
# Kajy University: Informatique

## Algorithmes et structures de données

**Author:** Dimby Rabearivony

**Date:** 28 avril 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

# 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 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

**Types de données :**

1. Écrivez un programme C pour déclarer une variable de chaque type de données de base et initialisez-les avec des valeurs.
2. Déclarez un tableau d'entiers de taille 10 et initialisez-le avec des valeurs de votre choix. Affichez ensuite ces valeurs à l'écran.
3. Créez une structure `Personne` avec des champs pour le nom, l'âge et la taille. Déclarez une variable de type `Personne` et initialisez-la avec des valeurs fictives. Affichez ensuite ces valeurs à l'écran.
4. Écrivez une fonction en C pour inverser une chaîne de caractères donnée.
5. Écrivez une fonction en C pour trier un tableau d'entiers en utilisant l'algorithme de tri à bulles.

**Types de données de base :**

6. Écrivez un programme C pour convertir un nombre entier en binaire.
7. Écrivez une fonction en C pour calculer la somme des chiffres d'un nombre entier.
8. Déclarez une variable de type `char` et utilisez-la pour stocker une lettre majuscule. Ensuite, utilisez une opération pour convertir cette lettre en minuscule.
9. Écrivez un programme C pour vérifier si un nombre donné est premier ou non.
10. Écrivez une fonction récursive en C pour calculer le factoriel d'un nombre entier.

**Types de données dérivés :**

11. Écrivez une fonction en C pour concaténer deux chaînes de caractères données.
12. Déclarez un tableau de structures `Etudiant` avec des champs pour le nom, l'âge et la moyenne. Initialisez-le avec des valeurs fictives et affichez ensuite ces valeurs à l'écran.
13. Écrivez une fonction en C pour ajouter un élément à une liste chaînée.
14. Implémentez une file (queue) en utilisant des listes chaînées en C.
15. Écrivez une fonction en C pour supprimer un élément d'un arbre binaire de recherche.

**Typedef**

16. **Création d'un alias pour un type de base**

    Utilisez 'typedef' pour créer un alias pour un type de données de base. Ensuite, déclarez une variable de ce type et affichez sa valeur.

```
1  typedef ...
2  ... var = 25;
3
4  printf("La␣valeur␣de␣var␣est␣%d\\n", var);
```

17. **Utilisation de typedef avec des pointeurs**

    Utilisez 'typedef' pour créer un alias pour un pointeur vers un entier. Initialisez le pointeur

avec l'adresse d'une variable entière et affichez la valeur à laquelle le pointeur fait référence.

```
1  typedef ...
2  int a = 10;
3  ... ptr = &a;
4
5  printf("La␣valeur␣pointee␣par␣le␣pointeur␣est␣%d\\n", *
       ptr);
```

18. **Utilisation de typedef avec des structures**

Utilisez 'typedef' pour créer un alias pour une structure représentant une personne avec des champs comme le nom, l'âge, et la taille. Ensuite, déclarez une variable de ce type et attribuez des valeurs aux champs.

```
1  typedef struct {
2    char nom[50];
3    int age;
4    float taille;
5  } ...;
6
7  ... p1;
8
9  strcpy(p1.nom, "Alice");
10 p1.age = 25;
11 p1.taille = 1.65;
```

19. **Utilisation de typedef avec des énumérations**

Utilisez 'typedef' pour créer un alias pour une énumération représentant les jours de la semaine. Déclarez une variable de ce type et attribuez-lui un jour spécifique.

```
1  typedef enum {
2    LUNDI,
3    MARDI,
4    MERCREDI,
5    ...
6  } ...;
7
8  ... aujourd'hui␣=␣MERCREDI;
9
␣␣
```

20. **Utilisation de typedef avec des unions**

Utilisez 'typedef' pour créer un alias pour une union contenant un entier, un flottant, et un

caractère. Déclarez une variable de ce type et attribuez des valeurs aux différents éléments de l'union.

```
1  typedef union {
2    int entier;
3    float flottant;
4    char caractere;
5  } ...;
6
7  ... d;
8  d.entier = 10;
9  d.flottant = 5.5;
```

# Chapitre Algorithmes de recherche et de tri

## 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

Pour pratiquer les algorithmes de recherche, voici quelques exercices qui vous aideront à comprendre et à appliquer ces concepts :

(a). **Recherche linéaire** : Implémentez une recherche linéaire pour trouver un élément spécifique dans un tableau. Quelles sont les limites de cette méthode ?

(b). **Recherche binaire** : Implémentez une recherche binaire sur un tableau trié. Que se passe-t-il si le tableau n'est pas trié ?

(c). **Recherche par hachage** : Expliquez comment une table de hachage peut accélérer la recherche. Quels sont les risques associés à cette méthode ?

(d). **Recherche interpolée** : Pourquoi la recherche interpolée est-elle plus efficace que la recherche binaire pour les listes avec des valeurs bien réparties ? Implémentez cet algorithme et montrez ses avantages.

(e). **Recherche par saut** : Implémentez une recherche par saut sur un tableau de grande taille. Pourquoi cette méthode peut-elle être plus rapide que la recherche linéaire pour des tableaux de grande taille ?

**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
```

# 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 $O$, $\Omega$, et $\theta$.

## 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 $\theta$

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 ($\theta$)** : 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
```

# 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.

Exemple d'implémentation d'une liste chaînée en C :

```c
typedef struct Node {
  int data;
  struct Node* next;
} Node;

void insert(Node** head, int new_data) {
  Node* new_node = (Node*)malloc(sizeof(Node));
  new_node->data = new_data;
  new_node->next = *head;
  *head = new_node;
}

void delete(Node** head, int key) {
  Node* temp = *head, *prev;
  if (temp != NULL && temp->data == key) {
    *head = temp->next;
    free(temp);
    return;
  }
  while (temp != NULL && temp->data != key) {
    prev = temp;
    temp = temp->next;
  }
  if (temp == NULL) return;
  prev->next = temp->next;
  free(temp);
}
```

## 4.2 Piles

Les piles (ou LIFO, Last In First Out) permettent des opérations de push et pop. Elles suivent le principe du dernier entré, premier sorti. Les piles sont couramment utilisées pour les opérations de retour en arrière, comme dans les navigateurs Web ou les algorithmes de récursion.

Voici un exemple simple d'utilisation de pile pour vérifier les parenthèses équilibrées :

```
int areParenthesesBalanced(char expr[]) {
  Stack* stack = createStack(strlen(expr));
  for (int i = 0; i < strlen(expr); i++) {
    if (expr[i] == '(') {
      push(stack, expr[i]);
    } else if (expr[i] == ')') {
      if (isEmpty(stack)) {
        return 0;
      }
      pop(stack);
    }
  }
  return isEmpty(stack);
}
```

## 4.3 Files

Les files (ou FIFO, First In First Out) permettent des opérations d'enfilage et de défilage. Elles sont utiles pour la gestion des tâches, les files d'attente, et les algorithmes de parcours.

Exemple d'une file utilisant une structure de données basée sur des listes chaînées :

```
typedef struct QueueNode {
  int data;
  struct QueueNode* next;
} QueueNode;

typedef struct Queue {
  QueueNode* front;
  QueueNode* rear;
} Queue;

Queue* createQueue() {
```

```
12    Queue* queue = (Queue*)malloc(sizeof(Queue));
13    queue->front = queue->rear = NULL;
14    return queue;
15  }
16
17  void enqueue(Queue* queue, int data) {
18    QueueNode* new_node = (QueueNode*)malloc(sizeof(QueueNode));
19    new_node->data = data;
20    new_node->next = NULL;
21    if (queue->rear == NULL) {
22      queue->rear = queue->front = new_node;
23    } else {
24      queue->rear->next = new_node;
25      queue->rear = new_node;
26    }
27  }
28
29  int dequeue(Queue* queue) {
30    if (queue->front == NULL) {
31      return -1;
32    }
33    QueueNode* temp = queue->front;
34    queue->front = queue->front->next;
35    if (queue->front == NULL) {
36      queue->rear = NULL;
37    }
38    int data = temp->data;
39    free(temp);
40    return data;
41  }
```

## 4.4 Arbres binaires

Les arbres binaires sont des structures de données où chaque nœud a au maximum deux enfants, généralement appelés gauche et droite. Les arbres binaires sont utilisés pour des opérations de recherche et des algorithmes de parcours comme pré-ordre, en-ordre et post-ordre.

Exemple d'implémentation d'un arbre binaire :

```c
typedef struct TreeNode {
  int data;
  struct TreeNode* left;
  struct TreeNode* right;
} TreeNode;

TreeNode* createNode(int data) {
  TreeNode* new_node = (TreeNode*)malloc(sizeof(TreeNode));
  new_node->data = data;
  new_node->left = NULL;
  new_node->right = NULL;
  return new_node;
}

void insert(TreeNode** root, int data) {
  if (*root == NULL) {
    *root = createNode(data);
  } else {
    if (data < (*root)->data) {
      insert(&(*root)->left, data);
    } else {
      insert(&(*root)->right, data);
    }
  }
}
```

## 4.5 Arbres

Les arbres sont des structures de données génériques qui peuvent avoir plusieurs enfants. Ils sont utilisés dans des contextes comme les arbres de syntaxe abstraite, les arbres de préfixes, et les arbres de décision.

Exemple d'utilisation d'un arbre pour un arbre de décision :

```c
typedef struct DecisionNode {
  char* question;
  struct DecisionNode* yes;
  struct DecisionNode* no;
} DecisionNode;

```

```
7  DecisionNode* createDecisionNode(char* question) {
8    DecisionNode* node = (DecisionNode*)malloc(sizeof(
         DecisionNode));
9    node->question = question;
10   node->yes = NULL;
11   node->no = NULL;
12   return node;
13 }
14
15 void addDecision(DecisionNode* root, char* question, int
       answer) {
16   if (answer == 1) {
17     root->yes = createDecisionNode(question);
18   } else {
19     root->no = createDecisionNode(question);
20   }
21 }
```

## 4.6 Graphes

Les graphes sont des structures de données qui contiennent des nœuds (ou sommets) et des arêtes (qui relient les nœuds). Les graphes sont utilisés pour des problèmes complexes comme la recherche de plus courts chemins, la détection de cycles, et le partitionnement.

Exemple d'implémentation d'un graphe avec des listes d'adjacence :

```
1  typedef struct GraphNode {
2    int vertex;
3    struct GraphNode* next;
4  } GraphNode;
5
6  typedef struct Graph {
7    int num_vertices;
8    GraphNode** adj_lists;
9  } Graph;
10
11 Graph* createGraph(int num_vertices) {
12   Graph* graph = (Graph*)malloc(sizeof(Graph));
13   graph->num_vertices = num_vertices;
```

```
14      graph->adj_lists = (GraphNode**)malloc(num_vertices * sizeof
            (GraphNode*));
15      for (int i = 0; i < num_vertices; i++) {
16        graph->adj_lists[i] = NULL;
17      }
18      return graph;
19    }
20
21    void addEdge(Graph* graph, int src, int dest) {
22      GraphNode* new_node = (GraphNode*)malloc(sizeof(GraphNode));
23      new_node->vertex = src;
24      new_node->next = graph->adj_lists[dest];
25      graph->adj_lists[dest] = new_node;
26
27      new_node = (GraphNode*)malloc(sizeof(GraphNode));
28      new_node->vertex = dest;
29      new_node->next = graph's adj_lists[src];
30  graph's adj_lists[src] = new_node;
31    }
```

## 4.7 Exercices

Voici quelques exercices pour vous aider à pratiquer et consolider vos connaissances des structures de données avancées :

1. Implémentez une liste chaînée avec des opérations d'insertion et de suppression. Testez votre implémentation avec plusieurs cas d'utilisation.
2. Créez un algorithme qui utilise une pile pour évaluer une expression mathématique simple, comme "3 + 5 * 2".
3. Implémentez une file qui suit le principe FIFO (First In, First Out) avec des opérations d'enfilage et de défilage. Testez-la avec différents types de données.
4. Créez un arbre binaire qui peut insérer des valeurs en maintenant une structure de recherche binaire. Ajoutez des fonctions de recherche, de suppression, et de parcours.
5. Implémentez un graphe qui utilise des listes d'adjacence pour représenter des connexions entre des sommets. Ajoutez des fonctions pour ajouter et supprimer des arêtes, et pour trouver des chemins entre des sommets.

# Chapitre  Applications d'algorithmes en IA

## 5.1 Greedy algorithm

# Bibliographie

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

[2] Knuth, D. E. (1986). *The TEX Book*. Addison-Wesley Professional.