**Ciência da Computação - CEULP/ULBRA**

**Análise de Algoritmos**

**Lista Duplamente Encadeada e Não Ordenada**

**Jhemeson Mota, Luan Gomes, Pedro Sousa**

**Palmas-TO**

**2017**

**1. Descrição Geral**

É um tipo de lista encadeada que pode ser vazia (NULL) ou que pode ter um ou mais nós, sendo que cada nó possui no mínimo três campos: um para o dado ou informação à ser armazenada e dois ponteiros onde um aponta para o nó antecessor e outro que aponta para o nó sucessor.

A lista duplamente encadeada pode ser circular (o último termo aponta para o primeiro) ou não e ordenada (crescente/decrescente) ou desordenada.

A vantagem da utilização da lista duplamente encadeada sobre a lista simplesmente encadeada é a maior facilidade para navegação, que pode ser feita nos dois sentidos, ou seja, do início para o fim e do fim para o início.

Não existindo a necessidade de percorrer a lista de trás para frente, a lista simplesmente encadeada é mais interessante por ser mais simples.

**2. Algoritmo No(param, param, param)**

**Descrição**

O algoritmo recebe três parâmetros, sendo que o primeiro deles, chamado nesse contexto de “anterior”, representará um ponteiro do nó atual que apontará para o valor anterior (na verdade, guardará a referência de um valor, mas para fins didáticos, chamaremos aqui de valor). O segundo deles, representa o valor do nó em si. O último, representa um ponteiro do nó atual que apontará para o próximo valor na lista.

**Algoritmo**

1 class No():

2 def \_\_init\_\_ (self,anterior = None, valor = None, proximo = None):

3 self.info = valor

4 self.prox = proximo

5 self.ant = anterior

**Complexidade - ϴ(1)**

3. ϴ(1) - Atribuição

4. ϴ(1) - Atribuição

5. ϴ(1) - Atribuição

**3. Inserção no início da lista**

**Descrição**

O algoritmo recebe como parâmetro o valor que o se deseja inserir na lista. Se a lista estiver vazia, cria-se um nó com o valor, e as variáveis que apontam para o primeiro (self.prim) e para o último (self.ult) elemento receberão este nó como valor. Se a lista não estiver vazia, a variável que aponta para o primeiro elemento (self.prim), receberá um nó com o valor que se deseja inserir, e que tem o ponteiro “prox” (que representa o próximo valor da lista a partir do atual) apontando para o valor atual da variável “self.prim”, e o valor da variável “ant” (que representa o elemento anterior da lista a partir do atual) do nó que era o antigo “self.prim” receberá o valor do “self.prim” atual, desta forma, fazendo com que cada elemento da lista aponte para o nó anterior (menos o primeiro elemento, visto que o mesmo não possui um nó anterior), e para o próximo valor (com exceção do último elemento, já que não existem valores depois deste).

**Algoritmo**

1 def inserirInicio(self,valor):

2 if self.quant == 0:

3 self.prim = No(None, valor, None)

4 self.ult = self.prim

5 else:

6 self.prim = No(None, valor, self.prim)

7 self.prim.prox.ant = self.prim

8 self.quant+=1

**Complexidade - ϴ(1)**

2. ϴ(1) - Condicional

3. ϴ(1) - Função No(param, param, param)

4. ϴ(1) - Atribuição

6. ϴ(1) - Função No(param, param, param)

7. ϴ(1) - Atribuição

8. ϴ(1) - Atribuição

**4. Inserção no fim da lista**

**Descrição**

O algoritmo recebe o valor que deseja inserir na lista. Se a lista estiver vazia, cria um novo nó com o valor e os ponteiros que aponta para o primeiro e último nó, passam a apontar para o novo nó criado. Se a lista não estiver vazia, cria um novo nó, com o nó anterior apontando para o último e o ponteiro para o último nó passa a apontar para o novo nó criado. O antigo último nó passa a ter como o próximo nó, o novo nó criado.

**Algoritmo**

1 def inserirFim(self, valor):

2 if self.quant == 0:

3 self.prim = self.ult = No(None, valor, None)

4 else:

5 self.ult = No(self.ult, valor, None)

6 self.ult.ant.prox = self.ult

7 self.quant += 1

**Complexidade - ϴ(1)**

2. ϴ(1) - Condicional

3. ϴ(1) - Atribuição

5. ϴ(1) - Algoritmo No(param, param, param)

6. ϴ(1) - Atribuição

7. ϴ(1) - Atribuição

**5. Inserção Ordenada**

Por se tratar de uma lista não ordenada, esta função não será aplicável.

**6. Consultar toda a lista**

**Descrição**

Inicia a partir do primeiro elemento. Entra em um laço e percorre todos os elementos, em que a cada iteração um novo nó é acessado. O laço fecha quando todos os elementos forem acessados.

**Algoritmo**

1 def consultarLista(self):  
 2 aux = self.prim  
 3 while aux != None:  
 4 aux = aux.prox

**Complexidade - ϴ(n)**

2. ϴ(1) - Atribuição

3. ϴ(n) - Laço que percorre a lista

4. ϴ(1) - Atribuição

**7. Remoção**

**Descrição**

Primeiramente, é importante destacar que o simples fato de não haver nenhum ponteiro apontando para um elemento faz com que o *Garbage Collector* apague-o posteriormente, portanto, essa ação é o suficiente para excluirmos um item da lista.

A função “remover” recebe como parâmetro o valor a ser removido da lista. Uma variável auxiliar receberá uma referência para o primeiro elemento da lista (self.prim), então será iniciado um laço que repetirá enquanto aux!= None.

Se o valor de aux for igual ao valor desejado iremos conferir se há apenas um elemento na lista, caso haja apenas um elemento na lista, os ponteiro *self.prim* e *self.ult* apontarão para o vazio, portanto 0. Caso haja mais de um elemento na lista, caso o elemento anterior ao *aux* não seja vazio, o campo *prox* do elemento anterior a *aux* apontará para o próximo elemento depois do atual. Caso o elemento anterior a aux seja vazio, significa que estamos lidando com o primeiro item da lista, portanto, *self.prim* irá apontar para o próximo elemento, e o campo *ant* do próximo elemento apontará para o vazio, visto que não há valor algum antes do primeiro elemento da lista.

Depois disso (Ainda dentro do “se o valor de aux for igual ao procurado”), confere-se se o próximo elemento não é vazio, caso não seja, o campo *ant* do próximo valor irá apontar para anterior do valor atual. Caso o próximo elemento seja vazio (significando aqui, que o valor procurado é o último elemento), *self.ult* irá apontar para o valor anterior ao valor atual, e o campo *prox* do valor anterior (novo último) irá apontar para o vazio (visto que, não deve haver nenhum valor depois do último valor da lista).

Se qualquer remoção for feita, o valor de *self.quant* será decrementado em uma unidade.

**Algoritmo**

1 def remover(self, valor):  
 2 aux = self.prim  
 3 while aux != None:  
 4 if aux.info == valor:  
 5 if self.quant == 1:  
 6 self.prim = self.ult = None  
 7 else:  
 8 if aux.ant != None:  
 9 aux.ant.prox = aux.prox  
 10 else:  
 11 self.prim = aux.prox  
 12 aux.prox.ant = None  
 13 if aux.prox != None:  
 14 aux.prox.ant = aux.ant  
 15 else:  
 16 self.ult = aux.ant  
 17 aux.ant.prox = None  
 18 self.quant -= 1  
 19 aux = None  
 20 aux = aux.prox

**Complexidade - O(n)**

2. ϴ(1) - Atribuição

3. O(n) - Repetição

4. ϴ(1) - Condicional

5. ϴ(1) - Condicional

6. ϴ(1) - Atribuição

8. ϴ(1) - Condicional

9. ϴ(1) - Atribuição

11. ϴ(1) - Atribuição

12. ϴ(1) - Atribuição

13. ϴ(1) - Condicional

14. ϴ(1) - Atribuição

16. ϴ(1) - Atribuição

17. ϴ(1) - Atribuição

18. ϴ(1) - Atribuição

19. ϴ(1) - Atribuição

20. ϴ(1) - Atribuição

**8. Esvaziar a lista**

**Descrição**

Define a quantidade de nós como zero, e define os ponteiros “self.prim” e “self.ult” para nulo.

**Algoritmo**

1 def esvaziarLista(self):

2 self.quant = 0

3 self.prim = self.ult = None

**Complexidade - ϴ(1)**

2. ϴ(1) - Atribuição

3. ϴ(1) - Atribuição