Estruturas de Dados Fundamentais

Arranjos e listas encadeadas

Prof. Marcelo de Souza

45EST – Algoritmos e Estruturas de Dados
Universidade do Estado de Santa Catarina

UDESC

Material de apoio

Leitura principal:

- Capítulo 3 de Goodrich et al. (2014)[1] – Estruturas de dados fundamentais.

Leitura complementar:

- Capítulo 4 de Preiss (2001)[2] – Estruturas de dados fundamentais.
- Capítulo 2 de Pereira (2008)[3] – Listas lineares.

[1] Michael T Goodrich et al. (2014). *Data structures and algorithms in Java*. 6ª ed. John Wiley & Sons.

[2] Bruno R Preiss (2001). *Estruturas de dados e algoritmos: padrões de projetos orientados a objetos com Java*. Campus.

[3] Silvio do Lago Pereira (2008). *Estruturas de Dados Fundamentais: Conceitos e Aplicações*.

Arranjos

Ou seja, *arrays*/vetores

Arranjos são **estruturas de dados sequenciais**, armazenando sequências finitas e ordenadas de valores de um mesmo tipo. Por exemplo:

- **Números:** 1, 2, 4, 5, 7, 8, ...
- **Strings:** "Brasil", "Alemanha", "Croácia", ...
- **Veículos:** ("Corcel", 1977), ("Fusca", 1968), ("Passat", 1984), ...

Arranjos

Ou seja, *arrays*/vetores

Arranjos são **estruturas de dados sequenciais**, armazenando sequências finitas e ordenadas de valores de um mesmo tipo. Por exemplo:

- **Números:** 1, 2, 4, 5, 7, 8, ...
- **Strings:** "Brasil", "Alemanha", "Croácia", ...
- **Veículos:** ("Corcel", 1977), ("Fusca", 1968), ("Passat", 1984), ...

A principal característica dos arranjos é a **alocação contígua** em memória.

- **Vantagens:**
  - Fácil de usar;
  - Acesso rápido (tempo constante).
- **Desvantagens:**
  - Tamanho fixo (aumentar implica copiar elementos);
  - Inserção e remoção [interna] custosas (*shift* de elementos).

Listas simplesmente encadeadas

Encadeamento

Queremos uma estrutura de dados dinâmica que permita **expandir** e **contrair** com eficiência.

Listas simplesmente encadeadas

Encadeamento

Queremos uma estrutura de dados dinâmica que permita **expandir** e **contrair** com eficiência.

Lista encadeada: coleção de nodos formados em uma sequência linear.

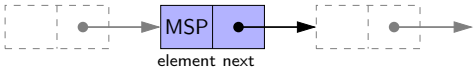
Listas simplesmente encadeadas

Encadeamento

Queremos uma estrutura de dados dinâmica que permita **expandir** e **contrair** com eficiência.

Lista encadeada: coleção de nodos formados em uma sequência linear.

Lista simplesmente encadeada: cada nodo armazena os dados do elemento e uma referência ao próximo nodo. A alocação em memória **não é contígua**.



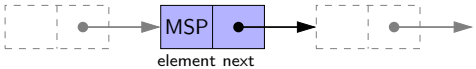
Listas simplesmente encadeadas

Encadeamento

Queremos uma estrutura de dados dinâmica que permita **expandir** e **contrair** com eficiência.

Lista encadeada: coleção de nodos formados em uma sequência linear.

Lista simplesmente encadeada: cada nodo armazena os dados do elemento e uma referência ao próximo nodo. A alocação em memória **não é contígua**.



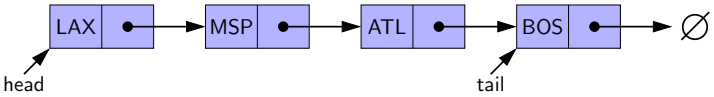
Benefícios:

- Tamanho dinâmico;
- Consumo de memória dinâmico;
- Fácil inserção e remoção de elementos.

Listas simplesmente encadeadas

Encadeamento

Exemplo: uma lista simplesmente encadeada para armazenar aeroportos dos EUA.



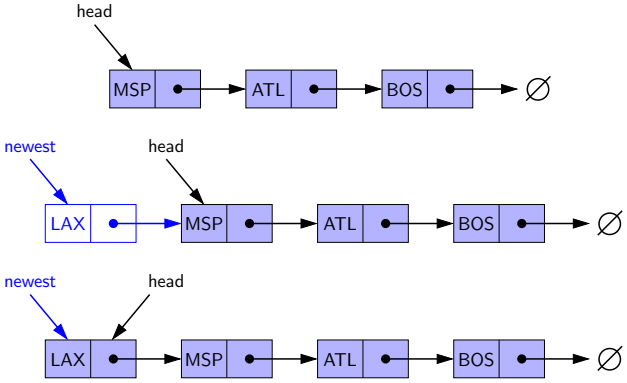
Elementos:

- `head`: referência ao primeiro elemento da lista;
- `tail`: referência ao último elemento da lista;
- O próximo elemento do último elemento aponta para `null`.

Listas simplesmente encadeadas

Operações

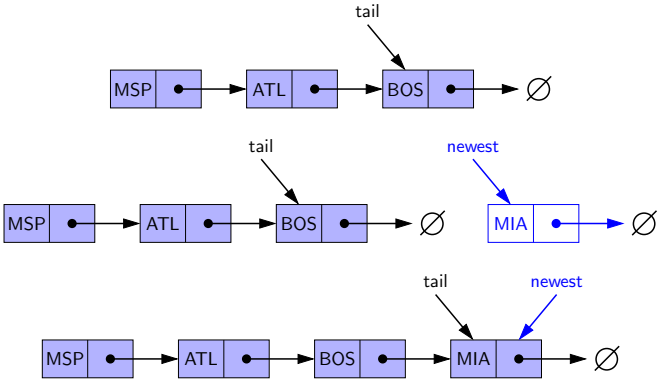
Inserção de elemento no início



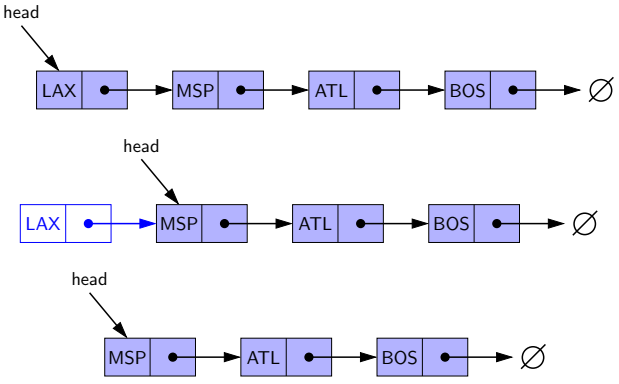
Listas simplesmente encadeadas

Operações

Inserção de elemento no final



Remoção de elemento do início



Listas simplesmente encadeadas

Implementação

```java
public class SinglyLinkedList<E> {

  private static class Node<E> {
    private E element;
    private Node<E> next;

    public Node(E e, Node<E> n) {
      element = e;
      next = n;
    }

    public E getElement() { return element; }
    public Node<E> getNext() { return next; }
    public void setNext(Node<E> n) { next = n; }
  }

  private Node<E> head = null;
  private Node<E> tail = null;
  private int size = 0;

  // ...
}
```

Listas simplesmente encadeadas

Implementação

```
1  public class SinglyLinkedList<E> {
2
3    private static class Node<E> {
4      private E element;
5      private Node<E> next;
6
7      public Node(E e, Node<E> n) {
8        element = e;
9        next = n;
10     }
11
12     public E getElement() { return element; }
13     public Node<E> getNext() { return next; }
14     public void setNext(Node<E> n) { next = n; }
15   }
16
17   private Node<E> head = null;
18   private Node<E> tail = null;
19   private int size = 0;
20
21   // ...
22 }
```

Detalhes:

- Usamos **genéricos** (`<E>`) para suportar qualquer tipo de dados.
  - e.g., podemos ter uma lista de inteiros, Strings, veículos, ...
- A classe `Node` define um nodo, que contém um elemento e uma referência ao próximo nodo da lista.
  - `Node` é uma **nested class**, pois queremos encapsular o nodo.
- A lista contém referências para o primeiro e último nodos (`head` e `tail`) e o seu tamanho (número de nodos), inicialmente zero.

Listas simplesmente encadeadas

Implementação

Métodos `size` e `isEmpty`:

```
1  public int size() { return size; }
2  public boolean isEmpty() { return size == 0; }
```

Listas simplesmente encadeadas

Implementação

Métodos `size` e `isEmpty`:

```
1  public int size() { return size; }
2  public boolean isEmpty() { return size == 0; }
```

Métodos `first` e `last`:

```
1  public E first() {
2    if (isEmpty()) return null;
3    return head.getElement();
4  }
5
6  public E last() {
7    if (isEmpty()) return null;
8    return tail.getElement();
9  }
```

Note que:

- O método retorna o elemento armazenado pelo primeiro (ou último) nodo.
- A estrutura da lista (`Node`) é transparente (encapsulada).

Listas simplesmente encadeadas

Implementação

Método `addFirst`:

```
1  public void addFirst(E e) {
2    head = new Node<>(e, head);
3    if (size == 0)
4      tail = head;
5    size++;
6  }
```

- Criamos um novo nodo para armazenar o elemento, que passa a ser o novo `head` e aponta para o antigo `head`.
- Caso a lista esteja vazia, ele é o novo `tail`.

Listas simplesmente encadeadas

Implementação

Método `addFirst`:

```
1  public void addFirst(E e) {
2    head = new Node<>(e, head);
3    if (size == 0)
4      tail = head;
5    size++;
6  }
```

- Criamos um novo nodo para armazenar o elemento, que passa a ser o novo `head` e aponta para o antigo `head`.
- Caso a lista esteja vazia, ele é o novo `tail`.

Método `addLast`:

```
1  public void addLast(E e) {
2    Node<E> newest = new Node<>(e, null);
3    if (isEmpty())
4      head = newest;
5    else
6      tail.setNext(newest);
7    tail = newest;
8    size++;
9  }
```

- O novo nodo (`newest`) passa a ser o próximo nodo do atual `tail`, antes de assumir a posição do `tail`.
- Caso a lista esteja vazia, o novo nodo passa a ser o `head` e o `tail`.

Listas simplesmente encadeadas

Implementação

Método `removeFirst`:

```
1  public E removeFirst() {
2    if (isEmpty()) return null;
3    E answer = head.getElement();
4    head = head.getNext();
5    size--;
6    if (size == 0) tail = null;
7    return answer;
8  }
```

- Retorna o elemento do nodo removido.
- O nodo `head` passa a ser o próximo nodo do `head` atual.
- Se a remoção deixa a lista vazia, o `tail` passa a ser `null`.

Listas simplesmente encadeadas

Implementação

Método `removeFirst`:

```
1  public E removeFirst() {
2    if (isEmpty()) return null;
3    E answer = head.getElement();
4    head = head.getNext();
5    size--;
6    if (size == 0) tail = null;
7    return answer;
8  }
```

- Retorna o elemento do nodo removido.
- O nodo `head` passa a ser o próximo nodo do `head` atual.
- Se a remoção deixa a lista vazia, o `tail` passa a ser `null`.

Método `toString`:

```
1  public String toString() {
2    StringBuilder sb = new StringBuilder("(");
3    Node<E> walk = head;
4    while (walk != null) {
5      sb.append(walk.getElement());
6      if (walk != tail) sb.append(", ");
7      walk = walk.getNext();
8    }
9    sb.append(")");
10   return sb.toString();
11 }
```

- Percorremos a estrutura com um `while`, visitando um nodo por vez.
- Para cada nodo, recuperamos seu elemento e o adicionamos na string de resultado.
- O próximo nodo do `tail` é `null`, o que viola a condição do `while` e interrompe a execução do laço.

Listas encadeadas circulares

Conceito e motivação

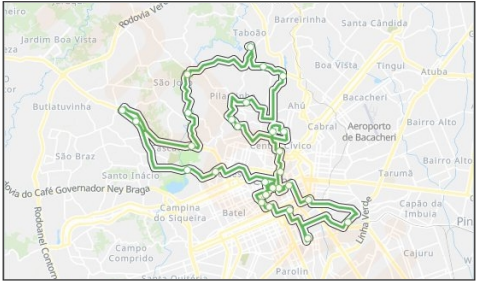
Lista circular: é uma lista encadeada com ordenação cíclica. Ou seja, cada elemento tem um anterior e um seguinte, mas não exite início e fim definidos. Na prática, o "último" elemento aponta para o "primeiro".

Listas encadeadas circulares

Conceito e motivação

Lista circular: é uma lista encadeada com ordenação cíclica. Ou seja, cada elemento tem um anterior e um seguinte, mas não exite início e fim definidos. Na prática, o "último" elemento aponta para o "primeiro".

Aplicações: rotas de transporte público (e.g. ônibus)



Listas encadeadas circulares

Conceito e motivação

Lista circular: é uma lista encadeada com ordenação cíclica. Ou seja, cada elemento tem um anterior e um seguinte, mas não exite início e fim definidos. Na prática, o "último" elemento aponta para o "primeiro".

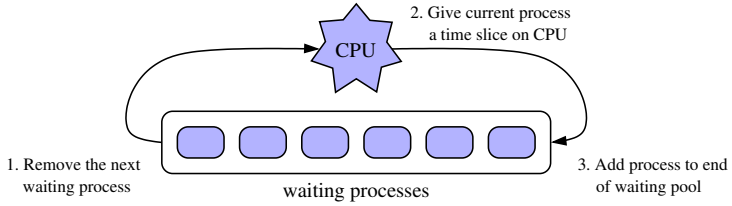
Aplicações: jogos por turnos

Listas encadeadas circulares

Conceito e motivação

Lista circular: é uma lista encadeada com ordenação cíclica. Ou seja, cada elemento tem um anterior e um seguinte, mas não exite início e fim definidos. Na prática, o "último" elemento aponta para o "primeiro".

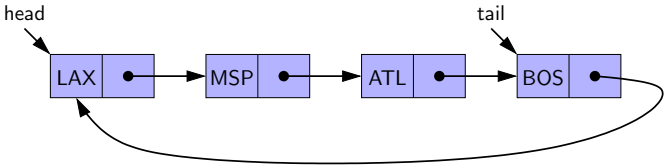
Aplicações: escalonamento de processos



Listas encadeadas circulares

Conceito e motivação

Exemplo: uma lista encadeada circular para armazenar aeroportos dos EUA.



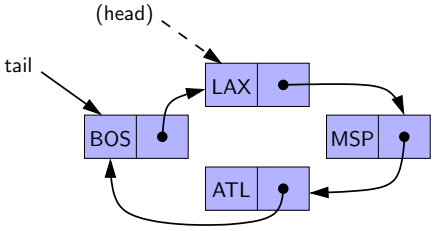
Novidades:

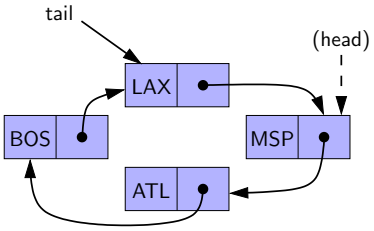
- Não é necessária a referência para `head` (`tail.getNext()`);
- Novo método `rotate`, que avança um elemento na lista, atualizando a referência `tail`.

Listas encadeadas circulares

Operações

Rotação (avanço na lista circular)

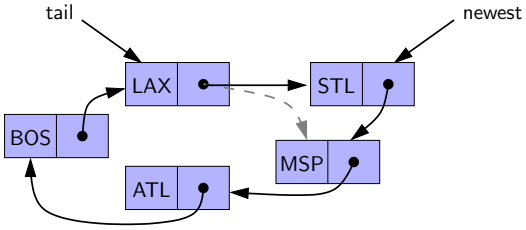




Listas encadeadas circulares

Operações

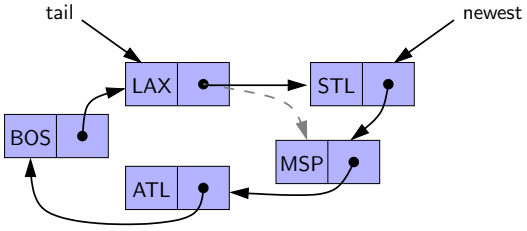
Inserção de elemento no início (i.e., após o `tail`)



Listas encadeadas circulares

Operações

Inserção de elemento no início (i.e., após o `tail`)



Inserção de elemento no final (i.e., passa a ser o `tail`)

- Insere no início (conforme acima) e rotaciona (`rotate`).

Remoção de elemento do início (i.e., o elemento seguinta ao `tail`)

- Basta atualizar a referência do `tail`.

Listas encadeadas circulares

Implementação

```java
public class CircularlyLinkedList<E> {

  // Definição da classe Node [...]

  private Node<E> tail = null;
  private int size = 0;

  public int size() { return size; }
  public boolean isEmpty() { return size == 0; }

  public E first() {
    if (isEmpty()) return null;
    return tail.getNext().getElement();
  }

  public E last() {
    if (isEmpty()) return null;
    return tail.getElement();
  }

  //...
}
```

Listas encadeadas circulares

Implementação

```java
public class CircularlyLinkedList<E> {

  // Definição da classe Node [...]

  private Node<E> tail = null;
  private int size = 0;

  public int size() { return size; }
  public boolean isEmpty() { return size == 0; }

  public E first() {
    if (isEmpty()) return null;
    return tail.getNext().getElement();
  }

  public E last() {
    if (isEmpty()) return null;
    return tail.getElement();
  }

  //...
}
```

Detalhes:

- Usamos a mesma classe aninhada `Node` e o tipo genérico `<E>`.
- Só mantemos referência ao "último" elemento da lista (`tail`). O método `last` retorna o elemento do `tail`, enquanto o método `first` retorna o elemento de `tail.getNext()`.
- Caso haja somente um elemento na lista, `tail.getNext()` é o mesmo que `tail`, pois esse único nodo referencia a si mesmo como próximo (visto que a lista é circular).

Listas encadeadas circulares

Implementação

Método `rotate`:

```
1  public void rotate() {
2    if (tail != null)
3    tail = tail.getNext();
4  }
```

Listas encadeadas circulares

Implementação

Método `rotate`:

```
1  public void rotate() {
2    if (tail != null)
3    tail = tail.getNext();
4  }
```

Métodos `addFirst` e `addLast`:

```
1  public void addFirst(E e) {
2    if (size == 0) {
3      tail = new Node<>(e, null);
4      tail.setNext(tail);
5    } else {
6      Node<E> newest = new Node<>(e, tail.getNext());
7      tail.setNext(newest);
8    }
9    size++;
10 }
11
12 public void addLast(E e) {
13   addFirst(e);
14   rotate();
15 }
```

No método `addFirst`:

- Caso seja o primeiro elemento, ele passa a ser o `tail` e aponta a si mesmo.
- Caso contrário, o novo nodo é o próximo do `tail` atual e passa a ser o novo `tail`.

No método `addLast`:

- Após adicionar o elemento como primeiro (`addFirst`), atualizamos o `tail`.

Listas encadeadas circulares

Implementação

Método `removeFirst`:

```
1  public E removeFirst() {
2    if (isEmpty()) return null;
3    Node<E> head = tail.getNext();
4    if (head == tail) tail = null;
5    else tail.setNext(head.getNext());
6    size--;
7    return head.getElement();
8  }
```

- Se `head` for igual ao `tail`, só há um elemento na lista e o `tail` passa a ser `null`.
- Caso contrário, o próximo nodo do `head` é o novo `tail`, removendo o `head`.
- Ao final, retorna o elemento do `head`.

Listas encadeadas circulares

Implementação

Método `removeFirst`:

```
1  public E removeFirst() {
2    if (isEmpty()) return null;
3    Node<E> head = tail.getNext();
4    if (head == tail) tail = null;
5    else tail.setNext(head.getNext());
6    size--;
7    return head.getElement();
8  }
```

- Se `head` for igual ao `tail`, só há um elemento na lista e o `tail` passa a ser `null`.
- Caso contrário, o próximo nodo do `head` é o novo `tail`, removendo o `head`.
- Ao final, retorna o elemento do `head`.

Método `toString`:

```
1  public String toString() {
2    if (tail == null) return "()";
3    StringBuilder sb = new StringBuilder("(");
4    Node<E> walk = tail;
5    do {
6      walk = walk.getNext();
7      sb.append(walk.getElement());
8      if (walk != tail) sb.append(", ");
9    } while (walk != tail);
10   sb.append(")");
11   return sb.toString();
12 }
```

- O critério de parada do percurso muda, pois sempre haverá um próximo nodo (i.e. `getNext()` nunca será `null` em uma lista não vazia).
- O percurso inicia pelo `tail` e termina quando retorna ao `tail` (percurso completo).

Listas duplamente encadeadas

Conceito e motivação

Problemas do encadeamento simples:

- Não conseguimos **remover o último nodo** de forma eficiente.
  - Precisamos atualizar a referência `next` do nodo anterior.
  - Para chegar no penúltimo, precisamos **percorrer a lista**.
- Remover um nodo (que não seja o primeiro) tendo apenas a sua referência é custoso.

Listas duplamente encadeadas

Conceito e motivação

Problemas do encadeamento simples:

- Não conseguimos **remover o último nodo** de forma eficiente.
  - Precisamos atualizar a referência `next` do nodo anterior.
  - Para chegar no penúltimo, precisamos **percorrer a lista**.
- Remover um nodo (que não seja o primeiro) tendo apenas a sua referência é custoso.

Lista duplamente encadeada: cada nodo mantém a referência do anterior (`prev`) e do próximo (`next`) nodos.

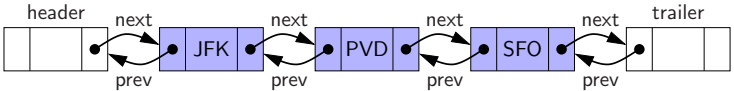
Listas duplamente encadeadas

Conceito e motivação

Problemas do encadeamento simples:

- Não conseguimos **remover o último nodo** de forma eficiente.
  - Precisamos atualizar a referência `next` do nodo anterior.
  - Para chegar no penúltimo, precisamos **percorrer a lista**.
- Remover um nodo (que não seja o primeiro) tendo apenas a sua referência é custoso.

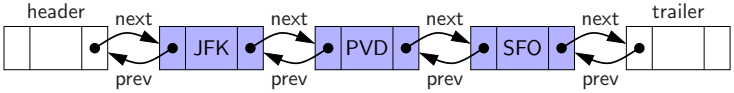
Lista duplamente encadeada: cada nodo mantém a referência do anterior (`prev`) e do próximo (`next`) nodos.



Listas duplamente encadeadas

Sentinelas

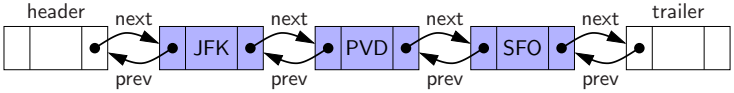
Na implementação dessas listas, usamos uma técnica muito útil: uso de **nodos sentinelas**. Trata-se de nodos vazios ("fictícios") no início (`header`) e no fim (`trailer`) da lista.



Listas duplamente encadeadas

Sentinelas

Na implementação dessas listas, usamos uma técnica muito útil: uso de **nodos sentinelas**. Trata-se de nodos vazios ("fictícios") no início (`header`) e no fim (`trailer`) da lista.



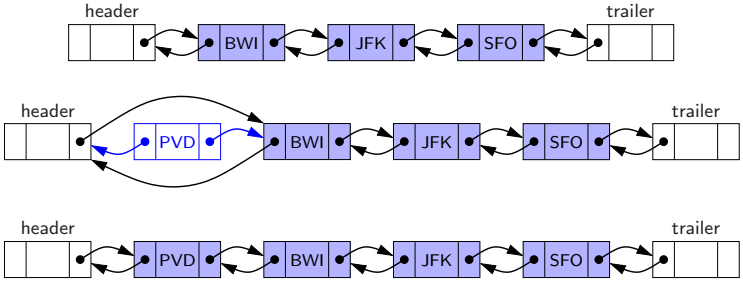
Facilidades (na implementação):

- Certeza de que cada nodo possui dois vizinhos;
- Toda inserção será entre dois nodos.
  - Nunca inserimos no verdadeiro início ou fim;
  - Casos excepcionais (lista vazia ou com um nodo) não acontecem.

Listas duplamente encadeadas

Operações

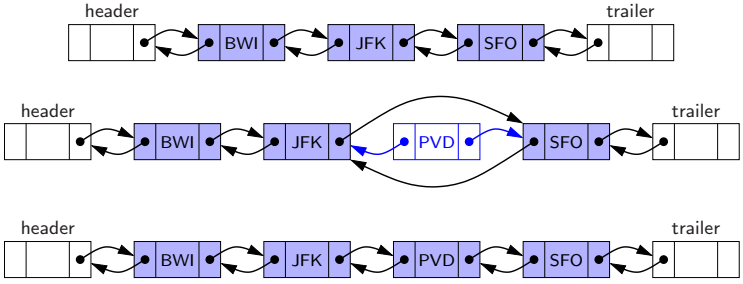
Inserção de elemento no início



Listas duplamente encadeadas

Operações

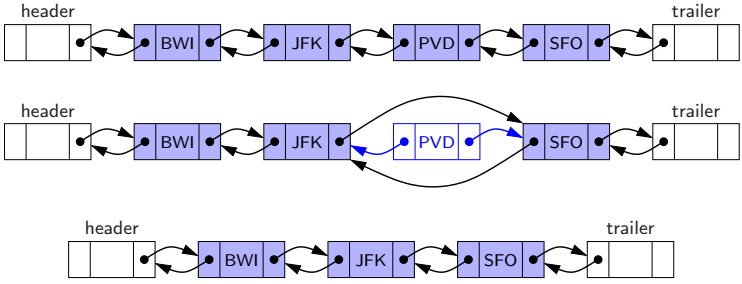
Inserção arbitrária de elemento



Listas duplamente encadeadas

Operações

Remoção de elemento



Listas duplamente encadeadas

Implementação

```
1  public class DoublyLinkedList<E> {
2
3    private static class Node<E> {
4      private E element;
5      private Node<E> prev, next;
6
7      public Node(E e, Node<E> p, Node<E> n) {
8        element = e;
9        prev = p;
10       next = n;
11     }
12
13     public E getElement() { return element; }
14     public Node<E> getPrev() { return prev; }
15     public Node<E> getNext() { return next; }
16
17     public void setPrev(Node<E> p) { prev = p; }
18     public void setNext(Node<E> n) { next = n; }
19   }
20
21   private Node<E> header, trailer;
22   private int size = 0;
23
24   // ...
25 }
```

Listas duplamente encadeadas

Implementação

```
1  public class DoublyLinkedList<E> {
2
3    private static class Node<E> {
4      private E element;
5      private Node<E> prev, next;
6
7      public Node(E e, Node<E> p, Node<E> n) {
8        element = e;
9        prev = p;
10       next = n;
11     }
12
13     public E getElement() { return element; }
14     public Node<E> getPrev() { return prev; }
15     public Node<E> getNext() { return next; }
16
17     public void setPrev(Node<E> p) { prev = p; }
18     public void setNext(Node<E> n) { next = n; }
19   }
20
21   private Node<E> header, trailer;
22   private int size = 0;
23
24   // ...
25 }
```

Detalhes:

- A classe `Node` agora define referências para os nodos anterior (`prev`) e próximo (`next`).
- O método construtor dessa classe recebe não só o elemento (`e`), mas também os nodos vizinhos (`p` e `n`).
- São definidos *setters* e *getters* para o elemento e para cada nodo vizinho.
- A lista agora possui os nodos sentinelas (`header` e `trailer`) e não possui referências para início e fim.
  - O primeiro elemento da lista está em `header.getNext()` e o último elemento está em `trailer.getPrev()`.

Listas duplamente encadeadas

Implementação

Método construtor:

```
1  public DoublyLinkedList() {
2    header = new Node<>(null, null, null);
3    trailer = new Node<>(null, header, null);
4    header.setNext(trailer);
5  }
```

- Inicialmente a lista contém os nodos sentinelas, que referenciam um ao outro.
- O `next` do `header` é o `trailer`; o `prev` do `trailer` é o `header`.

Listas duplamente encadeadas

Implementação

Método construtor:

```
1  public DoublyLinkedList() {
2    header = new Node<>(null, null, null);
3    trailer = new Node<>(null, header, null);
4    header.setNext(trailer);
5  }
```

- Inicialmente a lista contém os nodos sentinelas, que referenciam um ao outro.
- O `next` do `header` é o `trailer`; o `prev` do `trailer` é o `header`.

Métodos `first` e `last`:

```
1  public E first() {
2    if (isEmpty()) return null;
3    return header.getNext().getElement();
4  }
5
6  public E last() {
7    if (isEmpty()) return null;
8    return trailer.getPrev().getElement();
9  }
```

Listas duplamente encadeadas

Implementação

Método `addBetween`:

```
1  private void addBetween(E e, Node<E> p, Node<E> n) {
2    Node<E> newest = new Node<>(e, p, n);
3    p.setNext(newest);
4    n.setPrev(newest);
5    size++;
6  }
```

- Método genérico e encapsulado (privado) para inserir um novo nodo com o elemento desejado entre dois nodos vizinhos.
- Método auxiliar para qualquer inserção.

Listas duplamente encadeadas

Implementação

Método `addBetween`:

```
1  private void addBetween(E e, Node<E> p, Node<E> n) {
2    Node<E> newest = new Node<>(e, p, n);
3    p.setNext(newest);
4    n.setPrev(newest);
5    size++;
6  }
```

- Método genérico e encapsulado (privado) para inserir um novo nodo com o elemento desejado entre dois nodos vizinhos.
- Método auxiliar para qualquer inserção.

Métodos `addFirst` e `addLast`:

```
1  public void addFirst(E e) {
2    addBetween(e, header, header.getNext());
3  }
4
5  public void addLast(E e) {
6    addBetween(e, trailer.getPrev(), trailer);
7  }
```

- Usam o método `addBetween` para inserir no início (entre `header` e seu sucessor) e no final (entre `trailer` e seu antecessor).
- **Nota:** com sentinelas, não é necessário checar se a lista é vazia ou se tem só um nodo.

Listas duplamente encadeadas

Implementação

Método `remove`:

```
1  private E remove(Node<E> node) {
2    Node<E> predecessor = node.getPrev();
3    Node<E> successor = node.getNext();
4    predecessor.setNext(successor);
5    successor.setPrev(predecessor);
6    size--;
7    return node.getElement();
8  }
```

- Método genérico e encapsulado (privado) para remover um nodo, atualizando as referências entre seus vizinhos. O novo `next` do predecessor é o sucessor; o novo `prev` do sucessor é o predecessor.
- Método auxiliar para qualquer remoção.

Listas duplamente encadeadas

Implementação

Método `remove`:

```
1  private E remove(Node<E> node) {
2    Node<E> predecessor = node.getPrev();
3    Node<E> successor = node.getNext();
4    predecessor.setNext(successor);
5    successor.setPrev(predecessor);
6    size--;
7    return node.getElement();
8  }
```

- Método genérico e encapsulado (privado) para remover um nodo, atualizando as referências entre seus vizinhos. O novo `next` do predecessor é o sucessor; o novo `prev` do sucessor é o predecessor.
- Método auxiliar para qualquer remoção.

Métodos `removeFirst` e `removeLast`:

```
1  public E removeFirst() {
2    if (isEmpty()) return null;
3    return remove(header.getNext());
4  }
5
6  public E removeLast() {
7    if (isEmpty()) return null;
8    return remove(trailer.getPrev());
9  }
```

- Usam o método `remove` para remover o primeiro (`header.getNext()`) e o último (`trailer.getPrev()`) nodos.
- **Nota:** com sentinelas, o processo é o mesmo para qualquer nodo sendo removido.

Listas duplamente encadeadas

Implementação

Método `toString`:

```
1  public String toString() {
2    StringBuilder sb = new StringBuilder("(");
3    Node<E> walk = header.getNext();
4    while (walk != trailer) {
5      sb.append(walk.getElement());
6      walk = walk.getNext();
7      if (walk != trailer)
8        sb.append(", ");
9    }
10   sb.append(")");
11   return sb.toString();
12 }
```

- O percurso termina quando `walk` é igual ao `trailer`.

Comparação de listas encadeadas

Implementação

Na comparação de objetos, o Java compara os ponteiros. Em vez disso, podemos sobrescrever o método `equals` para comparar os elementos armazenados na lista.

```
1  public boolean equals(Object o) {
2    if (o == null) return false;
3    if (getClass() != o.getClass()) return false;
4    SinglyLinkedList other = (SinglyLinkedList) o;
5    if (size != other.size) return false;
6    Node<E> walkA = head;
7    Node<E> walkB = other.head;
8    while (walkA != null) {
9      if (!walkA.getElement().equals(walkB.getElement()))
10       return false;
11     walkA = walkA.getNext();
12     walkB = walkB.getNext();
13   }
14   return true;
15 }
```

Cuidados: referência nula, classes distintas, tamanho da lista.

Cópia de listas encadeadas

Implementação

Objetos em Java são copiados por referência. Em vez disso, podemos clonar uma lista (1) implementando a interface `Cloneable` e (2) sobcrescrevendo o método `clone`.

```
1  public class SinglyLinkedList<E> implements Cloneable {
2    //...
3    public SinglyLinkedList<E> clone() throws CloneNotSupportedException {
4      SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone();
5      if (size > 0) {
6        other.head = new Node<>(head.getElement(), null);
7        Node<E> walk = head.getNext();
8        Node<E> otherTail = other.head;
9        while (walk != null) {
10         Node<E> newest = new Node<>(walk.getElement(), null);
11         otherTail.setNext(newest);
12         otherTail = newest;
13         walk = walk.getNext();
14       }
15     }
16     return other;
17   }
18 }
```

Nota: o elemento também pode ser clonado, se necessário/desejado.

45EST – Algoritmos e Estruturas de Dados
Prof. Marcelo de Souza