数据结构实验报告
张恒硕　　2212266　　智科 2 班

实验目标：顺序栈，链式栈，并利用栈实现数制转换与括号匹配，并测试。
（以下分为栈、顺序栈、链式栈、数制转换、括号匹配五部分，其中数制转换的代码与测试包含在前两部分中）

零、栈

栈（Stack）是一种后进先出（LIFO）的数据结构，只允许在一端（称为栈顶）进行插入和删除操作。

一、顺序栈

实验原理：

顺序栈是一种基于数组实现的数据结构，用于模拟栈的行为。其原理主要包括以下几个方面：

1、数据存储：使用数组来存储数据元素，数组的每个元素都代表栈中的一个元素。数组的第一个元素作为栈底，最后一个元素作为栈顶。栈的大小由数组长度决定。

2、栈顶指针：有一个用于指示栈顶元素在数组中位置的栈顶指针。其初始值为-1，表示栈为空。当有元素入栈时加 1，当有元素出栈时减 1。

3、入栈操作：首先检查栈是否已满，如果未满，则将新元素存储在数组的栈顶位置，并将栈顶指针加 1。

4、出栈操作：首先检查栈是否为空，如果非空，则将栈顶元素取出并返回，同时将栈顶指针减 1。

5、栈的容量：顺序栈的容量取决于数组的长度。由于数组的大小是固定的，因此顺序栈的容量也是固定的。在实际应用中，根据需求选择合适的数组长度。

代码：

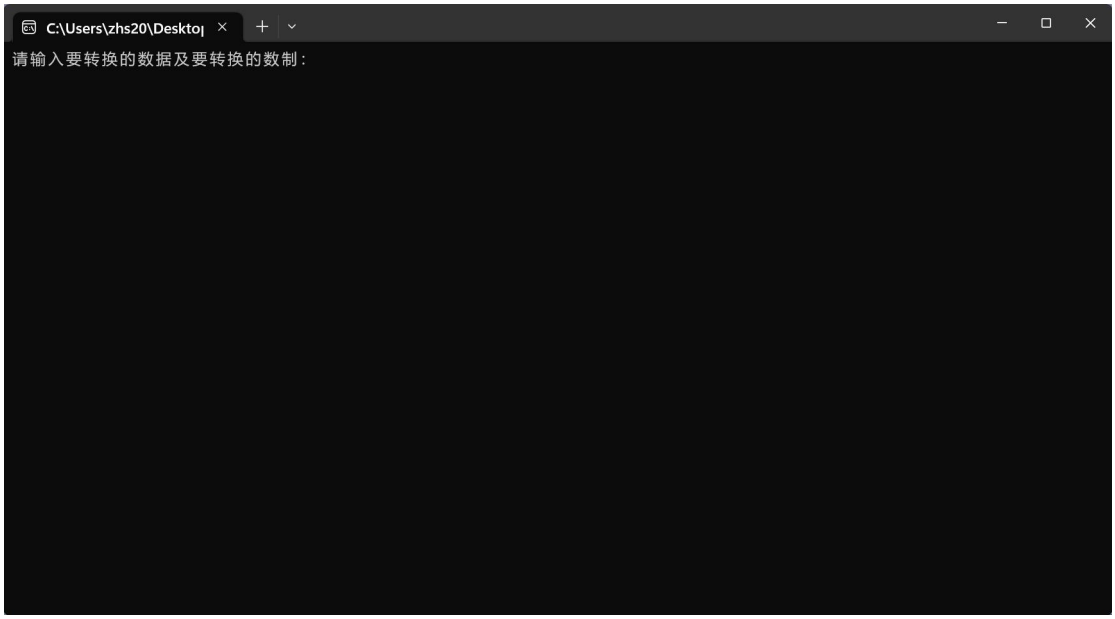
```
#include <iostream>
using namespace std;
const int MAX_SIZE = 100; //栈的最大容量
class Stack { //顺序栈类
private:
    int data[MAX_SIZE];
    int top_index;
public:
    Stack() { //构造函数
        data[0] = '\0';
        top_index = -1;
    }
    ~Stack(){} //析构函数
    bool full() { //判断满
        return top_index == MAX_SIZE - 1;
    }
```

```
    bool empty() { //判断空
        return top_index == -1;
    }
    void push(int val) { //入栈
        if (full()) {
            cout << "栈已满，无法入栈" << endl;
            return;
        }
        top_index++;
        data[top_index] = val;
    }
    int pop() { //出栈
        if (empty()) {
            cout << "栈已空，无法出栈" << endl;
            return -1;
        }
        int a = data[top_index];
        top_index--;
        return a;
    }
    int top() { //取出栈顶元素（不出栈）
        if (empty()) {
            cout << "栈为空，无法取出栈顶元素" << endl;
            return -1;
        }
        return data[top_index];
    }
};
void change_sz(int ori, int sz) { //转换数制
    Stack stack;
    while (ori / sz > 0) {
        stack.push(ori % sz);
        ori /= sz;
    }
    stack.push(ori);
    while (!stack.empty()) {
        cout << stack.pop();
    }
    cout << "(" << sz << "进制)" << endl;
}
int main() {
    int val, sz;
    cout << "请输入要转换的数据及要转换的数制：" << endl;
    cin >> val >> sz;
```

```
    change_sz(val, sz);
    return 0;
}
```

输入样例:
100 8

运行结果:



实现操作:
构造函数
析构函数
判断满
判断空
入栈
出栈
取出栈顶元素（不出栈）

分析:
顺序栈的优点是实现简单、效率高，支持快速入栈和出栈操作。然而，由于顺序栈的容量固定，在某些场景下可能会受到限制。

二、链式栈

实验原理:
链式栈是一种基于链表实现的数据结构，用于模拟栈的行为。其原理主要包括以下几个方面:
1、数据存储：使用链表来存储数据元素，每个链表节点代表栈中的一个元素。链表节点的结构通常包括数据域和指针域，其中数据域存储元素的值，指针域指向下一个节点。

2、栈顶指针：有一个用于指示栈顶元素在数组中位置的栈顶指针，其初始值为 null，表示栈为空。当有元素入栈时，将新元素添加到链表头部，并将栈顶指针指向新节点；当有元素出栈时，将链表头部节点移除，并将栈顶指针指向下一个节点。
3、入栈操作：首先创建一个新的链表节点，并将新元素存储在该节点的数据域中。然后将新节点添加到链表头部，即将新节点的指针域指向原来的链表头部节点，并将栈顶指针指向新节点。
4、出栈操作：首先检查栈是否为空。如果非空，则将链表头部节点移除，即将链表头部节点的下一个节点成为新的链表头部节点，并将栈顶指针指向新的链表头部节点。
5、栈的容量：链式栈的容量取决于链表的长度。由于链表的大小是动态的，因此链式栈的容量也是动态的。在实际应用中，可以根据需求动态调整链表的长度。

代码：

```
#include <iostream>
using namespace std;
class Node { //节点类
public:
    int data;
    Node* next;
};
class Stack { //链式栈类
private:
    Node* top;
public:
    Stack() { //构造函数
        top = NULL;
    }
    ~Stack() { //析构函数
        while (top != NULL) {
            Node* temp = top;
            top = top->next;
            delete temp;
        }
    }
    bool empty() { //判断空
        return top == NULL;
    }
    void push(int value) { //入栈
        Node* newNode = new Node;
        newNode->data = value;
        newNode->next = top;
        top = newNode;
    }
    int pop() { //出栈
        if (empty()) {
```

```
            cout << "栈已空，无法出栈" << endl;
            return -1;
        }
        Node* temp = top;
        int value = temp->data;
        top = top->next;
        delete temp;
        return value;
    }
    int top_() { //取出栈顶元素（不出栈）
        if (empty()) {
            cout << "栈为空，无法取出栈顶元素" << endl;
            return -1;
        }
        return top->data;
    }
};
void change_sz(int ori, int sz) { //转换数制
    Stack stack;
    while (ori / sz > 0) {
        stack.push(ori % sz);
        ori /= sz;
    }
    stack.push(ori);
    while (!stack.empty()) {
        cout << stack.pop();
    }
    cout << "(" << sz << "进制)" << endl;
}
int main() {
    int val, sz;
    cout << "请输入要转换的数据及要转换的数制：" << endl;
    cin >> val >> sz;
    change_sz(val, sz);
    return 0;
}
```

输入样例:
100 9

运行结果:

实现操作：
构造函数
析构函数
判断空
入栈
出栈
取出栈顶元素（不出栈）

分析：

链式栈的优点是可以动态扩展容量，不需要预先分配固定大小的数组空间。然而，由于需要为每个元素分配额外的指针空间来存放指针域，因此在存储相同数量的元素时，链式栈的空间利用率相对较低。此外，链式栈的操作也比顺序栈稍微复杂一些。

三、数制转换

实验原理：

数制转换是指将一个数字从一个进制转换为另一个进制，可使用栈数据结构实现：
1、初始化一个空栈。
2、将待转换的十进制数除以目标进制数，并将余数压入栈中。
3、更新待转换数为上一步的商。
4、重复步骤 2 和 3，直到待转换数为 0。
5、依次弹出栈中的元素，并将它们按照目标进制的表示方法组合起来，即可得到转换后的数。

分析：

利用栈进行数制转换的优势主要表现在以下几个方面：
1、简化操作过程：使用栈可以将数制转换的过程简化为一系列简单的余数入栈和出栈操作。
2、高效性：使用栈进行数制转换的时间复杂度为 O(n)，其中 n 是待转换数的位数。这是因

为每个数字只需要进行一次入栈和出栈操作，所以整个转换过程的时间复杂度与数字的位数成正比。
3、通用性：利用栈进行数制转换的方法不仅适用于整数，还可以扩展到小数和任意进制之间的转换。通过适当地调整入栈和出栈的规则，我们可以很容易地实现不同类型和进制之间的数制转换。
4、易于扩展：在实际应用中，我们可以将利用栈进行数制转换的方法与其他算法和数据结构相结合，以实现更复杂的功能。

存在的问题：
无法输出正确的大于十的进制的形式。

四、括号匹配

实验原理：
用栈实现括号匹配的原理主要是通过栈的后进先出（LIFO）特性来检查括号序列的正确性。括号包括左括号（如“(”、“[”和“{”）和右括号（如“)”、“]”和“}”）。
1、初始化一个空栈。
2、遍历输入字符串中的每个字符。
3、如果当前字符是左括号，将其压入栈中。
4、如果当前字符是右括号，检查栈顶元素是否为与之匹配的左括号。如果是，将栈顶元素弹出；如果不是，说明匹配失败。
5、重复步骤 2 和 3，直到遍历完整个字符串。
6、最后，如果栈为空，说明所有括号都已正确匹配；否则，说明还有未匹配的左括号，匹配失败。

代码：
（直接使用了栈的头文件）

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;
int main() {
    string str;
    cout << "输入测试样例：" << endl;
    cin >> str;
    stack<char> s;
    for (int i = 0; i < str.size(); i++) {
        if (str[i] == '(' || str[i] == '[' || str[i] == '{') {
            s.push(str[i]);
        }
        else if (str[i] == ')') {
            if (s.top() == '(') {
                s.pop();
            }
```

```
            else {
                break;
            }
        }
        else if (str[i] == ']') {
            if (s.top() == '[') {
                s.pop();
            }
            else {
                break;
            }
        }
        else if (str[i] == '}') {
            if (s.top() == '{') {
                s.pop();
            }
            else {
                break;
            }
        }
    }
    if (s.empty()) {
        cout << "匹配成功" << endl;
    }
    else {
        cout << "匹配失败" << endl;
    }
    return 0;
}
```

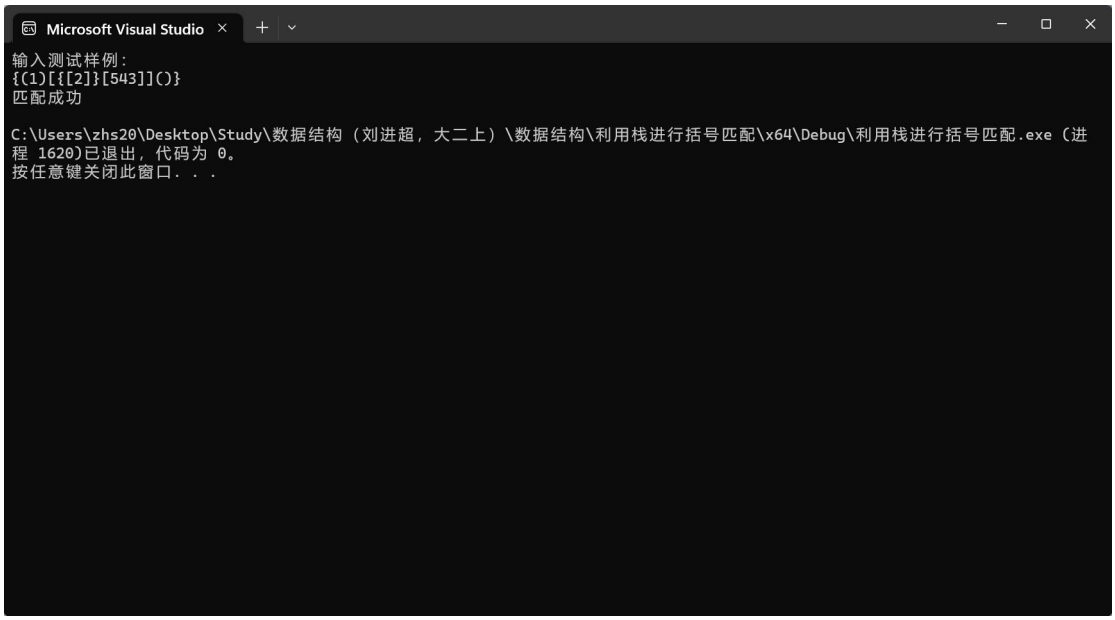
输入样例:

1、{(1)[{[2]}[543]]()}

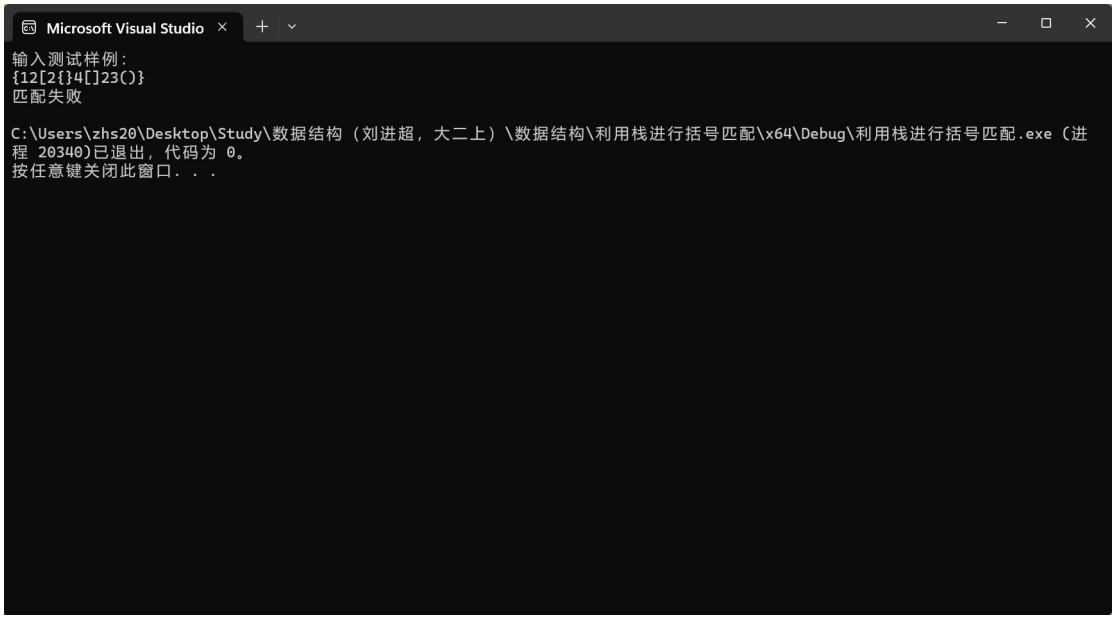
2、{12[2{}4[]23()}

运行结果:

1、



2、



分析：

利用栈进行括号匹配的优势主要表现在以下几个方面：

1、简化操作过程：使用栈可以将括号匹配的过程简化为一系列简单的左括号入栈和遇到右括号时出栈的操作。

2、高效性：使用栈进行括号匹配的时间复杂度为 O(n)，其中 n 是输入字符串的长度。这是因为每个字符只需要进行一次入栈或出栈操作，所以整个匹配过程的时间复杂度与输入字符串的长度成正比。

3、通用性：利用栈进行括号匹配的方法不仅适用于常见的圆括号、方括号和大括号，还可以扩展到其他类型的括号，如尖括号、单引号等。通过适当地调整入栈和出栈的规则，我们可以很容易地实现不同类型括号之间的匹配。

4、易于扩展：在实际应用中，我们可以将利用栈进行括号匹配的方法与其他算法和数据结构相结合，以实现更复杂的功能。