



# Análise e Projeto de Algoritmos

## Aula 3

---

Thiago Cavalcante – [thiago.cavalcante@penedo.ufal.br](mailto:thiago.cavalcante@penedo.ufal.br)

08 de novembro de 2019

Universidade Federal de Alagoas – UFAL

Campus Arapiraca

Unidade de Ensino de Penedo

# Tipos abstratos de dados

- Contêineres (filas, pilhas)
- Dicionários
- Filas de prioridade

Tipos abstratos de dados são implementados com **estruturas de dados**

# Estruturas contíguas × ligadas

- **Único** pedaço de memória
- Vetores (array 1D)
- Matrizes (array 2D)
- Heaps
- Tabelas de dispersão (*hash tables*)
- **Vários** pedaços de memória conectados por **ponteiros**
- Listas
- Árvores
- Listas de adjacência (grafos)

# Arrays

Estruturas de **dados com tamanho fixo** onde cada elemento pode ser acessado de forma **eficiente** pelo seu **índice** (ou **endereço**)

# Vantagens de um array

- Acesso aos elementos é  $O(1)$ , dado o índice
- Eficiência de espaço: composto puramente de dados com o mesmo tamanho
- Continuidade física dos dados na memória

# Principal desvantagem de um array: tamanho fixo

Possíveis soluções

- Alocar um array grande o suficiente (possível desperdício de memória)
- Arrays **dinâmicos** (ex.)

# Ponteiros e estruturas ligadas

- Representam **endereços** na memória
- Oferecem maior **flexibilidade**
- Em C: operadores **\*** e **&**, valor **NULL**



# Implementação de uma lista

```
typedef struct lista {  
    tipo dado; // -> dados  
    Lista *prox; // -> ponteiro para o  
} Lista;        //      proximo elemento
```

# Características da lista ligada

- Cada elemento armazena um ou mais dados
- Cada elemento possui também um **ponteiro para o próximo elemento** (ocupa mais espaço na memória)
- É necessário sempre o **ponteiro para o primeiro elemento** da lista
- Estrutura ligada mais simples
- Pode ser **duplamente** encadeada (ponteiros para o anterior e o próximo, ocupa mais espaço)

# Busca em uma lista

```
Lista *busca_lista(Lista *l, tipo x) {  
    if (l == NULL) // lista vazia  
        return(NULL);  
  
    if (l->dado == x) // dado encontrado  
        return(l);  
    else // dado nao encontrado  
        return(busca_lista(l->prox, x));  
}
```

# Inserção em uma lista

```
void inserir_lista(Lista **l, tipo x) {  
    Lista *p; // ponteiro auxiliar  
  
    p = malloc(sizeof(Lista));  
    p->dado = x;  
    p->prox = *l;  
    *l = p;  
}
```

# Remoção em uma lista

```
Lista *anterior(Lista *l, tipo x) {  
    if ((l == NULL) || (l->prox == NULL)) {  
        printf("Nao encontrado");  
        return(NULL);  
    }  
    if ((l->prox)->dado == x)  
        return(l);  
    else  
        return(anterior(l->prox, x));  
}
```

# Remoção em uma lista

```
void remover_lista(Lista **l, tipo x) {  
    Lista *p, *pred;  
    p = busca_lista(*l,x);  
    if (p != NULL) {  
        pred = anterior(*l,x);  
        if (pred == NULL)  
            *l = p->prox;  
        else  
            pred->prox = p->prox;  
        free(p);  
    }  
}
```

# Vantagens das listas

- Uma lista nunca vai "estourar" a memória, a não ser que a memória do PC acabe
- Inserção e remoção são mais simples
- Gerenciamento de ponteiros é mais simples em larga escala

# Desvantagens das listas

- Precisam de mais espaço para os ponteiros
- Acesso aleatório não é eficiente
- Memória espalhada diminui a performance



**Listas e arrays são estruturas  
recursivas**

# Pilhas (stacks)

- *last-in, first-out* (LIFO)
- Simples de implementar
- Eficientes
- Ordem não importa
- Operações: `push(p, x)` e `pop(p)`
- Aparecem em algoritmos recursivos

# Filas (queues)

- *first-in, first-out* (FIFO)
- Complicadas de implementar
- Minimizam o tempo máximo de espera
- Ordem importa
- Operações: enqueue(*f*, *x*) e dequeue(*p*)

**Pilhas e filas podem ser implementadas tanto com arrays quanto com listas**

# Dicionários

Armazenam um conjunto de **dados**, indexados por chaves.

```
{"chave1": dado1, "chave2": dado2, ... }
```

# Operações com dicionários

- $\text{Busca}(D, c)$
- $\text{Inserção}(D, x)$
- $\text{Remoção}(D, x)$
- $\text{Máx}(D) / \text{Mín}(D)$
- $\text{Anterior}(D, k) / \text{Próximo}(D, k)$

Exemplo: lista de elementos únicos

# **Exercício: implementação de dicionário com array**

Array ordenado × não ordenado

<b>Operação</b>	<b>Array não ordenado</b>	<b>Array ordenado</b>
Busca		
Inserção		
Remoção		
Próximo		
Anterior		
Mín		
Máx		

Assumindo que o número  $n$  de elementos no array é dado



Operação	Array não ordenado	Array ordenado
Busca	$O(n)$	
Inserção		
Remoção		
Próximo		
Anterior		
Mín		
Máx		

Assumindo que o número  $n$  de elementos no array é dado

Operação	Array não ordenado	Array ordenado
Busca	$O(n)$	
Inserção	$O(1)$	
Remoção		
Próximo		
Anterior		
Mín		
Máx		

Assumindo que o número  $n$  de elementos no array é dado

Operação	Array não ordenado	Array ordenado
Busca	$O(n)$	
Inserção	$O(1)$	
Remoção	$O(1)^*$	
Próximo		
Anterior		
Mín		
Máx		

Assumindo que o número  $n$  de elementos no array é dado

Operação	Array não ordenado	Array ordenado
Busca	$O(n)$	
Inserção	$O(1)$	
Remoção	$O(1)^*$	
Próximo	$O(n)$	
Anterior		
Mín		
Máx		

Assumindo que o número  $n$  de elementos no array é dado

Operação	Array não ordenado	Array ordenado
Busca	$O(n)$	
Inserção	$O(1)$	
Remoção	$O(1)^*$	
Próximo	$O(n)$	
Anterior	$O(n)$	
Mín		
Máx		

Assumindo que o número  $n$  de elementos no array é dado

Operação	Array não ordenado	Array ordenado
Busca	$O(n)$	
Inserção	$O(1)$	
Remoção	$O(1)^*$	
Próximo	$O(n)$	
Anterior	$O(n)$	
Mín	$O(n)$	
Máx		

Assumindo que o número  $n$  de elementos no array é dado

Operação	Array não ordenado	Array ordenado
Busca	$O(n)$	
Inserção	$O(1)$	
Remoção	$O(1)^*$	
Próximo	$O(n)$	
Anterior	$O(n)$	
Mín	$O(n)$	
Máx	$O(n)$	

Assumindo que o número  $n$  de elementos no array é dado

Operação	Array não ordenado	Array ordenado
Busca	$O(n)$	$O(\log n)$
Inserção	$O(1)$	
Remoção	$O(1)^*$	
Próximo	$O(n)$	
Anterior	$O(n)$	
Mín	$O(n)$	
Máx	$O(n)$	

Assumindo que o número  $n$  de elementos no array é dado



Operação	Array não ordenado	Array ordenado
Busca	$O(n)$	$O(\log n)$
Inserção	$O(1)$	$O(n)$
Remoção	$O(1)^*$	
Próximo	$O(n)$	
Anterior	$O(n)$	
Mín	$O(n)$	
Máx	$O(n)$	

Assumindo que o número  $n$  de elementos no array é dado

Operação	Array não ordenado	Array ordenado
Busca	$O(n)$	$O(\log n)$
Inserção	$O(1)$	$O(n)$
Remoção	$O(1)^*$	$O(n)$
Próximo	$O(n)$	
Anterior	$O(n)$	
Mín	$O(n)$	
Máx	$O(n)$	

Assumindo que o número  $n$  de elementos no array é dado

Operação	Array não ordenado	Array ordenado
Busca	$O(n)$	$O(\log n)$
Inserção	$O(1)$	$O(n)$
Remoção	$O(1)^*$	$O(n)$
Próximo	$O(n)$	$O(1)$
Anterior	$O(n)$	
Mín	$O(n)$	
Máx	$O(n)$	

Assumindo que o número  $n$  de elementos no array é dado

Operação	Array não ordenado	Array ordenado
Busca	$O(n)$	$O(\log n)$
Inserção	$O(1)$	$O(n)$
Remoção	$O(1)^*$	$O(n)$
Próximo	$O(n)$	$O(1)$
Anterior	$O(n)$	$O(1)$
Mín	$O(n)$	
Máx	$O(n)$	

Assumindo que o número  $n$  de elementos no array é dado

Operação	Array não ordenado	Array ordenado
Busca	$O(n)$	$O(\log n)$
Inserção	$O(1)$	$O(n)$
Remoção	$O(1)^*$	$O(n)$
Próximo	$O(n)$	$O(1)$
Anterior	$O(n)$	$O(1)$
Mín	$O(n)$	$O(1)$
Máx	$O(n)$	

Assumindo que o número  $n$  de elementos no array é dado

Operação	Array não ordenado	Array ordenado
Busca	$O(n)$	$O(\log n)$
Inserção	$O(1)$	$O(n)$
Remoção	$O(1)^*$	$O(n)$
Próximo	$O(n)$	$O(1)$
Anterior	$O(n)$	$O(1)$
Mín	$O(n)$	$O(1)$
Máx	$O(n)$	$O(1)$

Assumindo que o número  $n$  de elementos no array é dado

# Exercício: implementação de dicionário com lista

Lista simplesmente ligada ordenada × lista  
simplesmente ligada não ordenada × lista  
duplamente ligada ordenada × lista  
duplamente ligada não ordenada

<b>Operação</b>	<b>Simp. não ord.</b>	<b>Dup. não ord.</b>	<b>Simp. ord.</b>	<b>Dup. ord.</b>
Busca				
Inserção				
Remoção				
Próximo				
Anterior				
Mín				
Máx				



# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$			
Inserção				
Remoção				
Próximo				
Anterior				
Mín				
Máx				

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$			
Inserção	$O(1)$			
Remoção				
Próximo				
Anterior				
Mín				
Máx				

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$			
Inserção	$O(1)$			
Remoção	$O(n)^*$			
Próximo				
Anterior				
Mín				
Máx				

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$			
Inserção	$O(1)$			
Remoção	$O(n)^*$			
Próximo	$O(n)$			
Anterior				
Mín				
Máx				

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$			
Inserção	$O(1)$			
Remoção	$O(n)^*$			
Próximo	$O(n)$			
Anterior	$O(n)$			
Mín				
Máx				

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$			
Inserção	$O(1)$			
Remoção	$O(n)^*$			
Próximo	$O(n)$			
Anterior	$O(n)$			
Mín	$O(n)$			
Máx				

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$			
Inserção	$O(1)$			
Remoção	$O(n)^*$			
Próximo	$O(n)$			
Anterior	$O(n)$			
Mín	$O(n)$			
Máx	$O(n)$			

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$		
Inserção	$O(1)$			
Remoção	$O(n)^*$			
Próximo	$O(n)$			
Anterior	$O(n)$			
Mín	$O(n)$			
Máx	$O(n)$			



# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$		
Inserção	$O(1)$	$O(1)$		
Remoção	$O(n)^*$			
Próximo	$O(n)$			
Anterior	$O(n)$			
Mín	$O(n)$			
Máx	$O(n)$			

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$		
Inserção	$O(1)$	$O(1)$		
Remoção	$O(n)^*$	$O(1)$		
Próximo	$O(n)$			
Anterior	$O(n)$			
Mín	$O(n)$			
Máx	$O(n)$			

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$		
Inserção	$O(1)$	$O(1)$		
Remoção	$O(n)^*$	$O(1)$		
Próximo	$O(n)$	$O(n)$		
Anterior	$O(n)$			
Mín	$O(n)$			
Máx	$O(n)$			

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$		
Inserção	$O(1)$	$O(1)$		
Remoção	$O(n)^*$	$O(1)$		
Próximo	$O(n)$	$O(n)$		
Anterior	$O(n)$	$O(n)$		
Mín	$O(n)$			
Máx	$O(n)$			

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$		
Inserção	$O(1)$	$O(1)$		
Remoção	$O(n)^*$	$O(1)$		
Próximo	$O(n)$	$O(n)$		
Anterior	$O(n)$	$O(n)$		
Mín	$O(n)$	$O(n)$		
Máx	$O(n)$			

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$		
Inserção	$O(1)$	$O(1)$		
Remoção	$O(n)^*$	$O(1)$		
Próximo	$O(n)$	$O(n)$		
Anterior	$O(n)$	$O(n)$		
Mín	$O(n)$	$O(n)$		
Máx	$O(n)$	$O(n)$		

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$	$O(n)$	
Inserção	$O(1)$	$O(1)$		
Remoção	$O(n)^*$	$O(1)$		
Próximo	$O(n)$	$O(n)$		
Anterior	$O(n)$	$O(n)$		
Mín	$O(n)$	$O(n)$		
Máx	$O(n)$	$O(n)$		

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$	$O(n)$	
Inserção	$O(1)$	$O(1)$	$O(n)$	
Remoção	$O(n)^*$	$O(1)$		
Próximo	$O(n)$	$O(n)$		
Anterior	$O(n)$	$O(n)$		
Mín	$O(n)$	$O(n)$		
Máx	$O(n)$	$O(n)$		



# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$	$O(n)$	
Inserção	$O(1)$	$O(1)$	$O(n)$	
Remoção	$O(n)^*$	$O(1)$	$O(n)^*$	
Próximo	$O(n)$	$O(n)$		
Anterior	$O(n)$	$O(n)$		
Mín	$O(n)$	$O(n)$		
Máx	$O(n)$	$O(n)$		

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$	$O(n)$	
Inserção	$O(1)$	$O(1)$	$O(n)$	
Remoção	$O(n)^*$	$O(1)$	$O(n)^*$	
Próximo	$O(n)$	$O(n)$	$O(1)$	
Anterior	$O(n)$	$O(n)$		
Mín	$O(n)$	$O(n)$		
Máx	$O(n)$	$O(n)$		

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$	$O(n)$	
Inserção	$O(1)$	$O(1)$	$O(n)$	
Remoção	$O(n)^*$	$O(1)$	$O(n)^*$	
Próximo	$O(n)$	$O(n)$	$O(1)$	
Anterior	$O(n)$	$O(n)$	$O(n)^*$	
Mín	$O(n)$	$O(n)$		
Máx	$O(n)$	$O(n)$		

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$	$O(n)$	
Inserção	$O(1)$	$O(1)$	$O(n)$	
Remoção	$O(n)^*$	$O(1)$	$O(n)^*$	
Próximo	$O(n)$	$O(n)$	$O(1)$	
Anterior	$O(n)$	$O(n)$	$O(n)^*$	
Mín	$O(n)$	$O(n)$	$O(1)$	
Máx	$O(n)$	$O(n)$		

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$	$O(n)$	
Inserção	$O(1)$	$O(1)$	$O(n)$	
Remoção	$O(n)^*$	$O(1)$	$O(n)^*$	
Próximo	$O(n)$	$O(n)$	$O(1)$	
Anterior	$O(n)$	$O(n)$	$O(n)^*$	
Mín	$O(n)$	$O(n)$	$O(1)$	
Máx	$O(n)$	$O(n)$	$O(1)^*$	

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Inserção	$O(1)$	$O(1)$	$O(n)$	
Remoção	$O(n)^*$	$O(1)$	$O(n)^*$	
Próximo	$O(n)$	$O(n)$	$O(1)$	
Anterior	$O(n)$	$O(n)$	$O(n)^*$	
Mín	$O(n)$	$O(n)$	$O(1)$	
Máx	$O(n)$	$O(n)$	$O(1)^*$	

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Inserção	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Remoção	$O(n)^*$	$O(1)$	$O(n)^*$	
Próximo	$O(n)$	$O(n)$	$O(1)$	
Anterior	$O(n)$	$O(n)$	$O(n)^*$	
Mín	$O(n)$	$O(n)$	$O(1)$	
Máx	$O(n)$	$O(n)$	$O(1)^*$	

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Inserção	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Remoção	$O(n)^*$	$O(1)$	$O(n)^*$	$O(1)$
Próximo	$O(n)$	$O(n)$	$O(1)$	
Anterior	$O(n)$	$O(n)$	$O(n)^*$	
Mín	$O(n)$	$O(n)$	$O(1)$	
Máx	$O(n)$	$O(n)$	$O(1)^*$	



# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Inserção	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Remoção	$O(n)^*$	$O(1)$	$O(n)^*$	$O(1)$
Próximo	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Anterior	$O(n)$	$O(n)$	$O(n)^*$	
Mín	$O(n)$	$O(n)$	$O(1)$	
Máx	$O(n)$	$O(n)$	$O(1)^*$	

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Inserção	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Remoção	$O(n)^*$	$O(1)$	$O(n)^*$	$O(1)$
Próximo	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Anterior	$O(n)$	$O(n)$	$O(n)^*$	$O(1)$
Mín	$O(n)$	$O(n)$	$O(1)$	
Máx	$O(n)$	$O(n)$	$O(1)^*$	

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Inserção	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Remoção	$O(n)^*$	$O(1)$	$O(n)^*$	$O(1)$
Próximo	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Anterior	$O(n)$	$O(n)$	$O(n)^*$	$O(1)$
Mín	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Máx	$O(n)$	$O(n)$	$O(1)^*$	

# Estruturas de Dados

Operação	Simp. não ord.	Dup. não ord.	Simp. ord.	Dup. ord.
Busca	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Inserção	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Remoção	$O(n)^*$	$O(1)$	$O(n)^*$	$O(1)$
Próximo	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Anterior	$O(n)$	$O(n)$	$O(n)^*$	$O(1)$
Mín	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Máx	$O(n)$	$O(n)$	$O(1)^*$	$O(1)$

# Árvores binárias de busca

- Até agora: busca rápida ou atualização flexível, não ambas
- Busca binária requer acesso aos nós medianos **acima** e **abaixo** do que se procura
- Árvore binária de busca é uma "lista ligada" com dois ponteiros por elemento

# Definição recursiva de árvore binária

1. Vazia, ou
2. Elemento chamado de **raiz** com duas outras árvores binárias, chamadas **subárvores** da **esquerda** e da **direita**

A ordem dos elementos importa

**Em uma árvore binária de busca, todos os nós na subárvore da esquerda são menores que a raiz e todos os nós na subárvore da direita são maiores que a raiz**

Com uma árvore binária de  $n$  elementos e um conjunto de  $n$  valores, só existe **uma maneira** de criar uma árvore binária de busca

# Implementação de uma árvore

```
typedef struct arvore {  
    tipo dado;  
    struct Arvore *pai; // opcional  
    struct Arvore *esquerda;  
    struct Arvore *direita;  
} Arvore;
```



# Operações básicas

- Busca
- Travessia
- Inserção
- Remoção

# Busca em uma árvore

```
Arvore *busca_arvore(Arvore *l, tipo x) {  
    if (l == NULL) return(NULL);  
    if (l->dado == x) return(l);  
    if (x < l->dado)  
        return(busca_arvore(l->esquerda, x));  
    else  
        return(busca_arvore(l->direita, x));  
}
```

Roda em tempo  $O(h)$ , onde  $h$  é a altura da árvore

# Mínimo/máximo em uma árvore

```
Arvore *minimo(Arvore *t) {  
    Arvore *min; // ponteiro auxiliar  
    if (t == NULL) return(NULL);  
    min = t;  
    while (min->esquerda != NULL)  
        min = min->esquerda;  
    return(min);  
}
```

# Atravessando uma árvore

```
void atravessar_arvore(Arvore *l) {  
    if (l != NULL) {  
        atravessar_arvore(l->esquerda);  
        processar_elemento(l->dado);  
        atravessar_arvore(l->direita);  
    }  
}
```

A travessia ocorre em **ordem ascendente** dos dados, em um tempo  $O(n)$  (cada um dos  $n$  elementos é processado uma vez)

# Inserção em uma árvore

Realizada no ponto onde a busca pelo elemento que se deseja inserir falha (encontra um elemento **NULL**)

Roda em tempo  $O(h)$ , onde  $h$  é a altura da árvore

## Estruturas de Dados

```
insere_arvore(Arvore **l, tipo x, Arvore *pai) {  
    Arvore *p; // ponteiro auxiliar  
    if (*l == NULL) {  
        p = malloc(sizeof(Arvore)); // alocação  
        p->dado = x;  
        p->esquerda = p->direita = NULL;  
        p->pai = pai;  
        *l = p; // conectando a arvore  
        return;  
    }  
    if (x < (*l)->dado)  
        insere_arvore(&((*l)->esquerda), x, *l);  
    else  
        insere_arvore(&((*l)->direita), x, *l);  
}
```

# Remoção em uma árvore

Três possibilidades

1. Elemento não tem filhos
2. Elemento tem um filho
3. Elemento tem dois filhos (complexo)

Roda em tempo  $O(h)$ , onde  $h$  é a altura da árvore

### Pior caso

Altura  $h$  é igual ao número  $n$  de elementos, árvore vira uma lista encadeada e todos os algoritmos rodam no tempo  $O(n)$



### Melhor caso

A árvore é **perfeitamente balanceada** (dois filhos por elemento), a altura é igual a  $\lceil \log_2 n \rceil$  e todos os algoritmos rodam no tempo  $O(\log_2 n)$

Na **média**, existe uma **alta probabilidade** de a árvore ter altura  $O(\log_2 n)$

# Exercício: lendo $n$ números e imprimindo na ordem com um dicionário balanceado

1. Usando inserção e travessia
2. Usando inserção, mínimo e próximo
3. Usando inserção, mínimo e remoção