Cahier des charges
Démarche & Extensions (ChargeCraft)

Message IMPORTANT

Si vous avez bien exploré le starter livré, vous avez sans doute constaté qu'il est bien garni : il compile, démontre un flux d'événements, structure les données et propose une base solide. J'espère que vous avez pris le temps de le tester (cas nominaux et limites) et de vous l'approprier : comprendre l'architecture, les API, et la logique des structures.

1) Objet et positionnement

Ce document est un complément au sujet initial et au starter déjà fournis. Il ne modifie pas le sujet : il précise la démarche attendue après la prise en main du starter, et définit des fonctionnalités avancées (modèles) et des scénarios applicatifs à réaliser pour valoriser la maîtrise des structures et l'algorithmique.

2) Rappel : Contenu du starter (résumé)

• Données : izivia_tp_subset.csv (300 lignes), izivia_tp_min.json (10 enregistrements).

• Structures : SList (MRU), Queue (Event), Stack (règles postfix), AVL (index stations), N-ary (squelette).

• Moteur de règles postfix (pile) : opérateurs >=, <=, >, <, ==, &&, ||.

• Démonstration : chargement CSV/JSON, ingestion d'événements, affichages de base.

3) Démarche attendue

1) Maîtriser le starter : exécuter, lire, et comprendre l'enchaînement données → structures → affichages.

2) Mettre en place au moins deux **modules avancés** parmi le menu ci-dessous (section 4).

3) Démontrer un **scénario applicatif** complet (section 5).

4) Soigner un minimum la **qualité** (messages clairs, libérations mémoire, complexité en commentaire).

4) Modèles de fonctions avancées (menu A : choisissez 2 à 3)

A1) Range queries sur AVL

But : extraire rapidement des IDs dans un intervalle, ou compter par contrainte simple.

```
int si_range_ids(StationNode* r, int lo, int hi, int* out, int cap);
// retourne le nb écrit
int si_count_ge_power(StationNode* r, int P);
// compte stations avec power>=P
/* Idée : parcours in-order borné (couper les branches inutiles).
   Complexité typique : O(k + log n), k = nb d'éléments retournés. */
```

A2) Top-K par score (min-heap local)

But : classer les stations selon un score composite (slots, puissance, prix).

```
int si_top_k_by_score(StationNode* r, int k, int* out_ids,
                      int alpha, int beta, int gamma);
/* score = slots_free*alpha + power_kW*beta - price_cents*gamma.
   Idée : parcours global → min-heap local (taille k).
   Complexité : O(n log k). */
```

A3) Filtrage mixte : pré-filtres + règle postfix

But : accélérer le filtrage avec des conditions simples avant l'évaluation de la règle.

```
int filter_ids_with_rule(StationNode* r, char* toks[], int n, int* out,
int cap,
                         int min_power, int min_slots);
/* D'abord pré-filtres C (ex. power>=min_power && slots>=min_slots),
puis règle postfix pour la précision.
   Complexité : O(n) en pratique (parcours + filtrage). */
```

A4) MRU capée (sans doublons)

But : maintenir pour chaque véhicule un historique borné, sans doublon, en capant la longueur.

```
void mru_add_station(SList* mru, int station_id, int MRU_CAP);
/* Étapes : ds_slist_remove_value(mru, station_id) ; insert head ; si
longueur>CAP, remove tail.
   Complexité : O(L) avec L<=CAP (CAP petit ⇒ coût pratique O(1)). */
```

A5) Export snapshot CSV

But : produire un export trié pour visualisation/partage.

```
int si_export_csv(StationNode* r, const char* path);
/* Parcours in-order → fprintf.
   Retour : nb de lignes écrites ; gérer l'ouverture/erreurs
simplement. */
```

A6) Plus proches voisins (naïf, sans index spatial)

But : illustrer une recherche géographique simple et ses limites (O(n)).

```
double haversine(double lat1, double lon1, double lat2, double lon2);
int nearest_k(StationIndex* idx, double lat, double lon, int k, int*
out_ids);
/* Parcours toutes stations ; garder les k plus proches via un petit
heap local. */
```

5) Modèles de scénarios applicatifs (menu B : choisissez 1)

B1) Heure de pointe

• Charger le CSV ; appliquer une rafale d'événements (arrivées massives).

• Comparer Top-K par score avant/après ; lister les candidates après pré-filtres + règle postfix.

B2) Panne & dégradation

• Forcer des états dégradés (ex. slots_free=0) pour certaines stations ; visualiser l'impact.

• Option : si_delete (retirer) puis si_add (retour service) ; comparer avant/après.

B3) Campagne tarifaire

• Appliquer une réduction sur price_cents pour un intervalle d'ID ou une sélection.

• Montrer Top-K avant/après et exporter un snapshot CSV des meilleures stations.

6) Qualité minimale attendue (sans imposer l'écriture de tests unitaires)

• Messages clairs en cas d'erreur I/O (fichiers manquants, lignes invalides).

• Libération mémoire (si_clear, MRU, queue).

• Complexité en commentaire pour chaque fonction ajoutée (O(...) + 1 phrase de justification).

• (Option) Une cible Makefile `debug` avec `-fsanitize=address,undefined`.

7) Évaluation (proposition)

• Fonctions avancées (2 à 3 du menu A)

• Un scénario complet (menu B)

• Lisibilité & explications (complexités, commentaires, logs)

8) Exemple minimal de démonstration (extrait)

```
// Charger CSV
int c = ds_load_stations_from_csv("izivia_tp_subset.csv", &idx);
if (c <= 0) { printf("[ERR] CSV vide/absent\n"); return 1; }

// Rafale d'événements (heure de pointe)
for (int i = 0; i < DS_EVENTS_COUNT; ++i) q_enqueue(&q, DS_EVENTS[i]);
process_events(&q, &idx);

// Top-5 par score
int top[5];
int k = si_top_k_by_score(idx.root, 5, top, /*alpha=*/2, /*beta=*/1,
/*gamma=*/1);
printf("Top-5: "); for (int i=0;i<k;i++) printf("%d ", top[i]);
printf("\n");

// Règle postfix + pré-filtres
char* rule[] = {"slots","1",">=","power","50",">=","&&"};
int filt[64], m = filter_ids_with_rule(idx.root, rule, 7, filt, 64,
/*min_power=*/50, /*min_slots=*/1);
printf("Candidats: "); for (int i=0;i<m;i++) printf("%d ", filt[i]);
printf("\n");
```

9) Conseils de présentation

- Un README court expliquant : modules choisis, scénario, et comment lancer la démo.
- Des impressions lisibles (titres/sections) pour valoriser le résultat en séance.