AED2 - Algoritmos e Estr. de Dados II

Aula 9: Heaps e Compressão de Huffman

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Fevereiro de 2026

Sumário

1. O Conceito de Fila de Prioridade

Em uma fila comum (FIFO), o primeiro a chegar é o primeiro a ser atendido. Em uma **Fila de Prioridade**, cada elemento tem uma "urgência" associada. O elemento de **maior prioridade** é sempre atendido primeiro, não importa quando chegou. Exemplos:

- Triagem de Hospital (Emergência > Dor de garganta).
- Agendador de CPU (Processo de Sistema > Processo de Usuário).
- Algoritmo de Dijkstra (Menor distância atual > Maior distância).

A estrutura de dados mais eficiente para implementar isso é o **Heap Binário**.

2. Heap Binário: Definição e Representação

Heap: árvore quase completa. Max-Heap: pai $\geq$ filhos. Min-Heap: pai $\leq$ filhos. Em vetor: pai $i/2$, filhos $2i$ e $2i+1$.

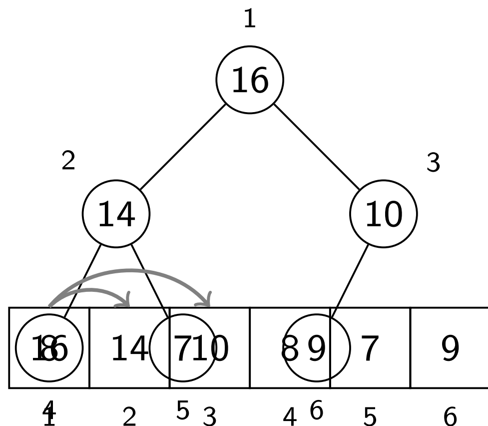


Figure: Heap estruturado e mapeado em vetor.

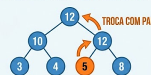
3. Heap Binário: Operações Principais

Operações Principais

1. **Insert ($O(\log N)$):** Adiciona o elemento no **final** do vetor e faz o **Sift-Up** (sobe trocando com o pai). 2.

Extract-Max ($O(\log N)$):

- Retira a Raiz (índice 1).
- Coloca o **último** elemento na Raiz.
- Faz o **Sift-Down** (desce trocando com o maior filho).



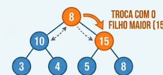


Figure: Processo de Sift (Ajuste).

4. Prática em Java: Min-Heap

O Java possui a classe `PriorityQueue` implementando um **Min-Heap**.

```java
import java.util.PriorityQueue;

public class ExemploMinHeap {
    public static void main(String[] args) {
        // Padrao: Min-Heap (Menor sai primeiro)
        PriorityQueue<Integer> minh = new PriorityQueue<>();

        minh.add(10);
        minh.add(5);
        minh.add(20);

        System.out.println(minh.poll()); // Saida: 5
        System.out.println(minh.poll()); // Saida: 10
    }
}
```

5. Prática em Java: Max-Heap

Para usar como **Max-Heap**, utilizamos um comparador reverso:

```java
// Uso de Collections.reverseOrder()
PriorityQueue<Integer> maxh = new PriorityQueue<>(
    Collections.reverseOrder()
);

maxh.add(10);
maxh.add(5);
maxh.add(20);

System.out.println(maxh.poll()); // Saida: 20
```

6. Aplicação: HeapSort

Algoritmo HeapSort

Uma aplicação direta de Heaps é ordenar um vetor em $O(N \log N)$ e espaço $O(1)$ extra (se feito in-place).

1. Construa um Max-Heap a partir vetor desordenado.
2. Troque a Raiz (maior) com o último.
3. Reduza o tamanho do heap e chame Sift-Down na nova raiz.
4. Repita até que o vetor esteja ordenado.

7. Alfabeto & Codificação (Tamanhos Fixos)

Considere um alfabeto contendo n símbolos e uma mensagem composta por símbolos deste alfabeto. Deseja-se codificar a mensagem convertendo cada símbolo para binário (0s e 1s).

Exemplo: Alfabeto = {A, B, C, D}

Mensagem a codificar: **ABACCDA** (7 letras).

Se utilizarmos representações de tamanho fixo:

- $A \rightarrow 010$, $B \rightarrow 100$, $C \rightarrow 000$, $D \rightarrow 111$ (3 bits) $\Rightarrow$ Mensagem final usa **21 bits** (7×3). Ineficiente!
- $A \rightarrow 00$, $B \rightarrow 01$, $C \rightarrow 10$, $D \rightarrow 11$ (2 bits) $\Rightarrow$ Mensagem final usa **14 bits** (7×2). Mais eficiente.

Pergunta: Podemos fazer algo melhor (usar menos bits no total)?

8. Alfabeto & Codificação (Tamanho Variável)

Se utilizarmos **códigos de tamanho variável**, podemos atribuir códigos mais curtos aos símbolos mais frequentes!

Exemplo de Código Variável

Símbolos e Frequências na mensagem **ABACCDA**: $A = 3, C = 2, B = 1, D = 1$.

Tentativa de codificação:

- Símbolos frequentes *curtos*: $A \rightarrow 0$, $C \rightarrow 10$.
- Símbolos pouco frequentes *longos*: $B \rightarrow 110$, $D \rightarrow 111$.

Mensagem codificada: **0110010101110**.
Aqui serão necessários apenas **13 bits** $\rightarrow$ Codificação eficiente!

9. A Regra do Prefixo

O problema dos códigos de tamanho variável é a decodificação. Como o computador sabe onde termina uma letra e começa a outra?

O código de um símbolo não deve ser prefixo do outro!

Se $A \rightarrow 0$ e $C \rightarrow 01$, a string "01" poderia ser "A" seguido de algo livre, ou "C". É ambíguo!

Figure: A regra do prefixo garante decodificação determinística.

Para garantir a Regra do Prefixo de forma ótima, utilizamos as **Árvores de Huffman**.

10. Intuição do Algoritmo

Mecânica Básica

1. Cada símbolo é um nó folha com sua frequência.
2. Pegue os **dois nós de menor frequência** e una-os num novo nó.
3. A frequência do novo nó é a soma dos dois filhos.
4. Repita até sobrar só um nó (a raiz da árvore).

Figure: Exemplo de estrutura árvore para Huffman.

11. Compressão de Huffman: Algoritmo

O algoritmo de Huffman usa uma Fila de Prioridade para compressão de dados **Lossless** (sem perda). A ideia é usar códigos binários menores para caracteres mais frequentes.

Algoritmo Guloso (Geração da Árvore)

1. Conte a frequência de cada caractere na string. 2. Crie um nó folha para cada caractere e insira em uma **Min-PriorityQueue**. 3. Enquanto houver mais de 1 nó na fila: a. Remova os dois nós de menor frequência (A e B). b. Crie um novo nó interno P com frequência $=$ $freq(A) + freq(B)$. c. Faça A e B serem filhos de P. d. Insira P na fila.

12. Compressão de Huffman: Exemplo "banana"

Exemplo Passo 0

Texto: "banana" $(b:1, n:2, a:3)$. Fila: `[(b,1), (n,2), (a,3)]` 1. Tira `(b,1)` e `(n,2)`.

Soma = 3. Cria nó `(bn, 3)`. Fila: `[(bn, 3), (a, 3)]`. 2. Tira `(bn, 3)` e `(a, 3)`. Soma = 6. Cria Raiz `(Total, 6)`. Códigos: Caminhar para esquerda = 0, direita = 1. Se 'a' ficar

na direita da raiz → Código `1` (1 bit).

Implementação Simplificada do Nó em Java:

```java
class HuffmanNode implements Comparable<HuffmanNode> {
    char c;
    int frequency;
    HuffmanNode left, right;

    public int compareTo(HuffmanNode other) {
        return this.frequency - other.frequency;
    }
}
```

14. Huffman: Algoritmo de Construção

```java
// 1. Criar nos folha para cada caractere
PriorityQueue<HuffmanNode> pq = new PriorityQueue<>();
for (char c : frequencies.keySet()) {
    pq.add(new HuffmanNode(c, frequencies.get(c)));
}

// 2. Unir ate sobrar apenas a raiz
while (pq.size() > 1) {
    HuffmanNode left = pq.poll();  // Menor
    HuffmanNode right = pq.poll(); // Segundo menor

    // Novo no interno (soma das frequencias)
    HuffmanNode parent = new HuffmanNode(' ',
        left.frequency + right.frequency, left, right);

    pq.add(parent);
}

HuffmanNode root = pq.poll(); // Raiz final
```

15. Aplicações de Heaps em Grafos

Heaps são o "motor" de algoritmos vitais de Grafos: 1. **Algoritmo de Dijkstra:** O heap decide qual vértice explorar em seguida (o de menor distância acumulada). Isso reduz a complexidade de $O(V^2)$ para $O(E \log V)$. 2. **Algoritmo de Prim:** Para Árvore Geradora Mínima, similar ao Dijkstra. **Exercícios Sugeridos:**

- **Beecrowd 1252:** Sort! (Custom Comparator).
- **LeetCode 215:** Kth Largest Element in an Array (Use um Min-Heap de tamanho K).
- **LeetCode 23:** Merge k Sorted Lists.

16. A Estrutura da Árvore de Huffman

A Árvore de Huffman é uma **Árvore Binária Estrita**, construída de baixo para cima (*bottom-up*).

Características

- **Folhas:** Caracteres originais.
- **Caminho:** Determina o código binário (Esq=0, Dir=1).

Objetivo: Letras frequentes ficam perto da raiz (códigos curtos).

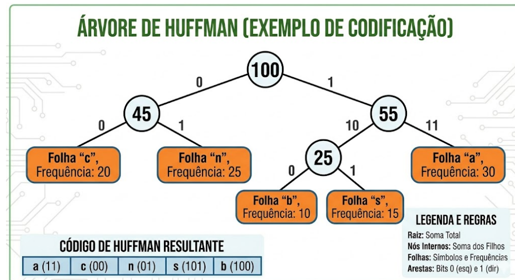


Figure: Resumo da estrutura.

17. Princípio do Algoritmo Guloso

A construção da árvore usa uma abordagem **gulosa** (greedy) para garantir uma compressão ótima passo a passo.

Por que escolhemos sempre os dois menores?

- Ao unir os elementos de *menor frequência* na base da árvore, forçamos os seus caminhos a serem os mais longos.
- Como eles aparecem poucas vezes no texto, o custo de ter um código longo para eles é minúsculo comparado a poupar bits nas letras frequentes.

Resultado: Minimização do tamanho total do arquivo $\sum(\text{freq}_i \times \text{profundidade}_i)$.

18. Como os Dados são Trafegados?

Ao compactar, não basta apenas converter o texto com base nos novos códigos curtos. O programa descompactador **precisa saber** qual foi a árvore gerada!

Estrutura de Arquivo Compactado (Huffman)

1. **Cabeçalho (Header):** A tabela de frequências (ou a árvore serializada) para reconstruir o dicionário.
2. **Padding:** O número de bits "extras" adicionados no fim, pois arquivos são sempre gravados em bytes inteiros (múltiplos de 8).
3. **Dados:** A sequência de bits real (o texto empacotado).

Importante: Sem salvar a árvore junto (ou defini-la de forma padrão), o arquivo binário é inútil.

19. Exemplo Prático: Árvore de Huffman

Para montar uma Árvore de Huffman, o segredo é sempre manter a lista de nós ordenada e usar uma **Fila de Prioridade (Min-Heap)**.

Dados Iniciais (Caracteres e Frequências)

H	F	G	B	C	D	A	E	I
1	4	6	6	7	12	15	25	25

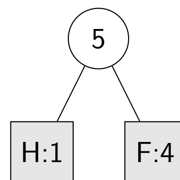
Algoritmo (Revisão)

1. Pegar os dois menores da fila.
2. Unir em um novo nó (Soma das frequências).
3. Reinserir na fila.
4. Repetir até sobrar apenas um nó (Raiz).

Passo 1: Fila: [H:1, F:4, G:6, B:6, C:7, D:12, A:15, E:25, I:25]

Unir H(1) e F(4)
Novo nó (HF): 5

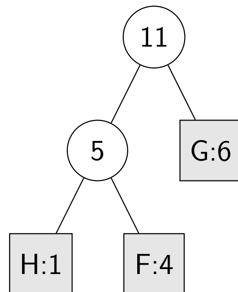
Nova Fila:
[HF:5, G:6, B:6, C:7, D:12, A:15, E:25, I:25]



Passo 2: Fila: [HF:5, G:6, B:6, C:7, D:12, A:15, E:25, I:25]

Unir HF(5) e G(6)
Novo nó (HFG): 11

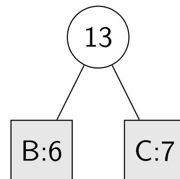
Nova Fila:
[B:6, C:7, HFG:11, D:12, A:15, E:25, I:25]



Passo 3: Fila: [B:6, C:7, HFG:11, D:12, A:15, E:25, I:25]

Unir B(6) e C(7)
Novo nó (BC): 13

Nova Fila:
[HFG:11, D:12, BC:13, A:15, E:25, I:25]



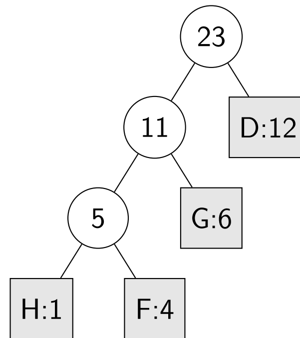
Nota Pedagógica

Repare que **BC:13** é uma nova árvore isolada. Ela ainda não se une à árvore **HFG:11** porque, no momento, elas são árvores independentes em uma "floresta".

Passo 4: Fila: [HFG:11, D:12, BC:13, A:15, E:25, I:25]

Unir HFG(11) e D(12)
Novo nó (HFGD): 23

Nova Fila:
[BC:13, A:15, HFGD:23, E:25, I:25]



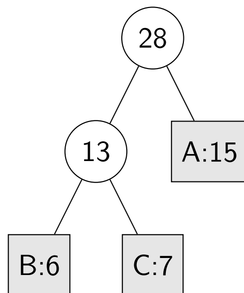
Nota

Agora sim, a árvore **HFG** foi unida ao nó **D**, pois eram os dois menores da fila atual. A árvore **BC** continua aguardando sua vez.

Passo 5: Fila: [BC:13, A:15, HFGD:23, E:25, I:25]

Unir BC(13) e A(15)
Novo nó (BCA): 28

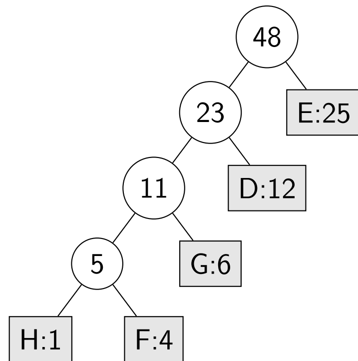
Nova Fila:
[HFGD:23, E:25, I:25, BCA:28]



Passo 6: Fila: [HFGD:23, E:25, I:25, BCA:28]

Unir HFGD(23) e E(25)
Novo nó (HFGDE): 48

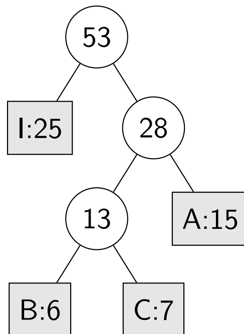
Nova Fila:
[I:25, BCA:28, HFGDE:48]



Passo 7: Fila: [I:25, BCA:28, HFGDE:48]

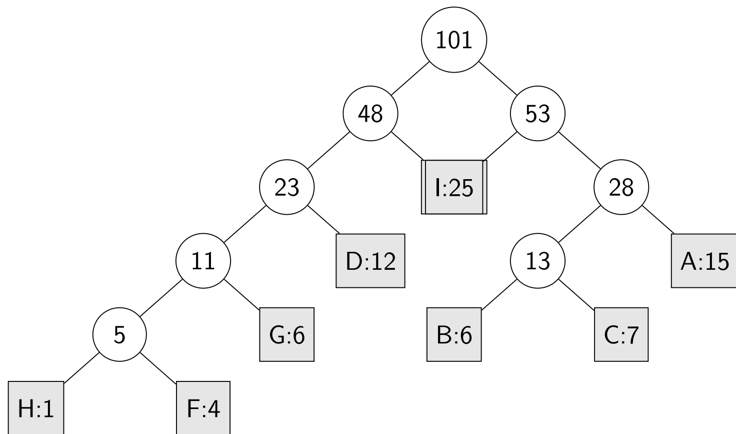
Unir I(25) e BCA(28)
Novo nó (IBCA): 53

Nova Fila:
[HFGDE:48, IBCA:53]



Fusão Final
Raiz: 101

Concluído!
Árvore otimizada para codificação.



20. Insights da Árvore de Huffman

Observe a relação entre a frequência e o caminho até a folha:

- **Frequência vs. Profundidade**: Caracteres mais frequentes (E, I, A) ficaram com caminhos **curtos**. O 'H' (freq 1) ficou com o caminho **longo**.
- **Cálculo de Bits**:
  - Se fosse ASCII (8 bits): $101 \times 8 = 808$ bits.
  - Com Huffman: $\sum(\text{freq}_i \times \text{profundidade}_i)$.

21. Huffman: Empate e Desafio

O Pulo do Gato (Empates)

Se houver empate na fila (ex: Passo 3, unir 'B' com 'C' ou com 'G'), a estrutura da árvore muda, mas **a compressão total em bits permanece idêntica**.

Desafio para Turma

Qual o código binário da letra **F** nesta árvore?
(Siga: Esquerda=0, Direita=1).