数据结构

谢浩哲
cshzxie@gmail.com

数据结构概述

- 在计算机科学中,数据结构(Data Structure)是计算机中存储、组织数据的方式.通常情况下,精心选择的数据结构可以带来最优效率的算法.
- 常见的数据结构:
  - 堆栈(Stack)
  - 队列(Queue)
  - 链表 (Linked List)
  - 树(Tree)
  - 堆(Heap)
  - 图(Graph)

数据结构

[Array]数组

数组(Array)

- **什么是数组?**
  - 假设我们需要存储一系列具有相同类型数据(Integer)，计算这一系列数值的平均值，并且记录有多少数值大于平均值。
  - 如果没有数组，我们是这样做的...

数组(Array)

- **什么是数组?**
  - 假设我们需要存储一系列具有相同类型数据(Integer)，计算这一系列数值的平均值，并且记录有多少数值大于平均值。
  - 如果没有数组，我们是这样做的...

```
var
   n1, n2, n3, ..., n100: integer;
   average: integer;
   count: integer;
begin
   read(n1, n2, n3, ..., n100);
   average:= (n1 + n2 + ... + n100) / 100;
   if (n1 > average) then Inc(count);
   if (n2 > average) then Inc(count);
   ......
   if (n100 > average) then Inc(count);
   writeln(average, count);
end.
```

数组(Array)

- **什么是一维数组?**
  - 一维数组是存储于计算机的连续存储空间中的多个具有统一类型的数据元素。
  - 同一数组的不同元素通过不同的下标标识。

  $(a_1, a_2, a_3, \ldots, a_n)$
- **如何声明一维数组?**
  - Pascal描述

```
var
    a:array[1..100] of integer;
```

  - C/C++语言描述

```
int a[100];
```

数组(Array)

- **对数组进行初始化**
  - Pascal描述
    - 使用循环进行初始化

```
for i:= 1 to n do
    a[i]:= 0;
```

    - 使用fillchar()函数

```
fillchar(a, sizeof(a), 0);
```

  - C/C++描述
    - 在声明时进行初始化

```
int a[100] = {0};
```

    - 使用循环进行初始化

```
for (i = 0; i < n; i++)
    a[i] = 0;
```

数组(Array)

- **在数组中插入元素**
  - 在Array[Index]处插入元素Element
  - Array:[1..SIZE]: integer; [已存在n个元素(n < SIZE)]
  - Pascal描述

```
for i:= n downto Index do
   Array[i + 1]:= Array[i];
Array[Index]:= Element;
```

数组(Array)

- **在数组中插入元素**
  - 在Array[Index]处插入元素Element
  - Array:[1..SIZE]: integer; [已存在n个元素(n < SIZE)]
  - C/C++描述

```
for (int i = n - 1; i >= Index; i -- )
    Array[i + 1] = Array[i];
Array[Index] = Element;
```

数组(Array)

- **在数组中删除元素**
  - 删除在Array[Index]处的元素
  - Array:[1..SIZE]: integer; [已存在n个元素(n < SIZE)]
  - Pascal描述

```
for i:= (Index + 1) to n do
    Array[i - 1]:= Array[i];
Array[n]:= 0;
```

数组(Array)

- **在数组中删除元素**
  - 删除在Array[Index]处的元素
  - Array:[1..SIZE]: integer; [已存在n个元素(n < SIZE)]
  - C/C++描述

```
for (int i = Index + 1; i < n; i ++)
    Array[i - 1] = Array[i];
Array[n - 1] = 0;
```

数组(Array)

- **使用一维数组**
  - **计算斐波那契数列的前n项(n ≤ 30)**
    - 1, 1, 2, 3, 5, 8...
    - 对于第n项(n ≥3)有：$a_n = a_{n-1} + a_{n-2}$
  - Pascal描述

```
for i:= 3 to n do
    a[i]:= a[i - 1] + a[i - 2];
```

  - C/C++描述

```
for (i = 3; i <= n; i ++)
    a[i]= a[i - 1] + a[i - 2];
```

数组(Array)

- 使用一维数组
  - 计算斐波那契数列的前n项
    - 计算结果如下(C/C++)：
    - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...
    - 当 n > 40时

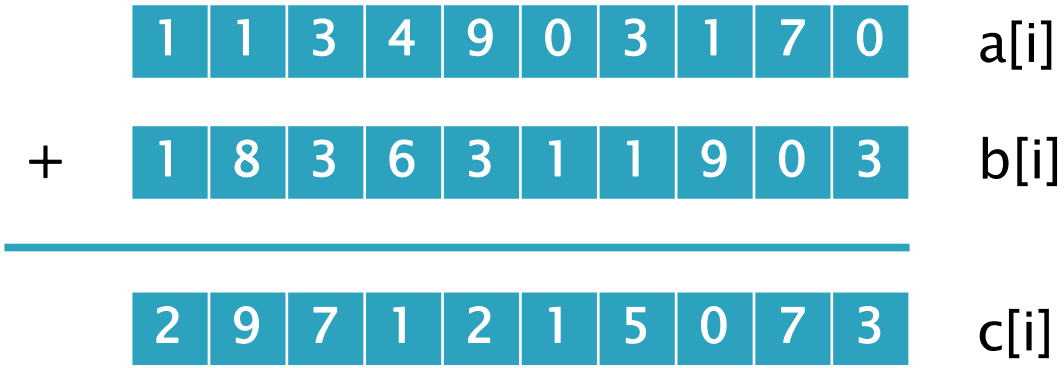
41: 165580141
42: 267914296
43: 433494437
44: 701408733
45: 1134903170
46: 1836311903
47: -1323752223

正确结果:2971215073

在Pascal中 会提示
Error Code: 206

数组(Array)

- **使用一维数组**
  - 高精度算法(High-Precision)
    - 高精度加法



- Pascal描述
  - c[i]:= a[i] + b[i] + x;
  - x:= (a[i] + b[i]) **div** 10;
  - c[i]:= c[i] **mod** 10;

数组(Array)

- **什么是多维数组?**
  - 多维数组类似于数学上的矩阵，数学上的矩阵：

$$A = \begin{pmatrix} 2 & 4 & 8 \\ 4 & 16 & 64 \end{pmatrix}$$

  - 计算机中的多维数组：
  - Pascal描述

```
var
    a:array[1..2, 1..3] of integer;
const
   a:array[1..2, 1..3] of integer = ((2, 4, 8), (4, 16, 64));
```

  - C/C++描述

```
int  a[2][3] ={
                    {2, 4, 8},
                    {4, 16, 64},
                } ;
```

数组(Array)

- **什么是多维数组?**
  - 普通数组采用一个整数来作下标。我们在多维数组之中采用一系列有序的整数来标注，如在[3, 1, 5] 。这种整数列表之中整数的个数始终相同，且被称为数组的“维度”。关于每个数组维度的边界称为“维”。维度为*k*的数组通常被称为*k*维。

- **使用多维数组**
  - 输入4名学生语文数学、英语、物理、化学五门课的考试成绩，求出每名学生的平均分，打印出表格。

数组(Array)

- **使用多维数组**
  - 引用二维数组
  - 计算机中一般使用双重循环来存取二维数组中的元素：
  - Pascal描述

```
for i:= 1 to m do
begin
    for j:= 1 to n do
        write(a[i][j], ' '); //write(a[i, j], ' ');
    writeln();
end;
```

数组(Array)

- **使用多维数组**
  - 引用二维数组
  - 计算机中一般使用双重循环来存取二维数组中的元素：
  - C语言描述

```
for (i = 0; i < m; i ++)
{
    for (j = 0; j < n; j ++)
        printf("%d ", a[i][j]);
    printf("\n");
}
```

数组(Array)

- **使用多维数组**
  - 引用二维数组
  - 计算机中一般使用双重循环来存取二维数组中的元素：
  - C++描述

```
for (i = 0; i < m; i ++)
{
    for (j = 0; j < n; j ++)
        cout << a[i][j] << " ";
    cout << endl;
}
```

数组(Array)

- **数组在计算机中的存储**
  - 在Pascal、C、C++等语言中，数组形式上依赖内存分配而成的，所以必须在使用前预先请求空间。这使得数组有以下特性：
    - 1.请求空间以后大小固定，不能再改变(数据溢出问题)；
    - 2.在内存中有空间连续性的表现，中间不会存在其他程序需要调用的数据，为此数组的专用内存空间；
    - 3.在某些编程语言中(如C语言和C++)，程序不会对数组的操作做下界判断，也就有潜在的越界(Out Of Bound)风险。

数组(Array)

- 一维数组在计算机中的存储
  - 数组的物理实现是一块连续的存储空间，计算机通过首地址的位移来访问数组中的元素
  - For Example:
    - a:array[0..9] of longint;
    - Address of a = Address of a[0]
    - Address of a[1] = Address of a[0] + 1 × sizeof(longint)
    - .....
    - Address of a[n] = Address of a[0] + n × sizeof(longint)
  - 数组中的每一个元素都相当于一个与数组同类型的普通变量.所以,访问数组的一个元素就如同访问一个变量,关键就在于要能找到该数组元素的地址

数组(Array)

- **多维数组在计算机中的存储**
  - 由于计算机内存是一维的，多维数组的元素应排成线性序列后存入存储器
  - 1) 行优先存储(Pascal、C、C++等)
    - 将数组元素按行向量排列，第i+1个行向量紧接在第i个行向量后面
    - 【例】二维数组A_{mn}的按行优先存储的线性序列为：

      $a_{11},a_{12},\ldots,a_{1n},a_{21},a_{22},\ldots,a_{2n},\ldots\ldots,\ a_{m1},a_{m2},\ldots,\ a_{mn}$
  - 2) 列优先存储(FORTRAN等)
    - 将数组元素按列向量排列，第i+1个列向量紧接在第i个列向量后面。
    - 【例】二维数组A_{mn}的按列优先存储的线性序列

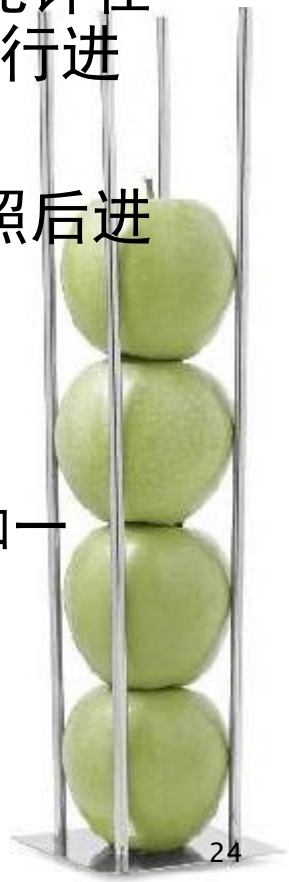
      $a_{11},a_{21},\ldots,a_{m1},a_{12},a_{22},\ldots,a_{m2},\ldots\ldots,\ a_{1n},a_{2n},\ldots,\ a_{mn}$

数据结构

[Stack]堆栈

堆栈(Stack)

- **什么是堆栈？**
  - 堆栈(Stack)，也可直接称栈。在计算机科学中，是一种特殊的串行形式的数据结构，它的特殊之处在于只能允许在链结串行或阵列的一端(称为堆栈顶端指标，Top)进行进栈(Push)和出栈(Pop)操作。
  - 由于堆栈数据结构只允许在一端进行操作，因而按照后进先出(LIFO, Last In First Out)的原理运作。
  - 堆栈数据结构使用两种基本操作：
    - 进栈(Push)：将数据放入堆栈的顶端，堆栈顶端指标加一
    - 出栈(Pop)： 将堆栈顶端数据输出，堆栈顶端指标减一

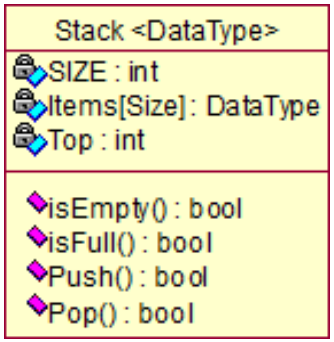


堆栈(Stack)

- **如何定义堆栈?**
- Pascal描述

```
var
    Stack: array[1..SIZE] of integer;
    Top: integer;
```

Stack

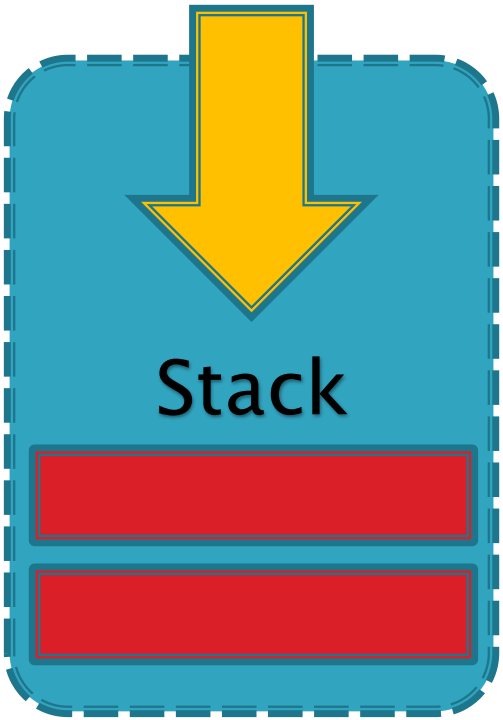


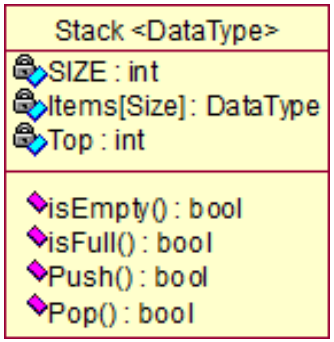
堆栈(Stack)

- 如何定义堆栈?
- Pascal描述

```
var
   Stack: array[1..SIZE] of integer;
   Top: integer;

Procedure Push(n: integer)
Begin
    if Top = SIZE then writeln('OverFlow!');
    else
        begin
            Top:= Top + 1; Stack[Top] = n;
        end;
End;
```



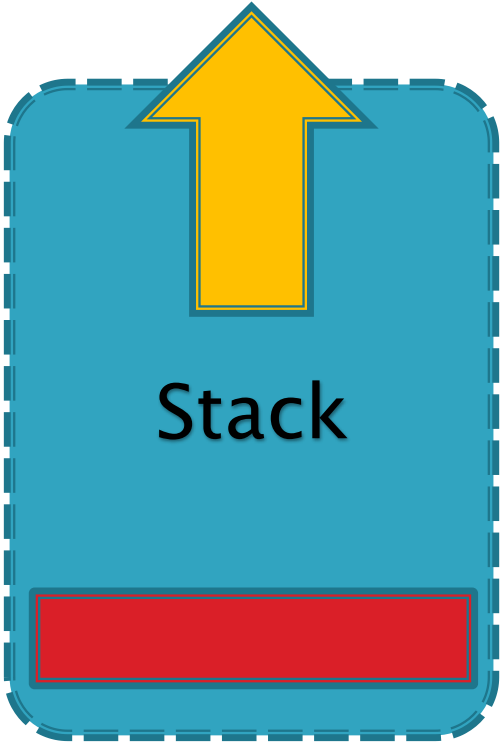


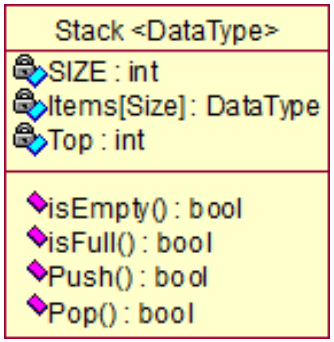
堆栈(Stack)

- 如何定义堆栈?
- Pascal描述

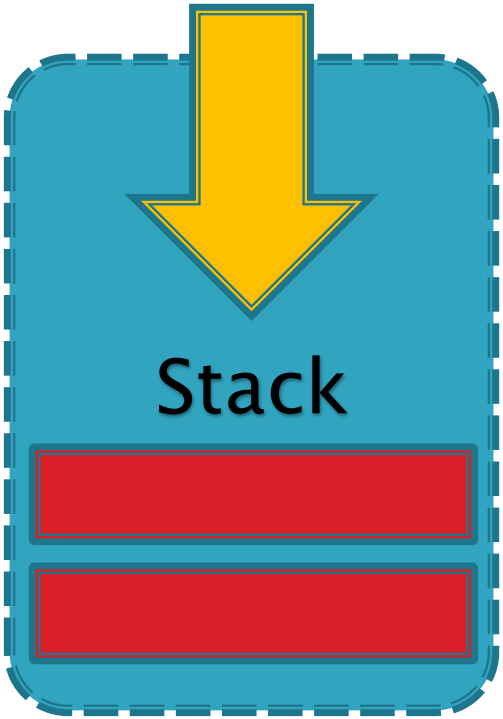
```
var
   Stack: array[1..SIZE] of integer;
   Top: integer;

Function Pop: integer
Begin
    if Top = 0 then writeln('UnderFlow!');
    else
        begin
            Pop:= Stack[Top]; Top:= Top - 1;
        end;
End;
```





堆栈(Stack)

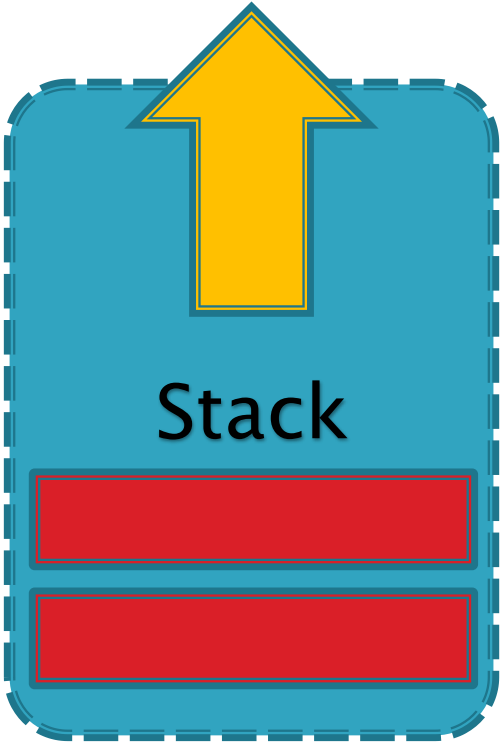


- **如何定义堆栈?**
- C/C++描述

```
int Stack[SIZE] = {0};
int Top = 0;

bool Push(const int &n)
{
    if (Top >= Size) return false;
    else
    {
        Stack[Top++] = n; return true;
    }
}
```

堆栈(Stack)



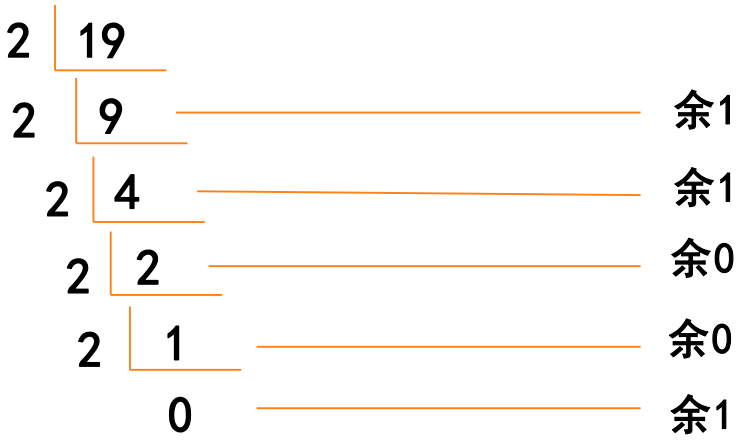
- **如何定义堆栈?**
- C/C++描述

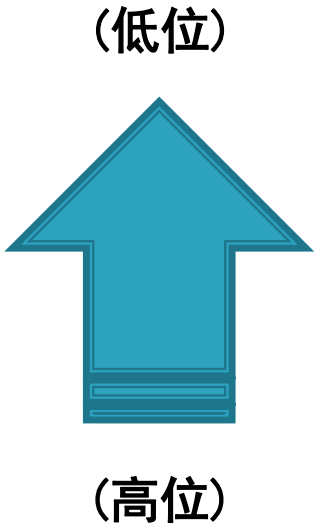
```
int Stack[SIZE] = {0};
int Top = 0;

bool Pop(int &n)
{
    if (Top <= 0) return false;
    else
    {
        n = Stack[--Top]; return true;
    }
}
```

进制间的转换

- **使用堆栈**
  - 将N(Integer)转换成二进制数
  - 如何将一个数转化为二进制？
    - 纯整数部分的转换: 除2取余





    - $\therefore (19)_{10}=(10011)_2$

堆栈(Stack)

- **使用堆栈**
  - 将N(Integer)转换成二进制数

Pascal描述**(Use Array)**

```
while (n > 0) do
begin
    a[i]:= n mod 2;
    n:= n div 2;
    i:= i + 1;
end;

for i:= trunc(log2n) + 1 downto 1 do //倒序输出
    write(a[i]);
```

堆栈(Stack)

- **使用堆栈**
  - 将N(Integer)转换成二进制数

Pascal描述**(Use Stack)**

```
while (n > 0) do
begin
    Push(n mod 2);
    n: = n div 2;
end;

while (IsEmpty = false) do
    write(Pop);
```

堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **判断表达式中括号是否匹配**

      Input:　　12 + [4 * (2 + 3) / 2] – 4

      Output:　Yes

  - 判定原理
    - 使用堆栈进行判定，将‘(’，‘[’，或‘{’入栈；当遇到‘}’，‘]’，或‘)’时，检查当前栈顶元素是否是对应的‘(’，‘[’，或‘{’；若是则弹出元素，否则返回表示不配对。当整个算术表达式检查完毕时，若栈为空则表示括号配对正确，否则不配对。

堆栈(Stack)

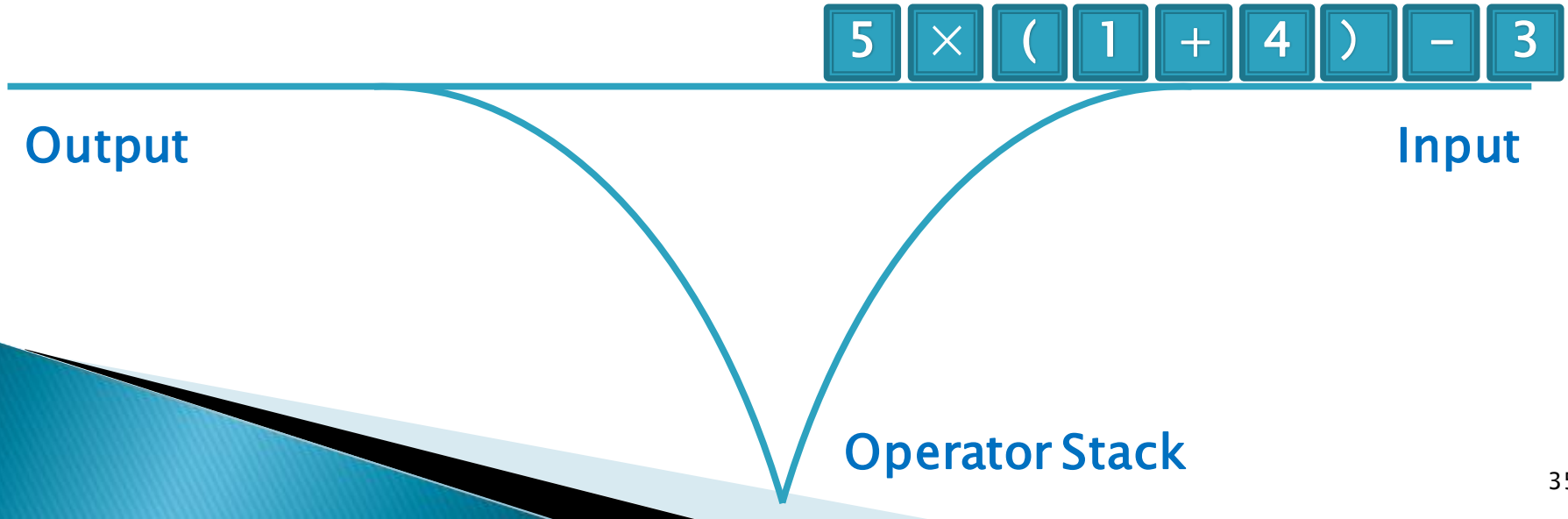
- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**
    - 什么是后缀表达式？
      - 在逆波兰记法(Reverse Polish notation, RPN)中，所有操作符置于操作数的后面，因此也被称为后缀表示法．逆波兰记法不需要括号来标识操作符的优先级．
      - 例如：
        - 中缀表示法：3 + 4
        - 前缀表示法：+ 3 4
        - 后缀表示法：3 4 +

堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**

      Input: 5 × (1 + 4) − 3

      Output: 5 1 4 + × 3 −
    - **Shunting Yard (调度场)算法**

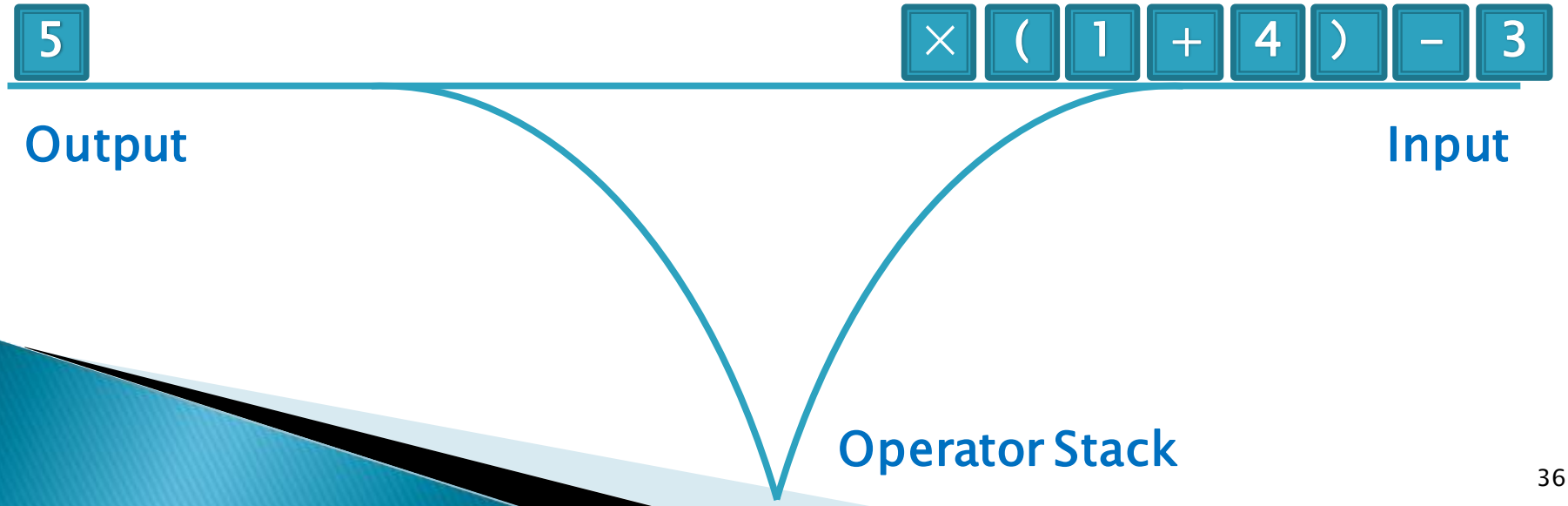


堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**

      Input: 5 × (1 + 4) − 3

      Output: 5 1 4 + × 3 −
    - **Shunting Yard (调度场)算法**

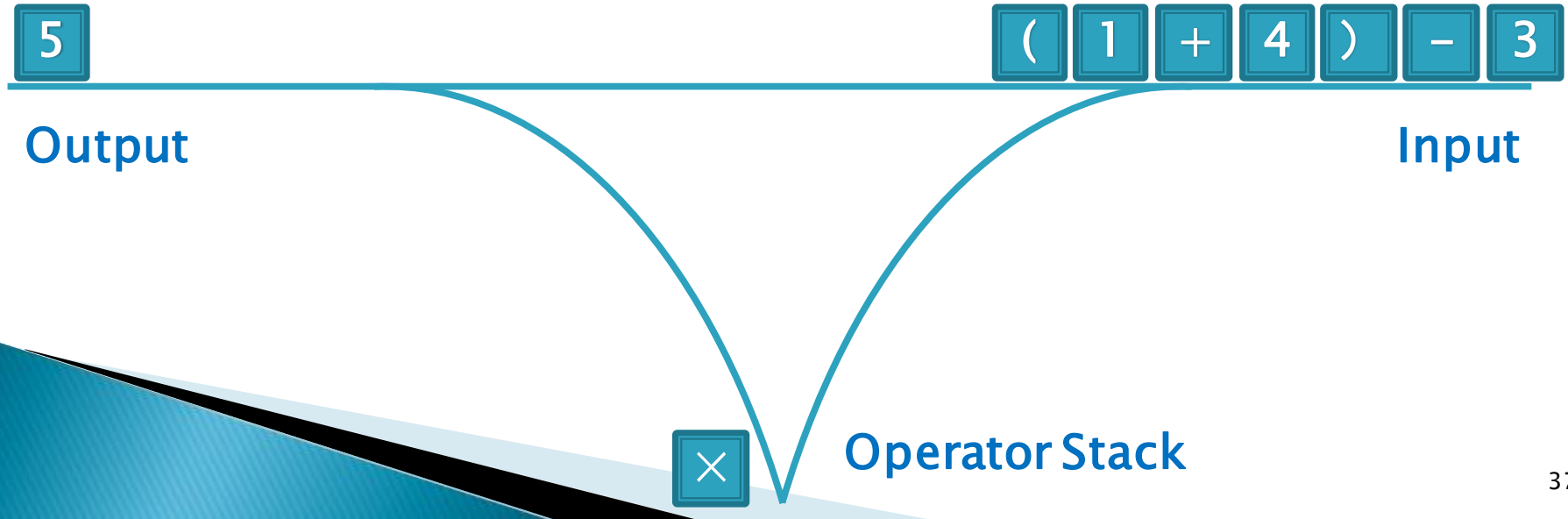


堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**

      Input: 5 × (1 + 4) − 3

      Output: 5 1 4 + × 3 −
    - **Shunting Yard (调度场)算法**

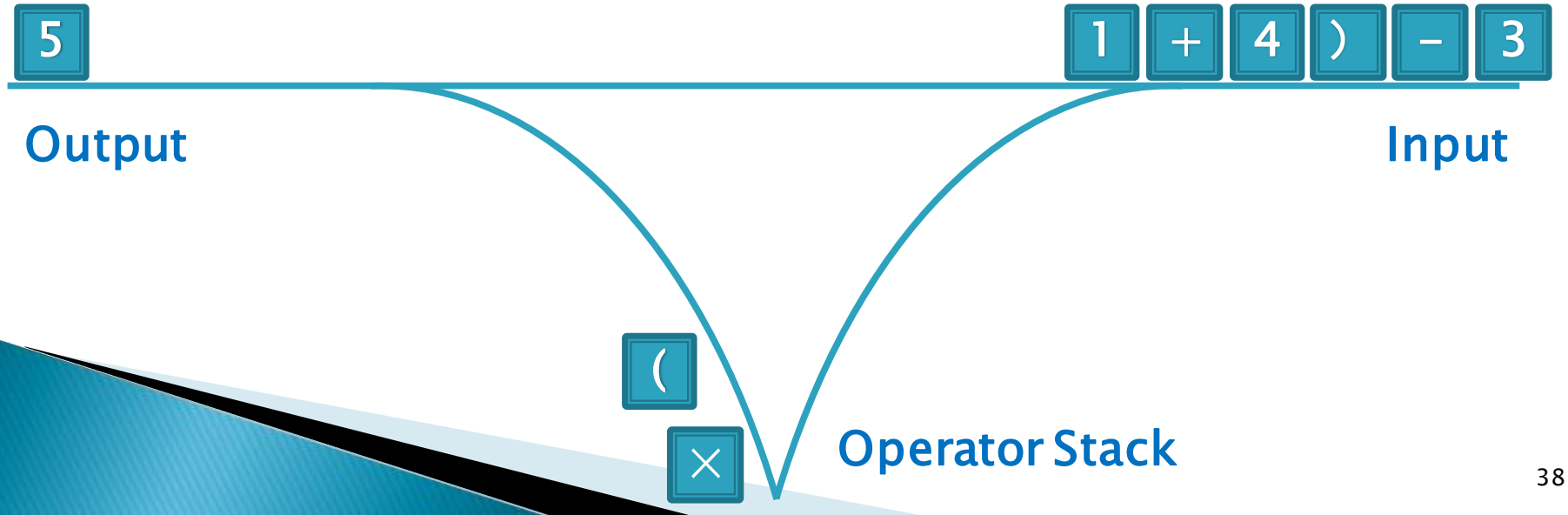


堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**

      Input: 5 × (1 + 4) − 3

      Output: 5 1 4 + × 3 −
    - **Shunting Yard (调度场)算法**

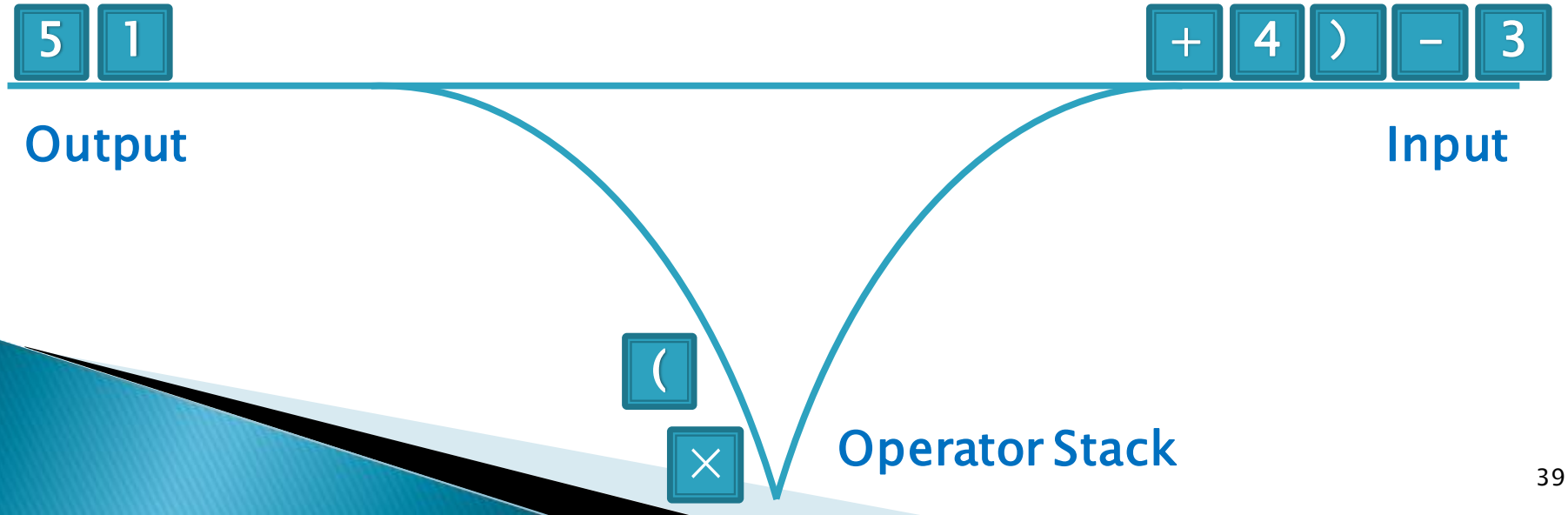


堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**

      Input: 5 × (1 + 4) − 3

      Output: 5 1 4 + × 3 −
    - **Shunting Yard (调度场)算法**

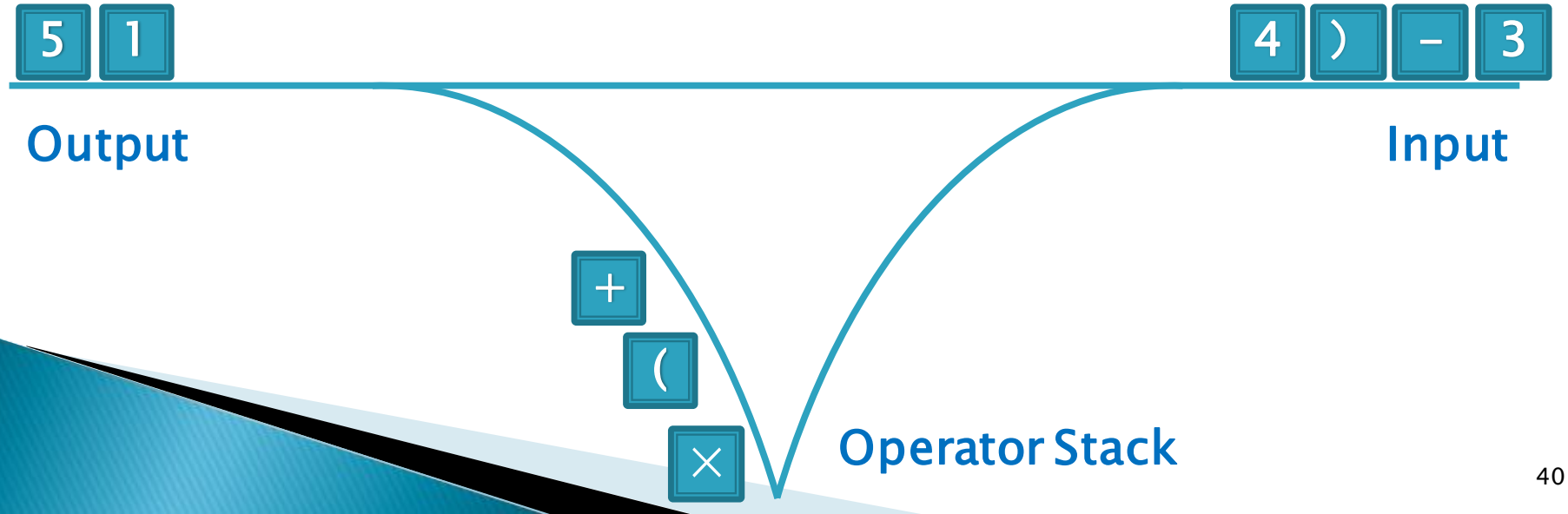


堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**

      Input: 5 × (1 + 4) − 3

      Output: 5 1 4 + × 3 −
    - **Shunting Yard (调度场)算法**

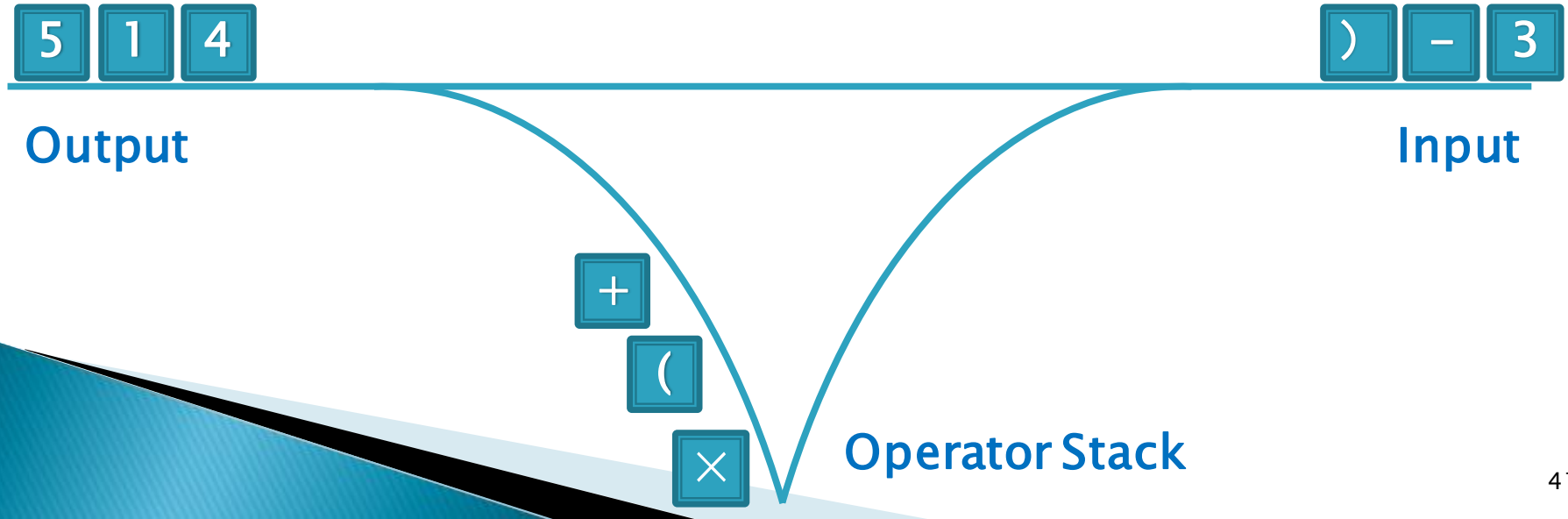


堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**

      Input: 5 × (1 + 4) − 3

      Output: 5 1 4 + × 3 −
    - **Shunting Yard (调度场)算法**

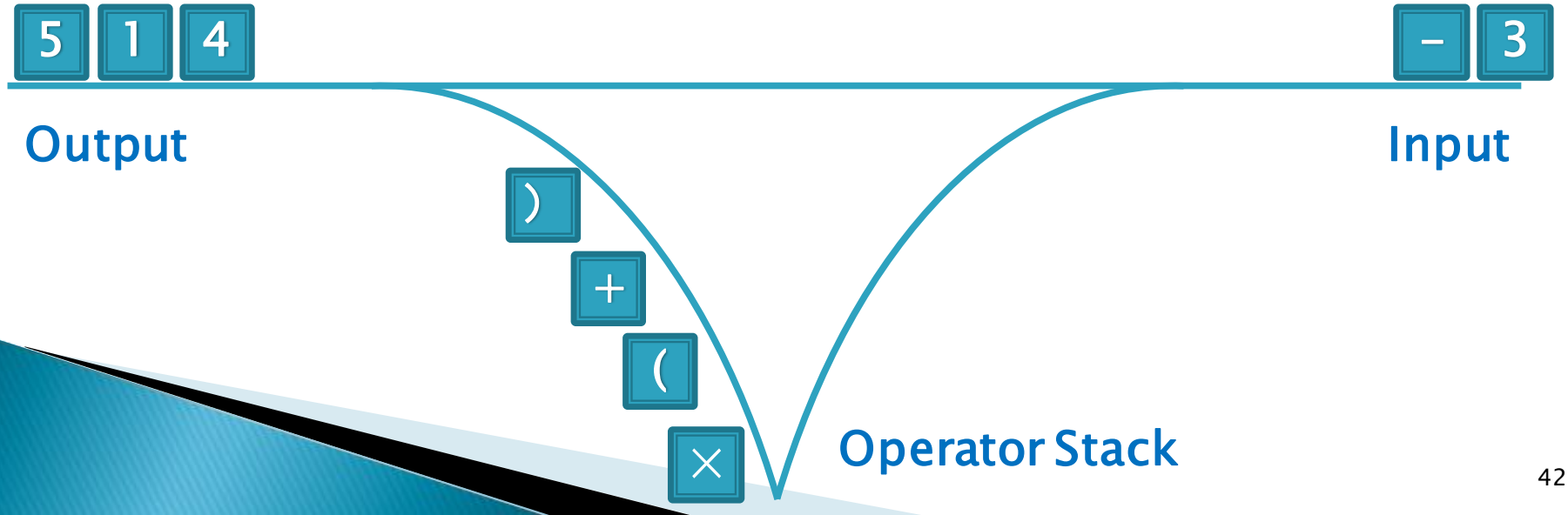


堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**

      Input: 5 × (1 + 4) − 3

      Output: 5 1 4 + × 3 −
    - **Shunting Yard (调度场)算法**

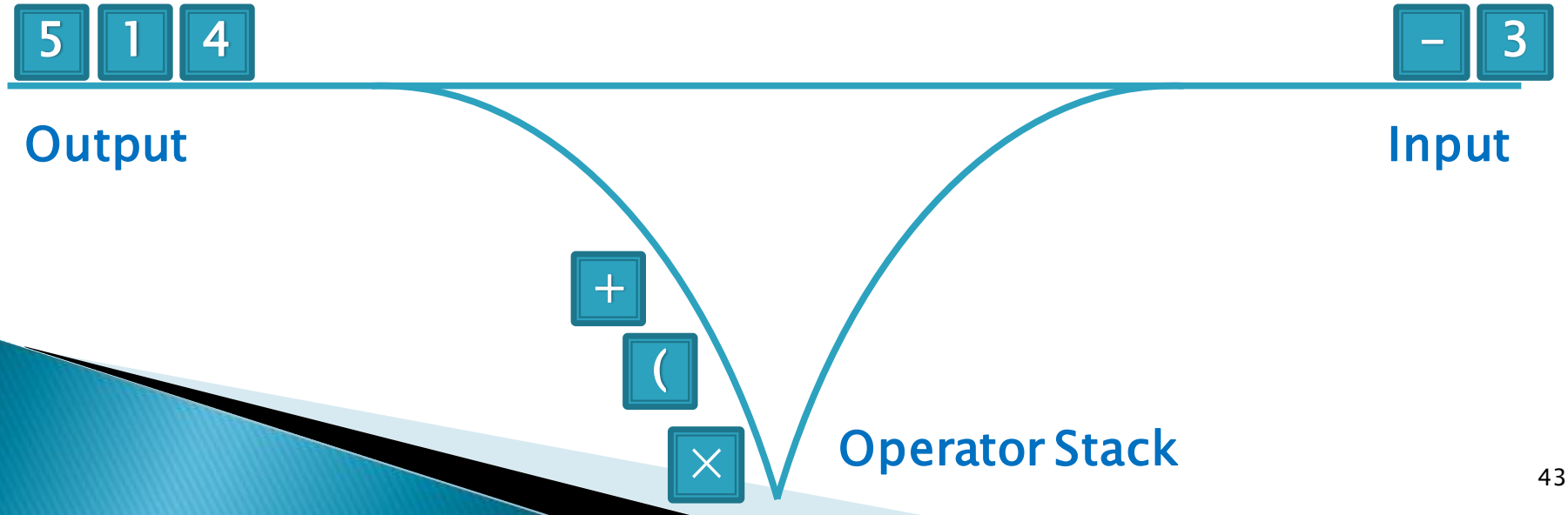


堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**

      Input: 5 × (1 + 4) − 3

      Output: 5 1 4 + × 3 −
    - **Shunting Yard (调度场)算法**

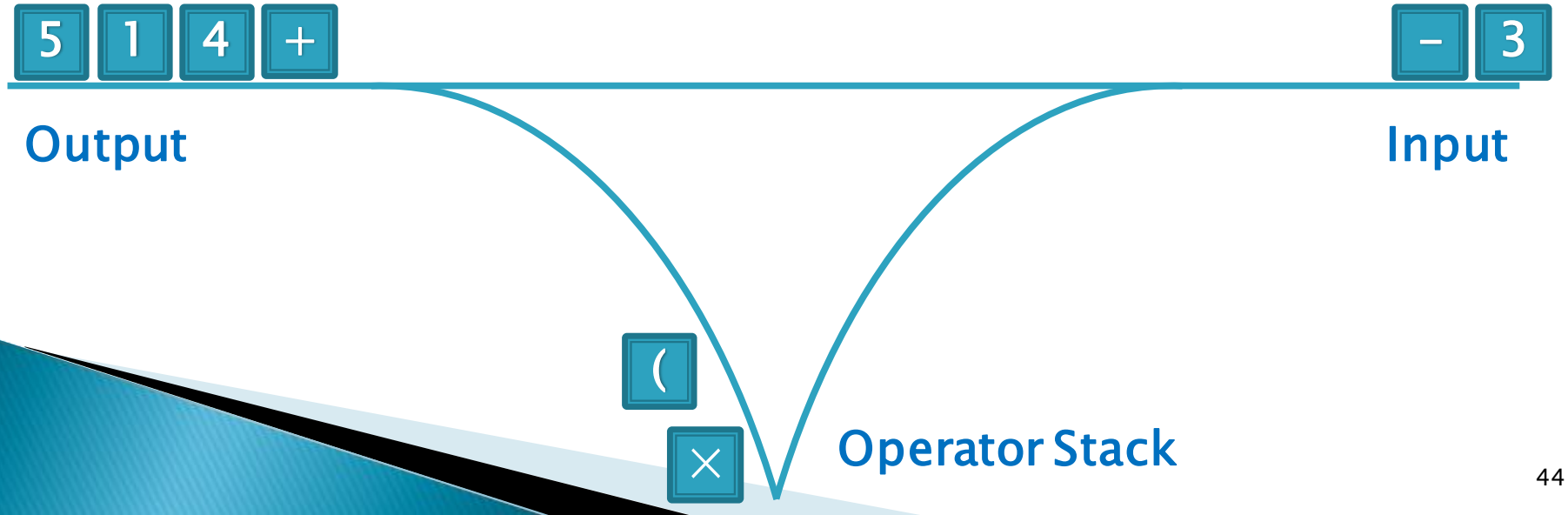


堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**

      Input: 5 × (1 + 4) − 3

      Output: 5 1 4 + × 3 −
    - **Shunting Yard (调度场)算法**

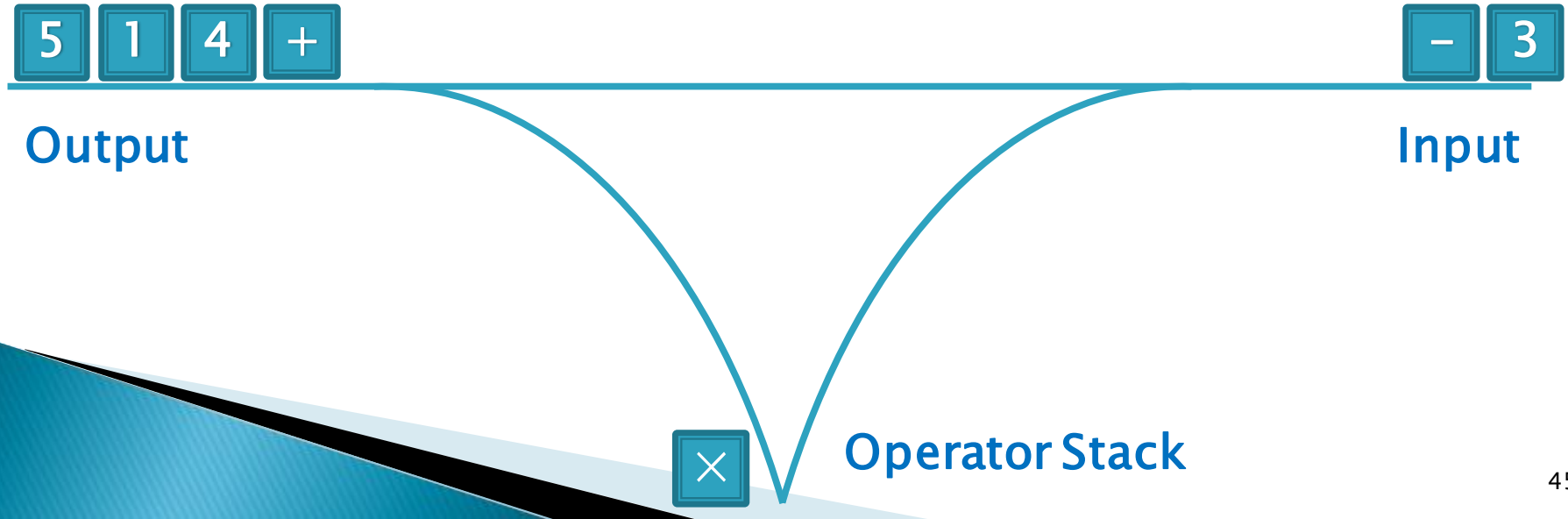


堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**

      Input: 5 × (1 + 4) − 3

      Output: 5 1 4 + × 3 −
    - **Shunting Yard (调度场)算法**

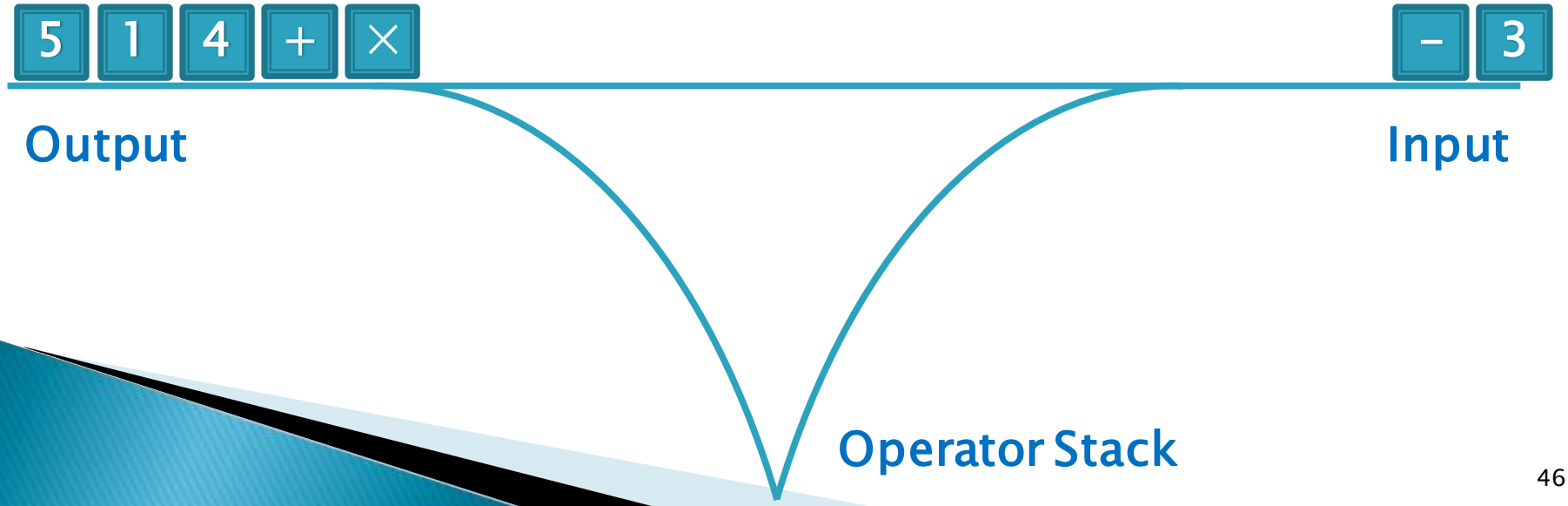


堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**

      Input: 5 × (1 + 4) − 3

      Output: 5 1 4 + × 3 −
    - **Shunting Yard (调度场)算法**

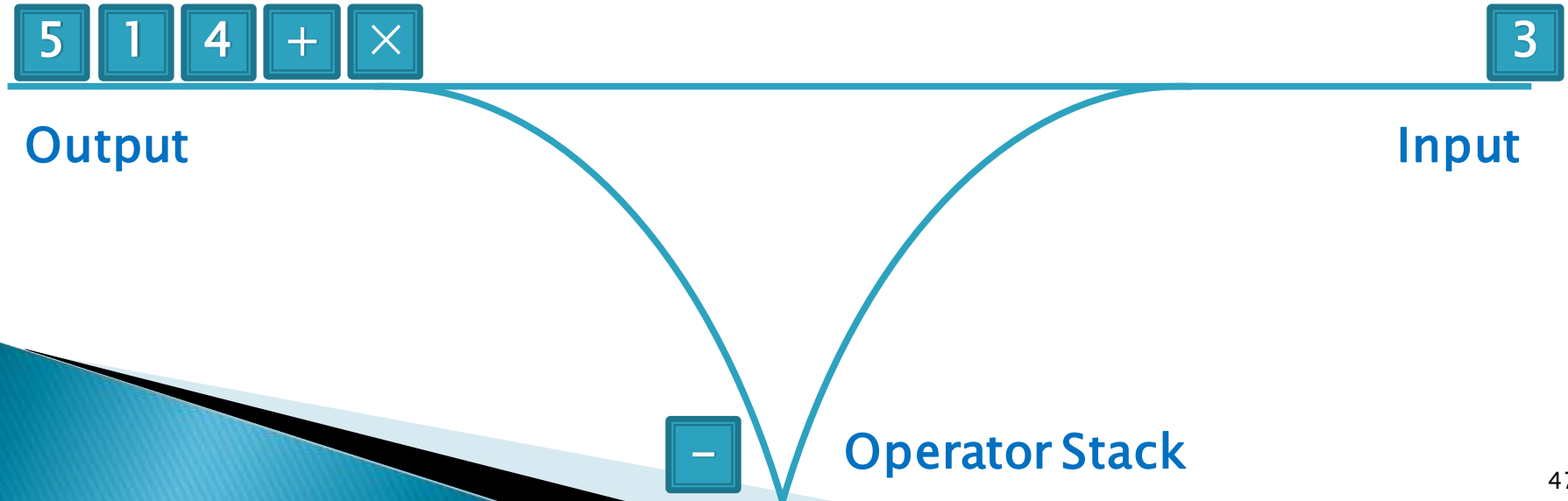


堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**

      Input: 5 × (1 + 4) − 3

      Output: 5 1 4 + × 3 −
    - **Shunting Yard (调度场)算法**

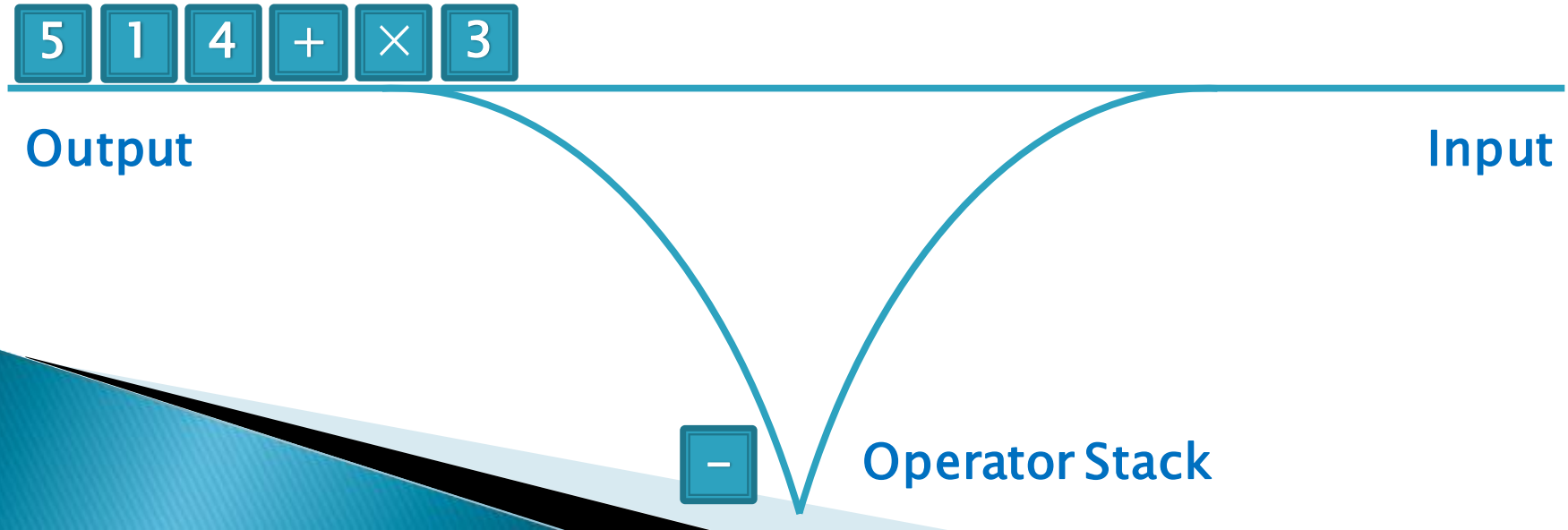


堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**

      Input: 5 × (1 + 4) − 3

      Output: 5 1 4 + × 3 −
    - **Shunting Yard (调度场)算法**

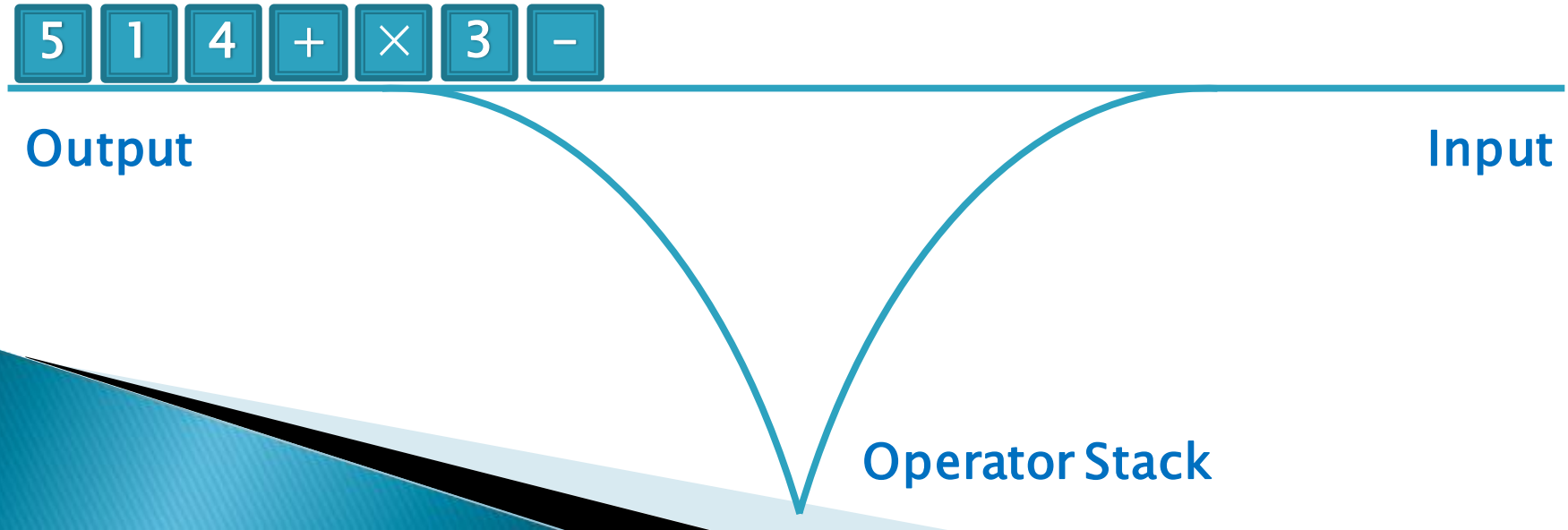


堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**

      Input: 5 × (1 + 4) − 3

      Output: 5 1 4 + × 3 −
    - **Shunting Yard (调度场)算法**



堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **Shunting Yard (调度场)算法描述**
    - 当还有记号可以读取时：
      - 读取一个记号.
      - 如果这个记号表示一个数字, 那么将其添加到输出队列中.
      - 如果当前运算符为'(', 直接入栈；
      - 如果当前运算符为')', 将'('之前的元素全部出栈.
      - 如果为其它运算符, 比较运算符栈栈顶元素与当前元素的优先级：
        - 如果栈顶元素是'(', 当前元素直接入栈；
        - 如果栈顶元素优先级>=当前元素优先级, 出栈并顺序输出运算符直到栈顶元素优先级<当前元素优先级, 然后当前元素入栈；
        - 如果栈顶元素优先级<当前元素优先级, 当前元素直接入栈.
      - 重复第三点直到表达式扫描完毕.
    - 顺序出栈并输出运算符直到栈元素为空.

堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **中缀表达式与后缀表达式的转换**

      Input: 5 + ((1 + 2) * 4) − 3

      Output: 5 1 2 + 4 * + 3 −
    - **后缀表达式的作用**
      - 将复杂表达式转换为可以依靠简单的操作得到计算结果的表达式
      - 这样就能通过入栈和出栈求解任何普通表达式的运算. 如果当前字符为变量或者为数字, 则压栈, 如果是运算符, 则将栈顶两个元素弹出作相应运算, 结果再入栈, 最后当表达式扫描完后, 栈里的就是结果.

堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**

      Input:　　12 + [4 * (2 + 3)　/ 2] - 4

      Output:　18

  - 求值原理
    - 建立两个栈，一个是数据栈，一个是计算符号栈.
      - **规定运算符的优先顺序**
        - 符号优先级如下:
        - 1. '(' Or ')'　　-1
2. '+' Or '-'　　0
3. '*' Or '/'　　1
        - 数值越大, 优先级越高, **同级别的比较时 优先计算先出现的运算符**.

堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**

      Input:　　12 + [4 * (2 + 3) / 2] - 4

      Output:　18

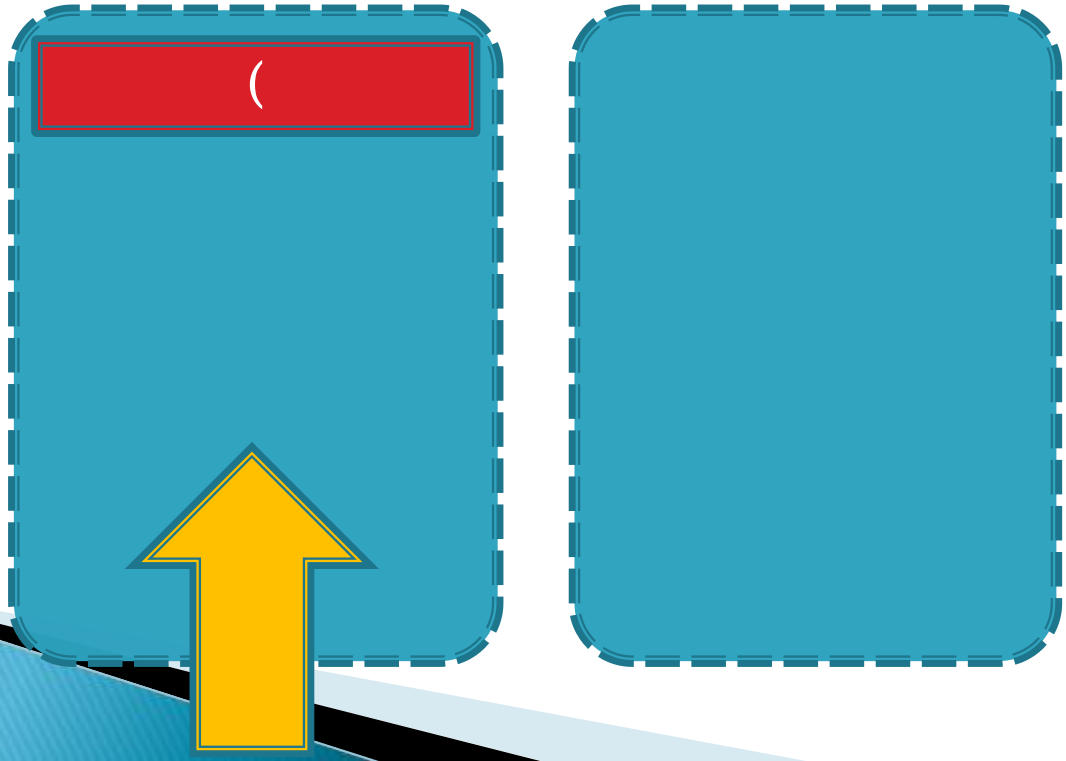
  - 求值原理
    - 建立两个栈，一个是数据栈，一个是计算符号栈.
      - **计算条件**
        - 当前运算符不等于‘(’ Or ‘)’...
        - 出栈口的运算符优先级高于将要入栈的运算符时或者两者可对消时

堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**
      - 以表达式 (6 + 2 * 5) / 4 为例说明计算过程：

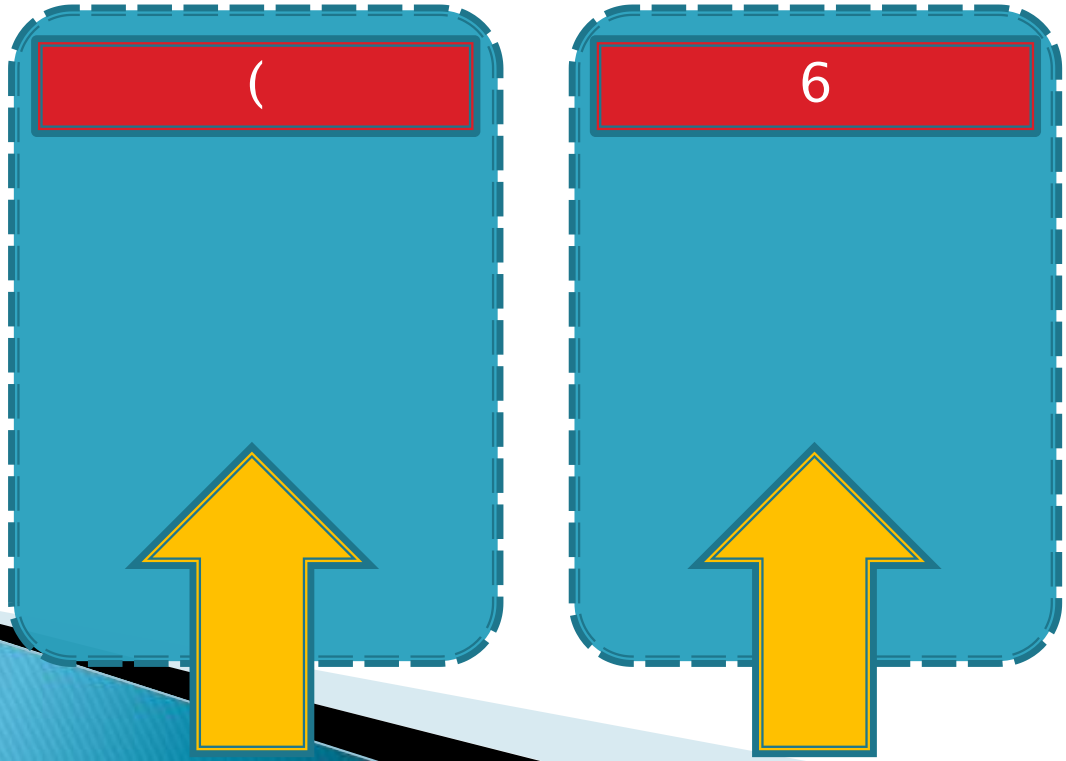
堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**
      - 以表达式 (6 + 2 * 5) / 4 为例说明计算过程：



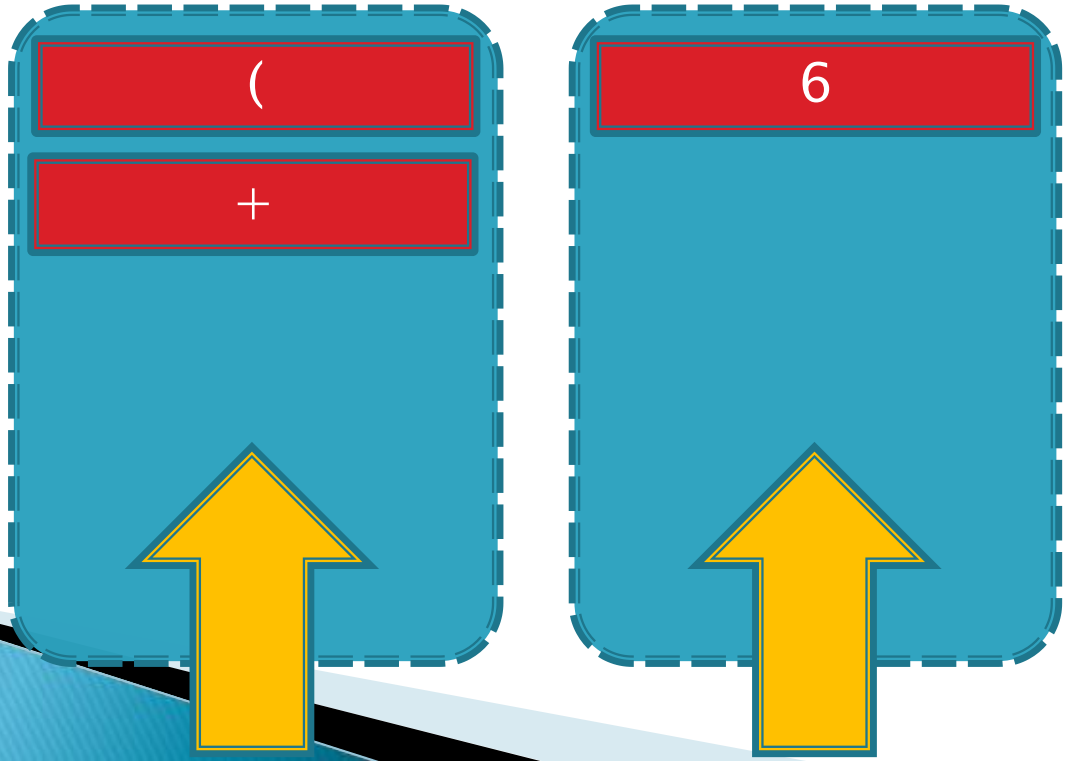
堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**
      - 以表达式 (6 + 2 * 5) / 4 为例说明计算过程：



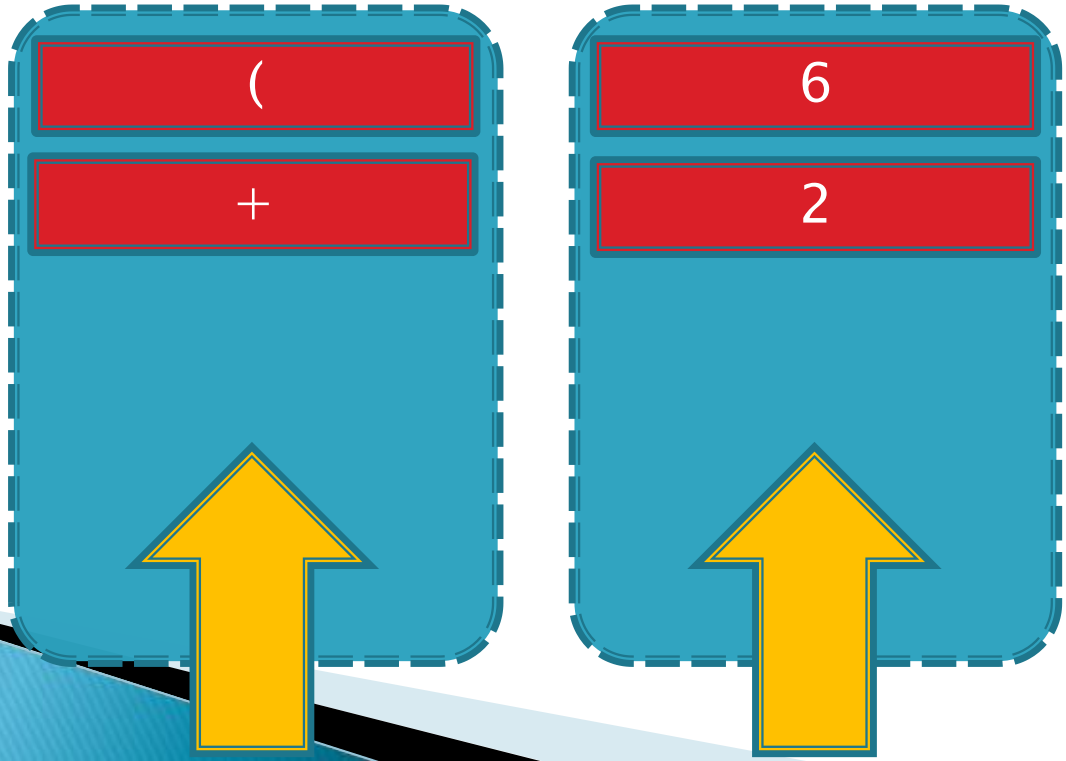
堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**
      - 以表达式 (6 + 2 * 5) / 4 为例说明计算过程：



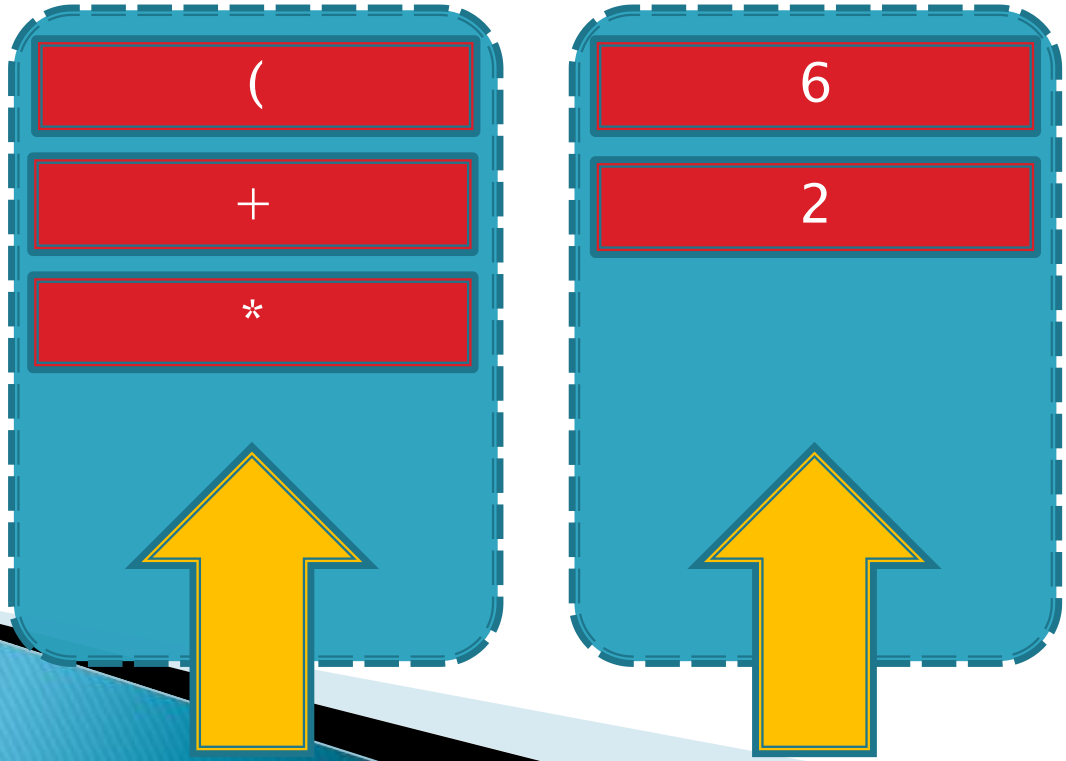
堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**
      - 以表达式 (6 + 2 * 5) / 4 为例说明计算过程：



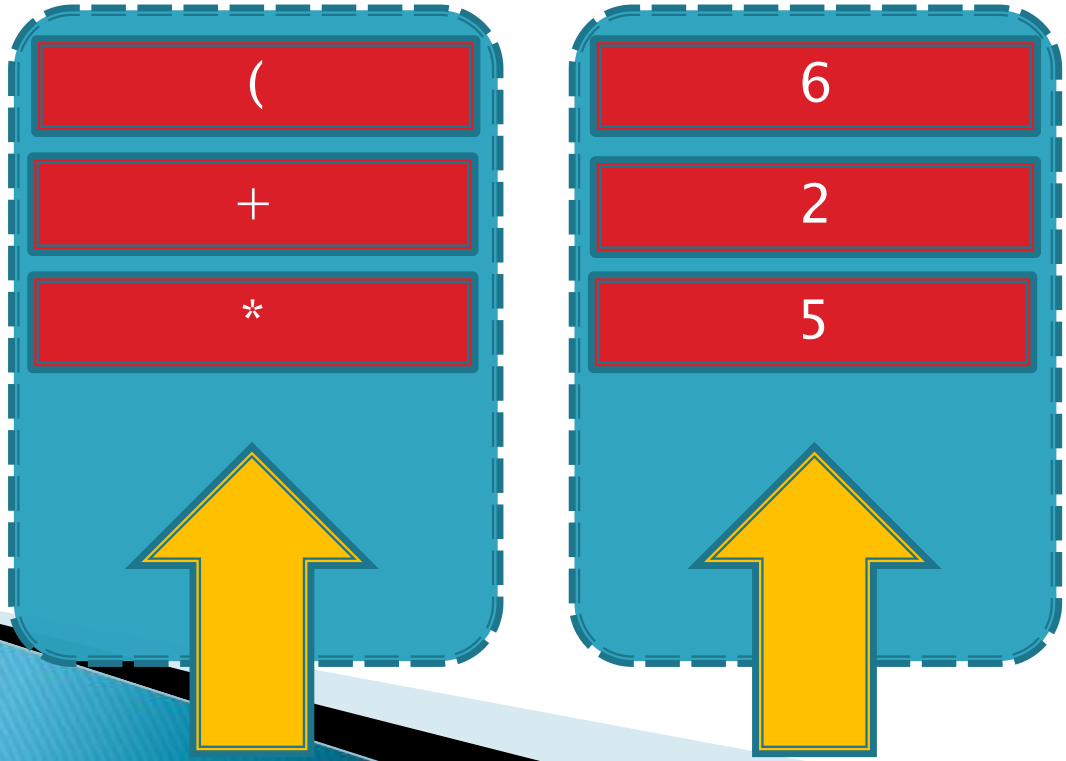
堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**
      - 以表达式 (6 + 2 * 5) / 4 为例说明计算过程：



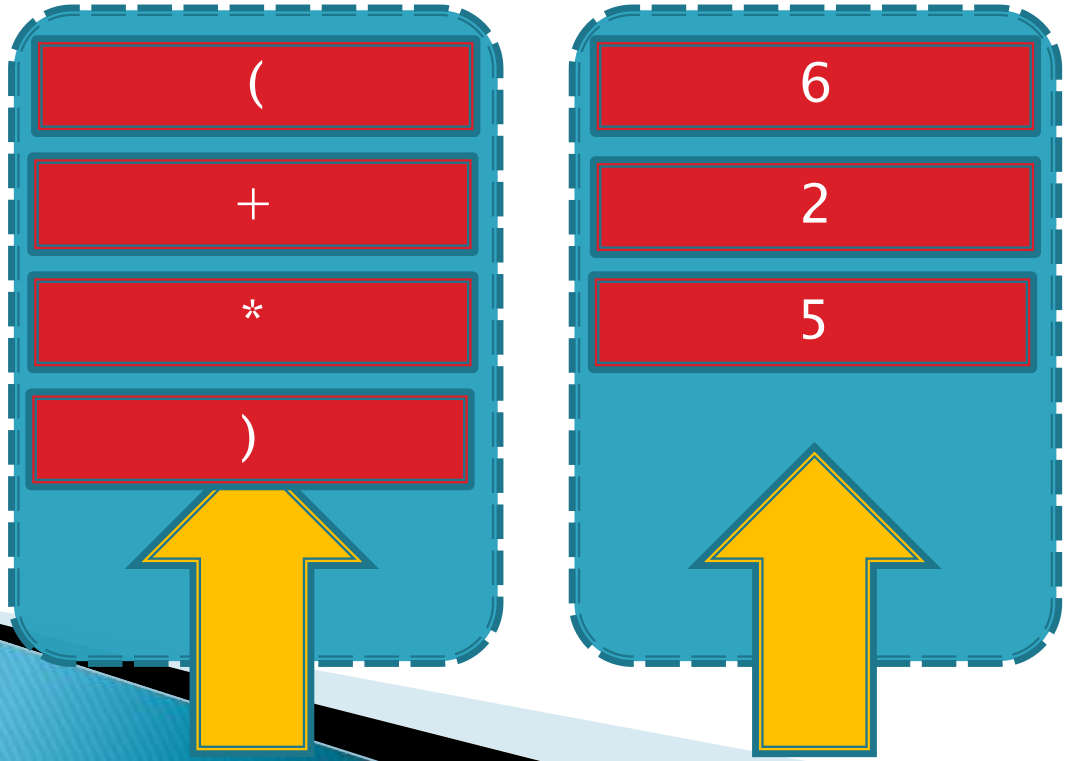
堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**
      - 以表达式 (6 + 2 * 5) / 4 为例说明计算过程：



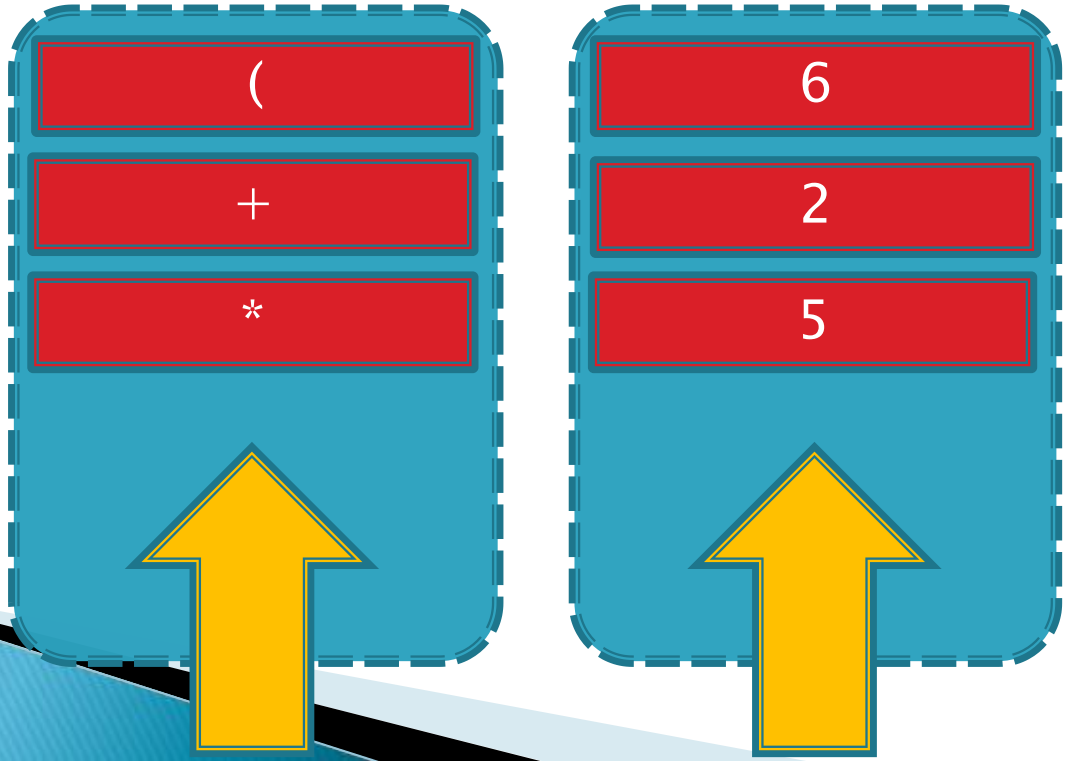
堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**
      - 以表达式 (6 + 2 * 5) / 4 为例说明计算过程：



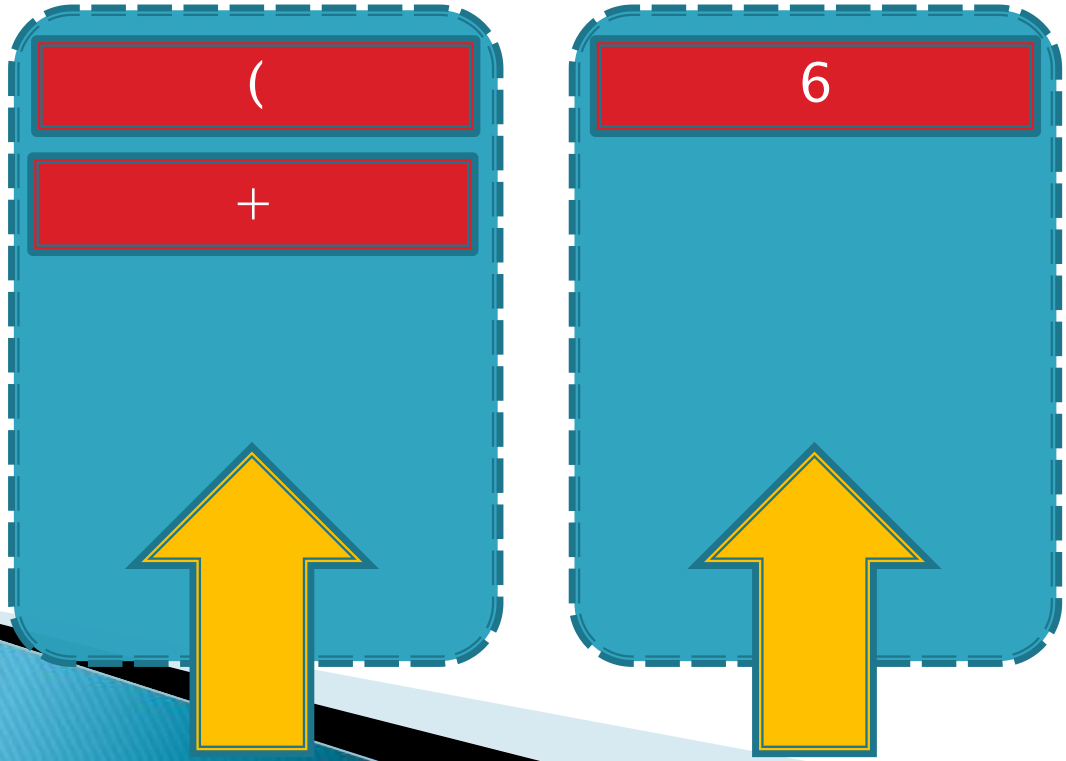
堆栈(Stack)

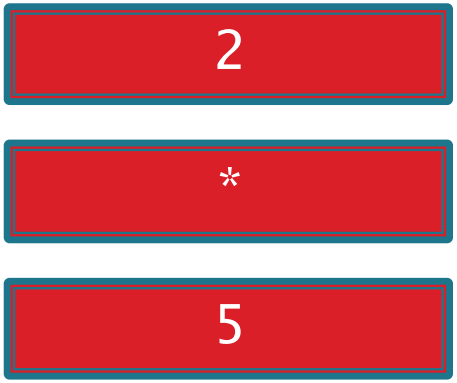
- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**
      - 以表达式 (6 + 2 * 5) / 4 为例说明计算过程：



堆栈(Stack)

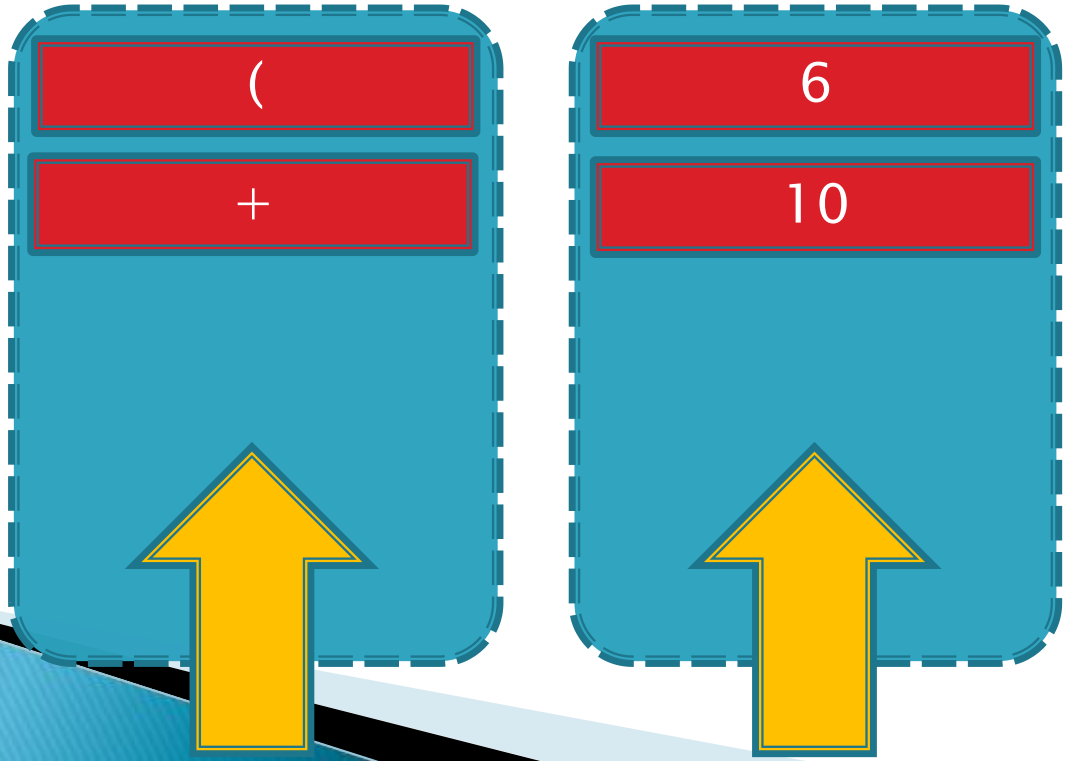
- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**
      - 以表达式 (6 + 2 * 5) / 4 为例说明计算过程：





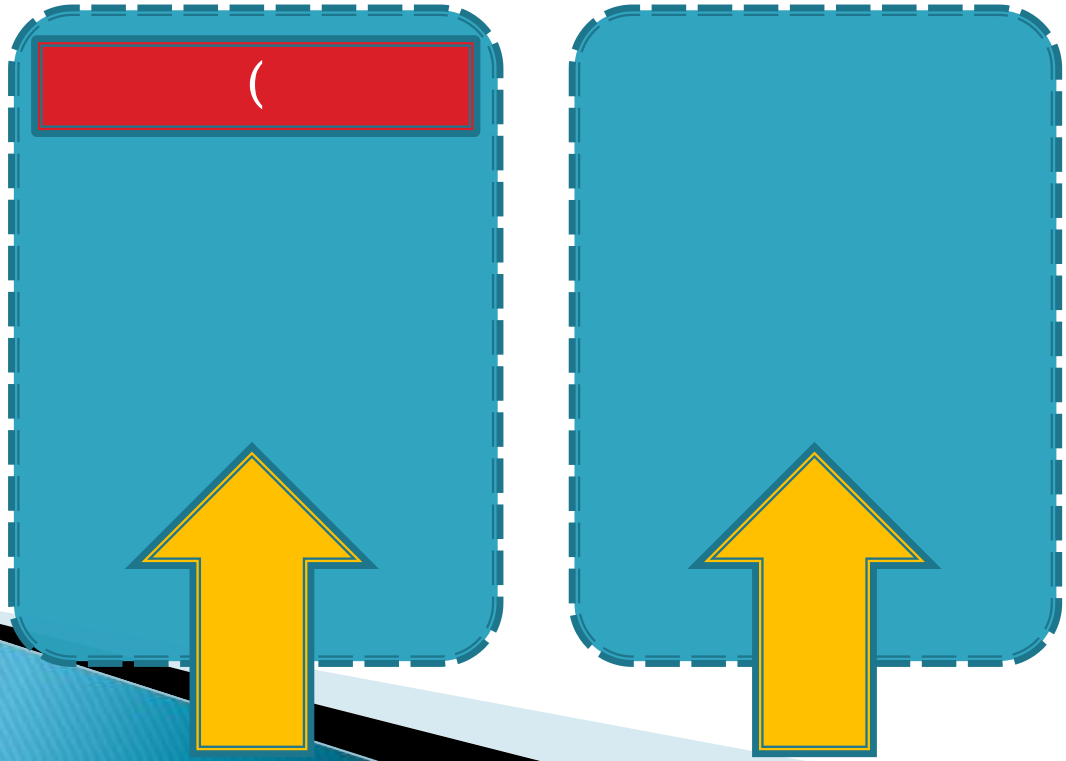
堆栈(Stack)

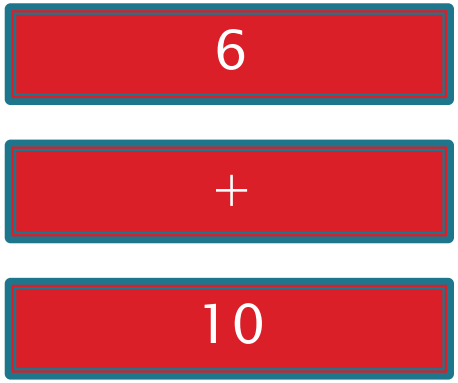
- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**
      - 以表达式 (6 + 2 * 5) / 4 为例说明计算过程：



堆栈(Stack)

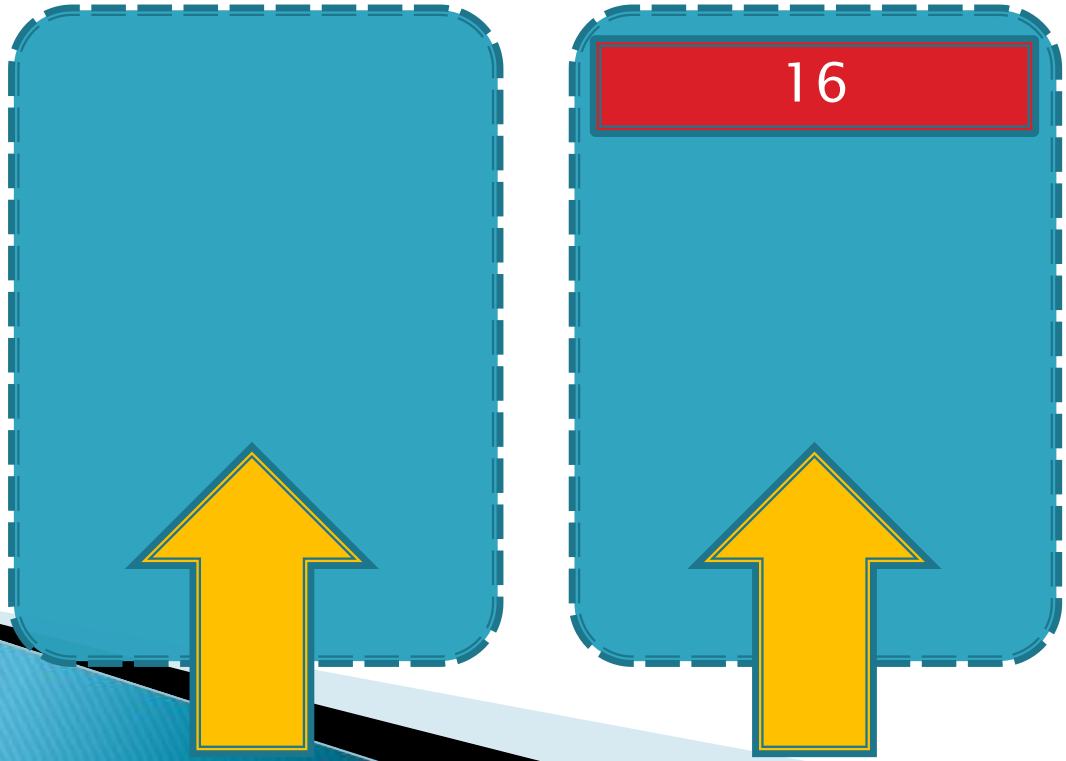
- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**
      - 以表达式 (6 + 2 * 5) / 4 为例说明计算过程：





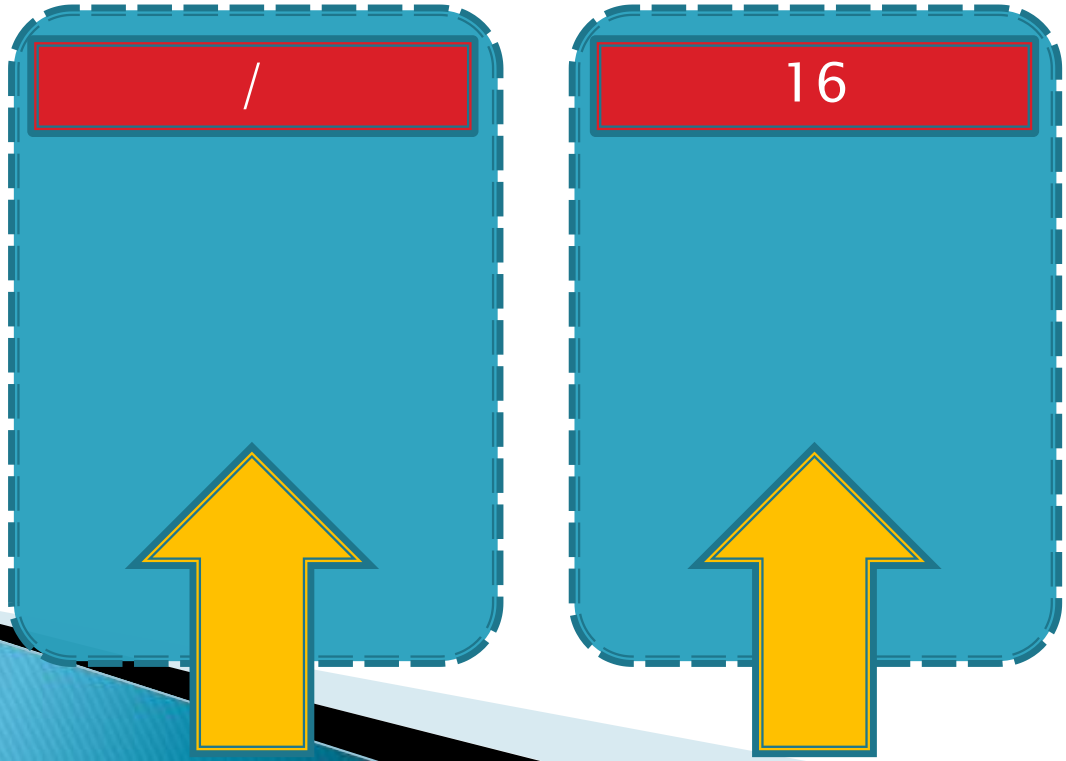
堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**
      - 以表达式 (6 + 2 * 5) / 4 为例说明计算过程：



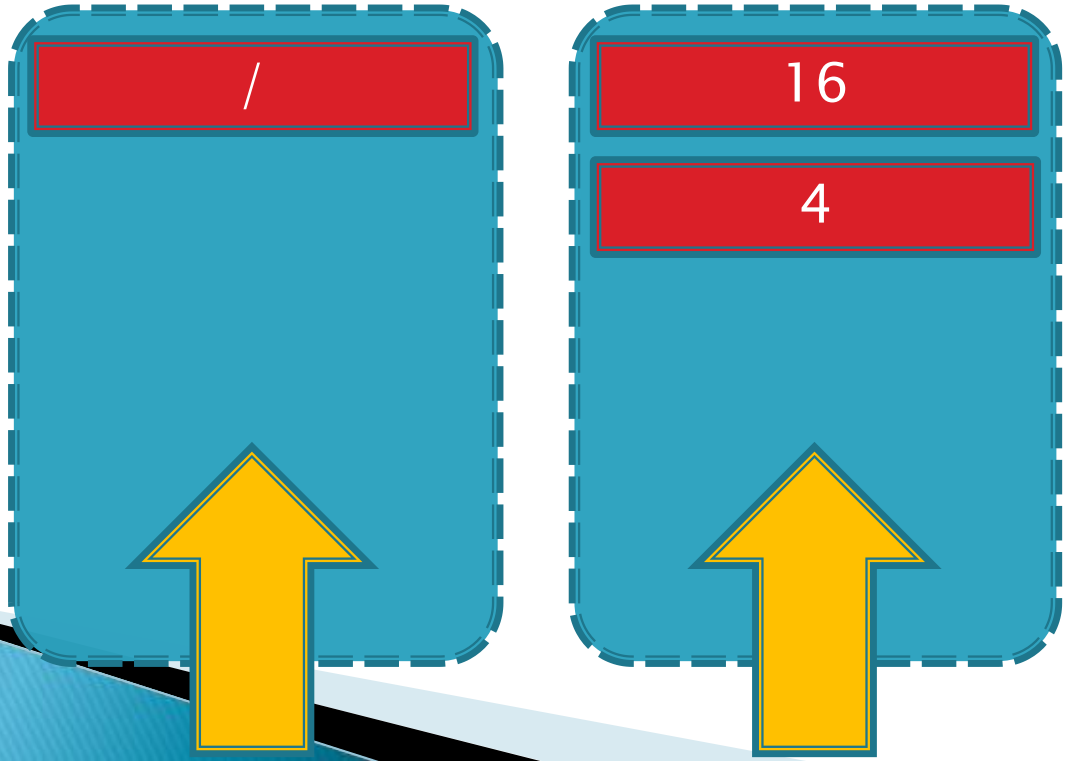
堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**
      - 以表达式 (6 + 2 * 5) / 4 为例说明计算过程：



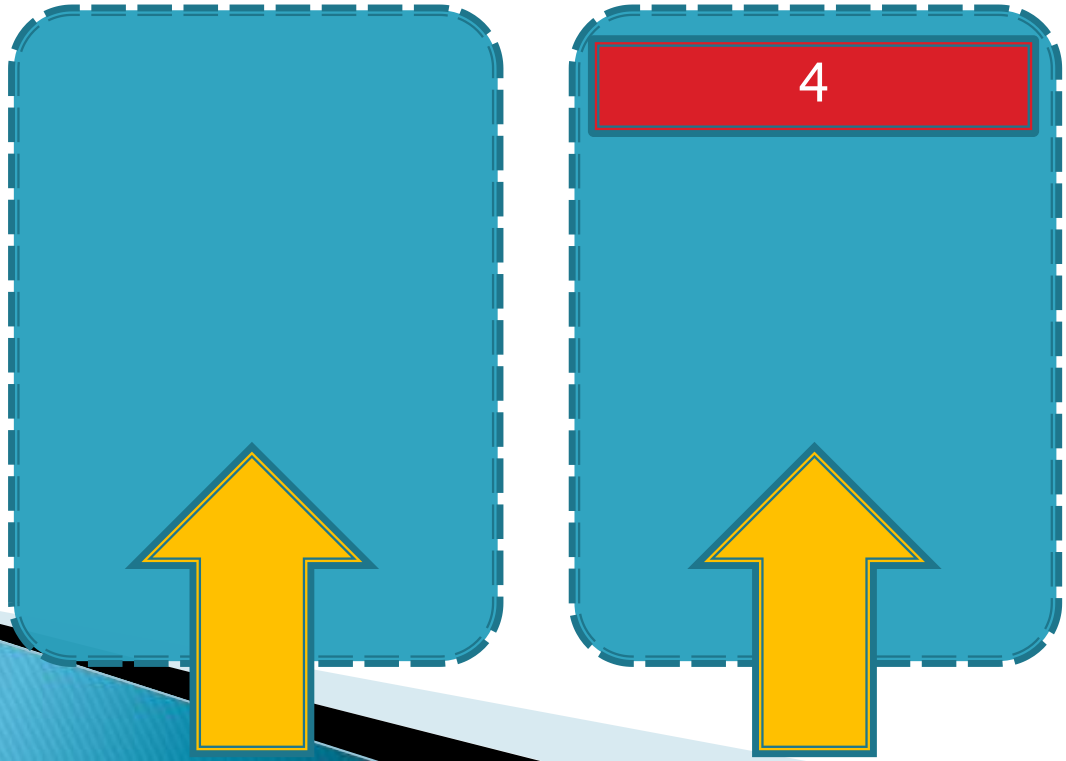
堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**
      - 以表达式 (6 + 2 * 5) / 4 为例说明计算过程：



堆栈(Stack)

- **使用堆栈**
  - 栈在表达式计算中的作用
    - **计算表达式的值**
      - 以表达式 (6 + 2 * 5) / 4 为例说明计算过程：



堆栈(Stack)

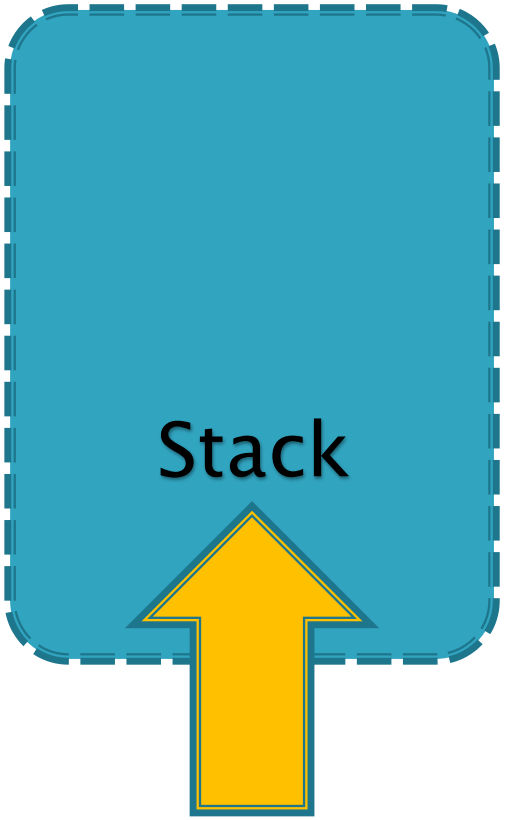
- **使用堆栈**
  - 合法的出栈序列

堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - 什么是递归?
    - 在数学和计算机科学中，递归指由一种(或多种)简单的基本情况定义的一类对象或方法，并规定其他所有情况都能被还原为其基本情况.
    - 典型的递归案例:
    - Case I：阶乘
      - $F(n) = n!$
      - $F(n) = n \times F(n-1)\ (n \geq 1)$
      - 规定:
        - $F(0) = 1$; (递归终止条件)

堆栈(Stack)

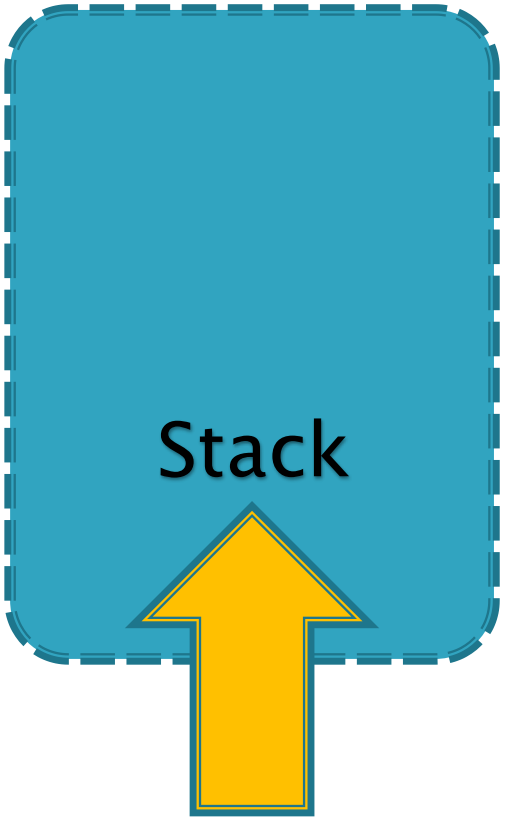
- **使用堆栈**
  - 递归(Recursion)
    - Case I：阶乘
    - 规定： $F(0) = 1$; (递归终止条件)
    - 递推关系式：$F(n) = n \times F(n - 1)$



堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case I：阶乘
    - 规定： $F(0) = 1$; (递归终止条件)
    - 递推关系式：$F(n) = n \times F(n-1)$

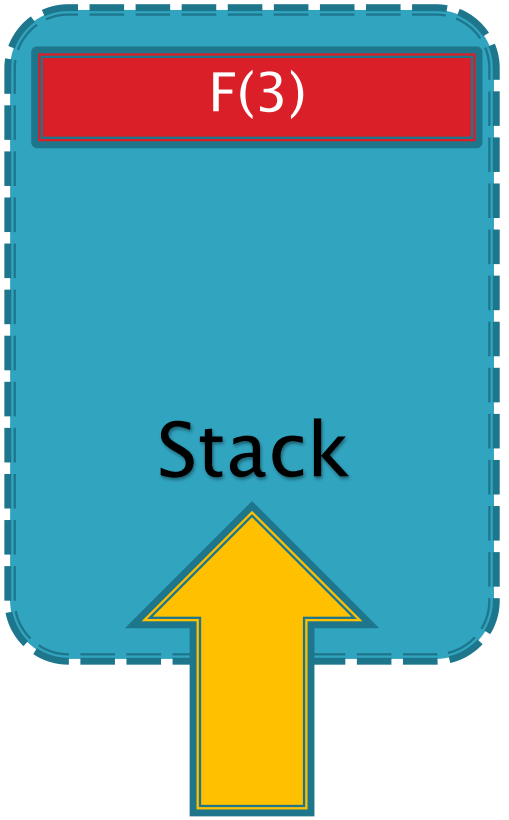
F(3)



堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case I：阶乘
    - 规定： $F(0) = 1$; (递归终止条件)
    - 递推关系式：$F(n) = n \times F(n-1)$

F(3) $= 3 \times F(2)$

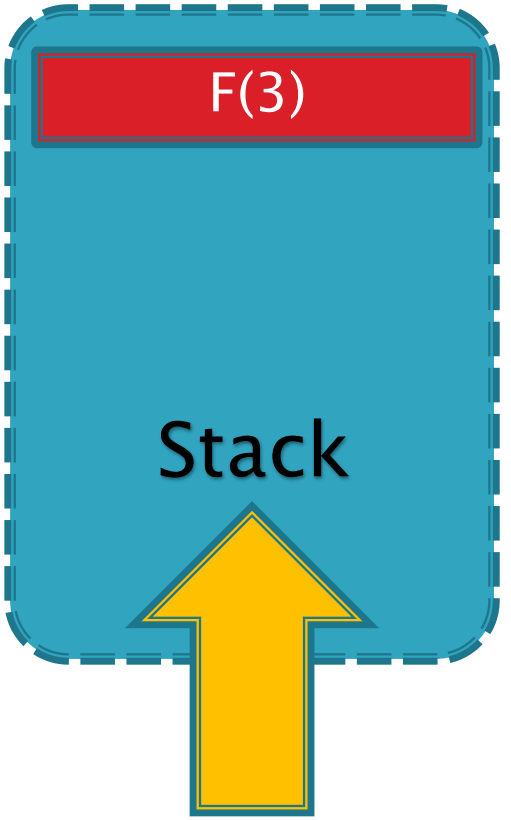


堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case I：阶乘
    - 规定： $F(0) = 1$; (递归终止条件)
    - 递推关系式：$F(n) = n \times F(n-1)$

F(3) $= 3 \times F(2)$

F(2)

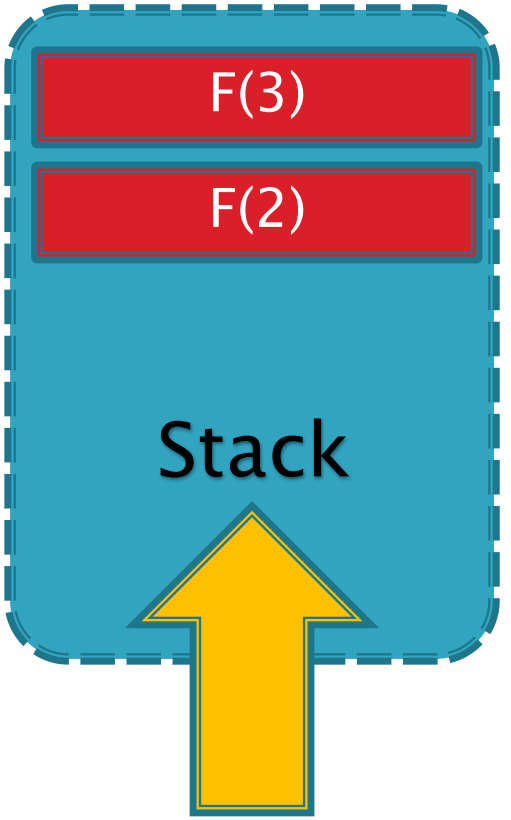


堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case I：阶乘
    - 规定： $F(0) = 1$; (递归终止条件)
    - 递推关系式：$F(n) = n \times F(n-1)$

F(3) $= 3 \times F(2)$

F(2) $= 2 \times F(1)$



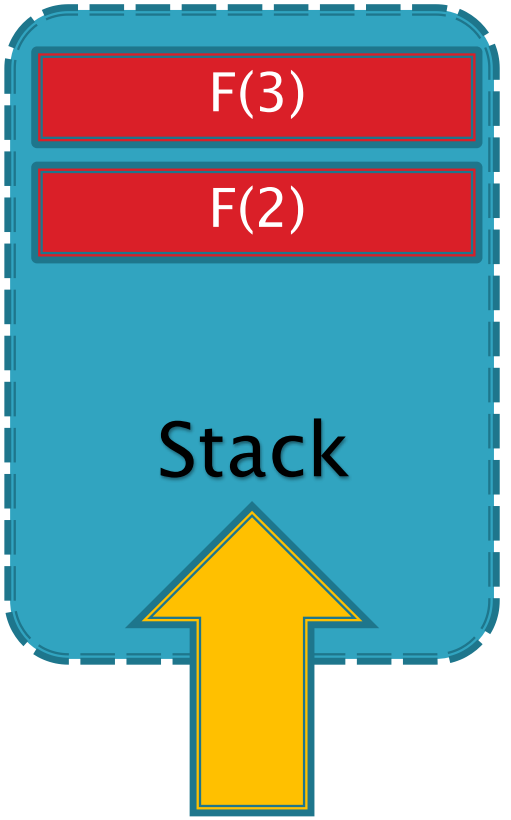
堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case I：阶乘
    - 规定： F(0) = 1; (递归终止条件)
    - 递推关系式：F(n) = n × F(n – 1)

F(3) = 3 × F(2)

F(2) = 2 × F(1)

F(1)



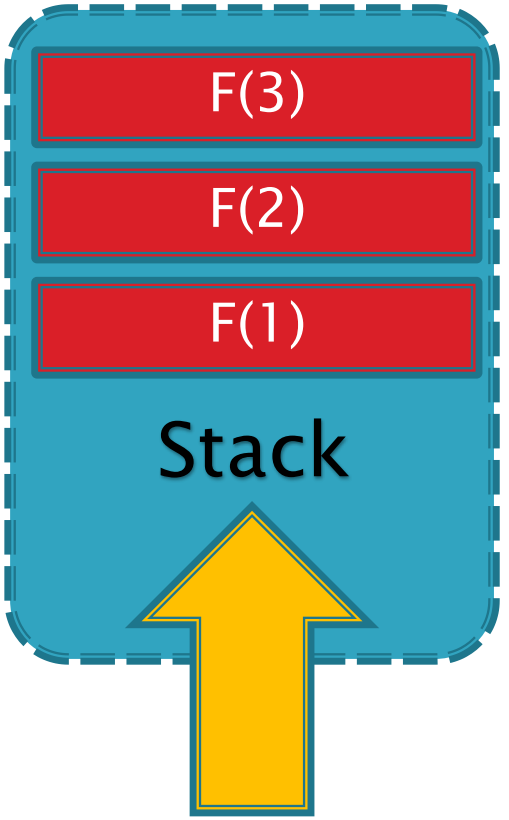
堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case I：阶乘
    - 规定： $F(0) = 1$; (递归终止条件)
    - 递推关系式：$F(n) = n \times F(n-1)$

$$F(3) = 3 \times F(2)$$

$$F(2) = 2 \times F(1)$$

$$F(1) = 1 \times F(0)$$



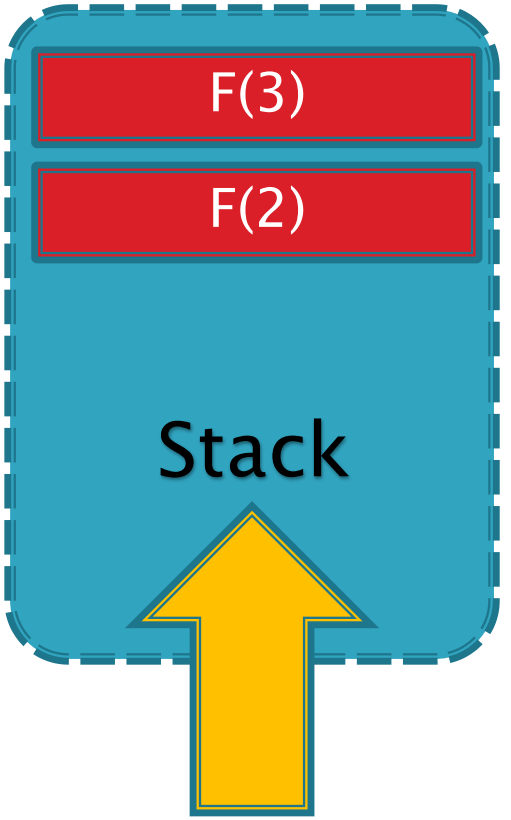
堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case I：阶乘
    - 规定： F(0) = 1; (递归终止条件)
    - 递推关系式：F(n) = n × F(n – 1)

F(3) = 3 × F(2)

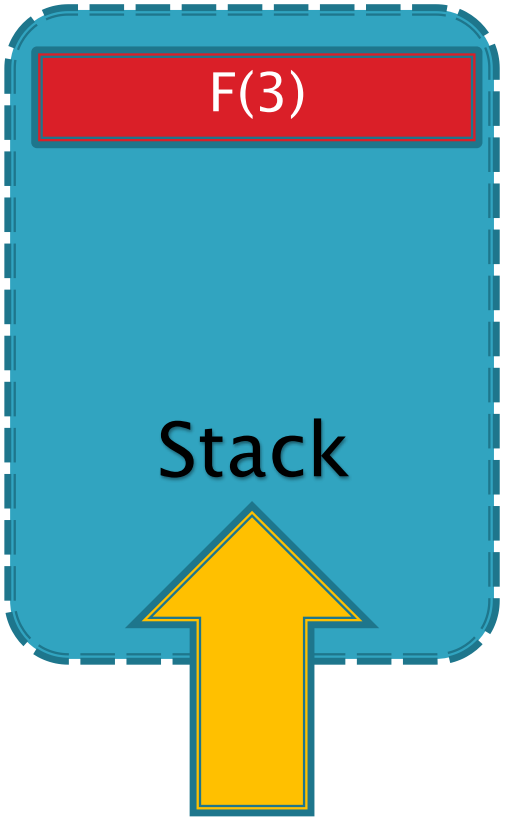
F(2) = 2 × F(1)

F(1) = 1 × F(0) = 1 × 1 = 1



堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case I：阶乘
    - 规定： $F(0) = 1$; (递归终止条件)
    - 递推关系式：$F(n) = n \times F(n-1)$



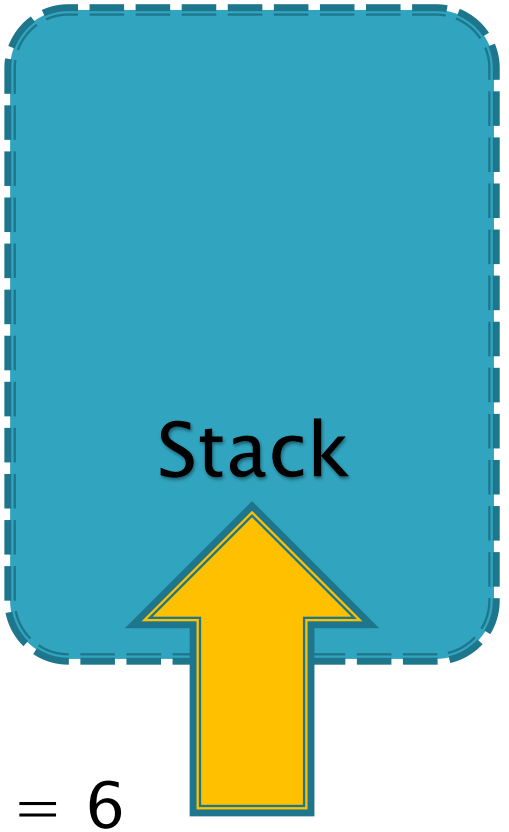
F(3) $= 3 \times F(2)$

F(2) $= 2 \times F(1) = 2 \times 1 = 2$

F(1) $= 1 \times F(0) = 1 \times 1 = 1$

堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case I：阶乘
    - 规定： F(0) = 1; (递归终止条件)
    - 递推关系式：F(n) = n × F(n – 1)



F(3) = 3 × F(2) = 3 × 2 = 6

F(2) = 2 × F(1) = 2 × 1 = 2

F(1) = 1 × F(0) = 1 × 1 = 1

堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case I：阶乘

Pascal描述

```
Function Factorial(n: integer): longint;
begin
    if (n = 0) then
        Factorial:= 1
    else
        Factorial:= n * Factorial(n - 1);
end;
```

堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case I：阶乘

C/C++描述

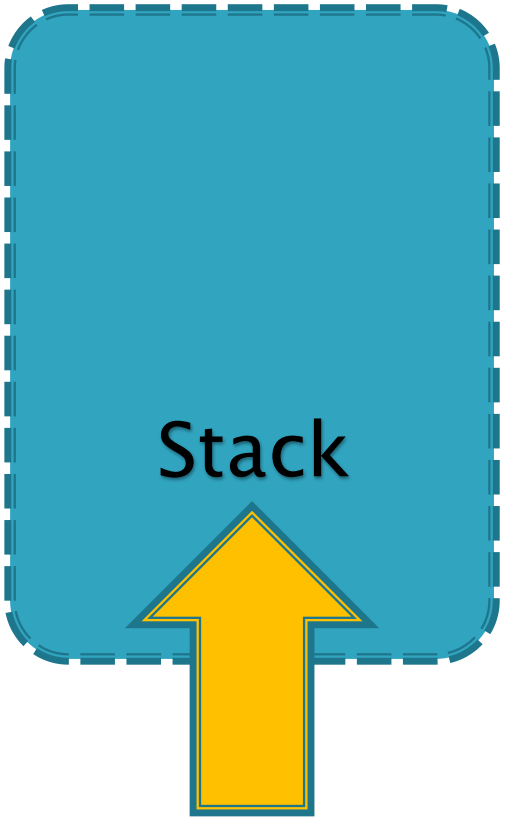
```
int Factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * Factorial(n - 1);
}
```

堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - 什么是递归?
    - 在数学和计算机科学中，递归指由一种(或多种)简单的基本情况定义的一类对象或方法，并规定其他所有情况都能被还原为其基本情况.
    - 典型的递归案例：
    - Case II：斐波那契数列
      - $F(n) = F(n-1) + F(n-2)\ (n \geq 2)$
      - 规定：
        - $F(0) = 0;\ F(1) = 1;$ (递归终止条件)

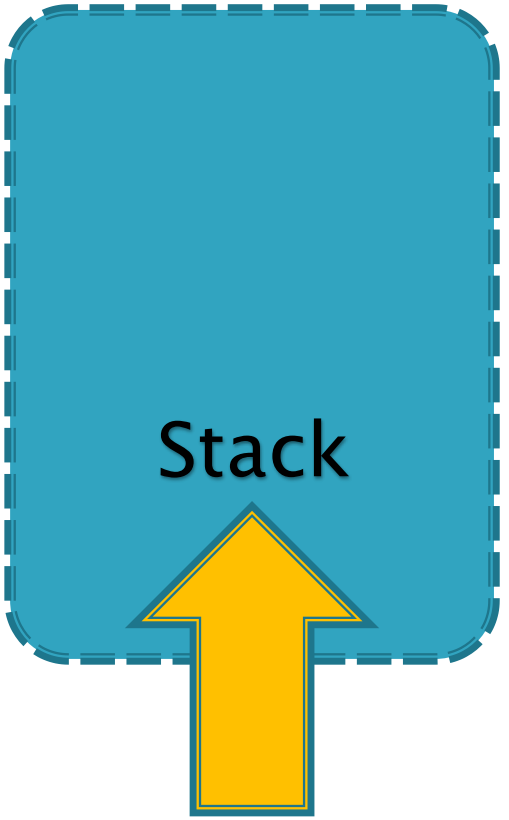
堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case II：斐波那契数列
    - 递推关系式：F(n) = F(n – 1) + F(n – 2)
    - 规定：F(0) = 0; F(1) = 1;



堆栈(Stack)

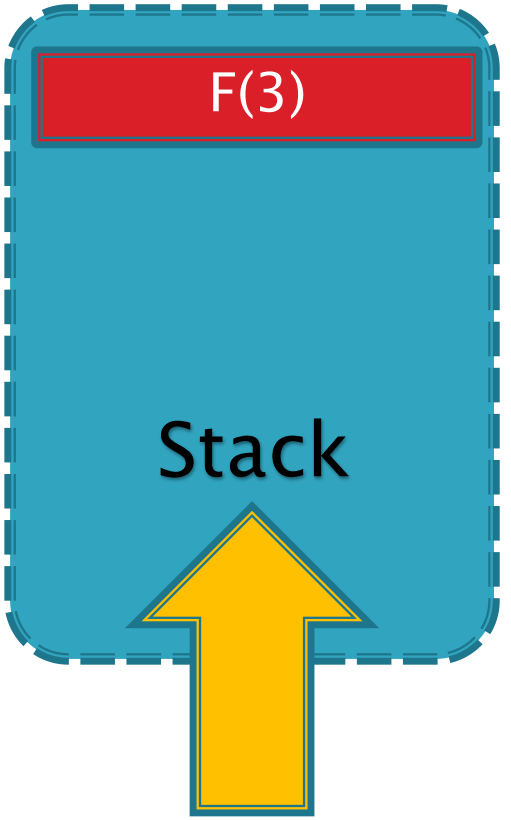
- **使用堆栈**
  - 递归(Recursion)
    - Case II：斐波那契数列
    - 递推关系式：F(n) = F(n – 1) + F(n – 2)
    - 规定：F(0) = 0; F(1) = 1;



F(3)

堆栈(Stack)

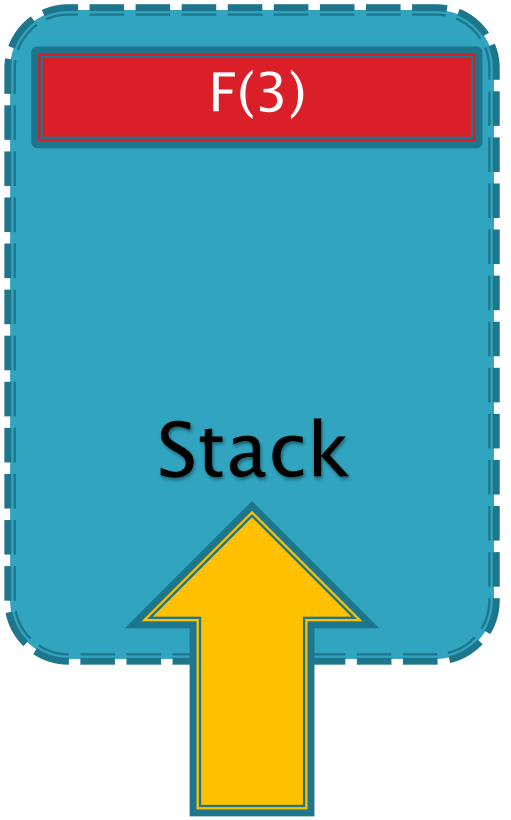
- **使用堆栈**
  - 递归(Recursion)
    - Case II：斐波那契数列
    - 递推关系式：F(n) = F(n – 1) + F(n – 2)
    - 规定：F(0) = 0; F(1) = 1;



F(3) = F(2) + F(1)

堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case II：斐波那契数列
    - 递推关系式：F(n) = F(n – 1) + F(n – 2)
    - 规定：F(0) = 0; F(1) = 1;

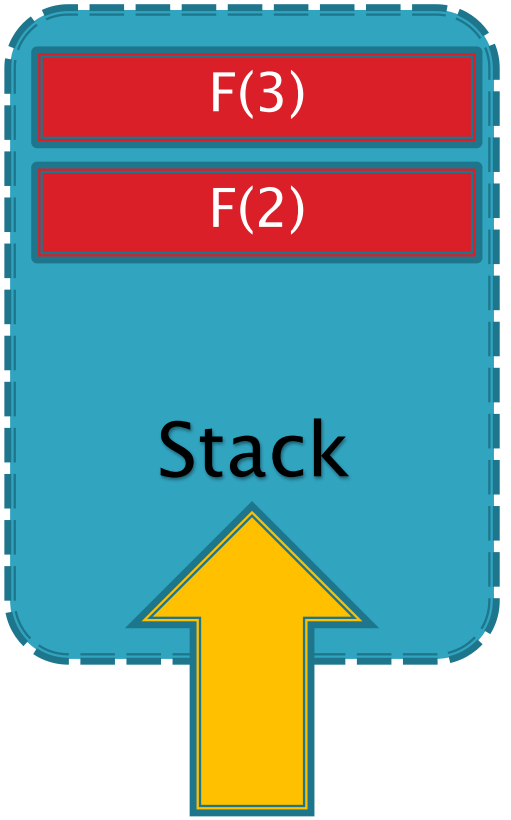


F(3) = F(2) + F(1)

F(2)

堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case II：斐波那契数列
    - 递推关系式：F(n) = F(n – 1) + F(n – 2)
    - 规定：F(0) = 0; F(1) = 1;

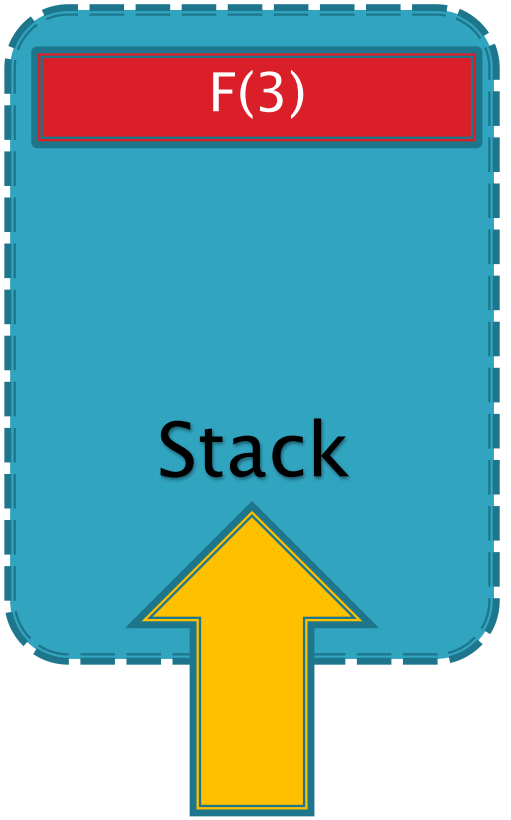


F(3) = F(2) + F(1)

F(2) = F(1) + F(0)

堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case II：斐波那契数列
    - 递推关系式：F(n) = F(n – 1) + F(n – 2)
    - 规定：F(0) = 0; F(1) = 1;

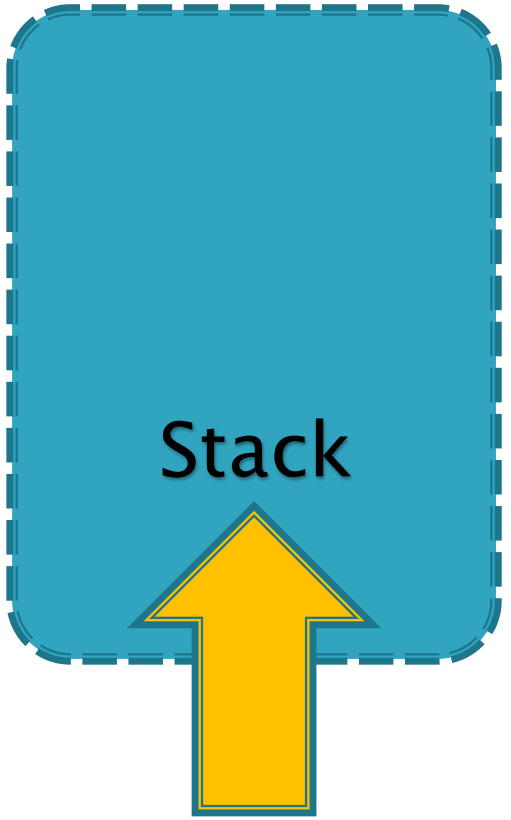


F(3) = F(2) + F(1)

F(2) = F(1) + F(0) = 1 + 0 = 1

堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case II：斐波那契数列
    - 递推关系式：F(n) = F(n – 1) + F(n – 2)
    - 规定：F(0) = 0; F(1) = 1;



F(3) = F(2) + F(1) = 1 + 1 = 2

F(2) = F(1) + F(0) = 1 + 0 = 1

堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case II：斐波那契数列

Pascal描述

```
Function Fibonacci(n: integer): longint;
begin
    if (n = 0) then
        Fibonacci:= 0
    else if  (n = 1) then
        Fibonacci:= 1
    else
        Fibonacci:= Fibonacci(n - 1) + Fibonacci(n - 2);
end;
```

堆栈(Stack)

- **使用堆栈**
  - 递归(Recursion)
    - Case II：斐波那契数列

C/C++描述

```
int Fibonacci (int n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

数据结构

[Queue]队列

队列(Queue)

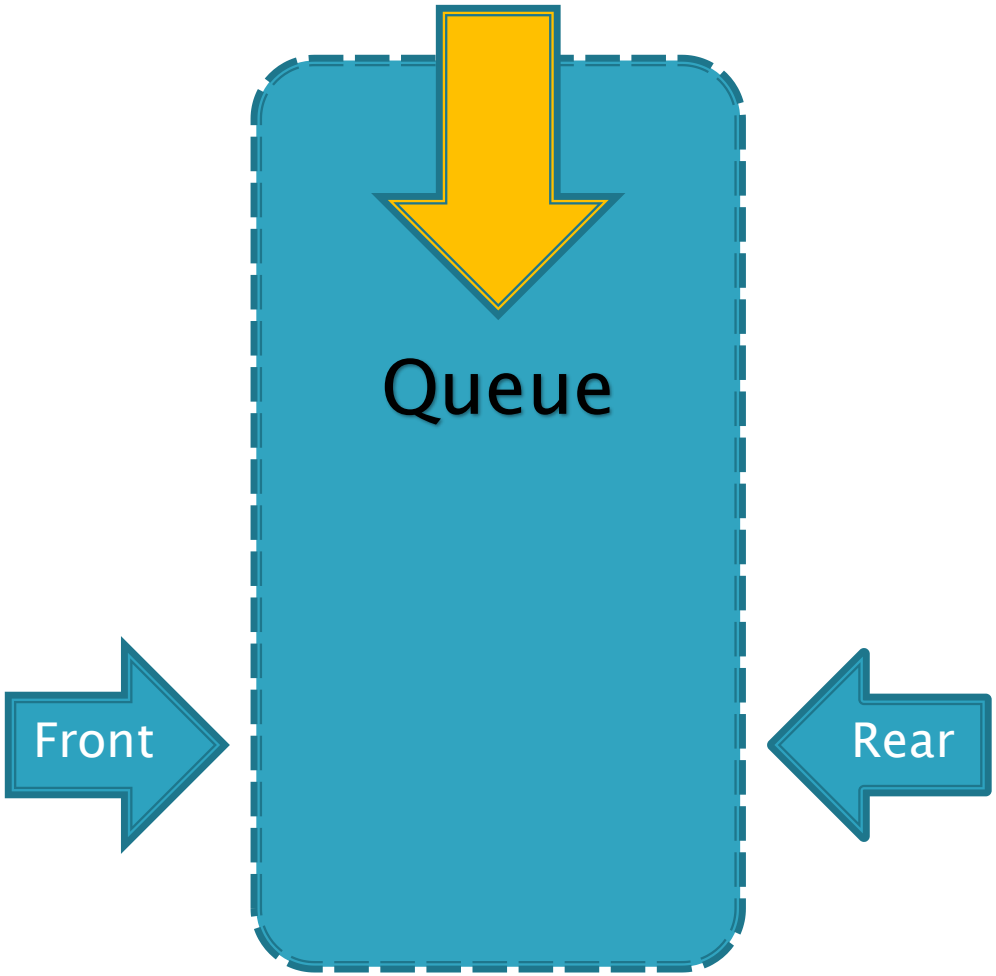
- **什么是队列？**
  - 队列(Queue)，又称伫列。在计算机科学中，是一种特殊的串行形式的数据结构，在具体应用中通常用链表或者数组来实现。队列只允许在后端(称为Rear)进行插入操作，在前端(称为Front)进行删除操作。
  - 由于队列数据结构允许在两端进行操作，因而按照后进先出(FIFO, First In First Out)的原理运作。
  - 堆栈数据结构使用两种基本操作：
    - 队列的操作方式和堆栈类似，唯一的区别在于队列只允许新数据在后端进行添加。

队列(Queue)

- 什么是**顺序队列**?

Size = 3

Status: Empty

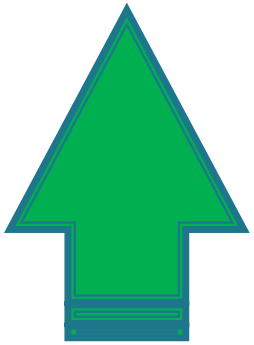


队列(Queue)

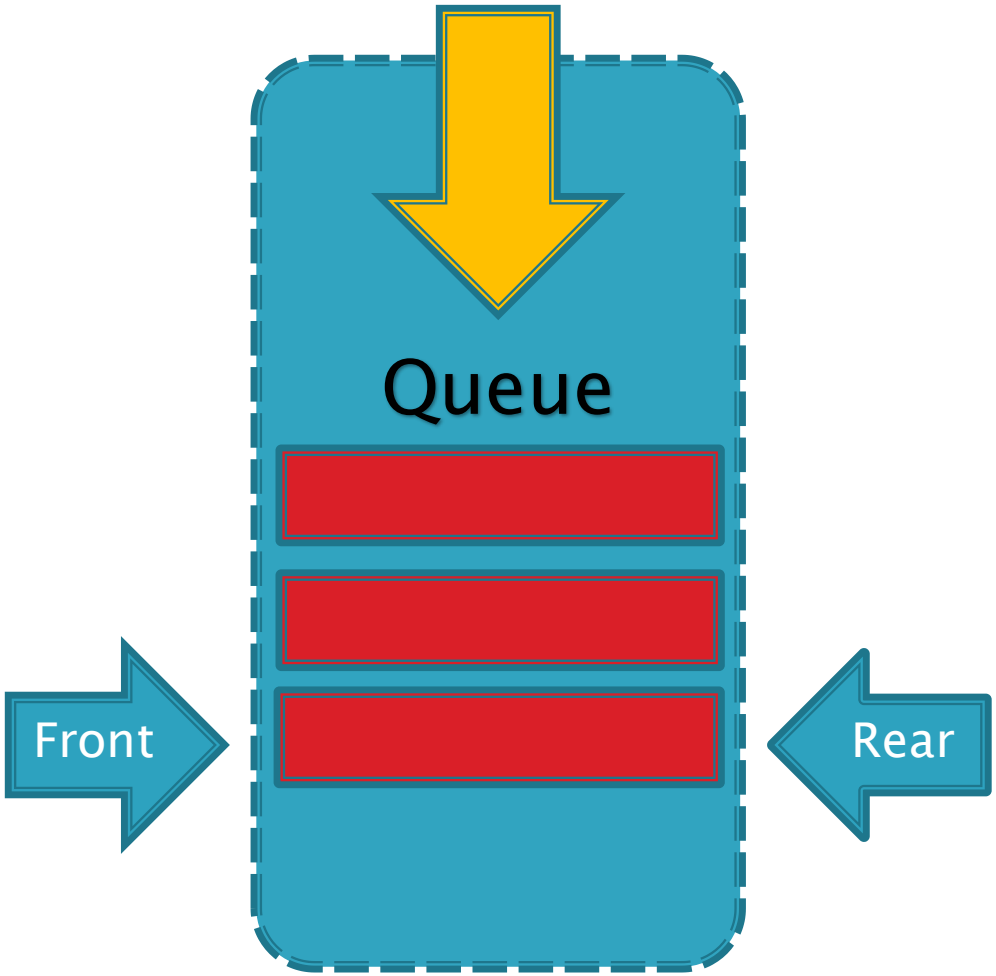
- 什么是**顺序队列**?

Size = 3

Status: Full



Rear > Size

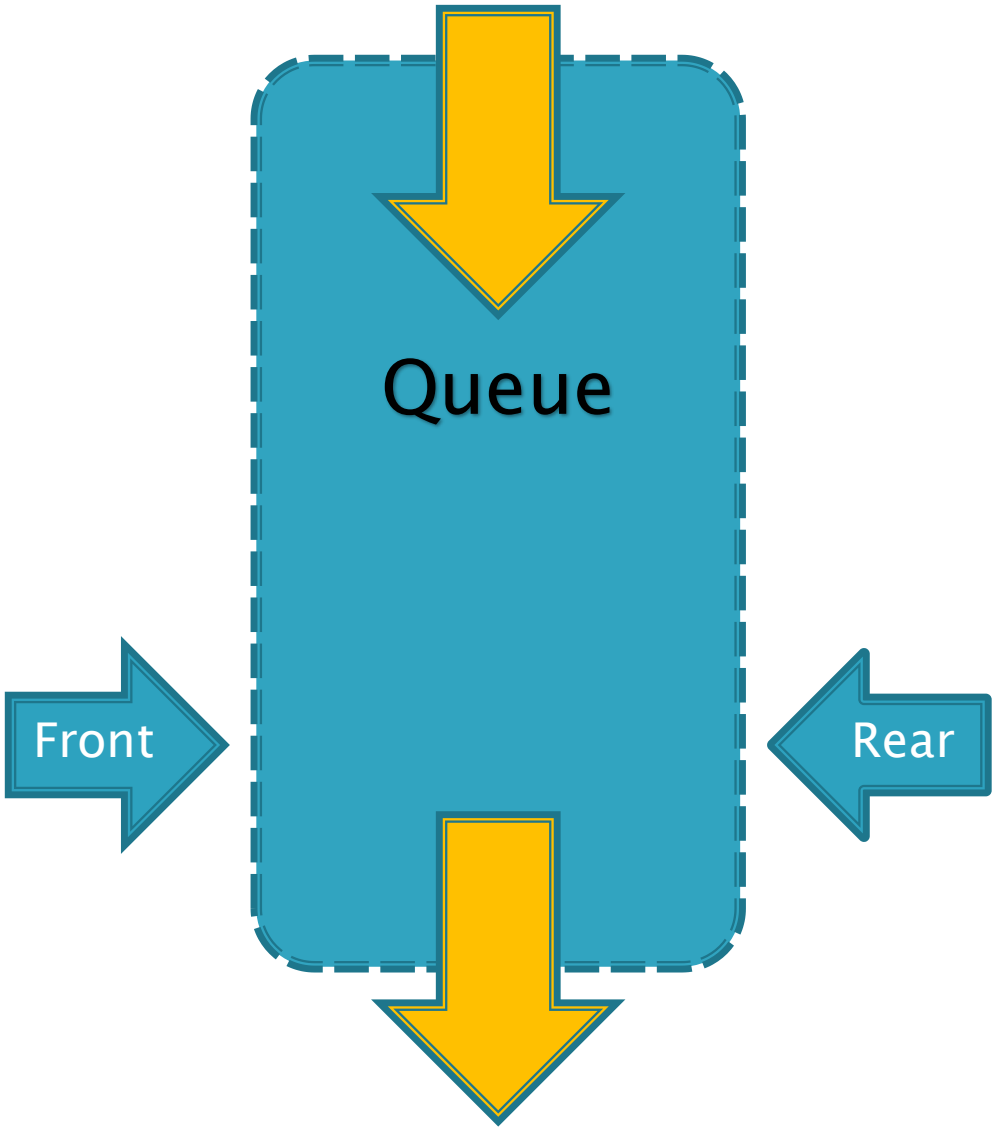


队列(Queue)

- 什么是**顺序队列**?

Size = 3

Status: Empty?

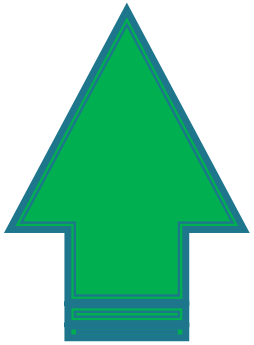


队列(Queue)

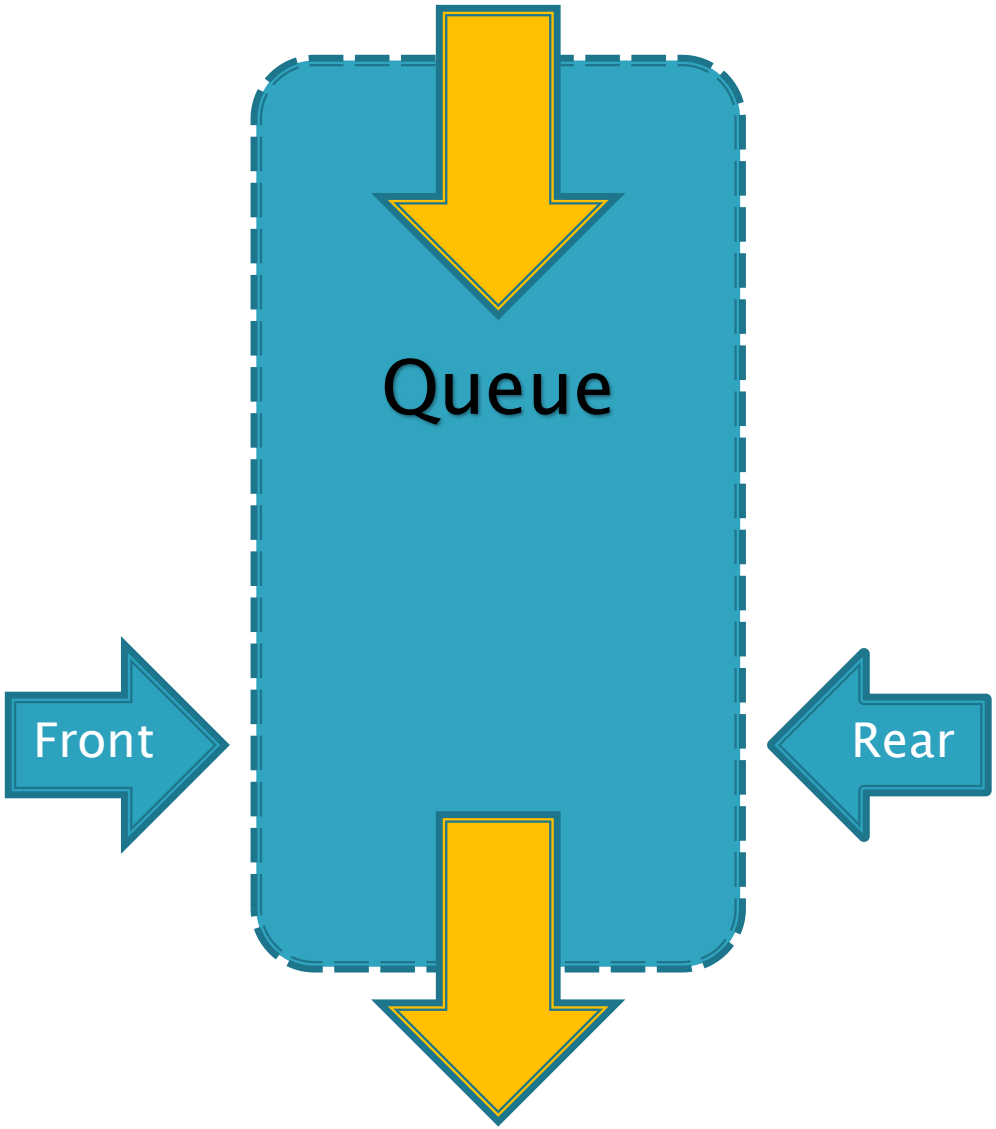
- 什么是**顺序队列**?

Size = 3

Status: Empty?



Rear > Size

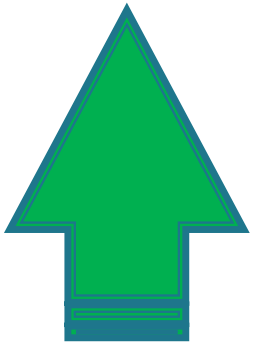


队列(Queue)

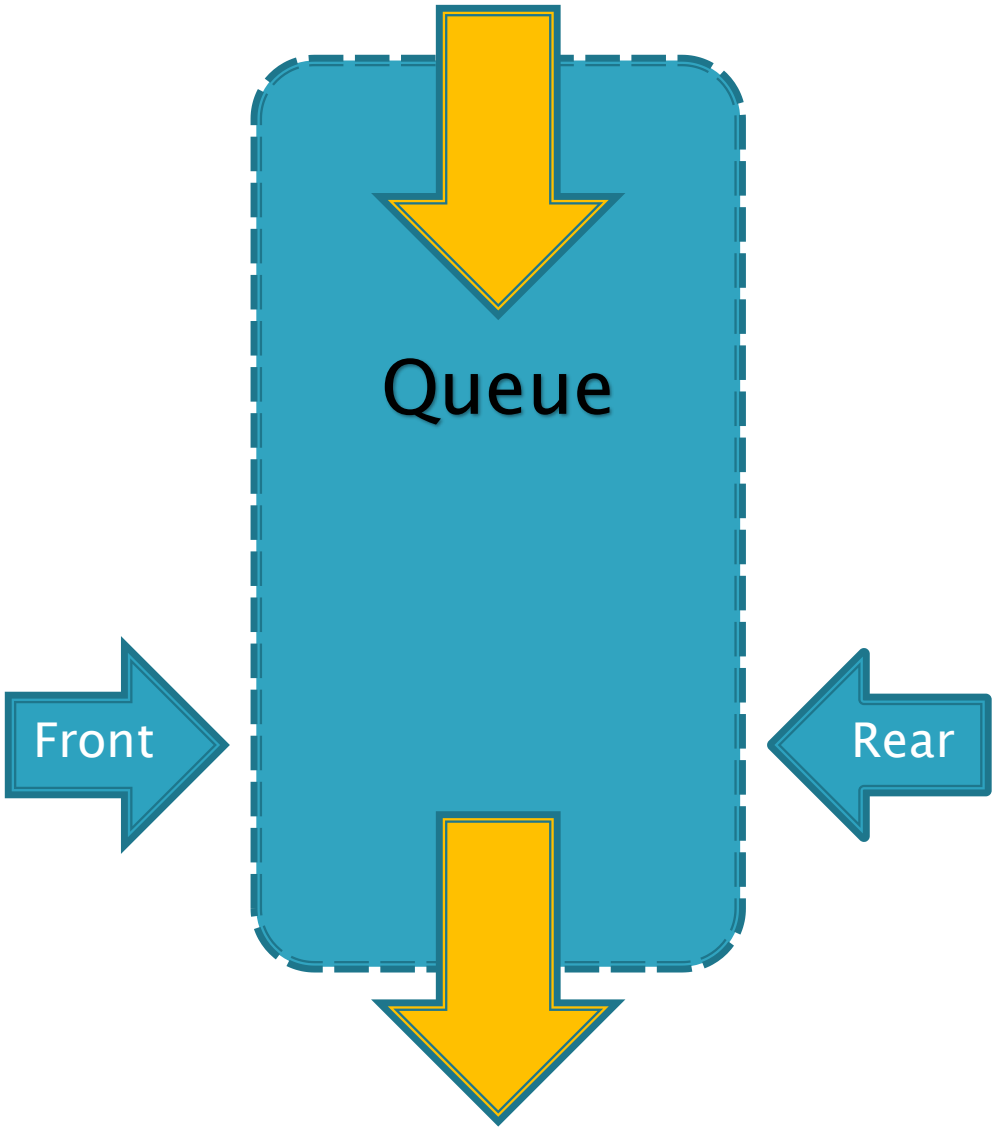
- 什么是**顺序队列**?

Size = 3

Status: Full



Rear > Size

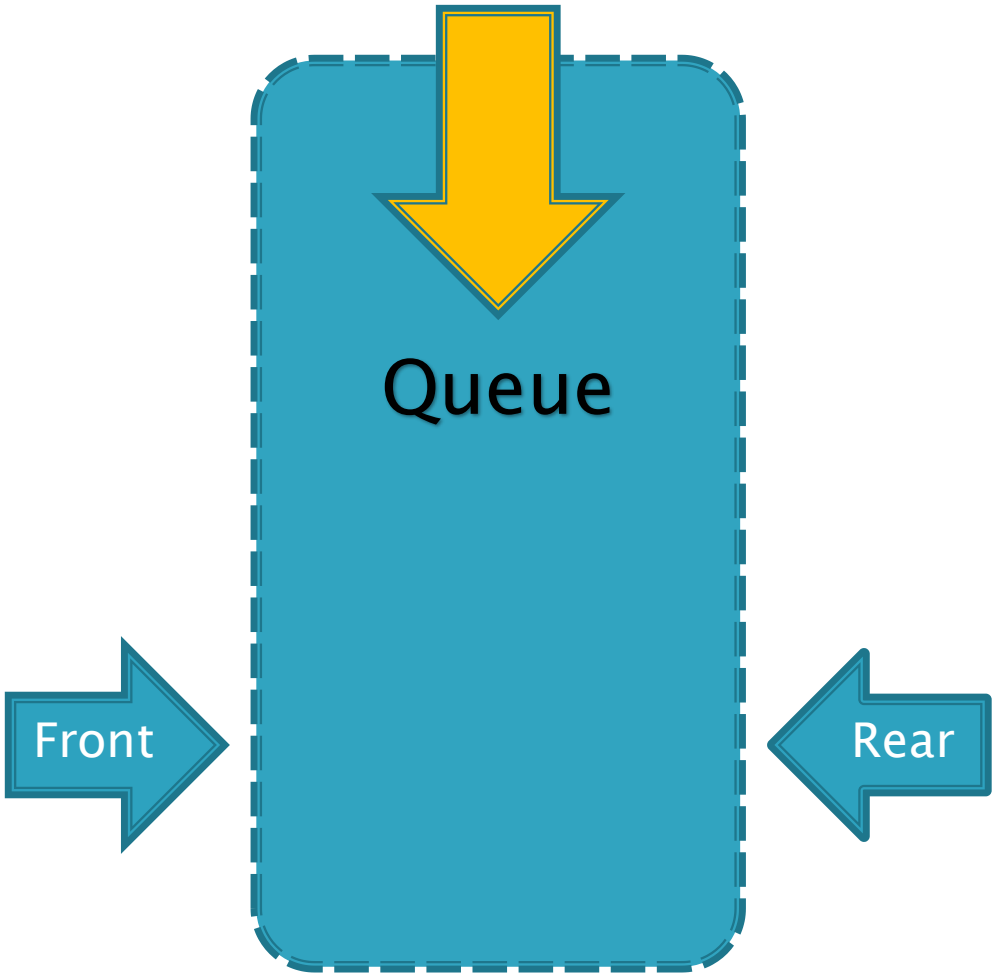


队列(Queue)

- 什么是**循环队列**?

Size = 3

Status: Empty

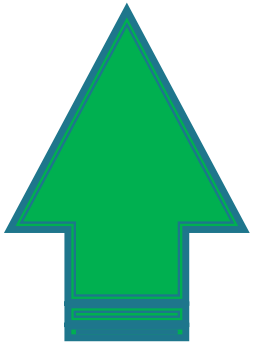


队列(Queue)

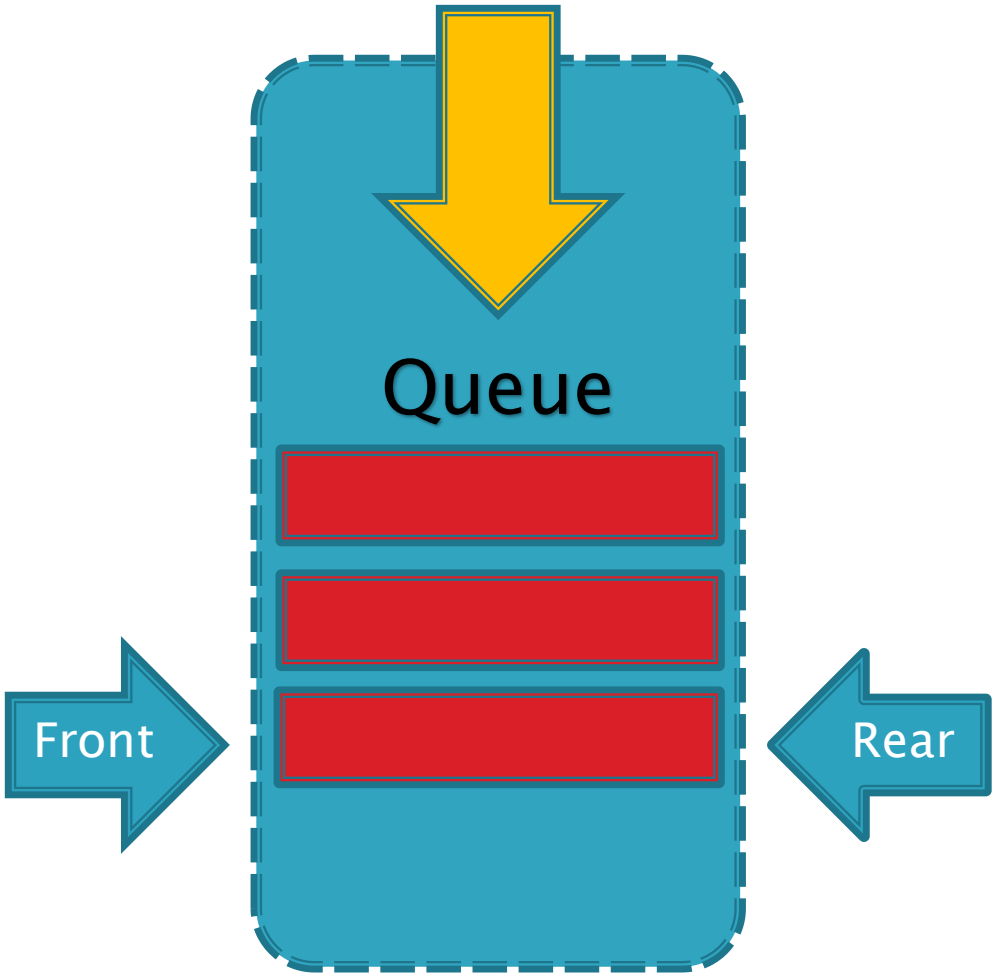
- 什么是**循环队列**?

Size = 3

Status: Full



Rear > Size

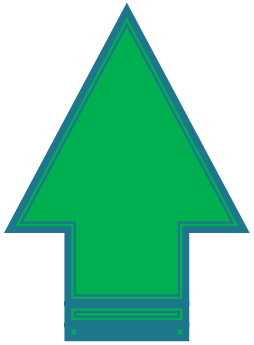


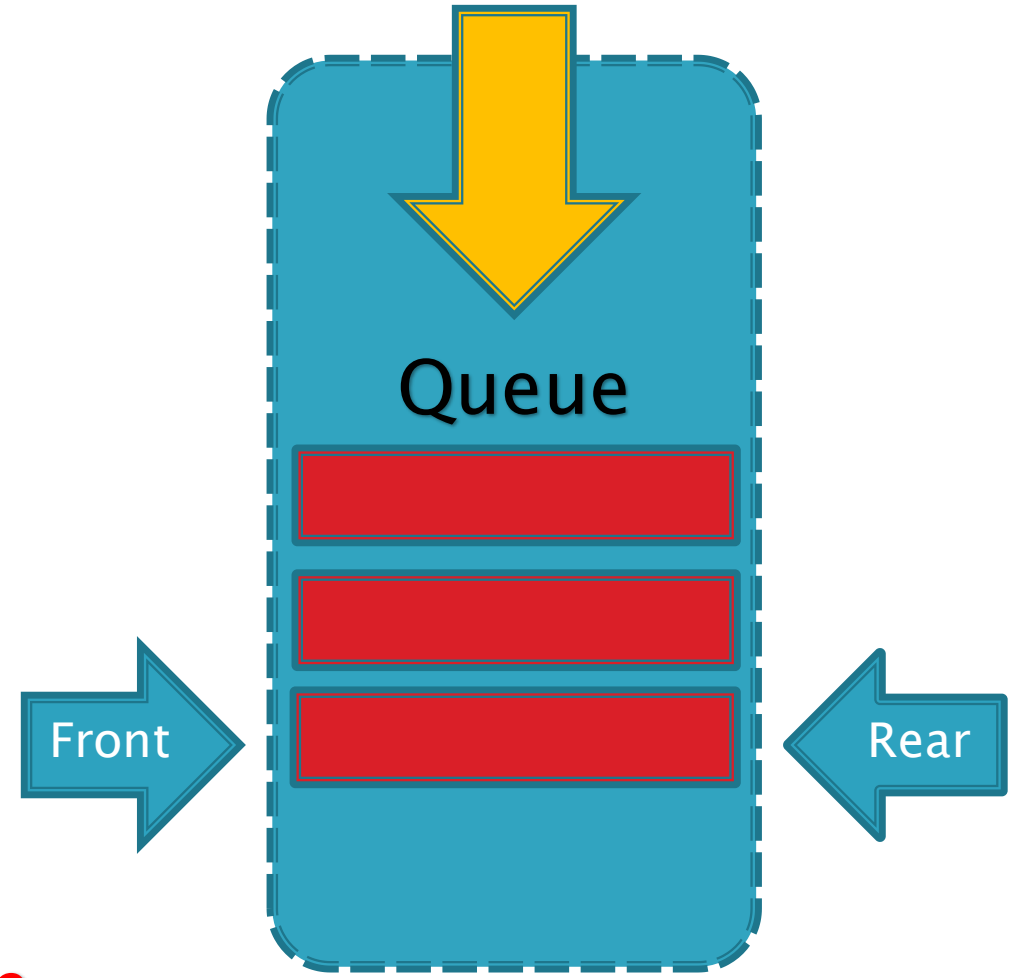
队列(Queue)

- 什么是**循环队列**?

Size = 3

Status: Full





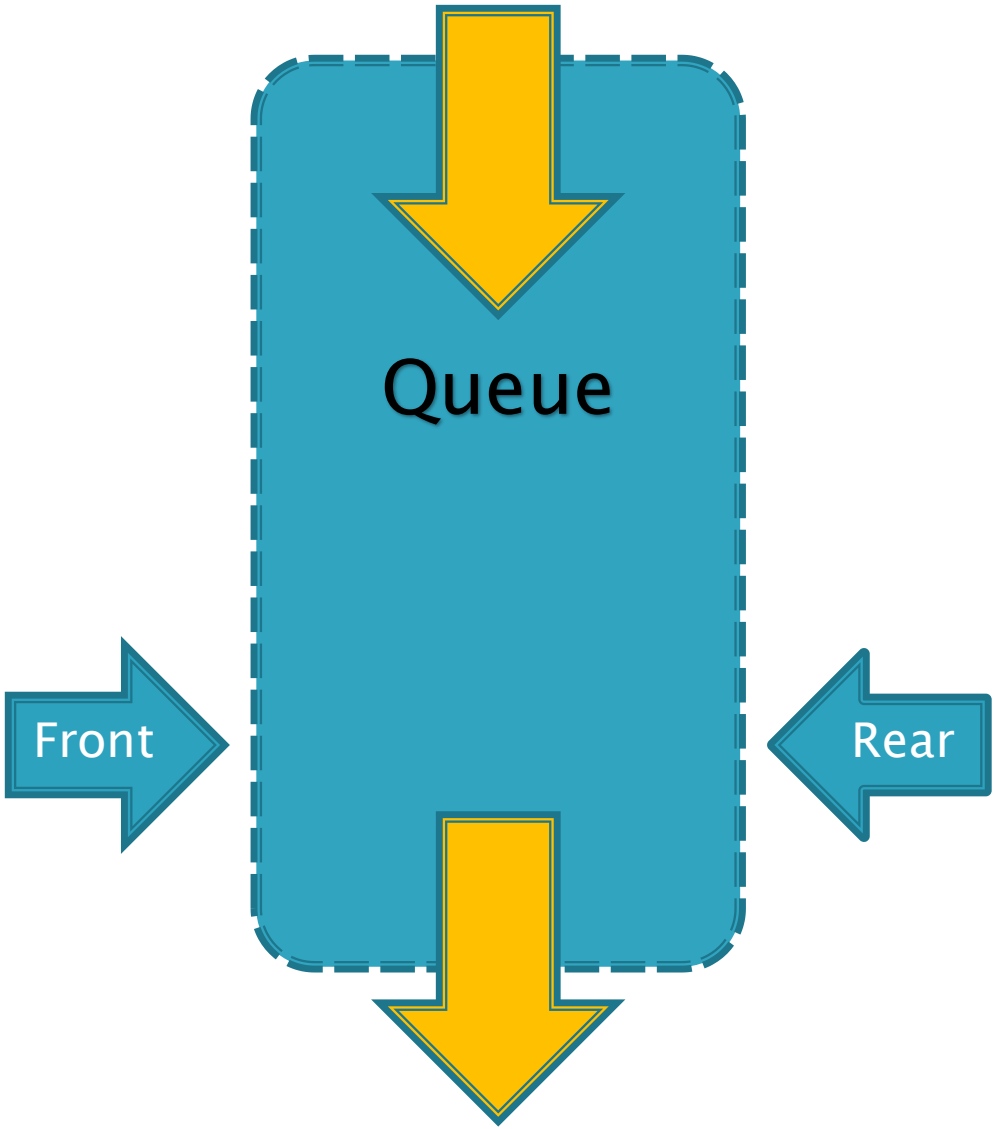
|Rear – Front| = Size

队列(Queue)

- 什么是**循环队列**?

Size = 3

Status: Empty?

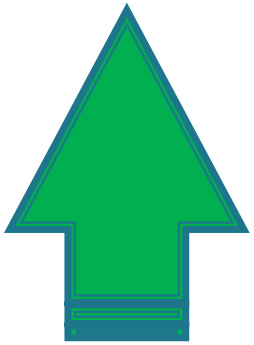


队列(Queue)

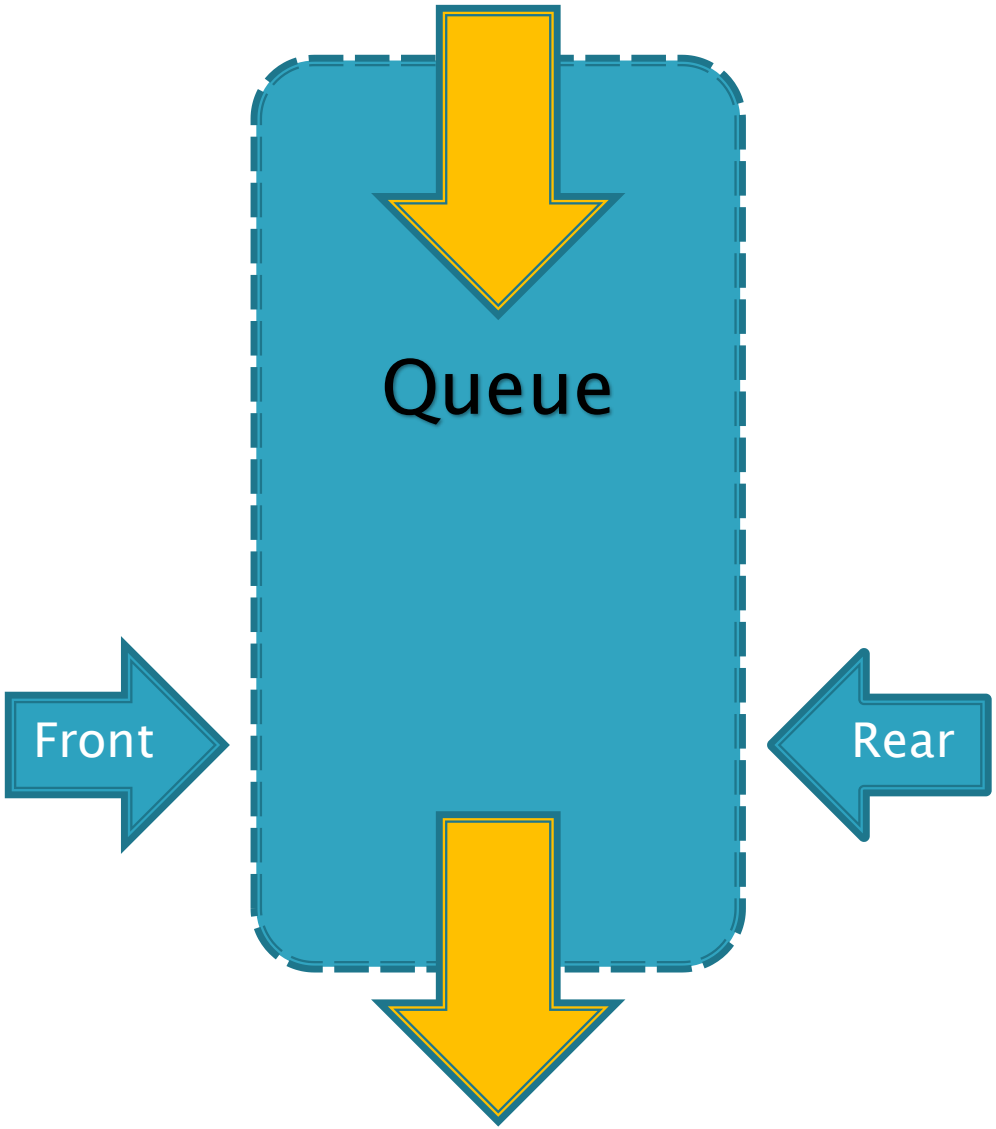
- 什么是**循环队列**?

Size = 3

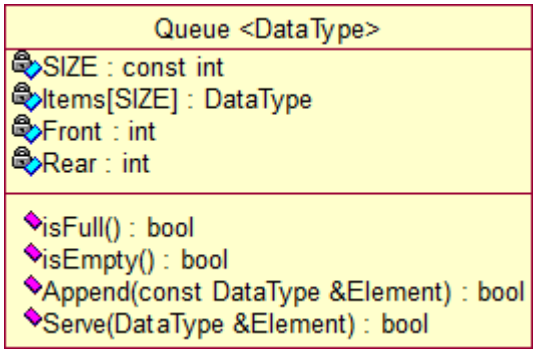
Status: Empty



Rear = Front



队列(Queue)

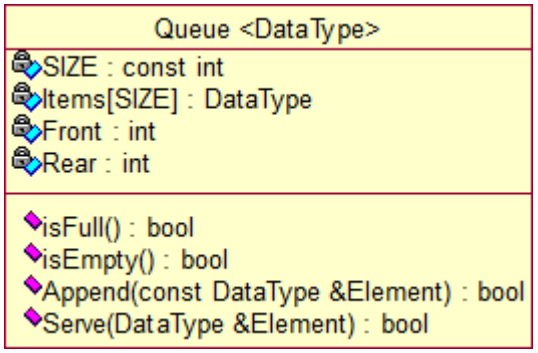


- 如何定义**循环队列**?
- Pascal描述

```
var
   Queue: array[1..SIZE] of integer;
   Front, Rear: integer;

Procedure Append(x: integer)
Begin
    if (Abs(Rear - Front) = SIZE) then writeln('OverFlow!');
    else
        begin
            Queue[Rear mod SIZE]:= x;  Inc(Rear);
        end;
End;
```

队列(Queue)



- 如何定义**循环队列**?
- Pascal描述

```
var
   Queue: array[1..SIZE] of integer;
   Front, Rear: integer;

Function Serve: integer
Begin
    if (Rear = Front) then writeln('UnderFlow!');
    else
        begin
            Serve:= Queue[Front mod SIZE];  Inc(Front);
        end;
End;
```

队列(Queue)

- 如何定义**循环队列**?
- C/C++描述

```
int Queue[SIZE] = {0};
int Front, Rear = 0;

bool Append(const int &x)
{
    if (abs(Rear - Front) == SIZE) return false;
    else
    {
        Queue[Rear++ % N] = x;
        return true;
    }
}
```

队列(Queue)

- 如何定义**循环队列**?
- C/C++描述

```
int Queue[SIZE] = {0};
int Front, Rear = 0;

bool Serve(int &x)
{
    if (Front == Rear) return false;
    else
    {
        x = Queue[Front++ % N];
        return true;
    }
}
```

队列(Queue)

- **使用队列**
  - 约瑟夫(Josephus)问题
  - N个人围成一圈，从第一个开始报数，每次报到M的人出列，直到最后一个，求剩下的人的序号。

    Input：　6　5

    Output：　1

队列(Queue)

- **使用队列**
  - 约瑟夫(Josephus)问题
  - N个人围成一圈，从第一个开始报数，每次报到M的人出列，直到最后一个，求剩下的人的序号。

Pascal描述

```
while  (isEmpty() = false) do
begin
    Person:= Serve();  Inc(k);
    if (k mod m = 0) then continue;
    else Append(Person);
end;
```

队列(Queue)

- **使用队列**
  - 约瑟夫(Josephus)问题
  - N个人围成一圈，从第一个开始报数，每次报到M的人出列，直到最后一个，求剩下的人的序号。

C/C++描述

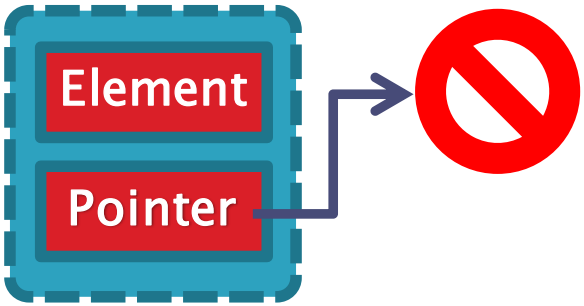
```
while  (isEmpty() == false)
{
    Serve(Person);  k++;
    if (k % m == 0) continue;
    else  Append(Person);
}
```

数据结构

[Linked List]链表

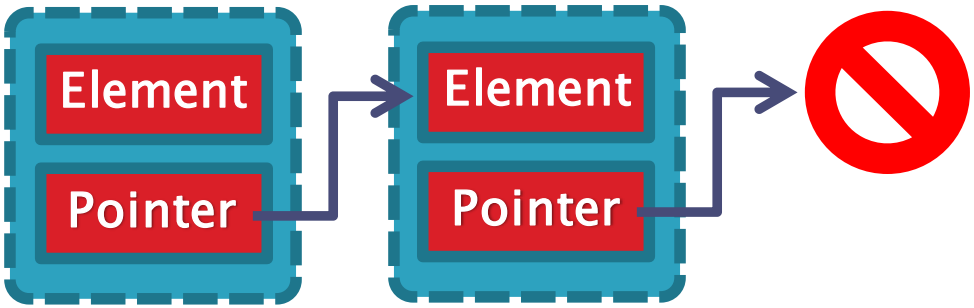
链表(Linked List)

- **什么是链表？**
  - 链表(Linked list)是一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个结点里存到下一个结点的指针(Pointer)。



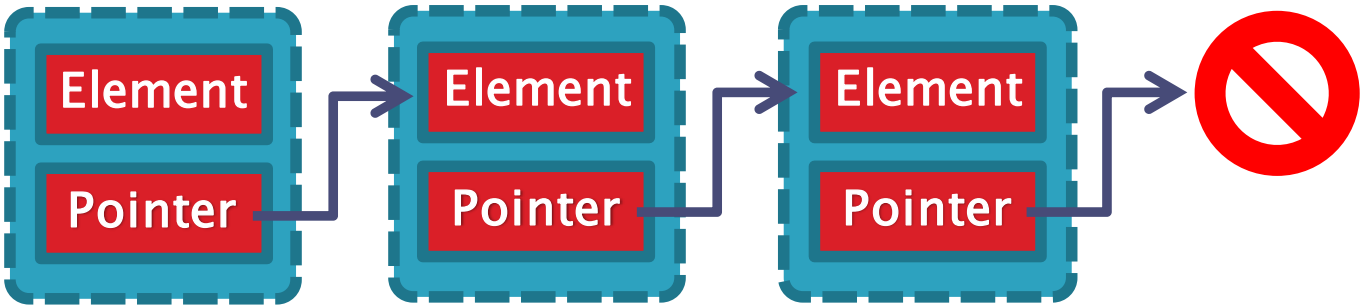
链表(Linked List)

- **什么是链表？**
  - 链表(Linked list)是一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个结点里存到下一个结点的指针(Pointer)。



链表(Linked List)

- **什么是链表？**
  - 链表(Linked list)是一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个结点里存到下一个结点的指针(Pointer)。



链表(Linked List)

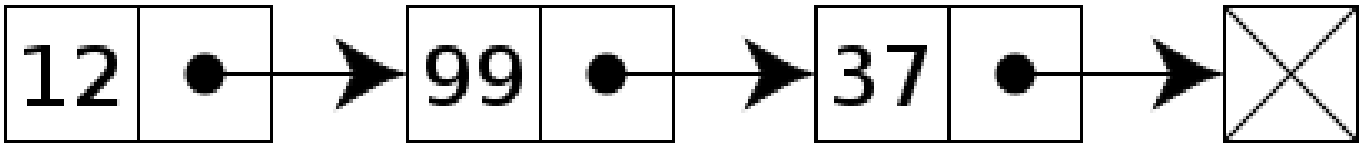
- **链表的特点**
  - 链表通常由一连串结点组成，每个结点包含任意的实例数据(Data Fields)和一或两个用来指向明上一个/下一个结点的位置的链接(Links)
  - 使用链表结构可以克服数组链表需要预先知道数据大小的缺点，链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。

链表(Linked List)

- **链表与顺序表**
  - 相关操作时间复杂度的比较：
    - 查找/访问元素：
      - 顺序表： O(1)　　链表：O(n)
    - 添加/删除元素：
      - 顺序表： O(n)　　链表：O(1)
  - 因此，链表常用于组织删除、检索较少，而添加、遍历较多的数据。

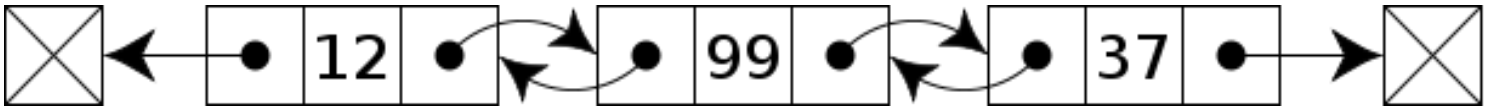
链表(Linked List)

- **链表的结构**
  - **单向链表**
  - 链表中最简单的一种是单向链表，它包含两个域，一个信息域和一个指针域。这个链接指向列表中的下一个结点，而最后一个结点则指向一个空值。



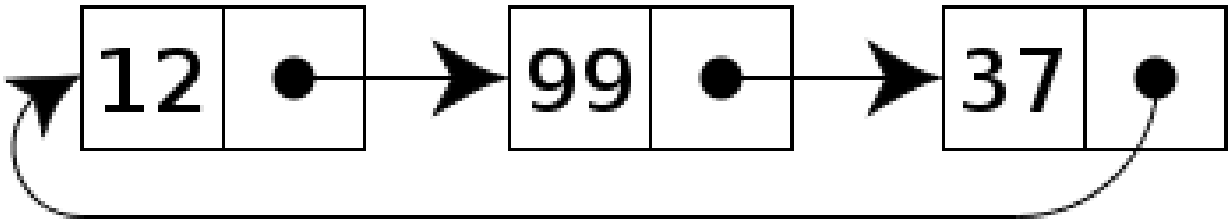
链表(Linked List)

- **链表的结构**
  - **双向链表**
  - 一种更复杂的链表是双向链表 。每个结点有两个连接：一个指向前一个结点，(当此“连接”为第一个“连接”时，指向空值或者空列表)；而另一个指向下一个结点，(当此“连接”为最后一个“连接”时，指向空值或者空列表)



链表(Linked List)

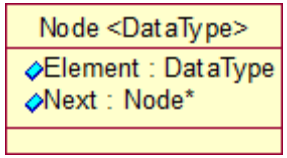
- **链表的结构**
  - **循环链表**
  - 在一个 循环链表中，首结点和末结点被连接在一起。这种方式在单向和双向链表中皆可实现。要转换一个循环链表，你开始于任意一个结点然后沿着列表的任一方向直到返回开始的结点。



链表(Linked List)

- 链表的结构
  - 如何定义单向链表？
  - Pascal描述

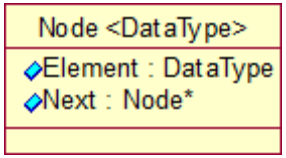
```
Type
    Pointer = ^Node;
    Node = Record
        Element: integer;
        Next: Pointer;
end;
```



链表(Linked List)

- 链表的结构
  - 如何定义单向链表？
  - C语言描述

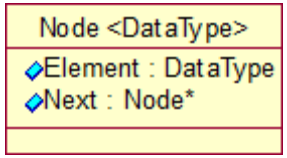
```
typedef struct StructNode
{
    int m_Element;
    struct StructNode*  m_Next;
} Node;
```



链表(Linked List)

- 链表的结构
  - 如何定义单向链表？
  - C++描述

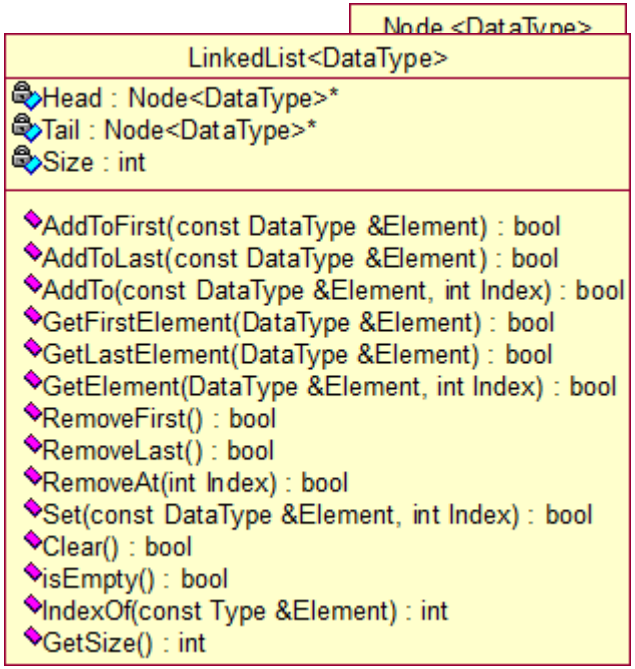
```
template <typename Type>
class Node
{
   public:
      Type m_Element;
      Node* m_Next;
      Node();
      Node(Type Element);
};
```



链表(Linked List)

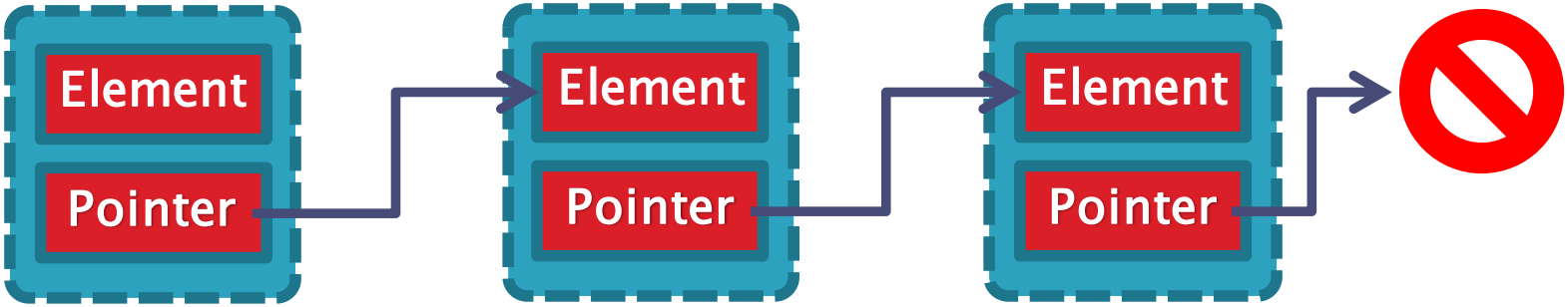
- 链表的结构
  - 如何定义**单向链表**?
  - **C++描述**

```
template <typename Type>
class Node
{
   public:
      Type m_Element;
      Node* m_Next;
      Node();
      Node(Type Element);
};
```



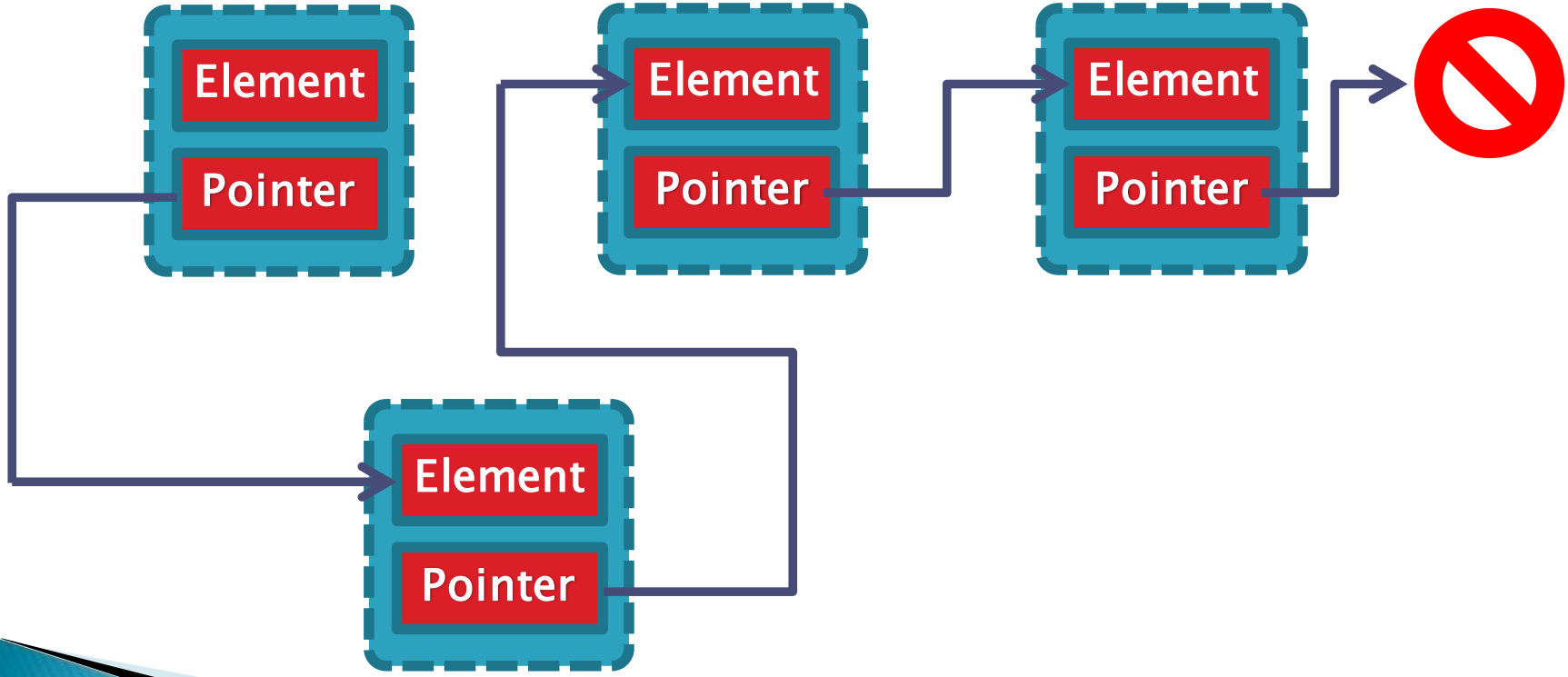
链表(Linked List)

- **链表的结构**
  - 在**单向链表**中插入元素



链表(Linked List)

- **链表的结构**
  - 在**单向链表**中插入元素



链表(Linked List)

- **链表的结构**
  - 在**单向链表**中插入元素
  - **Pascal描述**

```
Function AddTo(const Element: integer; IndexOf: integer): boolean;
begin
    Current:= startNode;
    for i:= 2 to IndexOf - 1 do
      Current:= Current^.m_Next;
    New(NewNode);
    NewNode^.m_Element:= Element;
    NewNode^.m_Next:= Current^.m_Next;
    Current^.m_Next:= NewNode;
    inc(Size);
    exit(true);
end;
```

链表(Linked List)

- **链表的结构**
  - 在**单向链表**中插入元素
  - **C语言描述**

```
_Bool  AddTo(const int* Element, int Index)
{
    Node* Current = startNode;
    for (int i = 1; i < Index; i ++)
        Current = Current->m_Next;
    Node* NewNode = (Node*)malloc(sizeof(Node));
    NewNode->m_Element = *Element;
    NewNode->m_Next = Current->m_Next;
    Current->m_Next = NewNode;
    Size++;
    return TRUE;
}
```

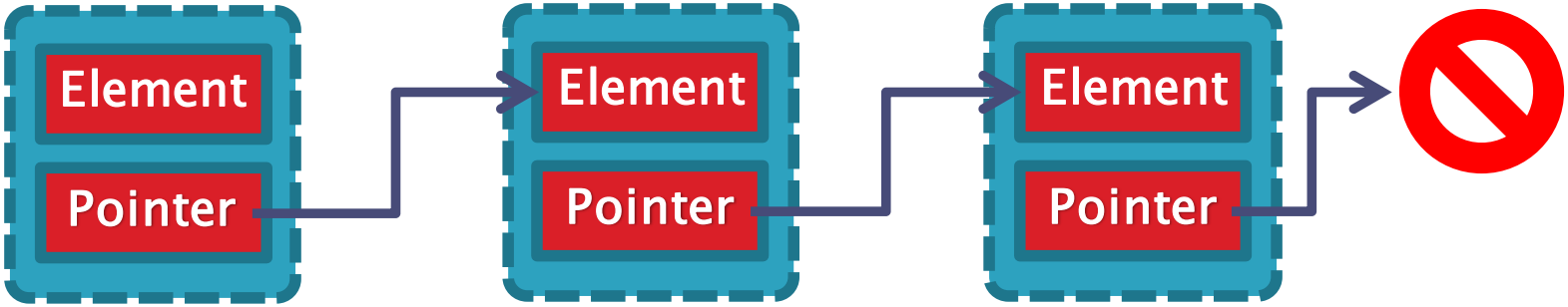
链表(Linked List)

- **链表的结构**
  - 在**单向链表**中插入元素
  - **C++描述**

```
template<typename Type>
bool LinkedList<Type>::AddTo(const Type &Element, int Index)
{
    Node<Type>* Current = startNode;
    for (int i = 1; i < Index; i ++)
        Current = Current->m_Next;
    Node<Type>* NewNode = Current->m_Next;
    Current->m_Next = new Node<Type>(Element);
    (Current->m_Next)->m_Next = NewNode;
    Size++;
    return true;
}
```

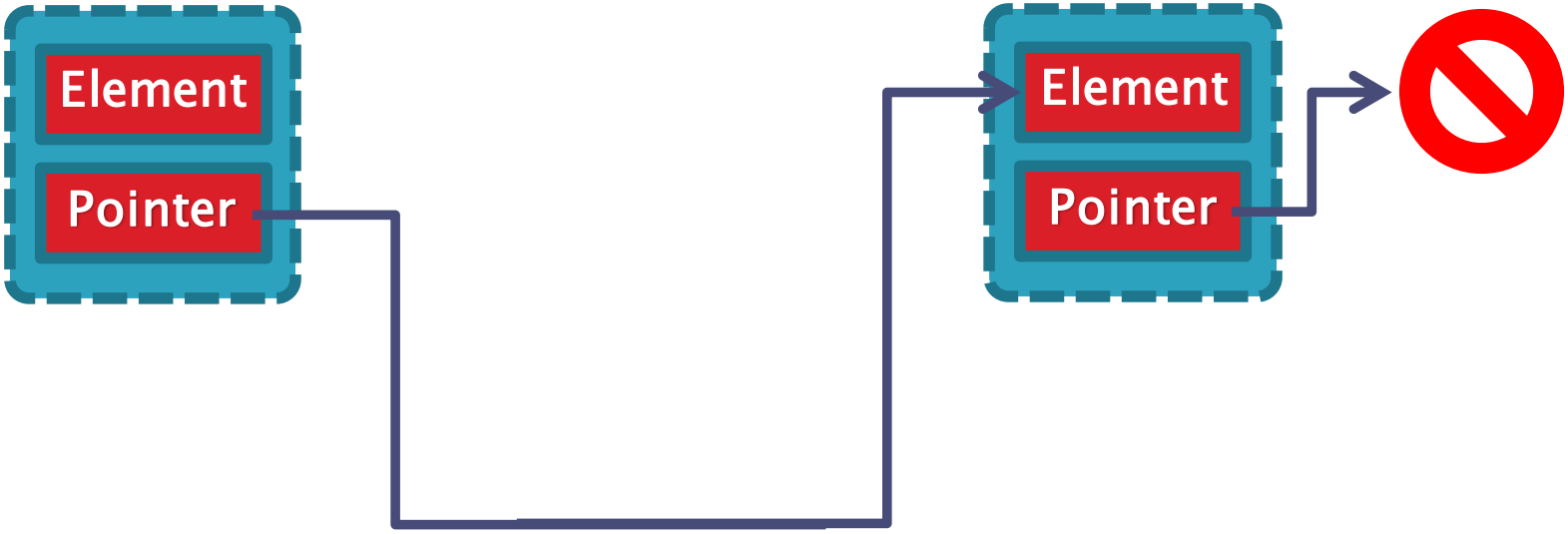
链表(Linked List)

- **链表的结构**
  - 删除**单向链表**中的元素



链表(Linked List)

- **链表的结构**
  - 删除**单向链表**中的元素



链表(Linked List)

- **链表的结构**
  - 删除**单向链表**中的元素
  - **Pascal描述**

```
Function RemoveAt(IndexOf: integer): Boolean;
begin
Current:= startNode;
    for i:= 1 to IndexOf - 2 do
      Current:= Current^.m_Next;
    Current^.m_Next:= (Current^.m_Next)^.m_Next;
    Dispose(Current^.m_Next);
    dec(Size);
    exit(true);
  end;
end;
```

链表(Linked List)

- **链表的结构**
  - 删除**单向链表**中的元素
  - **C语言描述**

```
_Bool RemoveAt(int Index)
{
    int i = 0;
    Node* Current = startNode;
    for (i= 0;  i < Index - 1; i ++)
        Current = Current->m_Next;
    Current->m_Next = Current->m_Next->m_Next;
    free(Current->m_Next);
    Size --;
    return TRUE;
}
```

链表(Linked List)

- **链表的结构**
  - 删除**单向链表**中的元素
  - **C++描述**

```
template <typename Type>
bool LinkedList<Type>::RemoveLast()
{
    Node<Type>* Current = startNode;
    for (int i = 0; i < Size - 2; i ++)
        Current = Current->m_Next;
    delete Tail;
    Tail = Current;
    Tail->m_Next = NULL;
    Size --;
    return true;
}
```

链表(Linked List)

- **链表的结构**
  - 链栈和链队列
  - 栈的链式存贮结构,也称为链栈，它是一种限制运算的链表，即规定链表中的插入和删除运算只能在链表开头进行。在链栈中，只会出现栈空和非空两种状态。

链表(Linked List)

- **链表的相关问题**
  - Finding a **Loop** in a Singly Linked List

  - If a linked list has a cycle:
    - The malformed linked list has no end (no node ever has a null next node pointer)
    - The malformed linked list contains two links to some node
    - Iterating through the malformed linked list will yield all nodes in the loop multiple times

链表(Linked List)

- **链表的相关问题**
  - Finding a Loop in a Singly Linked List
  - Inefficient solution

```
function boolean hasLoop(Node startNode) {
    HashSet nodesSeen = new HashSet();
    Node currentNode = startNode;
    do {
        if (nodesSeen.contains(currentNode))
            return true;
        nodesSeen.add(currentNode);
    }
    while (currentNode = currentNode.next());
    return false;
}
```

链表(Linked List)

- **链表的相关问题**
  - Finding a Loop in a Singly Linked List
  - Best solution

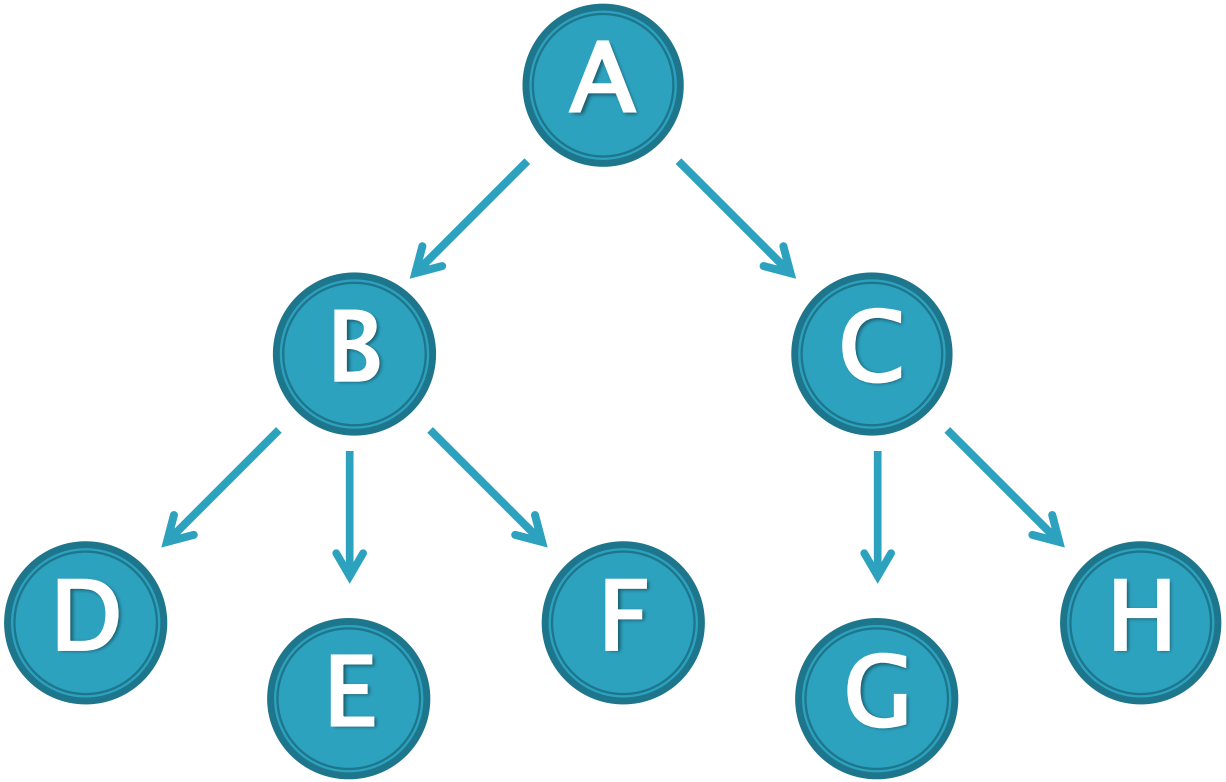
```
function boolean hasLoop(Node startNode) {
    Node *quick_node = startNode->next,
    Node *slow_node = startNode;
    if (startNode == NULL || startNode->next == NULL) {
        return false;
    }
    while (quick_node != slow_node) {
        if (quick_node == NULL || slow_node == NULL)
            break;
        quick_node = quick_node->next->next;
        slow_node = slow_node->next;
    }
    if (quick_node != NULL && slow_node != NULL)
        return true;
    return false;
}
```

数据结构

[Tree]树

树(Tree)

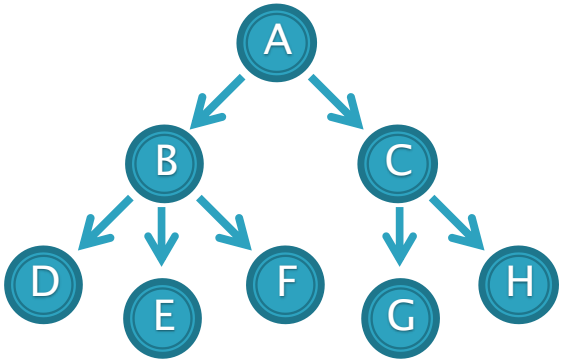
- 什么是树？



树(Tree)

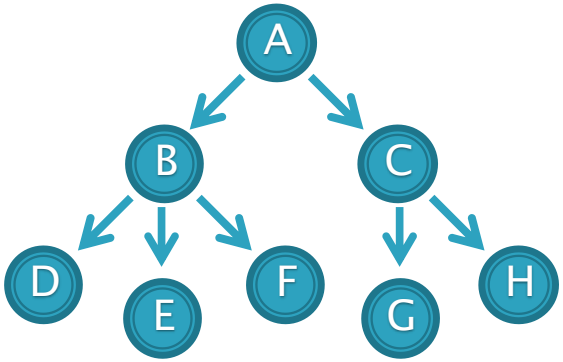
- **什么是树？**
  - 树是一种数据结构，它是由n(n>=1)个有限结点组成一个具有层次关系的集合。把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。它具有以下的特点：
    - 每个结点有零个或多个子结点；
    - 每一个子结点只有一个父结点；
    - 没有前驱的结点为根结点；
    - 除了根结点外，每个子结点可以分为m个不相交的子树；

树(Tree)



- **树的基本概念：**
  - **结点的度**：一个结点含有的子树的个数称为该结点的度；
  - **叶结点**(终端结点)：度为零的结点称为叶结点；
  - **非终端结点**(分支结点)：度不为零的结点；
  - **双亲结点**(父结点)：若一个结点含有子结点，则这个结点称为其子结点的父结点；
  - **孩子结点**(子结点)：一个结点含有的子树的根结点称为该结点的子结点；
  - **兄弟结点**：具有相同父结点的结点互称为兄弟结点；
  - **树的度**：一棵树中，最大的结点的度称为树的度；

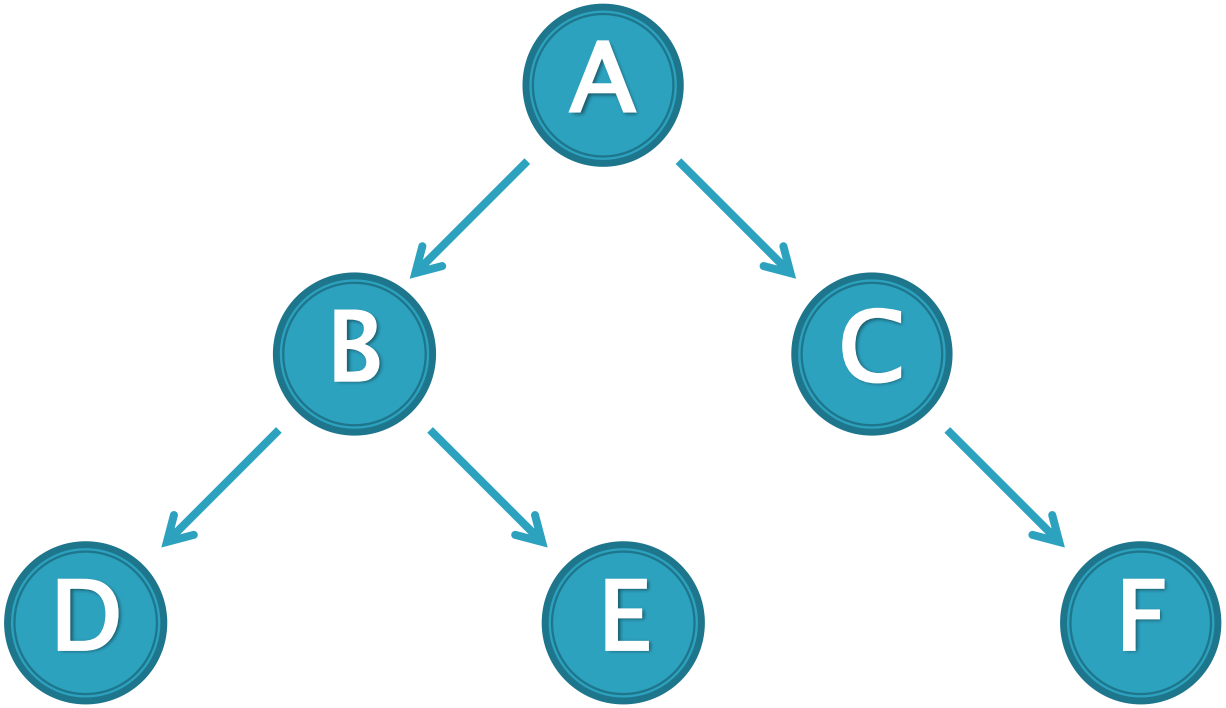
树(Tree)



- **树的基本概念：**
  - **结点的层次**：从根开始定义起，根为第1层，根的子结点为第2层，以此类推；
  - **树的高度**(深度)：树中结点的最大层次；
  - **堂兄弟结点**：双亲在同一层的结点互为堂兄弟；
  - **结点的祖先**：从根到该结点所经分支上的所有结点；
  - **子孙**：以某结点为根的子树中任一结点都称为该结点的子孙；
  - **森林**：由m(m>=0)棵互不相交的树的集合称为森林。

二叉树(Binary Tree)

- 什么是二叉树？

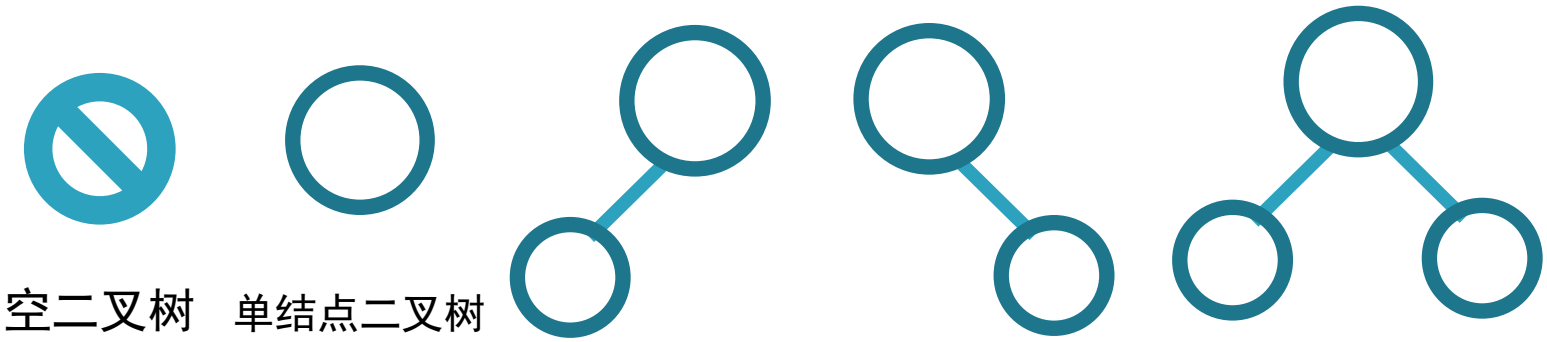


二叉树(Binary Tree)

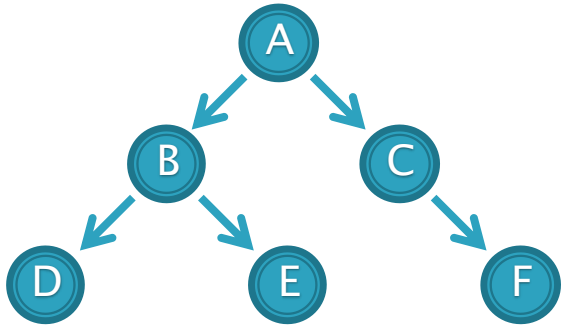
- **什么是二叉树？**
  - 在计算机科学中，二叉树是n个结点的有限集合，其中n≥0，当n=0时，T为空树，否则，其中有一个结点为根结点，其余结点划分为两个互不相交的子集T_L、T_R，并且T_L、T_R分别构成二叉树T的左、右子树。
  - 二叉树的五种形态：

二叉树(Binary Tree)

- **什么是二叉树？**
  - 在计算机科学中，二叉树是n个结点的有限集合，其中n≥0，当n=0时，T为空树，否则，其中有一个结点为根结点，其余结点划分为两个互不相交的子集T_L、T_R，并且T_L、T_R分别构成二叉树T的左、右子树。
  - 二叉树的五种形态：



二叉树(Binary Tree)



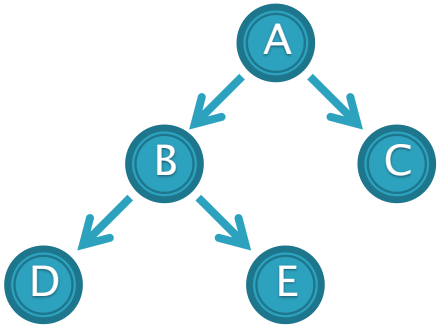
- 二叉树的性质：
  - 性质1：在二叉树的第i层上的结点数≤ $2^{i-1}(i > 0)$
  - 性质2：深度为k的二叉树的结点数≤ $2^k - 1(k > 0)$
  - 性质3：对任一棵非空的二叉树T，若其叶子数为n_0，度为2的结点数为n_2，则：$n_0 = n_2 + 1$
    - 证明：

      设T的总结点数为n，度为1的结点数为n_1，则T的结点数满足关系式：$n = n_0 + n_1 + n_2$

      一方面，这些分支在这n个结点中，除根以外，每个结点有一个分支进入，因而总分支数为n - 1,。另一方面，这些分支仅从度为1和2的结点发出，因而有下列关系式成立：$n - 1 = n_1 + 2n_2$

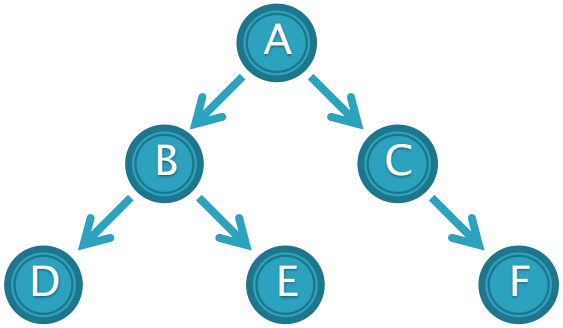
      因此有：$n_0 = n_2 + 1$

二叉树(Binary Tree)



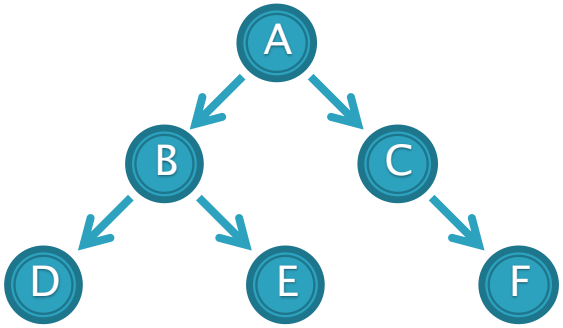
- 满二叉树和完全二叉树
  - **满二叉树**：指每层都有最大结点的二叉树，即满二叉树是高度为k且有$2^k - 1$个结点的二叉树。
  - **完全二叉树**：指在满二叉树的最下层从右到左连续地删除若干个所得到的二叉树。
- 完全二叉树的性质：
  - 性质1：有n个结点的完全二叉树的深度为$\lfloor \log_2 n + 1 \rfloor$
  - 性质2：在编号的完全二叉树中，各结点编号之间的关系为：
    - ①若编号为i的结点存在左孩子结点，则左孩子结点的编号为2i
    - ②若编号为i的结点存在右孩子结点，则左孩子结点的编号为2i+1
    - ③若编号为i的结点存在父结点，则其父结点的编号为$\lfloor i/2 \rfloor$

二叉树(Binary Tree)



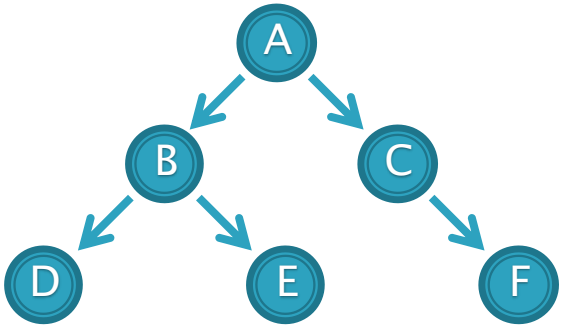
- 二叉树的遍历(Traversal)
  - **什么是遍历？**
    - 遍历是指按某种顺序访问二叉树中每个结点一次(**仅一次**)
    - 一棵非空的二叉树由根结点及左、右子树三个基本部分组成。因此，在任一给定结点上，可以按某种次序执行三个操作：
      - ①访问结点本身(N)
      - ②遍历该结点的左子树(L)
      - ③遍历该结点的右子树(R)
    - 以上三种操作有六种执行次序：
      - NLR，NRL，LNR，RNL，LRN，RLN

二叉树(Binary Tree)



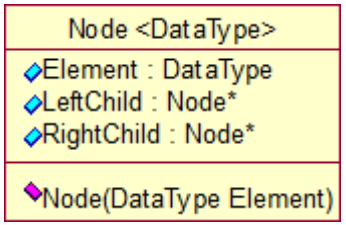
- 二叉树的遍历(Traversal)
  - 基本遍历方法：
    - **先序遍历**(Preorder Traversal)
      - NLR：先访问根结点，然后分别遍历左右子树
      - A → B → D → E → C → F
    - **中序遍历**(Inorder Traversal)
      - LNR：访问根结点在遍历其左右子树之间
      - D → B → E → A → C → F
    - **后序遍历**(Postorder Traversal)
      - LRN：先遍历左右子树，然后再访问根结点
      - D → E→ B → F→ C → A

二叉树(Binary Tree)

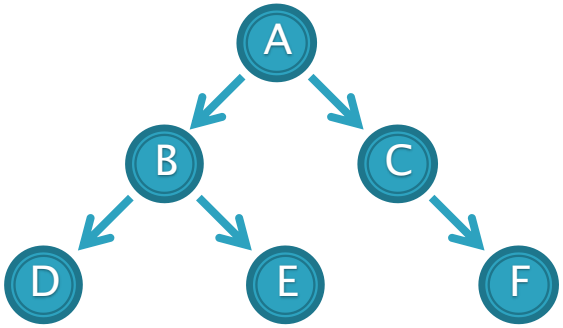


- **二叉树的结构**
  - 如何定义二叉树？
  - **Pascal描述**

```
Type
   Pointer = ^Node;
   Node = Record
      m_Element: integer;
      m_LeftChild: Pointer;
      m_RightChild: Pointer;
end;
```

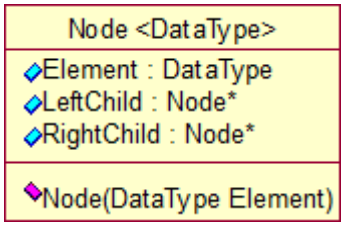


二叉树(Binary Tree)

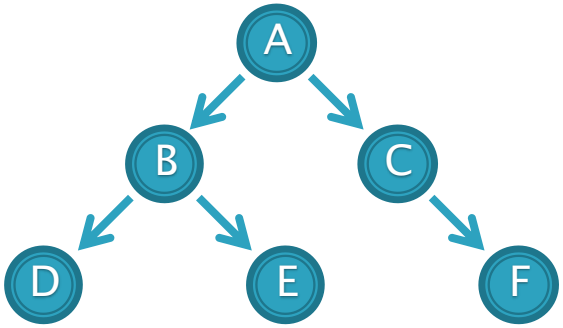


- **二叉树的结构**
  - 如何定义二叉树？
  - **C语言描述**

```
typedef struct StructNode
{
    int m_Element;
    struct StructNode*  m_LeftChild;
    struct StructNode*  m_RightChild;
} Node;
```

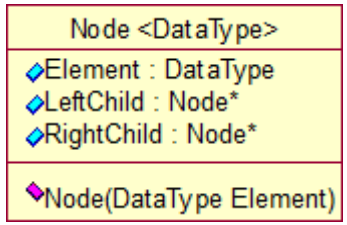


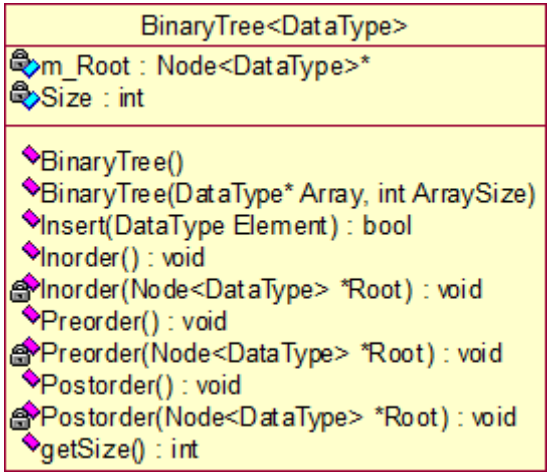
二叉树(Binary Tree)



- **二叉树的结构**
  - 如何定义二叉树？
  - **C++描述**

```
class Node
{
   public:
      Type m_Element;
      Node* m_LeftChild;
      Node* m_RightChild;
      Node();
      Node(Type Element);
};
```





二叉树(Binary Tree)

- 二叉树的结构
  - 向二叉树中插入节点
  - **Pascal 描述**

```
Parent:= nil;  Current:= Root;
while (Current <> nil) do
    begin
        if (Element < Current^.m_Element) then
            begin
                Parent:= Current;
                Current:= Current^.m_LeftChild;
            end
        else if (Element > Current^.m_Element) then
            ...
        else
            exit(false);
end;
```

二叉树(Binary Tree)

- 二叉树的结构
  - 向二叉树中插入节点
  - **Pascal 描述**

```
if (Element < Parent^.m_Element) then
    begin
        New(NewNode);
        NewNode^.m_Element:= Element;
        NewNode^.m_LeftChild:= nil;
        NewNode^.m_RightChild:= nil;
        Parent^.m_LeftChild:= NewNode;
    end
else
    begin
        ...
    end;
```

二叉树(Binary Tree)

- 二叉树的结构
  - 向二叉树中插入节点
  - **C语言描述**

```
Node<Type>* Parent = NULL;
Node<Type>* Current = m_Root;
while (Current != NULL) {
    if (Element < Current->m_Element) {
        Parent = Current;
        Current = Current->m_LeftChild;
    }
    else if (Element > Current->m_Element) {
        Parent = Current;
        Current = Current->m_RightChild;
    }
    else
        return FALSE;
}
```

二叉树(Binary Tree)

- 二叉树的结构
  - 向二叉树中插入节点
  - **C语言描述**

```
if (Element < Parent->m_Element) {
    Parent->m_LeftChild = (Node*)malloc(sizeof(Node));
    Parent->m_LeftChild->m_Element = Element;
    Parent->m_LeftChild->m_LeftChild = NULL;
    Parent->m_LeftChild->m_RightChild = NULL;
}
else
    ...
return TRUE;
```

二叉树(Binary Tree)

- 二叉树的结构
  - 向二叉树中插入节点
  - **C++ 描述**

```
Node<Type>* Parent = NULL;
Node<Type>* Current = m_Root;
while (Current != NULL) {
    if (Element < Current->m_Element) {
        Parent = Current;
        Current = Current->m_LeftChild;
    }
    else if (Element > Current->m_Element) {
        Parent = Current;
        Current = Current->m_RightChild;
    }
    else
        return false;
}
```

二叉树(Binary Tree)

- 二叉树的结构
  - 向二叉树中插入节点
  - C++ 描述

```
if (Element < Parent->m_Element)
    Parent->m_LeftChild = new Node<Type>(Element);
else
    Parent->m_RightChild = new Node<Type>(Element);
return true;
```

二叉树(Binary Tree)

- 二叉树的结构
  - 二叉树的中序遍历
  - **Pascal 描述**

```
Procedure Inorder(Root :Pointer);
begin
  if (Root = nil) then
     exit
  else
    begin
      Inorder(Root^.m_LeftChild);
      write(Root^.m_Element, ' ');
      Inorder(Root^.m_RightChild);
    end;
end;
```

二叉树(Binary Tree)

- 二叉树的结构
  - 二叉树的中序遍历
  - C语言描述

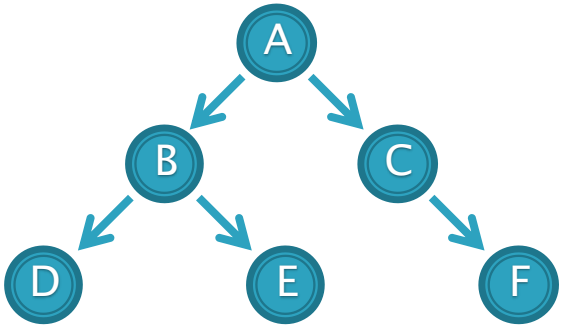
```
void Inorder(Node*  Root)
{
    if (Root == NULL)
        return;
    else
    {
        Inorder(Root->m_LeftChild);
        printf("%d ", Root->m_Element);
        Inorder(Root->m_RightChild);
    }
}
```

二叉树(Binary Tree)

- 二叉树的结构
  - 二叉树的中序遍历
  - C++ 描述

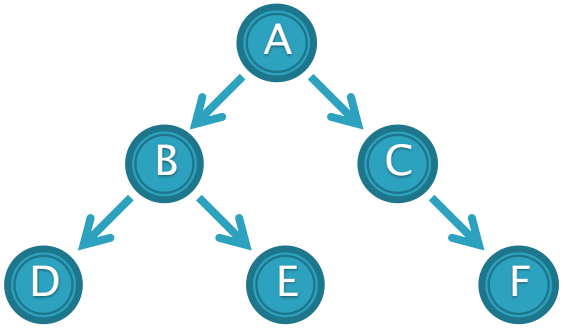
```
template <typename Type>
void BinaryTree<Type>::Inorder(Node<Type>  *Root)
{
    if (Root == NULL)
        return;
    else
    {
        Inorder(Root->m_LeftChild);
        std::cout << Root->m_Element << " ";
        Inorder(Root->m_RightChild);
    }
}
```

二叉树(Binary Tree)



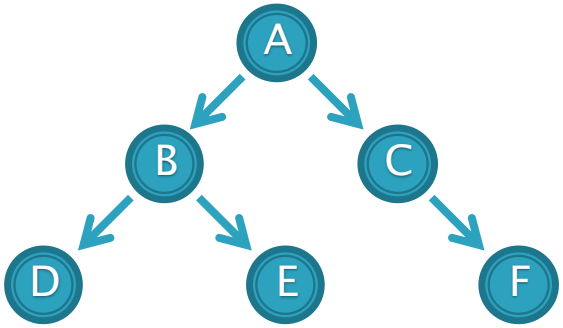
- 深度优先搜索(Deep-First-Search)
  - **什么是深度优先搜索？**
    - 深度优先搜索算法，是搜索算法的一种. 是**沿着树的深度遍历树的节点，尽可能深的搜索树的分支.** 这一过程一直进行到已发现从源节点可达的所有节点为止. 如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止.

二叉树(Binary Tree)

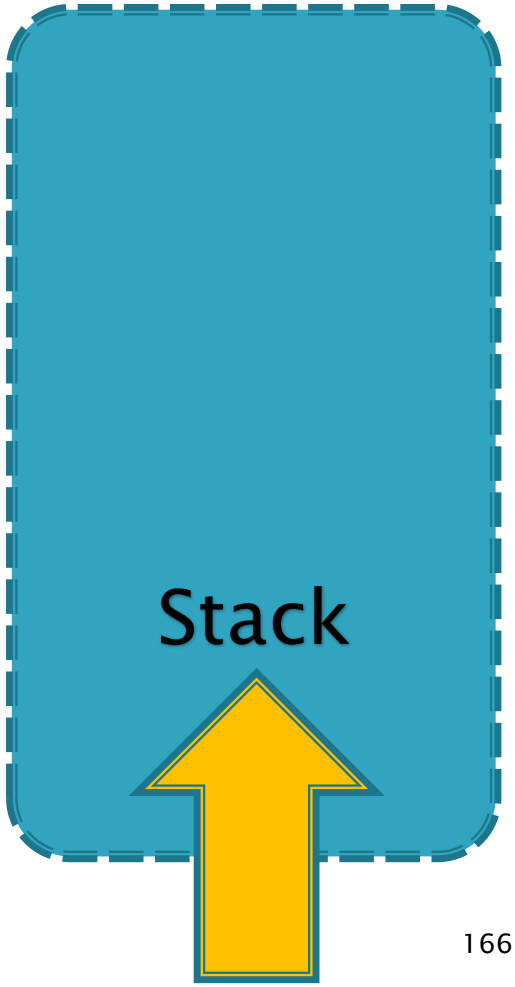


- 深度优先搜索(Deep-First-Search)
  - **深度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和D、然后是E. 然后再是C、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用堆栈的结构, 因为堆栈是一个先进后出的顺序.

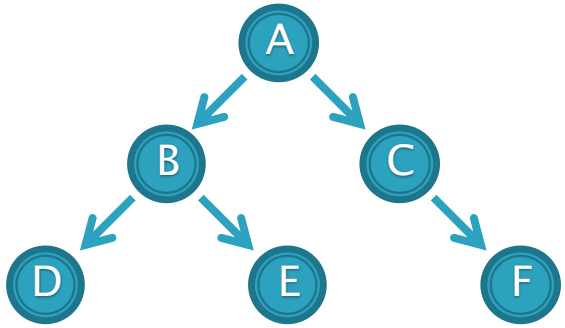
二叉树(Binary Tree)



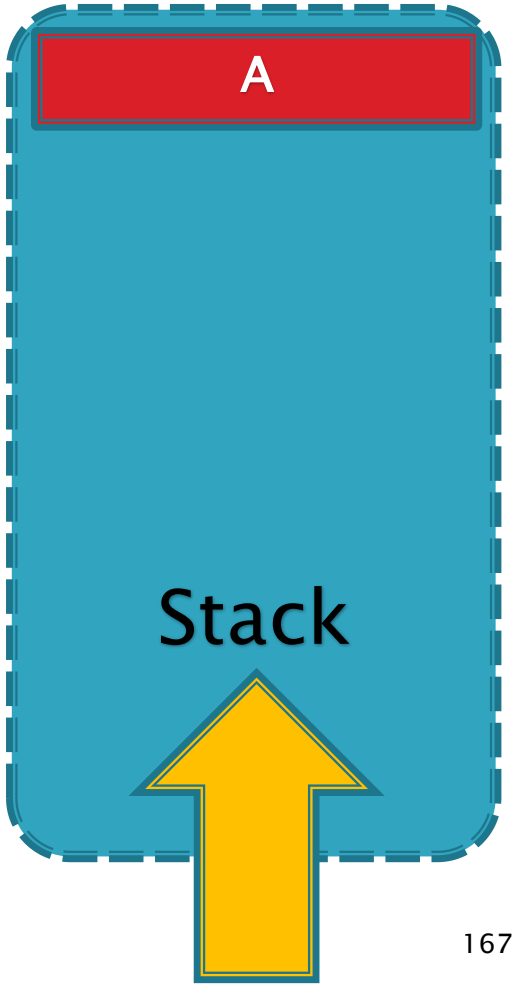
- 深度优先搜索(Deep-First-Search)
  - **深度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和D、然后是E. 然后再是C、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用堆栈的结构, 因为堆栈是一个先进后出的顺序.



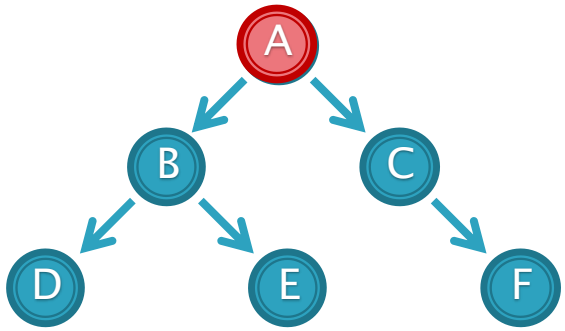
二叉树(Binary Tree)



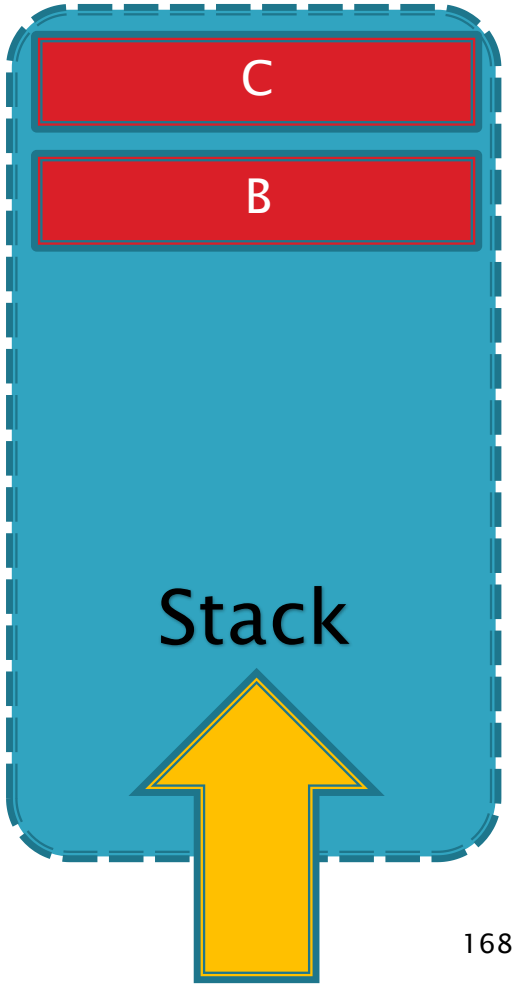
- 深度优先搜索(Deep-First-Search)
  - **深度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和D、然后是E. 然后再是C、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用堆栈的结构, 因为堆栈是一个先进后出的顺序.



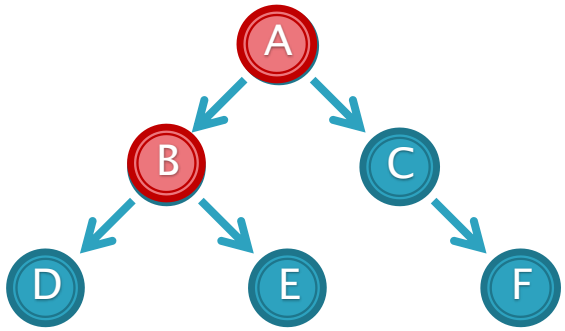
二叉树(Binary Tree)



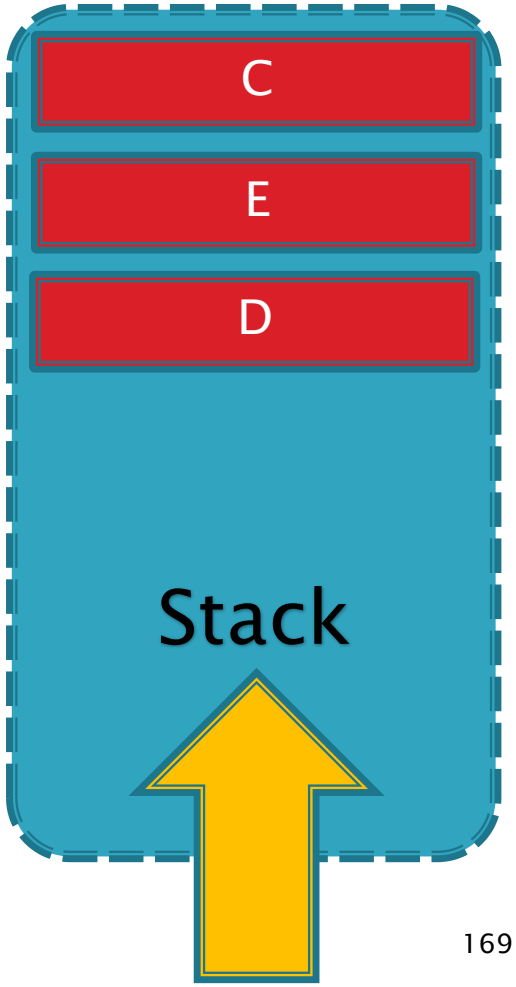
- 深度优先搜索(Deep-First-Search)
  - **深度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和D、然后是E. 然后再是C、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用堆栈的结构, 因为堆栈是一个先进后出的顺序.



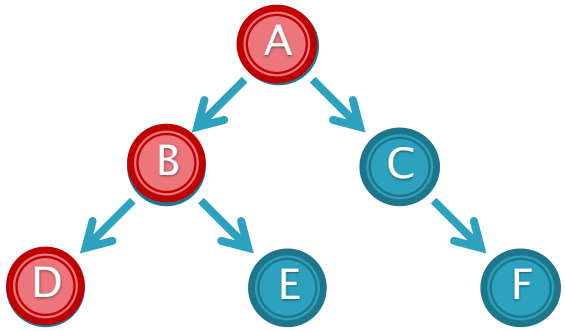
二叉树(Binary Tree)



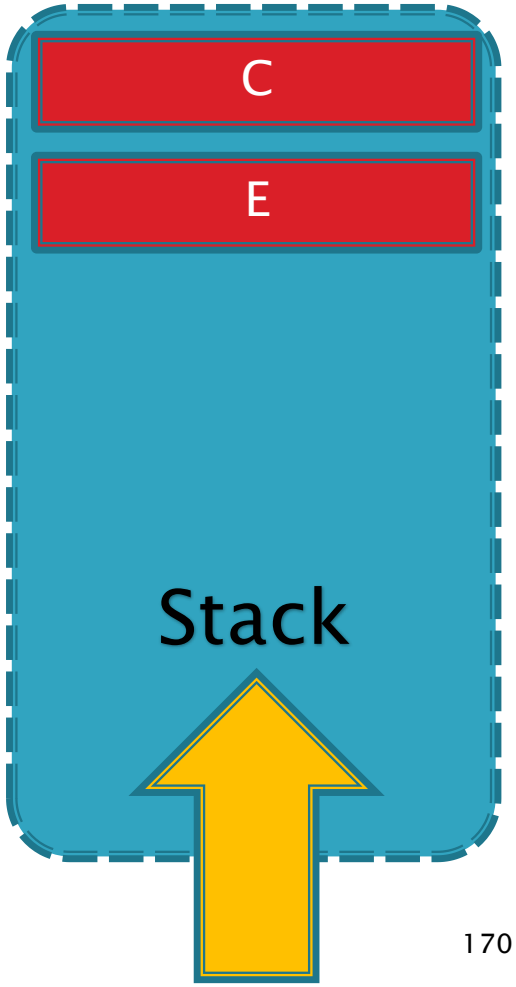
- 深度优先搜索(Deep-First-Search)
  - **深度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和D、然后是E. 然后再是C、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用堆栈的结构, 因为堆栈是一个先进后出的顺序.



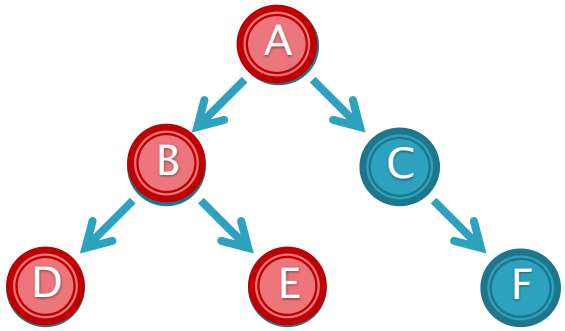
二叉树(Binary Tree)



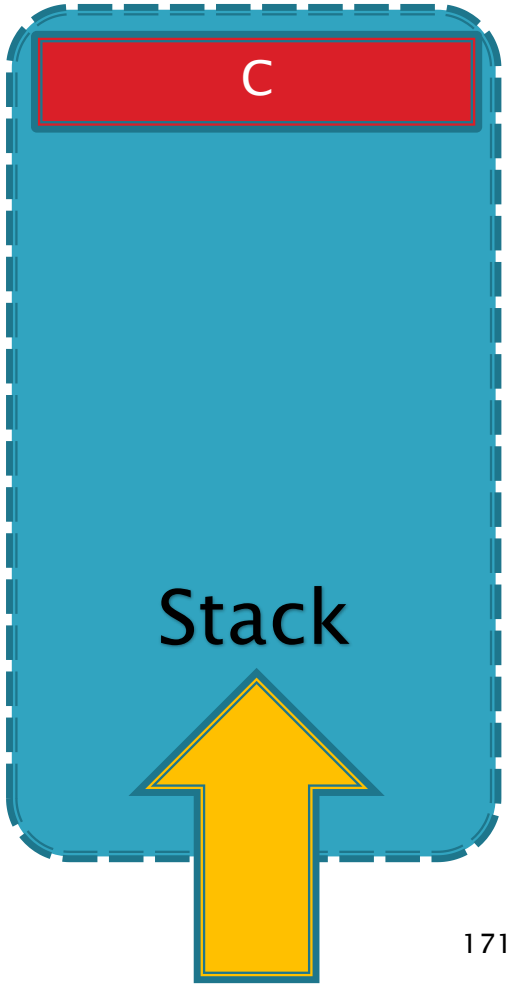
- 深度优先搜索(Deep-First-Search)
  - **深度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和D、然后是E. 然后再是C、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用堆栈的结构, 因为堆栈是一个先进后出的顺序.



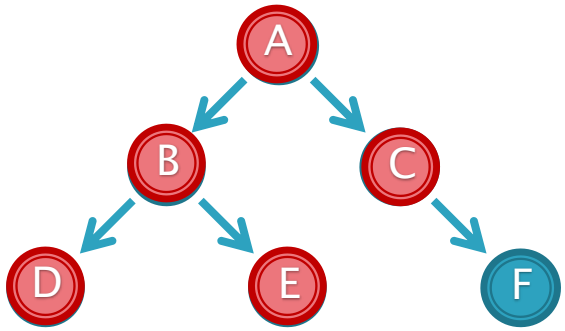
二叉树(Binary Tree)



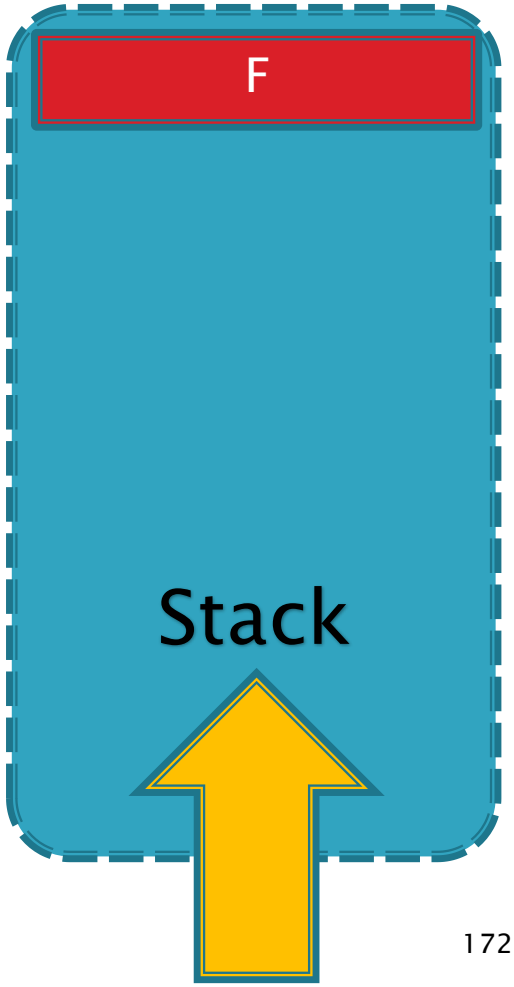
- 深度优先搜索(Deep-First-Search)
  - **深度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和D、然后是E. 然后再是C、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用堆栈的结构, 因为堆栈是一个先进后出的顺序.



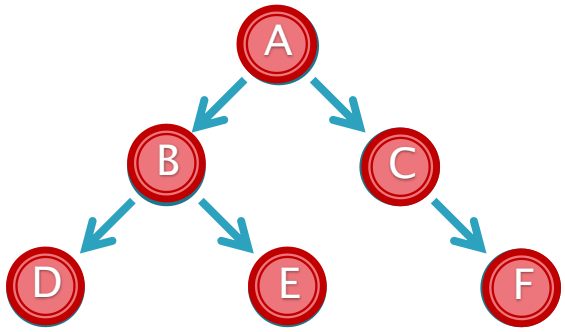
二叉树(Binary Tree)



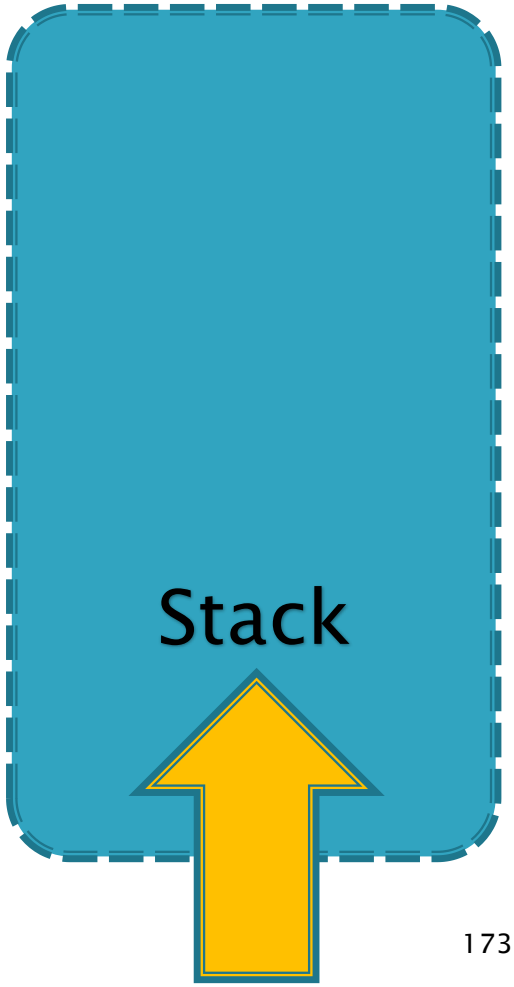
- 深度优先搜索(Deep-First-Search)
  - **深度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和D、然后是E. 然后再是C、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用堆栈的结构, 因为堆栈是一个先进后出的顺序.



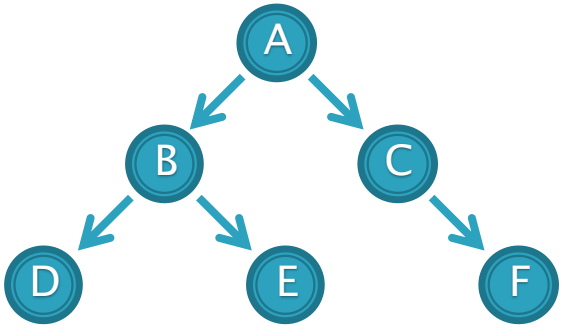
二叉树(Binary Tree)



- 深度优先搜索(Deep-First-Search)
  - **深度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和D、然后是E. 然后再是C、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用堆栈的结构, 因为堆栈是一个先进后出的顺序.



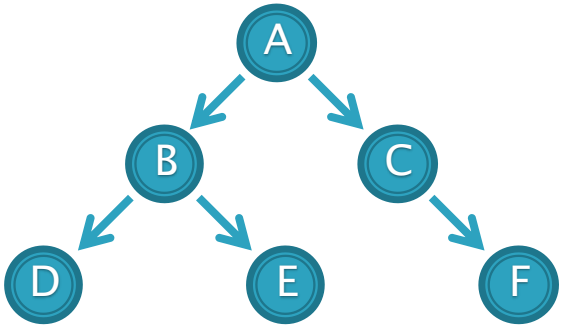
二叉树(Binary Tree)



- 深度优先搜索(Deep-First-Search)
  - C++ 描述

```
UnVisited.push(Tree.getRoot());
while (!UnVisited.empty()) {
        Node<int>* Current = UnVisited.top();
        UnVisited.pop();
        cout << "Now we are at " << Current->m_Element << ".\n";
        // Push right sub tree into the stack
        if (Current->m_RightChild != NULL)
            UnVisited.push(Current->m_RightChild);
        // Push left sub tree into the stack
        if (Current->m_LeftChild != NULL)
            UnVisited.push(Current->m_LeftChild);
}
```

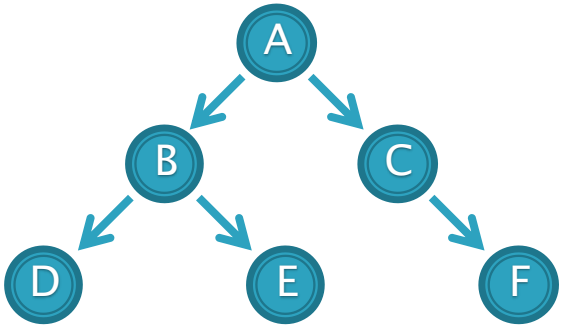
二叉树(Binary Tree)



- 深度优先搜索(Deep-First-Search)
  - Pascal 描述

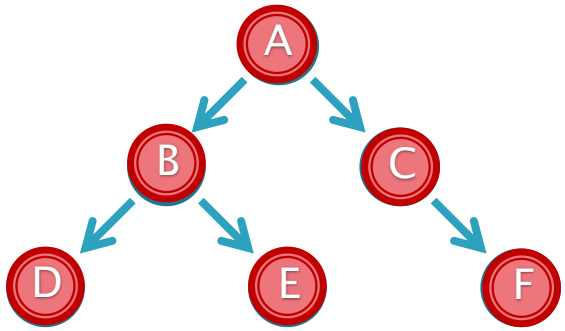
```
Procedure DeepFirstSearch()
    begin
        push(Root);
        while (isEmpty() = false) do
            begin
                Current:= pop();
                write(Current^.m_Element, ' ');
                if (Current^.m_RightChild <> nil) then
                    push(Current^.m_RightChild);
                else
                    push(Current^.m_LeftChild);
            end;
    end;
```

二叉树(Binary Tree)



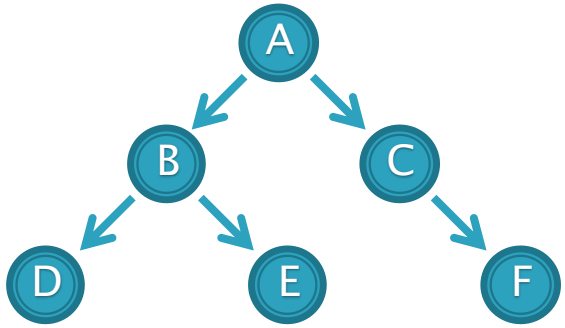
- 广度优先搜索(Breadth-First-Search)
  - **什么是广度优先搜索？**
    - 广度优先搜索算法，是搜索算法的一种. 是**沿着树的宽度遍历树的节点，尽可能宽的搜索树的分支.** 这一过程一直进行到已发现从源节点可达的所有节点为止. 如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止.

二叉树(Binary Tree)



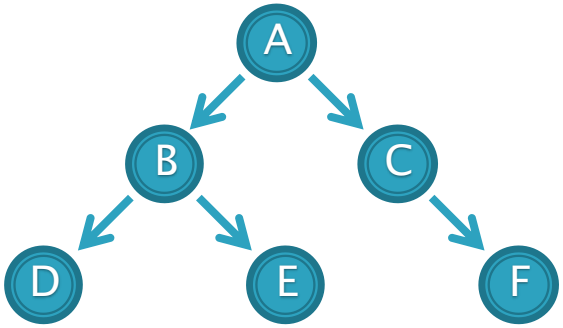
- 广度优先搜索(Breadth-First-Search)
  - **什么是广度优先搜索？**
    - 广度优先搜索算法，是搜索算法的一种. 是**沿着树的宽度遍历树的节点，尽可能宽的搜索树的分支.** 这一过程一直进行到已发现从源节点可达的所有节点为止. 如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止.

二叉树(Binary Tree)



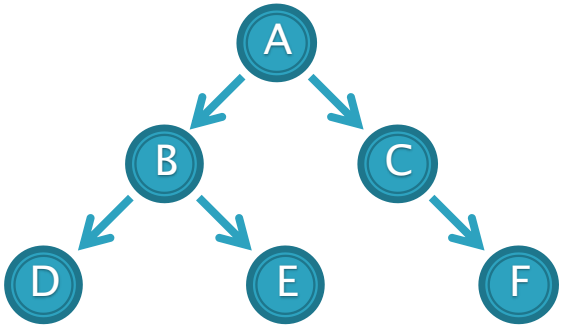
- 广度优先搜索(Breadth-First-Search)
  - **广度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和C然后再是D、E、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用队列的结构, 因为队列是一个先进先出的顺序.

二叉树(Binary Tree)



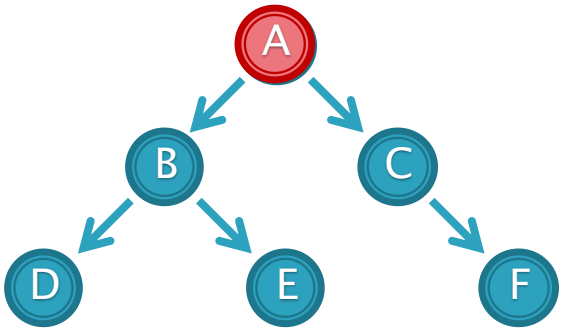
- 广度优先搜索(Breadth-First-Search)
  - **广度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和C然后再是D、E、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用队列的结构, 因为队列是一个先进先出的顺序.

二叉树(Binary Tree)

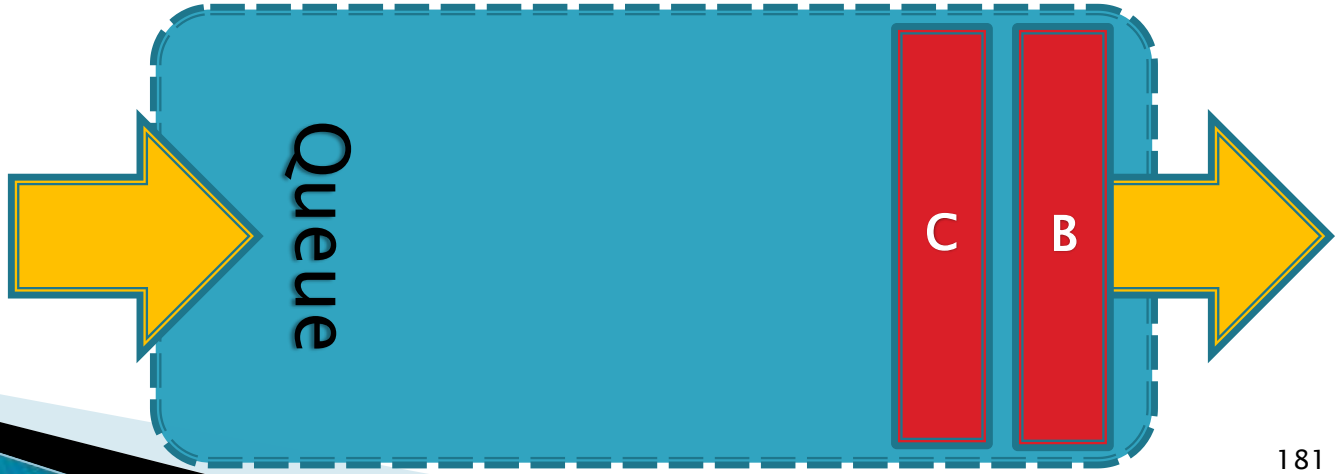


- 广度优先搜索(Breadth-First-Search)
  - **广度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和C然后再是D、E、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用队列的结构, 因为队列是一个先进先出的顺序.

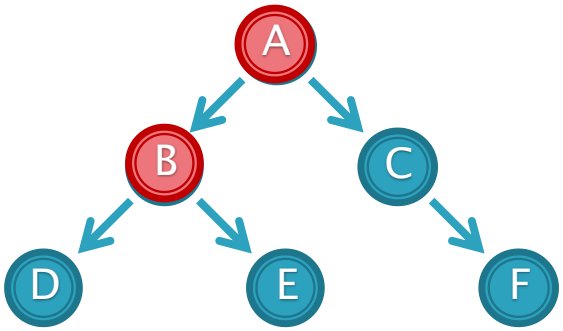
二叉树(Binary Tree)



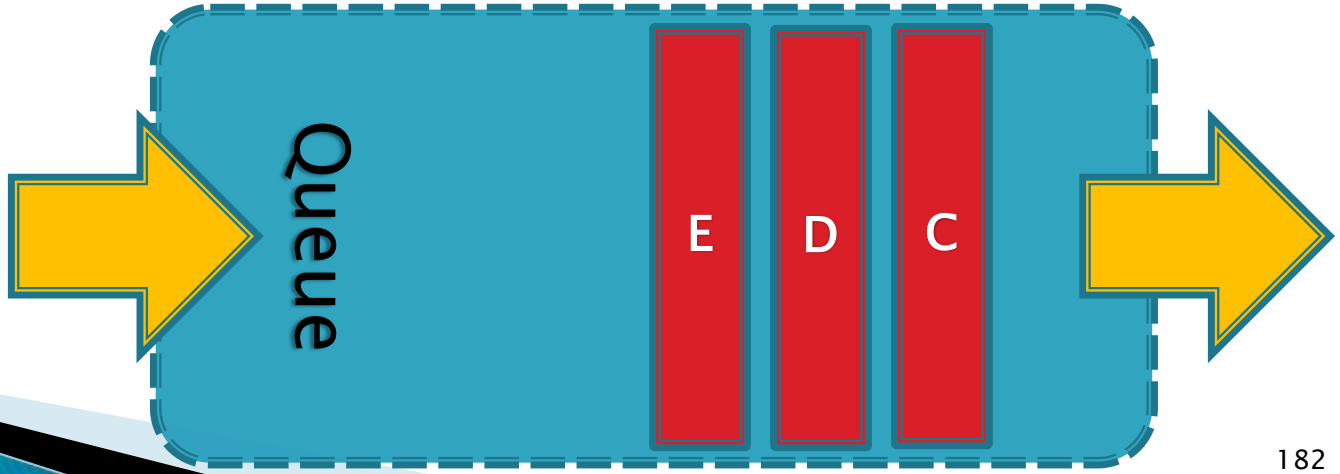
- 广度优先搜索(Breadth-First-Search)
  - **广度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和C然后再是D、E、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用队列的结构, 因为队列是一个先进先出的顺序.



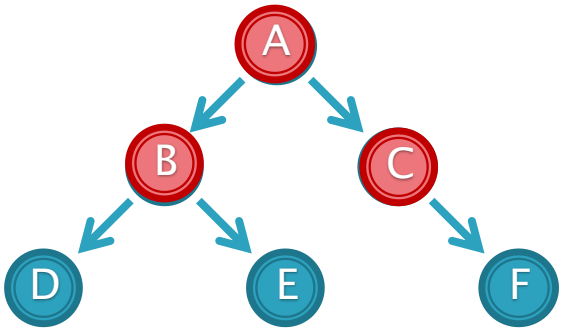
二叉树(Binary Tree)



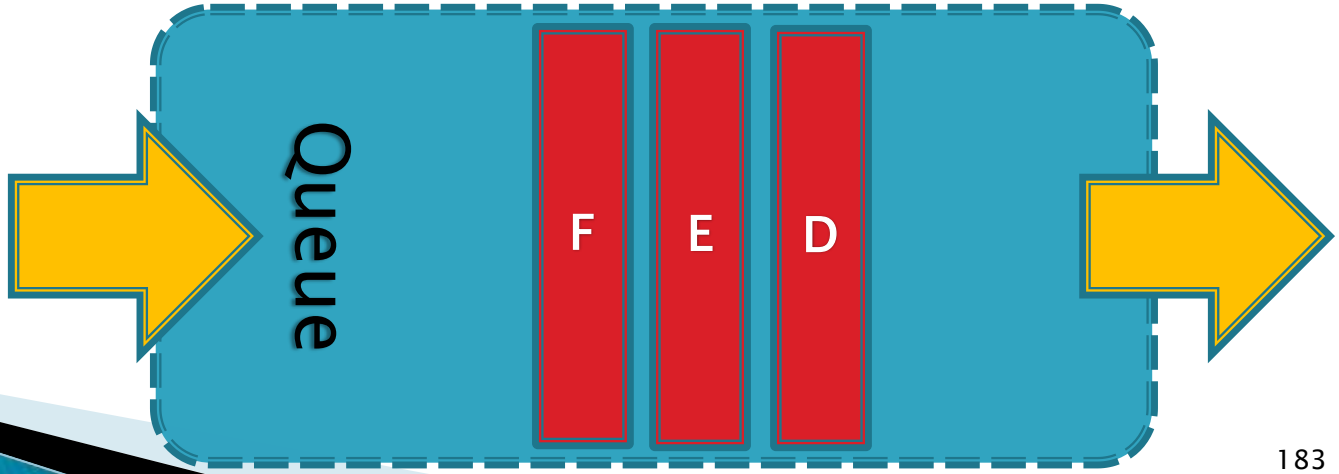
- 广度优先搜索(Breadth-First-Search)
  - **广度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和C然后再是D、E、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用队列的结构, 因为队列是一个先进先出的顺序.



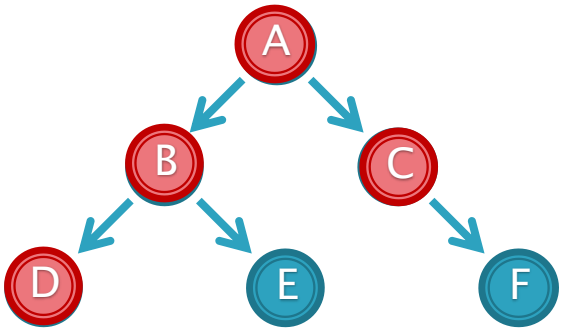
二叉树(Binary Tree)



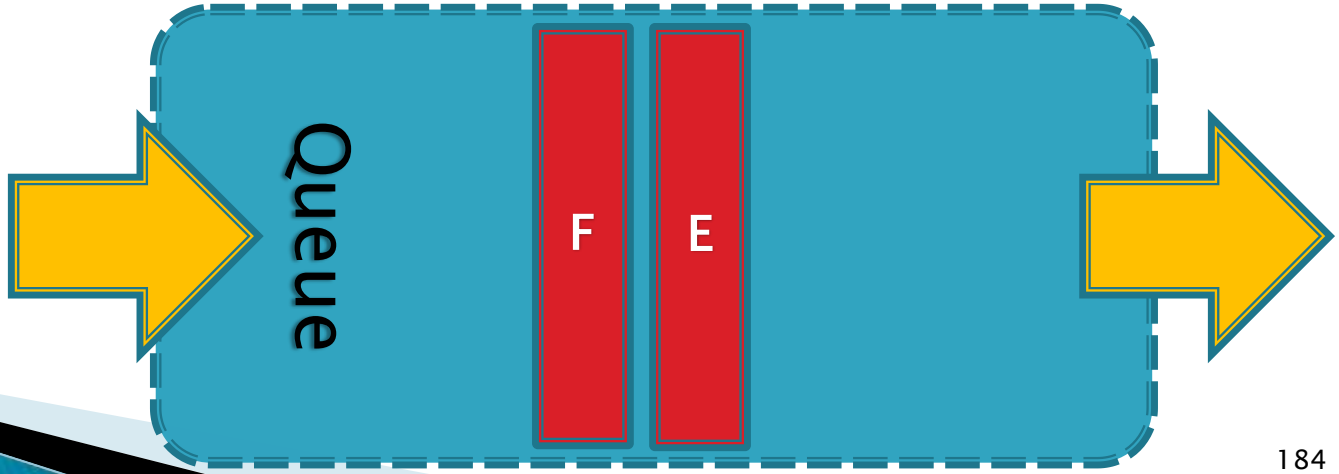
- 广度优先搜索(Breadth-First-Search)
  - **广度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和C然后再是D、E、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用队列的结构, 因为队列是一个先进先出的顺序.



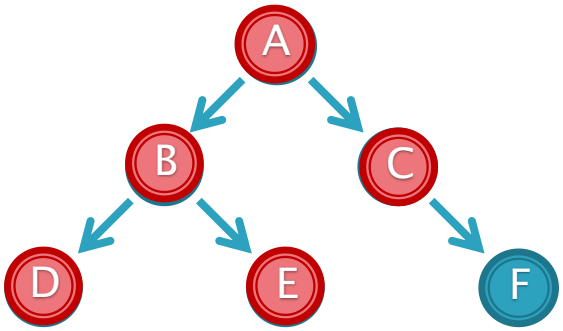
二叉树(Binary Tree)



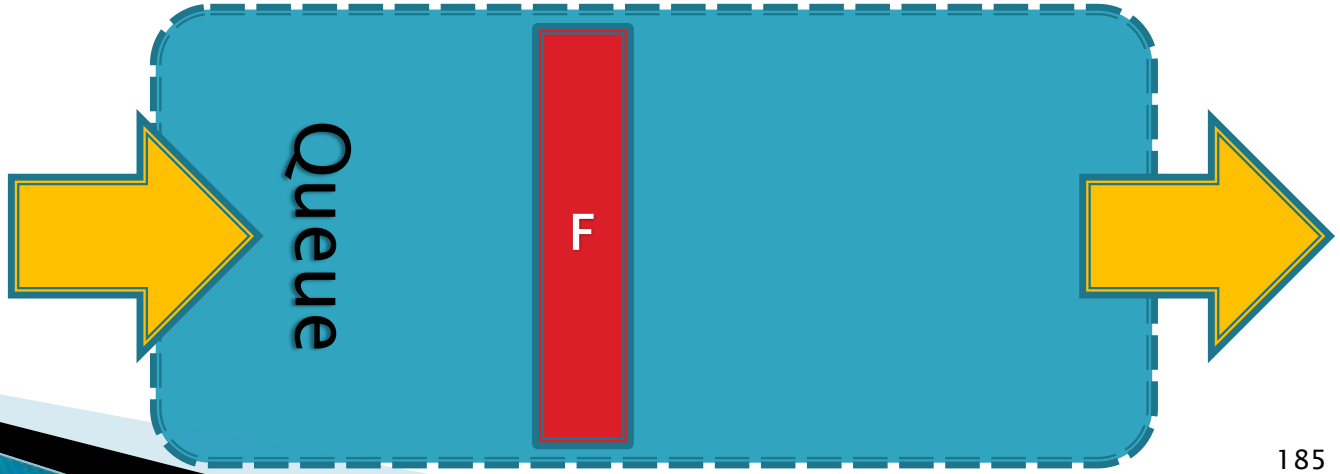
- 广度优先搜索(Breadth-First-Search)
  - **广度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和C然后再是D、E、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用队列的结构, 因为队列是一个先进先出的顺序.



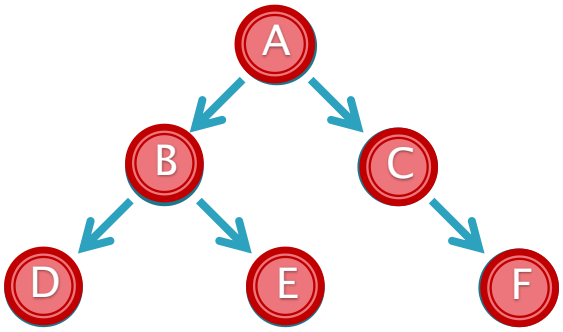
二叉树(Binary Tree)



- 广度优先搜索(Breadth-First-Search)
  - **广度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和C然后再是D、E、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用队列的结构, 因为队列是一个先进先出的顺序.

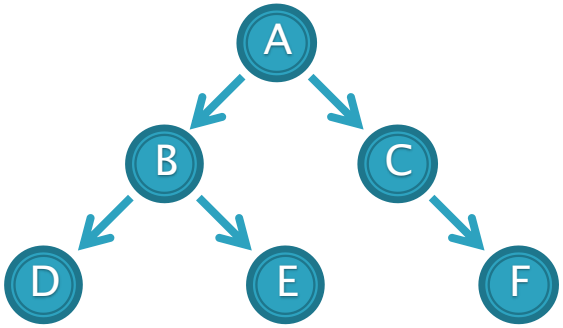


二叉树(Binary Tree)



- 广度优先搜索(Breadth-First-Search)
  - **广度优先搜索算法**
    - 我们在搜索一个树的时候, 从一个节点开始, 能首先获取的是它的两个子节点.
    - A是第一个访问的, 然后顺序是B和C然后再是D、E、F. 那么我们怎么来保证这个顺序呢?
    - 这里就应该用队列的结构, 因为队列是一个先进先出的顺序.

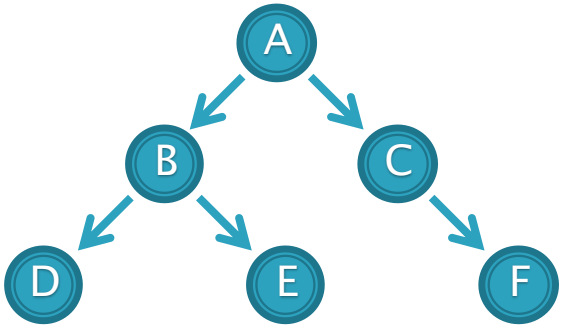
二叉树(Binary Tree)



- 广度优先搜索(Breadth-First-Search)
  - C++ 描述

```
UnVisited.push(Tree.getRoot());
while (!UnVisited.empty()) {
        Node<int>* Current = UnVisited.front();
        UnVisited.pop();
        cout << "Now we are at " << Current->m_Element << ".\n";
        // Push left sub tree into the queue
        if (Current->m_LeftChild != NULL)
            UnVisited.push(Current->m_LeftChild);
        // Push right sub tree into the queue
        if (Current->m_RightChild != NULL)
            UnVisited.push(Current->m_RightChild);
}
```

二叉树(Binary Tree)



- 广度优先搜索(Breadth-First-Search)
  - Pascal 描述

```
Procedure BreadthFirstSearch()
    begin
        Append(Root);
        while (isEmpty() = false) do
            begin
                Current:= Serve();
                write(Current^.m_Element, ' ');
                if (Current^.m_RightChild <> nil) then
                    Append(Current^.m_RightChild);
                else
                    Append(Current^.m_LeftChild);
            end;
    end;
```

二叉树(Binary Tree)

- **二叉树的相关问题**
  - 求二叉树的最大深度
  - **Pascal 描述**

```
Function getDepth(Root : Pointer) : integer;
var
   DepthLeft, DepthRight : integer;
begin
   if (Root = nil) then
      getDepth:= 0
   else
      begin
         DepthLeft:= getDepth(Root^.m_LeftChild);
         DepthRight:= getDepth(Root^.m_RightChild);
         if DepthLeft > DepthRight then
            getDepth:= DepthLeft + 1;
         else
            getDepth:= DepthRight + 1;
      end;
end;
```

二叉树(Binary Tree)

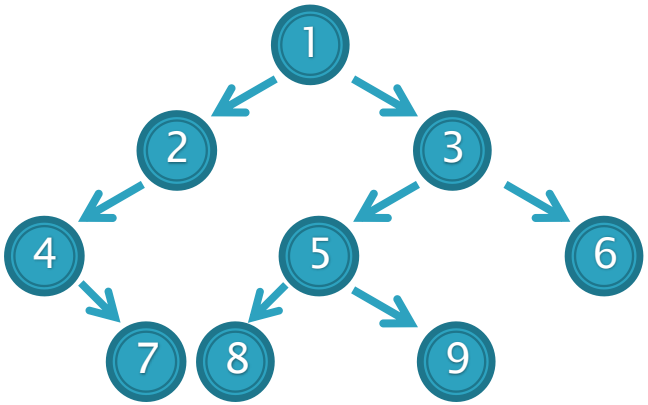
- **二叉树的相关问题**
  - 求二叉树的最大深度
  - **C++ 描述**

```
template <typename Type>
int getDepth(Node<Type>* Root) {
    if (Root == NULL)
        return 0;
    else {
        int DepthLeft = getDepth(Root->m_LeftChild);
        int DepthRight = getDepth(Root->m_RightChild);
        return DepthLeft > DepthRight ? DepthLeft + 1 : DepthRight + 1;
    }
}
```

二叉树(Binary Tree)

- **二叉树的相关问题**
  - 已知前序和中序遍历，返回后序遍历

Input： 9
1 2 4 7 3 5 8 9 6
4 7 2 1 8 5 9 3 6



Output： 7 4 2 8 9 5 6 3 1

二叉树(Binary Tree)

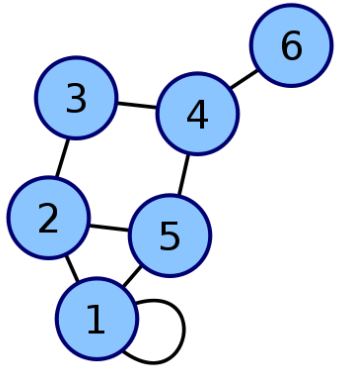
- **二叉树的相关问题**
  - 已知前序和中序遍历，返回后序遍历
  - **C++ 描述**

```
void BuildTree(int* Preorder, int* Inorder, int Length) {
  int Index = 0;
  if (Length > 0) {
    Stack[Top ++] = Preorder[0];
    for (Index = 0; Inorder[Index] != Preorder[0]; ++ Index);
    BuildTree(Preorder + Index + 1, Inorder + Index + 1, Length - Index - 1);
    BuildTree(Preorder + 1, Inorder, Index);
  }
  return;
}
```

数据结构

[Graph]图

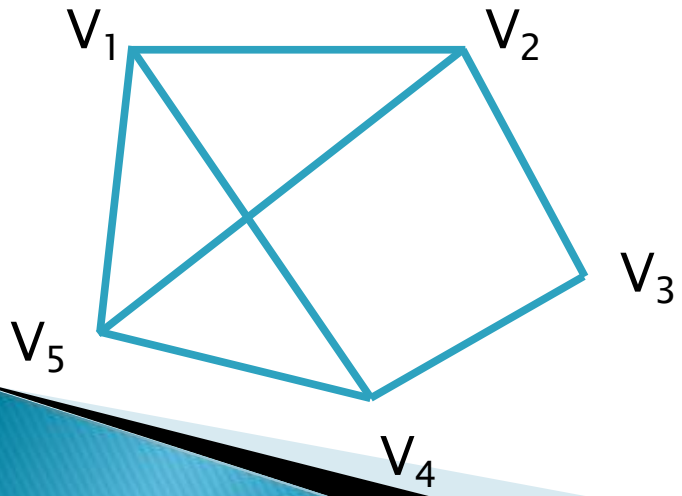
图(Graph)



- **什么是图?**
  - 在数学上，一个图(Graph)是表示物件与物件之间的关系的方法，是图论的基本研究对象。一个图看起来是由一些小圆点(称为顶点或结点)和连结这些圆点的直线或曲线(称为边)组成的。
  - 图有各种变体，包括简单图/多重图;有向图/无向图等。
  - 图的二元组定义：
    - 图G是一个二元组(V, E)，其中V称为顶点集，E称为边集。它们亦可写成V(G)和E(G)。 的元素是一个二元组数对，用(x, y)表示，其中x, y $\in$ V。

图(Graph)

- **图的分类**
  - **有向图/无向图**
    - 如果给图的每条边规定一个方向，那么得到的图称为有向图；相反，边没有方向的图称为无向图。
    - 无向图的定义：
      - 在图G=(V，E)中，如果对于任意的a，b ∈ V，当(a，b) ∈ E时，必有(b，a)∈E(即关系R对称)，对称此图为无向图。

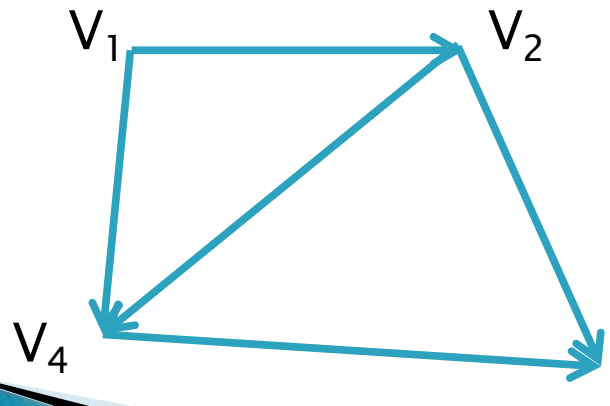


$V = \{V_1, V_2, V_3, V_4, V_5\}$

E = {(V1,V2), (V2,V3), (V3,V4), (V4,V5),
(V5,V1), (V2,V5), (V4,V1)}

图(Graph)

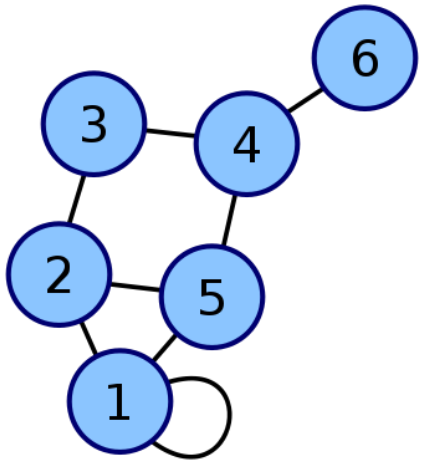
- **图的分类**
  - **有向图/无向图**
    - 如果给图的每条边规定一个方向，那么得到的图称为有向图；相反，边没有方向的图称为无向图。
    - 有向图的定义：
      - 如果对于任意的a，b ∈ V，当(a，b)∈E时 ，(b，a)∈E未必成立，则称此图为有向图。



$V = \{V_1, V_2, V_3, V_4\}$

$E = \{<V_1,V_2>, <V_2,V_3>, <V_1,V_4>, <V_4,V_3>, <V_2,V_4>\}$

图(Graph)

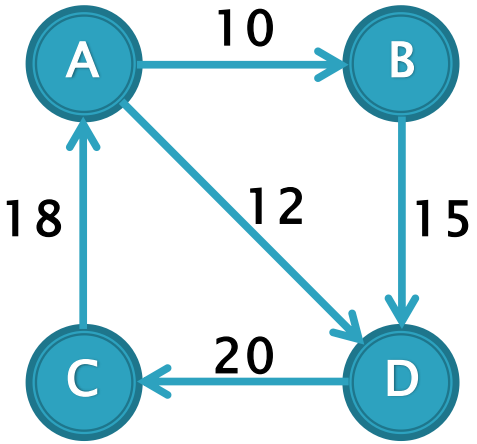


- **图的分类**
  - **简单图/多重图**
    - 一个图如果满足：
      - ① 没有两条边，它们所关联的两个点都相同(在有向图中，没有两条边的起点终点都分别相同)；
      - ② 每条边所关联的是两个不同的顶点；

      则称为简单图(Simple Graph)，简单的有向图和无向图都可以使用 “二元组的定义”，但形如(x, x)的序对不能属于E。而无向图的边集必须是对称的，即如果(x, y) ∈ E，则 (y, x) ∈ E
    - 若允许两结点间的边数多于一条，又允许顶点通过同一条边和自己关联，则为多重图。

图(Graph)

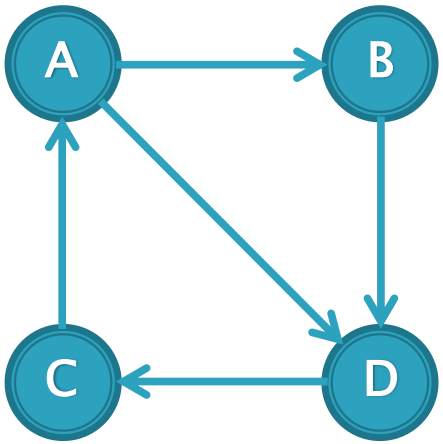
- **图的基本概念：**
  - 网络(Network)：若图G中每条边对应一个数值表示关系的程度, 则称图G为网络.
  - 子图(Sub-Graph)：图G’称作图G的子图如果V(G’) ⊆ V(G)以及E(G’) ⊆ E(G)



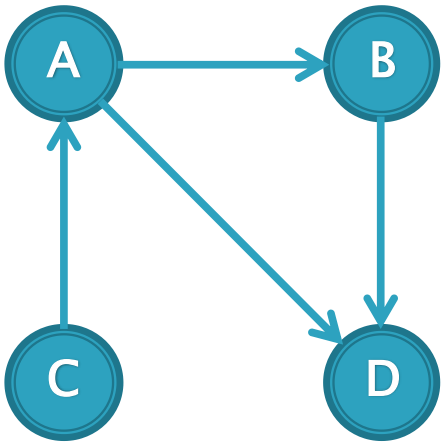
图G

图(Graph)

- **图的基本概念：**
  - 网络(Network)：若图G中每条边对应一个数值表示关系的程度, 则称图G为网络.
  - 子图(Sub-Graph)：图G’称作图G的子图如果V(G’) ⊆ V(G) 以及E(G’) ⊆ E(G)



图G



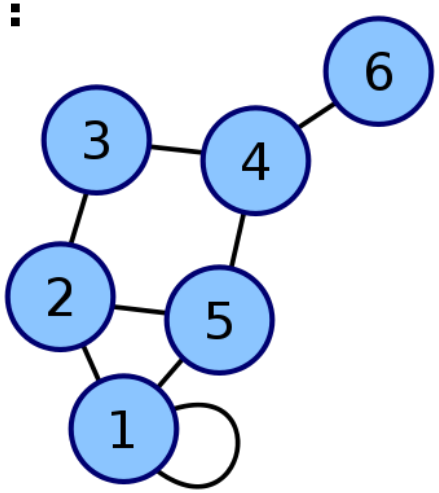
图G’

图(Graph)

- **图的基本概念：**
  - 度(Degree)：是一个顶点的度是指与该顶点相关联的总边数，顶点v的度记作d(v).度和边有如下关系：
    $\sum_{v \in V} d(v) = 2|E|$
  - 对有向图而言，顶点的度还可分为出度和入度：
    - **出度(Out-Degree)**：一个顶点的出度为d_0，是指有d_0条边以该顶点为起点，或说与该点关联的出边共有d_0条.
    - **入度(In-Degree)**：一个顶点的入度为d_0，是指有d_0条边以该顶点为终点，或说与该点关联的入边共有d_0条.

图(Graph)

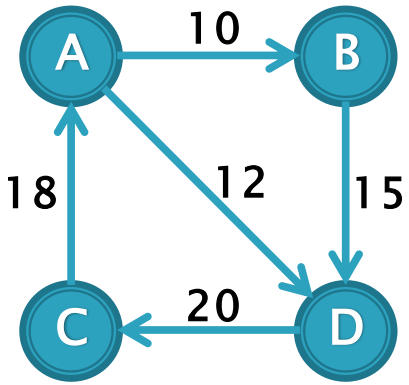
- **图的存储表示：**
  - **邻接矩阵**
    - **邻接矩阵的定义：**
      - 阶为n的图G的邻接矩阵A是 n × n的. 将G的顶点标签为v_1, v_2, ..., v_n。若$(v_i, v_j) \in E(G)$, $A_{ij} = 1$, 否则$A_{ij} = 0$.
      - 无向图的邻接矩阵是对称矩阵.
    - **例如：**



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

图(Graph)

- **图的存储表示：**
  - **距离矩阵**
    - **距离矩阵的定义：**
      - 一个距离矩阵是一个包含一组点两两之间距离的矩阵(即 二维数组). 因此给定N个欧几里得空间中的点，其距离矩阵就是一个非负实数作为元素的N×N的对称矩阵. 距离矩阵可以看成是邻接矩阵的加权形式.
    - **例如：**



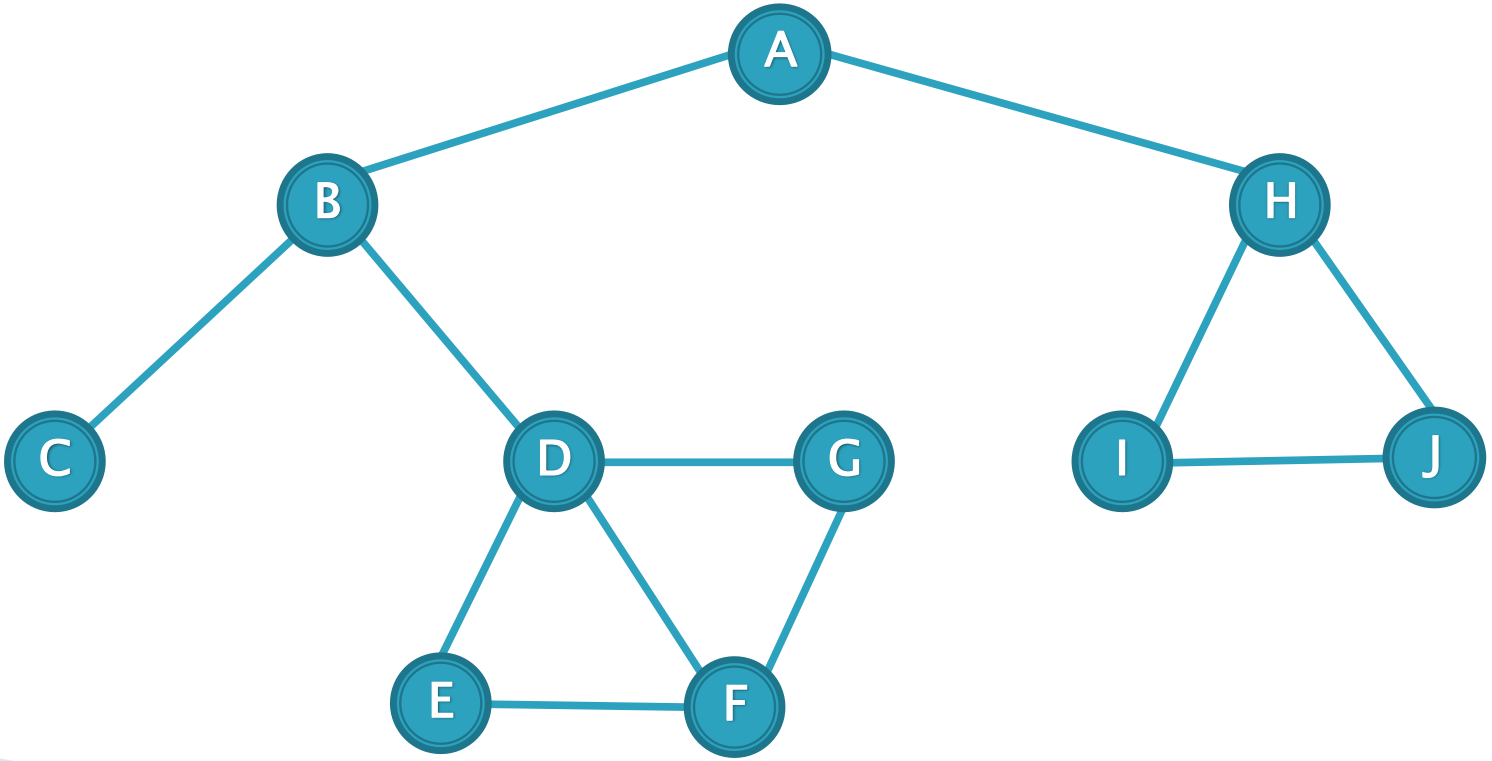
$$\begin{pmatrix} 0 & 10 & \infty & 12 \\ 10 & 0 & \infty & 15 \\ \infty & \infty & 0 & 20 \\ 12 & 15 & 20 & 0 \end{pmatrix}$$

图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)
    - **深度优先搜索**是树的 先序遍历的推广，它的基本思想是：从图G的某个顶点v_0出发，访问v_0，然后选择一个与v_0相邻且没被访问过的顶点v_i访问，再从vi出发选择一个与v_i相邻且未被访问的顶点v_j进行访问，依次继续. 如果当前被访问过的顶点的所有邻接顶点都已被访问，则退回到已被访问的顶点序列中最后一个拥有未被访问的相邻顶点的顶点w，从w出发按同样的方法向前遍历，直到图中所有顶点都被访问.

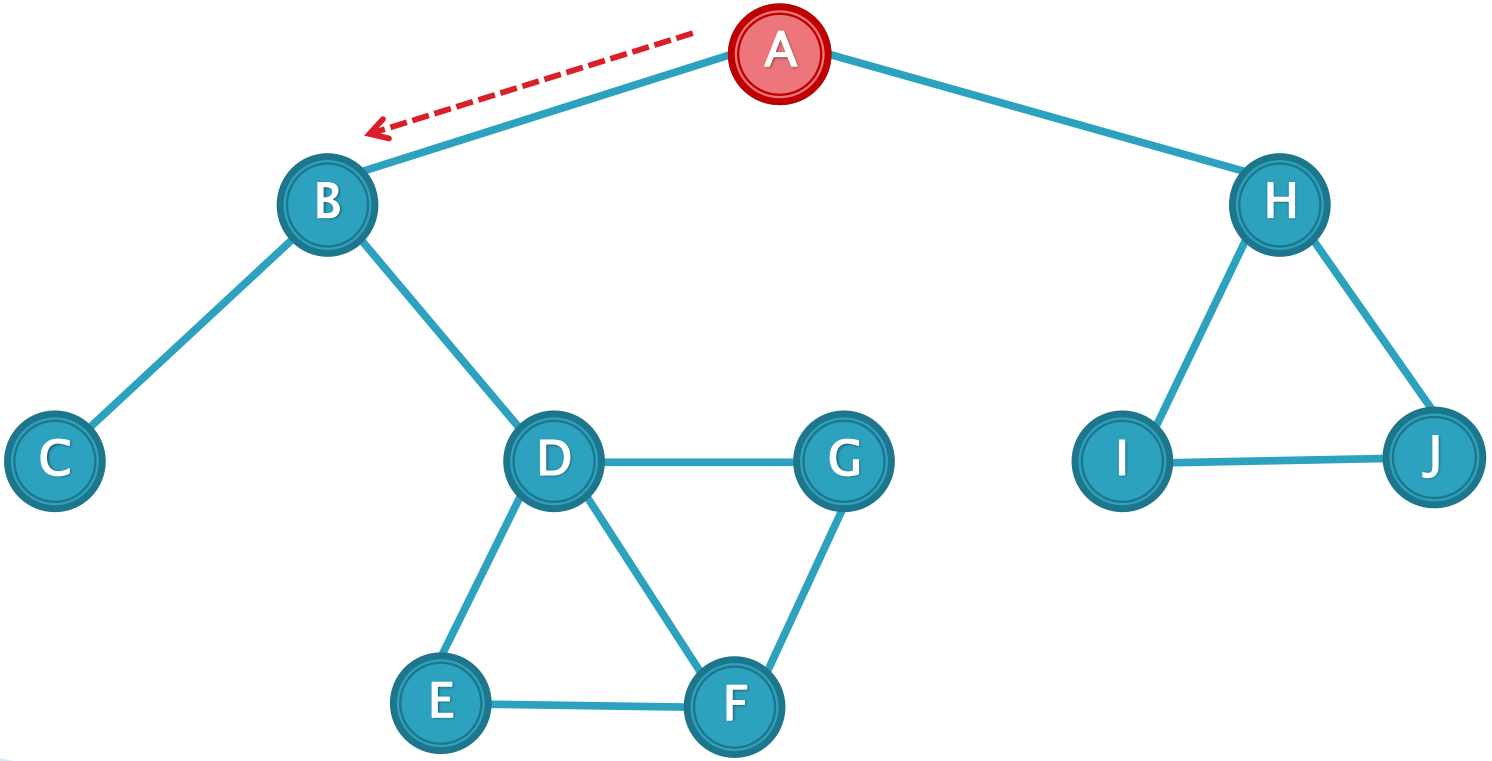
图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)



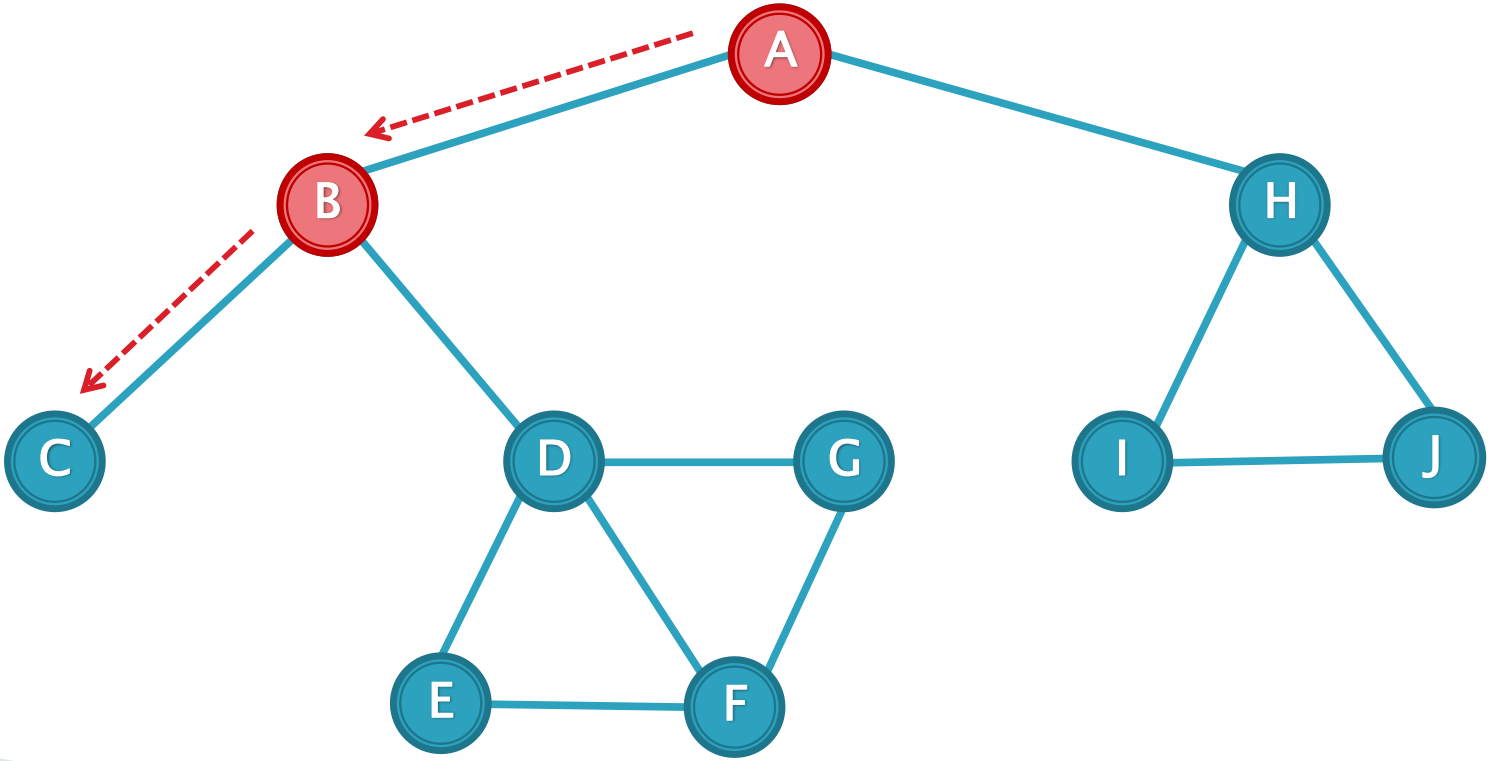
图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)



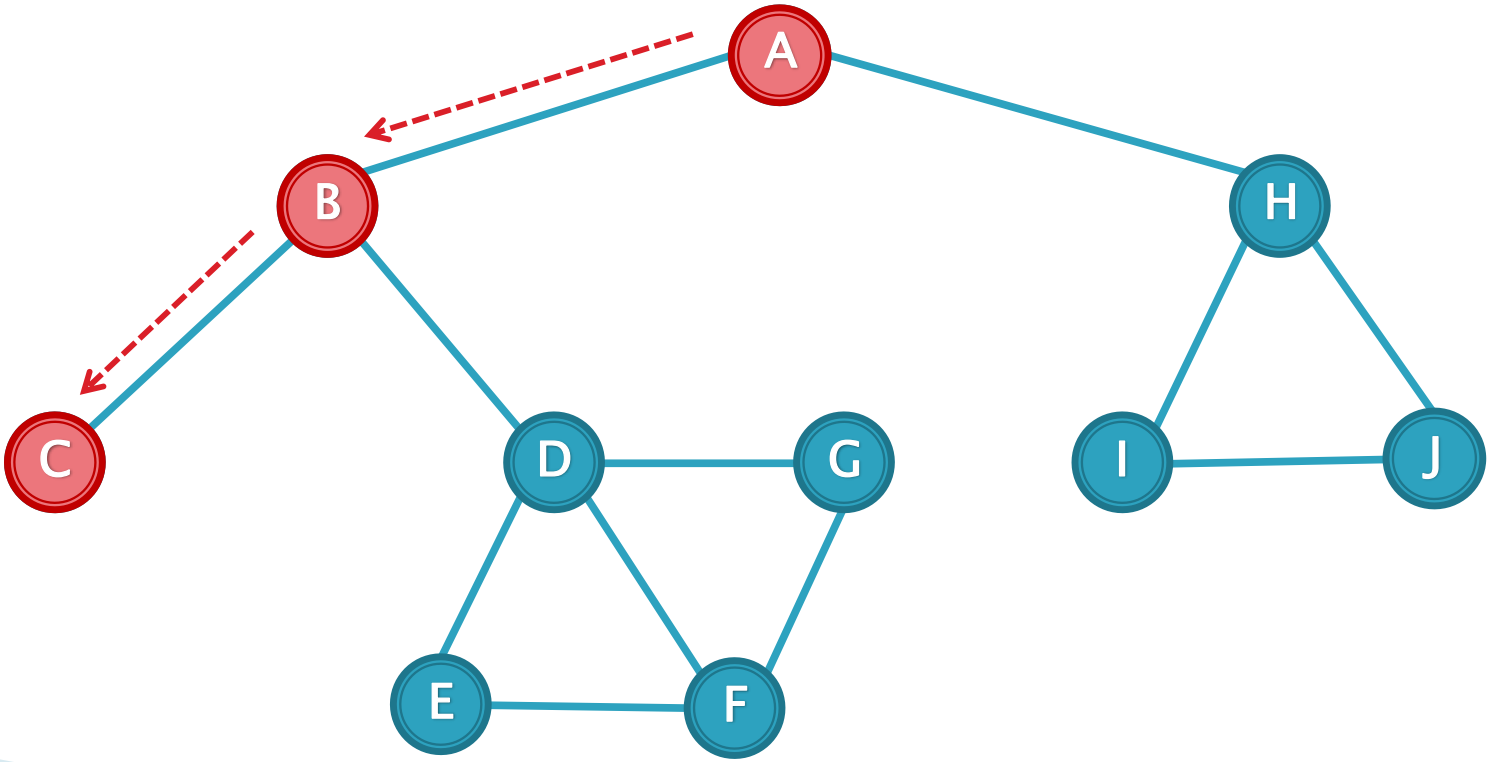
图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)



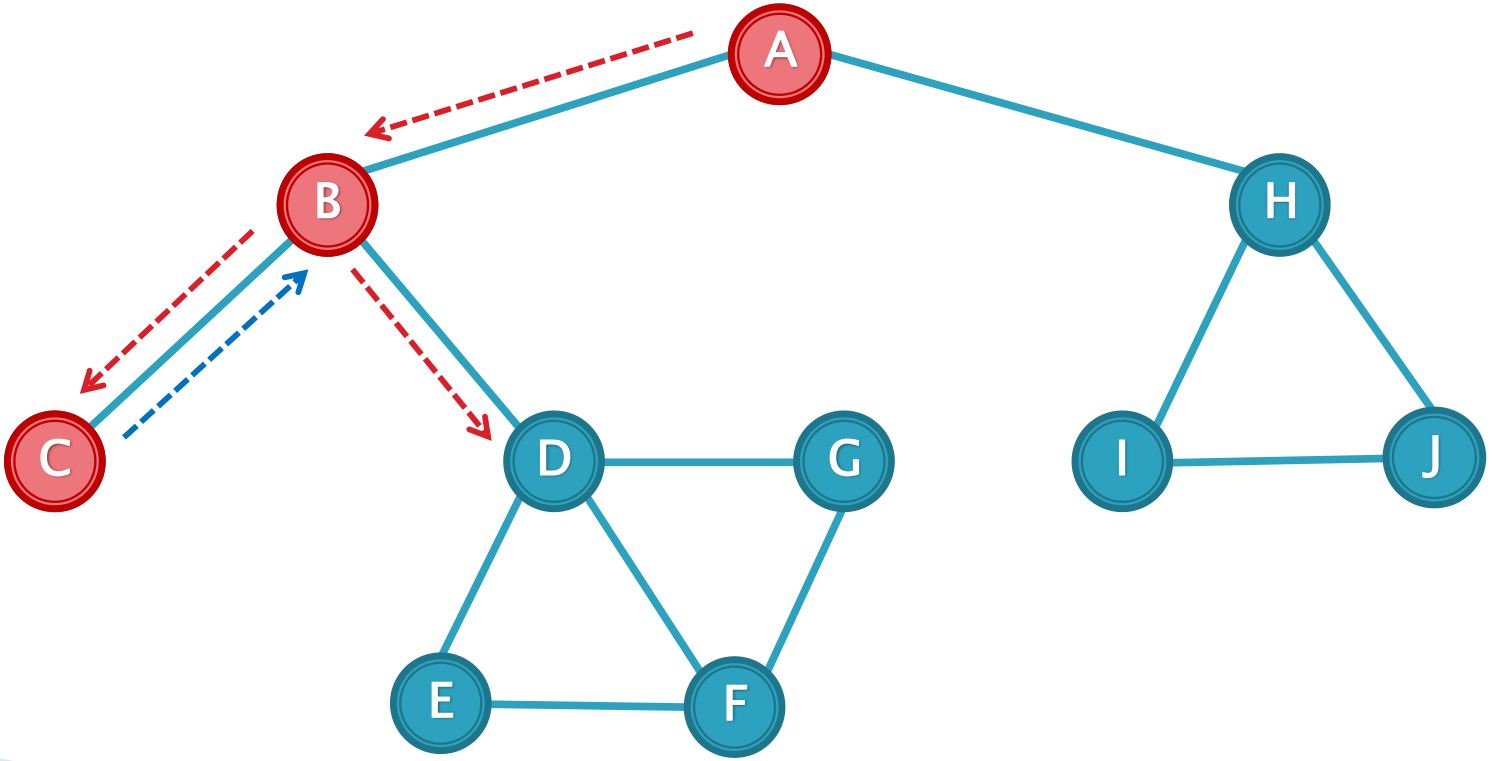
图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)



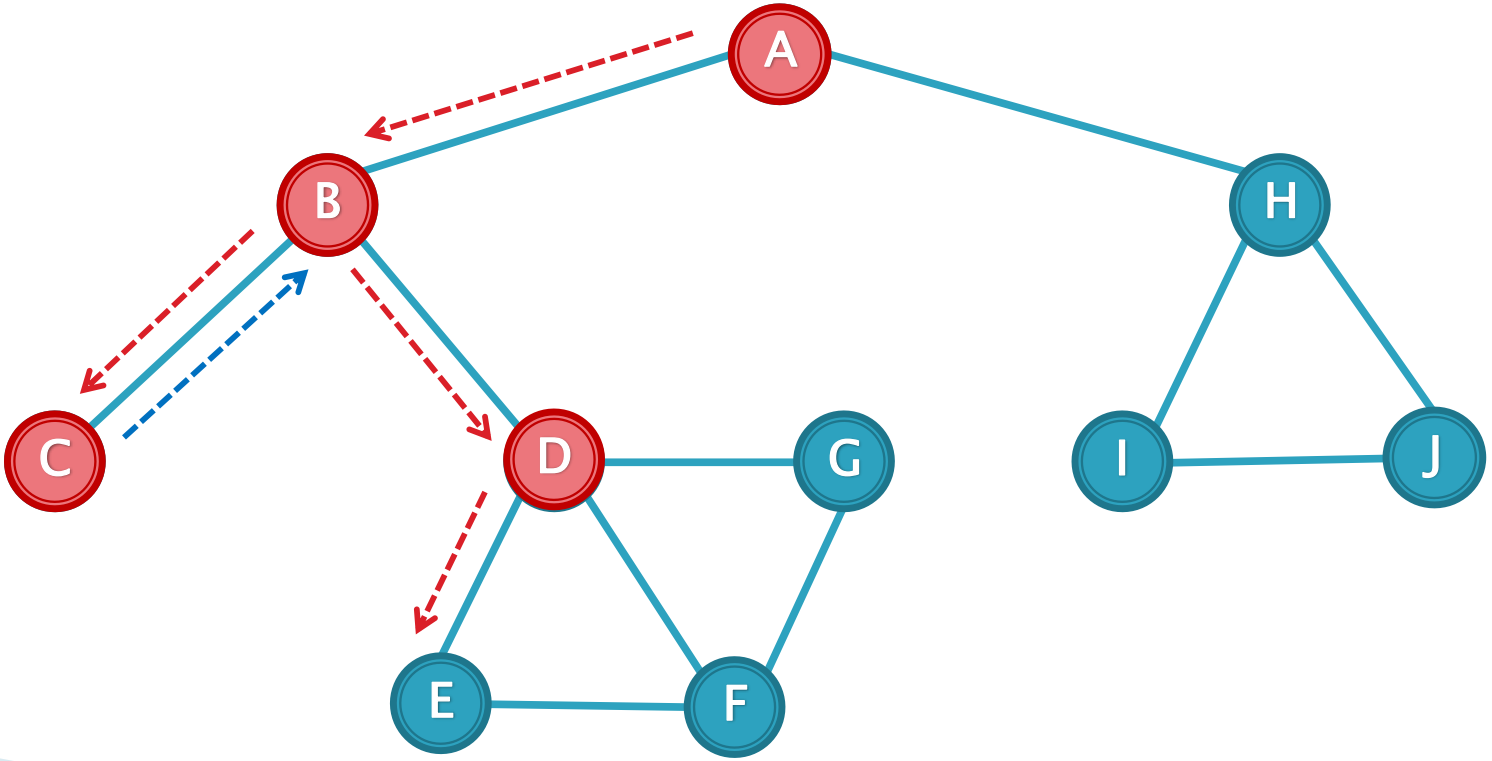
图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)



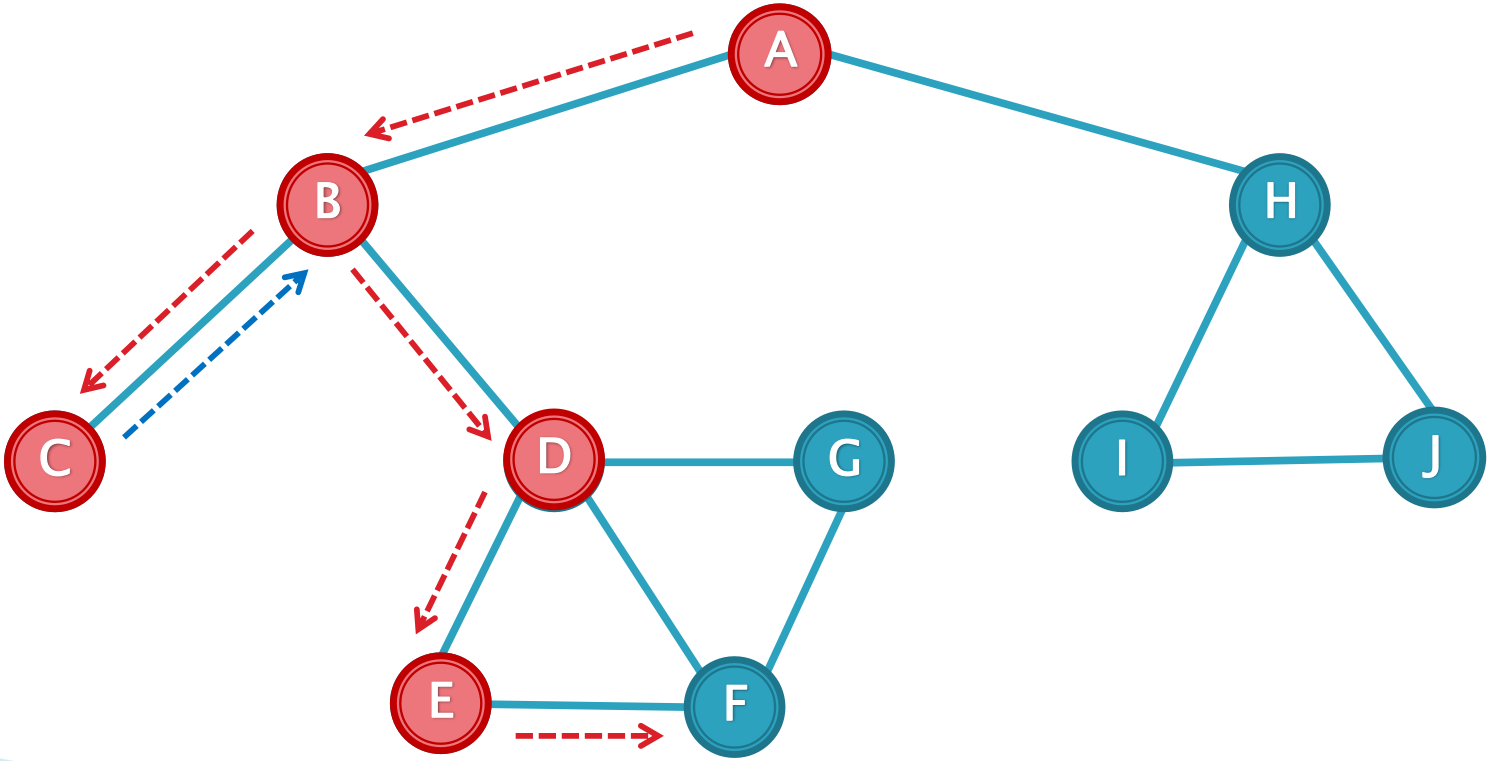
图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)



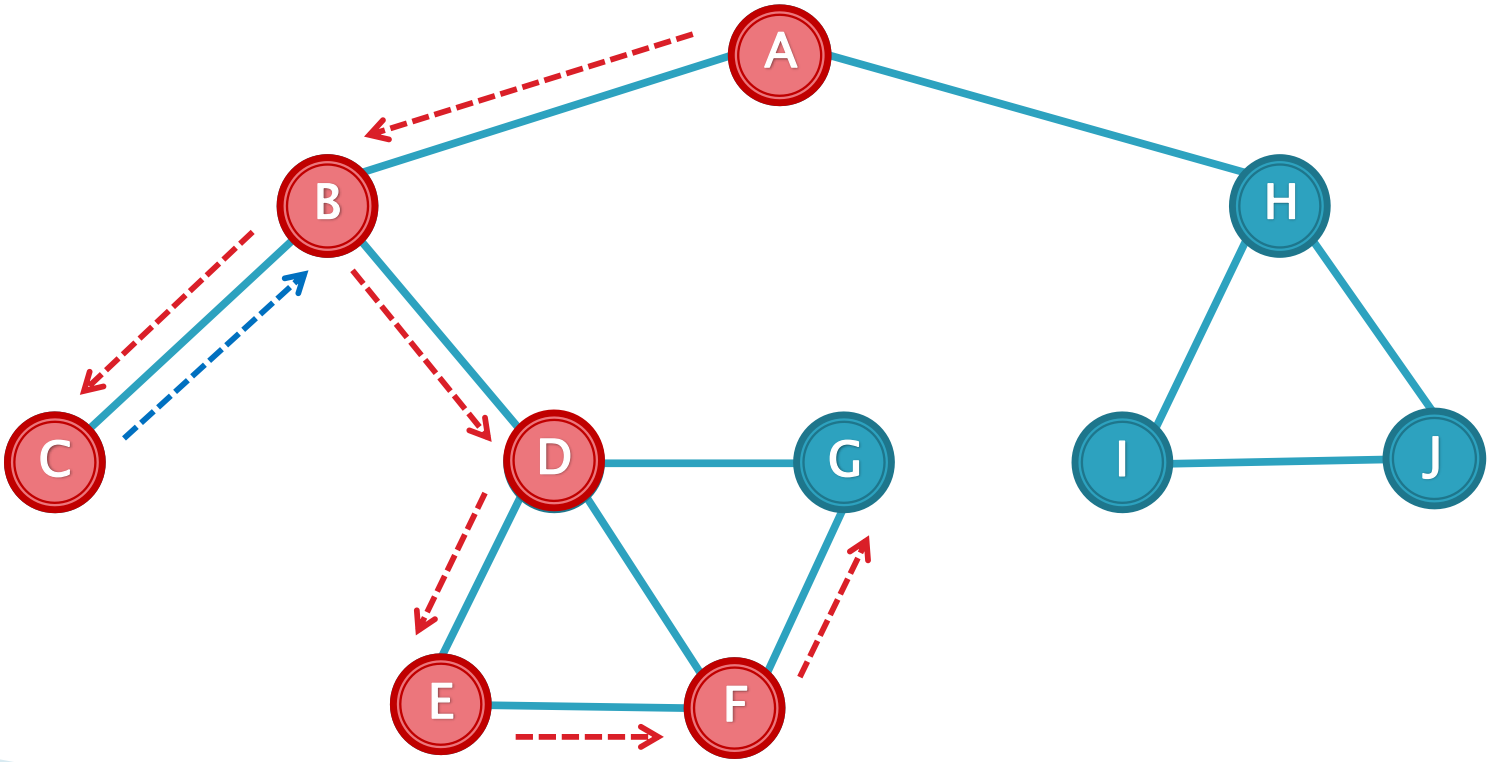
图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)



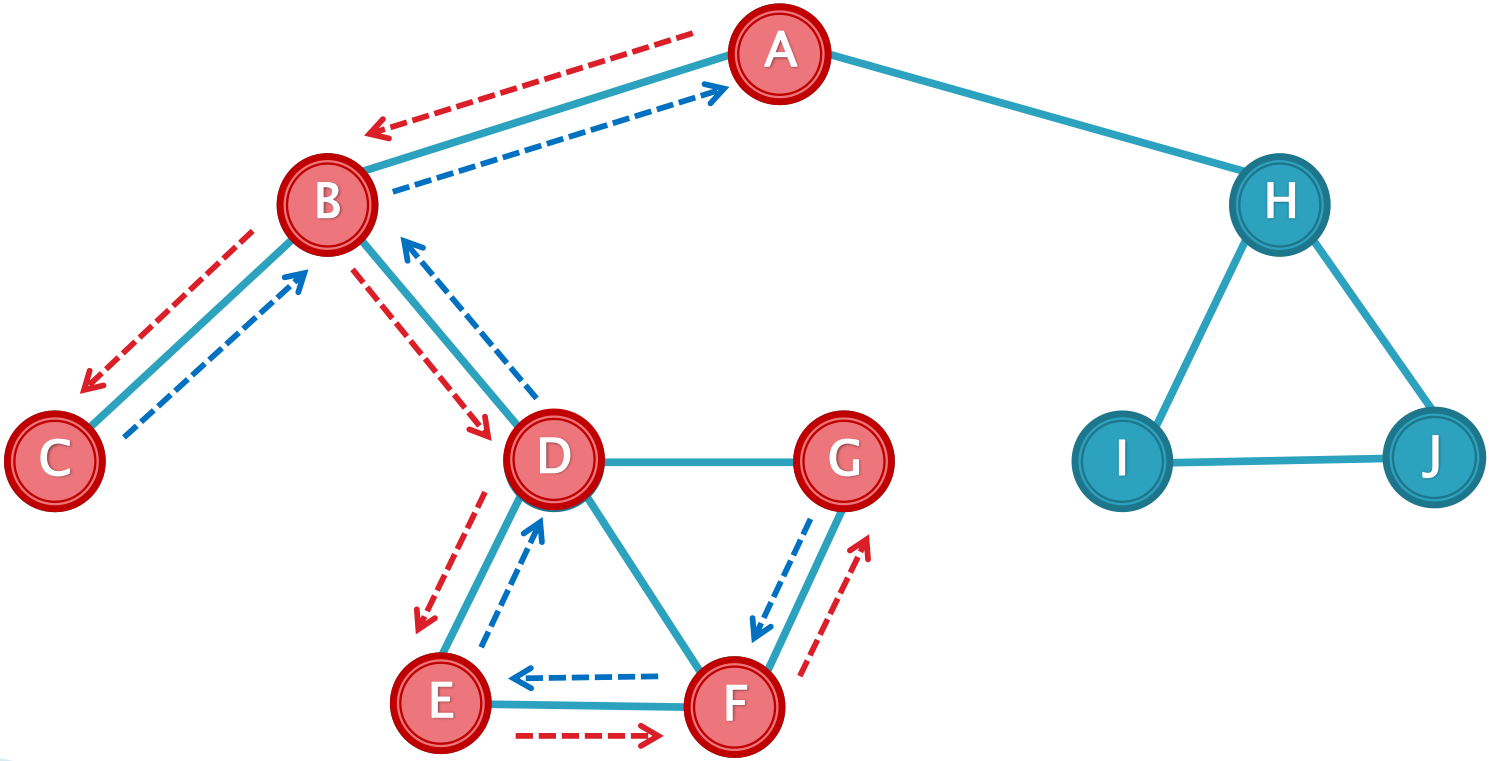
图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)



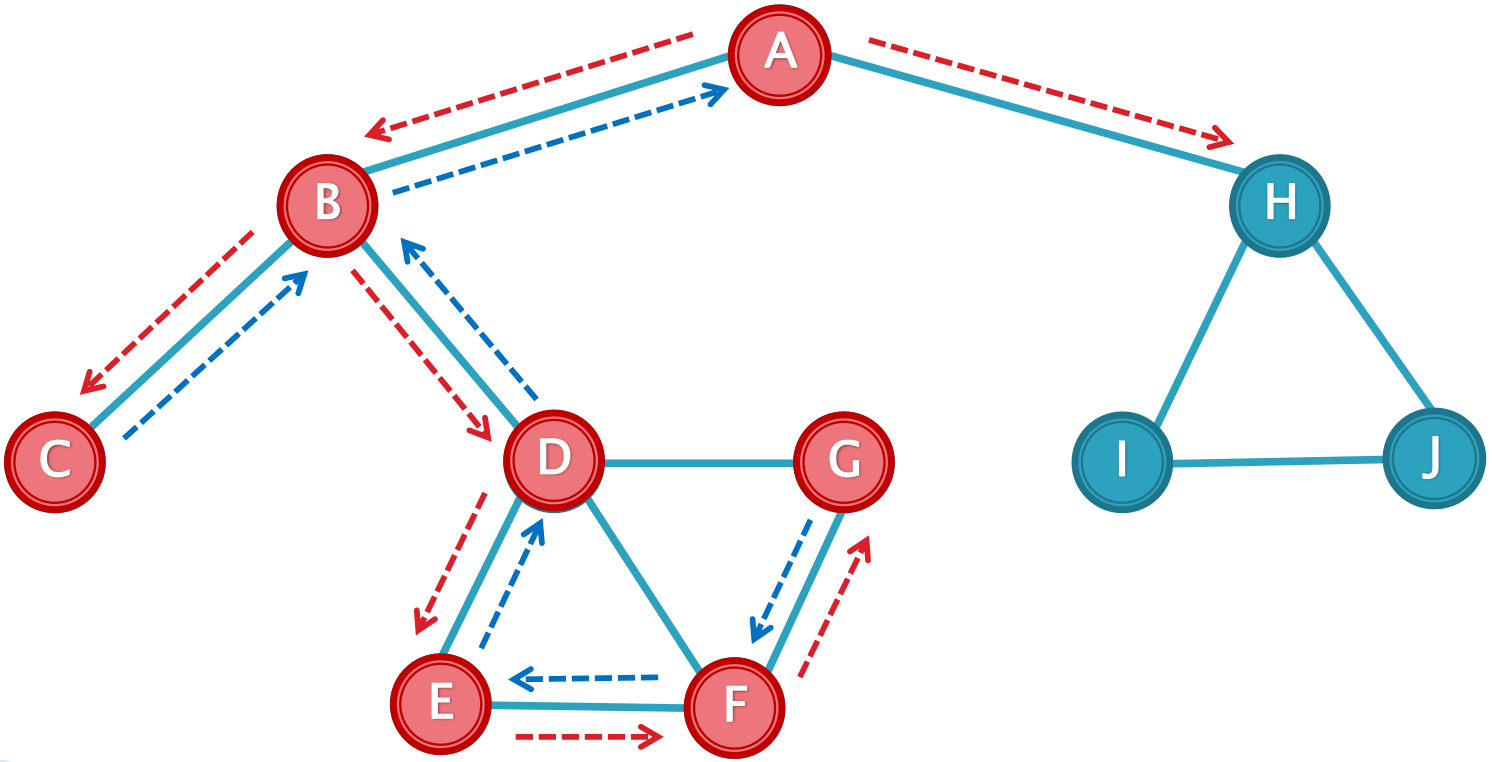
图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)



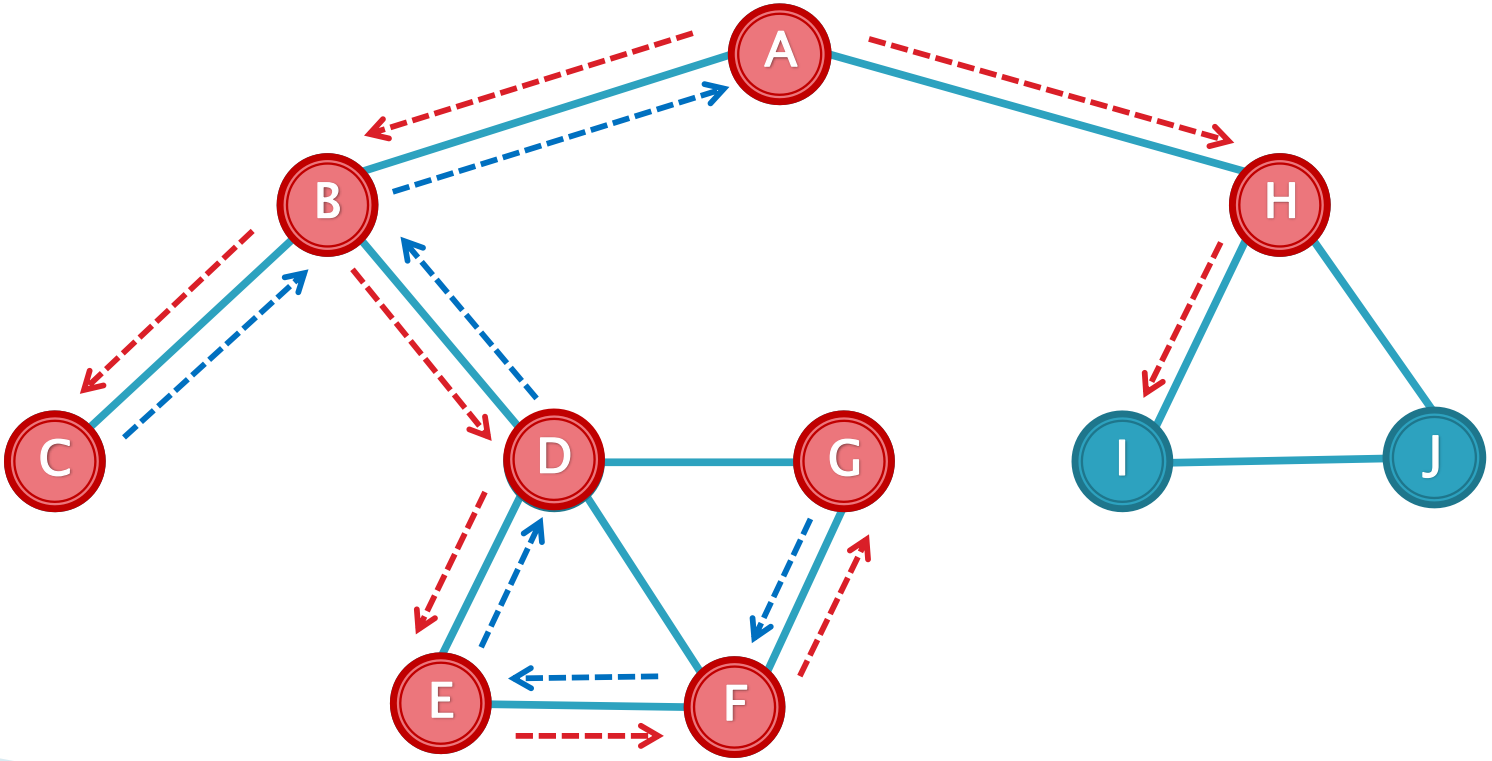
图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)



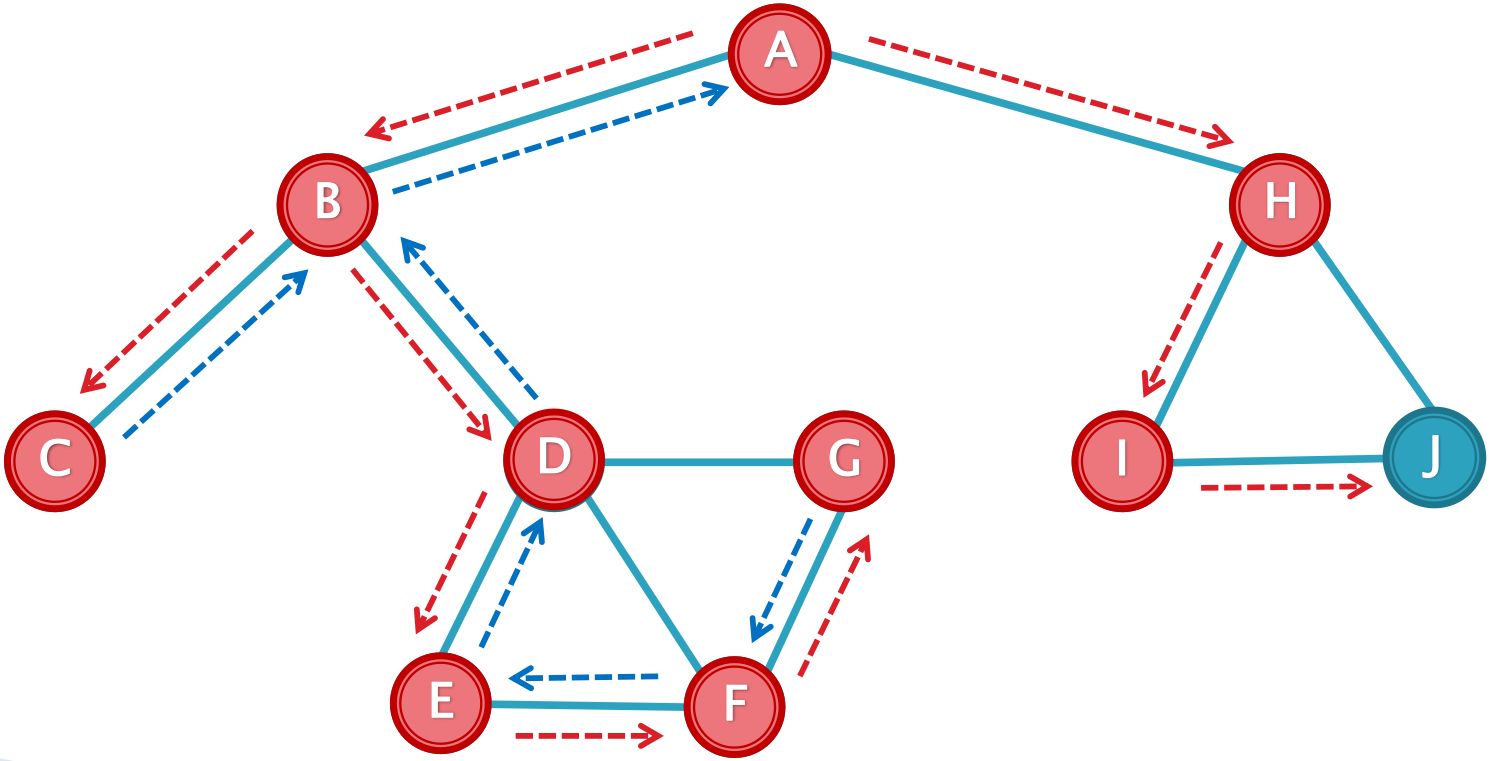
图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)



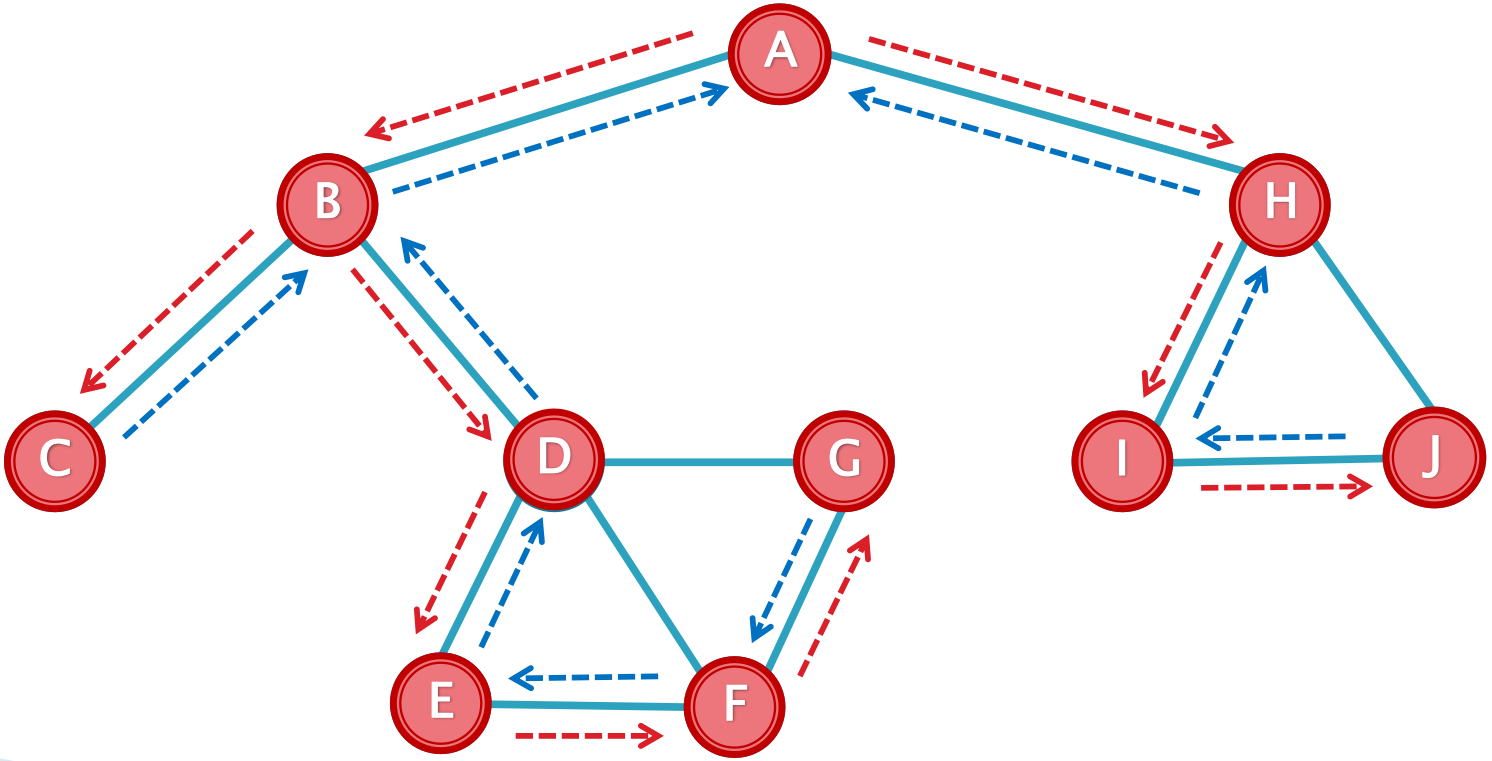
图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)



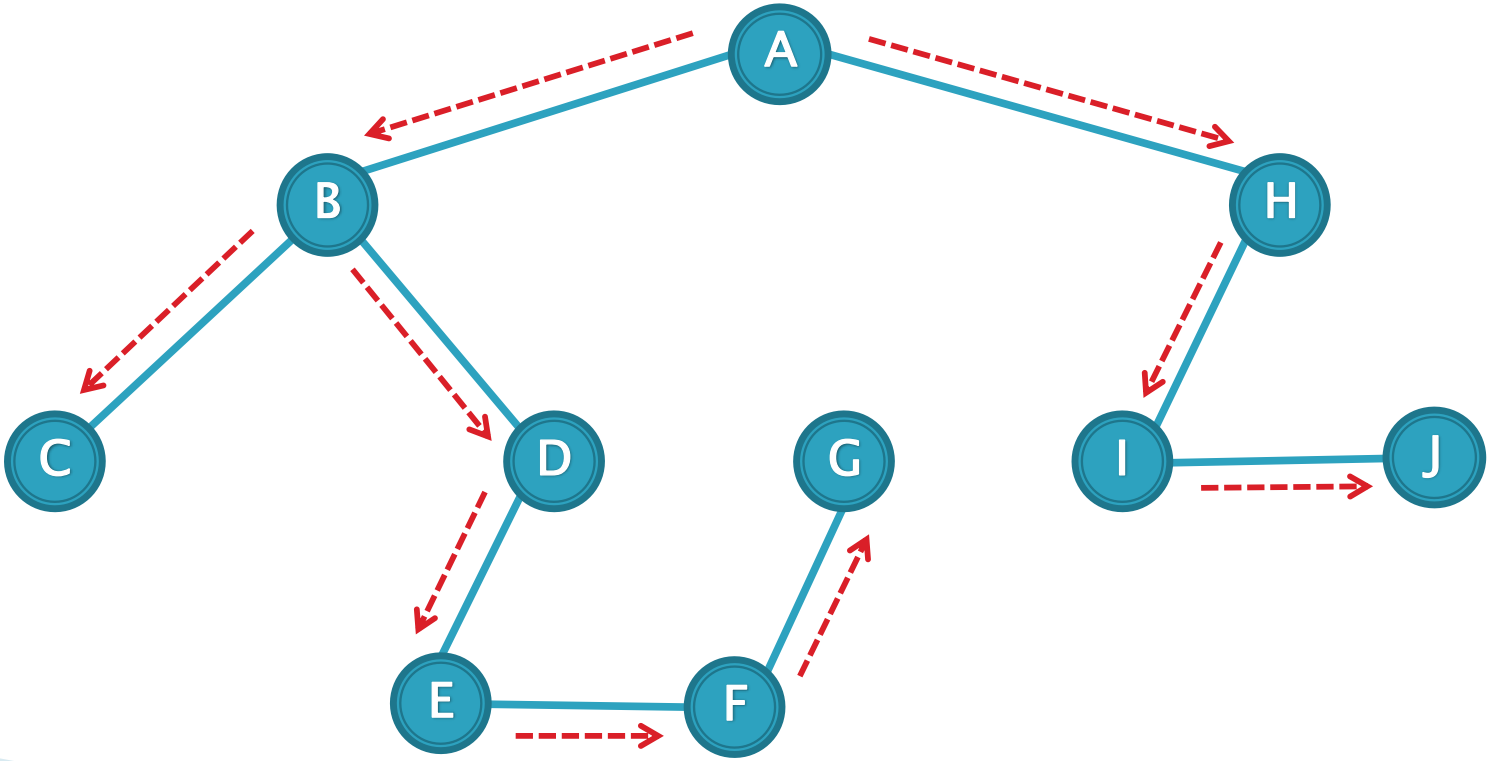
图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)



图(Graph)

- **图的遍历**
  - 深度优先搜索**生成树**



图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)
  - **Pascal 描述**

```
Procedure DeepFirstSearch();
var
  i : integer;
begin
  for i:= 1 to NUMBER_OF_VERTEX do
    if (isVisited[i] = false) then
      DFS(i);
end;
```

图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)
  - **Pascal 描述**

```
Procedure DFS(Point : integer);
var
  i : integer;
begin
  isVisited[Point]:= true;
  writeln('We are at ', Point);
  for i:= 1 to NUMBER_OF_VERTEX do
    if ((isVisited[i] = false) and (Matrix[Point][i] <> 0))
then
      DFS(i);
end;
```

图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)
  - **C++ 描述**

```
void Graph::DeepFirstSearch()
{
    bool* isVisited = new bool[Vertex]();
    for (int i = 0; i < Vertex; i ++)
        if (!isVisited[i])
            DeepFirstSearch(isVisited, i);
    return;
}
```

图(Graph)

- **图的遍历**
  - 深度优先搜索(Deep-First-Search)
  - **C++ 描述**

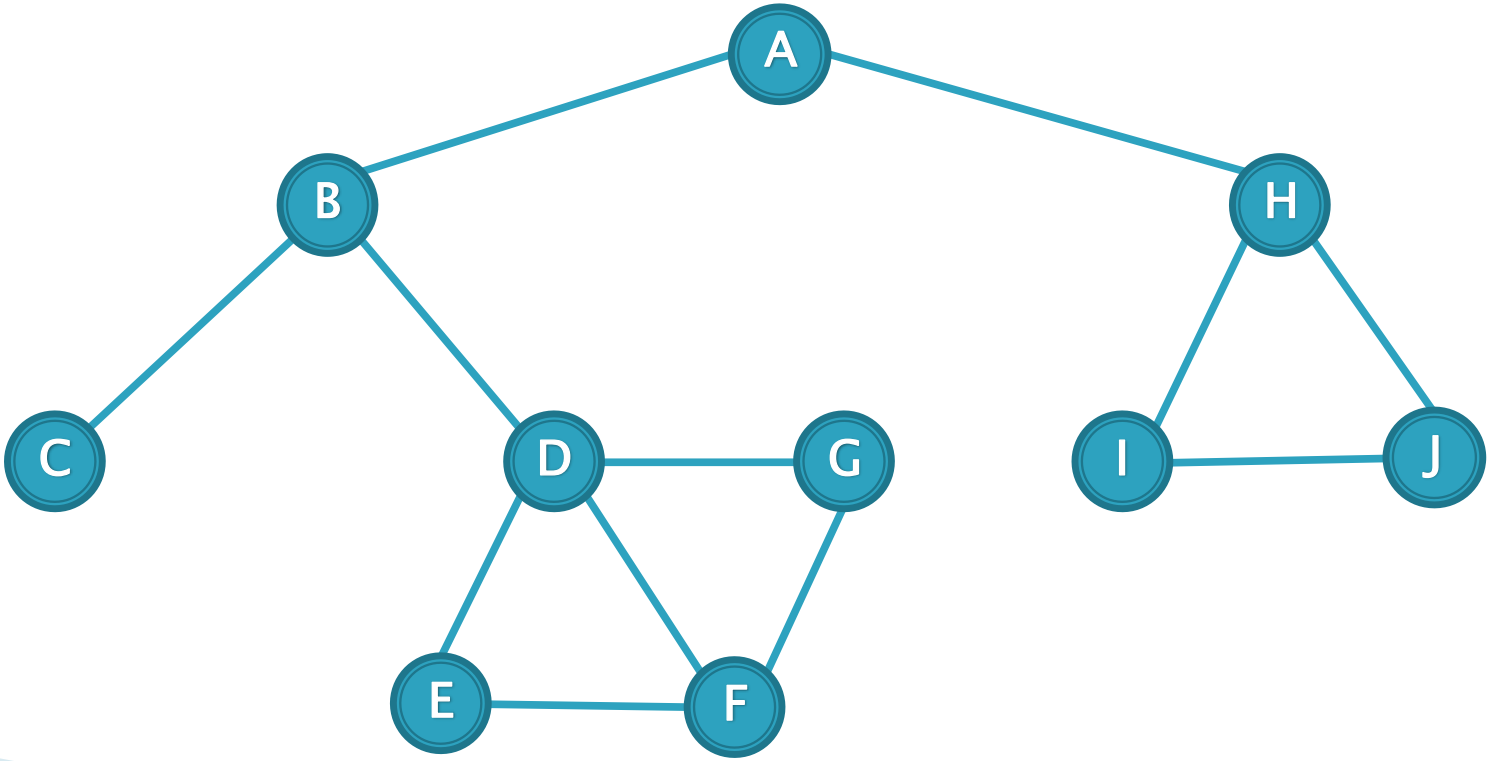
```
void Graph::DeepFirstSearch(bool* isVisited, int Point)
{
    isVisited[Point] = true;
    std::cout << "Now we are at " << Point << ".\n";
    for (int i = 0; i < Vertex; ++ i)
        if (!isVisited[i] && Matrix[Point][i])
            DeepFirstSearch(isVisited, i);
    return;
}
```

图(Graph)

- **图的遍历**
  - 广度优先搜索(Breadth-First-Search)
    - **广度优先搜索**是树的按层次遍历的推广，它的基本思想是：首先访问初始点vi，并将其标记为已访问过，接着访问vi的所有未被访问过的邻接点vi1,vi2,…, vi t，并均标记已访问过，然后再按照vi1,vi2,…, vi t的次序，访问每一个顶点的所有未被访问过的邻接点，并均标记为已访问过，依次类推，直到图中所有和初始点vi有路径相通的顶点都被访问过为止。

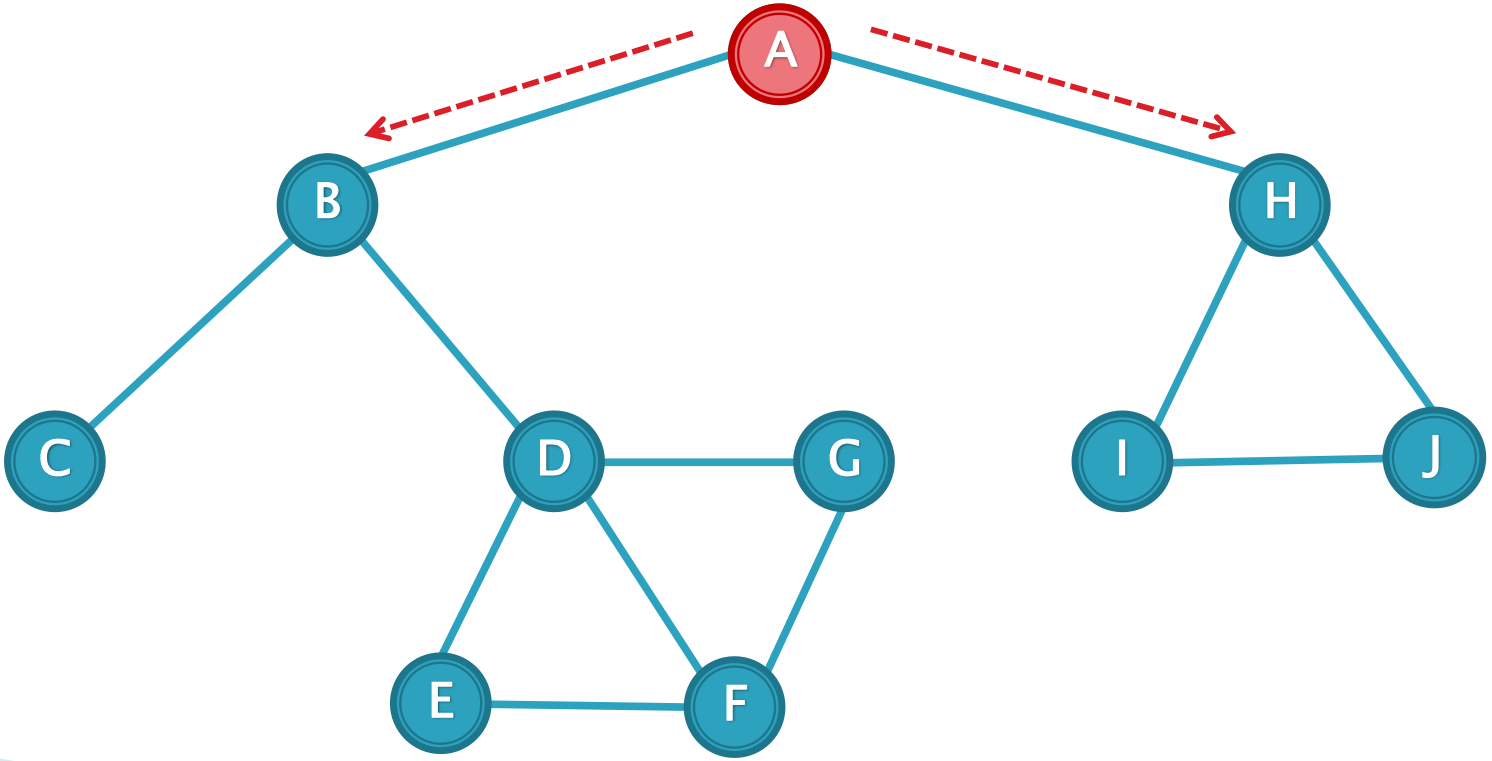
图(Graph)

- **图的遍历**
  - 广度优先搜索(Breadth-First-Search)



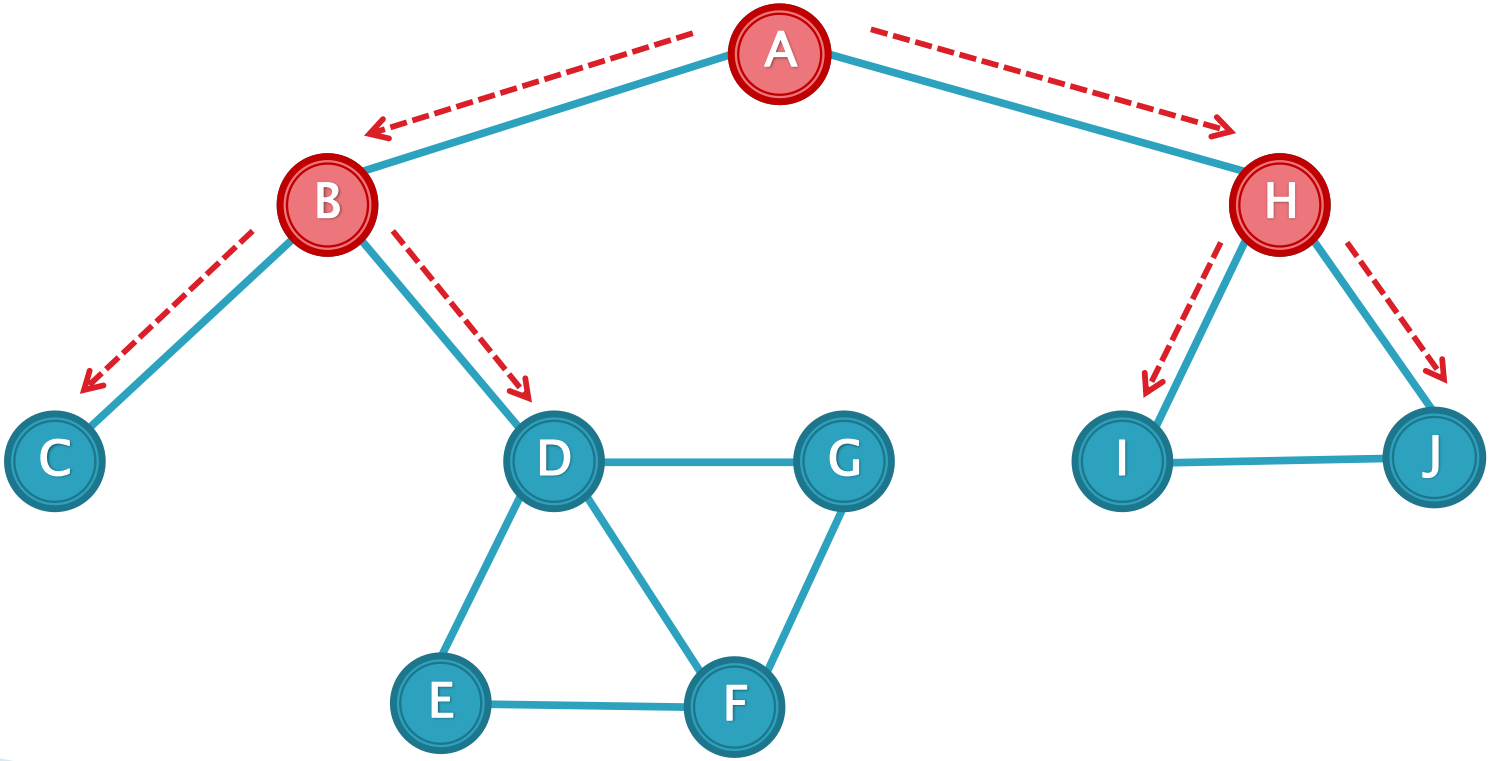
图(Graph)

- **图的遍历**
  - 广度优先搜索(Breadth-First-Search)



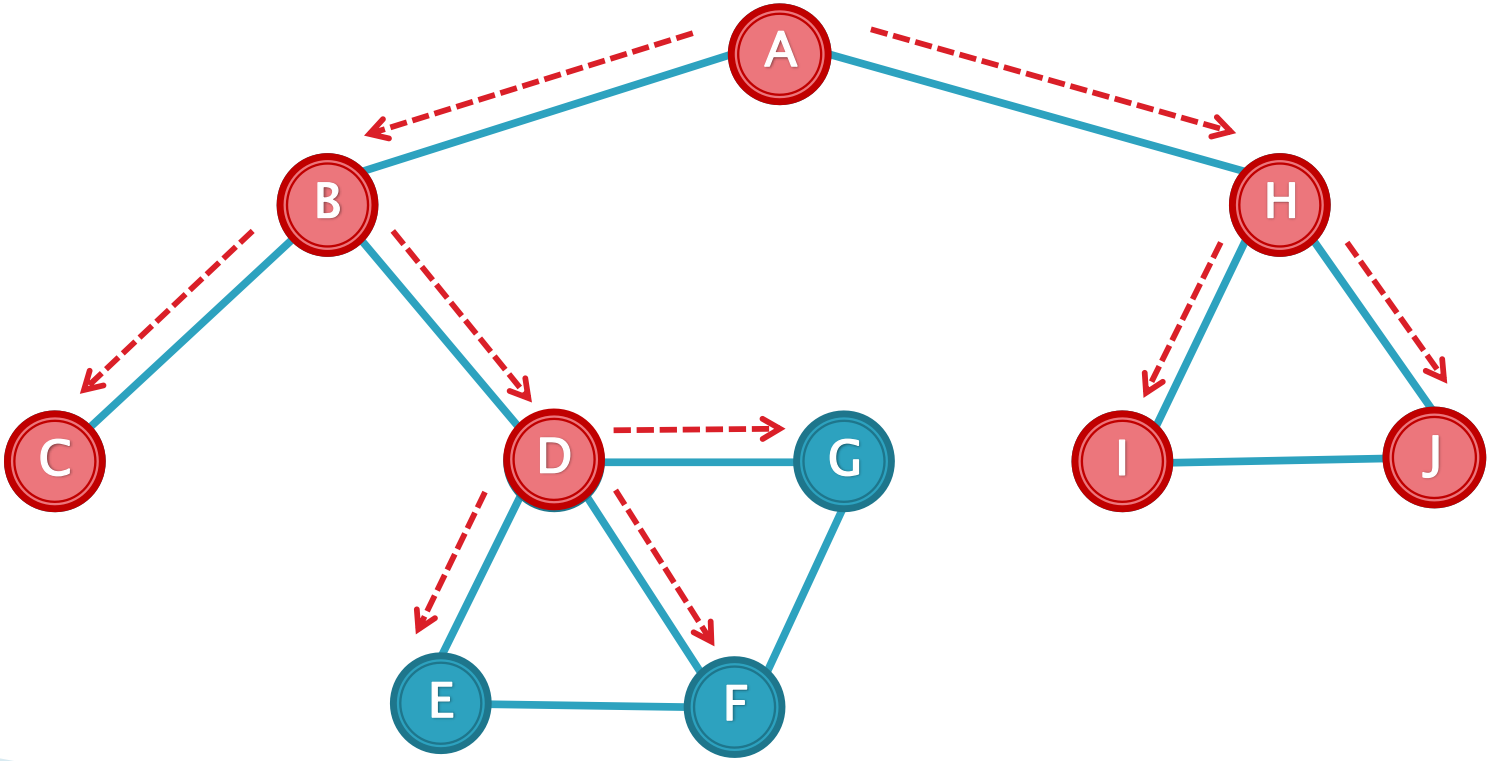
图(Graph)

- **图的遍历**
  - 广度优先搜索(Breadth-First-Search)



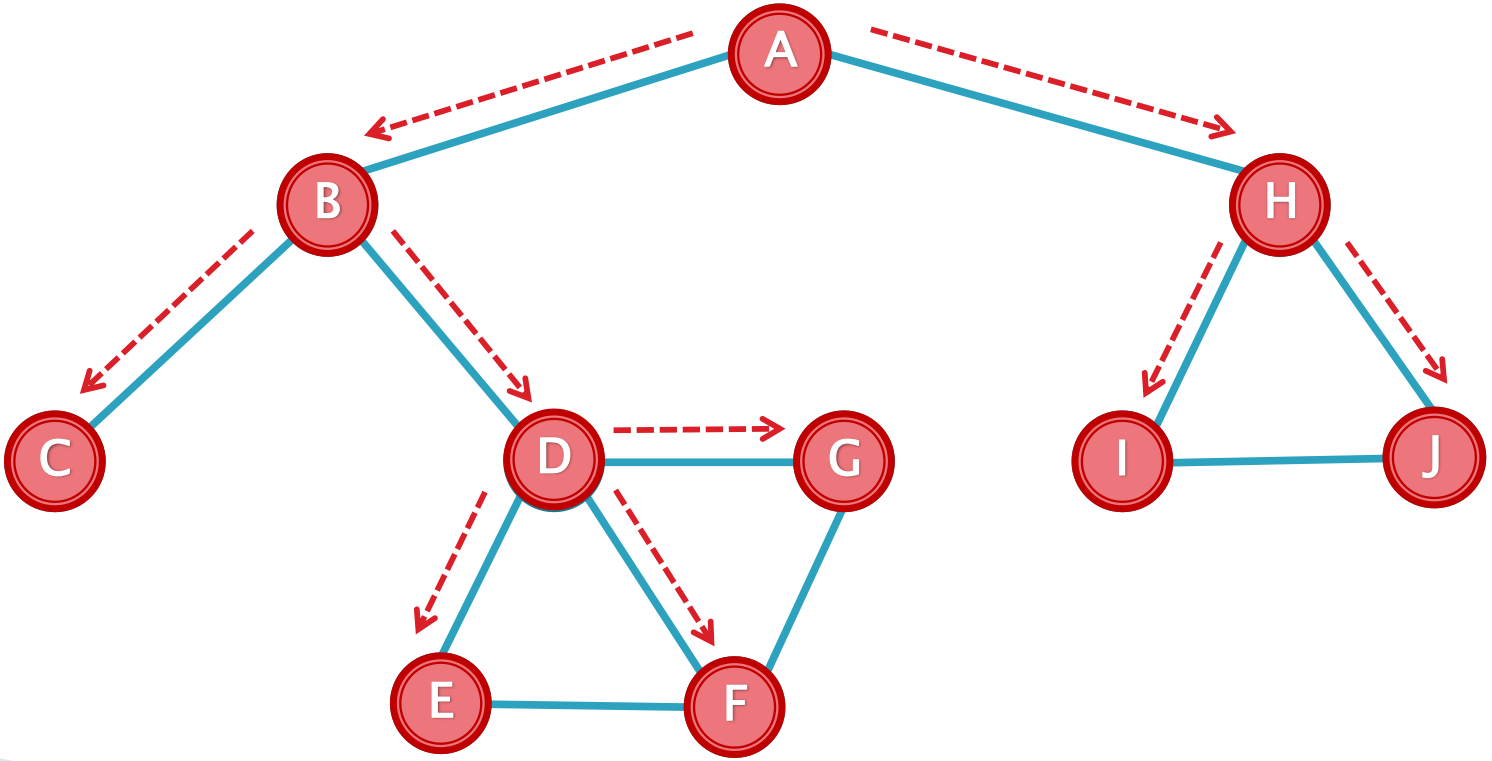
图(Graph)

- **图的遍历**
  - 广度优先搜索(Breadth-First-Search)



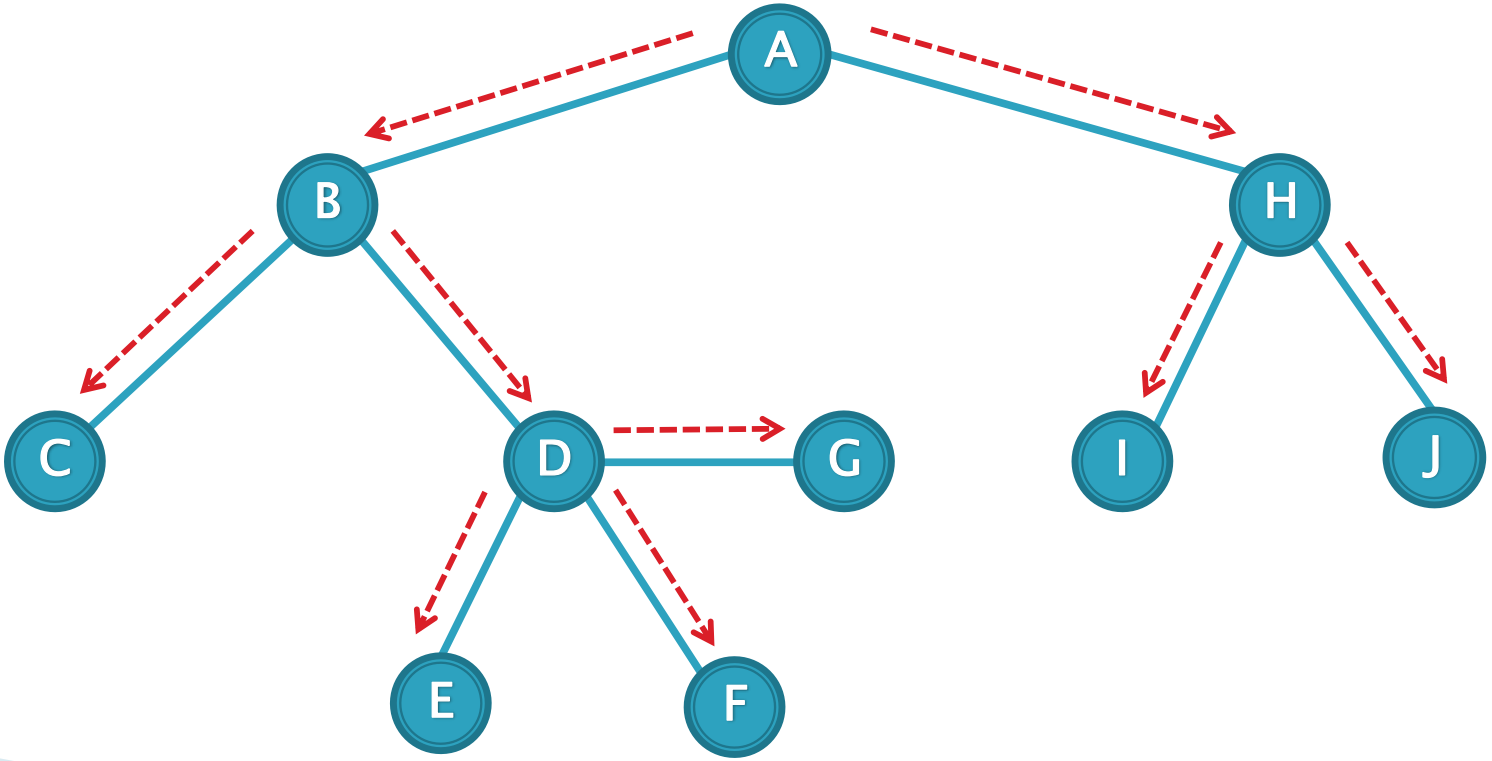
图(Graph)

- **图的遍历**
  - 广度优先搜索(Breadth-First-Search)



图(Graph)

- **图的遍历**
  - 广度优先搜索**生成树**



图(Graph)

- **图的遍历**
  - 广度优先搜索(Breadth-First-Search)
  - **Pascal 描述**

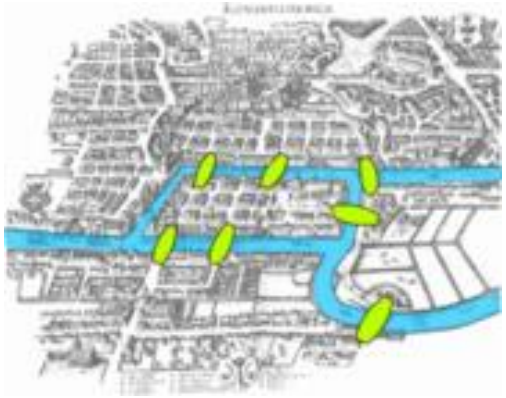
```
while (isEmpty() = false) do
  begin
    CurrentPoint:= Serve();
    for i:= 1 to NUMBER_OF_VERTEX do
      if ((isVisited[i] = false) and (Matrix[CurrentPoint][i] <> 0)) then
      begin
        isVisited[i]:= true;
        writeln('Now we are at ', i);
        Append(i);
      end;
  end;
```

图(Graph)

- **图的遍历**
  - 广度优先搜索(Breadth-First-Search)
  - **C++ 描述**

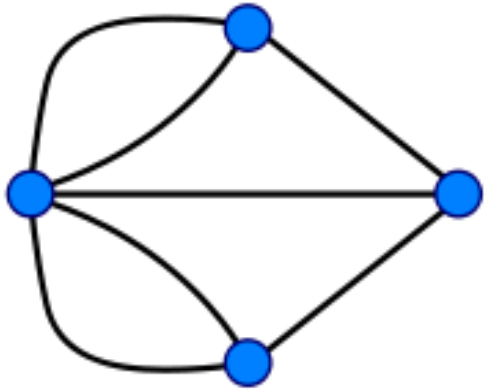
```
while (!Queue.empty()) {
        int CurrentPoint = Queue.front();
        Queue.pop();
        for (int i = 0; i < Vertex; ++ i)
            if (!isVisited[i] && Matrix[CurrentPoint][i])
            {
                isVisited[i] = true;
                std::cout << "Now we are at " << i << ".\n";
                Queue.push(i);
            }
    }
```

图(Graph)



- **图的相关问题**
  - 柯尼斯堡七桥问题
    - 柯尼斯堡七桥问题是图论中的著名问题. 这个问题是基于一个现实生活中的事例：当时东普鲁士柯尼斯堡市区跨普列戈利亚河两岸, 河中心有两个小岛. 小岛与河的两岸有七条桥连接. 在所有桥都只能走一遍的前提下, 如何才能把这个地方所有的桥都走遍？

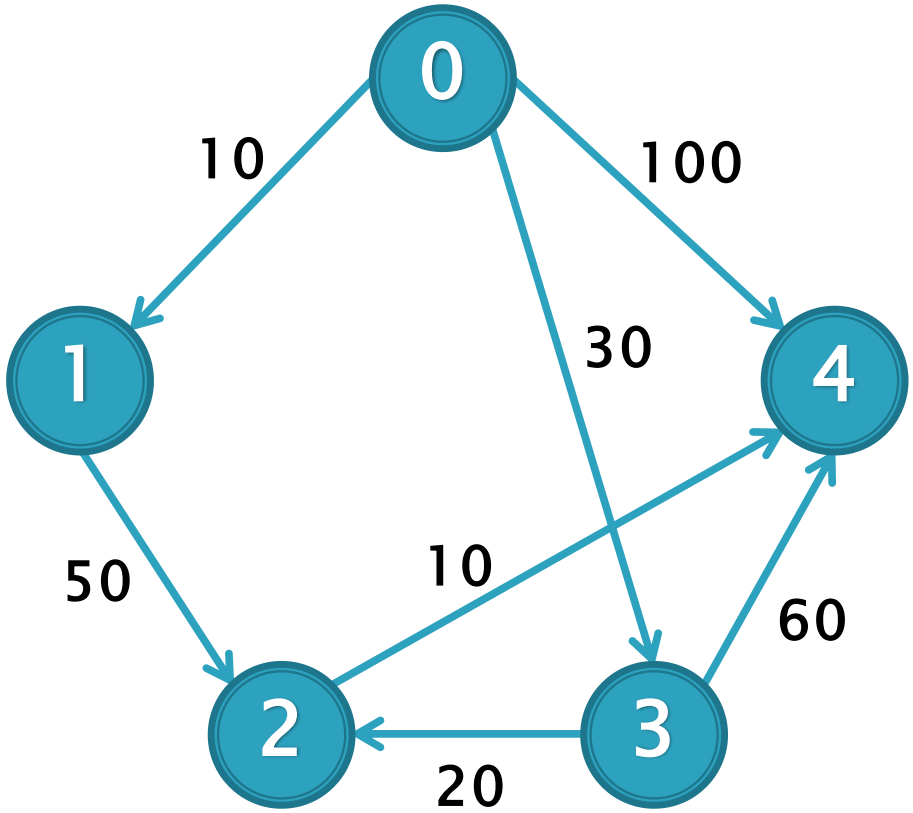
图(Graph)



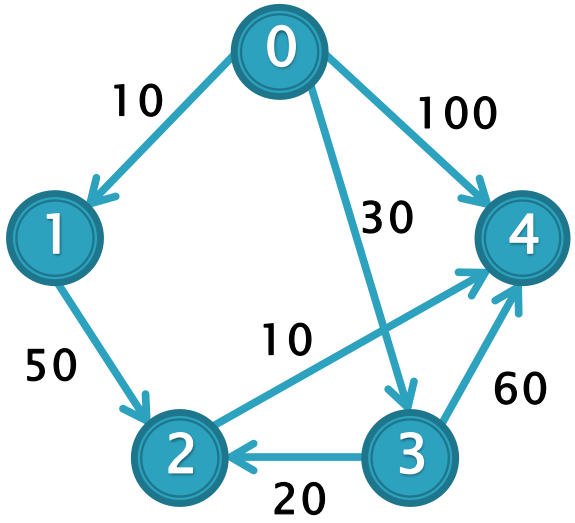
- **图的相关问题**
  - 柯尼斯堡七桥问题
    - 莱昂哈德·欧拉（Leonhard Euler）在1735年圆满地解决了这一问题，证明这种方法并不存在，也顺带解决了一笔画问题. 他在圣彼得堡科学院发表了图论史上第一篇重要文献.
    - 欧拉最后给出任意一种河——桥图能否全部走一次的判定法则. **如果通奇数座桥的地方不止两个，那么满足要求的路线便不存在了**. 如果只有两个地方通奇数座桥，则可从其中任何一地出发找到所要求的路线. 若没有一个地方通奇数座桥，则从任何一地出发，所求的路线都能实现，他还说明了怎样快速找到所要求的路线.

图(Graph)

- **图的相关问题**
  - 最短路径问题



图(Graph)

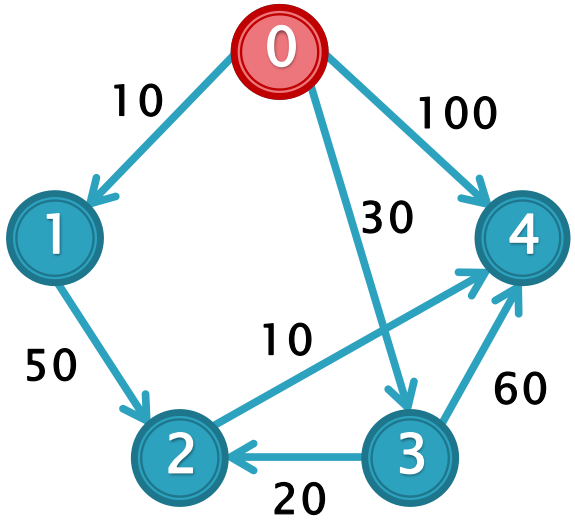


- **图的相关问题**
  - 最短路径问题（Dijkstra算法）

结点	队列Queue	D(0)	D(1)	D(2)	D(3)	D(4)

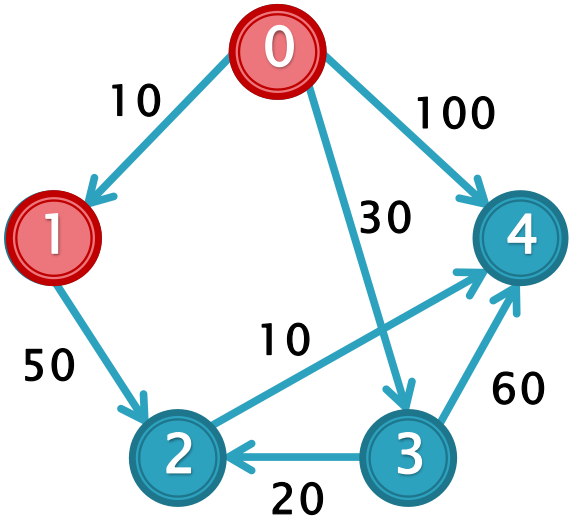
图(Graph)

- **图的相关问题**
  - 最短路径问题（Dijkstra算法）



结点	队列Queue	D(0)	D(1)	D(2)	D(3)	D(4)
0	{1, 3, 4}	0	10	∞	30	100

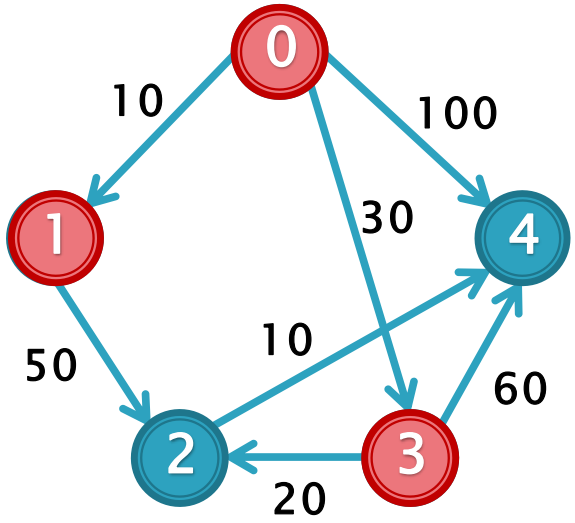
图(Graph)



- **图的相关问题**
  - 最短路径问题（Dijkstra算法）

结点	队列Queue	D(0)	D(1)	D(2)	D(3)	D(4)
0	{1, 3, 4}	0	10	∞	30	100
1	{3, 4, 2}	0	10	60	30	100

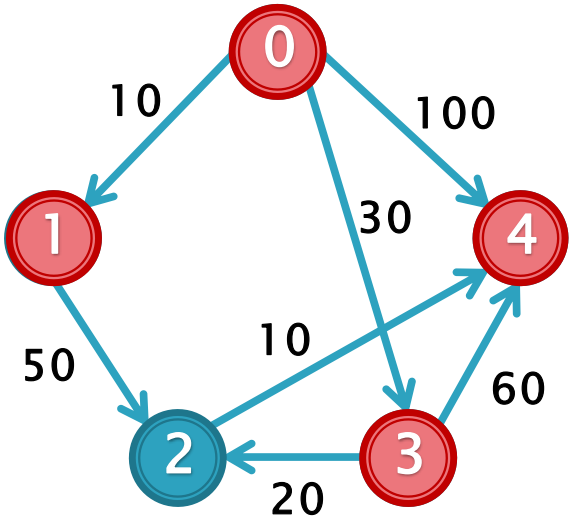
图(Graph)



- **图的相关问题**
  - 最短路径问题（Dijkstra算法）

结点	队列Queue	D(0)	D(1)	D(2)	D(3)	D(4)
0	{1, 3, 4}	0	10	∞	30	100
1	{3, 4, 2}	0	10	60	30	100
3	{4, 2}	0	10	50	30	90

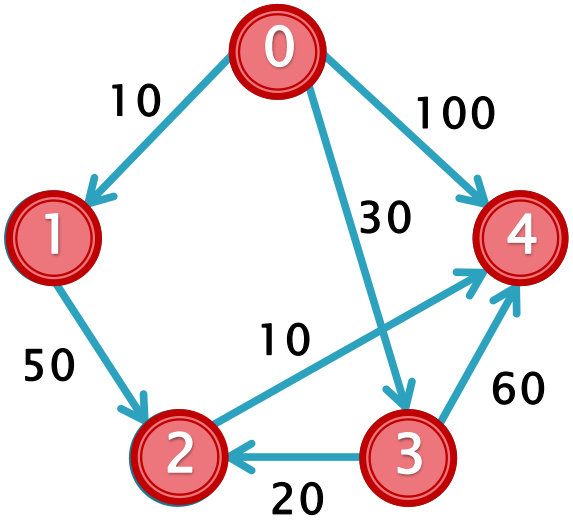
图(Graph)



- **图的相关问题**
  - 最短路径问题（Dijkstra算法）

结点	队列Queue	D(0)	D(1)	D(2)	D(3)	D(4)
0	{1, 3, 4}	0	10	∞	30	100
1	{3, 4, 2}	0	10	60	30	100
3	{4, 2}	0	10	50	30	90
4	{2}	0	10	50	30	90

图(Graph)



- **图的相关问题**
  - 最短路径问题（Dijkstra算法）

结点	队列Queue	D(0)	D(1)	D(2)	D(3)	D(4)
0	{1, 3, 4}	0	10	∞	30	100
1	{3, 4, 2}	0	10	60	30	100
3	{4, 2}	0	10	50	30	90
4	{2}	0	10	50	30	90
2	∅	0	10	50	30	60

图(Graph)

- **图的相关问题**
  - Dijkstra算法
  - **Pascal 描述**

```
Append(Point);
while (isEmpty() = false) do
begin
  CurrentPoint:= Serve();
  for i:= 1 to NUMBER_OF_VERTEX do
    if (Distance[i] > Distance[CurrentPoint] + Matrix[CurrentPoint][i]) then
    begin
      Distance[i]:= Distance[CurrentPoint] + Matrix[CurrentPoint][i];
      Append(i);
    end;
end;
```

图(Graph)

- **图的相关问题**
  - Dijkstra算法
  - **C++ 描述**

```
Queue.push(Point);
  while (!Queue.empty()) {
    int CurrentPoint = Queue.front();
    Queue.pop();
    for (int i = 0; i < Vertex; ++ i)
      if (Distance[i] > Distance[CurrentPoint] + Matrix[CurrentPoint][i]) {
        Distance[i] = Distance[CurrentPoint] + Matrix[CurrentPoint][i];
        Queue.push(i);
      }
  }
```

Thank You

谢浩哲
cshzxie@gmail.com