Apontamentos de Estruturas de Dados

Stacks

Ricardo Santos| rjs@estg.ipp.pt

Escola Superior de Tecnologia e Gestão
Instituto Politécnico do Porto

Última versão: Outubro de 2022

Índice

6. STACKS 1

6.1. INTRODUÇÃO 1

6.2. APLICAÇÃO DE PILHAS 1

6.3. STACK ADT 2

6.4. INTERFACE STACK 3

6.5. USAR PILHAS – UM EXEMPLO 5

6.6. IMPLEMENTAR UMA STACK ADT COM RECURSO A UM ARRAY 6

6.7. IMPLEMENTAR UMA STACK ADT COM RECURSO A UMA LISTA LIGADA 9

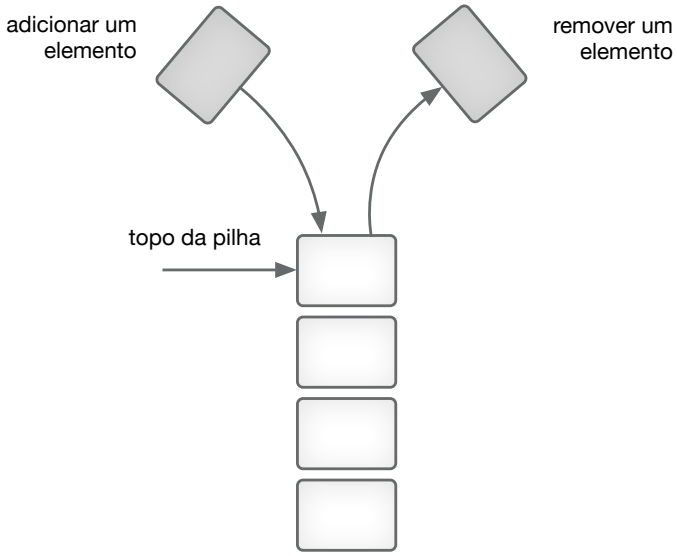
6.8. NOTAS FINAIS 12

6.9. EXERCÍCIOS PROPOSTOS 12

6. Stacks

6.1. Introdução

Uma *Stack* (pilha) é uma estrutura de dados linear na qual todas as inserções e remoções de dados são feitas apenas numa extremidade da pilha, geralmente denominada de topo da pilha. Por esse motivo, uma pilha é denominada de estrutura *last-in-first-out* (LIFO), ou seja último a entrar primeiro a sair.



Uma metáfora frequentemente usada é a ideia de uma pilha de pratos. Nessa pilha, apenas é visível e acessível o prato superior, todos os outros pratos permanecem ocultos. À medida que novos pratos são adicionados, cada novo prato irá ser colocado no topo da pilha escondendo cada prato abaixo. À medida que o prato superior é removido da pilha, um novo prato fica visível e torna-se o topo da pilha.

6.2. Aplicação de Pilhas

As pilhas são usadas extensivamente em todos os níveis de um sistema computacional moderno. Por exemplo, um PC moderno usa pilhas ao nível arquitetural, onde são usadas na conceção de um sistema operativo para o tratamento de interrupções e chamadas de funções do sistema operativo. Entre outros usos, as pilhas são utilizadas para executar a Máquina

Virtual Java (*Java Virtual Machine*) e a própria linguagem Java possui uma classe denominada *Stack*, que pode ser utilizada pelo programador.

As pilhas têm muitas outras aplicações. Por exemplo, quando o processador executa um programa através da chamada de uma função, a função tem de saber como retornar ao programa original. Para que isso aconteça é colocado na pilha o endereço atual. Uma vez finalizada a função, o endereço que foi guardado é removido da pilha e a execução do programa é retomada. Se ocorrer uma série de chamadas de funções, os sucessivos valores de retorno são colocados na pilha segundo a ordem LIFO para que cada função possa retornar ao ponto onde foi feita a chamada. As pilhas suportam chamadas de funções recursivas usadas por exemplo na resolução de expressões com notação polonesa inversa (*reversed polished notation*).

Para problemas de pesquisa as pilhas também são essenciais dependendo do tipo de abordagem que se pretenda adotar, no entanto na sua grande maioria são usadas para manter o percurso percorrido ou identificar os elementos que faltam pesquisar. Outro uso comum de pilhas ao nível arquitetural é como meio de alocação e acesso à memória.

6.3. Stack ADT

As pilhas são containers e, como tal, contêm valores de algum tipo. Devemos, portanto, falar da `Stack ADT` de `T`, onde `T` é o tipo dos elementos contidos na pilha.

O conjunto de valores deste tipo é o conjunto de todas as pilhas que contêm elementos do tipo `T`. Assim, o conjunto de valores inclui a pilha vazia, as pilhas com um elemento do tipo `T`, as pilhas com dois elementos do tipo `T` e assim por diante. As operações essenciais do tipo são as seguintes.

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

Além das operações da `Stack ADT` acima, também podemos enunciar alguns axiomas para nos ajudar a entender a `Stack ADT`. Por exemplo, considere os seguintes axiomas:

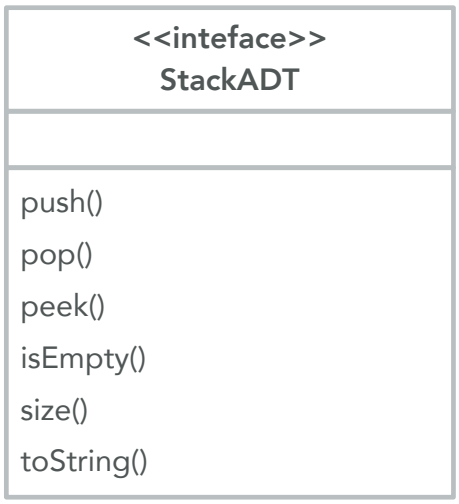
> Para qualquer `Stack s` e elemento `e`, `pop(push(s,e)) = s`.
> Para qualquer `Stack s` e elemento `e`, `top(push(s,e)) = e`.
> Para qualquer `Stack s` elemento `e`, `isEmpty(push(s,e)) = false`.

O primeiro axioma diz-nos que a operação `pop` desfaz o que a operação `push` efetua. A segunda diz-nos que quando um elemento é adicionado na `Stack` torna-se o elemento do topo da pilha. A terceira diz-nos que adicionar um elemento numa `Stack` vazia nunca pode torná-la vazia. Com os axiomas corretos, podemos caracterizar completamente o comportamento das operações do ADT sem precisar descrevê-las informalmente em português (o problema é saber quando temos os axiomas corretos). Geralmente não realizamos esta especificação axiomática de ADTs, embora por vezes possamos usar os axiomas para explicar algumas operações de ADTs.

6.4. Interface Stack

As operações da `Stack ADT` podem referenciar as pilhas através de argumentos e resultados. Isso é o que se espera de funções matemáticas que referenciam o conjunto de valores de um ADT entre si. No entanto, ao implementar pilhas numa linguagem orientada a objetos que nos permite criar uma classe `Stack` com instâncias de `Stack`, não existe necessidade de passar ou retornar valores de `Stack` — as instâncias de `Stack` guardam valores que são transformados noutros valores pelas operações da pilha. Consequentemente,

quando especificamos numa implementação orientada a objetos da `Stack ADT`, todos os argumentos da pilha assim como valores de retorno desaparecem. A mesma coisa ocorre quando estudarmos outros ADTs e suas implementações: os argumentos do conjunto de valores do ADT e os valores de retorno irão desaparecer das assinaturas das operações nas classes de implementação do ADT e, em vez disso, as instâncias são transformadas de um conjunto de valores para outros à medida que as operações são executadas.




```
1. public interface StackADT<T> {
2.    /**  Adds one element to the top of this stack.
3.     *   @param element element to be pushed onto stack
4.     */
5.    void push(T element);
6.    /**  Removes and returns the top element from this stack.
7.     *   @return T element removed from the top of the stack
8.     */
9.    T pop();
10.   /**  Returns without removing the top element of this stack.
11.    *   @return T element on top of the stack
12.    */
13.   T peek();
14.   /**  Returns true if this stack contains no elements.
15.    *   @return boolean whether or not this stack is empty
16.    */
17.   boolean isEmpty();
```

```
18.    /**  Returns the number of elements in this stack.
19.     *   @return int number of elements in this stack
20.     */
21.    int size();
22.    /**  Returns a string representation of this stack.
23.     *   @return String representation of this stack
24.      */
25.    @Override
26.    String toString();
27.}
```

6.5. Usar Pilhas – Um Exemplo

Expressões *Postfix* são expressões matemáticas onde para cada par de Operador e Operandos, os operadores vêm depois dos operandos. Portanto, a ordem de avaliação é da esquerda para a direita. Aqui os operadores são escritos após os operandos.

Por exemplo, `XY+` é uma expressão *postfix* e o seu equivalente *infix* é `X+Y`. Os compiladores geralmente usam notações *postfix* para avaliar uma determinada expressão com facilidade. Torna-se mais fácil avaliar uma determinada expressão devido à ordem dos operadores e operandos. Considere a expressão *postfix*:

`8 2 3 * + 7 / 1 –`

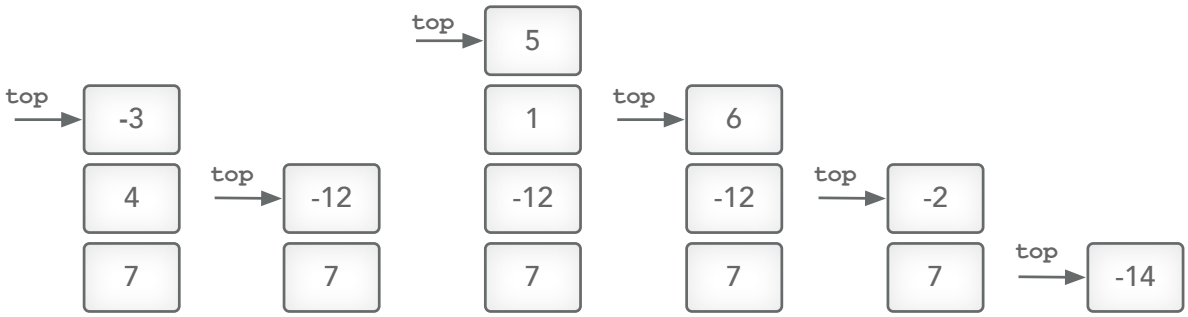
O resultado após a avaliação desta expressão será `1`. Inicialmente, `2*3 = 6`, a sua soma com `8` dá `14`. `14` é então dividido por `7` para dar `2`. Depois de subtrair `1` com `2` obtemos `1` que é o resultado.

Algoritmo de Avaliação *Postfix*

Vamos usar uma pilha para fazer a operação, a pilha irá conter os operandos e o valor resultante de cada par de operandos para cada operador. No final do algoritmo a pilha *postfix* irá conter o valor resultante obtido da expressão total.

A expressão é avaliada da esquerda para a direita:

- 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




```
7  4  -3  *  1  5  +  /  *
```

6.6. Implementar uma Stack ADT com recurso a um array

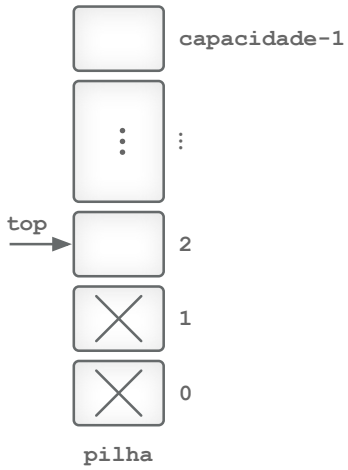
Existem duas abordagens para implementar uma `Stack ADT`: uma implementação contígua com recurso a *arrays* e uma implementação através de listas simplesmente ligadas.

A implementação de pilhas de elementos do tipo `T` com recurso a *arrays* requer um *array* `T` para armazenar o conteúdo da pilha e um "apontador" para acompanhar o topo da pilha. O "apontador" pode registar a localização do elemento do topo ou a localização da próxima posição livre que irá ser ocupada com a próxima operação `push()`. Qualquer uma das abordagens é válida não sendo uma melhor do que a outra desde que o programador saiba claramente o que representa o apontador ao escrever o código.

Se for usado um *array* de tamanho fixo a pilha irá ficar cheia. Se for usado um *array* dinâmico (redimensionável), a pilha será essencialmente ilimitada. Normalmente, redimensionar um *array* é uma computacionalmente pesada porque tem de ser alocado um novo *array* e o

conteúdo do *array* antigo copiado para o novo. Essa flexibilidade tem um custo que nem sempre é compatível com a solução que se está a implementar.

Na nossa implementação teremos um *array* dinâmico para armazenar os elementos da pilha e um índice denominado `top` para acompanhar o topo da pilha. A figura seguinte ilustra essa estrutura de dados.



O índice é denominado de `top` porque mantém o controlo da localização da próxima posição vazia do *array* onde será colocado o novo elemento através da operação `push`. Não esquecer que o *array* cresce do índice `0` para cima, por esse motivo *top* também regista quantos elementos estão na pilha.

O *array* é instanciado para conter um determinado número de elementos, sendo esse o número máximo de elementos que podem ser colocados na pilha até que seja expandida. A figura anterior mostra dois elementos na pilha destacados com uma cruz assim como a variável `top`. Observe como a variável `top` também indica a localização do *array* onde o próximo elemento será armazenado quando for colocado na pilha. A pilha está vazia quando a `top` é `0` e deve ser expandida quando a contagem for igual à capacidade e pretendermos adicionar outro elemento.

A implementação das operações da `Stack ADT` com esta estrutura de dados é bastante direta. Por exemplo, para implementar a operação `push()` primeiro é feita uma verificação para ver se o armazenamento deve ser expandido através da verificação da variável `top` (se

`top` for igual à capacidade deve ser expandido). Em caso afirmativo, o *array* é expandido. De seguida o valor é colocado no local e a contagem é incrementada.

Uma classe que especifica esta implementação pode ser chamada de `ArrayStack(T)`. Esta irá implementar a interface `StackADT(T)` e terá o *array* de armazenamento e a variável de contagem como atributos privados e as operações da pilha como métodos públicos. O construtor irá criar uma pilha vazia. A variável de capacidade pode ser mantida pela classe `ArrayStack` ou fazer parte da implementação de *arrays* dinâmicos na linguagem de implementação.

```
1. public class ArrayStack<T> implements StackADT<T> {
2.    /**
3.     * constant to represent the default capacity of the array
4.     */
5.    private final int DEFAULT_CAPACITY = 100;
6.    /**
7.     * int that represents both the number of elements and the next
8.     * available position in the array
9.     */
10.   private int top;
11.   /**
12.    * array of generic elements to represent the stack
13.    */
14.   private T[] stack;
15.   /**
16.    * Creates an empty stack using the default capacity.
17.    */
18.   public ArrayStack() {
19.      top = 0;
20.      stack = (T[])(new Object[DEFAULT_CAPACITY]);
21.   }
22.   /**
23.    * Creates an empty stack using the specified capacity.
24.    * @param initialCapacity represents the specified capacity
25.    */
26.   public ArrayStack (int initialCapacity)
```

```
27.    {
28.        top = 0;
29.        stack = (T[])(new Object[initialCapacity]);
30.    }
```

6.7. Implementar uma Stack ADT com recurso a uma lista ligada

Uma implementação ligada de uma `Stack ADT` usa uma estrutura de dados ligada para representar os valores do conjunto. Vamos revisitar os fundamentos das estruturas de dados ligadas:

Nó: Uma variável agregada composta por dados e campos de ligações (*links*) para os nós através das suas referências.

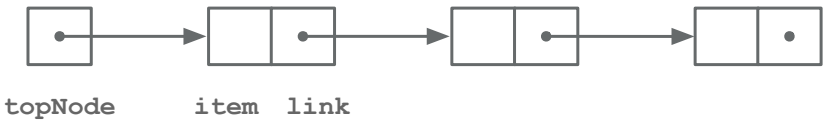
Estrutura ligada (dados): uma coleção de nós formados através dos campos de *links*.

Os nós podem conter um ou mais campos de dados e *links* dependendo da necessidade. estes últimos podem formar uma coleção de nós em estruturas de dados ligadas de formas e tamanhos arbitrários. Entre as estruturas de dados ligadas mais importantes estão as seguintes:

- **Lista simplesmente ligada:** Uma estrutura de dados ligada cujos nós têm um único campo de *link* usado para formar uma sequência de nós. Cada *link* de nó, exceto o último, contém uma referência para o próximo nó na lista; o *link* do último nó contém `null` (um valor de referência especial que não se refere a nada).
- **Lista duplamente ligada:** Uma estrutura ligada cujos nós têm cada um dois campos com *links* usados para formar uma sequência de nós. Cada nó exceto o primeiro tem um *link* predecessor com uma referência ao nó anterior na lista, e cada nó exceto o último tem um *link* sucessor com uma referência ao próximo nó na lista.
- **Árvore ligada:** Uma estrutura ligada cujos nós formam uma árvore.

A estrutura ligada necessária para uma pilha é muito simples exigindo apenas uma lista ligada, portanto, os nós da lista precisam conter apenas um valor do tipo `T` e uma referência ao próximo nó. O elemento superior da pilha é armazenado no início da lista, portanto, os únicos

dados que a estrutura de dados da pilha deve acompanhar são a referência para o início da lista. A figura seguinte ilustra essa estrutura de dados. A referência do início da lista é chamada `topNode`. Cada nó possui um campo de item (para valores do tipo `T`) e um campo de *link* (para as referências).



A figura mostra uma pilha com três elementos; o topo da pilha é o primeiro nó da lista. A pilha está vazia quando `topNode` é nulo; a pilha nunca fica cheia (a menos que a memória seja esgotada).

Implementar as operações da `Stack ADT` através de uma estrutura de dados ligada é bastante simples. Por exemplo, para implementar a operação `push()` é criado um novo nó para o novo elemento e a referência do *link* fica a do valor atual de `topNode`. A variável `topNode` é então atribuída uma referência ao novo nó.

Uma classe que realiza essa implementação pode ser chamada de `LinkedStack(T)`. A `LinkedStack(T)` implementa a interface `StackADT(T)` e teria `top` (o mesmo que `topNode`) como um atributo privado e as operações da pilha como métodos públicos. Deve também ter uma classe nó interna e privada para instâncias dos vários nós, é esta a estrutura ligada que irá manter os dados. O construtor deve criar uma pilha vazia. À medida que são adicionados novos nós, a pilha irá crescer através da instanciação de novos nós com os elementos que constitui a lista ligada. Quando os valores são retirados da pilha, as instâncias de nós são removidas sendo o espaço libertado. Uma `LinkedStack` usa portanto apenas o espaço necessário para os elementos que contém.

O elemento que representa o nó segue de seguida através da classe `LinearNode<T>`:

```
1. public class LinearNode<T> {
2.     /** reference to next node in list */
3.     private LinearNode<T> next;
```

```
4.    /** element stored at this node */
5.    private T element;
6.    /**Creates an empty node.*/
7.    public LinearNode() {
8.       next = null;
9.       element = null;
10.   }
11.   /**
12.    * Creates a node storing the specified element.
13.    * @param elem element to be stored */
14.   public LinearNode(T elem) {
15.      next = null;
16.      element = elem;
17.   }
18.   /**
19.    * Returns the node that follows this one.
20.    * @return LinearNode<T> reference to next node*/
21.   public LinearNode<T> getNext() {
22.      return next;
23.   }
24.   /**
25.    * Sets the node that follows this one.
26.    * @param node node to follow this one*/
27.   public void setNext(LinearNode<T> node) {
28.      next = node;
29.   }
30.   /**
31.    * Returns the element stored in this node.
32.    * @return T element stored at this node*/
33.   public T getElement() {
34.      return element;
35.   }
36.   /**
37.    * Sets the element stored in this node.
38.    * @param elem element to be stored at this node*/
39.   public void setElement(T elem) {
40.      element = elem;
41.   }
```

```
42.}
```

6.8. Notas Finais

Ambas as formas de implementação de pilhas são simples e eficientes, mas a implementação contígua coloca uma restrição do tamanho da pilha ou usa uma técnica de realocação pesada se a pilha ficar muito grande. Se as pilhas implementadas de forma contígua forem muito grandes para garantir que não "estourem", o espaço pode ser desperdiçado.

Uma implementação ligada é essencialmente ilimitada, portanto, a pilha nunca fica cheia. Também é muito eficiente no uso do espaço porque aloca apenas nós suficientes para armazenar os valores realmente mantidos na pilha. No geral, a implementação ligada de pilhas é melhor do que a implementação contígua.

6.9. Exercícios Propostos

Exercício 1

Suponha que uma `ArrayStack` seja implementada de forma a que os elementos do topo sejam sempre armazenados na posição `[0]` do *array*. Quais são as vantagens ou desvantagens dessa abordagem?

Exercício 2

Uma `LinkedStack` deve ter um atributo de contagem? Explique a sua resposta.

Exercício 3

O elemento superior poderia ser armazenado na cauda de uma LinkedStack? Quais seriam as consequências que isso tem para a implementação?

Exercício 4

Uma `LinkedStack` pode ser implementada usando uma lista duplamente ligada. Quais são as vantagens ou desvantagens dessa abordagem?

Exercício 5

Implemente uma calculadora *postfix* com recurso a um `ArrayStack`.

Exercício 6

Substituir a `ArrayStack` pela `LinkedStack` na implementação da calculadora *posfix*. Enumere as alterações teve de fazer na implementação da calculadora para que esta apresentasse o mesmo comportamento.