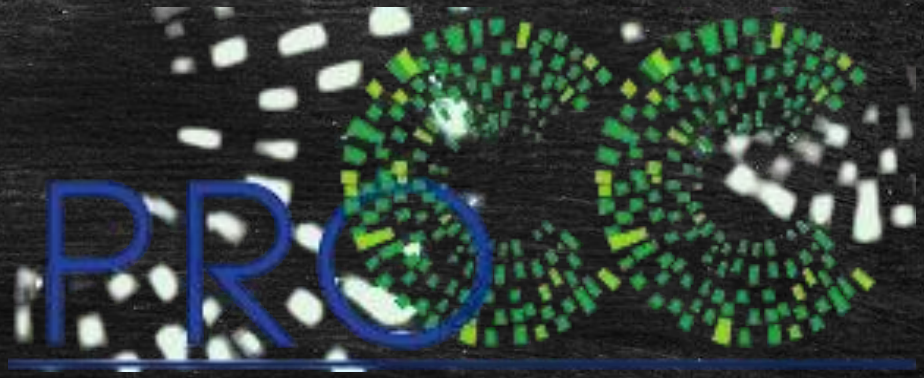


PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Atividade:

Árvore Binária de Busca Ótima - OBST

Otimização de buscas considerando probabilidades.

Professor:

Dr. Leonardo Nogueira Matos

Equipe

Francisco Farias Gomes

Silas Lopes Santos Silva Amancio do Vale

Luciano Torres Marques

Ramon Adller de Santana

Introdução

- Uma Árvore Binária de Busca (BST – Binary Search Tree) é uma estrutura de dados que organiza chaves de forma hierárquica, permitindo buscas, inserções e remoções em tempo médio de $O(\log n)$, quando balanceada.
- Porém, em cenários onde as chaves têm diferentes probabilidades de acesso, o simples uso de uma BST comum pode ser ineficiente.
- É nesse contexto que surge a Optimal Binary Search Tree (OBST): uma árvore binária de busca otimizada de acordo com as frequências de acesso às chaves, de modo a minimizar o custo esperado de busca.

Árvore Binária de Busca Ótima - OBST

- DEFINIÇÃO: Uma Árvore Binária de Busca onde o custo esperado de busca é minimizado.
- CONTEXO: Na BST, nem todas as chaves são acessadas com a mesma frequência. Por esse motivo, uma Árvore Binária de Busca – BST não atenderia, a depender da organização, uma busca com custo minimizado.
- OBJETIVO: Organizar as chaves para que as mais frequentes fiquem mais próximas da raiz.

Conceitos Fundamentais

- Uma BST comum organiza chaves em ordem, mas o custo médio de busca depende da forma da árvore.
- Se algumas chaves são mais acessadas que outras, uma árvore "mal escolhida" pode aumentar o tempo médio de busca.
- A OBST organiza as chaves em ordem e escolhe as raízes/subárvores de modo a minimizar o custo esperado de busca.

Problema

Suponha um dicionário com palavras:

- Chaves: ["amor", “carro", “zebra", “xilofone"]
- Frequências: “amor" (40%), “xilofone" (10%), “zebra" (20%), “carro" (30%)

Como organizar estas palavras na árvore para minimizar o tempo médio de busca?

Dados do Problema

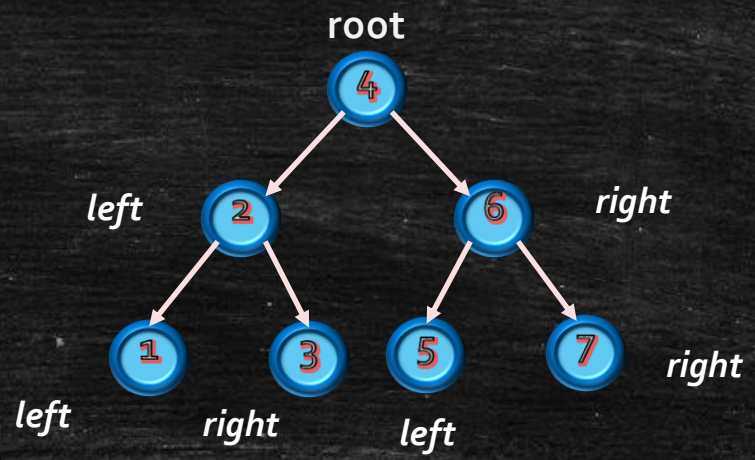
Temos 4 chaves, com suas probabilidades de acesso (frequência já somando 100%)

- Amor -> 0.40
- Carro -> 0.30
- Zebra-> 0.20
- Xilofone -> 0.10

Dados do Problema

Uma Árvore Binária de Busca (BST) organiza chaves em ordem alfabética (ou numérica) para facilitar a busca:

- Nó à esquerda é menor,
- Nó à direita é maior.



O problema: dependendo da ordem de inserção, a árvore pode ficar desequilibrada e aumentar o custo médio de busca.

Dados do Problema – Solução OBST

A OBST (Optimal Binary Search Tree) resolve isso escolhendo quem será a raiz e como organizar os filhos de forma que o custo esperado de busca seja mínimo, considerando as frequências (ou probabilidades) de acesso às chaves.

O objetivo é minimizar o número total de comparações ou o custo médio das buscas, considerando as frequências de acesso (probabilidades) de cada chave, para que as chaves mais acessadas estejam mais próximas da raiz da árvore.

Definições

Temos um conjunto de n chaves ordenadas: $k_1 < k_2 < \cdots < k_n$

- Cada chave ki possui uma probabilidade de acesso (frequência de busca): p_i, com i = 1, 2,..., n.

- Além disso, temos as probabilidades de falha de busca (elementos não encontrados): $q_0, q_1, q_2, \ldots, q_n$.

  q_i, = probabilidade de uma busca falhar entre k_i e k_{i+1}.

Definições

O peso de um subintervalo de chaves é dado por:

$$w(i,j) = \sum_{t=i+1}^{i} p_t + \sum_{t=i}^{i} q_t$$

- Onde i e j são os índices do intervalo considerado, com $0 \leq i \leq j \leq n$.
- Esse valor representa a soma das probabilidades de todas as chaves $k_{i+1}, \ldots, k_j$ e das falhas correspondentes.

Definições

O custo esperado mínimo para subárvore que contém $k_{i+1}, \ldots, k_j$ é dado pela recorrência:

$$e[i,j] = \begin{cases} q_i & \text{se j} = \text{i (subárvore vazia)} \\ \min_{r=i+1}^{j} (e[i,r-1] + e[r,j] + w(i,j)) & se\ e < j \end{cases}$$

- $e[i,j]$ = Custo esperado da subárvore contendo $k_{i+1}, \ldots, k_j$
- r = índice escolhido como raiz da subárvore.
- Custo Ótimo = $e[0,n]$

Fórmula do Custo

Se cada chave tem uma frequência (probabilidade de ser buscada), o custo esperado da árvore é:

$$C = \sum_{i=1}^{n} Profundidade(Chave_i) . Frequencia(Chave_i)$$

- *Profundidade*: distância da raiz até o nó (raiz = 1).
- *Frequência*: probabilidade de busca dessa chave

Ordens das chaves

- As chaves devem estar em ordem alfabética para respeitar a propriedade da BST:
  - ✓Amor (40%)
  - ✓Carro (30%)
  - ✓Xilofone (10%)
  - ✓Zebra (20%)

Pelo critério de OBST, a raiz Carro ou Amor tende a ser melhor, pois concentra mais peso no topo.

Exemplo

Amor, Carro, Zebra, Xilofone com as respectivas probabilidades: $p_1 = 0,40, p_2 = 0.30, p_3 = 0.20, p_4 = 0.10$.

Para simplificar este exemplo vamos assumir nenhuma probabilidade de busca malsucedida (ou seja, $q_1 = q_2 = q_3 = q_4 = 0$).

Exemplo - Dados

- Chaves (ordenadas):

$$k_1 = Amor, k_2 = Carro, k_3 = Zebra, k_4 = Xilofone$$

Probabilidade: $p = [0, 40, 0, 30, 0, 20, 0, 10]$

Falhas: $q = [0, 0, 0, 0, 0]$

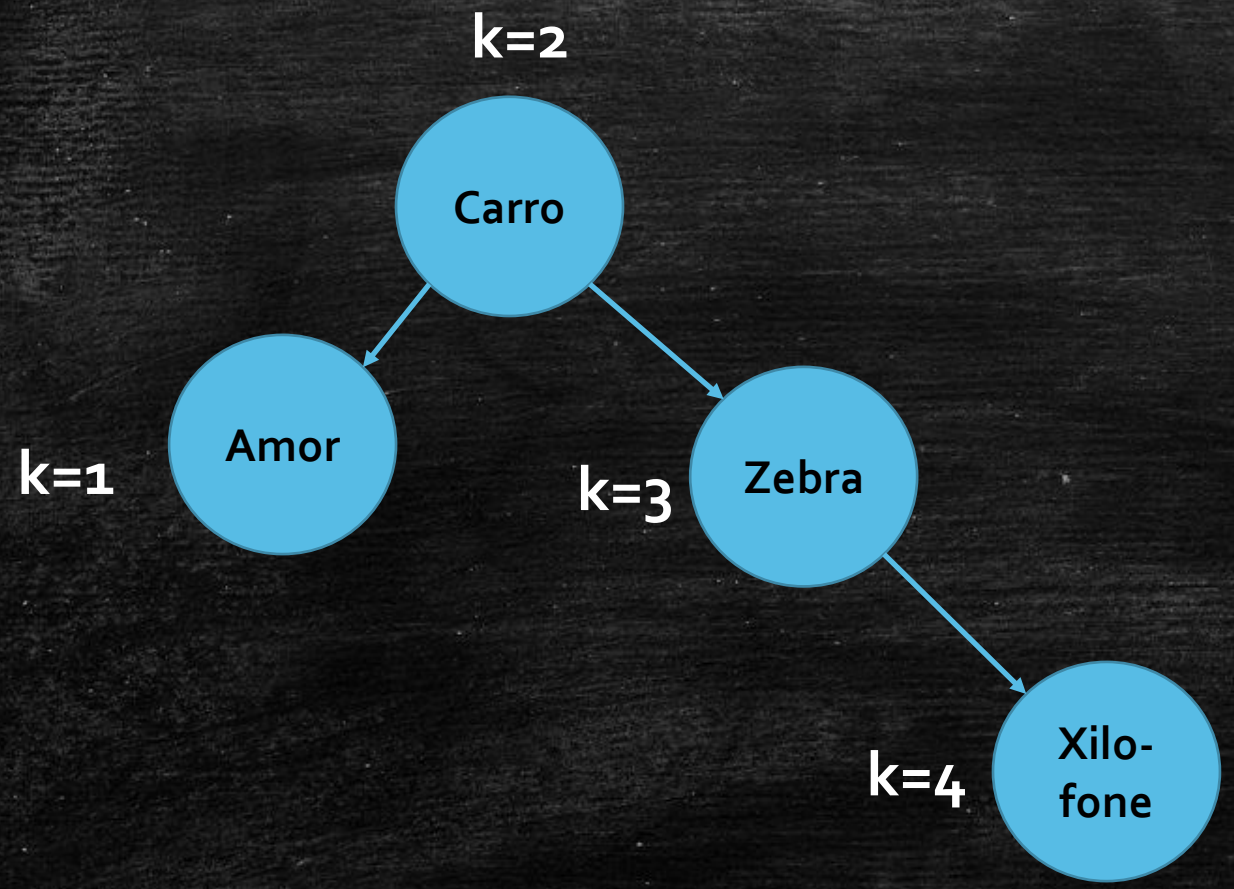
Exemplo – Fórmulas Usadas

- Peso: $w(i,j) = \sum_{t=i+1}^{i} p_t + \sum_{t=i}^{i} q_t$
- Caso base: $e[i,j] = q_i$
- Recorrência:

$$e[i,j] = \left\{ min_{r=i+1}^{j} (e[i,r-1] + e[r,j] + w(i,j)) \right.$$

- Custo ótimo total: $e[0,n]$ onde n é o total de chaves.

Opção A: Raiz = Carro (0.30)



Profundidade = 1

Profundidade = 2

Profundidade = 3

Opção A: Raiz = Carro (0.30)

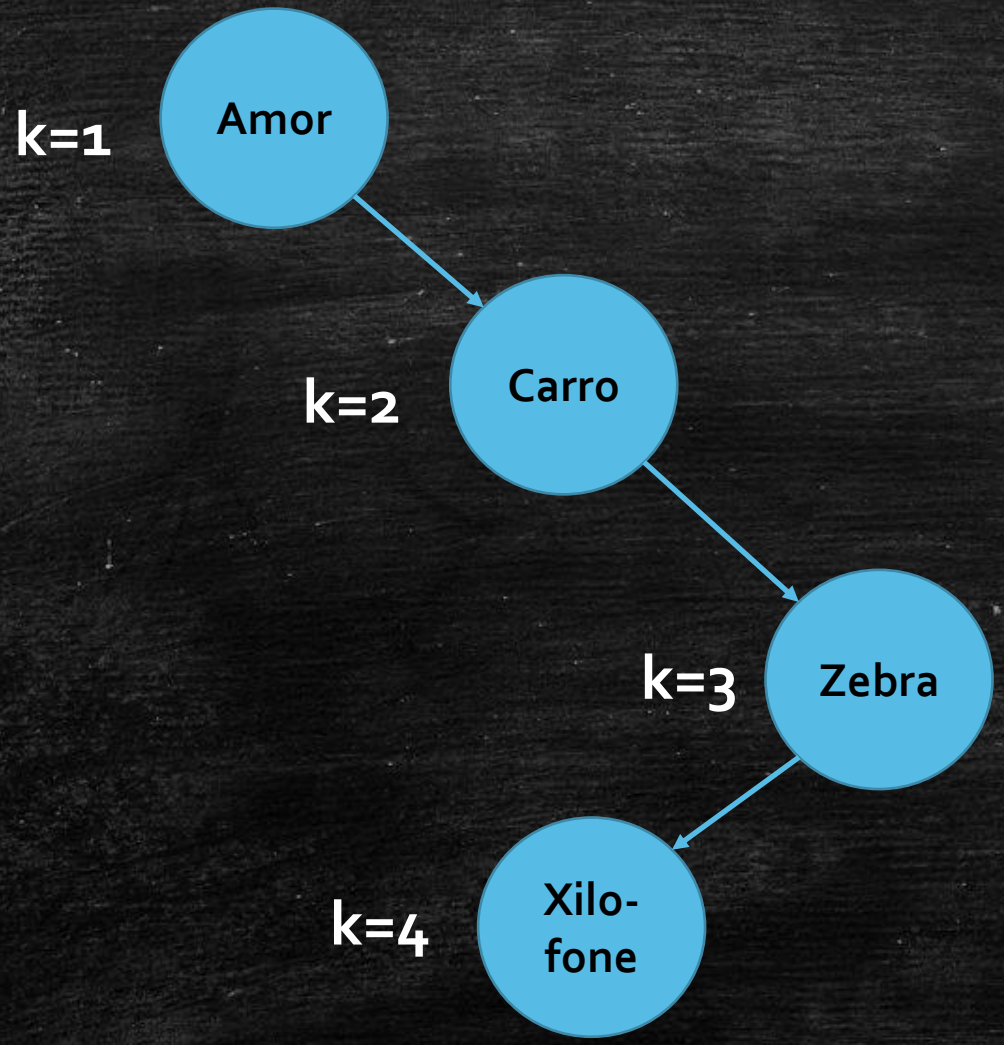
- Carro (k_2): $d_2 = 1$
- Amor (k_1): $d_1 = 2$
- Zebra (k_3): $d_3 = 2$
- Xilofone (k_4): $d_4 = 3$
- Custo esperado: $\sum_{i=1}^{4} p_i \cdot d_i$

d é a profundidade

$$= 0{,}40 \cdot 2 + 0{,}30 \cdot 1 + 0{,}20 \cdot 2 + 0{,}10 \cdot 3$$
$$= 0{,}80 + 0{,}30 + 0{,}40 + 0{,}30 = 1{,}80$$

Confere com $e[0, 4] = 1{,}80$.

Opção B: Raiz = Amor (0.40)



Profundidade = 1

Profundidade = 2

Profundidade = 3

Profundidade = 4

Opção A: Raiz = Carro (0.30)

- Carro (k_2): $d_2 = 2$
- Amor (k_1): $d_1 = 1$
- Zebra (k_3): $d_3 = 3$
- Xilofone (k_4): $d_4 = 4$
- Custo esperado: $\sum_{i=1}^{4} p_i \,.\, d_i$

$$= 0{,}40 \cdot 1 + 0{,}30 \cdot 2 + 0{,}20 \cdot 3 + 0{,}10 \cdot 4$$
$$= 0{,}40 + 0{,}60 + 0{,}60 + 0{,}40 = 2{,}00$$

Confere com $e[0, 4] = 2{,}00$.

Comparação de Custos

Raiz Amor: Custo total = $0.40 + 0.60 + 0.60 + 0.40 = 2.00$

Raiz Carro: Custo total = $0.30 + 0.80 + 0.40 + 0.30 = 1.80$

CONCLUSÃO: O menor custo esperado é: 1.80 obtido quando CARRO é a raiz da árvore.

Algoritmo OBST - Python

```python
# ============================================================
# ÁRVORE BINÁRIA DE BUSCA ÓTIMA (Optimal Binary Search Tree - OBST)
# Implementação baseada no livro de Cormen
# ============================================================

# Chaves em ordem alfabética:
# Amor < Carro < Xilofone < Zebra
keys = ["Amor", "Carro", "Xilofone", "Zebra"]

# Probabilidades de acesso (sucesso na busca de cada chave)
p = [0.40, 0.30, 0.10, 0.20]
n = len(keys)
```

Algoritmo OBST - Python

```python
# ============================================================
# Passo 1: Inicializar matrizes
# ============================================================
e = [[0.0 for _ in range(n+2)] for _ in range(n+2)]
w = [[0.0 for _ in range(n+2)] for _ in range(n+2)]
root = [[0   for _ in range(n+2)] for _ in range(n+2)]

# ============================================================
# Passo 2: Casos base
# ============================================================
for i in range(1, n+1):
    e[i][i] = p[i-1]
    w[i][i] = p[i-1]
    root[i][i] = i
```

Programação Dinâmica – Caso de falha na busca

Algoritmo OBST - Python

```python
# ============================================================
# Passo 3: Preencher tabelas
# ============================================================
for l in range(2, n+1):              # l = tamanho do subintervalo
    for i in range(1, n-l+2):        # i = início do intervalo
        j = i + l - 1                # j = fim do intervalo
        e[i][j] = float("inf")       # inicializa com infinito
        w[i][j] = w[i][j-1] + p[j-1]  # soma dos pesos até j

        for k in range(i, j+1):      # testamos cada chave como raiz
            cost = ( (e[i][k-1] if k > i else 0) +
                     (e[k+1][j] if k < j else 0) +
                     w[i][j] )
            if cost < e[i][j]:       # guardamos o menor custo
                e[i][j] = cost
                root[i][j] = k       # registramos quem foi a raiz
```

Algoritmo OBST - Python

```python
# ================================================================
# Passo 4: Reconstrução da árvore com níveis e custo individual
# ================================================================
def build_tree(i, j, nivel=1, parent=None, side=None):
    if i > j:
        return
    r = root[i][j]                  # raiz ótima desse intervalo
    node = keys[r-1]                # nome da chave
    prob = p[r-1]                   # sua probabilidade
    custo_individual = round(prob * nivel, 2)  # custo local = p * profundidade

    if parent is None:
        print(f"Raiz: {node} (nível {nivel}, custo {custo_individual})")
    else:
        print(f"{side} de {parent}: {node} (nível {nivel}, custo {custo_individual})")

    # recursivamente constrói as subárvores esquerda e direita
    build_tree(i, r-1, nivel+1, node, "esquerda")
    build_tree(r+1, j, nivel+1, node, "direita")
```

Algoritmo OBST - Python

```python
# ============================================================
# Passo 5: Exibir resultados
# ============================================================
print("Matriz de custos e[i][j]:")
for row in e[1:n+1]:
    print([round(x,2) if x != float("inf") else "∞" for x in row[1:n+1]])

print("\nMatriz de raízes root[i][j]:")
for row in root[1:n+1]:
    print(row[1:n+1])

print("\nÁrvore ótima:")
build_tree(1, n)

print("\nCusto esperado ótimo =", round(e[1][n], 2))
```

Algoritmo OBST - Python

```
Matriz de custos e[i][j]:
[0.4, 1.0, 1.3, 1.8]
[0.0, 0.3, 0.5, 1.0]
[0.0, 0.0, 0.1, 0.4]
[0.0, 0.0, 0.0, 0.2]

Matriz de raízes root[i][j]:
[1, 1, 1, 2]
[0, 2, 2, 2]
[0, 0, 3, 4]
[0, 0, 0, 4]

Árvore ótima:
Raiz: Carro (nível 1, custo 0.3)
esquerda de Carro: Amor (nível 2, custo 0.8)
direita de Carro: Zebra (nível 2, custo 0.4)
esquerda de Zebra: Xilofone (nível 3, custo 0.3)

Custo esperado ótimo = 1.8
```

O objetivo da OBST

Organizar os dados de forma a minimizar o número total de comparações ou o custo médio das buscas, considerando as frequências de acesso (probabilidades) de cada chave, para que as chaves mais acessadas estejam mais próximas da raiz da árvore.

As principais áreas e situações de aplicação

- Otimização de Buscas em Estruturas Estáticas.
- Compiladores e Análise Léxicas/Sintática.
- Dicionários Digitais/Autocomplete.
- Compressão de dados (Similar a Huffman). *Símbolos com maior frequência recebem códigos mais curtos, e os menos frequentes recebem códigos mais longos*.

Conceito Chave

- A ideia central é colocar as chaves com maior probabilidade de acesso o mais próximo possível da raiz da árvore.
- Enquanto árvores de busca binária balanceadas (como AVL ou Rubro-Negra), que buscam o equilíbrio perfeito de altura, a OBST busca o equilíbrio de custos ponderados para alcançar o desempenho de busca mais eficiente possível.

Conclusão

Principais características:

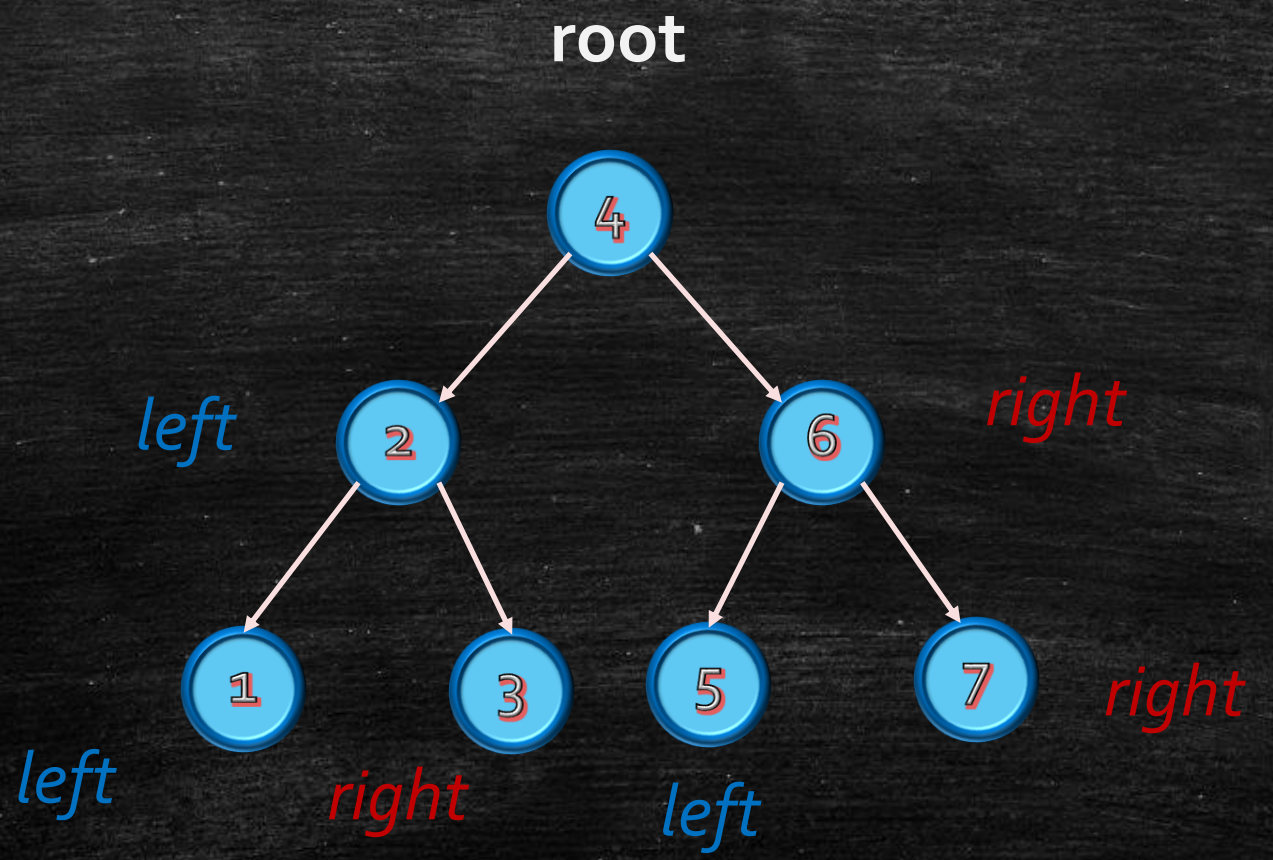
- ✓ Minimiza o custo médio de busca, considerando probabilidades de acesso.
- ✓ Estrutura eficiente para cenários com buscas desbalanceadas.
- ✓ Baseada em programação dinâmica para determinar a disposição ótima dos nós.

Conclusão

Pontos negativos:

- ✓ Construção complexa e com alto custo computacional $O(n^3)$ no pior caso.
- ✓ Requer conhecimento prévio das probabilidades de acesso.
- ✓ Pouco prática em ambientes dinâmicos com inserções e remoções frequentes.

Obrigado



Árvore Binária de Busca Ótima - OBST

Otimização de buscas considerando probabilidades.