数据结构 - 栈和队列复习资料

详细版本

Contents

Date: June 7, 2025.

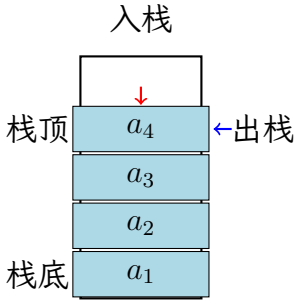
1. 栈的基本概念

1.1. 栈的定义.

定义 **1.1** (栈). 栈（*stack*）是限定仅在表的一端进行插入和删除操作的线性表。允许插入和删除的一端称为栈顶（*stack top*），另一端称为栈底（*stack bottom*），不含任何数据元素的栈称为空栈。

1.2. 栈的特性. 栈中元素具有以下特性:

(1) 后进先出（**LIFO**）: Last In First Out，最后进入栈的元素最先出栈
(2) 线性关系: 栈中元素具有一对一的前驱后继关系
(3) 操作受限: 只能在栈顶进行插入（入栈、压栈）和删除（出栈、弹栈）操作



1.3. 栈的抽象数据类型定义.

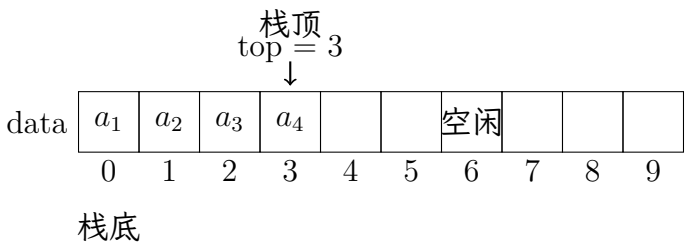
```
ADT Stack {
    数据对象: D = {ai | ai    ElemType, i = 1,2,...,n, n    0}
    数据关系: R1 = {<ai-1, ai> | ai-1, ai    D, i = 2,...,n}
    基本操作:
        InitStack(S);              // 栈的初始化
        DestroyStack(S);           // 栈的销毁
        Push(S, x);                // 入栈操作
        Pop(S);                    // 出栈操作
        GetTop(S);                 // 取栈顶元素
        Empty(S);                  // 判断栈是否为空
} ADT Stack
```

LISTING 1. 栈的抽象数据类型定义

2. 栈的存储结构及实现

2.1. 顺序栈.

2.1.1. 顺序栈的存储结构. 顺序栈是栈的顺序存储结构，用一维数组存储栈中元素，并用一个整型变量 top 记录栈顶元素在数组中的位置。



2.1.2. 顺序栈的实现.

```
const int StackSize = 100;
template <typename DataType>
class SeqStack {
public:
    SeqStack();                     // 构造函数
    ~SeqStack();                    // 析构函数
    void Push(DataType x);          // 入栈操作
    DataType Pop();                 // 出栈操作
    DataType GetTop();              // 取栈顶元素
    int Empty();                    // 判断栈是否为空
private:
    DataType data[StackSize];       // 存放栈元素的数组
    int top;                        // 栈顶指针
};
```

LISTING 2. 顺序栈类定义

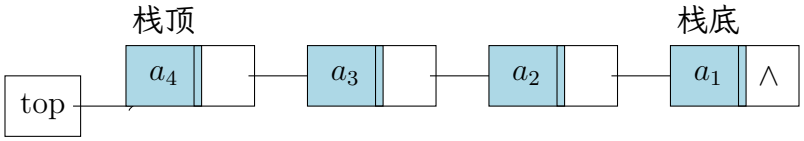
基本操作实现:

```
// 构造函数
template <typename DataType>
SeqStack<DataType>::SeqStack() {
    top = -1;    // 栈空时top = -1
}

// 入栈操作
template <typename DataType>
void SeqStack<DataType>::Push(DataType x) {
    if (top == StackSize - 1)
        throw "栈上溢";
    data[++top] = x;
}

// 出栈操作
template <typename DataType>
DataType SeqStack<DataType>::Pop() {
    if (top == -1)
        throw "栈下溢";
    return data[top--];
}

// 取栈顶元素
template <typename DataType>
DataType SeqStack<DataType>::GetTop() {
    if (top == -1)
        throw "栈为空";
    return data[top];
}

// 判空操作
template <typename DataType>
int SeqStack<DataType>::Empty() {
    return (top == -1) ? 1 : 0;
}
```

Listing 3. 顺序栈基本操作

2.2. 链栈.

2.2.1. 链栈的存储结构. 链栈是栈的链式存储结构，用单链表实现，通常以链表的头部作为栈顶。



2.2.2. 链栈的实现.

```
template <typename DataType>
struct Node {
    DataType data;
    Node<DataType>* next;
};

template <typename DataType>
class LinkStack {
public:
    LinkStack();                    // 构造函数
    ~LinkStack();                   // 析构函数
    void Push(DataType x);          // 入栈操作
    DataType Pop();                 // 出栈操作
    DataType GetTop();              // 取栈顶元素
    int Empty();                    // 判断栈是否为空
private:
    Node<DataType>* top;            // 栈顶指针
};
```

Listing 4. 链栈类定义

```
// 入栈操作
template <typename DataType>
void LinkStack<DataType>::Push(DataType x) {
    Node<DataType>* s = new Node<DataType>;
    s->data = x;
    s->next = top;
    top = s;
}

// 出栈操作
template <typename DataType>
DataType LinkStack<DataType>::Pop() {
    if (top == nullptr)
        throw "栈下溢";
    DataType x = top->data;
    Node<DataType>* p = top;
    top = top->next;
    delete p;
    return x;
}
```

Listing 5. 链栈基本操作

2.3. 顺序栈与链栈的比较.

比较项目	顺序栈	链栈
存储空间	预分配固定大小	动态分配
空间利用率	可能浪费空间	充分利用空间
存储密度	高（无指针开销）	低（有指针开销）
操作效率	高	相对较低
栈满判断	容易判断	依赖内存状态
适用场景	元素个数相对稳定	元素个数变化较大

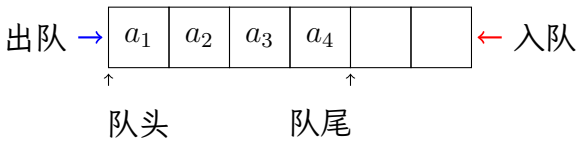
3. 队列的基本概念

3.1. 队列的定义.

定义 **3.1** (队列). 队列（*queue*）是只允许在一端进行插入操作，在另一端进行删除操作的线性表。允许插入（入队）的一端称为队尾（*rear*），允许删除（出队）的一端称为队头（*front*）。

3.2. 队列的特性. 队列中元素具有以下特性：

(1) 先进先出（**FIFO**）：First In First Out，最先进入队列的元素最先出队
(2) 线性关系：队列中元素具有一对一的前驱后继关系
(3) 操作受限：只能在队尾插入，在队头删除



3.3. 队列的抽象数据类型定义.

```
ADT Queue {
    数据对象: D = {ai | ai    ElemType, i = 1,2,...,n, n    0}
    数据关系: R1 = {<ai-1, ai> | ai-1, ai    D, i = 2,...,n}
    基本操作:
        InitQueue(Q);              // 队列的初始化
        DestroyQueue(Q);           // 队列的销毁
        EnQueue(Q, x);             // 入队操作
        DeQueue(Q);                // 出队操作
        GetHead(Q);                // 取队头元素
        Empty(Q);                  // 判断队列是否为空
} ADT Queue
```

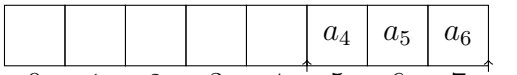
Listing 6. 队列的抽象数据类型定义

4. 队列的存储结构及实现

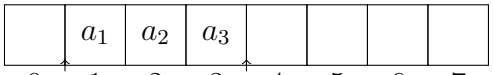
4.1. 顺序队列的问题. 如果用顺序存储实现队列，会遇到”假溢出”问题：

状态 2：假溢出

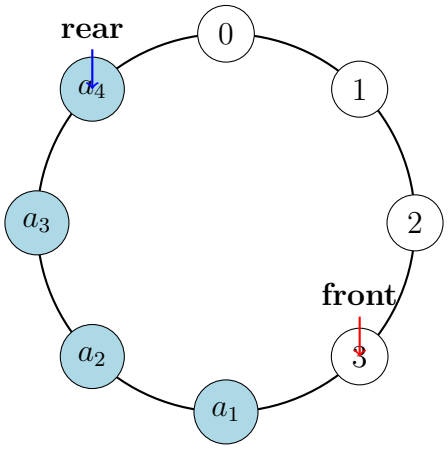
空闲但无法使用



状态 1：正常队列



4.2. 循环队列. 为了解决假溢出问题，将存储队列的数组看成头尾相接的循环结构，称为循环队列。



循环队列示意图

4.2.1. 循环队列的队空和队满判断. 问题：循环队列中，队空和队满时都有 front == rear，如何区分？

解决方案：牺牲一个存储单元，队满条件为：$(rear+1)\%QueueSize == front$

状态	判断条件
队空	front == rear
队满	(rear + 1) % QueueSize == front
队列长度	(rear - front + QueueSize) % QueueSize

4.2.2. 循环队列的实现.

```
const int QueueSize = 100;
template <typename DataType>
class CirQueue {
public:
    CirQueue();                     // 构造函数
    ~CirQueue();                    // 析构函数
    void EnQueue(DataType x);       // 入队操作
    DataType DeQueue();             // 出队操作
    DataType GetHead();             // 取队头元素
    int Empty();                    // 判断队列是否为空
private:
    DataType data[QueueSize];       // 存放队列元素的数组
    int front, rear;                // 队头和队尾指针
```

```
};
```

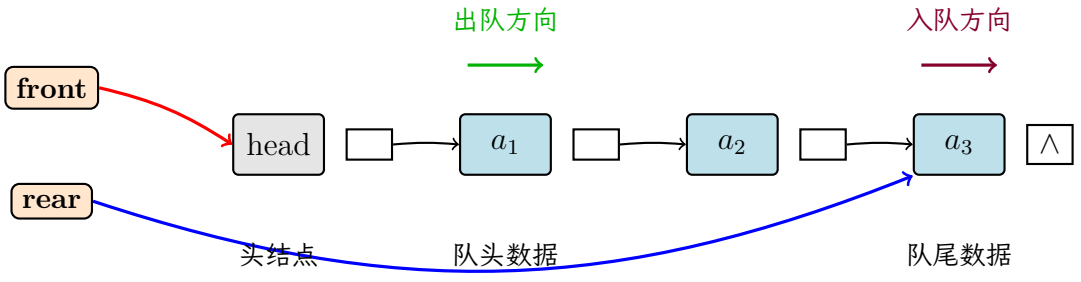
Listing 7. 循环队列类定义

```
// 构造函数
template <typename DataType>
CirQueue<DataType>::CirQueue() {
    front = rear = 0;    // 队空时front = rear
}

// 入队操作
template <typename DataType>
void CirQueue<DataType>::EnQueue(DataType x) {
    if ((rear + 1) % QueueSize == front)
        throw "队列上溢";
    rear = (rear + 1) % QueueSize;
    data[rear] = x;
}

// 出队操作
template <typename DataType>
DataType CirQueue<DataType>::DeQueue() {
    if (front == rear)
        throw "队列下溢";
    front = (front + 1) % QueueSize;
    return data[front];
}

// 取队头元素
template <typename DataType>
DataType CirQueue<DataType>::GetHead() {
    if (front == rear)
        throw "队列为空";
    return data[(front + 1) % QueueSize];
}
```

Listing 8. 循环队列基本操作

4.3. 链队列.

4.3.1. 链队列的存储结构. 链队列是队列的链式存储结构，用单链表实现，需要设置队头指针 front 和队尾指针 rear。



链队列存储结构
蓝色为数据节点，灰色为头结点，∧ 表示空指针

4.3.2. 链队列的实现.

```
template <typename DataType>
class LinkQueue {
public:
    LinkQueue();                        // 构造函数
    ~LinkQueue();                       // 析构函数
    void EnQueue(DataType x);           // 入队操作
    DataType DeQueue();                 // 出队操作
    DataType GetHead();                 // 取队头元素
    int Empty();                        // 判断队列是否为空
private:
    Node<DataType>* front, * rear; // 队头和队尾指针
};
```

LISTING 9. 链队列类定义

```
// 构造函数
template <typename DataType>
LinkQueue<DataType>::LinkQueue() {
    Node<DataType>* s = new Node<DataType>;
    s->next = nullptr;
    front = rear = s;    // 头结点
}

// 入队操作
template <typename DataType>
void LinkQueue<DataType>::EnQueue(DataType x) {
    Node<DataType>* s = new Node<DataType>;
    s->data = x;
    s->next = nullptr;
    rear->next = s;
    rear = s;
}

// 出队操作
template <typename DataType>
DataType LinkQueue<DataType>::DeQueue() {
    if (front == rear)
        throw "队列下溢";
    Node<DataType>* p = front->next;
    DataType x = p->data;
    front->next = p->next;
    if (rear == p) rear = front;  // 队列变空
    delete p;
    return x;
}
```

LISTING 10. 链队列基本操作

5. 栈和队列的应用

5.1. 括号匹配问题.

例题 **5.1** (括号匹配算法). 设计算法判断表达式中括号是否正确配对。

算法思想:

(1) 用栈保存未配对的左括号
(2) 遇到左括号时入栈
(3) 遇到右括号时出栈一个左括号与之配对
(4) 最后栈空则配对成功

Algorithm 1 括号匹配算法

Require: 表达式字符串 str
Ensure: 匹配结果：0 表示匹配，1 表示多左括号，-1 表示多右括号
栈 S 初始化
for $i = 0$ to $strlen(str) - 1$ **do**
  if $str[i] == '('$ **then**
    $S.Push('(')$
  else if $str[i] == ')'$ **then**
    if $S.Empty()$ **then**
      return -1 {多右括号}
    else
      $S.Pop()$
    end if
  end if
end for
if $S.Empty()$ **then**
  return 0 {正确匹配}
else
  return 1 {多左括号}
end if

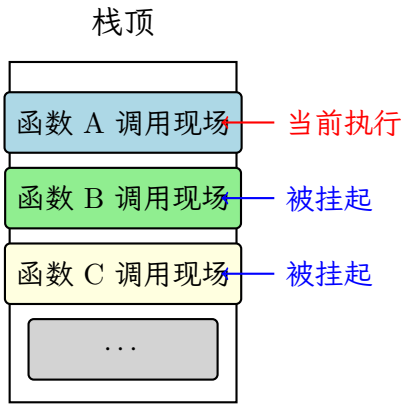
5.2. 表达式求值.

例题 **5.2** (表达式求值算法). 利用两个栈（操作数栈和运算符栈）实现中缀表达式求值。
算法过程：

(1) 扫描表达式，操作数直接入操作数栈
(2) 运算符与运算符栈顶比较优先级
(3) 当前运算符优先级高，入运算符栈
(4) 当前运算符优先级低，弹出栈顶运算符计算

当前字符	操作数栈	运算符栈	说明
(		#, (	(入运算符栈
4	4	#, (	4 入操作数栈
+	4	#, (, +	+ 入运算符栈
2	4, 2	#, (, +	2 入操作数栈
)	6	#, (	计算 $4 + 2 = 6$
)	6	#	括号匹配，(出栈
*	6	#, *	* 入运算符栈
3	6, 3	#, *	3 入操作数栈
#	18	#	计算 $6 * 3 = 18$，结束

5.3. 函数调用栈. 函数的嵌套调用使用系统栈保存调用现场：



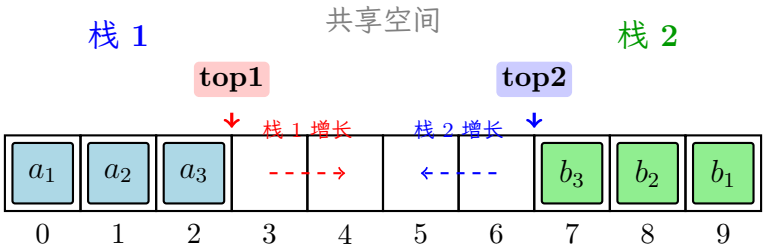
函数调用栈结构

5.4. 队列的应用场景.

(1) 操作系统：进程调度、作业队列
(2) 打印缓冲：打印任务排队
(3) 键盘缓冲：按键事件队列
(4) **BFS** 遍历：广度优先搜索算法
(5) 银行排队：客户服务系统

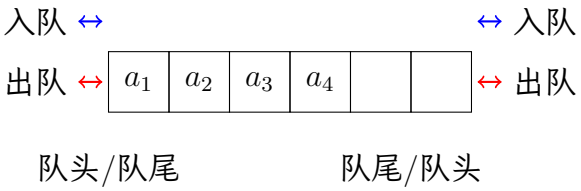
6. 特殊栈和队列

6.1. 共享栈. 利用一个数组实现两个栈，充分利用存储空间：



栈满条件：top1 + 1 == top2

6.2. 双端队列. 允许在队列两端进行插入和删除操作：



7. 算法复杂度分析

7.1. 时间复杂度.

操作	顺序栈	链栈	说明
Push	$O(1)$	$O(1)$	入栈操作
Pop	$O(1)$	$O(1)$	出栈操作
GetTop	$O(1)$	$O(1)$	取栈顶元素
Empty	$O(1)$	$O(1)$	判空操作

操作	循环队列	链队列	说明
EnQueue	$O(1)$	$O(1)$	入队操作
DeQueue	$O(1)$	$O(1)$	出队操作
GetHead	$O(1)$	$O(1)$	取队头元素
Empty	$O(1)$	$O(1)$	判空操作

7.2. 空间复杂度.

- 顺序栈：$O(n)$，需要预分配固定大小的数组
- 链栈：$O(n)$，动态分配，但有指针开销
- 循环队列：$O(n)$，需要预分配固定大小的数组
- 链队列：$O(n)$，动态分配，但有指针开销

8. 常见考点总结

8.1. 重点掌握内容.

(1) 栈和队列的基本概念：LIFO 和 FIFO 特性
(2) 循环队列：队空和队满的判断条件
(3) 栈的应用：括号匹配、表达式求值、函数调用
(4) 队列的应用：BFS 算法、操作系统调度
(5) 出入栈序列：判断序列的合法性
(6) 算法设计：用栈和队列解决实际问题

8.2. 常见题型.

(1) 判断给定的出栈序列是否合法
(2) 循环队列的队空、队满判断
(3) 用栈实现括号匹配算法
(4) 用栈实现中缀表达式求值
(5) 用栈实现中缀转后缀表达式
(6) 用队列实现 BFS 算法
(7) 共享栈的设计与实现

8.3. 解题技巧.

(1) 出栈序列判断：利用栈的 LIFO 特性，模拟入栈出栈过程
(2) 循环队列：注意取模运算和牺牲一个存储单元
(3) 表达式求值：掌握运算符优先级和两个栈的配合
(4) 递归转迭代：用栈模拟递归调用过程

9. 典型例题解析

例题 **9.1** (出栈序列判断). 栈的入栈序列为 *1,2,3,4,5*，判断以下哪些是合法的出栈序列：

(1) 5,4,3,2,1
(2) 4,5,3,2,1
(3) 4,3,5,1,2

解：

(1) ***5,4,3,2,1***：合法。全部入栈后依次出栈。
(2) ***4,5,3,2,1***：合法。*1,2,3,4* 入栈，*4* 出栈，*5* 入栈，*5* 出栈，*3,2,1* 依次出栈。
(3) ***4,3,5,1,2***：不合法。当 *5* 出栈时，*1,2* 还在栈中，不可能 *2* 比 *1* 先出栈。

例题 **9.2** (循环队列容量计算). 循环队列的存储空间为 *Q[0..m-1]*，约定 *front* 指向队头元素前一个位置，*rear* 指向队尾元素位置。当 *front=rear* 时队列为空，当 *(rear+1)*

解：设队列最多存储 n 个元素。当队列满时，有一个位置被浪费（用于区分队空和队满），因此：

$$n = m - 1$$

该循环队列最多可以存储 $m-1$ 个元素。

例题 **9.3** (共享栈设计). 设计一个共享栈，用一个数组 *S[0..n-1]* 存储两个栈，栈 *1* 从数组低端开始，栈 *2* 从数组高端开始。写出入栈和出栈算法。

解：

```
// 共享栈结构
struct SharedStack {
    int data[MAXSIZE];
    int top1, top2;  // 两个栈顶指针
};

// 初始化
void InitStack(SharedStack &S) {
    S.top1 = -1;
    S.top2 = MAXSIZE;
}

// 栈满判断
bool StackFull(SharedStack S) {
    return S.top1 + 1 == S.top2;
}

// 入栈操作
bool Push(SharedStack &S, int i, int x) {
    if (StackFull(S)) return false;
    if (i == 1)
        S.data[++S.top1] = x;
    else
        S.data[--S.top2] = x;
    return true;
}

// 出栈操作
int Pop(SharedStack &S, int i) {
    if (i == 1) {
        if (S.top1 == -1) throw "栈1下溢";
        return S.data[S.top1--];
    } else {
        if (S.top2 == MAXSIZE) throw "栈2下溢";
        return S.data[S.top2++];
    }
}
```