# Trabalho Prático 3

Estrutura de Dados 2024/1

Felipe Dias de Souza Martins - 2023002073
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
felipedsm@ufmg.br

# 1. Introdução

Este trabalho tem como objetivo desenvolver um sistema eficiente para a gestão e consulta de pontos de recarga de veículos elétricos, utilizando uma estrutura de dados avançada chamada Quadtree. A Quadtree é especialmente útil para armazenar e manipular dados espaciais, como coordenadas geográficas, devido à sua capacidade de dividir o espaço em regiões menores de forma hierárquica. A implementação deste projeto é feita na linguagem de programação C + +, que oferece alto desempenho e controle sobre a memória, características essenciais para a manipulação de grandes volumes de dados. O sistema desenvolvido é capaz de ler os dados de estações de recarga a partir de um arquivo de base, organizá-los em uma Quadtree e realizar consultas para encontrar as estações mais próximas de uma dada localização. Além disso, permite a ativação e desativação dinâmica das estações, refletindo mudanças em tempo real no estado das estações de recarga. O programa também foi projetado para processar comandos fornecidos em um arquivo de eventos, onde diferentes operações, como consultas e atualizações, são executadas de acordo com as instruções presentes no arquivo.

# 2. Métodos

## 2.1 Configurações do Computador

O programa foi desenvolvido utilizando a linguagem C++, compilado pelo G++, em um notebook, modelo ideapad S145 da Lenovo, com um processador intel i5, 8gb de memória ram, utilizando WSL 2 com o Visual Studio Code.

## 2.2 Bibliotecas

Neste trabalho, foi necessário utilizar várias bibliotecas do C++.

<fstream>:

Fornece classes e funções para manipulação de arquivos, permitindo a leitura e gravação de dados em arquivos.
Utilizada para abrir, ler e fechar os arquivos de base e de eventos. A leitura das linhas dos arquivos é feita através de objetos std::ifstream.

<sstream>:

Oferece classes para manipulação de strings como fluxos de entrada e saída, facilitando a conversão e extração de dados de strings.
Usada para processar e extrair os dados de cada linha lida dos arquivos. Com std::stringstream, as partes das linhas são separadas e convertidas para os tipos de dados adequados (como int, double, etc.).

<iostream>:

Fornece objetos e funções padrão para operações de entrada e saída (I/O) de dados.

Utilizada para exibir mensagens e resultados no console, como para informar o sucesso ou falha na abertura de arquivos e para mostrar as saídas das operações realizadas, como consultas e ativações/desativações de estações.

<iomanip>:

Fornece manipuladores para formatar a entrada e saída de dados, especialmente relacionado a números.
Empregada para definir a precisão das saídas numéricas. O manipulador std::setprecision é usado para ajustar o número de casas decimais exibidas em coordenadas e distâncias.

<cstring>:

Oferece funções para manipulação de strings em estilo C, como strcpy, strcmp, entre outras.
Utilizada para copiar e comparar strings que representam identificadores e outras informações textuais no formato de C-strings, que são necessárias para lidar com os dados de forma eficiente.

## 2.3 Estrutura de Dados

A **Quadtree** é a espinha dorsal do sistema de estações de recarga, fornecendo uma estrutura eficiente para a organização espacial dos pontos. Com ela, o sistema pode responder rapidamente a consultas de proximidade, realizar operações de inserção e remoção sem comprometer a performance, e gerenciar grandes volumes de dados de forma eficaz, contudo, foi necessário implementar diversas outras funções que auxiliam a Quadtree, que serão especificadas abaixo:

**RechargeStation:**

Essa estrutura armazena as informações pedidas no enunciado, como as coordenadas x e y, cep, endereço por escrito, entre outras. A única alteração foi a criação de um booleano para saber se o ponto está ativado ou não.

**Point:**

Estrutura utilizada para representar um ponto de forma mais intuitiva no plano cartesiano ($R^2$) com dois valores double representando as coordenadas x e y.

**PoitID:**

É uma estrutura composta que armazena um Point junto com um identificador (id) de tipo string.

**DistPair:**

É uma estrutura que armazena um par de valores: um id (identificador da estação) e uma dist (distância entre a estação e um ponto alvo), ambos necessários para as operações de busca de vizinhos mais próximos (KNN).

Além disso, para atender as especificações do trabalho, utilizei de dois conceitos que foram abordados em sala de aula, a **Tabela Hash** e o **Heap**.

A **Tabela Hash** foi escolhida devido à necessidade crítica de ativar ou desativar estações de recarga de forma rápida e eficiente. Como o sistema deve gerenciar um grande número de estações e permitir mudanças rápidas no status dessas estações, ela fornece a solução ideal, permitindo que essas operações sejam realizadas quase instantaneamente.

E o **Heap** foi responsável por realizar a busca por vizinhos mais próximos no sistema, especialmente para localizar as estações de recarga mais próximas de um veículo elétrico. **O MaxHeap** foi escolhido porque permite que o sistema mantenha e atualize eficientemente a lista dos k vizinhos mais próximos, garantindo que as respostas sejam precisas e rápidas.

.

## 2.4 Métodos

Para a execução do código, pensando em superar os desafios propostos, foram implementados, no total, mais de vinet métodos, porém, os mais importantes são os métodos da **QuadTree**, **MaxHeap e Hash**, que serão brevemente explicados abaixo:

***Principais métodos da QuadTree***

insert(PointID p):
**Descrição**: Este método insere um novo ponto na Quadtree. A inserção é feita recursivamente, dividindo o espaço em quadrantes até que o ponto seja inserido em uma folha apropriada. Garante que o ponto seja armazenado na área correta da Quadtree, facilitando futuras consultas e operações.

activate(Point p, std::string id):
**Descrição**: Ativa um ponto específico na Quadtree, marcando-o como disponível ou utilizável. Isso envolve a busca do ponto pela Quadtree e a alteração do estado do nó correspondente. Permite que o sistema marque uma estação de recarga como ativa, tornando-a disponível para consultas de proximidade.

knn(Point target, int k):
**Descrição**: Este método busca os k pontos mais próximos ao ponto alvo target. A busca é feita explorando recursivamente os quadrantes da Quadtree e utilizando um MaxHeap para rastrear os k vizinhos mais próximos durante a busca. É fundamental para determinar quais estações de recarga estão mais próximas de um veículo elétrico, um aspecto central do sistema.

***Principais Métodos da Tabela Hash***

insert(std::string id, Point p):

**Descrição:** Insere um novo par (id, p) na Tabela Hash. A chave id é utilizada para calcular o índice de hash, e o ponto p é armazenado nesse índice, tratando colisões se necessário. Este método permite que novas estações de recarga sejam rapidamente adicionadas à Tabela Hash, garantindo que possam ser acessadas de forma eficiente.

search(std::string id):
**Descrição:** Busca o ponto p associado ao ID fornecido. Se o ID existir na Tabela Hash, o método retorna o ponto correspondente; caso contrário, retorna um valor nulo ou erro. Proporciona acesso rápido aos dados de uma estação de recarga, fundamental para operações de ativação, desativação e consultas.

remove(std::string id):
**Descrição**: Remove o par (id, p) da Tabela Hash. O ID é utilizado para localizar o ponto, que é então removido, possivelmente tratando colisões que possam ter ocorrido. Permite a exclusão de estações de recarga da Tabela Hash quando elas são removidas do sistema.

***Principais Métodos do MaxHeap***

enqueue(DistPair dp):
**Descrição**: Insere um novo par (id, dist) no MaxHeap. O heap é então reorganizado para garantir que a maior distância esteja na raiz. Este método permite que o MaxHeap rastreie os vizinhos mais próximos durante a busca KNN, substituindo o vizinho mais distante quando um mais próximo é encontrado.

getSize():
**Descrição:** Retorna o número de elementos atualmente no MaxHeap. Utilizado para verificar se o MaxHeap já atingiu o tamanho k, o que é crucial durante a busca KNN para decidir se novos elementos devem substituir os existentes.

## 3. Análise de Complexidade

Para analisar a complexidade do programa, é necessário pontuar todas as principais estruturas e métodos dos dados, sendo assim, selecionei as três estruturas principais, e o quicksort, algoritmo também já visto em sala de aula, que auxiliou a selecionar os pontos mais próximos.

### 3.1 QuadTree

***Inserção de Pontos:***

- **Complexidade:** O(log n)
- A inserção em uma Quadtree envolve a divisão do espaço e o posicionamento do ponto no quadrante correto. Devido à natureza de divisão em quadrantes, a inserção segue uma profundidade

logarítmica em relação ao número de pontos, resultando em **O(log n).**

***Ativação e Desativação de Pontos:***

- **Complexidade:** O(log n)
- Ativar ou desativar um ponto na Quadtree envolve localizar o ponto e alterar seu estado. A localização do ponto segue a estrutura hierárquica da Quadtree, o que é feito em **O(log n).**

***Consultas de Vizinhos (KNN):***

- **Complexidade:** O(k log n)
- A busca dos k vizinhos mais próximos envolve explorar a Quadtree e utilizar um MaxHeap para manter os k vizinhos mais próximos encontrados. A complexidade total inclui explorar a árvore **(O(log n))** e manter a lista dos k vizinhos (**O(log k)** por inserção), resultando em **O(k log n).**

***Conclusão*:**
A complexidade é, portanto, **O(3logn+klogn) = O(klogn)**

## 3.2 Tabela Hash

***Inserção:***

- **Complexidade:** O(1) (em média)
- A inserção em uma tabela hash é realizada em tempo constante na média, utilizando a função de hash para determinar a posição de inserção. Colisões são tratadas usando endereçamento aberto quadrático, que ainda mantém a complexidade média em **O(1)**.

***Remoção:***

- **Complexidade:** O(1) (em média)
- A remoção de um item da tabela hash segue a mesma lógica da inserção e busca, sendo feita em tempo constante **O(1**) na média, com o tratamento de colisões mantido

***Busca:***

- **Complexidade:** O(1) (em média)
- A busca por um item na tabela hash é feita em tempo constante na média, através da função de hash que determina a posição do item. O tratamento de colisões é eficiente e também mantém a complexidade média em **O(1).**

***Conclusão:***

Por tanto, na média, é que a tabela hash tem complexidade constante;

## 3.3 MaxHeap

***Inserção e Remoção:***

- **Complexidade:** O(log k)
- Inserir um novo elemento no MaxHeap requer a reorganização do heap para manter a propriedade de heap. Essa operação é feita em **O(log k**), onde k é o número de elementos no heap.O mesmo acontece para remover o maior elemento (raiz) e reorganizar o heap é feito em **O(log k)**. A reorganização garante que a propriedade do heap seja mantida após a remoção

***Acesso ao Topo:***

- **Complexidade**: O(1)
- A operação de acessar o elemento do topo (maior valor) é feita em tempo constante O(1), pois o topo do MaxHeap é sempre o maior elemento.

***Conclusão:***

Dito isso, partindo da soma das complexidades, é visto que a complexidade do MaxHeap é **O(log k).**

### 3.4 QuickSort

***Ordenação no Pior Caso e no Caso Médio***

- **Complexidade**: O(n log n) (em média), O($n^2$) (no pior caso)

## 4. Estratégia de Robustez

Uma das estratégias de robustez utilizadas foi a de implementar testes de unidade nos métodos. Os testes funcionam para simular entradas e funções nas tabelas hash, quadtree e max heap, além disso, a fim de de organizar melhor as alocações e nas deslocações, fiz questão de comentar e isolar as partes mencionadas, com isso, os erros que viriam com o código de segmentação e memória se tornaram muito mais previsíveis para futuras correções. Além disso, também foi necessário verificar se as entradas correspondem a números positivos, deixando o código mais resistente a entradas inválidas, fazendo com que o programa encerre.

Outra estratégia foi utilizar as flags, essas que solicitam os arquivos de leitura, fazendo com que o programa possa ser testado em diferentes máquinas e arquivos, desde que sigam as especificações de formatação.

Além disso, foram criados métodos auxiliares para que a inserção da Quadtree e ordenação do quicksort fosse feita de forma mais eficiente, utilizando recursividade e apontadores.

## 5. Análise Experimental

Para as análises experimentais foram utilizadas duas bibliotecas de C++, uma é responsável por calcular o tempo, chamada “chrono”, a outra foi usada para medir o gasto de memória, chamada “sys/resource.h”.

É importante citar que o programa disponibilizado pelo professor (naive) só analisa até 10 pontos mais próximos, sem a possibilidade de

aumentar ou fazer ativações e desativações, portanto, para a análise, irei comparar o programa novo com o antigo somente nas condições que o antigo consegue operar, posteriormente, analisarei isoladamente o aplicativo novo.

## 5.1 Análise do Naive

**Estrutura de Dados**: O programa utiliza vetores e cálculo direto para encontrar os pontos mais próximos. Essa abordagem pode ser ineficiente em termos de tempo quando se lida com grandes quantidades de dados.

**Complexidade de Tempo e Memória**:

**O(n log k)**: Se for necessário ordenar todos os pontos por distância e então pegar os 10 mais próximos, a complexidade é O(n log n) para a ordenação e O(k log k) para a seleção dos k menores elementos.

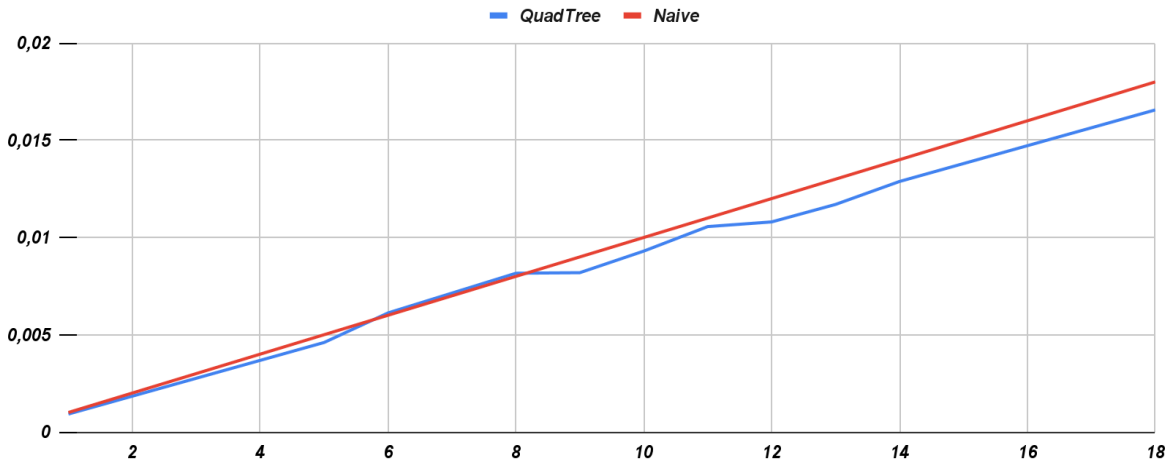
**O(n * k)**: Se for realizado um cálculo direto para cada ponto e selecionar os k menores, a complexidade pode chegar a O(n * k), onde n é o número total de pontos e k é o número de pontos mais próximos desejado, que, no caso do naive, é 10.

**Complexidade de Memória**:

**O(n):** O único espaço utilizado é o vetor de pontos de recarga, que armazena todos os n pontos.

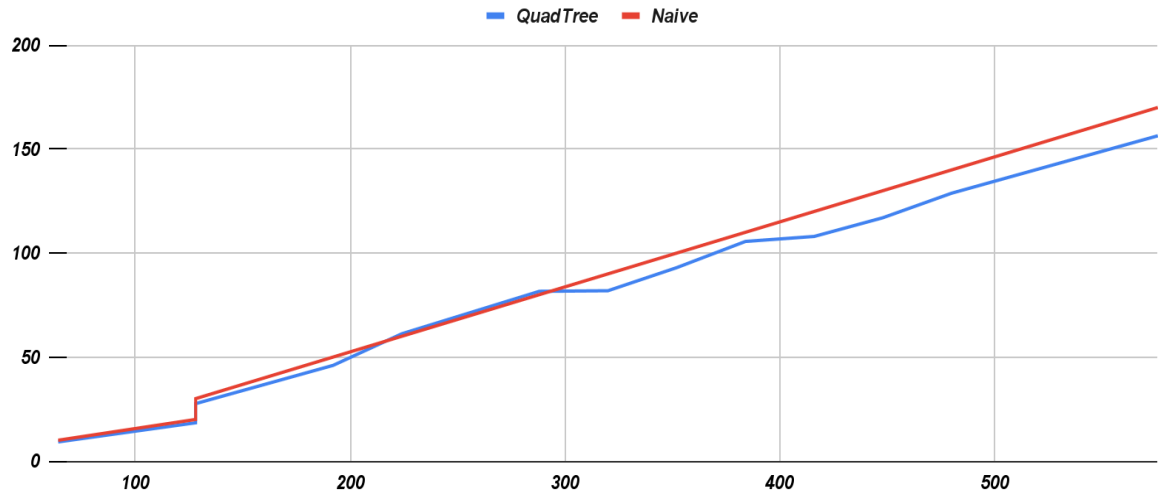
## *5.2 Comparação de Tempo*

Para a análise de tempo, utilizei o eixo X como o número de consultas e Y para os milissegundos.




## *5.3 Comparação de Gasto de Memória*

Nesta análise, o eixo X representa o valor em KiloBytes e o Y as consultas




## *5.4 Comparação Entre os Aplicativos*

A principal desvantagem do naive é sua escalabilidade. À medida que o número de pontos de recarga e o número de consultas aumentam, o tempo de execução cresce rapidamente, tornando o algoritmo impraticável para grandes conjuntos de dados, visto que ele analisa todos os pontos de recarga para encontrar os mais próximos. Esse processo é repetido para cada consulta, o que leva a uma complexidade total proporcional ao número de consultas vezes o número de pontos de recarga. Esse crescimento quadrático torna o naive ineficiente para um grande número de consultas ou pontos de recarga.

Enquanto que o consumo de memória do naive é relativamente baixo, pois ele não utiliza estruturas de dados complexas. Sua memória é essencialmente constante em relação ao número de consultas, uma vez que ele só precisa armazenar os dados de entrada (pontos de recarga) e variáveis temporárias para processar cada consulta.Com o aumento do número de consultas, o tempo total de processamento se torna muito elevado, resultando em um desempenho significativamente pior.

O aplicativo novo é significativamente mais eficiente em termos de tempo, especialmente com um grande número de pontos de recarga e consultas. A estrutura de dados quadtree reduz a quantidade de comparações necessárias, e o heap otimiza a extração dos pontos mais próximos.Ele também é mais escalável e pode lidar com grandes volumes de dados e consultas de forma mais eficiente.

## *5.5 Análise Isolada*

Análise isolada do projeto desenvolvido.

| ***QUANTIDADE DE ENDEREÇOS*** | ***NÚMERO DE CONSULTAS*** | ***TEMPO GASTO (ms)*** | ***MEMÓRIA GASTA (KB)*** |
|---|---|---|---|
| ***100*** | ***20*** | ***0,02*** | ***1640*** |
| ***100*** | ***50*** | ***0,14*** | ***1860*** |
| ***100*** | ***150*** | ***0,21*** | ***2000*** |

| | | | |
|---|---|---|---|
| ***100*** | ***300*** | ***0,24*** | ***2056*** |
| ***500*** | ***20*** | ***0,09*** | ***8040*** |
| ***500*** | ***50*** | ***0,7*** | ***8200*** |
| ***500*** | ***150*** | ***0,23*** | ***8400*** |
| ***500*** | ***300*** | ***0,27*** | ***8800*** |
| ***1000*** | ***20*** | ***0,11*** | ***16000*** |
| ***1000*** | ***50*** | ***0,8*** | ***16400*** |
| ***1000*** | ***150*** | ***0,3*** | ***168000*** |
| ***1000*** | ***300*** | ***0,38*** | ***17640*** |

Como pode ser observado, a memória gasta no novo aplicativo é bem elevada, principalmente quando o número de linhas a serem lidas (endereços) aumentam, isso se deve ao gasto das estruturas, que são complexas e demandam memória para serem criadas e utilizadas, contudo, a eficiência em relação ao tempo é ótima, crescendo de forma equilibrada.

## 6 Conclusões

O trabalho mostrou como uma boa escolha de estrutura de dados pode melhorar a eficiência de um projeto, além de que ele adiciona diversas funcionalidades que, pensando na vida real, seriam úteis e imprescindíveis para os usuários. O aplicativo antigo até poderia ter um desempenho levemente superior no gasto de memória em comparação com o desenvolvido no projeto, no entanto, ao se comparar o tempo, é visto que não a memória adicional vale a pena.

## 7 Bibliografia


1. XAVIER, Carlos. **Introdução aos Algoritmos**, 2012
2. ZIVIANI, N. **Projetos de Algoritmos com Implementações em Pascal e C**, 2011
3. Slides Disponibilizados pelos Professores Eder Ferreira e Wagner Meira, Disponível em *https://virtual.ufmg.br/20241/course/view.php?id=10097*