

栈Stack

by 波波微课 & William Fiset

1

大家好，欢迎回到波波微课。

栈Stack是一种非常知名的数据结构，
也是我最喜欢的数据结构之一。

本课会分成三部分，除了第一部分，在
第二部分和第三部分，我还会演示如何
基于链表Linked List来实现栈。

大纲

- 介绍栈Stack
 - 什么是栈？
 - 栈的使用场景有哪些？
 - 复杂度分析
 - 栈的使用样例
- 实现细节
 - 元素入栈
 - 元素出栈
- 代码实现

2

先来过一下本课的大纲。

首先我会来解释什么是栈？它有哪些使用场景。然后我会演示一些可以用栈来解决的比较有意思的问题，包括基本的语法检查问题，还有汉诺塔问题。

然后，我会来演示栈的内部是如何工作的，也会分析栈操作的时间复杂度。

最后，我会通过代码演示如何实现**Stack**抽象数据类型。

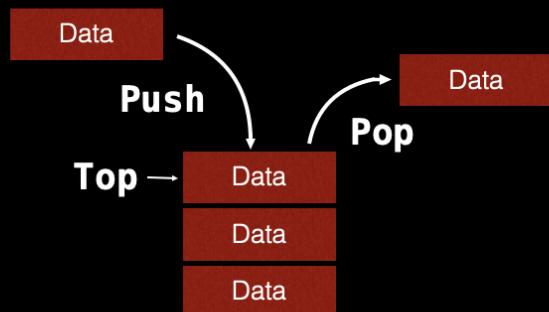
介绍栈

3

好，下面我来介绍栈。

什么是栈

栈是一种线性数据结构，它只支持在一端操作。
它支持的操作主要包括 **push** 和 **pop** .



那么什么是栈呢？

[读PPT]

PPT上给出了栈的一个示意图。其中有一个数据刚刚被弹出栈，另外一个数据准备要进入栈。需要注意，栈有一个 **Top** 指针，它指向栈顶的元素。元素的出栈和入栈，都只能通过栈顶来操作，这个行为也称为 **L-I-F-O**, **Last In First Out**，翻译成中文就是后进先出，最后

进来的最先出去。

下面我们来看一个具体的出栈和入栈的例子。

什么是栈？

指令

```
pop()  
push('洋葱')  
push('芹菜')  
push('西瓜')  
pop()  
pop()  
push('莴苣')
```

苹果
土豆
白菜
大蒜

5

请看PPT，右边有一个栈，里头有一些元素[读PPT]，然后左边是一些栈操作指令，下面我们来模拟执行这些指令，看看栈是如何工作的。

什么是栈？

指令

```
→ pop()
push('洋葱')
push('芹菜')
push('西瓜')
pop()
pop()
push('莴苣')
```



6

第一个指令是pop，所以我们移除栈顶的一个元素。

什么是栈？

指令

```
→ pop()
push('洋葱')
push('芹菜')
push('西瓜')
pop()
pop()
push('莴苣')
```



7

第一个元素苹果出栈。

什么是栈？

指令

```
→ pop()
push('洋葱')
push('芹菜')
push('西瓜')
pop()
pop()
push('莴苣')
```



8

现在栈中剩下三个元素。

什么是栈？

指令

```
pop()  
→ push('洋葱')  
push('芹菜')  
push('西瓜')  
pop()  
pop()  
push('莴苣')
```



9

下一个指令是将元素洋葱入栈， 所以我们将洋葱推到栈顶。

什么是栈？

指令

```
pop()  
→ push('洋葱')  
push('芹菜')  
push('西瓜')  
pop()  
pop()  
push('莴苣')
```



10

现在栈又变成四个元素， 洋葱在栈顶。

什么是栈？

指令

```
pop()  
push('洋葱')  
→ push('芹菜')  
push('西瓜')  
pop()  
pop()  
push('莴苣')
```



11

下一个指令是将芹菜推到栈顶。

什么是栈？

指令

```
pop()  
push('洋葱')  
→ push('芹菜')  
push('西瓜')  
pop()  
pop()  
push('莴苣')
```



12

现在栈里头有5个元素， 芹菜在栈顶。

什么是栈？

指令

```
pop()  
push('洋葱')  
push('芹菜')  
→ push('西瓜')  
pop()  
pop()  
push('莴苣')
```



13

下一个指令再将西瓜推到栈顶，也就是在芹菜之上。

什么是栈？

指令

```
pop()  
push('洋葱')  
push('芹菜')  
→ push('西瓜')  
pop()  
pop()  
push('莴苣')
```



14

现在西瓜在栈顶，栈里头总共有6个元素。

什么是栈

指令

```
pop()  
push('洋葱')  
push('芹菜')  
push('西瓜')  
→pop()  
pop()  
push('莴苣')
```

西瓜
芹菜
洋葱
土豆
白菜
大蒜

15

下一个指令是一个**pop**操作，所以我们要将栈顶的元素移除，这个元素就是我们刚刚添加的西瓜。

什么是栈？

指令

```
pop()  
push('洋葱')  
push('芹菜')  
push('西瓜')  
→pop()  
pop()  
push('莴苣')
```



16

所以，我们将刚刚入栈的西瓜再推出栈。

什么是栈？

指令

```
pop()  
push('洋葱')  
push('芹菜')  
push('西瓜')  
→pop()  
pop()  
push('莴苣')
```

芹菜
洋葱
土豆
白菜
大蒜

17

西瓜已经出栈了

什么是栈？

指令

```
pop()  
push('洋葱')  
push('芹菜')  
push('西瓜')  
pop()  
→pop()  
push('莴苣')
```



18

下一个指令还是一个pop，所以我们要将目前在栈顶的元素，也就是芹菜出栈。

什么是栈？

指令

```
pop()  
push('洋葱')  
push('芹菜')  
push('西瓜')  
pop()  
→pop()  
push('莴苣')
```



19

将芹菜出栈。

什么是栈？

指令

```
pop()  
push('洋葱')  
push('芹菜')  
push('西瓜')  
pop()  
→pop()  
push('莴苣')
```

洋葱
土豆
白菜
大蒜

20

现在栈里头剩下四个元素。

什么是栈？

指令

```
pop()  
push('洋葱')  
push('芹菜')  
push('西瓜')  
pop()  
pop()  
→ push('莴苣')
```



21

最后一个操作是将莴苣推到栈顶，所以我们将莴苣入栈。

什么是栈？

指令

```
pop()  
push('洋葱')  
push('芹菜')  
push('西瓜')  
pop()  
pop()  
→ push('莴苣')
```



22

现在栈里头有五个元素， 莴苣在栈顶。

栈有哪些使用场景？

- 实现文本编辑器的相反(undo)机制。
- 在编译器语法检查中匹配括号。
- 可以建模一堆书或者碟子。
- 支持嵌套或者递归的底层实现，栈可以保存和跟踪之前的函数调用链。
- 可用于对图进行深度优先(DFS)搜索。

23

既然我们对栈有了基本的理解，下面来看看栈有哪些使用场景？栈的使用场景其实是非常广泛的。

在文本编辑器中，栈可以用来实现相反(undo)机制，这样，你就可以用浏览器不断退回到上一个页面。

在编译器的语法检查中，栈可以用来检查左右括号是否匹配。

栈也可以用于建模现实世界中的书本，甚至是像汉诺塔一样的游戏。

栈也用于支持嵌套或者递归调用的底层实现，它可以保存和跟踪之前的函数调用链。当一个函数调用返回，它的栈帧就会被从栈顶上弹出，然后下一个栈顶对应的函数继续调用。你可能会有点吃惊，栈在编程语言中被大量使用，但是我们却从未察觉。

栈还有其它一些应用场景，比如栈可以用于对图进行深度优先搜索。深度优先搜索可以用手工维护栈的方式来实现，也可以使用递归来实现，当然，我们前面刚提到，递归底层其实也是基于栈实现的。

复杂度分析

24

下面我们来分析下栈操作的复杂度。

复杂性

入栈Pushing	$O(1)$
出栈Popping	$O(1)$
查看Peeking	$O(1)$
查找	$O(n)$
Size	$O(1)$

25

PPT上有一个关于栈操作的复杂度表，这里我们假定栈是基于链表实现的。

入栈Pushing是常量级的，因为我们对栈顶有引用。同样，出栈Popping和查看栈顶元素Peeking也都是常量级的。

但是查找是线性级的。我们要找的元素未必在栈顶，它可能在栈底，也可能不存在，也就是说我们必须查找栈中的所

有元素。

获取栈的大小是常量级的操作，因为我们
可以始终维护记录栈大小的一个计数器。
元素入栈的时候，我们将计数器 $+1$ ，元素
出栈的时候，我们将计数器 -1 ，很简单。

例子 – 括号匹配问题

问题：给出一个由圆()方[]花{}等括号所组成的字符串，检查字符串中的括号是否匹配。

[{}]	→	合法
((()))	→	合法
{}	→	非法
[(())())	→	非法
[]{}({})	→	合法

26

下面我们来看一个可以用栈来解决的具体问题。

[读问题]

比方说，[演示ppt]

在我给出答案之前，你不妨先自己思考，如何用栈来解决这个问题。

例子 - 括号匹配问题

括号序列：

[[{ }] ()]

当前括号： \emptyset

反括号： \emptyset

27

考虑下PPT上的这个括号序列，下面我来演示如何用栈来解决这个问题。随着我们从左到右陆续处理字符串，我会在PPT上显示当前处理的括号，也显示和它对应的反括号。好，我们开始。

例子 - 括号匹配问题

括号序列：

[[{ }] ()]

当前括号： [

反括号：]

[

28

每次碰到一个左括号，我们就将它入栈。
因为括号序列中的第一个括号(蓝色标示的)是左方括号，所以我们将它入栈。

例子 – 括号匹配问题

括号序列：

[[{}]()]

当前括号： [

反括号：]

[
[

29

对下一个左方括号，做同样的操作。

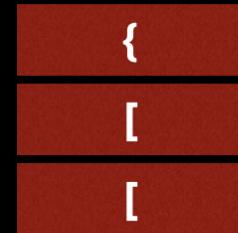
例子 – 括号匹配问题

括号序列：

[[{ }] ()]

当前括号： {

反括号： }



30

下面碰到一个左花括号，入栈。

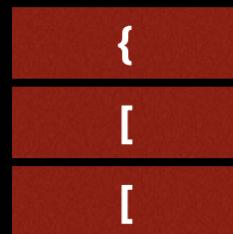
例子 - 括号匹配问题

括号序列：

[[{}]()]

当前括号： }

反括号： {



31

好的，下一个是最右花括号。每次我们遇到右括号，我们要做两个检查。首先我们要检查当前栈是否是空，如果是空，那么括号序列就是合法的。但是如果栈不空，那么我们就要将栈顶元素弹出，并且和当前括号的反括号去进行比对，看是否相等，如果相等，那么我们可以继续检查，如果不相等，那么我们可以判定输入序列是非法的。

现在我们发现，当前栈顶的元素和反括号是相等的，所以我们可以继续。

因为我们刚刚找到一堆匹配的括号，所以我们不需要入栈，而是要将栈顶元素出栈。

例子 - 括号匹配问题

括号序列：

[[{ }] ()]

当前括号： }

反括号： {



32

将栈顶的左花括号出栈，现在栈里头剩两个待匹配的元素。

例子 - 括号匹配问题

括号序列：

[[{}]()]

当前括号：]

反括号： [



33

下一个字符是右方括号，我们先看栈是否为空？当前栈不空，我们继续。再看栈顶元素和反括号是否相等？我们看到是相等的，所以继续。

例子 - 括号匹配问题

括号序列：

[[{}]()]

当前括号：]

反括号： [

[

34

因为我们刚刚找到一对匹配括号，所以我们将栈顶元素出栈。现在栈里头只剩一个待匹配的元素。

例子 – 括号匹配问题

括号序列：

[[{ }] ()]

当前括号： (

反括号：)

(

[

下一个字符是左圆括号，入栈。

例子 - 括号匹配问题

括号序列：

[[{ }] ()]

当前括号：)

(

反括号： (

[

下一个是最外层的右圆括号，这个时候栈空吗？
不空，我们继续。栈顶元素和反括号相等吗？是的，我们继续。

例子 – 括号匹配问题

括号序列：

[[{}]()]

当前括号：)

反括号： (

[

37

因为刚刚找到一对匹配括号，所以将栈顶元素出栈。现在栈里头剩一个待匹配括号。

例子 - 括号匹配问题

括号序列：

[[{}]()]

当前括号：]

反括号： [

[

38

再发现一个右方括号，栈空吗？不空，我们继续。栈顶元素和反括号相等吗？是的，我们继续。

因为我们刚刚找到一对匹配括号，所以我们要将栈顶元素出栈。

例子 - 括号匹配问题

括号序列：

[[{}]()]

当前括号：]

反括号： [

39

现在我们的栈为空，我们的输入括号序列也已经全部检查完，没有下一个括号了。

例子 – 括号匹配问题

括号序列：

[[{}]()] → 合法

当前括号：]

反括号： [

40

这种情况下，我们可以判定输入括号序列是合法的。

例子 – 括号匹配问题

再看一个例子：

[{}) []

当前括号： \emptyset

反括号： \emptyset

41

好的， 下面我们再看另外一个括号序列。

例子 - 括号匹配问题

括号序列：

[{}) []

当前括号： [

反括号：]

[

42

第一个括号是左方括号，我们将它入栈。

例子 – 括号匹配问题

括号序列：

[{ }) []

当前括号： {

{

反括号： }

[

43

第二个左花括号，入栈。

例子 - 括号匹配问题

括号序列：

[{ }) []

当前括号： }

{

反括号： {

[

44

下一个字符是右花括号，我们先判断栈是否为空？不空。再看栈顶元素是否等于反括号？相等，我们继续。

例子 - 括号匹配问题

括号序列：

[{ }) []

当前括号： }

反括号： {

[

45

因为刚刚找到一对匹配括号，所以将栈顶元素出栈。现在栈顶只有一个左方括号。

例子 - 括号匹配问题

括号序列：

[{})([]

当前括号：)

反括号： (

[

46

下一个括号是右圆括号， 栈空吗？ 不空，
我们继续判断。

例子 – 括号匹配问题

括号序列：

[{})([] → 非法

当前括号：)

反括号： (

[

47

当前栈顶的括号和反括号相等吗？不相等，所以这个输入括号序列是不合法的。

括号匹配算法伪代码

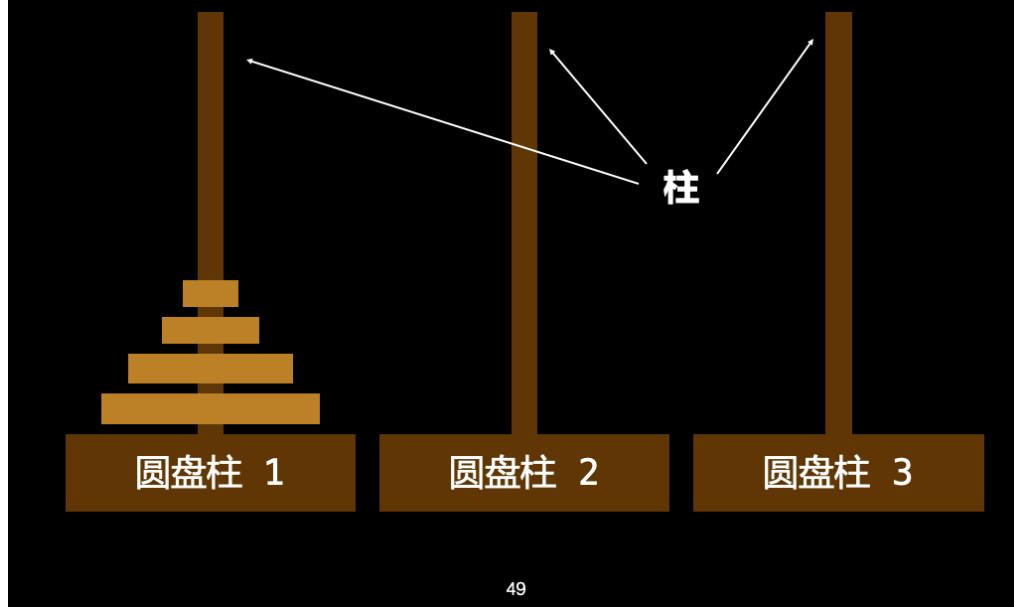
假定S是一个栈

```
For bracket in bracket_string:  
    rev = getReversedBracket(bracket)  
  
    If isLeftBracket(bracket):  
        S.push(bracket)  
  
    Else If S.isEmpty() or S.pop() != rev:  
        return false // 非法  
  
return S.isEmpty() // 如果S为空则合法
```

48

PPT上给出了括号匹配问题的算法伪代码，下面我们来过一下这个算法。

汉诺塔游戏



49

下面我们来看一个叫汉诺塔的游戏，这是一个非常著名的游戏，不少数学家和计算机科学家也玩这个游戏。我们来看看这个游戏和栈有什么关系。

这个游戏是这样玩的：

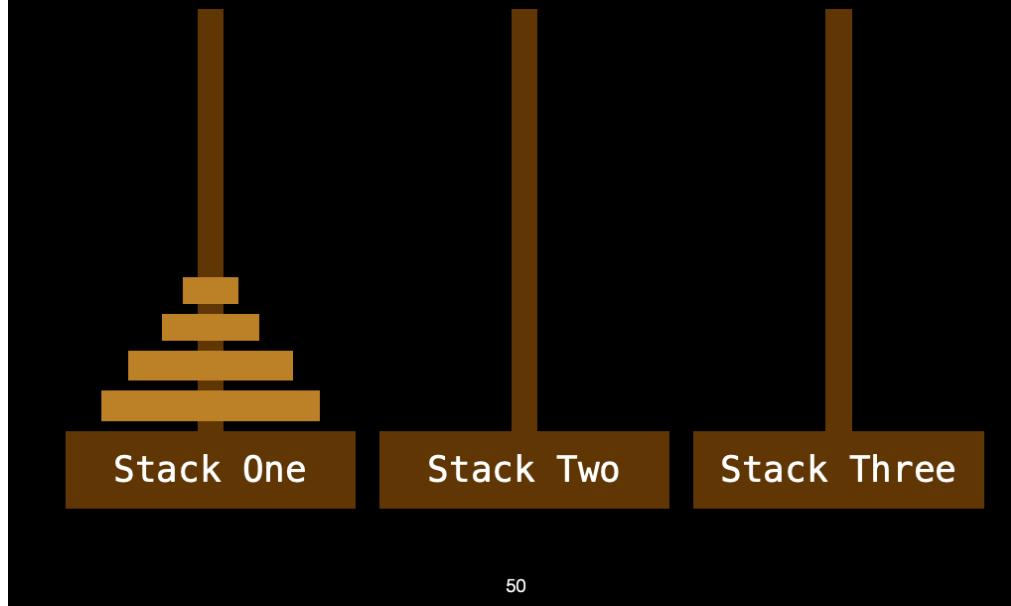
刚开始圆盘都在左边的圆盘柱1上面，游戏的目标是要将所有圆盘移到右边的圆盘柱3上面去。每一步，你只能移动

一个圆盘，你可以借助中间或者两边的圆盘柱，但是不允许出现将大圆盘放在小圆盘上的情况。

传说，真正的汉诺塔有64个圆盘，如果将所有64个圆盘，按照规则都移动到圆盘柱3上面去，它所需的时间非常庞大，基本上到那个时候，整个世界都已经毁灭了。

我们可以把每一个圆盘柱看作是一个栈，因为每次我们只能移动栈顶的圆盘，被移动的圆盘也只能放在其它圆盘柱的栈顶。

汉诺塔游戏

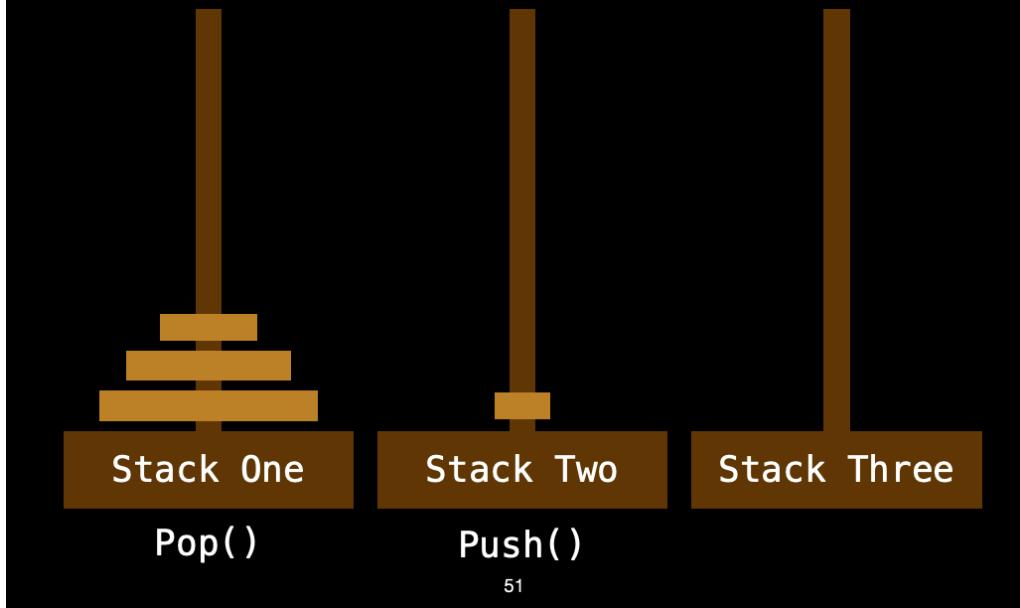


50

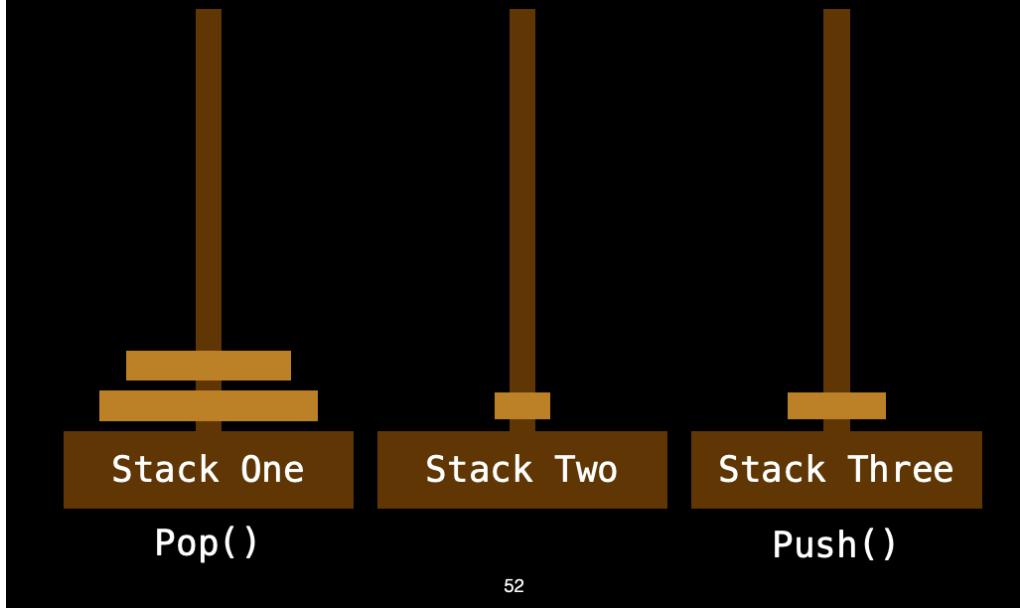
如果你对这个游戏感兴趣，那么不妨暂停视频，先自己尝试玩一下。

好，这边会直接以动画方式演示玩这个游戏的过程，从动画中，你可以看到每一个圆盘柱都类似一个栈。这个动画很酷，让我来开始。。。。

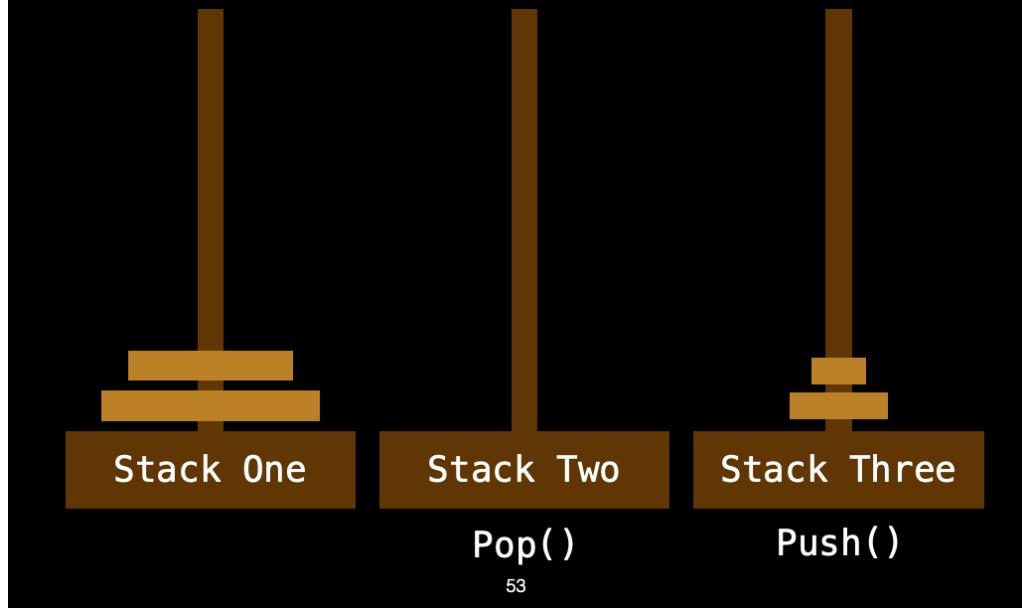
汉诺塔游戏



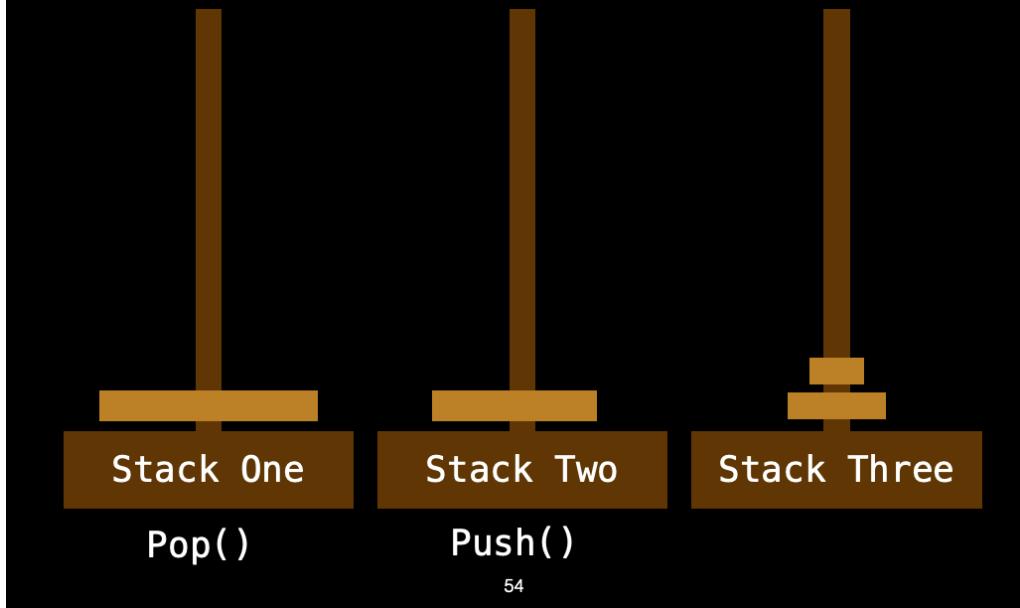
汉诺塔游戏



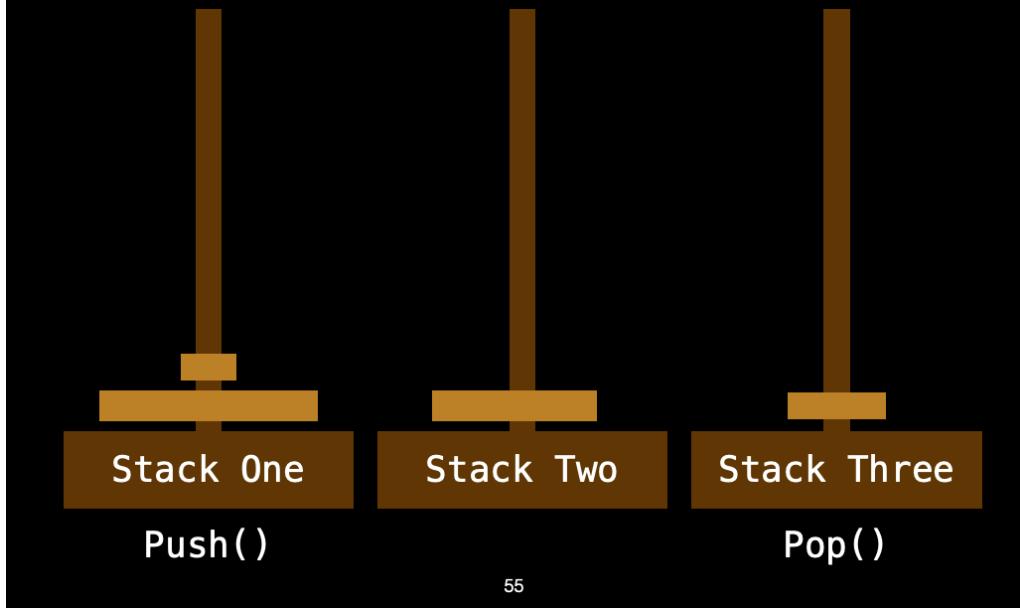
汉诺塔游戏



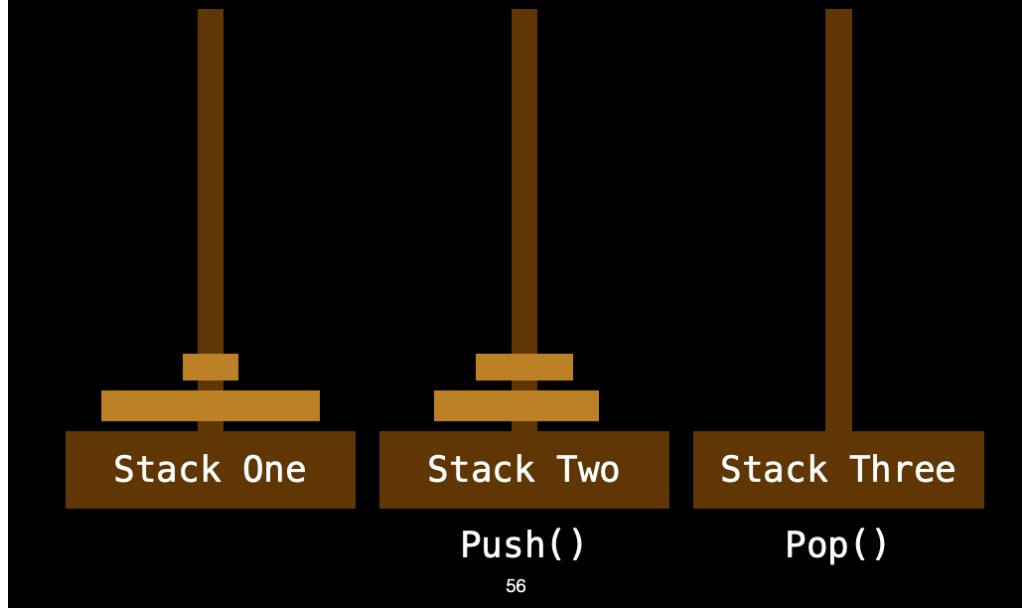
汉诺塔游戏



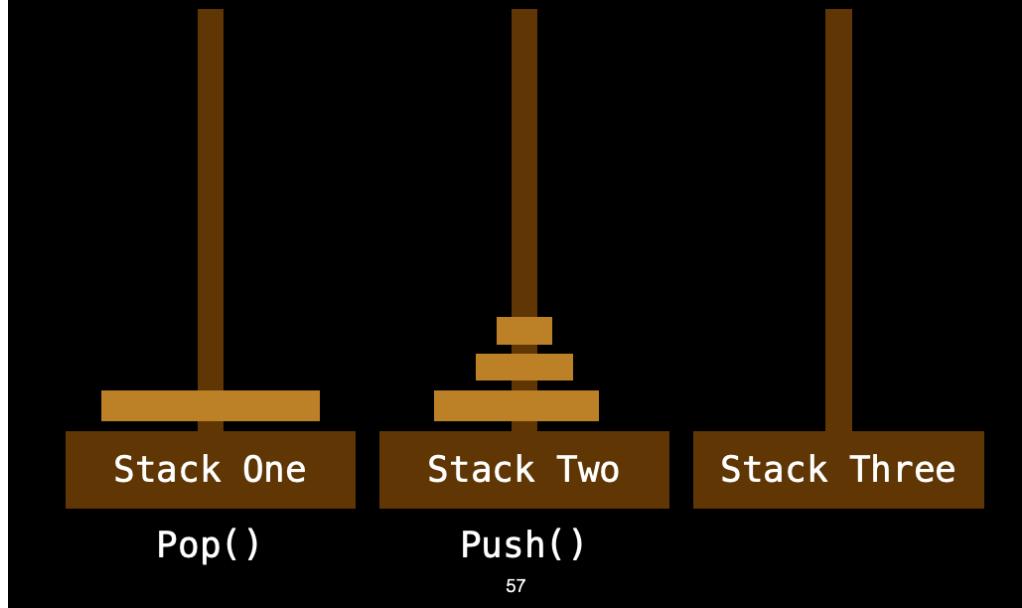
汉诺塔游戏



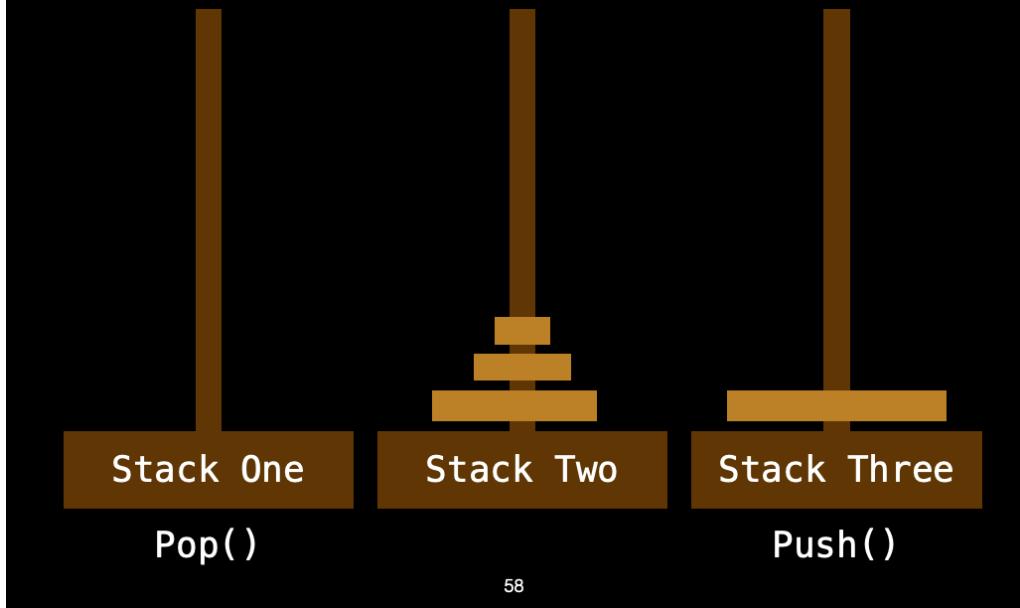
汉诺塔游戏



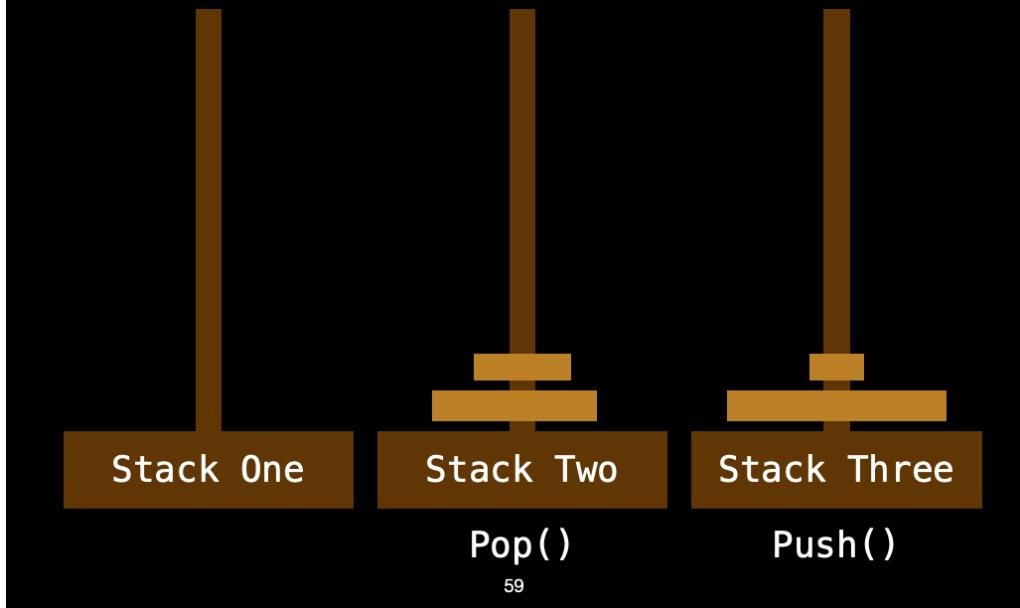
汉诺塔游戏



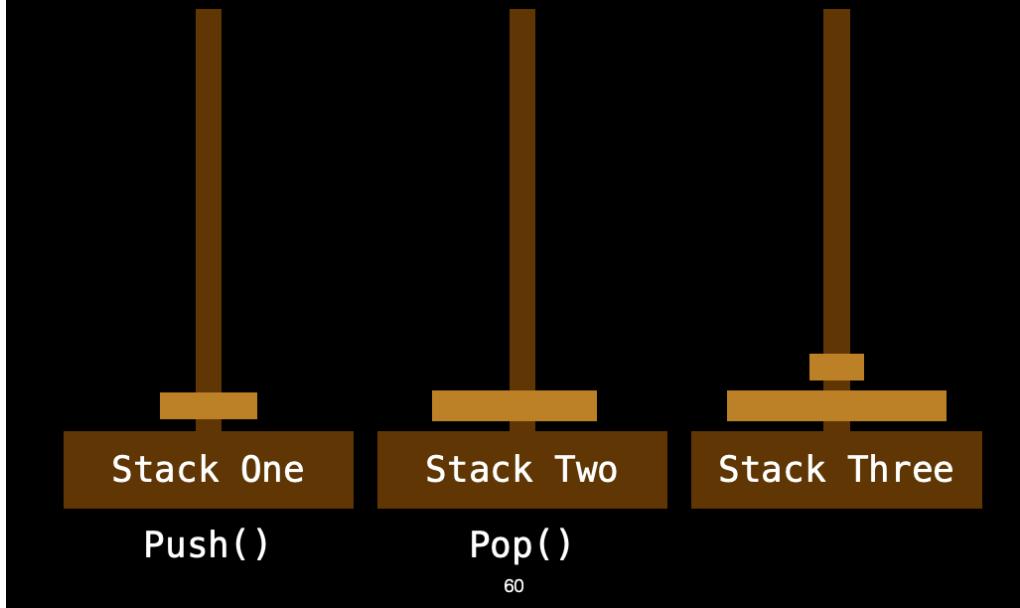
汉诺塔游戏



汉诺塔游戏

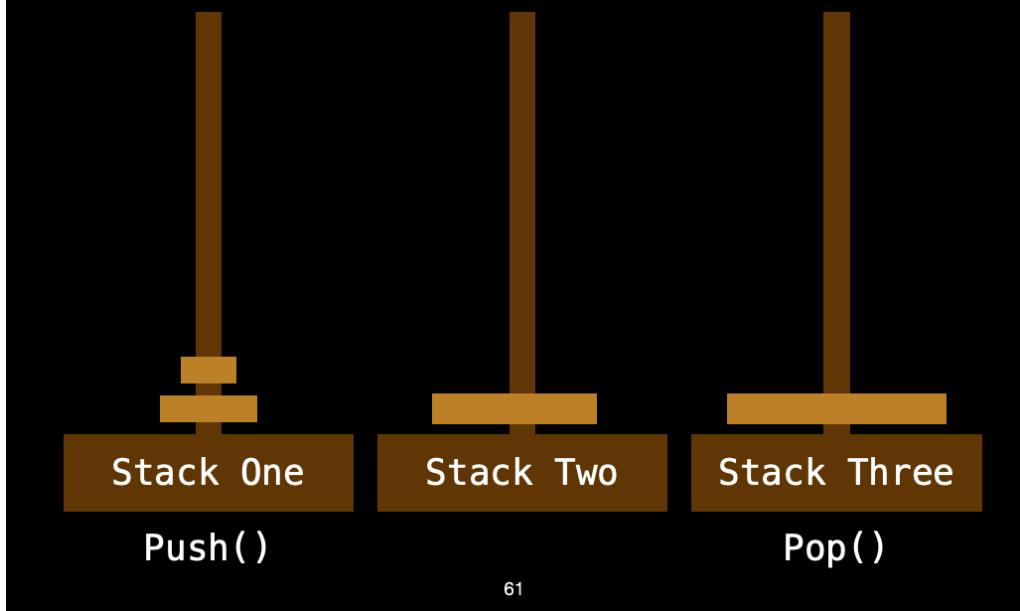


汉诺塔游戏

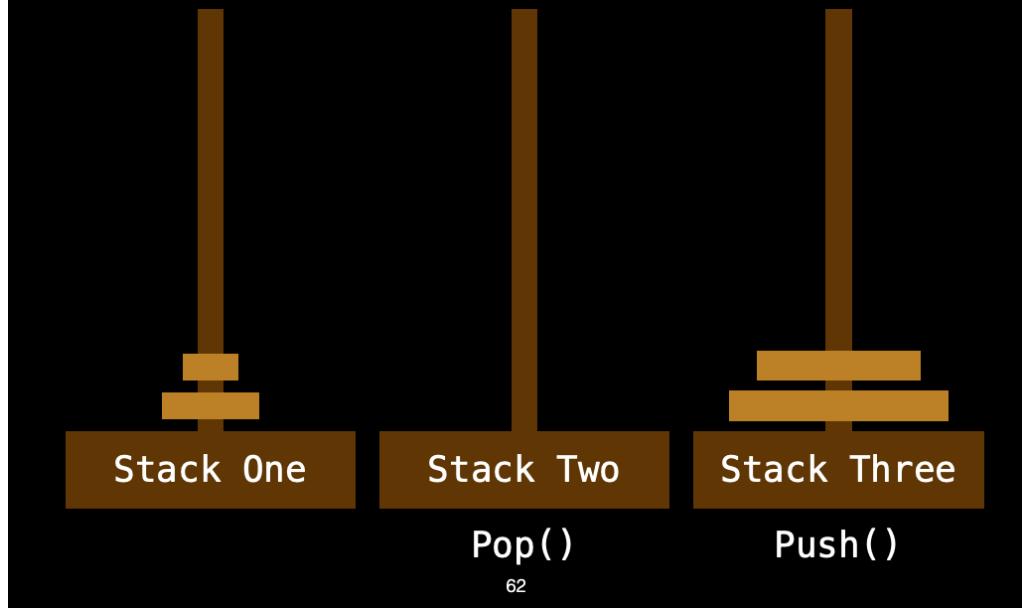


60

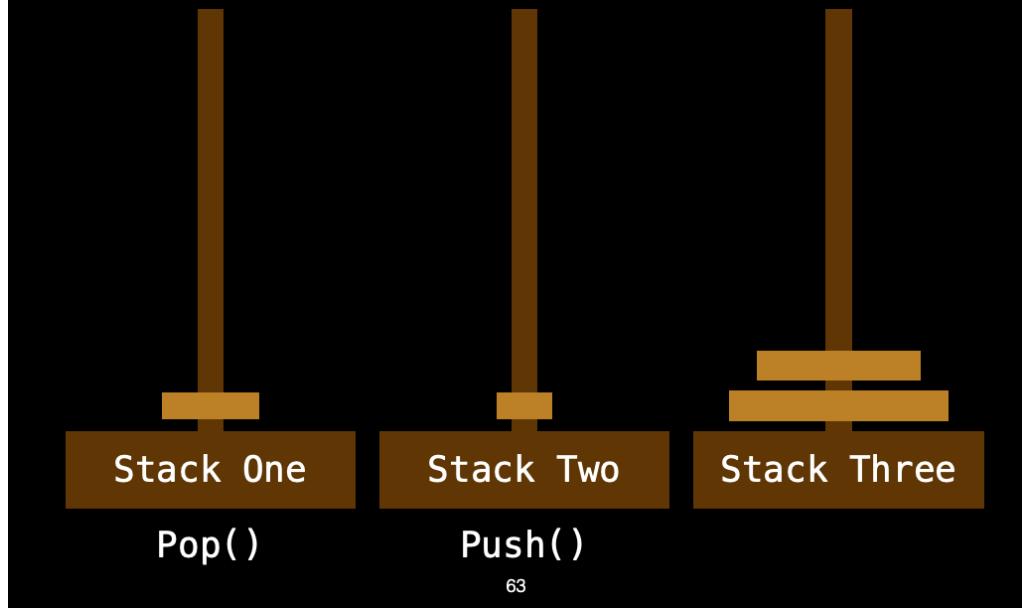
汉诺塔游戏



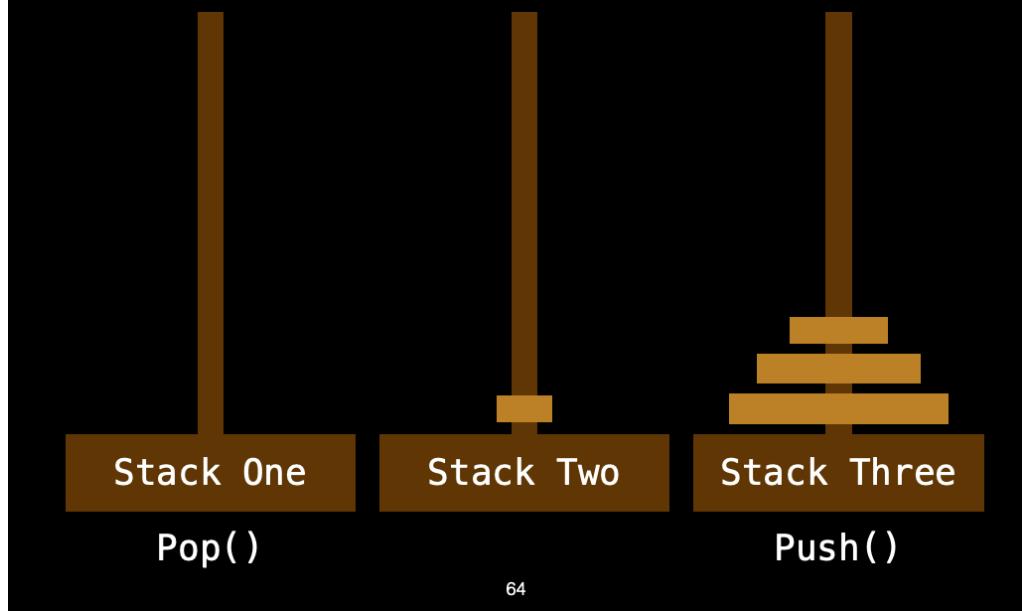
汉诺塔游戏



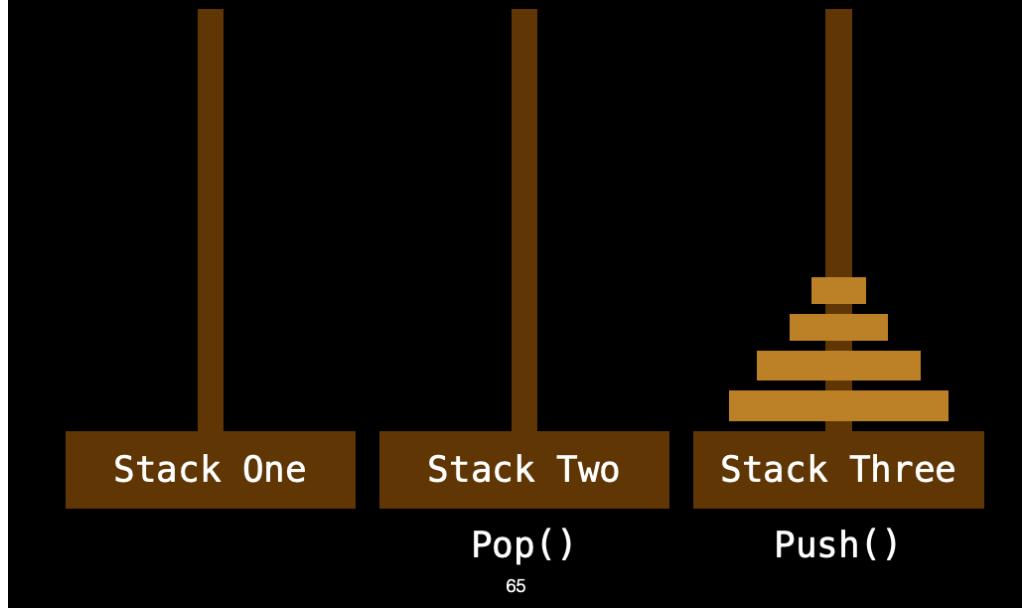
汉诺塔游戏



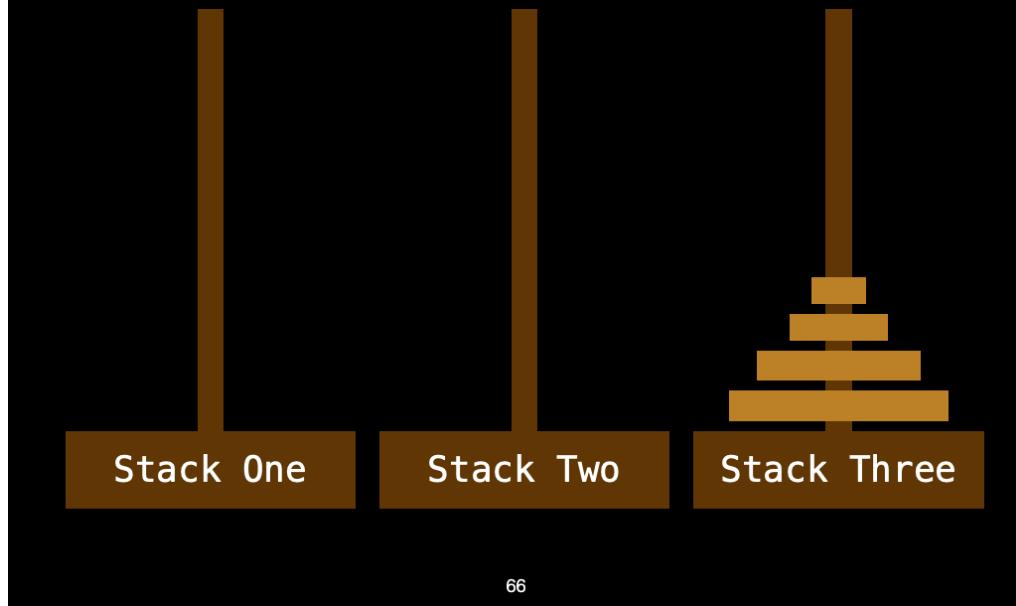
汉诺塔游戏



汉诺塔游戏



汉诺塔游戏



66

好的，游戏演示结束。

现在你已经看到，将圆盘从一个圆盘柱移动到另外一个圆盘柱，就像从一个栈的栈顶弹出，然后再进入另外一个栈的栈顶。限制条件是你不能将大的圆盘放在小的圆盘之上。

栈操作

by 波波微课 & William Fiset

67

欢迎回到波波微课，这一课是关于栈的第二次课，我会以PPT的方式演示栈是如何实现的。

Pushing

Instructions

```
Push(4)  
Push(2)  
Push(5)  
Push(13)
```

68

栈通常可以用数组、单向链表，或者甚至还可以用双向链表来实现。本节课我会以PPT方式演示如何用单向链表来实现栈。在下一节课，我会以代码方式来演示如何基于双向链表来实现栈。

Pushing

Instructions

```
Push(4)  
Push(2)  
Push(5)  
Push(13)
```



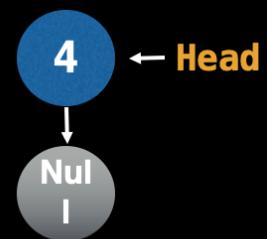
69

因为刚开始栈是空的，所以我们先创建一个空的头指针，也就是`head = null`。

Pushing

Instructions

→ Push(4)
Push(2)
Push(5)
Push(13)



70

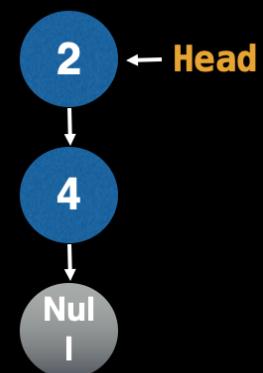
使用单向链表来创建栈，为了实现Push操作，比较巧妙的地方是，我们应该在头节点的前面插入新元素，而不是在链表的尾部。这样，我们的头指针始终指向栈顶元素，后面如果要弹出元素也没有问题，后面我也会演示弹出。

下一个要入栈的元素是2。

Pushing

Instructions

Push(4)
→ Push(2)
Push(5)
Push(13)



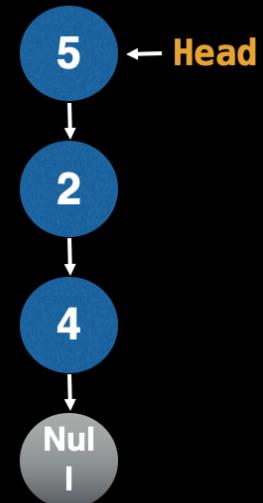
71

所以我们创建一个新节点，调整头指针指向最新的节点，并且将新的头节点的下一个指针指向原来的头节点。

Pushing

Instructions

Push(4)
Push(2)
→ Push(5)
Push(13)



72

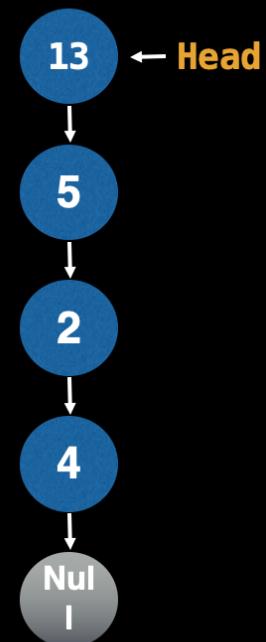
5和13两个元素的入栈也是一样的。

5入栈。

Pushing

Instructions

Push(4)
Push(2)
Push(5)
→ Push(13)



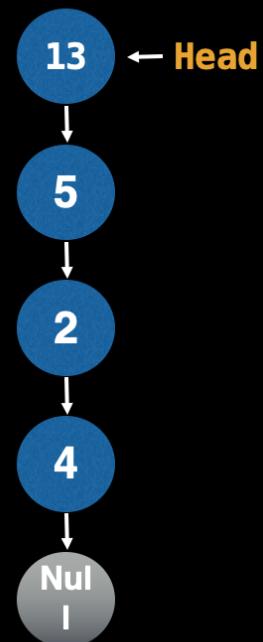
73

13入栈。

Popping

Instructions

```
Pop()  
Pop()  
Pop()  
Pop()
```



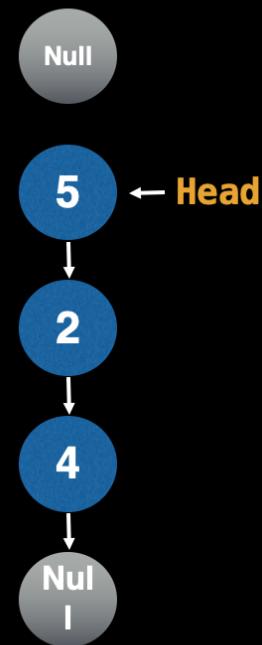
74

现在我们来演示如何弹出(poping)元素。
弹出元素也很简单，只需要将头指针移
动到下一个节点，并且释放原来的头节
点。

Popping

Instructions

→ Pop()
Pop()
Pop()
Pop()



75

现在，我们已经将第一个节点弹出栈了，并且将这个节点的数据和下一个节点引用都设置成null了。这样，如果是Java实现的话，这个节点就可以被垃圾回收了，因为已经没有其它节点再引用它了。

如果你采用像C或者C++这些需要显式释放内存的语言来实现，那么你就需要显式释放这个节点，以免发生内存泄漏。

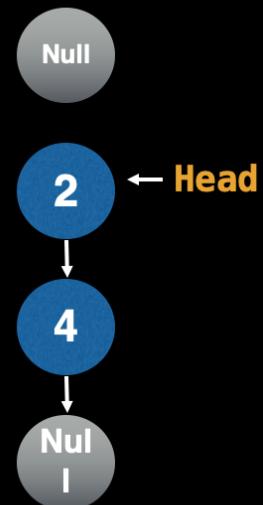
数据结构一般会被重用，所以在其中发生内存泄漏是很糟糕的事情。关于内存释放问题，你不仅在本次课要注意，而且在整个这门数据结构课程中都需要注意。

如果你发现我实现的代码中有潜在的内存泄漏问题，请务必告诉我，或者给我的github仓库发一个pull request。

Popping

Instructions

Pop()
→ Pop()
Pop()
Pop()



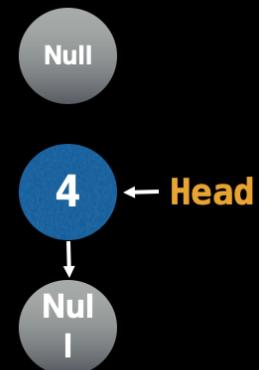
76

下面我们可以继续Pop弹出元素，也就是将头节点往下移动。

Popping

Instructions

Pop()
Pop()
→ Pop()
Pop()



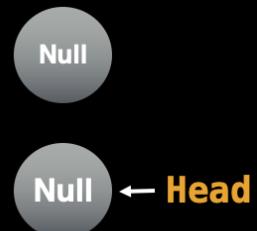
77

继续弹出。

Popping

Instructions

Pop()
Pop()
Pop()
→ Pop()



78

现在我们把最后一个元素也弹出了，栈变成了空。

Popping

Instructions

```
Pop()  
Pop()  
Pop()  
Pop()
```



79

好，本次课我们演示了如何基于单向链表来实现栈。下节课我会继续演示如何基于双向链表来实现栈，我们下节课再见。