

ESTRUTURAS DE DADOS

2024/2025

Aula 04

- Colecção *Stack*
- Interface `StackADT`
- Implementação em *array*
- Implementação em Lista Ligada



**ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO**

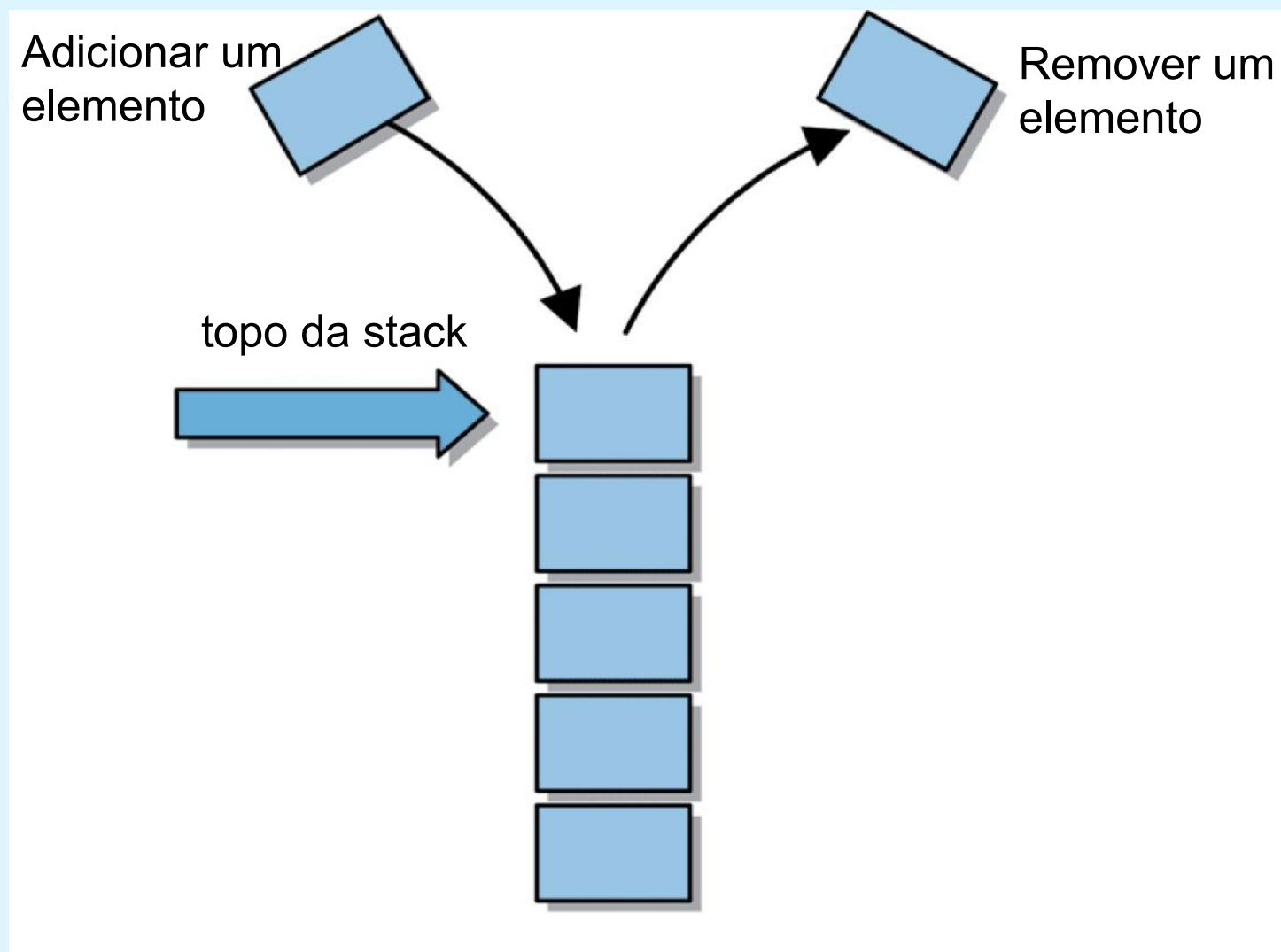
Introdução

- Nas listas a ordem pela qual são colocados os itens não é considerada importante
- Existem situações onde esta informação tem de ser mantida
- A **Stack** representa uma colecção em que os elementos são colocados pela ordem em que são inseridos
- Em qualquer momento, apenas um elemento da **Stack** pode ser removido, e esse é o elemento que menos tempo esteve na **Stack**

Coleção Stack

- Uma coleção de pilha (**Stack**) organiza os elementos da seguinte forma: ultimo a entrar primeiro a sair (**Last-in-First-Out** LIFO)
- É muito similar a uma pilha de pratos, livros, etc
- Apenas podemos colocar novos elementos no topo da pilha (**Stack**)
- Apenas podemos remover elementos do topo da pilha (**Stack**)

Vista Conceptual de uma Stack



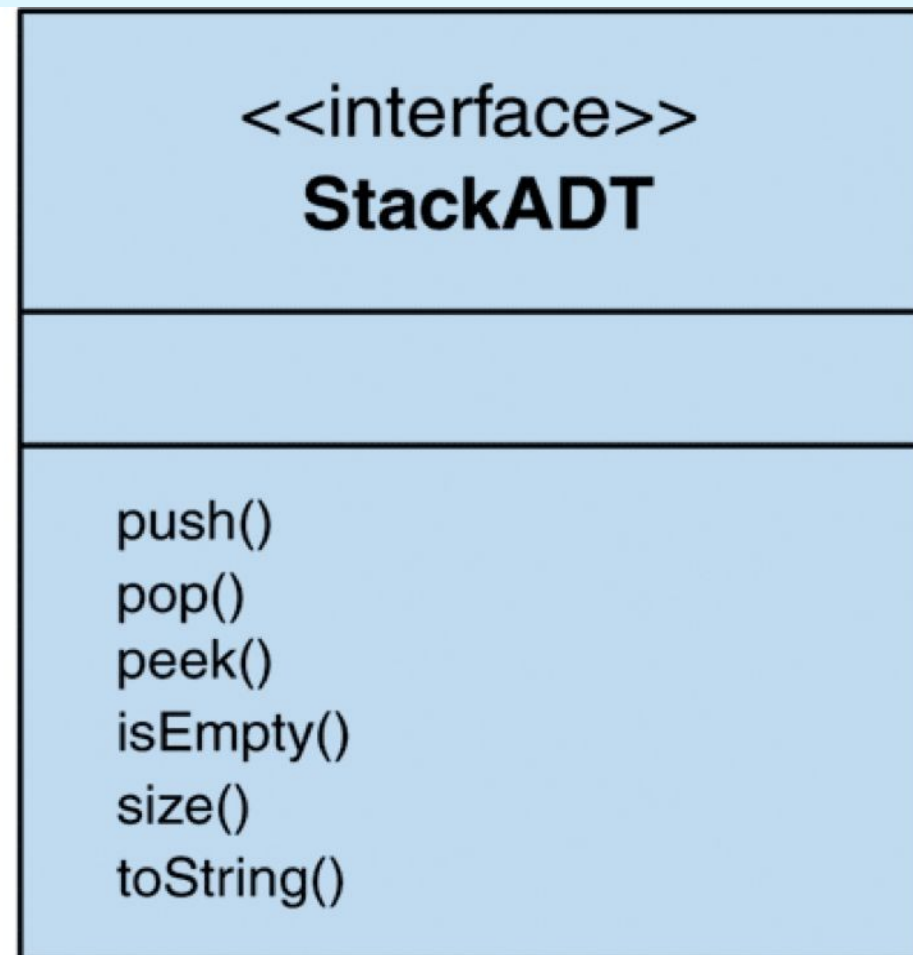
Operações da Coleção

- Cada colecção tem um conjunto de operações que definem a forma como interagimos com ela
- Geralmente incluem funcionalidades para o utilizador:
 - adicionar e remover elementos
 - determinar se a colecção está vazia
 - determinar o tamanho da colecção

Operações da Stack

Operação	Descrição
push	Adiciona um elemento ao topo da pilha
pop	Remove um elemento do topo da pilha
peek	Examina o elemento no topo da pilha
isEmpty	Determina se a pilha está vazia
size	Determina o número de elementos da pilha
toString	Representação da pilha em string

Interface StackADT



Interface StackADT

```
public interface StackADT<T> {  
  
    /** Adds one element to the top of this stack.  
     * @param element element to be pushed onto stack  
     */  
    public void push(T element);  
  
    /** Removes and returns the top element from this stack.  
     * @return T element removed from the top of the stack  
     */  
    public T pop();  
  
    /** Returns without removing the top element of this stack.  
     * @return T element on top of the stack  
     */  
    public T peek();  
}
```



```
/** Returns true if this stack contains no elements.
 * @return boolean whether or not this stack is empty
 */
public boolean isEmpty();

/** Returns the number of elements in this stack.
 * @return int number of elements in this stack
 */
public int size();

/** Returns a string representation of this stack.
 * @return String representation of this stack
 */
@Override
public String toString();
}
```

Usar Stacks

- As pilhas (**Stacks**) são particularmente úteis na resolução de certos tipos de problemas
- Considere a operação de voltar atrás de uma operação numa aplicação
 - mantém o controlo das operações mais recentes em ordem inversa

Expressões *Postfix*

- Vamos examinar um programa que utiliza uma pilha para avaliar expressões ***postfix***
- Numa expressão ***postfix***, o operador vem depois dos seus dois operandos
- Geralmente nós usamos a notação ***infix***, com parênteses para forçar a precedência:

(3 + 4) * 2

- Em notação **postfix** seria escrito da seguinte forma

3 4 + 2 *

- O cálculo:

((1 + 2) * 4) + 3

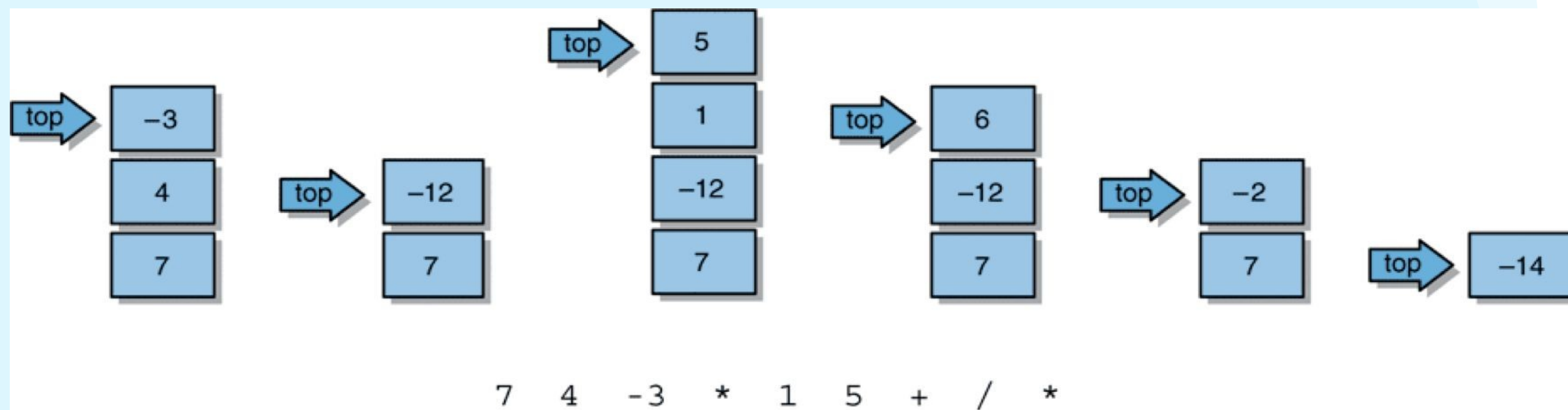
- também pode ser escrito com recurso à notação **postfix** com a vantagem de não ser necessária nenhuma regra de precedência nem parêntesis

1 2 + 4 * 3 +

- A expressão é avaliada da esquerda para a direita com recurso a uma pilha (**Stack**):
 - quando for encontrado um operando fazer o push
 - quando for encontrado um operador fazer o pop de dois operandos e avaliar o resultado, de seguida fazer o push do resultado

Usar uma *stack* para avaliar expressões *postfix*

14



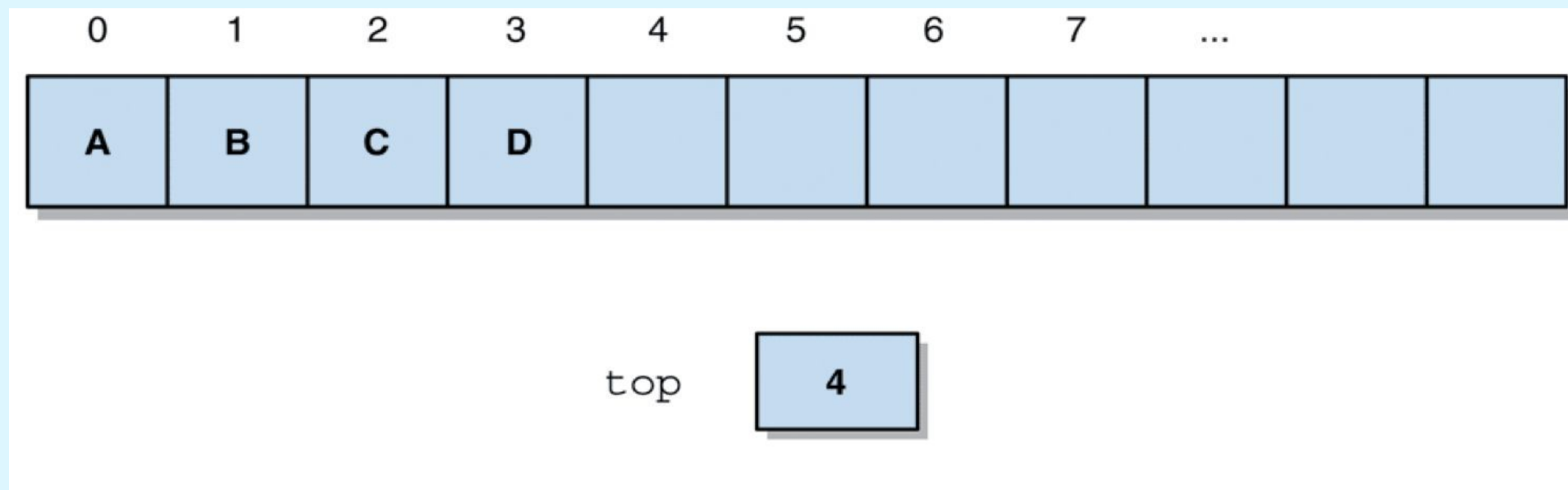
Implementar uma *stack* com recurso a um *array*

15

- Agora vamos examinar a implementação de uma pilha (***Stack***) baseada num *array*

- Vamos tomar as seguintes decisões de projecto a seguir:
 - manter um **array** de referências genéricas
 - o fundo da pilha está no índice **0**
 - os elementos da pilha estão em ordem e contíguas
 - uma variável inteira **top** armazena o índice do próximo **slot** disponível no **array**
- Esta abordagem permite que a pilha possa aumentar e diminuir nos maiores índices

Implementação de uma *stack* com um *array*



Classe ArrayStack

```
public class ArrayStack<T> implements StackADT<T> {  
  
    /**  
     * constant to represent the default capacity of the array  
     */  
    private final int DEFAULT_CAPACITY = 100;  
  
    /**  
     * int that represents both the number of elements and the next  
     * available position in the array  
     */  
    private int top;  
  
    /**  
     * array of generic elements to represent the stack  
     */  
    private T[] stack;  
}
```

```
/**
 * Creates an empty stack using the default capacity.
 */
public ArrayStack()
{
    top = 0;
    stack = (T[]) (new Object[DEFAULT_CAPACITY]);
}

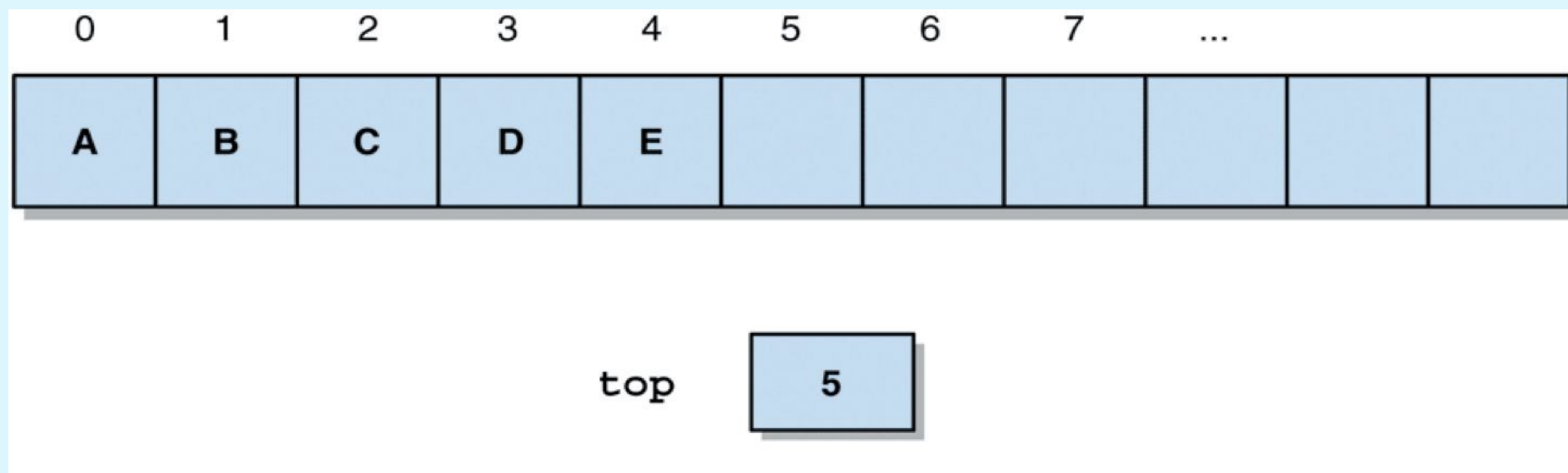
/**
 * Creates an empty stack using the specified capacity.
 * @param initialCapacity represents the specified capacity
 */
public ArrayStack (int initialCapacity)
{
    top = 0;
    stack = (T[]) (new Object[initialCapacity]);
}
```

ArrayStack: Operação push

```
/**
 * Adds the specified element to the top of this stack,
 * expanding the capacity of the stack array if necessary.
 * @param element generic element to be pushed onto stack
 */
public void push (T element)
{
    if (size() == stack.length)
        expandCapacity();

    stack[top] = element;
    top++;
}
```

Stack depois de realizar o push do elemento E



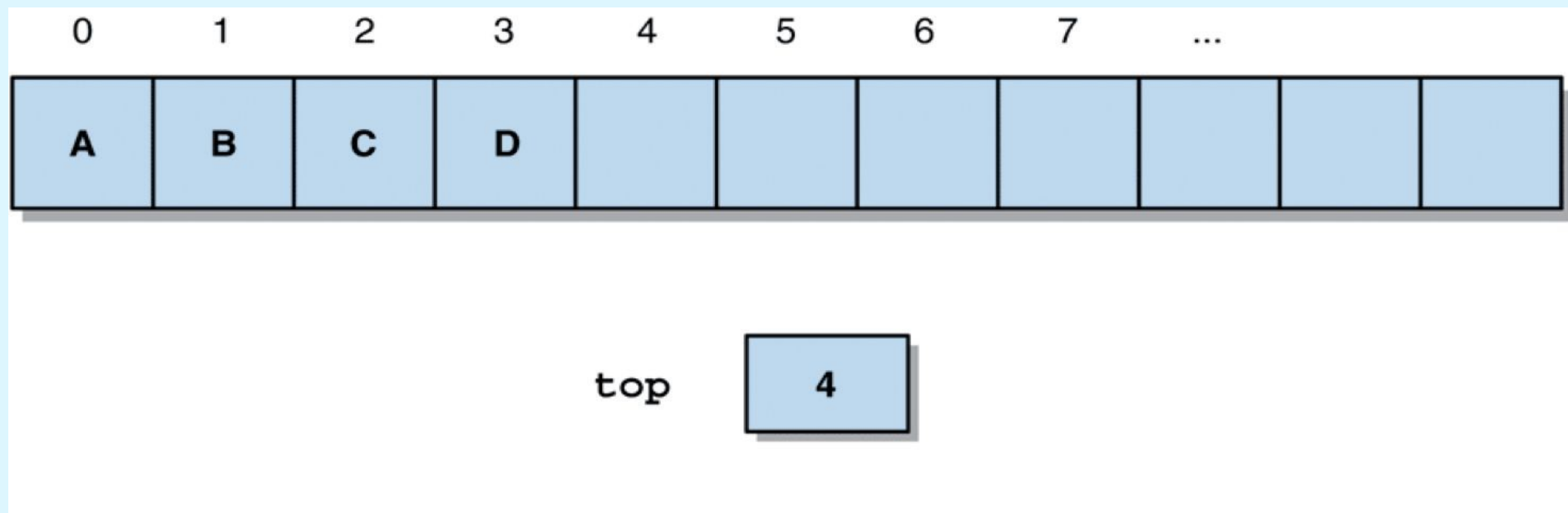
ArrayStack: Operação pop

```
/**
 * Removes the element at the top of this stack and
 * returns a reference to it.
 * Throws an EmptyCollectionException if the stack is empty.
 * @return T element removed from top of stack
 * @throws EmptyCollectionException if a pop
 * is attempted on empty stack
 */
public T pop() throws EmptyCollectionException {
    if (isEmpty())
        throw new EmptyCollectionException("Stack");

    top--;
    T result = stack[top];
    stack[top] = null;

    return result;
}
```

Stack depois de realizar o pop do elemento top



ArrayStack: Operação peek

```
/**
 * Returns a reference to the element at the top of this stack.
 * The element is not removed from the stack.
 * Throws an EmptyCollectionException if the stack is empty.
 * @return T element on top of stack
 * @throws EmptyCollectionException if a
 * peek is attempted on empty stack
 */
public T peek() throws EmptyCollectionException
{
    if (isEmpty())
        throw new EmptyCollectionException("Stack");

    return stack[top-1];
}
```


Outras Operações

- A operação `size` implica encontrar uma forma de devolver o número de elemento na **Stack**
- A operação `isEmpty` retorna `true` se a **Stack** estiver vazia e `false` caso contrário
- A operação `toString` concatena uma `String` composta pelo resultado das operações `toString` de cada um dos elementos na **Stack**

Análise das Operações da *Stack*

26

- Como as operações da **Stack** funcionam todas numa ponta da colecção são geralmente eficientes
- As operações `push` e `pop` para a implementação em array são **$O(1)$**
- Pelo mesmo motivo as outras operações serão também **$O(1)$**

**Agora vamos realizar a
implementação da *Stack* recorrendo a
uma Lista Ligada**

Implementar uma *Stack* com uma Lista Ligada

- Podemos usar uma lista ligada para implementar uma colecção ***Stack***
- No entanto, primeiro teremos de criar a classe que irá representar o nó na lista

Classe LinearNode

```
public class LinearNode<T> {  
    /** reference to next node in list */  
    private LinearNode<T> next;  
    /** element stored at this node */  
    private T element;  
  
    /**Creates an empty node.*/  
    public LinearNode() {  
        next = null;  
        element = null;  
    }  
  
    /**  
    * Creates a node storing the specified element.  
    * @param elem element to be stored */  
    public LinearNode(T elem) {  
        next = null;  
        element = elem;  
    }  
}
```

```
/**
 * Returns the node that follows this one.
 * @return LinearNode<T> reference to next node*/
public LinearNode<T> getNext() {
    return next;
}

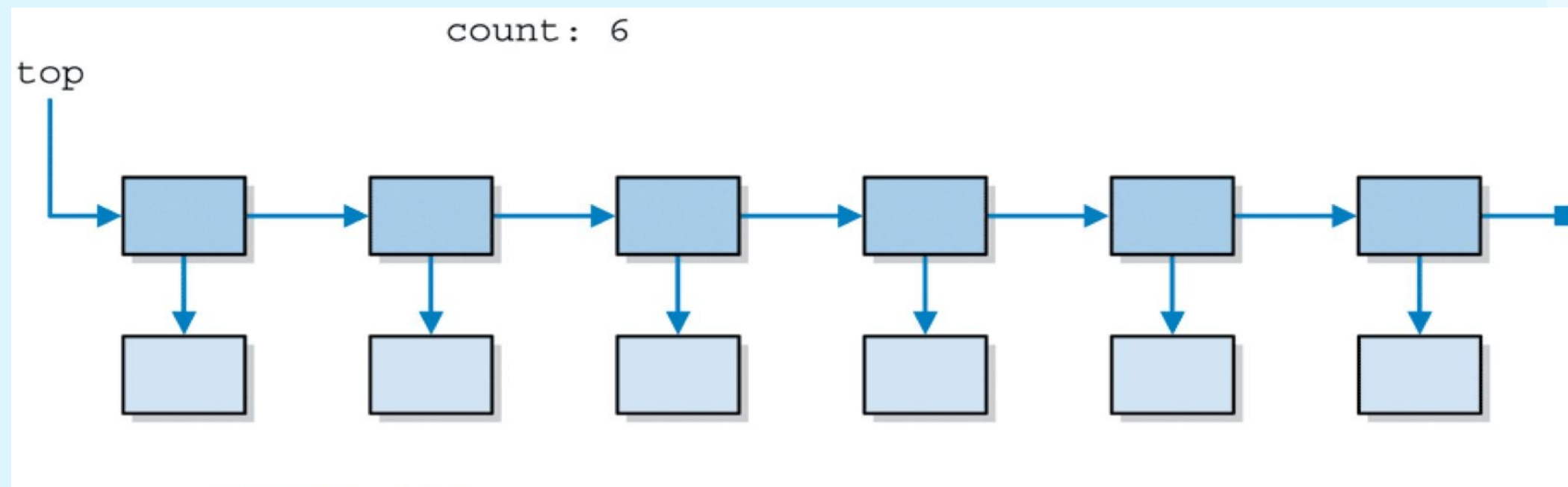
/**
 * Sets the node that follows this one.
 * @param node node to follow this one*/
public void setNext(LinearNode<T> node) {
    next = node;
}

/**
 * Returns the element stored in this node.
 * @return T element stored at this node*/
public T getElement() {
    return element;
}

/**
 * Sets the element stored in this node.
 * @param elem element to be stored at this node*/
public void setElement(T elem) {
    element = elem;
}
}
```

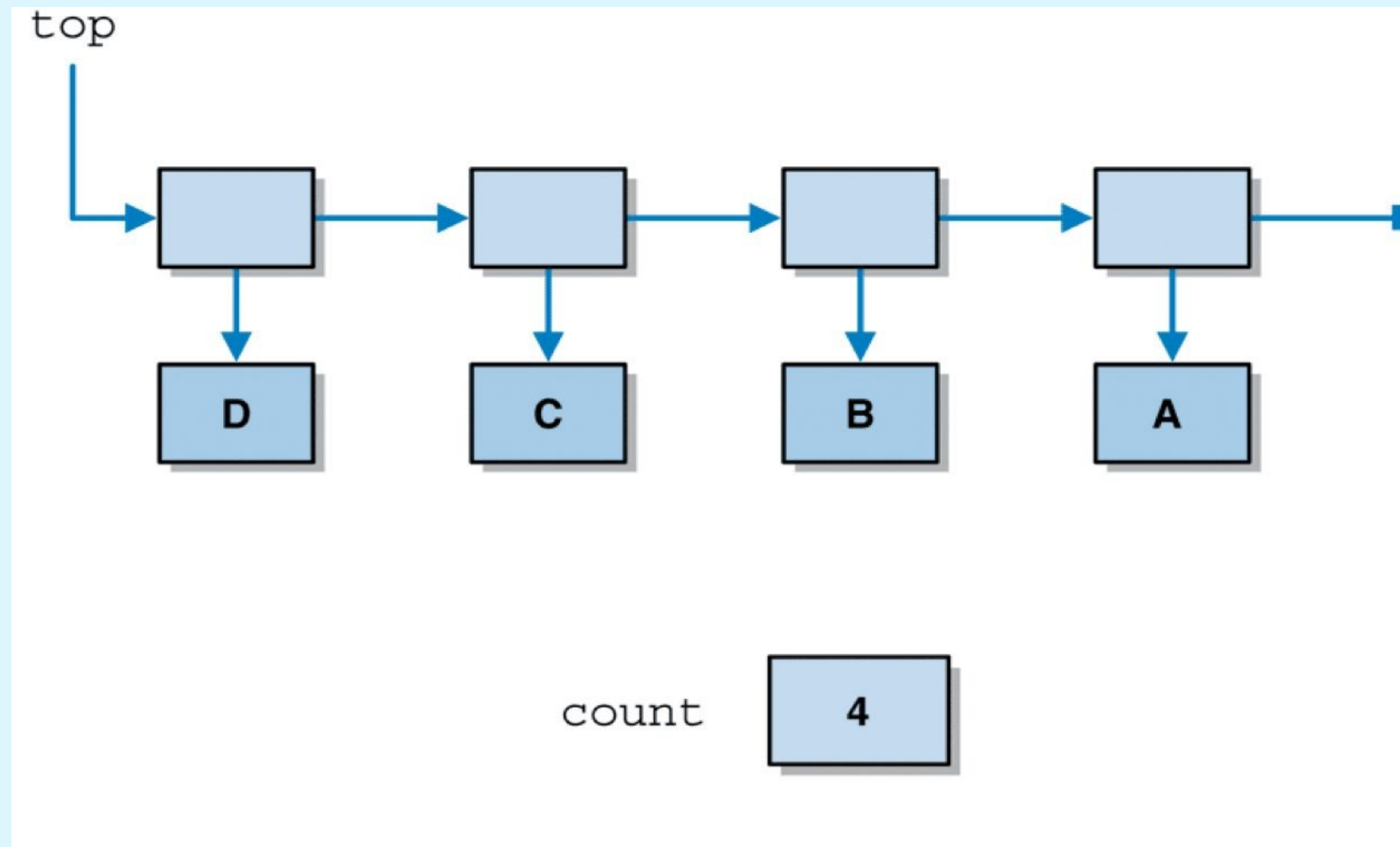
Implementação de uma colecção *Stack* com uma Lista Ligada

31



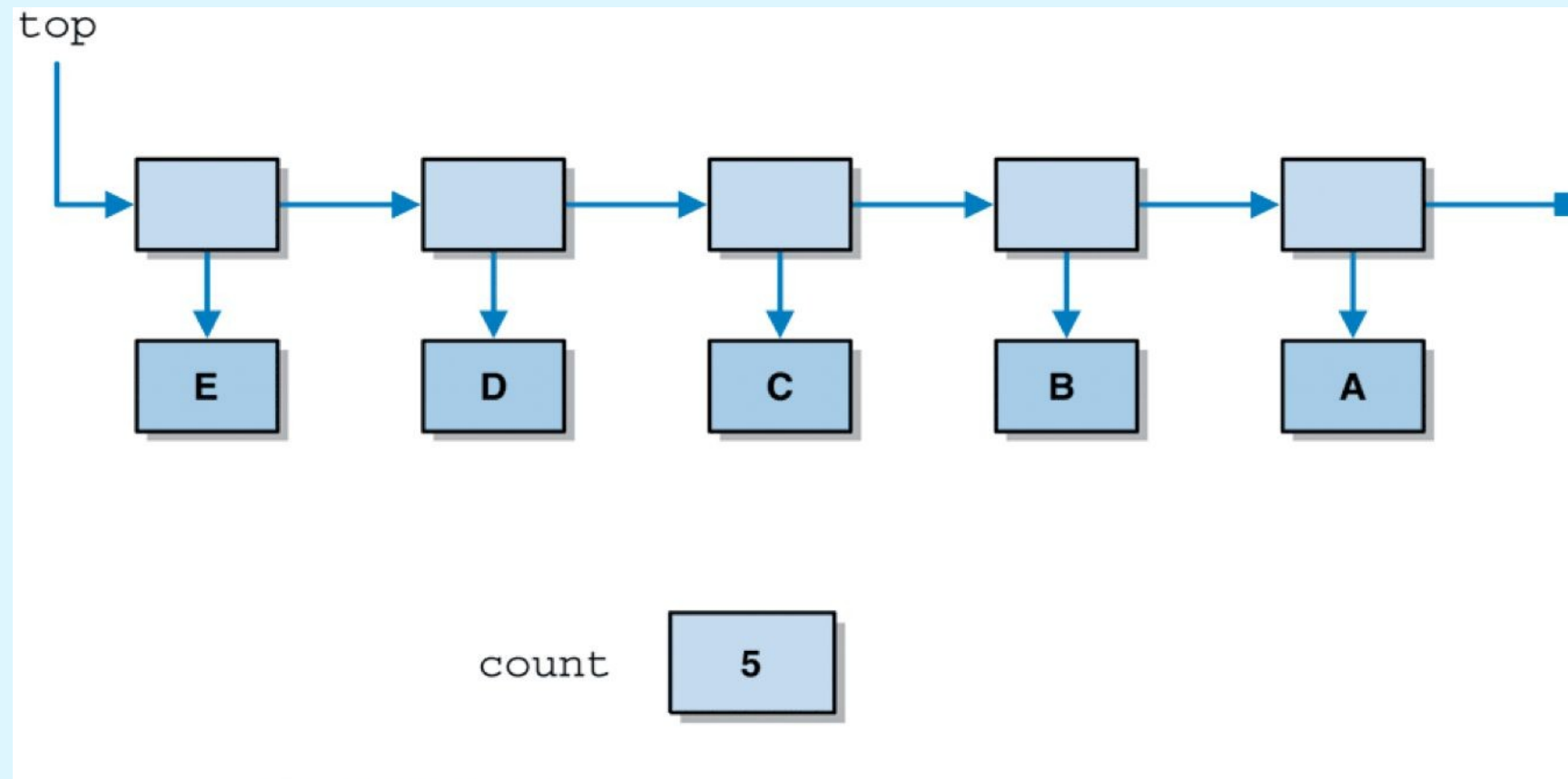
- **Exercício:** Implementar a `LinkedStack`

Stack inicial



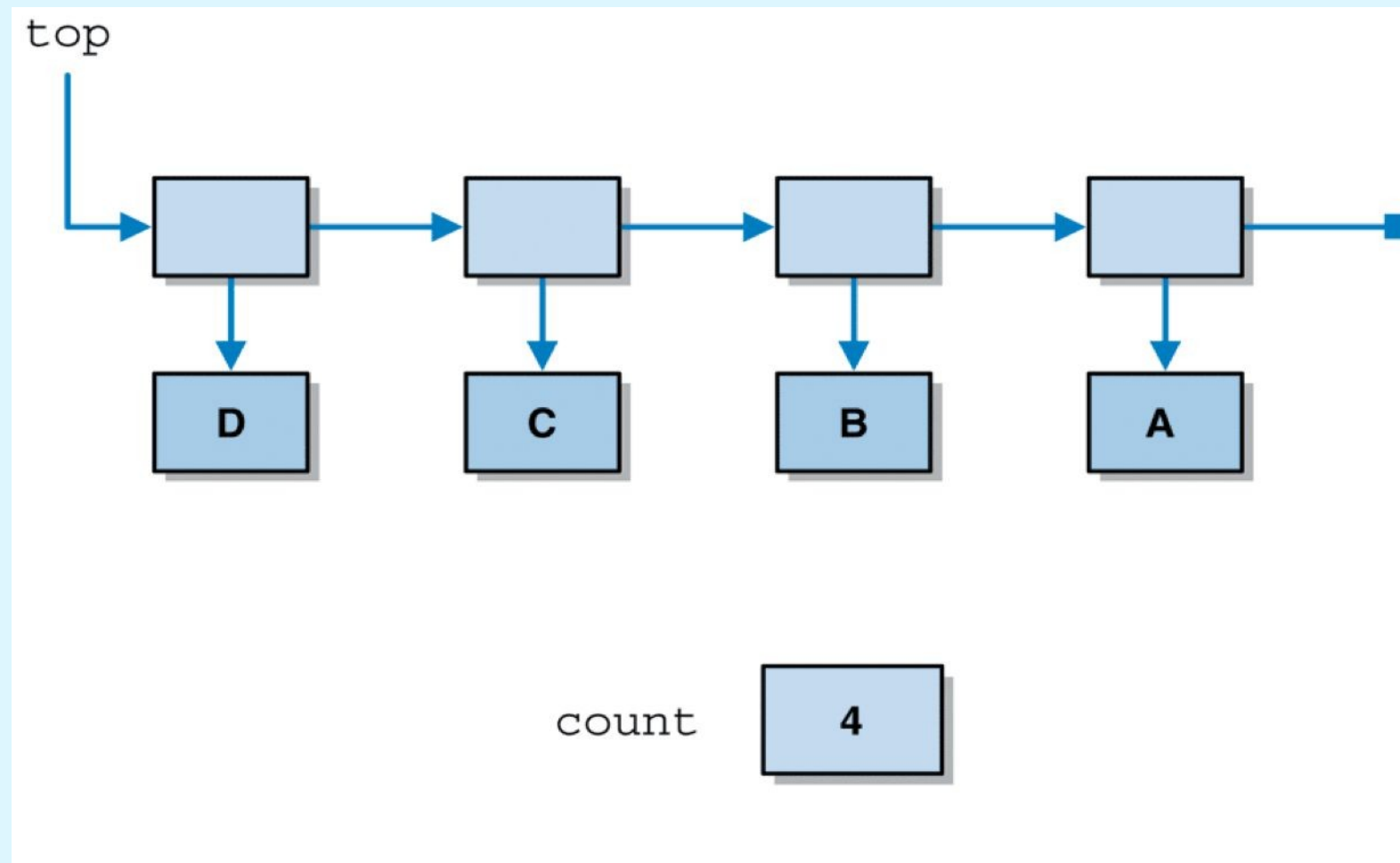
- Recorrendo a uma implementação com Lista Ligada

LinkedList: Operação push



- Como implementar esta operação?

LinkedList: Operação pop



- Como implementar esta operação?

- **Exercício:** Implementar as operações:
 - push
 - pop

Outras operações

- Ao usarmos a implementação em lista ligada a operação `peek` é implementada retornando a referência para `top`
- A operação `isEmpty` retorna `true` se o contador de elementos é **0**, e falso caso contrário
- A operação `size` retorna simplesmente o contador do elementos da pilha (***Stack***)
- A operação `toString` pode ser implementada percorrendo a lista ligada

- **Exercício:** Implementar estas operações!

Análise das operações da *stack*

39

- Tal como para as operações da `ArrayStack`, as operações da `LinkedStack` trabalham numa ponta da colecção o que a torna de uma forma geral eficiente
- As operações `push` e `pop` da implementação em lista ligada são **$O(1)$**
- Assim como as outras operações que são também **$O(1)$**

A classe `java.util.Stack`

- A plataforma de colecções do *Java* define uma classe `Stack` com operações similares
- É uma classe que deriva da classe `Vector` e apresenta algumas características que não são apropriadas para uma pilha (***Stack***) pura
- A classe `java.util.Stack` já existe desde a versão original do *Java* e tem sido adaptada para a plataforma de colecções actual

