# Relatorio de Estrutura de Dados

Weverson da Silva Pereira

November 2020

## 1 Lista Dinâmica Duplamente Encadeada (LDDE)

### 1.1 Introdução

A lista dinâmica duplamente encadeada é uma derivação da lista dinâmica encadeada (LDE), mas agora com 2 ponteiros. Em um nó são armazenados, além do dado do nó, um ponteiro para o próximo elemento da lista e outro para o elemento anterior da lista, como na imagem a seguir:

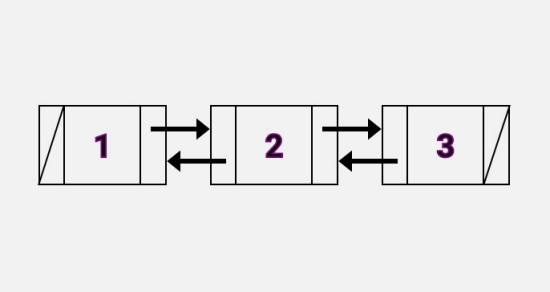


Figure 1: Lista Dinâmica Duplamente Encadeada

Note que o primeiro elemento não pode apontar para o nó anterior, logo ele aponta para NULL. O mesmo acontece com o último elemento da lista e o apontamento para o nó posterior.

### 1.2 Prós e Contras

Prós:

- Tamanho ilimitado
- Não ocupa mais memória do que precisa como as listas estáticas, por exemplo
- É possível acessá-la pelo começo ou pelo fim

- O tamanho da lista não é fixa
- Partindo de qualquer nó podemos acessar todos os outros, diferente da LDE

Contras:

- Os elementos não estão alocados de forma contínua na memória, dificultando o acesso
- Alguns algoritmos de busca, como a Binária, não são possíveis nessa estrutura e por isso se usa a busca linear.

## 1.3 Pseudocódigo

```
class Node( input Value )
    Initialize value to input Value
    Initialize next to NULL
    Initialize previous to NULL

class LDDE
    Initialize HEAD to NULL
    Initialize TAIL to NULL
```

**Algorithm 1:** Insert(new Value)

**Result:** True or false
Initialize NODE as Node with new Value;
Initialize ACTUAL as Node pointing to NULL;
Initialize PREVIOUS as Node pointing to NULL;
**while** *ACTUAL is not NULL and ACTUAL.value smaller than new Value* **do**
    PREVIOUS = ACTUAL;
    ACTUAL = ACTUAL.next;
**end**
Updates PREVIOUS.next to points to NODE;
Updates ACTUAL.previous to points to NODE;
Update NODE.previous to PREVIOUS;
Update NODE.next to ACTUAL;
Update HEAD if necessary;
Update TAIL if necessary;

**Algorithm 2:** Remove(target Value)

**Result:** True or false
Initialize ACTUAL as Node pointing to HEAD;
Initialize PREVIOUS as Node pointing to NULL;
**while** *ACTUAL is not NULL and ACTUAL.value smaller than target Value* **do**
    PREVIOUS = ACTUAL;
    ACTUAL = ACTUAL.next;
**end**
**if** *ACTUAL = NULL or ACTUAL.value and target Value are different* **then**
    return false;
**end**
Initialize NEXT as Node pointing to ACTUAL.next;
Updates PREVIOUS.next to points to NEXT;
Update HEAD if necessary;
Updates NEXT.previous to points to PREVIOUS;
Update TAIL if necessary;
Delete ACTUAL; return true;

**Algorithm 3:** Search(target Value)

**Result:** True or false
Initialize ACTUAL as Node pointing to HEAD;
**while** *ACTUAL is not NULL and ACTUAL.value smaller than target Value* **do**
    ACTUAL = ACTUAL.next;
**end**
**if** *ACTUAL is not NULL and ACTUAL.value = target Value* **then**
    return true;
**else**
    return false;
**end**

# 2 Tabela HASH

## 2.1 Introdução

É um estrutura que associa chaves de pesquisas a valores para promover uma busca mais eficiente. Essas chaves são são atribuidas aos valores através de uma função h(x) como na imagem a seguir. Existem vários métodos para a função HASH, a usada aqui foi o método da divisão: h(x) = x mod m, em que m é a quantidade de elementos no vetor.

## 2.2 Prós e Contras

Prós:

- Busca, Inserção e Remoção em O(1) na maioria dos casos

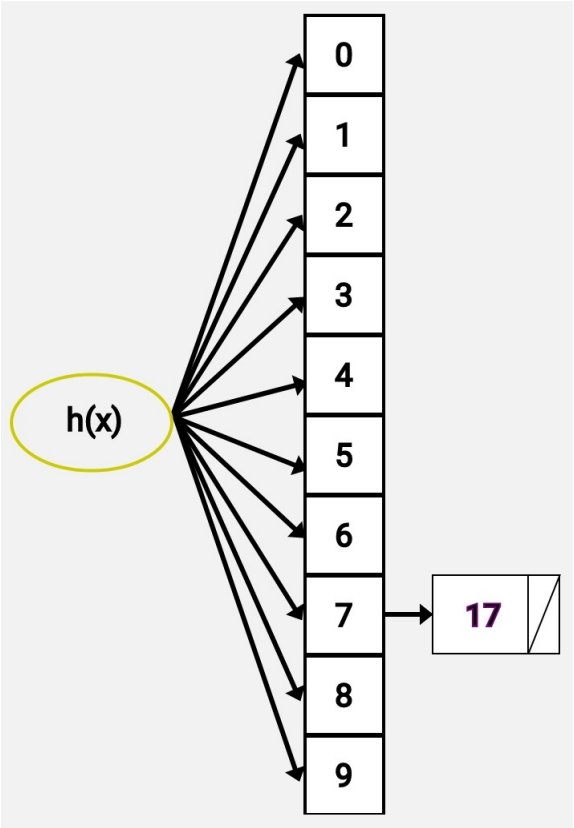


Figure 2: Tabela HASH

- Se torna mais viável quando o numero de elementos aumenta

Contras:

- Necessário tratamento de colisões, ou então a busca se aproximará de O(n)
- Perde-se a ordem da inserção, e não é possível ordená-la após isso

## 2.3 Pseudocódigo

```
class HASH
    Initialize MAX with the number of keys
    Initialize Hash Dictionary as a array of LDEs with size MAX
```

**Algorithm 4:** Hash Function(Value)

**Result:** Value mod MAX
return Value mod MAX;

**Algorithm 5:** Insert(new Value)

```
Result: True or false
Initialize INDEX as Hash Function(new Value);
Push value in Hash Dictionary[INDEX];
return true;
```

**Algorithm 6:** Remove(target Value)

```
Result: True or false
Initialize INDEX as Hash Function(target Value);
Initialize ACTUAL as Node pointing to Hash Dictionary[INDEX].HEAD;
if ACTUAL = NULL then
    return false;
end
while ACTUAL is not NULL and ACTUAL.value smaller than target Value do
    ACTUAL = ACTUAL.next;
end
if ACTUAL = NULL or ACTUAL.value and target Value are different then
    return false;
end
Delete ACTUAL;
return true;
```

**Algorithm 7:** Search(target Value)

```
Result: True or false
Initialize INDEX as Hash Function(target Value);
Initialize ACTUAL as Node pointing to Hash Dictionary[INDEX].HEAD;
while ACTUAL is not NULL and ACTUAL.value smaller than target Value do
    ACTUAL = ACTUAL.next;
end
if ACTUAL is not NULL and ACTUAL.value = target Value then
    return true;
else
    return false;
end
```

# 3 HEAP

## 3.1 Introdução

Um Heap é uma estrutura de árvore binária que segue baiscamante duas regras:

- Quando o último nível da árvore não estiver completa, as folhas devem estar mais a esquerda da árvore
- A folha Pai tem um valor sempre maior que a folha dos filhos

O Heap adota uma caracteristica da fila de prioridade (FIFO) e consegue lidar com valores repetidos Ele pode ser armazenado em uma árvore binária ou em um vetor, como mostra as imagens.

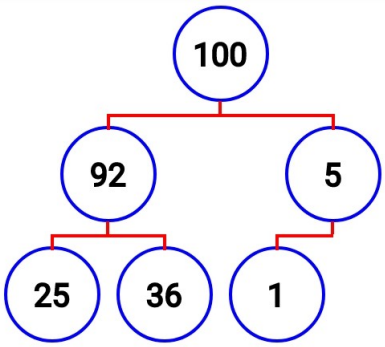


Figure 3: Heap em formato de árovre

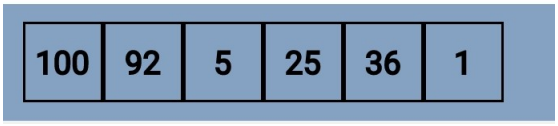


Figure 4: Heap em formato de vetor

## 3.2 Prós e Contras

Prós:

- Tempo O(1) para remoção e inserção no melhor dos casos, ou O(log n) no pior
- Também leva tempo O(1) para encontrar o maior elemento, já que ele sempre estará na posição 0 do vetor
- Se comparada com a árvore binária de busca, a Heap pode ser construída mais rapidamente, levando O(n). Enquanto a ABS leva O(n*log(n))

Contras:

- Os elementos não estão alocados de forma contínua na memória, dificultando o acesso
- Alguns algoritmos de busca, como a Binária, não são possíveis nessa estrutura e por isso se usa a busca linear.

## 3.3 Pseudocódigo

```
class HEAP
    Initialize Heap Array as an empty array
```

**Algorithm 8:** Left Son(Index)

```
Return 2*Index + 1;
```

**Algorithm 9:** Right Son(Index)

```
Return 2*Index + 2;
```

**Algorithm 10:** Father(Index)

```
Return (Index - 1) / 2;
```

**Algorithm 11:** Last Father()

```
Initialize size as the size of Heap Array;
if size less or equal 1 then
    Return -1;
else
    Return Father(size-1);
end
```

**Algorithm 12:** Sift(Index)

```
if Index bigger than Last Father(Index) then
    Return;
end
Initialize Bigger Son as the value of the largest node between the two
 childreen;
if Heap Array[Bigger Son] bigger than Heap Array[Index] then
    Swap Heap Array[Bigger Son] with Heap Array[Index] values;
    Call Sift again with Bigger Son as Index;
end
```

**Algorithm 13:** Insert(new Value)

```
Result: True or false
Push new Value to the Heap Array;
Initialize DAD as Last Father();
while DAD bigger than 0 do
    Call Sift(DAD);
    DAD now is equal his father;
end
Call again Sift with 0 as Index;
Return true;
```

**Algorithm 14:** Remove()

**Result:** True or false
**if** *Heap Array is empty* **then**
    Return false;
**end**
Swap the values at the first position of the Heap Array with the Last position;
Delete the last element of the Heap Array Call Sift from the first postion of the Heap Array Return true;