Universidade Cruzeiro do Sul

Estruturas de dados

Conteúdo:

- Tipos de dados.
- Tipos abstratos de dados (TAD).
- Estrutura de dados pilha.
  - Pilhas estáticas sequenciais.

Autores

Prof. Manuel F. Paradela Ledón
Profª Cristiane P. Camilo Hernandez
Prof. Amilton Souza Martha
Prof. Daniel Calife

Tipos de dados - exemplos

Universidade Cruzeiro do Sul

- Um **tipo de dados** especifica um **conjunto** ou coleção **de valores** que uma constante, variável ou expressão pode assumir (domínio de valores) e um grupo de **operações** que podem ser executadas com dados desse tipo.

```
Exemplo:   int  idade;              //a variável idade poderá armazenar
                                    //valores inteiros, como 4, 20 ou 76

Exemplo:   x * 2.0 + 3.1            //a expressão  x * 2.0 + 3.1  assumirá um
                                    //valor dentro do conjunto de números reais

Exemplo:   int  quad(int v) {       //a função 'quad' retornará um valor
               return(v*v);         //dentro do conjunto de números inteiros
           }
```

Tipos	Valores
Inteiro	-20, 0, 47, 99
Real	23.4, 68.2, -15.0
String	"dados", "árvore", "pilha"
Símbolos	Janeiro, Fevereiro, Março

Os operadores poderão ter significados (operações) diferentes, dependendo dos tipos de dados dos operandos:

```
x + 2.0                 //soma dois valores reais (adição)
"Uma"  +  " casa"       //concatena, junta, duas strings
```

Tipos de dados primitivos

- São os tipos de dados básicos. Dependem da linguagem de programação utilizada.

- São aqueles a partir dos quais podemos definir os demais tipos ou organizações de informações, quase sempre mais complexas.

- Possuem operações que podem ser realizadas com dados desse tipo. Exemplos: adição, subtração, multiplicação, divisão etc.

tipos de dados primitivos

→ **tipos de dados estruturados**

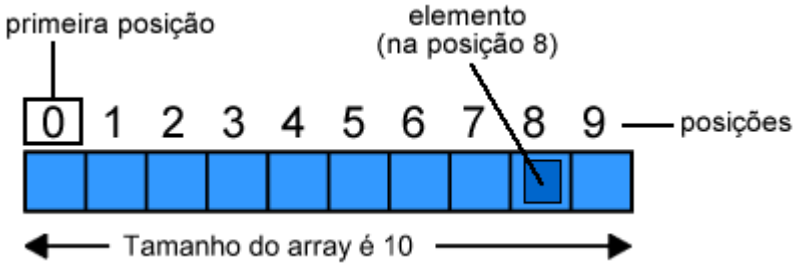
→ **estruturas de dados**

Exemplo: tipos de dados primitivos em Java

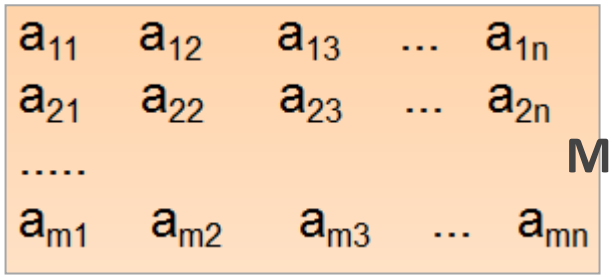
Tipo	Descrição
boolean	Pode assumir o valor true ou o valor false
char	Serve para a armazenagem de um valor alfanumérico. Também pode ser usado como um dado inteiro com valores na faixa entre 0 e 65535.
byte	Inteiro de 8 bits (1 byte) em notação de complemento de dois. Pode assumir valores entre -2^7=-128 e 2^7-1=127.
short	Inteiro de 16 bits em notação de complemento de dois. Os valores possíves cobrem a faixa de -2^{15}=-32.768 a 2^{15}-1=32.767
int	Inteiro de 32 bits (4 bytes) em notação de complemento de dois. Pode assumir valores entre -2^{31} = -2.147.483.648 e 2^{31}-1 = 2.147.483.647.
long	Inteiro de 64 bits (8 bytes) em notação de complemento de dois. Pode assumir valores entre -2^{63} e 2^{63}-1.
float	Representa números em notação de ponto flutuante normalizada em precisão simples de 32 bits em conformidade com a norma IEEE 754-1985. O menor valor positivo representável por esse tipo é 1.40239846e-46 e o maior é 3.40282347e+38
double	Representa números em notação de ponto flutuante normalizada em precisão dupla de 64 bits em conformidade com a norma IEEE 754-1985. O menor valor positivo representável é 4.94065645841246544e-324 e o maior é 1.7976931348623157e+308

Tipos de dados estruturados

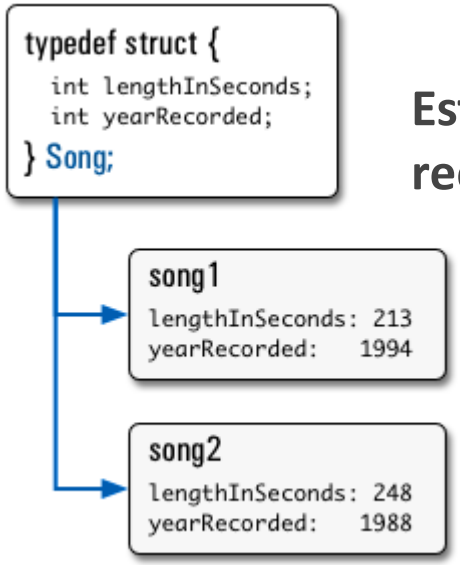
- São organizações de dados com determinada estrutura ou organização, obtidas normalmente a partir dos tipos de dados primitivos (mais poderiam guardar itens mais complexos, objetos).
- Frequentemente as linguagens de programação fornecem alguns tipos estruturados. Exemplos:



Vetores (lembre Java e JavaScript)



Matrizes



Estruturas, records (registros)

Listas em Python

lista = [4.0, 9.2, 3.1, 9.1, 1.5, 3.2, 2.6]

Abstração (antes de estudar o conceito de TAD)

- Um computador ainda tem sérias limitações físicas para representar a complexidade do mundo real. Nosso mundo contêm ricos detalhes que não podem ser inseridos ou representados diretamente no computador.
- É **abstraindo** nossa realidade que **podemos capturar o que existe de mais relevante ou importante** em uma situação real, tornando possível a construção de **modelos** que podem ser implementados nos computadores por meio de uma linguagem de programação.

- **abstração -** processo mental que consiste em isolar um aspecto determinado de um estado de coisas relativamente complexo, a fim de simplificar a sua avaliação, classificação ou para permitir a comunicação do mesmo

  Dicionário Houaiss.

Tipo Abstrato de Dados (TAD)

- Um **TAD** apresenta, em forma geral, um conjunto de **dados** e uma lista de **operações** que atuam sobre esses dados, mas sem entrar em detalhes.
- Um **TAD** é uma **descrição dos elementos mais relevantes ou importantes de um componente de software, de uma estrutura de dados etc**.
- O TAD será uma primeira etapa de projeto e em uma segunda etapa esse TAD será implementado detalhadamente.
- Em um TAD mostramos uma pequena descrição de cada método e, opcionalmente, adicionamos pré-condições e pós-condições para as diferentes operações de um TAD.

Neste material analisaremos as diferenças que existem entre estes dois conceitos: Tipo Abstrato de Dados (TAD) e Estrutura de Dados (ED).

Tipo Abstrato de Dados (TAD) como contrato

O **TAD** deverá especificar um "**contrato**" (digamos que é um **compromisso**) para que futuros implementadores desenvolvam adequadamente as estruturas de dados com base nesse modelo.

Este contrato (no **TAD**) **considera**:

- os nomes das operações deste TAD;
- os tipos de dados retornados e tipos de dados dos parâmetros;
- o comportamento ou descrição geral das operações;
- opcionalmente: as pré-condições e pós-condições das operações do TAD.

Veja que o contrato (TAD), normalmente, **não considera**:

- a representação específica dos dados para implementar este TAD;
- nem especifica os algoritmos detalhados para efetuar a implementação.

Vantagens de utilizar um Tipo Abstrato de Dados (TAD)

- Vantagens de utilizar TAD:
  - Especificar a integridade e a consistência entre os dados.
  - Ao estabelecer critérios de desenho, facilitam a manutenção.
  - Reutilização (porque podemos implementar esse TAD em várias formas diferentes).
  - Um TAD é um compromisso que, na próxima etapa de desenvolvimento do software, os programadores deverão respeitar.

- Um TAD está desvinculado de sua implementação, ou seja, quando definimos um TAD estamos preocupados com o que ele faz e não como ele faz (não entramos em detalhes de como serão implementados os dados e as operações).

Tipo Abstrato de Dados (TAD) nos livros

Universidade Cruzeiro do Sul

Goodrich e Tamassia, Estruturas de Dados e Algoritmos em Java

"**Um TAD é um modelo matemático** de estruturas de dados que especifica os tipos dos dados armazenados, as operações definidas sobre estes dados e os tipos de parâmetros destas operações. Um TAD define o que cada operação faz, mas não como o faz. **Em Java, um TAD pode ser expresso por uma interface*, que é uma simples lista de declarações de métodos.**"

"Um TAD é materializado por uma estrutura de dados concreta que, em Java, é modelada por uma **classe**. **Uma classe define os dados que serão armazenados e as operações suportadas**..."

> Analisar a diferença entre **declarar** e **definir**. Lembrar os conceitos de: interfaces, classes abstratas e classes concretas. Veja o **exemplo** no projeto **InterfaceClasse.zip**.

Shaffer, Data Structures and Algorithm Analysis

"Um **tipo abstrato de dados** (**TAD**) é a realização de um tipo de dados como um componente de software. A **interface do TAD** é definida em termos de um tipo e um conjunto de operações sobre esse tipo..."

"Uma **estrutura de dados** é a implementação de um TAD."

Exemplo: um TAD chamado TAD_NotaFinal

```
public interface TAD_NotaFinal {

   public float calculaNotaFinal();  //poderia ter parâmetros
        //Calcula a nota final do aluno.
        // pré-condições:
        //    cada nota deverá ser >= 0
        //    a soma das notas deverá ser <= 10
        // pós-condições:
        //    a nota final será a soma de todas as notas

   public float calculaMediaDasNotas();
        //Calcula a média das notas do aluno.
        //pré-condições:
        //    cada nota deverá ser >= 0
        //    a soma das notas deverá ser <= 10
        //pós-condições:
        //    a média será a soma de todas as notas
        //    dividida pela quantidade de notas
}
```

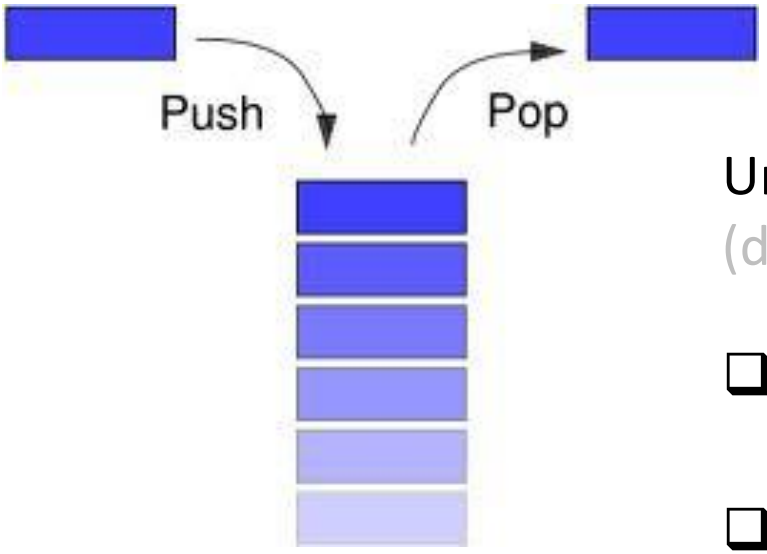
Exemplo: uma classe que implementa o TAD

```
public class EAD_NotaFinal  implements TAD_NotaFinal {

   private float n1 = -1, n2 = -1, n3 = -1;

   public float calculaNotaFinal() {
      if( n1>=0 && n2>=0 && n3>=0 && (n1+n2+n3)<=10 )
         return (n1+n2+n3); else return -1;
   }

   public float calculaMediaDasNotas() {
     if( n1>=0 && n2>=0 && n3>=0 && (n1+n2+n3)<=10 )
         return (n1+n2+n3)/3; else return -1;
   }

   //... construtores e outros métodos

}
```

Pilhas

- Uma pilha utiliza um tipo de dado abstrato de dados (TAD), que define as características principais para implementação de uma estrutura de dados conhecida pelo critério de inserção/retirada dos elementos: **Last In, First Out** (**LIFO**), em português: o último que entrou será o primeiro a sair;
- Uma pilha mantém seus elementos em sequência;
- Caracterizada pela restrição de acesso apenas ao último elemento inserido (acessar somente o elemento no topo);
- **Inserções** e **remoções** são feitas numa mesma extremidade denominada **topo**.

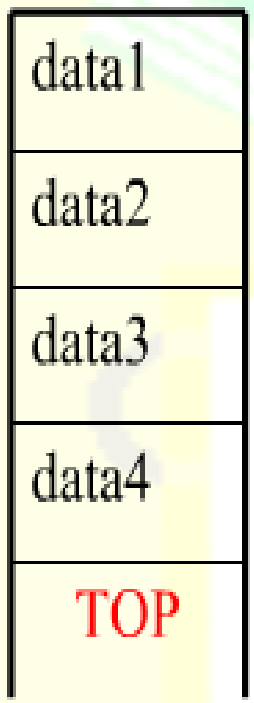
- Resumindo, em uma pilha temos acesso somente ao topo da pilha, ou seja, quando **inserimos** um elemento na pilha (push) o colocamos no topo e se **excluimos** um elemento da pilha (retirar, pop), só podemos excluir aquele que está no topo.

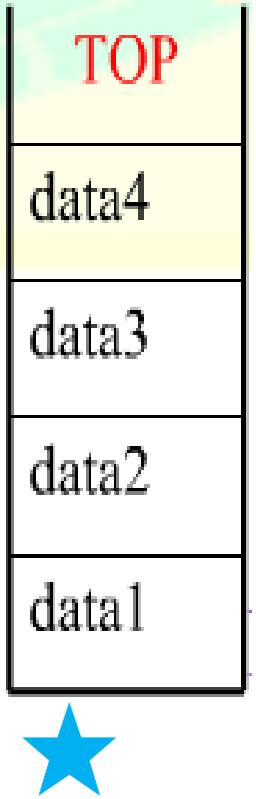


Uma pilha possui três operações básicas (depois adicionaremos outras operações):

- **top**: acessa (retorna, sem eliminar) o elemento posicionado no topo;
- **push**: insere um novo elemento no topo da pilha;
- **pop**: remove e retorna o elemento que se encontra no topo da pilha.

Quatro representações gráficas de uma pilha





são as preferidas, mas intuitivas

Universidade Cruzeiro do Sul

Os estados de uma pilha (de uma pilha guardada em um vetor)

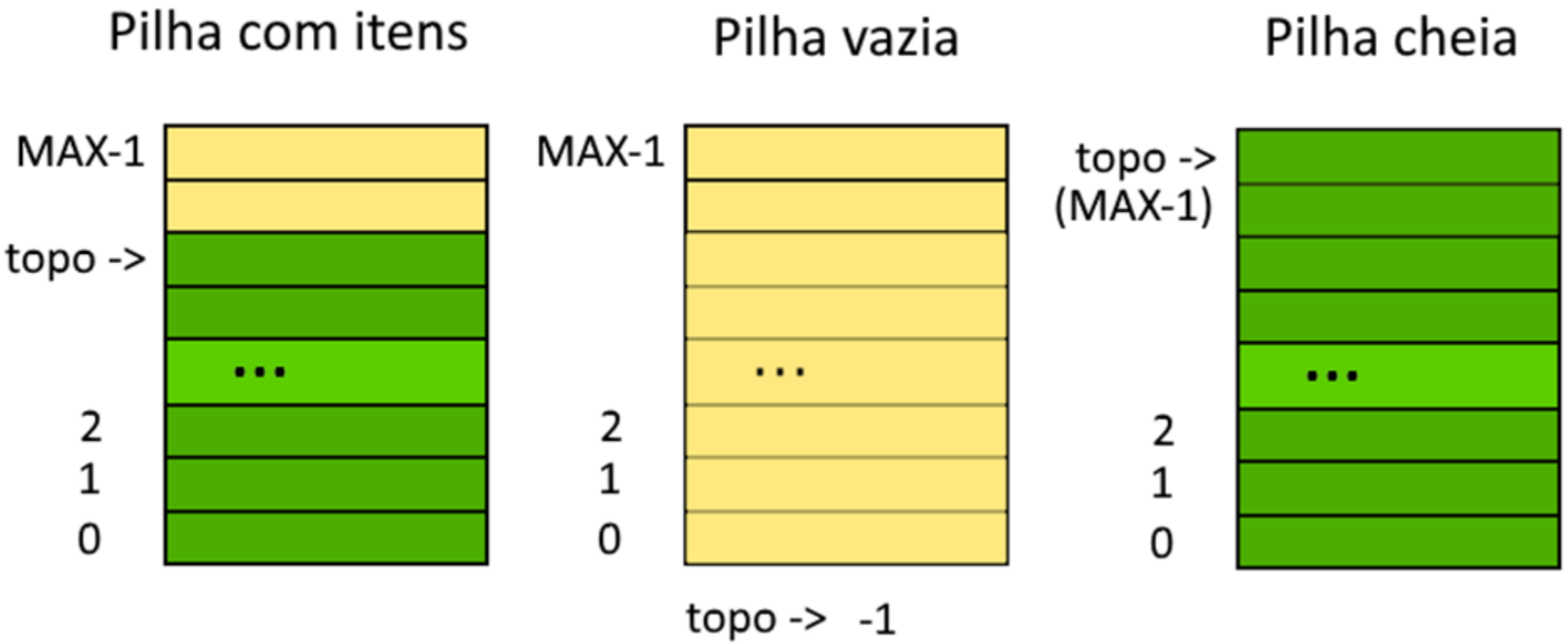


Tabela com exemplos - uma pilha P

--------	P:[ ]	Inicialmente a Pilha está Vazia
push(a)	P:[a]	Insere o elemento a na Pilha P
push(b)	P:[a, b]	Insere o elemento b na Pilha P. Note que agora o topo é b
push(c)	P:[a, b, c]	Insere o elemento c na Pilha P. Agora o topo contém c
pop()	P:[a,b]	Remove o elemento do topo e retorna o conteúdo. No caso c
push(d)	P:[a, b, d]	Insere o elemento d no topo.
top()	P:[a, b, d]	Não altera a Pilha, apenas retorna o valor do topo da Pilha (d)
push(top())	P:[a, b, d, d]	Insere no topo mais uma cópia do topo

Veja os exercícios em **Exercícios-1-e-2-Pilhas.txt**

Reflexões

Universidade Cruzeiro do Sul

- Pense em uma pilha de pratos, onde sempre colocamos o prato no topo da pilha e retiramos também do topo.
- Não é possível inserir pratos indefinidamente, pois existe um limite (no caso o teto) e nem tirar pratos indefinidamente (pilha vazia).
- Na implementação computacional também existem tais limites:
    - a quantidade de elementos não pode exceder a quantidade de memória alocada;
    - não podemos retirar elementos de uma pilha vazia.
- Para suprir as restrições anteriores, precisamos de mais operações:
    - **isEmpty** - para verificar se a pilha está vazia;
    - **isFull** - para verificar se a pilha está "cheia" (se chegou no limite da capacidade, se acabou a memória);
    - **toString** - para retornar os itens (os objetos) da pilha.

Exemplos de utilização de pilhas

- Análise de expressões matemáticas:
  - identifica se os elementos separadores de abertura '[', '{' e '(' são encerrados de forma correta com os respectivos elementos separadores de encerramento ')', '}' e ']'.
- Avaliação de uma expressão pós-fixa:
  - Infixa: **((1 + 2) * 4) + 3**
  - Pós-fixa: **1 2 + 4 * 3 +**
  - Utilizando uma pilha consideramos:
    - empilhar quando encontrar um operando;
    - desempilhar dois operandos e achar o valor quando encontrar uma operação;
    - empilhar o resultado.
- Chamadas de métodos e retornos (nos compiladores).
- Histórico de navegadores (browsers) tem 'comportamento' de pilha.

Um TAD chamado TAD_Pilha (1)

```
public interface TAD_Pilha {

   public Object push (Object x);
        //Método para inserir um novo item x no topo da pilha.
        // pré-condições:
        //    a pilha deverá ter espaço para inserir: verificar !isFull()
        //    o objeto a inserir x não poderá ser nulo
        // pós-condições:
        //    retornará diferente de nulo se a inserção foi possível
        //    o topo será modificado, apontando ao novo item

   public Object pop ();
        //Método para retirar o item que se encontra no topo da pilha.
        // pré-condições:
        //    não poderá retirar da pilha se estiver vazia: verificar !isEmpty()
        // pós-condições:
        //    retornará o objeto que se encontra no topo da pilha
        //    retornará nulo se a pila está vazia
        //   o topo será modificado, apontando ao próximo item
```

Obs: estas operações possuem tempo de execução constante em relação à entrada: **O(1)**.

Um TAD chamado TAD_Pilha (2)

```
    public boolean isEmpty ();
        //Método que verifica se a pilha está vazia.
        //Exemplo: não poderão ser retirados valores se a pilha estiver vazia.

    public boolean isFull ();
        //Método que verifica se a pilha está cheia, ou seja, não existe
        //memória para adicionar mais elementos.

    public Object top();
       //Método para retornar o item que se encontra no topo da
       //pilha, sem eliminar o mesmo
       // pré-condições:
       //    não poderá retornar o item se a pilha se estiver vazia
       // pós-condições:
       //    retornará o objeto que se encontra no topo da pilha
       //    retornará nulo se a pila está vazia
       //    o topo não será modificado, nem a pilha

    public String toString();   //retorna os objetos contidos na pilha
}
```

Obs: estas operações possuem tempo de execução constante em relação à entrada: **O(1)**. Somente toString() será de **O(n)**.

Implementação estática sequencial de pilhas

Estática

A alocação ou reserva de memória pode ser estática ou dinâmica. Neste material estudaremos pilhas com alocação estática de memória.
Entendemos o termo *estática* como uma quantidade máxima fixa de memória reservada, seja na compilação ou na execução do programa.

Sequencial

Os elementos de uma pilha poderiam ser colocados sequencialmente (um depois do outro, na sequência, dentro de um vetor ou uma lista, por exemplo) ou encadeados/enlaçados/ligados na memória interna. Nesta primeira etapa estudaremos pilhas com elementos colocados em forma sequencial.

A seguir, mostraremos uma estrutura de dados **Pilha**, como uma implementação **estática** e **sequencial** do **TAD_Pilha** anterior, utilizando a linguagem Java, com um vetor de objetos. Em próximas aulas veremos outra implementação: **dinâmica** e **encadeada**.

Implementação estática sequencial de pilhas em Java

```
package pilhaestaticasequencial;

public class Pilha implements TAD_Pilha {
        private int topo;              //topo da pilha
        private int MAX;               //tamanho máximo da pilha (do vetor)
        private Object memo[];         //elementos da pilha
```

Utilizamos um vetor de **Object**, que é uma classe geral de onde todas as classes herdam características. Portanto, podemos guardar qualquer objeto dentro desse vetor, como objetos Integer, Float, String etc. ou de outra classe qualquer (Automovel, Trabalhador, Aluno).

```
//Método construtor que inicializa a pilha com um tamanho máximo desejado
public Pilha (int qtde)
{
        topo = -1;   // esta é a convenção que utilizaremos para "pilha vazia"
        MAX = qtde;
        memo = new Object[MAX];
}

//Método que verifica se a pilha está Vazia
public boolean isEmpty ()
{
        if( topo == -1) return true; else return false;
}

//Método que verifica se a pilha está cheia (o vetor que guarda a pilha cheio)
public boolean isFull ()
{
        return (topo==MAX-1); (ou utilizando um comando if)
}
```

```java
//Método para inserir um valor na Pilha
public Object push(Object x)
{
        if( !isFull() && x != null ) {
                memo[++topo]=x;
                return x;
        }
        else return null;
}
```

```java
//Método para retornar o topo da Pilha e remove-lo
public Object pop()
{
   if(!isEmpty())
      return memo[topo--];
   else
      return null;
}
```

Obs: os métodos mostrados retornam **null** nos casos de situações anormais (erradas). Outra abordagem seria que os métodos lancem uma exception.

Observe a utilização correta do operador ++ pré-fixo e também do operador -- pós-fixo.

memo[++topo]=x;

é equivalente a:

topo = topo + 1;
memo[topo] = x;

return memo[topo--];

é equivalente a:

Object temp = memo[topo]
topo = topo - 1;
return temp;

```java
//Método que retorna o topo da pilha sem removê-lo
public Object top() {
    if(!isEmpty()) return memo[topo];
                   else return null;
}


//Método para retornar o conteúdo da Pilha
public String toString () {
        if(!isEmpty()) {
                String msg = "";
                for(int i=0; i<=topo; i++) {
                  msg += memo[i].toString();
                  if(i != topo) msg += ", ";
                }
                return ("P: [ " + msg + " ]");
        }
        else return "Pilha Vazia!";
}
```

Exemplo de operações com uma pilha

Universidade Cruzeiro do Sul

```
public static void main(String[] args) {
    Pilha p = new Pilha(10);
    p.push("Lima");
    p.push("Roma");
    p.push("Brasília");
    p.push("Washington");
    p.push("Havana");
    p.push("Buenos Aires");
    p.push("Bogotá");
    System.out.println("Pilha inicial:");
    System.out.println(p.toString());
    System.out.println("Retiremos dois elementos da pilha:");
    System.out.println(p.pop());
    System.out.println(p.pop());
    System.out.println(p.toString());
    System.out.println("Adicionamos Montevidéu:");
    p.push("Montevidéu");
    System.out.println(p.toString());
    System.out.println("No topo está: \n" + p.top() );
    System.out.println("Retirando e esvaziando a pilha:");
    while ( !p.isEmpty() ) {
       System.out.println(p.pop());
    }
    if ( p.isEmpty() ) System.out.println("Impossível retirar da pilha: Pilha vazia.");
  }
```

```
Pilha inicial:
P: [ Lima, Roma, Brasília, Washington, Havana, Buenos Aires, Bogotá ]
Retiremos dois elementos da pilha:
Bogotá
Buenos Aires
P: [ Lima, Roma, Brasília, Washington, Havana ]
Adicionamos Montevidéu:
P: [ Lima, Roma, Brasília, Washington, Havana, Montevidéu ]
No topo está:
Montevidéu
Retirando e esvaziando a pilha:
Montevidéu
Havana
Washington
Brasília
Roma
Lima
Impossível retirar da pilha. Pilha vazia.
```

Solução completa no projeto (NetBeans) PilhaEstaticaSequencial

Mais sobre pilhas e tipos de dados

- Uma estrutura de dados (como a pilha) pode ser capaz de guardar diferentes tipos de dados: podemos criar pilhas com objetos inteiros, objetos reais, de pratos, de alunos, de veículos, de trabalhadores etc. e inclusive de 'objetos polimorfos';
- Em Java, todos as classes são herdeiras de uma classe chamada Object como, por exemplo, as classes Float, Integer, String etc. Portanto, utilizamos na implementação anterior um Object, que consegue armazenar uma referência a qualquer objeto (endereço de um objeto de qualquer classe).

Sobre conversões

Exemplo: efetuando "casting" para a classe Trabalhador.

```
while ( !p.isEmpty()  )  {
        Trabalhador tr = (Trabalhador) p.pop();
        //o casting anterior foi utilizado porque pop() retorna Object
        System.out.println("Dados do trabalhador: \n" + tr.toString());
        if(tr.getSalario() > 4000.00f) {  //podemos usar métodos da classe Trabalhador
            System.out.println("O trabalhador ganha mais de R$ 4.000,00");
       }
}
```

Comparando objetos

- Um dos métodos herdados da classe Object é a função **toString()**, que converte os dados do objeto em uma String. As classes derivadas normalmente redefinem esta função toString().
- toString() é útil para visualizar objetos e também para poder comparar elementos, visto que com o sinal de igual não é possível.

```
Object a, b;

if(a == b)... ;  //está errado, porque só compararia as referências

//está correto desta forma:
if(a.toString().equals(b.toString()) {
   ...
}
```

Exercício - para entregar

Criar uma **pilha** com objetos da classe **Veículo**. Considere os atributos: placa, marca, modelo e ano de fabricação para a classe Veículo.

- Leia ou crie vários objetos de veículos e insira os mesmos na pilha.
- Retire os objetos da pilha e mostre seus dados.

Exercício proposto, apenas para praticar ou refletir

Modifique o exercício anterior de forma a utilizar um **ArrayList** para guardar um grupo de objetos da classe Trabalhador em uma pilha, em lugar do vetor estático que foi utilizado na implementação mostrada neste material para a classe **Pilha**.

Bibliografia oficial da disciplina

BIBLIOGRAFIA BÁSICA	BIBLIOGRAFIA COMPLEMENTAR
CORMEN, T. H.; et al. Algoritmos: teoria e prática. 3. ed. Rio de Janeiro: Elsevier, 2012. GOODRICH, M. T.; TAMASSIA, R. Estrutura de dados e algoritmos em java. 5. ed. Porto Alegre: Bookman, 2013. (livro físico e e-book) CURY, T. E., BARRETO, J. S., SARAIVA, M. O., et al. Estrutura de Dados (1. ed.) ISBN 9788595024328, Porto Alegre: SAGAH, 2018 (e-book)	ASCENCIO, A. F. G.; ARAÚJO, G. S. Estruturas de Dados: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Hall, 2010. (eBook) PUGA, S.; RISSETTI, G. Lógica de programação e estruturas de dados, com aplicações em Java. 3. ed. São Paulo: Pearson Education do Brasil, 2016. (eBook) DEITEL, P.;DEITEL, H. Java como programar. 10. ed. São Paulo: Pearson Education do Brasil, 2017. (eBook) BARNES, D. J.; KOLLING, M. Programação Orientada a Objetos com Java: uma introdução prática usando o Blue J. São Paulo: Pearson Prentice Hall, 2004. (eBook) BORIN, V. POZZOBON. Estrutura de Dados. ISBN: 9786557451595, Edição: 1ª . Curitiba: Contentus, 2020 (e-book)

Knowledge Center. Stack data structure || Stacks || LIFO (Índia)
https://youtu.be/xC8AXn1TDA8