天津大学

程序设计综合实践课程报告

数据结构实验

学生姓名 陈昊昆

学院名称 智算学部

专　　业 软件工程

学　　号 **3021001196**

1. ip 转换

1.1 题目分析

将 32 位 ip 字符串分割成四段

分别转换成十进制

再一起输出

1.2 题目代码（带注释）

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

using namespace std;

string binary_to_decimal(string binary) {
    //  将一个长度为 32 的二进制数转换成由点分隔的四个十进制数，并返回字符串表示的结果
    // 将二进制字符串分隔成四个字节（每个字节 8 位）
    string byte1 = binary.substr(0, 8);
    string byte2 = binary.substr(8, 8);
    string byte3 = binary.substr(16, 8);
    string byte4 = binary.substr(24, 8);

    // 将每个字节转换为十进制数字
    int dec1 = stoi(byte1, nullptr, 2);
    int dec2 = stoi(byte2, nullptr, 2);
    int dec3 = stoi(byte3, nullptr, 2);
    int dec4 = stoi(byte4, nullptr, 2);

    // 将四个数字组合成点分十进制表示的 ip 地址
    string ip = to_string(dec1) + "." + to_string(dec2) + "." + to_string(dec3) + "." + to_string(dec4);
```

```
    return ip;
}

int main() {
    int n;
    cin >> n;  // 输入样例个数

    for (int i = 0; i < n; i++) {
        string binary_ip;
        cin >> binary_ip;

        string decimal_ip = binary_to_decimal(binary_ip);
        cout << decimal_ip << endl;
    }

    return 0;
}
```

2. 进制转换

2.1 题目分析

根据进制转换的规则，十进制的数对进制 R 取余数

然后除以 R

循环上面的过程

2.2 题目代码（带注释）

```
#include <bits/stdc++.h>
using namespace std;
string decimalToRadix(int decimal, int radix);

int main() {
    int n, R;
    while (cin >> n) {
        cin >> R;
        string result = decimalToRadix(abs(n), R);
        if (n < 0) cout << "-";
        cout << result << endl;
    }
}
// 将十进制数 decimal 转换成 radix 进制数的字符串表示
string decimalToRadix(int decimal, int radix) {
    string result = "";    //先将结果置空
    while (decimal > 0) {
        int remainder = decimal % radix;    //取进制 R 的余数
        if (remainder < 10) {
            result = (char)('0' + remainder) + result;
        } else {
            result = (char)('A' + remainder - 10) + result;
        }
        decimal /= radix;
    }
    return result;
}
```

3. 简单计算器

3.1 题目分析

关键是将输入的中缀表达式转化成后缀表达式

通过栈后进先出来实现

同时还要注意计算的优先级：先乘除，后加减

3.2 题目代码（带注释）

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

using namespace std;

// 运算符的优先级
int priority(char op) {
    if (op == '+' || op == '-') return 1;
    else if (op == '*' || op == '/') return 2;
    else return 0;
}

// 计算操作数 1 op 操作数 2 的结果
double calculate(double operand1, double operand2, char op) {
    switch (op) {
        case '+':
            return operand1 + operand2;
        case '-':
            return operand1 - operand2;
        case '*':
            return operand1 * operand2;
        case '/':
            return operand1 / operand2;
        default:
            return 0.0;
    }
}

// 将中缀表达式转化为后缀表达式
string infixToPostfix(string infix) {
```

```
    string postfix = "";
    stack<char> s;
    for (int i = 0; i < infix.length(); i++) {
        if (infix[i] >= '0' && infix[i] <= '9') {
            // 如果是操作数，直接加入后缀表达式
            postfix += infix[i];
        } else if (infix[i] == '+' || infix[i] == '-' || infix[i] == '*'
|| infix[i] == '/') {
            // 如果是运算符
            while     (!s.empty()     &&     priority(s.top())     >=
priority(infix[i])) {
                // 栈顶的运算符优先级大于等于当前运算符的优先级，弹出栈顶运算
符加入后缀表达式
                postfix += s.top();
                s.pop();
            }
            // 将当前运算符入栈
            s.push(infix[i]);
        }
    }
    // 将栈中剩余的运算符弹出加入后缀表达式
    while (!s.empty()) {
        postfix += s.top();
        s.pop();
    }
    return postfix;
}

// 计算后缀表达式的值
double postfixCalculate(string postfix) {
    stack<double> s;
    for (int i = 0; i < postfix.length(); i++) {
        if (postfix[i] >= '0' && postfix[i] <= '9') {
            // 如果是操作数，入栈
            s.push(postfix[i] - '0');
        } else if (postfix[i] == '+' || postfix[i] == '-' || postfix[i]
== '*' || postfix[i] == '/') {
            // 如果是运算符，弹出栈顶的两个操作数进行计算，将计算结果入栈
            double operand2 = s.top();
            s.pop();
            double operand1 = s.top();
            s.pop();
            s.push(calculate(operand1, operand2, postfix[i]));
        }
```

```
    }
    return s.top();
}

int main() {
    string infix;
    while (getline(cin, infix) && infix != "0") {
        string postfix = infixToPostfix(infix);
        double result = postfixCalculate(postfix);
        printf("%.2f\n", result);
    }
    return 0;
}
```

4. 队列和栈

4.1 题目分析

使用标准库中的队列和栈 实现 **pop** 和 **push**

输出时队列正常输出

栈需要再进行逆序之后输出

4.2 题目代码（带注释）

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 105;
int n;
queue<int> q;
stack<int> stk;

void clear() {
    while (!q.empty()) q.pop();
    while (!stk.empty()) stk.pop();
}

int main() {
    int T;
    cin >> T;
    while (T--) {
        clear();
        cin >> n;
        string op;
        int k;
        bool qflag = true, stkflag = true;
        while (n--) {
            cin >> op;
            if (op == "push") {
                cin >> k;
                q.push(k);
                stk.push(k);
            } else {
```

```
                if (q.empty() || stk.empty()) {
                    qflag = false;
                    stkflag = false;
                    continue;
                }
                if (op == "pop") {
                    q.pop();
                    stk.pop();
                }
            }
        }
        if (qflag) {
            int size = q.size();
            for (int i = 0; i < size; i++) {
                cout << q.front() << (i == size - 1 ? "\n" : " ");
                q.push(q.front());
                q.pop();
            }
        } else {
            cout << "error\n";
        }
        if (stkflag) {
            int size = stk.size();
            int *m = new int[size];
            for (int i = 0; i < size; i++) {
                m[i] = stk.top();
                stk.pop();
            }
            for (int i = size-1; i >=0; i--) {
                cout << m[i] << " ";
            }
            cout << endl;
        } else {
            cout << "error\n";
        }
    }
    return 0;
}
```

5. 报数

5.1 题目分析

创建一个长度为 n 的数组 students，其中 students[i]表示第 i 个同学是否已经被移出队伍。

count 变量记录当前报数，

index 变量记录当前同学的下标。

在循环中使用模运算更新 index 的值，使得 index 的范围在[0, n-1]之间循环。

在每次循环中，如果当前同学还没有被移出队伍，则 count++，如果 count 是 7 的倍数，则当前同学被移出队伍，students[index]置为 1，并将 remaining 减 1。

最后遍历 students 数组，输出未被移出队伍的同学的编号。

5.2 题目代码（带注释）

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int T;
    cin >> T;
    while (T--) {
        int n;
        cin >> n;
        vector<int> students(n, 0);
        int count = 0; // 报数
        int index = 0; // 当前同学的下标
```

```
        int remaining = n; // 剩余的同学数
        while (remaining >= 7) {
            if (students[index] == 0) {
                count++;
                if (count % 7 == 0) {
                    students[index] = 1;
                    remaining--;
                }
            }
            index = (index + 1) % n;
        }
        for (int i = 0; i < n; i++) {
            if (students[i] == 0) {
                cout << i + 1 << " ";
            }
        }
        cout << endl;
    }
    return 0;
}
```

6. 二叉树遍历 1

6.1 题目分析

根据先序遍历字符串建立二叉树，可以采用递归方式。

对建立好的二叉树进行中序遍历，可以采用非递归方式，使用栈模拟。

6.2 题目代码（带注释）

```
#include <bits/stdc++.h>
using namespace std;

struct TreeNode {
    char val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(char x) : val(x), left(nullptr), right(nullptr) {}
};

// 递归建树
TreeNode* buildTree(string& str, int& index) {
    if (str[index] == '#') {
        index++;
        return nullptr;
    }
    TreeNode* root = new TreeNode(str[index++]);
    root->left = buildTree(str, index);
    root->right = buildTree(str, index);
    return root;
}

// 非递归中序遍历
void inOrderTraversal(TreeNode* root) {
    stack<TreeNode*> stk;
    TreeNode* p = root;
    while (p != nullptr || !stk.empty()) {
        while (p != nullptr) {
            stk.push(p);
            p = p->left;
```

```
        }
        p = stk.top();
        stk.pop();
        cout << p->val << ' ';
        p = p->right;
    }
    cout << endl;
}

int main() {
    string str;
    while (getline(cin, str)) {
        if (str.empty()) break;
        int index = 0;
        TreeNode* root = buildTree(str, index);
        inOrderTraversal(root);
    }
    return 0;
}
```

7. 复原二叉树

7.1 题目分析

先通过先序遍历和中序遍历的结果构造二叉树

关键是利用中序遍历分割出左右子树

然后递归的构造左右子树

最后得到后续遍历的结果

7.2 题目代码（带注释）

```
#include <bits/stdc++.h>
using namespace std;

struct TreeNode {
    char val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(char x) : val(x), left(NULL), right(NULL) {}
};

// 递归构建二叉树，返回根节点指针
TreeNode* buildTree(string preorder, string inorder, int preStart, int
inStart, int inEnd) {
    // 递归结束条件，如果当前子树为空，返回空指针
    if (preStart > preorder.size() - 1 || inStart > inEnd) {
        return nullptr;
    }

    auto root = new TreeNode(preorder[preStart]);

// 在中序遍历中查找当前根节点的位置，以便分割左右子树
    int inIndex = 0;
    for (int i = inStart; i <= inEnd; ++i) {
        if (inorder[i] == root->val) {
            inIndex = i;
            break;
        }
```

```
    }
// 递归构建左右子树，更新当前节点的左右孩子指针
    root->left = buildTree(preorder, inorder, preStart + 1, inStart,
inIndex - 1);
    root->right = buildTree(preorder, inorder, preStart + inIndex -
inStart + 1, inIndex + 1, inEnd);

    return root;
}

void postorderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    cout << root->val;
}

int main() {
    string preorder, inorder;
    while (cin >> preorder >> inorder) {
        TreeNode*   root   =   buildTree(preorder,   inorder,   0,   0,
inorder.size() - 1);
        postorderTraversal(root);
        cout << endl;
    }
    return 0;
}
```

8. 合并果子（堆）

8.1 题目分析

将所有果子数目存入小根堆中

然后循环合并(n-1)次

每次取出小根堆中排名前两的元素，累计体力值

再将合并后的元素放回小根堆中

最后输出累计的体力值即为所求。

8.2 题目代码（带注释）

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    priority_queue<int, vector<int>, greater<int>> pq; // 小根堆，存储果堆的大小（个数）
    int n;
    cin >> n;

    for (int i = 0; i < n; i++) {
        int m;
        cin >> m;
        pq.push(m); // 将每个果堆的大小存入小根堆中
    }

    long long ans = 0;

    while (pq.size() > 1) { // 只要堆的大小大于 1，就继续合并
        int a = pq.top(); pq.pop();
        int b = pq.top(); pq.pop();
        ans += a + b; // 合并这两个堆，记录体力耗费
        pq.push(a + b); // 将新的果堆大小推回小根堆，继续等待合并
    }

    cout << ans << endl; // 输出最小体力耗费值
```

```
    return 0;
}
```