数据结构：栈 (Data Structure: Stack)

I. 栈的核心概念与定义 (Core Concepts and Definition)

1. **引言：栈与队列 (Introduction: Stacks and Queues)**
   - **English:** Stacks and queues are linear lists where insertions and deletions are restricted to the "endpoints" of the list. They are essentially linear lists with limited operations.
   - **中文:** 栈和队列是特殊的线性表，它们的插入和删除操作被限制在表的“端点”进行。本质上，它们是操作受限的线性表。
   - **操作对比 (Operations Comparison):**
     - **List (列表):** `Insert(L, i, x)` (在第i个位置插入), `Delete(L, i)` (删除第i个位置元素)。
     - **Stack (栈):** `Insert(S, n+1, x)` (总是在末尾插入-入栈), `Delete(S, n)` (总是在末尾删除-出栈)。
     - **Queue (队列):** `Insert(Q, n+1, x)` (总是在末尾插入-入队), `Delete(Q, 1)` (总是在开头删除-出队)。
2. **栈的正式定义 (Formal Definition of a Stack)**
   - **English:** A stack is a linear list where insertions and deletions are restricted to the end of the list. This end is called the **top**, and the other end is called the **bottom**. A stack with no elements is called an **empty stack**.
   - **中文:** 栈是一种只允许在表的一端进行插入和删除操作的线性表。允许操作的这一端被称为 **栈顶 (top)**，另一端固定不动，被称为 **栈底 (bottom)**。不含任何元素的栈称为 **空栈 (empty stack)**。
3. **栈的工作原则：后进先出 (LIFO - Last In, First Out)**
   - **English:** A stack processes data elements according to the "Last In, First Out" (LIFO) principle. The last element pushed into the stack is the first one to be popped out.
   - **中文:** 栈按照“后进先出”(LIFO) 的原则处理数据。这意味着最后进入栈的元素将最先离开。
     - **Push Order (入栈顺序):** $a_1, a_2, ..., an$
     - **Pop Order (出栈顺序):** $an, an-1, ..., a_1$

II. 栈的抽象数据类型 (Abstract Data Type - ADT)

1. **ADT 定义 (ADT Definition)**
   - **English:** The ADT for a Stack includes data elements, data relationships, and basic operations.

    - **Data elements:** $D = \{ a_i \mid a_i \in ElemSet, i=1,2,...,n, n\geq 0 \}$
    - **Data relationship:** $R_1 = \{ <a_i\text{-}1, a_i> \mid a_i\text{-}1, a_i \in D, i=2,...,n \}$ (Linear relationship, where a_n is the top and a_1 is the bottom).
    - **Basic Operations:** ... (see below)
- **中文:** 栈的抽象数据类型包括数据元素、数据关系和基本操作。
    - **数据元素:** $D = \{ a_i \mid a_i \in ElemSet, i=1,2,...,n, n\geq 0 \}$
    - **数据关系:** $R_1 = \{ <a_i\text{-}1, a_i> \mid a_i\text{-}1, a_i \in D, i=2,...,n \}$ (线性关系， a_n 为栈顶， a_1 为栈底)。
    - **基本操作:** ... (见下文)

2. **基本操作详解 (Detailed Basic Operations)**
    - `InitStack(&S)`：
    - **初始化 (Initialize)**. 构造一个空栈S。
    - `DestroyStack(&S)`：
    - **销毁 (Destroy)**. 栈S存在时，销毁它。
    - `StackLength(S)`：
    - **获取长度 (Get Length)**. 返回栈S中的元素个数。
    - `StackEmpty(s)`：
    - **判空 (Check if Empty)**. 若栈S为空，返回 `TRUE` ，否则返回 `FALSE` 。
    - `GetTop(S, &e)`：
    - **获取栈顶元素 (Get Top Element)**. 获取栈S的栈顶元素，并通过 `e` 返回，栈不变。
    - `ClearStack(&S)`：
    - **清空 (Clear)**. 将栈S清空为一个空栈。
    - `Push(&S, e)`：
    - **入栈 (Push / Insert)**. 插入元素 `e` 作为新的栈顶。
    - `Pop(&S, &e)`：
    - **出栈 (Pop / Remove)**. 删除栈S的栈顶元素，并用 `e` 返回其值。
    - `StackTravers(S, visit())`：
    - **遍历 (Traverse)**. 从栈底到栈顶依次对每个元素执行 `visit()` 操作。

III. 栈的实现方式 (Implementation of Stacks)

A. 顺序存储结构 (Sequential Storage Structure - Array)

1. **原理 (Principle):**
    - **English:** A set of consecutive memory cells (an array) is used to store elements from bottom to top. A top pointer (`top`) indicates the position (index) of the top element. The bottom is fixed, while the top changes with push and pop operations.

- **中文:** 使用一组地址连续的存储单元（数组）自底向上存放元素。用一个整型变量 `top` 作为指针，指示栈顶元素在数组中的位置。栈底固定，栈顶随着 `push` 和 `pop` 操作动态变化。
- **状态 (States):**
  - **Empty Stack (空栈):** `top = -1`
  - **Stack Overflow (栈满/上溢):** `top` 达到数组最大索引时再进行入栈。
  - **Stack Underflow (栈空/下溢):** `top` 为-1时再进行出栈。

2. **C语言实现 (C Language Implementation):**
   - **结构定义 (Struct Definition):**

```c
#define MAXSIZE n // n is the maximum possible number of data elements
typedef struct {
    elemtype stack[MAXSIZE];
    int top;
} sqstack;
```

   - **初始化 (Initialization):**

```c
void Initstack(sqstack &s) {
    s.top = -1; // Indicating the stack is empty
}
```

   - **入栈 (Push):**

```c
status push(sqstack &s, elemtype x) {
    if (s.top >= MAXSIZE - 1) // full stack, overflow
        return ERROR;
    s.top++;
    s.stack[s.top] = x; // In the presentation, s.stack[top]=x might be a typo for s.st
    return OK;
}
```

   - **出栈 (Pop):**

```c
status pop(sqstack &s, elemtype &e) {
    if (s.top < 0) // empty stack, underflow
        return ERROR;
    e = s.stack[s.top];
    s.top--;
    return OK;
}
```

- **指针作为参数 (Using Pointers):** 当使用结构指针 `sqstack *s` 时，成员访问从 `s.top` 变为 `s->top` 。

B. 链式存储结构 (Linked Storage Structure - Link List)

1. **原理 (Principle):**
   - **English:** When the maximum required capacity cannot be estimated in advance, using a linked storage structure is an effective method. It is typically implemented with a singly linked list where the head pointer is the top of the stack.
   - **中文:** 当所需容量无法预先估计时，链式存储是一种有效的方法。通常用单链表实现，链表的头指针即为栈顶。
   - **优势 (Advantage):** 动态分配内存，不存在栈满上溢的问题。
2. **C语言实现 (C Language Implementation):**
   - **结构定义 (Struct Definition):**

```
typedef struct snode {
    elemtype data;
    struct snode *next;
} *linkstack;
```

   - **入栈 (Push):**

```
status push(linkstack top, elemtype x) {
    t = (linkstack)malloc(sizeof(snode));
    if (t == NULL) return ERROR; // no enough memory space
    t->data = x;
    t->next = top;
    top = t;
    return OK;
}
```

   - **出栈 (Pop):**

```
status pop(linkstack top, elemtype &e) {
    if (top == NULL) return NULL; // empty stack
    linkstack p = top;
    top = top->next;
    e = p->data;
    free(p);
    return OK;
}
```

好的，遵照您的要求，这里是 **第四部分：栈的应用实例** 的完整、未删节的详细笔记，包含了演示文稿中提供的所有算法思想、代码示例和解释。

IV. 栈的应用实例 (Examples of Stack Applications)

1. 数制转换 (Numbering System Conversion)

- **基本原理 (Fundamental Principle):**
  - **English:** The conversion is based on the division-remainder method, represented by the formula: `N = (N div d) * d + N mod d`. The remainders, which form the digits of the new base number, are generated from the least significant to the most significant. A stack's LIFO property is perfect for reversing this order.
  - **中文:** 转换基于“除d取余法”，其数学原理为 `N = (N div d) × d + N mod d`。计算过程中产生的余数是新数制的各位数字，但产生顺序是从最低位到最高位。栈的“后进先出”特性恰好能将这个顺序反转，得到正确的结果。
- **算法步骤 (Algorithm Steps):**
  i. 初始化一个空栈 `S`。
  ii. 当待转换的十进制数 `N` 不为0时，循环执行：
    - 将 `N % d` (N对d取余) 的结果入栈 `S`。
    - 将 `N` 更新为 `N / d` (N整除d的商)。
  iii. 当循环结束后，只要栈 `S` 不为空，循环执行：
    - 弹出栈顶元素并输出。
- **示例：$(1348)_{10} \rightarrow (2504)_8$**
  - `1348 % 8 = 4` → `4` 入栈。 `N = 168`。
  - `168 % 8 = 0` → `0` 入栈。 `N = 21`。
  - `21 % 8 = 5` → `5` 入栈。 `N = 2`。
  - `2 % 8 = 2` → `2` 入栈。 `N = 0`。循环结束。
  - 出栈顺序：`2`，`5`，`0`，`4`。 结果为 $(2504)_8$。
- **代码实现 (Code Implementation):**

```c
/* 伪代码 (Pseudocode from slide 46) */
void conversion() {
    InitStack(S);
    scanf("%d", &N);
    while (N) { // while (N != 0)
        Push(S, N % 8);
        N = N / 8;
    }
    while (!StackEmpty(S)) {
        Pop(S, e);
        printf("%d", e);
    }
}

/* 完整C程序示例 (Complete C Program from slides 47-49) */
#define stacksize 100;
typedef struct {
    int base[stacksize];
    int top;
} stack;

int push(stack *s, int e) {
    if (s->top >= stacksize - 1) return 0; // Overflow
    s->top++;
    s->base[s->top] = e;
    return 1;
}

int pop(stack *s, int *e) {
    if (s->top < 0) return 0; // Underflow
    *e = s->base[s->top];
    s->top--;
    return 1;
}

main() {
    stack s1;
    int m, e, n;
    s1.top = -1; // Initialize stack
    m = 1348;
    n = 8;
    while (m) {
        push(&s1, m % n);
```

```
            m = m / n;
        }
        while (s1.top != -1) {
            pop(&s1, &e);
            printf("%d", e);
        }
    }
```

2. 括号匹配检查 (Checking Brackets Matching)

- **问题描述 (Problem Description):**
  - **English:** In an expression, brackets must be correctly paired and nested. For example, `([]())` is correct, while `[(])` or `((])` are incorrect. The problem can be solved by considering the "urgency of expectation": an opening bracket creates an expectation for its specific closing counterpart.
  - **中文:** 表达式中的括号必须正确配对和嵌套。例如，`([]())` 是正确的，而 `[(])` 或 `((])` 是错误的。这个问题可以用“期望的紧迫性”概念来描述：一个左括号的出现，就产生了一个对与之匹配的右括号的“期望”。
- **设计思想 (Design Idea):**
  i. **English:** Whenever a left parenthesis appears, push it onto the stack.
  ii. **English:** Whenever a right parenthesis appears, first check if the stack is empty. If it is, this right parenthesis is redundant (mismatch). Otherwise, compare it with the top element of the stack. If they match, pop the left parenthesis. Otherwise, it's a mismatch.
  iii. **English:** When the expression checking ends, if the stack is empty, the parentheses are matched correctly. Otherwise, there are extra "left parentheses".
  - **中文:**
    a. 凡是出现左括号，就入栈。
    b. 凡是出现右括号，首先检查栈是否为空。若栈空，则该右括号多余（不匹配）；否则，将其与栈顶元素比较，若匹配，则弹出栈顶的左括号；若不匹配，则表达式错误。
    c. 当表达式检查结束后，若栈为空，则括号匹配正确；否则，有多余的“左括号”。
- **代码实现 (Code Implementation - for `()` only):**

```
status matching(string& exp) { // Considers parentheses only
    int state = 1, i = 0;
    InitStack(S); // Assume stack S is available
    while (i <= Length(exp) - 1 && state) {
        switch (exp[i]) {
            case '(':
                Push(S, exp[i]);
                i++;
                break;
            case ')':
                if (!stackEmpty(S) && GetTop(S) == '(') {
                    Pop(S, e);
                    i++;
                } else {
                    state = 0; // Mismatch
                }
                break;
            default: // Ignore other characters
                i++; // In the slide, default sets state=0 which might be for expressions w
                     // Here we assume other characters are ignored.
        }
    }
    if (StackEmpty(S) && state) return OK;
    else return ERROR;
}
```

3. 行编辑器问题 (Line Editor Program Problem)

- **问题描述 (Problem Description):**
  - **English:** Implement a simple line editor buffer. The user can type characters, use a special character `#` for backspace (delete the last character), and another character `@` to cancel the entire line (delete all characters on the current line).
  - **中文:** 实现一个简单的行编辑器缓冲区。用户可以输入字符，使用特殊字符 `#` 作为退格符（删除前一个字符），使用 `@` 作为退行符（删除当前行所有字符）。
- **实现思想 (Implementation Idea):**
  - **English:** Use a stack as an input buffer. This avoids storing incorrect input directly.
    - If the character received is a normal character, push it onto the stack.
    - If the character is a backspace (`#`), pop one character from the top of the stack.
    - If the character is a line cancellation (`@`), empty the stack.
  - **中文:** 使用一个栈作为输入缓冲区，这样可以避免立即存储错误输入。
    - 如果收到的是普通字符，则将其入栈。

    - 如果收到的是退格符（`#`），则从栈顶弹出一个字符。
    - 如果收到的是退行符（`@`），则将整个栈清空。
- **代码实现 (Code Implementation):**

```
void LineEdit() {
    InitStack(S);
    ch = getchar();
    while (ch != EOF) {
        while (ch != EOF && ch != '\n') {
            switch (ch) {
                case '#': Pop(S, ch); break;
                case '@': ClearStack(S); break; // Empty stack S
                default: Push(S, ch); break;
            }
            ch = getchar(); // Receive next character
        }
        // ... Transfer the characters in the stack to the data area;
        ClearStack(S); // Reset S to an empty stack for the next line
        if (ch != EOF) ch = getchar();
    }
    DestroyStack(S);
}
```

4. 迷宫求解 (Maze Solving)

- **基本思想 (Basic Idea):**
    - **English:** It's an "exhaustive solution" using backtracking. The core logic is trial and error.
        - If the current position is "passable", it is included in the path (pushed to a stack) and we continue to move forward.
        - If the current position is "impassable" (a wall or already visited), it is removed from the path (popped from the stack), and we go back (backtrack) to the previous position to change direction and continue exploring.
    - **中文:** 这是一种使用回溯法的“穷举求解”。核心是不断试错。
        - 如果当前位置“可通”，则将其纳入路径（入栈），并继续向前探索。
        - 如果当前位置“不通”(是墙或已走过)，则将其从路径中移除（出栈），我们退回到上一个路口，并换一个方向继续探索。
- **算法伪代码 (Algorithm Pseudocode):**

```
Let the current position be the entrance;
do {
    if (current position is passable) {
        push current position to the stack;
        if (this is the exit) {
            algorithm ends; // Path found
        } else {
            go to its next position (e.g., right) as current position;
        }
    } else { // Current position is impassable
        if (the stack is not empty and the top of the stack has other directions to be expl
            // Backtrack and change direction
            set the new current position to be: the next adjacent block of the top position
        }
        if (the stack is not empty and the top of the stack is now impassable in all direct
            // This path is a dead end, backtrack further
            pop the top position out;
        }
    }
} while (the stack is not empty);
// If the loop ends and exit was not found, there is no path.
```

5. 表达式求值 (Evaluating Expressions)

- **引言 (Introduction):**
  - **English:** Evaluating expressions is a fundamental problem in the compilation of programming languages, and its implementation is a typical example of stack application. The presentation introduces the "operator precedence method," which executes expressions according to the rules of operator precedence relations.
  - **中文:** 表达式求值是程序设计语言编译中的一个基本问题，其实现是栈应用的又一个典型范例。文稿中介绍了一种直观、广泛应用的算法——**“算符优先法”**，它依据算符之间的优先关系，实现对表达式的编译或解释执行。

A. 表达式的表示法 (Expression Notation Methods)

- **三种表示法 (Three Notation Methods):**
  - **Infix Notation (中缀表示法):** The operator is between the operands. This is the standard human-readable format. (e.g., `a + b`)
  - **Prefix Notation (前缀表示法 / 波兰式):** The operator precedes the operands. (e.g., `+ a b`)

- **Postfix Notation (后缀表示法 / 逆波兰式):** The operator follows the operands. (e.g., `a b +`)

- **示例 (Example):** `Exp = a × b + (c - d / e) × f`
  - **Prefix:** `+ × a b × - c / d e f`
  - **Infix:** `a × b + (c - d / e) × f`
  - **Postfix:** `a b × c d e / - f × +`
- **关键特性 (Key Properties):**
  i. **English:** The relative order between operands remains unchanged in all three notations.
  ii. **English:** The relative order of operators is different. In prefix/postfix notations, the order of operators determines the actual order of computation.
  iii. **English:** The infix expression relies on parentheses and precedence rules. When parentheses are removed, the order of operations can become uncertain. Prefix and postfix notations do not need parentheses.
  - **中文:**
    a. 三种表示法中，**操作数的相对次序不变**。
    b. **运算符的相对次序不同**。在前缀/后缀表示法中，运算符的顺序决定了计算的实际顺序。
    c. 中缀表达式依赖括号和优先级规则。若去掉括号，运算次序可能不确定。而前缀和后缀表示法**无需括号**即可表示运算次序。
- **运算规则 (Operational Rules):**
  - **Prefix Rule:** Scan from right to left. The first operator found is applied to the two operands immediately to its right.
  - **Postfix Rule:** Scan from left to right. An operator is applied to the two operands immediately preceding it. The order in which operators appear in the postfix expression is the exact order of operations.

B. 算法一：中缀表达式转换为后缀表达式 (Algorithm 1: Infix to Postfix Conversion)

- **目标 (Goal):** 将人类易读的中缀表达式，转换为计算机易于处理的后缀表达式。
- **核心思想 (Core Idea):** 使用一个**运算符栈**来处理运算符的优先级和括号。
- **转换规则 (Conversion Rules):**
  i. **初始化 (Initialize):** 建立一个运算符栈，并在栈底预置一个低优先级的“哨兵”符号，如 `#`。
  ii. **从左到右扫描 (Scan):** 逐个字符地读取中缀表达式。
  iii. **遇到操作数 (Operand):** 直接将其发送（添加）到后缀表达式的末尾。
  iv. **遇到运算符 (Operator):**
    - 将当前运算符的优先级与栈顶运算符的优先级进行比较。
    - 如果当前运算符的优先级 **高于** 栈顶运算符，则将当前运算符压入栈中。
    - 否则（低于或等于），不断地从栈顶弹出运算符并发送到后缀表达式，直到栈顶运算符的优先级低于当前运算符。然后，将当前运算符压入栈中。

v. **遇到左括号 （ (Left Parenthesis):** 直接压入运算符栈。左括号在栈内的优先级被视为最低，但在与栈外运算符比较时最高，确保任何运算符都能被压入其上。
vi. **遇到右括号 ） (Right Parenthesis):** 不断地从栈顶弹出运算符并发送到后缀表达式，直到遇到左括号（。最后，将这个左括号从栈中弹出（但不发送到后缀表达式），从而消除一对括号。
vii. **扫描完成 (End of Expression):** 当中缀表达式扫描完毕后，将运算符栈中所有剩余的运算符依次弹出，并发送到后缀表达式。

- **代码实现 (Code Implementation):**

```
// Slides 73-74 Pseudocode
void transform(char suffix[], char exp[]) {
    // S is the operator stack, preset with '#'
    // OP is the set of operators
    InitStack(S); Push(S, '#');
    p = exp; ch = *p;

    while (!StackEmpty(S)) { // Loop until the stack is cleared at the end
        if (!IN(ch, OP)) { // If ch is an operand
            Pass(suffix, ch); // Append operand to suffix string
            p++; ch = *p;      // Move to next char in expression
        } else { // If ch is an operator
            // 'A:' block handles the logic for operators
            switch (ch) {
                case '(':
                    Push(S, ch);
                    break;
                case ')':
                    Pop(S, c);
                    while (c != '(') {
                        Pass(suffix, c);
                        Pop(S, c);
                    }
                    break;
                default: // For other operators like +, -, *, /
                    // While stack is not empty and top operator has higher or equal preced
                    while (Gettop(S, c) && precede(c, ch)) {
                        Pass(suffix, c);
                        Pop(S, c);
                    }
                    Push(S, ch);
                    break;
            }
            if (/* not end of expression */) { p++; ch = *p; }
        }
    }
}
```

C. 算法二：后缀表达式的求值 (Algorithm 2: Postfix Expression Evaluation)

- **核心思想 (Core Idea):** 使用一个**操作数栈**。从左到右扫描后缀表达式，遇到数字则入栈，遇到运算符则取出栈顶的两个数字进行计算，并将结果压回栈中。
- **求值规则 (Evaluation Rules):**
  i. **初始化 (Initialize):** 建立一个操作数栈 `S`。
  ii. **从左到右扫描 (Scan):** 逐个读取后缀表达式的字符（直到结束符 `#`）。
  iii. **遇到操作数 (Operand):** 将该操作数压入操作数栈 `S`。
  iv. **遇到运算符 (Operator `optr`):**
    - 从栈中弹出两个数。第一个弹出的为第二操作数（`x2`），第二个弹出的为第一操作数（`x1`）。
    - 执行运算 `x1 optr x2`。
    - 将运算结果压回操作数栈 `S`。
  v. **扫描完成 (End of Expression):** 当表达式扫描完毕，操作数栈中唯一剩下的那个数就是最终的计算结果。
- **代码实现 (Code Implementation):**

```
// Slide 84 Pseudocode
int cal(char suffix_exp[]) {
    // S is the stack of operands, OP is the set of operators
    InitStack(S);
    i = 0; ch = suffix_exp[0];

    while (ch != '#') { // Assuming '#' is the end marker
        if (!IN(ch, OP)) { // If ch is an operand
            Push(S, ch); // Push operand onto the stack
        } else { // if ch is an operator
            x2 = POP(S); // Pop operand 2
            x1 = POP(S); // Pop operand 1
            result = calculate(x1, ch, x2); // Perform operation
            PUSH(S, result); // Push the result back
        }
        i++; ch = suffix_exp[i];
    }
    return GetTop(S); // The final result is at the top of the stack
}
```

D. 算法三：中缀表达式的直接求值 (Algorithm 3: Direct Infix Evaluation)

- **核心思想 (Core Idea):** 同时使用两个栈：一个**操作数栈 (OPND)** 和一个**运算符栈 (OPTR)**。
- **求值规则 (Evaluation Rules):**

i. **初始化 (Initialize):** OPND 为空栈，OPTR 预置一个栈底“哨兵”符号 `#` 。
ii. **从左到右扫描 (Scan):** 逐个读取中缀表达式的字符。
iii. **遇到操作数 (Operand):** 压入 OPND 栈。
iv. **遇到运算符 (Operator):** 将当前运算符与 OPTR 的栈顶运算符进行优先级比较。
    - 若当前运算符优先级 **高于** OPTR 栈顶运算符，则入栈 OPTR。
    - 若当前运算符优先级 **低于或等于** OPTR 栈顶运算符，则从 OPTR 弹出一个运算符，从 OPND 弹出两个操作数，进行计算，并将结果压回 OPND 栈。然后，重复此步骤，继续比较当前运算符与新的 OPTR 栈顶运算符。
v. **扫描完成 (End of Expression):** 当表达式扫描至结束符 `#` 时，依次弹出 OPTR 和 OPND 中的元素进行计算，直到 OPTR 栈只剩一个 `#` ，此时 OPND 栈中唯一的数就是结果。

- **代码实现 (Code Implementation):**

```
// Slide 80 Pseudocode
OperandType EvaluateExpression() {
    InitStack(OPTR); Push(OPTR, '#'); // Operator stack
    InitStack(OPND); // Operand stack
    c = getchar();

    while (c != '#' || GetTop(OPTR) != '#') {
        if (!In(c, OP)) { // c is not an operator (it's an operand)
            Push(OPND, c);
            c = getchar();
        } else { // c is an operator
            switch (Precede(GetTop(OPTR), c)) { // Compare precedence
                case '<': // Stack top has lower priority
                    Push(OPTR, c);
                    c = getchar();
                    break;
                case '=': // Parentheses match
                    Pop(OPTR, x); // Pop the '('
                    c = getchar();
                    break;
                case '>': // Stack top has higher or equal priority
                    theta = Pop(OPTR);
                    b = Pop(OPND);
                    a = Pop(OPND);
                    Push(OPND, Operate(a, theta, b)); // Calculate and push result
                    // Note: 'c' does not advance here, it needs to be compared again
                    break;
            }
        }
    }
    return GetTop(OPND);
}
```

6. 嵌套与递归 (Nesting and Recursion)

- **函数调用机制 (Function Call Mechanism):**
  - **English:** The system uses a **call stack** to manage function execution. A function call involves pushing a "stack frame" (containing parameters, return address, and local variables). A function return involves popping this frame. This mechanism follows a LIFO order: "The latter call returns first!"
  - **中文:** 系统使用 **调用栈** 来管理函数的执行。函数调用时会压入一个包含参数、返回地址和局部变量的 **“栈帧”**。函数返回时会弹出该栈帧。这个过程遵循“后调用者先返回”的后进先出规则。

- **递归 (Recursion):**
  - **English:** A recursive function call is handled the same way. Each recursive call pushes a new, independent stack frame. This allows the function to have its own set of parameters and local variables for each level of recursion.
  - **中文:** 递归函数的调用与普通函数调用机制相同。每次递归调用都会在调用栈上创建一个新的、独立的栈帧，使得每一层递归都有自己独立的参数和局部变量副本。
- **经典示例：汉诺塔 (Classic Example: Tower of Hanoi):**
  - **问题:** 将 n 个盘子从塔 X 借助塔 Y 移动到塔 Z，每次只能移动一个，且大盘不能在小盘之上。
  - **递归解法:**
    a. `if n == 1`, 直接将盘子从 X 移动到 Z。
    b. `else`,
       - 将 `n-1` 个盘子从 X 借助 Z 移动到 Y (`hanoi(n-1, x, z, y)`)。
       - 将第 `n` 个盘子从 X 移动到 Z (`move(x, n, z)`)。
       - 将 `n-1` 个盘子从 Y 借助 X 移动到 Z (`hanoi(n-1, y, x, z)`)。
  - **代码实现 (Code Implementation):**

```
void hanoi(int n, char x, char y, char z) {
    // Move n disks from tower x to tower z, using y as auxiliary.
    if (n == 1) {
        move(x, 1, z); // moves the disc numbered 1 from x to z
    } else {
        hanoi(n - 1, x, z, y); // Move n-1 from x to y, using z
        move(x, n, z); // moves disc numbered n from x to z
        hanoi(n - 1, y, x, z); // Move n-1 from y to z, using x
    }
}
```

7. 回文判断 (Palindrome Game)

- **问题描述:** 判断一个字符串（忽略空格）是否是回文，即正读和反读都一样。
- **算法思想 (Algorithm from Code):**
  i. 获取字符串 `S` 的长度 `n`。
  ii. 将字符串的前半部分（ `0` 到 `n/2 - 1` ）的字符依次压入栈 `T`。
  iii. 从字符串的后半部分开始，与从栈中弹出的字符进行比较。
  iv. 如果所有比较都相等，则是回文；一旦出现不相等，则不是回文。
- **代码实现 (Code Implementation):**

```c
int IsHuiwen(char *S) {
    SeqStack T;
    int i, n;
    char t1;
    InitStack(&T);
    n = strlen(S); // Length of the vector
    for (i = 0; i < n / 2; i++) {
        Push(&T, S[i]); // Push half of the characters onto the stack
    }
    // Adjust starting index for the second half based on odd/even length
    int j_start = (n % 2 == 0) ? n / 2 : n / 2 + 1;
    for (i = j_start; i < n; i++) {
        t1 = Pop(&T); // Pop a character
        if (t1 != S[i]) return 0; // If not equal, return 0
    }
    return 1; // If the comparison is complete and equal, return 1
}
```

(Note: The code on slide 97 had a logical error in the while loop condition. The code above is a corrected, more common implementation of the described algorithm.)

8. 地图四色问题 (Map Four Staining Problem)

- **问题描述与理论 (Problem Description and Theorem):**
  - **English:** The four-color theorem, one of the famous theorems in computer science, states that any map can be colored with at least four colors in such a way that no two adjacent regions have the same color. The goal is to prove or find a valid coloring for a given map using a stack and backtracking.
  - **中文:** 四色定理是计算机科学中的著名定理之一，它指出任何地图都可以用至少四种颜色进行着色，以确保没有两个相邻的区域颜色相同。我们的目标是使用栈和回溯法来证明或找出一个给定地图的有效着色方案。
- **算法思想 (Algorithm Idea):**
  - **English:** The core idea is to use backtracking. A stack is used to record the color choices for each region.
    a. **Start:** Begin with the first administrative zone (e.g., region 1).
    b. **Trial:** For the current region, sequentially test each of the four distinct colors.
    c. **Check & Advance:** For a given color, check if it conflicts with any already colored adjacent regions. If there is no conflict (the color is valid), record this choice in the stack and advance to the next region.

    d. **Backtrack:** If all four colors conflict for the current region, it means a previous coloring choice was incorrect. The system must backtrack. According to the slide, this is where it will "pop the value from the top of the stack to correct the current color code." This implies reversing the last successful decision to try a new path.
  - **中文:** 核心思想是使用回溯法，并利用栈来记录每个区域的颜色选择。
    a. **开始:** 从第一个行政区（如区域1）开始。
    b. **试探:** 对当前区域，按顺序尝试四种不同的颜色。
    c. **检查与前进:** 对于某个尝试的颜色，检查它是否与已着色的相邻区域冲突。如果不冲突（颜色有效），则将此选择记录入栈，并前进到下一个区域。
    d. **回溯:** 如果当前区域的所有四种颜色都发生冲突，意味着之前的某个着色选择是错误的。系统必须回溯。根据幻灯片的描述，此时系统将“从栈顶弹出该值以纠正当前颜色代码”，这意味着撤销上一步成功的决策，以便为上一个区域尝试新的颜色。
- **示例与实现 (Example and Implementation):**
  - **场景 (Scenario):** 一个有7个行政区、4种颜色（a,b,c,d，或1,2,3,4）的地图，其邻接关系由一个 `R[7][7]` 矩阵表示。
  - **代码分析 (Code Analysis):** 演示文稿中的C代码通过循环和变量巧妙地模拟了栈回溯的行为，而不是使用一个显式的数据结构。
    - `int s[]`：这个数组扮演了栈的角色， `s[a]` 存储了区域 `a` 的颜色。
    - `int a`：这个变量是指针，指向当前正在尝试着色的区域。 `a++` 相当于“入栈”（前进到下一层决策）。
    - `int J`：这个变量代表当前正在为区域 `a` 尝试的颜色。
    - **前进 (Advance):** 当为区域 `a` 找到一个不冲突的颜色 `J` 时（ `ELSE { s[a]=J; a=a+1; J=1; }` ），代码记录颜色，并将指针 `a` 移到下一个区域，准备为新区域从颜色1开始尝试。这相当于将 `(a, J)` 这个决策压入栈。
    - **回溯 (Backtrack):** 当为区域 `a` 尝试了所有颜色（ `J` 从1到4）都失败后（ `IF (J>4)` ），代码执行 `a=a-1;` ，这相当于回退到上一个区域。然后执行 `J=s[a]+1;` ，即获取上一个区域当时使用的颜色，并从它的下一个颜色开始新的尝试。这个组合操作 `a--` 和 `J=s[a]+1` 正是回溯的核心，相当于“弹出旧决策，并修改状态以尝试新决策”。

```c
// Slide 82 C Code
void mapcolor(int R[][], int n, int s[]) {
    s[1] = 1;                    // Initialize: color Region 1 with color 1
    int a = 2, J = 1;            // a: current region, J: current color trial
    while (a <= n) {
        // Loop to try colors J=1..4 for region 'a'
        while ((J <= 4) && (a <= n)) {
            int k = 1;
            // Loop to check for conflicts with previously colored regions k=1..a-1
            while ((k < a) && (s[k] * R[a - 1][k - 1] != J)) {
                k = k + 1;
            }

            if (k < a) { // Conflict found (inner while loop broke early)
                J = J + 1; // Try next color for region 'a'
            } else { // No conflict found (inner while loop completed)
                s[a] = J;      // Assign valid color
                a = a + 1;     // Move to next region
                J = 1;         // Reset color trial for the new region
            }
        }
        if (J > 4) { // All 4 colors failed for region 'a', need to backtrack
            a = a - 1;      // Go back to the previous region
            J = s[a] + 1;   // Try the next color for that previous region
        }
    }
}
```

9. N皇后问题 (N-Queens Problem)

- **问题描述 (Problem Description):**
  - **English:** The N-Queens problem is to place N chess queens on an N×N chessboard so that no two queens threaten each other. The constraint is that no two queens can be on the same row, column, or diagonal.
  - **中文:** N皇后问题是在一个N×N的棋盘上放置N个皇后，使得任意两个皇后都不能互相攻击。约束条件是：任意两个皇后不能位于同一行、同一列或同一条对角线上。
- **解的表示 (Solution Representation):**
  - **English:** Since each queen must be in a different column, a one-dimensional array `x` of size N can represent a solution, where the index `i` represents the column and the value `x[i]` represents the row of the queen in that column.

- **中文:** 由于每个皇后必须占据不同的一列，我们可以用一个大小为N的一维数组 `x` 来表示一个解。其中，数组的下标 `i` 代表列号，而 `x[i]` 的值代表放在该列的皇后的行号。

- **算法思想：回溯法 (Algorithm Idea: Backtracking):**
  - **English:** The solution is built column by column using recursion, which implicitly uses the call stack.
    a. **Place:** Start with the first column (`j=1`). Try to place a queen in the first row (`i=1`).
    b. **Check:** Check if this position is "safe" (legal) according to the rules.
    c. **Recurse:** If it is safe, recursively call the function to place a queen in the next column (`j+1`).
    d. **Backtrack:** If the recursive call for `j+1` does not lead to a solution (it returns), it means the placement at column `j` was a dead end. The algorithm then backtracks. In a recursive function, this happens automatically when the function returns. In an iterative implementation, you would pop from a stack. The code then tries the next available row in column `j` .
    e. **Solution:** If a queen is successfully placed in the last column (`j > n`), a complete, valid solution has been found.
  - **中文:** 通过递归（隐式地使用调用栈）来逐列构建解决方案。
    a. **放置:** 从第一列（ `j=1` ）开始，尝试在第一行（ `i=1` ）放置一个皇后。
    b. **检查:** 检查该位置是否“安全”(合法)。
    c. **递归:** 如果安全，则递归调用函数去放置下一列（ `j+1` ）的皇后。
    d. **回溯:** 如果为 `j+1` 列的递归调用未能找到解（即函数返回了），说明在第 `j` 列的当前位置是一个死胡同。算法需要回溯。在递归中，回溯是自动发生的（函数返回）。然后代码会继续尝试第 `j` 列的下一行。
    e. **找到解:** 如果成功地在最后一列（ `j > n` ）放置了皇后，那么就找到了一个完整的有效解。

- **代码实现 (Code Implementation):**
  - **`ok(i, j)` 函数:** 检查在第 `j` 列的第 `i` 行放置皇后是否安全。它会检查与前面 `j-1` 列的所有皇后是否有行冲突或对角线冲突。
  - **`queen(j)` 函数:** 核心递归函数，用于放置第 `j` 列的皇后。
    - **Base Case (基本情况):** `if (j > n)` ，表示所有N列都已成功放置皇后，找到了一个解，打印它。
    - **Recursive Step (递归步骤):** `for(i=1; i<=n; i++)` 循环尝试当前列 `j` 的每一行 `i` 。
      - `if(ok(i,j))` : 如果位置 `(i,j)` 安全。
      - `a[j]=i;` : 放置皇后。
      - `queen(j+1);` : 递归处理下一列。当 `queen(j+1)` 返回后，无论它是否找到了解，程序都会继续 `for` 循环的下一次迭代（尝试第 `j` 列的下一行 `i+1` ），自动覆盖 `a[j]` 的值，这本身就是一种隐式的回溯。

```c
// Slide 87-88 C Code
#include <stdio.h>
#define n 8 // 8 queens problem
int m = 0, a[n + 1]; // m: solution count, a[]: stores queen positions

// Check if placing a queen at row i, col j is safe
int ok(int i, int j) {
    int j1, i1, ok1 = 1;
    // Check row conflict with previous columns
    for (j1 = 1; j1 < j; j1++) {
        if (a[j1] == i) {
            ok1 = 0;
            break;
        }
    }
    // Check diagonal conflicts
    if (ok1) {
        for (j1 = 1; j1 < j; j1++) {
            if ((a[j1] == i - (j - j1)) || (a[j1] == i + (j - j1))) {
                ok1 = 0;
                break;
            }
        }
    }
    return ok1;
}

void queen(int j) { // Place queen for column j
    if (j > n) { // Base case: solution found
        m++;
        printf("m=%d ", m);
        for (int i = 1; i <= n; i++) printf("%d", a[i]);
        printf("\n");
    } else { // Recursive step
        for (int i = 1; i <= n; i++) { // Try each row i for column j
            if (ok(i, j)) {
                a[j] = i;       // Place queen
                queen(j + 1);   // Recurse for the next column
                // Backtracking is implicit: the loop continues to the next 'i',
                // overwriting a[j] if another valid row is found.
            }
        }
    }
```

```
}

int main() {
    queen(1); // Start the process from the first column
    return 0;
}
```