) << ",低级括号包含高级括号" << endl;
        }
        else if (type == NOLITTLE)
        {
            cout << "有一个" << BRAKET(FIRST) << "中没有下一级括号" << endl;
        }
```

```
        else if (type == CANTMATCH)
        {
            cout << BRAKET(FIRST) << "前括号不可以和" << BRAKET(SECOND) <<
"后括号匹配,或者少了一个" << BRAKET(FIRST) << "后括号" << endl;
        }
        else if (type == CHAREXTRA)
        {
            cout << "有冗余字符" << endl;
        }
    };
    //使用栈
    //第一个元素存储括号的级别
    //第二个元素存储后面是否出现了比本括号只小一级的括号
    //因为中括号中必须要有小括号、大括号中必须要有中括号
    stack<pair<int, int>>unpaired;
    //NeXTsMALLER 表示该前括号和对应的后括号之间的括号都应比本括号的级别要小，
因为中括号中不许有大括号，小括号中不许有中括号和大括号
    //NOW 表示现在遍历到的括号的级别
    int NeXTsMALLER = LARGE, NOW = NONE;
    for (int i = 0; i < len; i++)
    {
        switch (s[i]) {
        //如果是前括号，获得其级别
        case '(':
            NOW = (NOW == NONE ? LITTLE : NOW);
        case '[':
            NOW = (NOW == NONE ? MIDDEL : NOW);
        case '{':
            NOW = (NOW == NONE ? LARGE : NOW);
            //对于所有的前括号操作相同
            //如果当前的括号比前面的括号级别高，则不满足中括号中不许有大括号、
小括号中不许有中括号和大括号的要求
            if (NOW > NeXTsMALLER)
            {
                raiseruntimerror(BIGinSMALL, NeXTsMALLER, NOW);
                return false;
            }
            //更改下一个括号的级别要求
            NeXTsMALLER = NOW;
            //向栈中放入相应的前括号
            //如果是小括号，则只需要使用 NeXTsMALLER 判断是否满足级别不会再增
大，而不需要再判断括号中是否有只小一级的括号，因为小括号级别最小
            unpaired.push({ NOW,(NOW == LITTLE ? NeXTHAVED : NeXTHAVENT) });
            break;
        //如果是后括号，获得其级别
        case ')':
            NOW = (NOW == NONE ? LITTLE : NOW);
        case ']':
            NOW = (NOW == NONE ? MIDDEL : NOW);
        case '}':
            NOW = (NOW == NONE ? LARGE : NOW);
            //对所有后括号的操作相同
            //如果前面没有对应的前括号匹配
            if (unpaired.empty())
```

```
            {
                raiseruntimerror(CHAREXTRA);
                return false;
            }
            //如果这两个括号之间没有只小一级的括号，则不满足中括号中必须要有小括号、大括号中必须要有中括号的要求
            else if (unpaired.top().second == NeXTHAVENT)
            {
                raiseruntimerror(NOLITTLE, unpaired.top().first);
                return false;
            }
            //如果前括号和后括号类型不匹配
            else if (unpaired.top().first != NOW)
            {
                raiseruntimerror(CANTMATCH, unpaired.top().first, NOW);
                return false;
            }
            //匹配成功，出栈
            unpaired.pop();
            //如果此次储栈的括号是叠加在另一个更大的括号中的
            if (!unpaired.empty())
            {
                //获得更大括号的类型
                NeXTsMALLER = unpaired.top().first;
                //如果刚刚出栈的括号是只小一级的括号，或者是同级括号，则栈顶前括号的第二个元素可以变成 NeXTHAVED
                if (NeXTsMALLER == NOW || NeXTsMALLER == NOW + 1)
                {
                    unpaired.top().second = NeXTHAVED;
                }
            }
            else
            {
                //如果本段括号匹配完毕，则将最初的级别限制放宽
                NeXTsMALLER = LARGE;
            }
            break;
        default:
            break;
        }
        //还未读取的下一个字符
        NOW = NONE;
    }
    //如果最后还有前括号，则说明括号冗余
    if (!unpaired.empty())
    {
        raiseruntimerror(CHAREXTRA);
        return false;
    }
    return true;
}
int main()
{
    char* strInput = (char*)malloc(sizeof(char) * LENGTH);
    int isContinue = 2;
```

```
    while (isContinue)
    {
        cout << "请输入要检查的字符串:" << endl;
        char* charTOINPUT = strInput, justIN;
        if (isContinue == 1)
        {
            justIN = cin.get();
        }
        while (justIN = cin.get())
        {
            if (justIN == '\n' || !charTOINPUT)
            {
                break;
            }
            *charTOINPUT++ = justIN;
        }
        int inputSize = charTOINPUT - strInput;
        bool judgeResult = isValid(strInput, inputSize);
        if (judgeResult)
        {
            cout << "满足要求." << endl;
        }
        else
        {
            cout << "所以不满足要求." << endl;
        }
        cout << "\n是否继续? 0|1" << endl;
        cin >> isContinue;
        if (isContinue != 1)
        {
            break;
        }
    }
    free(strInput);
    return 0;
}
```

五、程序复杂度分析

1、时间复杂度分析：

在程序中我使用了栈的数据结构，栈结构中的每个操作都是 O(1)；对于 isValid()函数来说，要遍历其中的每个字符，每遇到一个前括号必定都要入栈一次，复杂度为 O(1)，每遇到一个后括号，要检查前面是否有前括号，如果有，则弹出，复杂度是 O(1)；如果没有，则结束。

故综上，假设总共有 n 个字符，则每个字符操作是 O(1)，总的复杂度是 O(1)*n=O(n)。

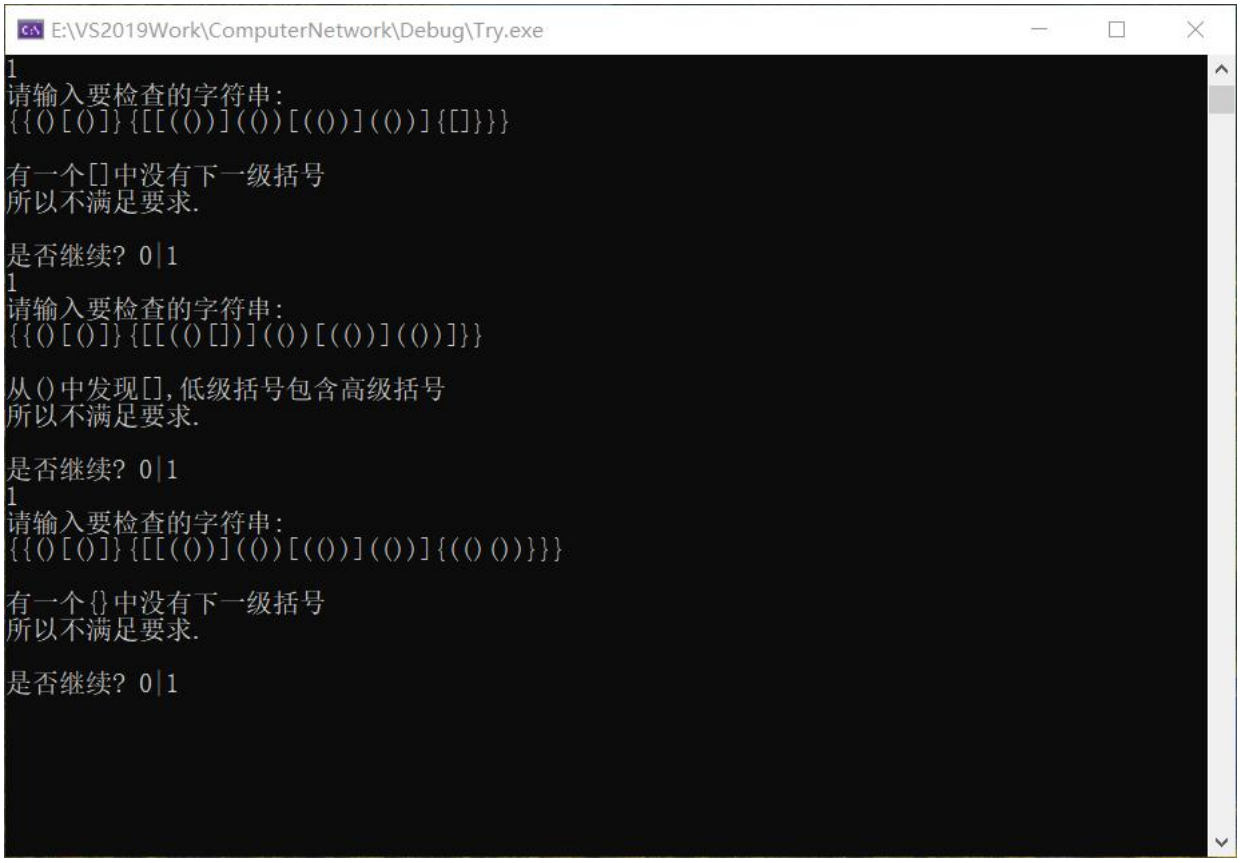
2、空间复杂度分析：

程序中只有前括号入栈，所以如果匹配成功，前括号就有 n/2 个，空间复杂度是 O(n/2)；如果匹配不成功，最坏情况下，全都是前括号，空间复杂度是 O(n)；综上，最坏的复杂度是 O(n)。

六、程序运行结果

我的实验环境是：WIN10+VSCODE/VS2019+Cpp20
以下是我的实验用例和实验结果：
正确的例子：
1、{{()[()]}{[[(())](())[(())](())]{(()())[()]}}}
2、{{()[()]}{[[(())](())[(())](())]}}
3、[()()((((()))))[[()][[[()]]][()]][[()]][(())]]
错误的例子和人为设置的错误：
1、{{()[()]}{[[(())](())[(())](())]{[]}}}：有一个[]没有()
2、{{()[()]}{[[(()[])](())[(())](())]}}:有一个()包含了[]
3、{{()[()]}{[[(())](())[(())](())]{(()())}}}：有一个{}中没有[]

程序实验的结果：



可以看到程序运行的结果是正确的，并且可以检查出相应的错误。