# 第3章 受限线性表：栈，队列和串

#数据结构与算法

## 3.1 操作受限线性表：栈

- 栈又称为堆栈，仅能在表尾进行删除和插入操作
- 栈有栈底（bottom）和栈顶（top），有着**先进后出**的特点：FILO
- 栈存在溢出的风险：上溢和下溢

## 3.2 栈的存储结构

### 3.2.1 顺序栈的定义与实现

顺序栈使用数组实现，栈顶指针 `top` 表示当前栈顶位置。

- 初始化：`top = -1`（空栈）
- 入栈（push）：`top++`，数组 `stack[top]` 存元素；若 `top == maxsize-1` 则上溢
- 出栈（pop）：元素 = `stack[top]`，`top--`；若 `top == -1` 则下溢

```
#define MAXSIZE 100
struct SqStack {
    int data[MAXSIZE];
    int top;
};

void InitStack(SqStack* s) { s->top = -1; }
bool Push(SqStack* s, int e) {
    if (s->top == MAXSIZE - 1) return false; // 上溢
    s->data[++s->top] = e;
    return true;
}
bool Pop(SqStack* s, int* e) {
    if (s->top == -1) return false; // 下溢
    *e = s->data[s->top--];
    return true;
}
```

### 3.2.2 链栈的定义与实现

链栈使用链表实现，无大小限制，栈顶为链表头。

- 每个节点：`data` + `next`
- 入栈：新节点插入头
- 出栈：删除头节点
- 示例C++实现：

```
struct StackNode {
    int data;
    StackNode* next;
};
typedef StackNode* LinkStackPtr;

bool Push(LinkStackPtr* top, int e) {
    LinkStackPtr s = new StackNode;
    if (!s) return false;
    s->data = e;
    s->next = *top;
    *top = s;
    return true;
}
bool Pop(LinkStackPtr* top, int* e) {
    LinkStackPtr p;
    if (*top == nullptr) return false;
    *e = (*top)->data;
    p = *top;
    *top = (*top)->next;
    delete p;
    return true;
}
```

# 3.3 栈的应用

## 3.3.1 栈的括号匹配

使用栈检查括号是否匹配（如`()`、`{}`、`[]`）。

- 算法：扫描字符串，遇左括号入栈，遇右括号检查栈顶匹配并出栈；最终栈空则匹配
- 示例：字符串`{(a+b)}` → 匹配；`{(a+b]` → 不匹配
- 实现思路：栈存左括号类型，pop时比对

## 3.3.2 栈与递归

递归本质上是栈的调用：函数调用栈管理局部变量和返回地址

比如汉诺塔问题（n个盘子从A移到C）：

- 递归定义：`hanoi(n, A, B, C)` = hanoi(n-1, A, C, B) + 移n从A到C + hanoi(n-1, B, A, C)
- 栈模拟：用栈压入子问题参数，非递归实现避免栈溢出

## 3.3.3 表达式求值问题

- 编译系统中，人们把操作数，运算符与分界符统称为token。
- 人们在计算算术表达式时，遵循四则运算法则。而计算机在实现时，要把人们遵循的运算法则变成优先级表

将中缀表达式转换为后缀表达式（没优先级）

```
    3+5*y
-->  35y*+
```

```
    3+5*(y-5)
-->  35y5-*+
```

计算机的实现方式：

- S1：读取表达式的一个单词x；
- S2：若x是操作数，则原样输出；否则执行S3；
- S3：x是操作符，若x是（，则将栈中元素输出，直到遇到)。将（从栈中删除，执行S4；
- S4：x不是)，循环比较栈顶元素和x的优先级，若不小于，则栈顶元素出栈并输出；
- S5：x进栈；
- S6：若表达式已读完，则执行S1，否则执行S7；

### 算法核心思想

该算法通过一个 **运算符栈** 和一个 **输出流**（用于构建后缀表达式）来实现转换。它从左到右扫描中缀表达式，根据遇到的Token类型（操作数、运算符、括号）执行不同操作。

### 详细步骤

1. **遇到操作数**：直接拼接到输出流。
2. **遇到左括号** `(`：直接压入运算符栈。
3. **遇到右括号** `)`：将运算符栈的元素弹出并拼接到输出流，直到遇到左括号 `(`。最后将 `(` 弹出并丢弃。
4. **遇到运算符**（如 `+`, `*`）：
   - 若栈为空、栈顶为 `(`，或当前运算符优先级**高于**栈顶运算符，则直接入栈。
   - 否则，将栈顶运算符弹出并拼接到输出流，重复此过程，直到满足入栈条件，再将当前运算符入栈。

5. **表达式扫描完毕**：将栈中所有剩余的运算符依次弹出并拼接到输出流。

**示例：** `3 + 5 * ( y - 5 )`

| 当前Token | 操作 | 输出流 | 运算符栈 |
|---|---|---|---|
| 3 | 操作数，输出 | 3 | [] |
| + | 栈空，入栈 | 3 | [+] |
| 5 | 操作数，输出 | 3 5 | [+] |
| * | 优先级 * > +，入栈 | 3 5 | [+, *] |
| ( | 左括号，入栈 | 3 5 | [+, *, (] |
| y | 操作数，输出 | 3 5 y | [+, *, (] |
| - | 栈顶为 (，入栈 | 3 5 y | [+, *, (, -] |
| 5 | 操作数，输出 | 3 5 y 5 | [+, *, (, -] |
| ) | 弹栈直到 (，输出 - | 3 5 y 5 - | [+, *] |
| **扫描结束** | 依次弹栈并输出 | 3 5 y 5 - * + | [] |

最终结果为 `3 5 y 5 - * +`。

# 3.4 操作受限线性表：队列

- 队列是一种**先进先出**（FIFO）的线性表，仅在队列尾（rear）插入，在队列头（front）删除
- 队空：`front == rear`；队满：需特殊判断（如循环队列用 `(rear+1) % maxsize == front`）
- 应用：任务调度、缓冲区

## 3.4.1 顺序队列的定义与实现

使用数组，front/rear指针。

- 入队（enqueue）：`rear = (rear+1) % maxsize`，存元素
- 出队（dequeue）：元素 = `q[front]`，`front = (front+1) % maxsize`
- 缺点：假溢出（一端空空间浪费）

对于带有优先级的火车售票系统，我们可以使用两个队列来实现：一个优先队列（`priorityQ`）和一个普通队列（`normalQ`）。

- **买票**：有优先权的乘客进入优先队列，普通乘客进入普通队列。
- **退票**：乘客退票后，票额可以重新分配。
- **售票逻辑**：售票时，优先检查并服务 `priorityQ` 中的乘客。只有当 `priorityQ` 为空时，才服务 `normalQ` 中的乘客。

```
// 假设已有一个基础的队列实现 Queue

struct TicketSystem {
    Queue priorityQ; // 优先队列
    Queue normalQ;   // 普通队列
    int tickets_available; // 可用票数
};

// 乘客买票
void BuyTicket(TicketSystem* ts, int passengerID, bool hasPriority) {
    if (hasPriority) {
        EnQueue(&ts->priorityQ, passengerID); // 进入优先队列
    } else {
        EnQueue(&ts->normalQ, passengerID);   // 进入普通队列
    }
}

// 出票
bool SellTicket(TicketSystem* ts, int* passengerID) {
    if (ts->tickets_available <= 0) return false;

    // 优先队列不为空，先出票
    if (DeQueue(&ts->priorityQ, passengerID)) {
        ts->tickets_available--;
        return true;
    }
    // 否则，处理普通队列
    else if (DeQueue(&ts->normalQ, passengerID)) {
        ts->tickets_available--;
        return true;
    }

    return false; // 无人排队
}
```

- **实现示例（循环队列）：**

```
#define MAXSIZE 100
struct Queue {
    int data[MAXSIZE];
    int front, rear;
};
```

```
void InitQueue(Queue* q) { q->front = q->rear = 0; }
bool EnQueue(Queue* q, int e) {
    if ((q->rear + 1) % MAXSIZE == q->front) return false; // 队满
    q->data[q->rear] = e;
    q->rear = (q->rear + 1) % MAXSIZE;
    return true;
}
bool DeQueue(Queue* q, int* e) {
    if (q->front == q->rear) return false; // 队空
    *e = q->data[q->front];
    q->front = (q->front + 1) % MAXSIZE;
    return true;
}
```

### 3.4.2 链队与双端队列

链队：链表实现，头尾指针，无大小限。

- 入队：尾插入；出队：头删除
- 双端队列（deque）：两端均可插入/删除，如输入限制的deque

```
struct QNode {
    int data;
    QNode* next;
};

struct LinkQueue {
    QNode* front;
    QNode* rear;
};

void InitLinkQueue(LinkQueue* q) { q->front = q->rear = nullptr; }
```

## 3.5 串（字符串）的定义与实现（上课没讲）

- 串：由零个或多个字符组成的有限序列，如 `S = "abc"`
- 存储：顺序串（数组）或链式（链表）
- 操作：求长、比较、子串、连接、模式匹配

### 串的基本操作

- 求长：`StrLength(S)`

- **连接：** `S = StrConcat(S1, S2)`

```
#include <cstring>
class String {
private:
    char* str;
public:
    String(const char* s = "") {
        int len = strlen(s);
        str = new char[len + 1];
        strcpy(str, s);
    }
    ~String() { delete[] str; }
    int StrLength() const {
        return strlen(str);
    }
    String StrConcat(const String& other) const {
        int len1 = StrLength(), len2 = other.StrLength();
        String result;
        delete[] result.str; // 假设有重置
        result.str = new char[len1 + len2 + 1];
        strcpy(result.str, str);
        strcat(result.str, other.str);
        return result;
    }
};
```

上一章 第2章 线性表
下一章 第4章 扩展线性表—数组与广义表