safwan.chendeb@lametice.com janvier 2026

ChargeCraft

I. Brief du projet

ChargeCraft : Noyau d'un système EV & bornes "intelligent"

Introduction

L'électromobilité explose : plus de véhicules, plus de bornes, plus de décisions… Où se recharger ? Quand ? À quelle puissance ? Et que faire quand une station est saturée ? ChargeCraft vous met dans la peau de l'équipe "Core" qui conçoit le cœur algorithmique d'un mini-système de supervision & recommandation pour bornes de recharge. L'objectif n'est pas de faire du ML, mais de bâtir des fondations solides (structures de données en C) pour un futur moteur data/ML.

Vous allez ingérer un flux d'événements (branché/débranché/panne), maintenir des historiques récents, indexer les stations en AVL pour des accès rapides, structurer la géographie en arbre n-aire, et proposer des requêtes utiles (top-N, filtrage par règles postfix). Le tout, propre, robuste, et commenté en complexité.

Travail à réaliser

1) Ingestion & modèle d'événements

• Implémenter une file FIFO (Queue) pour absorber un flux d'Event{ts, vehicle_id, station_id, action}.

• Consommer la file et maintenir l'état courant du système :

– Mise à jour du journal récent (MRU) des stations visitées par chaque véhicule via une SList head-only (insertion en tête, taille bornée).

– Mise à jour de l'index des stations (cf. point 2).

2) Index "station" en AVL (accès garantis O(log n))

• Concevoir un AVL indexé par station_id avec au minimum :

```
typedef struct StationInfo {
   int power_kW;
   int price_cents;
   int slots_free;
   int last_ts;
} StationInfo;
```

• Fournir : insert/update, find, delete, min/max, to_array (in-order) et un affichage "sideways" de l'arbre.

• Commenter systématiquement la complexité (temps/espace) de chaque opération.

3) Historiques & files locales

• MRU par véhicule (SList) : éviter doublons, évincer en queue si capacité atteinte.

• Option plus : modéliser la file d'attente locale d'une station avec une DList (pour retirer un véhicule connu en O(1) si on a son pointeur).

4) Règles de filtrage (moteur postfix)

• Implémenter une pile (Stack) et un évaluateur de règles postfix sans parsing compliqué (tokens déjà séparés).

• Exemples de règles :

```
slots 1 >= power 50 >= &&   /* au moins 1 slot libre ET ≥ 50 kW */
price 250 <=                /* prix ≤ 2,50 € */
```

• L'évaluateur reçoit un StationInfo* et retourne 1/0.

5) Hiérarchie géographique (arbre n-aire) & agrégations

• Construire une petite taxonomie Country → Region → City → StationGroup.

• Implémenter un arbre n-aire avec :

– création de nœud, attachement d'enfants (tableau dynamique),

– parcours BFS (Queue d'NNode*),

– accumulation simple (ex. somme des slots_free dans un sous-arbre).

• Imprimer la hiérarchie (BFS) et afficher quelques stats par niveau.

6) Requêtes "utiles" (démonstration)

• Top-N stations d'une zone (délimitée par un nœud de l'arbre n-aire), filtrées par une règle postfix :

1) parcourir l'AVL (in-order → tableau d'IDs),

2) filtrer par règle,

3) sélectionner N (méthode simple suffisante).

• Historique MRU de quelques véhicules (impression lisible).

• Affichages "pretty" de l'AVL et de l'arbre n-aire pour faciliter la démonstration.

7) Qualité, robustesse, complexités

• Vérifier les cas limites (listes/queues vides, suppression racine AVL, etc.).

• Commenter la complexité (Big-O) en tête de chaque fonction.

• Fournir un mini-rapport (≈1 page) comparant BST vs AVL sur des inserts défavorables (IDs triés).

Livrables attendus

• Code C11 compilable (−std=c11 −Wall −Wextra), organisé proprement :

slist.h/.c queue.h/.c stack.h/.c
station_index.h/.c (AVL)
nary.h/.c rules.c
events.h/.c main.c Makefile

• Démonstration console : ingestion d'un petit dataset, impression MRU, requêtes top-N + règle, affichage de l'AVL et de l'arbre n-aire.

• Commentaires de complexité + mini-rapport (BST vs AVL).

Critères d'évaluation (grille synthétique)

• Structures & API (Queue, SList, Stack, AVL, n-aire) correctes et robustes

• Fonctionnalités démontrées (règles, top-N, agrégations BFS, affichages)

• Qualité du code (lisibilité, modularité, commentaires, complexités)

• Démo & rapport (cohérence, explications, pertinence)

Bonus (facultatif)

• Suppression avancée en AVL + tests systématiques.

• Liste circulaire pour faire tourner une sélection de bornes candidates à l'affichage.

• Petit benchmark expérimental (compteur d'opérations) pour illustrer l'intérêt de l'AVL.

II. Addendum Données & Starter

ChargeCraft : Addendum Données & Starter

Contexte

Cet addendum introduit des jeux de données « TP-ready » (CSV & JSON) pour le projet ChargeCraft et décrit l'intégration dans le Starter (chargeurs CSV/JSON, Makefile et main mis à jour). Objectif : permettre une démonstration complète (chargement stations → insertion dans AVL → ingestion d'événements → requêtes/affichages).

Jeux de données fournis

• CSV (300 lignes) : izivia_tp_subset.csv

• JSON minimal (10 enregistrements) : izivia_tp_min.json

Colonnes CSV : id_station_itinerance, nom_operateur, nom_station, adresse_station, code_insee_commune, puissance_nominale, nbre_pdc, condition_acces, latitude, longitude.

Travail à réaliser

1) Écrire un chargeur CSV : lire chaque ligne, extraire un identifiant numérique de station depuis id_station_itinerance (ex. FRIZI_1101 → 1101), créer StationInfo et insérer/mettre à jour dans l'AVL.

2) (Option) Écrire un chargeur JSON minimal : même logique avec un petit fichier JSON fourni.

3) Démontrer : chargement des données → affichage AVL → application d'une règle postfix (ex. slots≥1 && power≥50) → MRU d'un véhicule.

Starter : contenu

• csv_loader.h/.c : charge stations depuis CSV IRVE-like

• json_loader.h/.c : charge stations depuis JSON minimal

• station_index.h/.c : AVL (insert/delete/to_array/print)

• slist.h/.c : MRU (SList head-only)

• queue.h/.c : FIFO d'Event

• stack.h/.c : pile (règles postfix) & rules.c , évaluation

• nary.h/.c : arbre n-aire (squelette + BFS print)

• events.h/.c : mini flux d'événements

• main.c : démo intégrée (chargement CSV/JSON, ingestion, affichages)

• Makefile : compile tout (C11)

Compilation & exécution

make
./ev_demo

Signatures & extraits utiles

StationInfo :

```
typedef struct StationInfo {
   int power_kW;
   int price_cents;
   int slots_free;
   int last_ts;
} StationInfo;
```

Chargeur CSV :

```
int ds_load_stations_from_csv(const char* path, StationIndex* idx);
/* parse_station_id("FRIZI_1101") -> 1101 */
```

Chargeur JSON minimal :

```
int ds_load_stations_from_json(const char* path, StationIndex* idx);
/* parsing naïf adapté à izivia_tp_min.json */
```

Règle postfix (exemple) : power>=50 && slots>=1

```
char* rule[] = { "slots","1",">=","power","50",">=","&&" };
/* filtrer les ids issus de si_to_array(...) avec eval_rule_postfix(...) */
```

Boucle démo (extrait main) :

```
int c1 = ds_load_stations_from_csv("izivia_tp_subset.csv", &idx);
int c2 = ds_load_stations_from_json("izivia_tp_min.json", &idx);
for(int i=0;i<DS_EVENTS_COUNT;i++) q_enqueue(&q, DS_EVENTS[i]);
process_events(&q, &idx);
si_print_sideways(idx.root);
```

Conseils pédagogiques

• Fixer un snapshot des données dans le dépôt de TP pour la reproductibilité.

• Commencer par le CSV (plus direct), puis le JSON (validation parsing).

• Demander la complexité (temps/espace) en commentaire sur chaque fonction.

Annexe
Choix de structures pour un système EV & bornes

Dans un prototype pédagogique, un AVL indexé par station_id est pertinent pour pratiquer les rotations, l'équilibrage et le O(log n). Côté applicatif, les requêtes métiers portent surtout sur la proximité géographique, la disponibilité instantanée et des contraintes de prix/puissance. Cette annexe propose une lecture pragmatique des structures adaptées, sans modifier le plan du module.

1) Trouver des bornes proches (recherche spatiale)

• Besoin : « autour de (lat,lon) » et k plus proches voisins.

• Structures adaptées :

– k-d tree (2D) pour k-NN exact ; partitionne l'espace.

– R-tree pour indexer des zones et accélérer les requêtes spatiales.

– Grilles hiérarchiques (geohash, H3/S2) pour voisinage approximatif très rapide et agrégations par niveau.

2) Choisir la "meilleure" borne maintenant (classement dynamique)

• Besoin : top-N selon un score dynamique (slots libres, puissance, prix, attente).

• Structure adaptée : priority queue (tas/heap) avec update (increase/decrease-key) en O(log n).

3) Accès direct et mises à jour par identifiant

• Besoin : mise à jour rapide de l'état d'une station lors d'un événement.

• Structure adaptée : table de hachage (O(1) amorti) id → record station.

4) Où l'AVL reste utile en "métier"

• En tant qu'index secondaire trié (par prix ou puissance) pour des requêtes d'intervalle :

– « prix ≤ 250 cts », « puissance ≥ 100 kW » (parcours borné en in-order).

• Vous conservez l'intérêt algorithmique (équilibrage) et répondez à un cas d'usage concret.

5) Synthèse “besoin → structure”

Besoin concret	Structure conseillée
Lookup par station_id	Hash table (chaînage)
Autour de (lat,lon) / k-NN	k-d tree / R-tree / Grille géohash/H3
Top-N meilleures bornes (score)	Priority queue (heap)
Filtrage par plages (prix, puissance)	AVL (index secondaire trié)
Ingestion d’événements	Queue FIFO
Historique récent par véhicule	SList (MRU head-only)

6) Recommandation pratique

Conserver l’AVL pour l’apprentissage, mais l’illustrer côté métier comme index secondaire trié (prix ou puissance). Mentionner qu’un système de production combinerait typiquement : HashMap (MAJ par id), index spatial (proximité) et Heap (classement dynamique), en plus des structures déjà travaillées (Queue, SList).

7) Pistes bibliographiques

• Cormen, Leiserson, Rivest, Stein : Introduction to Algorithms (MIT Press).

• Samet : Foundations of Multidimensional and Metric Data Structures (Morgan Kaufmann).

• Preparata & Shamos : Computational Geometry (Springer).

• Knuth : The Art of Computer Programming, Vol. 3 (Sorting and Searching).