数据结构与算法（A）-W03/栈与队列

北京大学 陈斌

2024.09.18

第3章　栈与队列

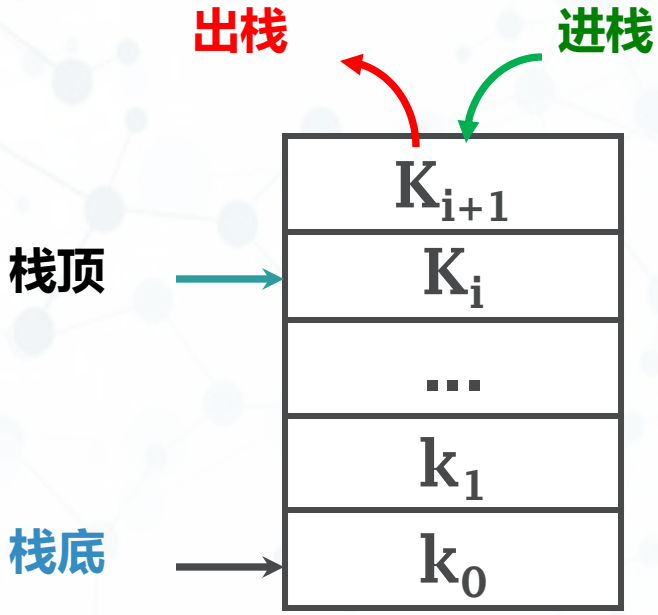
- 栈
- 队列
- 栈的应用
  递归到非递归的转换

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

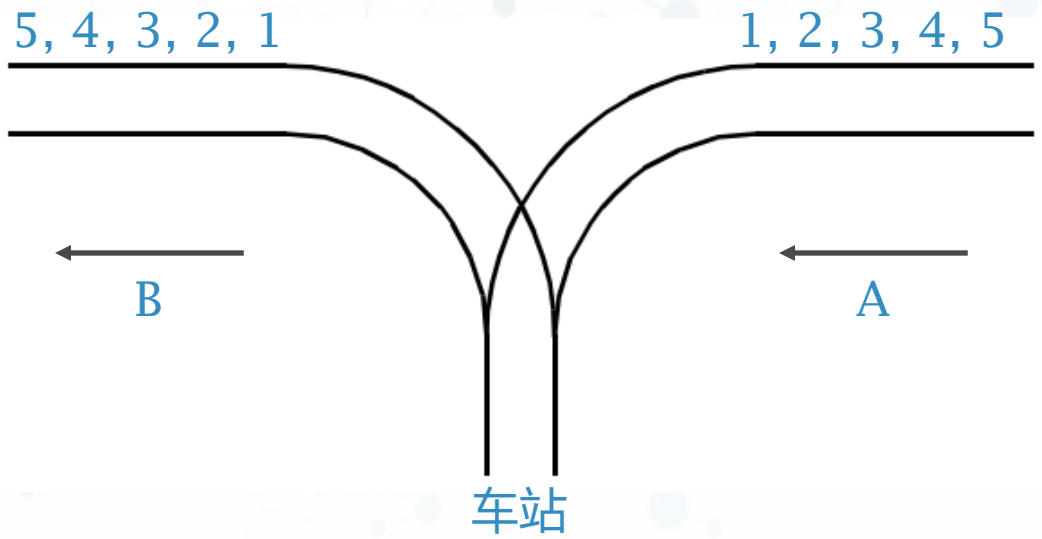


栈的抽象数据类型

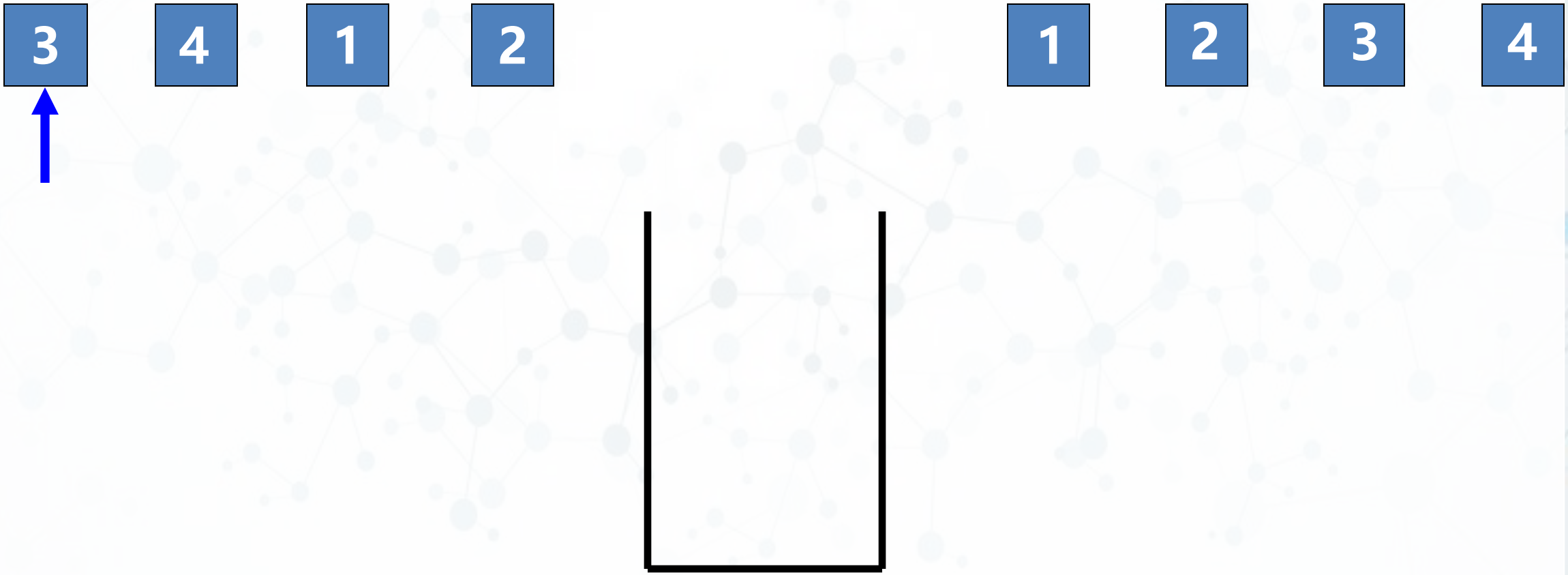
```
template <class T>
class Stack {
public:                          // 栈的运算集
    void clear();                // 变为空栈
    bool push(const T item);
                                 // item入栈，成功返回真，否则假
    bool pop(T &item);           // 返回栈顶内容并弹出，成功返回真，否则假
    bool top(T &item);           // 返回栈顶但不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

若入栈顺序为1,2,3,4的话，则出栈的顺序可以有哪些?

1234

1243

1324

1342

1423

1432

2134

2143

思考

- 思考题
  - 给定一个入栈序列，和一个出栈序列，请你写出一个程序，判定出栈序列是否合法?
  - 给定一个入栈序列，序列长度为N，请计算有多少种出栈序列?

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)**

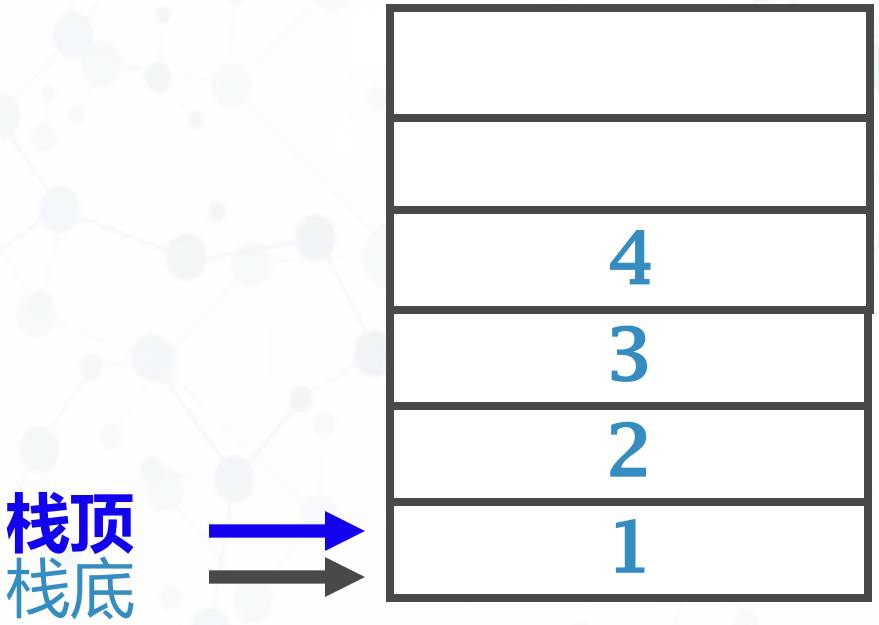
  用单链表方式存储，其中指针的方向是从栈顶向下链接

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

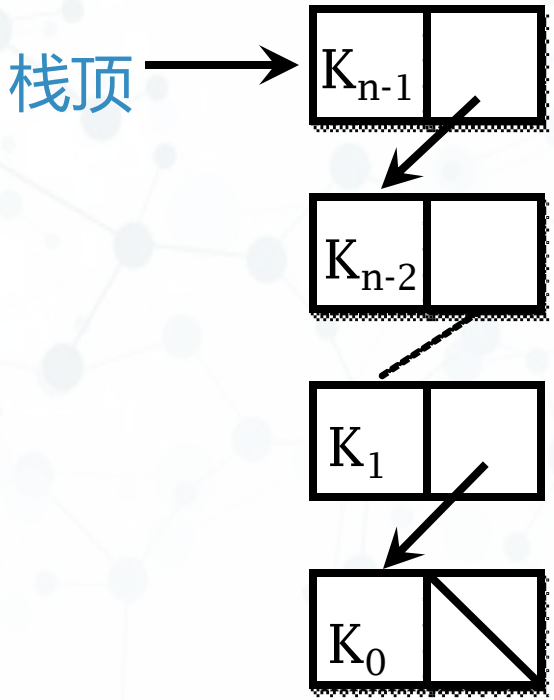
```
bool arrStack<T>::push(const T item) {
    if (top == mSize-1) {          // 栈已满
        cout << "栈满溢出" << endl;
        return false;
    } else {                       // 新元素入栈并修改栈顶指针
        st[++top] = item;
        return true;
    }
}
```

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**
- **实际应用中，顺序栈比链式栈用得更广泛**

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 + / +

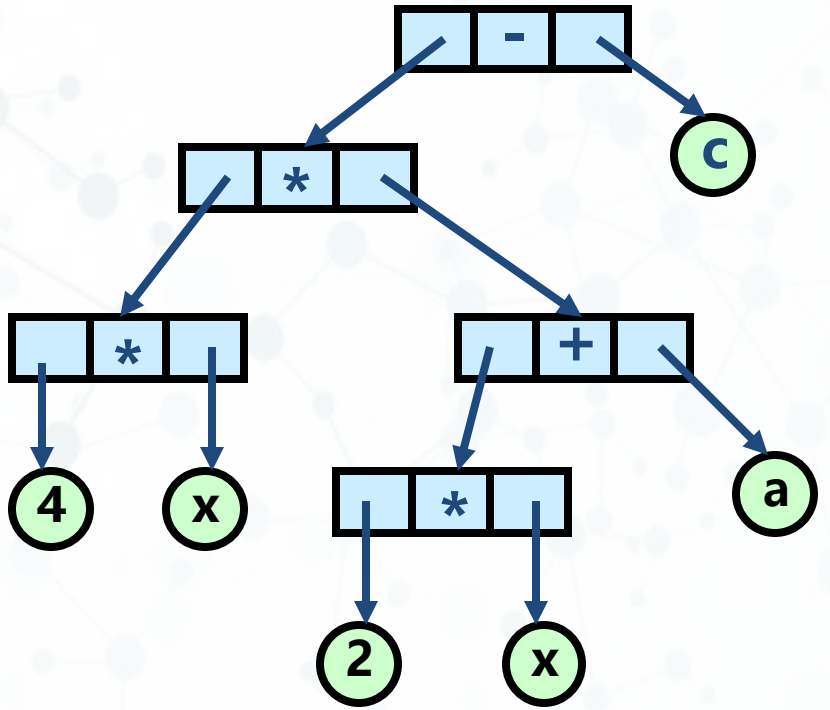
中缀表达式

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



中缀表达式的语法公式（BNF）

<表达式> ::= <项> + <项> | <项> - <项> | <项>

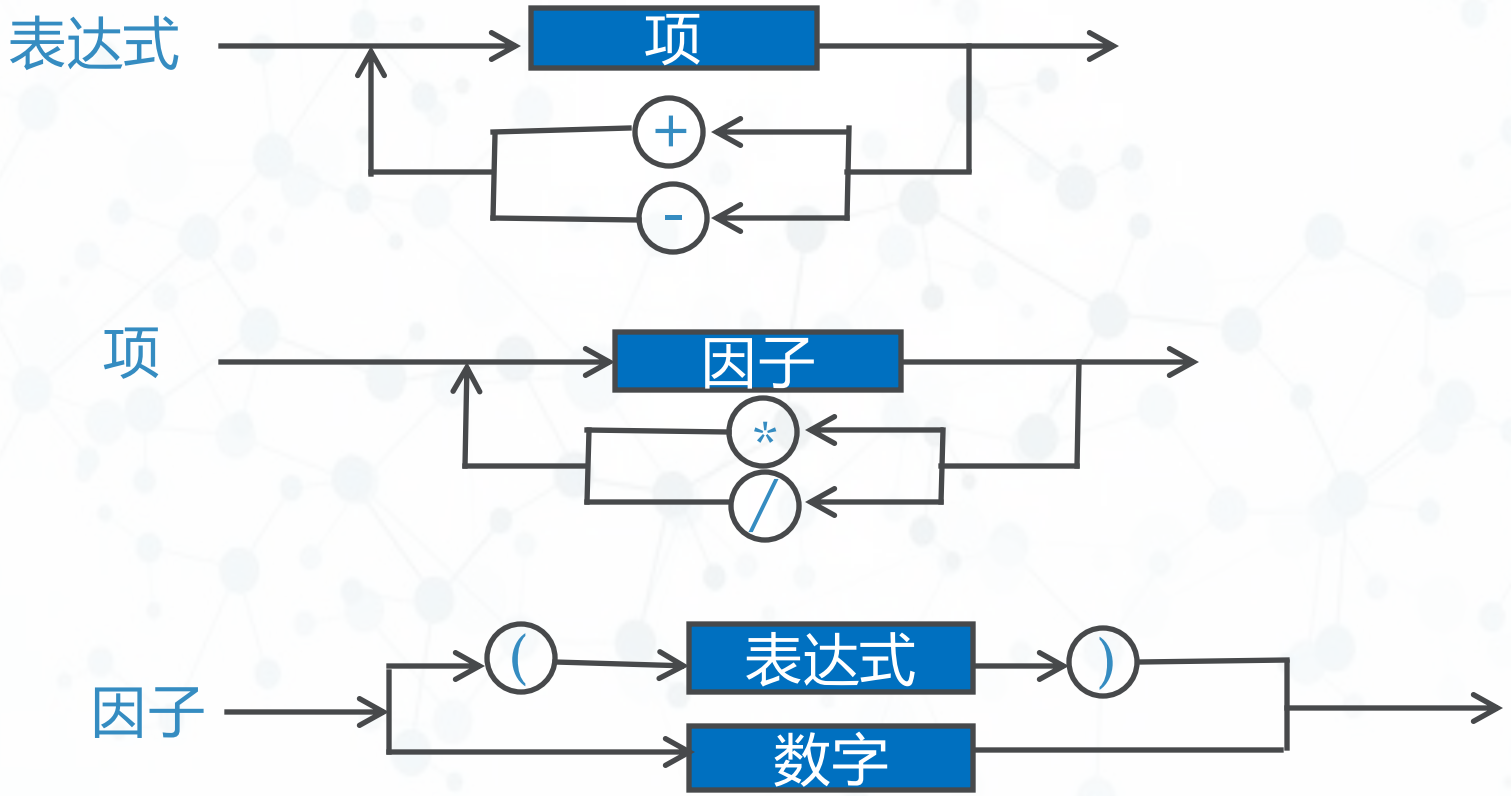
<项> ::= <因子> * <因子> | <因子> / <因子> | <因子>

<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



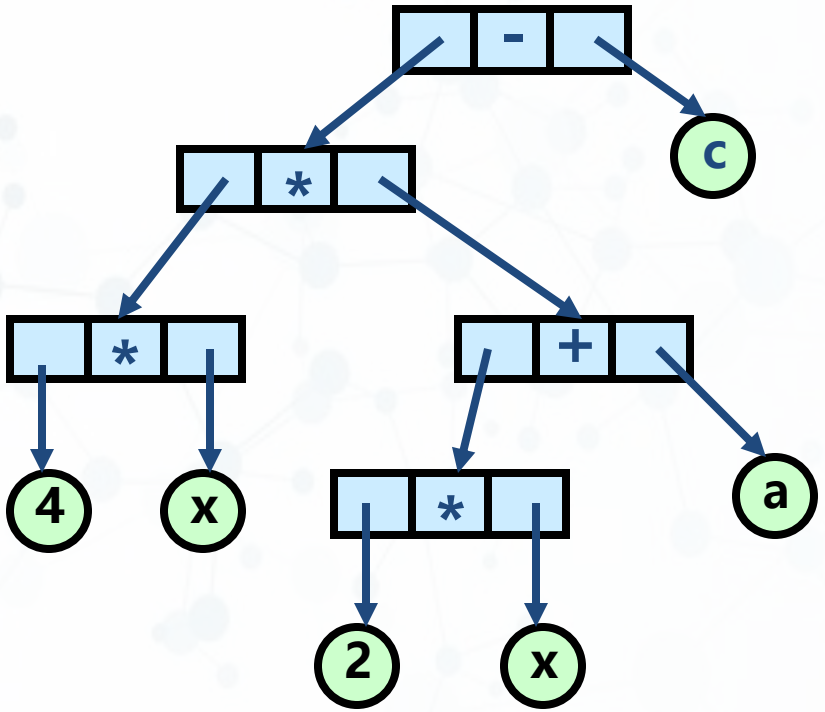
后缀表达式

后缀表达式

4 x * 2 x * a + * c –

运算符在后面

完全不需要括号



后缀表达式的语法公式（BNF）

<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

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;
                S.Push(newoperand);
                break;
        }
    }
    if (!S.IsEmpty())
        cout << S.Pop() << endl;          // 印出求值的最后结果
}
```

中缀到后缀表达式的转换

思路：利用栈来记录表达式中的运算符

- 当输入是操作数，直接输出到后缀表达式序列
- 当输入的是左括号时，也把它压栈
- 当输入的是运算符时
  - While (以下循环)
    - If（栈非空 *and* 栈顶不是左括号 *and* 输入运算符的优先级 “≤”栈顶运算符的优先级）时
      - 将当前栈顶元素弹栈，放到后缀表达式序列中（此步反复循环，直到上述if条件不成立）；将输入的运算符压入栈中。
    - Else 把输入的运算符压栈（>当前栈顶运算符才压栈！）

中缀到后缀表达式的转换

- 当输入的是右括号时，先判断栈是否为空
  - 若为空（括号不匹配），异常处理，清栈退出；
  - 若非空，则把栈中的元素依次弹出
    - 遇到第一个左括号为止，将弹出的元素输出到后缀表达式的序列中（弹出的开括号不放到序列中）
    - 若没有遇到开括号，说明括号也不匹配，做异常处理，清栈退出
- 最后，当中缀表达式的符号序列全部读入时，若栈内仍有元素，把它们全部依次弹出，都放到后缀表达式序列尾部。
  - 若弹出的元素遇到开括号时，则说明括号不匹配，做错误异常处理，清栈退出

中缀表达式→后缀表达式

输入中缀表达式： 23 + (34* 45) /(5 + 6 + 7)

栈的状态

输出后缀表达式：

第3章　栈与队列

队列

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

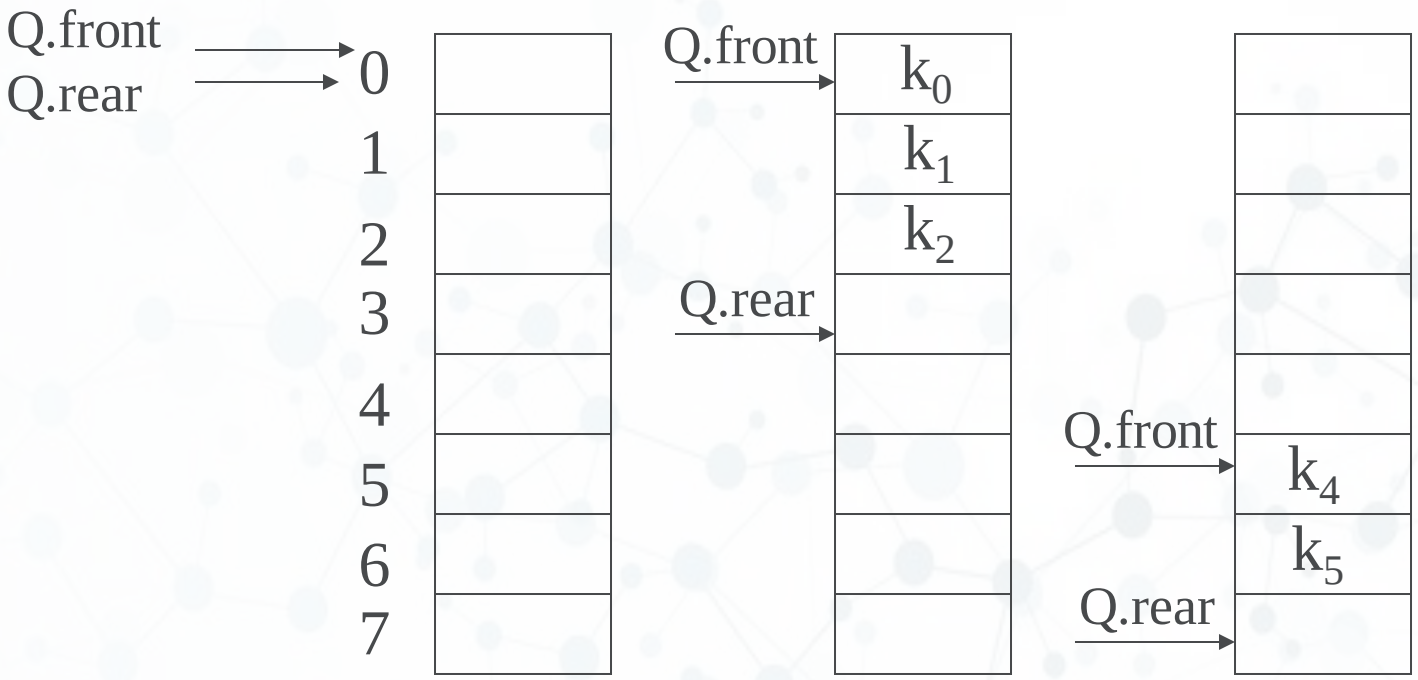
- **链式队列**

  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

顺序队列

- 用向量存储队列元素，用两个变量分别指向队列的前端(front)和尾端(rear)
  - *front*：指向当前待出队的元素位置（地址）
  - *rear*：指向当前待入队的元素位置（地址）

顺序队列



队列空

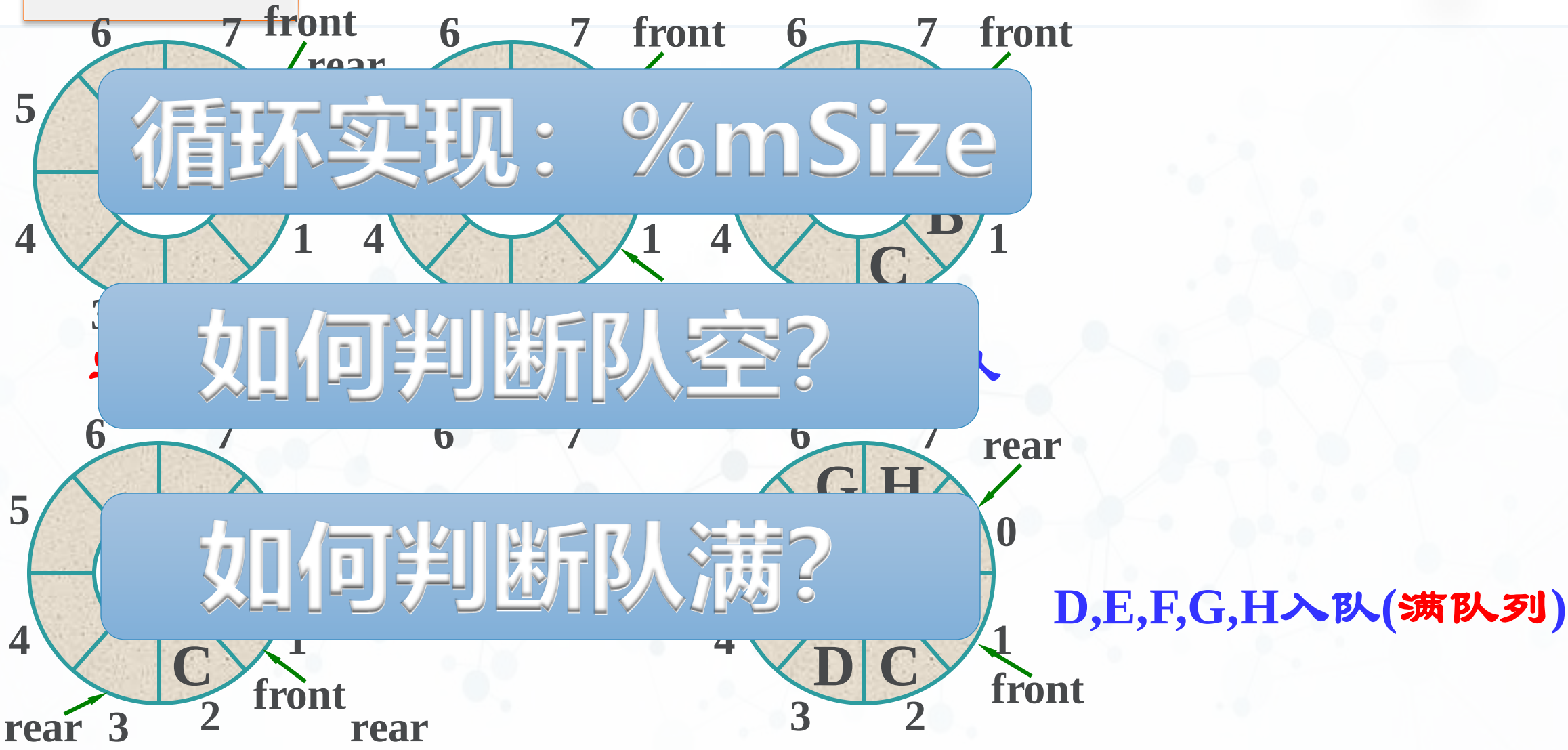
再进队一个元素如何?

顺序队列

队列的溢出

- 上溢
  - 当队列满时，再做进队操作，所出现的现象
- 下溢
  - 当队列空时，再做删除操作，所出现的现象
- 假溢出
  - 当 rear = mSize-1 时，再作插入运算就会产生溢出，如果这时队列的前端还有许多空位置，这种现象称为假溢出

循环队列运行示意图



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;                   // 表示待入队元素所在位置的下标
     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
      mSize = size +1;                   // 浪费一个存储空间，以区别队列空和队列满
      qu = new T [mSize];
      front = rear = 0;
     }
      ~arrQueue()  {                        // 消除该实例，并释放其空间
      delete [] qu;
     }
}
```

入队列的操作

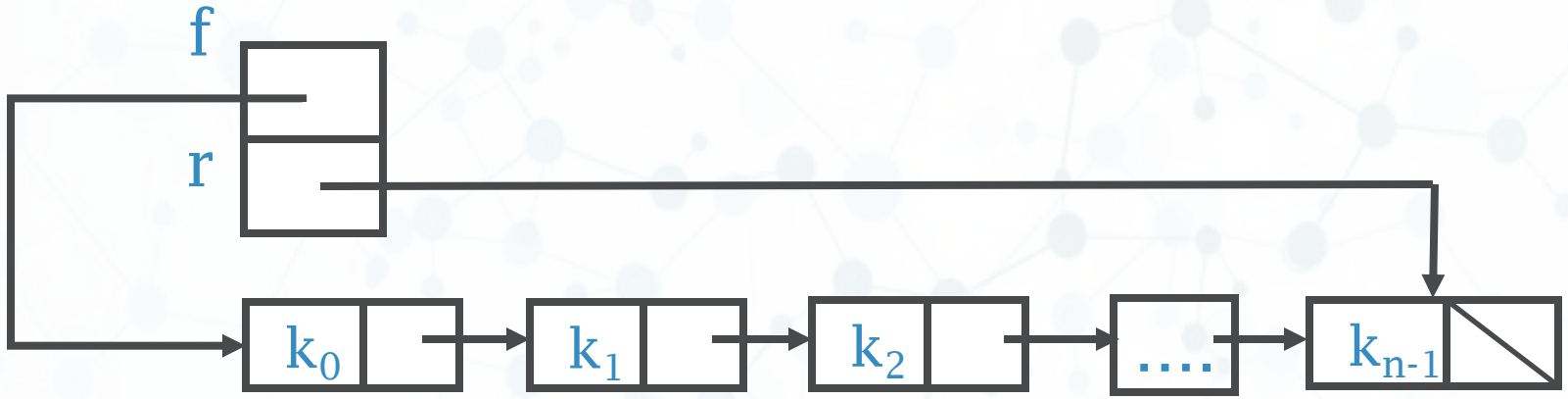
```
bool arrQueue<T> :: enQueue(const T item)  {
                        //  item入队，插入队尾
   if (((rear + 1 ) % mSize) == front) {
       cout << "队列已满，溢出" << endl;
       return false;
   }
   qu[rear] = item;
   rear = (rear +1) % mSize;                  // 循环后继
   return true;
}
```

出队列的操作

```
bool arrQueue<T> :: deQueue(T &item)  {
                    // 返回队头元素并从队列中删除
  if ( front == rear)  {
      cout << "队列为空" << endl;
      return false;
  }
  item = qu[front];
  front = (front +1) % mSize;
  return true;
}
```

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

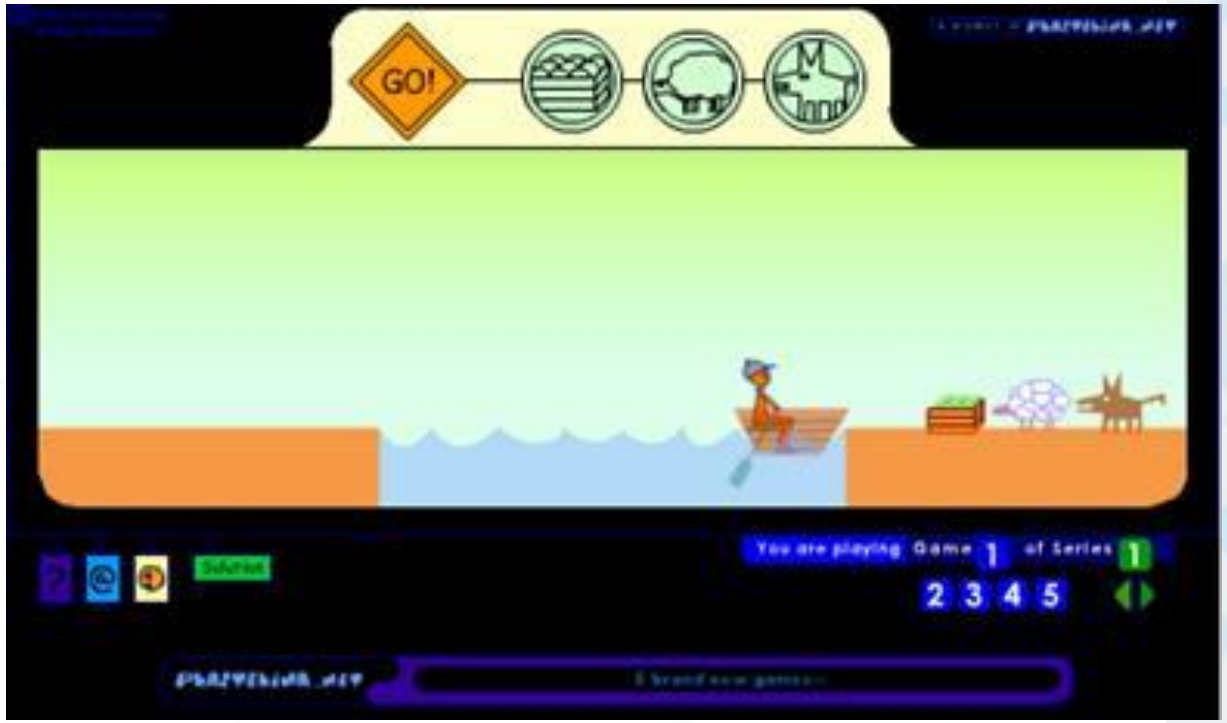
  消息缓冲器、邮件缓冲器、打印缓冲器

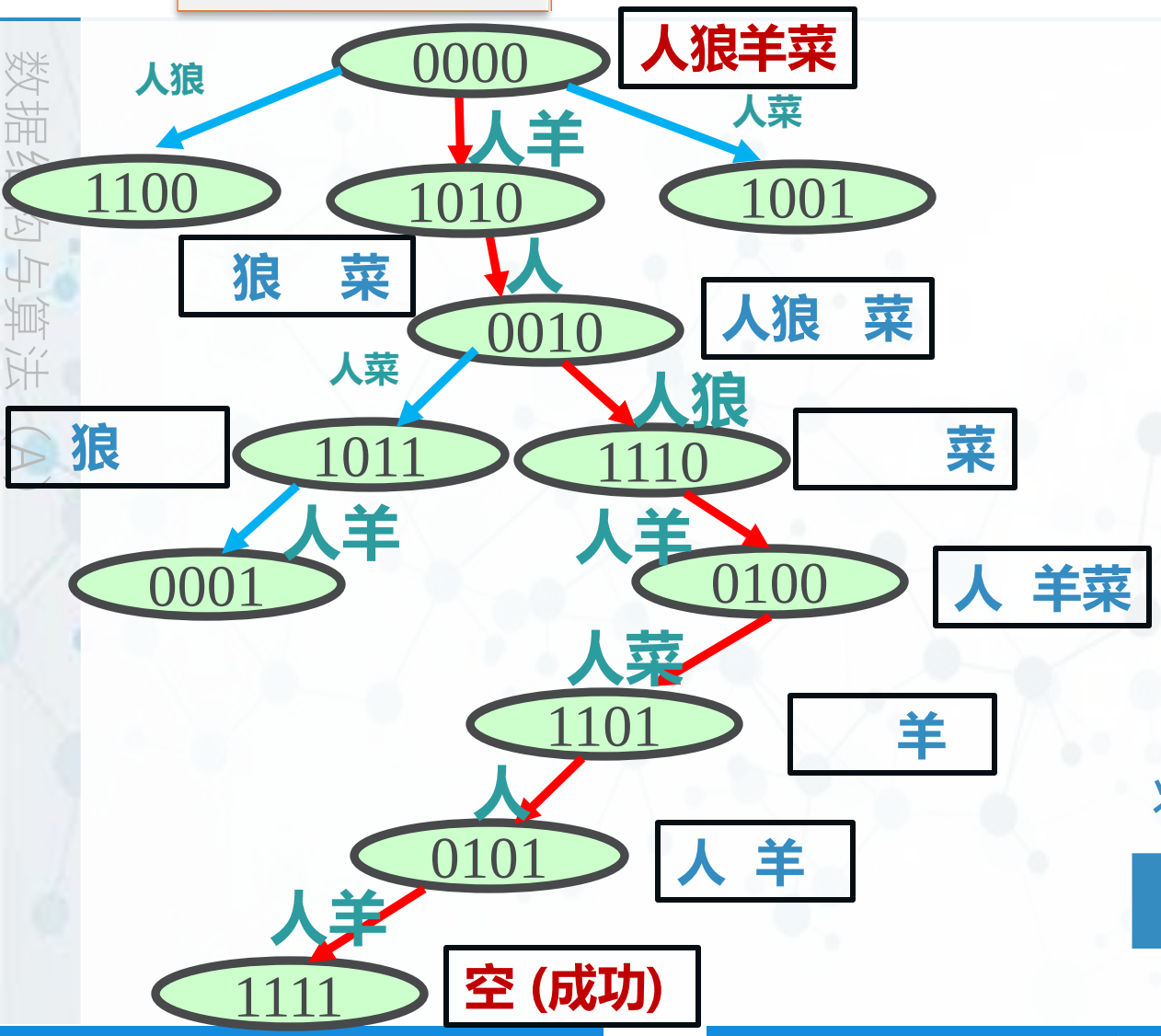
  计算机硬设备之间的通信也需要队列作为数据缓冲

  操作系统的资源管理
- 宽度优先搜索BFS

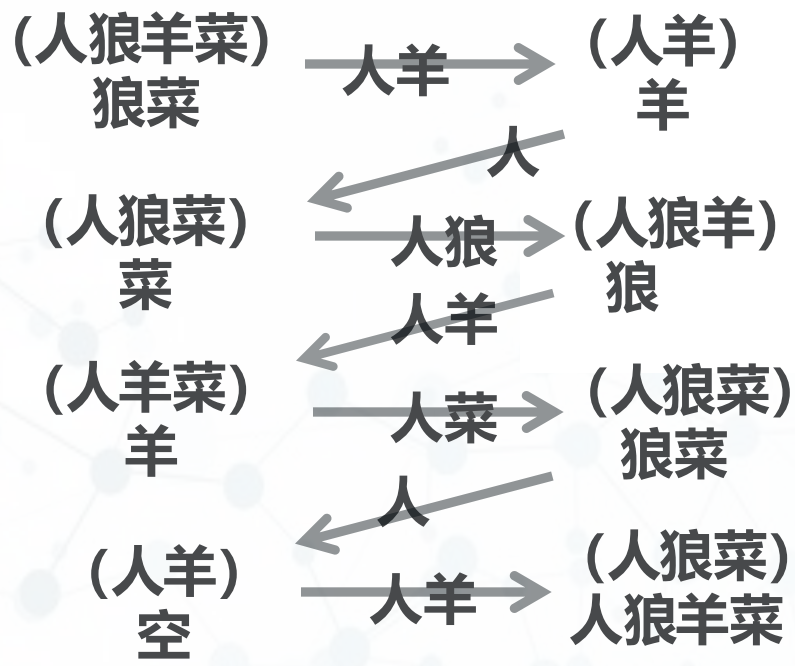
农夫过河问题

- **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





算法示意



状态位向量，物体在起始为0，否则为1

0	1	0	1
人	狼	羊	菜

问题分析

- 求解该问题最简单的方法是使用试探法，即一步一步进行试探，每一步都搜索所有可能的选择，对前一步合适的选择再考虑下一步的各种方案。

- 用计算机实现上述求解的搜索过程可以采用两种不同的策略：

广度优先搜索：搜索该步的所有可能状态，再进一步考虑后面的各种情况；（队列应用）

深度优先搜索：沿某一状态走下去，不行再回头。（栈应用）

问题分析

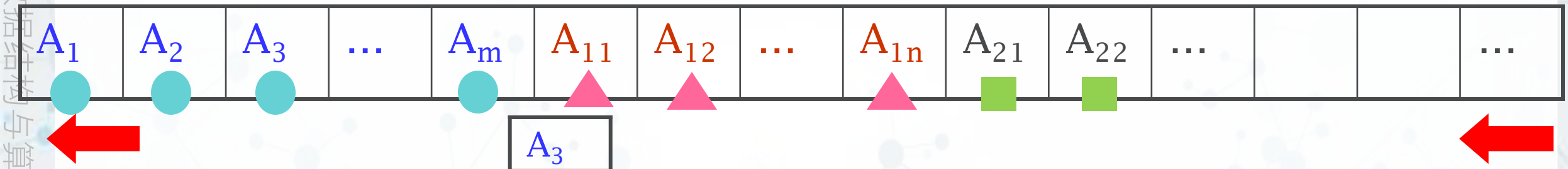
- 假定采用广度优先搜索解决农夫过河问题：

采用队列做辅助结构，把下一步所有可能达到的状态都放在队列中，然后顺序取出对其分别处理，处理过程中再把下一步的状态放在队列中，……。

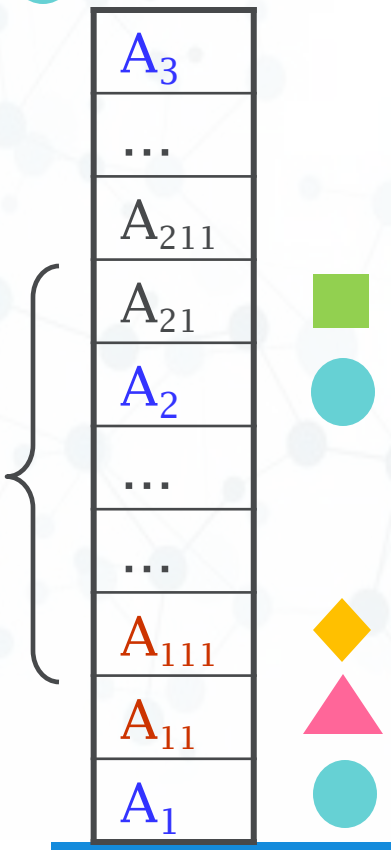
由于队列的操作按照先进先出原则，因此只有前一步的所有情况都处理完后才能进入下一步。

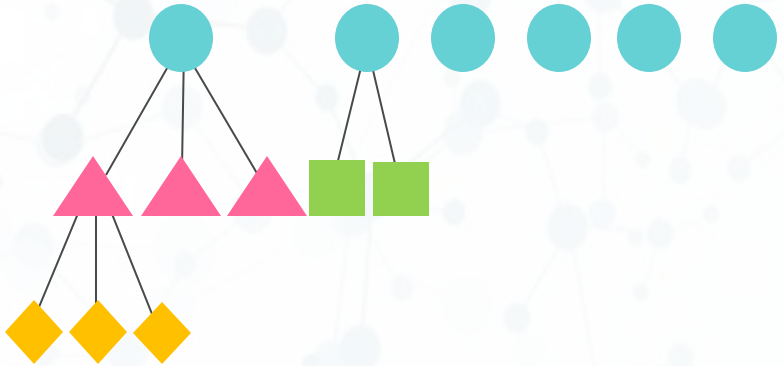
问题分析

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、羊和菜，四个角色依次各用一位，角色在起始岸位置：0，目标岸：1

0	1	1	0

如 0110 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 goat(int status)
    {  return ((status & 0x02)  != 0); }
bool cabbage(int status)
    {  return ((status & 0x01)  != 0); }
```

人	狼	羊	菜
1	x	x	x
x	1	x	x
x	x	1	x
x	x	x	1

人	狼	羊	菜
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);  // 相当于enQueue
        route[0]=0;
```

算法实现

人狼羊菜

0 0 1 0

1 1 1 0

```
while (!moveTo.empty() && route[15] == -1) {
        // 得到现在的状态
        status = moveTo.front();
        moveTo.pop();  // 相当于deQueue
        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)) {
                    // 随农夫移动以后的状态
                        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;

}
```

栈和队列的模拟

- 如何用两个栈模拟一个队列?
- 如何用两个队列模拟一个栈?

第3章　栈与队列

栈的应用

递归到非递归的转换

递归转非递归

- 简单的递归转换
- 递归函数调用原理
- 机械的递归转换

递归转非递归

- 简单的递归函数转换
  - 容易写出递推（迭代）公式，即$f(n)=F(f(n-1))$
    - 例如：阶乘 $f(n)=\begin{cases}1, & n\le 0\\ n\times f(n-1), & n>0\end{cases}$
  - 直接可以转为非递归形式

递归转非递归

- 阶乘 $f(n)=\begin{cases}1, & n\leq 0\\ n\times f(n-1), & n>0\end{cases}$

(a) 递归实现

```
long factorial(long n)
{
    if (n <= 0)
        return 1;
   return n * factorial(n-1);
}
```

(b) 非递归实现

```
long factorial(long n)
{
    long m = 1;
    for (long i = 1; i <= n; i++)
      m = m * i;
    return m;
}
```

递归转非递归

- 一类特殊的递归函数—尾递归
  - 指函数的最后一个动作是调用函数本身的递归函数，是递归的一种特殊情形
  - 尾递归的本质是：将单次计算的结果缓存起来，传递给下次调用，相当于自动累积

```
function story() {
从前有座山，山上有座庙，庙里有个老和尚，一天老和尚对小和尚讲故事：story()，小和尚听了，找了块豆腐撞死了 // 非尾递归，下一个函数结束以后此函数还有后续，所以必须保存本身的环境以供处理返回值。
}
```

```
function story() {
从前有座山，山上有座庙，庙里有个老和尚，一天老和尚对小和尚讲故事：story() // 尾递归，进入下一个函数不再需要上一个函数的环境了，得出结果以后直接返回。
}
```

递归转非递归

- 阶乘 $f(n)=\begin{cases}1, & n\le 0\\ n\times f(n-1), & n>0\end{cases}$

(a) 递归实现

```
long factorial(long n)
{
    if (n <= 0)
        return 1;
   return n * factorial(n-1);
}
```

(b) 尾递归实现

```
long factorial(long n, long x)
{
   if (n <= 0)
        return x;
   return factorial(n-1, x*n);
}
```

递归转非递归

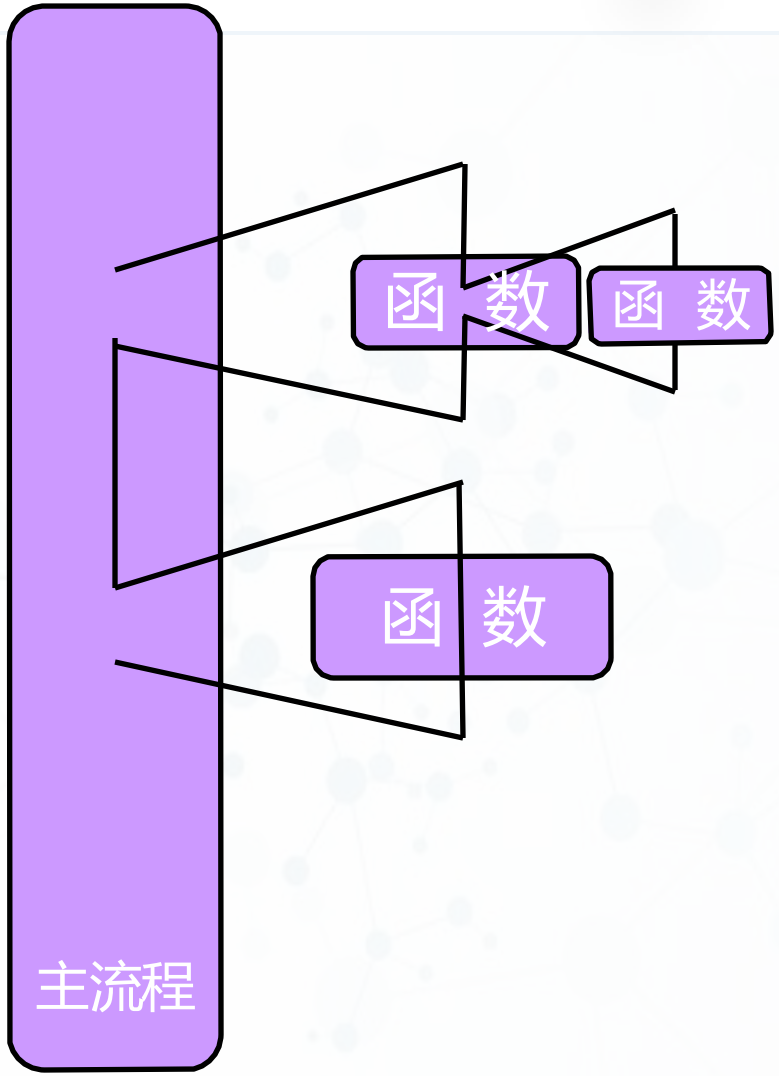
- 尾递归?伪递归?
  - 计算仅占用常量栈空间
  - 命令式语言：编译器可以对尾递归进行优化，没有必要存储函数调用栈信息，不会出现栈溢出 （例如 gcc -O2）
  - 函数式语言：靠尾递归来实现循环

递归转非递归

- 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、全局变量

递归函数调用原理

- 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）



递归的实现

- 一个问题能否用递归实现，看其是否具有下面特点
  - 有递推公式（1个或多个）
  - 有递归结束条件（1个）
- 编写递归函数时，程序中必须有相应的语句
  - 一个（或者多个）递归调用语句
  - 测试结束语句
  - 先测试，后递归调用
- 递归程序的特点
  - 易读、易编，但占用额外内存空间。

函数运行时的存储分配

- 静态分配
  - 在非递归情况下，数据区的分配可以在程序运行前进行，一直到整个程序运行结束才释放，这种分配称为静态分配。
  - 采用静态分配时，函数的调用和返回处理比较简单，不需要每次分配和释放被调函数的数据区
- 动态分配
  - 在递归（函数）调用的情况下，被调函数的局部变量不能静态地分配某些固定单元，而必须每调用一次就分配一份，以存放当前所使用的数据，当返回时随即释放。【大小不确定，值不确定】
  - 动态分配在内存中开辟一个称为运行栈的足够大的动态区

动态存储分配

- 用作动态数据分配的存储区，分为堆（heap）和栈（stack）
  - 堆区域则用于不符合LIFO（诸如指针的分配）的动态分配
  - 栈用于分配发生在后进先出LIFO风格中的数据（诸如函数的调用）

存放全局变量和静态变量

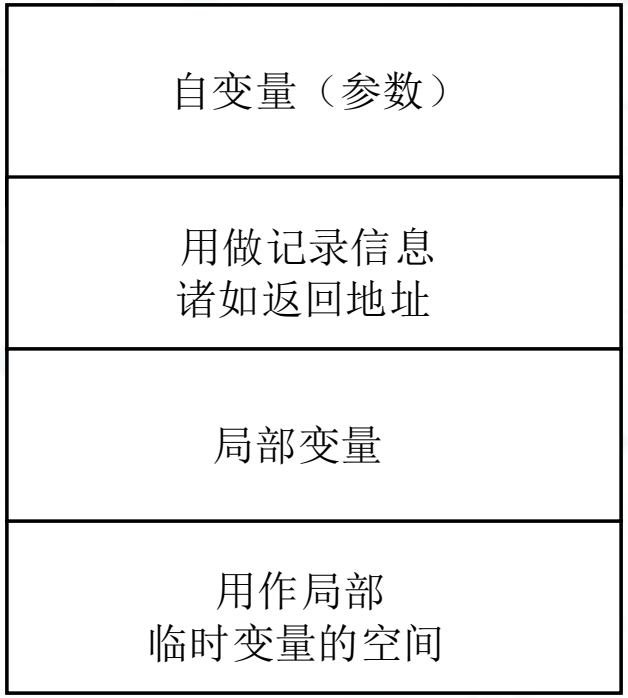
存放函数的形式参数/局部变量，由编译器分配并自动释放

例如Linux下的栈空间大小默认为10M

用malloc或者new申请的内存空间，需要手工释放

运行栈中的活动记录

- 函数活动记录是动态存储分配中的基本单元
  - 当调用函数时，函数的活动记录包含为其局部数据分配的存储空间



运行栈中的活动记录

- 运行栈随着程序执行时发生的调用链或生长或缩小
  - 每次调用执行进栈操作，把被调函数的活动信息压入栈顶
  - 函数返回执行出栈操作，恢复到上次调用所分配的数据区
- 一个函数在运行栈上可以有若干不同的活动记录，每个都代表了一个不同的调用
  - 递归深度决定运行栈中活动记录的数目
  - 同一局部变量在不同的递归层次被分配给不同的存储空间

举例1：阶乘n!

- 阶乘n！的递归定义

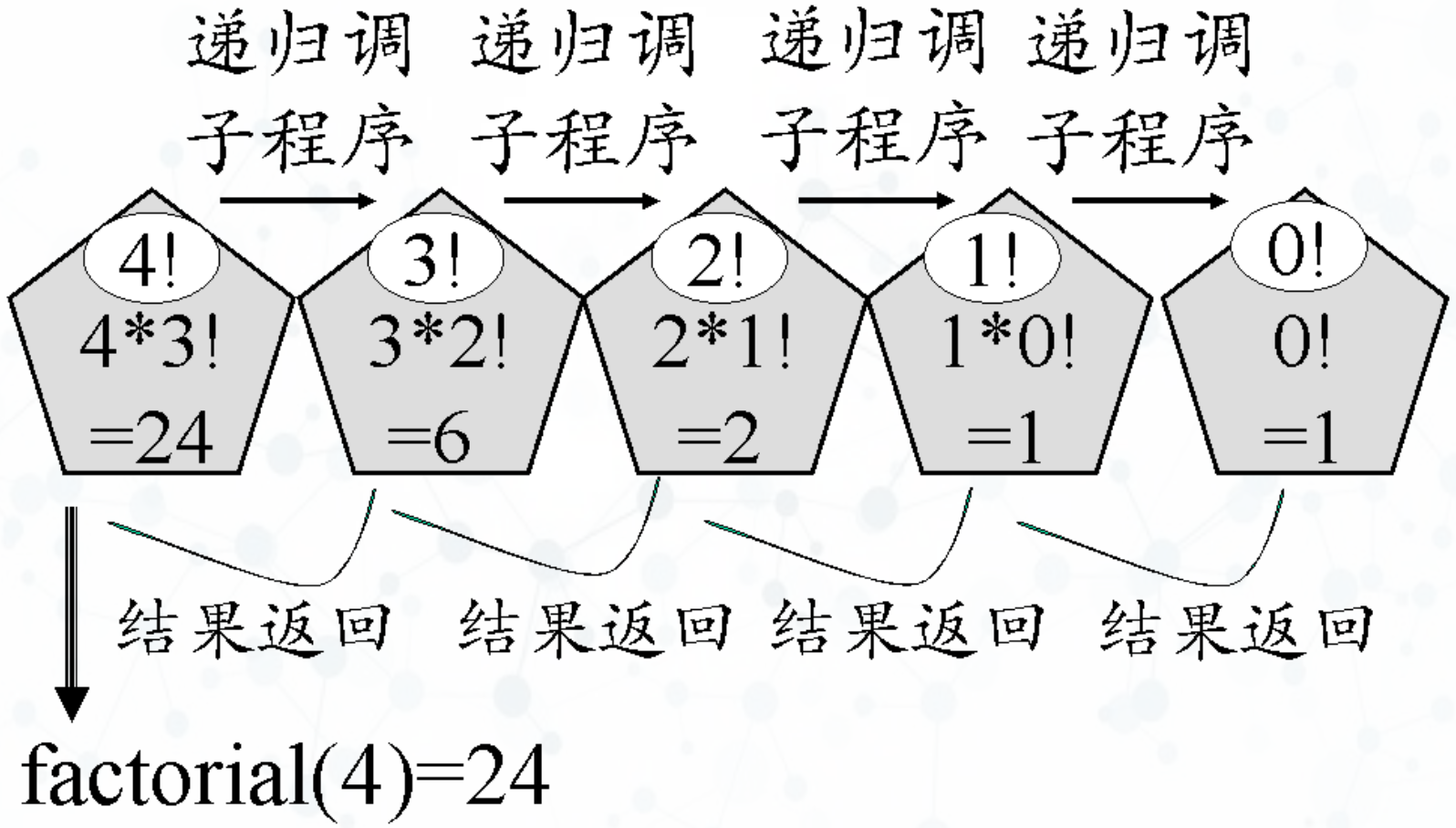
$$factorial(n+1)=\begin{cases}1, & if\ n\le 0\\ n*factorial(n-1), & if\ n>0\end{cases}$$

- 递归入口
  - factorial(n-1)
- 递归出口
  - n≤0

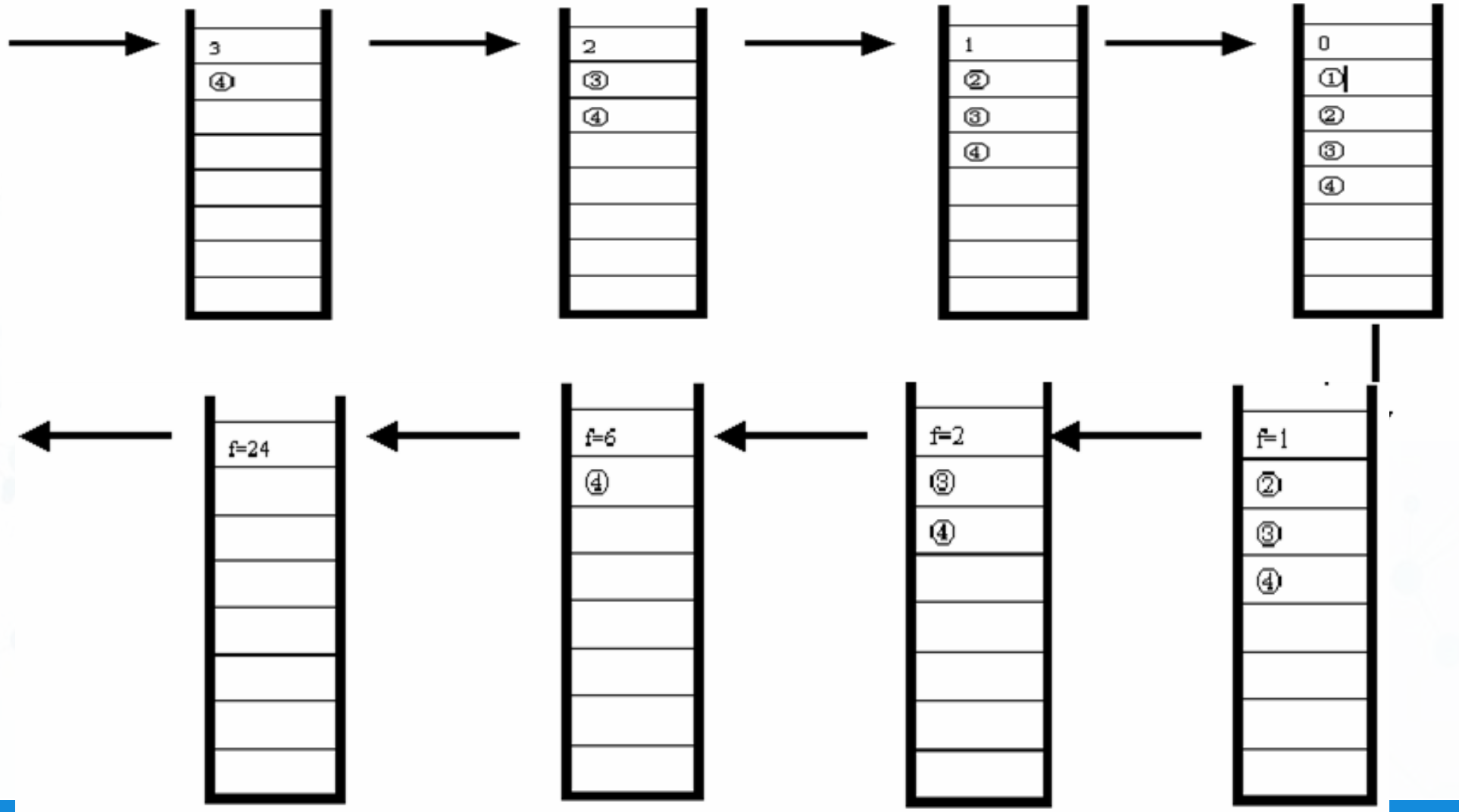
```
//递归函数
long factorial(long n)
{
  if (n==0)
       return 1;
  else
   //递归调用
  return n * factorial( n-1) ;
}
```

```
//非递归函数
long factorial(long n)
{
  int   m = 1;
  int   i ;
  if (n>0)
      for ( i = 1;  i  <= n;  i++ )
              m  = m * i ;
  return m ;
}
```

递归过程示意图(4!)


递归调 递归调 递归调 递归调
子程序 子程序 子程序 子程序
4!
4*3!
=24
3!
3*2!
=6
2!
2*1!
=2
1!
1*0!
=1
0!
0!
=1
结果返回 结果返回 结果返回 结果返回
factorial(4)=24

栈的变化



机械的递归转换

- 递归转非递归的通用方法
  - 1. 设置一工作栈保存当前工作记录
  - 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

机械的递归转换 - 以背包问题为例

[简化的0-1背包问题] 我们有n件物品，物品i的重量为w[i] 。如果限定每种物品（0）要么完全放进背包（1）要么不放进背包；即物品是不可分割的。问：能否从这n件物品中选择若干件放入背包，使其重量之和恰好为s。

机械的递归转换 – 以背包问题为例

$$knap(s,n)=\begin{cases} true, & 当s=0 \\ false, & 当s<0或s>0且n<1 \\ knap(s-w[n-1],n-1)||knap(s,n-1), & 当s>0且n\geq 1 \end{cases}$$

递归算法

```
bool knap (int s, int n) {
    if (s == 0) return true;
    if ((s < 0) || (s > 0 && n < 1)) return false;
    if (knap(s - w[n-1], n-1)) {
        cout << w[n-1];
        return true;
     }
     else return knap(s, n-1);
  }
```

机械的递归转换

- 递归出口
  - 1. s == 0时，不选择任何物品即可，解为true
  - 2. s > 0 且n < 1时，无物品可选，解为false
- 递归规则
  - 1. w[n-1]包含在解中，求解 knap(s – w[n-1], n-1)
  - 2. w[n-1]不包含在解中，求解 knap(s, n-1)

机械的递归转换

- 递归返回的三种情况
  - 1. 计算 knap(s, n) 完毕，返回到调用本函数的其他函数
  - 2. 计算 knap(s – w[n-1], n-1) 完毕，返回到本调用函数
  - 3. 计算 knap(s， n-1) 完毕，返回到本调用函数

机械的递归转换

- 1. 设置一工作栈保存当前工作记录
  - 在函数中出现的所有参数和局部变量都必须用栈中的数据成员代替
    - 返回情况标号
    - 函数参数（值参、引用型）
    - 局部变量

```
enum rdType {0, 1, 2}; //对应三种返回情况
public class knapNode{
    int s, n;           // 背包容量和物品数目
    rdType rd;     // 返回情况标号
    bool k;             // 结果单元
};
```

机械的递归转换

- 2. 设置t+2个语句标号
  - label 0: 第一个可执行语句
  - label t+1: 设置在函数体结束处
  - label i (1 <= i <= t): 第i个递归返回处

机械的递归转换

- 3. 增加非递归入口
  - 将第一个数据成员（递归起点）压入栈

```
Stack<knapNode> stack;
knapNode tmp;
tmp.s = s; tmp.n = n, tmp.rd = 0;
stack.push(tmp);     // 入栈
```

机械的递归转换

- 4. 替换第i (i=1, …, t)个递归规则
  - 若函数体第i (i=1, …, t)个递归调用语句形如recf(a1, a2, …, am);则用以下语句替换:
    - S.push(i, a1, ..., am);
    - goto label0;
  - 并增加标号为i的退栈语句
    - label i: S.pop(&x);
    - //根据取值x进行相应的返回处理

机械的递归转换

- 5. 所有递归出口处增加语句
  - 递归出口增加跳转语句，以进行递归结束后的相关处理
    - `goto label t+1;`

机械的递归转换

- 6. 标号为t+1的语句
  - 标号为 t+1 的语句形如:

```
S.pop(&tmp);
switch (tmp.rd) {
   case 0: return;
   case 1: goto label1; break;
   // ……
   cast t: goto labelt; break;
   default: break;
}
```

机械的递归转换

$$knap(s,n)=\begin{cases} true, & 当s=0 \\ false, & 当s<0或s>0且n<1 \\ knap(s-w[n-1],n-1)||knap(s,n-1), & 当s>0且n\geq 1 \end{cases}$$

递归算法

```
bool knap (int s, int n) {
    if (s == 0) return true;                                    // label0
    if ((s < 0) || (s > 0 && n < 1)) return false;
    if (knap(s - w[n-1], n-1)) {                                // label1
        cout << w[n-1];
        return true;
    }
    else return knap(s, n-1);                                   // label2
}
```

```
bool nonRecKnap(int s, int n) {
    tmp.s = s; tmp.n = n, tmp.rd = 0;
    stack.push(tmp);     // 非递归调用入口
label 0:                 // 递归调用入口
    stack.pop(&tmp);     // 查看栈顶并分情况处理
    if (tmp.s == 0) {    // 若满足递归出口条件
        tmp.k = true;
        stack.push(tmp);
        goto label3;     // 转向递归出口处理
    }
    if ((tmp.s < 0) || (tmp.s >0 && tmp.n < 1)) {
        tmp.k = false;
        stak.push(tmp);
        goto label3;
    }
    stack.push(tmp);     // 尚未满足出口条件
    x.s = tmp.s – w[tmp.n-1]; x.n = tmp.n – 1; x.rd
    = 1;
    stack.push(x);       // 按照规则1进行压栈
    goto label0;
label1:                  // 规则1对应的返回处理
    stack.pop(&x);  // 查看栈顶并分情况处理
    if (tmp.k == true) { // 某层结果单元为true
        x.k = true;      // 把true上传给调用层
        stack.push(x);
        cout << w[x.n-1] << endl; // 输出物品
        goto label3;
    }
    stack.push(x);       // 某层结果单元为false
    tmp.s = x.s; tmp.n = x.n – 1; tmp.rd = 2;
    stack.push(tmp);  // 回溯，应用规则2
    goto label0;
label2:                  // 规则2对应的返回处理
    stack.pop(&x);
    x.k = tmp.k;         // 结果单元上传给调用层
    stack.push(x);
label3:                  //递归出口处理
    stack.pop(&tmp);
    switch(tmp.rd) {
        case 0: return tmp.k;  // 算法结束
        case 1: goto label1;
        case2: goto label2;
    }
}
```