**Structures de Données et Algorithmes en C#**

Les **structures de données** et les **algorithmes** sont essentiels en programmation pour **stocker, organiser et manipuler** efficacement les données.

**1️⃣ Structures de Données en C#**

Les structures de données sont des **conteneurs** qui stockent et organisent des données de manière efficace.

## **1.1 Tableaux (**Array**)**

* Stocke **plusieurs éléments** du même type.
* Taille **fixe**.

### **Exemple :**

**int[] nombres = { 1, 2, 3, 4, 5 };**

**Console.WriteLine(nombres[2]); // ✅ Affiche 3**

**Limitation** : Impossible de modifier la taille après création.

## **1.2 Listes (**List<T>**)**

* Taille **dynamique**.
* Fournit **plus de méthodes** que les tableaux (Add(), Remove()…).

### **Exemple :**

**List<int> nombres = new List<int> { 1, 2, 3 };**

**nombres.Add(4); // ✅ Ajoute 4 à la liste**

**Console.WriteLine(nombres.Count); // ✅ Affiche 4**

**Avantage** : Flexible et facile à manipuler.

## **1.3 Dictionnaires (**Dictionary<TKey, TValue>**)**

* Stocke des **paires clé-valeur**.
* Accès rapide aux valeurs via la clé.

### **Exemple :**

**Dictionary<string, int> ages = new Dictionary<string, int>();**

**ages["Alice"] = 25;**

**ages["Bob"] = 30;**

**Console.WriteLine(ages["Alice"]); // ✅ Affiche 25**

**Avantage** : Accès aux données en **temps constant O(1)**.

## **1.4 Files (**Queue<T>**)**

* **Premier arrivé, premier sorti** (FIFO).

### **Exemple :**

**Queue<string> fileAttente = new Queue<string>();**

**fileAttente.Enqueue("Alice");**

**fileAttente.Enqueue("Bob");**

**Console.WriteLine(fileAttente.Dequeue()); // ✅ Alice sort en premier**

## **1.5 Piles (**Stack<T>**)**

* **Dernier arrivé, premier sorti** (LIFO).

### **Exemple :**

**Stack<string> historique = new Stack<string>();**

**historique.Push("Page1");**

**historique.Push("Page2");**

**Console.WriteLine(historique.Pop()); // ✅ Page2 sort en premier**

**Utilisation** : Annulation d'actions (Ctrl+Z).

2️⃣ Algorithmes en C#

Les **algorithmes** sont des instructions pour résoudre un problème.

## **2.1 Recherche Linéaire (O(n))**

* Parcourt **chaque élément** pour trouver une valeur.

### **Exemple :**

**int RechercheLineaire(int[] tableau, int cible)**

**{**

**for (int i = 0; i < tableau.Length; i++)**

**{**

**if (tableau[i] == cible)**

**return i; // ✅ Retourne l'index**

**}**

**return -1; // ❌ Pas trouvé**

**}**

**int[] nombres = { 10, 20, 30, 40 };**

**Console.WriteLine(RechercheLineaire(nombres, 30)); // ✅ 2**

**Inconvénient** : Lent sur **grandes** listes.

## **2.2 Recherche Binaire (O(log n))**

* **Plus rapide** mais nécessite un **tableau trié**.

### **Exemple :**

**int RechercheBinaire(int[] tableau, int cible)**

**{**

**int gauche = 0, droite = tableau.Length - 1;**

**while (gauche <= droite)**

**{**

**int milieu = (gauche + droite) / 2;**

**if (tableau[milieu] == cible) return milieu;**

**if (tableau[milieu] < cible) gauche = milieu + 1;**

**else droite = milieu - 1;**

**}**

**return -1; // ❌ Pas trouvé**

**}**

**int[] nombres = { 10, 20, 30, 40, 50 };**

**Console.WriteLine(RechercheBinaire(nombres, 30)); // ✅ 2**

**Avantage** : Très rapide sur **grandes** listes triées.

## **2.3 Tri à Bulles (Bubble Sort) (O(n²))**

* Compare **chaque paire d’éléments** et les échange si nécessaire.

### **Exemple :**

**void TriBulles(int[] tableau)**

**{**

**int n = tableau.Length;**

**for (int i = 0; i < n - 1; i++)**

**{**

**for (int j = 0; j < n - i - 1; j++)**

**{**

**if (tableau[j] > tableau[j + 1])**

**{**

**// Échange**

**int temp = tableau[j];**

**tableau[j] = tableau[j + 1];**

**tableau[j + 1] = temp;**

**}**

**}**

**}**

**}**

**int[] nombres = { 5, 1, 4, 2, 8 };**

**TriBulles(nombres);**

**Console.WriteLine(string.Join(", ", nombres)); // ✅ 1, 2, 4, 5, 8**

**Inconvénient** : Très **lent** pour **grandes** listes.

## **2.4 Tri Rapide (QuickSort) (O(n log n))**

* Utilise **un pivot** pour diviser et trier.

### **Exemple :**

**void QuickSort(int[] tableau, int gauche, int droite)**

**{**

**if (gauche < droite)**

**{**

**int pivot = Partition(tableau, gauche, droite);**

**QuickSort(tableau, gauche, pivot - 1);**

**QuickSort(tableau, pivot + 1, droite);**

**}**

**}**

**int Partition(int[] tableau, int gauche, int droite)**

**{**

**int pivot = tableau[droite];**

**int i = gauche - 1;**

**for (int j = gauche; j < droite; j++)**

**{**

**if (tableau[j] < pivot)**

**{**

**i++;**

**(tableau[i], tableau[j]) = (tableau[j], tableau[i]); // Échange**

**}**

**}**

**(tableau[i + 1], tableau[droite]) = (tableau[droite], tableau[i + 1]); // Place pivot**

**return i + 1;**

**}**

**int[] nombres = { 10, 7, 8, 9, 1, 5 };**

**QuickSort(nombres, 0, nombres.Length - 1);**

**Console.WriteLine(string.Join(", ", nombres)); // ✅ 1, 5, 7, 8, 9, 10**

**Avantage** : Beaucoup plus **rapide** que le tri à bulles.

**3️⃣ Résumé**

| **Structure** | **Caractéristiques** | **Complexité d'Accès** |
| --- | --- | --- |
| **Tableau** | Taille fixe | O(1) |
| **Liste (List<T>)** | Taille dynamique | O(1) |
| **Dictionnaire (Dictionary<K,V>)** | Accès par clé | O(1) |
| **File (Queue<T>)** | FIFO | O(1) |
| **Pile (Stack<T>)** | LIFO | O(1) |
|  |  |  |

| **Algorithme** | **Utilisation** | **Complexité** |
| --- | --- | --- |
| **Recherche linéaire** | Liste non triée | O(n) |
| **Recherche binaire** | Liste triée | O(log n) |
| **Tri à bulles** | Simple mais lent | O(n²) |
| **Tri rapide** | Efficace pour grands jeux de données | O(n log n) |