Université Abdelmalek Essaidi

Ecole Nationale des Sciences Appliquées de Tétouan

RAPPORT DE PROJET

Mini-Supermarché

Filière : GI 1
Module : Structures de données en C

Groupe N°4 :

- 23065715 | EL ASRI | Fatima Zahra
- 23065841 | BOUFNAR | Douae
- 23065632 | DAHBI | Aya
- 25063668 | EL MOUADAN | Taha

2025 - 2026

ENSA TETOUAN
PR. HACHCHANE Imane

1 Introduction

Ce projet a pour objectif de développer un système complet de gestion pour un mini-supermarché en langage C, combinant rigueur technique et application pratique. Il permet de gérer les produits, les clients, le passage en caisse et l'historique des transactions, offrant ainsi une solution opérationnelle pour un commerce de proximité.

Le programme exploite efficacement les structures de données classiques :

- **Tables de hachage** pour un accès rapide aux produits par identifiant.
- **Arbres binaires de recherche (ABR)** pour organiser les clients.
- **Listes chaînées** pour gérer dynamiquement les paniers avant validation.
- **Files d'attente** pour gérer le flux des clients à la caisse.
- **Piles** pour conserver l'historique des transactions.

Le système inclut une interface interactive complète. Chaque panier validé génère un ticket détaillé, tandis que les données sont sauvegardées automatiquement, garantissant fiabilité et persistance.

2 Structures de Données

Table de Hachage

Rôle Stratégique : *Accès instantané $O(1)$ à l'inventaire.*

Cette structure est choisie pour sa rapidité. Dans un supermarché contenant des milliers d'articles, le scan en caisse doit être immédiat. Le hachage évite un parcours séquentiel lent en transformant l'ID produit directement en adresse mémoire.

```
typedef struct Produit {
    int id; char nom[50]; float prix; int quantite;
    struct Produit* suivant; // Liste chainee pour collisions
} Produit;
Produit* tableHachage[100];
```

Arbre Binaire de Recherche

Rôle Stratégique : *Organisation alphabétique dynamique.*

L'ABR est idéal pour gérer la base client car il maintient les données triées naturellement. Contrairement à un tableau qui nécessiterait un tri coûteux après chaque ajout, l'ABR insère le client directement à la bonne place, facilitant les recherches par nom.

```
typedef struct Client {
    int id; char nom[30]; float totalDepense;
    struct Client *gauche, *droite;
} Client;
```

File (Queue)

Rôle Stratégique : *Gestion équitable du flux caisse.*

Pour simuler une caisse réaliste, nous utilisons le principe FIFO (First In, First Out). La file d'attente garantit que l'ordre d'arrivée des clients est strictement respecté lors du traitement des paiements, assurant l'équité du service.

```
typedef struct ElementFile {
    Client* client;
    struct ElementFile* suivant;
} ElementFile;
typedef struct { ElementFile *front, *rear; } FileAttente;
```

3 Fonctions Principales du Programme

Nous avons divisé le programme en modules distincts pour assurer la maintenabilité.

FONCTIONS PRODUITS	
`ajouterProduit()`	Cette fonction calcule l'index unique via le modulo de l'ID. Elle gère ensuite les collisions en insérant le nouveau produit en tête de la liste chaînée.
`rechercherProduitParId()`	Fonction critique pour la performance, elle accède directement à la case mémoire calculée par le hachage. Si la case contient plusieurs produits (collision), elle parcourt la courte liste chaînée pour renvoyer le pointeur vers le produit recherché
`supprimerProduit()`	Elle permet de retirer un produit de l'inventaire grâce à son ID. La fonction localise l'élément dans la liste chaînée correspondante et modifie les pointeurs pour exclure le nœud visé avant de libérer la mémoire associée.
`afficherStock()`	Parcourt l'intégralité de la table de hachage (toutes les alvéoles et leurs listes chaînées respect.

Historique des transactions	
`pushTransaction()`	Empile une nouvelle transaction (ID client, montant, date) au sommet de la pile. Cela permet de mémoriser instantanément la toute dernière action effectuée dans le système
`apopTransaction()`	Dépile l'élément du sommet pour annuler la dernière opération. Elle est utilisée pour corriger une erreur de saisie, permettant de restaurer les stocks et le solde du client à leur état précédent.
`afficherHistorique()`	Parcourt la pile du sommet vers la base pour afficher la liste chronologique inverse des ventes, permettant de voir les opérations les plus récentes en premier.

FONCTIONS CLIENTS (ABR)

`insererClient()`	Compare récursivement le nom du nouveau client avec les nœuds existants. Elle descend à gauche ou à droite jusqu'à trouver une feuille libre pour l'insertion.
`rechercherClient()`	Permet de retrouver un client spécifique par son nom ou son ID. Elle exploite la structure de l'ABR pour éliminer la moitié des candidats à chaque étape de comparaison, garantissant une recherche rapide.
`supprimerClient()`	Retire un client de l'arbre. Cette opération est complexe car elle doit réorganiser l'ABR pour conserver la propriété de tri, notamment en remplaçant le nœud supprimé par son successeur immédiat (le plus petit élément du sous-arbre droit).
`afficherClientsInfixe()`	Réalise un parcours infixe (Gauche → Racine → Droite) de l'arbre. Cela permet d'afficher la liste de tous les clients triée par ordre alphabétique croissant.

Passage de Caisse

`enfilerClient()`	Ajoute un client à la fin de la file d'attente (structure FIFO) lorsqu'il se présente à la caisse. Elle met à jour le pointeur arrière (rear) de la file.
`defilerClient()`	Retire le client situé en tête de la file pour traiter ses achats. Elle met à jour le pointeur avant (front) et renvoie l'identifiant du client à servir.
`traiterAchat()`	Fonction centrale qui orchestre la vente. Elle récupère le panier du client, vérifie la disponibilité des stocks via la table de hachage, met à jour les quantités restantes et calcule le montant total dû.
`genererTicket()`	Produit un récapitulatif de la transaction une fois le paiement validé. Elle affiche ou sauvegarde la liste des articles achetés, la date, et le montant total pour le client

4 Analyse de Complexité

Complexité : Recherche produit via hachage :

Moyenne : $O(1)$ | **Pire :** $O(n)$
La recherche d'un produit par son identifiant est quasi-instantanée avec une complexité moyenne de $O(1)$. Grâce au calcul de l'index par la fonction de hachage (modulo), on accède directement à la case mémoire du produit sans avoir besoin de parcourir tout le tableau.

Complexité : Parcours / tri liste chaînée :

Moyenne : $O(\log n)$ | **Pire :** $O(n)$
Le parcours d'une liste chaînée (utilisée pour les collisions ou le panier) possède une complexité linéaire de $O(n)$. Contrairement à un tableau, l'accès n'est pas direct ; il est nécessaire de visiter chaque maillon un par un en suivant les pointeurs pour atteindre un élément ou effectuer un tri.

Complexité : Insertion

Complexité : $O(n)$

lorsque les insertions sont déjà triées. L'arbre binaire se dégénère en liste chaînée, nécessitant la comparaison avec tous les nœuds existants.

Complexité : Rcherche

Complexité : $O(n)$ **Dans le cas d'un arbre non équilibré**

La recherche parcourt successivement tous les nœuds jusqu'à trouver le nom recherché.

Complexité : Supression

Complexité : $O(n)$ **Dans le pire des cas.**

La recherche du nœud puis du successeur dépend de la hauteur de l'arbre, qui peut atteindre n.

Complexité : File : enfiler / défiler :

Complexité : $O(1)$
La gestion de la file d'attente s'effectue en temps constant $O(1)$. Comme nous maintenons des pointeurs directs vers le début (tête) et la fin (queue) de la file, l'ajout d'un client ou son passage en caisse se fait instantanément, quel que soit le nombre de personnes en attente.

Complexité : Pile : push / pop :

Complexité : $O(1)$

Les opérations sur l'historique des transactions sont également en $O(1)$. L'empilement (sauvegarde) et le dépilement (annulation) se font toujours au sommet de la pile, ce qui garantit une exécution immédiate sans aucune boucle de recherche.

5 Défis Rencontrés et Améliorations

Obstacles Techniques

- **Gestion des Pointeurs :** La manipulation des listes chaînées pour les collisions dans la table de hachage a nécessité une gestion rigoureuse pour éviter les fuites de mémoire.
- **Synchronisation des Structures :** Mettre à jour le stock (Hachage) instantanément lors de la validation d'un panier (File) a demandé une logique d'interaction complexe pour maintenir la cohérence des données.
- **Cas Limites :** Gérer proprement les erreurs critiques, comme la tentative de vente d'un produit en rupture de stock ou la recherche d'un client inexistant dans l'arbre.

Perspectives d'Évolution

- **Modes de paiement flexibles :** permettre au client de choisir entre paiement en espèces ou par TPE.
- **Gestion fiscale automatique :** calculer et afficher la TVA directement sur le ticket, pour plus de transparence.
- **Options personnalisées pour le client :** proposer des sacs ou emballages et inclure leur coût dans le total, ainsi que la gestion des cartes de fidélité pour récompenser les clients réguliers.
- **Services complémentaires :** intégrer un système de livraison associé au magasin, afin de proposer une expérience client complète et connectée.

Conclusion

Ce projet a permis de concrétiser la manipulation des structures de données et la gestion de la mémoire en C. Nous avons appris l'importance de la modularité du code pour créer un système fiable et évolutif.

L'utilisation combinée des tables de hachage, des arbres binaires et des files d'attente a permis d'optimiser chaque aspect de la gestion du supermarché, garantissant performance et fluidité.