SEMINÁRIO BASEADO NO LIVRO DE JOHN BENTLEY

# Programming Pearls

## Column 13: Searching


UNIVERSIDADE

Universidade Vila Velha

PROFESSOR

Abrantes

DISCIPLINA

Teoria da Computação

INTEGRANTES

André Luiz Gomes dos Santos Filho

Thomás Kriger

# O que esta coluna aborda?

PERGUNTA CENTRAL

"Como encontrar rapidamente um elemento dentro de um grande conjunto de dados?"

## A coluna discute:

- Algoritmos de busca
- Diferentes estruturas de dados
- Organização de dados para facilitar buscas
- Comparação de eficiência entre métodos

IDEIA CENTRAL DE BENTLEY

*"A forma como organizamos os dados é tão importante quanto o algoritmo de busca."*

Programming Pearls

# O Problema da Busca



## Exemplos Práticos

### Listas Simples

Procurar um nome específico em uma lista de presença ou lista telefônica.

### Dicionários

Buscar a definição de uma palavra (chave) para obter seu significado (valor).

### Bancos de Dados

Encontrar um registro de usuário (ID) em tabelas com milhões de linhas.

### Motores de Busca

Google/Bing indexando e recuperando páginas da web relevantes.

## Abstração do Problema

Estrutura geral independente da linguagem

problem_definition.c

```c
/**
* Definição formal do problema de busca
*/
struct Result search(DataSet S, Element x) {
    // S: Conjunto de dados onde procuramos (The Haystack)
    // x: O elemento específico desejado (The Needle)

    if (x exists_in S) {
        return x.location(); // Retorna índice, ponteiro ou objeto
    } else {
        return NULL; // Elemento não encontrado
    }
}
```

**Desafio de Design**

A eficiência desta função `search` depende inteiramente de como `S` está organizado (lista, array ordenado, árvore, hash...).

# Busca Linear (Sequencial)



## Características

### Ideia Central

Percorrer todos os elementos um por um até encontrar $x$.

### Vantagens

Simples de implementar e funciona em qualquer conjunto de dados (mesmo desordenados).

### Limitação

Extremamente lenta para grandes volumes de dados.

COMPLEXIDADE DE TEMPO

**O(n)**

Pior caso

Percorre n elementos

## Implementação

Algoritmo de força bruta

linear_search.c

```c
/**
* Busca linear: percorre o array A de tamanho n
*/
int linear_search(int A[], int n, int x) {
    int i;

    // Loop percorre de 0 até o último elemento
    for (i = 0; i < n; i++) {

        // Se encontrar o elemento, retorna o índice
        if (A[i] == x) {
            return i;
        }
    }

    return -1; // Elemento não encontrado
}
```

# Melhorando a busca



PREMISSA

"Se os dados estão ordenados, podemos fazer muito melhor."

## Exemplo de Dados Ordenados

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 5 | 9 | 12 | 20 | 30 |

✓ ORDENADO (CRESCENTE)

DESBLOQUEIA

**Busca Binária**

NOVA COMPLEXIDADE

`O(log n)`

INSIGHT DE ENGENHARIA

### O custo da organização

"Investir em ordenação previamente pode reduzir drasticamente o custo de buscas futuras."

Ideal para conjuntos de dados estáticos ou onde a frequência de busca é muito maior que a de inserção.

# Busca Binária



## Exemplo Visual

Buscando o elemento **20** no vetor ordenado:

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 5 | 9 | **12** | 20 | 30 |

**Passo 1** **Analisar o Meio**

Meio = índice 3 (valor 12)

Como **20 > 12**, eliminamos a metade esquerda.

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 5 | 9 | 12 | **20** | 30 |

**Passo 2** **Novo Meio**

Novo intervalo: [20, 30]. Novo meio = 20.

**Encontrado!**

## Algoritmo: Divide & Conquer

Estratégia logarítmica para dados ordenados

`O(log n)`

binary_search.c

```c
int binary_search(int A[], int n, int x) {
    int low = 0, high = n - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (A[mid] == x)
            return mid; // Encontrado!
        else if (A[mid] < x)
            low = mid + 1; // Busca na direita
        else
            high = mid - 1; // Busca na esquerda
    }
    return -1;
}
```

**Eficiência Exponencial**

A cada passo, eliminamos metade dos dados restantes. Para 1 milhão de itens, precisamos de apenas ~20 comparações.

# Comparação de Algoritmos



## ANÁLISE TÉCNICA

| Característica | Busca Linear | Busca Binária |
|---|---|---|
| Complexidade (Tempo) | `O(n)` | `O(log n)` |
| Pré-requisito | Nenhum | Dados **Ordenados** |
| Complexidade (Código) | Muito Simples | Moderada (cuidado com bugs) |
| Crescimento | Linear (Proporcional) | Logarítmico (Muito lento) |

**Trade-off Fundamental**

A Busca Binária é exponencialmente mais rápida, mas exige o investimento inicial de manter os dados ordenados.

# Árvores de Busca (BST)



### A Regra de Ouro

Para qualquer nó **N**:

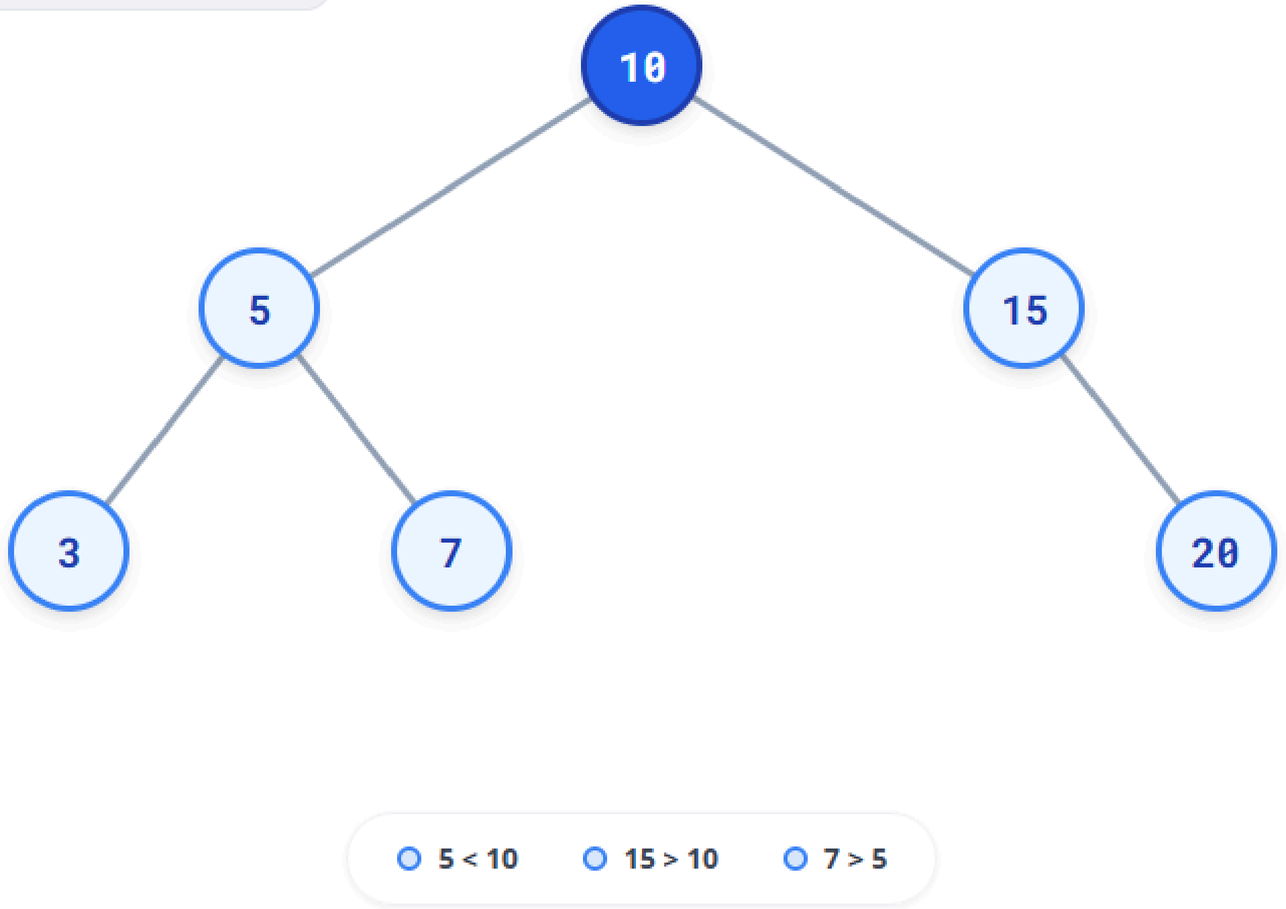
- **Esquerda:** Menores que N
- **Direita:** Maiores que N

### Como a busca funciona?

Semelhante à busca binária, a cada passo eliminamos metade da subárvore restante.

| | |
|---|---|
| MÉDIA | O(log n) |
| PIOR CASO | O(n) |

EXEMPLO: BST BALANCEADA



# Hashing (Tabelas Hash)



IDEIA FUNDAMENTAL

"Transformar a chave de busca em um endereço direto (posição) na memória."

## Performance e Aplicações

**O(1)** **Tempo Médio Constante**
Acesso direto, independente do tamanho N.

- Bancos de Dados (Indexação)
- Compiladores (Tabelas de Símbolos)
- Sistemas de Cache

# Espaço de Soluções



**ESPECTRO DE OPÇÕES**

Bentley mostra que existem muitas soluções possíveis para busca. Não existe uma "bala de prata".

## Principais Estruturas e Algoritmos

- **Busca Linear:** Simples, para pequenos conjuntos.
- **Busca Binária:** Eficiente para arrays estáticos.
- **Árvores:** BST, AVL, Red-Black (dados dinâmicos).
- **Hashing:** Acesso direto rápido O(1).
- **Especializadas:** Tries, B-trees, Índices.

CONCEITO CHAVE

## Design Space de Algoritmos

> Não se trata apenas de escolher "o mais rápido", mas de combinar a **organização de dados** correta com o **algoritmo adequado** para o contexto do problema.

Memória | Velocidade | Complexidade | Manutenção

# Trade-offs: Velocidade vs. Memória



## COMPARATIVO DE ESTRUTURAS

| Estrutura | Velocidade | Memória | Requisitos |
|---|---|---|---|
| Lista Simples | **Baixa (Lenta)** | **Pouca (Eficiente)** | Nenhum |
| Busca Binária | **Média (Rápida)** | **Pouca (Eficiente)** | Dados **Ordenados** |
| # Hash Table | **Muito Alta** | **Mais Memória** | Função Hash + Espaço Extra |

### A Lei da Troca Equivalente

Em programação, escolher algoritmos significa escolher compromissos (trade-offs). Geralmente, trocamos **memória** por **velocidade**, ou vice-versa.

### Qual escolher?

- **Pequenos dados?** Lista Simples (simplicidade vence).
- **Memória restrita?** Busca Binária (compacta e rápida).
- **Velocidade crítica?** Hash Table (performance máxima).

IDEIA PRINCIPAL

# *"Organize seus dados corretamente e o algoritmo se torna simples."*

**Jon Bentley**
PROGRAMMING PEARLS

### O algoritmo não é tudo

Escrever código complexo para dados desorganizados é ineficiente e propenso a erros.

### Estrutura define desempenho

A escolha da estrutura de dados determina quais algoritmos podem ser aplicados e sua velocidade.

# O que aprendemos com a coluna



## Lições Principais

### Problema Fundamental

A busca é uma operação onipresente e crítica na computação.

### Diversidade de Estratégias

Não existe solução única; existem várias abordagens para cada contexto.

### Estrutura vs. Performance

A escolha da estrutura de dados impacta diretamente o desempenho final.

### Complexidade é Essencial

Entender O(n) vs O(log n) distingue bons programas de ótimos programas.

# Por que essa coluna ainda é atual?

Conceitos fundamentais de 1986 que impulsionam a tecnologia de 2026.

## Aplicações Modernas

### Bancos de Dados
Otimização de índices e queries em SQL e NoSQL.

### Motores de Busca
Algoritmos de ranking e recuperação de informação (Google, Bing).

### Redes Sociais
Busca em grafos para conectar pessoas e conteúdos.

### Inteligência Artificial
Sistemas RAG (Retrieval-Augmented Generation) e busca vetorial.

## Infraestrutura Crítica

- **Sistemas de Arquivos:** Indexação rápida de milhões de arquivos.
- **Compiladores:** Tabelas de símbolos para variáveis e funções.
- **Caches:** Algoritmos de substituição e busca rápida (LRU).

CONCLUSÃO

"O problema da busca continua sendo central na computação moderna. A escala mudou, mas os princípios permanecem."

CONCLUSÃO

"Bons programas dependem de boas estruturas de dados."

Obrigado!