



Análise de algoritmos

# LISTA ENCADEADA

*Baseado no livro Entendendo Algoritmos, Um guia ilustrado para programadores e outros curiosos. Aditya Y. Bhargava, Novatec, 2017.*





Análise de algoritmos

# LISTA ENCADEADA

O computador possui uma série de compartimentos com endereço e slot de memória. Quando precisamos armazenar uma lista de elementos na memória, a primeira opção que nos vem à mente é realizar esse armazenamento de maneira contígua (sequencial), mas na prática, não é dessa forma que os dados são armazenados.

Nem sempre haverão espaços disponíveis na memória devido estarem alocados em outros programas e processos. Uma abordagem seria buscar um novo espaço na memória que comporte os dados, mas adicionar itens ao *array* pode ser lento. Reservando espaços superior ao esperado poderia evitar deslocamento mas, por outro lado, gerar desperdício de memória e dependendo do volume de dados, não sendo possível evitar a ocorrência de deslocamento.

Para resolver esse problema, as listas encadeadas não possuem um tamanho fixo e os elementos são conectados por meio de ponteiros.





Análise de algoritmos

# LISTA ENCADEADA

Em outras palavras, podemos manter os itens em qualquer lugar da memória, pois o item atual armazena o endereço do item seguinte e os endereços aleatórios de memória são mantidos interligados, não sendo necessário deslocamento se houver escala nos dados e nem mesmo desperdício de memória. E é por isso que geralmente, as listas encadeadas é uma opção melhor do que arrays.

## APLICAÇÕES

- Alocação de memória: alocação dinâmica e liberação eficiente de memória.
- Compiladores: estrutura sintática da linguagem.
- Editores de texto: armazenamento e conteúdo de textos.
- Sistemas de arquivos: armazenamento de metadados para acesso e organização eficiente dos dados.
- Representação de sequências: aplicações que exijam sequência de dados.





Análise de algoritmos

# LISTA ENCADEADA

Em estrutura de dados, podemos encontrar o algoritmo de lista encadeada nos seguintes casos:

- Pilhas (LIFO): armazenamento de elementos onde o último elemento é o primeiro a ser removido, aplicados em cenários de desfazer e refazer, gerenciamento de memória e expressões matemáticas.
- Filas (FIFO): armazenamento de elementos em que o primeiro elemento adicionado é o primeiro a ser removido. Encontra-se em filas de impressão, *buffer* de dados e agendamento de tarefas.
- Árvores: estruturas hierárquicas com raízes, ramos e folhas. Listas encadeadas podem representar conexões entre nós da árvore, permitindo operações eficientes como busca, inserção e remoção de elementos.
- Grafos: representam relações entre objetos, sendo o algoritmo de lista encadeada essencial para armazenar adjacências de cada nó no grafo, possibilitando implementação de algoritmo de busca como busca em profundidade e largura.





# Q LISTA ENCADEADA



```
package main

import "fmt"

type LinkedElement interface {
    Data() interface{}
    Next() *LinkedElement
    SetNext(*LinkedElement)
}

type Node struct {
    value interface{}
    next *Node
}
```





# Q LISTA ENCADEADA



```
func (n *Node) Data() interface{} {  
    return n.value  
}  
  
func (n *Node) Next() *Node {  
    return n.next  
}  
  
func (n *Node) SetNext(next *Node) {  
    n.next = next  
}
```





# Q LISTA ENCADEADA



```
type LinkedList interface {  
    InsertFirst(value interface{})  
    InsertLast(value interface{})  
    DeleteFirst()  
    DeleteLast()  
    Search(value interface{}) *Node  
    IsEmpty() bool  
    Length() int  
}
```

```
type LinkedListImplementation struct {  
    firstElement *Node  
    lastElement *Node  
}
```





# Q LISTA ENCADEADA



```
func (list *LinkedListImplementation) InsertFirst(value interface{}) {  
    newNode := &Node{value: value}  
    newNode.next = list.firstElement  
  
    if list.lastElement == nil {  
        list.lastElement = newNode  
    }  
    list.firstElement = newNode  
}
```







# Q LISTA ENCADEADA



```
func (list *LinkedListImplementation) InsertLast(value interface{}) {  
    newNode := &Node{value: value}  
  
    if list.lastElement == nil {  
        list.firstElement = newNode  
        list.lastElement = newNode  
    } else {  
        list.lastElement.next = newNode  
        list.lastElement = newNode  
    }  
}
```





# Q LISTA ENCADEADA



```
func (list *LinkedListImplementation) DeleteFirst() {  
    if list.IsEmpty() {  
        return  
    }  
    list.firstElement = list.firstElement.next  
  
    if list.firstElement == nil {  
        list.lastElement = nil  
    }  
}
```

```
func (list *LinkedListImplementation) DeleteLast() {  
    if list.IsEmpty() {  
        return  
    }  
}
```





# Q LISTA ENCADEADA



```
if list.firstElement == list.lastElement {  
    list.firstElement = nil  
    list.lastElement = nil  
    return  
}  
var previousNode *Node  
currentNode := list.firstElement  
  
for currentNode.next != nil {  
    previousNode = currentNode  
    currentNode = currentNode.next  
}  
previousNode.next = nil  
list.lastElement = previousNode  
}
```





# Q LISTA ENCADEADA



```
func (list *LinkedListImplementation) Search(value interface{}) *Node {
    currentNode := list.firstElement

    for currentNode != nil {
        if currentNode.value == value {
            return currentNode
        }
        currentNode = currentNode.next
    }
    return nil
}

func (list *LinkedListImplementation) IsEmpty() bool {
    return list.firstElement == nil
}
```





# Q LISTA ENCADEADA



```
func (list *LinkedListImplementation) Length() int {  
    count := 0  
    currentNode := list.firstElement  
  
    for currentNode != nil {  
        count++  
        currentNode = currentNode.next  
    }  
    return count  
}
```

```
func (list *LinkedListImplementation) PrintList() {  
    currentNode := list.firstElement
```





# Q LISTA ENCADEADA



```
    for currentNode != nil {  
        fmt.Print(currentNode.Data(), " -> ")  
        currentNode = currentNode.Next()  
    }  
    fmt.Println("nil")  
}
```

```
func main() {  
    list := &LinkedListImplementation{}  
  
    list.InsertFirst("A")  
    list.InsertFirst("B")  
    list.InsertFirst("C")  
  
    fmt.Println("list:", list)
```





## Q LISTA ENCADEADA



```
element := "C"
node := list.Search(element)

if node != nil {
    fmt.Println("element", element, "found in position", node)
} else {
    fmt.Println("element", element, "not found.")
}

list.DeleteFirst()
fmt.Println("list:", list)
}
```





# SAÍDA DO PROGRAMA



## CENÁRIO OTIMISTA

list: &{0xc000008078 0xc000008048}  
element C found in position &{C 0xc000008060}  
list: &{0xc000008060 0xc000008048}

## CENÁRIO PESSIMISTA

list: &{0xc000094060 0xc000094030}  
element D not found.  
list: &{0xc000094048 0xc000094030}







Análise de algoritmos

# CONSIDERAÇÕES

Para o exemplo apresentado, seguiu-se as boas práticas de desenvolvimento com *clean code* e SOLID, na tentativa de simular um cenário otimista.

Essa iniciativa vai de encontro com a ideia de trazer conteúdos relevantes altamente abordados em processos seletivos e desmitificar a ideia de algoritmos e estrutura de dados. Espero que seja de bom proveito e bons estudos.