Algoritmos e Estruturas de Dados

Árvores Binárias - Guia Completo

Sumário

---

1. Conceitos Fundamentais de Árvores

1.1 Definições Básicas

Árvore: Estrutura de dados hierárquica composta por nós conectados por arestas, sem ciclos.

Terminologia:

- **Raiz:** Nó superior da árvore (sem pai)
- **Nó:** Elemento que contém dado e referências para filhos
- **Folha:** Nó sem filhos
- **Nó Interno:** Nó com pelo menos um filho
- **Pai:** Nó que possui filhos
- **Filho:** Nó descendente direto de outro
- **Irmãos:** Nós com mesmo pai
- **Ancestral:** Qualquer nó no caminho da raiz até o nó
- **Descendente:** Qualquer nó alcançável seguindo filhos

- **Subárvore:** Árvore formada por um nó e todos seus descendentes

Propriedades Importantes:

- **Altura do Nó:** Comprimento do caminho mais longo até uma folha
- **Altura da Árvore:** Altura da raiz
- **Profundidade/Nível:** Distância do nó até a raiz
- **Grau do Nó:** Número de filhos
- **Grau da Árvore:** Maior grau entre todos os nós

1.2 Árvore Binária

Definição: Árvore onde cada nó tem no máximo 2 filhos (esquerdo e direito).

```
Estrutura NóBinário:
    chave: TipoChave
    esquerda: ponteiro para NóBinário
    direita: ponteiro para NóBinário
    (opcional) pai: ponteiro para NóBinário
```

Tipos Especiais:

- **Árvore Binária Cheia:** Todos os níveis completamente preenchidos
- **Árvore Binária Completa:** Todos os níveis cheios exceto possivelmente o último (preenchido da esquerda)
- **Árvore Binária Perfeita:** Todas as folhas no mesmo nível
- **Árvore Degenerada:** Cada nó tem apenas um filho (equivale a lista)

Propriedades Matemáticas:

- Número máximo de nós no nível i: 2^i
- Número máximo de nós em árvore de altura h: $2^{(h+1)} - 1$
- Altura mínima para n nós: $\lceil \log_2(n+1) \rceil - 1$
- Árvore binária completa com n nós tem altura $O(\log n)$

1.3 Percursos em Árvores Binárias

1.3.1 Percurso em Pré-Ordem (Preorder)

Visita: Raiz → Esquerda → Direita

```
PreOrdem(raiz):
    se raiz ≠ NULL então:
        visitar(raiz)
        PreOrdem(raiz.esquerda)
        PreOrdem(raiz.direita)
```

Uso: Copiar árvore, obter expressão prefixada

1.3.2 Percurso em Ordem (Inorder)

Visita: Esquerda → Raiz → Direita

```
EmOrdem(raiz):
    se raiz ≠ NULL então:
        EmOrdem(raiz.esquerda)
        visitar(raiz)
        EmOrdem(raiz.direita)
```

Uso: Obter elementos ordenados em BST

1.3.3 Percurso em Pós-Ordem (Postorder)

Visita: Esquerda → Direita → Raiz

```
PosOrdem(raiz):
    se raiz ≠ NULL então:
        PosOrdem(raiz.esquerda)
        PosOrdem(raiz.direita)
        visitar(raiz)
```

Uso: Liberar memória, obter expressão posfixada

1.3.4 Percurso em Largura (Level Order)

Visita nível por nível, da esquerda para direita

```
EmLargura(raiz):
    fila ← nova fila
    fila.enfileirar(raiz)

    enquanto não fila.vazia():
        no ← fila.desenfileirar()
        visitar(no)

        se no.esquerda ≠ NULL:
            fila.enfileirar(no.esquerda)
        se no.direita ≠ NULL:
            fila.enfileirar(no.direita)
```

Complexidade dos percursos: O(n) tempo, O(h) espaço na pilha de recursão

---

2. Árvores Binárias de Busca (BST)

2.1 Propriedade Fundamental

Para todo nó x em uma BST:

- Todos os nós na subárvore esquerda têm chave < x.chave
- Todos os nós na subárvore direita têm chave > x.chave
- Ambas subárvores também são BSTs

2.2 Operações Básicas

2.2.1 Busca

```
Buscar(raiz, chave):
    se raiz = NULL ou raiz.chave = chave:
        retornar raiz

    se chave < raiz.chave:
        retornar Buscar(raiz.esquerda, chave)
    senão:
        retornar Buscar(raiz.direita, chave)
```

Complexidade:

- Melhor caso: O(1) - elemento é a raiz
- Pior caso: O(h) onde h é altura

- Árvore balanceada: O(log n)
- Árvore degenerada: O(n)

2.2.2 Inserção

```
Inserir(raiz, chave):
    se raiz = NULL:
        retornar novo nó com chave

    se chave < raiz.chave:
        raiz.esquerda ← Inserir(raiz.esquerda, chave)
    senão se chave > raiz.chave:
        raiz.direita ← Inserir(raiz.direita, chave)

    retornar raiz
```

Complexidade: O(h)

2.2.3 Remoção

Três casos:

1. **Nó folha:** Simplesmente remover
2. **Nó com 1 filho:** Substituir pelo filho
3. **Nó com 2 filhos:** Substituir pelo sucessor (menor da direita) ou predecessor (maior da esquerda)

```
Remover(raiz, chave):
    se raiz = NULL:
        retornar NULL

    se chave < raiz.chave:
        raiz.esquerda ← Remover(raiz.esquerda, chave)
    senão se chave > raiz.chave:
        raiz.direita ← Remover(raiz.direita, chave)
    senão:
        // Caso 1 e 2: 0 ou 1 filho
        se raiz.esquerda = NULL:
            retornar raiz.direita
        senão se raiz.direita = NULL:
            retornar raiz.esquerda

        // Caso 3: 2 filhos
        sucessor ← Minimo(raiz.direita)
        raiz.chave ← sucessor.chave
        raiz.direita ← Remover(raiz.direita, sucessor.chave)

    retornar raiz

Minimo(raiz):
    enquanto raiz.esquerda ≠ NULL:
        raiz ← raiz.esquerda
    retornar raiz
```

Complexidade: O(h)

2.2.4 Operações Auxiliares

Mínimo e Máximo:

```
Minimo(raiz):
    enquanto raiz.esquerda ≠ NULL:
        raiz ← raiz.esquerda
    retornar raiz

Maximo(raiz):
    enquanto raiz.direita ≠ NULL:
        raiz ← raiz.direita
    retornar raiz
```

Complexidade: O(h)

2.3 Vantagens e Desvantagens

Vantagens:

- Busca, inserção e remoção eficientes (quando balanceada)
- Operações de mínimo, máximo, sucessor, predecessor
- Percurso em ordem fornece elementos ordenados
- Simples de implementar

Desvantagens:

- Desempenho depende da altura
- Pode degenerar para O(n) se inserções ordenadas
- Não garante balanceamento

---

3. Árvores AVL

3.1 Conceito e Motivação

Árvore AVL: BST auto-balanceada onde, para todo nó, a diferença de altura entre subárvores esquerda e direita é no máximo 1.

Fator de Balanceamento (FB):

```
FB(nó) = altura(subárvore_esquerda) - altura(subárvore_direita)
```

- FB ∈ {-1, 0, 1} para todos os nós em AVL
- |FB| > 1 indica desbalanceamento

Altura garantida: O(log n) para n nós

3.2 Rotações

Operações para restaurar balanceamento após inserção/remoção.

3.2.1 Rotação Simples à Direita (LL)

Usado quando FB(nó) = 2 e FB(filho_esquerdo) ≥ 0

```
RotacaoDireita(y):
    x ← y.esquerda
    T2 ← x.direita

    x.direita ← y
    y.esquerda ← T2

    atualizar_altura(y)
    atualizar_altura(x)

    retornar x
```

```
    y                 x
   / \               / \
  x   T3   -->    T1    y
 / \                   / \
T1 T2                 T2  T3
```

3.2.2 Rotação Simples à Esquerda (RR)

Usado quando FB(nó) = -2 e FB(filho_direito) ≤ 0

```
RotacaoEsquerda(x):
    y ← x.direita
    T2 ← y.esquerda

    y.esquerda ← x
    x.direita ← T2

    atualizar_altura(x)
    atualizar_altura(y)

    retornar y
```

3.2.3 Rotação Dupla Esquerda-Direita (LR)

Usado quando FB(nó) = 2 e FB(filho_esquerdo) < 0

```
RotacaoEsquerdaDireita(z):
    z.esquerda ← RotacaoEsquerda(z.esquerda)
    retornar RotacaoDireita(z)
```

3.2.4 Rotação Dupla Direita-Esquerda (RL)

Usado quando FB(nó) = -2 e FB(filho_direito) > 0

```
RotacaoDireitaEsquerda(z):
    z.direita ← RotacaoDireita(z.direita)
    retornar RotacaoEsquerda(z)
```

3.3 Inserção em AVL

```
InserirAVL(raiz, chave):
    // 1. Inserção normal de BST
    se raiz = NULL:
        retornar novo nó com chave

    se chave < raiz.chave:
        raiz.esquerda ← InserirAVL(raiz.esquerda, chave)
    senão se chave > raiz.chave:
        raiz.direita ← InserirAVL(raiz.direita, chave)
    senão:
        retornar raiz  // Duplicata

    // 2. Atualizar altura
    raiz.altura ← 1 + max(altura(raiz.esquerda), altura(raiz.direita))

    // 3. Calcular fator de balanceamento
    fb ← obter_fb(raiz)

    // 4. Balancear se necessário

    // Caso LL
    se fb > 1 e chave < raiz.esquerda.chave:
        retornar RotacaoDireita(raiz)

    // Caso RR
    se fb < -1 e chave > raiz.direita.chave:
        retornar RotacaoEsquerda(raiz)

    // Caso LR
    se fb > 1 e chave > raiz.esquerda.chave:
        raiz.esquerda ← RotacaoEsquerda(raiz.esquerda)
        retornar RotacaoDireita(raiz)

    // Caso RL
    se fb < -1 e chave < raiz.direita.chave:
        raiz.direita ← RotacaoDireita(raiz.direita)
        retornar RotacaoEsquerda(raiz)
```

```
    retornar raiz
```

3.4 Complexidade

Operação	Complexidade
Busca	O(log n)
Inserção	O(log n)
Remoção	O(log n)
Mínimo/Máximo	O(log n)
Espaço	O(n)

Rotações: No máximo 2 rotações por inserção, O(log n) por remoção

3.5 Vantagens e Desvantagens

Vantagens:

- Altura garantida O(log n)
- Todas operações garantidas O(log n)
- Melhor para aplicações com muitas buscas

Desvantagens:

- Mais rotações que outras árvores balanceadas
- Overhead de armazenar altura em cada nó
- Rebalanceamento frequente em inserções/remoções

---

4. Árvores Rubro-Negras

4.1 Propriedades

Árvore binária de busca com coloração (vermelho/preto) que satisfaz:

1. **Todo nó é vermelho ou preto**
2. **Raiz é preta**
3. **Todas as folhas (NULL) são pretas**
4. **Nó vermelho tem filhos pretos** (sem dois vermelhos consecutivos)
5. **Todos caminhos de um nó até folhas descendentes têm mesmo número de nós pretos**

```
Estrutura NóRN:
    chave: TipoChave
    cor: {VERMELHO, PRETO}
    esquerda: ponteiro para NóRN
    direita: ponteiro para NóRN
    pai: ponteiro para NóRN
```

Altura Negra: Número de nós pretos em qualquer caminho até folha (excluindo o nó inicial)

4.2 Teorema Importante

Uma árvore rubro-negra com n nós internos tem altura no máximo $2 \cdot \log_2(n+1)$.

Prova: Subárvore com raiz x tem pelo menos 2^(hn(x)) - 1 nós internos, onde hn(x) é altura negra.

4.3 Rotações e Recolorações

Similar a AVL, mas usa cores para determinar rotações:

4.3.1 Rotação à Esquerda

```
RotacaoEsquerda(T, x):
    y ← x.direita
    x.direita ← y.esquerda

    se y.esquerda ≠ NULL:
        y.esquerda.pai ← x

    y.pai ← x.pai

    se x.pai = NULL:
        T.raiz ← y
    senão se x = x.pai.esquerda:
        x.pai.esquerda ← y
    senão:
        x.pai.direita ← y

    y.esquerda ← x
    x.pai ← y
```

4.4 Inserção em Árvore Rubro-Negra

Estratégia:

1. Inserir como BST normal
2. Colorir novo nó de VERMELHO
3. Corrigir violações de propriedades

```
InserirRN(T, z):
    // 1. Inserção BST normal
    y ← NULL
    x ← T.raiz

    enquanto x ≠ NULL:
        y ← x
        se z.chave < x.chave:
            x ← x.esquerda
        senão:
            x ← x.direita

    z.pai ← y

    se y = NULL:
        T.raiz ← z
    senão se z.chave < y.chave:
        y.esquerda ← z
    senão:
        y.direita ← z

    z.esquerda ← NULL
    z.direita ← NULL
    z.cor ← VERMELHO

    // 2. Corrigir violações
    CorrigirInsercao(T, z)
```

Correção de Inserção

Casos (quando pai é vermelho):

Caso 1: Tio é vermelho

- Recolorir pai, tio e avô
- Continuar no avô

Caso 2: Tio é preto, nó é filho direito (triangular)

- Rotação à esquerda no pai

- Transformar no caso 3

Caso 3: Tio é preto, nó é filho esquerdo (linear)

- Rotação à direita no avô
- Recolorir pai e avô

```
CorrigirInsercao(T, z):
    enquanto z.pai.cor = VERMELHO:
        se z.pai = z.pai.pai.esquerda:
            tio ← z.pai.pai.direita

            se tio.cor = VERMELHO:  // Caso 1
                z.pai.cor ← PRETO
                tio.cor ← PRETO
                z.pai.pai.cor ← VERMELHO
                z ← z.pai.pai

            senão:
                se z = z.pai.direita:  // Caso 2
                    z ← z.pai
                    RotacaoEsquerda(T, z)

                // Caso 3
                z.pai.cor ← PRETO
                z.pai.pai.cor ← VERMELHO
                RotacaoDireita(T, z.pai.pai)

        senão:  // Simétrico
            // ...

    T.raiz.cor ← PRETO
```

4.5 Complexidade

Operação	**Complexidade**
Busca	O(log n)
Inserção	O(log n)
Remoção	O(log n)
Espaço	O(n)

Rotações: No máximo 2 rotações por inserção, 3 por remoção

4.6 Comparação AVL vs Rubro-Negra

Aspecto	AVL	Rubro-Negra
Balanceamento	Mais rígido	Mais relaxado
Altura	$\leq 1.44 \cdot \log_2(n)$	$\leq 2 \cdot \log_2(n+1)$
Rotações (inserção)	Até 2	Até 2
Rotações (remoção)	O(log n)	Até 3
Busca	Ligeiramente mais rápida	Ligeiramente mais lenta
Inserção/Remoção	Mais lentas	Mais rápidas
Uso	Muitas buscas	Muitas modificações
Espaço extra	Altura	1 bit (cor)

---

5. Árvores B

5.1 Definição e Motivação

Árvore B de ordem m: Árvore balanceada multi-caminho otimizada para sistemas com acesso a disco.

Propriedades:

1. Todo nó tem no máximo m filhos
2. Todo nó interno (exceto raiz) tem pelo menos ⌈m/2⌉ filhos
3. Raiz tem pelo menos 2 filhos (se não é folha)
4. Todas as folhas estão no mesmo nível
5. Nó com k filhos contém k-1 chaves

```
Estrutura NóB:
    n: inteiro  // número de chaves
    chaves: array[m-1] de TipoChave
    filhos: array[m] de ponteiro para NóB
    folha: booleano
```

Invariantes:

- Chaves em cada nó estão ordenadas
- Para chave k no nó: filhos à esquerda < k < filhos à direita

5.2 Por Que Árvores B?

Problema: Acessar disco é 100.000× mais lento que memória

Solução: Minimizar número de acessos ao disco

- Nós grandes (múltiplas chaves)
- Árvore baixa (altura pequena)
- Um nó = um bloco de disco

Exemplo: Árvore B de ordem 1001:

- 1 milhão de chaves → altura ≤ 2
- 1 bilhão de chaves → altura ≤ 3

5.3 Busca em Árvore B

```
BuscarB(x, k):
    i ← 0

    // Busca binária ou sequencial no nó
    enquanto i < x.n e k > x.chaves[i]:
        i ← i + 1

    se i < x.n e k = x.chaves[i]:
        retornar (x, i)  // Encontrado

    se x.folha:
        retornar NULL  // Não encontrado

    senão:
        ler_disco(x.filhos[i])
        retornar BuscarB(x.filhos[i], k)
```

Complexidade:

- Acessos ao disco: $O(\log_m n)$
- Comparações por nó: $O(\log m)$ ou $O(m)$
- Total: $O(\log_m n)$ acessos, $O(m \log_m n)$ comparações

5.4 Inserção em Árvore B

Estratégia: Inserir sempre em folha, dividir nós cheios

Divisão de Nó

```
DividirFilho(x, i):
    // x.filhos[i] está cheio (2m-1 chaves)
    z ← novo nó
    y ← x.filhos[i]

    z.folha ← y.folha
    z.n ← m - 1

    // Copiar m-1 maiores chaves para z
    para j de 0 até m-2:
        z.chaves[j] ← y.chaves[j+m]

    se não y.folha:
        para j de 0 até m-1:
            z.filhos[j] ← y.filhos[j+m]

    y.n ← m - 1

    // Inserir chave do meio em x
    para j de x.n até i+1 (decrescente):
        x.filhos[j+1] ← x.filhos[j]

    x.filhos[i+1] ← z

    para j de x.n-1 até i (decrescente):
        x.chaves[j+1] ← x.chaves[j]

    x.chaves[i] ← y.chaves[m-1]
    x.n ← x.n + 1
```

Inserção Principal

```
InserirB(T, k):
    r ← T.raiz

    se r.n = 2m-1:  // Raiz cheia
        s ← novo nó
        T.raiz ← s
        s.folha ← FALSO
        s.n ← 0
        s.filhos[0] ← r
        DividirFilho(s, 0)
        InserirNaoCheio(s, k)
    senão:
        InserirNaoCheio(r, k)

InserirNaoCheio(x, k):
    i ← x.n - 1

    se x.folha:
        // Inserir diretamente
        enquanto i ≥ 0 e k < x.chaves[i]:
            x.chaves[i+1] ← x.chaves[i]
            i ← i - 1
        x.chaves[i+1] ← k
        x.n ← x.n + 1

    senão:
        enquanto i ≥ 0 e k < x.chaves[i]:
            i ← i - 1
        i ← i + 1

        se x.filhos[i].n = 2m-1:
            DividirFilho(x, i)
            se k > x.chaves[i]:
                i ← i + 1

        InserirNaoCheio(x.filhos[i], k)
```

5.5 Remoção em Árvore B

Casos complexos: Fusão de nós, redistribuição de chaves

Casos principais:

1. **Chave em folha:** Remover diretamente
2. **Chave em nó interno:**

- Substituir por predecessor/sucessor
- Remover recursivamente

Garantir mínimo de chaves:

- Se filho terá < m-1 chaves após remoção:
  - Emprestar de irmão (se possível)
  - Ou fundir com irmão

5.6 Complexidade

Operação	Acessos Disco	Tempo CPU
Busca	$O(\log_m n)$	$O(m \log_m n)$
Inserção	$O(\log_m n)$	$O(m \log_m n)$
Remoção	$O(\log_m n)$	$O(m \log_m n)$

5.7 Aplicações

- **Sistemas de arquivos**: ext4, NTFS, HFS+
- **Bancos de dados**: Índices em MySQL, PostgreSQL, SQLite
- **Armazenamento de grande volume**: Minimizar I/O

Variações:

- **B+ Árvore**: Chaves apenas em folhas, folhas encadeadas (melhor para varreduras)
- *B Árvore:** Atrasa divisões, maior fator de preenchimento

---

6. Heaps (Árvores de Heap)

6.1 Definição

Heap: Árvore binária completa que satisfaz propriedade de heap.

Propriedade de Max-Heap:

- Para todo nó i (exceto raiz): A[pai(i)] ≥ A[i]
- Raiz contém máximo elemento

Propriedade de Min-Heap:

- Para todo nó i (exceto raiz): A[pai(i)] ≤ A[i]

- Raiz contém mínimo elemento

6.2 Representação em Array

Árvore binária completa pode ser armazenada eficientemente em array:

```
Para nó no índice i:
    pai(i) = ⌊(i-1)/2⌋
    filho_esquerdo(i) = 2i + 1
    filho_direito(i) = 2i + 2
```

Exemplo (Max-Heap):

```
Array: [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]

        16
       /  \
     14    10
    / \   / \
   8  7  9  3
  / \
 2  4
```

6.3 Operações Fundamentais

6.3.1 Heapify (Ajustar para Baixo)

```
MaxHeapify(A, i, tamanho):
    esq ← 2i + 1
    dir ← 2i + 2
    maior ← i

    se esq < tamanho e A[esq] > A[maior]:
        maior ← esq

    se dir < tamanho e A[dir] > A[maior]:
        maior ← dir

    se maior ≠ i:
        trocar(A[i], A[maior])
        MaxHeapify(A, maior, tamanho)
```

Complexidade: O(log n) - altura da árvore

6.3.2 Construir Heap

```
ConstruirMaxHeap(A, n):
    para i de ⌊n/2⌋ - 1 até 0 (decrescente):
        MaxHeapify(A, i, n)
```

Complexidade: O(n) - não O(n log n)!

Análise: Maioria dos nós está perto das folhas:

- n/2 folhas: 0 operações
- n/4 nós a 1 nível das folhas: 1 operação
- n/8 nós a 2 níveis: 2 operações
- Soma: n(1/4 + 2/8 + 3/16 + ...) = O(n)

6.3.3 Inserção

```
InserirHeap(A, chave, tamanho):
    tamanho ← tamanho + 1
    i ← tamanho - 1
    A[i] ← chave

    // Ajustar para cima
    enquanto i > 0 e A[pai(i)] < A[i]:
        trocar(A[i], A[pai(i)])
        i ← pai(i)
```

Complexidade: O(log n)

6.3.4 Extrair Máximo/Mínimo

```
ExtrairMaximo(A, tamanho):
    se tamanho < 1:
        retornar ERRO

    max ← A[0]
    A[0] ← A[tamanho-1]
    tamanho ← tamanho - 1
    MaxHeapify(A, 0, tamanho)

    retornar max
```

Complexidade: O(log n)

6.3.5 Aumentar/Diminuir Chave

```
AumentarChave(A, i, nova_chave):
    se nova_chave < A[i]:
        retornar ERRO

    A[i] ← nova_chave

    enquanto i > 0 e A[pai(i)] < A[i]:
        trocar(A[i], A[pai(i)])
        i ← pai(i)
```

Complexidade: O(log n)

6.4 HeapSort

Algoritmo de ordenação usando heap:

```
HeapSort(A, n):
    // 1. Construir max-heap
    ConstruirMaxHeap(A, n)

    // 2. Extrair elementos em ordem decrescente
    para i de n-1 até 1 (decrescente):
        trocar(A[0], A[i])
        MaxHeapify(A, 0, i)
```

Complexidade:

- Construir heap: O(n)
- n-1 extrações: O(n log n)
- **Total: O(n log n)** no pior, médio e melhor caso
- **Espaço: O(1)** - in-place

Vantagens:

- Garantia de O(n log n)
- In-place (não requer memória extra)

Desvantagens:

- Não é estável
- Constantes maiores que QuickSort
- Pobre localidade de cache

6.5 Fila de Prioridade

Heap é a estrutura ideal para implementar filas de prioridade:

Operação	**Complexidade**
Inserir	O(log n)
Obter Máximo/Mínimo	O(1)
Extrair Máximo/Mínimo	O(log n)
Aumentar/Diminuir Chave	O(log n)
Construir	O(n)

Aplicações:

- Algoritmo de Dijkstra (caminhos mínimos)
- Algoritmo de Prim (árvore geradora mínima)
- Agendamento de tarefas
- Simulações de eventos
- Compressão de dados (Huffman)

6.6 Heap Binomial e Heap de Fibonacci

Heaps mais avançados com operações especiais:

Heap Binomial:

- União: O(log n)
- Inserção: O(log n)
- Extrair mínimo: O(log n)

Heap de Fibonacci:

- Inserção: O(1) amortizado
- União: O(1)
- Diminuir chave: O(1) amortizado
- Extrair mínimo: O(log n) amortizado
- **Uso**: Dijkstra, Prim com melhor complexidade teórica

---

7. Árvores de Partilha (Tries)

7.1 Definição

Trie (Árvore Digital, Árvore de Prefixos): Estrutura para armazenar strings, onde cada caminho da raiz representa um prefixo.

Características:

- Cada nó representa um caractere
- Caminho da raiz a um nó = string/prefixo
- Nós podem marcar fim de palavra
- Compartilha prefixos comuns

```
Estrutura NóTrie:
    filhos: array[ALFABETO] de ponteiro para NóTrie
    fim_palavra: booleano
    (opcional) valor: TipoValor
```

Exemplo: Palavras "cão", "casa", "caso"

```
         raiz
          |
          c
          |
          a
         / \
        s   o (fim)
        |
        a (fim)
        |
        o (fim)
```

7.2 Operações Básicas

7.2.1 Inserção

```
InserirTrie(raiz, palavra):
    atual ← raiz

    para cada caractere c em palavra:
        se atual.filhos[c] = NULL:
            atual.filhos[c] ← novo NóTrie
        atual ← atual.filhos[c]

    atual.fim_palavra ← VERDADEIRO
```

Complexidade: O(m) onde m = comprimento da palavra

7.2.2 Busca

```
BuscarTrie(raiz, palavra):
    atual ← raiz

    para cada caractere c em palavra:
        se atual.filhos[c] = NULL:
            retornar FALSO
        atual ← atual.filhos[c]

    retornar atual.fim_palavra
```

Complexidade: O(m)

7.2.3 Busca de Prefixo

```
ComecaCom(raiz, prefixo):
    atual ← raiz

    para cada caractere c em prefixo:
        se atual.filhos[c] = NULL:
            retornar FALSO
        atual ← atual.filhos[c]

    retornar VERDADEIRO
```

Complexidade: O(m)

7.2.4 Remoção

```
RemoverTrie(raiz, palavra, profundidade):
    se raiz = NULL:
        retornar NULL

    // Última caractere
    se profundidade = tamanho(palavra):
        se raiz.fim_palavra:
            raiz.fim_palavra ← FALSO

        // Se nó não tem filhos, pode ser deletado
        se TodosFilhosNulos(raiz):
            liberar raiz
            raiz ← NULL

        retornar raiz

    // Recursão
    c ← palavra[profundidade]
    raiz.filhos[c] ← RemoverTrie(raiz.filhos[c], palavra, profundidade+1)

    // Se nó não tem filhos e não é fim de palavra, deletar
    se TodosFilhosNulos(raiz) e não raiz.fim_palavra:
        liberar raiz
        raiz ← NULL

    retornar raiz
```

Complexidade: O(m)

7.3 Autocompletar

```
AutoCompletar(no, prefixo, sugestoes):
    se no.fim_palavra:
        sugestoes.adicionar(prefixo)

    para cada filho c de no:
        se no.filhos[c] ≠ NULL:
            AutoCompletar(no.filhos[c], prefixo + c, sugestoes)
```

7.4 Complexidade e Análise

Operação	Complexidade
Inserção	O(m)
Busca	O(m)

Operação	Complexidade
Remoção	O(m)
Prefixo	O(m)
Espaço	O(ALFABETO × n × m) no pior caso

m: comprimento da palavra

n: número de palavras

ALFABETO: tamanho do alfabeto (26 para inglês)

7.5 Otimizações

7.5.1 Trie Compacta (Radix Tree / Patricia Tree)

- Compacta cadeias de nós únicos
- Reduz espaço
- Cada nó pode representar múltiplos caracteres

Exemplo:

```
Normal:  r-o-m-a-n-o (fim)
              |
              n-c-e (fim)

Compacta: roman-o (fim)
               \
                ce (fim)
```

7.5.2 Trie Ternária

- Cada nó tem 3 filhos: menor, igual, maior
- Economiza espaço (sem array grande)
- Complexidade semelhante a BST

```
Estrutura NóTrieTernaria:
    caractere: char
    fim_palavra: booleano
    menor: ponteiro
    igual: ponteiro
    maior: ponteiro
```

7.6 Aplicações

- **Autocompletar**: Sugestões em buscadores, editores
- **Corretor ortográfico**: Verificação e sugestões
- **Roteamento IP**: Tabelas de roteamento (prefixos)
- **T9 (digitação preditiva)**: Celulares antigos
- **Busca de padrões**: Matching de strings
- **Compressão**: Algoritmos LZW
- **Dicionários**: Estrutura eficiente para palavras

7.7 Vantagens e Desvantagens

Vantagens:

- Busca muito rápida: O(m) independente de n
- Operações de prefixo naturais
- Ordenação alfabética implícita
- Não há colisões (como em hash)
- Eficiente para grandes conjuntos de strings

Desvantagens:

- Uso de memória pode ser grande
- Overhead de ponteiros
- Não eficiente para alfabetos grandes
- Complexidade de implementação

---

8. Comparação e Aplicações

8.1 Tabela Comparativa Geral

Estrutura	Busca	Inserção	Remoção	Balanceamento	Espaço Extra
BST	O(h)	O(h)	O(h)	Não	Mínimo
AVL	O(log n)	O(log n)	O(log n)	Rígido	Altura
Rubro-Negra	O(log n)	O(log n)	O(log n)	Relaxado	1 bit/nó
Árvore B	O($\log_m$ n)	O($\log_m$ n)	O($\log_m$ n)	Sim	Médio
Heap	O(n)*	O(log n)	O(log n)	Completa	Mínimo

Estrutura	Busca	Inserção	Remoção	Balanceamento	Espaço Extra
Trie	O(m)	O(m)	O(m)	N/A	Alto

*Busca de elemento arbitrário; obter mín/máx é O(1)

8.2 Quando Usar Cada Estrutura

BST Simples

- Dados já balanceados naturalmente
- Protótipos rápidos
- Dados pequenos onde performance não é crítica

AVL

- Muitas operações de busca
- Poucas inserções/remoções
- Dados que requerem balanceamento estrito
- Aplicações em tempo real

Rubro-Negra

- Equilíbrio entre busca e modificação
- Muitas inserções/remoções
- Implementações de biblioteca (map, set em C++, Java)
- Kernel do Linux (agendador)

Árvore B

- Dados em disco/armazenamento secundário
- Bancos de dados e sistemas de arquivos
- Grande volume de dados
- Minimizar I/O

Heap

- Fila de prioridade
- Algoritmos de grafos (Dijkstra, Prim)
- Ordenação (HeapSort)

- Top-K elementos
- Simulações de eventos

Trie

- Dicionários e strings
- Autocompletar
- Roteamento e tabelas de prefixos
- Correção ortográfica
- Jogos de palavras

8.3 Aplicações Práticas

Banco de Dados

- **Índices B-Tree**: MySQL, PostgreSQL
- **Índices Hash + B-Tree**: Hybrid approaches
- **LSM Trees**: Cassandra, RocksDB (basedas em merge)

Sistemas Operacionais

- **Árvores Rubro-Negras**: Escalonador CFS do Linux
- **Árvores B**: Sistemas de arquivos (ext4, NTFS)
- **Heaps**: Gerenciamento de memória

Redes

- **Tries**: Tabelas de roteamento IP
- **Árvores de Prefixos**: Firewalls, ACLs

Compiladores

- **BST/AVL**: Tabelas de símbolos
- **Tries**: Análise léxica

Aplicações Web

- **Tries**: Autocompletar, busca
- **Heaps**: Ranking de resultados
- **B-Trees**: Armazenamento de sessões

---

9. Exercícios e Questões de Prova

9.1 Árvores Binárias Básicas

Q1: Quantos nós tem uma árvore binária completa de altura 4?

```
Resposta: 2^5 - 1 = 31 nós
(Fórmula: 2^(h+1) - 1)
```

Q2: Qual a altura mínima de uma árvore com 100 nós?

```
Resposta: ⌈log₂(101)⌉ - 1 = 6
```

Q3: Dada a sequência de percurso:

- Pré-ordem: A B D E C F
- Em-ordem: D B E A F C

Reconstrua a árvore.

```
Resposta:
      A
     / \
    B   C
   / \   \
  D   E   F
```

Método:

1. Raiz é primeiro em pré-ordem: A
2. Dividir em-ordem por A: (D B E) | A | (F C)
3. Recursivamente para subárvores

9.2 BST

Q4: Insira 5, 3, 7, 1, 9, 4 em uma BST vazia. Desenhe a árvore resultante.

```
Resposta:
      5
     / \
    3   7
   / \   \
  1   4   9
```

Q5: Qual a complexidade de encontrar o k-ésimo menor elemento em uma BST?

```
Resposta: O(k) se não balanceada, O(log n + k) se balanceada
Método: Percurso em-ordem até k-ésimo elemento
```

Q6: Uma BST degenerada em lista tem que complexidade?

```
Resposta: O(n) para busca, inserção e remoção
É equivalente a uma lista encadeada
```

9.3 AVL

Q7: Insira 10, 20, 30 em uma árvore AVL. Mostre as rotações.

```
Inserir 10:     10

Inserir 20:     10
                  \
                   20

Inserir 30:     10 (FB = -2, RR case)
                  \
                   20
                     \
                      30

Após rotação esquerda em 10:
                   20
                  /  \
                10    30
```

Q8: Qual o número máximo de rotações em uma inserção AVL?

```
Resposta: 2 rotações (uma dupla = duas simples)
```

Q9: A altura de uma árvore AVL com n nós é no máximo?

```
Resposta: 1.44 × log₂(n)
Altura é sempre O(log n)
```

9.4 Rubro-Negras

Q10: Qual propriedade garante que altura é O(log n)?

```
Resposta: Propriedade 5 - todos caminhos têm mesmo número de nós pretos,
combinada com propriedade 4 (sem dois vermelhos consecutivos),
garante que caminho mais longo ≤ 2 × caminho mais curto
```

Q11: Após inserir nó vermelho, o pai é vermelho e tio é preto. Que fazer?

```
Resposta: Rotação + recoloração
- Se inserção forma linha: 1 rotação
- Se forma triângulo: 2 rotações
```

Q12: Por que árvores rubro-negras são preferidas em bibliotecas padrão?

```
Resposta: Menos rotações em modificações que AVL,
boa performance em busca (altura ainda O(log n)),
e apenas 1 bit extra por nó (cor).
```

9.5 Árvores B

Q13: Uma árvore B de ordem 5 tem no mínimo quantas chaves por nó interno?

```
Resposta: ⌈5/2⌉ - 1 = 2 chaves (3 filhos)
```

Q14: Por que árvores B são melhores para disco que AVL?

```
Resposta:
- Nós grandes (um nó = um bloco de disco)
- Menos níveis (menos acessos ao disco)
- Ordem 1000 → altura 2-3 para milhões de registros
```

Q15: Árvore B de ordem 100 com 1 milhão de chaves tem altura máxima?

```
Resposta: log₅₀(1.000.000) ≈ 3.5 → altura ≤ 4
(Usa-se ⌈m/2⌉ = 50 para nós internos)
```

9.6 Heaps

Q16: Construir max-heap com [4, 10, 3, 5, 1].

```
Resposta:
Array inicial: [4, 10, 3, 5, 1]

Começar de ⌊5/2⌋-1 = 1 (índice 1 = 10):
[4, 10, 3, 5, 1] → já satisfaz heap em índice 1

Índice 0 (4):
Comparar 4 com filhos 10 e 3
Trocar 4 com 10: [10, 4, 3, 5, 1]

Heapify em 4 (agora índice 1):
Comparar 4 com filhos 5 e 1
Trocar 4 com 5: [10, 5, 3, 4, 1]

Resultado: [10, 5, 3, 4, 1]

Árvore:    10
          /  \
         5    3
        / \
       4   1
```

Q17: Qual a complexidade de construir um heap de n elementos?

```
Resposta: O(n) - não O(n log n)!
Análise tight: soma de alturas × número de nós em cada nível
```

Q18: Por que HeapSort não é estável?

```
Resposta: Trocas de longa distância (raiz com último elemento)
destroem ordem relativa de elementos iguais
```

9.7 Tries

Q19: Insira "topo", "toca", "boca" em uma Trie. Desenhe.

```
Resposta:
         raiz
        /    \
       t      b
       |      |
       o      o
      / \     |
     p   c    c
     |   |    |
    o(F) a(F) a(F)

(F) = fim de palavra
```

Q20: Qual a vantagem de Trie sobre hash table para strings?

```
Resposta:
- Busca de prefixos eficiente
- Ordenação alfabética natural
- Sem colisões
- Autocompletar fácil
- Worst case O(m) garantido (hash pode ter colisões)
```

Q21: Trie com 1000 palavras de 10 caracteres (alfabeto 26) usa quanto espaço no pior caso?

```
Resposta: 1000 × 10 × 26 × tamanho_ponteiro
≈ 260.000 ponteiros (pode ser vários MB)
Por isso Tries compactas e ternárias são importantes
```

9.8 Questões Comparativas

Q22: Compare memória: AVL vs Rubro-Negra para 1 milhão de inteiros (32 bits).

```
Resposta:
AVL: 1M × (4 bytes + 2 ponteiros + altura) ≈ 1M × 12-16 bytes = 12-16 MB
RN:  1M × (4 bytes + 2 ponteiros + 1 bit) ≈ 1M × 12 bytes = 12 MB
Diferença pequena, mas RN mais eficiente
```

Q23: Qual estrutura para ranking de jogadores que muda frequentemente?

```
Resposta: Heap (min ou max)
- Inserir nova pontuação: O(log n)
- Ver top player: O(1)
- Atualizar pontuação: O(log n)
- Remover: O(log n)
```

Q24: Estrutura para dicionário de 100k palavras com busca e prefixo?

```
Resposta: Trie ou Trie Compacta
- Busca: O(m)
- Prefixo: O(m)
- Melhor que BST/AVL que seria O(m log n) para strings
```

Q25: Sistema de arquivos com bilhões de arquivos, qual estrutura?

```
Resposta: Árvore B ou B+
- Minimiza I/O de disco
- Altura pequena (3-4 níveis para bilhões)
- Cada nó = um bloco de disco
```

9.9 Análise de Algoritmos

Q26: Analise a complexidade:

```
função Misteriosa(raiz):
    se raiz = NULL:
        retornar 0

    esq ← Misteriosa(raiz.esquerda)
    dir ← Misteriosa(raiz.direita)

    retornar 1 + max(esq, dir)
```

```
Resposta: O(n) - calcula altura da árvore
Visita cada nó exatamente uma vez
```

Q27: Qual é mais eficiente para encontrar o 5° maior elemento? a) Heap de tamanho n b) AVL c) Array ordenado

```
Resposta:
a) Heap: extrair 5 vezes = O(5 log n) = O(log n)
b) AVL: percorso reverso em-ordem até 5° = O(log n + 5) = O(log n)
c) Array: acesso direto = O(1)

Array ordenado é O(1), mas assume dados já ordenados.
Para dados dinâmicos, AVL ou Heap são melhores.
```

---

10. Resumo para Prova

10.1 Fórmulas Essenciais

Árvore Binária:

- Altura mínima: $\lceil \log_2(n+1) \rceil - 1$
- Nós em árvore completa de altura h: 2^(h+1) - 1
- Nós máximos no nível i: 2^i

AVL:

- Altura ≤ 1.44 × $\log_2(n)$
- |FB| ≤ 1 para todos os nós
- Rotações por inserção: ≤ 2
- Rotações por remoção: O(log n)

Rubro-Negra:

- Altura ≤ 2 × $\log_2(n+1)$
- Rotações por inserção: ≤ 2
- Rotações por remoção: ≤ 3

Árvore B (ordem m):

- Mínimo filhos (não-raiz): $\lceil m/2 \rceil$
- Altura: $O(\log_m n)$

Heap:

- pai(i) = $\lfloor (i-1)/2 \rfloor$
- esq(i) = 2i + 1
- dir(i) = 2i + 2

- Construir: O(n)

10.2 Complexidades para Memorizar

Operação	BST	AVL	RN	B-Tree	Heap	Trie
Busca	O(h)	O(log n)	O(log n)	$O(\log_m n)$	O(n)	O(m)
Inserção	O(h)	O(log n)	O(log n)	$O(\log_m n)$	O(log n)	O(m)
Remoção	O(h)	O(log n)	O(log n)	$O(\log_m n)$	O(log n)	O(m)
Mín/Máx	O(h)	O(log n)	O(log n)	$O(\log_m n)$	O(1)	O(m)

10.3 Dicas de Prova

1. **Desenhar sempre ajuda:** Visualize árvores para entender rotações
2. **Conheça os casos de rotação:** LL, RR, LR, RL em AVL
3. **Propriedades de cores:** Memorize as 5 propriedades rubro-negras
4. **Heap é completa:** Representação em array depende disso
5. **Trie economiza:** Prefixos comuns são compartilhados
6. **B-Tree para disco:** Minimizar I/O é o objetivo
7. **Complexidade amortizada:** Alguns algoritmos têm análise especial

10.4 Erros Comuns a Evitar

❌ **Confundir altura com profundidade**

- Altura: distância até folha mais distante (de cima para baixo)
- Profundidade: distância da raiz (de baixo para cima)

❌ **Esquecer que heap não é ordenado**

- Apenas a propriedade de heap é garantida
- Não é possível busca eficiente de elemento arbitrário

❌ **Achar que construir heap é O(n log n)**

- É O(n)! Análise tight é importante

❌ **Confundir rotações simples e duplas**

- LL/RR: 1 rotação
- LR/RL: 2 rotações (dupla)

❌ **Esquecer casos especiais em remoção BST**

- 0 filhos, 1 filho, 2 filhos têm tratamentos diferentes

Recursos Adicionais para Estudo

Visualizadores Online

- VisuAlgo (visualgo.net) - Todos os tipos de árvores
- USF CS (cs.usfca.edu/~galles/visualization) - Animações interativas
- Toptal Sorting Visualizations - HeapSort

Livros Recomendados

- **Cormen et al. - "Introduction to Algorithms"** (Capítulos 10-14, 18-19)
- **Sedgewick - "Algorithms"**
- **Weiss - "Data Structures and Algorithm Analysis"**

Prática

- LeetCode: Problemas de árvores (easy → hard)
- HackerRank: Data Structures - Trees
- Codeforces: Tree problems

Material desenvolvido para preparação de provas de AED Estude com atenção as complexidades e propriedades fundamentais! Boa sorte nos estudos! 🌳