

Estrutura de dados

Conteúdo

- Filas **dinâmicas encadeadas**.



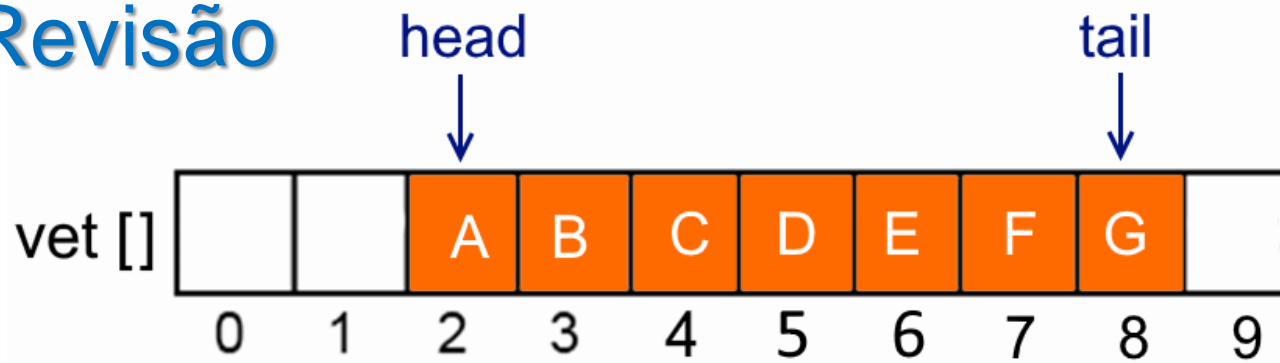
Autores

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

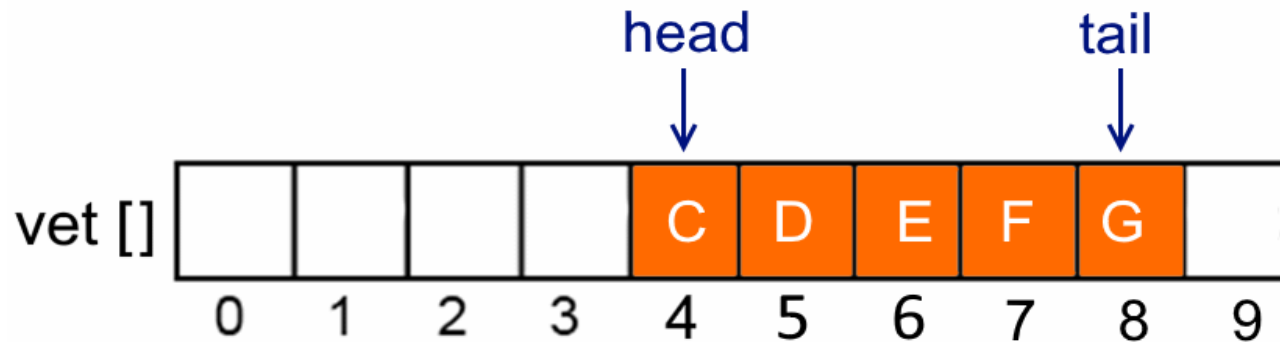
Fila Estática Sequencial

(revisão do design e da implementação anterior)

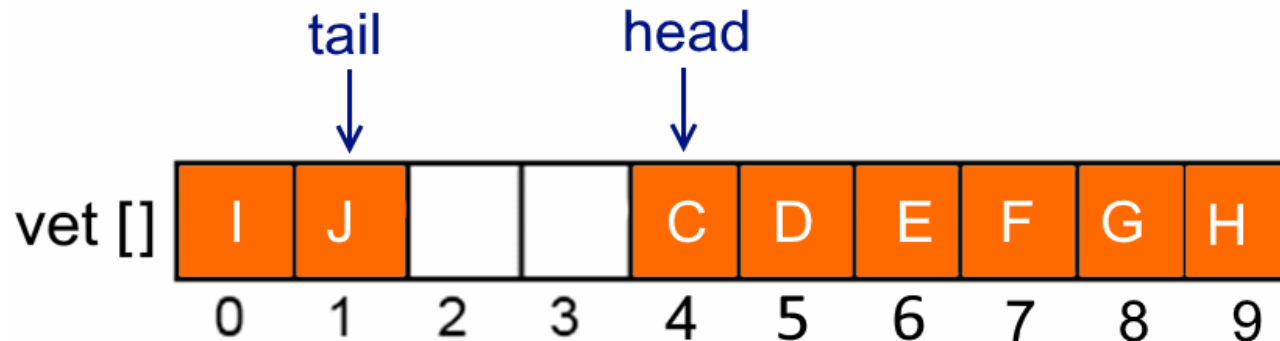
Revisão



Suponhamos uma pilha guardada no vetor `vet []` com este estado inicial.



Depois de retirar dois elementos, A e B, a fila ficaria assim.



Agora inserimos três elementos H, I e J.

Uma implementação de uma fila em um vetor, com funcionamento "circular", permite aproveitar toda a capacidade do mesmo.

TAD_Queue (o tipo abstrato de dados)

```
public interface TAD_Queue { // tipo abstrato de dado que descreve a Fila  
    //Retornam se a fila está vazia ou cheia:  
    public boolean isEmpty();  
    public boolean isFull(); //não será implementado na fila din. encadeada  
  
    //Insere um elemento no final da fila:  
    public Object enqueue(Object x);  
  
    //Remove um elemento do início da fila:  
    public Object dequeue();  
  
    //Retorna o objeto no início da fila (o primeiro da fila), sem eliminar:  
    public Object peek();  
  
    //Retorna o conteúdo (todos os elementos) da Queue:  
    public String toString();  
}
```

Implementação: os atributos da fila estática sequencial

- A implementação se encontrava dentro de um projeto NetBeans chamado FilaEstaticaSequencial.
- Atributos da classe para a ED Queue (implementação sequencial):

//Implementação de uma fila estática sequencial com funcionamento circular

```
public class Queue implements TAD_Queue {  
    private int total = 0; //total de elementos (convenção: 0 se a fila estiver vazia)  
    private int head = -1; //começo da queue (convenção: -1 se a fila estiver vazia)  
    private int tail = -1; //final da queue (convenção que usaremos: -1 se fila vazia)  
    private Object memo[]; //vetor para armazenar os objetos da fila  
    private int MAX; //capacidade máxima do vetor, diferente do atributo total
```

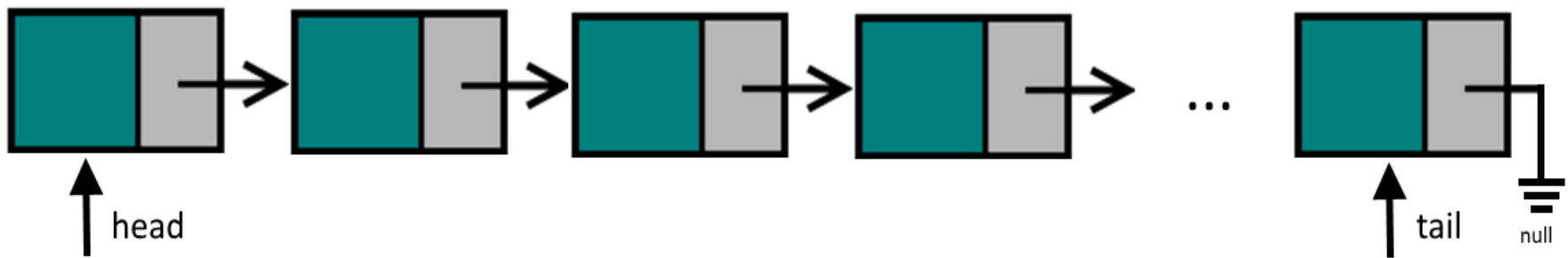
Fila (queue) dinâmica encadeada

(encadeada, enlaçada, ligada, linked)

Vantagens de utilizar uma fila dinâmica encadeada

- Alocaremos apenas a memória necessária para rodar a aplicação e solicitaremos mais memória à heap em tempo de execução, assim que for necessário inserir mais um elemento na fila (como em qualquer outra estrutura dinâmica).
- Não ocorre a desvantagem de uma fila estática sequencial, onde a memória do vetor é limitada e precisamos implementar uma fila circular para poder aproveitar todo seu espaço, com uma lógica mais complexa. Este inconveniente não ocorre aqui, pois a colocação dos elementos não é sequencial e sim encadeada, e a alocação de memória é dinâmica.

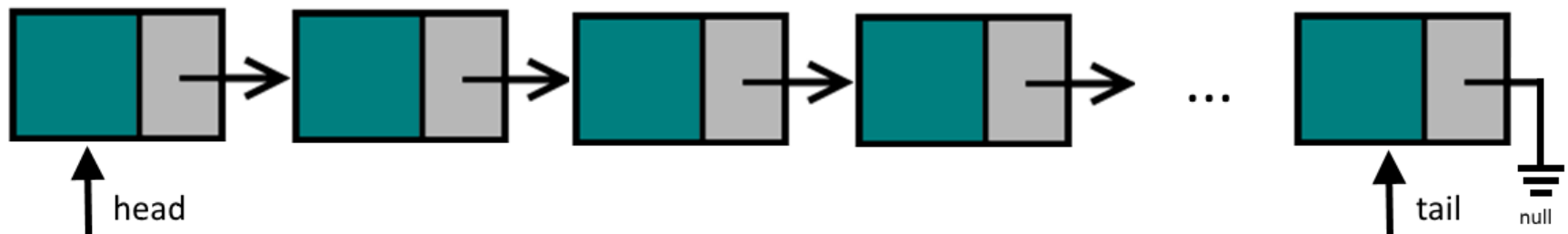
- Para criarmos uma fila dinâmica encadeada (enlaçada, ligada) usaremos uma estrutura onde **cada elemento da Fila será um nó (nodo)** contendo um dado (um objeto) e o endereço do próximo nó da Fila.



- A fila, por ser uma estrutura **FIFO** (o primeiro que entrou será o primeiro a sair, a ser atendido), deverá manipular as **duas extremidades** (começo e final, início e fim, **head** e **tail**), pois as inserções (**enqueue**) são feitas no final da fila e as remoções (**dequeue**) são realizadas no começo.
- Poderíamos usar um contador para armazenar a quantidade de elementos que a Fila possui. Na verdade, isto é opcional, a fila dinâmica encadeada pode ser implementada sem este contador.

Uma implementação de uma fila dinâmica encadeada

- utilizaremos nomes tradicionais em inglês
- implementada em projetos Java/NetBeans



```
public class Node {  
    private Object value; // valor do nodo  
    private Node next;    // enlace, endereço para acessar o próximo item  
  
    public Object getValue() { // retorna o valor do nodo  
        return value;  
    }  
  
    public void setValue(Object value) { // para alterar o valor do nodo  
        this.value = value;  
    }  
  
    public Node getNext() { // retorna o endereço do próximo item  
        return next;  
    }  
  
    public void setNext(Node next) { // para alterar o endereço do nodo  
        this.next = next;  
    }  
}
```

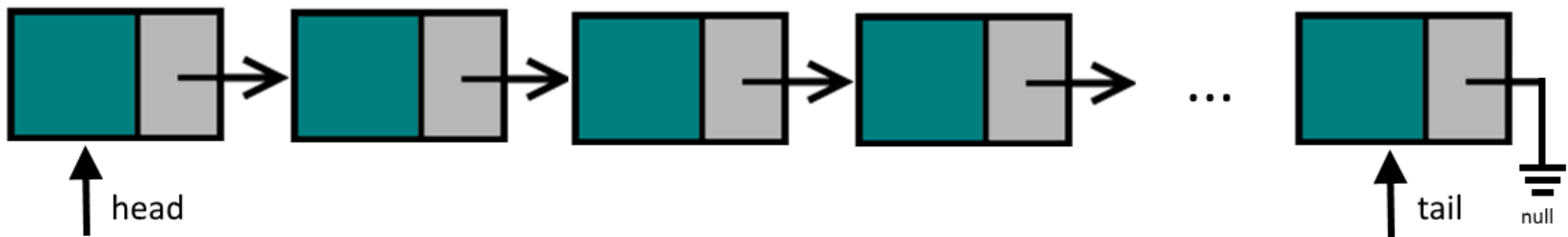


Um nodo (classe Node)

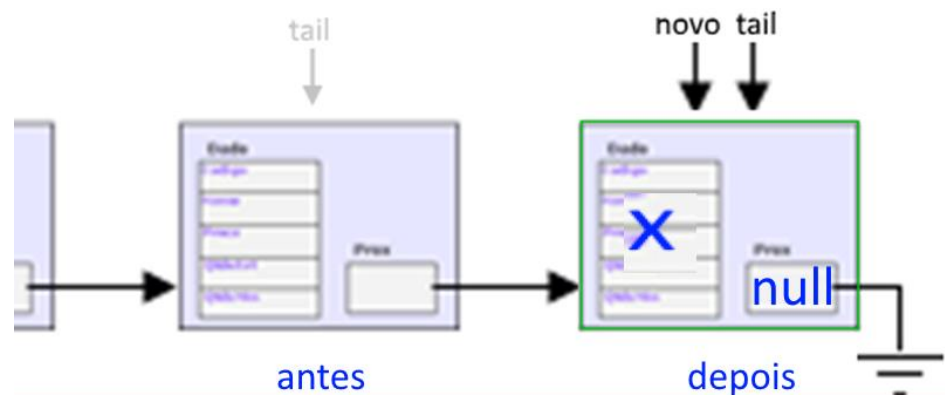
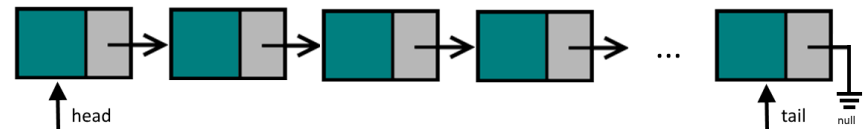
Revisão

```
class Queue implements TAD_Queue {  
    //em todos os métodos: quando uma operação não for possível,  
    //retornaremos null  
  
    private Node head=null, tail=null;  
    //convenção: ambos ponteiros serão null se a fila estiver vazia  
  
    public Queue() { // para construir uma fila vazia  
        head=null;  
        tail=null;  
    }  
  
    public boolean isEmpty () {  
        if(head == null) return true; else return false; //ou if (tail == null)  
    }  
}
```

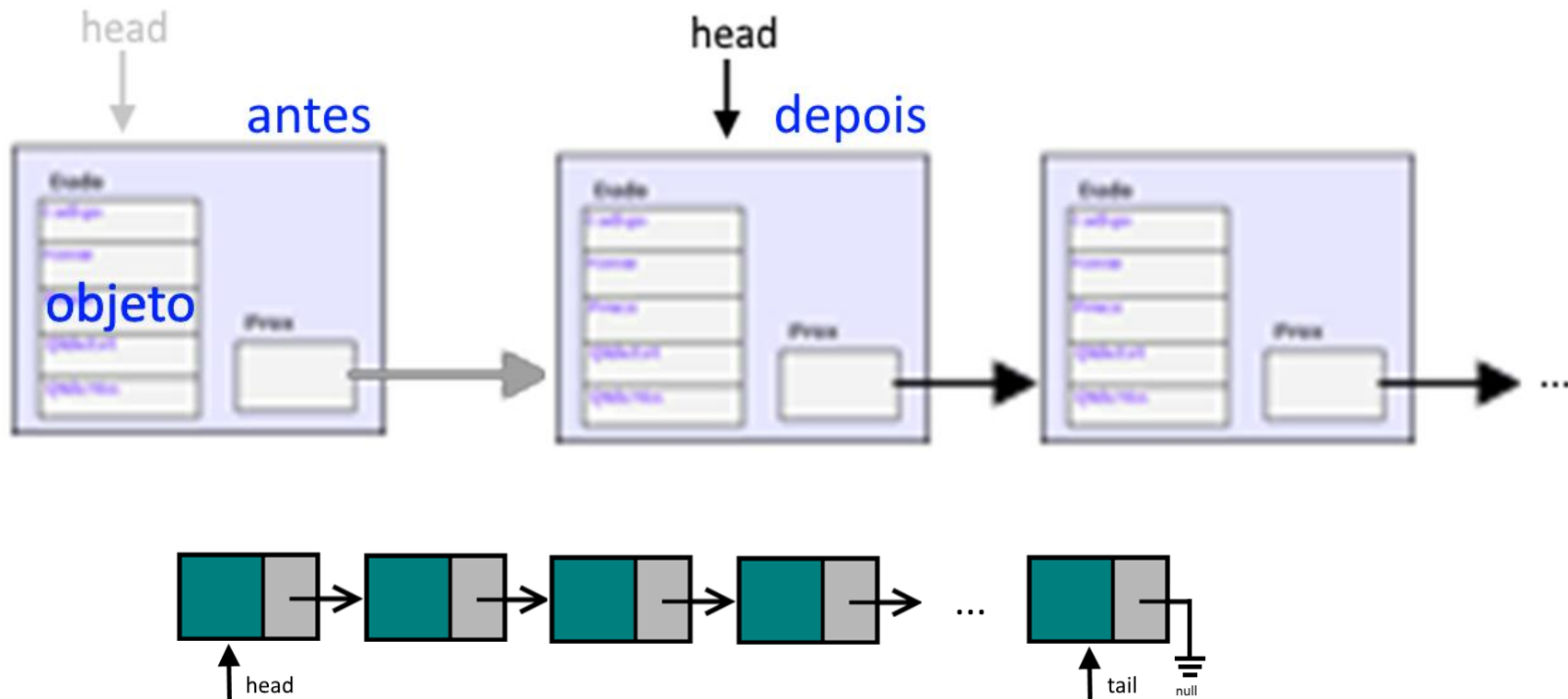
```
public Object peek () {  
    //retorna o objeto no início da fila (sem eliminar o mesmo),  
    //mas somente se a fila não estiver vazia:  
    if(head == null) return null; else return head.getValue();  
    //ou podemos usar: if(tail == null) ou if(isEmpty())  
}
```



```
public Object enqueue (Object x) {  
    if(x == null) return null; // não permitiremos inserir um objeto nulo  
    try { // verificamos se existe memória suficiente:  
        if(Runtime.getRuntime().freeMemory() < x.toString().length() + 1024) return null;  
        Node novo = new Node(); // alocamos memória para o novo nó  
        novo.setValue(x); // colocamos o objeto x no novo nodo  
        novo.setNext(null); // o enlace do novo nodo será nulo  
        if (tail == null) { // caso a fila estava vazia, head e tail apontarão para o novo/único nó:  
            head = novo; tail = novo;  
        }  
        else { // caso geral:  
            tail.setNext(novo); // a antiga cauda apontará para o novo nó  
            tail = novo; // a nova cauda é o novo nó  
        }  
        return x;  
    } catch (Exception ex) {  
        return null;  
    }  
}
```

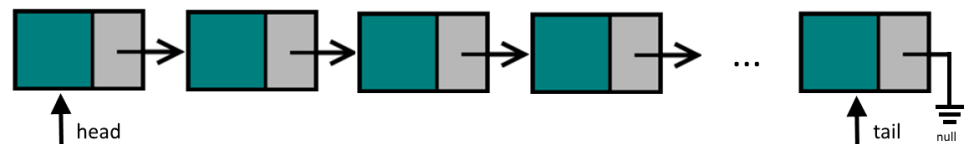


```
public Object dequeue () {  
    if (head == null) return null; // impossível retirar de uma fila vazia  
    Object objeto = head.getValue(); // pegamos o objeto na cabeça  
    head = head.getNext(); // avança a cabeça da fila  
    if( head == null) tail=null; // se a fila ficou vazia tail também será null  
    return objeto; // retornamos o objeto que estava na cabeça antiga  
}
```



```
public String toString () { //retorna o conteúdo (os objetos) da fila dinâmica

    if( !isEmpty() ) {
        String saida = "";
        Node aux = head; //ponteiro auxiliar começando na cabeça da fila
        while( aux!=null ) { //com o ponteiro aux percorremos a lista
            saida += aux.getValue().toString();
            aux = aux.getNext(); //avancamos o ponteiro
            // para separar objetos simples usamos , ou podemos separar com \n
            if ( aux != null ) saida += ", ";
        }
        return ( "f: [ " + saida + " ]" );
    }
    else return ( "f: [ ]" ); // fila vazia
}
```



Convenção utilizada para formatar a resposta ➔ f: [head, ... , tail]

```
public int size() { //retorna a quantidade de objetos na fila
    //count: temos que incrementar em enqueue e decrementar em dequeue
    //na versão em Fila1_v02.zip
    return count;
}
```

```
public Object[] toArray() {
    //retorna um vetor com os objetos guardados na fila, o que poderá
    //ser útil para ordenar dados ou processamento em geral
    if(isEmpty())return null; //operação impossível se a fila estiver vazia
    Object vet[] = new Object[count];
    Node aux = head;
    for(int i=0; i<count; i++) {
        vet[i] = aux.getValue();
        aux = aux.getNext();
    }
    return vet;
}
```

Veja estas duas implementações e testes no projeto **Fila1_v02.zip**

Testando o funcionamento de uma fila dinâmica encadeada

```
Queue fila = new Queue();
if( fila.isEmpty() ) {
    System.out.println("Incialmente: a fila está vazia");
}
fila.enqueue(5);
fila.enqueue(12);
fila.enqueue(6);
fila.enqueue(4);
if( !fila.isEmpty() ) {
    System.out.println("A fila agora não está vazia, temos os objetos: " + fila.toString());
}
System.out.println("O valor na cabeça (início) da fila é " + fila.peek());
while( !fila.isEmpty() ) {
    int valor = (Integer)fila.dequeue();
    System.out.println("Retirado o valor " + valor + " da fila");
}
System.out.println("Finalmente: " + fila.toString());
```

Testando: saída na tela - fila dinâmica encadeada

Inicialmente: a fila está vazia

Agora a fila não está vazia, temos os objetos: f: [5, 12, 6, 4]

O valor na cabeça (início) da fila é 5

Retirando o valor 5 da fila

Retirando o valor 12 da fila

Retirando o valor 6 da fila

Retirando o valor 4 da fila

Finalmente: f: []

Testando o funcionamento da fila dinâmica encadeada

```
Queue fila = new Queue();  
if (fila.isEmpty()) System.out.println("Inicialmente: a fila está vazia");  
  
fila.enqueue("mesa");  
fila.enqueue("janela");  
fila.enqueue("estante");  
fila.enqueue("cadeira");  
fila.enqueue("notebook");  
  
if (!fila.isEmpty()) {  
    System.out.println("Agora a fila não está vazia: " + fila.toString());  
}  
  
System.out.println("O valor na cabeça da fila é " + fila.peek());  
System.out.println("A fila guarda neste momento " + fila.size() + " objetos");
```

Testando o funcionamento da fila dinâmica encadeada

```
System.out.println("\nUm vetor com os objetos que se encontram na fila" );
Object vetor[] = fila.toArray();
for(int i=0; i < vetor.length; i++) {
    System.out.print(vetor[i] + " ");
}
System.out.println("\n");

while (!fila.isEmpty()) {
    Object obj = fila.dequeue();
    System.out.println("Retirando o valor " + obj.toString() + " da fila");
}

System.out.println("Finalmente: " + fila.toString());
```

Testando o funcionamento da fila dinâmica encadeada

Inicialmente: a fila está vazia

Agora a fila não está vazia: f: [mesa, janela, estante, cadeira, notebook
]

O valor na cabeça da fila é mesa

A fila guarda neste momento 5 objetos

Um vetor com os objetos que se encontram na fila

mesa janela estante cadeira notebook

Retirando o valor mesa da fila

Retirando o valor janela da fila

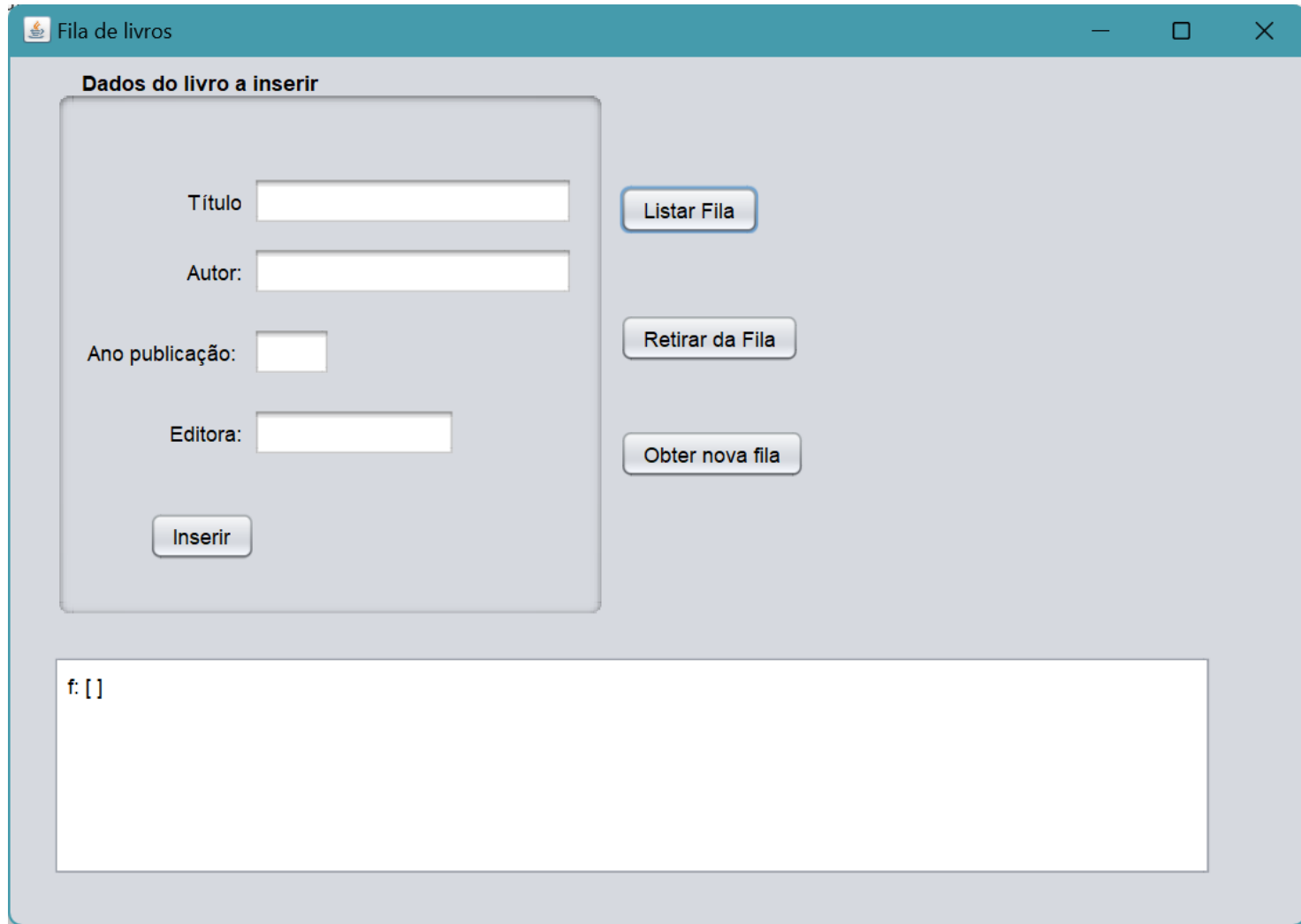
Retirando o valor estante da fila

Retirando o valor cadeira da fila

Retirando o valor notebook da fila

Finalmente: f: []

Exercício para praticar (não precisa entregar)



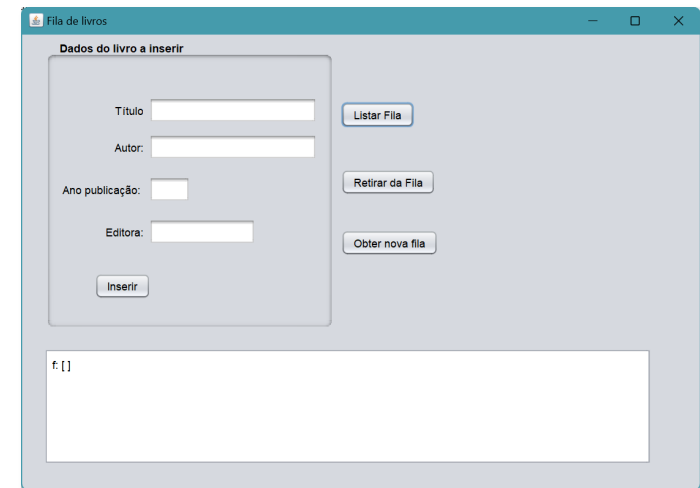
The screenshot shows a Java Swing window titled "Fila de livros". Inside the window, there is a section titled "Dados do livro a inserir" which contains four text input fields: "Título", "Autor:", "Ano publicação:", and "Editora:". Below these fields is an "Inserir" button. To the right of the input fields are three buttons: "Listar Fila", "Retirar da Fila", and "Obter nova fila". At the bottom of the window is a large text area labeled "f: []", which is currently empty.



A solução está disponibilizada em [ExercicioFilaLivros.zip](#).

Exercício para praticar (não precisa entregar)

- Criar uma fila dinâmica com dados de livros. Utilize como base uma classe Livro com os atributos: título, autor, ano de publicação e editora.
- Insira livros na fila, utilizando uma interface gráfica adequada.
- Implemente a lógica de um botão que, quando clicado, mostre todos os livros que se encontram na fila.
- Implemente a lógica de um botão para extrair um livro da fila (aquele que se encontra na cabeça da fila) e mostre seus dados na tela.
- Implemente um método que receba uma fila de livros, retire os mesmos e crie uma nova fila apenas com os livros publicados depois de 2002. Este método deverá deixar a fila original como estava antes.
- Ao clicar em um botão, mostre na tela a fila que foi criada com o método anterior.



Exercício 1 - para entregar em arquivo .txt, .doc, .jpg ou .png

Seja um fila dinâmica ligada F, qual será o estado final da fila F depois de executar os comandos a seguir?

Queue F = ["livro", "caneta", "borracha"]

Object ob = F.dequeue()

F.enqueue("clip")

F.enqueue(F.peek())

F.enqueue(ob)

Convenção utilizada → f: [head, ... , tail]

Exercício 2 - **para entregar** - classe Compra - com pilha e fila

Um app de compras guardará as compras efetuadas (botão **Guardar**) em uma ordem que permita retirar (cancelar por decisão do comprador) apenas a última compra que foi efetuada.

Outro botão, **Cancelar**, implementará esta lógica de cancelamento, mostrando os dados da compra que foi cancelada.

Na classe **Compra**, considere os atributos: nomeComprador, nomeProduto e valorProduto. Implemete construtores, get/set, toString.

Depois de um período de tempo (será a lógica do botão **Transferir**), todas as compras efetuadas serão colocadas em uma segunda estrutura de dados. Esta segunda estrutura deverá priorizar para atenção as solicitações de compras mais antigas, de forma a retirar a primeira que foi solicitada. Um último botão, **Atender**, implementará a lógica de retirar, mostrando os dados da compra priorizada que foi atendida.

Solução completa, posteriormente, no projeto PilhaFilaCompras, mas vamos demonstrar agora.

Bibliografia (oficial) para a disciplina

BIBLIOGRAFIA BÁSICA	BIBLIOGRAFIA COMPLEMENTAR
<p>CORMEN, T. H.; et al. Algoritmos: teoria e prática. 3. ed. Rio de Janeiro: Elsevier, 2012.</p> <p>GOODRICH, M. T.; TAMASSIA, R. Estrutura de dados e algoritmos em java. 5. ed. Porto Alegre: Bookman, 2013. (livro físico e e-book)</p> <p>CURY, T. E., BARRETO, J. S., SARAIVA, M. O., et al. Estrutura de Dados (1. ed.) ISBN 9788595024328, Porto Alegre: SAGAH, 2018 (e-book)</p>	<p>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)</p> <p>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)</p> <p>DEITEL, P.; DEITEL, H. Java como programar. 10. ed. São Paulo: Pearson Education do Brasil, 2017. (eBook)</p> <p>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)</p> <p>BORIN, V. POZZOBON. Estrutura de Dados. ISBN: 9786557451595, Edição: 1ª. Curitiba: Contentus, 2020 (e-book)</p>