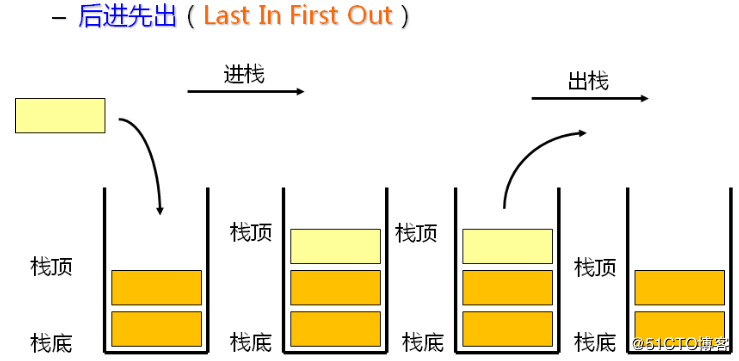
**3.1栈**

**3.1.1栈的类型定义**

* 栈（stack）是限定只能在表的一端进行插入和删除操作的线性表。
* 在表中，允许插入和删除的一端称作“栈顶（top）”，
* 不允许插入和删除的另一端称作“栈底（bottom）”。



* 栈必须按“后进先出”的规则进行操作
* 栈只允许在表尾一端进行插入和删除
* 通常称往栈顶插入元素的操作为“入栈”。
* 称删除栈顶元素的操作为“出栈”，
* 因此后入栈的元素先于先入栈的元素出栈，故被称为是一种“后进先出”的结构，因此又称LIFO（Last In First Out）表。

**类型定义**

ADT Stack {

数据对象：D＝{ai|ai∈ElemSet,i=1,2,...,n,n≥0}  
 数据关系：R1＝{<ai-1,ai>|ai-1,ai∈D,i=2,...,n}  
 约定an端为栈顶，a1端为栈底。  
 基本操作：  
 InitStack(&S)  
 操作结果：构造一个空栈S。  
  
 DestroyStack(&S)  
 初始条件：栈S已存在。  
 操作结果：栈S被销毁。

ClearStack(&S)  
 初始条件：栈S已存在。  
 操作结果：将S清为空栈。

StackEmpty(S)  
 初始条件：栈S已存在。  
 操作结果：若栈S为空栈，则返回TRUE，否则返回FALSE。

StackLength(S)  
 初始条件：栈S已存在。  
 操作结果：返回栈S中元素个数，即栈的长度。

Peer(S,&e)  
 初始条件：栈S已存在且非空。  
 操作结果：用e返回S的栈顶元素。

Push(&S,e)  
 初始条件：栈S已存在。  
 操作结果：插入元素e为新的栈顶元素。

Pop(&S,&e)  
 初始条件：栈S已存在且非空。  
 操作结果：删除S的栈顶元素，并用e返回其值。

StackTraverse(S,visit())  
 初始条件：栈S已存在且非空，visit()为元素的访问函数。  
 操作结果：从栈底到栈顶依次对S的每个元素调用函数visit()，

一旦visit()失败，则操作失败。

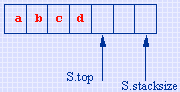
} ADT Stack

**3.1.2栈的存储表示和操作的实现**

**（1）顺序栈类型的定义**

结构定义：

typedef struct{  
 ElemType \*base; //存储空间基址  
 int top; //栈顶指针  
 int stacksize; //允许的最大存储空间以元素为单位  
} Stack;



void InitStack(Stack&S，int maxsize)  
{  
 //构造一个最大存储容量为maxsize的空栈S  
 if(maxsize==0)  
 maxsize=MAXLISTSIZE;  
 S.base=new SElemType[maxsize];  
 if(!S.base)exit(1); //存储分配失败  
 S.stacksize=maxsize;  
 S.top=0; //空栈中元素个数为0  
}  
bool GetTop(Stack S,ElemType&e)  
{  
 //若栈不空，则用e返回S的栈顶元素，并返回TRUE；否则返回FALSE  
 if(S.top==0) return FALSE;  
 e=\*(S.base+S.top-1); //返回非空栈中栈顶元素  
 return TRUE;  
}

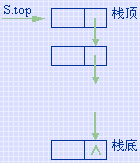
bool Push(Stack&S,ElemType e)  
{  
 //若栈的存储空间不满，则插入元素e为新的栈顶元素，并返回TRUE；否则返回FALSE  
 if(S.top==S.stacksize) //栈已满，无法进行插入  
 return FALSE;  
 \*(S.base+S.top)=e; //插入新的元素  
 ++S.top; //栈顶指针后移  
 return TRUE;  
}

bool Pop(Stack&S,ElemType&e)  
{  
 //若栈不空，则删除S的栈顶元素，用e返回其值，并返回TRUE；否则返回FALSE  
 if(S.top==0) return FALSE;  
 e=\*(S.base+S.top-1); //返回非空栈中栈顶元素  
 --S.top; //栈顶指针前移  
 return TRUE;  
}

**（2）链栈**

结构定义：

typedef struct {  
 SLink top; //栈顶指针  
 int length; //栈中元素个数  
} Stack;



void InitStack(Stack&S)  
{  
 //构造一个空栈S  
 S.top=NULL; //设栈顶指针的初值为"空"  
 S.length=0; //空栈中元素个数为0  
} //InitStack

void Push(Stack&S,ElemType e)  
{  
 //在栈顶之上插入元素e为新的栈顶元素  
 p=new LNode; //建新的结点  
 if(!p) exit(1); //存储分配失败  
 p->data=e;  
 p->next=S.top; //链接到原来的栈顶  
 S.top=p; //移动栈顶指针  
 ++S.length; //栈的长度增1  
}

bool Pop(Stack&S,SElemType&e)  
{  
 //若栈不空，则删除S的栈顶元素，用e返回其值，并返回TRUE；否则返回FALSE  
 if(!S.top)  
 return FALSE;  
 else {  
 e=S.top->data; //返回栈顶元素  
 q=S.top;  
 S.top=S.top->next; //修改栈顶指针  
 --S.length; //栈的长度减1  
 delete q; //释放被删除的结点空间  
 return TRUE;  
 }  
}

**3.1.3栈的应用举例**

例1：数制转换

（1348）10 = （2504）8

void conversion()  
{  
 //对于输入的任意一个非负十进制整数，打印输出与其等值的八进制数  
 InitStack(S); //构造空栈  
 cin>>N; //输入一个十进制数

while(N) {  
 Push(S,N%8); //“余数”入栈  
 N=N/8; //非零“商”继续运算  
 }  
 while(!StackEmpty) { //和"求余"所得相逆的顺序输出八进制的各位数  
 Pop(S,e);  
 cout<<e;  
 }  
}

例2：括弧匹配检验

考虑下列括号序列

[ ( [ ] [ ] ) ]

1 2 3 4 5 6 7 8

例3：迷宫求解问题

* 通常用的是“穷举求解”的方法，即从入口出发，顺某一方向向前探索，若能走通，则继续往前走；否则沿原路退回，换一个方向再继续探索，直至所有可能的通路都探索到为止。
* 如果所有可能的通路都试探过，还是不能走到终点，那就说明该迷宫不存在从起点到终点的通道。
* 所谓“下一位置”指的是“当前位置”四周四个方向（东、南、西、北）上相邻的方块。
* 假设以栈S记录“当前路径”，则栈顶中存放的是“当前路径上最后一个通道块”。
* “纳入路径”的操作即为“当前位置入栈”；
* “从当前路径上删除前一通道块”的操作即为“出栈”。

例3：表达式求值问题

* Exp=S1+OP+S2
* 称OP+S1+S2为表达式的前缀表示法（简称前缀式）
* 称S1+OP+S2为表达式的中缀表示法（简称中缀式）
* 称S1+S2+OP为表达式的后缀表示法（简称后缀式）



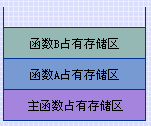
后缀式的运算规则为：  
运算符在式中出现的顺序恰为表达式的运算顺序；  
每个运算符和在它之前出现且紧靠它的两个操作数构成一个最小表达式；

"先找运算符，后找操作数。"

对后缀式从左向右“扫描”，遇见操作数则暂时保存，遇见运算符即可进行运算；此时参加运算的两个操作数应该是在它之前刚刚碰到的两个操作数，并且先出现的是第一操作数，后出现的是第二操作数。

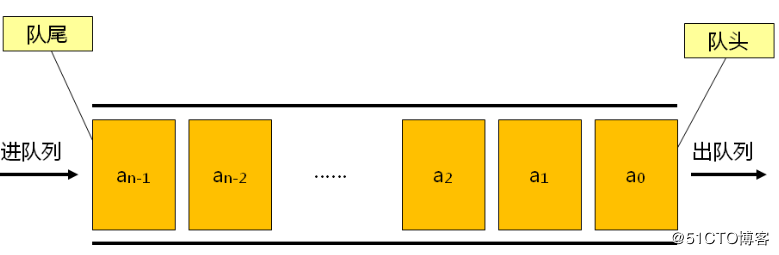
例4：递归函数的实现

* 当多个函数嵌套调用时，由于函数的运行规则是：后调用先返回，因此各函数占有的存储管理应实行“栈式管理”。
* 假设主函数调用函数A，函数A又调用函数B
* 一个递归函数的运行过程类似于多个函数的嵌套调用，差别仅在于"调用函数和被调用函数是同一个函数"。



**3.3队列**

* 队列（Queue）是限定只能在表的一端进行插入和在另一端进行删除操作的线性表。
* 在表中，允许插入的一端称作“队列尾（tail）”。
* 允许删除的另一端称作“队列头（front）”。
* 队列又称FIFO（First In First Out）表。



**3.3.1类型定义**

ADT Queue{

数据对象：D＝{ai|ai∈ElemSet,i=1,2,...,n,n≥0}

数据关系：R1＝{<ai-1,ai>|ai-1,ai∈D,i=2,...,n}

//约定其中a1端为队列头，an端为队列尾。

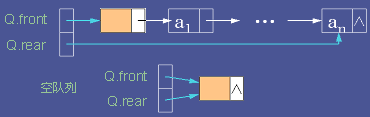
基本操作：  
 InitQueue(&Q)  
 操作结果：构造一个空队列Q。  
 DestroyQueue(&Q)  
 初始条件：队列Q已存在。  
 操作结果：队列Q被销毁，不再存在。

ClearQueue(&Q)  
 初始条件：队列Q已存在。  
 操作结果：将Q清为空队列。  
 QueueEmpty(Q)  
 初始条件：队列Q已存在。  
 操作结果：若Q为空队列，则返回TRUE，否则返回FALSE。  
 QueueLength(Q)  
 初始条件：队列Q已存在。  
 操作结果：返回Q的元素个数，即队列的长度。  
 GetHead(Q,&e)  
 初始条件：Q为非空队列。  
 操作结果：用e返回Q的队头元素。

EnQueue(&Q,e)  
 初始条件：队列Q已存在。  
 操作结果：插入元素e为Q的新的队尾元素。  
 DeQueue(&Q,&e)  
 初始条件：Q为非空队列。  
 操作结果：删除Q的队头元素，并用e返回其值。  
 QueueTraverse(Q,visit())  
 初始条件：队列Q已存在且非空，visit()为元素的访问函数。  
 操作结果：依次对Q的每个元素调用函数visit()，一旦visit()失败则操作失败。  
} ADT Queue

**3.3.2队列的存储表示和操作的实现**

**（1）链队列**



1）结构定义：

typedef SLink QueuePtr; //链队列的结点结构和单链表相同  
typedef struct{  
 QueuePtr front; //队列的头指针  
 QueuePtr rear; //队列的尾指针  
 int length; //队列元素个数  
}Queue; //链队列

2）构造一个空队列Q

void InitQueue(Queue&Q)  
{  
 Q.front=Q.rear=new LNode;  
 if(!Q.front)exit(1); //存储分配失败  
 Q.front->next=NULL;  
 Q.length=0;  
}  
 3）在当前队列的尾元素之后，插入元素e为新的队列尾元素

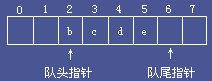
void EnQueue(Queue&Q,ElemType e)  
{  
 s=new LNode;  
 if(!s) exit(1); //存储分配失败  
 s->data=e; s->next=NULL;  
 Q.rear->next=s; //修改尾结点的指针  
 Q.rear=s; //移动队尾指针  
 ++Q.length; //队列长度增1  
}

4）若队列不空，则删除当前队列Q中的头元素，用e返回其值

bool DeQueue(Queue&Q,ElemType&e)  
{  
 if(Q.front==Q.rear) //链队列中只有一个头结点  
 return FALSE;  
 p=Q.front->next;  
 e=p->data; //返回被删元素  
 Q.front->next=p->next; //修改头结点指针  
 if(Q.rear==p)Q.rear=Q.front;  
 delete p; //释放被删结点  
 return TRUE;  
} //DeQueue

**（2）循环队列**

* 利用顺序分配存储结构实现队列
* 设立两个指针front和rear分别指示“队头”和“队尾”的位置
* 空队列是，令front=rear=0
* 头指针始终指向队头元素，而尾指针指向队尾元素的“下一个”位置



（1）结构定义

typedef struct{  
 ElemType\*elem; //存储空间基址  
 int rear; //队尾指针  
 int front; //队头指针  
 int queuesize; //允许的最大存储空间，以元素为单位  
} Queue;

循环队列的初始化需要添加一个"最大容量"的参数

（2）构造一个最大存储空间为maxsize的空队列

void InitQueue(Queue&Q，int maxsize)  
{  
 if(maxsize==0)  
 maxsize=MAXLISTSIZE;  
 Q.elem=new ElemType[maxsize]; //为循环队列分配存储空间  
 if(!Q.elem) exit(1); //存储分配失败  
 Q.queuesize=maxsize;  
 Q.front=Q.rear=0;  
} //InitQueue

（3）若队列不空，则删除当前队列Q中的头元素，用e返回其值

bool DeQueue(Queue&Q,ElemType&e)  
{  
 if(Q.front==Q.rear)  
 return FALSE;  
 e=Q.elem[Q.front];  
 Q.front=(Q.front+1)%Q.queuesize;  
 return TRUE;  
}

队列变空的条件应该是"两个指针指向循环队列中的同一位置"。

1. 若当前队列不满，这在当前队列的尾元素之后

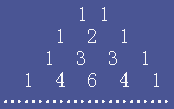
bool EnQueue(Queue&Q,ElemType e)  
{  
 if((Q.rear+1)%Q.queuesize==Q.front)  
 return FALSE;  
 Q.elem[Q.rear]=e;  
 Q.rear=(Q.rear+1)%Q.queuesize;  
 return TRUE;  
}

（5）返回队列Q中元素个数，即队列的长度

int QueueLength(Queue Q)  
{  
 return((Q.rear-Q.front+Q.queuesize)%Q.queuesize);  
}

**3.3.3队列应用举例**

* 编写一个打印二项式系数表（即杨辉三角）的算法。
* 假设队列中已存有第 k 行的计算结果，并为了计算方便，在两行之间添加一个"0"作为行界值，则在计算第 k+1 行之前，头指针正指向第 k 行的"0"，而尾元素为第 k+1 行的"0" 。



**练习**

* 双向栈
* 调度战入口有n节软席和硬席车厢,将所有软席调到硬席之前
* 识别以@结尾的字符串是否满足”序列1&序列2”,其中序列1,2中不含&,且序列2是序列1的逆序列.
* 判别表达式中三种括号是否匹配
* 编写如下递归函数的递归算法

0 m=0,n≥0

g(m,n)=

g(m-1,2n)+n m>0,n≥0

* 假设以带头结点的循环链表表示队列,并且只设一个指针指向队尾元素结点,编写对应的操作.
* 设循环队列,以rear和length分别表示队尾位置和队列长度,给出队列的基本操作.
* 判别输入的字符串是否回文序列,是则返回1,否则返回0,(以@结尾)
* 这里用字符串train表示火车,‘P’表示硬座,‘H’表示硬卧,‘S’表示软卧,最终按PSH的顺序排列(用输出受限的双端队列实现,E表示头端入队,D表示头端出队,A表示尾端入队)