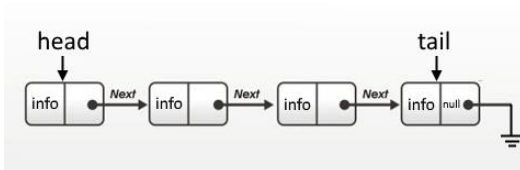


Estrutura de dados

Conteúdo

- Listas dinâmicas encadeadas.
Uma implementação em Java.
- A classe LinkedList da biblioteca Java da Oracle.

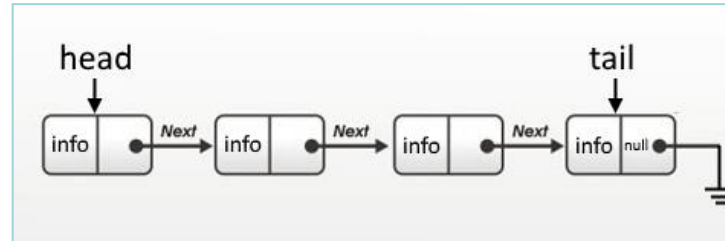


Elaboração:

Prof. Manuel Fdez. Paradela Ledón.

Listas ligadas dinâmicas

(lista geral simplesmente encadeada, ligada, enlaçada)



- Apresentamos neste material sugestões de operações e implementação, mas lembremos a classe LinkedList já existe em bibliotecas como, por exemplo, em Java/Oracle e C#/Microsoft.
- Normalmente temos dois pontos de acesso: o começo da lista (**head**) e o final (**tail**).
- Podemos achar bastante parecido com uma fila, mas: **são permitidas inserções e remoções em qualquer posição da lista (no início, no fim, no meio...), alterações dos itens, busca de itens na lista etc.** Ou seja, não existem as limitações das estruturas FIFO e LIFO!
- Dinâmica: observe, também, que não existe a restrição das estruturas estáticas sequenciais quanto aos limites do tamanho do vetor ou qualquer outra estrutura utilizada para armazenar os elementos.

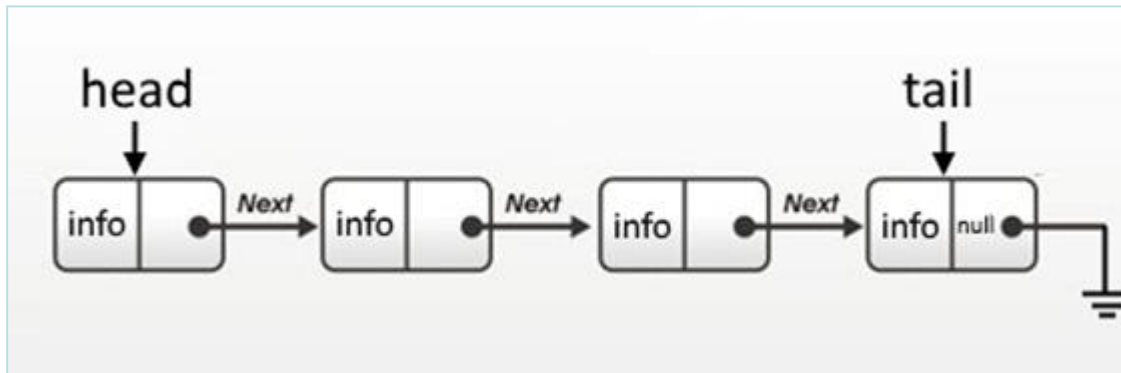


Um nodo (classe Node)

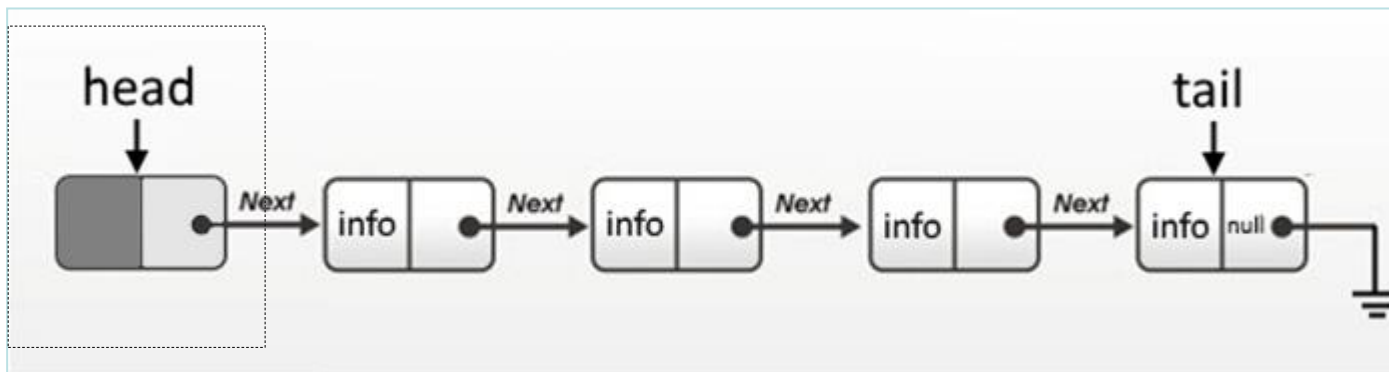
Utilizaremos
novamente a
mesma classe **Node**

```
public class Node {  
  
    private Object value;  
    private Node next;  
  
    public Object getValue() {  
        return value;  
    }  
  
    public void setValue(Object value) {  
        this.value = value;  
    }  
  
    public Node getNext() {  
        return next;  
    }  
  
    public void setNext(Node next) {  
        this.next = next;  
    }  
  
}
```

Listas ligadas dinâmicas sem ou com **nó cabeça**



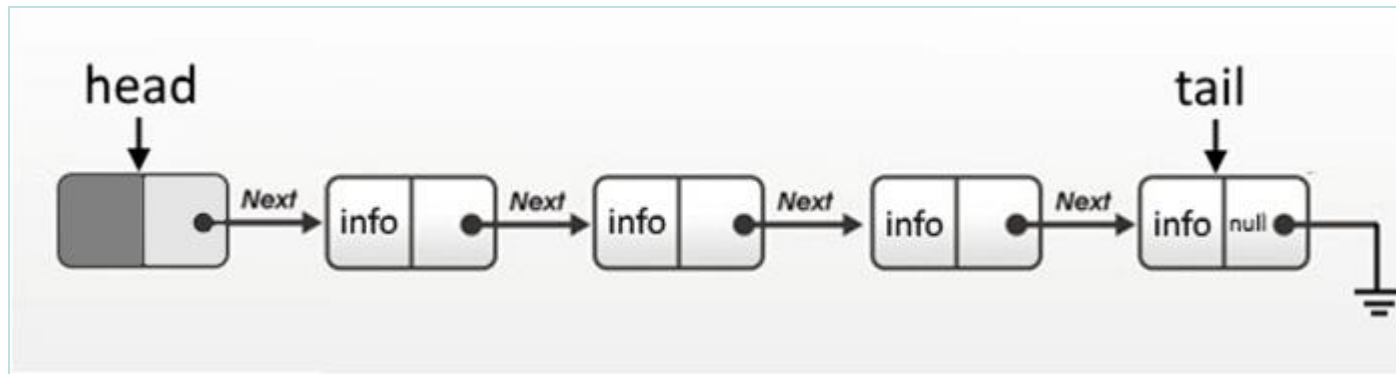
sem nó cabeça



com **nó cabeça** (**nó fictício** ou **nodo sentinela**: este nó não é um item, não guarda informação)

Alguns autores apresentam listas com nó cabeça (Cormen, Wirth etc.) e outros sem utilização de nó cabeça ou com ambos os estilos (Weiss, Goodrich, Shaffer, Edelweiss e Galante etc.).

Listas ligadas dinâmicas com **nó cabeça**



Lista duplamente ligada com nó cabeça ou sentinela . Fonte: Cormen. Introduction to Algorithms, 3d Edition.

- Vantagens da utilização do nó cabeça: **código mais claro, simplificação das verificações em algumas operações.**
- Utilizar com cuidado em situações de muitas listas pequenas, por causa da **utilização adicional de memória**: "We should use sentinels judiciously. When there are many small lists, the extra storage used by their sentinels can represent significant wasted memory... use only when they truly simplify the code.", Cormen.
- As responsabilidades por operações de atualização de enlaces, inserção e eliminação de nodos etc. poderiam ficar na classe Node.

Tipo Abstrato de Dados

TAD_LinkedList

- este **TAD_LinkedList** é uma proposta das principais operações de uma lista dinâmica encadeada;
- utilizaremos nomes tradicionais em inglês, como utilizados na bibliografia e como implementados nas classes `LinkedList` de Java/Oracle e C#/Microsoft;
- depois mostraremos uma implementação deste TAD na classe **LinkedList**, em um projeto NetBeans.

```
public interface TAD_LinkedList { // TAD que descreve uma lista ligada dinâmica

    //Verifica se a lista está vazia.
    public boolean isEmpty(); // O(1)

    //Retorna o tamanho (quantidade de itens) da lista.
    public int size(); // O(1)

    //Retorna o conteúdo (todos os elementos) da lista.
    public String toString(); // O(n)

    //Adiciona o item no início da lista (head). Retorna null se não foi possível adicionar.
    public Object addFirst(Object item); // O(1)

    //Adiciona o item no final da lista (tail). Retorna null se não foi possível adicionar.
    public Object addLast(Object item); // O(1)

    //Remove e retorna o item na cabeça da lista, o primeiro (retorna null se lista vazia).
    public Object removeFirst(); // O(1)

    //Retorna (sem eliminar) o primeiro item da lista (retorna null se lista vazia).
    public Object peekFirst(); // O(1)
```

//Retorna (sem eliminar) o último item da lista (null se lista vazia).

public Object peekLast(); // O(1)

//Limpa o conteúdo da lista.

public void clear(); // O(n).

//Retorna um vetor de objetos com os itens da lista encadeada dinâmica.

public Object[] toArray(); // O(n)

//Adiciona um item na lista, na posição idx. Retorna null se não foi possível adicionar.

//Erro se $idx < 0$ ou $idx \geq size()$.

public Object add(int idx, Object item); // O(n)

//Remove o item da posição idx. Retorna o item se sucesso, ou null caso contrário.

//Erro se $idx < 0$ ou $idx \geq size()$.

public Object remove(int idx); // O(n)

//Retorna o elemento que está na posição idx. Erro se $idx < 0$ || $idx \geq size()$.
public Object get(int idx); // O(n)

//Altera o item da posição idx da lista. Erro se $idx < 0$ || $idx \geq size()$.
public Object set(int idx, Object item); // O(n)

//Procura um objeto dentro da lista. Se encontra retornará true.
public boolean contains(Object x); // O(n)

//Elimina da lista o objeto x especificado, se for encontrado. Retorna true ou false.
public boolean remove(Object x); // O(n)

} //fim do TAD_LinkedList

Uma implementação de uma lista dinâmica ligada

- utilizamos nomes tradicionais em inglês;
- implementada como uma classe concreta (que implementa o TAD anterior), em um projeto NetBeans.

Atributos e operações básicas

class **LinkedList** implements **TAD_LinkedList** {


```
private Node head = null, tail = null; // cabeça e cauda (início e final da lista)
private int size = 0; // quantidade de itens na lista
```

```
public LinkedList () { // método construtor
    head = null; tail = null; size = 0; // convenção de lista vazia
}
```

```
public boolean isEmpty () {
    if(head == null) return true; else return false; // ou return (size == 0);
}
```


Outras operações básicas

```
public Object peekFirst () { //retorna o objeto no início da lista  
    if(head == null) return null; else return head.getValue(); //ou if( isEmpty() )  
}
```



parecido com
peek da fila

```
public Object peekLast () { //retorna o objeto no final da lista  
    if(head == null) return null; else return tail.getValue(); //ou if( isEmpty() )  
}
```



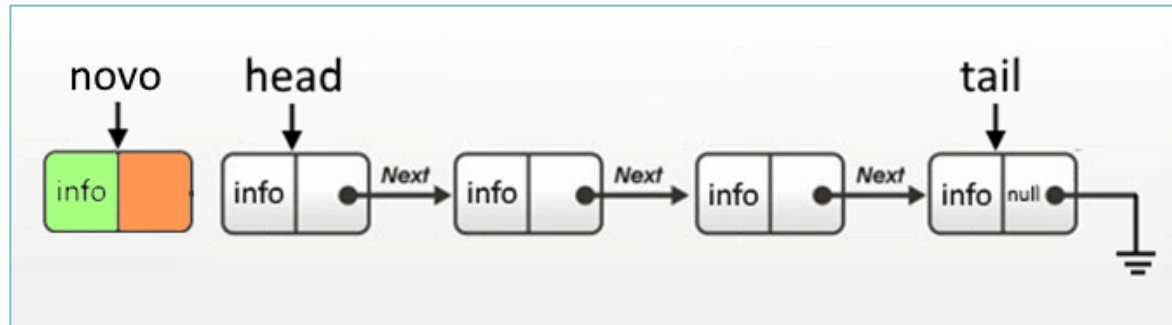
parecido com
top da pilha

```
public int size () { //retorna a quantidade de itens na lista  
    return size;  
}
```

Os métodos que retornam null, significa "operação não efetuada".

Adicionar um **novo** item no início da lista (head)

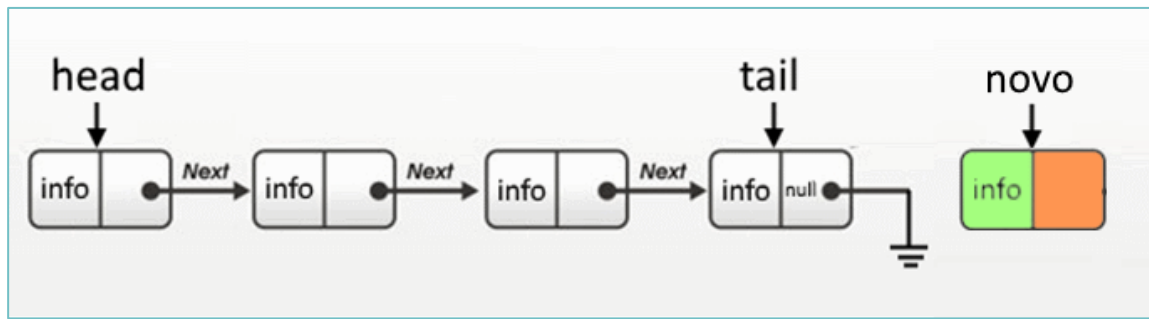
```
public Object addFirst (Object info) {
    if (info == null) return null;
    try { //para casos de memória insuficiente
        if(Runtime.getRuntime().freeMemory() < info.toString().length() + 1024) return null;
        Node novo = new Node();
        novo.setValue(info);
        novo.setNext(null);
        if (head == null) {
            head = novo;
            tail = novo;
        }
        else {
            novo.setNext(head); // liga no novo nodo com o antigo head
            head = novo;
        }
        size++; //porque a lista agora tem um item a mais
        return info;
    } catch(Exception ex) { return null; } // qualquer outro erro
}
```



Adicionar um **novo** item no final da lista (tail)

```
public Object addLast (Object info) {
    if (info == null) return null;
    try {
        if(Runtime.getRuntime().freeMemory() < info.toString().length() + 1024) return null;
        Node novo = new Node();
        novo.setValue(info);
        novo.setNext(null);
        if (head == null) head = novo; // caso especial: lista vazia, if ( isEmpty() )
            else tail.setNext(novo); // liga o nodo no final da lista

        tail = novo;
        size++; //porque a lista tem agora um item a mais
        return info;
    } catch(Exception ex) { return null; } // qualquer outro erro
}
```


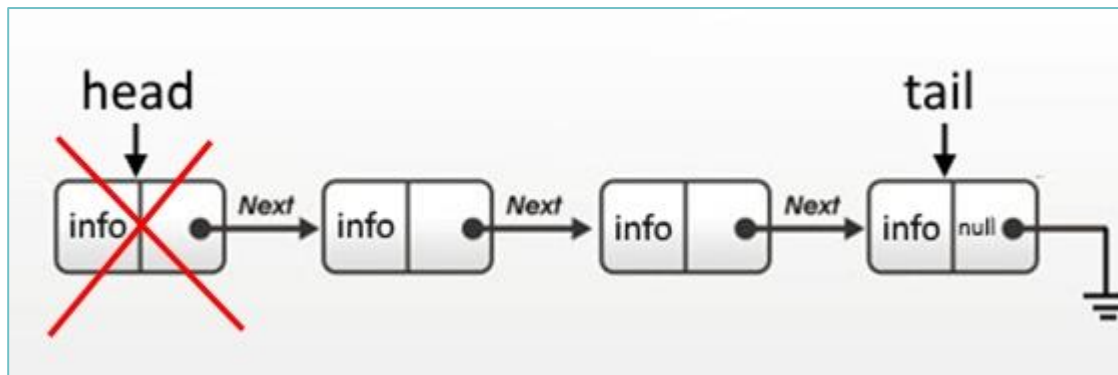


parecido com
enqueue da fila e
o push da pilha



Retirar o elemento que está na cabeça da lista (head)

```
public Object removeFirst () {  
    if (head == null) return null; // pode ser: if (size == 0) ou if (isEmpty())  
    Object info = head.getValue();  
    head = head.getNext(); //para avançar o ponteiro head  
    if( head == null) tail=null;  
    size--; //ou size = size - 1; porque agora a lista tem um item a menos  
    return info;  
}
```



parecido com
dequeue da fila

Retornando uma String com todos os itens da lista

//Retorna o conteúdo da lista dinâmica ligada no formato lista: [head, a, b, c, ..., tail]

```
public String toString () {  
    if( !isEmpty() ) {  
        String saida = "";  
        Node aux = head;  
        while( aux!=null ) { // o ponteiro aux percorre a lista (não alterar head)  
            saida += aux.getValue().toString(); // ou separar os objetos com \n  
            aux = aux.getNext();  
            if ( aux != null ) saida += ", "; // separar com , ou com \n  
        }  
        return ( "lista: [ " + saida + " ]" );  
    }  
    else return ( "lista: [ ]" ); // lista vazia  
}
```


Procura um item **x** na lista

//Procura um objeto dentro da lista.

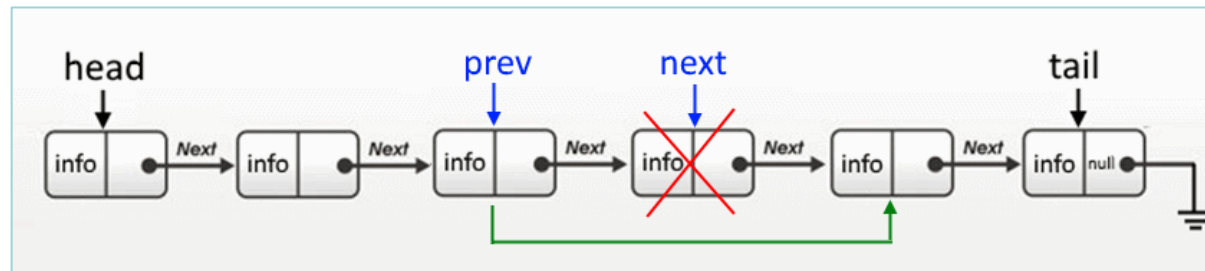
```
public boolean contains(Object x) {  
    if(x == null) return false;  
    Node aux = head;  
    boolean encontrado = false;  
    while ( aux!=null && !encontrado ) {  
        if(x.toString().equalsIgnoreCase(aux.getValue().toString())) encontrado = true;  
        aux = aux.getNext();  
    }  
    return encontrado;  
}
```

Algumas operações mais complexas com listas dinâmicas encadeadas

- São operações mais complexas, por se tratar de operações que podem ser efetuadas em qualquer lugar intermediário da lista e, ainda, vamos verificar casos especiais de inserção/eliminação no início ou no final da lista etc.
- Alguns destes métodos utilizam dois ponteiros (prev, next) que avançam simultaneamente para, uma vez encontrada a posição de inserção ou eliminação, facilitar a operação.
- Fornecemos estas implementações para quem deseje aprofundar ou quem necessite implementar estas operações em uma situação específica, mas **estas operações não serão avaliadas na A1 nem a AF.**

Eliminar e retornar o item que está na posição **idx** da lista

```
public Object remove (int idx) {
    if(isEmpty() || idx < 0 || idx >= size) return null; //pré-condições
    int pos = 0; Node next = head, prev = null;
    //avançar simultaneamente os ponteiros prev e next até encontrar a posição desejada pos==idx
    while(pos != idx) {
        prev = next;
        next = next.getNext();
        pos++;
    }
```



```
Object value = next.getValue(); //guardamos o valor a ser retornado
if(prev != null) prev.setNext(next.getNext()); //elimina a ligação do nodo eliminado
if(idx == 0) head = head.getNext(); //caso especial: eliminado o item na cabeça, => avançar head
if(idx == size-1) tail = prev; //caso especial: foi eliminado o item na cauda => modificar tail
if(head == null) tail = null; //para manter os dois ponteiros com valores nulos
next.setNext(null); next = null; //opcional: para liberar a memória do nodo eliminado
size--; // ou também: size = size - 1;
return value;
```

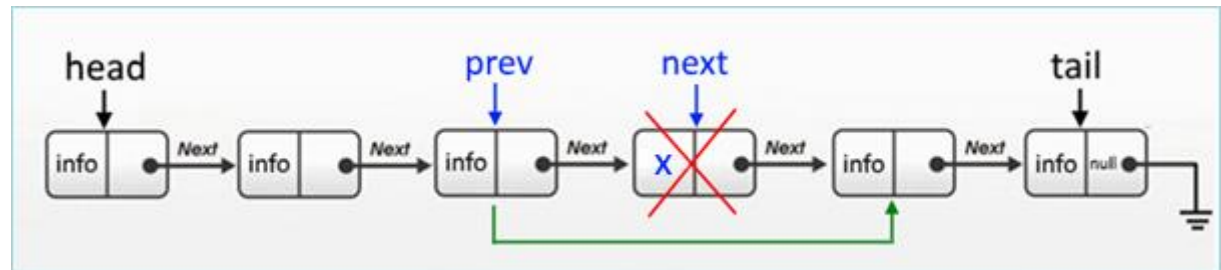
```
}
```

Eliminar um item **x** da lista

```

public boolean remove (Object x) {
    if(isEmpty() || x == null) return false;
    Node next = head, prev = null; boolean achou = false;
    //ciclo para avançar simultaneamente os ponteiros prev e next até encontrar o item x
    while(next!=null && !achou) {
        if(x.toString().equalsIgnoreCase(next.getValue().toString())) {
            achou = true; break;
        }
        prev = next;
        next = next.getNext();
    }
    if(!achou) return false;
    if(prev != null) prev.setNext(next.getNext()); //caso geral: o nodo eliminado não ficará ligado na lista
    if(prev == null) head = head.getNext(); //caso especial: eliminado o item na cabeça -> avançar head
    if(next.getNext() == null) tail = prev; //caso especial: foi eliminado o item na cauda -> modificar tail
    if(head == null) tail = null; //para manter os ponteiros com valores nulos
    next.setNext(null); next = null; //opcional: para liberar a memória do nodo eliminado
    size--; // ou também: size = size - 1;
    return true;
}

```

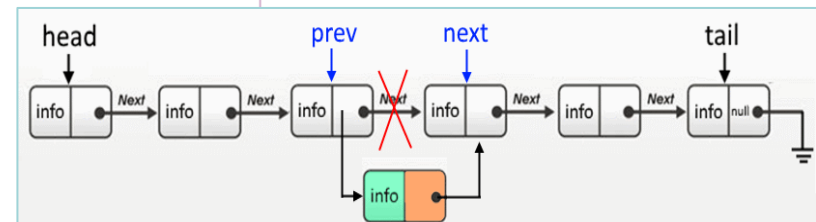


Adicionar um novo item na lista, na posição idx

```

public Object add(int idx, Object x) { //adicionar um item (nodo) na posição idx
    if(idx < 0 || idx > size || x == null) return null; //pré-condições
    if(idx == size) { //caso especial: inserção no final (depois do tail atual)
        Object res = addLast(x); return res;
    }
    Node novo = null;
    try {
        if(Runtime.getRuntime().freeMemory() < x.toString().length() + 1024) return null;
        novo = new Node();
        novo.setValue(value: x); novo.setNext(next: null);
    }
    catch (Exception ex) { return null; }
    int pos = 0; Node next = head, prev = null;
    //ciclo para avançar simultaneamente os ponteiros prev e next até encontrar a posição desejada pos==idx
    while(pos != idx) {
        prev = next; next = next.getNext(); pos++;
    }
    if(isEmpty()) {
        head = novo; tail = novo;
    }
    else if(idx == 0) { //caso especial: o novo nodo será inserido na cabeça da lista
        novo.setNext(next: head); head = novo;
    }
    else { //caso geral
        prev.setNext(next: novo); novo.setNext(next);
    }
    size++; //ou: size = size + 1; porque a lista tem agora mais um item
    return x; //retorna o mesmo objeto inserido na lista
}

```



Limpar a lista

```
public void clear () {  
    Node aux = head;  
    while(aux != null) {  
        Node tmp = aux;  
        aux = aux.getNext();  
        // liberar memória:  
        tmp.setNext(null); tmp = null;  
    }  
    head = null; tail = null; size = 0;  
}
```

Retornar um vetor de objetos

```
public Object[] toArray () {  
    if(isEmpty()) return null;  
    try { //para verificar memória insuficiente  
        Object[] vetor = new Object[size];  
        int i = 0;  
        Node aux = head;  
        while (aux != null) {  
            vetor[i++] = aux.getValue();  
            aux = aux.getNext();  
        }  
        return vetor;  
    }  
    catch(Exception exc) { return null; }  
}
```

//exemplo de uso: para ordenar o vetor retornado

A classe **LinkedList** da Oracle

- Existe uma classe pronta na biblioteca da Oracle: **LinkedList**.
Veja detalhes em:
<https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html>
- Esta classe possui todos os métodos que mostramos antes e bastantes outros. Por exemplo, também possui: `addFirst`, `addLast`, `removeFirst`, `removeLast`, `peekFirst`, `peekLast`, `isEmpty`, `toArray`, `clear`, `get`, `set`, `size`, `contains` etc.
- Com esta classe podemos implementar listas encadeadas, filas e pilhas! Veja nos próximos slides.

Reflexão: uma lista ligada (**LinkedList**) utilizada como **Fila**

- Uma lista ligada (LinkedList) poderia ser utilizada como fila (Queue), sempre que respeitado o critério **FIFO** de processamento.
 - A operação **enqueue** é equivalente a uma operação **addLast** (para inserir o item no final-cauda-tail da fila).
 - A operação **dequeue** é equivalente a uma operação **removeFirst** (eliminar/retornar o item que está na cabeça-head-início da fila).
 - A operação **peek** da fila é equivalente a uma operação **peekFirst** (para retornar, sem eliminar, o primeiro da fila).
 - Outras operações da LinkedList, como **isEmpty**, **toString** e **clear**, poderão ser aproveitadas para uma fila.
- Opcionalmente, poderia ser criada uma classe derivada (class **Queue** extends **LinkedList**) para implementação dos métodos originais de uma fila, baseados nos equivalentes da LinkedList.

Reflexão: uma lista ligada (**LinkedList**) utilizada como **Pilha**

- Uma lista ligada (LinkedList) poderia ser utilizada como pilha (Stack), sempre que respeitado o critério **LIFO** de processamento.
 - A operação **push** é equivalente a uma operação **addLast** (para inserir o item no topo da pilha).
 - A operação **pop** é equivalente a uma operação **removeLast** (para eliminar/retornar o item que está no topo da pilha).
 - A operação **top** é equivalente a uma operação **peekLast** (para retornar, sem eliminar, o item que se encontra no topo da pilha).
 - Outras operações da LinkedList, como **isEmpty**, **toString** e **clear**, poderão ser aproveitadas para uma pilha (toString ≈).
- Opcionalmente, poderia ser criada uma classe derivada (class **Stack** extends **LinkedList**) para implementar os métodos originais de uma pilha, baseados nos equivalentes da LinkedList.

Fila e Pilha, opcional: herdando da classe **LinkedList**

```
class Pilha extends LinkedList {
```

```
    //podemos implementar os métodos push, pop, top
```

```
    //com base em AddLast, RemoveLast, peekLast
```

```
}
```

```
class Fila extends LinkedList {
```

```
    // podemos implementar os métodos enqueue, dequeue, peek
```

```
    //com base em AddLast, RemoveFirst, peekFirst
```

```
}
```

Uma lista ligada (**LinkedList**) utilizada como **fila**

```
LinkedList fila = new LinkedList();
```

```
//enqueue:
```

```
fila.addLast("casa");
```

```
fila.addLast("mesa");
```

```
fila.addLast("janela");
```

```
System.out.println(fila.toString());
```

[casa, mesa, janela]

```
//dequeue:
```

```
System.out.println("dequeue:" + fila.removeFirst());
```

dequeue: casa

Uma lista ligada (**LinkedList**) utilizada como **pilha**

```
LinkedList pilha = new LinkedList();
```

```
//push:
```

```
pilha.addLast("verde");
```

```
pilha.addLast("azul");
```

```
pilha.addLast("amarelo");
```

```
System.out.println(pilha.toString());
```

[verde, azul, amarelo]

```
//pop:
```

```
System.out.println("pop: " + pilha.removeLast());
```

pop: amarelo

Exemplo: uma lista ligada (**LinkedList**) utilizada como **fila**

```
System.out.println("Fila2");
LinkedList fila2 = new LinkedList();
//adicionemos objetos inteiros (da classe Integer):
fila2.addLast(20);
fila2.addLast(30);
fila2.addLast(40);
fila2.addLast(50);
//um processamento geral qualquer:
int soma = 0;
for (Object obj : fila2) {
    int valor = (int) obj;
    soma += valor;
}
System.out.println(fila2.toString());
System.out.println("Média dos valores: " + soma / fila2.size());
```

Exemplo: uma lista ligada (**LinkedList**) utilizada como **fila**

```
System.out.println("Fila3");
LinkedList fila3 = new LinkedList();
//adicionemos objetos da classe Trabalhador:
fila3.addLast(new Trabalhador("Beatriz Lima", 6000.00f));
fila3.addLast(new Trabalhador("Luiz Alves", 3000.00f));
fila3.addLast(new Trabalhador("Amilton Lopes", 3000.00f));
//um processamento geral qualquer:
float somaSalarios = 0;
for (Object obj : fila3) {
    Trabalhador trab = (Trabalhador) obj;
    System.out.println(trab.toString()); //mostramos um trabalhador em uma linha
    somaSalarios += trab.getSalario(); //poderíamos chamar trab.getNome() etc.
}
//System.out.println(fila3.toString()); //mas ficariam todos em uma linha
System.out.println("Média dos salários: R$" + somaSalarios / fila3.size());
```

Ordenando os objetos da **fila** anterior (alteramos a fila)

Comparator **c** =

```
((obj1, obj2) -> (((Trabalhador) obj1).compareTo(((Trabalhador) obj2))));  
//O "comparador" anterior especifica como comparar trabalhadores e funciona  
//porque a classe Trabalhador implementou o método compareTo.
```

```
fila3.sort(c); //IMPORTANTE: a fila será alterada!!! Veja próxima versão →  
//O método de ordenação sort, é herdado pela classe LinkedList de List e,  
//segundo a Oracle, utiliza um método iterativo MergeSort, de  $O(n \lg n)$ .  
//Mostramos na tela os trabalhadores já ordenados:  
for (Object obj : fila3) {  
    //Se for necessário utilizar outros métodos da classe Trabalhador podemos  
    //fazer a conversão: Trabalhador trab = (Trabalhador)obj;  
    System.out.println(obj.toString());  
}
```

Ordenando os objetos de uma **fila**, mas sem alterar a fila

```
LinkedList fila4 = new LinkedList();  
fila4.addLast(new Trabalhador("Daniel Borelli", 6000.00f));  
fila4.addLast(new Trabalhador("Zoe Fernandes", 5000.00f));  
fila4.addLast(new Trabalhador("Ana Linares", 3000.00f));  
fila4.addLast(new Trabalhador("Luiz Alves", 3000.00f));  
fila4.addLast(new Trabalhador("Amilton Gomes", 3000.00f));  
fila4.addLast(new Trabalhador("Ada Hernández", 3000.00f));  
LinkedList filaClonada = (LinkedList) fila4.clone(); //criamos um clone  
filaClonada.sort(c); //vamos utilizar o mesmo comparador anterior  
System.out.println("Fila original");  
for (Object obj : fila4) System.out.println(obj.toString());  
System.out.println("Fila clonada ordenada");  
for (Object obj : filaClonada) System.out.println(obj.toString());
```


Objetos da **fila** (um clone) ordenados, sem alterar a fila original

Fila original sem alterar

Trabalhador{nome=Daniel Borelli, salario=R\$6000.0}

Trabalhador{nome=Zoe Fernandes, salario=R\$5000.0}

Trabalhador{nome=Ana Linares, salario=R\$3000.0}

Trabalhador{nome=Luiz Alves, salario=R\$3000.0}

Trabalhador{nome=Amilton Gomes, salario=R\$3000.0}

Trabalhador{nome=Ada Hernández, salario=R\$3000.0}

Fila clonada ordenada

Trabalhador{nome=Ada Hernández, salario=R\$3000.0}

Trabalhador{nome=Amilton Gomes, salario=R\$3000.0}

Trabalhador{nome=Ana Linares, salario=R\$3000.0}

Trabalhador{nome=Daniel Borelli, salario=R\$6000.0}

Trabalhador{nome=Luiz Alves, salario=R\$3000.0}

Trabalhador{nome=Zoe Fernandes, salario=R\$5000.0}

Exercícios para praticar (não precisa entregar)

Implemente, na classe **LinkedList** cujo código fonte foi fornecido, estes métodos que faltaram:

//Retorna o elemento da posição idx. Erro se $idx < 0$ || $idx \geq size()$.
public Object **get**(int idx); // $O(n)$

//Altera o item da posição idx da lista. Erro se $idx < 0$ || $idx \geq size()$.
public Object **set**(int idx, Object item); // $O(n)$

Exercício para praticar

(não precisa entregar - utilize a classe LinkedList)

- Criar uma **fila dinâmica ligada** com dados de livros, **utilizando a classe `LinkedList`**. 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 mostrar seus dados na tela.
- Implemente a lógica de um botão que execute um método que receba como parâmetro uma fila de livros, retire os mesmos e crie uma nova fila apenas com os livros publicados depois de 2002. Mostre na tela a nova fila. Deixe a fila original como se encontrava antes deste processamento.

Bibliografia sugerida 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> |