abstracct data type (ADT) 是一种数据结构模型

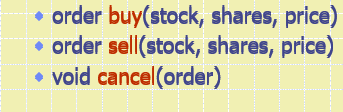
一个ADT是间接定义的，只有通过在他身上进行操作来定义它，一个ADT有

1. 所存储的数据
2. 对这些数据的操作
3. 与操作所联系的错误状况

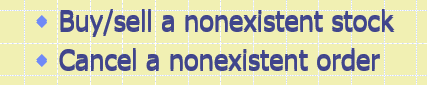
例如一个ADT来模拟简单的市场交易系统

所存储的数据：交易买卖订单

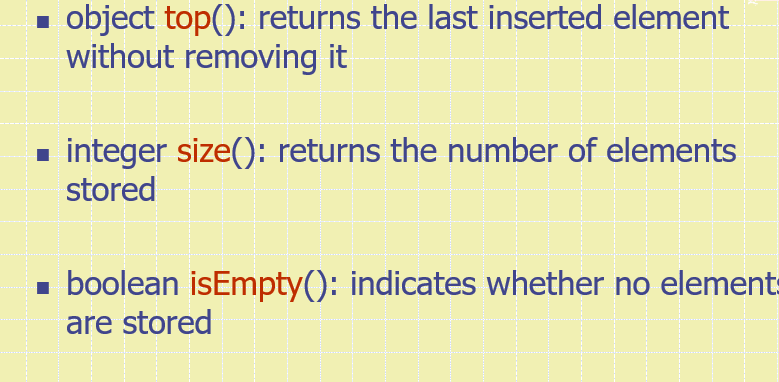
对这些数据的操作

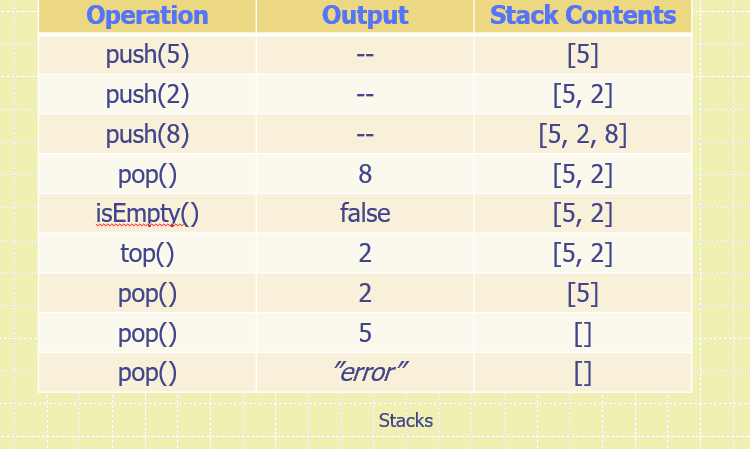


错误状况

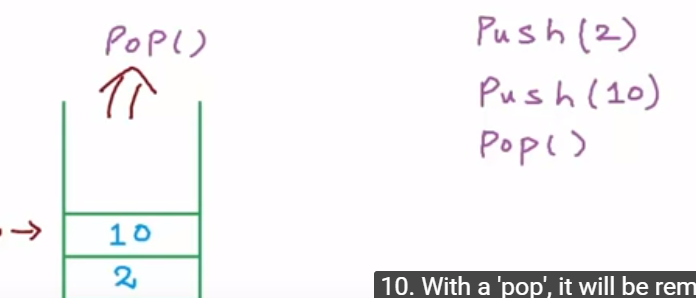


Stack的操作有

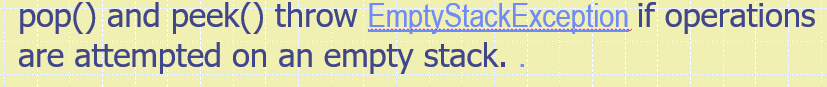




push输入，pop弹出，isEmpty确认是不是empty，top pop

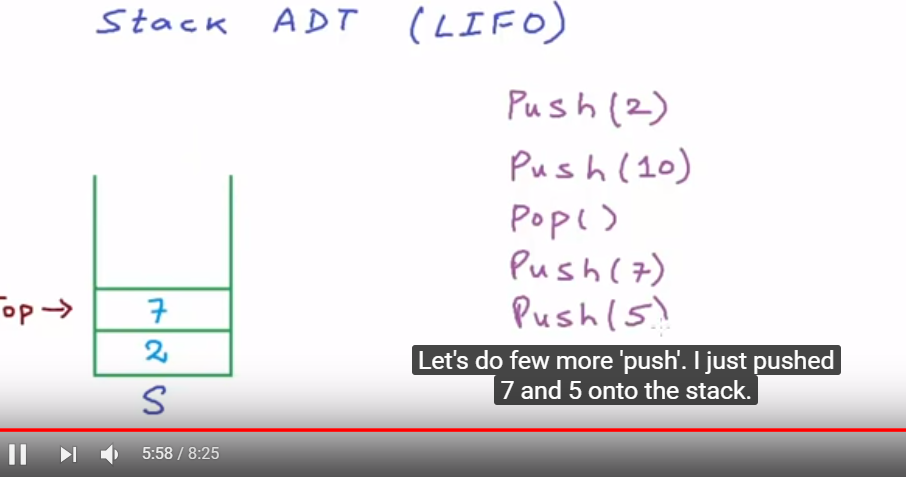


pop以后最顶上的10没了



peek（）就是java里的top()

对empty stack使用pop(), top() //peek（）会throw 一个exmpty stackexception



pop是没有参数的，他只能Pop最顶上的

Java有内置stack 

JVM java虚拟机中的Method Stack

jvm 用stack来追踪keep track那些被激活的method

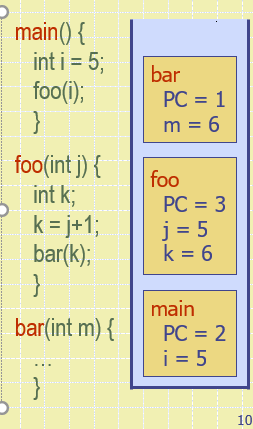
当一个method被call的时候，JVM把一个框架push到stack上，这个frame包括：

1.Local variable和return value

2.program counter程序计数器，来跟踪正在进行的句子

当一个method结束时，这个frame被pop出去，然后我们运行下一个stack顶端的Method

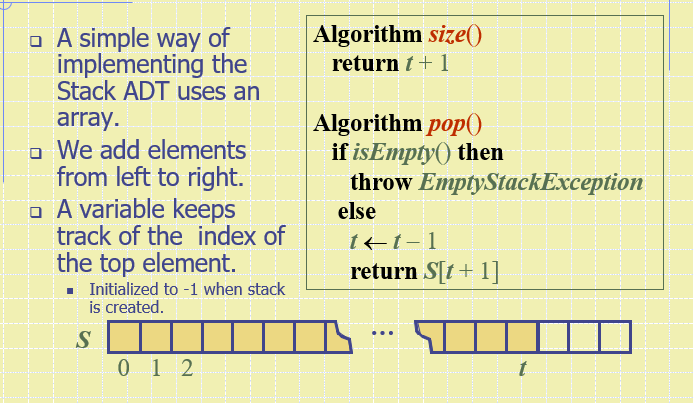
允许recursion

,PC是 program controller, 即保留所有的参数变量，当地变量，PC

建立在array上的stack

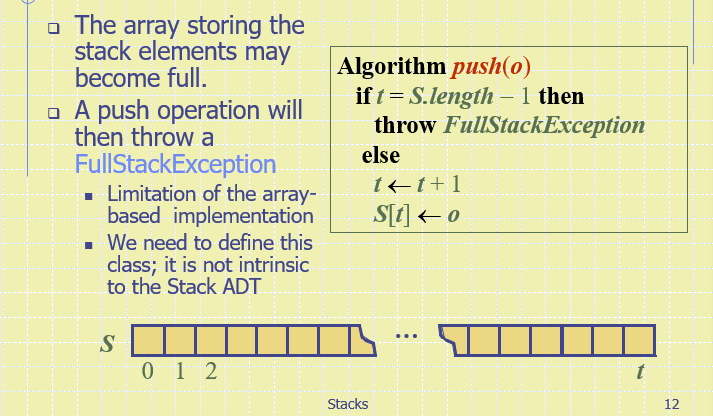
一个简单使用stack这个ADK的方法是用array

我们从左到右add element，并用一个变量来track最顶端的element(最右边的)



最右边的是t，pop如果empty,throw emptystackexception

不然就t小一位，然后returnS[t+1]



push,如果t=S.length-1就已经在最右边了，这时如果push，throw full stackException

不然t自动增1，（这时t是空白），S(t)输入o

注意点：一开始t的默认值是-1，因为t指的是现在存在数据的index，一开始INDEX0都没数据，那么自然是-1，

我们可以看到pop的操作其实并没有删除原来t上的数据，只是把T往前移，但是实际上，我们没必要删除原来T上的数据，因为push的时候我们T-1又会变成T，这时新输入的o会把老O覆盖

Performance and limitations

Performance

让n是stack中元素的数量

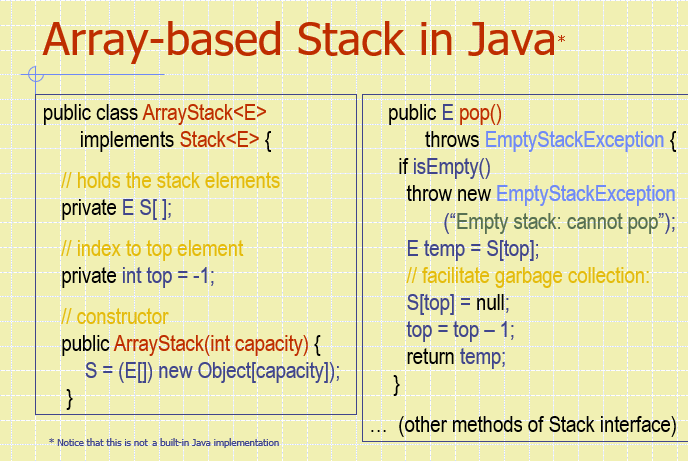
用的space 是O（n）

每一个操作是O(1)

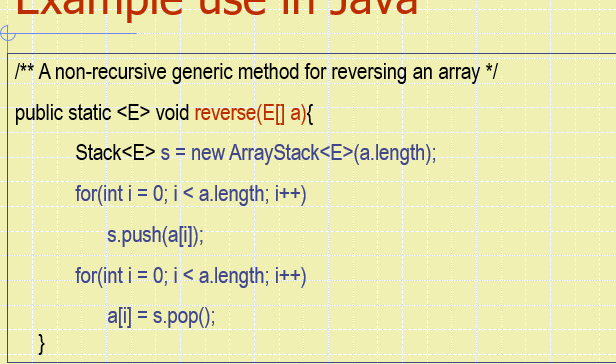
Limitations

stack的最大size必须被实现规定而且无法改变（一开始创造array的时候的arraysize）

full stack 的push会产生*implementation-specific* exception



这个arraylist会hold stack元素



reverse这个array

time complexity是O（n）

Space complexity是O（n）

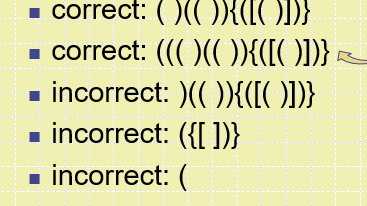
更好的Stack实现

用linked list来代替array

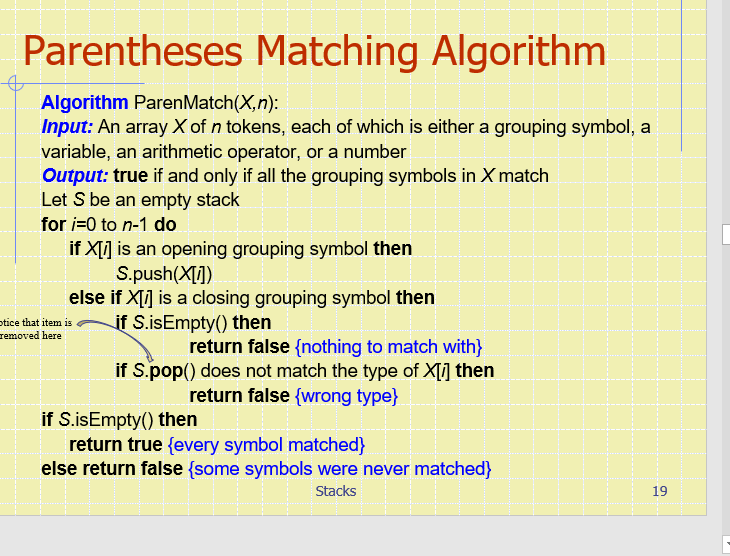
好处在于不用设定一个最大size

在push的时候只用增加一个新node在tail

POP的时候把这个node移除就完事儿了

parentheses matching

两个括号配对



X是输入array，n是token，可以是括号，可以是变量，可以是加减乘除符号

只有括号配对的时候才会输出true

从1到n-1,如果是左括号，push到stack里，如果是右括号，那么

1. Stack是empty的，return false (因为左括号都抵消完了，还有右括号，就没match成功)
2. Stack的pop（左括号）和X(i）（这时我们检测到的右括号）不是一个类型，false

遍历完了以后

如果这时STACK是empty，就return true。如果不是empty就returnfalse(还有一些左括号没被消去)

怎么用stack进行四则运算

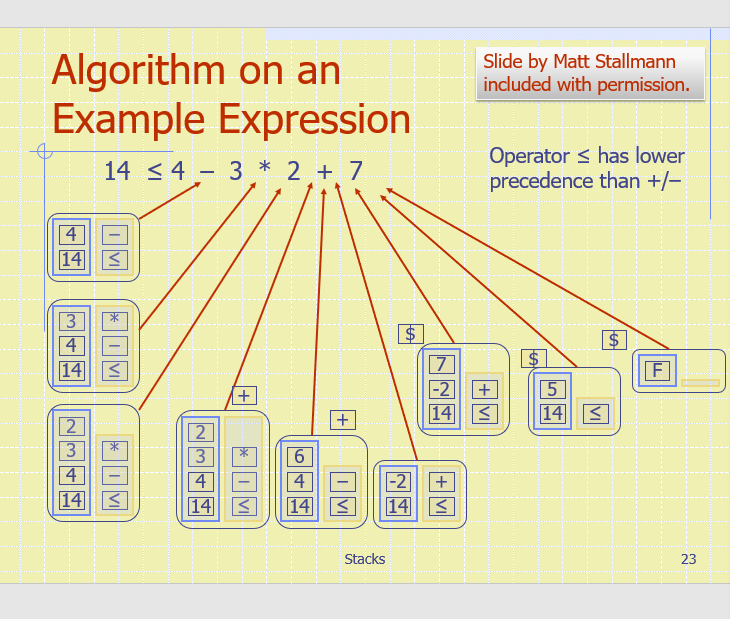
首先，\*/的优先级高于+-

其次，同级的就从左至右

X-Y+Z更倾向于(x-y)+z而不是x-(y+z)

总体IIDEA:把每个运算符号push到stack里，优先进行pop优先级高的符号





从左往右进行

左边放 右边放

14 <=

4 -

3 \* 因为3后面是乘号所以这里不进行4-3

2 这时候放+了，但是+的优先级小于上一个的\*,所以我们进行上一级的操作

2\*3=6 这里待定的是+号，与-号同优先级

按照从左往右的规则

这里进行-

变成了

-2 +

7

5

这时候比较

FALSSE

Computing spans()

用stack作为一个辅助数据结构

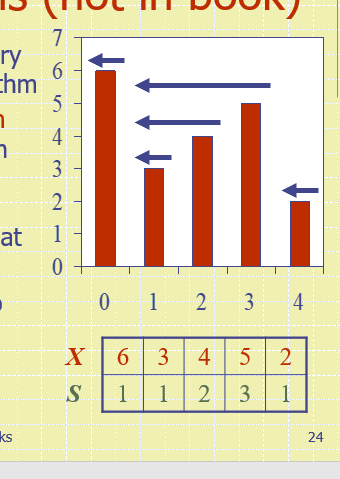
给你一个Array X,

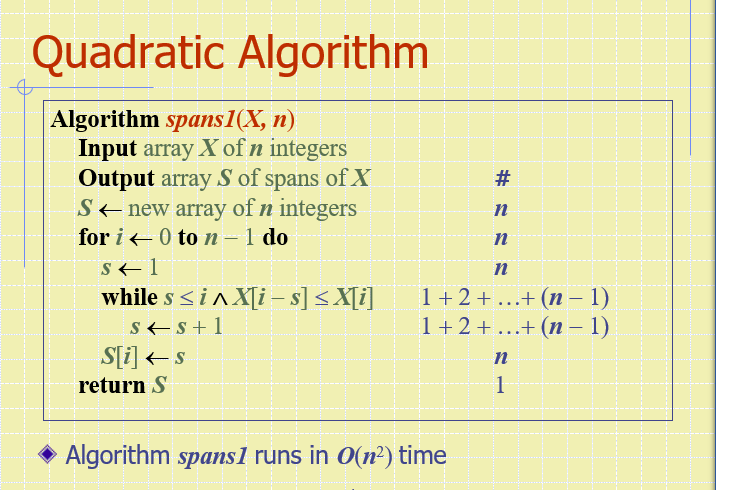
S(i)指的是最大的数字连续几个X都在增长或不变

比如

X(1)=1, X(2)=2， X(3)=1, X(4)=5

S(1)=1, s(2)=2 S(3)=1 s(4)=2





arrayX有n个整数

输出arrayS有X个span

创造新S的长度设置为n

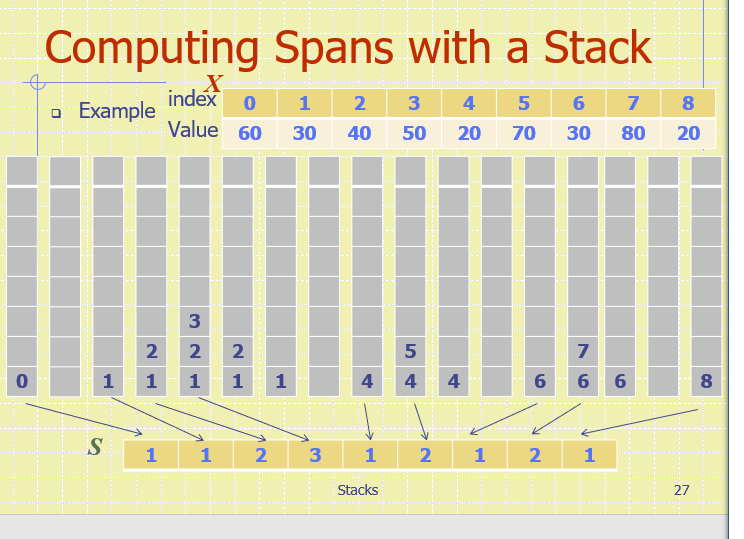
从0到N-1

S默认为1,

把X[i]往前遍历，WHILE X[i]大于X[i-1]，就会S+1，

最后return S

用stack来考虑computing spans



让i代表current index

只要INDEX比前面INDEX有更大的值，我们就push一次

否则我们就把所有的element全一个一个POP出去

这个array X里的所有X，只会被push一次，pop一次，stack多高并不会影响time complexity

STACK的高度代表了S的value

直到下一个比上一个高，我们一直累积STACK，0,0去掉，1,2,3，去掉，去掉，4,5，去掉，去掉，67，去掉，去掉

只有增加的时候会统计stack容量

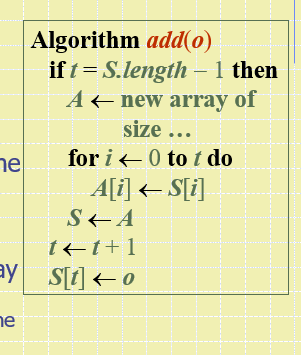
Linear算法，我们会遍历这个array n次，每个Index最多push一次,pop一次，那么就是n+n

我们总共会记录n次S

最后complexity是O（n）

当一个stack full（array长度到顶了）我们没法再push

这时我们可以用add，



如果t=S.length-1(到顶)

那么设置一个新ARRAY并且有着新size

一一复制过去

再把这个TEMP ARRAY A赋值给S（这时S的指针指向A）

t变大，把新add的element o 输入进去

换句话来说，我们创建了一个新array有着新长度并替代了原ARRAY，只是还用着相同的变量S来指代，在这个足够长的新S里，再加上我们要加的o

一般来说新array增加的长度是常数C或者直接翻倍SIZE