Trabalho Prático 3 - Estrutura de Dados

Lucas Ribeiro da Silva - 2022055564
Universidade Federal de Minas Gerais
Belo Horizonte - Minas Gerais - Brasil
lucasrsilvak@ufmg.br

1 Introdução

O problema proposto foi desenvolver, dada diferentes localizações de estações dentro da cidade de Belo Horizonte, desenvolver um aplicativo que retornasse a informação de quais estações ativas há mais próximas. O problema forneceu uma implementação anterior *'naive'* que funciona em tempo não-otimizado e requisitou a necessidade de otimizar o sistema para funcionamento em tempo sublinear, implementando uma Estrutura de Dados própria para dados de geolocalização, as **QuadTrees**.

O problema lidará com as operações de inserção e mostrará o tempo de inserção, ativação e busca dos comandos executados demonstrando as otimizações implementadas e as estruturas de dados utilizadas.

2 Método

2.1 Configurações de Máquina e Ambiente

O programa foi desenvolvido em C++ e compilado com G++ da GNU Compiler Collection. O ambiente de desenvolvimento utilizado foi:

- Windows 11 Home Single Language
- WSL 2 com Ubuntu 22.04.4 LTS

2.2 Estrutura de Dados

2.2.1 Estação

A Classe Estação foi implementada para satisfazer a necessidade de armazenar as informações da Estação, ou seja seu endereço e as informações do endereço e também a informação se a estação está ativa ou não.

2.2.2 Par

Uma classe Par foi criada para armazenar as informações das Estações com as suas distâncias para adicionar e ordenar informações sobre as estações na Heap.

```
1     Par(Estacao Ponto, double Distancia);
```

2.2.3 Heap

A Estrutura de Dados Heap foi implementada na forma de um max-heap para satisfazer a necessidade de encontrar e armazenar as k estações mais próximas enquanto percorremos a QuadTree.

2.2.4 Hash

A Estrutura de Dados Tabela Hash foi implementada para satisfazer a necessidade de armazenar o Index das estações na QuadTree e permitir o acesso em tempo constante. A Hash implementada salva o ID e o Index numa struct HashElemento que permite encontrar o Index a partir do ID.

2.2.5 QuadTree

A Estrutura de Dados QuadTree foi implementada para satisfazer a necessidade de armazenar as estações e permitir que a procura pelas estações mais próximas seja realizada com eficiência. A QuadTree implementada é uma **Point QuadTree vetorizada**, que armazena a estação contida, e subdivide a QuadTree no ponto da estação, guardando as informações da região delimitada por cada quadrante, assim como um ponteiro para os quadrantes subjacentes: nordeste, sudeste, sudoeste e noroeste. A escolha da implementação pela Point QuadTree foi decidida pela possibilidade de armazenar a memória de maneira eficiente de acordo com o número de estações.

3 Análise de Complexidade

3.1 Tempo

3.1.1 Hash

Hash() O construtor da tabela Hash aloca um vetor com uma struct Hash e os pré-seta, logo opera em O(n)

~Hash() O destrutor da Hash apenas libera a memória alocada e opera em O(1).

Inserir() O Inserir da tabela Hash opera em O(n) no pior caso em que há muitas colisões, e O(1) no caso médio.

Procurar() O Procurar opera em O(n) no pior caso em que há muitas colisões, e O(1) no caso médio.

3.1.2 Estação

Estacao() O construtor da Estacao tem todas as operações em tempo constante e opera em O(1).

∼Estacao() O destrutor da Estacao simplesmente desaloca a memória e por isso opera em O(1).

Ativar() O Ativar simplesmente muda o booleano Ativo e opera em O(1).

Desativar() O Desativar simplesmente muda o booleano Ativo e opera em O(1).

3.1.3 Heap

Heap() O construtor da Heap tem todas as operações em tempo constante e opera em O(1).

∼Heap() O destrutor da Heap simplesmente desaloca a memória e por isso opera em O(1).

HeapifyPorCima() HeapifyPorCima tem seu pior caso quando o nó é movido da folha até a raiz, nesse caso, o algoritmo opera em O(log n).

HeapifyPorBaixo() HeapifyPorBaixo tem seu pior caso quando o nó é movido da raiz até a folha, nesse caso, o algoritmo opera em O(log n).

Inserir() Inserção na Heap insere um elemento no vetor da Heap e depois chama o método HeapifyPorCima() que opera em O(log(n)), sendo essa a complexidade também do Inserir().

Remover() Inserção na Heap remove um elemento no vetor da Heap e depois chama o método HeapifyPorBaixo() para reorganizar o vetor, mas ele opera em O(log(n)), sendo essa a complexidade também do método Remover().

GetElemento() A obtenção de um elemento da Heap opera em tempo constante O(1).

GetTamanho() Obter quantos elementos tem na Heap opera em tempo constante O(1).

Inverter() O método de Inverter inverte a ordem da Heap e implementa um HeapSort, sendo a complexidade do HeapSort O(nlog(n)) esta é também a complexidade desse método.

Vazio() A Verificação se a Heap está Vazia opera em tempo constante O(1).

3.1.4 QuadTree

QuadTree() O construtor da QuadTree tem todas as operações em tempo constante e opera em O(1).

∼QuadTree() O destrutor da QuadTree tem que desalocar cada um dos nós nós e por isso opera em O(n).

Inserir() Inserir chama o InserirInterno, que segue a inserção numa árvore normal e usualmente opera em tempo sublinear O(log n).

Procurar() Procurar chama o método ProcurarInterno, que percorrerá em seu pior caso a árvore inteira, operando em O(n). Entretanto, devido a uma verificação de poda que remove os galhos que não podem ser utilizados, é mais provável que o método funcione em O(log n). Depois de percorrer a árvore, o ProcurarInterno ainda jogará os nós para dentro de uma Heap de tamanho constante predefinido k e operará em O(log k) para inserção na Heap. Logo, a complexidade média é sublinear e O(log n * log k) e o pior caso é O(n log k), onde é necessário visitar todos os nós da árvore. Há também uma execução de um HeapSort para ordenar os resultados da busca que funciona em O(k

log k), mas não influencia na complexidade final se k for constante.

Ativar() O método Ativar procura a estação através do index dela na QuadTree e opera em O(1).

Desativar() O método Ativar procura a estação através do index dela na QuadTree e opera em O(1).

3.2 Espaço

3.2.1 Hash

Hash() O construtor da tabela Hash aloca um vetor e opera em O(n)

∼Hash() O destrutor da Hash apenas libera a memória alocada e opera em O(1).

Inserir() O Inserir da tabela não aloca memória adicional e opera em O(1).

Procurar() O Procurar da tabela não aloca memória adicional e opera em O(1).

3.2.2 Estação

Estacao() O construtor da Estacao tem todas as operações em tempo constante e opera em O(1).

∼Estacao() O destrutor da Estacao simplesmente desaloca a memória e por isso opera em O(1).

Ativar() Ativar não aloca memória extra e opera em O(1).

Desativar() Desativar não aloca memória extra e opera em O(1).

3.2.3 Heap

Heap() O construtor da Heap aloca n espaços de memória e por isso opera em O(n).

∼Heap() O destrutor da Heap não aloca memória adicional e opera em O(1).

Inserir() A Inserção na Heap não aloca memória adicional se não houver redimensionamento e opera em O(1).

Remover() A Remoção na Heap não aloca memória adicional e opera em O(1).

GetElemento() A obtenção de um elemento da Heap não aloca memória adicional e opera em O(1).

GetTamanho() A obtenção do tamanho da Heap não aloca memória adicional e opera em O(1).

HeapifyPorCima() O HeapifyPorCima() não aloca memória adicional e opera em O(1).

HeapifyPorBaixo() O HeapifyPorBaixo() não aloca memória adicional e opera em O(1).

Inverter() O método de Inverter implementa o HeapSort que funciona dentro do próprio vetor, logo não é necessário alocar memória extra e opera em O(1).

Vazio() A Verificação se a Heap está vazia não aloca memória e opera em O(1).

3.2.4 QuadTree

QuadTree() O construtor da QuadTree aloca um vetor de tamanho n e opera em O(n).

∼QuadTree() O destrutor da QuadTree não aloca memória e opera em O(1).

Inserir() Inserir não aloca memória extra e opera em O(1).

Procurar() Procurar aloca uma Heap e logo opera em O(k), onde k é o tamanho constante da Heap.
Ativar() Ativar não aloca memória extra e opera em O(1).
Desativar() Desativar não aloca memória extra e opera em O(1).

4 Análise de Robustez

Para melhorar a legibilidade, métodos foram padronizados em PascalCase e variáveis foram nomeadas em português. O código segue o paradigma de Orientação a Objetos e as estruturas de dados foram implementadas com o mínimo de funções necessárias para o funcionamento, seguindo o princípio "Keep it Simple, Stupid". O Valgrind foi utilizado para verificar vazamentos de memória.

4.1 Otimizações

Para aumentar a robustez do programa e garantir a sua eficiência em tempo sublinear nos métodos em que isso foi exigido, foram utilizados diversos métodos de otimização.

4.1.1 Poda em Procurar()

A otimização da Poda em Procurar() foi definida de tal forma que, quando a distância mínima do quadrante verificado for maior que o valor que está no topo da Heap, o quadrante é desconsiderado e não é necessário verificar seus filhos, desta forma, podemos reduzir consideravelmente a ordem de complexidade do algoritmo, chegando na ordem sublinear desejada.

4.1.2 Tabela Hash em Ativar() e Desativar()

A utilização de uma Tabela Hash para guardar o Index das estações na QuadTree através do ID do endereço, permite encontrar a estação através do seu ID e permite que os métodos Ativar() e Desativar() sejam realizados na QuadTree sejam realizados através do Index e em tempo sublinear e possivelmente O(1).

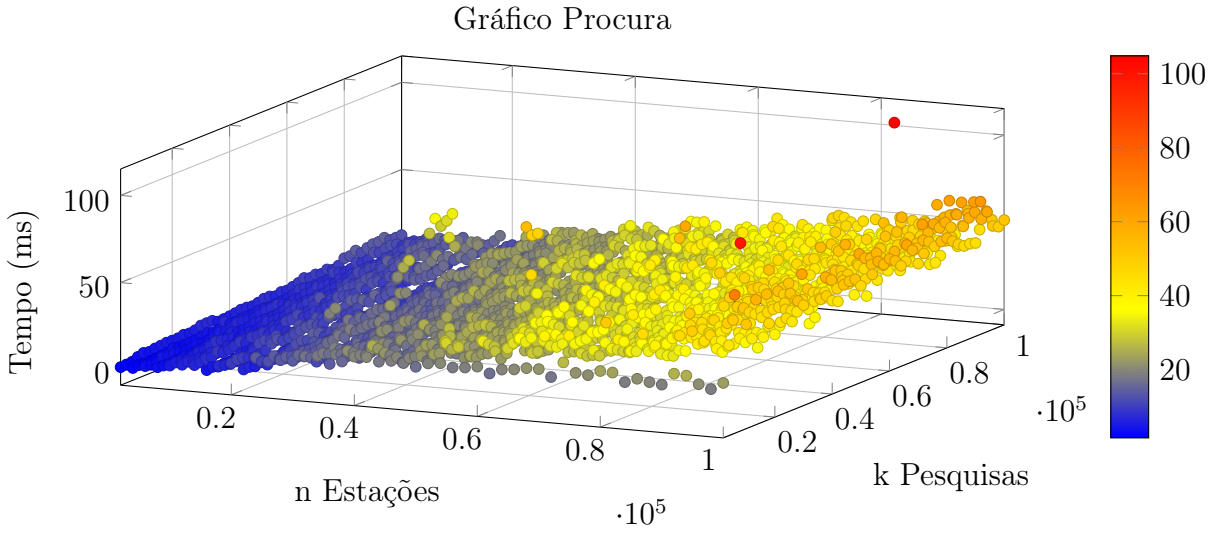
5 Análise Experimental

5.1 Complexidade Experimental

Nessa seção, faremos diversos experimentos para testar o funcionamento dos algoritmos implementados e suas ordens de complexidade.

5.1.1 Complexidade Experimental da Procura (KNN)

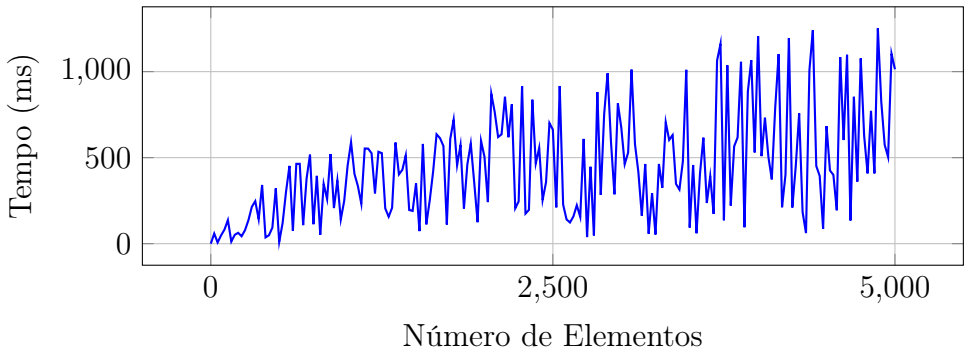
Para esse experimento, variamos o número de estações (n) de 0 a 50000 e o número de estações procuradas (k) de 50000 para comprovar a complexidade logarítmica do algoritmo de Procura.



Fica perceptível pelo gráfico a estabilidade do algoritmo e o crescimento logarítmico em relação ao número de estações enquanto o crescimento n-logarítmico em relação ao número de pesquisas, como esperado.

5.1.2 Procura com K variando

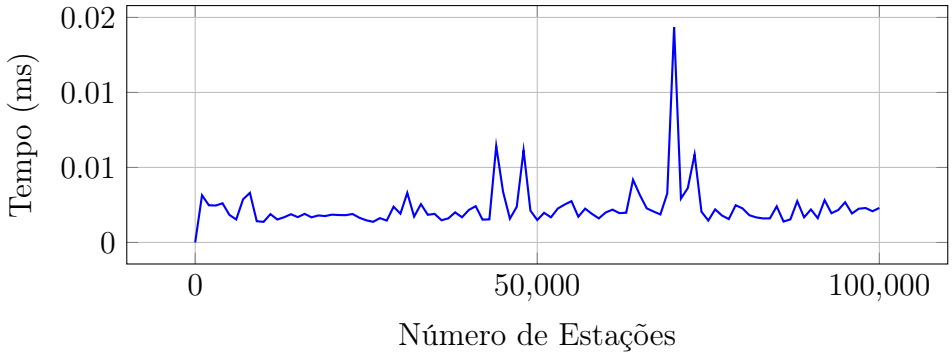
Para esse experimento, comparamos a implementação do Algoritmo de Procura fixado com 500000 estações e fazemos a procura de 0 estações até 5000.



Pelo gráfico, fica perceptível a deterioração do algoritmo conforme o número de estações a serem procuradas aumenta drasticamente, mas em alguns casos, quando a poda funciona logo de ínicio, o algoritmo encontra as estações em tempo adequado.

5.1.3 Ativar e Desativar

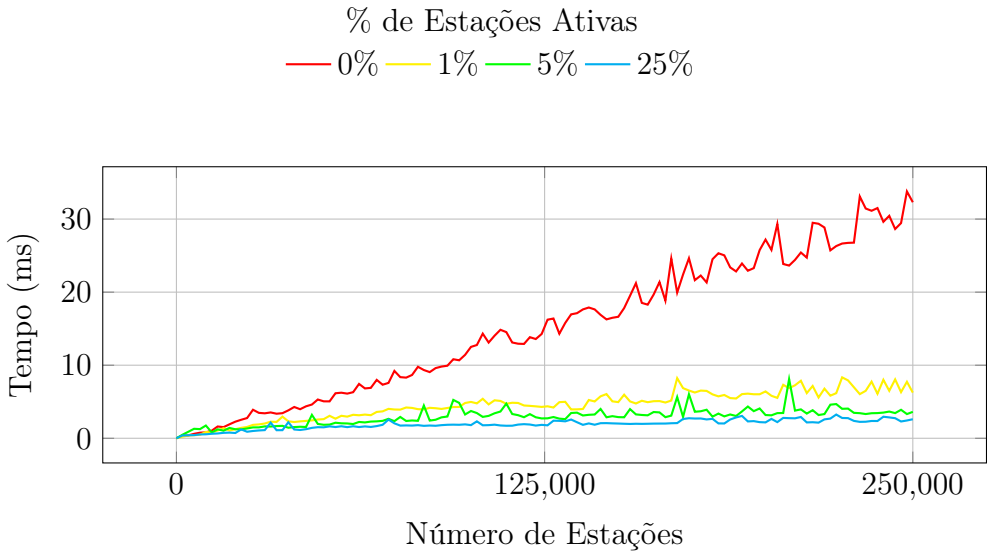
Para esse experimento, comparamos a implementação do Algoritmo de Ativação e Desativação e variamos o número de estações de 0 a 100000.



Pelo gráfico, fica perceptível que a velocidade do algoritmo praticamente independe do número de estações utilizada no programa.

5.1.4 Porcentagem de Estações Ativas

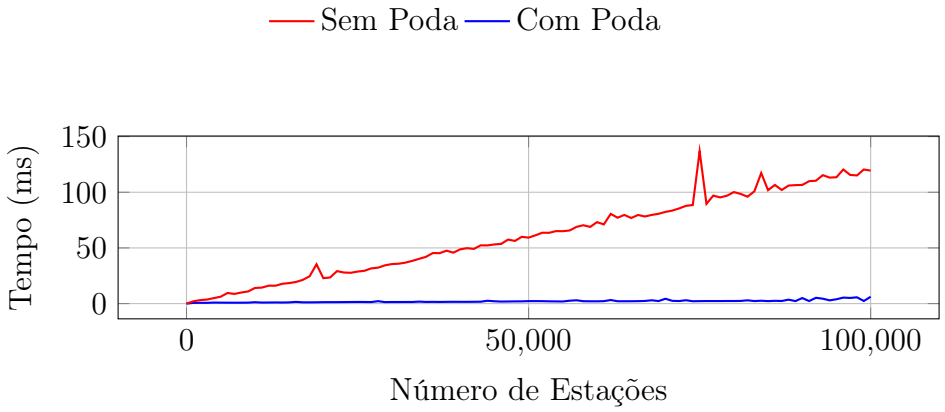
Para esse experimento, comparamos a implementação do Algoritmo de Ativação e Desativação e variamos a porcentagem de Estações Ativas.



Pelo gráfico, fica perceptível que quando não há estações ativas, o algoritmo deve percorrer toda a árvore procurando por estações, o que deixa o algoritmo no seu pior caso em relação ao número de estações, mas mesmo com 1% das estações o algoritmo já é muito mais rápido.

5.2 Análise da Poda

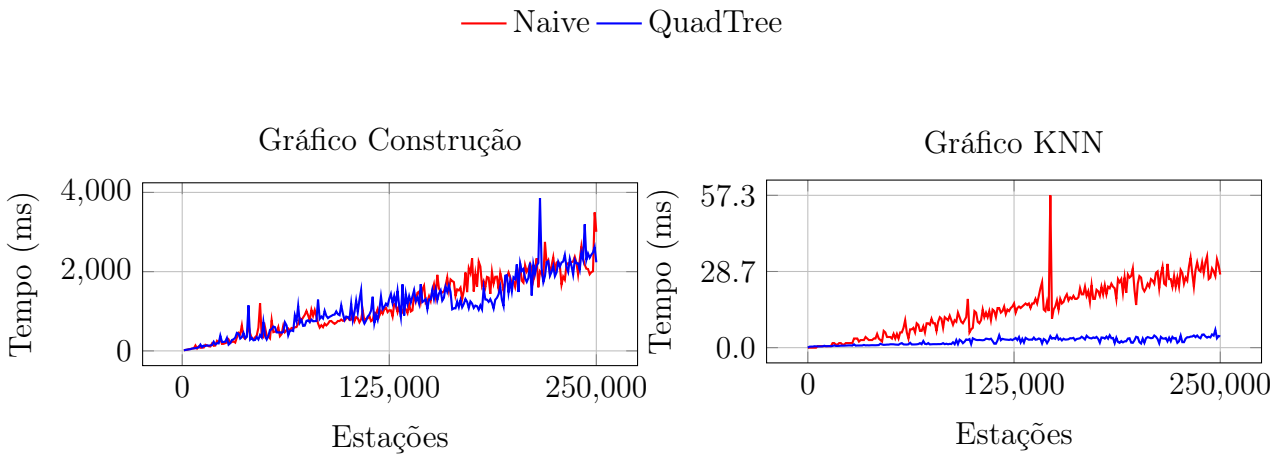
Para esse experimento, comparamos a implementação do Algoritmo de Procura com a poda e sem a poda. O número de estações procuradas são 10 e o número de estações possíveis varia de 0 a 100000.



Pelo gráfico, fica notável a diferença das implementações, com a implementação com poda sendo muito superior a implementação sem poda, podemos também observar o comportamento linear do gráfico sem poda, enquanto podemos observar um comportamento logarítmico no gráfico com poda. O algoritmo implementado sem a poda seria ineficiente para a solução do problema.

5.3 Naive x Otimizado

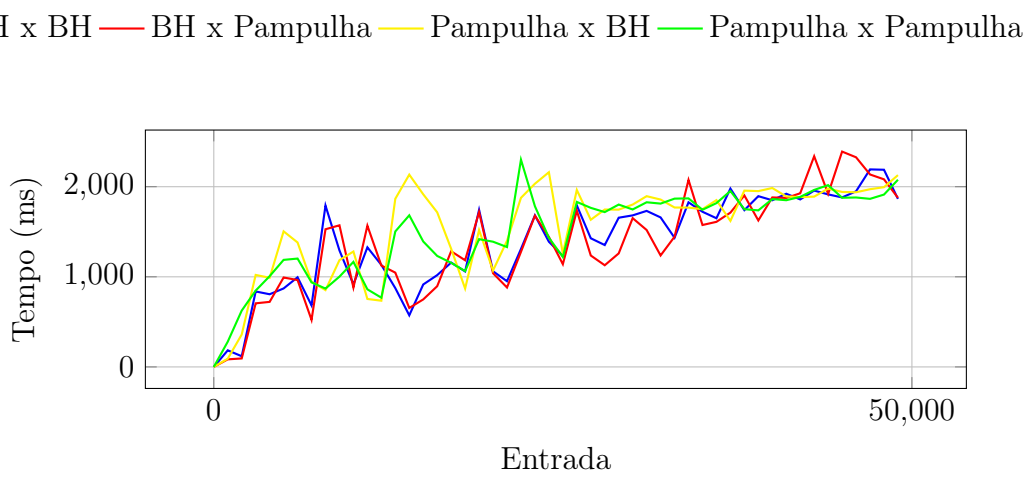
Para esse experimento, comparamos a execução do Algoritmo de Procura do o algoritmo Naive e com o algoritmo Otimizado. O número de estações procuradas são 10 e o número de estações possíveis varia de 0 a 500000.



Pelo gráfico, fica notável a diferença das implementações, com a implementação otimizada sendo similar a implementação *naive* no tempo de construção e muito superior no tempo de procura, com suas complexidades assintóticas mais eficientes.

5.4 Densidade Espacial

Para esse teste, utilizamos 50 mil estações na Pampulha e 50 mil aleatórias e 50 comandos de procura variando de 0 a 50000 estações, foram utilizadas de maneira cruzada tal que as procuras fossem feitas da seguinte maneira: Comandos em A x Estações em B



Podemos perceber que a complexidade assintótica permanece a mesma conforme o número de elementos a serem procurados aumenta, portanto a densidade espacial não influenciou nos testes.

5.5 Localidade de Referência

Para esse experimento, utilizamos os arquivos disponibilizados no moodle para o TP3 *geracarga.base* e *geracarga.ev* e utilizamos a ferramenta Valgrind com o comando –tool=cachegrind

Tabela 1: Localidade de Referência

Categoria	**Referências**	**Misses**
Instrução (I1)	3,748,258	12,560
I1 Miss Rate		0.34%
LLi Miss Rate		0.08%
Dados (D1)	1,391,069	16,883
D1 Miss Rate		1.2%
LLd Miss Rate		0.7%
Linha de Cache (LL)	29,443	13,209
LL Miss Rate		0.3%

6 Conclusões

O Trabalho Prático permitiu através do contexto da implementação de um aplicativo na cidade de Belo Horizonte uma exploração e otimização do algoritmo pré-definido *naive* e a utilização do conhecimento aprendido em sala para definir a melhor Estrutura de Dados para melhorar a eficiência dos comandos executados no programa para uma ordem de complexidade sublinear.

No caso do programa com dados georeferenciados, o projeto permitiu a exploração das tabelas Hashs e da QuadTree e suas implementações, assim como a possibilidade de explorar alternativas de trade-off entre memória e complexidade, além das heurísticas para a melhoria do funcionamento do sistema.

7 Bibliografia

Referências

[1] Chaimowicz, L. and Prates, R. (2020). Slides da Disciplina de Estruturas de Dados, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte. Disponível em: `https://virtual.ufmg.br/`

[2] Cormen, T., Leiserson, C., Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009. Versão Traduzida: Algoritmos – Teoria e Prática 3a. Edição, Elsevier, 2012