# 数据结构与算法（三）

张铭 主讲

采用教材：张铭，王腾蛟，赵海燕 编写
高等教育出版社，2008. 6（“十一五”国家级规划教材）

http://www.jpk.pku.edu.cn/pkujpk/course/sjjg

# 第3章　栈与队列

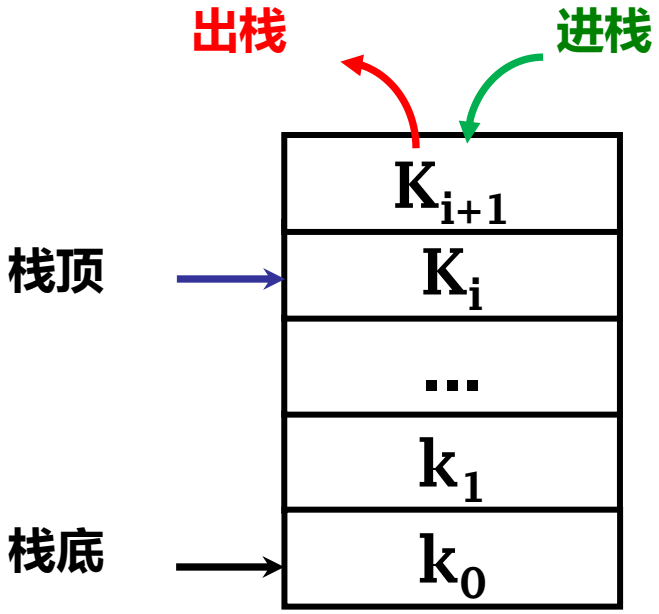
- **栈**
  - 栈与表达式求值
- **队列**
- **栈和队列的应用**
  - 队列的应用
  - 递归到非递归的转换（补充）

# 操作受限的线性表

- **栈 (Stack)**
  - 运算只在表的一端进行
- **队列 (Queue)**
  - 运算只在表的两端进行

# 栈定义

- **后进先出 (Last In First Out)**
  - 一种限制访问端口的线性表
- **主要操作**
  - 进栈 (push)　出栈 (pop)
- **应用**
  - 表达式求值
  - 消除递归
  - 深度优先搜索

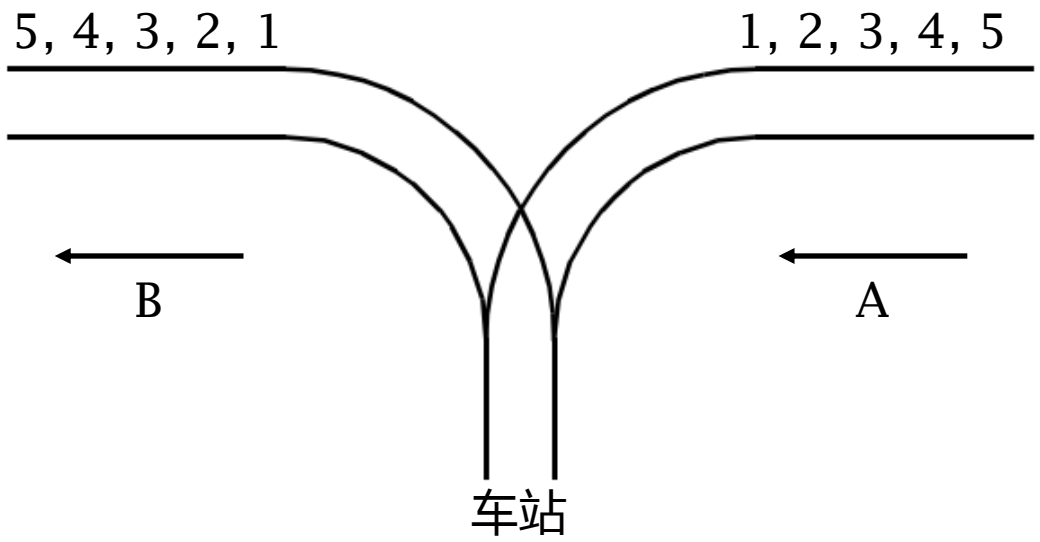


# 栈的抽象数据类型

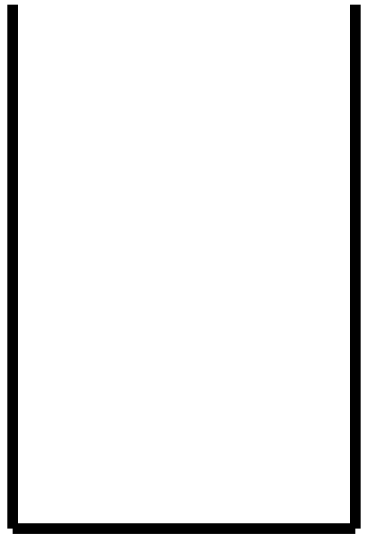
```
template <class T>
class Stack {
public:                          // 栈的运算集
    void clear();                // 变为空栈
    bool push(const T item);
                                 // item入栈，成功返回真，否则假
    bool pop(T& item);           // 返回栈顶内容并弹出，成功返回真，否则假
    bool top(T& item);           // 返回栈顶但不弹出，成功返回真，否则假
    bool isEmpty();              // 若栈已空返回真
    bool isFull();               // 若栈已满返回真
};
```

# 火车进出栈问题

- 判断火车的出栈顺序是否合法
  - http://poj.org/problem?id=1363
- 编号为1,2,…,n的n辆火车依次进站，给定一个n的排列，判断是否是合法的出站顺序？



# 利用合法的重构找冲突

# 思考

- 若入栈的顺序为1,2,3,4，
  那么出栈的顺序可以有哪些?

- 从初始输入序列1，2，…，n，希望利用一个栈得到输出序列$p_1$，$p_2$，…，$p_n$ (它们是1， 2，…，n的一种排列)。若存在下标i，j，k，满足i<j<k 同时 $P_j<P_k<P_i$，则输出序列是否合法？

# 栈的实现方式

- **顺序栈 (Array-based Stack)**
  - 使用向量实现，本质上是顺序表的简化版
    - 栈的大小
  - 关键是确定**哪一端**作为栈顶
  - 上溢，下溢问题

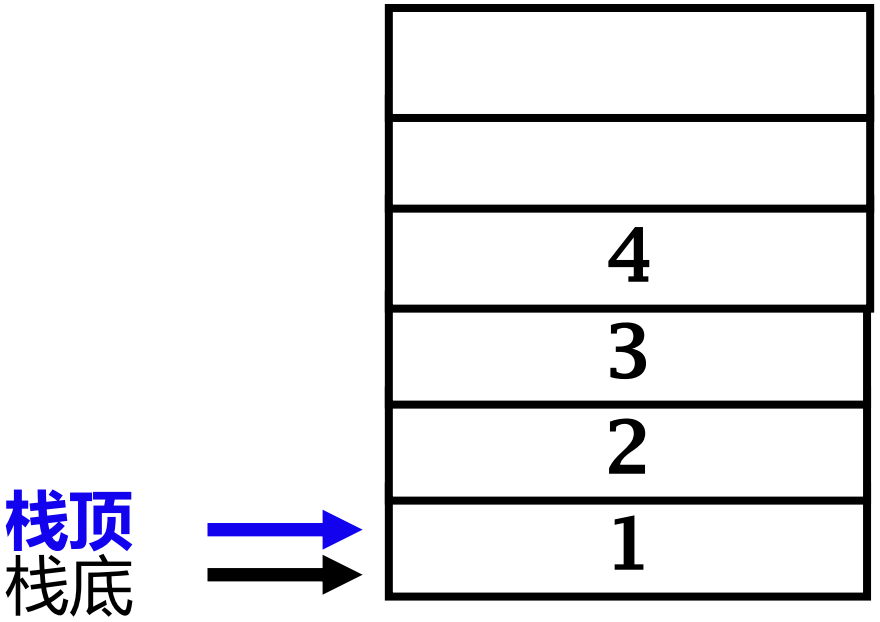
- **链式栈 ( Linked Stack )**
  - 用单链表方式存储，其中指针的方向是从栈顶向下链接

# 顺序栈的类定义

```
template <class T> class arrStack : public Stack <T> {
private:                                  // 栈的顺序存储
    int mSize;                            // 栈中最多可存放的元素个数
    int top;                              // 栈顶位置，应小于mSize
    T *st;                                // 存放栈元素的数组
public:                                   // 栈的运算的顺序实现
    arrStack(int size) {                  // 创建一个给定长度的顺序栈实例
        mSize = size; top = -1; st = new T[mSize];
    }
    arrStack() {                          // 创建一个顺序栈的实例
        top = -1;
    }
    ~arrStack() { delete [] st; }
    void clear() { top = -1; }  // 清空栈
}
```

# 顺序栈

- 按压入先后次序，最后压入的元素编号为4，然后依次为3,2,1



# 顺序栈的溢出

- **上溢 (Overflow)**
  - 当栈中已经有maxsize个元素时，如果再做进栈运算，所产生的现象
- **下溢 (Underflow)**
  - 对空栈进行出栈运算时所产生的现象

# 压入栈顶

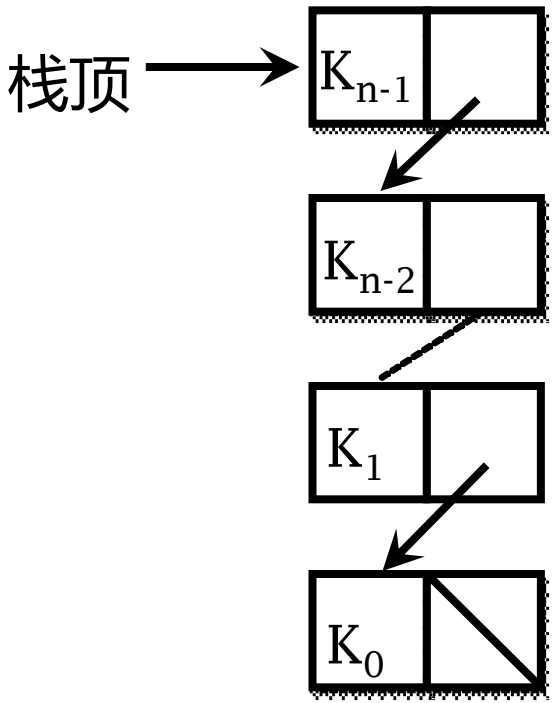
```
bool arrStack<T>::push(const T item) {
    if (top == mSize-1) {        // 栈已满
        cout << "栈满溢出" << endl;
        return false;
    } else {                     // 新元素入栈并修改栈顶指针
        st[++top] = item;
        return true;
    }
}
```

# 从栈顶弹出

```
bool arrStack<T>::pop(T & item) {   // 出栈
    if (top == -1) {                // 栈为空
        cout << "栈为空，不能执行出栈操作"<< endl;
        return false;
    } else {
        item = st[top--];   // 返回栈顶，并缩减1
        return true;
    }
}
```

# 链式栈的定义

- 用单链表方式存储
- 指针的方向从**栈顶向下**链接



# 链式栈的创建

```
template <class T> class lnkStack : public Stack <T> {
private:                                    // 栈的链式存储
    Link<T>* top;                           // 指向栈顶的指针
    int size;                               // 存放元素的个数
public:                                     // 栈运算的链式实现
    lnkStack(int defSize) {                 // 构造函数
        top = NULL; size = 0;
    }
    ~lnkStack() {                           // 析构函数
        clear();
    }
}
```

# 压入栈顶

```
// 入栈操作的链式实现
bool lnksStack<T>:: push(const T item) {
    Link<T>* tmp = new Link<T>(item, top);
    top = tmp;
    size++;
    return true;
}

Link(const T info, Link* nextValue)  {// 具有两个参数的Link构造函数
        data = info;
        next = nextValue;
}
```

# 从单链栈弹出元素

```
// 出栈操作的链式实现
bool lnkStack<T>:: pop(T& item) {
    Link <T> *tmp;
    if (size == 0) {
        cout << "栈为空，不能执行出栈操作"<< endl;
        return false;
    }
    item = top->data;
    tmp = top->next;
    delete top;
    top = tmp;
    size--;
    return true;
}
```

# 顺序栈和链式栈的比较

- **时间效率**
  - 所有操作都只需常数时间
  - 顺序栈和链式栈在时间效率上难分伯仲

- **空间效率**
  - 顺序栈须说明一个固定的长度
  - 链式栈的长度可变，但增加结构性开销

# 顺序栈和链式栈的比较

- 实际应用中，顺序栈比链式栈用得更广泛
  - 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$O(1)$，而链式栈则需要沿着指针链游走，显然慢些，读取第$k$个元素需要时间为$\mathbf{O}(k)$

- 一般来说，栈不允许“读取内部元素”，只能在栈顶操作

# 思考： STL中关于栈的函数

- top函数表示取栈顶元素，将结果返回给用户
- pop函数表示将栈顶元素弹出（如果栈不空）
  - pop函数仅仅是一个操作，并不将结果返回。
  - pointer = aStack.pop()？ **错误！**

- STL为什么这两个操作分开？ 为什么不提供ptop？

# 栈的应用

- 栈的特点：**后进先出**
  - 体现了元素之间的透明性
- 常用来处理具有递归结构的数据
  - 深度优先搜索
  - **表达式求值**
  - 子程序 / 函数调用的管理
  - **消除递归**

# 计算表达式的值

- 表达式的递归定义
  - 基本符号集：{0，1，...，9，+，-，*，/，（，）}
  - 语法成分集：{<表达式>，<项>，<因子>，<常数>，<数字>}
- 中缀表达式　　23+(34*45)/(5+6+7)
- 后缀表达式　　23 34 45 * 5 6 + 7 + / +

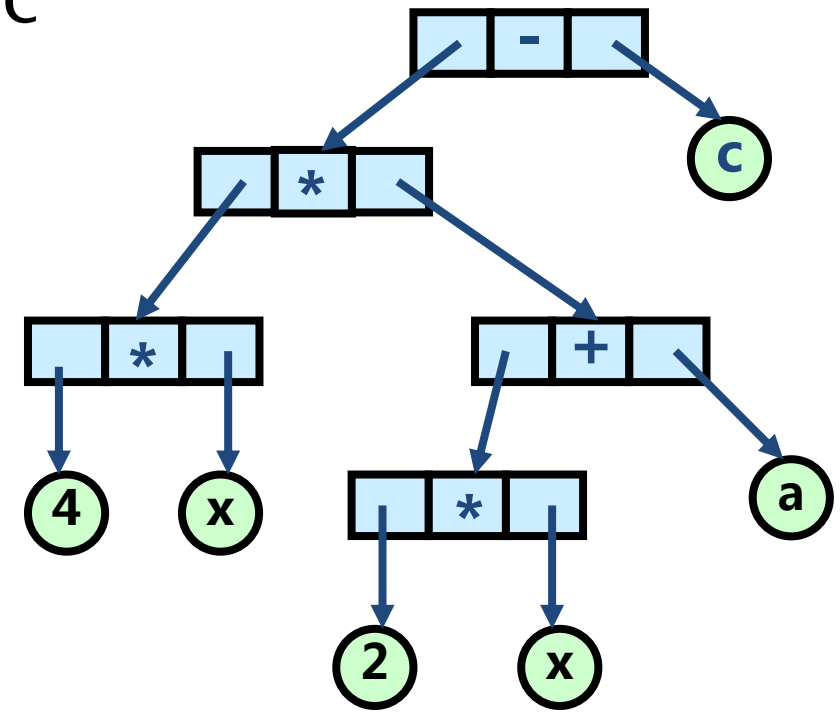
# 中缀表达式

- **中缀表达式**

  4 * x * (2 * x + a) – c

  – 运算符在中间

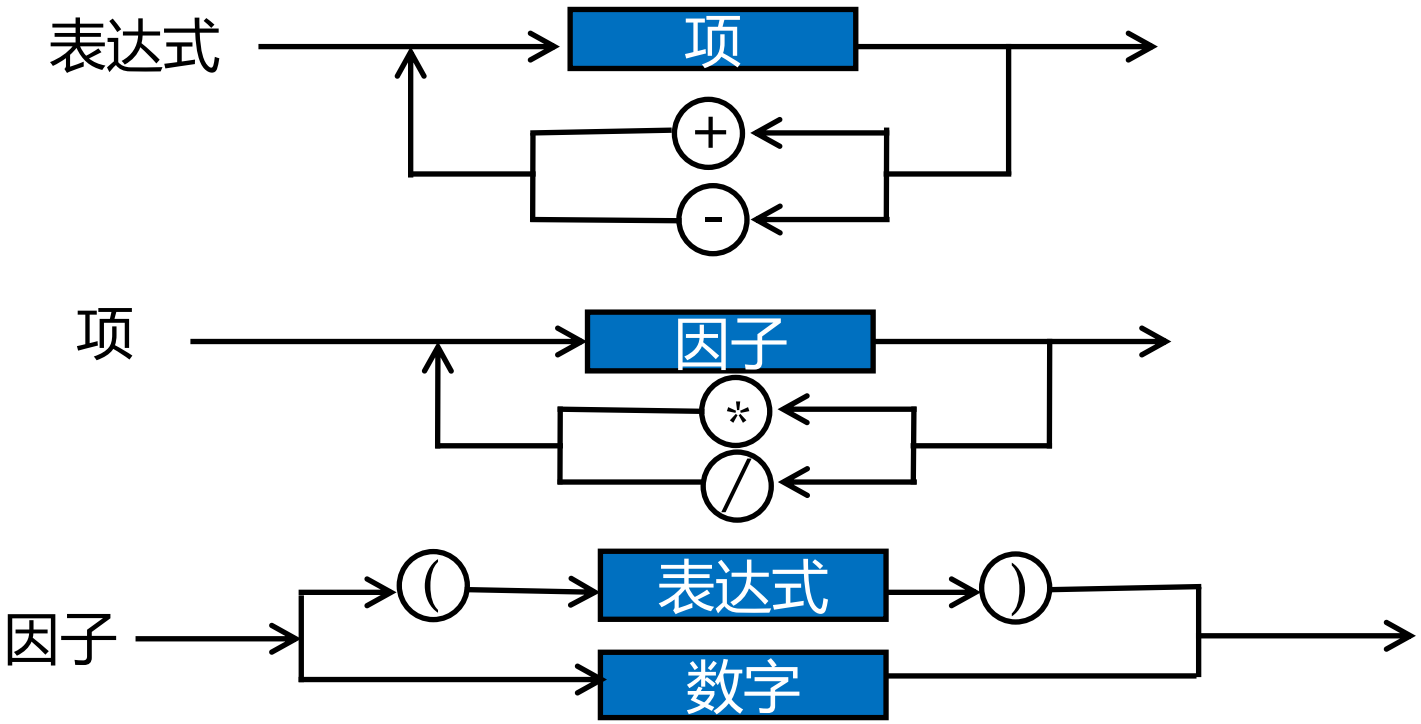
  – 需要括号改变优先级



# 中缀表达法的语法公式

<表达式> ::= <项> + <项>
| <项> - <项>
| <项>
<项> ::= <因子> * <因子>
| <因子> / <因子>
| <因子>
<因子> ::= <常数>
| ( <表达式> )
<常数> ::= <数字>
| <数字> <常数>
<数字> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

# 表达式的递归图示

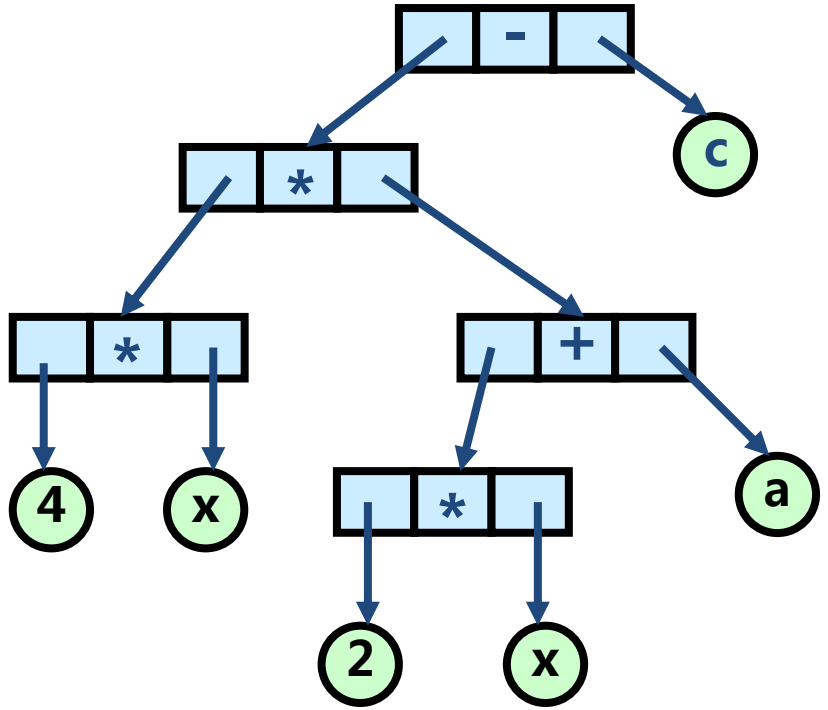


# 后缀表达式

- **后缀表达式**

  4 x * 2 x * a + * c -

  - 运算符在后面
  - 完全不需要括号



# 后缀表达式

<表达式> ::= <项> <项> +
| <项> <项> -
| <项>
<项> ::= <因子> <因子> *
| <因子> <因子> /
| <因子>
<因子> ::= <常数>
<常数> ::= <数字>
| <数字> <常数>
<数字> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

# 后缀表达式的计算

- 23 34 45 * 5 6 + 7 + / + = ?

计算特点？

中缀和后缀表达式的主要异同？

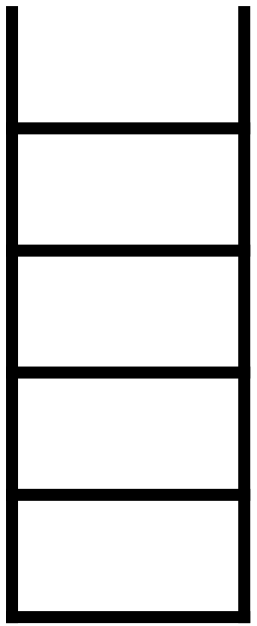
23 + 34 * 45 / ( 5 + 6 + 7 ) = ?

23 34 45 * 5 6 + 7 + / + = ?

**待处理后缀表达式：**

23 34 45 * 5 6 + 7 + / +

**栈状态的变化**



| | |
|---|---|
| | 1180780 |

计算　　结果

# 后缀表达式求值

- 循环：依次顺序读入表达式的符号序列（假设以 = 作为输入序列的结束），并根据读入的元素符号逐一分析
  1. 当遇到的是一个操作数，则压入栈顶
  2. 当遇到的是一个运算符，就从栈中两次取出栈顶，按照运算符对这两个操作数进行计算。然后将计算结果压入栈顶
- 如此继续，直到遇到符号 = ，这时栈顶的值就是输入表达式的值

# 后缀计算器的类定义

```
class Calculator {
private:
    Stack<double> s;              // 这个栈用于压入保存操作数
    // 从栈顶弹出两个操作数opd1和opd2
    bool GetTwoOperands(double& opd1,double& opd2);
    // 取两个操作数，并按op对两个操作数进行计算
    void Compute(char op);
public:
    Calculator(void){} ;          // 创建计算器实例，开辟一个空栈
    void Run(void);               // 读入后缀表达式，遇“=”符号结束
    void Clear(void);             // 清除计算器，为下一次计算做准备
};
```

# 后缀计算器的类定义

```
template <class ELEM>
bool Calculator<ELEM>::GetTwoOperands(ELEM& opnd1, ELEM& opnd2) {
    if (S.IsEmpty()) {
        cerr << "Missing operand!" <<endl;
        return false;
    }
    opnd1 = S.Pop(); // 右操作数
    if (S.IsEmpty()) {
        cerr << "Missing operand!" <<endl;
        return false;
    }
    opnd2 = S.Pop(); // 左操作数
    return true;
}
```

# 后缀计算器的类定义

```
template <class ELEM> void Calculator<ELEM>::Compute(char op) {
    bool result;  ELEM operand1, operand2;
    result = GetTwoOperands(operand1, operand2);
    if (result == true)
        switch(op) {
            case '+' : S.Push(operand2 + operand1); break;
            case '-' : S.Push(operand2 - operand1); break;
            case '*' : S.Push(operand2 * operand1); break;
            case '/' : if (operand1 == 0.0) {
                       cerr << "Divide by 0!" << endl;
                       S.ClearStack();
                   } else S.Push(operand2 / operand1);
                   break;
        }
    else S.ClearStack();
}
```

# 后缀计算器的类定义

```
template <class ELEM> void Calculator<ELEM>::Run(void) {
    char c; ELEM newoperand;
    while (cin >> c, c != '=') {
        switch(c) {
            case '+': case '-': case '*': case '/':
                Compute(c);
                break;
            default:
                cin.putback(c); cin >> newoperand;
                Enter(newoperand);
                break;
        }
    }
    if (!S.IsEmpty())
        cout << S.Pop() << endl;          // 印出求值的最后结果
}
```

# 思考

- 1. 栈往往用单链表实现。可以用双链表吗？哪个更好？

- 2. 请总结前缀表达式的性质，以及求值过程。

# 第3章　栈与队列

- 栈
  - 栈与表达式求值
- **队列**
- 栈和队列的应用
  - 队列的应用
  - 递归到非递归的转换（补充）

# 队列的定义

- **先进先出（First In First Out）**
  - 限制访问点的线性表
    - 按照到达的顺序来释放元素
    - 所有的插入在表的一端进行，所有的删除都在表的另一端进行
- **主要元素**
  - 队头（front）
  - 队尾（rear）

# 队列的主要操作

- 入队列（enQueue）
- 出队列（deQueue）
- 取队首元素（getFront）
- 判断队列是否为空（isEmpty）

# 队列的抽象数据类型

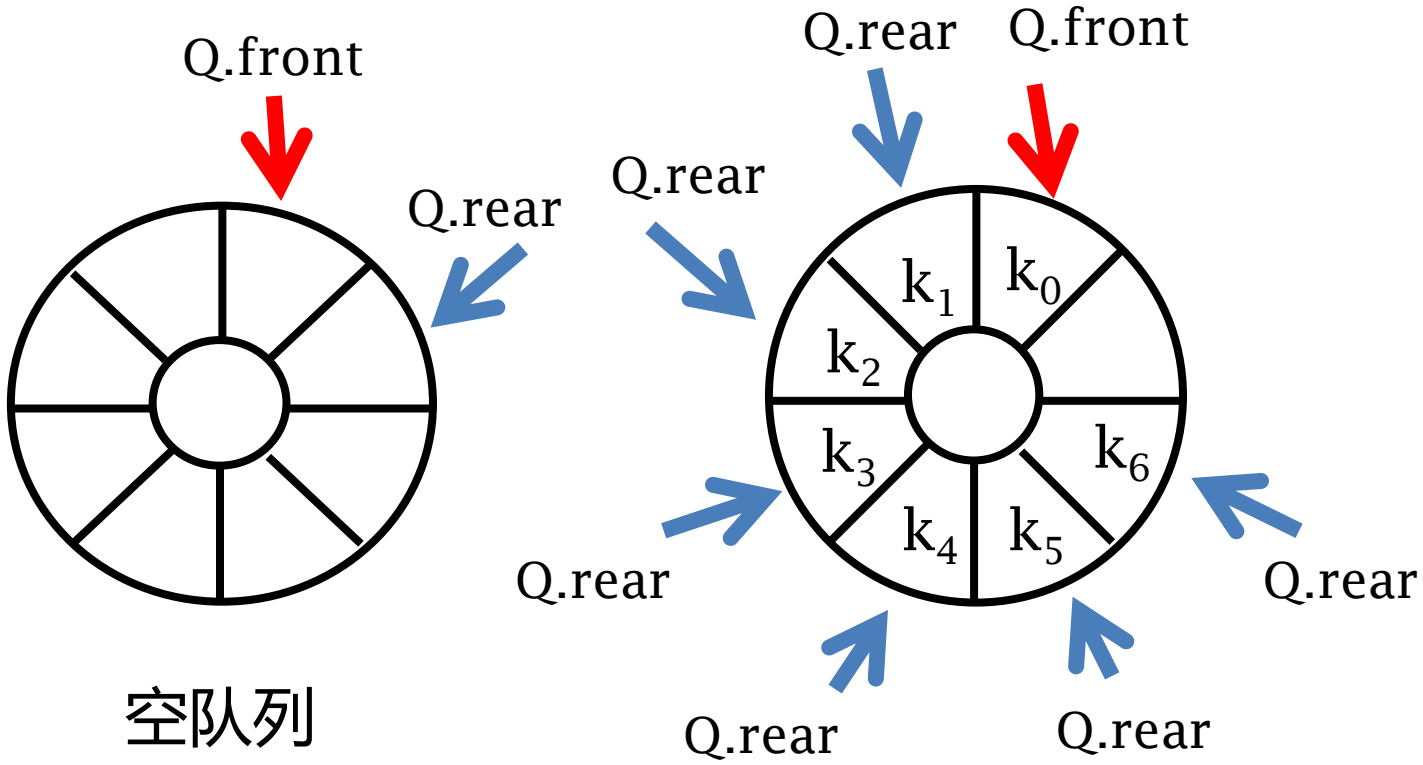
```
template <class T> class Queue {
public:                  // 队列的运算集
    void clear();        // 变为空队列
    bool enQueue(const T item);
                         // 将item插入队尾，成功则返回真，否则返回假
    bool deQueue(T & item) ;
                         // 返回队头元素并将其从队列中删除，成功则返回真
    bool getFront(T & item);
                         // 返回队头元素，但不删除，成功则返回真
    bool isEmpty(); // 返回真，若队列已空
    bool isFull();  // 返回真，若队列已满
};
```

# 队列的实现方式

- **顺序队列**
  - **关键**是如何防止假溢出

- **链式队列**
  - 用单链表方式存储，队列中每个元素对于链表中的一个结点

# 队列示意：环形(实指)

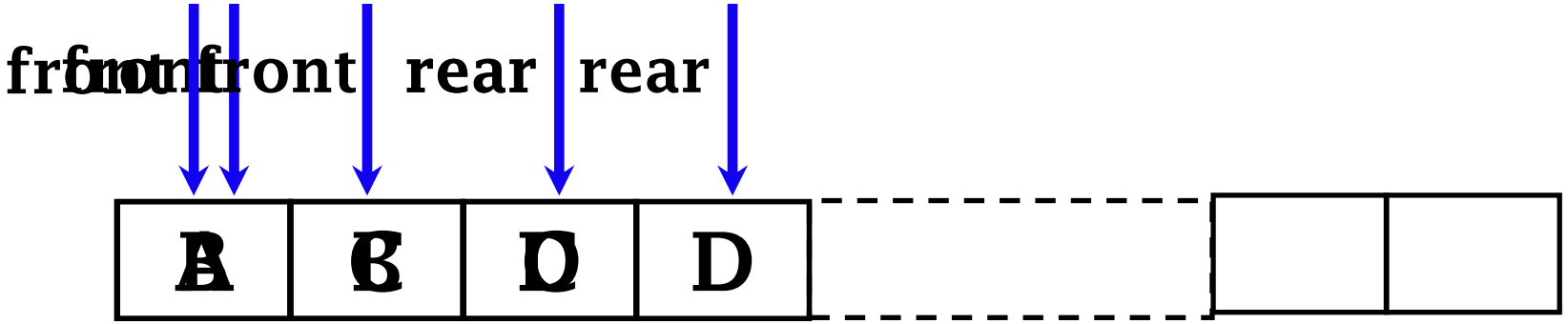

Q.front
Q.rear
空队列
Q.rear
Q.front
Q.rear
Q.rear
Q.rear
Q.rear
Q.rear
$k_1$
$k_0$
$k_2$
$k_3$
$k_6$
$k_4$
$k_5$

# 顺序队列的类定义

```
class arrQueue: public Queue<T> {
private:
    int mSize;              // 存放队列的数组的大小
    int front;              // 表示队头所在位置的下标
    int rear;               // 表示队尾所在位置的下标
    T * qu;                 // 存放类型为T的队列元素的数组
public:                     // 队列的运算集
    arrQueue(int size);     // 创建队列的实例
    ~arrQueue();            // 消除该实例，并释放其空间
}
```

# 顺序队列的维护

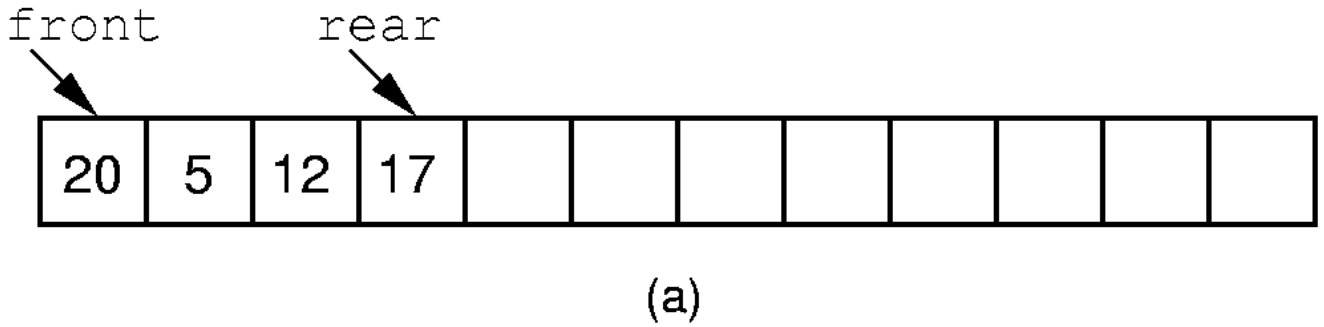
- rear实指



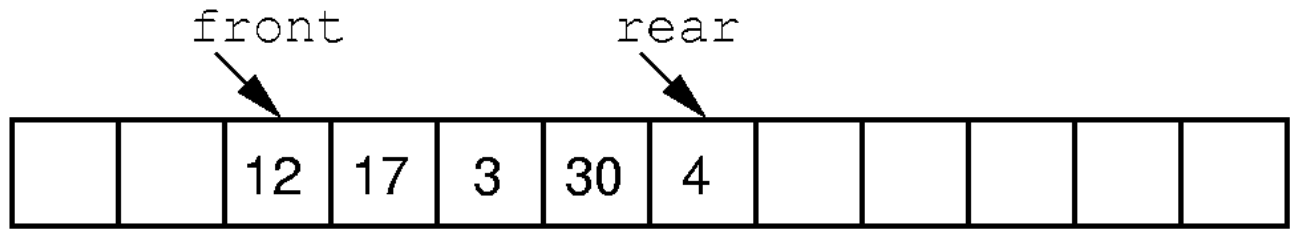

删除

# 顺序队列的维护

- front和rear都实指




(a)




(b)

# 顺序队列代码实现

```
template <class Elem> class Aqueue : public Queue<Elem> {
private:
    int size;                       // 队列的最大容量
    int front;                      // 队首元素指针
    int rear;                       // 队尾元素指针
    Elem *listArray;                // 存储元素的数组
public:
    AQueue(int sz=DefaultListSize) {
        // 让存储元素的数组多预留一个空位
        size = sz+1;                // size数组长，sz队列最大长度
        rear = 0; front = 1;        // 也可以rear=-1; front=0
        listArray = new Elem[size];
    }
    ~AQueue() { delete [] listArray; }
    void clear() { front = rear+1; }
```

# 顺序队列代码实现

```
bool enqueue(const Elem& it) {
    if (((rear+2) % size) == front) return false;
                                // 还剩一个空位，就要报满
    rear = (rear+1) % size; // 因为实指，需要先移动到下一个空位
    listArray[rear] = it;
    return true;
}
bool dequeue(Elem& it) {
    if (length() == 0) return false;
                                    // 队列为空
    it = listArray[front];          // 先出队，再移动front下标
    front = (front+1) % size;       // 环形增加
    return true;
}
```

# 顺序队列代码实现

```
bool frontValue(Elem& it) const {
    if (length() == 0)
        return false;       // 队列为空
    it = listArray[front]; return true;
}
int length() const {
    return (size +(rear - front + 1)) % size;
}
```

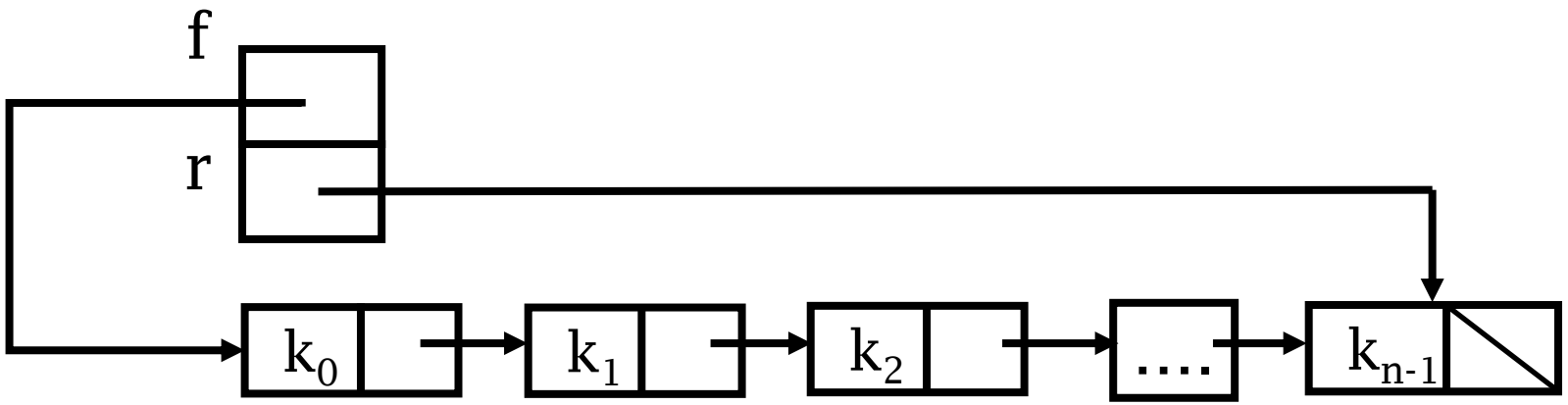
# 思考

- 1. 只是用 front, rear 两个变量，长度为 mSize = n 的队列，可以容纳的最大元素个数为多少？ 请给出详细的推导过程。

- 2. 如果不愿意浪费队列的存储单元，还可以采用什么方法？

# 链式队列的表示

- 单链表队列
- 链接指针的方向是从队列的前端向尾端链接



# 链式队列的类定义

```
template <class T>
class lnkQueue: public Queue<T> {
private:
    int size;                       // 队列中当前元素的个数
    Link<T>* front;                 // 表示队头的指针
    Link<T>* rear;                  // 表示队尾的指针
public:                             // 队列的运算集
    lnkQueue(int size);             // 创建队列的实例
    ~lnkQueue();                    // 消除该实例，并释放其空间
}
```

# 链式队列代码实现

```
bool enQueue(const T item) { //  item入队，插入队尾
    if (rear == NULL) {       // 空队列
        front = rear = new Link<T> (item, NULL);
    }
    else  {                   // 添加新的元素
        rear-> next = new Link<T> (item, NULL);
        rear = rear ->next;
    }
    size++;
    return true;
}
```

# 链式队列代码实现

```
bool deQueue(T* item)  {              // 返回队头元素并从队列中删除
    Link<T> *tmp;
    if (size == 0)  {                 // 队列为空，没有元素可出队
        cout << "队列为空" << endl;
        return false;
    }
    *item = front->data;
    tmp = front;
    front = front -> next;
    delete tmp;
    if (front == NULL)
        rear = NULL;
    size--;
    return true;
}
```

# 顺序队列与链式队列的比较

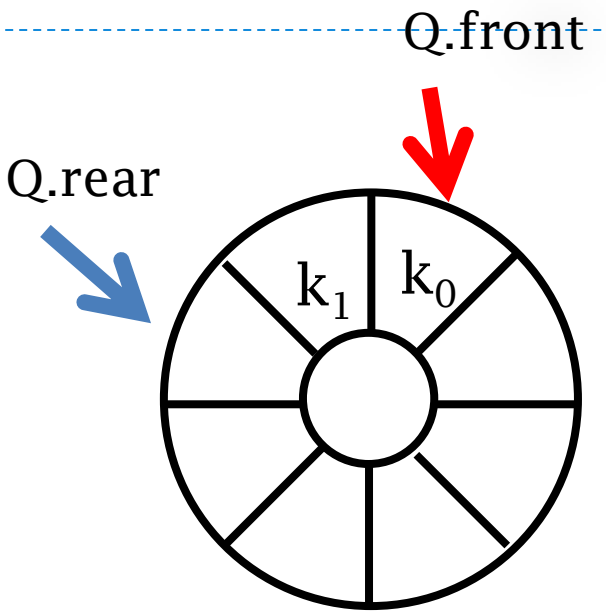
- **顺序队列**
  - 固定的存储空间
- **链式队列**
  - 可以满足大小无法估计的情况

都不允许访问队列内部元素

# 队列的应用

- 只要满足先来先服务特性的应用均可采用队列作为其数据组织方式或中间数据结构
- 调度或缓冲
  - 消息缓冲器
  - 邮件缓冲器
  - 计算机硬设备之间的通信也需要队列作为数据缓冲
  - 操作系统的资源管理
- 宽度优先搜索

# 思考




- 链式队列往往用单链表。
  为什么不用双链表来实现？

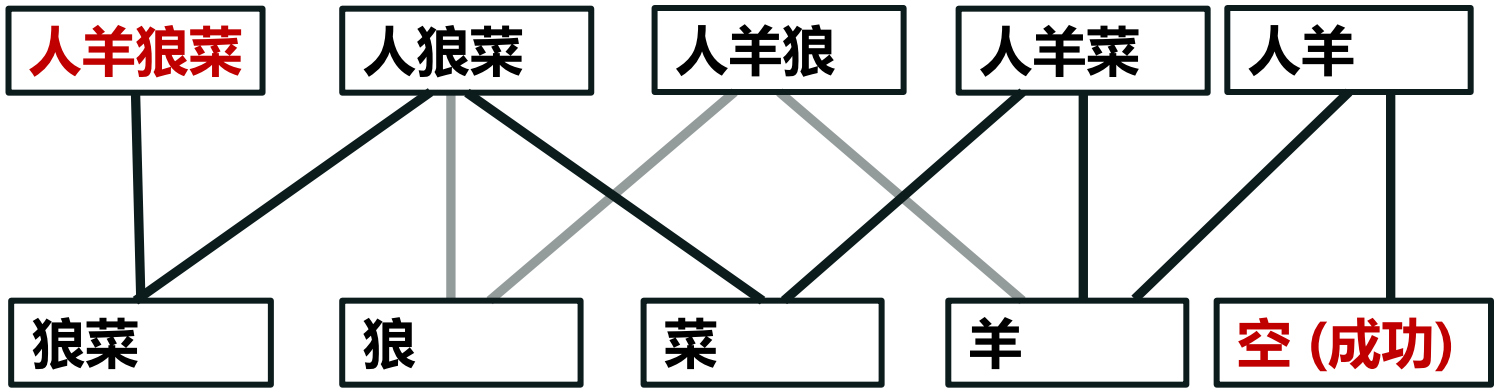
- 若采用虚指方法实现队尾指针(rear指向队尾元素后一个元素，和实指相比后移一位)，在具体实现上有何异同？哪一种更好？

# 第3章　栈与队列

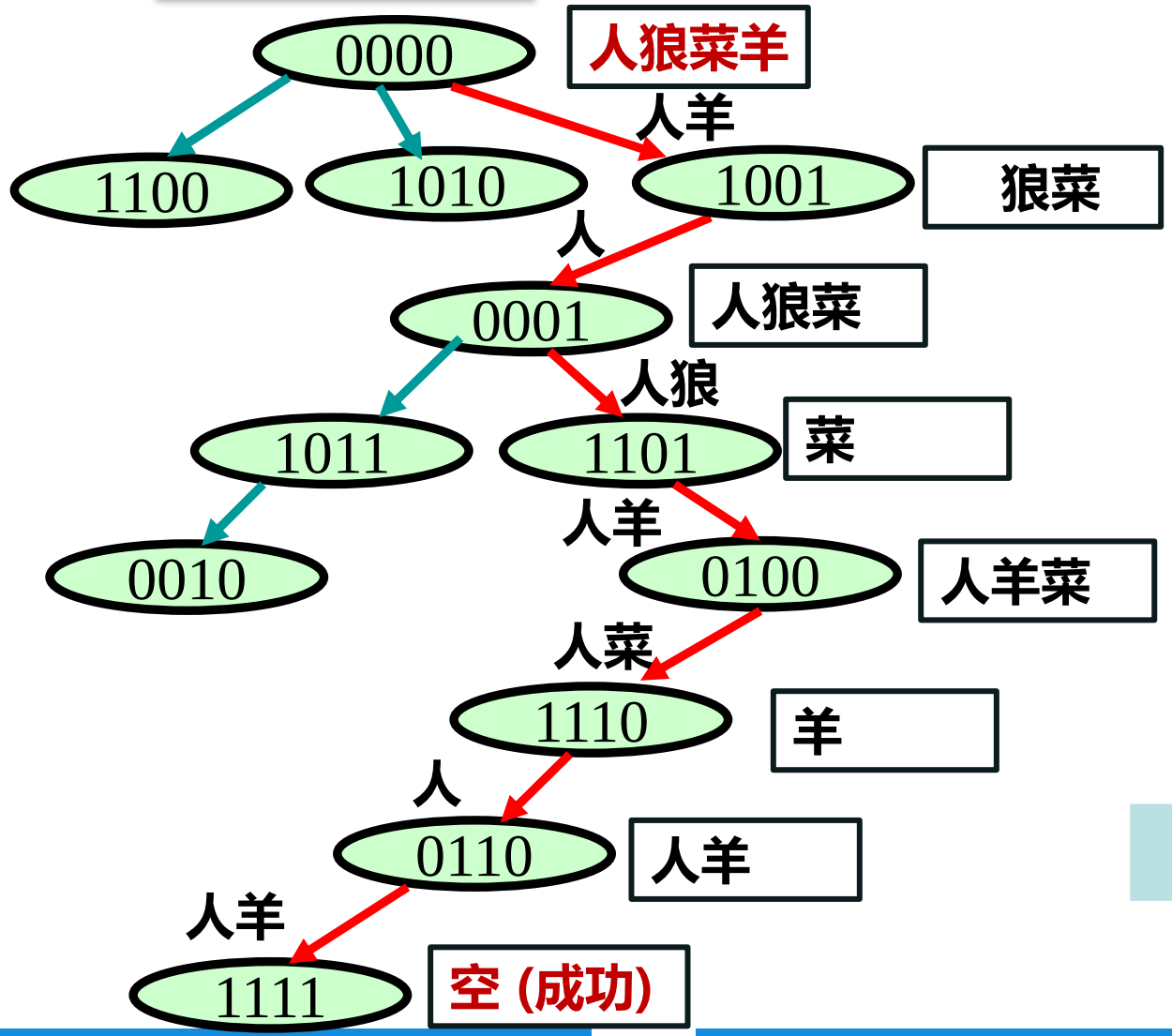
- 栈
  - 栈与表达式求值
- 队列
- **栈和队列的应用**
  - **队列的应用**
  - 递归到非递归的转换（补充）

# 农夫过河问题

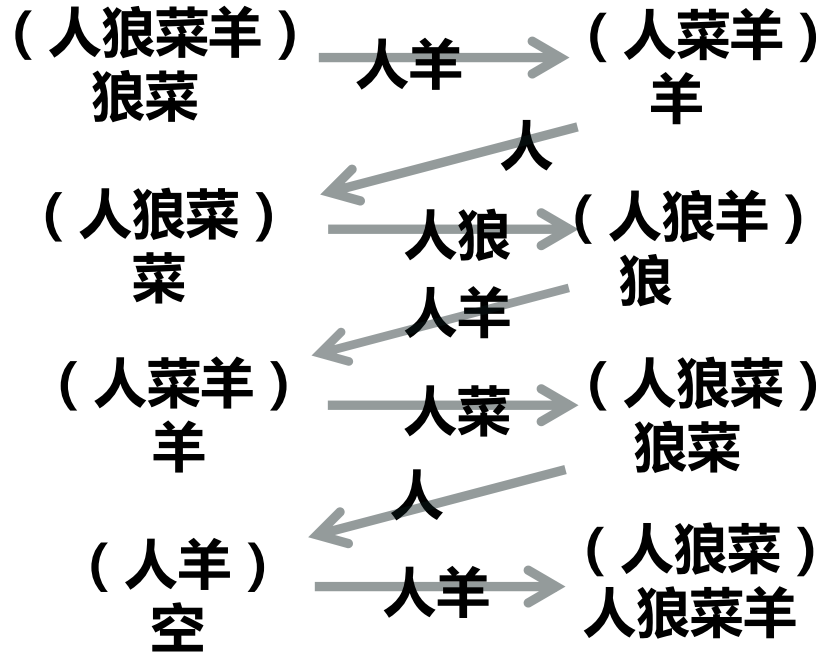

- **问题抽象**：“人狼羊菜”乘船过河
  - 只有人能撑船，船只有两个位置（包括人）
  - 狼羊、羊菜不能在没有人时共处



# 队列的应用



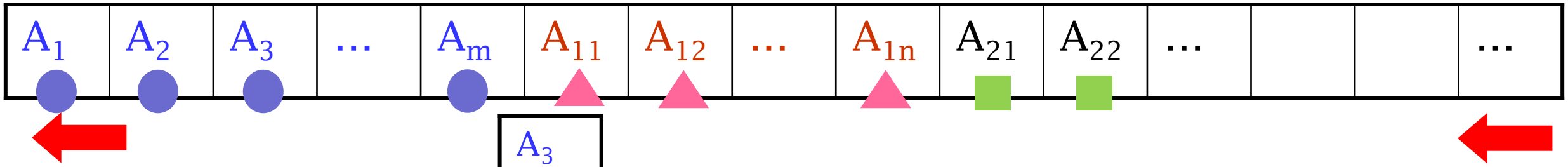

## 算法示意



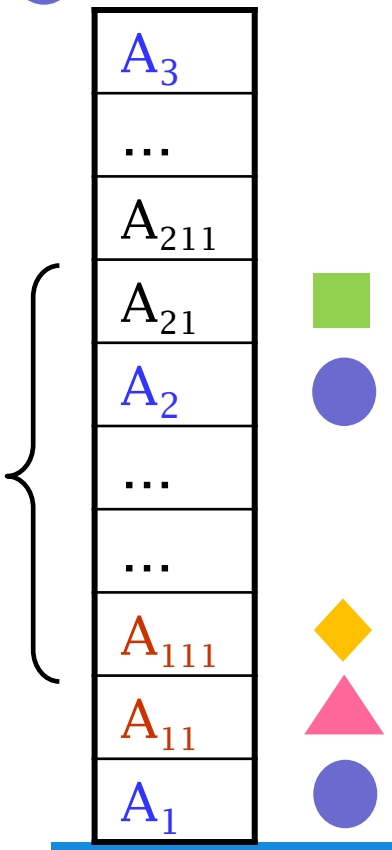

| 0 | 1 | 0 | 1 |
|---|---|---|---|
| 人 | 狼 | 菜 | 羊 |

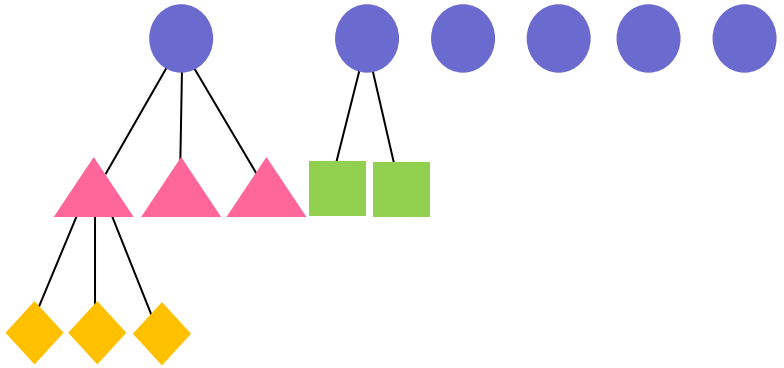
## 问题分析

**广度优先：(m 种状态 )**




**深度优先：(m 种状态 )**



# 数据抽象

- 每个角色的位置进行描述
  - 农夫、狼、菜和羊，四个目标各用一位（假定按照农夫、狼、白菜、羊次序），目标在起始岸位置：0，目标岸：1

| 0 | 1 | 0 | 1 |
|---|---|---|---|

  - 如 0101 表示农夫、白菜在起始岸，而狼、羊在目标岸（此状态为不安全状态）

# 数据的表示

- 用整数 status 表示上述四位二进制描述的状态
  - 整数 0x08 表示的状态

| 1 | 0 | 0 | 0 |
|---|---|---|---|

  - 整数 0x0F 表示的状态

| 1 | 1 | 1 | 1 |
|---|---|---|---|

- 如何从上述状态中得到每个角色所在位置？
  - 函数返回值为真（1），表示所考察人或物在目标岸
  - 否则，表示所考察人或物在起始岸

# 确定每个角色位置的函数

```
bool farmer(int status)
{  return ((status & 0x08)  != 0);      }

bool wolf(int status)
{  return ((status & 0x04)  != 0);      }

bool cabbage(int status)
{  return ((status & 0x02)  != 0);      }

bool goat(int status)
{  return ((status & 0x01)  != 0);      }
```

| 人 | 狼 | 菜 | 羊 |
|---|---|---|---|
| 1 | x | x | x |
| x | 1 | x | x |
| x | x | 1 | x |
| x | x | x | 1 |

| 人 | 狼 | 菜 | 羊 |
|---|---|---|---|
| 0 | 1 | 0 | 1 |

# 安全状态的判断

```
bool safe(int status)                    // 返回 true:安全，false:不安全
{
    if ((goat(status) == cabbage(status)) &&
        (goat(status) != farmer(status)))
            return(false);               // 羊吃白菜
    if ((goat(status) == wolf(status)) &&
        (goat(status) != farmer(status)))
            return(false);               // 狼吃羊
    return(true);                        // 其它状态为安全
}
```

# 算法抽象

- 问题变为

从状态0000（整数0）出发，寻找全部由安全状态构成的状态序列，以状态1111（整数15）为最终目标。

  - 状态序列中 每个 状态都可以从前一状态通过农夫（可以带一样东西）划船过河的动作到达。
  - 序列中不能出现 重复 状态

# 算法设计

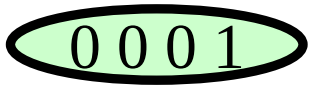
- 定义一个整数队列 moveTo，它的每个元素表示一个可以安全到达的中间状态
- 还需要定义一个数据结构 记录已被访问过的各个状态，以及已被发现的能够到达当前这个状态的路径
  - 用顺序表 route 的第 $i$ 个元素记录状态$i$是否已被访问过
  - 若 route[i] 已被访问过，则在这个顺序表元素中记入前驱状态值； -1表示未被访问
  - route 的大小（长度）为 16

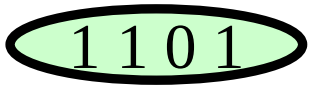
# 算法实现

```
void solve() {
    int movers, i, location, newlocation;
    vector<int> route(END+1, -1);
                        // 记录已考虑的状态路径
    queue<int> moveTo;
    // 准备初值
    moveTo.push(0x00);
    route[0]=0;
```

# 算法实现

人狼菜羊



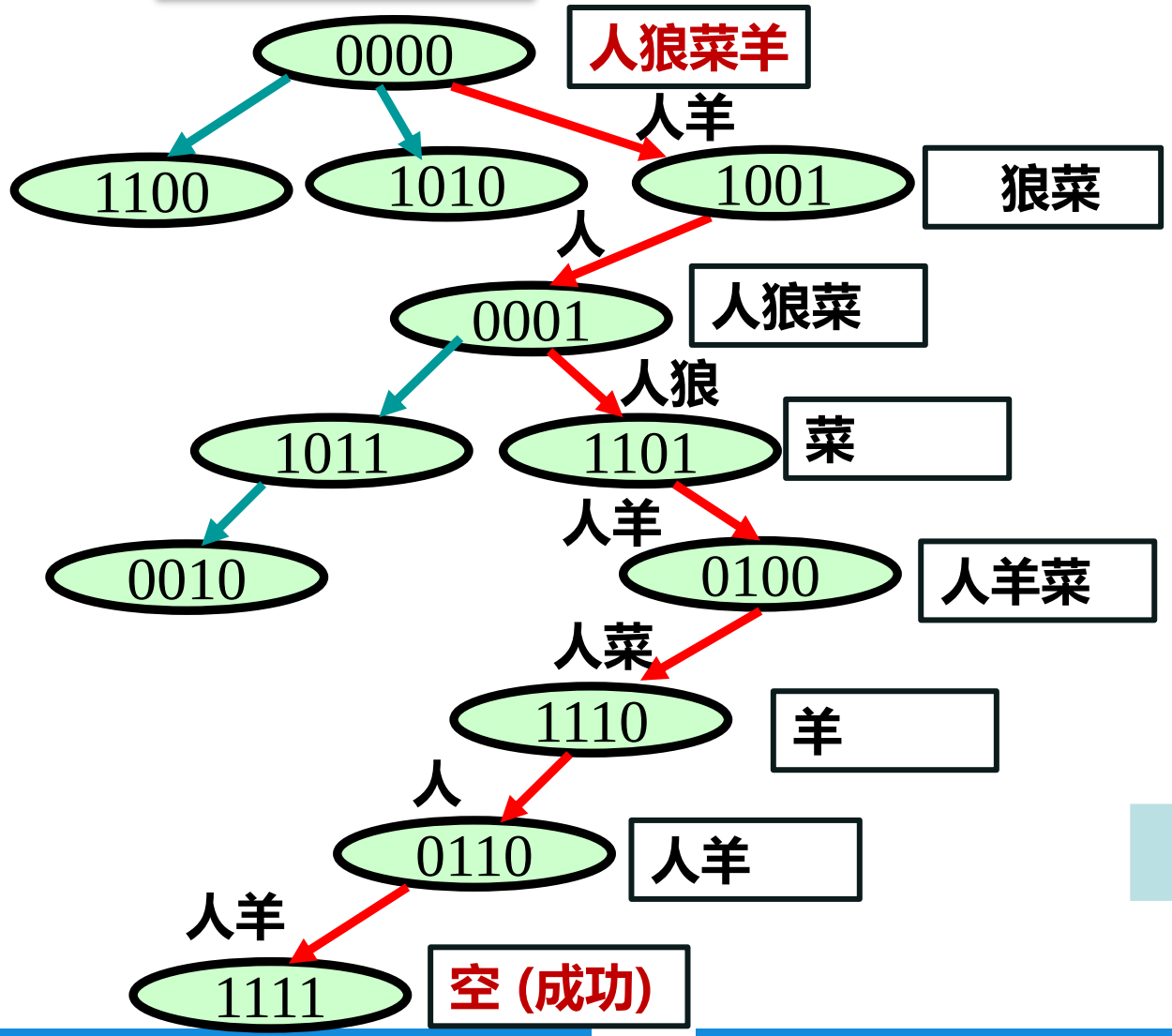




```
while (!moveTo.empty() && route[15] == -1) {
        // 得到现在的状态
        status = moveTo.front();
        moveTo.pop();
        for (movers = 1; movers <= 8; movers <<= 1) {
        // 农夫总是在移动，随农夫移动的也只能是在农夫同侧的东西
                if (farmer(status)  == (bool)(status & movers)) {
                        newstatus = status ^ (0x08 | movers);
                        // 安全的，并且未考虑过的走法
                        if (safe(newstatus) && (route[newstatus] == -1)) {
                                route[newstatus] = status;
                                moveTo.push(newstatus); }
                }
        }
}
```

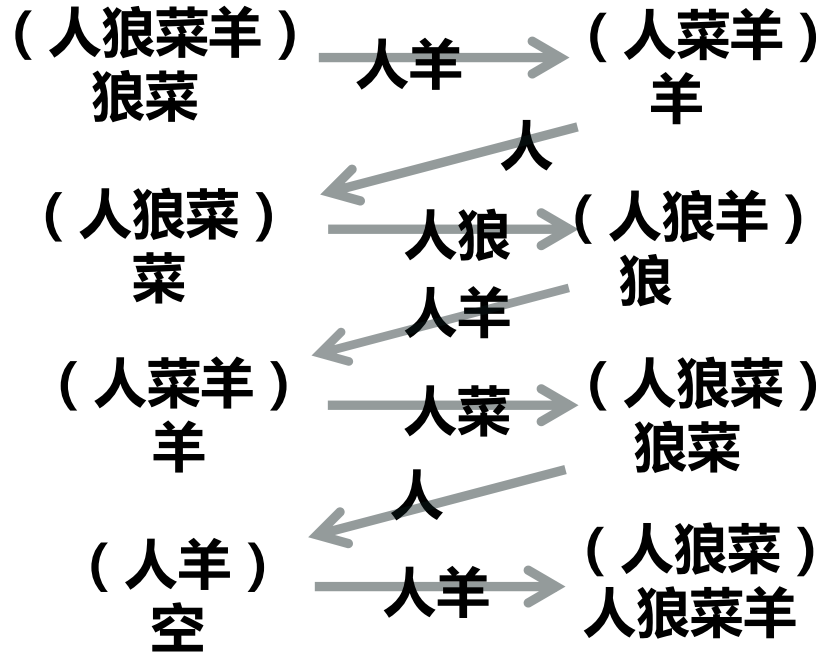
# 算法实现

```
    // 反向打印出路径
    if (route[15] != -1) {
        cout << "The reverse path is : “ << endl;
        for (int status = 15; status >= 0; status = route[status]) {
            cout << "The status is : “ << status << endl;
            if (status == 0) break;
        }
    }
    else
        cout << "No solution.“ << endl;
}
```

# 队列的应用



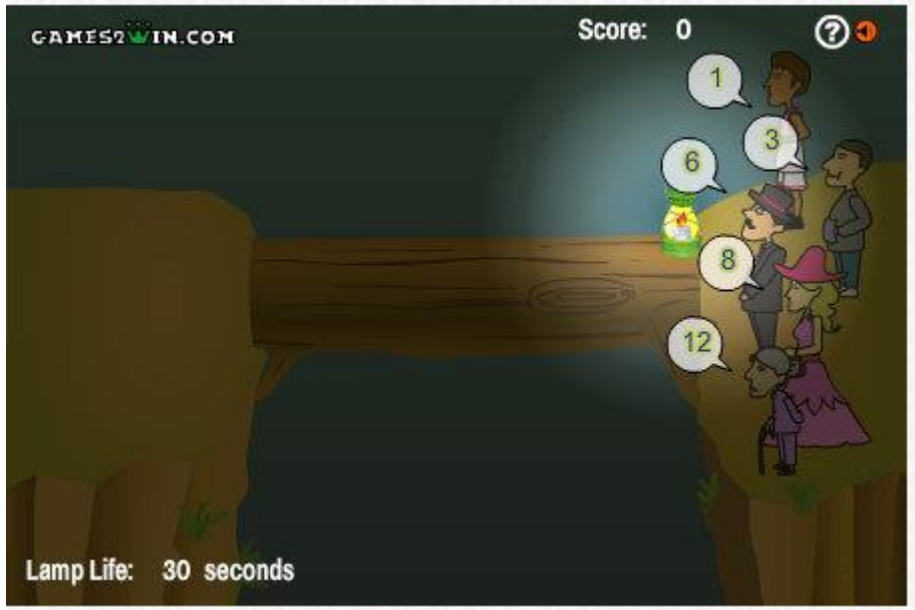

## 算法示意




| 0 | 1 | 0 | 1 |
|---|---|---|---|
| 人 | 狼 | 菜 | 羊 |

# 思考：另一个小游戏

- 五人提灯过独木桥：
  - 有一盏灯能使用30秒，要在灯熄灭前过这座桥；
  - 一家五口人每个人过桥的速度不同：哥哥 1 秒，弟弟 3 秒，爸爸 6 秒，妈妈 8 秒，奶奶 12 秒；
  - 每次只能过两个人。 过去后，对岸要有一个人再把灯送回来。



# 第3章　栈与队列

- 栈
- 队列
- **栈和队列的应用**
  - 队列的应用
  - **递归到非递归的转换（补充）**

# 递归转非递归

- 递归函数调用原理
- 机械的递归转换
- 优化后的非递归函数

# 递归的再研究

- 阶乘 $f(n)=\begin{cases} n\times f(n-1) & n\geq 1 \\ 1 & n=0 \end{cases}$
- **递归出口**
  - 递归终止的条件，即最小子问题的求解
  - 可以允许多个出口
- **递归规则**（递归体 + 界函数）
  - 将原问题划分成子问题
  - 保证递归的规模向出口条件靠拢

# 递归算法的非递归实现

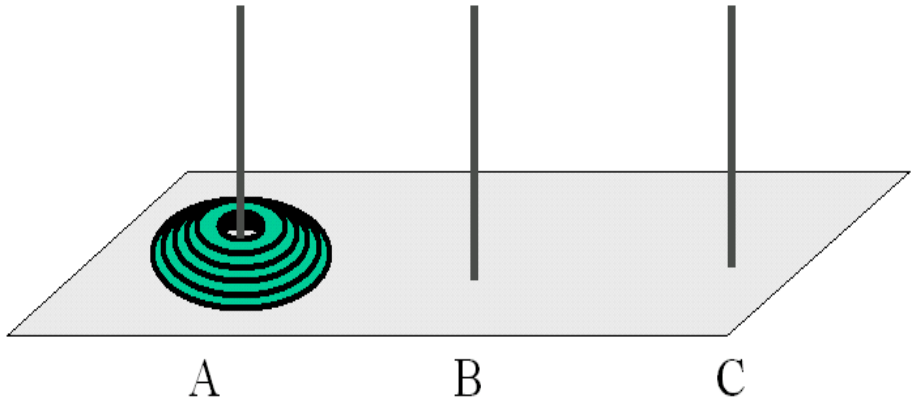
$$f(n)=\begin{cases} n\times f(n-1) & n\geq 1 \\ 1 & n=0 \end{cases}$$

- 阶乘的非递归方式
  - 建立迭代
  - 递归转换为非递归

- 以Hanoi塔为例呢？

# 河内塔问题的递归求解程序

http://www.17yy.com/f/play/89425.html

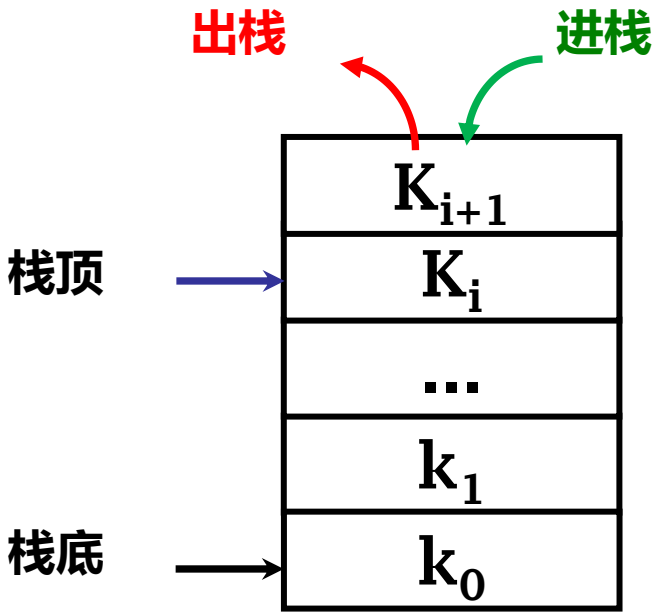



- hanoi(n,X,Y,Z)
  - 移动n个槃环
  - X柱出发，将槃环移动到Z 柱
  - X、Y、Z都可以暂存
    - 大盘不能压小盘
- 例如， hanoi(2, ‘B’, ‘C’ , ‘A’)
  - B柱上部的2个环槃移动 到A 柱

# 3.1.3 递归转非递归

```
void hanoi(int n, char X, char Y, char Z) {
    if (n <= 1)
        move(X,Z);
    else  { // X上最大的不动，其他 n - 1个环槃移到Y
        hanoi(n-1,X,Z,Y);
        move(X,Z);          // 移动最大环槃到Z，放好
        hanoi(n-1,Y,X,Z); // 把 Y上的n - 1个环槃移到Z
    }
}
void move(char X, char Y)   // 把柱X的顶部环槃移到柱Y
{
  cout  << "move"  <<  X  << "to" << Y  << endl;
}
```

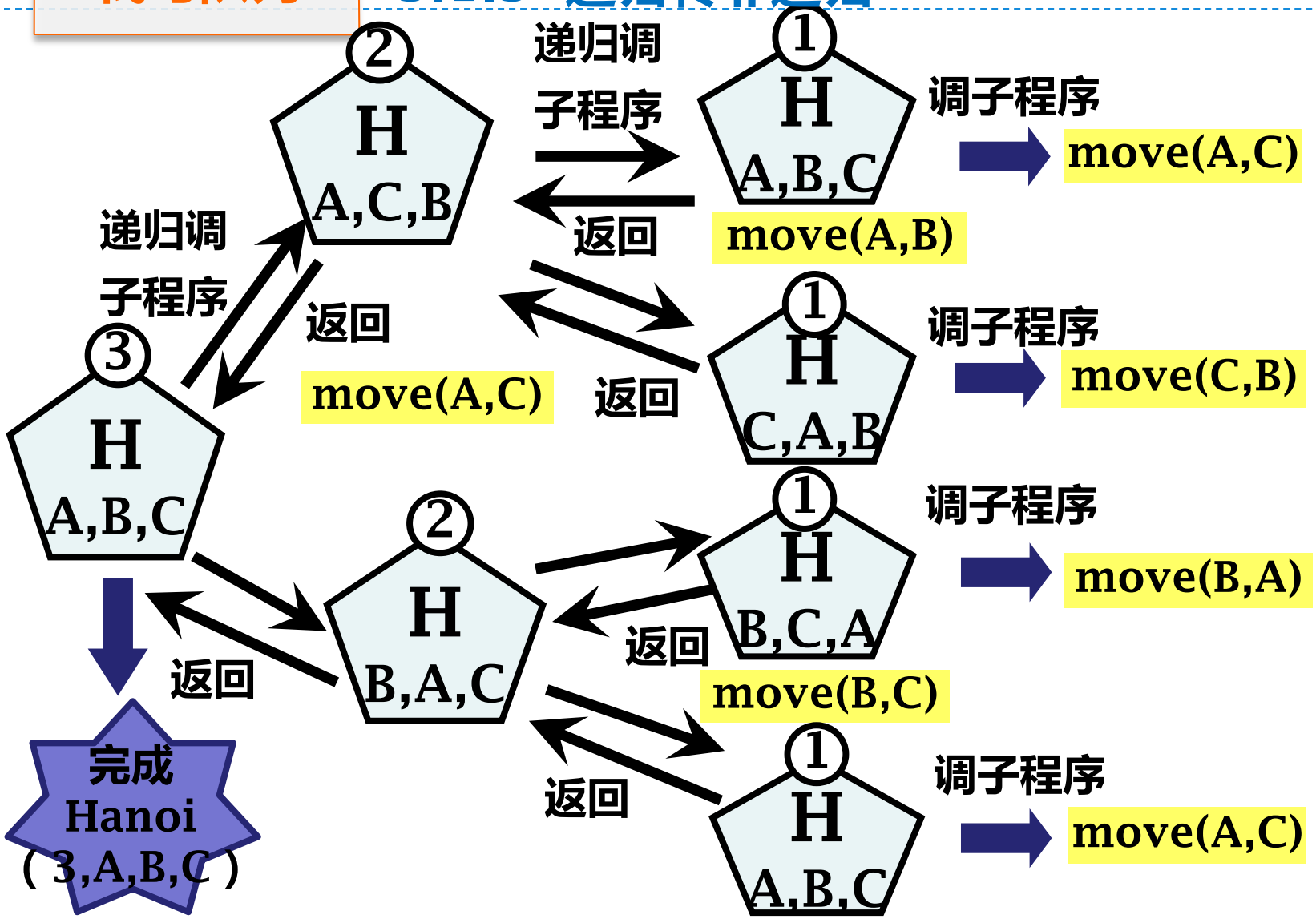
# Hanoi递归子程序的运行示意图

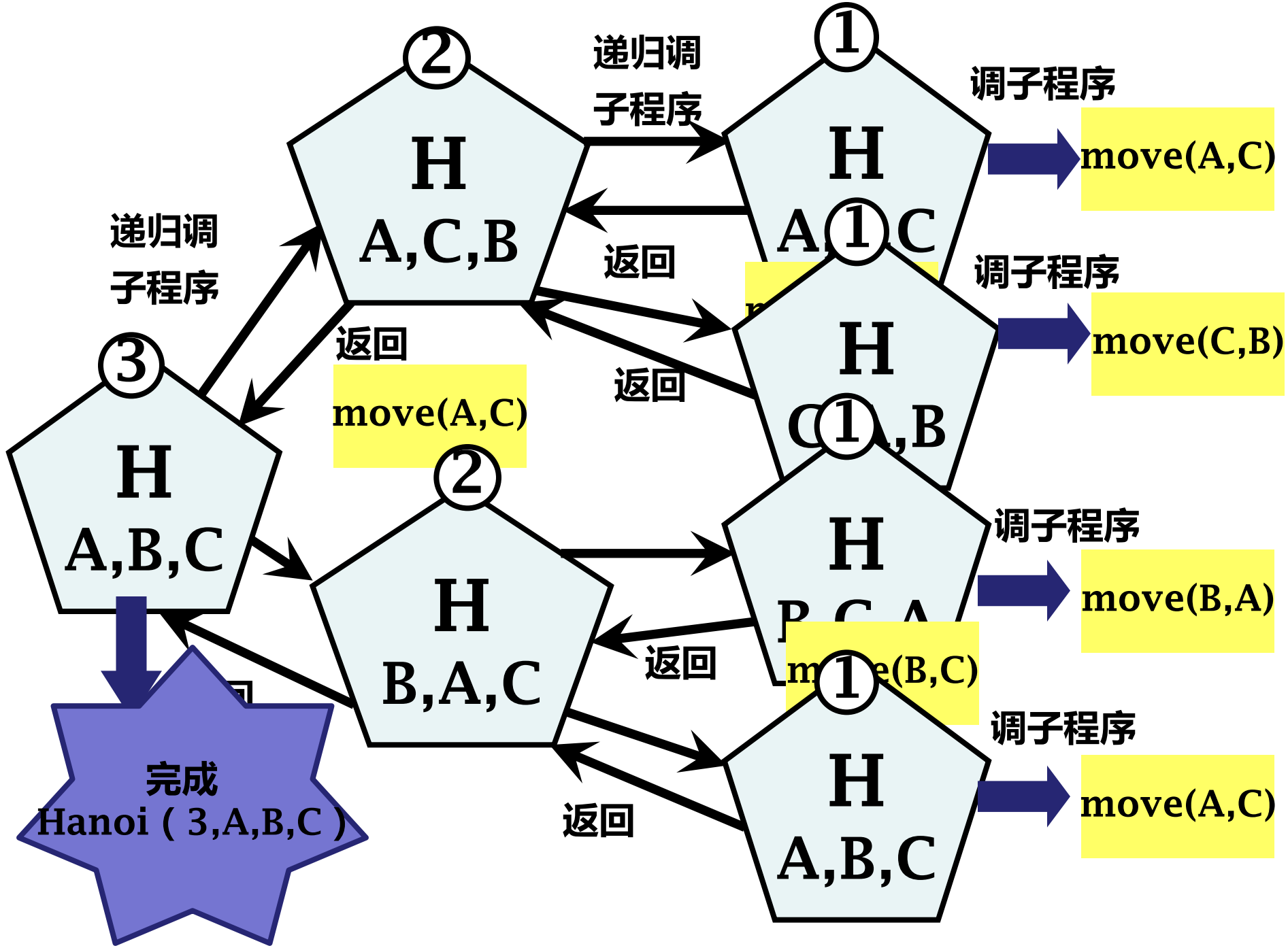




执行hanoi程序的指令流

通过内部栈和子程序交换信息

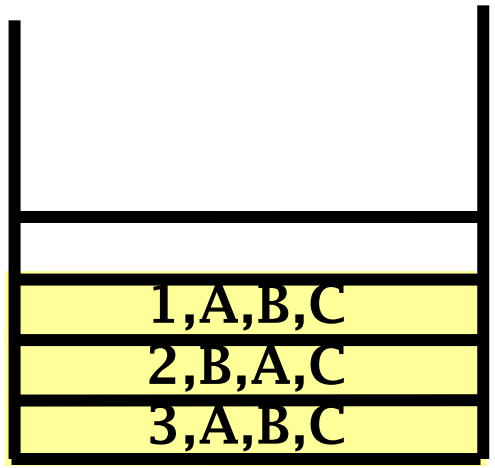
## 3.1.3 递归转非递归


②
H
A,C,B
递归调
子程序
①
H
A,B,C
调子程序
move(A,C)
返回
move(A,B)
递归调
子程序
返回
move(A,C)
③
H
A,B,C
①
H
C,A,B
调子程序
move(C,B)
返回
②
H
B,A,C
①
H
B,C,A
调子程序
move(B,A)
返回
move(B,C)
返回
①
H
A,B,C
调子程序
move(A,C)
完成
Hanoi
( 3,A,B,C )


递归调
子程序
调子程序
move(A,C)
返回
递归调
子程序
返回
move(A,C)
调子程序
move(C,B)
返回
调子程序
move(B,A)
返回
调子程序
move(A,C)
返回
完成
Hanoi ( 3,A,B,C )

# 递归运行时,堆栈的进退以及通过堆栈传递参数

hanoi(1,A,B,C)
hanoi(1,B,C,A)
hanoi(2,B,A,C)
hanoi(1,C,A,B)
hanoi(1,A,B,C)
hanoi(2,A,C,B)
hanoi(3,A,B,C)




执行move(A,C)

# 一个递归数学公式

$$fu(n)=\begin{cases} n+1 & \text{当}n<2\text{时} \\ fu(\lfloor n/2 \rfloor)*fu(\lfloor n/4 \rfloor) & n\geq 2\text{时} \end{cases}$$

# 递归函数示例

$$fu(n)=\begin{cases} n+1 & \text{当}n<2\text{时} \\ fu(\lfloor n/2 \rfloor)*fu(\lfloor n/4 \rfloor) & n\geq 2\text{时} \end{cases}$$

```
int f(int n) {
    if (n<2)
        return n+1;
    else
        return f(n/2) * f(n/4);
}
```

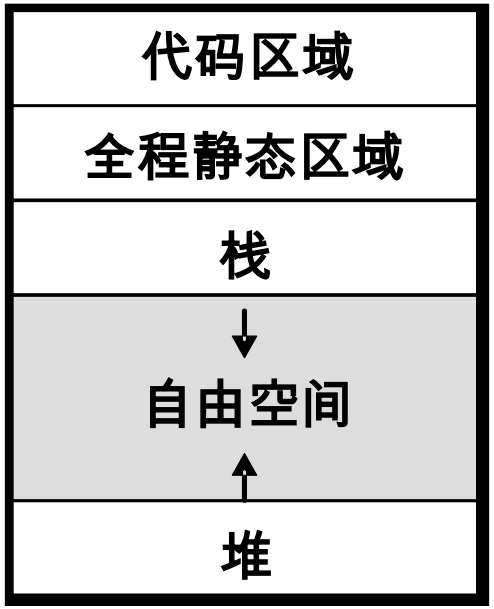
# 递归函数示例(转换一下)

```
void exmp(int n, int& f) {
    int u1, u2;
    if (n<2)
        f = n+1;
    else {
        exmp((int)(n/2), u1);
        exmp((int)(n/4), u2);
        f = u1*u2;
    }
}
```

$$fu(n)=\begin{cases} n+1 & \text{当}n<2\text{时} \\ fu(\lfloor n/2 \rfloor)*fu(\lfloor n/4 \rfloor) & n\geq 2\text{时} \end{cases}$$

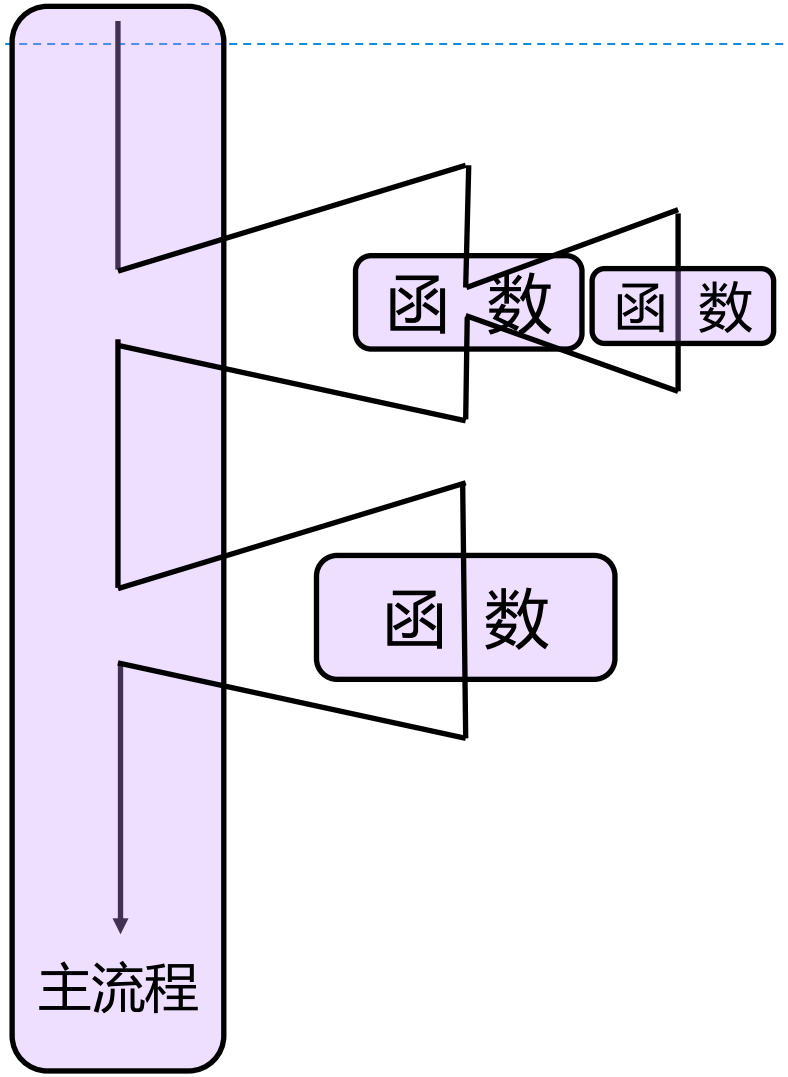
# 函数运行时的动态存储分配

- **栈（stack）**用于分配后进先出LIFO的数据
  - 如函数调用
- **堆（heap）**用于不符合LIFO的
  - 如指针所指向空间的分配



# 函数调用及返回的步骤

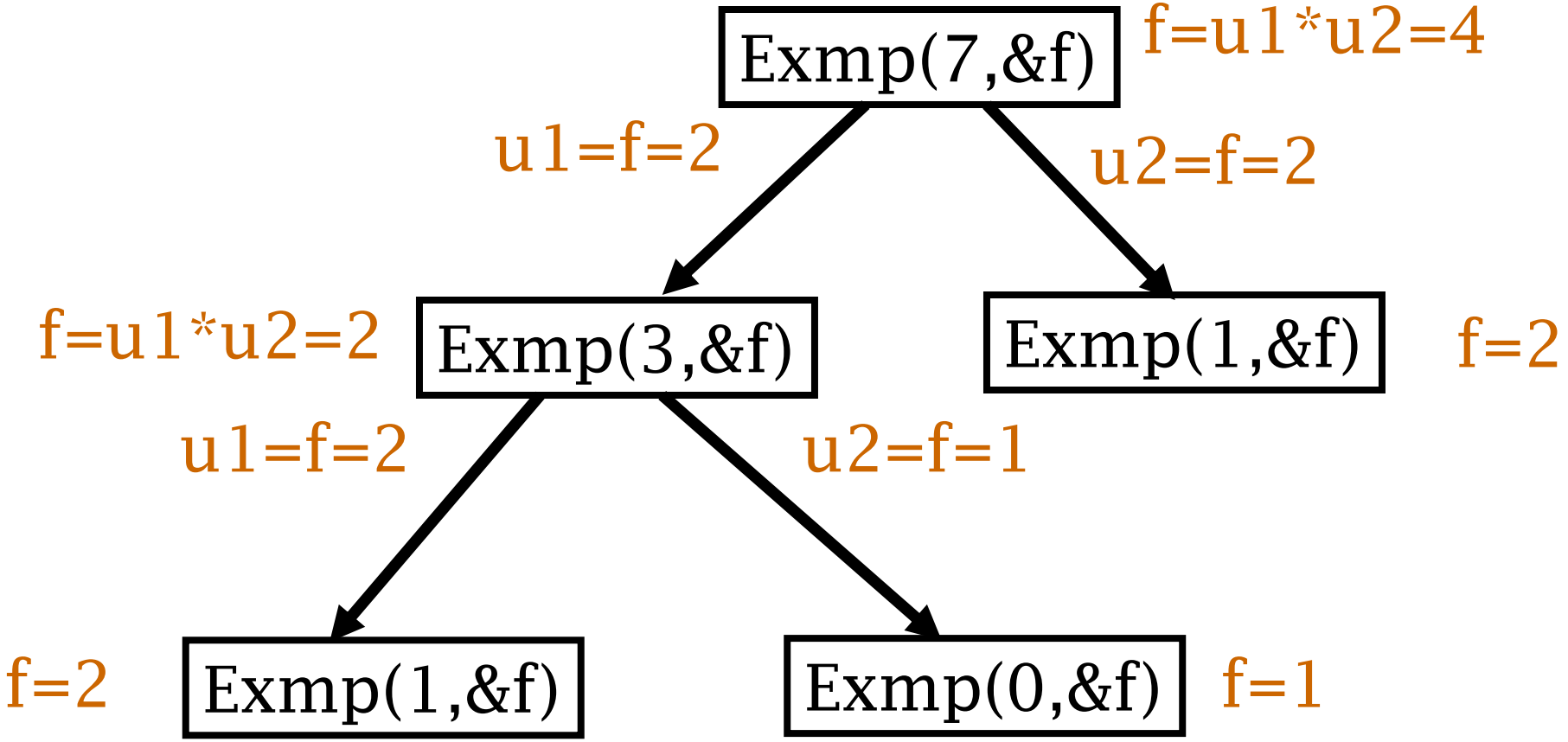
- **调用**
  - 保存调用信息（参数，返回地址）
  - 分配数据区（局部变量）
  - 控制转移给被调函数的入口
- **返回**
  - 保存返回信息
  - 释放数据区
  - 控制转移到上级函数（主调用函数）



## 3.1.3 递归转非递归

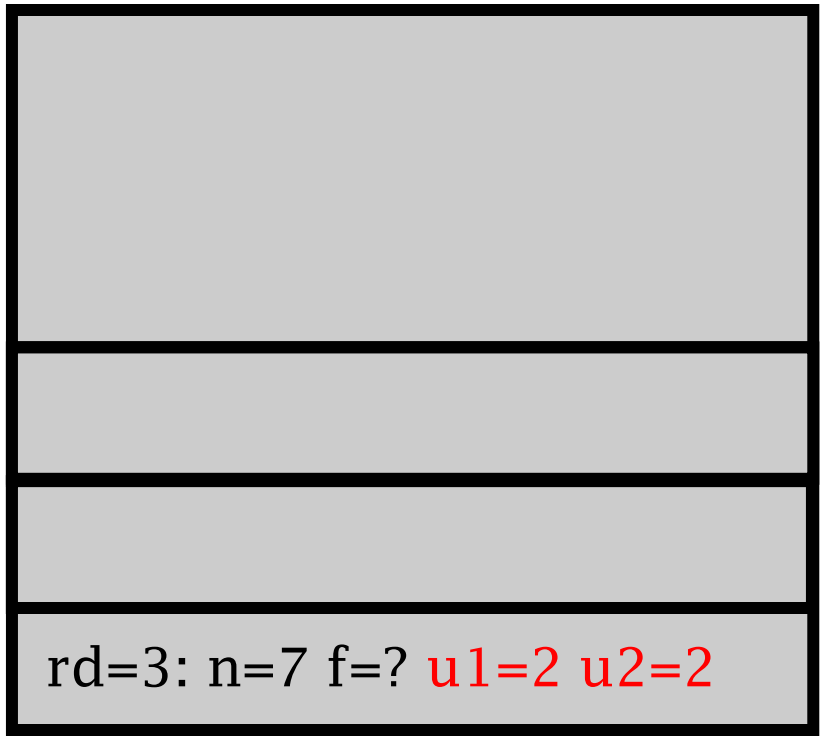
$$fu(n)=\begin{cases} n+1 & \text{当}n<2\text{时} \\ fu(\lfloor n/2 \rfloor)*fu(\lfloor n/4 \rfloor) & n\geq 2\text{时} \end{cases}$$

# 函数执行过程图解



# 用栈模拟递归调用过程

- 后调用，先返回（LIFO），所以用栈




```
void exmp(int n, int& f) {
    int u1, u2;
    if (n<2) f = n+1;
    else {
        exmp((int)(n/2), u1);
        exmp((int)(n/4), u2);
        f = u1*u2;
    }
}
```

# 思考

- 对以下函数，请画出n=4情况下的递归树，并用栈模拟递归的调用过程。
  - 阶乘函数
  - 2阶斐波那契函数

$$f_0=0,\ f_1=1,\ f_n = f_{n-1}+ f_{n-2}$$

# 第3章　栈与队列

- 栈
- **栈的应用**
  - **递归到非递归的转换**
- 队列

# 递归转非递归

- 递归函数调用原理
- **机械的递归转换**
- **优化后的非递归函数**

# 递归转换为非递归的方法

$$fu(n)=\begin{cases} n+1 & \text{当}n<2\text{时} \\ fu(\lfloor n/2 \rfloor)*fu(\lfloor n/4 \rfloor) & n\geq 2\text{时} \end{cases}$$

- 机械方法

1. 设置一工作栈当前工作记录
2. 设置 t+2个语句标号
3. 增加非递归入口
4. 替换第 i (i = 1, …, t)个递归规则
5. 所有递归出口处增加语句：goto label t+1;
6. 标号为t+1的语句的格式
7. 改写循环和嵌套中的递归
8. 优化处理

| |
|---|
| rd=2: n=0 f=? u1=? u2=? |
| rd=1: n=3 f=? u1=2 u2=? |
| rd=3: n=7 f=? u1=? u2=? |

## (2) 机械的递归转换

# 1. 设置一工作栈记录当前工作记录

- 在函数中出现的所有参数和局部变量都必须用栈中相应的数据成员代替
  - 返回语句标号域 (t+2个数值)
  - 函数参数(值参、引用型)
  - 局部变量

```
typedef struct elem { // 栈数据元素类型
    int rd;              // 返回语句的标号
    Datatypeofp1 p1;     // 函数参数
    …
    Datatypeofpm pm;
    Datatypeofq1 q1;     // 局部变量
    …
    Datatypeofqn qn;
} ELEM;
```

# 2. 设置t+2个语句标号

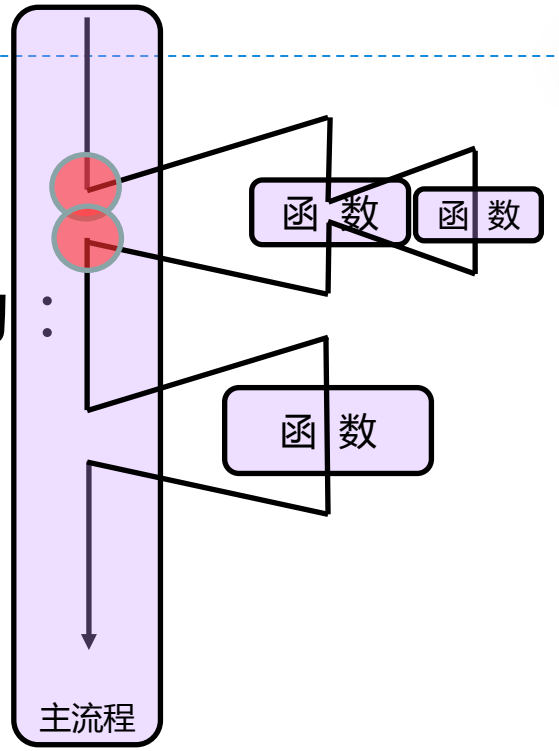
- label *0* ：第一个可执行语句
- label t+1 ：设在函数体结束处
- label i （1<=i<=t)： 第i个递归返回处

# 3. 增加非递归入口

```
// 入栈
S.push(t+1，p1, ... , pm , q1 , ...qn);
```

## (2) 机械的递归转换

# 4. 替换第i (i=1,…,t)个递归规则




- 假设函数体中第i (i=1, …, t)个递归调用语句为：recf(a1, a2, …,am) ;
- 则用以下语句替换：

```
S.push(i, a1, …, am) ;  // 实参进栈
goto label 0 ;
.....
label i :  x = S.top() ; S.pop();
```

/* 退栈，然后根据需要，将x中某些值赋给栈顶的工作记录S.top () — 相当于把引用型参数的值回传给局部变量 */

# 5. 所有递归出口处增加语句

- goto label *t+1*;

# 6. 标号为t+1的语句

```
switch ((x=S.top ()).rd) {
    case 0 :    goto label 0;
                break;
    case 1 : goto label 1;
               break;
    ……….
    case t+1 : item = S.top(); S.pop(); // 返回
               break;
    default : break;
}
```

# 7. 改写循环和嵌套中的递归

- 对于循环中的递归，改写成等价的goto型循环
- 对于嵌套递归调用

  例如，recf (... recf())

  改为：

  $exmp_1$ = recf ( );

  $exmp_2$ = recf ($exmp_1$);

  ...

  $exmp_k$ = recf ($exmp_{k-1}$)

  然后应用规则 4 解决

# 8. 优化处理

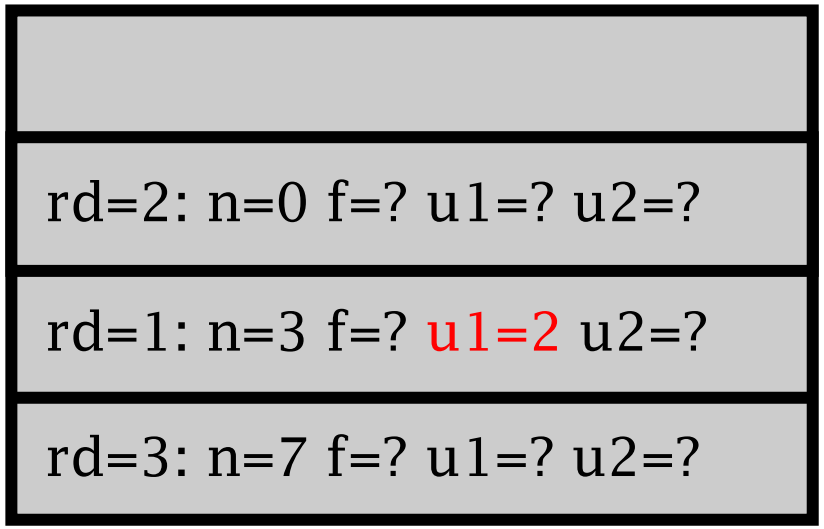
- 进一步优化
  - 去掉冗余进栈/出栈
  - 根据流程图找出相应的循环结构，从而消去goto语句

## (2) 机械的递归转换

# 数据结构定义

$$fu(n)=\begin{cases} n+1 & \text{当}n<2\text{时} \\ fu(\lfloor n/2 \rfloor)*fu(\lfloor n/4 \rfloor) & n\geq 2\text{时} \end{cases}$$

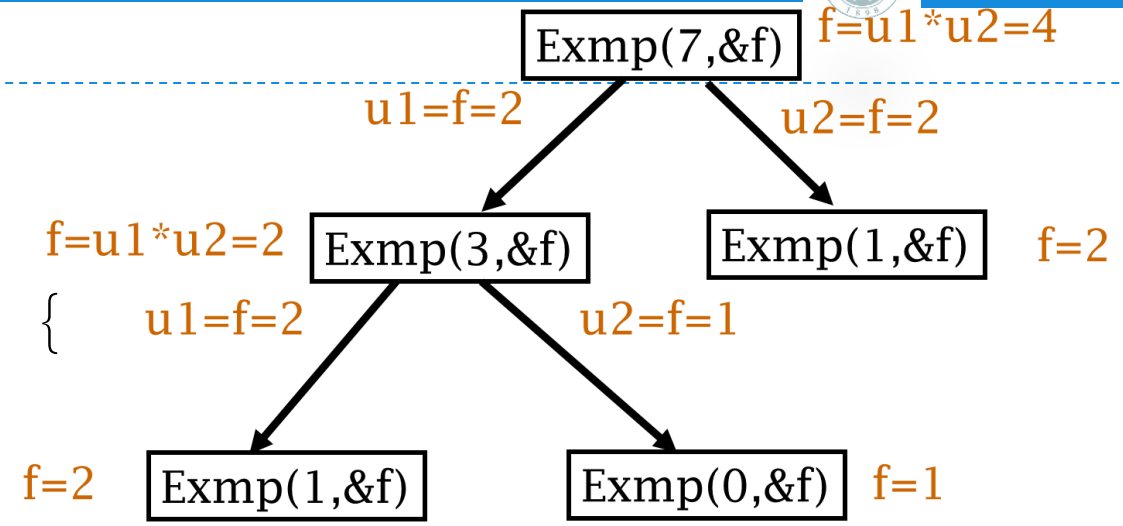
```
typedef struct elem {
    int rd, pn, pf, q1, q2;
} ELEM;
class nonrec {
    private:
        stack <ELEM> S;
    public:
        nonrec(void) { }      // constuctor
        void replace1(int n, int& f);
};
```



## (2) 机械的递归转换

# 递归入口

```
void nonrec::replace1(int n, int& f)   {
    ELEM x, tmp
    x.rd = 3;       x.pn = n;
    S.push(x);      // 压在栈底作"监视哨"
label0: if ((x = S.top()).pn < 2)   {
            S.pop( );
            x.pf = x.pn + 1;
            S.push(x);
            goto label3;
        }
```

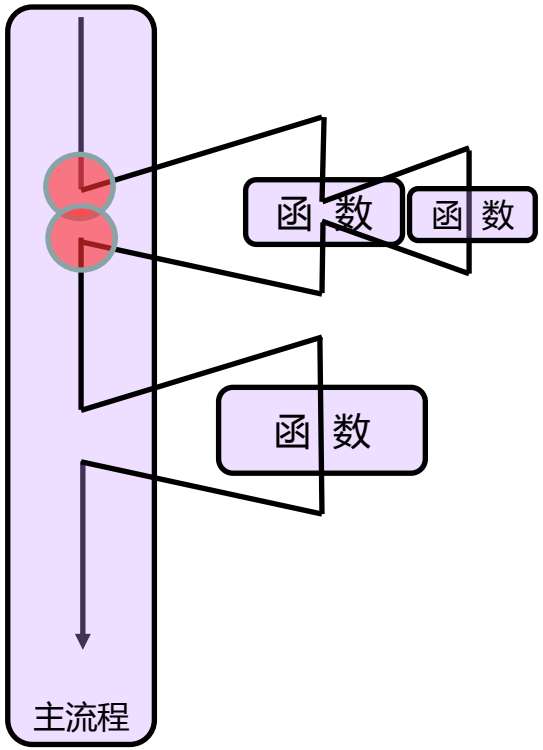


## (2) 机械的递归转换

# 第一个递归语句

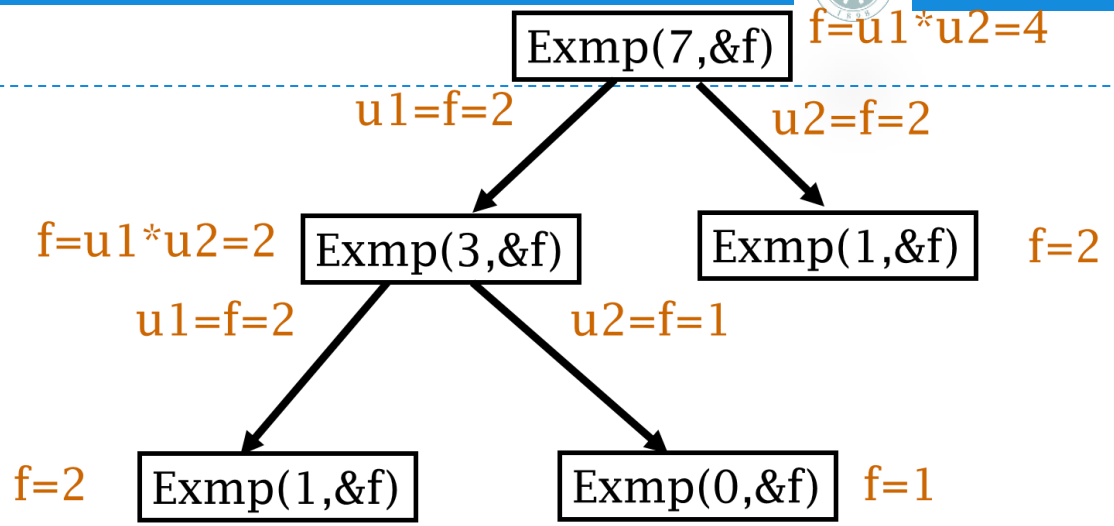
$$fu(n)=\begin{cases} n+1 & \text{当}n<2\text{时} \\ fu(\lfloor n/2 \rfloor)*fu(\lfloor n/4 \rfloor) & n\geq 2\text{时} \end{cases}$$

```
        x.rd = 1;          // 第一处递归
        x.pn = (int)(x.pn/2);
        S.push(x);
        goto label0;
label1: tmp = S.top(); S.pop();
        x = S.top(); S.pop();
        x.ql = tmp.pf; // 修改ul=pf
        S.push(x);
```



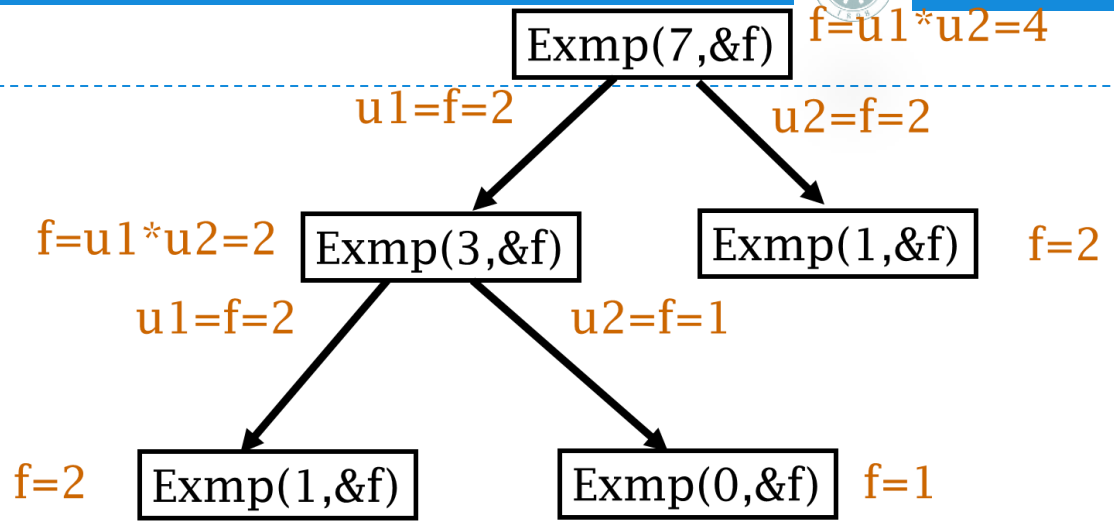
# 第二个递归语句

```
        x.pn = (int)(x.pn/4);
        x.rd = 2;
        S.push(x);
        goto label0;
label2: tmp = S.top(); S.pop();
        x = S.top(); S.pop();
        x.q2 = tmp.pf;
        x.pf = x.q1 * x.q2;
        S.push(x);
```



## (2) 机械的递归转换

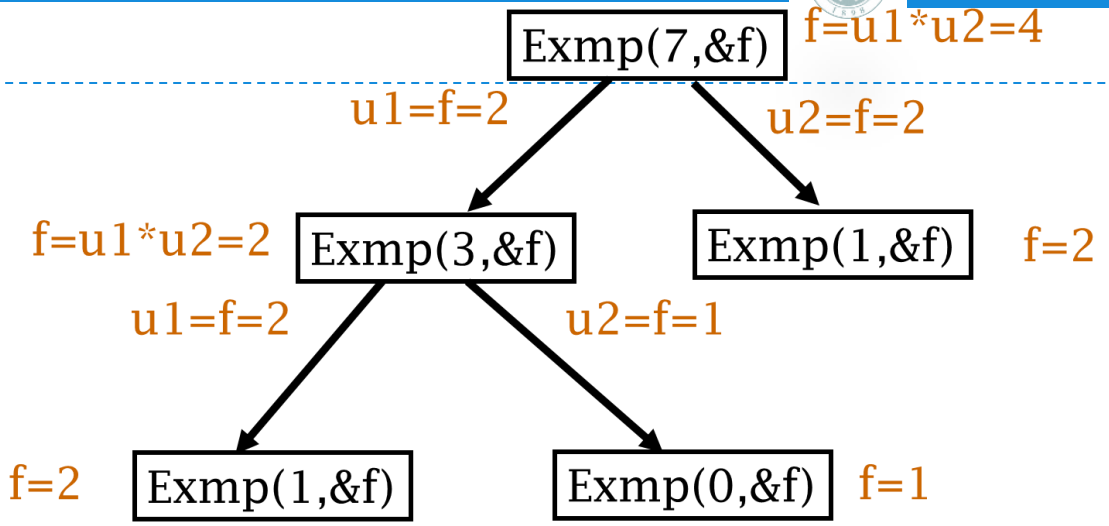
```
label3: x = S.top();
      switch(x.rd)  {
         case 1 : goto label1;
                  break;
         case 2 : goto label2;
                  break;
         case 3 : tmp = S.top(); S.pop();
                  f = tmp.pf;            //计算结束
                  break;
         default : cerr << "error label number in stack";
                   break;
       }
}
```



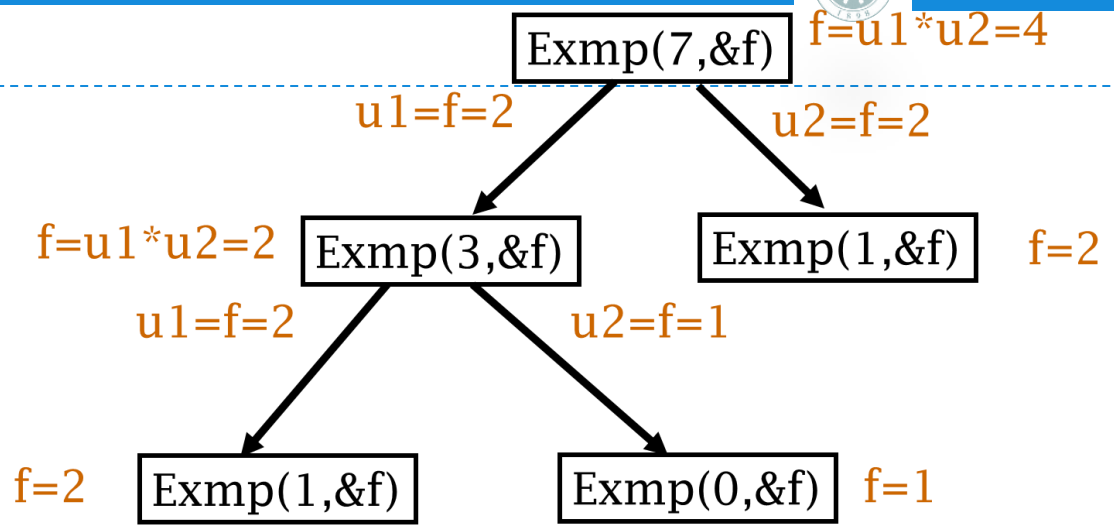
## （3）优化后的非递归函数

# 优化后的非递归算法

```
void nonrec::replace2(int n, int& f) {
    ELEM x, tmp;
     // 入口信息
    x.rd = 3;   x.pn = n;   S.push(x);
    do {
      // 沿左边入栈
       while ((x=S.top()).pn >= 2){
          x.rd = 1;
          x.pn = (int)(x.pn/2);
          S.push(x);
       }
```



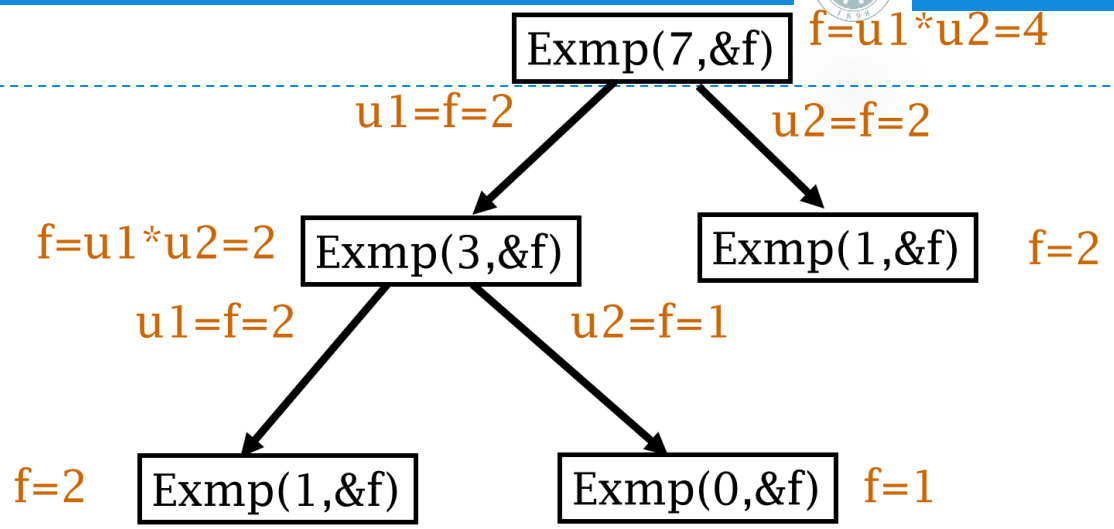
## （3）优化后的非递归函数




```
x = S.top(); S.pop();   // 原出口，n <= 2
x.pf = x.pn + 1;
S.push(x);
// 如果是从第二个递归返回，则上升
while ((x = S.top()).rd==2) {
  tmp = S.top(); S.pop();
  x = S.top(); S.pop();
  x.pf = x.q * tmp.pf;
  S.push(x);
}
```

## （3）优化后的非递归函数

```
        if ((x = S.topValue()).rd == 1) {
            tmp = S.top(); S.pop();
            x = S.top(); S.pop();
            x.q = tmp.pf;    S.push(x);
            tmp.rd = 2;  // 进入第二个递归
            tmp.pn = (int)(x.pn/4);
            S.push(tmp);
        }
    } while ((x = S.top()).rd != 3);
    x = S.top(); S.pop();
    f = x.pf;
}
```



## 递归转非递归的性能实验

# 快排的时间对照（单位ms）

| 方法 \ 数据量 | 10000 | 100000 | 1000000 | 10000000 |
|---|---|---|---|---|
| 递归快排 | 4.5 | 29.8 | 268.7 | 2946.7 |
| 机械方法的非递归快排 | 1.6 | 23.3 | 251.7 | 2786.1 |
| 非机械方法的非递归快排 | 1.6 | 20.2 | 248.5 | 2721.9 |
| STL中的sort | 4.8 | 59.5 | 629.8 | 7664.1 |

注：测试环境
Intel Core Duo CPU T2350
内存 512MB
操作系统 Windows XP SP2
编程环境 Visual C++ 6.0

# 递归与非递归处理问题的规模

- 递归求f(x)：

```
int f(int x) {
    if (x==0) return 0;
    return f(x-1)+1;
}
```

  - 在默认设置下，当x超过 11,772 时会出现堆栈溢出
- 非递归求f(x)，栈中元素记录当前x值和返回值
  - 在默认设置下，当x超过32,375,567时会出现错误

# 思考

- 用机械的转换方法
  - 阶乘函数
  - 2阶斐波那契函数

$$f_0=0,\ f_1=1,\ f_n = f_{n-1}+ f_{n-2}$$

  - Hanoi 塔算法

# 数据结构与算法

## 谢谢聆听

国家精品课“数据结构与算法”
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

**张铭，王腾蛟，赵海燕**
**高等教育出版社，2008. 6。“十一五”国家级规划教材**