# Tipos abstratos de dados (TADs)

- Um TAD é uma abstração de uma estrutura de dados
- Um TAD especifica:
  - Dados armazenados
  - Operações sobre os dados
  - Condições de erros associadas à opers
- Exemplo: TAD que modela um sistema de controle de estoque
  - Os dados são os pedidos de compra/venda
  - As operações suportadas são:
    - comprar(produto, preço)
    - vender(produto, preço)
    - cancelar(pedido)
  - Condições de erro:
    - Comprar/vender um produto não existente
    - Cancelar um pedido não existente

# O TAD Pilha

- O TAD Pilha armazena *objetos* arbitrários
- Inserções e remoções segue o esquema *LIFO*
- Exemplo: uma pilha de pratos
- Principais operações:
  - push(object): insere um elemento
  - object pop(): remove e returna o último elemento inserido
- Operações auxiliares:
  - object top(): retorna o último elemento inserido sem removê-lo
  - integer size(): retorna o número de lementos armazenados
  - boolean isEmpty(): indica se há ou não elementos na Pilha

# Exceções

- Ao executar uma operação em um TAD, pode-se causar uma condição de erro, que chamamos exceção
- Execções podem ser *levantadas* (*thrown*) por uma operação que não pode executá-la
- No TAD Pilha, as operações *pop* e *top* não podem ser realizadas se a pilha está vazia
- Executar *pop* ou *top* numa pilha vazia causa a exceção EPilhaVazia

# Aplicações de pilhas

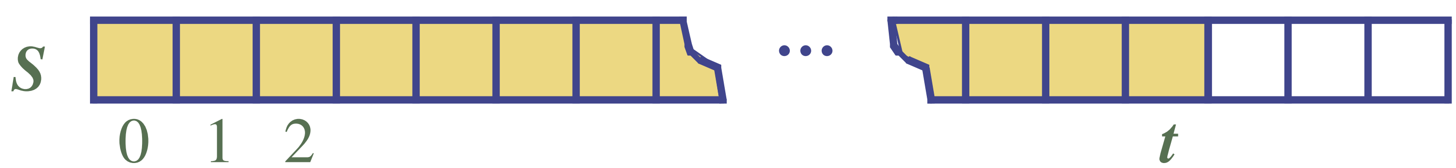
- Aplicações diretas
  - Histórico de páginas visitadas num navegador
  - Sequência de desfazer em um editor de textos
  - Cadeia de chamada de métodos num programa
- Aplicações indiretas
  - Estrutura de dados auxiliares para algoritmos
  - Componentes de outras estruturas de dados

# Pilhas baseadas em *Arrays*

- Uma forma simples de implementar uma pilha usa *arrays*
- Adicionamos elementos da esquerda para a direita
- Uma variável mantém o índice do elemento no topo da pilha

**Algoritmo** *size*()
  **retorne** $t + 1$

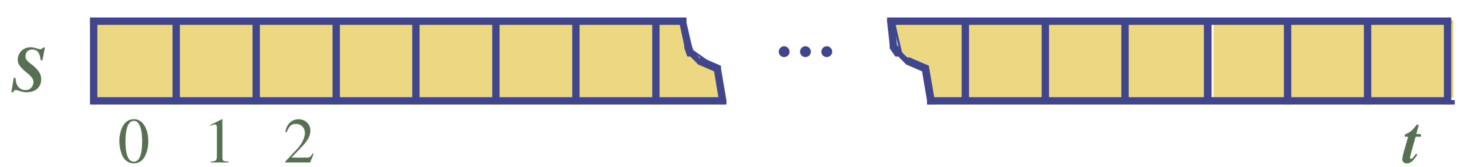
**Algoritmo** *pop*()
  **Se** (*estaVazia*())
    **throw** *EPilhaVazia*
  **senão**
    $t \leftarrow t - 1$
    **retorne** $S[t + 1]$



# Pilhas baseadas em *Arrays*

- O *array* pode ficar cheio
- A operação *push* pode então levantar a exceção EPilhaCheia
  - Está é uma limitação da implementgação baseada em *arrays*
  - Não é intrínsico do TAD Pilha

**Algoritmo** *push*(*o*)
  **Se** ($t$ = $S.length$ − 1)
    **throw** *EPilhaCheia*
  **senão**
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$

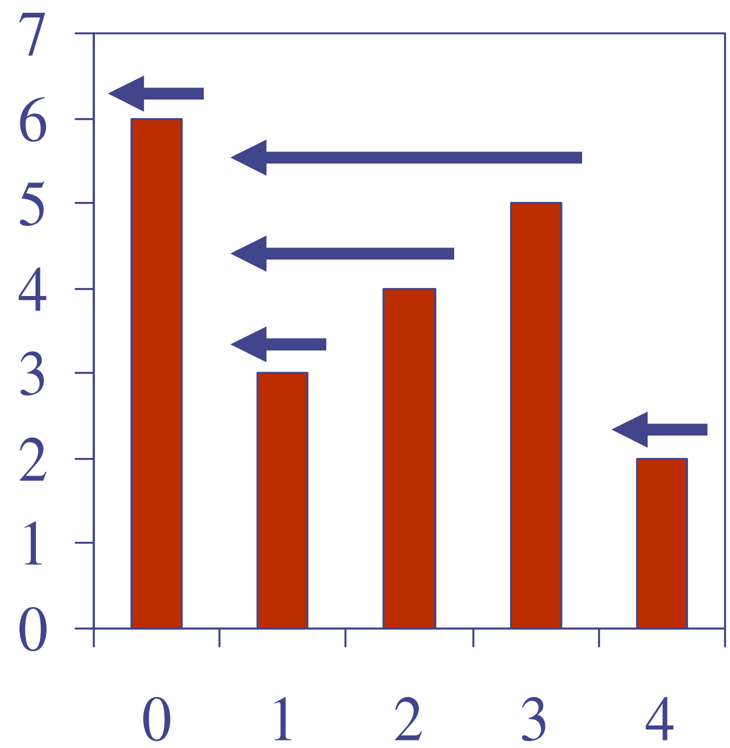


# Desempenho e limitações

- Desempenho
  - Seja $n$ o número de elemento na pilha
  - O espaço usado é $O(n)$
  - Cada operação roda em tempo $O(1)$
- Limitações
  - O tamanho máximo deve ser definido a priori e não pode ser mudado
  - Tentando colocar um novo elemento numa pilha cheia causa uma exceção específica da implementação (*array*)

# Alcance (*span*)

- Veremos a apliação da Pilha como estrutura de dados auxiliar em um algoritmo
- Dado um *array* $X$, o alcance $S[i]$ de $X[i]$ é o número máximo de elementos consecutivos $X[j]$ imediatamente precedendo $X[i]$ e $X[j] \leq X[i]$
- Alcances  têm aplicações em análise financeira
  - Estoque em 52 semanas




| | | | | | |
|---|---|---|---|---|---|
| $X$ | 6 | 3 | 4 | 5 | 2 |
| $S$ | 1 | 1 | 2 | 3 | 1 |

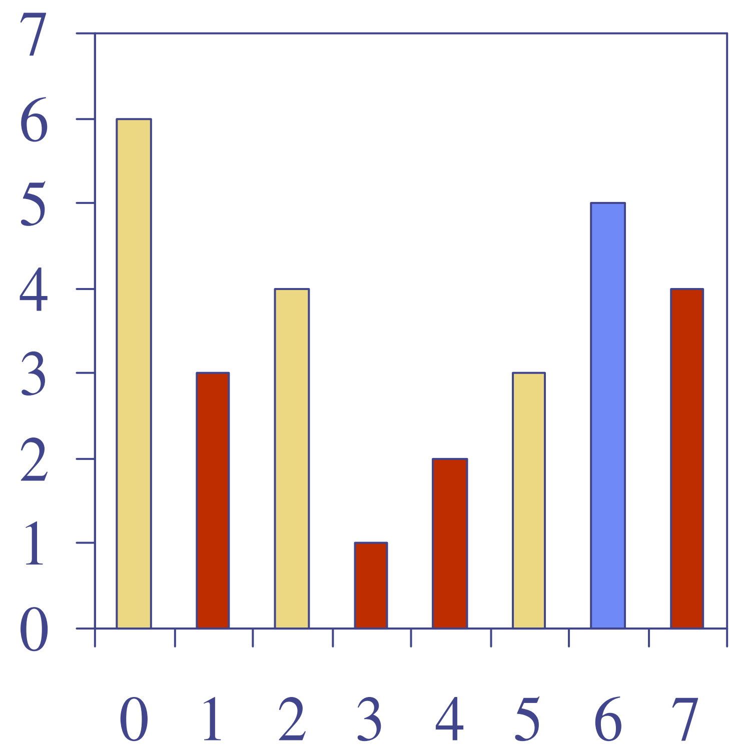
# Algoritmo quadrádico

| | |
|---|---|
| **Algoritmo** *alcance1*$(X, n)$ | |
|   **Entrada** *array* $X$ de $n$ inteiros | |
|   **Saída** *array* $S$ dos alcances de $X$ | # |
|   $S \leftarrow$ novo *array* de $n$ inteiros | $n$ |
|   **para** $i \leftarrow 0$ **até** $n - 1$ **faça** | $n$ |
|     $s \leftarrow 1$ | $n$ |
|     **enquanto** $(s \leq i \wedge X[i - s] \leq X[i])$ | $1 + 2 + \ldots + (n - 1)$ |
|       $s \leftarrow s + 1$ | $1 + 2 + \ldots + (n - 1)$ |
|     $S[i] \leftarrow s$ | $n$ |
|   **retorne** $S$ | 1 |

- Algoritmo *alcance1* roda em tempo $O(n^2)$

# Alcances com Pilhas

- Nós mantemos em uma Pilha os índices dos elementos visíveis quando “olhamos para trás”
- Nós percorremos o *array* da esquerda para a direita
  - Seja $i$ o índice atual
  - Pega-se os índices da pilha até encontrar $j$ tal que $X[i] < X[j]$
  - Realiza-se a atribuição $S[i] \leftarrow i - j$
  - Coloca-se $x$ na pilha



# Algoritmo Linear

- Cada índice do *array*
  - É colocado na pilha exatamente uma vez
  - É retirado uma vez
- Comandos no laço *while* são executados $n$ vezes
- Algoritmo *alcence2* roda em tempo $O(n)$

| **Algoritmo** *alcance2*$(X, n)$ | # |
| --- | --- |
| $S \leftarrow$ novo *array* de $n$ inteiros | $n$ |
| $A \leftarrow$ nova pilha vazia | 1 |
| **para** $i \leftarrow 0$ **até** $n - 1$ faça | $n$ |
| **enquanto** ( $\neg A.estaVazia() \wedge X[A.topo()] \leq X[i]$ ) **faça** | $n$ |
| $j \leftarrow A.retirar()$ | $n$ |
| **Se** ($A.estaVazia()$) **então** | $n$ |
| $S[i] \leftarrow i + 1$ | $n$ |
| **senão** | |
| $j \leftarrow$ A.topo(); | $n$ |
| $S[i] \leftarrow i - j$ | $n$ |
| $A.colocar(i)$ | $n$ |
| **retorne** $S$ | 1 |

# Pilha crescente baseada em *array*

- Em uma operação *push()*, quando o *array* está cheio, ao invés de levantar uma exceção, substituímos o *array* por um maior
- Qual o tamanho do *array*?
  - Estratégia incremental: aumentar o *array* usando uma constante $c$
  - Estratégia de duplicação: duplicar o tamanho do *array*

**Algoritmo** *colocar*($o$)
  **Se** ($t = S.length - 1$)
        **então**
    $A \leftarrow$ novo *array*
    **para** $i \leftarrow 0$ **até** $t$ **faça**
      $A[i] \leftarrow S[i]$
      $S \leftarrow A$
  $t \leftarrow t + 1$
  $S[t] \leftarrow o$

# Comparação de estratégias

- Comparamos a estratégia incremental e de duplicação analisando o tempo total $T(n)$ necessário para realizar uma série de $n$ operações *push*
- Assumimos que começamos com uma pilha vazia representada por um *array* de tamanho 1
- Chamamos tempo de amortização de uma operação *push* o tempo médio de uma operação sobre uma série de operações-$T(n)/n$

# Análise da estratégia incremental

- Substituímos o *array* $k = n/c$ vezes
- O número toral $T(n)$ de uma série de $n$ operações *push* é proporcional a

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- Como $c$ é uma constante, $T(n)$ é $O(n + k^2)$, i.e., $O(n^2)$
- O tempo amortizado de uma operação *push* é $O(n)$

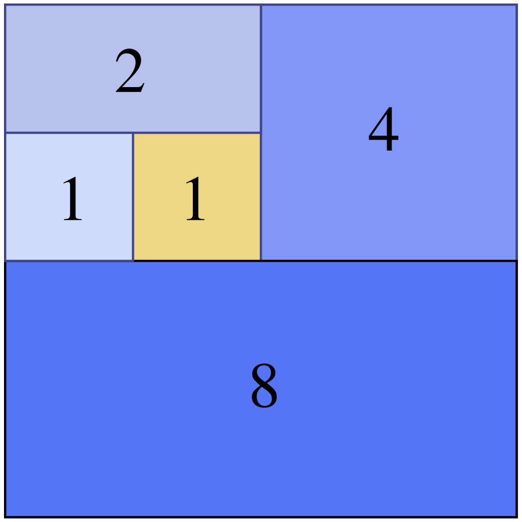
# Análise da estratégia de duplicação

- Substituímos o *array* $k = \log_2 n$ vezes
- O tempo total $T(n)$ de uma série $n$ de operações *push* é proporcional a

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 2 = 3n - 2$$

- $T(n)$ é $O(n)$
- O tempo amortizado de uma operação *push* é $O(1)$

Série geométrica



# A interface **Pilha** em JAVA

- Interface JAVA correspondente ao nosso TAD Pilha
- Requer a definição da classe EPilhaVazia
- Diferente da classe interna JAVA java.util.Stack

```
public interface Pilha {

  public int size();

  public boolean isEmpty();

  public Object top()
        throws EPilhaVazia;

  public void push(Object o);

  public Object pop()
         throws EPilhaVazia;
}
```

# Pilha baseada em *array* - JAVA

```
public class PilhaArray
      implements Pilha {

   // Armazena elementos da pilha
   private Object S[ ];

   // índice do elemento do topo
   private int t = -1;

   // construtor
   public PilhaArray(int tam) {
       S = new Object[tam]);
    }
```

```
    public Object pop()
          throws EPilhaVazia {
       if isEmpty()
          throw new EPilhaVazia
             (“Pilha vazia”);
       Object temp = S[topo];
       // facilita coleta de lixo
       S[topo] = null;
       t = t – 1;
       return temp;
    }
```