# 数 据 结 构

## 第三章 栈和队列 [1]

任课老师：郭艳

数据结构课程组

计算机学院

中国地质大学（武汉）2020年秋

# 上堂课要点回顾

- 单链表
  - 应用：有序线性表的有序插入
  - 案例：算法设计（遵循结构化程序设计方法）与描述（思想、步骤、效率分析）
- **ADT LinearList的实现方案3——循环单链表**
  - 特点、模型、优点和实现要点
  - 循环单链表（带头结点）类的定义和实现（“CircLinkedList.h“）
    - class CircList<T>类
      - 数据成员/函数成员
- **ADT LinearList的实现方案4——双向循环链表**
  - 特点、模型和特征
  - 双向循环链表（带头结点）类的定义和实现（“DBLinkedList.h“）
    - struct DblNode<T>类
      - 数据成员/函数成员
    - class DblList<T>类
      - 数据成员/函数成员bool Insert(int, T&, int)、bool Remove(int, T&, int)及性能
- 静态链表
- **顺序表和链表的应用场合的选择***

# 第四次课

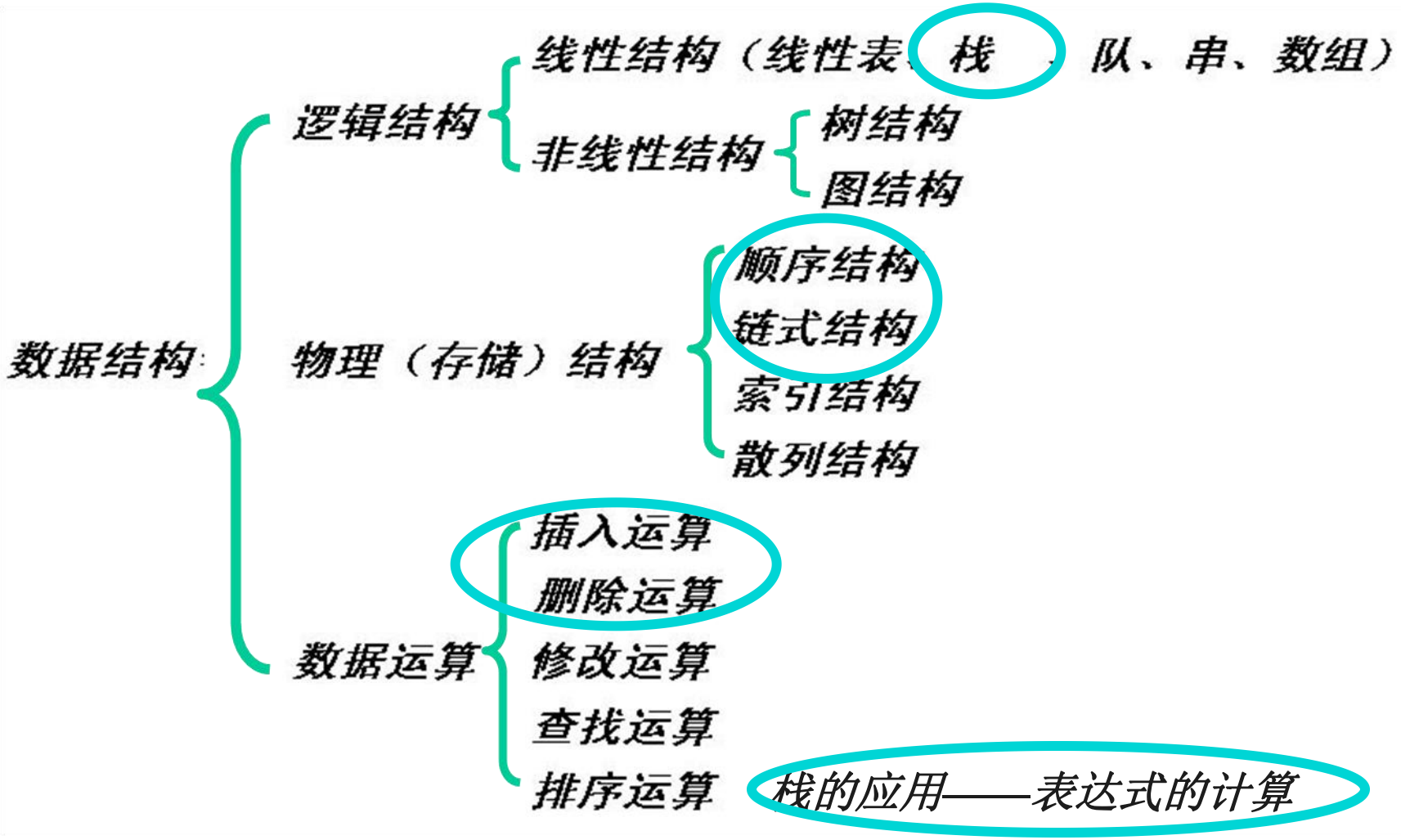
阅读：

殷人昆，第88-109页

练习：

作业4

# 数据结构课程内容


数据结构
逻辑结构
线性结构（线性表、栈、队、串、数组）
非线性结构
树结构
图结构
物理（存储）结构
顺序结构
链式结构
索引结构
散列结构
数据运算
插入运算
删除运算
修改运算
查找运算
排序运算
栈的应用——表达式的计算

# Chapter3栈和队列

## 1 栈

3.1.1 栈的定义、特征

3.1.2 顺序栈

3.1.3 链栈

3.1.4-5栈的应用

3.2 栈与递归

特征和应用是重点
——要求能正确应用它们解决实际问题

## 2 队列

3.3.1 队列的定义、特征

3.3.2 顺序队列和顺序循环队列

3.3.3 链队列

3.3.4 顺序队列的应用

3.4 优先级队列

3.5 双端队列（自学）

# 3.1 栈

## 3.1.1 栈（Stack）的基本概念

- 定义

什么是栈?

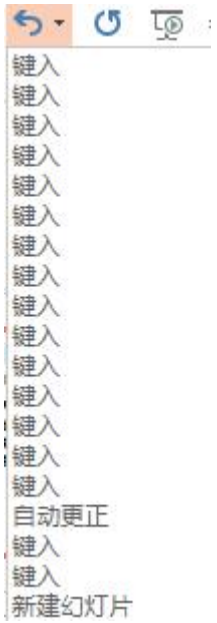
↓

盘子的放/取： 取出或放入一个盘子，在顶部操作才是最方便的

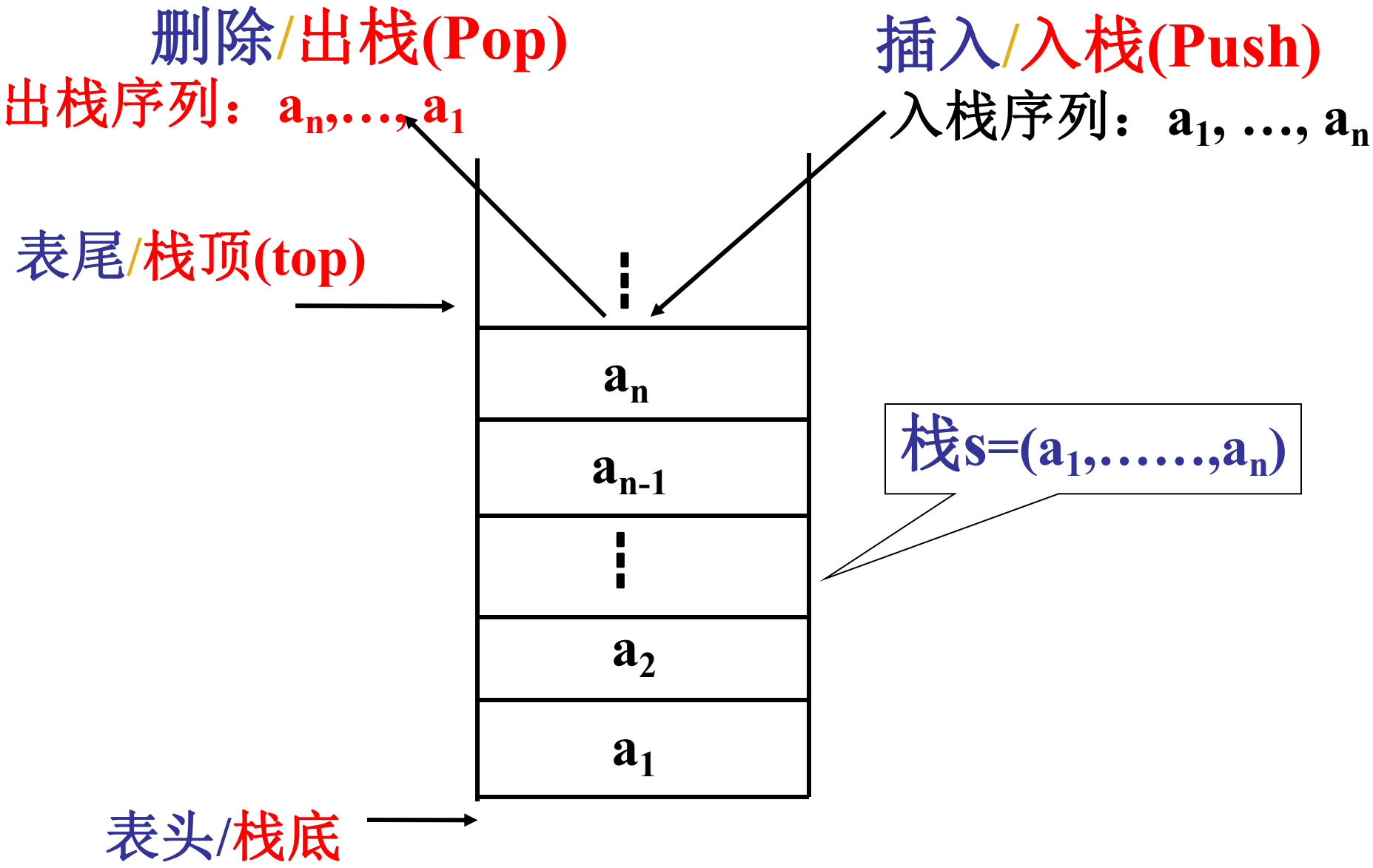
↓

栈是限定只能在表的一端进行插入和删除运算的线性表

Powerpoint撤销/恢复



例：


删除/出栈(Pop)
插入/入栈(Push)
出栈序列：$a_n,\ldots,a_1$
入栈序列：$a_1,\ldots,a_n$
表尾/栈顶(top)
$a_n$
$a_{n-1}$
$a_2$
$a_1$
栈$s=(a_1,\ldots\ldots,a_n)$
表头/栈底

## ◆ 栈的特征

每次取出（或删除）的总是刚压进的元素，而最先压入的元素则被放在栈底

**后进先出（Last In First Out，LIFO）**

**或**

**先进后出（First In Last Out，FILO）**

返回

# 栈的抽象基类 P88 程序3.1

```
const  int maxSize 50;
template <class T>
class Stack {              //stack.h实现栈的类定义
public:
    Stack(){ };                          //构造函数
    virtual void Push(const T& x) = 0;   /*进栈：在栈顶插入
                                          一个值为x的元素*/
    virtual bool Pop(T& x) = 0;                  /*删除栈顶元素，
                                  被删除的栈顶元素通过x带回。*/
    virtual bool getTop(T& x) const = 0; //取栈顶元素
    virtual bool IsEmpty() const = 0;     //判栈空
    virtual bool IsFull() const = 0;      //判栈满
    virtual bool getSize() const = 0; //计算栈中元素个数
}; //思考：栈接口的时间复杂度是0（？）
```

返回

# 3.1.2 顺序栈 P89 程序3.2

```
#include <assert.h>          //SeqStack.h文件
#include <iostream.h>
#include "stack.h"
const int stackIncreament=20;
template <class T>
class SeqStack : public Stack<T> {   //顺序栈类的定义
private:
    T *elements;                     //栈元素存放数组
    int top;                  //栈顶指针，指示栈顶下标
    int maxSize;                     //栈最大容量
    void overflowProcess();          //栈的溢出处理(自学)
```

```
public:
    SeqStack(int sz =50);                    //构造函数
    ～SeqStack() { delete []elements; }   //析构函数
    bool getTop(T& x) const;      //取栈顶元素
    void Push(const T& x);         //进栈
    bool Pop(T& x);                  //出栈
     bool IsEmpty() const { return top == -1; }//判栈空
    bool IsFull() const { return top == maxSize-1; }
    int getSize() const { return top+1; }//获取表项个数
    void makeEmpty() { top = -1; }      //置空表
    friend ostream& operator << (ostream& os,
SeqStack<T>& s);//输出栈中元素的重载操作<< (略，自学)
};
```

返回

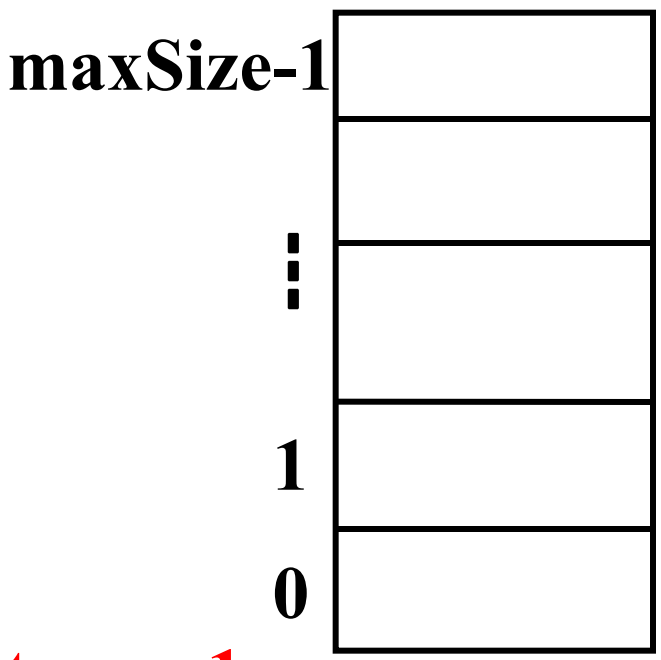
# 【顺序栈的构造函数】 P90 程序3.3（自学）

```
template<class T>
SeqStack<T>::SeqStack(int sz):top(-1),
maxSize(sz)
{
   elements = new T[maxSize];
   assert(elements != NULL);
}
```

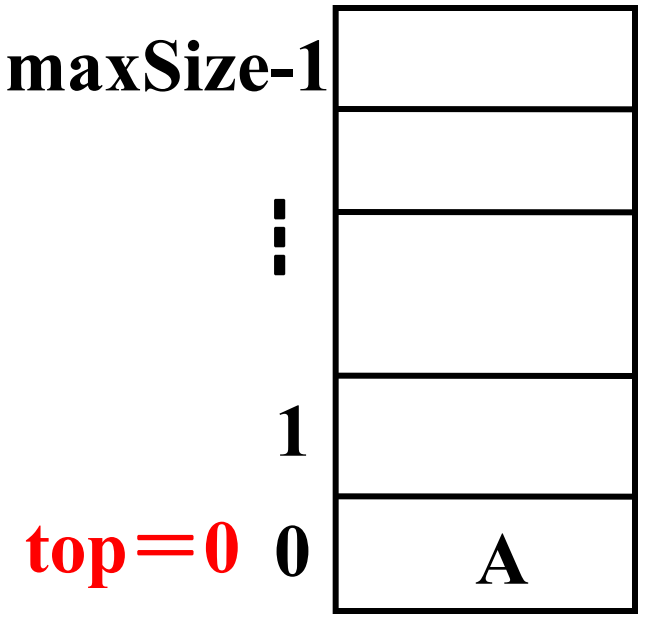
# 【顺序栈的扩充空间函数】P90 程序3.4(1) 自学

```
template<class T>
void SeqStack<T>::overflowProcess()
{//私有函数，扩充栈的存储空间（自学）
        T * newArray = new T[maxSize + stackIncreament];
        if(newArray = NULL)
        {
                cerr<< "存储分配失败!" << endl;
                exit(1);
        }
        for(int i =0;i<= top;i++)
                newArray[i] = elements[i];
        maxSize = maxSize + stackIncreament;
        delete[] elements;
        elements = newArray;
}
```

# 栈操作时栈顶指针的变化情况：

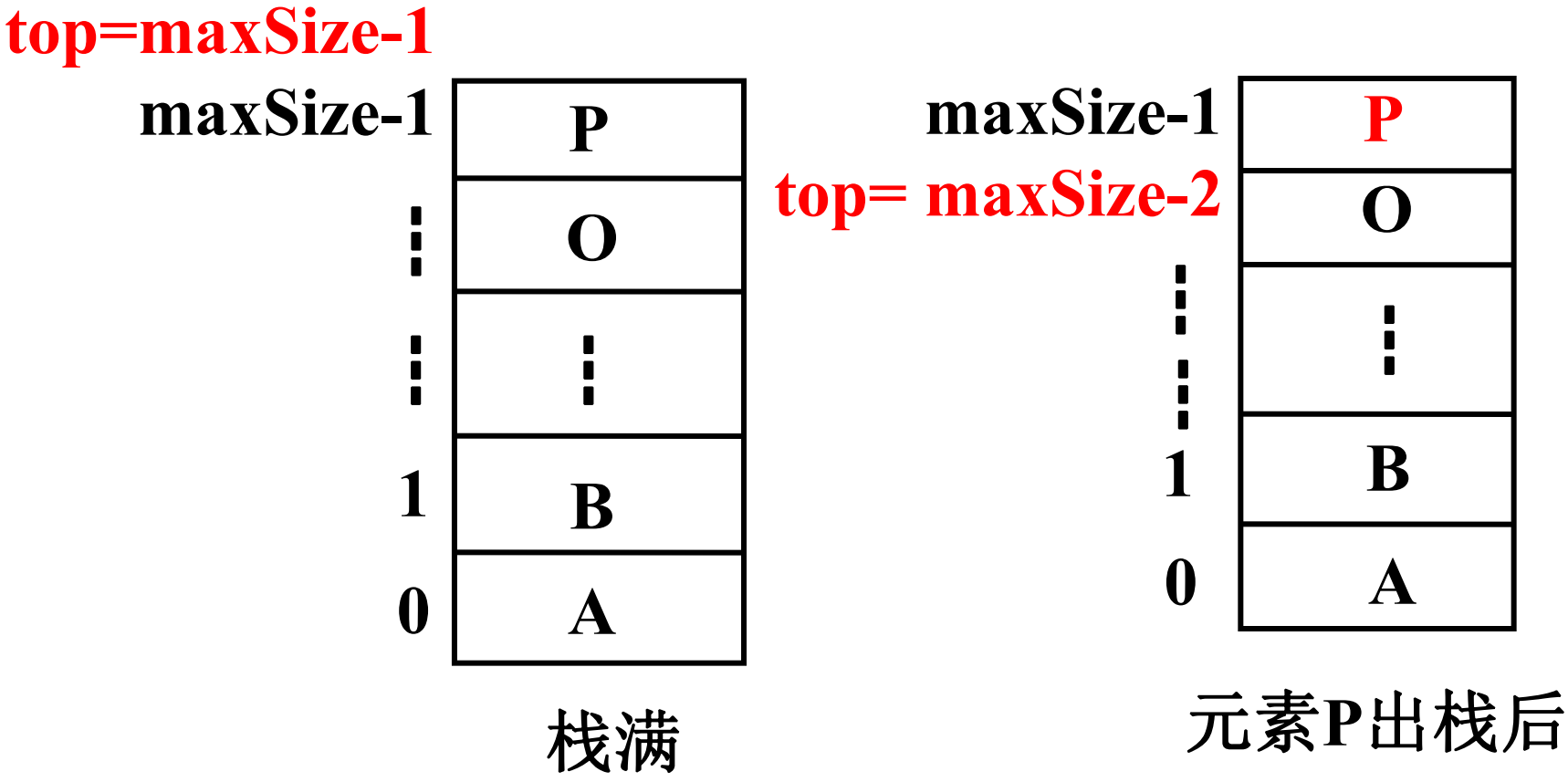



栈空




元素A入栈后

# 栈操作时栈顶指针的变化情况（续）：



# 【顺序栈的取栈顶元素操作】P91 程序90(4)

```
template <class T>
bool SeqStack<T>::getTop(T& x) const{
 //通过形参x返回栈顶元素的值
        if (IsEmpty() == true)
                return false;
        x = elements[top];
        return true;
}
 //时间复杂度：O(1)
```

# 【顺序栈的入栈操作】P91 程序90(2)

算法思想：先判断栈是否满，如果栈满则作溢出处理；如果栈不满则栈顶指针先加1，然后新元素再进栈。

```
template <class T>
void SeqStack<T>::Push(const T& x) {
      //若栈不满，则将元素x插入该栈栈顶，否则溢出处理
      if (IsFull() == true)  //栈满
            overflowProcess( );
     elements[++top] = x;     //栈顶指针先加1，再进栈
}
```

**//时间复杂度：O(1)**

# 【顺序栈的出栈操作】P91 程序90(3)

算法思想：先判断栈是否为空，如果栈空则出栈失败返回false；如果栈非空，则返回栈顶元素的值，然后栈顶指针减1，返回true。

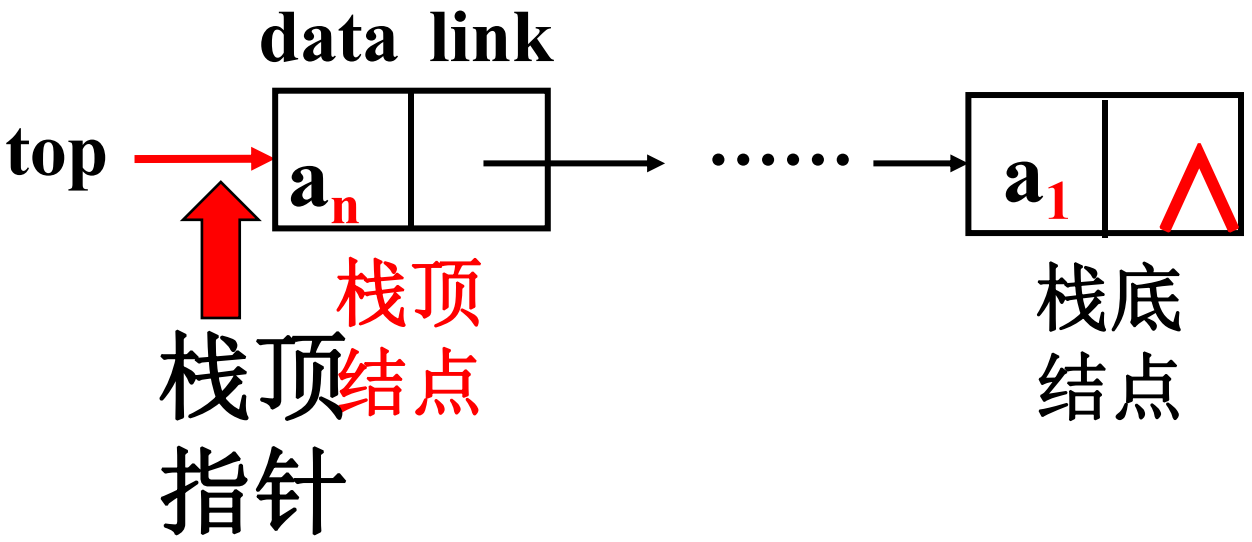
```
template <class T>
bool SeqStack<T>::Pop(T& x) {
//函数退出栈顶元素并返回栈顶元素的值
        if (IsEmpty() == true)
            return false;
        x = elements[top--];     //栈顶指针退1
        return true;                  //退栈成功
}
```

**//时间复杂度：O(1)**

# 3.1.3 单链栈及其实现

## ◆ 单链栈模型：指针方向从栈顶向下链接

非空链栈　　满足top!=NULL




空链栈　　满足top==NULL

# 链栈类的定义 P93 例3.6

```
#include <iostream.h>              //LinkedStack.h文件实现单链栈
#include "LinkedList.h"
#include "stack.h"
template <class T>
class LinkedStack:public Stack<T>{
private:
    LinkNode<T> *top;      //栈顶指针
public:
    LinkedStack():top(NULL) { }
    ~LinkedStack() {makeEmpty();}
    bool IsEmpty() const {return top==NULL;}
    bool IsFull() const {return false;}
    bool getTop(T& x) const;
    int getSize() const;
    void Push(const T& x);
    bool Pop(T& x);
     void makeEmpty();
    //friend ostream & operator << (ostream& os,SeqStack<T>& s);};
```

## 【单链栈的取栈顶函数】P93 程序3.6(5)

```
template<class T>bool LinkedStack<T>::getTop(T&
x)const
{       if(IsEmpty() == true) return false;
        x = top->data;
        return true;            }       //O(1)
}
```

## 【单链栈的求元素个数函数】P93 程序3.6(6)，自学

```
template<class T>int LinkedStack<T>::getSize() const
{       LinkNode<T> *p = top;  int k = 0;
        //p：遍历指针，k：计数器
        while (p != NULL)
        {       p = p->link;
                k++;  }
        return k;
}       //O(n)
```

# 【链式栈的入栈操作】P93 程序3.6(3)

```
template <class T>
void LinkedStack<T>::Push(const T& x) {
//将元素值x插入到链式栈的栈顶，即链头。
    top = new LinkNode<T> (x, top);  //创建新结点
    assert (top != NULL);                    //创建失败退出
}//O(1)
```

# 【链式栈的出栈操作】P93 程序3.6(4)

```
template <class T>
bool LinkedStack<T>::Pop(T& x) {
//删除栈顶结点, 返回被删栈顶元素的值
    if (IsEmpty() == true) return false;   //栈空返回
    LinkNode<T> *p = top;                  //暂存栈顶元素
    top = top->link;                       //退栈顶指针
    x = p->data;
    delete p;              //释放结点
    return true;
}//O(1)
```

# 【单链栈的置空函数】P93 程序3.6(2)，自学

```
template<class T>
void LinkedStack<T>::makeEmpty()
{//逐次删去链式栈中的元素直至栈顶指针为空
        LinkNode<T> *p ;      //指向被删结点
        while (top != NULL)
        {
                p = top;
                top = top->link;
                delete p ;
        }
}
```

# 小结

- 实际应用中，顺序栈比链栈用得更广泛
  - 顺序栈容易根据栈顶位置，进行相对位移、快速定位、并读取栈的内部元素
  - 顺序栈读取内部元素的时间为**O(1)**，而链栈为**O(n)**。一般来讲，栈不允许“读取内部元素”，只能在栈顶操作

# 3.1.4–5 栈的应用*

## 引　言

通常，**数据处理的逻辑或框架**是：

搜索（遍历）操作对象集合，在搜索（遍历）的同时访问（处理）各个对象。当遍历结束时，完成所有对象处理，得到输出。

当正在访问（处理）的对象当前**不能完成所有**对它的处理时，需要**缓存该对象**。当等待到合适时机，再**从缓存取出该对象**并完成所有对它的处理。

如果**最先**访问（处理）的对象**最后**才能完成所有对它的处理，或者，**最后**访问（处理）的对象**最先**完成所有对它的处理，则**采用栈**作为未完成访问（处理）的对象的**缓存数据结构**。

# 3.1.4-5 栈的应用

## ——计算算术表达式的值（P94 括号匹配问题 自学）

### 中缀表达式

- 运算符在两个操作数的中间
- 需要括号改变优先级，例如：

$$A/(B+C\times D)-E$$

### 后缀表达式

- 操作符在两个操作数的后面
- 后缀操作数的顺序同于中缀；操作符的顺序就是表达式计算的顺序，完全不需要括号。例如：

$$ABCD\times+/E-$$

- 因此，后缀表达式适合用来计算表达式的值

# 计算算术表达式值的两个步骤

对一个中缀表达式的计算可以通过两个子步骤来实现：

① 先把中缀表达式变换为后缀表达式；

② 根据后缀表达式计算表达式的值。

# Calculator.h文件实现中缀表达式求值 P97程序3.8

```
#include"SeqStack.h"
#include<iostream.h>
#include <ctype.h> //调用isdigit( )函数
#include<string.h>
#include <stdlib.h>
class Calculator
{ public:       Calculator(int sz):s(sz){};
                void Run();                 //根据后缀表达式求值
                void Clear(){s.makeEmpty();} //置栈为空栈
                void postfix(char *e);          // 中缀转为后缀
                int isp(char ch);  //操作符ch进s栈后的优先级数
                int icp(char ch);  //操作符ch的优先级数
  private:      SeqStack<double> s;           //缓存操作数的栈
                void AddOperand(double value); //操作数入s栈
                bool Get2Operands(double& left,double& right);
//从s栈得到操作数（右）和被操作数（左），成功返回true，否则返回false
                void DoOperator(char op); //进行op运算
};
```

返回

② 对后缀表达式如何进行计算呢?

很简单，只要从左到右依次扫描表达式的各单词。由于扫描到的操作数并不能马上参加运算，并且先扫描到的操作数将后参与计算，因此采用栈来缓存操作数。因此，如果当前读到的是操作数，存入一个栈（操作数栈）中，如果当前读到的是运算符，就取栈前面的两个操作数（即从栈顶依次弹出两个元素）进行运算，中间结果同样存入栈中，作为下一次运算的操作数。如此反复，直到表达式处理完毕。

```
//P87 程序 3.9(3)  操作数value入s栈
void Calculator::AddOperand(double value)
{  s.Push(value);
};
/*P87 程序 3.9(2) 从s栈得到操作数（right）和被操作数（left
），成功返回true，否则返回false*/
bool Calculator::Get2Operands(double&
left,double& right){
        if(s.IsEmpty()==true)
        {cerr<<"缺少右操作数！ "<<endl; return false;}
        s.Pop(right);
        if(s.IsEmpty()==true)
        {cerr<<"缺少左操作数！ "<<endl; return false;}
        s.Pop(left);
        return true;
};
```

```
//P87 程序 3.9(1)  进行op运算，中间结果value入s栈缓存
void Calculator::DoOperator(char op)
{  double left,right,value;
   bool result;
   result = Get2Operands(left,right);
   if(result==true)
      switch(op){
         case'+':value=left + right; s.Push(value);break;
         case'-':value=left - right; s.Push(value);break;
         case'*':value=left * right; s.Push(value);break;
         case'/':if(right==0.0){cerr<<"Divide by 0!"<<endl;Clear();}
                 else{value=left / right; s.Push(value);break;
         }
  else Clear();
};
```

```
void Calculator::Run()  //P98 程序3.10(1)
//输入后缀表达式（最后以’#’结束），输出计算结果
{  char ch;double newOperand;
   while(cin>>ch,ch!='#')
   {     switch(ch){
                 case'+':
                 case'-':
                 case'*':
                 case'/': DoOperator(ch);break;
                 default:cin.putback(ch);
                         cin>>newOperand;//重读
                         AddOperand(newOperand);}
   }
   s.getTop(newOperand);       cout << newOperand;
} //操作数先后以char和double格式从输入缓冲区被读入两次（先被读入一次，再putback()回输入缓冲区）
```

返回

# ① 如何将中缀表达式变换成后缀表达式？

利用运算符优先级法。中缀表达式的计算次序（算术四则运算规则）：

①在无括号或同层括号时，先乘除后加减；

②同级运算从左到右；

③在有括号时，先括号内后括号外。

从而得到

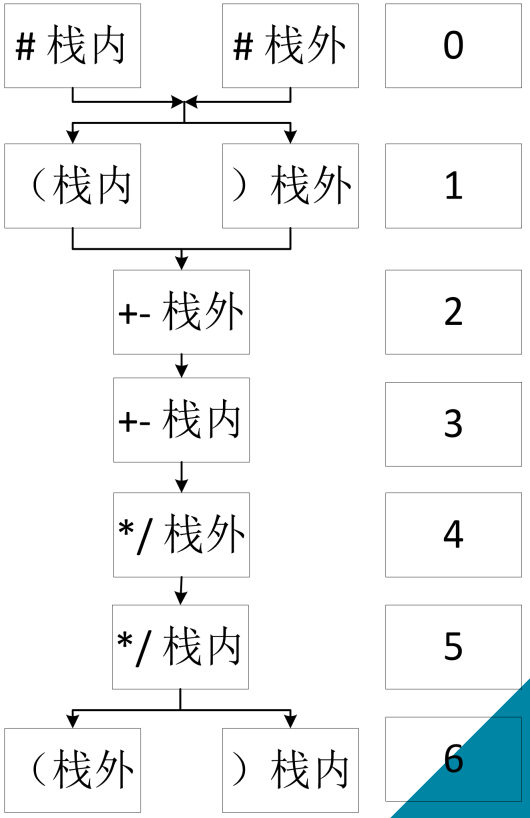
当前扫描到的运算符ch的优先级数icp(ch)

和

该操作符进栈后的优先级数isp(ch)：

| 操作符ch | # | ( | *, / | +, − | ) |
|---|---|---|---|---|---|
| isp (栈内) | 0 | 1 | 5 | 3 | 6 |
| icp (栈外) | 0 | 6 | 4 | 2 | 1 |

数值越大优先级越高



两个相继出现的运算符op（在前）和ch（在后）之间的优先关系：

| op \ ch | + | − | * | / | ( | ) | # |
|---|---|---|---|---|---|---|---|
| + | > | > | < | < | < | > | > |
| − | > | > | < | < | < | > | > |
| * | > | > | > | > | < | > | > |
| / | > | > | > | > | < | > | > |
| ( | < | < | < | < | < | = | ERROR3 |
| ) | > | > | > | > | ERROR1 | > | > |
| # | < | < | < | < | < | ERROR2 | = |

# 表达式“中缀->后缀”算法的基本思想

1．置运算符栈S为空栈，然后表达式起始符‘#’入栈。

2．循环：读入中缀表达式中的一个字符ch，

2.1 若是操作数，就将其输出，接着转2(即读下一字符)；

2.2 若是运算符，将ch的优先级icp和位于栈顶的操作符op的优先级isp比较优先级：

若icp(ch) > isp(op)：将ch进栈，转2(即读下一字符)；

若icp(ch) < isp(op)：则将栈顶运算符op出栈，并作为后缀表达式的一个单词输出，再转2.2。

若icp(ch) == isp(op)：若op为‘(’，ch为‘)’，

则‘(’退栈但不输出；转2(即读下一字符)。

直至整个表达式处理完毕(当‘#’＝‘#’时)。

例：利用上述算法，转换中缀表达式“#A／(B＋C＊D)－E为后缀表达式#”。

| 中缀表达式 | Stack | 输出(后缀表达式) |
| --- | --- | --- |
| A/(B+C*D)－E# | # | |
| /(B+C*D)－E# | # | A |
| (B+C*D)－E# | # ／ | A |
| B+C*D)－ E # | #／ ( | A |
| +C*D) －E # | #／ ( | AB |
| C*D) －E # | #／ ( ＋ | AB |
| *D） －E# | #／ ( ＋ | ABC |

| 中缀表达式 | Stack | 输出(后缀表达式) |
| --- | --- | --- |
| D）－E# | #／(+* | ABC |
| ）－E# | #／(+* | ABCD |
| ）－E# | #／(+ | ABCD* |
| ）－E# | #／( | ABCD*+ |
| －E# | #／ | ABCD*+ |
| －E# | # | ABCD*+/ |
| E# | #－ | ABCD*+/ |
| # | #－ | ABCD*+/E |
| # | # | ABCD*+/E－,结束 |

```
void Calculator::postfix(char *e)
// 中缀表达式转换为后缀表达式 P100 程序3.11
{  char ch,op;   int i=0;ch=e[i];
   SeqStack<char> s;            //操作符栈s
   s.makeEmpty();    s.Push('#');
   while(  s.IsEmpty()==false && ch!='#'  )
  { if ( isdigit(ch) ){    cout<<ch; i++; ch=e[i];  } //若读到的是操作数
    else{           s.getTop(op);                    //若读到的是操作符
                    if(  icp(ch) > isp(op)  )
                    {      s.Push(ch); i++; ch=e[i];                  }
                    else if(  icp(ch) < isp(op)  )
                    {      s.Pop(op);
                           if(  op!='#'  ) cout<<op;
                    }
                    else{    s.Pop(op);                 //优先级相等
                             if(op=='(') {i++; ch=e[i];}
                    }
         }
}//时间复杂度：O(n)
```

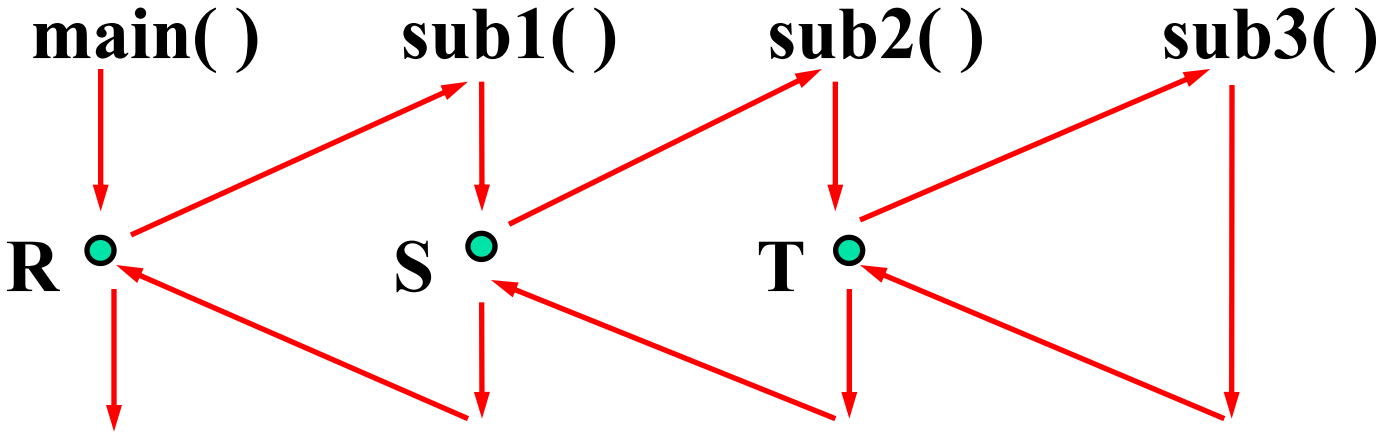
返回

```
void main()
{  Calculator c;
   char *exp="5.5*3+(2-4/6)#";
   c.postfix(exp);
   cout<<endl<<"运算表达式"<<exp<<"的值："<<endl;
   c.Run();
}
```

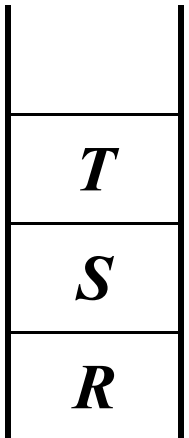
# 3.2 栈与递归

## 多个函数的嵌套调用

例：




需要保存返回地址




- 后调用先返回

存：R，S，T

取：T，S，R

# 递归（Recursion）函数:

自己（直接或间接地）调用自己的函数

- 例1：阶乘的递归定义

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

- 例2：2阶Fibonacci数列

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

# 阶乘n!的递归定义

$$factorial(n)=\begin{cases}1 & \text{当}n=0\text{时}\\ n\times factorial(n-1) & \text{当}n>0\text{时}\end{cases}$$

# 计算n！的两个函数

```
long  fact(int n)
//使用循环迭代方法，计算n!的函数
{ long m=1;int i;
   if(n>0)
   for(i=1;i<=n;i++)
   m=m*i;
   return m;
}
```

**//时间复杂度：O(n)**

//计算n!的递归函数 P101 程序3.12（改）

```
long Fact(int n)
{ long y;
   if (n<0) { cout<<“参数错误”<<endl; reurn -1;}
   if (n==0)  return 1;
   else
    {    y=Fact(n-1);
         return n*y;
     }
}
main( )
{  long fn=Fact(4);
   cout<<fn<<endl;}
```

# 递归示例：n!

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

- 递归函数必须包含一个递归出口。
- 递归调用部分所使用的参数值应比函数的参数值要小，以便重复调用能最终获得基本部分所提供的值。

```
long  fact(int n)
//使用循环迭代方法，计算n!的函数
{  long m=1;int i;
   if(n>0)
      for(i=1;i<=n;i++)
          m=m*i;
   return m;
} //时间复
```

```
long Fact(int n)
{ long y;
  if (n<0) reurn -1;
  if (n==0)  return 1;
  else
   {    y=Fact(n-1);
        return n*y;
    }
}//时间复杂度：O(n)
```

# 递归示例：2阶斐波那契数列

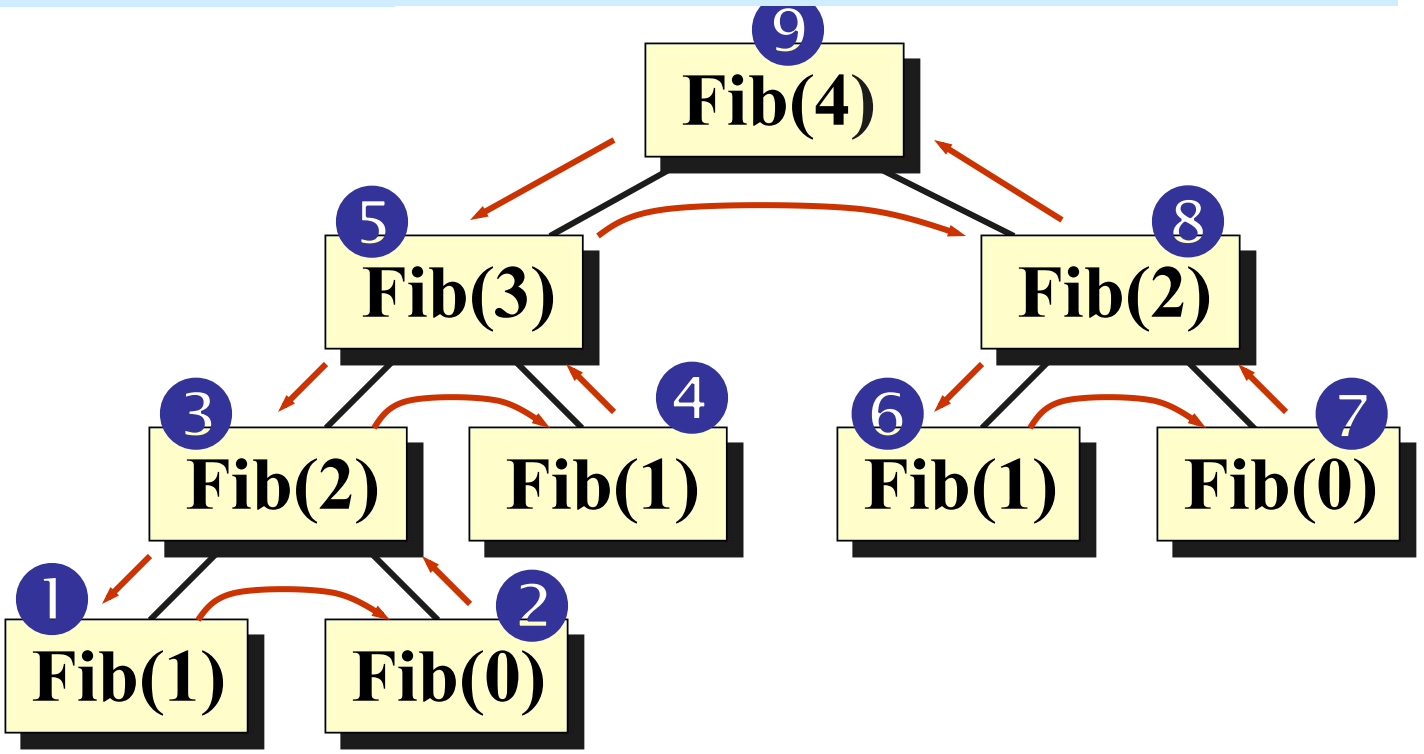
$$Fib(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ Fib(n-1)+Fib(n-2) & n>1\end{cases}$$

- 递归函数必须包含一个递归出口。
- 递归调用部分所使用的参数值应比函数的参数值要小，以便重复调用能最终获得基本部分所提供的值。

```
int fibonacci(int n)
{
    if(n==0) return 0;
    if(n==1) return 1;
    int oneBack=1;
    int twoBack=0;
    for(int i=2;i<=n;i++)
     {
          int current=oneBack+twoBack;
          twoBack=oneBack;
          oneBa
     }
      return cu
}
//时间复杂度
```

```
int Fibonacci(int n)
{
    if(n==0) return 0;
    if(n==1) return 1;
   return Fibonacci(n-1)
          +Fibonacci(n-2);
}//时间复杂度为O(2^n)
```

# 斐波那契数列的递归调用树和求解顺序




时间复杂度：调用次数 NumCall(n) = 2*Fib(n+1) - 1=O($2^n$) (n>0)。究其原因在于，计算过程中所出现的递归实例的重复度极高

空间复杂度O(n).

# 递归原理

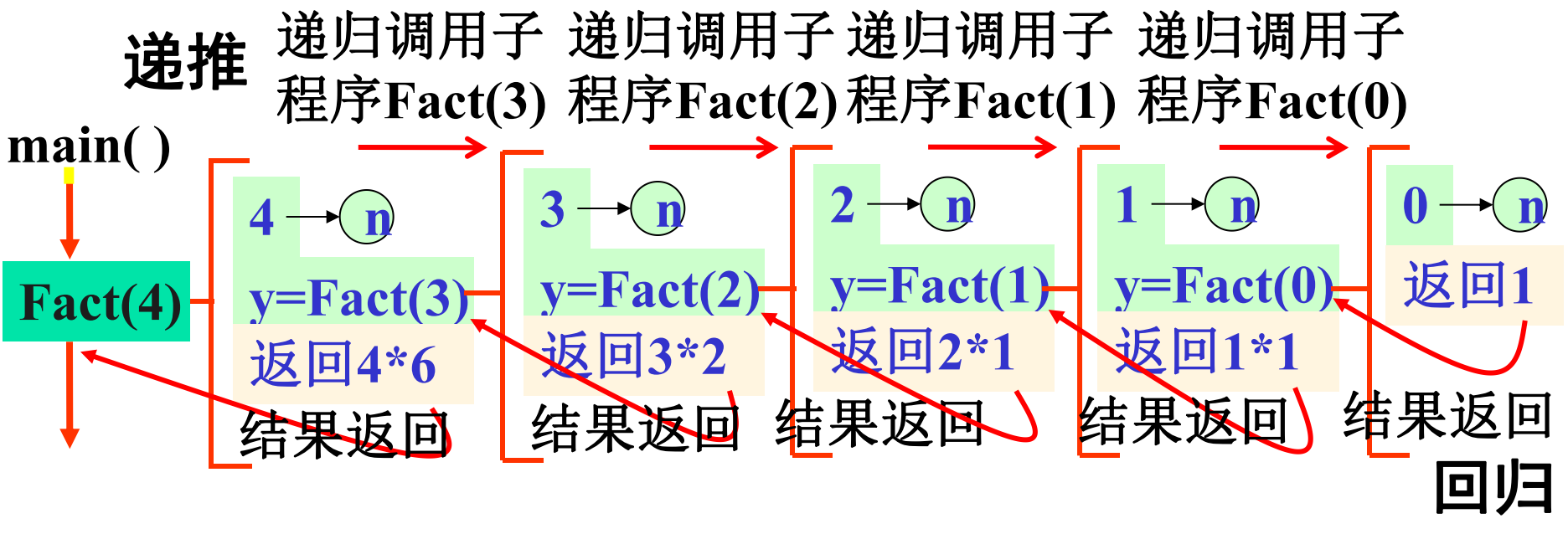
- 每一次递归调用时，需要为过程中使用的参数、局部变量等另外分配存储空间。
- 每层递归调用需分配的空间形成递归工作记录，按后进先出的栈组织。——递归工作栈

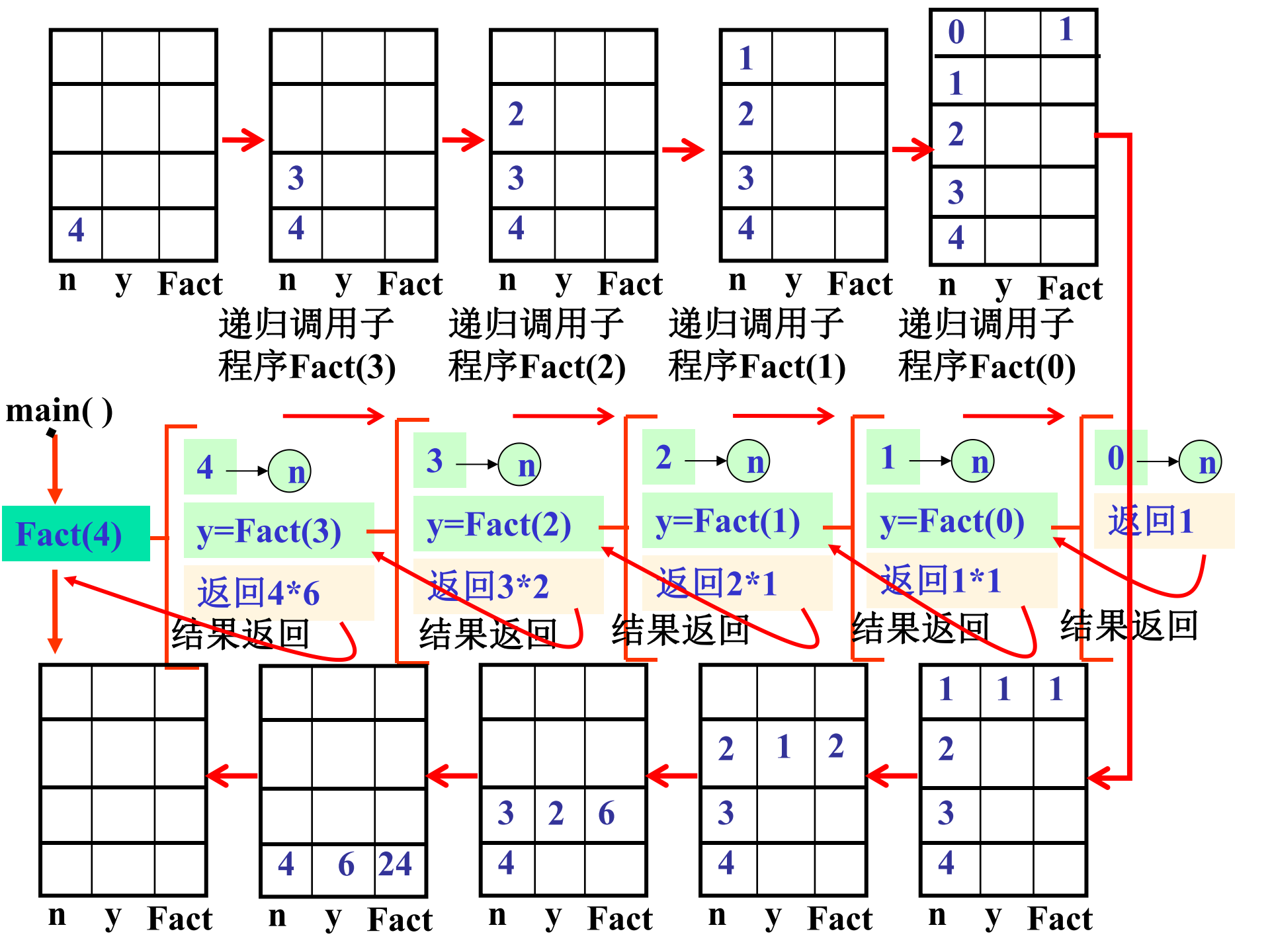
活动记录框架


递归工作记录

```
long Fact(int n)
{ long y;
   if (n<0) reurn -1;
   if (n==0)  return 1;
   else
    {    y=Fact(n-1);
         return n*y;
     }
}
```

```
main( )
{ long fn=Fact(4);
   cout<<fn<<endl ;
}
```




n y Fact
递归调用子程序Fact(3)
递归调用子程序Fact(2)
递归调用子程序Fact(1)
递归调用子程序Fact(0)
main( )
Fact(4)
y=Fact(3)
y=Fact(2)
y=Fact(1)
y=Fact(0)
返回1
返回1*1
返回2*1
返回3*2
返回4*6
结果返回

适宜于用递归算法求解的问题的充分必要条件是：

（1）问题可借用类同自身的子问题进行描述；

（2）某一有限步的子问题（也称作本原问题）有直接的解存在。

当一个问题存在上述两个基本要素时，该问题的递归算法的设计方法是：

（1）把对原问题的求解设计成包含有对子问题求解的形式；

（2）设计递归出口。

- 递归的价值在于，许多应用问题都可简洁而准确地描述为递归形式。
- 递归也是一种基本而典型的算法设计模式。
  - 可以对实际问题中反复出现的结构和形式做高度概括，并从本质层面加以描述与刻画，进而导出高效的算法。
  - 递归模式能够统筹纷繁多变的具体情况，避免复杂的分支以及嵌套的循环，从而更为简明地描述和实现算法，减少代码量，提高算法的可读性，保证算法的整体效率。
- 但是在追求更高效率的场合，应尽可能避免递归。可以通过显式地模拟调用栈的运转过程实现，精细裁剪栈中各帧内容，可以尽可能减低空间复杂度的常系数。

# 自 学

1、P90 程序3.4 (1,5) overflowProcess()、operator<<(ostream&,SeqStack<T>)

2、P92 程序3.5(1,2) push()、pop()

3、P93 程序3.6 (2,6,7)makeEmpty()、getSize()（有错误，找到错误处修改正确）、operator<<(ostream&,LinkedStack<T>)

4、P94-95 3.1.4章节 栈的应用之一——括号匹配

5、P106-109 用栈实现递归过程的非递归算法

6、P109-1143.2.3节 用回溯法求解迷宫问题

# 作业4

- 概念题：**P131 3.3、3.6。**

# 本堂课要点总结

- 栈
  - 栈的定义*、特征*、抽象基类
  - 顺序栈类及实现**SeqStack.h**
  - 单链栈类及实现**LinkedStack.h**
  - 栈的应用*
    - 计算算术表达式的值
      - 计算后缀表达式的值
      - 中缀表达式转为后缀表达式