



Fenofitia Nomenjanahary

Kajy University : Informatique

Algorithmes et structures de données

Author : Dimby Rabearivony

Date : 7 décembre 2024

Version : 1.1



"Ny fahalalana no valamparihiko."

Table des matières

1	Structures de données fondamentales	3
1.1	Variables	3
	Déclaration de variables	3
	Initialisation des variables	3
	Utilisation des variables	3
	Portée des variables	4
1.2	Pointeurs	5
	Déclaration de pointeurs	5
	Initialisation de pointeurs	5
	Utilisation de pointeurs	5
	Double pointeurs	6
	Utilisation avancée des pointeurs	6
	Pointeurs et mémoire dynamique	7
1.3	Types de données	7
	Types de données de base	7
	Types de données dérivés	8
	Typedef	9
1.4	Exercices	11
	Variables et types de données	11
	Pointeurs	12
	Tableaux	12
	Chaînes de caractères	12
	Structures	12
2	Algorithmes de tri et de recherche	14
2.1	Qu'est-ce qu'un algorithme ?	14
2.2	Récursion	14
2.3	Algorithmes de tri	15
2.4	Algorithmes de recherche	17
2.5	Exercices	17
3	Complexité algorithmique	20
3.1	Les bornes asymptotiques	20
3.2	La notation O	21
3.3	Les notations Ω et θ	21

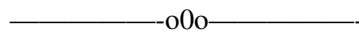
3.4	Complexité de certains algorithmes	22
	Temps constant ($O(1)$)	22
	Temps linéaire ($O(n)$)	22
	Temps quadratique ($O(n^2)$)	22
	Temps exponentiel ($O(2^n)$)	22
3.5	Exercices	23
4	Structures de données avancées	25
4.1	Listes chaînées	25
	Création d'une liste chaînée	25
	Insertion d'un nœud dans la liste chaînée	25
	Suppression d'un nœud de la liste chaînée	26
	Résumé	26
4.2	Piles	26
	Construction d'une pile	27
	Insertion d'un élément dans une pile (empilage)	27
	Suppression d'un élément dans une pile (dépileage)	27
	Vérification si la pile est vide	27
	Utilisations courantes des piles	28
	Résumé	28
4.3	Files	28
	Construction d'une file	29
	Insertion d'un élément dans une file (enfilage)	29
	Suppression d'un élément dans une file (défilage)	29
	Vérification si la file est vide	29
	Utilisations courantes des files	29
	Résumé	30
4.4	Arbres binaires	30
	Construction d'un arbre binaire	31
	Insertion dans un arbre binaire	31
	Utilisations courantes des arbres binaires	31
	Algorithmes de parcours des arbres binaires	31
	Résumé	32
4.5	Graphes	34
	Construction d'un graphe	35
	Ajout d'arêtes (liens) dans un graphe	35
	Utilisations courantes des graphes	35
	Algorithmes de parcours de graphes	35

Résumé	36
5 Applications d’algorithmes en IA	39
5.1 Introduction	39
5.2 Algorithmes d’Apprentissage Supervisé	39
Régression Linéaire	39
Régression Logistique	40
Clustering avec K-means	41
6 Corrigées des exercices	43
6.1 Structures de données fondamentales	43
Variables et types de données	43
Pointeurs	44
Tableaux	45
Chaînes de caractères	47
Structures	49
6.2 Algorithmes de tri et de recherche	51
Algorithmes de tri	51
Algorithmes de recherche	55
6.3 Complexité algorithmique	57
6.4 Structures de données avancées	58
Listes chaînées	58
Piles	61
Files	63
Arbres binaires	66
Graphes	68
6.5 Applications d’algorithmes en IA	72

Introduction

“En fait, je dirai que la différence entre un mauvais programmeur et un bon réside dans le fait qu’il considère son code ou ses structures de données comme plus importants. Les mauvais programmeurs se soucient du code. Les bons programmeurs se soucient des structures de données et de leurs relations.”

- Linus Torvalds



Bienvenue dans ce cours sur les structures de données et les algorithmes à Kajy University, Antananarivo, Madagascar. Ce cours est conçu pour les étudiants de première année qui souhaitent acquérir une solide compréhension des concepts fondamentaux en informatique. Les structures de données et les algorithmes constituent le fondement de la programmation et du développement logiciel, essentiels pour toute carrière en informatique ou en ingénierie logicielle.

L’objectif principal de ce cours est de vous familiariser avec les différentes structures de données, telles que les tableaux, les listes chaînées, les piles, les files, les arbres binaires, et les graphes. Vous apprendrez également les algorithmes associés à ces structures, tels que les algorithmes de tri, de recherche, et de parcours. Le cours abordera également la complexité algorithmique, un élément crucial pour évaluer l’efficacité des algorithmes.

Les structures de données et les algorithmes sont au cœur de nombreux aspects de l’informatique. Ils permettent de stocker, organiser, et manipuler des données de manière efficace. Comprendre ces concepts vous aidera à résoudre des problèmes complexes, à développer des programmes performants, et à améliorer vos compétences en programmation.

Voici ce que vous apprendrez au cours de ce semestre :

- **Structures de données fondamentales** : Vous apprendrez les concepts de base comme les variables, les pointeurs, les types de données, les tableaux, et les chaînes de caractères. Cela servira de fondation pour comprendre les structures de données avancées et les algorithmes.
- **Algorithmes de recherche et de tri** : Cette partie couvre les algorithmes courants de tri et de recherche, y compris le tri à bulles, le tri par insertion, le tri rapide, la recherche linéaire, et la recherche binaire. Ces algorithmes sont essentiels pour manipuler et organiser des données.
- **Complexité algorithmique** : Vous découvrirez les concepts de complexité algorithmique, y compris la notation O , Ω et Θ . Vous étudierez également la récursion et comprendrez comment elle affecte la complexité algorithmique.
- **Structures de données avancées** : Cette section traite des structures de données avancées comme les listes chaînées, les piles, les files, les arbres binaires, et les graphes. Vous apprendrez également des algorithmes de parcours et leurs applications.

- **Applications d’algorithmes en IA** : Cette dernière partie explore les applications pratiques des structures de données et des algorithmes dans le domaine de l’intelligence artificielle. Vous étudierez des algorithmes comme la régression linéaire, la régression logistique, et le clustering K-means. Ces algorithmes constituent une base fondamentale pour comprendre comment l’intelligence artificielle utilise les algorithmes pour résoudre des problèmes complexes.

Ce cours adopte une approche pratique. Vous aurez des sessions théoriques pour comprendre les concepts de base, suivies de travaux pratiques pour appliquer ces concepts dans des exercices et des projets. Vous travaillerez avec le langage C pour mettre en œuvre des structures de données et des algorithmes, et développer des compétences en programmation.

À la fin du cours, vous aurez une compréhension solide des structures de données et des algorithmes, et vous serez en mesure de les appliquer dans des contextes réels. Ces compétences vous seront utiles tout au long de vos études et de votre carrière professionnelle.

Nous sommes impatients de vous accompagner dans ce voyage d’apprentissage et de vous voir progresser dans le domaine passionnant des structures de données et des algorithmes. Bon cours !

Chapitre 1

Structures de données fondamentales

1.1 Variables

Les variables constituent l'un des concepts les plus fondamentaux en programmation. En langage C, une variable est un espace de stockage nommé qui peut contenir une valeur modifiable. Les variables sont utilisées pour stocker des données telles que des nombres, des caractères et des adresses mémoire.

Déclaration de variables

En langage C, on déclare une variable en précédant son nom le type de donnée qu'elle contient.

```
1  int age; // Declaration d'une variable de type entier appelee "age"
2  float prix; // Declaration d'une variable de type flottant appelee "
    prix"
3  char lettre ; // Declaration d'une variable de type caractere appelee "
    lettre"
```

Initialisation des variables

Les variables peuvent être initialisées lors de leur déclaration en leur attribuant une valeur initiale. Par exemple :

```
1  int nombre = 10; // Declaration et initialisation d'une variable de
    type entier avec la valeur 10
2  float pi = 3.14; // Declaration et initialisation d'une variable de
    type flottant avec la valeur 3.14
3  char grade = 'A'; // Declaration et initialisation d'une variable de
    type caractere avec la valeur 'A'
```

Utilisation des variables

Une fois déclarées et éventuellement initialisées, les variables peuvent être utilisées dans le programme pour stocker et manipuler des données. Par exemple :

```

1  #include <stdio.h>;
2  int x = 5;
3  int y = 10;
4  int somme = x + y; // Addition des valeurs des variables x et y
5  printf("La somme de %d et %d est %d\n", x, y, somme); // Affichage
    du resultat

```

Portée des variables

La portée d'une variable en C détermine où elle peut être utilisée dans le programme. Les variables peuvent être locales à une fonction, auquel cas elles ne sont accessibles que dans cette fonction, ou elles peuvent être globales, auquel cas elles sont accessibles dans tout le programme.

```

1  #include <stdio.h>;
2
3  int globalVar = 100; // Variable globale
4
5  void exampleFunction() {
6      int localVar = 50; // Variable locale a la fonction exampleFunction
7      printf("La variable globale est %d\n", globalVar); // Acces a la
        variable globale
8      printf("La variable locale est %d\n", localVar); // Acces a la
        variable locale
9  }
10
11 int main() {
12     printf("La variable globale est %d\n", globalVar); // Acces a la
        variable globale
13     // printf("La variable locale est %d\n", localVar); // Cela generera
        une erreur car localVar est locale a exampleFunction
14     exampleFunction();
15     return 0;
16 }

```

Les variables sont un élément essentiel en langage C et constituent la base de la manipulation des données dans les programmes. Il est crucial de comprendre leur déclaration, leur initialisation, leur utilisation et leur portée pour écrire des programmes efficaces et fonctionnels.

1.2 Pointeurs

Les pointeurs sont un concept essentiel en langage C. Un *pointeur* est une variable qui contient l'adresse mémoire d'une autre variable. En d'autres termes, un pointeur pointe vers l'emplacement en mémoire où une valeur est stockée.

Déclaration de pointeurs

En langage C, un pointeur est déclaré en précédant le nom de la variable avec l'opérateur *, qui indique que la variable est un pointeur. Voici un exemple de déclaration de pointeur :

```
1  int *ptr; // Declaration d'un pointeur vers un entier
2  float *ptr_float; // Declaration d'un pointeur vers un flottant
3  char *ptr_char; // Declaration d'un pointeur vers un caractere
```

Initialisation de pointeurs

Les pointeurs peuvent être initialisés avec l'adresse mémoire d'une variable existante à l'aide de l'opérateur d'adresse &. Voici un exemple d'initialisation de pointeur :

```
1  int var = 10; // Declaration et initialisation d'une variable
2  int *ptr; // Declaration d'un pointeur
3  ptr = &var; // Initialisation du pointeur avec l'adresse de la variable
    var
```

Utilisation de pointeurs

Une fois qu'un pointeur est initialisé, il peut être utilisé pour accéder à la valeur à laquelle il pointe ou pour modifier cette valeur. Voici quelques exemples :

```
1  #include <stdio.h>;
2  int var = 10; // Declaration et initialisation d'une variable
3  int *ptr; // Declaration d'un pointeur
4  ptr = &var; // Initialisation du pointeur avec l'adresse de la variable
    var
5  printf("La valeur de var est %d\n", var); // Affichage de la valeur de
    var
6  printf("L'adresse de var est %p\n", &var); // Affichage de l'adresse de
    var
7  printf("La valeur pointee par le pointeur est %d\n", *ptr); //
    Affichage de la valeur pointee par le pointeur
8  *ptr = 20; // Modification de la valeur pointee par le pointeur
```

```

9  printf("La nouvelle valeur de var est %d\n", var); // Affichage de la
    nouvelle valeur de var

```

Les pointeurs sont un concept puissant en langage C, mais ils nécessitent une manipulation prudente pour éviter les erreurs de segmentation et les fuites de mémoire.

Double pointeurs

Les double pointeurs (ou pointeurs vers des pointeurs) contiennent l'adresse mémoire d'un autre pointeur. Ils sont utiles pour passer des pointeurs par référence à des fonctions, pour des structures de données complexes comme des tableaux de tableaux, ou pour manipuler des listes chaînées.

Voici un exemple de déclaration et d'utilisation d'un double pointeur :

```

1  int a = 5;
2  int *ptr = &a; // Pointeur vers int
3  int **double_ptr = &ptr; // Double pointeur vers pointeur int
4
5  printf("La valeur de a est %d\n", a); // Affichage de la valeur de a
6  printf("La valeur pointée par ptr est %d\n", *ptr); // Affichage de la
    valeur pointée par ptr
7  printf("La valeur pointée par double_ptr est %d\n", **double_ptr);
    // Affichage de la valeur pointée par double_ptr

```

Les double pointeurs sont également utilisés pour retourner des pointeurs depuis des fonctions ou pour allouer dynamiquement de la mémoire pour des structures comme des tableaux 2D.

Utilisation avancée des pointeurs

Les pointeurs permettent des opérations avancées comme l'arithmétique des pointeurs, où vous pouvez ajouter ou soustraire des valeurs pour déplacer le pointeur en mémoire. Voici un exemple :

```

1  int tableau[5] = {1, 2, 3, 4, 5};
2  int *ptr = tableau;
3
4  printf("Premier element: %d\n", *ptr); // Affichage du premier
    element
5  ptr++; // Déplacement du pointeur vers le prochain element
6  printf("Deuxieme element: %d\n", *ptr); // Affichage du deuxieme
    element

```

Soyez prudent avec l'arithmétique des pointeurs, car elle peut conduire à des erreurs de segmentation si vous sortez des limites de la mémoire allouée.

Pointeurs et mémoire dynamique

Les pointeurs sont essentiels pour la gestion de la mémoire dynamique en langage C. Avec des fonctions comme 'malloc' et 'free', vous pouvez allouer et libérer de la mémoire au moment de l'exécution. Voici un exemple :

```

1  int *array = (int *)malloc(10 * sizeof(int)); // Allouer de la memoire
    pour 10 entiers
2  if (array == NULL) {
3      printf("Echec de l'allocation de memoire!\n");
4      return 1;
5  }
6
7  for (int i = 0; i < 10; i++) {
8      array[i] = i * 2; // Initialisation de la memoire allouee
9  }
10
11 free(array); // Liberer la memoire

```

La gestion de la mémoire est cruciale lors de l'utilisation des pointeurs. Assurez-vous de toujours libérer la mémoire allouée pour éviter les fuites de mémoire.

1.3 Types de données

Les types de données en langage C déterminent la nature des valeurs qu'une variable peut contenir. Le langage C prend en charge plusieurs types de données de base, ainsi que des types de données dérivés tels que les tableaux et les structures.

Types de données de base

Les types de données de base définissent les valeurs simples que peuvent contenir les variables en langage C. Voici quelques-uns des types de données de base les plus couramment utilisés, avec leurs tailles habituelles :

- a) **int** : Pour les entiers signés (généralement 4 octets).
- b) **float** : Pour les nombres à virgule flottante (généralement 4 octets).
- c) **double** : Pour les nombres à virgule flottante double précision (généralement 8 octets).
- d) **char** : Pour les caractères ASCII (généralement 1 octet).

Voici des exemples de déclaration de variables pour ces types de données :

```

1  int age = 30; // Declaration d'une variable de type entier
2  float poids = 75.5; // Declaration d'une variable de type flottant
3  double prix = 99.99; // Declaration d'une variable de type double
4  char grade = 'A'; // Declaration d'une variable de type caractere

```

Types de données dérivés

Outre les types de données de base, le langage C offre la possibilité de créer des types de données dérivés, comme les tableaux, les structures, les unions et les énumérations.

a) Tableaux :

Un tableau est une collection ordonnée d'éléments du même type. Les éléments d'un tableau sont accessibles via un index numérique. Voici un exemple de déclaration d'un tableau et d'opérations de base :

```

1  int tableau[5]; // Declaration d'un tableau de taille 5
2  tableau[0] = 10; // Attribuer la valeur 10 au premier element
3  tableau[1] = 20; // Attribuer la valeur 20 au deuxieme element

```

b) Structures :

Les structures permettent de regrouper des variables de types différents. Elles permettent de définir des types de données personnalisés. Voici un exemple d'utilisation des structures :

```

1  #include <stdio.h>;
2  #include <string.h>;
3
4  struct Personne {
5      char nom[50];
6      int age;
7      float taille ;
8  };
9
10 struct Personne p1; // Declaration d'une structure de type
    Personne
11 strcpy(p1.nom, "John Doe"); // Attribution d'une valeur au
    champ nom
12 p1.age = 30; // Attribution d'une valeur au champ age
13 p1.taille = 1.75; // Attribution d'une valeur au champ taille

```

c) Unions :

Les unions permettent de partager la même zone mémoire entre différents types de données. Voici un exemple de déclaration d'une union :

```

1  union Data {
2      int entier ;
3      float flottant ;
4      char caractere ;
5  };
6
7  union Data d;
8  d.entier = 10; // Utilisation de l'union avec un entier
9  d.flottant = 5.5; // Utilisation avec un flottant (ecrase l'
    entier )

```

d) Énumérations :

Les énumérations permettent de définir un ensemble de valeurs nommées. Elles sont utiles pour créer des listes de constantes. Voici un exemple d'utilisation des énumérations :

```

1  enum Jour {LUNDI, MARDI, MERCREDI, JEUDI,
    VENDREDI, SAMEDI, DIMANCHE};
2
3  enum Jour aujourd'hui = MERCREDI; // Declaration d'une
    enumeration et initialisation

```

Typedef

Le mot-clé *typedef* permet de créer des alias pour des types de données existants. Cela peut simplifier des déclarations complexes, rendre le code plus lisible, et améliorer l'abstraction des types. 'typedef' est couramment utilisé pour simplifier des déclarations de pointeurs, de structures, d'unions, et de fonctions.

Alias pour des types de base

Voici un exemple d'utilisation de 'typedef' pour créer un alias pour un type de base :

```

1  typedef int Entier; // Creation d'un alias pour int
2  Entier age = 30; // Utilisation du typedef pour creer un alias

```

Cet exemple montre comment 'typedef' peut simplifier des déclarations. Utiliser des alias descriptifs peut rendre le code plus lisible et compréhensible, en particulier lorsque les types de base ont des applications spécifiques.

Utilisation avec des pointeurs

Les pointeurs en C peuvent être complexes à déclarer, surtout pour les pointeurs de fonctions ou les structures dérivées. ‘typedef’ simplifie les déclarations de pointeurs en créant des alias :

```
1 typedef int* PointeurEntier; // Alias pour un pointeur vers int
2 PointeurEntier ptr = NULL; // Declaration d'un pointeur vers int
```

Vous pouvez également utiliser ‘typedef’ pour simplifier des déclarations de pointeurs de fonctions, rendant le code plus lisible :

```
1 typedef int (*Fonction)(int, int); // Alias pour un pointeur vers une
   fonction
```

Utilisation avec des structures

‘typedef’ est largement utilisé pour simplifier les déclarations de structures complexes. Cela facilite l’utilisation de structures pour regrouper des variables de types différents :

```
1 typedef struct {
2     char nom[50];
3     int age;
4     float taille ;
5 } Personne;
6
7 Personne p1; // Utilisation du typedef pour nommer une structure
```

Avec ‘typedef’, les déclarations de structures deviennent plus concises et lisibles, ce qui peut simplifier le code et faciliter les modifications futures.

Utilisation avec des unions

Les unions permettent de partager la même zone mémoire entre différents types de données. ‘typedef’ peut être utilisé pour nommer des unions, ce qui simplifie l’utilisation des structures complexes :

```
1 typedef union {
2     int entier ;
3     float flottant ;
4     char caractere;
5 } Donnees;
6
7 Donnees d; // Utilisation du typedef pour nommer une union
```

Utilisation avec des énumérations

Les énumérations permettent de définir des ensembles de valeurs nommées. ‘typedef’ facilite la déclaration et l’utilisation des énumérations :

```
1  typedef enum {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI,
2      SAMEDI, DIMANCHE} Jour;
3  Jour aujourd'hui = MERCREDI; // Utilisation de l'enumeration avec
    typedef
```

Avantages de ‘typedef’

‘typedef’ offre plusieurs avantages pour le code C, notamment :

- **Simplification des déclarations complexes** : ‘typedef’ permet de rendre le code plus concis en créant des alias pour des déclarations compliquées.
- **Lisibilité accrue** : Les alias créés avec ‘typedef’ rendent le code plus facile à lire et à comprendre.
- **Abstraction des types** : ‘typedef’ permet de créer des abstractions pour des types de données, ce qui facilite les modifications futures.
- **Réutilisation du code** : Les alias créés avec ‘typedef’ peuvent être réutilisés dans différents contextes, ce qui améliore la maintenabilité du code.

Dans l’ensemble, ‘typedef’ est un outil puissant en langage C qui permet de simplifier les déclarations complexes, de rendre le code plus lisible, et de faciliter la création de types de données personnalisés. Que ce soit pour des pointeurs, des structures, des unions, ou des fonctions, ‘typedef’ est un outil précieux pour les développeurs C.

1.4 Exercices

Variables et types de données

1. Quelle est la différence entre les types de données suivants :
 - (a). **int**
 - (b). **float**
 - (c). **char**
2. Écrivez un programme C qui déclare des variables pour chaque type de données mentionné et affiche leurs valeurs.
3. Écrivez un programme C qui utilise des variables de différents types de données pour calculer la somme d’un entier et d’un flottant, puis affichez le résultat.
4. Quelles sont les limites de valeurs pour les types de données suivants :
 - (a). **int**
 - (b). **float**

(c). double

Indication : Considérer le nombre de bits et octets. Par exemple, un caractère ASCII est représenté par un octet, c'est-à-dire 8 bits. Le caractère "A" est représenté par 01000001. Avec huit chiffres de 0 et 1, on peut représenter $2^8 = 256$ valeurs. Pour les nombres signés (int, float, double, ...), l'opposé est représenté différemment.

Pointeurs

1. Quelle est la différence entre une variable et un pointeur ?
2. Écrivez un programme C qui utilise des pointeurs pour modifier la valeur d'une variable.
3. Écrivez un programme C qui utilise des pointeurs pour échanger les valeurs de deux variables.
4. Expliquez ce qu'est un pointeur nul (null pointer) et ce qui se passe si vous tentez d'utiliser un pointeur nul.

Tableaux

1. Déclarez un tableau de 5 entiers et initialisez-le avec des valeurs. Écrivez un programme C pour afficher les valeurs du tableau.
2. Modifiez le programme pour inverser les éléments du tableau et afficher le tableau inversé.
3. Écrivez un programme C qui calcule la somme de tous les éléments d'un tableau d'entiers de longueur 'n'.
4. Que se passe-t-il si vous essayez d'accéder à un index hors limites (out of bounds) dans un tableau C ? Écrivez un programme pour le démontrer.

Chaînes de caractères

1. Quelle est la particularité des chaînes de caractères en C par rapport aux autres types de données ?
2. Écrivez un programme C qui déclare une chaîne de caractères et utilise la bibliothèque 'string.h' pour copier du texte dans la chaîne.
3. Écrivez un programme C qui concatène deux chaînes de caractères en utilisant 'strcat'.
4. Comment compare-t-on deux chaînes de caractères en C ? Écrivez un programme qui compare deux chaînes et indique si elles sont identiques ou non.

Structures

1. Déclarez une structure en C pour représenter une personne avec un nom, un âge, et une taille.
2. Écrivez un programme C qui utilise cette structure pour créer une personne, assigner des valeurs, et afficher les informations de la personne.

3. Écrivez un programme C qui utilise des structures pour représenter un point 3D avec des coordonnées x , y , z . Ensuite, écrivez une fonction qui calcule la distance entre deux points 3D.
4. Créez une structure pour représenter une voiture avec un nom, une année, et un prix. Écrivez un programme C qui utilise cette structure pour créer une voiture et afficher ses propriétés.

Chapitre 2

Algorithmes de tri et de recherche

2.1 Qu'est-ce qu'un algorithme ?

Un *algorithme* est une série d'étapes visant à résoudre un problème ou à accomplir une tâche spécifique. En informatique, les algorithmes permettent de manipuler, trier, rechercher, et transformer des données. Pour qu'un algorithme soit efficace, il doit être :

1. **Correct** : Donner le bon résultat pour tous les cas.
2. **Efficace** : Rapide et utilisant des ressources raisonnables.
3. **Simple** : Compréhensible et facile à mettre en œuvre.
4. **Flexible** : Adaptable à différentes situations.

Un exemple simple d'algorithme qui trouve le maximum de deux nombres :

Algorithm 1 Trouver le maximum de deux nombres

```
Si  $a > b$  Alors  
    retour a  
Sinon  
    retour b  
Fin Si
```

2.2 Récursion

La récursion est un concept fondamental en informatique où une fonction s'appelle elle-même. C'est souvent utilisé pour résoudre des problèmes qui peuvent être décomposés en sous-problèmes similaires. Les fonctions récursives nécessitent une condition de terminaison pour éviter des appels infinis, ce qui pourrait entraîner des débordements de pile et des erreurs critiques.

Un exemple classique de récursion est le calcul de la factorielle d'un nombre. Voici un algorithme qui montre comment la récursion fonctionne pour la factorielle :

Dans cet exemple, la fonction 'factorielle' utilise la récursion pour multiplier un nombre par la factorielle du nombre précédent, jusqu'à ce qu'elle atteigne le cas de base ($n == 0$). La condition de terminaison empêche la récursion infinie et garantit que la fonction finit par retourner une valeur.

Les fonctions récursives peuvent entraîner des complexités spatiales élevées en raison de l'utilisation de la pile pour stocker les appels récursifs. Chaque appel de fonction récursive crée un

Algorithm 2 Calcul de la factorielle d'un nombre

```

function FACTORIELLE(n)
  Si n == 0 Alors
    Retour 1
  Sinon
    Retour n × FACTORIELLE(n - 1)
  Fin Si
Fin function

```

nouveau contexte d'exécution dans la pile, ce qui peut augmenter l'utilisation de la mémoire. Pour éviter des débordements de pile, il est essentiel d'avoir des conditions de terminaison robustes et de gérer la profondeur de récursion.

Exemples d'autres problèmes résolus par la récursion

Calculer la suite de Fibonacci : La suite de Fibonacci peut être calculée récursivement en additionnant les deux termes précédents, avec des cas de base pour les premiers termes. Rappelons la définition de la suite de Fibonacci :

$$\begin{cases} U_0 = U_1 = 1 \\ U_{n+2} = U_{n+1} + U_n \quad \text{pour } n \geq 0. \end{cases}$$

La récursion est un outil puissant pour résoudre des problèmes de manière élégante, mais elle doit être utilisée avec précaution pour éviter des complications liées à la mémoire et à la performance.

2.3 Algorithmes de tri

Le *tri* consiste à organiser les données dans un ordre particulier. Les algorithmes de tri couramment utilisés comprennent :

1. **Tri à bulles (Bubble Sort)** : Cet algorithme compare des éléments adjacents (côte à côte) et les échange s'ils ne sont pas dans le bon ordre. On répète le processus jusqu'à ce que toute la liste soit triée. C'est le plus simple des algorithmes de tri, mais il n'est pas très efficace pour les grandes listes, car il doit comparer de nombreux éléments.
2. **Tri par insertion (Insertion Sort)** : Cet algorithme place chaque élément à la bonne position dans la liste, un par un. C'est efficace pour les petites listes ou les listes presque triées, car il insère chaque nouvel élément à sa position correcte sans avoir besoin de beaucoup de déplacements.
3. **Tri rapide (QuickSort)** : Cet algorithme utilise la technique de division et conquête. Il choisit un pivot (un élément central), divise la liste en deux parties (éléments plus petits d'un côté, plus grands de l'autre), puis trie chaque partie séparément. Cela en fait un algorithme rapide et efficace pour les grandes listes.

4. **Tri fusion (MergeSort)** : Cet algorithme divise la liste en deux moitiés, trie chaque moitié séparément, puis les fusionne pour créer une liste triée. C'est un tri stable, ce qui signifie qu'il maintient l'ordre relatif des éléments égaux. C'est souvent utilisé pour les grandes listes car il offre de bonnes performances.

Algorithm 3 Tri à bulles

```

1 : Tant que non triée Faire
2 :   Pour i de 0 à n - 2 Faire
3 :     Pour j de 0 à n - i - 2 Faire
4 :       Si tableau[j] > tableau[j + 1] Alors
5 :         échanger tableau[j] et tableau[j + 1]
6 :       Fin Si
7 :     Fin Pour
8 :   Fin Pour
9 : Fin Tant que
  
```

Algorithm 4 Tri par insertion

```

1 : Pour i de 1 à n - 1 Faire
2 :   clé := tableau[i]
3 :   j := i - 1
4 :   Tant que j >= 0 et tableau[j] > clé Faire
5 :     tableau[j + 1] = tableau[j]
6 :     j = j - 1
7 :   Fin Tant que
8 :   tableau[j + 1] = clé
9 : Fin Pour
  
```

Algorithm 5 Tri rapide (QuickSort)

```

1 : function QUICKSORT(tableau, bas, haut)
2 :   Si bas < haut Alors
3 :     pivot := PARTITION(tableau, bas, haut)
4 :     QUICKSORT(tableau, bas, pivot - 1)
5 :     QUICKSORT(tableau, pivot + 1, haut)
6 :   Fin Si
7 : Fin function
  
```

Les algorithmes de tri sont essentiels pour organiser les données efficacement. Chacun de ces algorithmes a ses avantages et inconvénients, avec des applications spécifiques en fonction des contraintes de temps et d'espace. Le choix de l'algorithme de tri dépendra de nombreux facteurs, y compris la taille de la liste, les caractéristiques des données, et les exigences de performance.

Tâche : Montrer que chacune des algorithmes de tri ci-dessus se termine.

Algorithm 6 Tri fusion (MergeSort)

```

1 : function MERGESORT(tableau, gauche, droite)
2 :   Si gauche < droite Alors
3 :     milieu := (gauche + droite) / 2
4 :     MERGESORT(tableau, gauche, milieu)
5 :     MERGESORT(tableau, milieu + 1, droite)
6 :     FUSION(tableau, gauche, milieu, droite)
7 :   Fin Si
8 : Fin function

```

2.4 Algorithmes de recherche

Les algorithmes de recherche permettent de trouver des éléments spécifiques dans un ensemble de données. Voici quelques-uns des types courants d'algorithmes de recherche, avec des explications détaillées et des exemples :

1. **Recherche linéaire (Linear Search)** : Ce type de recherche parcourt une liste d'éléments un par un, de manière séquentielle, jusqu'à ce qu'il trouve l'élément recherché ou atteigne la fin de la liste. C'est le plus simple des algorithmes de recherche, mais il peut être lent pour les grandes listes.
2. **Recherche binaire (Binary Search)** : La recherche binaire utilise une technique de division pour trouver rapidement un élément dans une liste triée. Elle divise la liste en moitiés, recherche le milieu, puis réduit le domaine de recherche en fonction de la valeur recherchée. Cela la rend plus rapide que la recherche linéaire, mais elle nécessite que la liste soit triée.
3. **Recherche interpolée (Interpolation Search)** : Cette recherche est une version améliorée de la recherche binaire pour les listes avec des valeurs bien réparties. Elle utilise une formule pour estimer la position de la valeur recherchée, ce qui la rend plus précise.
4. **Recherche par saut (Jump Search)** : C'est une recherche linéaire optimisée pour les grandes listes. Elle saute de blocs en blocs pour trouver la plage dans laquelle l'élément pourrait se trouver, puis fait une recherche linéaire à l'intérieur de cette plage.

Algorithm 7 Recherche linéaire

```

1 : Pour i de 0 à n - 1 Faire
2 :   Si tableau[i] == cible Alors
3 :     Retourner i
4 :   Fin Si
5 : Fin Pour
6 : Retourner -1

```

2.5 Exercices

1. Pour chacun des algorithmes de tri cités dans ce chapitre :

Algorithm 8 Recherche binaire

```

gauche := 0
droite := n - 1
Tant que gauche <= droite Faire
    milieu := gauche + (droite - gauche) / 2
    Si tableau[milieu] == cible Alors
        Retourner milieu
    Sinon Si tableau[milieu] < cible Alors
        gauche := milieu + 1
    Sinon
        droite := milieu - 1
    Fin Si
Fin Tant que
Retourner -1

```

Algorithm 9 Recherche interpolée

```

1 : bas := 0
2 : haut := n - 1
3 : Tant que bas <= haut et cible >= tableau[bas] et cible <= tableau[haut] Faire
4 :     pos := bas + ((cible - tableau[bas]) ÷ (tableau[haut] - tableau[bas])) × (haut - bas)
5 :     Si tableau[pos] == cible Alors
6 :         Retourner pos
7 :     Sinon Si tableau[pos] < cible Alors
8 :         bas := pos + 1
9 :     Sinon
10 :        haut := pos - 1
11 :    Fin Si
12 : Fin Tant que
13 : Retourner -1

```

Algorithm 10 Recherche par saut

```

1 : tailleBloc := Racine n
2 : i := 0
3 : Tant que i < n et tableau[i] < cible Faire
4 :     i := i + tailleBloc
5 : Fin Tant que
6 : Pour j de i - tailleBloc à Min(i, n) Faire
7 :     Si tableau[j] == cible Alors
8 :         Retourner j
9 :     Fin Si
10 : Fin Pour
11 : Retourner -1

```

- (a). Implémentez l'algorithme en C.
 - (b). Testez votre code avec les cas suivants :
 - I. Un tableau vide.
 - II. Un tableau contenant un seul élément.
 - III. Un tableau déjà trié.
 - IV. Le tableau [9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7].
2. Pour chacun des algorithmes de recherche cités dans ce chapitre :
- (a). Implémentez l'algorithme en C.
 - (b). Testez votre code en cherchant les éléments 8, 1, et 0 dans le tableau [9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7].
 - (c). Lequel de ces algorithmes n'est pas applicable au tableau précédent ? Pourquoi ?
3. Testez votre implémentation de la recherche binaire en cherchant les éléments 0, 2, 7 et 9 dans le tableau [0, 1, 4, 4, 4, 5, 6, 7, 9, 9].

Chapitre 3

Complexité algorithmique

La complexité algorithmique permet d'évaluer l'efficacité des algorithmes en termes de temps d'exécution et de consommation de ressources. Cela aide à comprendre combien de temps un algorithme prendra pour des entrées de tailles différentes. Les notations couramment utilisées pour décrire la complexité algorithmique sont O , Ω , et θ .

3.1 Les bornes asymptotiques

Les bornes asymptotiques, ou "asymptotic bounds," fournissent une approximation de la performance d'un algorithme pour des tailles d'entrée importantes. Elles aident à évaluer le comportement des algorithmes sous différents scénarios : meilleur cas, pire cas, et cas moyen.

- **Borne supérieure (Big O)** : Représentée par la notation $O(f(n))$, cette borne donne une estimation de la complexité maximale d'un algorithme, indiquant qu'il n'aura jamais une complexité supérieure à $O(f(n))$. Mathématiquement, cela signifie qu'il existe des constantes c et n_0 telles que :

$$T(n) \leq c \cdot f(n), \quad \text{pour tout } n \geq n_0.$$

- **Borne inférieure (Big Omega)** : Représentée par la notation $\Omega(f(n))$, cette borne donne une estimation de la complexité minimale, indiquant que l'algorithme ne peut pas être plus rapide que cette borne. Cela signifie qu'il existe des constantes c et n_0 telles que :

$$T(n) \geq c \cdot f(n), \quad \text{pour tout } n \geq n_0.$$

- **Borne moyenne (Big Theta)** : Représentée par la notation $\theta(f(n))$, cette borne indique une complexité moyenne ou attendue. Cela signifie que l'algorithme a une borne supérieure et inférieure qui convergent vers la même fonction. Mathématiquement, cela signifie qu'il existe des constantes c_1 , c_2 , et n_0 telles que :

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n), \quad \text{pour tout } n \geq n_0.$$

3.2 La notation O

La notation "O" (grand O) indique la complexité maximale d'un algorithme en termes de temps ou d'espace. Elle fournit une estimation du nombre maximum d'opérations nécessaires en fonction de la taille de l'entrée. Par exemple, la recherche binaire a une complexité de ' $O(\log n)$ ', car chaque étape réduit le problème de moitié.

Algorithm 11 Recherche binaire

```

1 : function RECHERCHE_BINAIRE(tableau, valeur)
2 :   gauche := 0
3 :   droite := taille(tableau) - 1
4 :   Tant que gauche <= droite Faire
5 :     milieu := (gauche + droite) / 2
6 :     Si tableau[milieu] == valeur Alors
7 :       Retourner milieu
8 :     Sinon Si tableau[milieu] < valeur Alors
9 :       gauche := milieu + 1
10 :    Sinon
11 :      droite := milieu - 1
12 :    Fin Si
13 :  Fin Tant que
14 :  Retourner -1
15 : Fin function

```

3.3 Les notations Ω et θ

Ces notations complètent la notation "O" pour donner une vision complète de la complexité algorithmique.

- **Borne inférieure (Ω)** : Représente la complexité minimale. Par exemple, la complexité de l'algorithme de somme de tableau est $\Omega(n)$, car chaque élément doit être parcouru.
- **Borne moyenne (θ)** : Indique la complexité moyenne ou attendue, définissant à la fois une borne inférieure et une borne supérieure qui convergent. Pour le même algorithme de somme de tableau, la complexité est $\theta(n)$.

Voici un exemple de la somme des éléments d'un tableau :

Algorithm 12 Somme de tableau

```

1 : function SOMME(tableau)
2 :   somme := 0
3 :   Pour i de 0 à taille(tableau) - 1 Faire
4 :     somme := somme + tableau[i]
5 :   Fin Pour
6 :   Retourner somme
7 : Fin function

```

3.4 Complexité de certains algorithmes

Voici quelques exemples d'algorithmes qui illustrent des complexités courantes, allant du temps constant à l'exponentiel.

Temps constant ($O(1)$)

Les algorithmes à temps constant ont une complexité qui ne dépend pas de la taille de l'entrée. Par exemple, accéder à un élément spécifique d'un tableau par son index a une complexité ' $O(1)$ '. Cela signifie qu'il y a toujours une seule opération, quelle que soit la taille du tableau.

Algorithm 13 Accès direct à un élément du tableau

```

1 : function ACCÈS_DIRECT(tableau, index)
2 :   Retourner tableau[index]
3 : Fin function

```

Temps linéaire ($O(n)$)

Les algorithmes à temps linéaire ont une complexité qui augmente proportionnellement avec la taille de l'entrée. Par exemple, le calcul de la somme des éléments d'un tableau a une complexité ' $O(n)$ ', car il faut parcourir tous les éléments.

Algorithm 14 Calcul de la somme des éléments d'un tableau

```

1 : function SOMME(tableau)
2 :   somme := 0
3 :   Pour i de 0 à taille(tableau) - 1 Faire
4 :     somme := somme + tableau[i]
5 :   Fin Pour
6 :   Retourner somme
7 : Fin function

```

Temps quadratique ($O(n^2)$)

Les algorithmes à temps quadratique ont une complexité qui augmente avec le carré de la taille de l'entrée. Par exemple, le tri par insertion a une complexité ' $O(n^2)$ ', car chaque élément doit être comparé avec tous les éléments précédents.

Temps exponentiel ($O(2^n)$)

Les algorithmes à temps exponentiel ont une complexité qui augmente de manière exponentielle avec la taille de l'entrée. Par exemple, un algorithme récursif pour résoudre le problème de la tour de Hanoï a une complexité ' $O(2^n)$ ', car chaque mouvement peut générer de multiples autres mouvements.

Algorithm 15 Tri par insertion

```

1 : Pour  $i$  de 1 à  $n - 1$  Faire
2 :   clé := tableau[ $i$ ]
3 :    $j := i - 1$ 
4 :   Tant que  $j \geq 0$  et tableau[ $j$ ] > clé Faire
5 :     tableau[ $j + 1$ ] = tableau[ $j$ ]
6 :      $j = j - 1$ 
7 :   Fin Tant que
8 :   tableau[ $j + 1$ ] = clé
9 : Fin Pour

```

Algorithm 16 Tour de Hanoï

```

1 : function HANOI( $n$ , source, cible, intermédiaire)
2 :   Si  $n == 1$  Alors
3 :     Déplacer un disque de la source vers la cible
4 :   Sinon
5 :     HANOI( $n - 1$ , source, intermédiaire, cible)
6 :     Déplacer un disque de la source vers la cible
7 :     HANOI( $n - 1$ , intermédiaire, cible, source)
8 :   Fin Si
9 : Fin function

```

3.5 Exercices

- Quelle est la complexité temporelle des opérations suivantes ?
 - Parcourir un tableau d'entiers de longueur n .
 - Multiplier deux matrices de taille $n \times n$.
 - Effectuer une recherche binaire sur un tableau trié.
- Quelle est la complexité temporelle du code suivant ?

```

1   for (int i = 0; i < n; i++) {
2     for (int j = 0; j < n; j++) {
3       printf("%d ", i * j);
4     }
5   }

```

Comment pouvez-vous améliorer cet algorithme pour réduire sa complexité ?

- Analysez le code suivant pour déterminer sa complexité temporelle :

```

1   void operationSurTableau(int arr[], int n) {
2     for (int i = 0; i < n; i++) {
3       arr[i] = arr[i] * 2;
4     }
5
6     for (int i = 0; i < n; i++) {

```

```
7     arr[i] = arr[i] + 1;  
8     }  
9 }
```

4. Expliquez ce que signifie $O(n \log n)$. Donnez un exemple d'algorithme qui a une complexité de $O(n \log n)$ et expliquez pourquoi. (*Indication* : Tri fusion)
5. Quel est le type de complexité associé aux opérations suivantes ?
 - (a). Additionner deux entiers.
 - (b). Additionner les éléments d'un tableau de longueur 'n'.
 - (c). Effectuer un tri à bulle sur un tableau de longueur 'n'.
 - (d). Effectuer un tri par insertion sur un tableau de longueur 'n'.

Chapitre 4

Structures de données avancées

4.1 Listes chaînées

Les *listes chaînées* sont des structures de données composées de *nœuds*, où chaque nœud contient des données et un pointeur vers le nœud suivant. Elles sont utiles pour des opérations d'insertion et de suppression rapides sans nécessiter de réallocation de mémoire.

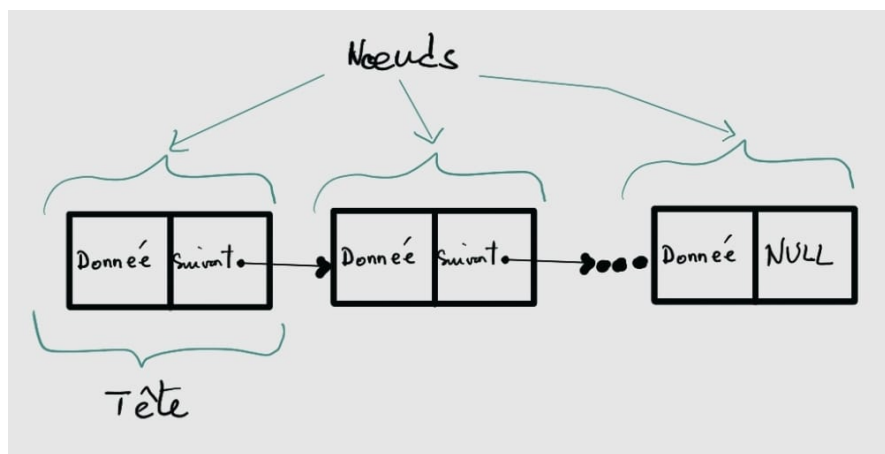


FIGURE 4.1 – Illustration d'une liste chaînée.

Création d'une liste chaînée

Pour créer une liste chaînée, commencez par définir la structure de base du nœud. Chaque nœud contient une donnée (comme un entier) et un pointeur vers le nœud suivant. Initialisez la tête de la liste à NULL pour indiquer qu'elle est vide.

Insertion d'un nœud dans la liste chaînée

Pour insérer un nœud dans une liste chaînée, vous avez deux approches principales :

- **Insérer au début de la liste** : Créez un nouveau nœud, définissez ses données, puis faites pointer ce nœud vers l'ancienne tête de la liste. Mettez ensuite à jour la tête pour qu'elle pointe vers le nouveau nœud.
- **Insérer à une position spécifique** : Parcourez la liste jusqu'à la position souhaitée, créez un nouveau nœud, puis ajustez les pointeurs pour insérer le nouveau nœud à la position correcte. Si la position est zéro, insérez au début de la liste.

Suppression d'un nœud de la liste chaînée

Pour supprimer un nœud d'une liste chaînée, vous avez deux approches principales :

- **Suppression d'un nœud par clé** : Recherchez un nœud avec une donnée spécifique. Si le nœud à supprimer est la tête de la liste, mettez à jour la tête pour qu'elle pointe vers le nœud suivant. Sinon, ajustez les pointeurs pour déconnecter le nœud du reste de la liste.
- **Suppression d'un nœud à une position spécifique** : Parcourez la liste jusqu'à la position spécifiée. Ajustez les pointeurs pour retirer le nœud à cette position. Si la position est hors limites, signalez une erreur.

Résumé

Les listes chaînées offrent une structure flexible pour stocker des données et permettent des opérations d'insertion et de suppression efficaces. L'utilisation de pointeurs est essentielle pour manipuler la structure. Il est également crucial de gérer correctement la mémoire, surtout lors de la suppression de nœuds ou de la libération de la liste entière.

Exercices

1. Création :

- Écrivez un programme en C pour créer une liste chaînée contenant les nombres suivants : 10, 20, 30.
- Affichez les éléments de la liste chaînée dans l'ordre.

2. Insertion :

- Ajoutez un élément 40 à la fin de la liste chaînée existante.
- Ajoutez un élément 5 au début de la liste chaînée.

3. Suppression :

- Supprimez l'élément 20 de la liste chaînée.
- Supprimez l'élément à la position 2 (indice basé sur 0).

4.2 Piles

Les *piles* (ou LIFO, Last In First Out) sont des structures de données où le dernier élément inséré est le premier à être retiré. Les piles sont utiles pour des opérations de retour en arrière, comme dans les navigateurs Web ou les algorithmes de récursion.

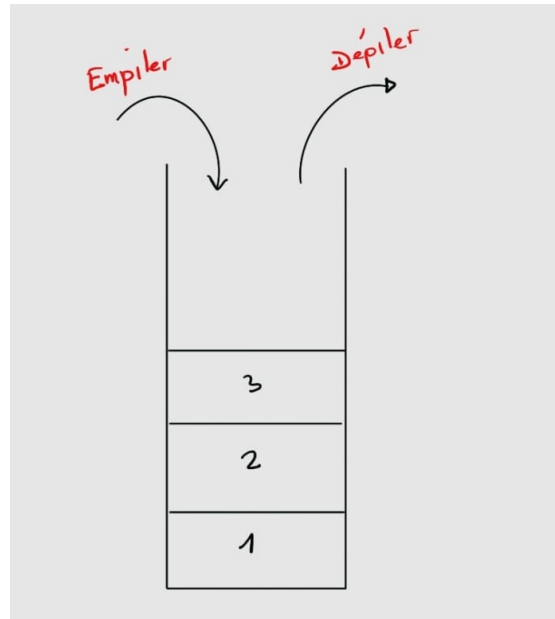


FIGURE 4.2 – Illustration d’une pile (LIFO), où le dernier élément ajouté est le premier retiré.

Construction d’une pile

Pour construire une pile, vous devez définir une structure qui contient des éléments (comme un tableau ou une liste) et un pointeur ou un indice qui indique le sommet de la pile. Les opérations principales associées aux piles sont l’empilage (ajout d’un élément au sommet) et le dépilage (retrait de l’élément au sommet).

Insertion d’un élément dans une pile (empilage)

L’empilage consiste à ajouter un élément au sommet de la pile. Pour cela, vous devez vérifier si la pile a de la place (si elle n’est pas pleine). Mettez ensuite à jour le sommet pour refléter le nouvel élément. Si vous utilisez un tableau, insérez l’élément à l’indice du sommet. Pour une pile basée sur des listes chaînées, créez un nouveau nœud et ajustez les pointeurs pour insérer le nœud au sommet.

Suppression d’un élément dans une pile (dépilage)

Le dépilage consiste à retirer l’élément au sommet de la pile. Comme pour l’empilage, vérifiez s’il y a des éléments dans la pile (pour éviter de dépiler une pile vide). Si la pile est vide, renvoyez une erreur ou une valeur spéciale pour indiquer qu’il n’y a rien à dépiler. Mettez ensuite à jour le sommet pour refléter le nouvel élément au sommet. Si vous utilisez un tableau, réduisez le sommet d’un indice.

Vérification si la pile est vide

Pour vérifier si la pile est vide, comparez le sommet à -1. Si le sommet est à -1, la pile est vide. Cette vérification est essentielle pour éviter des opérations incorrectes comme le dépilage

d'une pile vide, ce qui pourrait causer des erreurs.

Utilisations courantes des piles

Les piles sont utilisées dans de nombreux contextes, tels que les opérations de retour en arrière, les algorithmes de récursion, et les vérifications de syntaxe. Par exemple, les piles sont utiles pour vérifier si des parenthèses sont équilibrées dans une expression mathématique ou de programmation. Elles sont également utilisées dans les navigateurs Web pour gérer l'historique des pages visitées.

Résumé

Les piles offrent une structure simple mais puissante pour gérer des opérations où le dernier entré est le premier à sortir. Elles sont utilisées dans de nombreux domaines informatiques, y compris les algorithmes de récursion et les éditeurs de texte. Comprendre leur fonctionnement et savoir comment les utiliser efficacement est essentiel pour tout développeur.

Exercices

1. Création :

- (a). Implémentez une pile en C à l'aide d'un tableau de taille 5. Ajoutez les éléments suivants : 10, 20, 30.
- (b). Affichez les éléments actuels de la pile.

2. Empilage (Push) :

- (a). Ajoutez deux nouveaux éléments 40 et 50 dans la pile.
- (b). Essayez d'ajouter un élément 60 et gérez l'erreur si la pile est pleine.

3. Dépilage (Pop) :

- (a). Supprimez le dernier élément inséré et affichez l'état actuel de la pile.
- (b). Continuez à dépiler jusqu'à ce que la pile soit vide et gérez l'erreur si la pile est déjà vide.

4. Vérification :

- (a). Ajoutez une vérification pour savoir si la pile est vide ou pleine avant chaque opération.

4.3 Files

Les *files* (ou FIFO, First In First Out) sont des structures de données où le premier élément inséré est le premier à sortir. Contrairement aux piles, où le dernier élément ajouté est le premier à sortir, les files fonctionnent comme des files d'attente dans un magasin, où les premiers arrivés sont les premiers servis.

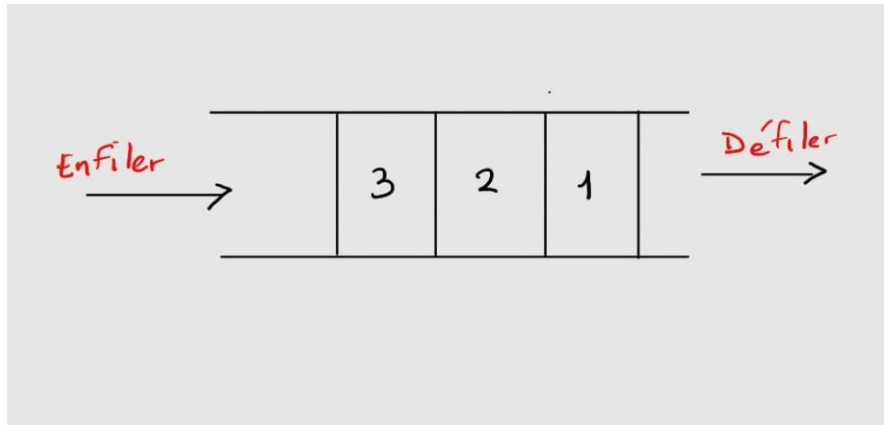


FIGURE 4.3 – Illustration d’une file (FIFO), où le premier élément ajouté est le premier retiré.

Construction d’une file

Pour construire une file, définissez une structure qui contient les éléments de la file et des pointeurs vers le début (avant) et la fin (arrière). Cette structure permet des opérations d’enfilage (ajout d’un élément à l’arrière) et de défilage (retrait du premier élément).

Insertion d’un élément dans une file (enfilage)

L’enfilage consiste à ajouter un élément à la fin de la file. Pour cela, assurez-vous qu’il y a de l’espace pour l’élément. Si la file est vide, l’élément ajouté devient à la fois l’avant et l’arrière. Si la file contient déjà des éléments, faites pointer le dernier élément vers le nouveau, puis mettez à jour l’arrière pour indiquer le nouvel élément.

Suppression d’un élément dans une file (défilage)

Le défilage consiste à retirer le premier élément de la file. Avant de le retirer, vérifiez que la file n’est pas vide. Si la file est vide, renvoyez une erreur ou une valeur spéciale. Sinon, mettez à jour l’avant pour qu’il pointe vers le prochain élément, puis libérez la mémoire de l’élément supprimé.

Vérification si la file est vide

Pour vérifier si la file est vide, voyez si l’avant est ‘NULL’. Si c’est le cas, la file n’a aucun élément, ce qui signifie que vous ne pouvez pas effectuer de défilage. Cette vérification empêche des erreurs liées au défilage d’une file vide.

Utilisations courantes des files

Les files sont largement utilisées dans des applications informatiques où des tâches doivent être effectuées dans un ordre spécifique. Par exemple, les files sont utilisées pour la gestion des tâches, les files d’attente dans les systèmes, et les algorithmes de parcours. Elles sont également

utiles pour des structures comme les queues d'impression, les files d'attente dans les réseaux, ou les systèmes d'exploitation qui gèrent des processus.

Résumé

Les files offrent une structure pratique pour des opérations où le premier élément inséré est le premier à sortir. Elles ont de nombreuses applications dans le monde informatique, de la gestion des tâches aux systèmes de files d'attente. Comprendre leur fonctionnement est essentiel pour travailler efficacement avec des structures FIFO.

Exercices

1. Création :

- (a). Implémentez une file en C à l'aide d'un tableau de taille 5. Ajoutez les éléments suivants : 1, 2, 3.
- (b). Affichez les éléments actuels de la file.

2. Enfilage (Enqueue) :

- (a). Ajoutez deux nouveaux éléments 4 et 5 dans la file.
- (b). Essayez d'ajouter un élément 6 et gérez l'erreur si la file est pleine.

3. Défilage (Dequeue) :

- (a). Supprimez le premier élément inséré et affichez l'état actuel de la file.
- (b). Continuez à défiler jusqu'à ce que la file soit vide et gérez l'erreur si la file est déjà vide.

4. Exploration :

- (a). Affichez tous les éléments de la file après chaque opération.

4.4 Arbres binaires

Les *arbres binaires* sont des structures de données où chaque nœud a au maximum deux enfants. Ils sont souvent utilisés pour des opérations de recherche, de tri, et pour implémenter des algorithmes de parcours comme le pré-ordre, l'en-ordre, et le post-ordre.

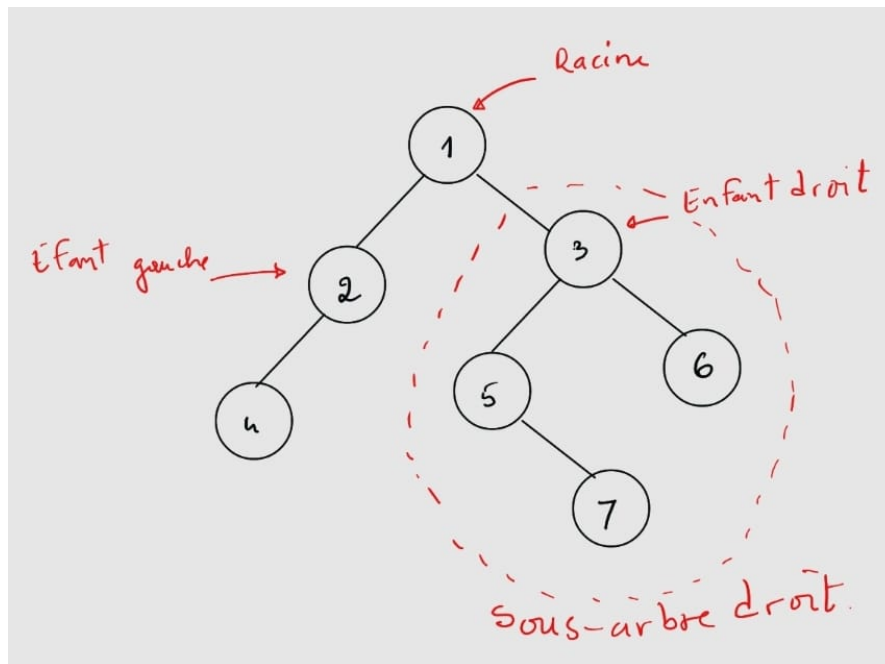


FIGURE 4.4 – Illustration d'un arbre binaire

Construction d'un arbre binaire

Pour construire un arbre binaire, définissez une structure de nœud qui contient des données (comme un entier) et des pointeurs vers les enfants gauche et droit. L'arbre binaire commence avec un seul nœud racine, et les autres nœuds sont ajoutés en fonction de leur relation avec le racine.

Insertion dans un arbre binaire

Pour insérer un élément dans un arbre binaire, commencez par la racine. Si le nouvel élément est plus petit que le racine, allez à gauche. S'il est plus grand, allez à droite. Continuez à descendre jusqu'à ce que vous trouviez une position vide. Ajoutez alors un nouveau nœud à cet endroit.

Utilisations courantes des arbres binaires

Les arbres binaires sont largement utilisés dans des algorithmes informatiques. Ils permettent des opérations de recherche rapides car les éléments peuvent être trouvés sans parcourir tout l'arbre. Les arbres binaires peuvent être utilisés pour des opérations comme la recherche binaire, le tri d'arbres, et le parcours de différentes manières (pré-ordre, en-ordre, post-ordre).

Algorithmes de parcours des arbres binaires

Les algorithmes de parcours des arbres binaires permettent d'explorer ou de traverser les arbres de différentes manières. Les méthodes courantes de parcours sont le parcours en-ordre, le parcours pré-ordre, et le parcours post-ordre.

a) Parcours en-ordre

Le parcours en-ordre consiste à visiter les nœuds d'un arbre binaire dans un ordre spécifique : d'abord le sous-arbre gauche, puis le nœud actuel, et enfin le sous-arbre droit. Cette méthode est souvent utilisée pour obtenir les éléments dans un ordre trié.

- Fonctionnement : Commencez par le sous-arbre gauche, puis visitez le nœud actuel, et terminez par le sous-arbre droit. Cela donne les éléments de l'arbre dans un ordre croissant.

b) Parcours pré-ordre

Le parcours pré-ordre visite d'abord le nœud actuel, puis les sous-arbres gauche et droit. Il est souvent utilisé pour copier des arbres binaires ou pour créer des représentations pré-ordre.

- Fonctionnement : Visitez le nœud actuel, puis parcourez le sous-arbre gauche, puis le sous-arbre droit. Cela peut être utilisé pour recréer l'arbre ou explorer tous les nœuds.

c) Parcours post-ordre

Le parcours post-ordre commence par les sous-arbres gauche et droit, puis visite le nœud actuel. Cette méthode est souvent utilisée pour supprimer des arbres ou pour des opérations nécessitant une évaluation de bas en haut.

- Fonctionnement : Commencez par le sous-arbre gauche, puis le sous-arbre droit, et enfin visitez le nœud actuel. Ce parcours est utile pour des opérations de suppression ou de nettoyage d'arbre.

Les algorithmes de parcours des arbres binaires ont des applications variées. Le parcours en-ordre est idéal pour obtenir des éléments dans un ordre trié, le pré-ordre est utilisé pour copier ou recréer des arbres, et le post-ordre est souvent utilisé pour des opérations de suppression ou de nettoyage. Comprendre ces différentes méthodes de parcours est essentiel pour travailler avec des arbres binaires et résoudre des problèmes complexes d'arborescence.

Résumé

Les arbres binaires sont des structures de données polyvalentes, idéales pour des opérations de recherche et de tri. Ils permettent des algorithmes efficaces pour parcourir des données de manière structurée. Comprendre comment insérer des nœuds et utiliser des arbres binaires est crucial pour travailler avec des structures de données avancées.

Exercices**1. Création (Complétez le code) :**

- (a). Le code ci-dessous initialise une structure pour un nœud d'arbre binaire. Complétez-le en ajoutant la fonction `createNode` pour allouer de la mémoire et initialiser un nœud.

```
struct Node {
```

```

int data;
struct Node* left;
struct Node* right;
};

--- Ajoutez la fonction createNode ici ---

int main() {
    struct Node* root = createNode(50);
    printf("Racine créée avec succès : %d", root->data);
    return 0;
}

```

2. Insertion (Ajoutez les lignes manquantes) :

- (a). Complétez la fonction d'insertion dans un arbre binaire. Ajoutez les conditions pour insérer un nœud à gauche ou à droite.

```

struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        // Ligne manquante pour insérer à gauche
    } else {
        // Ligne manquante pour insérer à droite
    }
    return root;
}

```

- (b). Testez votre fonction en insérant les éléments suivants : 30, 70, 20.

3. Parcours (Complétez le code) :

- (a). Complétez la fonction de parcours en pré-ordre en remplissant la logique pour visiter les enfants gauche et droit :

```

void preOrder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        --- Ajoutez l'appel pour visiter l'enfant gauche ---
        --- Ajoutez l'appel pour visiter l'enfant droit ---
    }
}

```

(b). Affichez les éléments de l'arbre en utilisant votre fonction.

4. Suppression (Avec guide) :

(a). Ajoutez la logique pour supprimer un nœud ayant deux enfants :

```
struct Node* deleteNode(struct Node* root, int data) {
    if (root == NULL) return root;
    if (data < root->data) {
        root->left = deleteNode(root->left, data);
    } else if (data > root->data) {
        root->right = deleteNode(root->right, data);
    } else {
        if (root->left == NULL && root->right == NULL) {
            free(root);
            return NULL;
        }
        // Ajoutez ici la logique pour trouver le successeur en-ordre
    }
    return root;
}
```

(b). Supprimez l'élément 70 de l'arbre existant et affichez les éléments en parcours en-ordre.

4.5 Graphes

Les *graphes* sont des structures de données composées de nœuds (ou sommets) et d'arêtes (les connexions entre les nœuds). Les graphes peuvent être orientés ou non orientés. Dans un graphe orienté, les arêtes ont une direction (comme une flèche). Dans un graphe non orienté, les arêtes n'ont pas de direction particulière. Les graphes sont utilisés pour résoudre des problèmes complexes comme la recherche des plus courts chemins, la détection de cycles, ou le partitionnement.

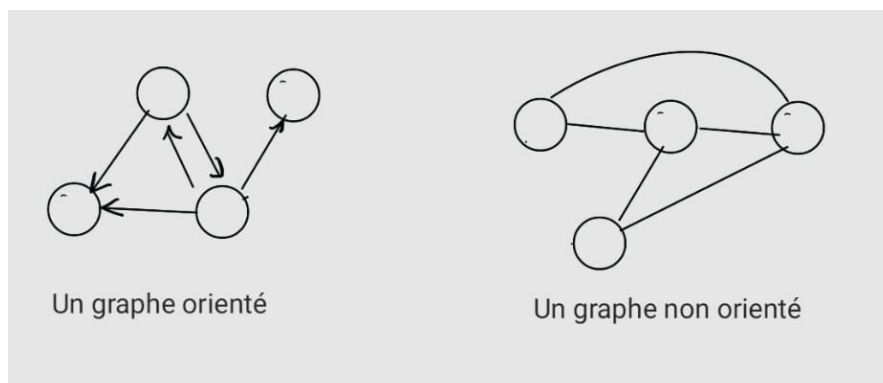


FIGURE 4.5 – Exemples des deux types de graphes

Construction d'un graphe

Pour construire un graphe, vous devez définir une structure qui contient une liste de sommets et des arêtes qui les relient. Les graphes peuvent être implémentés de différentes manières, comme avec des listes d'adjacence ou des matrices d'adjacence.

Dans le cas des listes d'adjacence, chaque sommet a une liste de connexions vers d'autres sommets. Les graphes peuvent être orientés (avec des flèches pour indiquer la direction des arêtes) ou non orientés (où les connexions sont bidirectionnelles). Pour construire un graphe, initialisez le nombre de sommets et allouez de la mémoire pour chaque liste d'adjacence.

Ajout d'arêtes (liens) dans un graphe

Pour ajouter des arêtes dans un graphe, vous devez d'abord déterminer les sommets que vous voulez relier. Si le graphe est orienté, assurez-vous de respecter la direction des arêtes. Pour un graphe non orienté, ajoutez des connexions bidirectionnelles.

Dans le cas des listes d'adjacence, créez un nouveau nœud pour chaque connexion, puis ajoutez-le à la liste du sommet correspondant. Assurez-vous que les arêtes sont correctement ajoutées pour éviter les erreurs de connexion.

Utilisations courantes des graphes

Les graphes sont utilisés dans de nombreux domaines informatiques. Ils permettent de modéliser des relations complexes entre des entités. Par exemple, les graphes peuvent représenter des réseaux de transport, des relations sociales, ou des connexions informatiques. Les graphes sont également utiles pour des algorithmes de parcours comme la recherche en profondeur (DFS) ou la recherche en largeur (BFS).

Les graphes peuvent résoudre des problèmes comme la recherche du plus court chemin, la détection de cycles, et la recherche de composants connexes. Comprendre leur fonctionnement est essentiel pour travailler avec des structures de données avancées.

Algorithmes de parcours de graphes

Les algorithmes de parcours de graphes permettent d'explorer ou de traverser les graphes de différentes manières. Les méthodes courantes de parcours incluent la recherche en profondeur (DFS) et la recherche en largeur (BFS).

a) Recherche en profondeur (DFS – [Depth First Search])

DFS explore le graphe en suivant chaque branche aussi loin que possible avant de revenir en arrière. Elle utilise une pile pour mémoriser les nœuds visités. DFS est souvent utilisée pour détecter des cycles, trouver des composants connexes, ou résoudre des problèmes de recherche de chemins.

- Fonctionnement : DFS commence à partir d'un nœud et explore tous ses voisins. Ensuite, il passe au nœud suivant non visité, poursuivant ainsi la profondeur jusqu'à ce qu'il n'y ait plus de nœuds à visiter. Puis, il revient en arrière pour explorer d'autres chemins.

b) **Recherche en largeur (BFS – [Breadth First Search])**

BFS explore le graphe couche par couche, en visitant tous les voisins d'un nœud avant de passer au suivant. Elle utilise une file pour mémoriser les nœuds à visiter. BFS est couramment utilisée pour trouver des plus courts chemins dans des graphes non pondérés ou pour des problèmes de recherche de niveaux.

- Fonctionnement : BFS commence par le premier nœud et visite tous ses voisins. Ensuite, il passe aux voisins de ces voisins, touchant tous les nœuds au même niveau avant de passer au niveau suivant.

Les algorithmes de parcours de graphes ont des applications variées. DFS est utilisé pour des analyses approfondies, comme la recherche de cycles ou de chemins profonds, tandis que BFS est idéal pour des analyses en surface, comme trouver des connexions courtes ou explorer des niveaux. Ces algorithmes sont essentiels pour résoudre des problèmes complexes de graphes.

Résumé

Les graphes sont des structures de données puissantes qui permettent de modéliser des connexions entre des sommets. Ils sont utilisés dans de nombreux domaines, comme les réseaux informatiques, les systèmes de transport, ou l'analyse sociale. Les graphes peuvent être orientés (où les arêtes ont une direction) ou non orientés (où les connexions sont bidirectionnelles).

Les algorithmes de parcours comme la recherche en profondeur (DFS) et la recherche en largeur (BFS) sont des outils essentiels pour travailler avec des graphes. DFS explore les graphes en profondeur, tandis que BFS explore les graphes en largeur, offrant des applications diverses pour des problèmes complexes.

Les graphes peuvent résoudre des problèmes tels que la recherche des plus courts chemins, la détection de cycles, ou la recherche de composants connexes. Ils permettent également de modéliser des systèmes complexes avec des connexions multiples. La compréhension des graphes, de leurs applications et des algorithmes de parcours est essentielle pour résoudre des problèmes avancés en informatique.

Exercices

1. Création (Complétez le code) :

- (a). Complétez la fonction pour créer un graphe. Ajoutez la logique pour initialiser chaque liste d'adjacence.

```
struct Graph* createGraph(int vertices) {
    struct Graph* graph = malloc(sizeof(struct Graph));
```



```

graph->numVertices = vertices;
graph->adjLists = malloc(vertices * sizeof(struct Node*));
for (int i = 0; i < vertices; i++) {
    --- Ajoutez la ligne pour initialiser chaque liste d'adjacence ---
}
return graph;
}

```

2. Ajout d'arêtes (Complétez la fonction) :

- (a). Complétez la fonction pour ajouter une arête entre deux sommets dans un graphe non orienté :

```

void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = malloc(sizeof(struct Node));
    newNode->vertex = dest;
    newNode->next = --- Ajoutez ici pour lier au sommet source ---
    graph->adjLists[src] = newNode;

    // Ajouter la même logique pour dest -> src
}

```

- (b). Ajoutez des arêtes entre les sommets A, B, C, et D.

3. Recherche en largeur (Complétez le code) :

- (a). Ajoutez la logique pour visiter tous les voisins d'un sommet dans la fonction BFS :

```

void BFS(struct Graph* graph, int startVertex) {
    int visited[graph->numVertices];
    for (int i = 0; i < graph->numVertices; i++) visited[i] = 0;

    struct Queue* queue = createQueue();
    visited[startVertex] = 1;
    enqueue(queue, startVertex);

    while (!isEmpty(queue)) {
        int currentVertex = dequeue(queue);
        printf("%d ", currentVertex);

        struct Node* temp = graph->adjLists[currentVertex];
        while (temp) {
            int adjVertex = temp->vertex;
            if (!visited[adjVertex]) {

```

```

visited[adjVertex] = 1;
--- Ajoutez ici la logique pour mettre en file le sommet voisin ---
}
temp = temp->next;
}
}
}

```

(b). Testez la fonction BFS à partir du sommet A.

4. Exploration (Ajoutez des vérifications) :

(a). Ajoutez une fonction pour vérifier si une arête existe entre deux sommets :

```

int edgeExists(struct Graph* graph, int src, int dest) {
struct Node* temp = graph->adjLists[src];
while (temp != NULL) {
if (temp->vertex == dest) {
--- Retournez une valeur indiquant que l'arête existe ---
}
temp = temp->next;
}
--- Retournez une valeur indiquant que l'arête n'existe pas ---
}

```

(b). Testez la fonction pour les sommets A et D.

Chapitre 5

Applications d'algorithmes en IA

5.1 Introduction

L'intelligence artificielle repose sur des algorithmes sophistiqués pour prendre des décisions, analyser des données et résoudre des problèmes complexes. Ce chapitre explore les algorithmes populaires de l'IA en mettant l'accent sur leur fonctionnement étape par étape et sur des exemples concrets qui illustrent leur application.

5.2 Algorithmes d'Apprentissage Supervisé

Régression Linéaire

La **régression linéaire** est utilisée pour prédire une valeur numérique continue (comme un prix ou une température) en fonction d'une ou plusieurs variables.

- **Idée de base** : Imaginez un nuage de points sur un graphique où l'axe x représente une variable (par exemple, la taille d'une maison) et l'axe y représente le résultat (comme le prix de la maison). La régression linéaire cherche à tracer une ligne qui passe le plus près possible de ces points.
- **Étapes de l'algorithme** :
 1. **Préparer les données** : Regroupez les données en paires (x, y) .
 2. **Déterminer l'équation** : Une ligne est définie par $y = ax + b$.
 3. **Ajuster la ligne** : Trouvez a et b pour minimiser l'erreur.
 4. **Prédire** : Utilisez $y = ax + b$ pour de nouvelles valeurs de x .

TABLE 5.1 – Données pour la régression linéaire

Surface (m ²)	Prix (MGA)
50	70,000,000
75	95,000,000
100	130,000,000

Exercice : Implémenter la Régression Linéaire

1. Écrivez un programme en C qui :
 - Accepte les paires (x, y) en entrée.

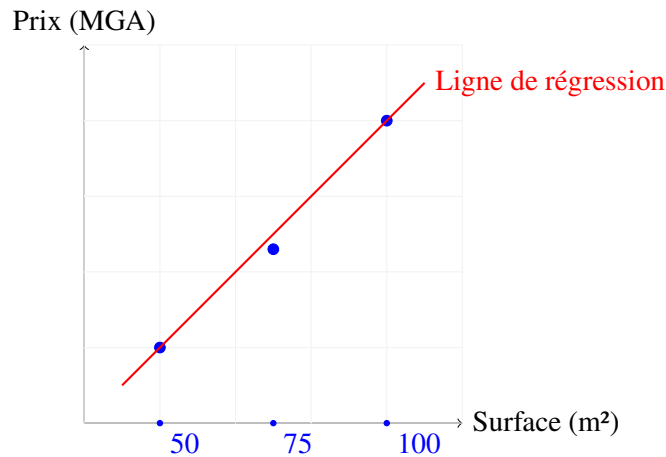


FIGURE 5.1 – Régression linéaire sur des données simulées.

- Calcule les coefficients a et b .
 - Prédit y pour une nouvelle valeur x .
2. Utilisez les données du tableau ci-dessus et prédisez y pour $x = 80$.

Régression Logistique

La **régression logistique** est utilisée pour prédire une probabilité binaire, comme "oui/non" ou "succès/échec".

- **Idée de base** : Contrairement à la régression linéaire, la régression logistique utilise une *fonction sigmoïde* pour transformer un score z en une probabilité :

$$P = \frac{1}{1 + e^{-z}}$$

- **Étapes de l'algorithme** :
 1. Calculez z avec $z = a \cdot x + b$.
 2. Appliquez la fonction sigmoïde pour obtenir P .
 3. Comparez P avec un seuil (e.g., 0,5) pour décider "oui" ou "non".

TABLE 5.2 – Exemple pour la régression logistique

Temps (minutes)	Pages visitées	z
12	7	$z = 2.3$

Exercice : Implémenter la Régression Logistique

1. Écrivez un programme en C qui :
 - Calcule z pour une valeur donnée.
 - Transforme z en P à l'aide de la fonction sigmoïde.
 - Compare P avec un seuil pour afficher "Oui" ou "Non".
2. Utilisez les données du tableau ci-dessus et prédisez si le client achètera.

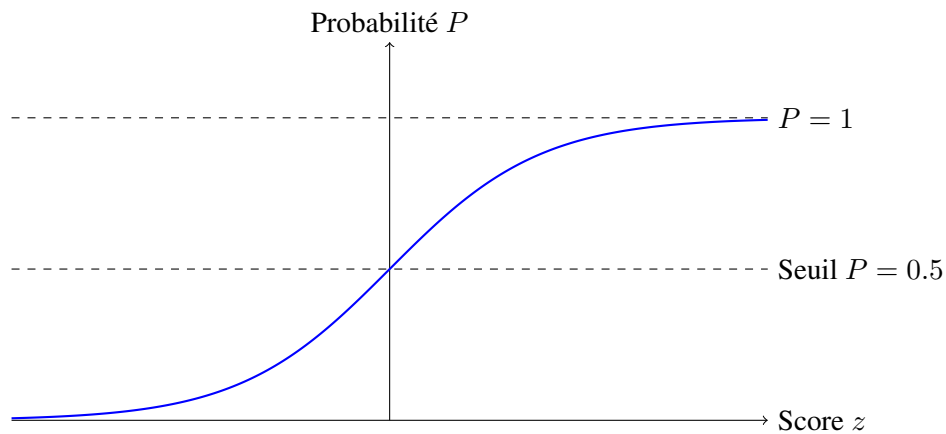


FIGURE 5.2 – Courbe sigmoïde pour la régression logistique.

Clustering avec K-means

L'algorithme **K-means** regroupe des données en plusieurs groupes appelés *clusters*.

- **Idée de base** : Divisez les données en K groupes, chaque groupe ayant un centre (*centroïde*).
- **Étapes de l'algorithme** :
 1. Choisissez K centroïdes aléatoires.
 2. Assignez chaque point au centroïde le plus proche.
 3. Recalculez les centroïdes en prenant la moyenne des points.
 4. Répétez jusqu'à ce que les centroïdes ne changent plus.

TABLE 5.3 – Données pour K-means

Point	Coordonnées (x, y)
A	(1, 2)
B	(3, 4)
C	(5, 6)
D	(8, 9)

Exercice : Implémenter K-means

1. Écrivez un programme en C qui :
 - Accepte une liste de points (x, y) .
 - Initialise 2 centroïdes aléatoires.
 - Assigne chaque point au centroïde le plus proche.
 - Recalcule les centroïdes et affiche les groupes finaux.
2. Utilisez les données du tableau ci-dessus.

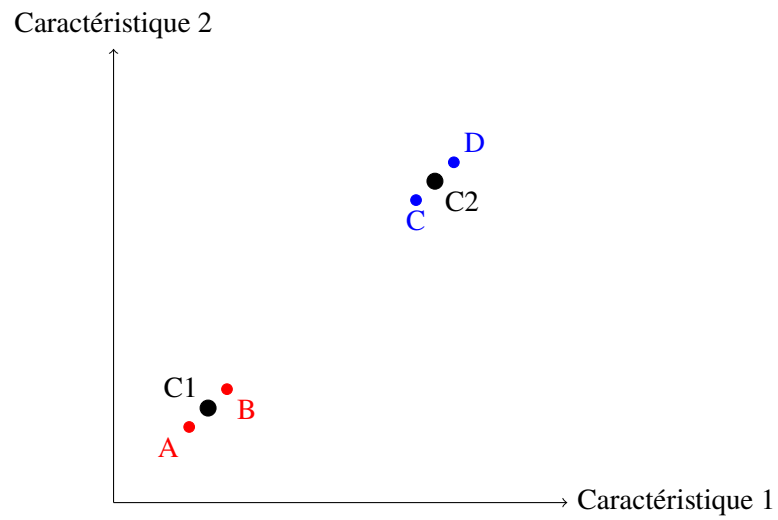


FIGURE 5.3 – Clustering K-means avec deux clusters.

Chapitre 6

Corrigées des exercices

6.1 Structures de données fondamentales

Variables et types de données

1. Quelle est la différence entre les types de données suivants :
 - (a). **int**
 - (b). **float**
 - (c). **char**
2. Écrivez un programme C qui déclare des variables pour chaque type de données mentionné et affiche leurs valeurs.

Réponse :

```
1      #include <stdio.h>
2
3      int main() {
4          int entier = 42;
5          float flottant = 3.14;
6          char caractere = 'A';
7
8          printf("Entier: %d\n", entier);
9          printf("Flottant: %.2f\n", flottant);
10         printf("Caractere: %c\n", caractere);
11
12         return 0;
13     }
```

3. Écrivez un programme C qui utilise des variables de différents types de données pour calculer la somme d'un entier et d'un flottant, puis affichez le résultat.

Réponse :

```
1      #include <stdio.h>
2
3      int main() {
4          int entier = 10;
```

```

5      float flottant = 2.5;
6
7      float somme = entier + flottant;
8
9      printf("La somme est: %.2f\n", somme);
10
11     return 0;
12 }
```

4. Quelles sont les limites de valeurs pour les types de données suivants :

- (a). **int**
- (b). **float**
- (c). **double**

Indication : Considérer le nombre de bits et octets. Par exemple, un caractère ASCII est représenté par un octet, c'est-à-dire 8 bits. Le caractère "A" est représenté par 01000001. Avec huit chiffres de 0 et 1, on peut représenter $2^8 = 256$ valeurs. Pour les nombres signés (int, float, double, ...), l'opposé est représenté différemment.

Réponse :

- (a). 'int' : généralement de '-2.147.483.648' à '2.147.483.647'.
- (b). 'float' : de '1,2E-38' à '3,4E+38'.
- (c). 'double' : de '2,3E-308' à '1,7E+308'.

Pointeurs

1. Quelle est la différence entre une variable et un pointeur ?
2. Écrivez un programme C qui utilise des pointeurs pour modifier la valeur d'une variable.

Réponse :

```

1      #include <stdio.h>
2
3      int main() {
4          int x = 10;
5          int *p = &x;
6
7          printf("Valeur_de_x_avant_modification: %d\n", x);
8
9          *p = 20;
10
11         printf("Valeur_de_x_apres_modification: %d\n", x);
12 }
```



```

13     return 0;
14 }

```

3. Écrivez un programme C qui utilise des pointeurs pour échanger les valeurs de deux variables.

Réponse :

```

1     #include <stdio.h>
2
3     void echanger(int *a, int *b) {
4         int temp = *a;
5         *a = *b;
6         *b = temp;
7     }
8
9     int main() {
10        int x = 10;
11        int y = 20;
12
13        printf("Avant l'echange: x = %d, y = %d\n", x, y);
14
15        echanger(&x, &y);
16
17        printf("Après l'echange: x = %d, y = %d\n", x, y);
18
19        return 0;
20    }

```

4. Expliquez ce qu'est un pointeur nul (null pointer) et ce qui se passe si vous tentez d'utiliser un pointeur nul.

Réponse : Un pointeur nul ('NULL') ne pointe pas vers une adresse mémoire valide. Tenter d'utiliser ou de déréférencer un pointeur nul peut provoquer des erreurs de segmentation ou des plantages du programme.

Tableaux

1. Déclarez un tableau de 5 entiers et initialisez-le avec des valeurs. Écrivez un programme C pour afficher les valeurs du tableau.
2. Modifiez le programme pour inverser les éléments du tableau et afficher le tableau inversé.

Réponse :

```

1  #include <stdio.h>
2
3  int main() {
4      int tableau[5] = {1, 2, 3, 4, 5};
5
6      printf("Tableau initial : ");
7      for (int i = 0; i < 5; i++) {
8          printf("%d ", tableau[i]);
9      }
10     printf("\n");
11
12     // Inverser le tableau
13     int temp;
14     for (int i = 0; i < 2; i++) {
15         temp = tableau[i];
16         tableau[i] = tableau[4 - i];
17         tableau[4 - i] = temp;
18     }
19
20     printf("Tableau inversE: ");
21     for (int i = 0; i < 5; i++) {
22         printf("%d ", tableau[i]);
23     }
24     printf("\n");
25
26     return 0;
27 }

```

3. Écrivez un programme C qui calcule la somme de tous les éléments d'un tableau d'entiers de longueur 'n'.

Réponse :

```

1  #include <stdio.h>
2
3  int somme(int tableau[], int taille) {
4      int total = 0;
5
6      for (int i = 0; i < taille; i++) {
7          total += tableau[i];

```

```

8         }
9
10        return total;
11    }
12
13    int main() {
14        int tableau[5] = {1, 2, 3, 4, 5};
15
16        int total = somme(tableau, 5);
17
18        printf("La somme totale est: %d\n", total);
19
20        return 0;
21    }

```

4. Que se passe-t-il si vous essayez d'accéder à un index hors limites (out of bounds) dans un tableau C ? Écrivez un programme pour le démontrer.

Réponse : Accéder à un index hors limites (out of bounds) peut provoquer des erreurs de segmentation ou des plantages du programme.

```

1    #include <stdio.h>
2
3    int main() {
4        int tableau[5] = {1, 2, 3, 4, 5};
5
6        // Essayer d'accéder a un index hors limites
7        printf("AccEs a un index hors limites: %d\n", tableau
8              [10]);
9        // Ceci peut provoquer des erreurs de segmentation
10
11        return 0;
12    }

```

Chaînes de caractères

1. Quelle est la particularité des chaînes de caractères en C par rapport aux autres types de données ?
2. Écrivez un programme C qui déclare une chaîne de caractères et utilise la bibliothèque 'string.h' pour copier du texte dans la chaîne.

Réponse :

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main() {
5      char chaine[50];
6
7      strcpy(chaine, "Bonjour, monde!");
8
9      printf("Chaîne: %s\n", chaine);
10
11     return 0;
12 }

```

3. Écrivez un programme C qui concatène deux chaînes de caractères en utilisant 'strcat'.

Réponse :

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main() {
5      char chaine1[50] = "Bonjour";
6      char chaine2[50] = ", monde!";
7
8      strcat(chaine1, chaine2); // Concaténer les deux chaînes
9
10     printf("Résultat de la concaténation: %s\n", chaine1);
11
12     return 0;
13 }

```

4. Comment compare-t-on deux chaînes de caractères en C ? Écrivez un programme qui compare deux chaînes et indique si elles sont identiques ou non.

Réponse : On peut comparer deux chaînes de caractères en utilisant 'strcmp' de la bibliothèque 'string.h'.

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main() {
5      char chaine1[50] = "Bonjour";

```

```

6      char chaine2[50] = "Bonjour";
7
8      int comparaison = strcmp(chaine1, chaine2);
9
10     if (comparaison == 0) {
11         printf("Les deux chaînes sont identiques.\n");
12     } else if (comparaison < 0) {
13         printf("La première chaîne est inférieure.\n");
14     } else {
15         printf("La première chaîne est supérieure.\n");
16     }
17
18     return 0;
19 }

```

Structures

1. Déclarez une structure en C pour représenter une personne avec un nom, un âge, et une taille.
2. Écrivez un programme C qui utilise cette structure pour créer une personne, assigner des valeurs, et afficher les informations de la personne.

Réponse :

```

1      #include <stdio.h>
2      #include <string.h>
3
4      typedef struct {
5          char nom[50];
6          int age;
7          float taille ;
8      } Personne;
9
10     int main() {
11         Personne p;
12         strcpy(p.nom, "John Doe");
13         p.age = 30;
14         p.taille = 1.75;
15
16         printf("Nom: %s\n", p.nom);

```

```

17     printf("Âge: %d\n", p.age);
18     printf("Taille: %.2f m\n", p.taille);
19
20     return 0;
21 }

```

3. Écrivez un programme C qui utilise des structures pour représenter un point 3D avec des coordonnées x, y, z. Ensuite, écrivez une fonction qui calcule la distance entre deux points 3D.

Réponse :

```

1     #include <stdio.h>
2     #include <math.h>
3
4     typedef struct {
5         float x;
6         float y;
7         float z;
8     } Point3D;
9
10    float distance(Point3D p1, Point3D p2) {
11        return sqrt(pow(p2.x - p1.x, 2) + pow(p2.y - p1.y, 2) +
12                    pow(p2.z - p1.z, 2));
13    }
14
15    int main() {
16        Point3D p1 = {0, 0, 0};
17        Point3D p2 = {3, 4, 5};
18
19        float dist = distance(p1, p2);
20
21        printf("Distance entre les deux points: %.2f\n", dist);
22
23        return 0;
24    }

```

4. Créez une structure pour représenter une voiture avec un nom, une année, et un prix. Écrivez un programme C qui utilise cette structure pour créer une voiture et afficher ses propriétés.

Réponse :

```

1  #include <stdio.h>
2  #include <string.h>
3
4  typedef struct {
5      char nom[50];
6      int annee;
7      float prix;
8  } Voiture;
9
10 int main() {
11     Voiture v;
12     strcpy(v.nom, "Toyota");
13     v.annee = 2020;
14     v.prix = 25000.50;
15
16     printf("Voiture: %s\n", v.nom);
17     printf("Année: %d\n", v.annee);
18     printf("Prix: %.2f\n", v.prix);
19
20     return 0;
21 }

```

6.2 Algorithmes de tri et de recherche

Algorithmes de tri

1. Pour chaque algorithme de tri, implémentez-le en C et testez-le avec les cas suivants :
 - (a). Un tableau vide.
 - (b). Un tableau contenant un seul élément.
 - (c). Un tableau déjà trié.
 - (d). Le tableau [9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7].

Tri à bulle (Bubble Sort)

Réponse :

```

1  #include <stdio.h>
2
3  void bubbleSort(int arr [], int n) {

```

```

4     for (int i = 0; i < n-1; i++) {
5         for (int j = 0; j < n-i-1; j++) {
6             if (arr[j] > arr[j+1]) {
7                 // Échanger les éléments
8                 int temp = arr[j];
9                 arr[j] = arr[j+1];
10                arr[j+1] = temp;
11            }
12        }
13    }
14 }

15
16 void printArray(int arr [], int size) {
17     for (int i = 0; i < size; i++) {
18         printf("%d ", arr[i]);
19     }
20     printf("\n");
21 }
22
23 int main() {
24     int arr [] = {9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7};
25     int n = sizeof(arr)/sizeof(arr[0]);
26
27     bubbleSort(arr, n);
28     printf("Array trié : ");
29     printArray(arr, n);
30
31     return 0;
32 }

```

Tri par insertion (Insertion Sort)

Réponse :

```

1     #include <stdio.h>
2
3     void insertionSort (int arr [], int n) {
4         for (int i = 1; i < n; i++) {
5             int key = arr[i];

```



```

6      int j = i - 1;
7      while (j >= 0 && arr[j] > key) {
8          arr[j + 1] = arr[j];
9          j = j - 1;
10     }
11     arr[j + 1] = key;
12 }
13 }
14
15 void printArray(int arr [], int size) {
16     for (int i = 0; i < size; i++) {
17         printf("%d ", arr[i]);
18     }
19     printf("\n");
20 }
21
22 int main() {
23     int arr [] = {9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7};
24     int n = sizeof(arr)/sizeof(arr[0]);
25
26     insertionSort(arr, n);
27     printf("Array trié : ");
28     printArray(arr, n);
29
30     return 0;
31 }

```

Tri rapide (Quick Sort)

Réponse :

```

1      #include <stdio.h>
2
3      int partition(int arr [], int low, int high) {
4          int pivot = arr[high];
5          int i = (low - 1);
6
7          for (int j = low; j < high; j++) {
8              if (arr[j] < pivot) {

```

```
9         i++;
10         int temp = arr[i];
11         arr[i] = arr[j];
12         arr[j] = temp;
13     }
14 }
15 int temp = arr[i + 1];
16 arr[i + 1] = arr[high];
17 arr[high] = temp;
18 return i + 1;
19 }
20
21 void quickSort(int arr [], int low, int high) {
22     if (low < high) {
23         int pi = partition(arr, low, high);
24         quickSort(arr, low, pi - 1);
25         quickSort(arr, pi + 1, high);
26     }
27 }
28
29 void printArray(int arr [], int size) {
30     for (int i = 0; i < size; i++) {
31         printf("%d ", arr[i]);
32     }
33     printf("\n");
34 }
35
36 int main() {
37     int arr [] = {9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7};
38     int n = sizeof(arr)/sizeof(arr[0]);
39
40     quickSort(arr, 0, n - 1);
41     printf("Array trié : ");
42     printArray(arr, n);
43
44     return 0;
45 }
```

Algorithmes de recherche

1. Pour chaque algorithme de recherche, implémentez-le en C et testez-le en cherchant les éléments 8, 1 et 0 dans le tableau [9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7].

Recherche linéaire (Linear Search)

Réponse :

```

1  #include <stdio.h>
2
3  int linearSearch(int arr [], int n, int key) {
4      for (int i = 0; i < n; i++) {
5          if (arr[i] == key) {
6              return i; // Retourne l'indice où l'élément est trouvé
7          }
8      }
9      return -1; // L'élément n'est pas trouvé
10 }
11
12 int main() {
13     int arr [] = {9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7};
14     int n = sizeof(arr)/sizeof(arr[0]);
15
16     printf("Recherche de l'élément 8: %d\n", linearSearch(arr, n,
17                                     8));
18     printf("Recherche de l'élément 1: %d\n", linearSearch(arr, n,
19                                     1));
20     printf("Recherche de l'élément 0: %d\n", linearSearch(arr, n,
21                                     0));
22
23     return 0;
24 }
```

Recherche binaire (Binary Search)

La recherche binaire n'est applicable que si le tableau est trié. Voici l'implémentation pour une recherche binaire dans un tableau trié.

Réponse :

```

1  #include <stdio.h>
```

```

2
3  int binarySearch(int arr [], int low, int high, int key) {
4      while (low <= high) {
5          int mid = low + (high - low) / 2;
6          if (arr[mid] == key) {
7              return mid;
8          }
9          if (arr[mid] < key) {
10             low = mid + 1;
11         } else {
12             high = mid - 1;
13         }
14     }
15     return -1; // L'élément n'est pas trouvé
16 }
17
18 int main() {
19     int arr [] = {0, 1, 4, 4, 4, 5, 6, 7, 9, 9};
20     int n = sizeof(arr)/sizeof(arr[0]) ;
21
22     printf("Recherche de l'élément 0: %d\n", binarySearch(arr, 0,
23         n - 1, 0));
24     printf("Recherche de l'élément 2: %d\n", binarySearch(arr, 0,
25         n - 1, 2));
26     printf("Recherche de l'élément 7: %d\n", binarySearch(arr, 0,
27         n - 1, 7));
28     printf("Recherche de l'élément 9: %d\n", binarySearch(arr, 0,
29         n - 1, 9));
30
31     return 0;
32 }

```

2. Lequel de ces algorithmes n'est pas applicable au tableau précédent ? Pourquoi ?

La recherche binaire n'est pas applicable au tableau [9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7] car il n'est pas trié. La recherche binaire fonctionne uniquement sur des tableaux triés, car elle divise le tableau en deux parties et suppose que les éléments à gauche sont inférieurs à ceux à droite. Dans ce cas, un tri préalable est nécessaire.

6.3 Complexité algorithmique

1. Quelle est la complexité temporelle des opérations suivantes ?

- (a). Parcourir un tableau d'entiers de longueur n .

Réponse : La complexité temporelle de l'opération "parcourir un tableau d'entiers" est $O(n)$. En effet, dans cette opération, chaque élément du tableau est visité une fois, ce qui signifie qu'il y a n opérations à effectuer, où n est la taille du tableau. La notation $O(n)$ exprime que la durée de l'opération est linéaire par rapport à la taille du tableau.

- (b). Multiplier deux matrices de taille $n \times n$.

Réponse : La multiplication de deux matrices carrées de taille $n \times n$ nécessite d'effectuer n^2 multiplications. En effet, chaque élément de la matrice résultat est le produit de n éléments (lignes de la première matrice et colonnes de la deuxième matrice). Donc, pour chaque élément de la matrice résultat (qui est n^2 éléments au total), nous devons effectuer n multiplications, ce qui donne une complexité de $O(n^3)$. Cette complexité est dans le cas général pour une multiplication de matrices à l'aide de l'algorithme classique. Des algorithmes plus avancés, comme l'algorithme de Strassen, peuvent réduire cette complexité à $O(n^{2.81})$, mais l'algorithme classique reste de complexité $O(n^3)$.

- (c). Effectuer une recherche binaire sur un tableau trié.

Réponse : La recherche binaire dans un tableau trié a une complexité temporelle de $O(\log n)$. L'idée principale de la recherche binaire est de diviser à chaque étape le tableau en deux sous-tableaux, réduisant ainsi de moitié l'espace de recherche à chaque itération. Cette opération est répétée jusqu'à ce que l'élément recherché soit trouvé ou que l'espace de recherche soit vide. La complexité $\log n$ provient du fait que l'on réduit par moitié la taille de la partie restante du tableau à chaque itération. La recherche binaire est donc très efficace pour les tableaux triés, car elle évite de parcourir l'intégralité du tableau. Contrairement à une recherche linéaire qui aurait une complexité $O(n)$, la recherche binaire est logarithmique.

2. Quelle est la complexité temporelle du code suivant ?

```

1   for (int i = 0; i < n; i++) {
2       for (int j = 0; j < n; j++) {
3           printf("%d ", i * j);
4       }
5   }
```

Réponse : La complexité temporelle de ce code est $O(n^2)$. Voici pourquoi :

- La première boucle `for (int i = 0; i < n; i++)` s'exécute n fois, car i varie de 0 à $n - 1$.

- La deuxième boucle imbriquée `for (int j = 0; j < n; j++)` s'exécute également n fois pour chaque itération de la première boucle.

Par conséquent, pour chaque itération de la première boucle (qui se répète n fois), la deuxième boucle s'exécute également n fois, ce qui donne un total de $n \times n = n^2$ itérations au total. C'est ce que l'on appelle une complexité quadratique.

- Le `printf` à l'intérieur des boucles effectue une opération constante à chaque itération (une simple impression), ce qui ne change pas la complexité globale de l'algorithme. L'opération `printf` elle-même, bien que coûteuse en termes de temps d'exécution réel, reste considérée comme une opération de complexité constante dans ce contexte.

3. Comment pouvez-vous améliorer cet algorithme pour réduire sa complexité ?

Réponse : Pour réduire la complexité de cet algorithme, il faudrait d'abord identifier s'il y a une optimisation possible dans la logique de l'algorithme. Ici, le problème semble être celui d'afficher tous les produits de $i \times j$ pour i, j allant de 0 à $n - 1$. Une possible amélioration serait de revoir la structure du code pour éviter des calculs redondants. En effet, si on cherche à afficher les produits $i \times j$ et que l'on parcourt les deux indices i et j indépendamment, cela entraîne une complexité $O(n^2)$. Cependant, il est possible de réorganiser le calcul en utilisant des propriétés symétriques des produits, comme dans le cas d'un tableau triangulaire ou d'autres structures spécifiques.

En fonction de l'objectif spécifique du programme (comme stocker les résultats ou effectuer un calcul particulier), il pourrait être possible de réduire la quantité de travail en évitant de calculer des produits pour des paires (i, j) identiques (par exemple, en exploitant la symétrie si $i \times j = j \times i$), mais cela dépend du contexte de l'application.

6.4 Structures de données avancées

Listes chaînées

1. Création d'une liste chaînée :

Réponse : Pour créer une liste chaînée, nous devons définir une structure pour les nœuds de la liste, où chaque nœud contiendra une donnée et un pointeur vers le nœud suivant. La tête de la liste est initialisée à `NULL` pour indiquer que la liste est vide.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Node {
5      int data;
6      struct Node* next;
7  };
8
```

```

9      struct Node* head = NULL; // Initialisation de la tête de la
      liste à NULL
10
11     // Fonction pour créer une nouvelle liste
12     void createList(int value) {
13         struct Node* newNode = malloc(sizeof(struct Node));
14         newNode->data = value;
15         newNode->next = NULL;
16         head = newNode;
17     }

```

Explication : La fonction `createList` alloue de la mémoire pour un nouveau nœud, définit sa donnée, et initialise son pointeur suivant à `NULL`. Elle affecte ensuite ce nœud à la tête de la liste.

2. Insertion d'un nœud au début de la liste :

Réponse : Pour insérer un nœud au début de la liste chaînée, nous créons un nouveau nœud et nous mettons à jour son pointeur `next` pour qu'il pointe vers l'ancienne tête de la liste. Ensuite, nous mettons la tête de la liste pour qu'elle pointe vers ce nouveau nœud.

```

1     // Fonction pour insérer un nœud au début de la liste
2     void insertAtBeginning(int value) {
3         struct Node* newNode = malloc(sizeof(struct Node));
4         newNode->data = value;
5         newNode->next = head; // Le nouveau nœud pointe vers l'
      ancienne tête
6         head = newNode; // La tête pointe maintenant vers le
      nouveau nœud
7     }

```

Explication : La fonction `insertAtBeginning` crée un nouveau nœud avec la valeur donnée, fait pointer ce nœud vers l'ancienne tête de la liste, puis met la tête de la liste pour qu'elle pointe vers le nouveau nœud. Cela insère efficacement le nœud au début de la liste.

3. Insertion d'un nœud à une position spécifique :

Réponse : Pour insérer un nœud à une position spécifique dans la liste, nous devons parcourir la liste jusqu'à la position souhaitée, puis insérer le nœud à cet endroit en ajustant les pointeurs. Si la position est zéro, on insère au début de la liste.

```

1     // Fonction pour insérer un nœud à une position spécifique
2     void insertAtPosition(int value, int position) {
3         if (position == 0) {
4             insertAtBeginning(value);

```

```

5      return;
6  }

7
8      struct Node* newNode = malloc(sizeof(struct Node));
9      newNode->data = value;
10     struct Node* current = head;
11
12     for (int i = 0; i < position - 1 && current != NULL; i++)
13     {
14         current = current->next;
15     }
16
17     if (current == NULL) {
18         printf("Position invalide\n");
19         return;
20     }
21
22     newNode->next = current->next;
23     current->next = newNode;

```

Explication : La fonction `insertAtPosition` parcourt la liste jusqu'à la position spécifiée. Si la position est zéro, elle appelle la fonction `insertAtBeginning`. Sinon, elle insère un nouveau nœud à la position donnée en ajustant les pointeurs. Si la position est invalide, un message d'erreur est affiché.

4. Suppression d'un nœud de la liste chaînée :

Réponse : Pour supprimer un nœud de la liste chaînée, nous devons parcourir la liste jusqu'à ce que nous trouvions le nœud à supprimer. Ensuite, nous ajustons le pointeur du nœud précédent pour qu'il pointe vers le nœud suivant du nœud à supprimer.

```

1      // Fonction pour supprimer un nœud d'une liste chaînée
2      void deleteNode(int value) {
3          struct Node* current = head;
4          struct Node* previous = NULL;
5
6          // Si la tête contient la valeur à supprimer
7          if (current != NULL && current->data == value) {
8              head = current->next;
9              free(current);

```



```

10     return;
11 }
12
13 // Parcours de la liste à la recherche du œnud
14 while (current != NULL && current->data != value) {
15     previous = current;
16     current = current->next;
17 }
18
19 // Si la valeur n'est pas trouvée
20 if (current == NULL) {
21     printf("Valeur non trouvée dans la liste\n");
22     return;
23 }
24
25 // Supprimer le œnud
26 previous->next = current->next;
27 free(current);
28 }

```

Explication : La fonction `deleteNode` commence par vérifier si le nœud à supprimer est le premier nœud de la liste (la tête). Si ce n'est pas le cas, elle parcourt la liste pour trouver le nœud correspondant à la valeur à supprimer. Une fois le nœud trouvé, elle ajustera le pointeur du nœud précédent pour qu'il saute le nœud à supprimer et libère la mémoire allouée pour ce nœud.

Piles

1. Construction d'une pile :

Réponse : Pour construire une pile, vous devez définir une structure qui contient les éléments de la pile et un pointeur ou un indice qui indique le sommet de la pile. La structure suivante utilise un tableau pour représenter la pile et un indice pour suivre le sommet de la pile.

```

1     #include <stdio.h>
2     #include <stdlib.h>
3
4     #define MAX 10
5
6     struct Stack {

```

```

7      int items[MAX];
8      int top;
9  };
10
11     // Fonction pour initialiser la pile
12     void initStack(struct Stack* stack) {
13         stack->top = -1; // La pile est vide au départ
14     }
15
16     // Fonction pour vérifier si la pile est vide
17     int isEmpty(struct Stack* stack) {
18         return stack->top == -1;
19     }

```

Explication : La fonction `initStack` initialise la pile en définissant le sommet à -1, ce qui indique que la pile est vide. La fonction `isEmpty` permet de vérifier si la pile est vide en vérifiant si l'indice du sommet est égal à -1.

2. Insertion d'un élément dans une pile (empilage) :

Réponse : L'empilage consiste à ajouter un élément au sommet de la pile. Voici la fonction pour empiler un élément dans la pile, en vérifiant d'abord si la pile n'est pas pleine.

```

1     // Fonction pour empiler un élément
2     void push(struct Stack* stack, int value) {
3         if (stack->top == MAX - 1) {
4             printf("Erreur : pile pleine\n");
5             return;
6         }
7         stack->items[++stack->top] = value; // Incrémente le
            sommet puis ajoute l'élément
8     }

```

Explication : La fonction `push` ajoute un élément au sommet de la pile. Si le sommet est égal à `MAX - 1`, cela signifie que la pile est pleine et une erreur est renvoyée. Sinon, l'élément est ajouté au sommet, et l'indice du sommet est incrémenté.

3. Suppression d'un élément dans une pile (dépile) :

Réponse : Le dépile consiste à retirer l'élément du sommet de la pile. Si la pile est vide, une erreur est renvoyée. Voici la fonction pour dépiler un élément de la pile :

```

1     // Fonction pour dépiler un élément
2     int pop(struct Stack* stack) {
3         if (isEmpty(stack)) {

```

```

4     printf("Erreur : pile vide\n");
5     return -1; // Valeur spéciale pour indiquer une erreur
6 }
7     return stack->items[stack->top--]; // Retourne l'élément au
        sommet puis décrémente le sommet
8 }

```

Explication : La fonction pop supprime l'élément au sommet de la pile. Si la pile est vide, elle renvoie une erreur. Sinon, elle retourne l'élément au sommet et décrémente l'indice du sommet pour retirer l'élément de la pile.

4. Test de la pile :

Réponse : Voici un exemple de code qui utilise les fonctions d'empilage et de dépilage sur une pile.

```

1     int main() {
2         struct Stack stack;
3         initStack(&stack);
4
5         push(&stack, 10);
6         push(&stack, 20);
7         push(&stack, 30);
8
9         printf("Dépiler : %d\n", pop(&stack)); // Devrait afficher
        30
10        printf("Dépiler : %d\n", pop(&stack)); // Devrait afficher
        20
11        printf("Dépiler : %d\n", pop(&stack)); // Devrait afficher
        10
12
13        return 0;
14    }

```

Explication : Dans cet exemple, nous empilons trois éléments (10, 20, 30) dans la pile et ensuite nous les dépilons un par un. Les éléments sont retirés dans l'ordre inverse de leur insertion, conformément au principe LIFO (Last In, First Out).

Files

1. Construction d'une file :

Réponse : Pour construire une file, vous devez définir une structure qui contient les éléments de la file, ainsi que des pointeurs pour l'avant et l'arrière. Ces pointeurs permettent

de gérer les opérations d'enfilage (ajout d'un élément à l'arrière) et de défilage (retrait du premier élément). Voici un exemple de structure de file en C :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Node {
5      int data;
6      struct Node* next;
7  };
8
9  struct Queue {
10     struct Node* front;
11     struct Node* rear;
12 };
13
14 // Fonction pour initialiser la file
15 void initQueue(struct Queue* q) {
16     q->front = NULL;
17     q->rear = NULL;
18 }
```

Explication : La structure Queue contient deux pointeurs, front et rear, qui pointent respectivement vers le premier et le dernier élément de la file. La fonction initQueue initialise ces pointeurs à NULL pour indiquer que la file est vide.

2. Insertion d'un élément dans une file (enfilage) :

Réponse : L'enfilage consiste à ajouter un élément à la fin de la file. Si la file est vide, l'élément ajouté devient à la fois l'avant et l'arrière de la file. Si la file contient déjà des éléments, vous devez ajuster le pointeur rear pour indiquer le nouvel élément.

```

1  // Fonction pour enfiler un élément
2  void enqueue(struct Queue* q, int value) {
3      struct Node* newNode = malloc(sizeof(struct Node));
4      newNode->data = value;
5      newNode->next = NULL;
6
7      if (q->rear == NULL) { // Si la file est vide
8          q->front = newNode;
9          q->rear = newNode;
10     } else {
```

```

11     q->rear->next = newNode;
12     q->rear = newNode;
13 }
14 }

```

Explication : La fonction enqueue crée un nouveau nœud et le place à la fin de la file. Si la file est vide, l'élément ajouté devient à la fois le premier et le dernier élément. Sinon, le nouvel élément devient le dernier de la file, et rear est mis à jour pour pointer vers ce nouvel élément.

3. Suppression d'un élément dans une file (défilage) :

Réponse : Le défilage consiste à retirer le premier élément de la file. Si la file est vide, la fonction retourne une erreur ou une valeur spéciale. Sinon, elle met à jour le pointeur front pour qu'il pointe vers le prochain élément de la file et libère la mémoire du nœud retiré.

```

1 // Fonction pour défiler un élément
2 int dequeue(struct Queue* q) {
3     if (q->front == NULL) {
4         printf("La file est vide!\n");
5         return -1; // Valeur d'erreur
6     }
7
8     struct Node* temp = q->front;
9     int value = temp->data;
10    q->front = q->front->next;
11
12    if (q->front == NULL) { // Si la file devient vide après le
        défilage
13        q->rear = NULL;
14    }
15
16    free(temp);
17    return value;
18 }

```

Explication : La fonction dequeue retire l'élément au début de la file en mettant à jour le pointeur front. Si la file devient vide après le défilage, le pointeur rear est également mis à NULL. La mémoire du nœud retiré est libérée, et la fonction retourne la valeur de l'élément retiré.

Arbres binaires

1. Construction d'un arbre binaire :

Réponse : Pour construire un arbre binaire, vous devez définir une structure qui représente chaque nœud de l'arbre. Chaque nœud contient des données et deux pointeurs, un pour l'enfant gauche et un pour l'enfant droit. L'arbre commence avec un seul nœud racine, et les autres nœuds sont ajoutés en fonction de leur relation avec la racine. Voici un exemple de code en C pour définir la structure d'un arbre binaire :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Node {
5      int data;
6      struct Node* left;
7      struct Node* right;
8  };
9
10 // Fonction pour créer un nouveau nœud
11 struct Node* createNode(int data) {
12     struct Node* newNode = malloc(sizeof(struct Node));
13     newNode->data = data;
14     newNode->left = NULL;
15     newNode->right = NULL;
16     return newNode;
17 }
18
19 // Fonction pour insérer un élément dans l'arbre binaire
20 void insert(struct Node** root, int data) {
21     if (*root == NULL) {
22         *root = createNode(data);
23         return;
24     }
25     if (data < (*root)->data)
26         insert(&(*root)->left, data);
27     else
28         insert(&(*root)->right, data);
29 }
```

Explication : La structure Node contient un entier data et deux pointeurs left et right

qui pointent respectivement vers l'enfant gauche et l'enfant droit. La fonction `createNode` crée un nouveau nœud avec la valeur donnée, et la fonction `insert` insère un élément dans l'arbre en suivant la règle de l'arbre binaire de recherche (les éléments plus petits vont à gauche et les plus grands à droite).

2. Insertion dans un arbre binaire :

Réponse : L'insertion dans un arbre binaire se fait en suivant la règle de comparaison avec la racine. Si le nouvel élément est plus petit que la racine, il va à gauche, sinon il va à droite. Le processus continue récursivement jusqu'à ce qu'un emplacement vide soit trouvé pour le nouvel élément.

Voici l'exemple d'insertion dans un arbre binaire :

```

1      int main() {
2          struct Node* root = NULL;
3          insert(&root, 50); // Insertion de 50 à la racine
4          insert(&root, 30); // Insertion de 30 à gauche de 50
5          insert(&root, 70); // Insertion de 70 à droite de 50
6          insert(&root, 20); // Insertion de 20 à gauche de 30
7          insert(&root, 40); // Insertion de 40 à droite de 30
8          insert(&root, 60); // Insertion de 60 à gauche de 70
9          insert(&root, 80); // Insertion de 80 à droite de 70
10         return 0;
11     }
```

Explication : Dans cet exemple, l'arbre est construit à partir de la racine et chaque élément est inséré à la bonne position en fonction de sa valeur par rapport à la racine. Les éléments plus petits que la racine vont à gauche et ceux plus grands vont à droite. L'insertion est effectuée récursivement jusqu'à ce que l'emplacement correct soit trouvé.

3. Algorithmes de parcours des arbres binaires :

Réponse : Les algorithmes de parcours des arbres binaires sont utilisés pour visiter tous les nœuds de l'arbre dans un ordre spécifique. Il existe trois principaux types de parcours d'arbres binaires :

- **Parcours pré-ordre :** Visite la racine, puis le sous-arbre gauche et enfin le sous-arbre droit.
- **Parcours en-ordre :** Visite le sous-arbre gauche, puis la racine, et enfin le sous-arbre droit. Ce type de parcours est utilisé dans les arbres binaires de recherche pour parcourir les éléments dans l'ordre croissant.
- **Parcours post-ordre :** Visite le sous-arbre gauche, puis le sous-arbre droit, et enfin la racine.

Exemple de code pour un parcours en-ordre en C :

```

1 // Fonction de parcours en-ordre
2 void inorder(struct Node* root) {
3     if (root == NULL) {
4         return;
5     }
6     inorder(root->left); // Visiter le sous-arbre gauche
7     printf("%d ", root->data); // Visiter la racine
8     inorder(root->right); // Visiter le sous-arbre droit
9 }
10
11 // Fonction pour afficher l'arbre en-ordre
12 int main() {
13     struct Node* root = NULL;
14     insert(&root, 50);
15     insert(&root, 30);
16     insert(&root, 70);
17     insert(&root, 20);
18     insert(&root, 40);
19     insert(&root, 60);
20     insert(&root, 80);
21
22     printf("Parcours en-ordre : ");
23     inorder(root); // Afficher les éléments de l'arbre en-ordre
24     return 0;
25 }

```

Explication : La fonction `inorder` parcourt l'arbre en-ordre en visitant d'abord le sous-arbre gauche, puis la racine, puis le sous-arbre droit. Le résultat du parcours en-ordre d'un arbre binaire de recherche est toujours les éléments dans l'ordre croissant.

Graphes

1. Construction d'un graphe :

Réponse : Pour construire un graphe, il faut définir une structure de données qui contient les sommets (ou nœuds) ainsi que les arêtes qui les relient. On peut utiliser une structure avec des listes d'adjacence ou des matrices d'adjacence pour représenter le graphe. Dans ce cas, nous allons utiliser des listes d'adjacence, où chaque sommet a une liste de connexions vers les autres sommets.

Voici un exemple de code C pour définir un graphe avec des listes d'adjacence :


```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Structure représentant un œnœud
5  struct Node {
6      int vertex;
7      struct Node* next;
8  };
9
10 // Structure représentant un graphe
11 struct Graph {
12     int numVertices;
13     struct Node** adjLists;
14 };
15
16 // Fonction pour créer un œnœud
17 struct Node* createNode(int vertex) {
18     struct Node* newNode = malloc(sizeof(struct Node));
19     newNode->vertex = vertex;
20     newNode->next = NULL;
21     return newNode;
22 }
23
24 // Fonction pour créer un graphe avec un nombre de sommets
    donné
25 struct Graph* createGraph(int vertices) {
26     struct Graph* graph = malloc(sizeof(struct Graph));
27     graph->numVertices = vertices;
28     graph->adjLists = malloc(vertices * sizeof(struct Node*));
29
30     for (int i = 0; i < vertices; i++) {
31         graph->adjLists[i] = NULL; // Initialisation de chaque
            liste d'adjacence
32     }
33     return graph;
34 }

```

Explication : Dans ce code, nous avons une structure Node qui représente un nœud avec

un sommet et un pointeur vers le nœud suivant (pour former une liste chaînée). La structure Graph contient un tableau de pointeurs vers les listes d'adjacence. Chaque élément de ce tableau représente un sommet du graphe, et chaque liste contient les sommets connectés à celui-ci.

2. Ajout d'arêtes dans un graphe :

Réponse : Pour ajouter des arêtes dans un graphe, on insère un nouveau nœud dans la liste d'adjacence du sommet de départ (source) et dans la liste d'adjacence du sommet de destination. Si le graphe est orienté, l'arête va seulement du sommet source vers le sommet destination. Si le graphe est non orienté, l'arête est bidirectionnelle, donc on doit ajouter des connexions dans les deux directions.

Voici un exemple de fonction en C pour ajouter une arête dans un graphe non orienté :

```

1 // Fonction pour ajouter une arête entre deux sommets dans un
   // graphe non orienté
2 void addEdge(struct Graph* graph, int src, int dest) {
3     // Ajouter une arête de src à dest
4     struct Node* newNode = createNode(dest);
5     newNode->next = graph->adjLists[src];
6     graph->adjLists[src] = newNode;
7
8     // Ajouter une arête de dest à src (pour un graphe non orient
   // é)
9     newNode = createNode(src);
10    newNode->next = graph->adjLists[dest];
11    graph->adjLists[dest] = newNode;
12 }
```

Explication : La fonction addEdge crée un nouveau nœud pour chaque arête et l'ajoute à la liste d'adjacence correspondante. Dans le cas d'un graphe non orienté, il y a deux ajouts : un de src à dest et un de dest à src, ce qui rend l'arête bidirectionnelle.

3. Recherche en largeur (Complétez le code) :

Réponse : Pour implémenter la recherche en largeur (BFS), on utilise une file (FIFO) pour explorer chaque niveau du graphe. On commence à partir du sommet source et on explore tous ses voisins, puis les voisins de ces voisins, et ainsi de suite.

Voici un exemple de code pour implémenter la recherche en largeur (BFS) :

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4
```

```

5 // Structure représentant une file
6 struct Queue {
7     int* items;
8     int front, rear, size;
9 };
10
11 // Fonction pour initialiser la file
12 void initQueue(struct Queue* q, int size) {
13     q->items = malloc(size * sizeof(int));
14     q->front = 0;
15     q->rear = 0;
16     q->size = size;
17 }
18
19 // Fonction pour ajouter un élément à la file
20 void enqueue(struct Queue* q, int item) {
21     q->items[q->rear++] = item;
22 }
23
24 // Fonction pour retirer un élément de la file
25 int dequeue(struct Queue* q) {
26     return q->items[q->front++];
27 }
28
29 // Fonction pour vérifier si la file est vide
30 bool isEmpty(struct Queue* q) {
31     return q->front == q->rear;
32 }
33
34 // Fonction pour effectuer une recherche en largeur (BFS)
35 void bfs(struct Graph* graph, int start) {
36     bool* visited = malloc(graph->numVertices * sizeof(bool));
37     for (int i = 0; i < graph->numVertices; i++) {
38         visited[i] = false;
39     }
40
41     struct Queue q;
42     initQueue(&q, graph->numVertices);

```

```

43
44     visited [ start ] = true;
45     enqueue(&q, start);
46
47     while (!isEmpty(&q)) {
48         int current = dequeue(&q);
49         printf ("%d ", current);
50
51         struct Node* temp = graph->adjLists[current];
52         while (temp) {
53             if (! visited [temp->vertex]) {
54                 visited [temp->vertex] = true;
55                 enqueue(&q, temp->vertex);
56             }
57             temp = temp->next;
58         }
59     }
60 }

```

Explication : La fonction bfs commence par marquer le sommet source comme visité et l'ajoute à la file. Ensuite, elle explore tous les voisins du sommet actuel, en ajoutant les non visités à la file. Cela continue jusqu'à ce que tous les sommets accessibles aient été visités.

6.5 Applications d'algorithmes en IA

Régression Linéaire

Données

Les données d'entrée sont les suivantes :

TABLE 6.1 – Données pour la régression linéaire

Surface (m ²)	Prix (MGA)
50	70,000,000
75	95,000,000
100	130,000,000

Nous allons utiliser ces données pour calculer les coefficients a et b de la régression linéaire, puis prédire y pour $x = 80$.

Implémentation en C

Voici l'implémentation en C pour résoudre l'exercice :

Listing 6.1 – Implémentation de la Régression Linéaire en C

```

1  #include <stdio.h>
2
3  // Fonction pour calculer les coefficients a et b
4  void linearRegression(int x[], int y[], int n, float *a, float *b) {
5      int sumX = 0, sumY = 0, sumXY = 0, sumX2 = 0;
6
7      // Calcul des sommes nécessaires
8      for (int i = 0; i < n; i++) {
9          sumX += x[i];
10         sumY += y[i];
11         sumXY += x[i] * y[i];
12         sumX2 += x[i] * x[i];
13     }
14
15     // Calcul des coefficients
16     *a = (n * sumXY - sumX * sumY) / (float)(n * sumX2 - sumX *
17         sumX);
18     *b = (sumY - (*a) * sumX) / (float)n;
19 }
20
21 // Fonction pour prédire y à partir de x
22 float predictY(float a, float b, int x) {
23     return a * x + b;
24 }
25
26 int main() {
27     // Données fournies
28     int x[] = {50, 75, 100}; // Surface des maisons en m²
29     int y[] = {70000000, 95000000, 130000000}; // Prix en MGA
30     int n = 3; // Nombre de paires de données
31
32     float a, b;
33
34     // Appel de la fonction de régression linéaire

```

```

34 linearRegression(x, y, n, &a, &b);
35
36 // Affichage des résultats
37 printf("Coefficients de la régression linéaire :\n");
38 printf("a = %.2f\n", a);
39 printf("b = %.2f\n\n", b);
40
41 // Prédire le prix pour x = 80
42 int x_predict = 80;
43 float predicted_y = predictY(a, b, x_predict);
44
45 printf("Le prix prédit pour une surface de %d m² est : %.2f MGA\n",
46       , x_predict, predicted_y);
47
48 return 0;
49 }

```

Explication du Code

Le programme calcule les coefficients a et b de la régression linéaire en utilisant les formules suivantes :

$$a = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{n \sum x_i^2 - (\sum x_i)^2}$$

$$b = \frac{\sum y_i - a \sum x_i}{n}$$

Ensuite, il utilise ces coefficients pour prédire y pour $x = 80$.

Résultats

En exécutant ce programme avec les données fournies, le programme affiche les coefficients de régression et la prédiction pour $x = 80$.

Régression Logistique

Implémentation en C

Voici l'implémentation en C pour résoudre l'exercice :

Listing 6.2 – Implémentation de la Régression Logistique en C

```

1 #include <stdio.h>
2 #include <math.h>

```

```

3
4 // Fonction pour calculer z
5 float computeZ(float beta0, float beta1, float x1) {
6     return beta0 + beta1 * x1;
7 }
8
9 // Fonction sigmoïde pour transformer z en probabilité P
10 float sigmoid(float z) {
11     return 1 / (1 + exp(-z));
12 }
13
14 int main() {
15     // Données d'exemple
16     float beta0 = 0.5; // Coefficient intercept
17     float beta1 = 0.1; // Coefficient pour x1 (temps)
18     float x1 = 12;     // Temps en minutes
19
20     // Calcul de z
21     float z = computeZ(beta0, beta1, x1);
22
23     // Calcul de la probabilité P
24     float P = sigmoid(z);
25
26     // Affichage des résultats
27     printf("Valeur de z : %.2f\n", z);
28     printf("Probabilité P : %.2f\n", P);
29
30     return 0;
31 }

```

Le programme utilise la fonction sigmoïde pour transformer la valeur z calculée en probabilité P , qui peut ensuite être utilisée pour prendre des décisions dans un modèle de régression logistique.

Explication

Le programme calcule d'abord z en utilisant la formule de la régression linéaire :

$$z = \beta_0 + \beta_1 \cdot x_1$$

Ensuite, il applique la fonction sigmoïde :

$$P = \frac{1}{1 + e^{-z}}$$

Cela donne la probabilité P associée à une certaine valeur x_1 (dans ce cas, 12 minutes).

K-means

Voici un programme en C qui résout l'exercice demandé. Le programme suit les étapes suivantes :

- Accepter une liste de points (x, y) .
- Initialiser 2 centroïdes aléatoires.
- Assigner chaque point au centroïde le plus proche.
- Recalculer les centroïdes et afficher les groupes finaux.

Données

Les données d'entrée sont les suivantes :

TABLE 6.2 – Données pour K-means

Point	Coordonnées (x, y)
A	(1, 2)
B	(3, 4)
C	(5, 6)
D	(8, 9)

Implémentation en C

Voici l'implémentation en C pour résoudre l'exercice :

Listing 6.3 – Implémentation de l'algorithme K-means en C

```

1  #include <stdio.h>
2  #include <math.h>
3  #include <stdlib.h>
4
5  // Structure pour un point (x, y)
6  typedef struct {
7      float x;
8      float y;
9  } Point;
10
11 // Fonction pour calculer la distance entre deux points

```



```

12  float distance(Point a, Point b) {
13      return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
14  }
15
16  // Fonction pour initialiser les centroïdes aléatoires
17  void initializeCentroids (Point points [], int k, Point centroids []) {
18      centroids [0] = points [0]; // Initialisation manuelle pour le
19      // premier centroïde
20      centroids [1] = points [3]; // Initialisation manuelle pour le
21      // deuxième centroïde
22  }
23
24  // Fonction pour assigner chaque point au centroïde le plus proche
25  void assignPointsToCentroids(Point points [], int n, Point centroids [],
26  // int clusters []) {
27      for (int i = 0; i < n; i++) {
28          float minDistance = distance(points[i], centroids [0]);
29          clusters [i] = 0;
30
31          for (int j = 1; j < 2; j++) {
32              float dist = distance(points[i], centroids [j]);
33              if (dist < minDistance) {
34                  minDistance = dist;
35                  clusters [i] = j;
36              }
37          }
38      }
39  }
40
41  // Fonction pour recalculer les centroïdes
42  void recalculateCentroids(Point points [], int n, Point centroids [], int
43  // clusters []) {
44      float sumX[2] = {0, 0}, sumY[2] = {0, 0};
45      int count[2] = {0, 0};
46
47      for (int i = 0; i < n; i++) {
48          sumX[clusters[i]] += points[i].x;
49          sumY[clusters[i]] += points[i].y;
50      }
51  }

```

```

46     count[clusters[i]]++;
47 }
48
49 for (int i = 0; i < 2; i++) {
50     centroids[i].x = sumX[i] / count[i];
51     centroids[i].y = sumY[i] / count[i];
52 }
53 }
54
55 int main() {
56     // Données des points
57     Point points[] = {{1, 2}, {3, 4}, {5, 6}, {8, 9}};
58     int n = 4; // Nombre de points
59
60     // Initialisation des centroïdes
61     Point centroids[2];
62     initializeCentroids(points, 2, centroids);
63
64     // Assignment des points aux centroïdes
65     int clusters[4];
66     assignPointsToCentroids(points, n, centroids, clusters);
67
68     // Affichage des groupes après la première assignation
69     printf("Assignment des points aux clusters :\n");
70     for (int i = 0; i < n; i++) {
71         printf("Point (%f, %f) -> Cluster %d\n", points[i].x, points[i].y,
72             clusters[i]);
73     }
74
75     // Recalcul des centroïdes
76     recalculateCentroids(points, n, centroids, clusters);
77
78     // Affichage des nouveaux centroïdes
79     printf("\nNouveaux centroïdes :\n");
80     for (int i = 0; i < 2; i++) {
81         printf("Centroïde %d : (%f, %f)\n", i + 1, centroids[i].x,
82             centroids[i].y);

```

```
82  
83     return 0;  
84 }
```

Le programme fonctionne comme suit : 1. Il initialise deux centroïdes à partir des points donnés (ici, manuellement sélectionnés pour simplifier). 2. Chaque point est assigné au centroïde le plus proche. 3. Les centroïdes sont recalculés en prenant la moyenne des points de chaque groupe. 4. Enfin, les résultats sont affichés, montrant les groupes de points ainsi que les nouveaux centroïdes.

Explication des résultats

Après avoir exécuté le programme, vous obtiendrez l'assignation des points à leurs clusters respectifs et les nouveaux centroïdes recalculés. L'algorithme peut être itéré plusieurs fois pour améliorer les résultats.

