

# ChargeCraft

## Rapport Technique

Structures de Données pour la Gestion de Bornes de Recharge

## 1 Introduction

Ce document analyse l'architecture et les choix algorithmiques du système ChargeCraft, une plateforme de gestion intelligente de bornes de recharge pour véhicules électriques. L'accent est mis sur la justification des structures de données employées et l'analyse de leurs complexités temporelles.

## 2 Structures de Données Principales

### 2.1 AVL Tree (Arbre Équilibré)

#### 2.1.1 Justification du choix

L'arbre AVL a été retenu comme structure principale d'indexation des stations pour plusieurs raisons critiques :

- **Recherche rapide par identifiant** : Les requêtes fréquentes nécessitent un accès direct aux informations d'une station donnée ( $O(\log n)$ )
- **Équilibrage automatique** : Garantit des performances stables même après de nombreuses insertions/suppressions
- **Parcours ordonné** : Permet d'extraire efficacement les stations par intervalles d'identifiants
- **Scalabilité** : Performances prévisibles pour un nombre croissant de stations

#### 2.1.2 Complexités temporelles

Opération	Complexité
Insertion ( <code>si_add</code> )	$O(\log n)$
Recherche ( <code>si_find</code> )	$O(\log n)$
Suppression ( <code>si_delete</code> )	$O(\log n)$
Rééquilibrage	$O(1)$ par rotation

#### 2.1.3 Implémentation

La structure `StationNode` stocke :

```
typedef struct StationNode {
    int station_id;           // Clé d'indexation
    StationInfo info;         // Données (power, slots, price)
    struct StationNode *left, *right;
    int height;               // Pour équilibrage AVL
} StationNode;
```

L'équilibrage repose sur le facteur de balance  $bf = h_{gauche} - h_{droite}$  et des rotations simples/- doubles lorsque  $|bf| > 1$ .

## 2.2 Queue (File FIFO)

### 2.2.1 Justification du choix

La queue est utilisée pour le traitement séquentiel des événements de charge/décharge. Ce choix s'impose car :

- **Ordre temporel** : Les événements doivent être traités dans leur ordre chronologique d'arrivée
- **Simplicité** : Structure adaptée au pattern producteur-consommateur
- **Équité** : Garantit que chaque événement sera traité sans priorité arbitraire

### 2.2.2 Complexités temporelles

Opération	Complexité
Enqueue ( <code>q_enqueue</code> )	$O(1)$
Dequeue ( <code>q_dequeue</code> )	$O(1)$
Test de vide	$O(1)$

## 2.3 Stack (Pile LIFO)

### 2.3.1 Justification du choix

La pile est employée exclusivement pour l'évaluation des règles en notation postfixe (Reverse Polish Notation). Cette structure est idéale car :

- **Notation postfixe** : L'évaluation d'expressions postfixes nécessite naturellement une pile
- **Pas de parenthèses** : Simplifie l'analyse syntaxique des règles
- **Évaluation efficace** : Un seul passage sur l'expression suffit

### 2.3.2 Exemple d'évaluation

Pour la règle : `power 50 >= slots 1 >= &&`

1. Push `power_kw` de la station
2. Push 50
3. Pop 2 valeurs, évalue  $\geq$ , push résultat (0 ou 1)
4. Push `slots_free`
5. Push 1
6. Pop 2 valeurs, évalue  $\geq$ , push résultat
7. Pop 2 booléens, évalue AND logique, retourne résultat final

Complexité :  $O(m)$  où  $m$  est le nombre de tokens dans la règle.

## 2.4 Linked List (Liste Simplement Chaînée)

### 2.4.1 Justification du choix

Les listes chaînées sont utilisées pour l'historique MRU (Most Recently Used) de chaque véhicule :

- **Insertions fréquentes en tête** : Chaque tentative de connexion ajoute une station en tête de liste
- **Taille variable** : Le nombre de stations visitées par véhicule n'est pas fixe
- **Pas d'accès aléatoire** : On ne consulte que les dernières stations visitées

## 2.4.2 Complexités

Opération	Complexité
Insertion en tête	$O(1)$
Suppression d'une valeur	$O(k)$ ( $k = \text{taille liste}$ )
Suppression en queue	$O(k)$

## 3 Requêtes et Algorithmes

### 3.1 Requête par Intervalle (si\_range\_ids)

#### 3.1.1 Description

Cette fonction extrait tous les identifiants de stations dans l'intervalle  $[lo, hi]$  en exploitant la structure ordonnée de l'AVL.

#### 3.1.2 Complexité

$$O(\log n + k)$$

où :

- $n$  = nombre total de stations
- $k$  = nombre de résultats retournés

Le terme  $\log n$  correspond à la descente initiale dans l'arbre, et  $k$  au coût de collecte des résultats.

#### 3.1.3 Optimisation clé

Le parcours sélectif évite d'explorer les sous-arbres ne contenant aucune valeur dans l'intervalle :

```
if (r->station_id > lo)
    explore_left(); // Peut contenir des valeurs >= lo
if (r->station_id < hi)
    explore_right(); // Peut contenir des valeurs <= hi
```

### 3.2 Comptage par Puissance (si\_count\_ge\_power)

#### 3.2.1 Description

Compte les stations ayant une puissance  $\geq P$  kW.

#### 3.2.2 Complexité

$$O(n)$$

**Limitation importante** : Contrairement à `si_range_ids`, cette fonction ne peut pas bénéficier pleinement de la structure AVL car :

- L'AVL est indexé par `station_id`, pas par `power_kW`
- Impossible d'élaguer des branches basées sur la puissance
- Nécessite un parcours quasi-complet de l'arbre

### 3.3 Filtrage par Règles (`filter_ids_with_rule`)

#### 3.3.1 Description

Sélectionne les stations satisfaisant une règle booléenne complexe exprimée en notation postfixe.

#### 3.3.2 Stratégie d'optimisation

Deux niveaux de filtrage :

1. **Pré-filtres rapides** : Tests simples sur `power` et `slots`
2. **Règle complète** : Évaluation postfixe uniquement si les pré-filtres réussissent

#### 3.3.3 Complexité

$$O(n \cdot m)$$

où :

- $n$  = nombre de stations
- $m$  = nombre de tokens dans la règle

En pratique :  $O(k \cdot m)$  avec  $k \ll n$  grâce aux pré-filtres.

## 4 Scénario de Résilience (B2)

### 4.1 Problématique

Simuler une panne sectorielle affectant les stations [1101, 1150] tout en maintenant le système opérationnel.

### 4.2 Mécanisme de dégradation

1. **Injection de panne** : Mise à zéro de `power_kW` pour les stations du secteur
2. **Traitements résilients** : Les événements continuent d'être traités via `process_events`
3. **Reroutage automatique** : Recherche d'alternatives via `filter_ids_with_rule`
4. **Recovery** : Restauration à `power_kW = 50`

### 4.3 Logique de reroutage

Lorsqu'une station refuse un branchement :

```
// Recherche d'alternatives (power >= 50, slots >= 1)
int n = filter_ids_with_rule(idx->root, rule, 7,
                             candidates, 16, 50, 1);

// Selection de la première station rellement disponible
for (int i = 0; i < n; i++) {
    if (alt->info.power_kW > 0 && alt->info.slots_free > 0) {
        // Reroutage vers cette station
        alt->info.slots_free--;
        break;
}
```

## 5 Cohérence et Vérifications

### 5.1 Sanity Check

La fonction `sanity_check_slots` vérifie l'invariant fondamental :

$$\forall \text{ stations } s, \quad s.\text{slots\_free} \geq 0$$

Complexité :  $O(n)$  (parcours in-order complet).

### 5.2 Garanties de cohérence

- Les événements ne modifient que l'état courant dans l'AVL
- Les stations inexistantes sont créées avec des valeurs par défaut
- Les reroutages préservent la cohérence des compteurs de slots

## 6 Conclusion

Les choix structurels de ChargeCraft sont guidés par les contraintes opérationnelles :

- **AVL** : Accès rapide et scalable aux stations par ID
- **Queue** : Traitement ordonné et équitable des événements
- **Stack** : Évaluation efficace de règles complexes
- **Liste chaînée** : Historique flexible par véhicule

Le système atteint un compromis entre performance ( $O(\log n)$  pour les opérations critiques) et flexibilité (règles configurables, résilience aux pannes).