Kajy University: Informatique

Algorithmes et structures de données

Author: Dimby Rabearivony

Date: 8 mai 2024

Version: 1.0

"Ny fahalalana no valamparihiko."

Table des matières

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

————————-o0o————————-

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.

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 language 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 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.

1.1.1 Déclaration de variables

En language 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"
```

1.1.2 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'
```

1.1.3 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
```

1.1.4 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 element essentiel en langage C et constituent la base de la manipulation des donnees dans les programmes. Il est crucial de comprendre leur declaration, leur initialisation, leur utilisation et leur portee pour ecrire 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.

1.2.1 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
```

1.2.2 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
```

1.2.3 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.

1.2.4 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 pointe par ptr est %d\\n", *ptr); //
       Affichage de la valeur pointee par ptr
7  printf("La valeur pointee par double_ptr est %d\\n", **
       double_ptr); // Affichage de la valeur pointee 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.

1.2.5 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++; // Deplacement 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.

1.2.6 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.

1.3.1 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
```

1.3.2 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 aujourdhui = MERCREDI; // Declaration d'une
          enumeration et initialisation
```

1.3.3 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, SAMEDI,
       DIMANCHE} Jour;
2
3  Jour aujourdhui = 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

1.4.1 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**

1.4.2 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.

1.4.3 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.

1.4.4 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.

1.4.5 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 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 :

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
```

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 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. Rapellons 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
```

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
```

Algorithm 7 Tri par tas (HeapSort)

```
1: function HeapSort(tableau)
2:     Construire le tas
3:     Pour i de n - 1 à 1 Faire
4:         échanger tableau[0] et tableau[i]
5:         Réorganiser le tas pour maintenir la propriété du tas
6:     Fin Pour
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.

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 par hachage (Hash Table Search)** : Ce type de recherche utilise une table de hachage pour trouver rapidement des éléments. Une fonction de hachage convertit chaque élément en une clé unique pour le stockage et la recherche. C'est très rapide pour les grandes listes, mais peut être complexe à mettre en place.
4. **Recherche dans un arbre binaire (Binary Tree Search)** : Cette recherche utilise une structure appelée arbre binaire, où chaque nœud a jusqu'à deux enfants. La recherche se fait en parcourant les nœuds selon des règles spécifiques, ce qui peut être efficace pour certaines opérations de recherche.
5. **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.
6. **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.

2.5 Exercices

1. Pour chacun des algorithmes de tri cités dans ce chapitre :
   (a). Implémentez l'algorithme en C.

Algorithm 8 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
```

Algorithm 9 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 10 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 11 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
```

   (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 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 12 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 13 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.

3.4.1 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 14 Accès direct à un élément du tableau

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

3.4.2 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 15 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
```

3.4.3 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.

Algorithm 16 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
```

3.4.4 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 17 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

1. Quelle est la complexité temporelle des opérations suivantes ?
   (a). Parcourir un tableau d'entiers de longueur n.
   (b). Multiplier deux matrices de taille $n \times n$.
   (c). Effectuer une recherche binaire sur un tableau trié.
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  }
```

   Comment pouvez-vous améliorer cet algorithme pour réduire sa complexité ?
3. 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 á bulle sur un tableau de longueur 'n'..
   (d). Effectuer un tri par insertion sur un tableau de longueur 'n'.

Chapitre 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.

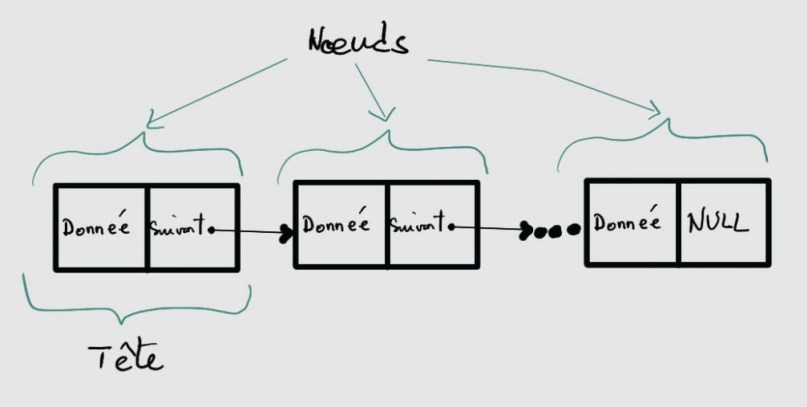

FIGURE 4.1 – Illustration d'une liste chaînée.

4.1.1 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.

4.1.2 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.

4.1.3 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.

4.1.4 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.

Voici un exemple d'implémentation d'une liste chaînée en C : [Repository containing codes to be pasted here]

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.

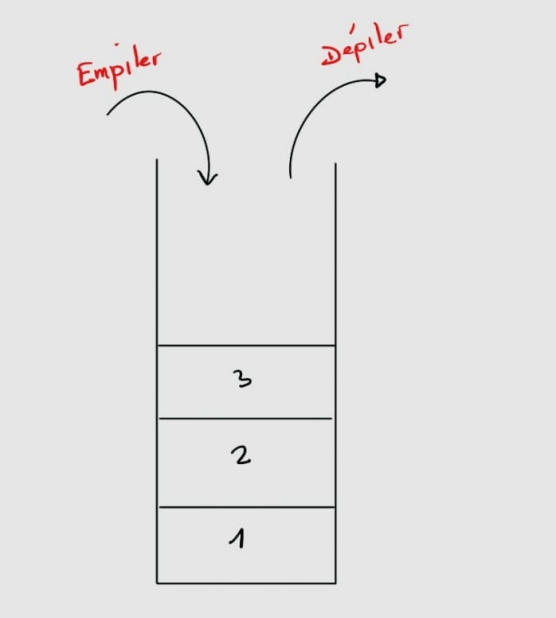

FIGURE 4.2 – Illustration d'une pile (LIFO), où le dernier élément ajouté est le premier retiré.

4.2.1 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).

4.2.2 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.

4.2.3 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.

4.2.4 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.

4.2.5 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.

4.2.6 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.

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.

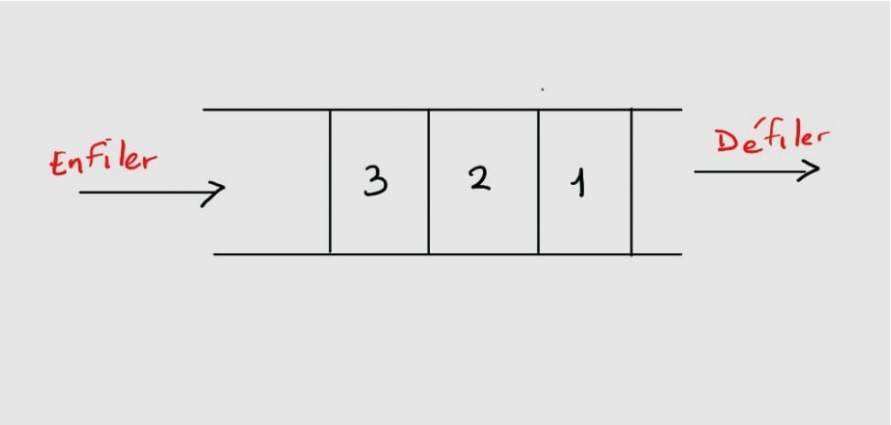

Figure 4.3 – Illustration d'une file (FIFO), où le premier élément ajouté est le premier retiré.

4.3.1 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).

4.3.2 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.

4.3.3 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.

4.3.4 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.

4.3.5 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.

4.3.6 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.

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.

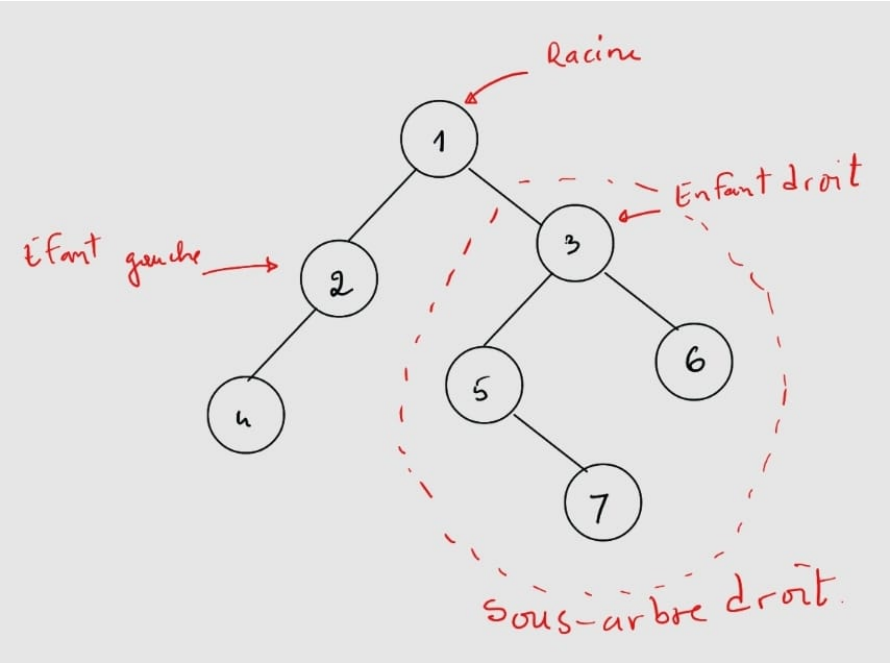

Figure 4.4 – Illustration d'un arbre binaire

4.4.1 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.

4.4.2 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.

4.4.3 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).

4.4.4 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.

4.4.5 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.

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.

4.5.1 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.

4.5.2 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.

4.5.3 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.

4.5.4 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)**

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)**

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.

4.5.5 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.

Chapitre Applications d'algorithmes en IA

5.1 Greedy algorithm

Bibliographie

[1] Karumanchi, N. (2017). Data structures and algorithms made easy.