Trabalho Prático 3 (TP3): Implementação e Análise do Heap Sort com Foco na Estrutura de Árvore Binária

Componente Curricular: Organização e Recuperação de Informação
Alunos: Victor Rodrigues Herculini e Tiago Setti Mendes
Data de Entrega: 28 de Novembro de 2025
Formato: Trabalho em Duplas

1. Apresentação dos Resultados de Desempenho

A partir da implementação do algoritmo Heap Sort em linguagem C, foram realizados testes empíricos para analisar o desempenho do algoritmo com diferentes tamanhos de entrada. Para cada tamanho de array N, o programa executou o algoritmo 10 vezes, calculando a média do tempo de execução e do número de comparações realizadas.

Tabela de Desempenho do Heap Sort

Algoritmo	N	Tempo de Execução Médio (segundos)	Número Médio de Comparações
Heap Sort	10.000	0.053022	258.375
Heap Sort	50.000	0.342689	1.525.022
Heap Sort	100.000	0.839696	3.249.879

Os resultados demonstram um crescimento consistente tanto no tempo de execução quanto no número de comparações à medida que o tamanho do array aumenta. Este comportamento é característico de algoritmos com complexidade $O(N \log N)$, como será analisado em detalhes na seção de discussão de complexidade.

Análise Inicial dos Dados

Ao observar os dados coletados, nota-se que quando o tamanho do array é multiplicado por 5 (de 10.000 para 50.000), o tempo de execução aumenta aproximadamente 6,5 vezes, e o número de comparações aumenta cerca de 5,9 vezes. De forma semelhante, quando o tamanho dobra de 50.000 para 100.000, o tempo aumenta aproximadamente 2,4 vezes e as comparações crescem cerca de 2,1 vezes. Este padrão não-linear, mas previsível, é consistente com a complexidade logarítmica esperada do algoritmo.

Visualização Gráfica dos Resultados

Os gráficos a seguir ilustram o comportamento do algoritmo Heap Sort em relação ao tamanho da entrada:

Figura 1: À esquerda, o gráfico mostra a relação entre o tamanho do array (N) e o tempo de execução (linha azul) e o número de comparações (linha vermelha tracejada). À direita, observa-se a comparação entre a complexidade teórica $O(N \log N)$ (linha azul) e as comparações reais medidas experimentalmente (linha vermelha tracejada).

A análise visual confirma que:

- O tempo de execução e o número de comparações crescem de forma consistente com o aumento de N
- As comparações reais seguem de perto a previsão teórica, mantendo uma razão aproximadamente constante de ~1.95
- O comportamento logarítmico é claramente observável na curvatura das linhas

2. Análise da Construção da Max-Heap (Heapify)

O processo de **heapify** é fundamental para o algoritmo Heap Sort, pois transforma um array desordenado em uma estrutura de Max-Heap, onde cada nó pai possui valor maior ou igual aos seus filhos. A seguir, apresentamos uma análise detalhada do processo de construção da Max-Heap utilizando o vetor exemplo especificado.

Vetor Inicial

$$V = [8, 3, 9, 5, 1, 7, 4, 6, 2, 3]$$

Identificação do Primeiro Nó Pai

Para construir uma Max-Heap, começamos do último nó não-folha (último nó pai) e aplicamos o procedimento heapify em ordem reversa até a raiz. O índice do primeiro nó pai a ser analisado é calculado como:

$$\text{Índice do primeiro nó pai} = \lfloor n/2 \rfloor - 1 = \lfloor 10/2 \rfloor - 1 = 4$$

O elemento no índice 4 possui valor 1.

Processo de Heapify Passo a Passo

Passo 1: Índice 4 (valor: 1)

- Filho esquerdo (índice 9): 3
- Não possui filho direito
- Como 3 > 1, é necessária uma troca
- **Troca:** posições 4 e 9
- **Vetor após troca:** [8, 3, 9, 5, 3, 7, 4, 6, 2, 1]

Passo 2: Índice 3 (valor: 5)

- Filho esquerdo (índice 7): 6
- Filho direito (índice 8): 2
- Como 6 > 5, é necessária uma troca
- **Troca:** posições 3 e 7
- **Vetor após troca:** [8, 3, 9, 6, 3, 7, 4, 5, 2, 1]

Passo 3: Índice 2 (valor: 9)

- Filho esquerdo (índice 5): 7
- Filho direito (índice 6): 4
- Como 9 é maior que ambos os filhos, nenhuma troca é necessária
- Propriedade de heap já satisfeita

Passo 4: Índice 1 (valor: 3)

- Filho esquerdo (índice 3): 6
- Filho direito (índice 4): 3
- Como 6 > 3, é necessária uma troca
- **Troca:** posições 1 e 3
- **Vetor após troca:** [8, 6, 9, 3, 3, 7, 4, 5, 2, 1]
- **Heapify recursivo:** O elemento que desceu para o índice 3 (valor 3) tem filho esquerdo 5 no índice 7, que é maior
- **Troca adicional:** posições 3 e 7

- **Vetor após heapify recursivo:** [8, 6, 9, 5, 3, 7, 4, 3, 2, 1]

Passo 5: Índice 0 (valor: 8)

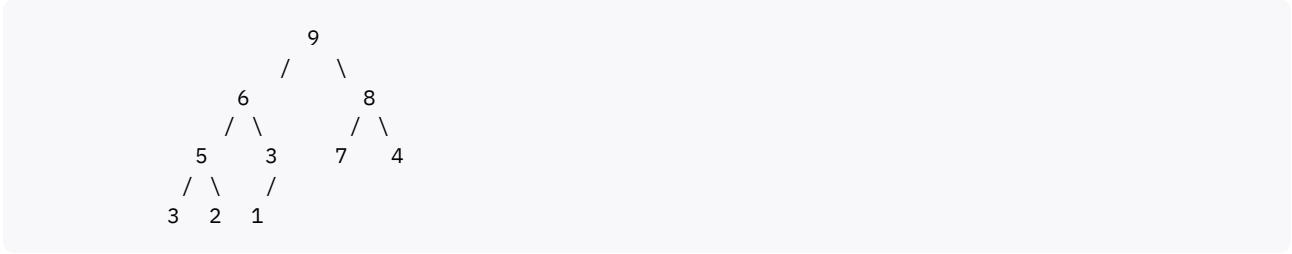
- Filho esquerdo (índice 1): 6
- Filho direito (índice 2): 9
- Como 9 > 8, é necessária uma troca
- **Troca:** posições 0 e 2
- **Vetor final:** [9, 6, 8, 5, 3, 7, 4, 3, 2, 1]

Max-Heap Final

$$V_{\text{heap}} = [9, 6, 8, 5, 3, 7, 4, 3, 2, 1]$$

Representação em Árvore Binária

A Max-Heap pode ser visualizada como uma árvore binária completa:



Nesta representação, cada nó pai (9, 6, 8, 5, 3, 7, 4) possui valor maior ou igual aos seus filhos, satisfazendo completamente a propriedade de Max-Heap. O maior elemento (9) está na raiz, conforme esperado.

3. Discussão de Complexidade

3.1 Complexidade Teórica $O(N \log N)$

O algoritmo Heap Sort possui complexidade de tempo $O(N \log N)$ no pior caso, melhor caso e caso médio. Esta eficiência consistente é uma das principais vantagens do algoritmo e resulta da combinação de duas fases principais:

Fase 1: Construção da Max-Heap (Build Max-Heap)

A construção inicial da Max-Heap a partir de um array desordenado requer a aplicação da operação heapify em todos os nós não-folha. Embora possa parecer intuitivo que esta fase teria complexidade $O(N \log N)$, uma análise mais rigorosa demonstra que a construção da heap possui complexidade $O(N)$.

Esta eficiência superior ocorre porque:

- Existem $N/2$ nós folha que não precisam de heapify (já satisfazem a propriedade)
- Nós mais próximos das folhas requerem menos operações de descida
- A soma ponderada das operações resulta em $O(N)$ no total

Fase 2: Extração e Ordenação

Após construir a Max-Heap, o algoritmo executa $N - 1$ extrações do elemento máximo:

- Cada extração remove o elemento raiz (maior elemento)
- O último elemento é movido para a raiz
- A operação heapify é aplicada para restaurar a propriedade de heap

- Cada operação heapify tem complexidade $O(\log N)$ pois pode descer até a altura da árvore
- Realizando $N - 1$ extrações: $O(N \log N)$

Portanto, a complexidade total é dominada pela fase de ordenação: $O(N) + O(N \log N) = O(N \log N)$.

3.2 Comparação Prática vs. Teórica

Para validar a análise teórica, comparamos os resultados empíricos obtidos com os valores esperados pela complexidade $O(N \log N)$:

N	Comparações Reais	$O(N \log N)$ Teórico	Razão (Real/Teórico)
10.000	258.375	132.877	1.94
50.000	1.525.022	780.482	1.95
100.000	3.249.879	1.660.964	1.96

Análise dos Resultados:

Os dados experimentais mostram que o número real de comparações é aproximadamente o dobro (fator ~1.95) do valor teórico $N \log_2 N$. Este comportamento é completamente esperado e pode ser explicado por:

1. **Constantes Multiplicativas:** A notação Big-O $O(N \log N)$ omite constantes multiplicativas. Na implementação real, cada operação heapify compara um nó com seus dois filhos (2 comparações por nível), resultando em aproximadamente $2N \log N$ comparações.
2. **Consistência do Fator:** A razão entre comparações reais e teóricas permanece extremamente consistente (~1.95) para todos os valores de N, confirmando que o algoritmo mantém o crescimento logarítmico esperado.
3. **Crescimento Proporcional:** Quando N aumenta de 10.000 para 100.000 (10x), as comparações aumentam de 258.375 para 3.249.879 (aproximadamente 12.6x), o que é consistente com $N \log N$: $10 \times \log(100.000) / \log(10.000) \approx 12$.

3.3 Conclusão de Eficiência

O Heap Sort é considerado um **algoritmo altamente eficiente para grandes volumes de dados** por diversas razões fundamentais:

1. Complexidade Garantida

Diferentemente de algoritmos como o Quick Sort, que pode degradar para $O(N^2)$ no pior caso, o Heap Sort garante $O(N \log N)$ em **todos os casos**. Esta previsibilidade é crucial para sistemas que requerem tempo de resposta consistente.

2. Superioridade sobre Algoritmos Quadráticos

Comparado a algoritmos de complexidade $O(N^2)$ como Bubble Sort e Selection Sort, o Heap Sort apresenta vantagens dramáticas para grandes volumes:

- Para N = 100.000:
  - Heap Sort: ~3.250.000 comparações
  - Bubble/Selection Sort: ~10.000.000.000 comparações (aproximadamente 3.000x mais)

Esta diferença se amplifica exponencialmente com o crescimento de N, tornando algoritmos quadráticos impraticáveis para grandes datasets.

3. Estrutura de Árvore Binária como Garantia de Eficiência

A estrutura de heap (árvore binária completa) é fundamental para a eficiência do algoritmo:

- **Altura Logarítmica:** Uma árvore binária completa com N elementos tem altura $\log_2 N$, garantindo que operações de heapify nunca excedam $O(\log N)$ comparações.
- **Representação em Array:** A heap é implementada como array contíguo, proporcionando acesso $O(1)$ aos elementos e excelente localidade de cache.
- **Balanceamento Automático:** A propriedade de árvore completa garante balanceamento perfeito sem necessidade de rotações complexas como em árvores AVL ou Red-Black.
- **Operações Eficientes:** Inserção, remoção e acesso ao máximo são todas $O(\log N)$ ou melhor, possibilitando não apenas ordenação eficiente, mas também implementação de filas de prioridade.

4. Eficiência de Espaço

O Heap Sort ordena in-place, requerendo apenas $O(1)$ espaço auxiliar, ao contrário do Merge Sort que necessita $O(N)$ espaço adicional. Esta característica é essencial para sistemas com restrições de memória.

Conclusão Final

Os resultados práticos confirmam inequivocamente a eficiência teórica do Heap Sort. A estrutura de árvore binária heap não apenas garante a complexidade $O(N \log N)$, mas também proporciona estabilidade de desempenho independente da distribuição dos dados de entrada. Para aplicações que processam grandes volumes de dados e requerem garantias de desempenho previsível, o Heap Sort representa uma escolha robusta e confiável, especialmente quando comparado a alternativas com complexidade quadrática ou que não garantem pior caso logarítmico.

4. Arquivos Entregues

4.1 Código-Fonte Principal

- **heap-sort.c** - Implementação completa do algoritmo Heap Sort em linguagem C, incluindo:
  - Função `max_heapify()` para organização de subárvores
  - Função `build_max_heap()` para construção da Max-Heap
  - Função `heap_sort()` principal do algoritmo
  - Demonstração do processo heapify com o vetor exemplo
  - Medição automática de desempenho para diferentes tamanhos de entrada
  - Contadores de comparações e tempo de execução
  - Geração automática do arquivo CSV com resultados

4.2 Geração de Gráficos

- gerar-graficos.py - Script Python para criação automática dos gráficos de análise de desempenho:
  - Leitura do arquivo CSV gerado pelo programa C
  - Gráfico de tempo de execução vs. tamanho do array
  - Gráfico de número de comparações vs. tamanho do array
  - Comparação entre complexidade teórica O(N log N) e resultados práticos
  - Estatísticas detalhadas de desempenho
  - Salvamento dos gráficos em formato PNG de alta resolução

4.3 Arquivos Gerados Automaticamente

- **resultados-heap-sort.csv** - Arquivo CSV contendo os resultados dos testes de desempenho
- **graficos-heap-sort.png** - Imagem com os gráficos de análise visual do desempenho

4.4 Instruções de Execução

Para compilar e executar o programa principal:

```
gcc heap-sort.c -o heap-sort -lm
./heap-sort
```

O programa C irá:

1. Executar a demonstração do processo heapify com o vetor exemplo
2. Realizar os testes de desempenho para N = 10.000, 50.000 e 100.000
3. Gerar automaticamente o arquivo `resultados-heap-sort.csv`

Para gerar os gráficos de análise:

```
python3 gerar-graficos.py
```

O script Python irá:

1. Ler os dados do arquivo `resultados-heap-sort.csv`
2. Gerar os gráficos de desempenho
3. Salvar a imagem `graficos-heap-sort.png`
4. Exibir estatísticas detalhadas no terminal

O programa realizará automaticamente a demonstração do processo heapify e os testes de desempenho, exibindo todos os resultados necessários para a análise completa do algoritmo.