第3章

知识要点

(1) 熟悉操作受限线性表中栈、队列与线性表的关系，顺序栈、循环队列与顺序表的关系，链栈、链队列与链表的关系。

(2) 重点掌握栈与队列的结构特点，了解分别在什么问题中应该使用栈结构与队列结构。

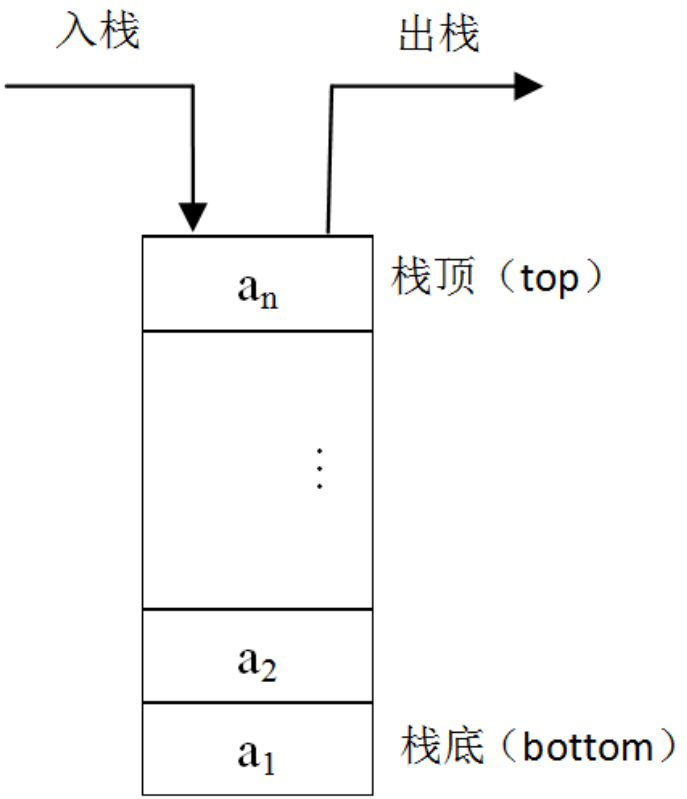
(3) 熟练掌握栈的基本操作在顺序栈和链栈上的实现，特别注意顺序栈和链栈的判空条件；熟练掌握队列的基本操作在循环队列和链队列上的实现，特别注意循环队列和链队列的判空条件。

(4) 熟悉栈与队列的上溢和下溢的概念，了解顺序队列中产生假上溢的原因，以及循环队列消除假上溢的方法。

(5) 了解数据类型受限的字符串结构，初步掌握字符串匹配算法。

3.1 操作受限线性表---栈

- **栈（stack），**又称为堆栈，是一种被限定仅在表尾进行插入和删除操作的线性表。
- 对于栈来说，能进行插入和删除操作的一端称为栈顶（top），另一端称为栈底（bottom）。
- 栈顶的位置是不断变化的，需要使用一个变量将当前的栈顶位置记录下来，该变量称为“**栈顶指针**”。
- 栈底的位置是不会变的。
- 在栈顶位置进行的插入操作称为**入栈**操作，而在栈顶位置进行的删除操作称为**出栈**操作。
- 若栈内不存在任何数据元素，则称为**空栈**。



- 设定栈S = (a_1 , a_2 ,…, a_n);
- 栈具有“**先进后出，后进先出**”的特点；

  FILO(First In Last Out)

算法3.1：栈的抽象数据类型

ADT Stack{

数据对象：D={ a_i| a_i ∈ElemSet,i=1,2,3,…,n,n≥0}

数据关系：R={ <a_{i-1},a_i,>| a_{i-1},a_i∈D,i=1,2,3,…,n} //a_1为栈底元素，a_n为栈顶元素

基本操作：

```
    StackInit( &S );          //构造一个空栈S
    IsEmpty( S );             //判断S是否为空栈，若是，则返回1；否则返回0
    IsFull( S );              //判断栈S是否为满栈，若满，则返回1，否则返回0
    StackClear( &S );         //将栈S清空
    StackDestroy( &S );       //销毁栈S
    StackLength( S );         //求栈S的长度，即栈S中的元素的个数
    GetTop( S );              //返回栈S的栈顶元素
    Push( &S, e );            //入栈操作，将数据元素e插入栈S的当前栈顶
    Pop( &S );                //出栈操作，删除栈S的栈顶元素，并返回其值
```

}ADT Stack

3.2 栈的存储结构

3.2.1 顺序栈的定义及实现

- 栈的顺序存储结构称为**顺序栈**（sequence stack）。
- 顺序栈也是用一组地址连续的存储空间依次存放从栈底到栈顶的所有数据元素，再定义一个指针变量来指示栈底的位置，一个指针变量来指示当前的栈顶元素的位置。
- **栈的顺序存储结构**是用一个一维数组依次存放从栈底到栈顶的所有元素，从而实现顺序栈的存储。

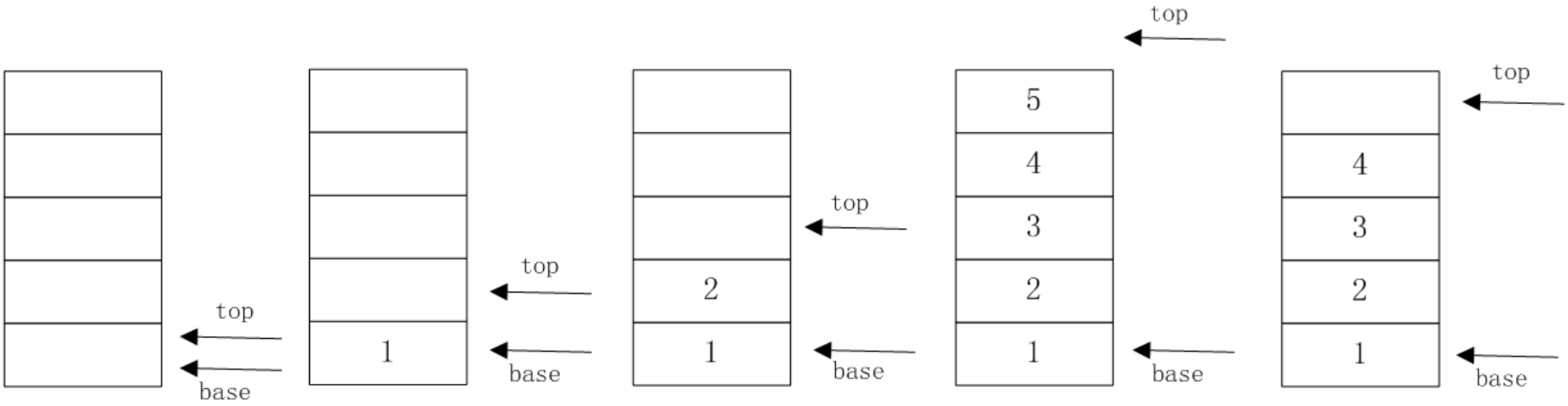


图3-2 顺序栈的示意图

- 由于栈中的元素个数动态变化，而数组的空间是固定的，所以当使用一个数组来存放数据元素时，可能会出现**溢出的现象**。
- 当数组为空时，再对数组进行出栈操作会出现溢出现象，这种溢出称为**下溢**。
- 若数组已满时，再对数组进行入栈操作也会出现溢出现象，这种溢出称为**上溢**。
- 因此，在对栈进行入栈或出栈操作前，应该**对栈当前的状态进行检查**。

算法3.2：顺序栈的数据类型定义

```
template<class Type>
class SeqStack{
private:
    Type *base;              //栈底指针
    int maxsize;             //顺序栈的最大空间
    int top;                 //栈顶指针
public:
   SeqStack(int size);       //构造函数,构造一个数组空间大小为size的空栈
   ~SeqStack();              //析构函数，销毁栈，释放栈空间
   int IsEmpty()const;       //判断栈是否为空栈，若是，返回1；否则返回0
   int IsFull();             //判断栈是否是满栈，若是，返回1；否则返回0
   void SeqStackClear();     //将栈清空
   int SeqStackLength();     //栈的长度，即栈中元素的个数
   void Push(Type e);        //入栈操作，将元素e插入栈顶
   Type &Pop();              //出栈操作，将当前栈顶元素删除，并返回其值
   Type &GetTop();           //返回栈顶元素
}
```

顺序栈基本操作

（1）顺序栈的创建操作：使用函数new()

顺序栈的构造函数产生了一个空间大小为size的空栈

算法3.3：顺序栈的构造函数

```
//构造容量大小为size的空栈
SeqStack<Type>::SeqStack(int size):maxsize(size){
    base = new Type[maxsize];
    if(base == NULL)   { cout<<"分配内存失败" ; exit(0);}
    top = 0;
}
```

（2）元素的入栈和出栈操作：函数Push()

算法3.4：元素的入栈

```
void SeqStack<Type>::Push(Type e){
    if( IsFull() ){
       cout<<"栈空间已满"<<endl;
       retrun;
    } else {
       base[top++] = e;
    }
}
```

- 判断栈当前状态为空或已满，使用函数IsEmpty()和IsFull()，原理都是通过判断栈顶指针指向的位置来实现。
- 若栈已满，则
  top-base = maxsize;
- 若栈为空，则
  top = base;

- **顺序栈的出栈：函数Pop()**

算法3.5：元素出栈

```
//将当前栈顶元素出栈
Type SeqStack<Type>::Pop(){
    if( IsEmpty())
    { cout<<"栈为空"<<endl; retrun -1; }
    else return base[--top];
}
```

（3）返回当前栈顶元素值的操作：函数GetTop()

- 需要判断栈的当前状态，在栈非空的状态下，输出当前栈顶元素的值。但是，在输出元素值后，不需要改变栈顶指针。
- 注意，栈顶指针指向的是当前栈顶指针的下一个元素空间。

算法3.6：返回当前栈顶元素

```
Type SeqStack<Type>::GetTop(){
    if( IsEmpty())
      { cout<<"栈为空"<<endl; return -1; }
    else  return base[top-1];
}
```

- 顺序栈必须预先分配存储空间，在应用中也必须考虑溢出的问题；
- 当同时使用多个栈的时候，若为每个栈分配较大的空间，可能有些栈已经出现溢出的现象，而有些栈还有很多的未用空间；
- 这个问题可以通过多个栈共享一个存储空间的方式来缓解。

只有当整个空间被两个栈全部占满（即两个栈顶相遇）时，才发生上溢现象。因此两个栈共用一个存储空间可以有效地防止空间的浪费和降低上溢发生的概率。

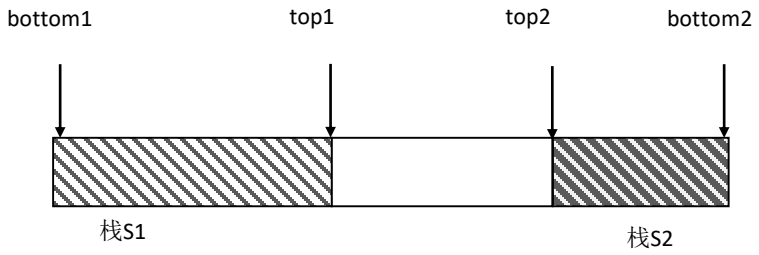
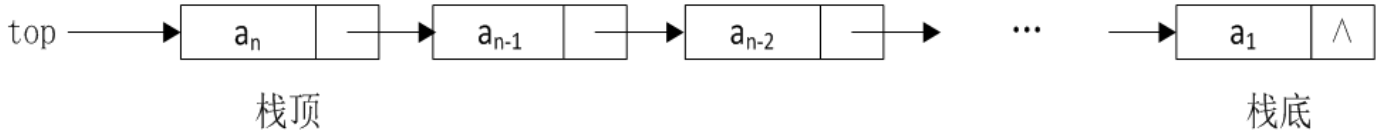


图3-3 两个栈共享存储空间示意图

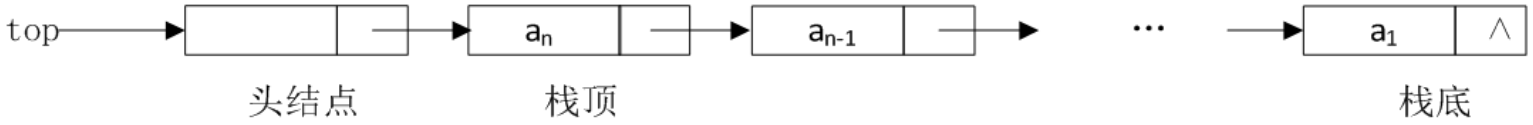
3.2.2 链栈的定义及实现

- 为了克服因为顺序栈预先分配空间可能产生的溢出和空间浪费的问题，可以使用链式存储结构来存储栈。
- **栈的链式存储结构又被称为链栈**(linked stack)，它将栈的所有数据元素依次保存在一个链表内。
- 链栈的结构与单链表的结构相似，但是它的操作是受限的。
- 若将单链表的插入和删除操作限定在表头进行，则它就变成了一个链栈，此时，单链表的头指针head就是对应栈的栈顶指针top。

- 链栈可以有头结点，也可以没有头结点。
- 若是带头结点的链栈，栈顶指针始终存放头结点的地址，而栈顶元素则存放于头结点之后的第一个节点内。
- 若是不带头结点的链栈，栈顶指针始终存放栈顶元素的地址。



（a）不带头结点的链栈



（b）带头结点的链栈

图 3-4 链栈的示意图

- 不带头结点的链栈在插入数据元素时直接将新节点插入到头指针的位置，删除数据元素时直接删除头指针指向的节点，操作方便且比较直观，**默认链栈不带头结点**。
- 链栈的数据类型定义如下：

算法3.7：链栈的数据类型定义

```
template<class Type> class LinkStack;          //链栈类的前视说明
template<class Type> class LinkStackNode{   //结点类的定义
    friend class LinkStack<Type>;          //将链栈类说明为结点类的友元
private:
    Type elem;                                 //结点类的数据域
    LinkStackNode<Type> *next;          //结点类的指针域
public:
    LinkStackNode(Type &e,LinkStackNode<Type> *p= NULL):elem (e), next(p) {};
};
```

```
template<class Type> class LinkStack{       //链栈类的定义
private:
    LinkStackNode<Type> *top;        //定义栈顶指针
public:
    LinkStack():top(NULL) {};        //构造函数，构造了一个空的链栈
    ~LinkStack();                    //析构函数，销毁链栈
    int IsEmpty() const  { return top == NULL; } //判断链栈是否为空
    void LinkStackClear();           //清空链栈
    int LinkStackLength() const ;    //求栈的长度
    Type GetTop();                   //返回链栈栈顶元素的值
    void Push(Type e);               //入栈操作，将元素e加入栈顶
    Type Pop();      //出栈操作，删除链栈的栈顶元素，并返回其值
};
```

（1）链栈的销毁操作

- 由于链栈是由多个结点连接而成，在销毁链栈时，需要依次释放从栈顶到栈底所有结点，最后栈顶指针指向空。

算法3.8：析构函数，销毁链栈

```
LinkStack<Type>::~LinkStack(){
        LinkStackNode<Type> *p;
        while(top!=NULL){
                p = top;
                top = top->next;
                delete p;
        }
}
```

（2）求链栈长度的操作：函数LinkStackLength()

- 返回链栈中数据元素的个数。与顺序栈不同，链栈是由多个结点连接而成，所以需要使用一个指针p从栈顶访问到栈底。

算法:3.9：求链栈的长度，即链栈中数据元素的个数

```
int LinkStack<Type>::LinkStackLength()const{
   LinkStackNode<Type> *p = top;
    int i = 0;          //初始化计数器
    while(p) { i++; p = p->next;  }
    return i;
}
```

（3）返回栈顶元素的操作：函数GetTop()

- 返回栈顶元素的值。
- 在有头结点的情况下，空的链栈仍有一个头结点，空栈的条件是头结点的指针域存放NULL。
- 若链栈没有头结点，则头指针直接指向栈顶元素，这时空栈的条件为栈顶指针指为空。

算法:3.10：返回栈顶元素的值

```
Type LinkStack<Type>::GetTop(){
    if(IsEmpty()) {  cout<<"链栈为空！ "<<endl; return -1; }
    else return top->elem;
}
```

（4）链栈的入栈和出栈操作

- 在没有头结点的链栈中，头指针直接指向栈顶节点。Push()函数实现了链栈的入栈操作

算法3.11：入栈操作，将元素e加入链栈栈顶

```
void LinkStack<Type>::Push(Type e){
    LinkStackNode<Type> *p;
    p = new LinkStackNode<Type>(e, top);
    if(p == NULL) { cout<<"分配失败！ "<<endl; return; }
    top=p;
}
```

- Pop()函数实现了链栈的出栈操作。用一个变量记录已出栈结点数据域的值，释放已出栈结点占用的存储空间，并返回记录的数据值。

算法3.12：出栈操作，删除栈顶元素，并返回其值

```
Type LinkStack<Type>::Pop(){
  if(IsEmpty())   { cout<<"链栈为空！"<<endl; return -1; }
  LinkStackNode<Type> *p = top;
  top = top->next;
  Type e = p->elem;
  delete p;
  return e;
}
```

3.3 栈的应用

3.3.1 括号匹配检验

括号匹配，是指在一个表达式中的左右括号不仅要个数相等，而且必须类型相同、先左后右的出现，以此来界定一个范围的起始和结束位置。

实现算术表达式中括号匹配检验可以使用一个栈来保存一个或多个嵌套的左括号，如果遇到右括号，则将栈顶的左括号弹出栈，并检查其是否与该右括号匹配，若匹配，接收下一个括号，若不匹配，程序报错。若算式表达式处理结束时，栈为空，则表达式中括号是匹配的；否则括号匹配出错。具体算法实现如下：

算法3.13：实现算术表达式中括号匹配检验

```
#include <iostream>
#include <cstring>
#include "LinkStack.cpp"
using namespace std;
int main(){
    LinkStack<char>  L;   //定义链栈对象
    string s;
    char left[3]={'(', '[', '{' };
    char right[3]={')', ']', '}' };
    cout<<"请输入表达式：";
    cin>>s;
    int n = s.size();
```

```
    for(int i=0;i<n;i++) {
            if(s[i] == '(' || s[i] == '[' || s[i] == '{' )   //遇到左括号入栈
            L.Push(s[i]);
            else if(s[i]==')' s[i] == ']' || s[i] == '}') {  //遇到右圆括号，判定是否匹配
                        for (int k=0; right[k]!=s[i]; k++);
                        if(!L.IsEmpty() && left[k]==L.GetTop())  L.Pop(); //左右括号匹配成功
                        else break;   //括号匹配出错
            }
    }
    if(L.IsEmpty() && i==n)  cout<<"括号匹配成功";
    else if (L.IsEmpty())  cout<<"缺少左括号";
    else if (i==n)  cout<<"缺少右括号";
    else  cout<<"左右括号不匹配";
    return 0;
}
```

3.3.2 栈与递归

递归算法包括递推和回归两部分：

（1）递推：将规模较大的原问题分解为一个或多个规模较小而又类似的原问题的子问题，确定一个或多个不需要分解、可直接求解的最小子问题。

（2）回归：当最小子问题得到解后，回归到原问题的解上。

求阶乘算法是非常典型的阶乘问题：

求n的阶乘可递归地定义为：

$$n! = \begin{cases} 1 & n = 0 \\ n * (n-1)! & n > 0 \end{cases}$$

算法:3.14：n的阶乘

```
class Factorial{
public:
     long Fa(int n);      //声明求阶乘的递归函数
};
long Factorial::Fa(int n)   //定义求阶乘的递归函数
{   int m;
   if (n==0) return 1;    //n=0时，直接返回1，同时也是递归的终止条件
   else m = n*Fa(n-1); //递归算式，实现直接递归调用。标记该地址为a2
   return m;
}
int main()
{   Factorial f;
int n=4;
long fn = f.Fa(n);         //标记该地址为a1
   cout<<n<<"!="<<fn<<endl;
   return 0;
}
```

- 递归过程是借助栈来实现的，这个栈称为递归工作栈。在递归执行过程中，每次调用都在栈顶保存一个工作记录，包括返回地址、参数、局部变量；当调用结束后则从栈顶释放相应的工作记录。
- 程序中使用了两个地址标记a1、a2，它们分别表示主函数调用递归函数和递归函数递归调用的返回地址.

调用层次	调用	参数n	返回地址	m值	退栈时计算结果
↑4	Fa(0)	0	a2	1	↓
↑3	Fa(1)	1	a2	1* Fa(0)	1*1=1 ↓
↑2	Fa(2)	2	a2	2 *Fa(1)	2*1=2 ↓
↑1	Fa(3)	3	a2	3*Fa(2)	3*2=6 ↓
↑0	Fa(4)	4	a1	4 *Fa(3)	4*6=24 返回

- 事实上，计算机系统中函数或过程调用都是借助栈实现的。
- 当有多个函数构成的嵌套调用时，总是遵循“先调用、后返回”的原则，因此调用函数和被调用函数之间的信息传递和控制转移都可以利用栈来实现。
- 每当一个函数被调用时，就为它在栈顶分配一个存储区，存储相应的工作记录；每当一个函数返回时，就要从栈顶释放它所占用的存储区。
- 一般通过以下几个步骤来完成调用函数的返回：

  (1) 从栈中弹出工作记录；
  (2) 将工作记录中的参数值赋给对应的变量；
  (3) 将函数值赋给相应的变量；
  (4) 转移至返回地址。

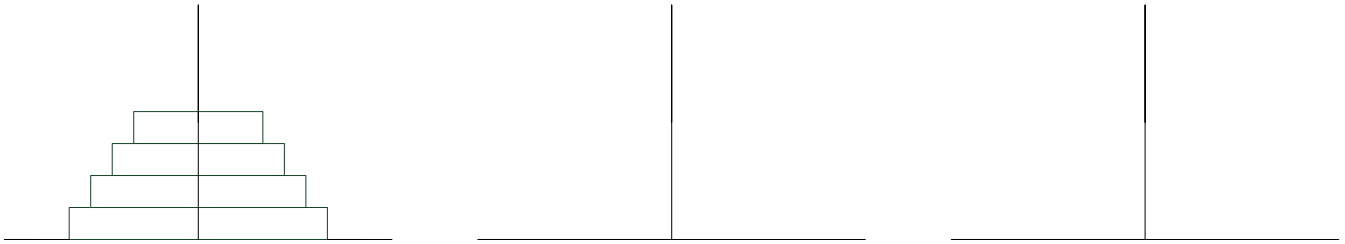
汉诺塔问题

设有三个分别命名为A、B、C的塔座，在塔座A上从上到下插有n个直径由小到大、各不相同的圆盘，编号分别为1、2、3、…、n（如图3-5）。现要求将A塔座上的n个圆盘全部移至塔座B上，并仍按相同的顺序叠放。在移动圆盘时，必须遵循下列规则：

（1）每次只能移动一个圆盘；

（2）圆盘可以插入A、B、C的任一个塔座上；

（3）任何时候都不能将一个较大的圆盘放在一个较小的圆盘上。



这个问题可以用递归的方法考虑。

- 当n=1时，问题可以直接求解，可直接将编号为1的圆盘从塔座A上移至塔座B上；
- 当n>1时，从上述移动过程中可以看出，汉诺塔问题可以分成三个步骤完成：

（1）将塔座A上最上面的n-1个圆盘移至塔座C上；

（2）将圆盘n从塔座A移至塔座B上；

（3）将塔座C上的n-1个圆盘全部移至塔座B上。

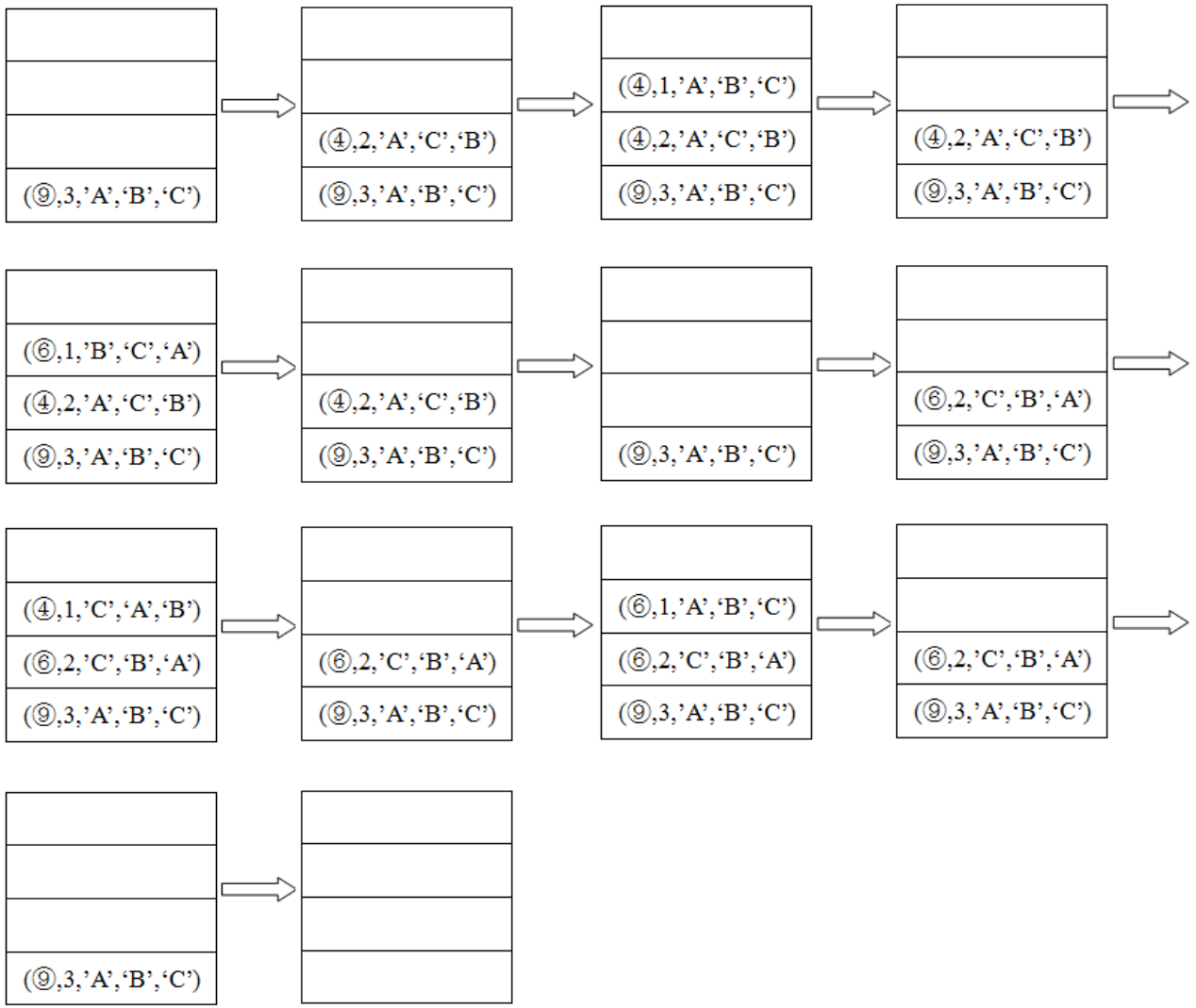
其中（1）和（3）是汉诺塔问题的两个子问题，而（2）可通过一次移动直接完成。

算法3.15：汉诺塔问题的递归算法

```
#include <iostream>
using namespace std;
class Hanoi{          //定义汉诺塔类
public:
  void HanoiTowers(int n, char a, char b, char c);    //声明递归函数
  void Move(int n, char s, char t);  //声明移动函数，将圆盘n从塔座s移至塔座t上
};
void Hanoi::HanoiTowers(int n, char a, char b, char c) {   //定义递归函数
    if(n==1) //定义终止条件
            Move(1,a,b);                    //①
    else {                                  //②
            HanoiTowers(n-1,a,c,b);         //③
            Move(n,a,b);                    //④
            HanoiTowers(n-1,c,b,a);         //⑤
    }                                       //⑥
    return;                                 //⑦
}
```

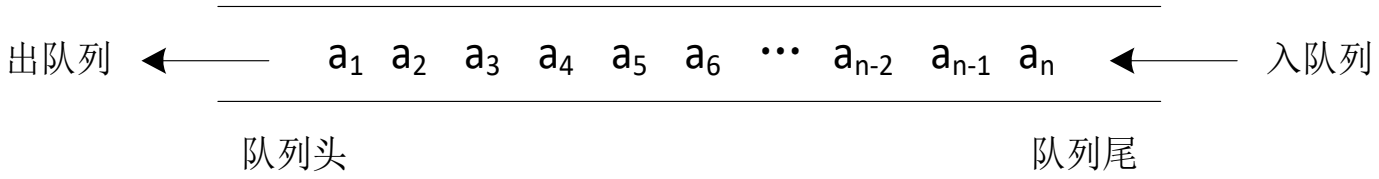
```
void Hanoi::Move(int n, char s, char t){
    cout<<”将圆盘”<<n<<”从塔座”<<s<<”移动至塔座”<<t<<endl;
    return;
}

int main(){
Hanoi h;
int n;
    cout<<"请输入圆盘的个数：";
    cin>>n;
    h.HanoiTowers(n, ’A’, ’B’, ‘C’);          //⑧
    return 0;                                  //⑨
}
```

状态栈的变化过程


(⑨,3,'A','B','C')
(④,2,'A','C','B')
(⑨,3,'A','B','C')
(④,1,'A','B','C')
(④,2,'A','C','B')
(⑨,3,'A','B','C')
(⑥,1,'B','C','A')
(④,2,'A','C','B')
(⑨,3,'A','B','C')
(⑥,2,'C','B','A')
(④,1,'C','A','B')
(⑥,2,'C','B','A')
(⑨,3,'A','B','C')
(⑥,1,'A','B','C')

3.4 操作受限线性表---队列

- 队列是限定只能在一端进行插入操作而在另一端进行删除操作的线性表。
- 可以进行插入操作的一端称为队列尾（rear）
- 可以进行删除操作的一端称为队列头（front）
- 插入操作又称为入队列操作
- 删除操作又称为出队列操作
- 队列性质：“**先进先出**”FIFO(First In First Out)



算法3.16：队列的抽象数据类型定义

ADT Quene{

数据对象：D={ a_i | a_i ∈ElemSet,i=1,2,3,…,n, n≥0}

数据关系：R={ <a_{i-1},a_i,> | a_{i-1},a_i∈D,i=2,3,…,n}　//a_1为队列头元素，a_n为队列尾元素

基本操作：

QueueInit(&Q);　//构造一个空队列Q

IsEmpty(Q);　//判断队列Q是否为空队列，若是，则返回1；否则返回0

IsFull(Q);　//判断队列Q是否为满队列，若满返回1，否则返回0

QueueClear(&Q);　//将队列Q清空

QueueDestroy(&Q);　//销毁队列Q

QueueLength(Q);　//求队列Q的长度，即队列Q中的元素的个数

GetFirst(Q);　//返回队列Q第一个元素的值

InQueue(&Q , e);　//入队列操作，将数据元素e插入队列Q的队列尾

OutQueue(&Q);　//出队列操作，删除队列Q的队列头元素，并返回其值

} ADT Queue

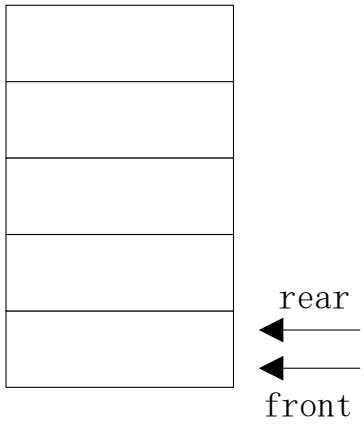
3.5 队列的存储结构及实现

3.5.1 顺序队列的定义及实现

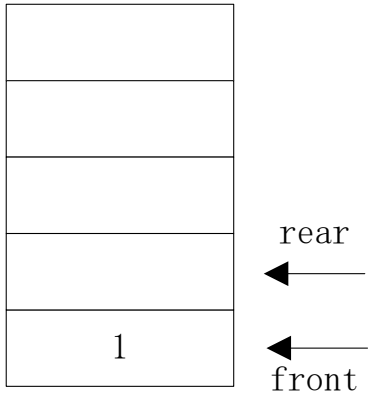
- 队列的顺序存储结构是用一组地址连续的存储空间，依次存放从队列头到队列尾的所有数据元素；
- 使用**队列头指针**（front）和**队列尾指针**（rear）分别记录队列头和队列尾的位置。
- 具有顺序存储结构的队列称为**顺序队列**。

- 在顺序队列中，队列头指针始终指向队列头元素的位置，而队列尾指针指向队列尾元素的下一个元素位置。
- 在顺序队列初始化的时候，通常将队列头指针front和队列尾指针rear初始化为0。
- 一般情况下，当队列头指针和队列尾指针指向同一个位置时，顺序队列是空队列。
- 当一个数据元素入队列时，先将入队列的数据元素添加到rear指向的位置，再对队列尾指针rear进行加1操作，使其后移一个位置，队列头指针front不变；
- 当一个数据元素出队列时，队列头指针front后移一个位置，队列尾指针rear不变。

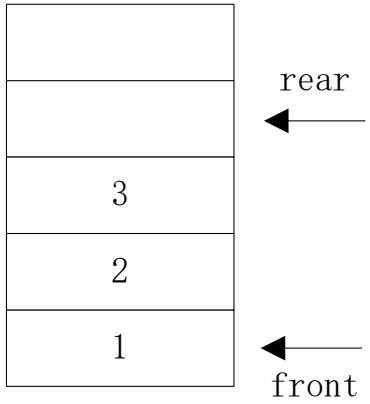
- 在数据元素进/出队列时，顺序队列的队列头指针和队列尾指针的移动情况：



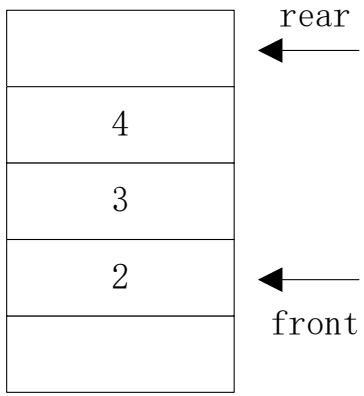
初始化时：
•队头指针
front=0
•队尾指针
rear=0。



入队元素1时：
front=0
rear =1。



2、3入队后：
front=0
rear =3。



4入队，1出队后：
front=1
rear =4。

- 解决顺序队列出现假溢出现象的一种有效途径是采用循环队列。
- **循环队列**就是将存放队列元素的存储空间首尾连接起来，构成一个顺序的环形结构。
- 在循环队列中，0位置于数组MaxSize-1之后，即，当队列头指针或队列尾指针等于MaxSize-1时，再向后移就是0位置。
- 循环队列中指针front和指针rear的循环后移可以通过对指针用MaxSize进行取余运算来实现。

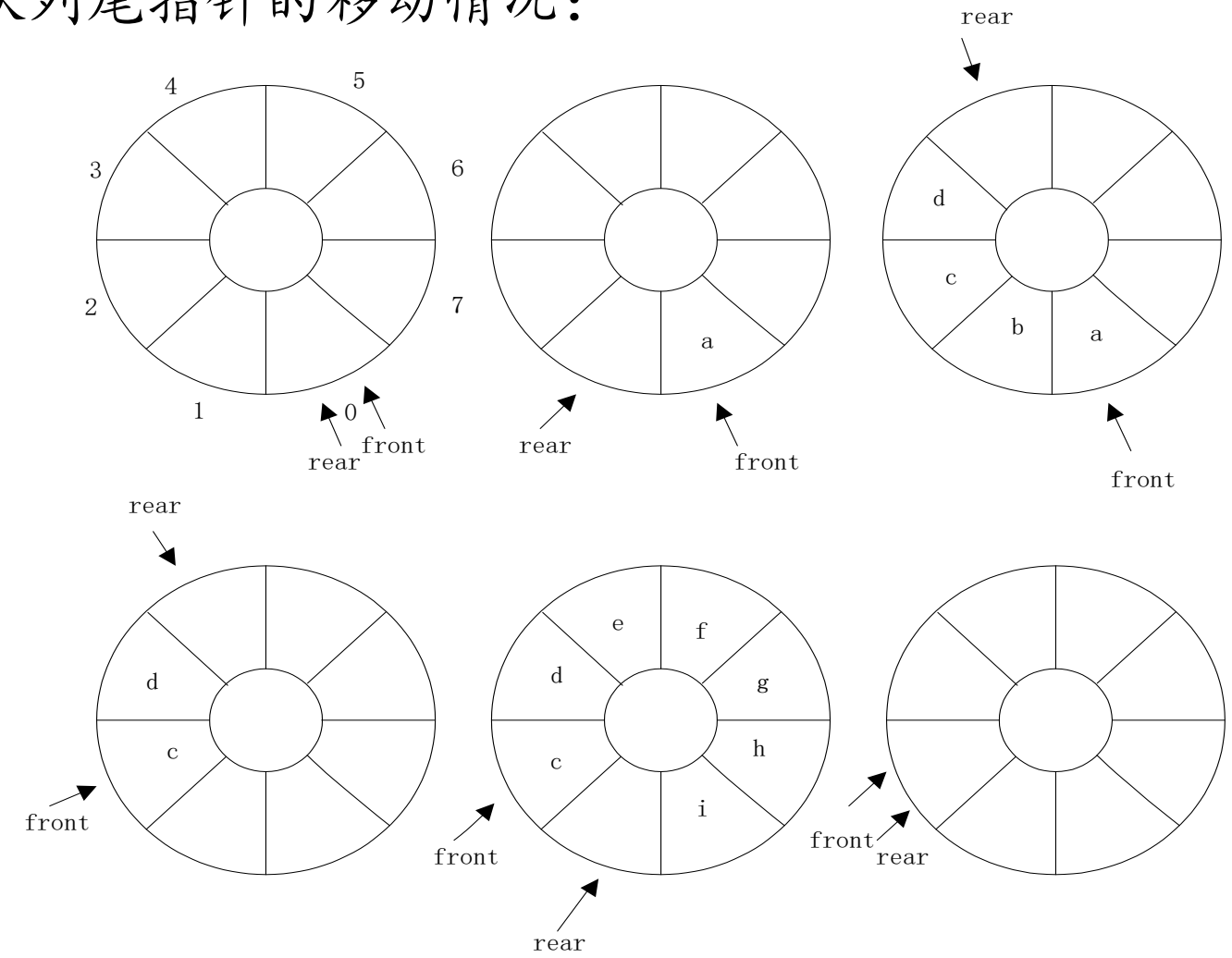
队列头指针front后移一个位置： front = (front + 1) % MaxSize

队列尾指针rear后移一个位置： rear = (rear + 1) % MaxSize

队列为空的判断条件为： front == rear

队列为满的判断条件为： front == (rear + 1) % MaxSize

- 在数据元素进出队列时，循环队列的队列头指针和队列尾指针的移动情况：



循环队列的数据类型定义：

算法3.17：循环队列的数据类型定义

```
template<class Type>
class SeqQueue{
  private:
    int front,rear;          //定义队列头指针、队列尾指针
    Type *queue;             //顺序队列存区指针，用来指向存放队列元素的数组
    int maxsize;             //存区的容量，即最大元素个数
  public:
    SeqQueue(int size);      //构造一个最大存储size个元素的空队列
    ~SeqQueue(){ delete [] elem; } //析构函数，销毁队列，释放队列空间
    int IsEmpty() const { return front == rear; } //判断队列是否为空
    int IsFull() const { return (rear + 1)%maxsize == front; } //判断队列是否是满队列
    void SeqQueueClear(Type &Q){front = rear =0;}        //将队列清空
    void SeqQueueLength() const {return (rear - front + maxsize)%maxsize;}
    void InQueue(Type e);  //入队列操作，将元素e从队列尾插入队列
    Type OutQueue();       //出队列操作，将当前队列头元素删除，并返回其值
    Type GetFront();       //返回队列头元素
}
```

（1）循环队列的创建操作：函数new()

- 用maxsize来记录存储空间大小，同时，将队列头指针front和队列尾指针rear初始化为0。

算法4.18：循环队列的构造函数

```
SeqQueue<Type>::SeqQueue(int size):front(0),rear(0),maxsize(size+1){
  queue = new Type[maxsize];
  if(!queue){cout<<"内存分配失败！"; return;}
}
```

（2）元素的入队列和出队列操作

- 首先要判断队列空间是否已满

算法4.19：入队列操作： 函数InQueue()

```
void SeqQueue<Type>::InQueue (Type e){
    if( IsFull() ){cout<<"队列已满！ "<<endl; return;}
    queue[rear] = e;
    rear = (rear+1)%maxsize;
}
```

- 循环队列的出队列操作:函数OutQueue()

算法:3.20： 出队列操作

```
Type SeqQueue<Type>::OutQueue(){
    if( IsEmpty() ){ cout<<"队列为空！ "<<endl; return -1; }
    Type e=queue[front];
    front = (front+1)%maxsize;
    return e;
}
```

（3）返回当前队列头元素操作:函数GetFront()

- 在返回值之前，需要判断队列的当前状态，在队列非空的状态下，输出当前队列头元素的值。
- 在输出元素值后，不需要改变队列头指针。

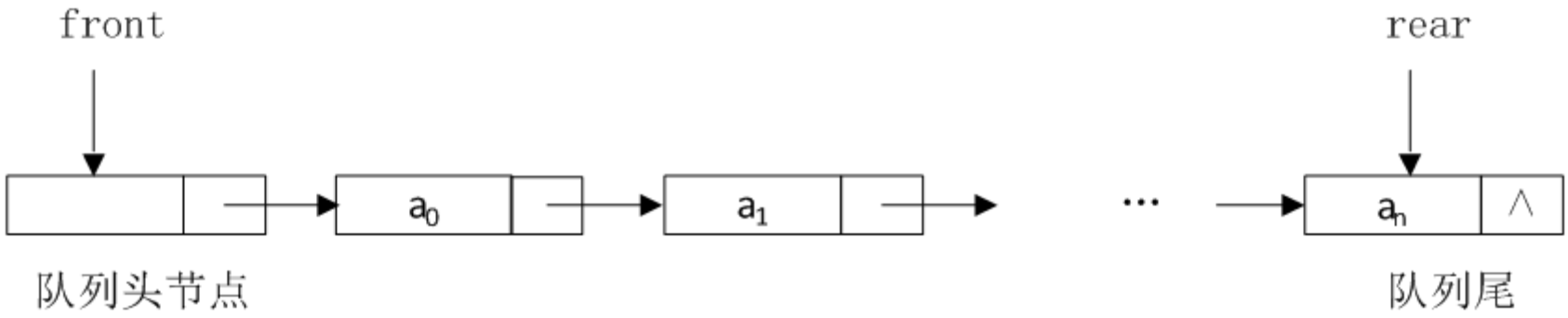
算法3.21：返回当前队列头元素

```
Type SeqQueue<Type>::GetFront(){
    if( IsEmpty() ){cout<<"队列为空！"<<endl; return -1;}
    return queue[front];
}
```

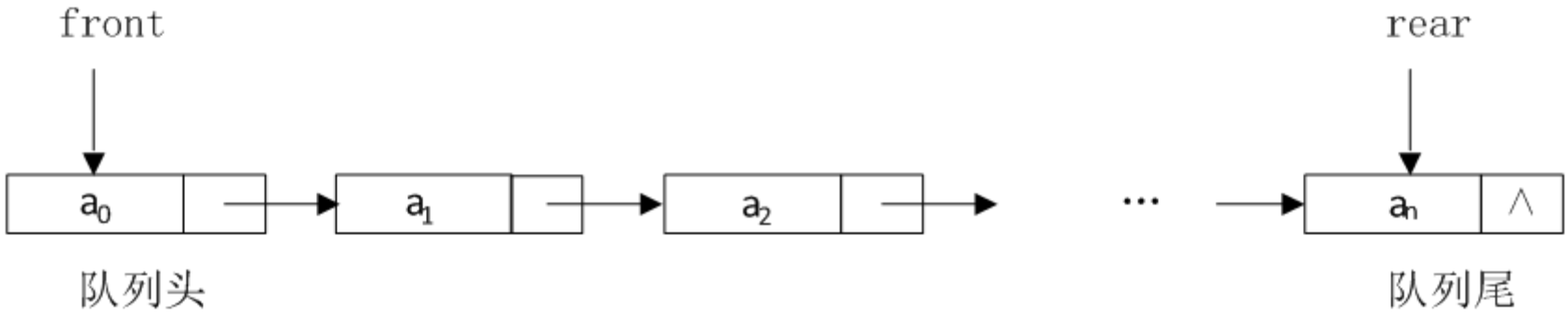
3.5.2 队列的链式存储结构及实现

- 队列的链式存储结构又称为链队列。
- 链队列就是用一个链表来依次存放从队列头到队列尾的所有数据元素。
- 由于在入队列和出队列过程中，队列头或队列尾的位置会发生改变，因此还需要使用两个指针来分别记录队列头和队列尾的当前位置。
- 存放队列头地址的指针称为队列头指针，存放队列尾地址的指针称为队列尾指针。

两种形式的链栈：



（a）带头结点的链队列示意图



（b）不带头结点的链队列示意图

不带头结点的链队列的数据类型定义：

算法3.22：不带头结点的链队列的**结点类型**定义

```
class LinkQueue;                          //链队列类的前视说明
class LinkQueueNode{                      //结点类的定义
    friend class LinkQueue<Type>;         //将链队列类说明为结点类的友元
private:
    Type elem;                            //结点类的数据域
    LinkQueueNode<Type> *next;            //结点类的指针域
public:
    LinkQueueNode(Type &e, LinkQueueNode<Type> *p= NULL):elem
     (e),next(p) {};                      //结点类的构造函数
};
```

```
class LinkQueue{          //定义链队列类
private:
   LinkQueueNode<Type> *front, *rear;      //定义队列头指针和队列尾指针
public:
   LinkQueue():front(NULL), rear(NULL) {};  //构造函数，构造一个空的链队列
   ~LinkQueue();                           //析构函数，销毁链队列
   int IsEmpty()const  { return front == NULL; }  //判断链队列是否为空
   void LinkQueueClear()                   //清空链队列
   int LinkQueueLength()const ;     //求链队列的长度，即链队列中元素的个数
   Type GetFront();                        //返回链队列头元素的值
   void InQueue(Type e);                   //入队列操作，将元素e加入队列顶
   Type OutQueue();   //出队列操作，删除链队列的队列顶元素，并返回其值
};
```

（1）链队列的销毁操作

- 由于链队列是由多个结点连接而成，所以，在销毁链队列时，需要依次释放从队列头到队列尾的所有结点，最后队列头指针指向空。

算法3.23：链队列的析构函数

```
LinkQueue<Type>::~LinkQueue(){   //析构函数，销毁链队列
        LinkQueueNode<Type> *p;
        while(front!=NULL){
                p = front;
                front = front->next;
                delete p;
        }
}
```

（2）求链队列长度的操作： 函数LinkQueueLength()

算法:3.24：求链队列的长度，即链队列中数据元素的个数

```
int LinkQueue<Type>::LinkQueueLength()const{
    LinkQueueNode<Type> *p = front;
    int i = 0;          //初始化计数器
    while(p) {
            i++;
            p = p->next;
    }
    return i;
}
```

（3）返回队列头元素的操作：函数GetFront()

算法:3.25：返回队列头元素的值

```
Type LinkQueue<Type>::GetFront(){
    if(IsEmpty()) {
      cout<<"链队列为空！"<<endl;
      return -1;
   } else return  front->elem;
}
```

（4）链队列的入队列和出队列操作

InQueue()函数实现了链队列的入队列操作。

OutQueue()函数实现了链队列的出队列操作。

算法3.26：入队列操作，将元素e加入链队列

```
void LinkQueue<Type>::InQueue(Type e){
  If (front == NULL)
    front = rear = new LinkQueueNode<Type>(e, NULL);
  else rear = rear->next = new LinkQueueNode<Type>(e, NULL);
}
```

算法3.27：出队列操作，删除队列头元素，并返回其值

```
Type LinkQueue<Type>::OutQueue(){
    if(IsEmpty()) { cout<<"链队列为空！"<<endl; return -1; }
    LinkQueueNode<Type> *p = front;
    Type e = p->elem;
    front = front->next;
    if(front == NULL ) rear = NULL;
    delete p;
    return e;
}
```

3.6 队列的应用

3.6.1 杨辉三角形

1

1 1

1 2 1

1 3 3 1

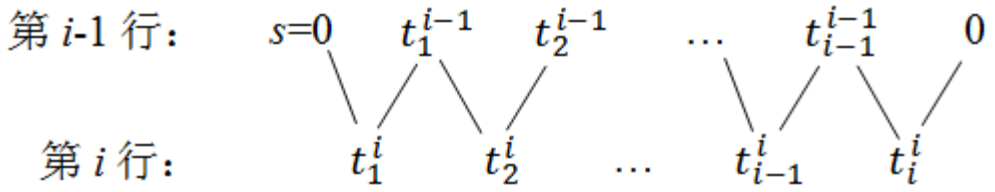
1 4 6 4 1

1 5 10 10 5 1

1 6 15 20 15 6 1

……

- 有个重要的性质是除第一行数字初始化为1之外，杨辉三角形从第二层开始，每个数字都是其左右肩上两个数字之和，如果其左肩或右肩上没有数字，默认其左肩或右肩上的数字为0，而后进行计算。
- 因此，若要求得杨辉三角形的第i行元素，可根据第i-1行的元素从左到右按依次计算。



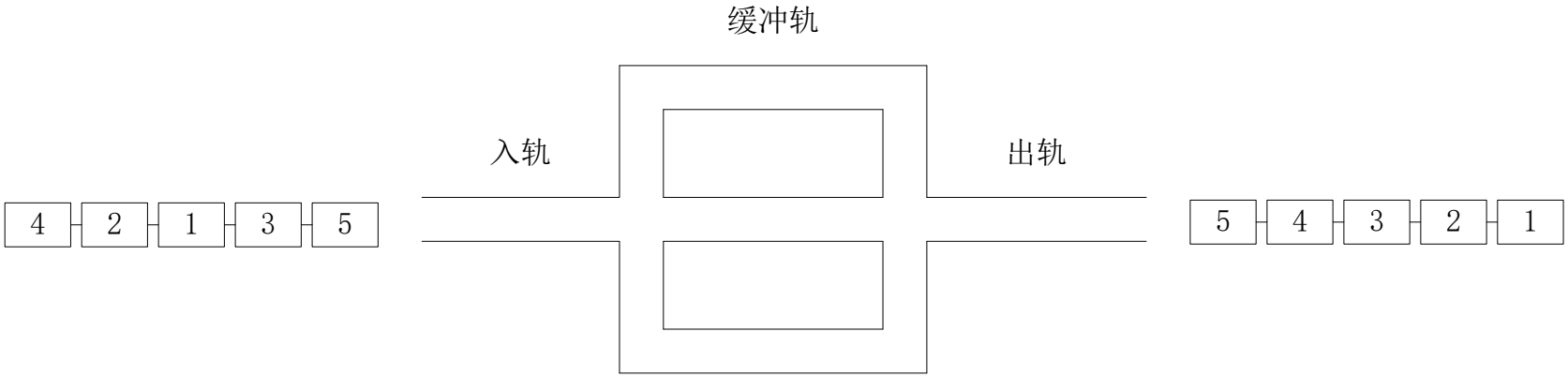
算法3.28：杨辉三角形计算

```
void YANGHUI ( int n ) {
  SeqQueue <int> q=new SeqQueue (n+1);  //循环队列初始化
  cout <<'1'<<endl;              //输出第1行
  q.InQueue (1);                 //将第1行加入循环队列
  int s=0;
  for ( int i=2; i<=n; i++ ) {        //逐行计算并输出
     q.InQueue(0);               //将行分隔符加入循环队列
     for ( int j=1; j<=i; j++ ) {    //输出第i行元素
        int t = q.DeQueue ( );
        s=s+t;
        cout<<s<<' ';
        q.EnQueue ( s );
        s=t;
     }
     cout <<endl;
  }
}
```

3.6.2 火车车厢重排

一列货运火车共n节车厢，每节车厢需要停放在不同的车站。假设一列火车有n节车厢，车厢编号分别为1~n，货运火车按照第n站到第1站的次序经过每个车站。车厢的编号与各个目的车站的编号相同。为了方便从火车上卸载相应的车厢，必须重排火车车厢，即将火车车厢按照一定的编号序列排列，使各个车厢从前到后按照编号从1到n的顺序排列。如果所有车厢按照这种顺序排列，每次只需要卸载火车最后一节车厢即可。

- 火车车厢重排工作安排在转轨站进行。转轨站中有一个入轨、一个出轨和h个缓冲铁轨。假设缓冲铁轨位于入轨和出轨之间，由于这些缓冲铁轨必须按照先进先出的原则运行，所以，可以将其视为队列。火车车厢重排示意图：



将一个编号为c的车厢从入轨移动到缓冲铁轨，需要遵循以下原则：

① 缓冲铁轨上所有的车厢编号都小于c；

② 如果有多个缓冲铁轨都满足条件（1），选择其中车厢编号最大的车厢所在的缓冲铁轨；

③ 若不满足条件（1），如果还有空的缓冲铁轨，则选择空的缓冲铁轨。

在实现火车车厢重排问题时，可以采用链队列。

3.7 类型受限线性表---字符串*

3.7.1 串的定义

字符串简称为串（String），是n(n≥0)个字符的一个有限序列。通常可记为：

$$S = "a_0\ a_1\ a_2\ \ldots\ldots\ a_{n-1}"$$

其中，S是串名，可以是串变量名，也可以是串常量名。用引号‘...’或"..."作为分界符括起来的叫做串值，其内的a_i,是串中的字符（$0 \le i < n$），n是串中的字符个数，也叫做串的长度，它不包括作为分界符的引号，也不包括串结束符‘\0’。长度为零的串叫做空串，除串结束符外，它不包含任何其他字符。

3.7.2 串的操作

s1="It is a car"

s2="jeep"

s3="car"

串的主要操作有：

（1）求串的长度。例如，s1的长度为11，s2的长度为4。

（2）把一个串赋给另一个串。若有s4＝s3，则s4的置为“car"。

（3）把两个串连接成一个新串。设s5为s2和s3连接形成的新串，则s5＝"jeepcar"，即将后面的串连接到前面的串的尾部。

（4）比较两个串的ASCII码值的大小。若str1小于str2，比较结果为-1；若str1等于str2，比较结构为0；若str1大于str2，比较结构为1。

（5）在一个串（称为主串）中查找是否存在与另一个串相等的子串。设在串str1中查找串str2，这里str1是主串，str2是子串。若主串中存在与str2相等的子串，则操作结果为str2在str1中首次出现的位置；若主串str1中不存在和串str2相等的子串，则操作结果为-1。

（6）在一个串中是否存在一个字符。设在串str中查找字符ch，若串str中存在该字符ch，则操作结果为字符ch在串str中首次出现的位置；若str中不存在字符ch，则操作结果为-1。

（7）截取子串形成新串。设str为要截取的串，pos为要截取的起始位置，length为要截取的长度，则形成的新串长度为length。如果s1串截取起始位置为3（这里子串是从0开始）、长度为2的子串放在s6，则s6="is"。

（8）在一个串插入另一个串。设把串str2插入到串str1中，pos为要查入的起始位置，则操作结果形成的新串长度为str1和str2之和。例如，把串s7=”not”，插入到s1的位置5处，则操作结果形成的新串s8="It is not a car"。

（9）从一个串中删除一个子串。设在串str中要删除长度为length的子串，pos为要删除的子串在str中的起始位置，则删除后的新串长度为原长减去length。例如，要在s8中删除长度为4的子串，起始位置为5，则删除后的新串为s1。

3.7.3 串的存储结构

串的存储方式可以有**两种处理方式**：

- 一种是将串定义成**字符型数组**，串的存储空间分配在编译时完成，不能更改，这种方式称为串的静态存储；
- 另一种是串的存储空间在程序运行时动态分配，这种方式称为**动态存储**。

串的静态存储结构即串的顺序存储结构，串的动态存储结构中常见的是链式存储结构。

（1）串的顺序存储结构

- 串的顺序存储结构有时称为**顺序串**。在顺序串中，串中的字符被依次存放在一组连续的存储单元里。一般来说，一个字节（8 位二进制）可以表示一个字符（即该字符的ASCII 码）。
- 串的静态存储结构有一下两个缺点：

①需要预先定义一个串允许的最大字符个数，当该值估计过大时，存储密度就会降低，浪费较多的存储空间；

②由于限定了串的最大字符个数，使串的某些操作，如置换、连接等擦作受到限制。

（2）串的链式存储结构

串的链式存储分为两类:

①结点大小为1的链式存储结构

一个串可以用一个单链表来表示。用单链表存放串时，链表中结点的个数等于串的长度。例如，S="XJTU"采用链式存储结构如图4-14所示。

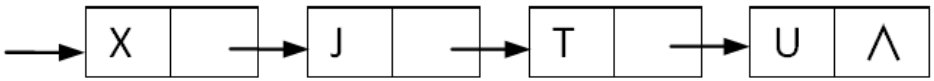


图 4-14 串的链式存储结构

- 链式存储结构的优点是对串的插入、删除操作可以方便地进行。但由于每个结点只存储一个字符，而指针域的存储空间一般要大于字符的存储空间，导致存储密度很低。

②结点大小为K的链式存储结构

采用结点大小为K的链式存储，可以有效地解决存储密度过低的问题，这种存储结构又叫块链存储。主要方法是让每个结点存储更多的字符，从而降低了指针域占存储空间的比例，提高了存储密度。比如一个字符串S="My XJTU"，且每个结点存储4个字符，那么S的存储结构如图4-15所示。

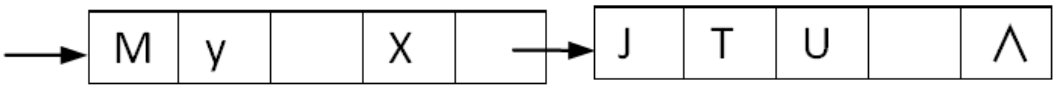


图 4-15 串的块链存储结构

显然块链存储结构能够增加串的存储密度，但在这种存储结构上对串进行插入、删除操作则显得很不方便。

3.7.4 串类及其实现

算法3.30： 字符串类定义

```
class String {
private:
    char *ch; //串存放数组
    int size; //存放数组的长度
public:
    String();
    String(const char *init);
    String(const String& ob);
    ～String() {delete []ch;}
    int size()const {return size}
    String& operator() (int pos, int len); //当0≤pos<maxSize 且0≤len 且
    pos+len<maxSize //则在串*this 中从pos 位置开始取len 个字符组成子串返回。
```

```
    int operator == (String& ob)const {return strcmp(ch,ob.ch)== 0;}
    int operator != (String& ob)const {return strcmp(ch,ob.ch) != 0;}
    int operator ! ()const {return size== 0;}
    String& operator = (String& ob);
    String& operatr=(char* s);
    String& operator += (String& ob);
    char& operator [] (int i);
    int find(const String& pat)const;//找到子串pat，并返回其位置
    int KMP_Find(const String &pat, int next[ ]);//KMP模式匹配算法
    void getNext(int next[]);//求字符串的next数组
}
```

算法3.31：字符串类部分函数的实现

```
String::String() { //串构造函数，不带初值
    size=0;
    ch = new char[size+1];
    if (ch == NULL) {cerr << "存储分配失败\n"; exit(1);}
    ch[0] = '\0';
}
String::String(const char *init) { //串构造函数，带初值
    size=strlen(init);
    ch = new char[size+1];
    if (ch == NULL) {cerr << "存储分配失败\n"; exit(1);}
    strcpy(ch,init);
}
```

```
String::String(const String& ob) { //串复制构造函数
    size=ob.size;
    ch = new char[size+1];
    if (ch == NULL) {cerr << "存储分配失败\n"; exit(1);}
    strcpy(ch,ob.ch);
}
String String::operator () (int pos, int len) {//求子串，即取子串
String temp;
    if (pos < 0 || pos+len-1 >= maxSize || len < 0) {
        temp.size = 0; temp.ch[0] = '\0';
    } else {
        if (pos+len-1 >= size) len = size-pos;
        temp.size = len;
        for (int i = 0,j = pos; i < len; i++, j++) temp.ch[i] = ch[j];
        temp.ch[len] = '\0';
    }
    return temp;
}
```

```
String& String::operator = (const String& ob) {//串重载操作: 串赋值
    if (&ob != this) { //若两个串相等为自我赋值
        delete []ch;
        ch = new char[size+1]; //重新分配，加一个字符放“\0”
        if (ch == NULL) {cerr << “存储分配失败!\n ”; exit(1);}
        size=ob.size;
        strcpy(ch,ob.ch);
    } else cout << “字符串自身赋值出错!\n”;
    return *this;
}
```

```
String& String :: operator += (const String& ob) {//串重载操作: 串连接
    char *temp = ch;       //暂存原串数组
    int n = size+ob.size;  //串长度累加
    ch = new char[n+1];
    if (ch == NULL) {cerr << "存储分配错!\n "; exit(1);}
    size = n;
    strcpy(ch, temp); //拷贝原串数组
    strcat(ch, ob.ch); //连接ob 串数组
    delete []temp;
    return *this;
}
```

3.7.5 串的模式匹配

- **模式匹配**是指子串在主串中的定位。
- 模式匹配成功是指在主串s中能够找到模式串t，否则，则说明模式串t在主串s中不存在。
- 在串类的方法中，函数find（String& pat）就是用来进行字符串的模式匹配的。
- 通常又将**子串称为模式串**。

1. Brute-Force算法

Brute-Force是**最朴素的一种字符串匹配算法**，其主要思想是：从主串ob="$s_0s_1\ldots s_{m-1}$"的第一个字符开始与模式串pat="$t_0t_1\ldots t_{n-1}$"的第一个字符比较：若相等，则继续比较有序字符；否则，从主串ob的第二个字符开始重新与模式串pat的第一个字符进行比较。如此进行，若在主串ob中有一个与模式串相等的连续字符序列，则匹配成功，函数返回模式串pat在主串ob中的位置；否则，匹配失败，函数返回-1。

设主串ob＝“abacabab”，模式串pat=“abab”，ob的长度为m=8,模式串的长度为n=4，用变量i指示主串ob的当前比较字符的下标，用j指示模式串pat的当前下标。模式匹配的过程如图所示。

图3-16 Brute-Force算法的字符串模式匹配过程

算法3.32：Brute-Force算法

```
int String::find(const String &pat)const{
int i = 0, j = 0;
    if ( size == 0 || pat.size == 0 )            return -1;
    while ( u <= size – 1 && j <= pat.size -1 ){
          if ( ch[i] == pat.ch[j] ){ i++; j++; }
          else {  i = i – j + 1; j = 0; }
    }
    if ( j == pat.size – 1 ) return i – pat.size;
    return -1;
}
```

- Brute-Force算法是一种带回溯的算法。
- 在最坏情况下，最多需要比较m-n+1趟，且若每趟比较都在最后才出现不匹配的情况，要做n次比较，那么总的比较次数就是(m-n+1)*n。
- 通常n会远小于m，所以最坏情况下运行时间是O(m*n)。
- 最好情况下，该算法的时间复杂度为O(m+n)

2. 模式匹配的KMP算法

模式中前k个字符的子串必须满足下列关系：

$$\text{“}t_0t_1\cdots t_{k-1}\text{”}=\text{“}s_{i-k}s_{i-k+1}\cdots s_{i-1}\text{”}$$

而已经得到的“部分匹配”的结果是：

$$\text{“}t_{j-k}t_{j-k+1}\cdots t_{j-1}\text{”}=\text{“}s_{i-k}s_{i-k+1}\cdots s_{i-1}\text{”}$$

由上面两式可推得下列等式：

$$\text{“}t_0t_1\cdots t_{k-1}\text{”}=\text{“}t_{j-k}t_{j-k+1}\cdots t_{j-1}\text{”}$$

因此，若模式串中存在满足上面式子的子串，则当匹配过程中，主串中第i个字符与模式中第j个字符比较不等时，仅需将模式串向右滑动至模式串中第k个字符和主串中第i个字符对齐。这样，下次比较就从主串的第i个字符和模式串的第k个字符开始，主串指针i也不必回溯。

KMP算法的思想：

设s为主串，t为模式串，i和j分别为指向主串和模式串中正在进行比较字符的指针。开始时，令i=0，j=0。如果$s_i=t_j$，则使i和j的分别加1；反之，i不变，j退回到j=next[j]的位置，然后再对s_i和t_j进行比较。依次类推，出现下列两种情况之一。

（1）j退回到某个j=next[j]时，若$s_i=t_j$，则指针i、j的值各加1，然后继续匹配。

（2）j值退回到j=-1，此时令指针i、j的值各加1，也即下一次对s_{i+1}和t_0进行比较。

- KMP算法和Brute-Force算法很相似，不同之处仅在于，当匹配过程中产生“失配”时，指针i不变，指针j退回到next[j]所指位置的字符，然后继续比较。假设已知next[j]，则KMP算法的描述如下。
- 若令next[j]=k，则next[j]表明当模式串中第j个字符与主串中相应字符“失配”时，在模式串中需重新和主串中该字符进行比较的字符位置。通过上面的分析，可以将next[j]定义为：

$$next[j]=\begin{cases}Max\{k \mid 0<k<j，且\ t_0t_1\cdots t_{k-1}=t_{j-k}t_{j-k+1}\cdots t_{j-1}\}，& \text{当此集合不空时}\\ 0 & \text{其它情况时}\\ -1 & \text{当 } j=0 \text{ 时}\end{cases}$$

算法3.33： KMP算法描述

```
int String::KMP_Find(const String &pat, int next[ ]){
    //在串中查找模式串t，若找到则返回t的首字符在ch中的位置，否则返回-1
    int i=0, j=0, pos;
    while ( i < size -1 && j < pat.size - 1){
        if ( j == -1 || ch[i] == pat.ch[j] ){  //当si=tj时，i，j分别加1然后继续比较；
                //当j== -1时要使si+1和t0进行比较，因而i,j分别加1继续比较
                i++; j++;
        }else j = next[j];
    }
    if ( j > pat.size – 1 ) pos = i – pat.size + 1;
    else pos = -1;
    return pos;
}
```

- 给出了KMP算法后，下面的问题就是：如何确定模式串的next[]数组？由next[]的定义可知，求模式串的next[j]值只与模式串本身相关，而与主串无关。

算法3.34：计算串的next[j]的算法

```
void String::getNext(int next[]){
    int j=0, k=-1;
    next[0]=-1;
    while( j < size -1){
        if ( k == -1 || ch[j] == ch[k] ){
        //当tj=tk时，j和k分别加1，同时计算next[j],再继续比较
        //当k==-1时，要使sj+1与t1比较，因而j和k分别加1，计算next[j],再继续比较
                j++; k++;
                next[j]=k;
        } else k=next[k];
    }
}
```