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° 5 :

- 23065645 | Garhani | Ilyas
- 23065640 | Farhan | Othmane
- 23065590 | Rahel |Salma
- 23065714 | Sabaoui | Siham

2025 - 2026

ENSA TETOUAN
PR. HACHCHANE Imane

1. Introduction :

L'informatisation des systèmes de gestion est devenue indispensable dans le domaine du commerce, même à petite échelle. Les mini-supermarchés doivent assurer une gestion efficace de leurs produits, de leurs clients et de leurs transactions afin de garantir un bon fonctionnement quotidien.

Ce projet a pour objectif d'implémenter un système de gestion d'un mini-supermarché en langage C, en utilisant les structures de données étudiées en cours Le programme permet de gérer le stock, les informations des clients, le passage en caisse et l'historique des transactions.

Pour assurer l'efficacité du système, plusieurs structures de données sont utilisées, notamment les listes chaînées, les tables de hachage, les arbres binaires de recherche, les files et les piles. Chaque structure est choisie selon les besoins de l'application. Le projet inclut également la gestion des fichiers pour assurer la sauvegarde des données et leur persistance.

2. Structures de Données Utilisées (1,5 pages)

A)- Structure des Produits : Liste Chaînée et Table de Hachage

La structure de données liste chaînée est utilisée pour représenter le catalogue des produits, car elle permet une gestion dynamique (ajout, suppression et modification) sans contrainte de taille. En complément, une table de hachage permet un accès rapide aux produits par leur identifiant, optimisant ainsi les opérations de recherche lors du passage en caisse.

```c
typedef struct Produit {
    int id;
    char nom[30];
    float prix;
    int quantite;
    struct Produit *suivant;
} Produit;

typedef struct Liste {
    Produit *tete;
    Produit *queue;
    int taille;
} Liste;

typedef struct {
    Produit *table[TAILLE_TABLE];
} TableHachage;
```

B)- Structure des Clients : Arbre Binaire de Recherche (ABR)

Les clients sont stockés dans une **structure de données de type arbre binaire de recherche**, qui maintient automatiquement les éléments triés par ordre alphabétique selon

le nom, tout en permettant des opérations efficaces de recherche, d'insertion et de suppression.

```c
typedef struct ClientNode {
    int id;
    char nom[30];
    float total;
    struct ClientNode *filsg;
    struct ClientNode *filsd;
} ClientNode;

typedef struct {
    ClientNode *racine;
} ArbreClients;
```

C)- Structure de la File d'Attente : Liste Chaînée FIFO

La file d'attente est implémentée à l'aide d'une **structure de données FIFO basée sur une liste chaînée**, garantissant un traitement des clients dans l'ordre d'arrivée selon le principe "premier arrivé, premier servi". avec des opérations simples et rapides.

```c
typedef struct ClientFile {
    int id;
    char nom[30];
    float total;
    struct ClientFile *suivant;
} ClientFile;

typedef struct File {
    ClientFile *tete;
    ClientFile *queue;
    int nef;
} File;
```

D)- Structure des Transactions : Pile LIFO

```c
typedef struct Transaction {
    int idClient;
    char nomClient[30];
    int idProduit;
    char nomProduit[30];
    int quantite;
    float total;
    char date[20];
    struct Transaction *suivant;
} Transaction;
typedef struct Pile {
    Transaction *tete;
    int nef;
} Pile;
```

L'historique des transactions est géré à l'aide d'une **structure de données de type pile (LIFO)**, adaptée aux besoins d'annulation, car la dernière transaction enregistrée est la première pouvant être retirée.

3. Fonctions Principales du Programme (2 pages)

1.1 Fonctions Produits

- **ajouterProduit() :**
  La fonction AjouterProduitFin ajoute un produit à la fin d'une liste chaînée après avoir vérifié la validité des données et l'absence de doublons. Elle met à jour les pointeurs de tête et de queue, incrémente la taille de la liste et reconstruit la table de hachage pour maintenir la cohérence des données.
- **supprimerProduit() :**
  La fonction supprimerProduit sert à retirer un produit d'une liste chaînée en utilisant son identifiant. Elle commence par vérifier si la liste est vide, puis cherche le produit dans la liste. S'il est trouvé, elle le supprime en mettant à jour les pointeurs (tête et queue si nécessaire), libère la mémoire occupée, diminue la taille de la liste et reconstruit la table de hachage pour garder des données cohérentes.
- **modifierProduit() :**
  La fonction modifierProduit permet de mettre à jour le nom, le prix ou la quantité d'un produit à partir de son identifiant. Si le produit est trouvé, l'utilisateur peut ajuster ses informations, puis la table de hachage est rafraîchie pour maintenir la cohérence des données
- **rechercherProduitParID()** (avec la table de hachage) :
  rechercherProduitHachage retrouve un produit dans la table de hachage à partir de son ID en parcourant la liste chaînée à l'index calculé, et renvoie le produit ou NULL.
  afficherProduits() + tri.
  Ces fonctions permettent d'afficher une liste de produits et de trier les produits par prix, quantité ou nom en parcourant la liste chaînée et en échangeant les informations des nœuds si nécessaire.

1.2 Fonctions Clients (ABR)

- **insererClient() :**
  La fonction insererClient ajoute un client dans un arbre binaire trié par nom. Elle place le client dans le sous-arbre gauche ou droit selon l'ordre alphabétique et affiche un message si le nom existe déjà.
- **rechercherClient() :**
  Les fonctions rechercherClientNom et rechercherClientID permettent de retrouver un client dans un arbre binaire. La première cherche par nom en suivant l'ordre alphabétique, la seconde cherche par identifiant en parcourant récursivement les sous-arbres gauche et droit, et retournent le client trouvé ou NULL si absent

- **supprimerClient() :**
  La fonction supprimerClientNom supprime un client dans un arbre binaire par son nom, en gérant les cas de feuille, un seul fils ou deux fils (en remplaçant par le successeur minimal). La fonction minNode trouve le nœud avec le plus petit nom dans le sous-arbre droit.
- **afficherClientsInfixe() :**
  La fonction afficherClients parcourt l'arbre en ordre croissant et affiche l'ID, le nom et le total de chaque client.

1.3 Passage en Caisse (File + Table de Hachage)

- **enfilerClient() :**
  La fonction enfilerClient ajoute un client à la fin d'une file. Elle met à jour les pointeurs de tête et de queue et incrémente le nombre d'éléments.
- **defilerClient() :**
  La fonction defilerClient supprime et retourne le premier client d'une file, en mettant à jour les pointeurs et le nombre d'éléments.
- **traiterAchat() :**
  La fonction servirClient défiler le client suivant dans la file, lui permet de choisir des produits en vérifiant la disponibilité et les quantités, ajoute les articles au panier et enregistre les transactions. Elle affiche et génère le ticket, met à jour le stock, reconstruit la table de hachage et actualise le total dépensé par le client dans l'arbre
- **genererTicket() :**
  La fonction afficherTicketEcran affiche à l'écran le ticket d'un client avec son ID, son nom, la date, la liste des produits achetés (quantité, prix, total) et le montant total.

1.4 Historique des Transactions (Pile)

- **pushTransaction() :**
  La fonction empiler ajoute une transaction en haut de la pile, met à jour le nombre d'éléments et enregistre l'opération dans l'historique
- **popTransaction() :**
  La fonction depiler retire et retourne la transaction en haut de la pile, en mettant à jour le nombre d'éléments, ou retourne NULL si la pile est vide.
- **afficherHistorique() :**
  La fonction afficherHistorique parcourt la pile des transactions et affiche, par client et par date, les produits achetés avec leurs quantités, prix, sous-totaux et le total du panier

4. Analyse de Complexité (1 page)

Recherche par hachage : rechercherProduitHachage() | moyenne : O(1) Complexité pire cas : O(n)

Explication : La fonction de hachage calcule directement l'indice (id % 13), permettant un accès instantané à la case correspondante. En cas de collision, il faut parcourir la liste chaînée de cette case. Avec une bonne distribution, le nombre de collisions reste faible.

- **Affichage : afficherListeProduits() | Complexité : O(n)**

Explication : Il faut visiter chaque nœud de la liste un par un, du début à la fin, donc proportionnel au nombre de produits.

- **Trie Fonctions : TrierListeProduitsPrix(), TrierListeProduitsQuantite(), TrierListeProduitsNom() = Complexité : O(n²)**

Explication : Utilisation du tri par sélection avec deux boucles imbriquées. Chaque élément est comparé avec tous les autres, ce qui génère n × n comparaisons.

- **Insertion d'un client : insererClient() | Complexité moyenne : O(log n) | pire cas : O(n)**

Explication : À chaque étape, on élimine la moitié de l'arbre (gauche ou droite) selon l'ordre alphabétique. Dans le pire cas (arbre dégénéré en liste), on doit parcourir tous les nœuds.

- **Recherche par nom : rechercherClientNom() | moyenne : O(log n) | pire cas : O(n)**

Explication : Même principe que l'insertion. La structure équilibrée de l'arbre permet de diviser l'espace de recherche à chaque niveau.

- **Suppression client: supprimerClientNom() | moyenne : O(log n) | pire cas : O(n)**

Explication : Recherche du nœud O(log n) + réorganisation locale O(1) = O(log n) au total.

- **Enfiler un client : enfilerClient() | Complexité : O(1)**

Explication : Ajout direct en fin de file grâce au pointeur queue. Aucun parcours nécessaire.

- **Défiler un client : defilerClient() | Complexité : O(1)**

Explication : Retrait direct du premier élément via le pointeur tete. Opération instantanée.

- **Empiler une transaction: empiler() | Complexité : O(1)**

Explication : Insertion en tête de pile. Opération immédiate sans parcours.

- **Dépiler une transaction: depiler() |Complexité : O(1)**

Explication : Retrait du premier élément (sommet de la pile). Accès direct.

5. Conclusion (½ page)

Ce projet nous a permis d'appliquer les notions fondamentales des structures de données à travers une application concrète. L'utilisation des listes chaînées, des tables de hachage, des arbres binaires, des files et des piles nous a aidés à mieux comprendre leur rôle et leur fonctionnement.

Nous avons renforcé nos compétences en programmation en langage C, notamment la manipulation des structures, la gestion de la mémoire dynamique, l'utilisation des pointeurs et la modularité du code. Toutefois, le manque de temps a constitué une difficulté importante, ainsi que certains problèmes techniques liés à la suppression des données et à la gestion des fichiers.

Avec plus de temps, des améliorations pourraient être apportées, comme l'amélioration de l'affichage, la détection des produits en rupture de stock, le calcul du chiffre d'affaires journalier et la génération d'un rapport de fin de journée.