第3章 栈和队列

日常生活中都有装箱和排队的经历。例如，在搬家时将物品装入纸箱，住进新居后还要将物品从纸箱里一件一件取出来，纸箱最上层的物品肯定是最先取出的，最下层的物品肯定是最后取出来的，这个纸箱就是一个“栈”。再如，读者都经历过排队乘车、排队进入景区以及排队购票等场景，排成的队伍无论长短都有共同的特点，那就是排在前面的人先于排在后面的人完成有关活动，新来的人只能排在队尾，不能插队，这就是“队列”。在计算机领域，栈和队列更是无处不在，函数的调用都会涉及栈，还有消息队列、输入输出队列和打印队列等各种各样的队列。本章将讲解栈和队列的有关概念和基本操作等内容。

3.1 栈

3.1.1 基本概念

1. 基本术语

栈(stack)又称为堆栈，是一种操作受限的线性表，它仅允许在线性表的一端进行插入和删除操作，允许进行插入、删除操作的一端称为**栈顶**(top)，另外一端称为**栈底**(bottom)，栈顶所保存的数据元素称为**栈顶元素**，当栈中没有数据元素时称该栈为**空栈**，栈中现有元素数称为栈的**长度**。

向一个栈中插入一个数据元素的操作称为**入栈**、**进栈**或**压栈**；从栈中删除一个数据元素的操作称为**出栈**、**退栈**或**弹栈**。当栈已满时，如果还要有元素入栈，就会产生“**上溢**”，上溢是一种错误，使得问题处理无法进行。当栈是空栈时，如果还要进行出栈操作，就会产生“**下溢**”，下溢一般表明问题处理结束。

栈具有“后进先出”(Last In First Out，LIFO)或者“先进后出”(First In Last Out，FILO)的操作特点，这是由于入栈和出栈操作仅允许在栈顶进行，先入栈的元素必定后出栈，后入栈的元素必定先出栈。栈及其入栈、出栈操作如图 3.1 所示。

用于列车编组的铁路转轨网络就是一种栈结构。被称为列车编组站教科书的郑州北站占地 5.3 平方千米，日均办理货车近 3 万辆，是全球最大的列车编组站，其中就有大量栈结

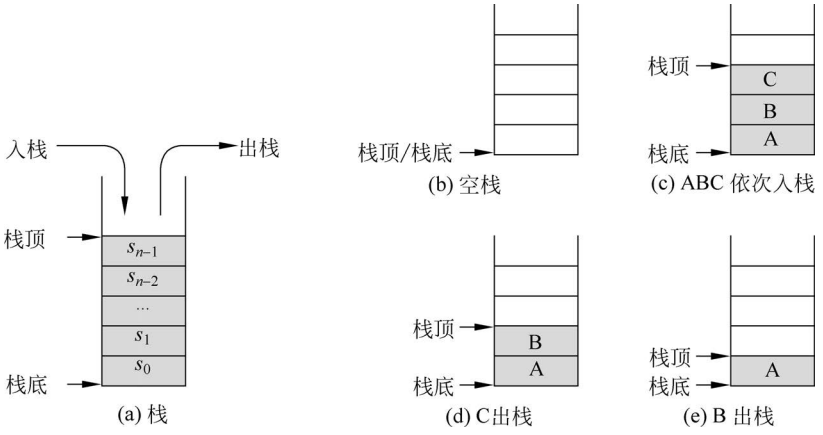


图 3.1 栈及其入栈、出栈操作示例

构的转轨线路。

2. 可能与不可能的出栈序列

将一个给定序列中的元素依次入栈，中间可以穿插出栈操作(前提是栈非空)，直到所有元素都出栈，并非能够得到任意的出栈序列，有些序列是不可能得到的，例如，ABCD 依次入栈，CBDA 就是可能的出栈序列：ABC 依次入栈，然后 CB 依次出栈，D 入栈，再 DA 依次出栈即得出栈序列 CBDA；而 CADB 就是不可能的出栈序列，无论如何操作都不可能得到该出栈序列。

对于一个给定的入栈序列，可能以及不可能的出栈序列的另一种判定方法如下。

在一个序列中，排在任意一个元素之后，并且先于该元素入栈的所有元素是倒序排列的，这样的序列才是可能的出栈序列，否则为不可能的出栈序列。

例如，ABCD 依次入栈，则

(1) CBDA 是可能的出栈序列，因为在该序列中 C 后面先于 C 的 BA 是倒序排列的，B 后面的 A 也是倒序排列的，D 后面只有 A 也是倒序的。

(2) DACB 是不可能的出栈序列，因为在该序列中 D 后面先于 D 的 ACB 不是倒序排列的。

(3) CADB 是不可能的出栈序列，因为在该序列中 C 后面先于 C 的 AB 不是倒序排列的。

实际上，如果用人眼直观观察，出栈序列中任意三个元素如果出现“后先中”情况的都是不可能的，否则都是可能的。例如，ABCD 依次入栈，CBDA 则是可能的出栈序列，而 DACB 则是不可能的出栈序列，因为其中任意三个元素无论是选 DAC 还是 DAB 都是按“后先中”顺序排列的。

3. 栈的抽象数据类型

栈的核心操作是入栈和出栈，但也有其他一些操作，下面给出栈的抽象数据类型定义。

```
ADT Stack {
    数据对象：D = {a_i | a_i ∈ ElementSet, i = 0, 1, 2, …, n - 1, n≥0}
    数据关系：R = {<a_i, a_i+1> | a_i, a_i+1 ∈ D, i = 0, 1, 2, …, n - 2, n≥0}
    基本操作：
        init()：初始化空栈。
```

```
        destroy(): 销毁栈,释放动态分配给栈的存储空间。
        getSize(): 返回栈的长度。
        isEmpty(): 判断栈是否为空,为空则返回 true,否则返回 false。
        isFull(): 判断栈是否为满,为满则返回 true,否则返回 false。
        push(e): 将数据元素 e 入栈。
        pop(): 出栈并返回出栈的数据元素。
        getTop(): 返回栈顶元素。
}
```

3.1.2 顺序栈

1. 顺序栈的定义

栈是一种操作受限的线性表,和线性表一样,栈也有两种存储结构:顺序存储结构和链式存储结构。

顺序栈利用一组地址连续的存储空间来依次存储栈中的元素,并选择一端作为栈顶。由于入栈和出栈都限定在栈顶进行操作,可以进行随机访问,因此,顺序栈的入栈和出栈操作的时间复杂度均为 $O(1)$。

顺序栈的定义由两部分构成:一是存储数据元素的数组 elements,二是表示栈顶位置的变量 top,假设数组 elements 有 n 个元素,依次为 elements[0], elements[1],…, elements[$n-1$],对于栈顶位置的选择及操作有如下一些方案,如表 3.1 所示。

表 3.1 顺序栈的定义方案

序号	top 的值	空栈	元素 e 入栈	元素 e 出栈
1	栈顶元素下标+1	top == 0	elements[top ++] = e;	e = elements[-- top];
2	栈顶元素下标	top == -1	elements[++ top] = e;	e = elements[top --];
3	栈顶元素下标-1	top == $n-1$	elements[top --] = e;	e = elements[++ top];
4	栈顶元素下标	top == n	elements[-- top] = e;	e = elements[top ++];

无论采用哪一种方案均可,本章选用第 1 种方案,该方案的 top 值也同时表示栈的长度,同时也比较直观,易于理解。如图 3.2 所示为采用该方案的栈。

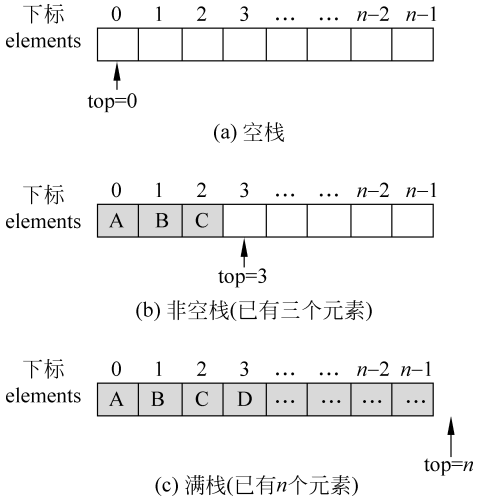


图 3.2 顺序栈

2. 顺序栈的类型定义及操作

下面给出顺序栈的存储结构定义及其操作。

算法 3.1 顺序栈的存储结构定义及其操作。

该算法的 C 语言描述如下。

```
#define MAXSIZE    100                          //MAXSIZE 为栈的最大长度
typedef struct {
    ElemType elements[MAXSIZE];
    int top;
} Stack;

void init(Stack *stack) {                       //初始化空栈
    stack->top = 0;
}

int getSize(Stack *stack) {                     //返回栈的长度
    return stack->top;
}

int isEmpty(Stack *stack) {                     //判断栈是否为空
    return stack->top == 0;
}

int isFull(Stack *stack) {                      //判断栈是否为满
    return stack->top == MAXSIZE;
}

void push(Stack *stack, ElemType e) {           //入栈操作,假设栈未满
    stack->elements[stack->top++] = e;
}

ElemType pop(Stack *stack) {                    //出栈操作,假设栈非空
    return stack->elements[--stack->top];
}

ElemType getTop(Stack *stack) {                 //取栈顶元素,假设栈非空
    return stack->elements[stack->top-1];
}
```

该算法的 Java 语言描述如下。

```
public class Stack {
    private final int MAXSIZE = 100;    //MAXSIZE 为栈的最大长度
    private ElemType[] elements;
    private int top;
    public Stack() {
        elements = new ElemType[MAXSIZE];
        top = 0;
    }

    public int getSize() {              //返回栈的长度
        return top;
    }

    public boolean isEmpty() {          //判断栈是否为空
        return top == 0;
```

```
    }

    public boolean isFull() {              //判断栈是否为满
        return top == MAXSIZE;
    }

    public void push(ElemType e) {         //入栈操作,假设栈未满
        elements[top ++] = e;
    }

    public ElemType pop() {                //出栈操作,假设栈非空
        return elements[ -- top];
    }

    public ElemType getTop() {             //取栈顶元素,假设栈非空
        return elements[top - 1];
    }
}
```

上述操作的时间复杂度均为 $O(1)$。

3. 双端栈

双端栈就是将一个顺序表的两端分别作为一个栈的栈底来实现两个栈，两个栈共享一组地址连续的存储空间。两个栈其中一个采用表 3.1 中的方案 1 或方案 2，另外一个采用表 3.1 中的方案 3 或方案 4。例如，如图 3.3 所示的双端栈即为一个栈采用表 3.1 中的方案 1(top1 值为该栈的栈顶元素下标＋1)，另外一个采用表 3.1 中的方案 3(top2 值为该栈的栈顶元素下标－1)。

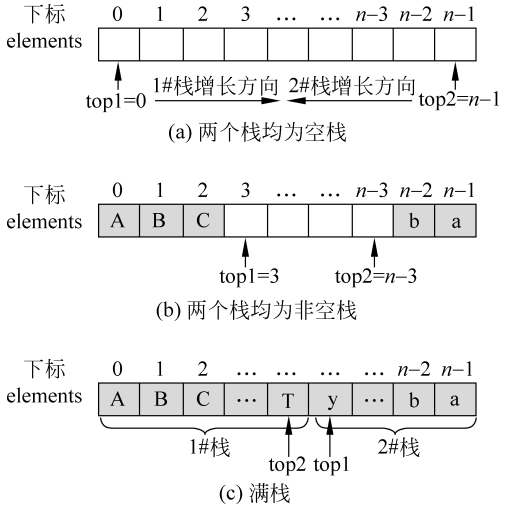


图 3.3　双端栈

4. 顺序栈的扩容

顺序栈的定义和操作都很简单，但需要预估栈的最大长度，如果最大长度取值小就可能造成上溢，如果最大长度取值大就可能造成不必要的空间浪费。

算法 3.1 中已指定栈空间的大小，不能动态改变，为了便于栈的扩容，可以动态申请适量的空间，当上溢时另外申请更大的空间(一般是原空间的 2 倍)，将栈的所有元素复制到新

的空间后，再将原空间释放。

3.1.3　链式栈

栈也可以采用链式存储结构，一般采用单链表，将单链表的头部作为栈顶，入栈和出栈操作均可在$O(1)$时间内完成。单链表带有或者不带附设头结点均可。如图 3.4 所示为链式栈。

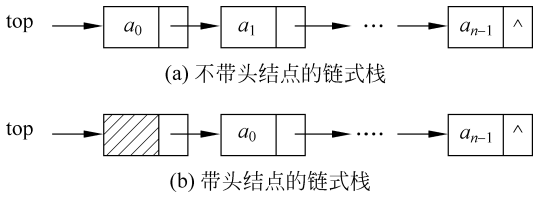


图 3.4　链式栈

对于不带头结点的链式栈：

(1) 空栈。

top 为空指针。

(2) 元素 e 入栈。

申请新结点 node，其数据域值为 e，将 node 插入在原来的首元结点之前。

(3) 元素 e 出栈。

当栈非空时，将首元结点数据域的值赋值给 e，并删除首元结点。

对于带头结点的链式栈：

(1) 空栈。

top 所指结点的链接域为空指针。

(2) 元素 e 入栈。

申请新结点 node，其数据域值为 e，将 node 插入在头结点之后。

(3) 元素 e 出栈。

当栈非空时，将首元结点的数据域赋值给 e，并删除首元结点。

算法 3.2　不带头结点的链式栈的存储结构定义及其操作。

该算法的 C 语言描述如下。

```
typedef struct node {
    ElemType data;
    struct node * next;
} StackNode;

typedef struct {
    StackNode * top;
    int size;
} Stack;

void init(Stack * stack) {                    //初始化空栈
    stack->top = NULL;
    stack->size = 0;
}
```

```
void destroy(Stack *stack) {                    //释放单链表所占空间
    StackNode *p = stack->top;
    while(p != NULL) {
        StackNode *next = p->next;
        free(p);
        p = next;
    }
    stack->size = 0;
}

int getSize(Stack *stack) {                     //返回栈的长度
    return stack->size;
}

int isEmpty(Stack *stack) {                     //判断栈是否为空
    return stack->top == NULL;                  //或者 stack->size == 0
}

void push(Stack *stack, ElemType e) {           //入栈操作
    StackNode *node = (StackNode *)malloc(sizeof(StackNode));
    node->data = e;
    node->next = stack->top;
    stack->top = node;
    stack->size ++;
}

ElemType pop(Stack *stack) {                    //出栈操作,假设栈非空
    StackNode *p = stack->top;
    ElemType e = p->data;
    stack->top = p->next;
    free(p);
    stack->size --;
    return e;
}

ElemType getTop(Stack *stack) {                 //取栈顶元素,假设栈非空
    return stack->top->data;
}
```

该算法的 Java 语言描述如下。

```
public class Stack {                            //链式栈的定义
    private SLinkNode top;
    int size;

    public Stack() {                            //初始化空栈
        top = null;
        size = 0;
    }

    public int getSize() {                      //返回栈的长度
        return size;
    }

    public boolean isEmpty() {                  //判断栈是否为空
        return top == null;                     //或者 size == 0
```

```
    }

    public void push(ElemType e) {          //e 入栈
        SLinkNode node = new SLinkNode(e, top);
        top = node;
        size ++;
    }

    public ElemType pop() {                 //出栈
        if(top == null) return null;
        ElemType e = top.getData();
        top = top.getNext();
        size --;
        return e;
    }

    public ElemType getTop() {              //取栈顶元素
        return top == null ? null : top.getData();
    }
}
```

3.2 队列

3.2.1 基本概念

1. 基本术语

队列(queue)简称为队,也是一种操作受限的线性表,它仅允许在线性表的一端进行插入操作,在另外一端进行删除操作,允许进行插入操作的一端称为**队尾**(rear),允许进行删除操作的一端称为**队头**(front),当队列中没有数据元素时称该队列为**空队**,队列中现有元素个数称为队列的**长度**。

向一个队列中插入一个数据元素的操作称为**入队**或**进队**;从队列中删除一个数据元素的操作称为**出队**或**离队**。当队列已满时,如果还要有元素入队,就会产生"**上溢**",上溢是一种错误,使得问题处理无法进行。当队列是空队时,如果还要进行出队操作,就会产生"**下溢**",下溢一般表明问题处理结束。

队列具有"先进先出"(First In First Out,FIFO)或者"后进后出"(Last In Last Out,LILO)的操作特点,这是由于入队操作仅允许在队尾进行,出队操作仅允许在队头进行,先入队的元素必定排在后入队的元素之前。

一个队列及其入队、出队操作如图 3.5 所示。

在日常生活中经常遇到需要排队的情况,一定要表现出良好的个人素质和社会公德,按序排队,不要插队。

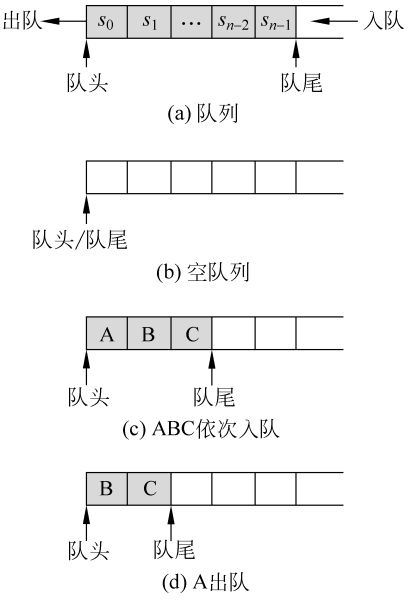


图 3.5 队列及其入队、出队操作示例

2. 队列的抽象数据类型

队列的核心操作是入队和出队，也有其他一些操作，下面给出队列的抽象数据类型定义。

```
ADT Queue {
    数据对象：D = {a_i | a_i ∈ ElementSet, i = 0, 1, 2, …, n - 1, n≥0}
    数据关系：R = {<a_i, a_i+1> | a_i, a_i+1 ∈ D, i = 0, 1, 2, …, n - 2, n≥0}
    基本操作：
        init()：初始化空队列。
        destroy()：销毁队列，释放动态分配给队列的存储空间。
        getSize()：返回队列的长度。
        isEmpty()：判断队列是否为空，为空则返回 true，否则返回 false。
        isFull()：判断队列是否为满，为满则返回 true，否则返回 false。
        enQueue(e)：将数据元素 e 入队。
        deQueue()：出队并返回出队的数据元素。
        getFront()：返回队头元素。
}
```

3.2.2 顺序队列

类似于顺序栈，顺序队列也是用一维数组来实现。

1. 顺序队列

假设所用一维数组为 elements，第一种实现方法是以 elements[0]作为队头元素，另外定义一个下标 last 来指示队尾元素下标，在元素 e 入队时将 last 增 1 并将 e 赋值给 elements[last]；但在出队时，则必须将 elements[1]～elements[last]依次前移并将 last 减 1。这种实现方法效率低，入队的时间复杂度为 $O(1)$，但出队的时间复杂度则为 $O(n)$。

第二种实现方法是将数组 elements[0 .. $n-1$]看作首尾相接的一个圆环，elements[0]接在 elements[$n-1$]之后，用 front 存储队头元素下标，rear 存储队尾元素下一个位置的下标。这样实现的顺序队列称为循环队列，如图 3.6 所示。

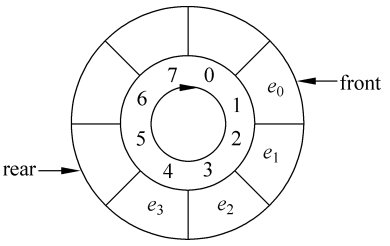
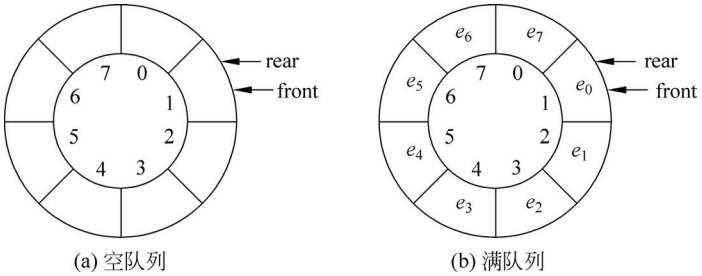


图 3.6　循环队列(1)

按照上述方案，队列的长度为(rear－front＋n) % n，初始化一个空队列就是将 front 和 rear 赋值为一个相同的值。但是，这里存在一个问题，当队列已满时，front 和 rear 的值也相同，这就产生了“二义性”，当 front = rear 时，无法区分队列是空的还是满的，如图 3.7 所示。



(a) 空队列　　(b) 满队列

图 3.7　循环队列

解决这个问题的方案有两个。

(1) 方案 1：少用一个元素单元。

假设用长度为 n 的数组 elements 来实现循环队列，限定队列的实际长度最大为 $n-1$，就能够有效区分循环队列是空队列和满队列两种状态。

- 当 front = rear 时队列为空。
- 当(rear+1) % n = front 时队列为满。
- 队列长度 =(rear−front+n) % n。

(2) 方案 2：引入队列长度 size。

假设用长度为 n 的数组 elements 来实现循环队列，引入一个 size 存储队列长度，队列的长度最大为 n，则当 size = 0 时队列为空队列，当 size = n 时队列为满队列。

上述两种方案的有关量、状态以及操作如表 3.2 所示。

表 3.2 循环队列有关量、状态以及操作

方案 1：少用一个元素单元		方案 2：引入队列长度 size	
量、状态及操作	描述及操作实现	量、状态及操作	描述及操作实现
front	队头元素下标	front	队头元素下标
rear	(队尾元素下标+1)%n	rear	(队尾元素下标+1)%n
队列长度	(rear−front+n)%n	队列长度	size
队列空	front==rear	队列空	size==0
队列满	(rear+1)%n==front	队列满	size==n
初始化空队列	front=rear=0;	初始化空队列	front=rear=size=0;
元素 e 入队	elements[rear]=e; rear=(rear+1)%n;	元素 e 入队	elements[rear]=e; rear=(rear+1)%n; size++;
元素 e 出队	e=elements[front]; front=(front+1)%n;	元素 e 出队	e=elements[front]; front=(front+1)%n; size--;

2. 循环队列的实现

算法 3.3 循环队列(方案 1：少用一个元素单元)。

该算法的 C 语言描述如下。

```
#define MAXSIZE         100                 //数组长度为 MAXSIZE,队列最长为 MAXSIZE - 1
typedef struct {
    ElemType elements[MAXSIZE];
    int front, rear;                    //front 为队头元素下标,rear 为队尾元素下一个位置下标
} Queue;

void init(Queue * queue) {                  //初始化空队列
    queue->front = queue->rear = 0;
}

int getSize(Queue * queue) {                //返回队列长度
    return (queue->rear - queue->front + MAXSIZE) % MAXSIZE;
}

int isEmpty(Queue * queue) {                //判断队列是否为空
    return queue->front == queue->rear;
```

```
}

int isFull(Queue * queue) {                    //判断队列是否为满
    return (queue->rear+1) % MAXSIZE == queue->front;
}

void enQueue(Queue * queue, ElemType e) {   //e 入队,假设队列未满
    queue->elements[queue->rear] = e;
    queue->rear = (queue->rear + 1) % MAXSIZE;
}

ElemType deQueue(Queue * queue) {           //e 出队,假设队列非空
    ElemType e = queue->elements[queue->front];
    queue->front = (queue->front + 1) % MAXSIZE;
    return e;
}

ElemType getFront(Queue * queue) {          //返回队头元素,假设队列非空
    return queue->elements[queue->front];
}
```

该算法的 Java 语言描述如下。

```
public class Queue {
    //数组长度为 MAXSIZE,队列最长为 MAXSIZE-1
    private final int MAXSIZE = 100;
    private ElemType[] elements;
    //front 为队头元素下标,rear 为队尾元素下一个位置下标
    private int front, rear;
    public Queue() {                    //初始化空队列
        elements = new ElemType[MAXSIZE];
        front = rear = 0;
    }

    public int getSize() {              //返回队列长度
        return (rear - front + MAXSIZE) % MAXSIZE;
    }

    public boolean isEmpty() {          //判断队列是否为空
        return front == rear;
    }

    public boolean isFull() {           //判断队列是否为满
        return (rear + 1) % MAXSIZE == front;
    }

    public void enQueue(ElemType e) { //e 入队,假设队列未满
        elements[rear] = e;
        rear = (rear + 1) % MAXSIZE;
    }

    public ElemType deQueue() {         //出队,假设队列非空
        ElemType e = elements[front];
        front = (front + 1) % MAXSIZE;
        return e;
    }
```

```
    public ElemType getFront() {      //返回队头元素,假设队列非空
        return elements[front];
    }
}
```

算法 3.4 循环队列(方案 2:引入 size 存放队列长度)。

该算法的 C 语言描述如下。

```
#define MAXSIZE        100            //数组长度为 MAXSIZE
typedef struct {
    ElemType elements[MAXSIZE];
    int front, rear;             //front 为队头元素下标,rear 为队尾元素下一个位置下标
    int size;                      //存放队列长度
} Queue;

void init(Queue * queue) {             //初始化空队列
    queue->front = queue->rear = queue->size = 0;
}

int getSize(Queue * queue) {           //返回队列长度
    return queue->size;
}

int isEmpty(Queue * queue) {           //判断队列是否为空
    return queue->size == 0;
}

int isFull(Queue * queue) {            //判断队列是否为满
    return queue->size == MAXSIZE;
}

void enQueue(Queue * queue, ElemType e) {//e 入队,假设队列未满
    queue->elements[queue->rear] = e;
    queue->rear = (queue->rear + 1) % MAXSIZE;
    queue->size ++;
}

ElemType deQueue(Queue * queue) {      //e 出队,假设队列非空
    ElemType e = queue->elements[queue->front];
    queue->front = (queue->front + 1) % MAXSIZE;
    queue->size --;
    return e;
}

ElemType getFront(Queue * queue) {     //返回队头元素,假设队列非空
    return queue->elements[queue->front];
}
```

该算法的 Java 语言描述如下。

```
public class Queue {
    private final int MAXSIZE = 100;
    private ElemType[] elements;
    //front 为队头元素下标,rear 为队尾元素下一个位置下标
    private int front, rear;
    private int size;                  //size 存队列长度
    public Queue() {                   //初始化空队列
        elements = new ElemType[MAXSIZE];
```

```
        front = rear = size = 0;
    }

    public int getSize() {              //返回队列长度
        return size;
    }

    public boolean isEmpty() {          //判断队列是否为空
        return size == 0;
    }

    public boolean isFull() {           //判断队列是否为满
        return size == MAXSIZE;
    }

    public void enQueue(ElemType e) {  //e 入队,假设队列未满
        elements[rear] = e;
        rear = (rear + 1) % MAXSIZE;
        size ++;
    }

    public ElemType deQueue() {         //出队,假设队列非空
        ElemType e = elements[front];
        front = (front + 1) % MAXSIZE;
        size --;
        return e;
    }

    public ElemType getFront() {        //返回队头元素,假设队列非空
        return elements[front];
    }
}
```

循环队列所有操作的时间复杂度均为 $O(1)$。

前面所讲循环队列中将 front 定义为队头元素下标,将 rear 定义为队尾元素下一个位置下标,也可以采用其他方案,例如,将 front 定义为队头元素前一个位置下标,而将 rear 定义为队尾元素下标,具体的实现与算法 3.3 和算法 3.4 有一点差异。

3.2.3 链式队列

队列也可以采用链式存储结构,一般采用单链表,将单链表的头部作为队头,尾部作为队尾,入队和出队操作均可在 $O(1)$时间内完成。单链表带有或者不带附设头结点均可,但一般采用带头结点的单链表。如图 3.8 所示为链式队列,其中,front 指向队头结点(对于不带头结点的单链表)或头结点(对于带头结点的单链表)。

对于不带头结点的链式队列,操作如下。

(1) 空队列:front 和 rear 均为空指针。

(2) 元素 e 入队。

① 申请新结点 node,其数据域值为 e。

② 如果队列为空,则将 front 和 rear 均指向 node。

③ 如果队列非空,则将 node 插入在 rear 所指结点之后,然后令 rear 指向 node。

(3) 元素 e 出队(假设队列非空)。

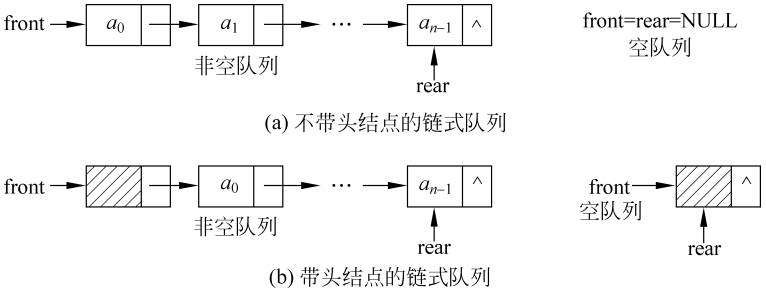


图 3.8　链式队列

① 将 front 所指结点数据域赋值给 e。

② 如果队列长度为 1，则删除 front 所指结点，将 front 和 rear 赋值为空指针。

③ 如果队列长度大于 1，则删除 front 所指结点，并令 front 指向下一个结点。

对于带头结点的链式队列的操作则比较简单。

(1) 空队列：front 和 rear 所指结点的链接域为空指针，或者 front == rear。

(2) 元素 e 入队：申请新结点 node，其数据域值为 e，将 node 插入在 rear 所指结点之后，令 rear 指向 node。

(3) 元素 e 出队：当队列非空时，将首元结点的数据域赋值给 e，并删除首元结点，如果队列变为空，则令 rear 指向头结点。

算法 3.5　带头结点的链式队列的存储结构定义及其操作。

该算法的 C 语言描述如下。

```
typedef struct node {
    ElemType data;
    struct node * next;
} QueueNode;

typedef struct {
    QueueNode * front, * rear;                    //front 指向头结点,rear 指向队尾结点
    int size;                                     //size 存队列长度
} LinkedQueue;

void init(LinkedQueue * queue) {                  //初始化空队列
    QueueNode * head = (QueueNode * )malloc(sizeof(QueueNode));
    head -> next = NULL;
    queue -> front = queue -> rear = head;
    queue -> size = 0;
}

void clear(LinkedQueue * queue) {                 //清空队列,队列长度变为 0
    QueueNode * ptr = queue -> front -> next;
    while(ptr != NULL) {
        QueueNode * next = ptr -> next;
        free(ptr);
        ptr = next;
    }
    queue -> rear = queue -> front;
    queue -> rear -> next = NULL;
    queue -> size = 0;
}
```

```
void destroy(LinkedQueue * queue) {                    //销毁队列,释放单链表所占空间
    clear(queue);
    free(queue->front);
}

int getSize(LinkedQueue * queue) {                     //返回队列长度
    return queue->size;
}

int isEmpty(LinkedQueue * queue) {                     //判断队列是否为空
    return queue->size == 0;
}

void enQueue(LinkedQueue * queue, ElemType e) {        //e 入队
    QueueNode * node = (QueueNode * )malloc(sizeof(QueueNode));
    node->data = e;
    node->next = NULL;
    queue->rear->next = node;
    queue->rear = node;
    queue->size ++;
}

ElemType deQueue(LinkedQueue * queue) {                //出队,假设队列非空
    QueueNode * node = queue->front->next;
    ElemType e = node->data;
    queue->front->next = node->next;
    free(node);
    queue->size --;
    if(queue->size == 0) {
        queue->rear = queue->front;
    }
    return e;
}

ElemType getFront(LinkedQueue * queue) {               //返回队头元素,假设队列非空
    return queue->front->next->data;
}
```

该算法的 Java 语言描述如下。

```
public class LinkedQueue {
    private SLinkNode front, rear;                     //front 指向头结点,rear 指向队尾结点
    private int size;                                  //size 存队列长度

    public LinkedQueue() {                             //初始化空队列
        front = new SLinkNode(null, null);
        rear = front;
        size = 0;
    }

    public void clear() {                              //清空队列,队列长度变为 0
        front.setNext(null);
        rear = front;
        size = 0;
    }
```

```
    public int getSize() {                          //返回队列长度
        return size;
    }

    public boolean isEmpty() {                      //判断队列是否为空
        return size == 0;
    }

    public void enQueue(ElemType e) {               //e 入队
        SLinkNode node = new SLinkNode(e, null);
        rear.setNext(node);
        rear = node;
        size ++;
    }

    public ElemType deQueue() {                     //出队
        if(size == 0) return null;
        SLinkNode node = front.getNext();
        ElemType e = node.getData();
        front.setNext(node.getNext());
        size --;
        if(size == 0) {
            rear = front;
        }
        return e;
    }

    public ElemType getFront() {                    //返回队头元素
        return size == 0 ? null : front.getNext().getData();
    }
}
```

3.2.4 优先队列

前面所讲队列都遵循“先进先出”的操作原则，但也有一种队列并不遵循该原则，那就是优先队列。

日常生活中也能够遇到类似于优先队列的场景，例如，很多车辆在排队通过某一路口，原则上应该是排在前面的车辆先通行，排在后面的车辆后通行，但此时来了一辆拉着患者的救护车或者赶往火灾现场的消防车，当然必须是救护车或者消防车优先通行。遇到这些特殊车辆，应该也必须礼让，让这些特殊车辆先行。

优先队列(priority queue)中的每个元素都具有用数值表示的优先级，在出队时，并非排在队头的元素出队，而是队列中优先级最高的元素出队；对于优先级相同的元素，可按先进先出或者任意顺序出队。

利用目前所学知识，可以得到如下优先队列实现方案。

1. 出队时选择具有最高优先级的元素

新元素入队时排在队尾，但在出队时，选择具有最高优先级的元素出队，空出的位置由原队尾元素填补。采用该方案，入队操作的时间复杂度为 $O(1)$，出队操作的时间复杂度最差为 $O(n)$。

下面以顺序存储结构的优先队列为例描述其存储结构和算法，只包含初始化、入队和出

队操作算法,其他操作算法请自行设计。

算法 3.6　顺序存储结构的优先队列。

用 C 语言描述如下。

```
#define MAXSIZE         100
typedef struct {
    ElemType data;
    int priority;
} QueueNode;
typedef struct {
    QueueNode elements[MAXSIZE];
    int size;
} PriorityQueue;

//初始化空队列
void init(PriorityQueue *queue) {
    queue->size = 0;
}

//node 入队
int enQueue(PriorityQueue *queue, QueueNode node) {
    if(queue->size == MAXSIZE) {
        return 0;
    }
    queue->elements[queue->size++] = node;
    return 1;
}

//出队,出队元素传给 *node
int deQueue(PriorityQueue *queue, QueueNode *node) {
    if(queue->size == 0) {
        return 0;
    }
    int idx = 0;
    for(int i = 1; i < queue->size; i++) {
        if(queue->elements[i].priority > queue->elements[idx].priority) {
            idx = i;
        }
    }
    *node = queue->elements[idx];
    if(idx != queue->size - 1) {
        queue->elements[idx] = queue->elements[queue->size-1];
    }
    queue->size--;
    return 1;
}
```

用 Java 语言描述如下。

```
public class QueueNode {
    private ElemType data;
    private int priority;
    public QueueNode(ElemType data, int priority) {
        this.data = data;
        this.priority = priority;
    }
```

```
    public ElemType getData() {
        return data;
    }
    public int getPriority() {
        return priority;
    }
    public void setData(ElemType data) {
        this.data = data;
    }
    public void setPriority(int priority) {
        this.priority = priority;
    }
}

public class PriorityQueue {
    private final int MAXSIZE = 100;
    private QueueNode[] elements;
    private int size;
    public PriorityQueue() {
        elements = new QueueNode[MAXSIZE];
        size = 0;
    }

    //node 入队
    public boolean enQueue(QueueNode node) {
        if(size == MAXSIZE) {
            return false;
        }
        elements[size ++] = node;
        return true;
    }

    //出队,返回出队元素
    public QueueNode deQueue() {
        if(size == 0) {
            return null;
        }
        int idx = 0;
        for(int i = 1; i < size; i ++) {
            if(elements[i].getPriority() > elements[idx].getPriority()) {
                idx = i;
            }
        }
        QueueNode node = elements[idx];
        if(idx != size - 1) {
            elements[idx] = elements[size - 1];
        }
        size --;
        return node;
    }
}
```

2. 入队时按优先级插入队列中

队列中的所有元素按优先级非递减排列,下标为 0 的元素优先级最低,下标为 size－1 的元素优先级最高,可见,该方案以下标为 0 的元素作为队尾元素,以下标为 size－1 的元素作为队头元素。新元素入队时插入队列中的适当位置,使得队列中所有元素依然是有序的,

出队时队头元素(下标为 size－1)出队。采用该方案,入队操作的时间复杂度最差为 $O(n)$,出队操作的时间复杂度为 $O(1)$。

算法 3.7　顺序存储结构的优先队列。

用 C 语言描述如下。

```
#define MAXSIZE         100
typedef struct {
    ElemType data;
    int priority;
} QueueNode;

typedef struct {
    QueueNode elements[MAXSIZE];
    int size;
} PriorityQueue;

//初始化空队列
void init(PriorityQueue *queue) {
    queue->size = 0;
}

//node 入队
int enQueue(PriorityQueue *queue, QueueNode node) {
    if(queue->size == MAXSIZE) {
        return 0;
    }
    int i = queue->size-1;
    while(i >= 0 && node.priority < queue->elements[i].priority) {
        queue->elements[i+1] = queue->elements[i];
        i--;
    }
    queue->elements[i+1] = node;
    queue->size++;
    return 1;
}

//出队,出队元素传给 *node
int deQueue(PriorityQueue *queue, QueueNode *node) {
    if(queue->size == 0) {
        return 0;
    }
    *node = queue->elements[--queue->size];
    return 1;
}
```

用 Java 语言描述如下。

```
public class QueueNode {
    private ElemType data;
    private int priority;
    public QueueNode(ElemType data, int priority) {
        this.data = data;
        this.priority = priority;
    }

    public ElemType getData() {
```

```
        return data;
    }
    public int getPriority() {
        return priority;
    }
    public void setData(ElemType data) {
        this.data = data;
    }
    public void setPriority(int priority) {
        this.priority = priority;
    }
}

public class PriorityQueue {
    private final int MAXSIZE = 100;
    private QueueNode[] elements;
    private int size;
    public PriorityQueue() {
        elements = new QueueNode[MAXSIZE];
        size = 0;
    }

    //node 入队
    public boolean enQueue(QueueNode node) {
        if(size == MAXSIZE) {
            return false;
        }
        int i = size - 1;
        while(i >= 0 && node.getPriority() < elements[i].getPriority()) {
            elements[i + 1] = elements[i];
            i --;
        }
        elements[i + 1] = node;
        size ++;
        return true;
    }

    //出队,返回出队元素
    public QueueNode deQueue() {
        return size == 0 ? null : elements[ -- size];
    }
}
```

第 8 章将介绍“堆”的相关知识,如果将队列中所有元素按照“堆”来组织,则入队和出队的时间复杂度均可达到 $O(\log n)$。

3.3 栈和队列的应用

3.3.1 栈的应用

1. 进制转换与括号匹配

进制转换是一个很常见的问题,例如,将一个正整数转换为二进制、八进制、十进制或者十六进制等形式。假设要将一个正整数 N 转换为 m 位 d 进制数(d 是大于或等于 2 的正整

数)$n_{m-1}n_{m-2}\cdots n_1 n_0$ 形式,其中,n_i 是它的各位数字($i=0,1,\cdots,m-1$),方法如下。

(1) 令 $i=0$。

(2) 令 $n_i=N\%d$,$N=N/d$(这里%和/分别是求余和整除运算),$i++$。

(3) 重复执行步骤(2),直到 $N=0$。

可以看出,最先求得的是最低位 n_0,然后是次低位 n_1,…,最后才是最高位 n_{m-1},而一般都是从高位到低位依次输出,这恰好符合栈"先进后出"的操作特点,可以用栈来依次保存求得的各位数字,然后再依次出栈并输出。

算法 3.8　将正整数 *N* 转换为 *d* 进制数。

该算法的 C 语言描述如下。

```
void conversion(int N, int d) {
    Stack stack;
    init(&stack);
    do {
        push(&stack, N % d);
        N = N / d;
    }
    while(N > 0);
    char digit[] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    while(!isEmpty(&stack)) {
        //出栈并输出一位数字
        int dig = pop(&stack);
        putchar(digit[dig]);
    }
}
```

该算法的 Java 语言描述如下。

```
public void conversion(int N, int d) {
    Stack stack = new Stack();
    do {
        stack.push(N % d);
        N = N / d;
    }
    while(N > 0);
    String digit = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    while(!stack.isEmpty()) {
        //出栈并输出一位数字
        int dig = stack.pop();
        System.out.print(digit.charAt(dig));
    }
}
```

当然,也可以用递归函数描述如下。

C 语言描述:

```
void conversion(int N, int d) {
    char digit[] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    if(N == 0) {
        putchar('0');
    }
    else {
        conversion(N / d);                    //将去掉最低位数字后的数值转换为 d 进制数
        putchar(digit[N % d]);                //输出最低位数字
```

```
    }
}
```

Java 语言描述：

```
void conversion(int N, int d) {
    String digit = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    if(N == 0) {
        System.out.print('0');
    }
    else {
        conversion(N / d);                          //将去掉最低位数字后的数值转换为 d 进制数
        System.out.print(digit.charAt(N % d)); //输出最低位数字
    }
}
```

可以看出，用递归形式更加简洁，形式上也看不到栈的应用。但实际上，对于函数调用尤其是函数的递归调用，栈都在“幕后”默默做着大量的工作。

还有一个典型问题可以用栈来解决，那就是表达式中的括号匹配问题。假设表达式中可以包含三类括号：圆括号（小括号）、方括号（中括号）和花括号（大括号），且可以相互嵌套，但必须能够正确配对。例如，({[]}(()){})就是正确的格式，而}{[()]}{或者{[(])}就是不正确的格式。

根据表达式括号的配对规则，对于先出现的左括号 1 和后出现的左括号 2，后出现的左括号 2 应先于先出现的左括号 1 得到对应的右括号的匹配，这恰好符合“先进后出”的特点，可以用栈来实现。

算法 3.9　对一个字符串中的括号进行配对，正确配对则返回 1（true），否则返回 0（false）。

该算法的 C 语言描述如下。

```
int bracketMatch(char str[], int n) {
    Stack stack;
    init(&stack);
    for(int i = 0; i < n; i ++) {
        char c = str[i];
        if(c == '(' || c == '[' || c == '{') {          //遇到左括号则入栈
            push(&stack, c);
        }
        else if(c == ')' || c == ']' || c == '}') {
        //遇到右括号则出栈并配对，期待出栈的是匹配的左括号
            if(isEmpty(&stack)) return 0;               //若栈为空则说明缺失匹配的左括号
            char left = pop(&stack);
            if(!(left == '(' && c == ')' || left == '[' && c == ']'
                || left == '{' && c == '}')) {
                return 0;                               //不匹配则返回 0
            }
        }
    }
    return isEmpty(&stack) ? 1 : 0;                     //若栈非空则说明存在多余的左括号
}
```

该算法的 Java 语言描述如下。

```
public boolean bracketMatch(char[] str, int n) {
```

```
    Stack stack = new Stack();
    for(int i = 0; i < n; i ++) {
        char c = str[i];
        if(c == '(' || c == '[' || c == '{') {          //遇到左括号则入栈
            stack.push(c);
        }
        else if(c == ')' || c == ']' || c == '}') {
        //遇到右括号则出栈并配对,期待出栈的是匹配的左括号
            if(stack.isEmpty()) return 0;               //若栈为空则说明缺失匹配的左括号
            char left = stack.pop();
            if(!(left == '(' && c == ')' || left == '[' && c == ']'
                || left == '{' && c == '}')) {
                return 0;                               //不匹配则返回 0
            }
        }
    }
    return stack.isEmpty() ? 1 : 0;                     //若栈非空则说明存在多余的左括号
}
```

2. 算术表达式计算

一个算术表达式由操作数(运算数)、操作符(运算符)和括号组成,其中,操作符(运算符)有加(+)、减(-)、乘(*)和除(/),假设括号只限定使用左右圆括号且必须正确配对。可以通过栈来实现算术表达式的计算。

日常遇到的算术表达式称为**中缀表达式**(infix expression),即操作符在两个操作数之间,如(3+5)*7-1。中缀表达式的计算规则如下。

(1) 先计算括号内,后计算括号外,如果有多层括号,则先处理内层括号,后处理外层括号。

(2) 无括号或者同层括号内,先乘除,后加减,即乘除运算优先于加减运算。

(3) 对于同一优先级的运算,从左向右依次计算。

运算的优先级对于中缀表达式的计算至关重要。假设整个表达式以"="结束,将左右括号和等号也考虑在内考察相邻运算符的优先级关系,如表 3.3 所示,其中,优先级比较结果的">"表示前一个运算符优先于相邻的后一个运算符,"<"表示后一个运算符优先于相邻的前一个运算符,"="表示左右括号配对。

表 3.3 前后两个相邻运算的优先级比较

前 \ 后	+	-	*	/	(	)	=
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	非法
)	>	>	>	>	非法	>	>

中缀表达式的计算过程如下。

(1) 初始化运算符栈 OPTR 和操作数栈 OPND。

(2) 从左向右扫描表达式。

(3) 遇到操作数,则操作数入栈 OPND。

(4) 遇到运算符 op2:

① 如果运算符栈 OPTR 为空,则 op2 入栈 OPTR。

② 如果 OPTR 非空,则取 OPTR 栈顶运算符 op1,比较 op1 和 op2 的优先级。

- 如果 op1<op2,则 op2 入栈 OPTR。
- 如果 op1>op2,则 op1 从 OPTR 出栈,从 OPND 出栈两个操作数 n2 和 n1(注意先出栈的是 n2,后出栈的是 n1),以 n1 为左操作数,n2 为右操作数,完成 op1 运算,结果入栈 OPND,并继续进行②的比较。
- 如果 op1=op2(左右括号配对),则将 OPTR 栈顶的左括号出栈。
- 如果是"非法",则结束,不再计算,因为表达式格式错误。

(5) 表达式结束时,OPTR 栈应该为空,如果不为空则表达式格式错误,否则 OPND 栈仅剩的唯一数值就是表达式的计算结果,如果 OPND 栈为空栈或者长度大于 1 则表达式格式错误。

算术表达式除了有中缀表达式以外,还有前缀表达式和后缀表达式。

前缀表达式(prefix expression)也称为波兰式,该表达式中不存在括号,只包含运算符和操作数,且运算符在两个操作数之前,例如,- * + 3 5 7 1 就是一个合规的前缀表达式,其中,3、5、7、1 各是一个操作数。

前缀表达式的计算规则如下。

(1) 从右向左扫描表达式。

(2) 遇到运算符 op 就将后面的两个操作数 n1 和 n2,以 n1 为左操作数,n2 为右操作数进行 op 运算,运算结果依然放在 op、n1 和 n2 原位。

(3) 一直扫描计算下去,直到产生计算结果。

例如,对于前缀表达式- * + 3 5 7 1,为了看着清晰,人为写出-,*,+,3,5,7,1。先计算 3+5,表达式变为-,*,8,7,1,再计算 8*7,表达式变为-,56,1,最后计算 56-1,结果为 55。

前缀表达式的计算过程如下。

(1) 初始化操作数栈 OPND。

(2) 从右向左扫描表达式。

(3) 遇到操作数,则操作数入栈 OPND。

(4) 遇到运算符 op,则从栈 OPND 出栈两个操作数 n1 和 n2(注意先出栈的是 n1,后出栈的是 n2),以 n1 为左操作数 n2 为右操作数,完成 op 运算,结果入栈 OPND。

(5) 重复上述过程(3)和(4),直到表达式最左端,最后栈 OPND 中仅剩的唯一数值就是表达式的计算结果,如果 OPND 长度不为 1 则说明表达式错误。

后缀表达式(suffix expression)也称为逆波兰式,该表达式中也不存在括号,只包含运算符和操作数,且运算符在两个操作数之后,例如,3 5 + 7 * 1 -就是一个合规的后缀表达式,其中,3、5、7、1 各是一个操作数。

后缀表达式的计算规则如下。

(1) 从左向右扫描表达式。

(2) 遇到运算符 op 就将前面的两个操作数 n1 和 n2,以 n1 为左操作数 n2 为右操作数进行 op 运算,运算结果依然放在 n1、n2 和 op 原位。

(3) 一直扫描计算下去,直到产生计算结果。

例如,对于后缀表达式 3 5 + 7 * 1−,为了看得清晰,人为写出 3,5,+,7,*,1,−。先计算 3+5,表达式变为 8,7,*,1,−,再计算 8*7,表达式变为 56,1,−,最后计算 56−1,结果为 55。

后缀表达式的计算过程如下。

(1) 初始化操作数栈 OPND。

(2) 从左向右扫描表达式。

(3) 遇到操作数,则操作数入栈 OPND。

(4) 遇到运算符 op,则从栈 OPND 出栈两个操作数 n2 和 n1(注意先出栈的是 n2,后出栈的是 n1),以 n1 为左操作数,n2 为右操作数,完成 op 运算,结果入栈 OPND。

(5) 重复上述过程(3)和(4),直到表达式最右端,最后栈 OPND 中仅剩的唯一数值就是表达式的计算结果,如果 OPND 长度不为 1 则说明表达式错误。

与中缀表达式相比,前缀表达式和后缀表达式的计算过程非常简单,也不用考虑运算符之间的优先级。

3.3.2 栈与递归

1. 函数调用中的栈

在编写任何稍大一些的程序时,都不可避免地会用到函数调用,而栈在“幕后”做了大量的工作。

当一个函数 A 调用函数 B 时,其基本执行过程如下。

(1) 函数 A 向函数 B 传递参数。

(2) 转到函数 B 的入口地址执行。

(3) 函数 B 取得函数 A 传递过来的参数,进行处理,获得处理结果。

(4) 函数执行完毕后,返回函数 A 中调用函数 B 的指令的下一条指令(返回地址处的指令)继续执行。

该过程如图 3.9 所示。

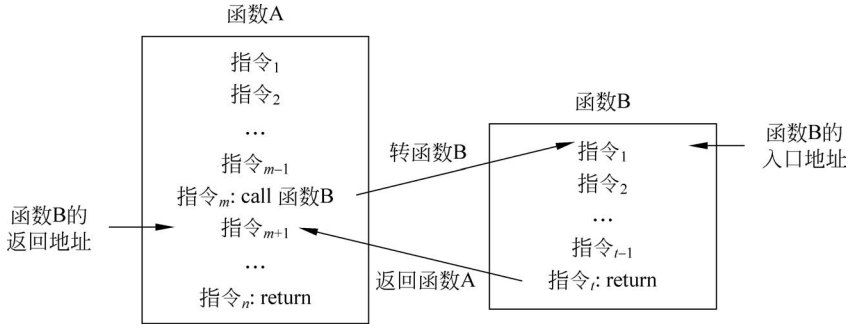


图 3.9 函数调用过程示例

这里存在一个问题：从函数 B 返回函数 A 继续执行时，函数 A 的后续指令需要和调用函数 B 之前的指令具有共同的上下文环境，这一环境可能由于函数 B 的执行而遭到破坏。解决这个问题的方法如下。

(1) 设置一个栈。

(2) 在函数 A 调用函数 B 而转到函数 B 执行之前，将必要的上下文保存在栈中(入栈，保护现场)。

(3) 在从函数 B 返回函数 A 继续执行时，恢复保存过的上下文环境(出栈，恢复现场)。

由于一个程序中普遍存在函数的嵌套调用，例如，函数 A 调用函数 B，函数 B 又调用函数 C，函数 C 又调用函数 D，所以这里保护与恢复现场要用栈来实现，先保存的现场后恢复，后保存的现场先恢复。

以 C 语言为例，函数 A 调用函数 B 以及从函数 B 返回的相关处理如下。

(1) 将函数 A 传递给函数 B 的参数值入栈，返回地址入栈。

(2) 转函数 B 执行。

(3) 函数 B 在栈中为自身的局部自动变量分配空间。

(4) 函数 B 执行自身的指令序列，通过栈访问入口参数，用寄存器保存返回值。

(5) 释放栈中局部自动变量所占空间，返回地址出栈，转返回地址执行。

(6) 函数 A 释放传递给函数 B 的参数所占的栈空间，从对应的寄存器获得函数 B 的返回值。

对应的栈的内容如图 3.10 所示。

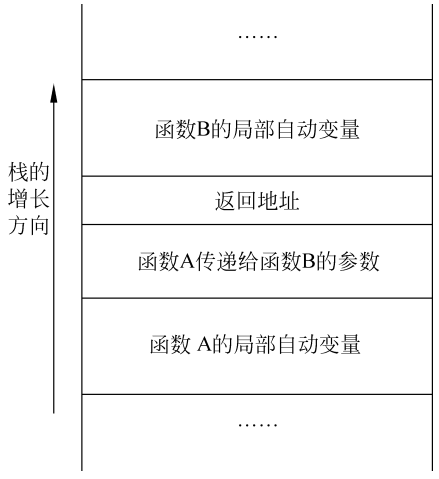


图 3.10　函数 A 调用函数 B 相关的栈的内容

2. 递归

递归(recursion)就是指在自身的定义中又直接或间接地引用了自身。

如果一个概念的定义中出现了用自身来定义自身，则该定义就是一个递归的定义，例如后面要学习的树结构中就有很多递归形式定义的术语。

在之前学习过的单链表结点类型的定义也涉及递归，例如：

```
struct SLinkNode {
    ElemType data;
```

```
    struct SLinkNode * next;
};
public class SLinkNode {
    private ElemType data;
    private SLinkNode next;
    …
}
```

如果一个算法直接或间接地调用了自身，则该算法就是一个递归算法。程序中的函数也一样存在递归函数。前面刚刚讲过，程序中的函数调用需要用栈来保护与恢复现场，递归的函数也必然如此。后续章节将涉及大量的递归算法。

很多数学函数也常常用到递归，例如阶乘函数：

$$\mathrm{fac}(n)=\begin{cases}1 & (n=0)\\ n\times\mathrm{fac}(n-1) & (n>0)\end{cases}$$

又如二阶斐波那契数列：

$$\mathrm{fib}(n)=\begin{cases}1 & (n=0)\\ 1 & (n=1)\\ \mathrm{fib}(n-1)+\mathrm{fib}(n-2) & (n>1)\end{cases}$$

如果用循环迭代方式来实现上述阶乘函数和二阶斐波那契数列，则时间复杂度为 $O(n)$，空间复杂度为 $O(1)$，而用递归函数来实现，则时间复杂度依然为 $O(n)$，但由于需要辅助的栈空间，其空间复杂度则为 $O(n)$。

再如，指数函数 $a^n(n\geqslant 0)$，如果采用连乘 n 个 a 的方法来实现，其时间复杂度为 $O(n)$，空间复杂度为 $O(1)$，而用递归则可写成如下形式。

$$\mathrm{pow}(a,\ n)=a^n=\begin{cases}1 & (n=0)\\ \left[\mathrm{pow}\left(a,\dfrac{n}{2}\right)\right]^2 & (n>0,n\text{ 为偶数})\\ \left[\mathrm{pow}\left(a,\dfrac{n-1}{2}\right)\right]^2\cdot a & (n>0,n\text{ 为奇数})\end{cases}$$

如果该指数函数用递归函数来实现，则时间复杂度和空间复杂度均为 $O(\log n)$。当然，如果根据上述递归公式，但不采用递归函数而是采用适当的循环迭代方法来实现，在保持时间复杂度为 $O(\log n)$的同时空间复杂度可降为 $O(1)$。

3. 递归与分治

在后面章节将学习的二叉树的遍历、快速排序和归并排序等都采用了分治法思想。

分治法的基本思想很容易理解。

(1) 求解一个问题所需时间与问题规模有关，问题规模越小，所需时间越少，问题规模越大，所需时间越多。

(2) 问题规模小到一定程度，可以直接求解。

(3) 对于不能直接求解的大规模问题，可将其分为若干规模较小的同类问题，如果这些小规模问题得以求解，即可将求解结果组合为大规模问题的解。

分治法算法一般由如下三部分组成。

(1) 分解。

将规模为 n 的问题,分解为 $k \geqslant 1$ 个子问题,每个子问题的规模严格小于原问题的规模。

(2) 治理。

如果子问题的规模小到可以直接解决则直接求解,否则递归地求解分解出的各个子问题。

(3) 组合。

将各个子问题的解组合为原问题的解。

在学习、工作和生活中经常会用到分治法,例如,上级主管部门要求学校报送某项涉及全校的材料,学校对应的分管职能部门向全校发出通知,要求各个二级单位报送该项材料,汇总之后报送上级主管部门,而每个二级单位的材料也是从其下属的部门收集来的,这也是分治法。在一个企事业单位中,面对一个大的工作任务,往往需要对任务进行层层分解,只要每个人、每个小组和每个团队都能够将分解指派的任务圆满完成,上下一心,通力合作,整个大的工作任务一定能够获得圆满解决,这也体现了分治法的强大能力。

例如,求具有 n 个整型元素的一维数组 a 的最大值,很容易想到如下算法。

```
int max(int a[], int n) {                    //求 a[0..n-1]的最大值
    int m = a[0];
    for(int i = 1; i < n; i ++) {
        if(a[i] > m) {
            m = a[i];
        }
    }
    return m;
}
```

如果采用分治法,可以将数组 a 分为元素数相近的两个子数组分别求其最大值(若子数组只有一个元素则该元素就是子数组的最大值),两个最大值的较大者即为整个数组的最大值,对应的分治法算法如下。

```
int max(int a[], int low, int high) {          //求 a[low..high]的最大值
    if(low == high) return a[low];
    int mid = (low + high) / 2;
    int m1 = max(a, low, mid);
    int m2 = max(a, mid + 1, high);
    return m1 > m2 ? m1 : m2;
}
```

该问题的分治法算法的时间性能并未得到提高,反而由于递归所需的辅助栈空间使得空间复杂度增加了,但该例子能够简单直观地体现分治法思想。

汉诺塔问题也是一个能够简单直观体现分治法思想的典型问题:有 A、B 和 C 三个位置,在 A 位置从下往上按照由大到小的尺寸摞着 64 个圆盘,要求每次只能在三个位置之间移动一个圆盘,且在任何位置小圆盘上面不能出现大圆盘,最终将 A 位置的 64 个圆盘借助 B 位置移到 C 位置,如何移动?

可以将该问题的分治法解决方案描述如下。

(1) 将 A 位置的 63 个圆盘借助 C 位置移动到 B 位置。

(2) 将 A 位置仅剩的一个圆盘直接移动到 C 位置。

(3) 将 B 位置的 63 个圆盘借助 A 位置移动到 C 位置,至此,问题得到解决。

(4) 至于(1)所述的 63 个圆盘和(3)所述的 63 个圆盘的移动方案,可以依据同样的道理进行分解。

该解决方案的算法描述如下。

```
//将 n 个圆盘从 A 借助 B 移到 C
void hanoi(int n, int A, int B, int C) {
    if(n == 1) {
        move(A, C);                 //如果只有 1 个圆盘则从 A 直接移到 C
    }
    else {
        hanoi(n-1, A, C, B);        //将 n-1 个圆盘从 A 借助 C 移到 B
        move(A, C);                 //A 的 1 个圆盘直接移到 C
        hanoi(n-1, B, A, C);        //将 n-1 个圆盘从 B 借助 A 移到 C
    }
}
```

3.3.3 队列的应用

在后面的树结构中的按层次遍历和图结构中的广度优先搜索都要利用队列来实现。在计算机领域中,队列有着广泛的应用。

在计算机操作系统的进程调度策略中,就有一种称为先来先服务(First Come First Service,FCFS)的策略,就绪进程组织为一个队列,而操作系统总是把当前处于就绪队列队头的进程调度到运行状态。

在计算机网络中,当一台路由器收到一个 IP 数据报时,它总是根据 IP 数据报头中的目的 IP 地址和自身的路由表,将 IP 数据报放入对应接口的转发队列中,排队等待转发到下一台路由器。通过这样的分段接力式的存储转发机制,一个 IP 数据报就能够从源主机发送到目的主机。

进程之间可以利用消息队列跨平台地进行消息传递,进行分布式系统的集成。发送者进程将要传递给其他进程的消息放入消息队列,而接收者则从消息队列接收消息。

为了缓和 CPU 和 I/O 设备速度不匹配的矛盾,提高 CPU 和 I/O 设备之间的并行性,绝大多数的 I/O 设备在与 CPU 进行数据交换时,都须通过缓冲区来实现,而缓冲区的组织形式就是队列。键盘输入有缓冲区,文件访问也有缓冲区,打印输出和网络通信等也都有对应的缓冲区。

队列的应用实例还有很多,在此不一一列举。

小结

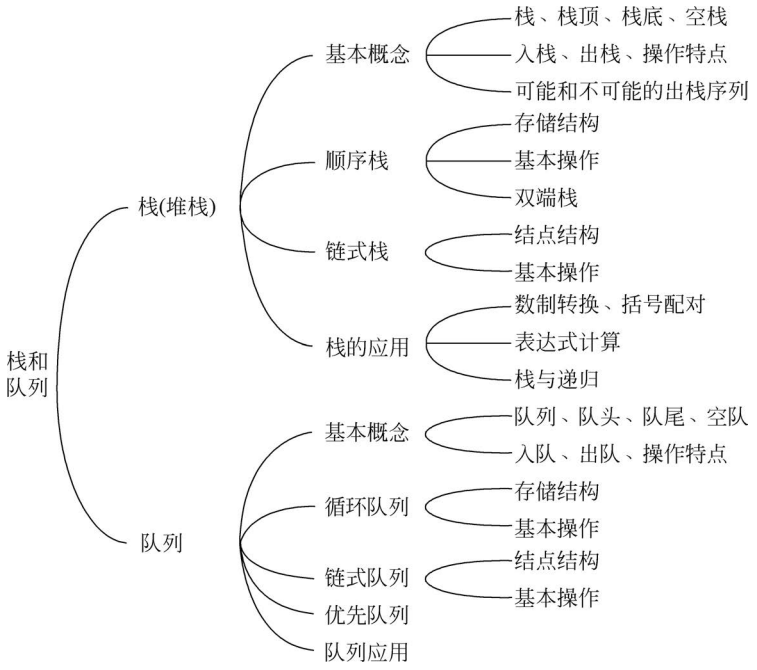
本章的知识点归纳总结如下。

本章中需要重点掌握的内容有:

(1) 栈和队列的基本概念。

(2) 顺序栈和链式栈的存储结构与基本操作。

(3) 循环队列和链式队列的存储结构与基本操作。



习题 3

1. 设计一个支持 push(入栈)、pop(出栈)、getTop(取栈顶元素)和 getMin(取最小元素)操作的栈,不考虑溢出情况,写出该栈的存储结构定义及算法,要求上述 4 个操作的时间复杂度均为 $O(1)$。

2. 对于入栈序列(0,1,…,$n-1$),设计一个算法,判断序列(list[0],list[1],…,list[$n-1$])(该序列是入栈序列的某一个排列)是否是可能的出栈序列。

3. 写出计算表达式((3+4) * 7−4) * 3 的值时的操作数栈和运算符栈的变化情况。