栈的应用——实验报告

- **班级**：通信2301
- **学号**：U202342641
- **姓名**：陶宇轩

一、编程实验名称与内容概述

- **实验名称**：栈的应用
- **内容概述**：设以字符序列 A、B、C、D作为顺序栈st的输入，

  利用 push(进栈)和pop(出栈)操作，输出所有可能的出栈序列并编程实现整个算法

二、程序设计思路

数据结构

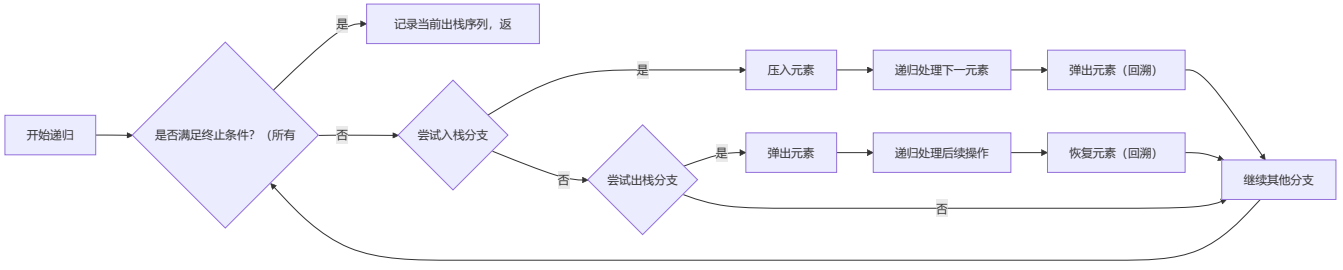
- 使用栈来模拟当前栈的状态
- 递归函数参数包括当前栈、当前的输出序列、当前已入栈的元素数目

算法步骤

1. 初始时，栈为空，输出序列为空，已入栈 0 个元素
2. 在每一步：
    1. 如果还有未入栈的元素，则可以选择入栈
    2. 如果栈非空，则可以选择出栈
3. 当所有元素都已入栈且栈为空时，记录当前的输出序列

三、代码说明

流程图



递归函数 `backtrack`

参数

- `stack<char>& s`：当前栈的状态
- `string output`：当前已生成的出栈序列
- `int next_input`：下一个待处理的输入元素索引

代码

```
void backtrack(stack<char>& s, const string& output, int next_input) {
    // 递归终止条件：所有元素已处理且栈为空
    if (next_input == input.size() && s.empty()) {
        all_sequences.push_back(output);
        return;
    }

    // 入栈分支
    if (next_input < input.size()) {
        s.push(input[next_input]);          // 压入当前元素
        backtrack(s, output, next_input + 1); // 递归处理下一元素
        s.pop();                            // 回溯：恢复栈状态
    }

    // 出栈分支
    if (!s.empty()) {
        char c = s.top();
        s.pop();                            // 弹出栈顶元素
        backtrack(s, output + c, next_input); // 递归处理后续操作
        s.push(c);                          // 回溯：恢复栈状态
    }
}
```

主函数

- 初始化空栈并启动递归
- 遍历并输出所有生成的出栈序列

四、运行结果与复杂度分析

运行结果

```
所有可能的出栈序列：
DCBA
CDBA
CBDA
CBAD
BDCA
BCDA
BCAD
```

```
BADC
BACD
ADCB
ACDB
ACBD
ABDC
ABCD
```

复杂度分析

- **时间复杂度**
  - 最坏情况下为`O(2^{2n})`通过剪枝优化后实际为`O(Catalan(n) * n)`
  - 其中`Catalan(n)`为第 n 个卡特兰数`Catalan(n) ≈ 4^n / (n^(3/2))`
- **空间复杂度**
  - 递归栈深度为`O(n)`，存储结果需`O(Catalan(n))`空间

五、改进方向与心得体会

改进方向

- **迭代替代递归**：避免递归深度过大导致的栈溢出问题
- **备忘录机制**：记录已处理的子状态，避免重复计算

心得体会

- **栈的应用**：在回溯算法中使用了栈的“后进先出”特性
- **回溯法**：通过递归+状态恢复穷举解空间
- **剪枝重要性**：合理剪枝可显著提升算法效率
- 学到了一个新的数学知识：**卡特兰数**
  - 了解了其在 合法括号序列，路径计数等问题上的应用

后缀表达式2025——实验报告

- **班级**：通信2301
- **学号**：U202342641
- **姓名**：陶宇轩

一、编程实验名称与内容概述

- **实验名称**：后缀表达式2025
- **内容概述**：统计满足特定条件的后缀表达式字符串的数量。条件包括：字符串由4个数字和3个运算符组成，构成合法的后缀表达式，计算结果等于给定的K，并且运算过程中没有除零或模零的情况。

二、程序设计思路

数据结构

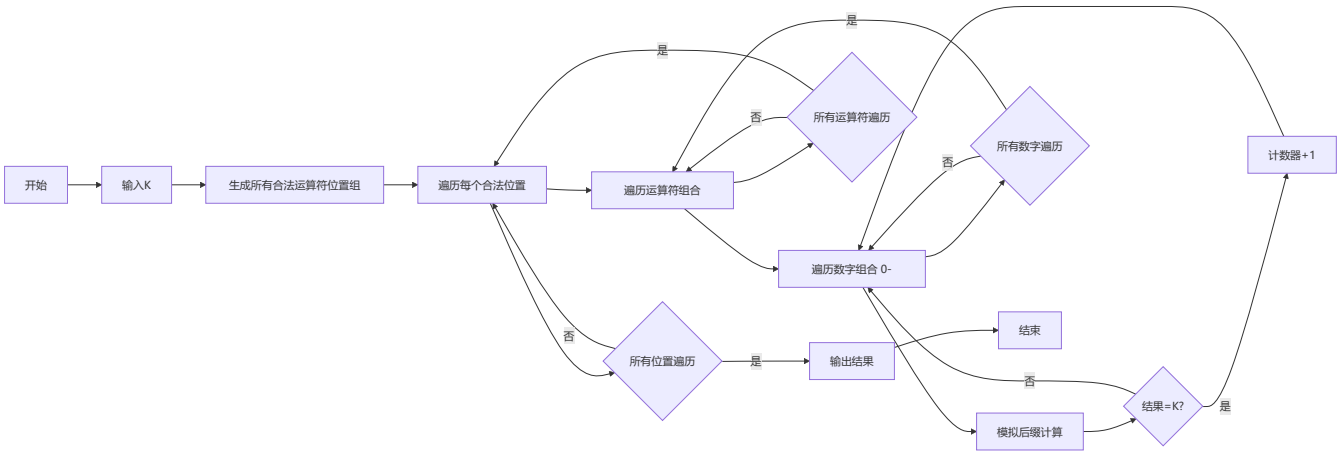
1. **向量（`vector`）**：存储运算符位置、数字映射关系及有效位置组合
2. **栈（`stack`）**：模拟后缀表达式的计算过程
3. **数组**：预定义运算符列表（`ops_list`）和数字位置映射

算法步骤

1. **生成合法运算符位置**
   - 遍历所有可能的3个运算符位置组合（共 C(7,3) = 35 种）通过栈模拟验证合法性
   - 合法条件：遍历表达式时，栈大小始终 ≥2 且最终栈大小为 1
2. **遍历运算符与数字组合**
   - 对每个合法位置组合，生成所有可能的运算符 5^3=125 种 和 4位数字 0 - 9999 共 10000 种组合
3. **模拟表达式计算**
   - 用栈处理表达式，遇到数字压栈，遇到运算符弹出栈顶两元素计算后压栈结果
   - 处理除零错误，记录有效结果

三、代码说明

流程图



核心函数

1. 合法性验证函数 `is_valid`

- 通过栈大小动态验证表达式结构，而非实际计算值
- 剪枝提前过滤掉无效位置组合

```cpp
bool is_valid(const vector<int>& pos) {
    vector<bool> is_op(7, false);  // 标记运算符位置
    for (int p : pos) is_op[p] = true;

    int stack_size = 0;  // 模拟栈大小
    for (int i = 0; i < 7; i++) {
        if (is_op[i]) {  // 遇到运算符
            if (stack_size < 2) return false;  // 栈元素不足，非法
            stack_size--;  // 弹出两个元素，压入一个结果
        } else {  // 遇到数字
            stack_size++;  // 压栈操作
        }
    }
    return stack_size == 1;  // 最终栈需只剩一个元素
}

```

2. 数字位置预处理

- 将离散的数字位置（如 `[0,2,4,6]`）映射到连续的4位数字索引（0→0, 2→1, 4→2, 6→3）

```cpp
vector<int> digits_pos, digit_map(7, -1);
for (int i = 0; i < 7; i++) {
    if (!is_op[i]) {  // 数字位置处理
        digits_pos.push_back(i);  // 记录数字索引顺序
        digit_map[i] = digits_pos.size() - 1;  // 建立位置到数字索引的映射
    }
}
```

3. 运算符预处理

- 将离散的运算符位置（如`[1,3,5]`）映射到运算符组合索引（1→0号运算符，3→1号，5→2号）

```
vector<int> op_indices(7, -1);
for (int k = 0; k < 3; k++) {
    op_indices[valid_pos[k]] = k;  // 标记每个运算符在组合中的顺序
}
```

4. 表达式计算核心逻辑

- 计算表达式

```
for (int i = 0; i < 7 && valid; i++) {
    if (op_indices[i] != -1) { // 处理运算符
        if (sp < 2) { valid = false; continue; }
        int a = stack[sp-2], b = stack[sp-1];
        sp -= 2;
        char op = ops[op_indices[i]];

        if ((op == '/' || op == '%') && b == 0) {
            valid = false;
            continue;
        }

        int res;
        switch(op) {
            case '+': res = a + b; break;
            case '-': res = a - b; break;
            case '*': res = a * b; break;
            case '/': res = a / b; break;
            case '%': res = a % b; break;
        }
        stack[sp++] = res;
    } else { // 处理数字
        stack[sp++] = digits[digit_map[i]];
    }
}
```

5. 数字组合生成

- 包含所有4位数的排列组合（0000-9999）

```
for (int num = 0; num < 10000; num++) {
    vector<int> digits = {
        num / 1000,         // 千位
        (num / 100) % 10,   // 百位
        (num / 10) % 10,    // 十位
        num % 10            // 个位
    };
}
```

四、运行结果与复杂度分析

运行结果

```
6561
5
```

复杂度分析

时间复杂度

- 生成合法位置组合：`O(7^3)`
- 遍历运算符组合：`O(5^3=125)`
- 遍历数字组合：`O(10^4=10000)`
- 总复杂度：`O(7^3 × 125 × 10000 × 7) ≈ 8.75×10^9` 次操作

空间复杂度

- 存储有效位置组合：`O(C)`（C为合法位置数）
- 临时存储数字和运算符：`O(1)`

五、改进方向与心得体会

改进方向

- **剪枝优化**：在生成数字组合时，若中间计算结果已超过 `K` 的可能范围，提前终止计算
- **并行计算**：将运算符和数字组合的遍历拆分为多线程任务，提升效率

心得体会

- **后缀表达式特性**：通过栈模拟计算
- **暴力枚举的局限性**：一开始的时候尝试用暴力枚举，但时间复杂度较高，后续使用剪枝优化
- **预处理的思想**：数字位置和字符位置的预处理
    - 将离散的位置映射到连续的索引
    - 将O(n)计算转为O(1)查询

斐波那契编码——实验报告

- **班级**：通信2301
- **学号**：U202342641
- **姓名**：陶宇轩

一、编程实验名称与内容概述

- **实验名称**：斐波那契编码
- **内容概述**：按斐波那契编码规则进行编码，从键盘输入正整数，程序运行后输出对应的斐波那契编码，以下是编码规则：
  1. 将输入的正整数分解为不连续的斐波那契数之和。
  2. 按斐波那契数列的顺序（从小到大）生成二进制位，每个斐波那契数对应一个位：选中的数为 1，未选中的数为 0。
  3. 在二进制表示的末尾添加一个 1 作为结束符。

二、程序设计思路

数据结构

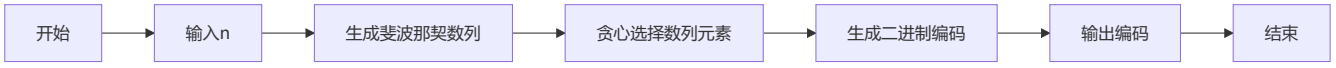
- `vector<int> fib`：用于存储斐波那契数列
- `vector<bool> selected`：标记选中的斐波那契数

算法步骤

1. 生成斐波那契数列直至超过输入值 n
2. 从最大斐波那契数开始，依次选择不超过剩余值的数，标记并减去该数
3. 根据标记数组生成二进制字符串，末尾添加终止符 1

三、代码说明

流程图



主函数

1. 输入整数 n
2. 生成斐波那契数列

```cpp
vector<int> fib = {1, 2};
while (true) {
    int next = fib.back() + fib[fib.size()-2];
    if (next > n) break;
    fib.push_back(next);
}
```

3. 贪心选择斐波那契数

```cpp
int remaining = n;
for (int i = max_idx; i >= 0; --i) {
    if (fib[i] <= remaining) {
        selected[i] = true;
        remaining -= fib[i];
    }
}
```

4. 生成编码并输出

```cpp
string code;
for (bool used : selected) {
    code += used ? '1' : '0';
}
code += '1'; // 添加终止符 1
```

四、运行结果与复杂度分析

运行结果

复杂度分析

- **时间复杂度**：
  - 生成斐波那契数：`O(log(n))`
  - 贪心选择斐波那契数：`O(log(n))`
  - 总时间复杂度：`O(log(n))`
- **空间复杂度**：`O(log(n))`，存储斐波那契数列和标记数组

五、改进方向与心得体会

改进方向

- 寻找复杂度更低的算法

心得体会

- 用预生成的斐波那契数列可以方便调用，节省时间

线性表编程实验2025——实验报告

- **班级**：通信2301
- **学号**：U202342641
- **姓名**：陶宇轩

一、编程实验名称与内容概述

实验名称：基于线性表的命令行操作实现
实验内容：设计一个存储整数（测试用例仅使用不超过1000的非负整数）的线性表（建议用链表实现，也可以用STL库），根据标准输入对线性表进行操作。

二、程序设计思路

数据结构

- 用`std::vector<int>`模拟线性表

命令实现分析

1. **创建线性表**
   命令：C m
   - m 为非负整数，表示创建一个长度为 m 的线性表，并依次填充 0 到 m-1 的整数。
   - m=0 时创建空表。
   - **测试用例保证 C 是第一个命令且合法。**
2. **插入整数**
   命令：I x y
   - 在位置 x（从0开始编号）插入值 y。
   - 若 x 不合法（如越界），输出 X 并终止。
3. **删除单个元素**
   命令：D x
   - 删除位置 x 的元素。
   - 若 x 不合法，输出 X 并终止。
4. **批量删除元素**
   命令：E x y
   - 删除区间 [x, y]（含 x 和 y）内的所有元素。
   - 若 x 或 y 不合法，输出 X 并终止。
5. **清除线性表**
   命令：CLR
   - 清空线性表，无输出。
6. **返回线性表长度**
   命令：LEN

- 输出当前线性表的长度并终止程序。例如，长度为3时输出 3。

7. **返回指定位置元素**
   命令：GET pos
   - 输出位置 pos 的元素值并终止程序。
   - 若 pos 不合法，输出 X 并终止。
8. **输出线性表**
   命令：P
   - 输出线性表的所有元素（空格分隔）。
   - 空表输出 EMPTY。

三、代码说明

关键代码段与注释

1. 初始化命令（C m）

```
// 处理初始化命令C，格式：C m（创建包含0到m-1的列表）
getline(cin, line);
istringstream iss(line);
string cmd;
iss >> cmd;
if (cmd != "C") {
    cout << "X" << endl;  // 首命令非C则报错
    return 0;
}
int m;
iss >> m;
list.clear();
for (int i = 0; i < m; ++i) {
    list.push_back(i);  // 初始化0,1,...,m-1
}
```

2. 插入操作（I x y）

```
// 处理插入命令I，格式：I x y（在位置x插入y）
if (cmd == "I") {
    int x, y;
    iss >> x >> y;
    if (x < 0 || x > list.size()) {  // 检查x有效性
        cout << "X" << endl;
        return 0;
    }
    list.insert(list.begin() + x, y);
}
```

3. 删除单个元素（D x）

```
// 处理删除命令D，格式：D x（删除位置x的元素）
else if (cmd == "D") {
    int x;
    iss >> x;
    if (x < 0 || x >= list.size()) {
        cout << "X" << endl;
        return 0;
    }
    list.erase(list.begin() + x);
}
```

4. 批量删除操作（E x y）

```
// 处理范围删除命令E，格式：E x y（删除x到y的元素）
else if (cmd == "E") {
    int x, y;
    iss >> x >> y;
    if (x < 0 || y >= list.size() || x > y) {  // 检查范围有效性
        cout << "X" << endl;
        return 0;
    }
    list.erase(list.begin() + x, list.begin() + y + 1);
}
```

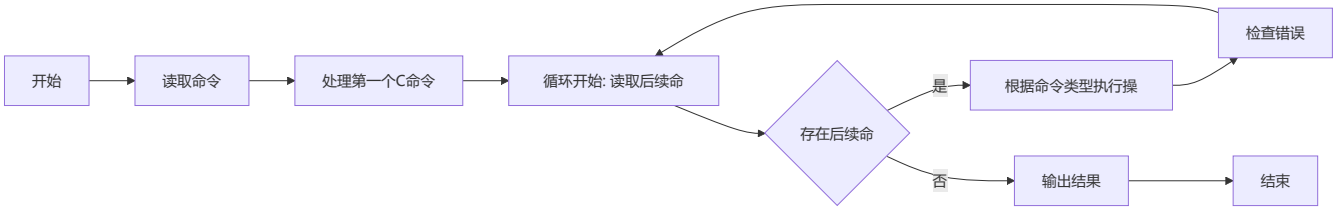
5. 返回指定位置元素 (GET pos)

```
// 处理元素获取命令GET，格式：GET pos（输出指定位置元素）
else if (cmd == "GET") {
    int pos;
    iss >> pos;
    if (pos < 0 || pos >= list.size()) {
        cout << "X" << endl;
        return 0;
    }
    cout << list[pos] << endl;
    terminated = true;
}
```

6. 输出线性表（P）

```
// 处理打印命令P（输出所有元素）
else if (cmd == "P") {
    if (list.empty()) {
        cout << "EMPTY" << endl;
    } else {
        for (size_t i = 0; i < list.size(); ++i) {
            if (i > 0) cout << " ";
            cout << list[i];
        }
        cout << endl;
    }
    terminated = true;
}
```

流程图



四、运行结果与复杂度分析

时间复杂度分析

时间复杂度

命令	时间复杂度	说明
C m	O(m)	初始化需要填充m个元素
I x y	O(n)	插入需要移动元素
D x	O(n)	删除需要移动元素
E x y	O(n)	批量删除需要移动元素
CLR	O(1)	直接清空容器
LEN	O(1)	直接返回size()
GET pos	O(1)	直接访问元素
P	O(n)	遍历所有元素输出

空间复杂度

- **空间复杂度**：O(m)，其中m是线性表的最大长度

五、改进方向与心得体会

改进方向

- 使用链表代替`vector`，可将插入和删除操作的时间复杂度降低到`O(1)`
- 在命令解析时提前判断参数数量是否合法
- 添加更多异常处理逻辑，例如输入非数字字符时的容错

心得体会

- `vector`在随机访问和简单操作中表现良好，但插入/删除频繁时效率较低