Explication du programme C (main.c)

Le programme gère une liste d'informations sur des stations électriques, organisées dans un arbre AVL (une structure de données d'arbre binaire équilibré). Il lit des données depuis un fichier CSV, insère ces données dans un arbre AVL, puis sauvegarde les résultats dans un autre fichier CSV. Voici ce que chaque fonction fait :

La fonction create_node :

Cette fonction crée un nœud AVL avec des informations sur une station. Elle alloue de la mémoire pour un nouveau nœud, l'initialise avec l'ID de la station, la capacité, la consommation, et la hauteur de l'arbre (initialisée à 1). Ce nœud n'a pas de fils au départ, donc ses fils gauche et droit sont définis à NULL.

En entrée : un station_id, une capacity et une consumption.

En sortie : un pointeur vers le nouveau nœud.

AVLNode *create_node(int station_id, long capacity, long consumption) { ... }

```
AVLNode *create_node(int station_id, long capacity, long consumption) {
    AVLNode *node = malloc(sizeof(AVLNode));
    if (!node) {
        fprintf(stderr, "Erreur d'allocation mémoire pour le nœud\n");
        exit(1);
    }
    node->station_id = station_id;
    node->capacity = capacity;
    node->consumption = consumption;
    node->height = 1;
    node->left = node->right = NULL;
    return node;
}
```

La fonction height

Cette fonction retourne la hauteur d'un nœud AVL. La hauteur est utilisée pour équilibrer l'arbre. Si le nœud est NULL, la hauteur est 0.

En Entrée : un pointeur vers un nœud.

En sortie : la hauteur du nœud.

int height(AVLNode *node) { ... }

```
int height(AVLNode *node) {
    return node ? node->height : 0;
}
```

La fonction get_balance :

Cette fonction calcule l'équilibre du nœud. Elle soustrait la hauteur du sous-arbre droit de celle du sous-arbre gauche. Un équilibre positif signifie que l'arbre est plus lourd à gauche, un équilibre négatif signifie qu'il est plus lourd à droite.

En entrée : un pointeur vers un nœud.

En sortie : l'équilibre du nœud.

int get_balance(AVLNode *node) { ... }

```
int get_balance(AVLNode *node) {
    return node ? height(node->left) - height(node->right) : 0;
}
```

La fonction rotate_right et rotate_left

Ces fonctions sont responsables de la **rotation** d'un sous-arbre pour maintenir l'équilibre de l'arbre AVL. Lorsqu'un sous-arbre est trop lourd d'un côté (à gauche ou à droite), une rotation est effectuée pour équilibrer l'arbre.

rotate_right : Lorsque l'arbre est trop lourd à gauche.

rotate_left : Lorsque l'arbre est trop lourd à droite.

Les deux fonctions ajustent également la hauteur des nœuds après la rotation.

AVLNode *rotate_right(AVLNode *y) { ... }

AVLNode *rotate_left(AVLNode *x) { ... }

```
AVLNode *rotate_right(AVLNode *y) {
    AVLNode *x = y->left;
    AVLNode *T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));
    x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));

    return x;
}
```

Et

```
AVLNode *rotate_left(AVLNode *x) {
    AVLNode *y = x->right;
    AVLNode *T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));
    y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));

    return y;
}
```

La fonction insert_node

Cette fonction insère un nouveau nœud dans l'arbre AVL en respectant l'ordre des IDs de station (les nœuds sont triés par ID de station). Si un nœud avec le même ID existe déjà, la fonction additionne la nouvelle consommation à l'ancienne.

Après chaque insertion, la fonction ajuste la hauteur du nœud et vérifie l'équilibre. Si l'arbre devient déséquilibré, elle effectue une ou deux rotations pour rééquilibrer l'arbre.

En entrée : le nœud racine de l'arbre, un station_id, capacity et consumption.

En sortie : le nouveau nœud racine après insertion (si l'arbre a changé).

AVLNode *insert_node(AVLNode *node, int station_id, long capacity, long consumption) { ... }

```
AVLNode *insert_node(AVLNode *node, int station_id, long capacity, long consumption) {
    if (!node) return create_node(station_id, capacity, consumption);

    if (station_id < node->station_id)
        node->left = insert_node(node->left, station_id, capacity, consumption);
    else if (station_id > node->station_id)
        node->right = insert_node(node->right, station_id, capacity, consumption);
    else {
        node->consumption += consumption;
        return node;
    }

    node->height = 1 + (height(node->left) > height(node->right) ? height(node->left) : height(node->right));

    int balance = get_balance(node);

    if (balance > 1 && station_id < node->left->station_id)
        return rotate_right(node);

    if (balance < -1 && station_id > node->right->station_id)
        return rotate_left(node);

    if (balance > 1 && station_id > node->left->station_id) {
        node->left = rotate_left(node->left);
        return rotate_right(node);
    }

    if (balance < -1 && station_id < node->right->station_id) {
        node->right = rotate_right(node->right);
        return rotate_left(node);
    }

    return node;
}
```

La fonction free_tree

Cette fonction libère la mémoire allouée pour l'arbre AVL en faisant un parcours en profondeur et en libérant chaque nœud un par un.

En entrée : un pointeur vers la racine de l'arbre.

En sortie : aucune (la mémoire est libérée).

void free_tree(AVLNode *node) { ... }

```
void free_tree(AVLNode *node) {
    if (!node) return;
    free_tree(node->left);
    free_tree(node->right);
    free(node);
}
```

La fonction save_in_order

Cette fonction enregistre l'arbre AVL dans un fichier, dans l'ordre croissant des IDs des stations. Elle parcourt l'arbre en ordre (gauche, racine, droite) et écrit les informations de chaque station dans le fichier de sortie.

En entrée : la racine de l'arbre et un fichier de sortie.

En sortie : écrit les données dans le fichier.

void save_in_order(AVLNode *root, FILE *output) { ... }

```
void save_in_order(AVLNode *root, FILE *output) {
    if (root) {
        save_in_order(root->left, output);
        fprintf(output, "%d:%ld:%ld\n", root->station_id, root->capacity, root->consumption);
        save_in_order(root->right, output);
    }
}
```

Le main

La fonction principale du programme. Elle fait plusieurs choses :

- Vérifie que le programme est exécuté avec un fichier d'entrée.
- Ouvre le fichier CSV, lit chaque ligne et extrait les données.
- Insère les données dans l'arbre AVL en appelant insert_node.
- Une fois tout le fichier traité, elle écrit le contenu de l'arbre dans un fichier de sortie result.csv.
- Elle libère la mémoire allouée pour l'arbre à la fin.
- Entrée : un fichier CSV contenant les informations sur les stations.
- Sortie : un fichier CSV avec les stations triées par ID.

int main(int argc, char **argv) { ... }

```c
int main(int argc, char **argv) {
    if (argc < 2) {
        printf("Usage: %s <input_csv>\n", argv[0]);
        return 1;
    }

    const char *input_file = argv[1];
    FILE *file = fopen(input_file, "r");
    if (!file) {
        fprintf(stderr, "Erreur : impossible d'ouvrir le fichier %s.\n", input_file);
        return 1;
    }

    AVLNode *root = NULL;
    char line[MAX_LINE_LENGTH];
    while (fgets(line, sizeof(line), file)) {
        char *token;
        int col = 0;
        int station_id = 0, lv_station = 0, company = 0;
        long capacity = 0, load = 0;
        printf("Découpage de la ligne : %s\n", line);
        token = strtok(line, ";");
        while (token != NULL) {
            printf("Colonne %d : %s\n", col, token);
            switch (col) {
                case 0: station_id = atoi(token); break;
                case 1: lv_station = atoi(token); break;
                case 2: company = atoi(token); break;
                case 3:
                    if (strcmp(token, "-") != 0) {
                        capacity = atol(token);
                    } else {
                        capacity = 0;
                    }
                    break;
                case 4:
                    load = atol(token);
                    break;
            }
            col++;

            token = strtok(NULL, ";");
        }

        if (lv_station > 0 && load > 0) {
            printf("Insertion : LV Station: %d, Capacity: %ld, Load: %ld\n", lv_station, capacity, load);
            root = insert_node(root, lv_station, capacity, load);
        }
    }

    fclose(file);

    FILE *output = fopen("output/result.csv", "w");
    if (!output) {
        perror("Erreur d'ouverture du fichier de sortie");
        free_tree(root);
        return 1;
    }
    save_in_order(root, output);
    fclose(output);

    free_tree(root);

    return 0;
}
```

Explication du programme c-wire.h

1) Protection contre les inclusions multiples

```
#ifndef CWIRE_HEADER
# define CWIRE_HEADER
```

Ces deux lignes garantissent que le contenu de ce fichier d'en-tête (cwire.h) ne sera inclus qu'une seule fois, même si plusieurs fichiers source incluent ce fichier. Cela évite les erreurs de redéfinition.

2) Inclusions des bibliothèques standard

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
```

Ces lignes incluent les bibliothèques standards nécessaires :

- **<stdio.h>** : Basique … Pour les opérations d'entrée/sortie (par exemple, printf, fopen).
- **<stdlib.h>** : Basique …Pour les fonctions d'allocation mémoire (malloc, free) et d'autres utilitaires.
- **<string.h>** : Pour la manipulation de chaînes de caractères (strtok, strcmp).

3) Définition de la constante MAX_LINE_LENGTH

```
# define MAX_LINE_LENGTH 1024
```

Ici, la constante **MAX_LINE_LENGTH** est définie à 1024. On l'utilisera plus tard dans le programme pour limiter la longueur maximale des lignes lues depuis un fichier (en l'occurrence, lors de la lecture des données du fichier CSV).

4) Définition de la structure AVLNode

```
typedef struct AVLNode {
    int station_id;
    long capacity;
    long consumption;
    int height;
    struct AVLNode *left;
    struct AVLNode *right;
} AVLNode;
```

C'est la définition de la structure **AVLNode** qui représente un **nœud de l'arbre AVL**. Chaque nœud contient :

- **station_id** : Un identifiant unique pour chaque station.
- **capacity** : La capacité de la station (peut-être en termes de production ou d'autres ressources).
- **consumption** : La consommation associée à cette station.
- **height** : La hauteur de ce nœud dans l'arbre AVL, utilisée pour équilibrer l'arbre.
- **left et right** : Pointeurs vers les sous-arbres gauche et droit de ce nœud.

5) **Prototypes des fonctions**

```
// Prototypes des fonctions
AVLNode *create_node(int station_id, long capacity, long consumption);
int height(AVLNode *node);
int get_balance(AVLNode *node);
AVLNode *rotate_right(AVLNode *y);
AVLNode *rotate_left(AVLNode *x);
AVLNode *insert_node(AVLNode *node, int station_id, long capacity, long consumption);
void free_tree(AVLNode *node);
void save_in_order(AVLNode *root, FILE *output);
```

Cette partie définit les **prototypes des fonctions** qui sont implémentées dans le fichier source cwire.c. Ces fonctions interagiront avec la structure AVLNode et seront utilisées pour manipuler l'arbre AVL. Voici ce que chaque fonction fait :

- **create_node** : Crée un nouveau nœud avec les informations d'une station.
- **height** : Retourne la hauteur d'un nœud.
- **get_balance** : Calcule l'équilibre d'un nœud.
- **rotate_right** et **rotate_left** : Effectuent des rotations pour rééquilibrer l'arbre.
- **insert_node** : Insère un nouveau nœud dans l'arbre AVL.
- **free_tree** : Libère la mémoire utilisée par l'arbre AVL.
- **save_in_order** : Sauvegarde l'arbre AVL dans un fichier dans l'ordre croissant des IDs des stations.

6) **Fin de la protection contre les inclusions multiples**

```
#endif
```

Cette ligne marque la fin de la protection contre les inclusions multiples, ce qui permet de s'assurer que le fichier d'en-tête n'est inclus qu'une seule fois dans chaque unité de traduction.

Explication du programme cwire.sh

1) **Vérification des arguments fournis**

```
if [ "$#" -lt 3 ]; then
    echo "Usage: $0 <chemin_csv> <type_station> <type_consommateur> [id_centrale]"
    echo "Options :"
    echo "  <type_station> : hvb, hva, lv"
    echo "  <type_consommateur> : comp, indiv, all"
    exit 1
fi
```

Le script commence par vérifier si un nombre suffisant d'arguments a été fourni ($# représente le nombre d'arguments). Si le nombre d'arguments est inférieur à 3, il affiche un message d'erreur et quitte.

2) Assignation des variables

```
CSV_PATH=$1
TYPE_STATION=$2
TYPE_CONSUMER=$3
CENTRALE_ID=$4
OUTPUT_DIR="output"
TMP_DIR="tmp"
GRAPHS_DIR="graphs"
```

Les arguments passés au script sont assignés à des variables :

- **CSV_PATH** : chemin du fichier CSV à traiter.
- **TYPE_STATION** : type de station (hvb, hva, ou lv).
- **TYPE_CONSUMER** : type de consommateur (comp, indiv, ou all).
- **CENTRALE_ID** : ID de la centrale (optionnel).
- **Répertoires de sortie** : output, tmp, et graphs sont définis pour organiser les résultats et les fichiers temporaires.

3) Vérification des valeurs d'argument

Le script vérifie que les types de station et de consommateur sont valides. Si l'un d'eux ne correspond pas aux valeurs attendues, le script affiche un message d'erreur et quitte.

4) Restrictions sur les types de consommateur et de station

```
if [[ "$TYPE_STATION" != "hvb" && "$TYPE_STATION" != "hva" && "$TYPE_STATION" != "lv" ]]; then
    echo "Erreur : type de station invalide. Choisir parmi : hvb, hva, lv."
    exit 1
fi

if [[ "$TYPE_CONSUMER" != "comp" && "$TYPE_CONSUMER" != "indiv" && "$TYPE_CONSUMER" != "all" ]]; then
    echo "Erreur : type de consommateur invalide. Choisir parmi : comp, indiv, all."
    exit 1
fi

if [[ ("$TYPE_STATION" == "hvb" || "$TYPE_STATION" == "hva") && "$TYPE_CONSUMER" != "comp" ]]; then
    echo "Erreur : seuls les consommateurs de type 'comp' sont autorisés pour hvb et hva."
    exit 1
fi
```

Ce bloc impose une restriction supplémentaire : pour les stations de type hvb ou hva, seuls les consommateurs de type comp sont autorisés. Si ce n'est pas le cas, un message d'erreur est affiché et le script quitte.

5) Création des répertoires nécessaires

```
mkdir -p "$OUTPUT_DIR" "$TMP_DIR" "$GRAPHS_DIR"
```

Les répertoires de sortie (output, tmp, graphs) sont créés s'ils n'existent pas déjà.

6) Vérification de l'existence du fichier CSV

```
if [ ! -f "$CSV_PATH" ]; then
    echo "Erreur : le fichier $CSV_PATH n'existe pas."
    exit 1
fi
```

Si le fichier CSV spécifié n'existe pas, le script affiche un message d'erreur et quitte.

7) Vérification de l'existence du programme C compilé

```
if [ ! -f "./cwire" ]; then
    echo "Compilation du programme C..."
    make
    if [ "$?" -ne 0 ]; then
        echo "Erreur : Échec de la compilation du programme C."
        exit 1
    fi
fi
```

Le script vérifie si le programme C cwire a été compilé et existe dans le répertoire actuel. Si ce n'est pas le cas, il tente de le compiler en utilisant make. Si la compilation échoue, le script affiche un message d'erreur et quitte.

8) Filtrage des données CSV

```
FILTERED_CSV="$TMP_DIR/filtered.csv"
if [ -n "$CENTRALE_ID" ]; then
    awk -F";" -v station="$TYPE_STATION" -v central="$CENTRALE_ID" '$1 == central && $2 == station' "$CSV_PATH" > "$FILTERED_CSV"
else
    awk -F";" -v station="$TYPE_STATION" '$2 == station' "$CSV_PATH" > "$FILTERED_CSV"
fi
```

Le script filtre le fichier CSV en fonction du type de station et, si un CENTRALE_ID est fourni, en fonction de cet ID également. Il utilise **awk** pour cela et enregistre les résultats filtrés dans un fichier temporaire filtered.csv.

9) Exécution du programme C

```
START_TIME=$(date +%s)

./cwire "$FILTERED_CSV" "$TYPE_STATION" "$TYPE_CONSUMER" > "$OUTPUT_DIR/result_${TYPE_STATION}_${TYPE_CONSUMER}.csv"
if [ "$?" -ne 0 ]; then
    echo "Erreur : échec du traitement des données par le programme C."
    exit 1
fi

END_TIME=$(date +%s)
DURATION=$((END_TIME - START_TIME))
```


```
START_TIME=$(date +%s)

./cwire "$FILTERED_CSV" "$TYPE_STATION" "$TYPE_CONSUMER" > "$OUTPUT_DIR/result_${TYPE_STATION}_${TYPE_CONSUMER}.csv"
if [ "$?" -ne 0 ]; then
    echo "Erreur : échec du traitement des données par le programme C."
    exit 1
fi

END_TIME=$(date +%s)
DURATION=$((END_TIME - START_TIME))
```

Le script exécute le programme C cwire avec les données filtrées et les options fournies en argument. Le temps de traitement est mesuré et affiché à la fin.

10) Affichage du résultat

```
echo "Traitement terminé en $DURATION secondes. Résultats dans $OUTPUT_DIR."
```

Enfin, le script affiche un message indiquant que le traitement est terminé et fournit la durée de l'exécution ainsi que l'emplacement des résultats dans le répertoire output.

Explication du makefile

```
all: main

main: main.c
    gcc main.c -o main

clean:
    rm -f main

.PHONY: all clean
```

1) **all: main**

- **Que fait cette ligne ?** C'est la **cible par défaut** du Makefile. Lorsqu'on tape make sans autre argument, c'est cette cible qui sera exécutée.

- Pourquoi main ? La cible main est un programme (ou une règle) qui va être généré. main dépend de la compilation de main.c, donc cela va dire à make de créer le fichier exécutable main à partir de main.c.

2) **main: main.c**

- Que fait cette ligne ? Cette règle dit à make qu'il faut créer l'exécutable main à partir du fichier source main.c.

- Comment ça fonctionne ? La commande gcc main.c -o main sera exécutée. Ce sont les instructions pour compiler le fichier main.c et produire un exécutable appelé main.

-gcc est le compilateur C.

-main.c est le fichier source qu'on veut compiler.

- -o main signifie que le programme compilé doit être nommé main (par défaut, le compilateur le nommerait a.out).

3) **clean:**

- Que fait cette ligne ? Cette cible permet de supprimer les fichiers créés pendant la compilation (comme l'exécutable main).

- Pourquoi faire ça ? Si on veut repartir de zéro, ou nettoyer notre répertoire de travail, on peut exécuter make clean pour supprimer tous les fichiers qui ont été générés pendant la compilation.

- Que fait rm -f main ? La commande rm supprime le fichier main. L'option -f force la suppression sans demander confirmation, même si le fichier n'existe pas.

4) **.PHONY: all clean**

- Que fait cette ligne ? Elle indique que make ne doit pas chercher des fichiers nommés all ou clean dans le répertoire. Ces noms sont réservés pour des cibles spéciales, donc .PHONY les marque comme des "cibles fictives".

- Cela permet de s'assurer que make exécute toujours les règles all et clean, peu importe si un fichier du même nom existe dans le répertoire.