Algoritmos e Estruturas de Dados II

Capítulo da Aula 9: Heaps e Compressão de Huffman

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br
CEFET-MG - Campus Timóteo

Fevereiro de 2026

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

Um Heap é uma Árvore Binária **Quase Completa** (preenchida da esquerda p/ direita) que satisfaz a **Propriedade do Heap**:

- **Max-Heap:** Pai $\geq$ Filhos (Raiz é o Maior).
- **Min-Heap:** Pai $\leq$ Filhos (Raiz é o Menor).

Representação em Vetor (Array)

Como a árvore é completa, não precisamos de ponteiros `left` e `right`. Podemos armazenar tudo em um vetor linear! Para um nó no índice i (indexado em 1):

- **Pai:** $\lfloor i/2 \rfloor$
- **Filho Esquerdo:** $2 \times i$
- **Filho Direito:** $2 \times i + 1$

Figure 1: Heap binário representado em vetor: pai em i/2, filhos em 2i e 2i+1.

Operações Principais

1. **Insert ($O(\log N)$):** Adiciona o elemento no **final** do vetor (primeira folha livre) e faz o **Sift-Up** (troca com o pai enquanto for maior que ele). 2. **Extract-Max ($O(\log N)$):**

- Retira a Raiz (índice 1).
- Coloca o **último** elemento do vetor na Raiz.
- Faz o **Sift-Down** (troca com o maior filho até restaurar a propriedade).

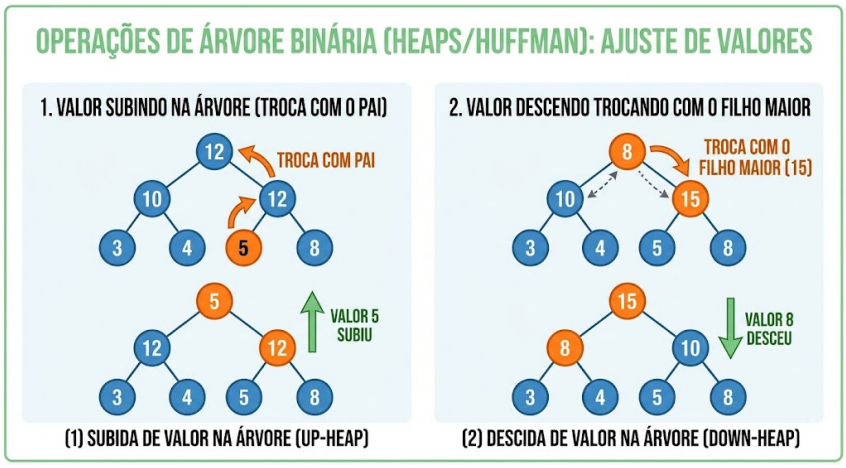

Figure 2: Sift-up: subir trocando com o pai. Sift-down: descer trocando com o maior filho.

3. Prática em Java (‘PriorityQueue‘)

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

```java
import java.util.PriorityQueue;
import java.util.Collections;

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

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

        System.out.println(minh.poll()); // Remove e imprime 5
        System.out.println(minh.poll()); // Remove e imprime 10

        // Para Max-Heap: Usar Collections.reverseOrder()
        PriorityQueue<Integer> maxh = new PriorityQueue<>(Collections.rever

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

        System.out.println(maxh.poll()); // Remove e imprime 20
    }
}
```

HeapSort

Uma aplicação direta 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.

4. O Algoritmo de Compressão de Huffman

A compressão de Huffman, apresentada por David A. Huffman, marca um divisor de águas na base computacional e é um dos ápices da Teoria da Informação. Consiste integralmente em um método de compressão mecânica de dados do tipo **Lossless** (sem perdas algorítmicas) movido primordialmente pelas frequências absolutas estatísticas de aparição nativas de um arquivo.

4.1. Entropia e Códigos Livres de Prefixo

Nos mapeamentos padrão computacionais fixos, como ASCII ou UTF-8, cada fragmento ou caractere sequestra um bloco fixo engessado do hardware (ex.: 8 bits). Porém, baseando-se por entropia linguística prática, uma vogal 'A' reaparece enxurradas de vezes a mais que uma obscura letra 'Z'. O princípio motor da técnica de Huffman assume que

caracteres estatisticamente redundantes devem usurpar códigos incrivelmente curtos, arrastando os enfileiramentos maiores de bytes para caracteres quase inexistentes.

Contudo, ao lidar com peças binárias irregulares e de tamanhos soltos em um fluxo elétrico desordenado, deve-se salvaguardar estritamente a decodificação matemática da ambiguidade. Para tanto, o ambiente adota rigorosamente as leis dos **Códigos Livres de Prefixo** (*Prefix-Free Code*). Nenhuma estola ou corrente de bits encarregada de estampar uma letra pode pertencer paralelamente ao princípio exato das bits de outra letra.

4.2. A Árvore Tridimensional (Trie) Matemática de Decodificação

A imensa sacada filosófica para decodificar prefixos variados foi isolá-los estritamente aos nós **Folhas** de uma vasta malha binária. Durante a tradução estruturada da máquina:

- Adentra-se livremente na geometria do nó raiz para começar o processo a cada laço novo de bytes;
- Mapear e pinçar o bit elétrico `0` faz os ponteiros descerem agressivamente na aresta gravitacional à **Esquerda**.
- Mapear e pinçar o bit elétrico `1` arremessa as coordenadas para o Nó ponteiro contíguo à gravitacional **Direita**.
- Ao estampar em uma folha limite terminal sem galhos subjacentes, extrai-se imediatamente a string e o ponteiro interno virtual entra num portal, reiniciando livre na Raiz Superior para quebrar o bit lógico subsequente.

4.3. Estratégia Greedy e Motor de Heaps

Para garantir a otimização dimensional perfeita total de bits por área alocada, desenham-se as ramificações brutalmente subvertidas: a Árvore consolida-se de **baixo para o topo** afunilando e esmagando os nós de origens raras do arquivo. Essa manobra complexa só opera mecanicamente em instâncias perfeitas pela mediação em blocos através da impiedosa **Fila de Prioridades (Min-Heap)**.

Passo a Passo Físico:

1. **Pesagem Global:** Leia e consuma o arquivo cru nativo empacotando e contabilizando com Hashes todos os bytes repetidos em frequência pura numérica.

2. **Isolamento Alocado:** Extraia cada byte como um Nó Solteiro desatrelado folhado carregando a assinatura e frequência. Abasteça todos impiedosamente sem vínculos nas entranhas processuais do **Min-Heap**.

3. **Aglutinação Combinatória:** Enquanto repousar livre mais de 1 vértice dentro do fluxo orgânico iterativo Heap:

   - Consuma sequencialmente os **dois** únicos nós possuidores do menor indício de incidência do conjunto.
   - Suba e instancie temporariamente neles um "Nó Pai Controlador Cego" puramente abstrato alocado sem byte que abrigará o somatório total absoluto dessas duas frequências combinadas.

- Acople impiedosamente o mais esquecido/menor índice para a restrição da *esquerda* nativa e o colega acoplado à *direita*.
- Recarregue (enfileire) o Nó Pai aglutinador gordo de volta ao **Min-Heap**. O rebalanceamento da fila aplicará o Sift-Up matemático normalizando globalmente prioridades.

4 **O Remanescente:** Incondicionalmente, o último formidável mega Nó estagnado sozinho abarcando a integralidade combinatória no topo encerra a Árvore totalizada daquele ecossistema base como a imponente **Raiz Final**.

4.4. A Clássica Mecânica no Fluxo de Execução - "BANANA"

Simulando empiricamente as contagens isoladas em **"BANANA"**.
Extrações Nativas: `B:1`, `N:2`, `A:3`.
Carga empilhada primária arremessada no Min-Heap Físico.

1 A fila expele os minúsculos: O estrito nó 'B' (1) e o elo 'N' (2).

2 Cria-se na memória intersecional o nó Pai Controlador de massa (1+2)=3. O ponteiro interno esquerdo crava em 'B' e ponteiro direito engata o nó 'N'.

3 A carga do sistema empurra de volta subvertendo esse massivo PAI Controlador (Volume 3) na malha de triagem Min-Heap.

4 A Fila isola agora duas entidades idênticas remanescentes globais: o invólucro `PAI Controlador(3)` e a estática folha virgem de fábrica `'A'(3)`. A Fila processa ambos despejando a suprema **Raiz Final** da carga inteira aglutinando massa de 6.

O Fechamento Matemático do Dicionário Binário:

- 'A': Escavado da base central para lateral direita limpa → Código espremido `1` (Apenas 1 mísero de bit limite).
- 'B': Isolado e afundado duplamente pro abismo da esquerda → Código sub-dimensionado `00` (2 bits pesados).
- 'N': Afastado pro canal secundário inferior (esquerda e reescalado na direita) → Código estruturado matriz `01` (2 bits brutos).

Um array computacional para "BANANA" rodaria como `00` (B) | `1` (A) | `01` (N) | `1` (A) | `01` (N) | `1` (A). Convertendo blocos enxugamos matematicamente as pesagens elétricas a absurdos 9 meros bits absolutos formatados encadeados em bloco no HDD `001011011`. Contraposição brutal à força estante mecânica burra de conversões massivas limitadas de Trens ASCII de 48 pesados bits físicos.

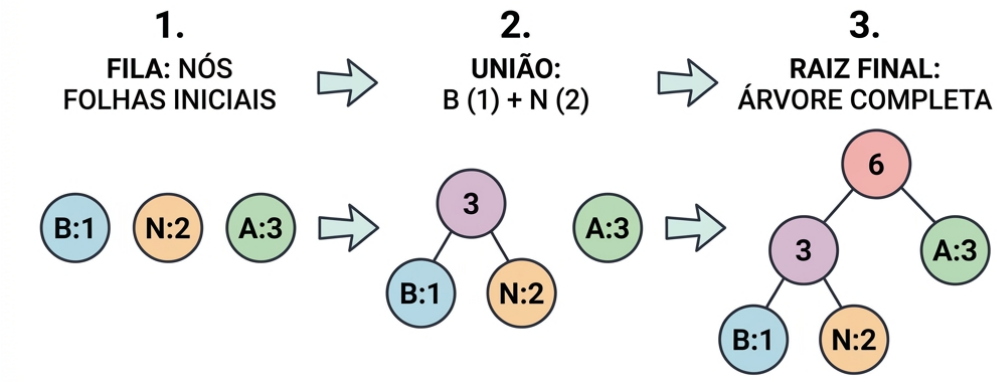

Figure 3: Processo iterativo em camadas ilustrando a convergência final estrutural de Huffman para "BANANA".

4.5. Engenharia Estrutural e Overhead Analítico

- **Big-O da Base (Construção da Fila):** Operando em Min-Heaps puríssimas efetuamos transações limitadas ao número de classes exclusivas (O alfabeto nativo do pacote estrito $= K$). A absorção massiva na mecânica combinatural causa loop $\mathcal{O}(K \log K)$. E varreduras dos gigas totais nativos espreme a matriz total apenas dominando em tráfego limite logístico linear o gigante L: finalizando $\mathcal{O}(L)$.
- **Overhead e Serialização Real:** Em arquivos gigantes um Dicionário base empacota pesagens pífias, mas a grande rasteira empírica e pegadinha da engenharia real atinge arquivos minúsculos estritos por Overhead! Sem a Árvore codificada original decifratória exportada no exato disco alvo reverso não há mágica matemática capaz de retro-traduzir chaves isoladas. Logo exigiremos incrustar cabeçalhos estruturais injetando integralmente todos os metadados de Huffman e limites das raízes nos bytes de frente para as assinaturas no descompactador receptor do arquivo base, prejudicando e estragando atrocidades estáticas a compactação final líquida se as tramas puras não

compensarem o gigantesco escopo de bytes do recém-nascido Dicionário imposto.

5. 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.

6. Exercícios

6.1. Exercícios Conceituais

1 Explique a propriedade do heap (heap property) para Max-Heap e Min-Heap. Por que a representação em vetor é eficiente para heaps binários?

2 Compare Heap com BST em termos de operações suportadas, complexidade e uso de memória. Quando cada estrutura é mais apropriada?

3 Descreva o algoritmo de compressão de Huffman e explique por que ele é considerado um algoritmo guloso. Qual é a propriedade que garante a otimalidade?

6.2. Exercícios Analíticos

1 Construa um Max-Heap a partir do vetor `[4, 10, 3, 5, 1, 8, 7]` usando o algoritmo bottom-up (heapify). Mostre cada etapa da construção.

2 Simule a inserção do valor 15 em um Max-Heap existente e a remoção do elemento máximo, mostrando todas as etapas de sift-up e sift-down.

3 Para o texto "ENGENHARIA", construa a árvore de Huffman passo a passo, calculando as frequências de cada caractere e mostrando os códigos binários resultantes.

6.3. Exercícios de Programação

1 Implemente um Heap Binário completo em Java (Max-Heap e Min-Heap) com operações `insert`, `extractMax`/`extractMin`, `peek` e `heapify`. Compare com `PriorityQueue` do Java.

2 Implemente o algoritmo de compressão de Huffman:

- Leia um texto e calcule frequências.
- Construa a árvore de Huffman.
- Gere os códigos binários.
- Codifique e decodifique o texto.

3 Resolva problemas de juiz online que usam heaps, como:

- **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.
- **LeetCode 347:** Top K Frequent Elements.

Para cada problema, analise a complexidade de tempo e espaço.