数据结构与算法

DATA STRUCTURE

第十二讲 栈和队列

胡浩栋

信息管理与工程学院

2018 - 2019 第一学期

课堂内容

- 作业回顾
- Stack
- Queue

2.1

在IntArray基础上实现insert/delete算法

```
class IntArray
{
public:
    IntArray(int size);

    virtual ~IntArray();

    IntArray(const IntArray & other);

    IntArray & operator= (const IntArray & rhs);

    void Insert(int index, int value);
    void Delete(int index);

    inline int Length() const { return _size; }

    void Reserve(int capacity);

    int GetRandom(int min, int max);

    int & operator[] (int index);
    const int & operator[] (int index) const;

    friend ostream & operator<< (ostream & out, const IntArray & list)
    {
        for (int i = 0; i < list._size; i++)
        {
            cout << list[i] << " ";
        }

        return out;
    }

private:
    int * _pData;
    int _capacity;
    int _size;

    void GetArray(int capacity);
    void Reset();
};
```

实现insert/delete

```
void IntArray::Insert(int index, int value)
{
    assert(index >= 0 && index < _size);

    if (_size + 1 >= _capacity)
    {
        Reserve(_capacity * 2 + 16);
    }

    if (_capacity > 0)
    {
        _size++;
        for (int i = _size - 1; i > index; i--)
        {
            _pData[i] = _pData[i-1];
        }
        _pData[index] = value;
    }
}
```

```
void IntArray::Delete(int index)
{
    assert(index >= 0 && index < _size);
    for (int i = index + 1; i < _size; i++)
    {
        _pData[i-1] = _pData[i];
    }
    _size--;

    if (_size > 16 && _size < _capacity / 4)
    {
        Reserve(_capacity / 2);
    }
}
```

```
void IntArray::Reserve(int capacity)
{
    if (capacity < _size)
    {
        return;
    }

    int * pBuffer = new (nothrow) int[capacity] {0};
    if (pBuffer != nullptr)
    {
        for (int i = 0; i < _size; i++)
        {
            pBuffer[i] = _pData[i];
        }

        Reset();
        _pData = pBuffer;
        _capacity = capacity;
    }
}

void IntArray::GetArray(int capacity)
{
    //避免分配负内存引起的exception
    if (capacity <= 0)
    {
        return;
    }

    Reset();

    // 申请大小为capacity的动态数组内存
    _pData = new (nothrow) int[capacity] {0};

    if (_pData != nullptr)
    {
        _capacity = capacity;
    }
}

void IntArray::Reset()
{
    if (_pData != nullptr)
    {
        delete [] _pData;
        _pData = nullptr;
        _capacity = 0;
        _size = 0;
    }
}
```

2.2翻转单链表

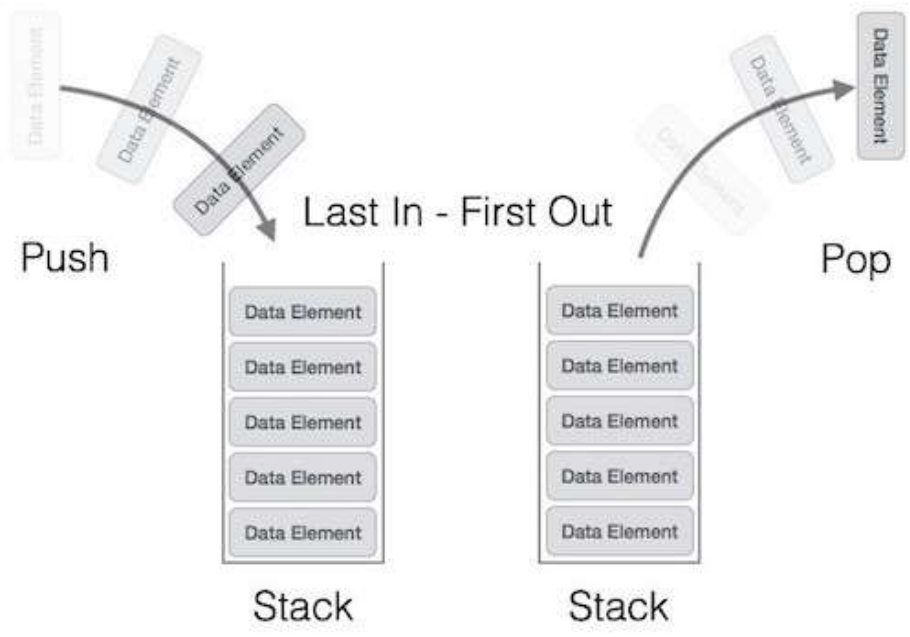
```
Node * Reverse(Node * head)
{
    Node * prev = nullptr;
    Node * curr = head;
    while (curr != nullptr)
    {
        Node * temp = curr->next;
        curr->next = prev;
        prev = curr;
        curr = temp;
    }

    return prev;
}
```

Stack

栈抽象

- 后进先出LIFO

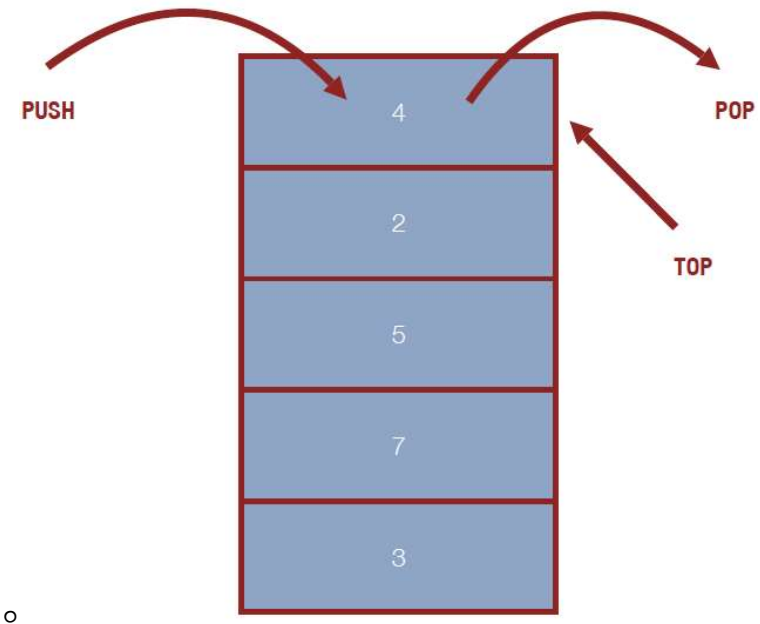
- 可以用数组实现
- 也可以用链表实现
- 可以是固定栈大小
- 也可以是动态改变栈大小



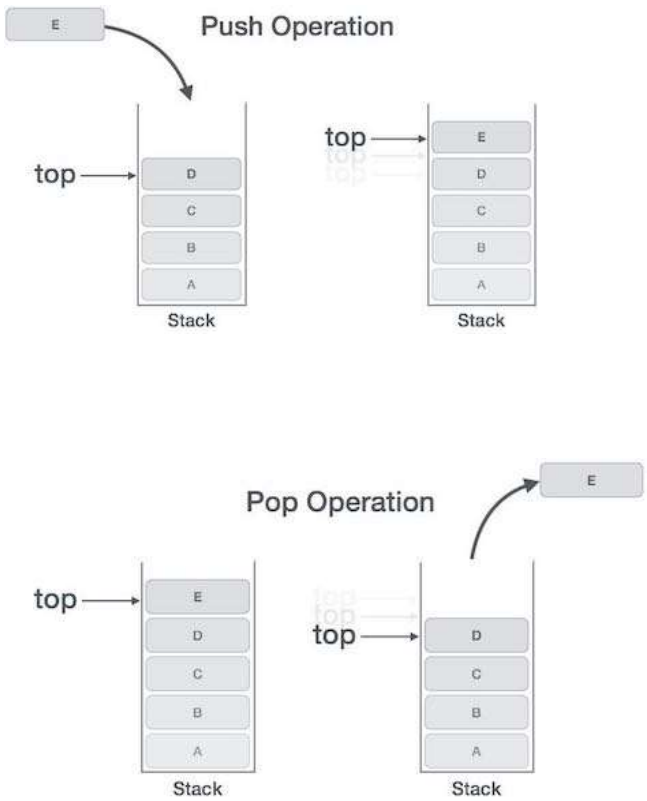
数据结构Stack

- Push(*value*) - 在最顶层加入数据。
- Pop() - 返回并移除最顶层的数据。
- Top()或者 Peek() - 返回最顶层数据的值，但不移除它。

- IsEmpty() -返回一个布尔值，表示当前stack是否为空栈。
- IsFull() - 返回一个布尔值，表示当前栈是否满了。



用静态数组实现IntStack




```
class IntStack
{
public:
    IntStack() : _top(0) {};

    inline bool IsEmpty() {return _top <= 0;}
    inline bool IsFull() { return _top >= Ssize; }

    int Peek()
    {
        assert (_top > 0);
        return _data[_top - 1];
    }

    void Push(int i)
    {
        assert (i < Ssize);
        _data[_top++] = i;
    }

    int Pop()
    {
        assert (_top > 0);
        return _data[--_top];
    }

 private:
    static const int Ssize = 100;
    int _data[Ssize];
    int _top;
};
```

链表实现Stack

- 没有大小限制
- 保存数据指针（不是数据自身）

```
template <class T>
class Stack
{
public:
    Stack() : _pHead(nullptr) {}
    ~Stack()
    {
        while (_pHead != nullptr)
        {
            Pop();
        }
    }

    bool IsEmpty() const { return _pHead == nullptr; }

    T* Peek() const
    {
        return _pHead != nullptr ? _pHead->pData : nullptr;
    }

    void Push(T* element)
    {
        _pHead = new Node(element, _pHead);
    }

    T* Pop()
    {
        if (_pHead == nullptr)
        {
            return nullptr;
        }

        T* pElement = _pHead->pData;
        Node* oldHead = _pHead;
        _pHead = _pHead->pNext;
        delete oldHead;

        return pElement;
    }

 private:
    struct Node
    {
        T * pData;
        Node * pNext;
        Node(T* data, Node * next) : pData(data), pNext(next) {}
    };

    Node *_pHead;
};
```

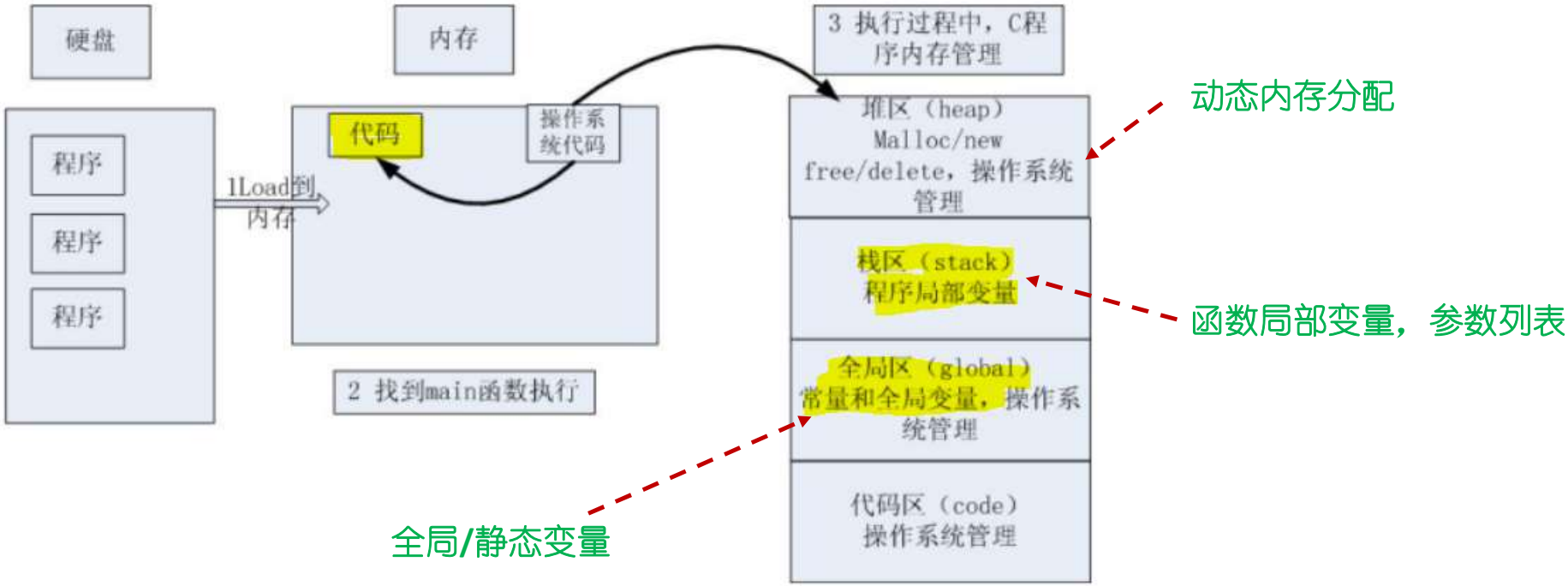
Stack优缺点

- 只能读取最上面的元素
- 之前的数据只能通过Pop
- 不能查找

- 读取数据元素非常快的动态数据结构
- Push，Pop，Top操作都是O(1)
- 使用std::stack<float>
- 现实中有好多stack的例子，包括C++语言本身，树遍历……

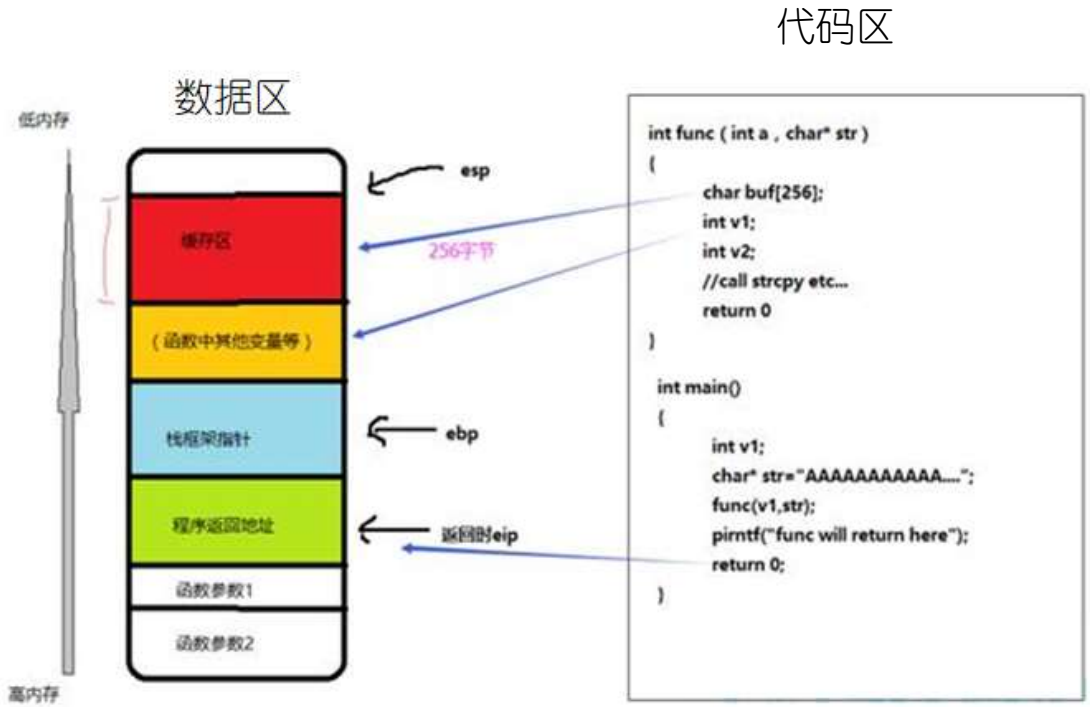
思考问题：能否在IntStack加上操作GetMin()，如果要求时间O(1)呢？

程序调用内存管理示意图



函数调用单个函数存储内容

- 函数调用时，指令从代码区的函数入口开始执行
- 数据区所需存储内容：返回值，参数列表，局部变量
- 栈运作方式



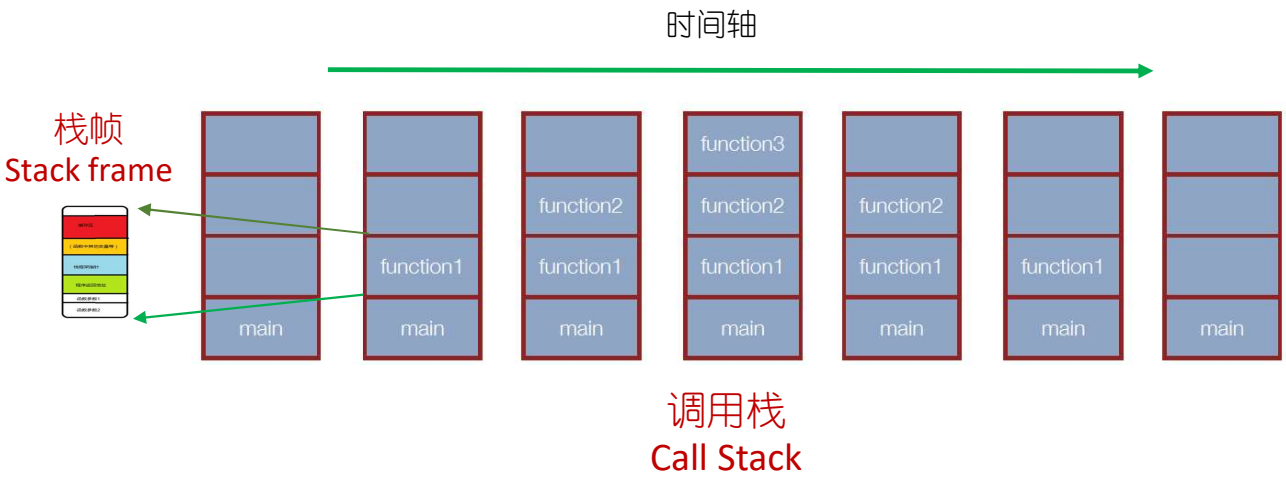
函数调用栈:

- Call stack, LIFO顺序
- 其实是固定大小的，数组实现的栈

```
main() {
    function1();
    return;
}
```

```
function1() {
    function2();
    return;
}
```

```
function2() {
    function3();
    return;
}
```

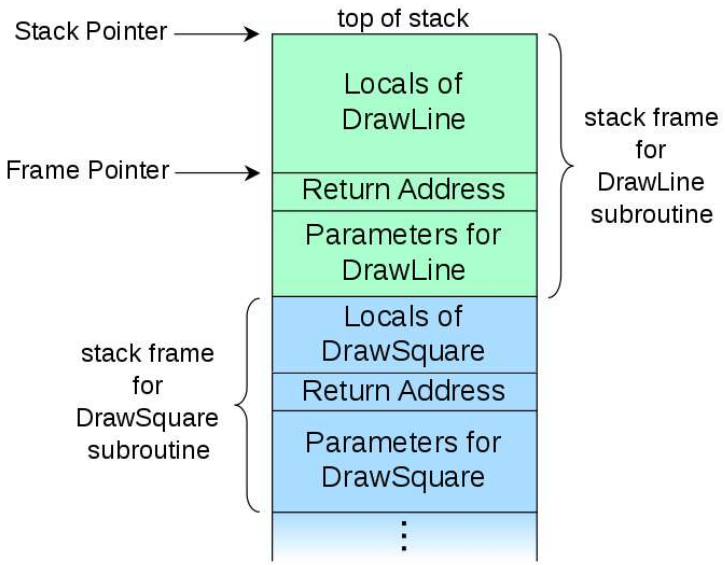


函数调用的额外开销

- 函数调用时要把一些变量压到调用栈
  - 参数列表
  - 局部变量
  - （返回值）

- 函数返回时，恢复被调用前的场景，然后返回

```
void DrawlLine(int x1, int x2, int y)
{
    ...
}
void DrawSquare(int left, int top, int right, int bottom)
{
    DrawLine(left, right, top);
    ...
}
```



Debug例子

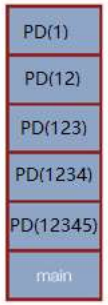
```
48  void PrintDigit(int x)
49  {
50      if (x > 10)
51      {
52          PrintDigit(x / 10);
53      }
54      cout << x % 10 << " ";
55  }
56
57  int main()
58  {
59      int num = 0;
60      cin >> num;
61      PrintDigit(num);
62
63      return 0;
64  }
```

栈帧2
Stack frame

```
> frame 2
#2  0x004014b4 in PrintDigit (x=123) at D:\Blocks\test\main.cpp:52
□□D:\Blocks\test\main.cpp:52:917:beg:0x4014b4
```

Call stack

Nr	Address	Function	File	Line
0		PrintDigit (x=1)	D:\Blocks\test\main.cpp	54
1	0x4014b4	PrintDigit(x=12)	D:\Blocks\test\main.cpp	52
2	0x4014b4	PrintDigit(x=123)	D:\Blocks\test\main.cpp	52
3	0x4014b4	PrintDigit(x=1234)	D:\Blocks\test\main.cpp	52
4	0x4014b4	PrintDigit(x=12345)	D:\Blocks\test\main.cpp	52
5	0x401534	main()	D:\Blocks\test\main.cpp	61



X=1

X=12

X=123

X=1234

X=12345

Memory

Address: 0x6afe50 Bytes: 256 Go

(e.g. 0x401060, or &variable, or $$eax)

```
0x6afe50: 7c fe 6a 00 ff ff ff ff|78 fe 6a 00 b4 14 40 00   |þj.ÿÿÿÿxþj.´.@.
0x6afe60: 01 00 00 00 b8 fe 6a 00|bc fe 6a 00 f7 0d 39 77   ....¸þj.¼þj.÷.9w
0x6afe70: ff ff ff ff 7c fe 6a 00|98 fe 6a 00 b4 14 40 00   ÿÿÿÿ|þj.þj.´.@.
0x6afe80: 0c 00 00 00 59 14 af 1a|94 fe 6a 00 f4 61 02 77   ....Y.¯.þj.ôa.w
0x6afe90: 9c 0c 03 00 a0 26 40 00|b8 fe 6a 00 b4 14 40 00   ... &@.¸þj.´.@.
0x6afea0: 7b 00 00 00 50 fe 6a 00|1c 23 07 77 ec fe 6a 00   {...Þþj..#.wìþj.
0x6afeb0: f8 5f 02 77 08 00 00 01|d8 fe 6a 00 b4 14 40 00   ø_.w....Øþj.´.@.
0x6afec0: d2 04 00 00 80 12 40 00|80 12 40 00 00 a0 2b 00   Ò....@..@.. +.
0x6afed0: 80 12 40 00 80 12 40 00|08 ff 6a 00 34 15 40 00   .@..@..ÿj.4.@.
0x6afee0: 39 30 00 00 3d 92 b9 32|fe ff ff ff 8a 6a 01 77   90..=¹2þÿÿÿj.w
0x6afef0: 7d 6b 01 77 f0 c4 41 00|94 ff 6a 00 39 30 00 00   }k.wðÄA.ÿj.90..
0x6aff00: f0 c4 41 00 20 ff 6a 00|94 ff 6a 00 fd 10 40 00   ðÄA. ÿj.ÿj.ý.@.
0x6aff10: 23 00 00 00 02 00 00 00|00 a0 2b 00 fd 10 40 00   #........ +.ý.@.
0x6aff20: 01 00 00 00 60 16 03 00|b0 1f 03 00 00 90 48 00   ....`...°....H.
0x6aff30: 50 ff 6a 00 ff ff ff ff|54 ff 6a 00 70 c8 01 77   Pÿj.ÿÿÿÿTÿj.pÈ.w
0x6aff40: fd 91 b9 32 fe ff ff ff|66 c5 01 77 80 c5 01 77   ý¹2þÿÿÿfÅ.wÅ.w
```

程序崩溃：stack overflow

- 函数调用栈空间有限，2M~4M大小
- 嵌套调用太多（注意不是总的调用次数），会造成程序崩溃。
- 比如

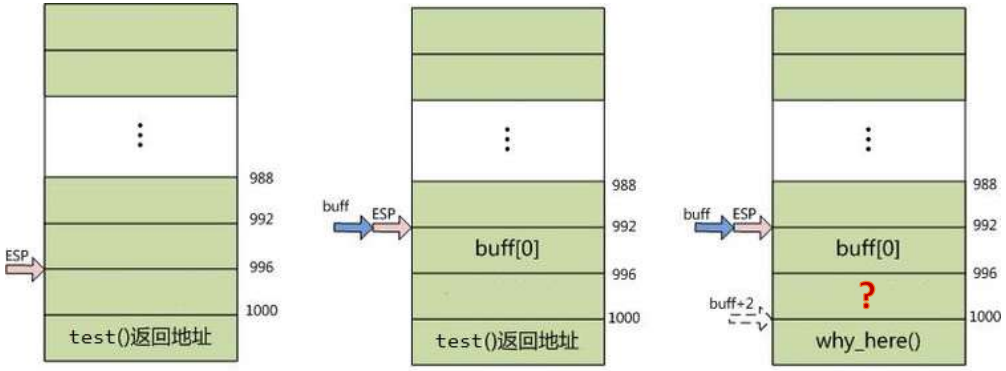
```
void DonotCallMe()
{
    int a = 0;
    DonotCallMe();
}

int main()
{
    DonotCallMe();
}
```

```
#65130 0x00401b29 in DonotCallMe() ()
#65131 0x00401b29 in DonotCallMe() ()
#65132 0x00401b29 in DonotCallMe() ()
#65133 0x00401b29 in DonotCallMe() ()
#65134 0x00401b29 in DonotCallMe() ()
#65135 0x00401b29 in DonotCallMe() ()
#65136 0x00401b29 in DonotCallMe() ()
#65137 0x00401b29 in DonotCallMe() ()
#65138 0x00401b29 in DonotCallMe() ()
#65139 0x00401b29 in DonotCallMe() ()
#65140 0x00401b29 in DonotCallMe() ()
#65141 0x00401b29 in DonotCallMe() ()
#65142 0x00401b29 in DonotCallMe() ()
#65143 0x00401b29 in DonotCallMe() ()
#65144 0x00401b29 in DonotCallMe() ()
#65145 0x00401b3b in main ()
(gdb)
```

函数调用时发生缓冲区溢出

```c
//Have we invoked this function?
void why_here(void)
{
    printf("why u r here?!\n");
    exit(9999);
}

void test()
{
    long buff[1];
    buff[2] = (int)why_here;
}

int main()
{
    test();
    return 0;
}
```

```
why u r here?!

Process returned 9999 (0x270F)
Press any key to continue.
```



进入main()函数，编译器字自动将返回地址和返回值压入栈中

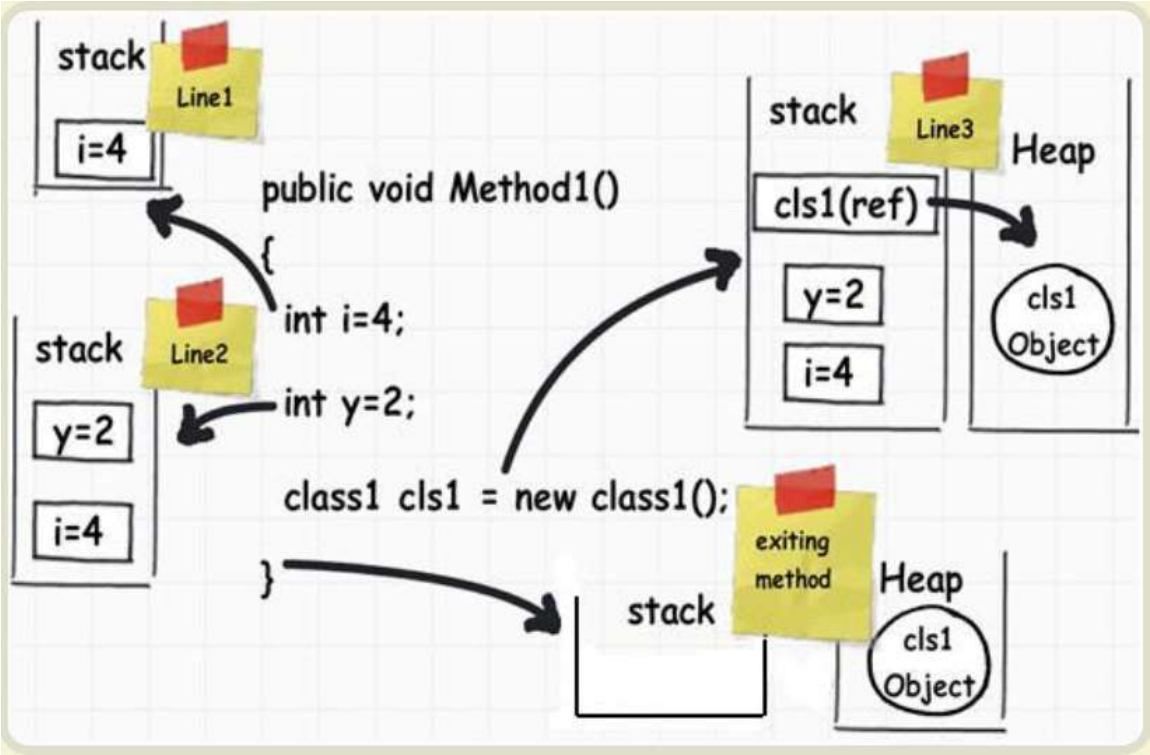
int buff[1]; 声明一个数组buff[1],在栈中为其开辟存储空间

buff[2]=(int)why_here; 由于不进行越界检查，引用buff[2]不报错,buff[2]中的内容被修改，main()返回地址被why_here()入口地址取代

- **如果** `buff[1] = (int)why_here;` **？？？**

内存栈和堆

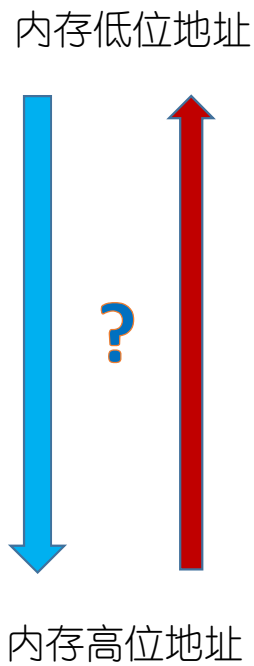
```
void Method1()
{
    int i = 4;
    int y = 2;

    Class1 cls1 = new Class1();
}
```



如何判断栈的增长方向

```c
void SecondCall(int *a)
{
    int b = 0;
    printf("Stack grows from 0x%x to 0x%x\n",a,&b);
}

void FirstCall()
{
    int a = 0;
    SecondCall(&a);
}

int main(int argc, char** argv)
{
    FirstCall();
    return 0;
}
```

```
Stack grows from 0x61fefc to 0x61fecc
```



作业

1. 用IntArray实现IntStack类
   - IntStack没有最大空间限制

```
class IntStack
{
public:
    IntStack();

    bool IsEmpty();
    int Peek();
    void Push(int x);
    int Pop();

 private:

};
```

2. 用课件stack<char>检查程序（字符串）里的括号是否匹配:( ) [ ] { }

```
bool CheckParentheses(std::string input)
```

```
5   int main()
6   {
7       for (int i = 0; i < 10; i++) {
8           cout << i;
9
10      return 0;
11  }
12
```

Line	Message
	=== Build: Debug in test (compiler: GNU GCC Compiler) ===
	In function 'int main()':
11	error: expected '}' at end of input
	=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===

栈的应用：**计算表达式**

- 比如如何计算： 5 * 4 - 8 / 2 + 9

- 需要知道优先级，左结合/右结合，括号来消除歧义

  ➔ ((5 * 4) – (8 / 2)) + 9

- 中缀infix表示法，符合人的习惯。但是计算机不习惯
- 后缀postfix表示法，5 4 * 8 2 / - 9 +

后缀表示法

- 运算符优先级

  $a + b * c \Rightarrow abc * +$

  $(a + b) * c \Rightarrow ab + c *$

- 运算符左结合/右结合

  $a + b\text{^}3\text{^}2 \;\Rightarrow\; ab32\text{^}\text{^} +$

  $a + (b\text{^}3)\text{^}2 \;\Rightarrow\; ab3\text{^}2\text{^} +$

Sr. No.	Infix Notation	Postfix Notation
1	a + b	a b +
2	(a + b) * c	a b + c *
3	a * (b + c)	a b c + *
4	a / b + c / d	a b / c d / +
5	(a + b) * (c + d)	a b + c d + *
6	((a + b) * c) - d	a b + c * d -

- 后缀表示法不需要括号，就能区分优先级和结合顺序

1. 如何计算后缀表达式

- 使用栈
- 从左到右扫描表达式
- 如果看到操作数，压入栈
- 如果看到运算符，从栈里Pop出两个操作数，并计算
- 把计算结果再压入栈
- 执行完所有操作后，最后的结果在栈里（还剩一个元素）

- $3 * (2 + 1) \Rightarrow 3\ 2\ 1\ + *$

栈：$3\ 2\ 1 \rightarrow 3 \xrightarrow{+} 3\ 3 \xrightarrow{*} 9$

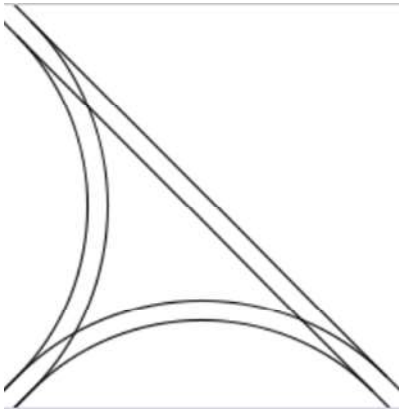
```
int Evaluate(const string& expression)
{
    IntStack stk;
    for (size_t i = 0; i < expression.size(); i++)
    {
        char ch = expression.at(i);
        if (isspace(ch))
        {
            continue;
        }

        if (isdigit(ch))
        {
            int val = ch - '0';

            while (i + 1 < expression.size() && isdigit(expression.at(i + 1)))
            {
                val = val * 10 + expression.at(++i) - '0';
            }

            stk.Push(val);
            continue;
        }

        int op2 = stk.Pop();
        int op1 = stk.Pop();
        switch(ch)
        {
        case '+':
           stk.Push(op1 + op2);
           break;
        case '-':
           stk.Push(op1 - op2);
           break;
        case '*':
           stk.Push(op1 * op2);
           break;
        case '/':
           stk.Push(op1 / op2);
           break;
        case '^':
            {
            int val = 1;
            for (int j = 0; j < op2; j++)
            {
                val *= op1;
            }
            stk.Push(val);
            break;
            }
        default:
            assert(false);
            break;
         }
    }

    return stk.Pop();
}
```

2. 如何把中缀转换成后缀

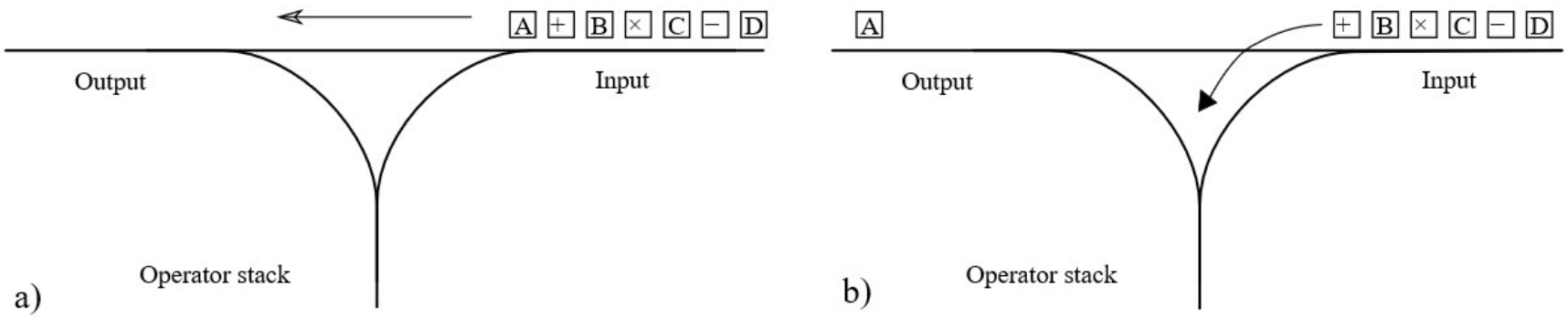
Shunting-yard algorithm

由 **Edsger Dijkstra** 发明的

- 需要一个输出列表output,
- 和一个运算符栈operator stack,
- 然后从左到右扫描中缀表达式



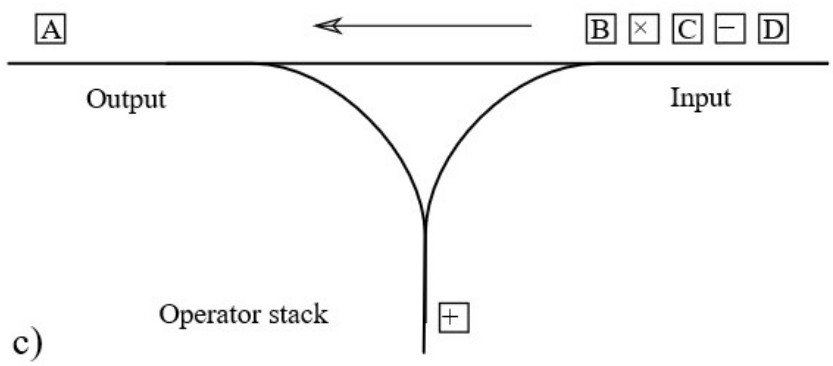
a. 转换$A + B \times C - D$

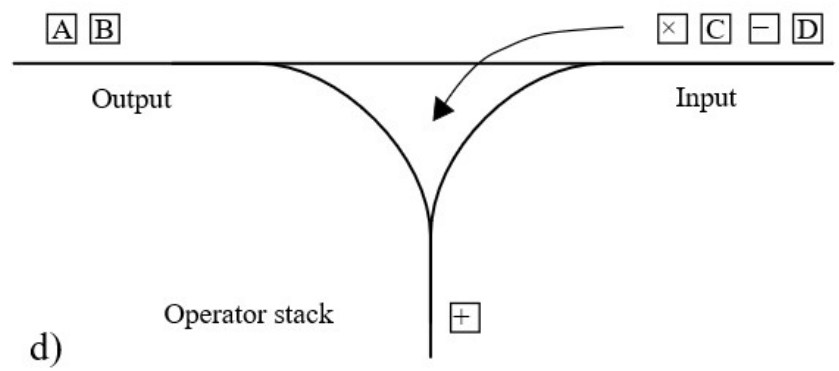


需要一个输出列表output，和一个运算符栈operator stack，然后从左到右扫描中缀表达式

1. 操作数：直接输出到结果列表
2. 运算符：压入栈，如果栈是空的话

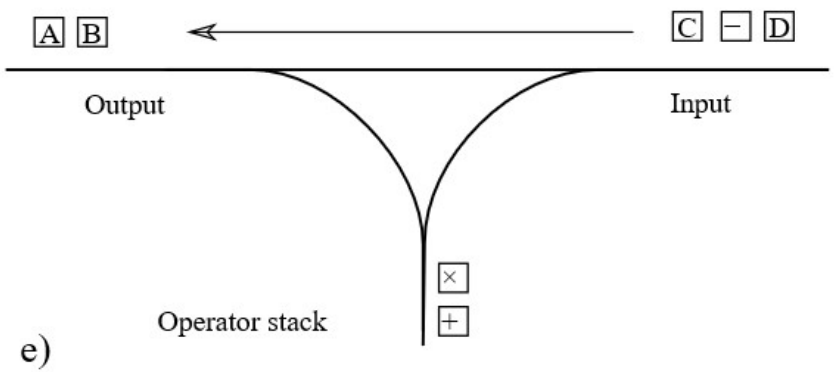
b. 转换$A + B \times C - D$

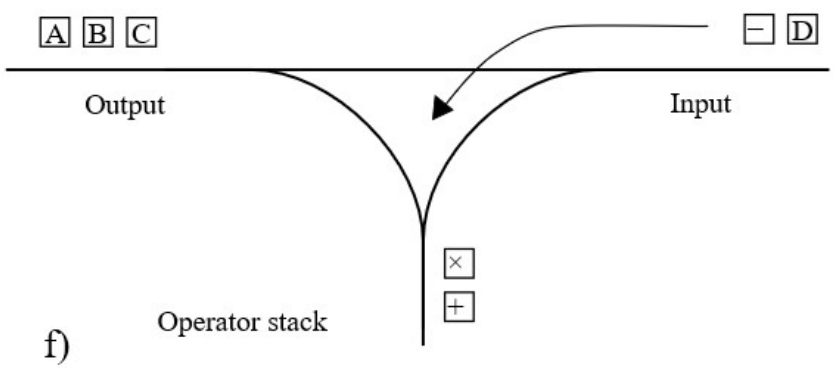




1. 操作数B：直接输出到结果列表
2. 运算符×：压入栈，如果栈顶的运算符优先级低

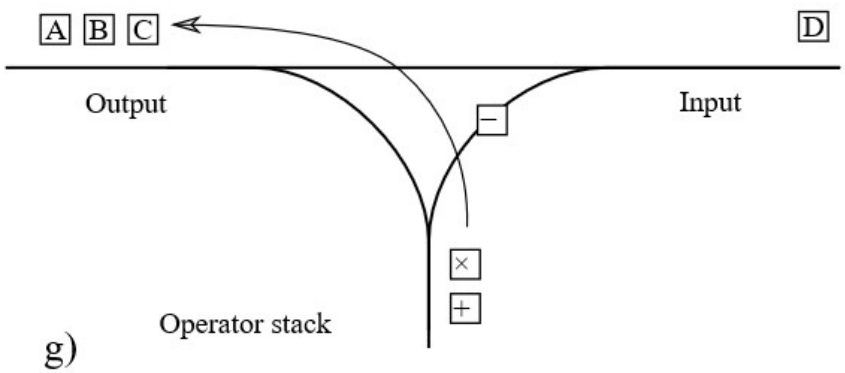
c. 转换$A + B \times C - D$

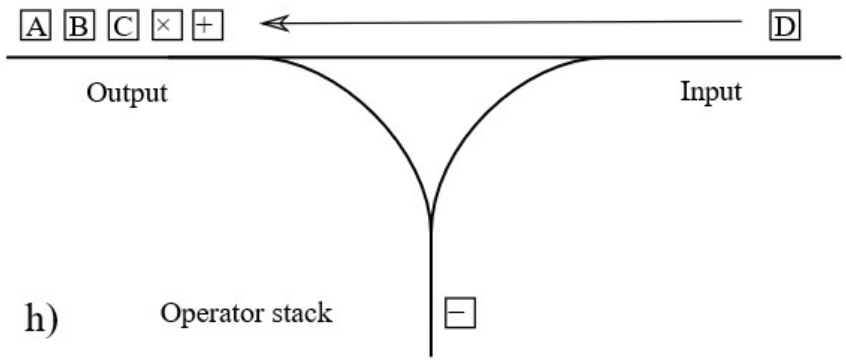




1. 操作数C：直接输出到结果列表
2. 运算符 − ：不能直接压入栈，因为栈顶的运算符优先级高

d. 转换$A + B \times C - D$

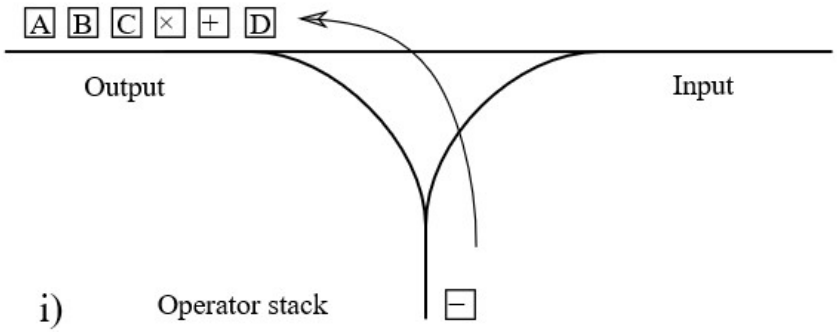


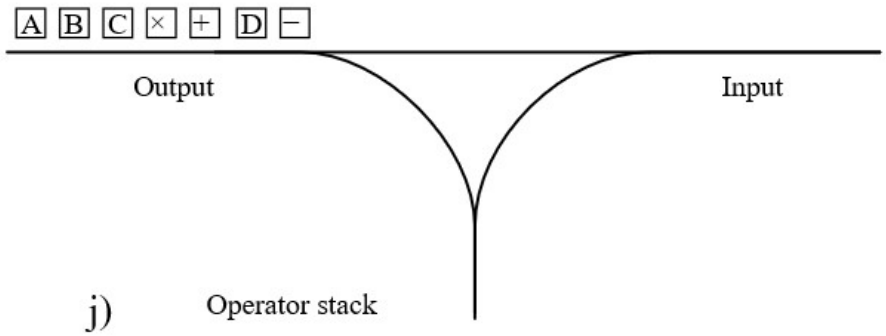


在压入运算符“ − ”之前，需要先把满足以下条件的栈顶运算符输出到结果列表

1. 运算符 × ：优先级高
2. 运算符+：优先级一样，但是+是左结合的

e. 转换$A + B \times C - D$





最后，把栈里面的剩余运算符都输出到结果列表output

调度场算法的原理

- 后缀表达式的算术符次序就是正确计算的次序

$$A + B \times C - D \quad \rightarrow \quad AB \times C + D -$$

- 调度场算法解决了通过一遍扫描决定算术符正确次序
- 把中缀次序的算术符放入栈里等待调度
- 根据优先级和结合方向决定出栈次序

提高：其实数字也可以当作算术符，看作最优先出栈

带括号的处理

Input: 3 + 4 × 2 ÷ (1 − 5) ^ 2 ^ 3

operator	precedence	associativity
^	4	Right
×	3	Left
÷	3	Left
+	2	Left
−	2	Left

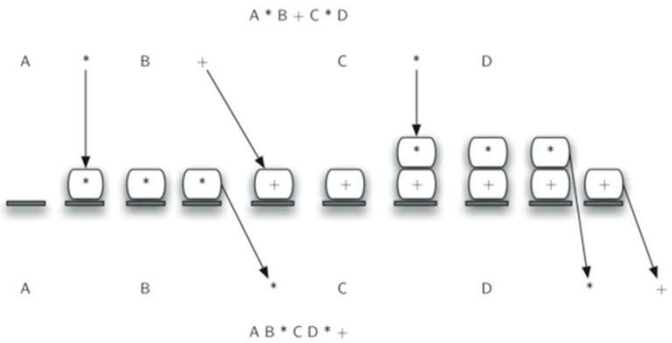
The symbol ^ represents the power operator.

- 如果运算符是左括号，直接压入栈
- 如果栈顶是左括号，运算符压入栈
- 如果运算符是右括号，把栈顶的运算符都输出到结果列表，直到左括号
- 括号不输出到结果列表

Token	Action	Output (in RPN)	Operator stack	Notes
3	Add token to output	3		
+	Push token to stack	3	+	
4	Add token to output	3 4	+	
×	Push token to stack	3 4	× +	× has higher precedence than +
2	Add token to output	3 4 2	× +	
÷	Pop stack to output	3 4 2 ×	+	÷ and × have same precedence
	Push token to stack	3 4 2 ×	÷ +	÷ has higher precedence than +
(	Push token to stack	3 4 2 ×	(÷ +	
1	Add token to output	3 4 2 × 1	(÷ +	
−	Push token to stack	3 4 2 × 1	− (÷ +	
5	Add token to output	3 4 2 × 1 5	− (÷ +	
)	Pop stack to output	3 4 2 × 1 5 −	(÷ +	Repeated until "(" found
	Pop stack	3 4 2 × 1 5 −	÷ +	Discard matching parenthesis
^	Push token to stack	3 4 2 × 1 5 −	^ ÷ +	^ has higher precedence than ÷
2	Add token to output	3 4 2 × 1 5 − 2	^ ÷ +	
^	Push token to stack	3 4 2 × 1 5 − 2	^ ^ ÷ +	^ is evaluated right-to-left
3	Add token to output	3 4 2 × 1 5 − 2 3	^ ^ ÷ +	
end	Pop entire stack to output	3 4 2 × 1 5 − 2 3 ^ ^ ÷ +		

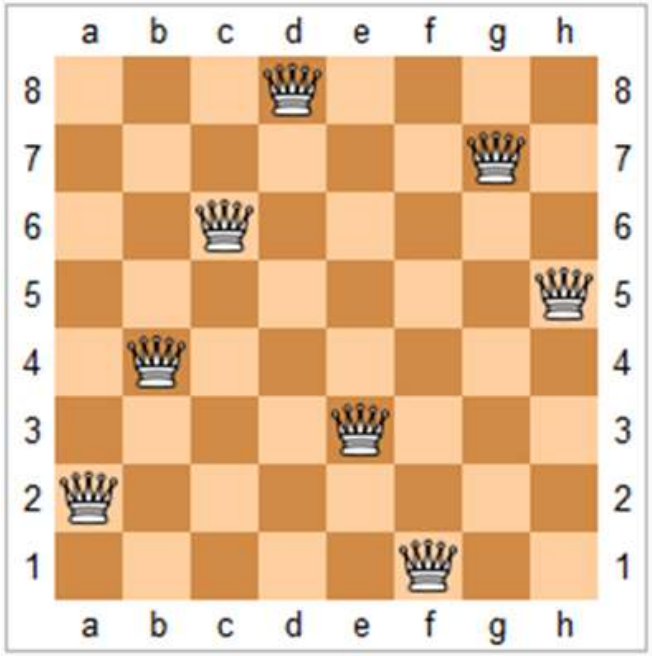
调度场算法实现

- 数字直接输出到结果列表
- 建一个栈放运算符，分情况输出到结果列表
  1) 如果栈顶的优先级低，或左括号，运算符压入栈
  2) 如果栈顶的优先级高，或者优先级一样不过运算符是左结合的，把栈里这些运算符输出到结果列表之后，再把运算符压入栈
  3) 如果运算符是左括号，直接压入栈
  4) 如果运算符是右括号，把栈顶的运算符都输出到结果列表，直到左括号
- 最后的运算符依次加入结果列表

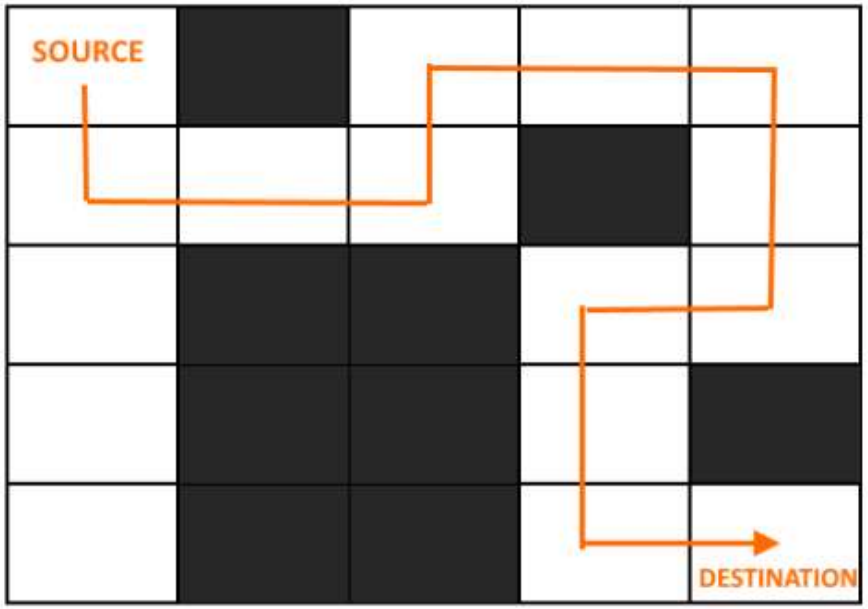



```
string ConvertToPostfix(const string & infix)
{
    string postfix;
    Stack<char> stk;
    stk.Push(new char('#'));

    for (size_t i = 0; i < infix.size(); i++)
    {
        char ch = infix[i];

        if (!IsOperator(ch))
        {
            postfix.append(new char(ch), 1);
            continue;
        }

        if(ch == '(')
        {
            stk.Push(new char(ch));
            continue;
        }

        if(ch == ')')
        {
            while (*stk.Peek() != '(')
            {
               postfix.append(stk.Pop(), 1);
            }

            stk.Pop();//pop out (.
            continue;
        }

        if(Precedence(ch) > Precedence(*stk.Peek()))
        {
            stk.Push(new char(ch));
            continue;
        }

        while(Precedence(ch) <= Precedence(*stk.Peek()))
        {
            postfix.append(stk.Pop(), 1);
        }

        stk.Push(new char(ch));
    }

    while(*stk.Peek() != '#')
    {
        postfix.append(stk.Pop(), 1);
    }

    return postfix;
}
```

Puzzle



八皇后问题



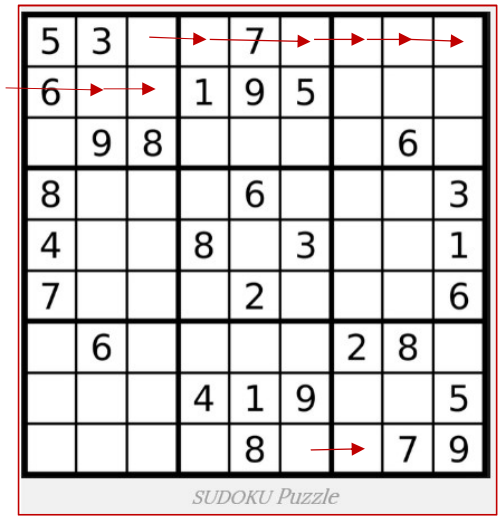
迷宫问题

数独（九宫格）

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

SUDOKU Puzzle

用栈穷举




```
bool Solve()
{
    for (int row = 0; row < 9; row++)
    {
        for (int col = 0; col < 9; col++)
        {
            // Those are initial values at (row, col)
            if (puzzle[row][col] > 0)
            {
                continue;
            }

            // A loop to enumerate possible values, until we find one.
            // If no legal value found at current possible, pop previous setting and retry.
            while (true)
            {
                puzzle[row][col]++;

                // Whenever the new value at (row, col) does not break, push to stack.
                if (CheckCol(row, col) && CheckRow(row, col) && CheckGroup(row, col))
                {
                    stk.Push(row, col);
                    break;
                }

                // If all values break, pop previous one and retry.
                while (!(puzzle[row][col] %= 9))
                {
                    if (stk.IsEmpty())
                    {
                        return false;
                    }
                    stk.Pop(row, col);
                }
            }
        }
    }

    return true;
}
```

队列Queue

队列抽象

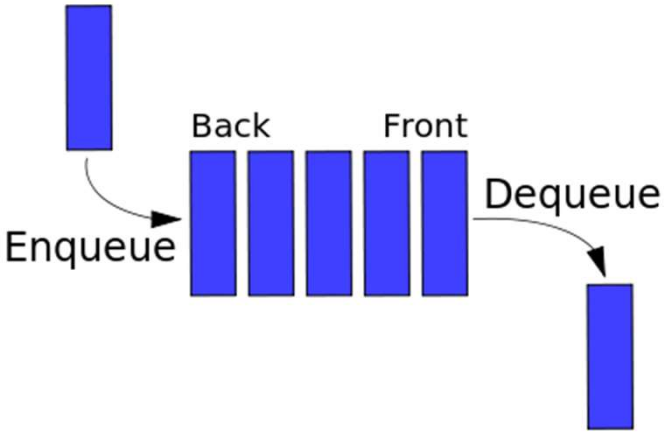
- 先进先出FIFO
- 队列有头尾，头部总是移去，尾部总是加入

- 可以用数组实现
- 也可以用链表实现

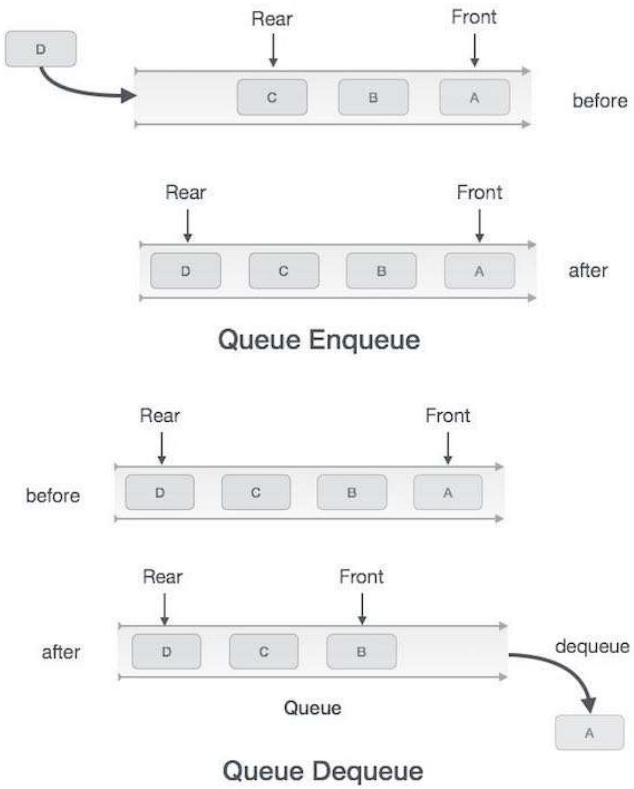
数据结构Queue

- Enqueue(*value*) - 在队尾加入数据。
- Dequeue() - 返回并移除队首的数据。

- Peek() - 返回队首数据的值，但不移除它。
- IsEmpty() -返回一个布尔值，表示当前队列是否为空。
- Size() - 返回当前队列数据个数。

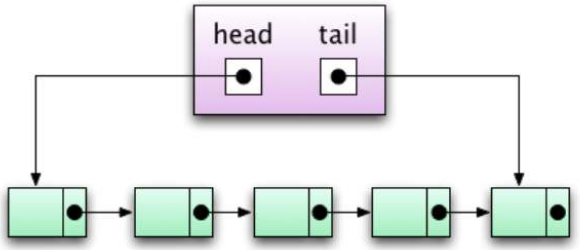


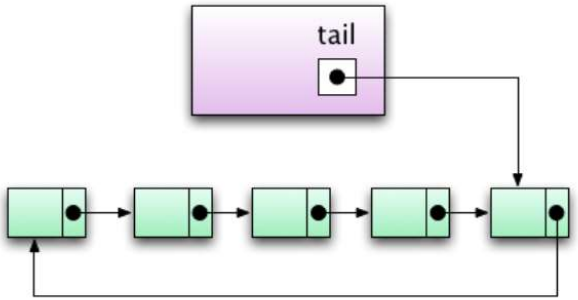
用数组实现Queue



- 参考书里
- 注意“假溢出”
- 改进是使用循环队列

用链表实现Queue






```
template <class T>
class Queue
{
public:
    Queue() : _pHead(nullptr),_pTail(nullptr) {}
    ~Queue()
    {
        while (_pHead != nullptr)
        {
            delete Dequeue();
        }
    }

    void Enqueue(T* element)
    {
        if (_pTail == nullptr)
        {
            _pHead = _pTail = new Node(element, nullptr);
            return;
        }

        _pTail->pNext = new Node(element, nullptr);
        _pTail = _pTail->pNext;
    }
    T* Dequeue()
    {
        if (IsEmpty())
        {
            return nullptr;
        }

        T* pElement = _pHead->pData;
        Node* oldHead = _pHead;
        _pHead = _pHead->pNext;
        delete oldHead;

        if (_pHead == nullptr)
        {
            _pTail = _pHead;
        }

        return pElement;
    }

    T* Peek()
    {
        return _pHead != nullptr ? _pHead->pData : nullptr;
    }

    inline bool IsEmpty() const { return _pHead == nullptr; }

 private:
    struct Node
    {
        T * pData;
        Node * pNext;
        Node(T* data, Node * next) : pData(data), pNext(next) {}
    };

    Node *_pHead;
    Node *_pTail;
};
```

Queue特点

- 只在队列尾部插入新元素
- 只在队列头部移去元素
- FIFO
- Std::Queue<double>()
- 简单且常用的动态数据结构
- Enqueue，Dequeue，Peek操作都是O(1)。
- 现实中有好多Queue的例子，包括打印机任务队列，买票窗口，……

选择题：

```
while (!queue.isEmpty()) {
    cout << queue.dequeue() << " ";
}
```

```
int origQSize = queue.size();
for (int i=0; i < origQSize; i++)
```

```
Queue<int> queue;
// produce: {1, 2, 3, 4, 5, 6}
for (int i = 1; i <= 6; i++) {
    queue.enqueue(i);
}
for (int i = 0; i < queue.size(); i++) {
    cout << queue.dequeue() << " ";
}
cout << queue.toString() << "  size " << queue.size() << endl;
```

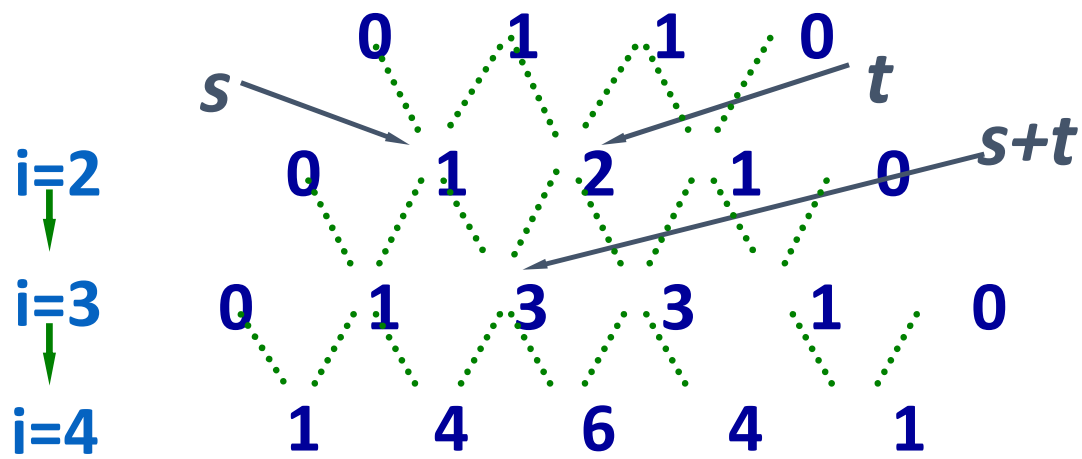
A. 1 2 3 4 5 6 {} size 0
B. 1 2 3 {4,5,6} size 3
C. 1 2 3 4 5 6 {1,2,3,4,5,6} size 6
D. none of the above

例子：打印二项展开式$(a+b)^n$的系数

杨辉三角形 **(Pascal's Triangle)**

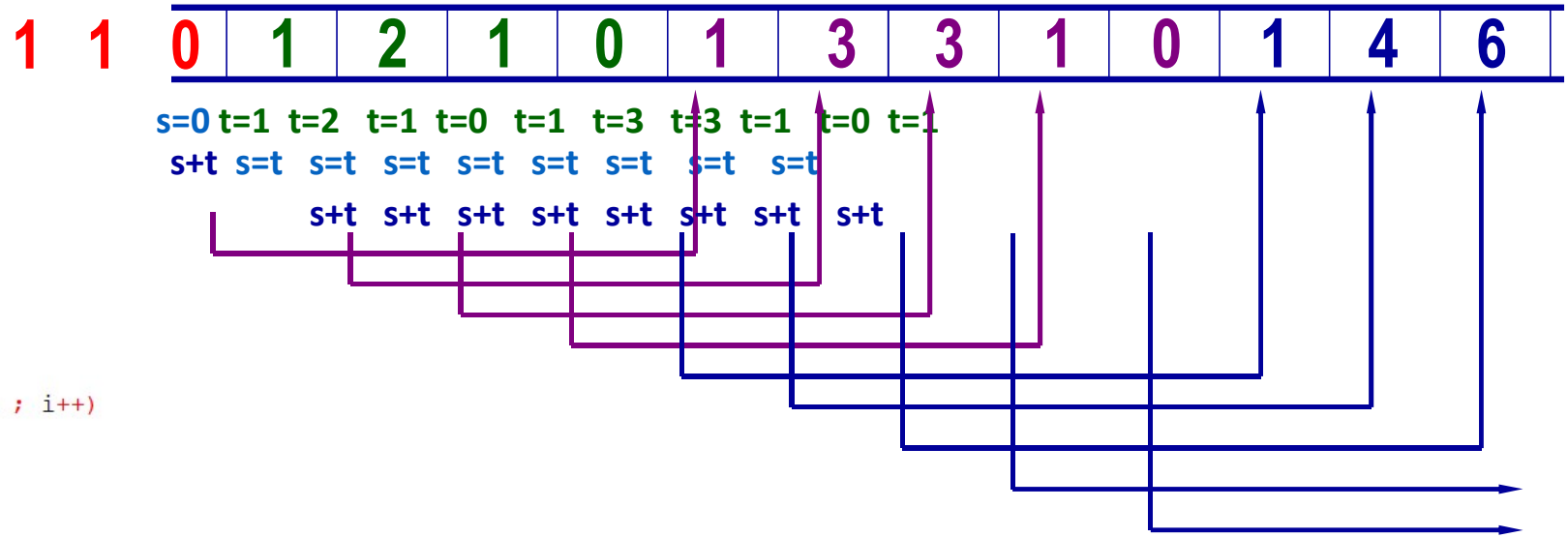
							1		1								$i =$	1
						1		2		1								2
					1		3		3		1							3
				1		4		6		4		1						4
			1		5		10		10		5		1					5
		1		6		15		20		15		6		1				6

分析第i行元素与第$i+1$行元素的关系



- 每行的两侧各添加一个0，设s是第i行第j-1个元素的值，s初值为0，t是第i行第j个元素的值，则s+t是第i+1行第j个元素的值。

代码




```
void Pascal(int n)
{
    queue<int> que;
    que.push(1);

    for (int i = 0; i < n ; i++)
    {
        cout << endl;
        que.push(0);
        int prev = 0;

        for (int j = 0; j < i+2; j++)
        {
            int curr = que.front();
            que.pop();
            que.push(curr + prev);
            prev = curr;

            if (j != i+1)
            {
                cout << setw(5) << prev << " ";
            }
        }
    }
}
```

```
1
1     1
1     2     1
1     3     3     1
1     4     6     4     1
1     5    10    10     5     1
1     6    15    20    15     6     1
1     7    21    35    35    21     7     1
1     8    28    56    70    56    28     8     1
1     9    36    84   126   126    84    36     9     1
```

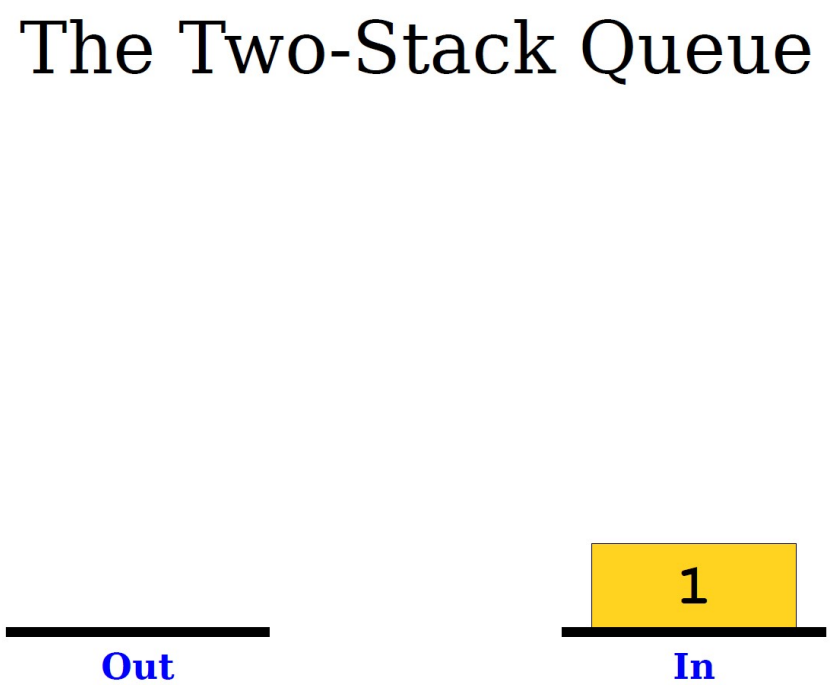
思考题

如何用Stack实现Queue

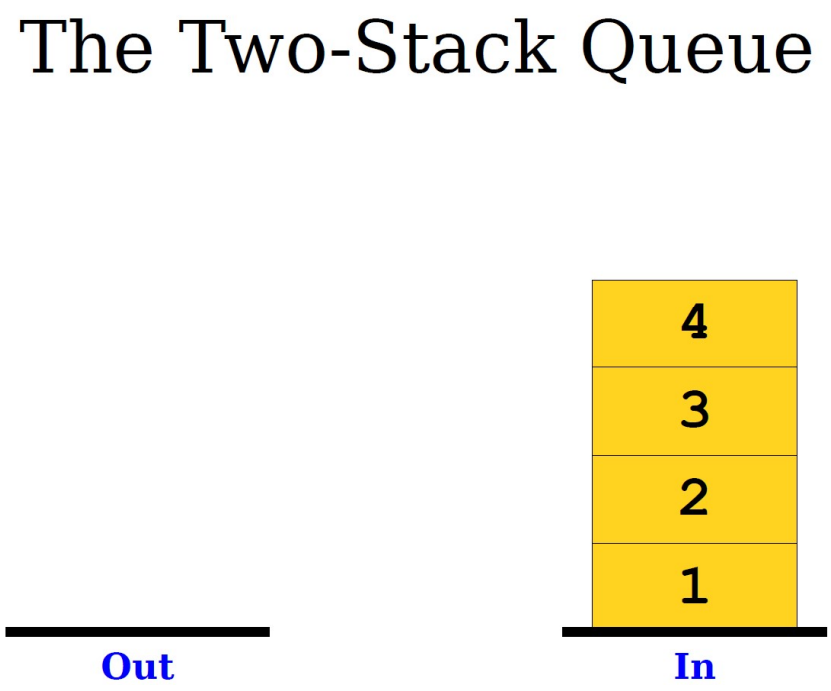
用两个Stack实现Queue

The Two-Stack Queue

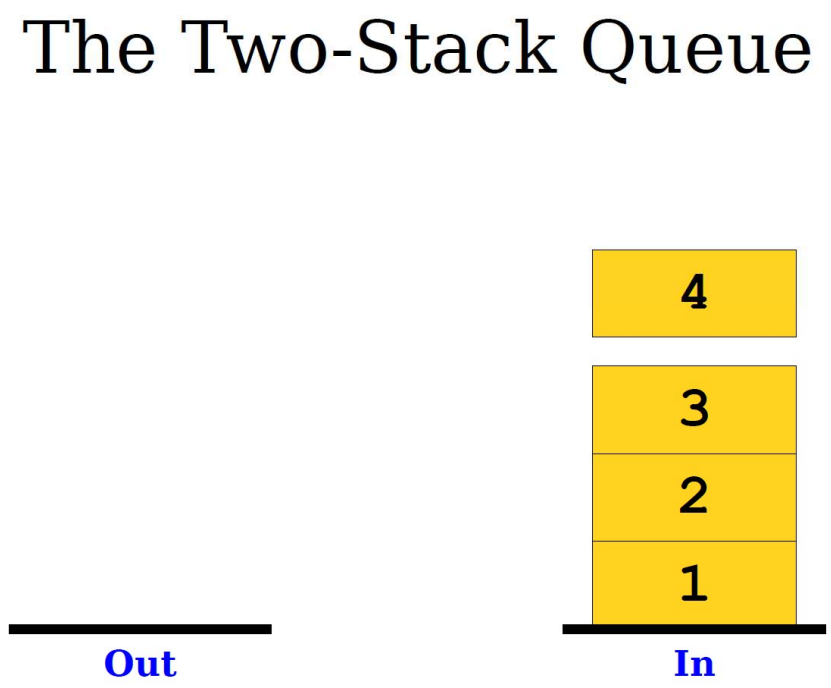
在右边的stack执行Enqueue



4个Enqueue操作后,

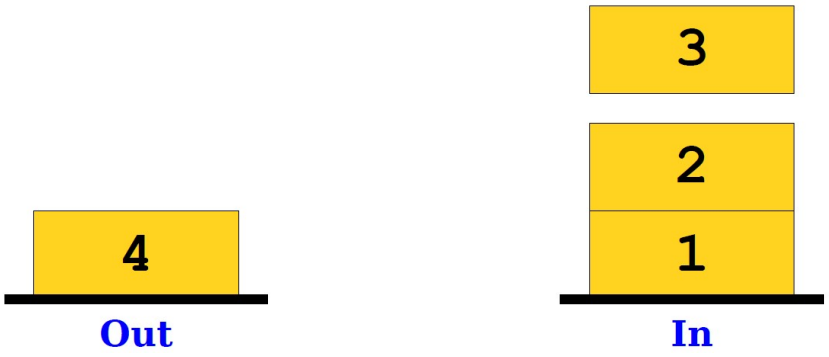


在执行Dequeue前，把右边的移到左边



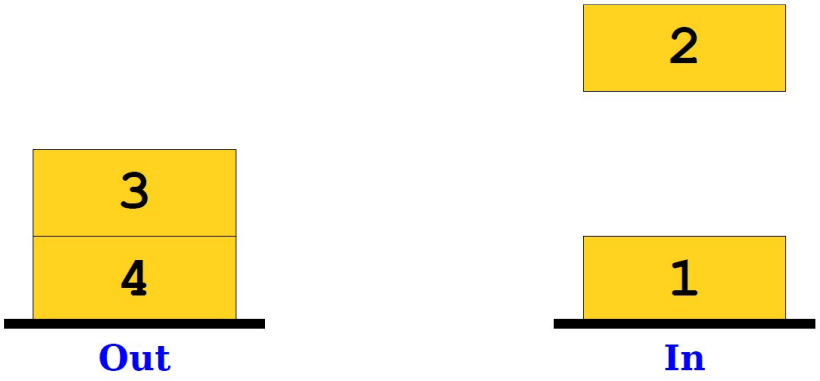
The Two-Stack Queue

The Two-Stack Queue

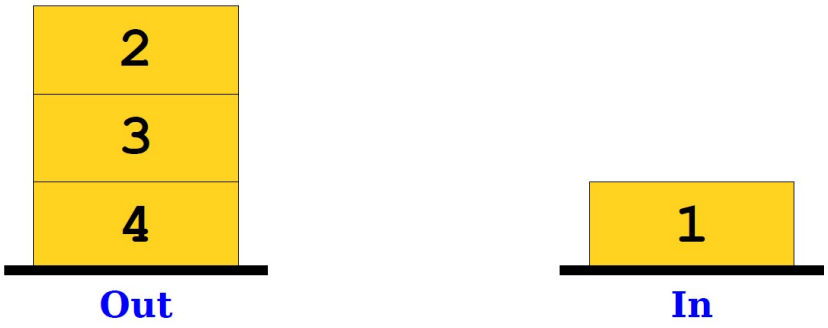


The Two-Stack Queue

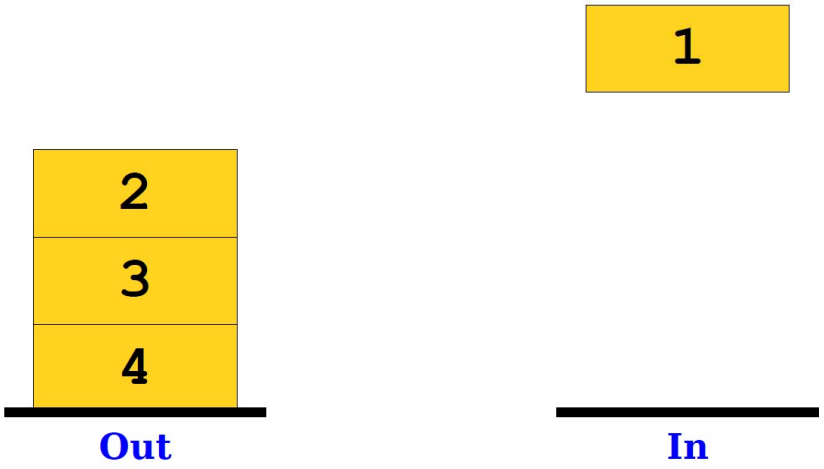
The Two-Stack Queue



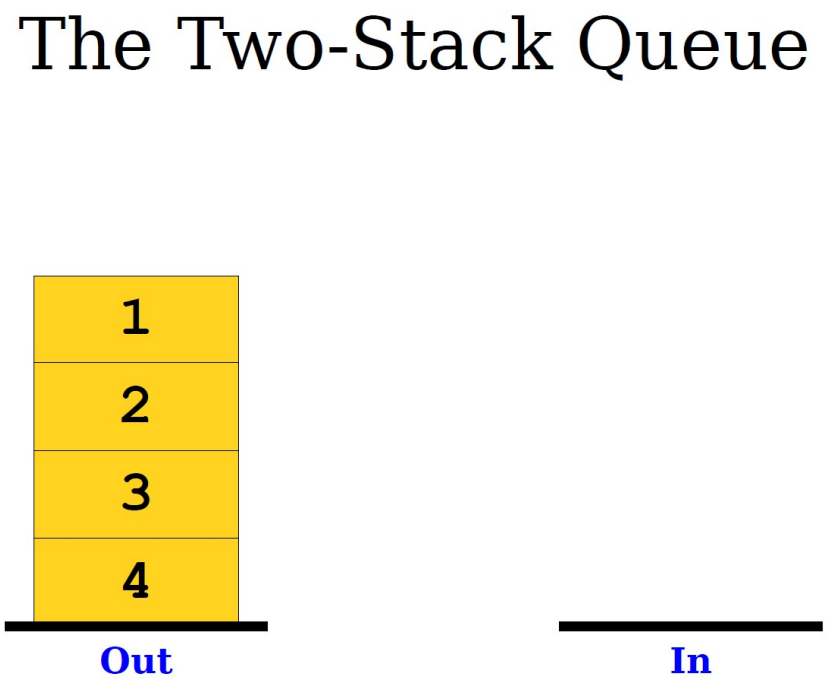
The Two-Stack Queue



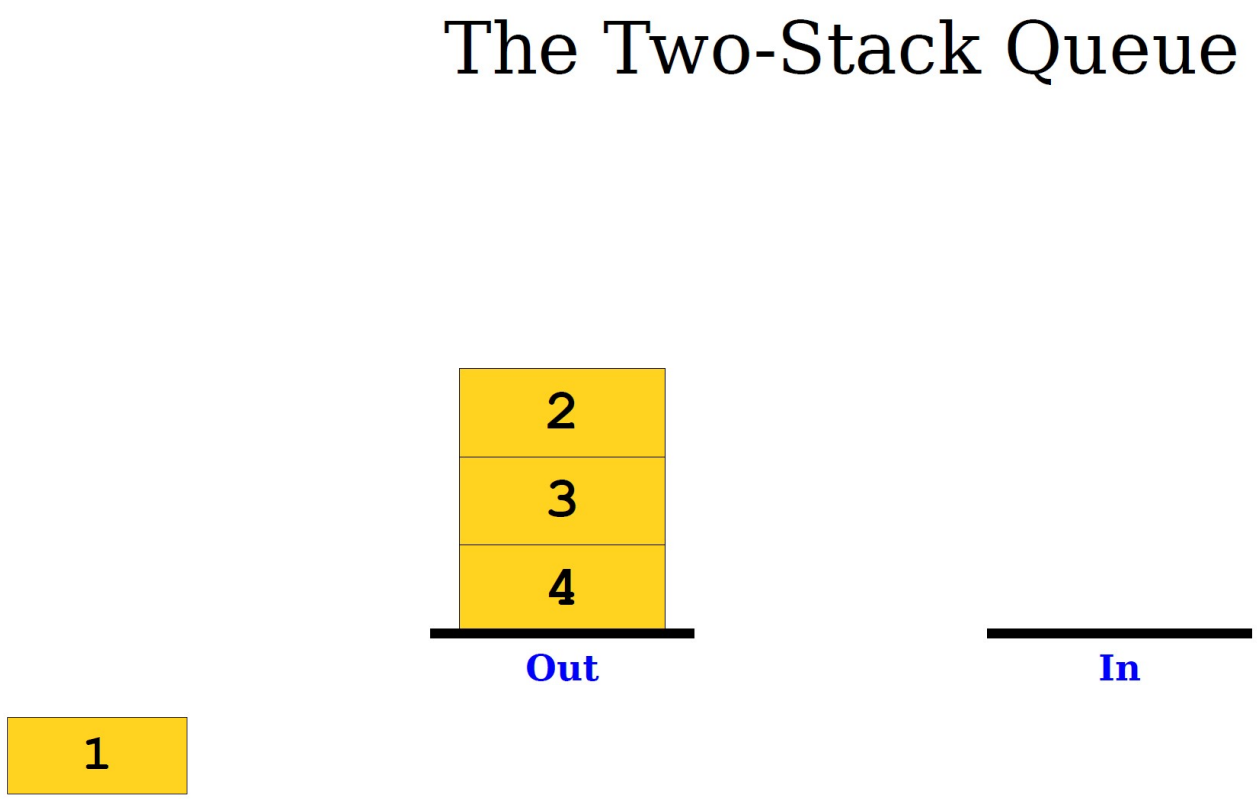
The Two-Stack Queue



直到右边的变空，

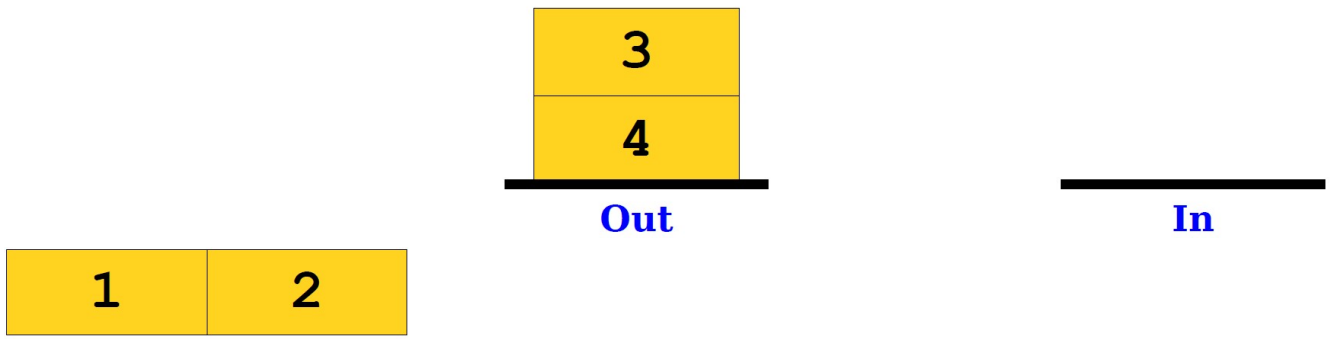


第一次Dequeue



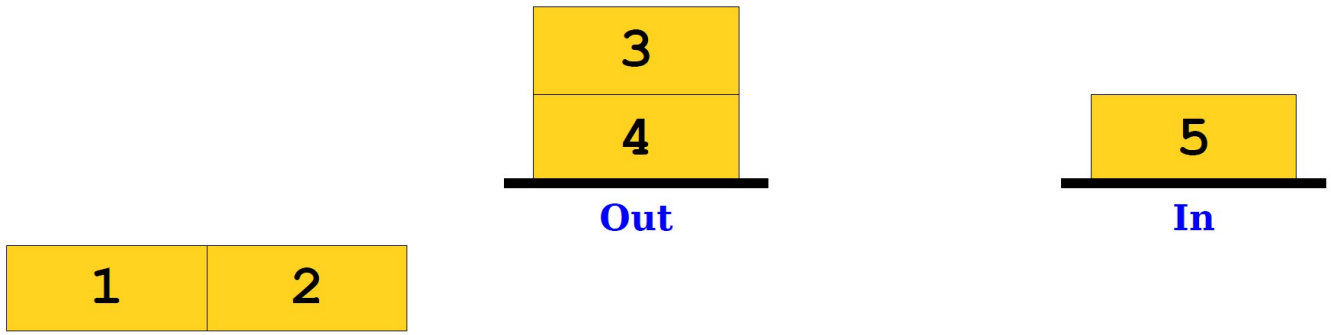
第二次Dequeue

The Two-Stack Queue



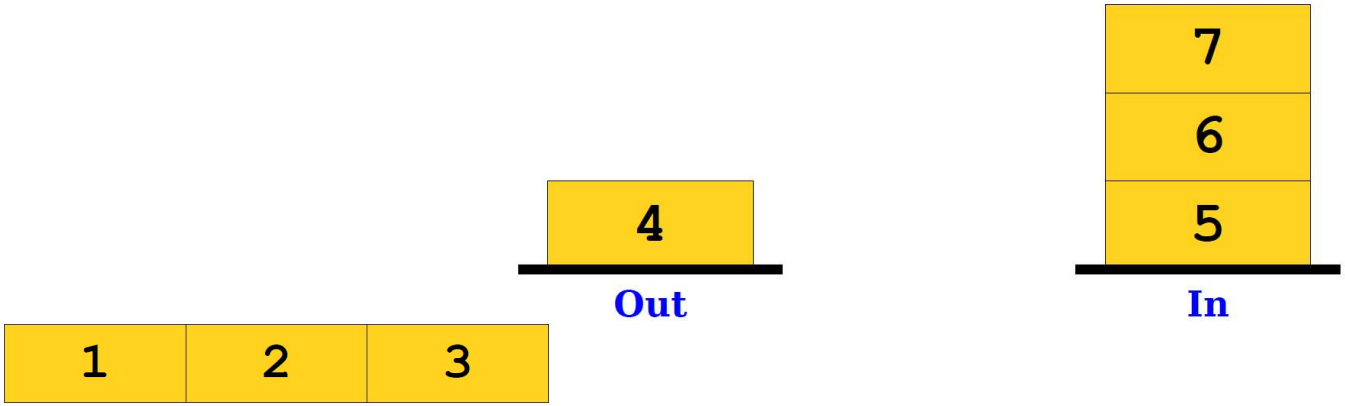
同时有一个Enqueue

The Two-Stack Queue



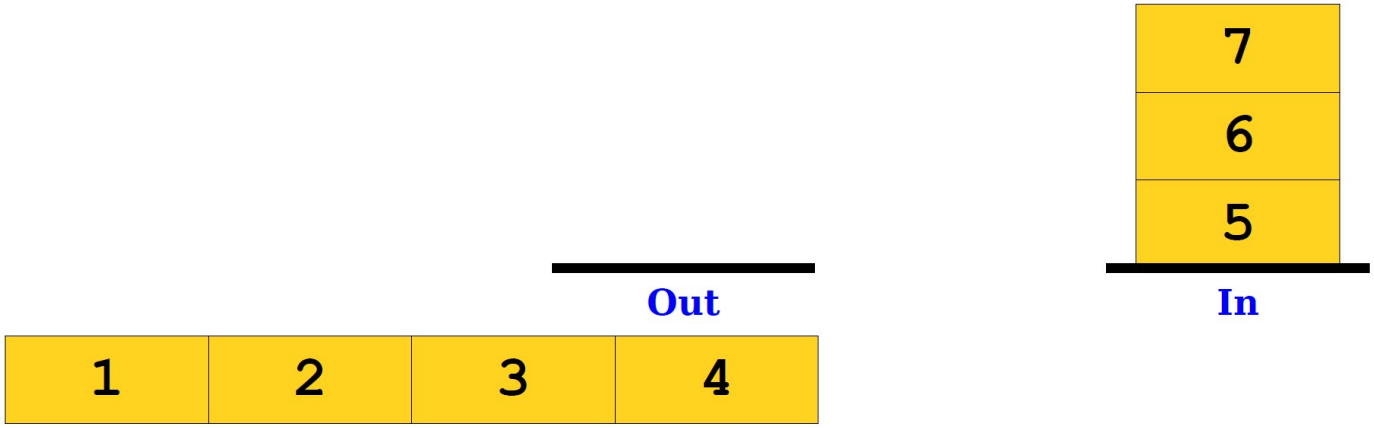
第三次Dequeue

The Two-Stack Queue



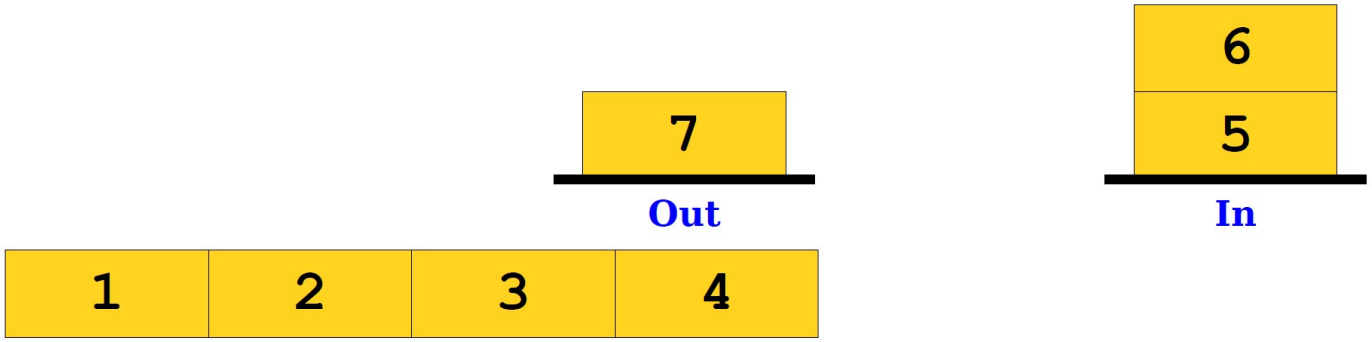
在左边变空前，还有两次Enqueue

The Two-Stack Queue



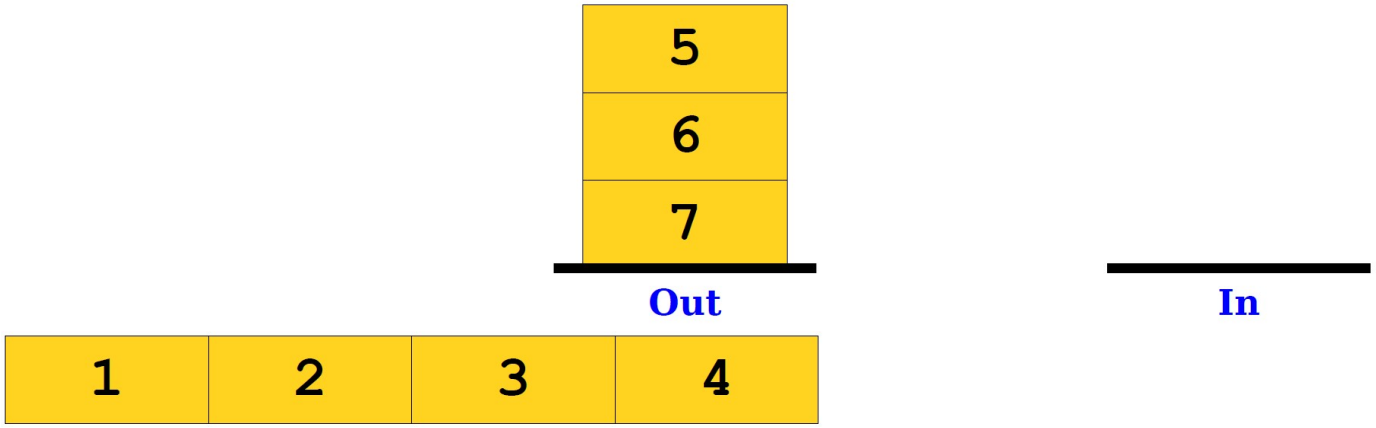
如果继续Dequeue，需要先倒腾

The Two-Stack Queue



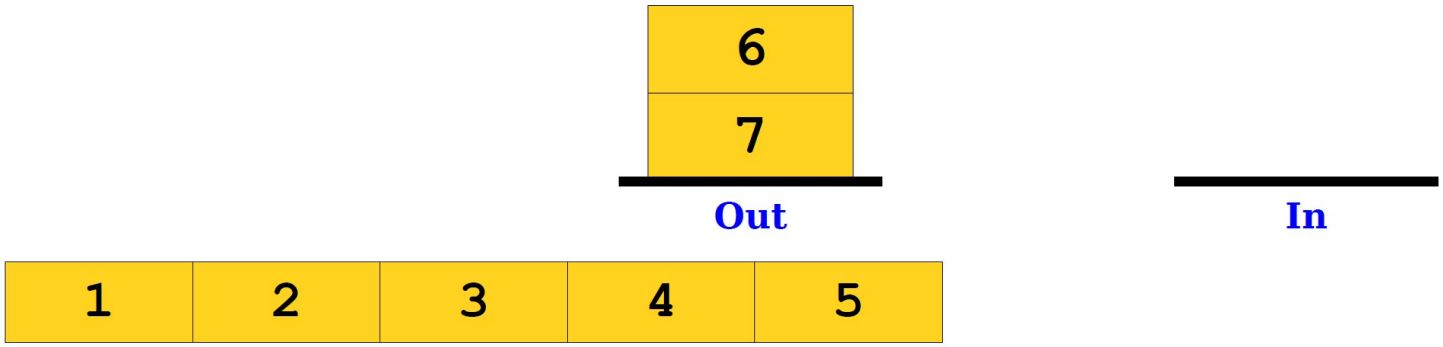
把右边的全部移到左边之后

The Two-Stack Queue



才能继续Dequeue

The Two-Stack Queue

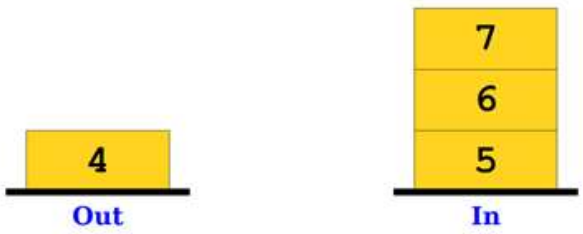


小结

- 需要建立两个栈，一个In，一个Out
- 执行Enqueue操作时，直接放入In栈
- 执行Dequeue操作时，
  - 如果Out栈是空的，把In栈里的全部移入Out栈
  - 然后把Out栈的顶部元素Pop后输出

- 时间复杂度？

实现




```
class IntQueue
{
public:
    IntQueue(){}

    inline bool IsEmpty() {return _in.IsEmpty() && _out.IsEmpty();}

    int Peek()
    {
        if (!_out.IsEmpty())
        {
            return _out.Peek();
        }

        assert(!_in.IsEmpty());

        while (!_in.IsEmpty())
        {
            _out.Push(_in.Pop());
        }

        return _out.Peek();
    }

    void Enqueue(int x)
    {
        _in.Push(x);
    }

    int Dequeue()
    {
        Peek();
        return _out.Pop();
    }

 private:
    IntStack _in;
    IntStack _out;
};
```

作业

- 用std::queue实现IntStack类

- 使用std::queue<int>
- #include<queue>
  - empty
  - size
  - front
  - back
  - push_back == Enqueue
  - pop_front == Dequeue

```
class IntStack
{
public:
    IntStack();

    bool IsEmpty();
    int Peek();
    void Push(int x);
    int Pop();

 private:
    //use std::queue<int>
};
```

Q&A

Thanks!