FICHE ALGORIHTMES DE TRI EN JAVA

Algorithmes de Tri

Les algorithmes de tri sont essentiels en informatique pour organiser des collections de données. Voici quelques algorithmes de tri courants, du plus simple au plus complexe.

- **Tri à Bulles (Bubble Sort)**
  - C'est l'un des algorithmes de tri les plus simples, mais aussi l'un des moins efficaces.
  - Il fonctionne en comparant les éléments adjacents et en les échangeant si ils sont dans le mauvais ordre.
  - Ce processus est répété jusqu'à ce que toute la collection soit triée.
  - Exemple en Java:

```java
public static void bubbleSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // échange arr[j+1] et arr[j]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

- **Tri par Insertion (Insertion Sort)**
  - Cet algorithme construit le tableau final trié un élément à la fois.
  - Il prend un élément et l'insère à sa bonne place dans le tableau trié.
  - Il est plus efficace que le tri à bulles, surtout pour les petits ensembles de données ou les données presque triées.
  - Exemple en Java:

```java
public static void insertionSort(int[] arr) {
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

  - Version récursive:

```java
public static void insertionSortRecursive(int[] arr, int n) {
    if (n <= 1) {
        return;
    }
    insertionSortRecursive(arr, n - 1);
    int key = arr[n - 1];
    int j = n - 2;
    while (j >= 0 && arr[j] > key) {
        arr[j + 1] = arr[j];
        j--;
    }
```

```
        arr[j + 1] = key;
    }
```

- **Tri par Sélection (Selection Sort)**
  - Il trouve l'élément minimum de la collection et le place au début.
  - Ce processus est répété pour le reste de la collection.
  - Il est simple à comprendre mais pas très efficace pour les grands ensembles de données.
  - Exemple en Java:

```
public static void selectionSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

  - Version récursive:

```
public static void selectionSortRecursive(int[] arr, int n, int index) {
    if (index >= n - 1) {
        return;
    }
    int min_idx = index;
    for (int j = index + 1; j < n; j++) {
        if (arr[j] < arr[min_idx]) {
            min_idx = j;
        }
    }
    int temp = arr[min_idx];
    arr[min_idx] = arr[index];
    arr[index] = temp;
   selectionSortRecursive(arr, n, index+1);
}
```

- **Tri Fusion (Merge Sort)**
  - Un algorithme de tri "diviser pour régner" efficace.
  - Il divise la collection en sous-collections, les trie récursivement, puis les fusionne.
  - Il a une complexité temporelle de O(n log n) dans tous les cas, ce qui le rend plus efficace pour les grands ensembles de données.
  - Exemple en Java:

```
public static void mergeSort(int[] arr, int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
public static void merge(int[] arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
```

```
        int[] L = new int[n1];
        int[] R = new int[n2];
        for (int i = 0; i < n1; ++i)
            L[i] = arr[l + i];
        for (int j = 0; j < n2; ++j)
            R[j] = arr[m + 1 + j];
        int i = 0, j = 0;
        int k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }
```

- **Tri Rapide (Quick Sort)**
  - Autre algorithme de tri “diviser pour régner” efficace, mais sa performance peut varier en fonction du choix du pivot.
  - Il sélectionne un élément “pivot” et partitionne la collection autour de ce pivot.
  - Il est généralement plus rapide que le tri fusion en pratique, mais sa complexité dans le pire des cas est de O(n^2).
  - Exemple en Java:

```
    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }

    public static int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i = (low - 1);
        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
```

```
        return i + 1;
    }
```

Itérateurs

Un **itérateur** est un objet qui permet de parcourir une collection d'éléments. En Java, l'interface `Iterator` est utilisée pour itérer sur les collections.

- **Fonctionnement:**
  - Un itérateur fournit une manière standard de parcourir une collection, sans avoir à connaître la structure interne de la collection.
  - Il permet de vérifier s'il reste des éléments à parcourir (`hasNext()`), d'obtenir l'élément suivant (`next()`), et de supprimer l'élément courant (`remove()`).
- **Méthodes principales de l'interface `Iterator`:**
  - `boolean hasNext()`: Retourne `true` s'il y a encore des éléments à parcourir.
  - `E next()`: Retourne l'élément suivant dans la collection et avance l'itérateur.
  - `void remove()`: Supprime le dernier élément retourné par `next()`. (optionnel)
- **Exemple d'utilisation avec `HashSet`:**

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("A");
        set.add("B");
        set.add("C");

        Iterator<String> iterator = set.iterator();
        while (iterator.hasNext()) {
            String element = iterator.next();
            System.out.println(element);
        }
    }
}
```

- **ListIterator**:
  - L'interface `ListIterator` est une extension de `Iterator` et est spécifiquement utilisée pour les listes (`List`).
  - Elle permet d'itérer dans les deux sens (avant et arrière), de modifier la liste pendant l'itération et d'obtenir l'index de l'élément courant.
  - **Méthodes supplémentaires de `ListIterator`**:
    - `boolean hasPrevious()`: Retourne `true` s'il y a des éléments précédents.
    - `E previous()`: Retourne l'élément précédent.
    - `int nextIndex()`: Retourne l'index de l'élément qui serait retourné par `next()`.
    - `int previousIndex()`: Retourne l'index de l'élément qui serait retourné par `previous()`.
    - `void add(E e)`: Insère un élément dans la liste.
    - `void set(E e)`: Remplace le dernier élément retourné par `next()` ou `previous()`.
  - **Exemple d'utilisation avec `ArrayList`**:

```
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

public class Main {
   public static void main(String[] args) {
```

```
        List<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
        list.add("C");

        ListIterator<String> listIterator = list.listIterator();
        while (listIterator.hasNext()) {
           String element = listIterator.next();
           System.out.println(element);
            if (element.equals("B")) {
                 listIterator.add("D");
           }
       }
        System.out.println(list);

      // Itération en arrière
        while (listIterator.hasPrevious()) {
            String element = listIterator.previous();
            System.out.println(element);
        }
    }
}
```

Points importants :

- **Choix de l'algorithme de tri :** Le choix de l'algorithme de tri dépend du contexte. Pour les petits ensembles de données, les tris simples comme le tri par insertion ou sélection peuvent suffire. Pour les grands ensembles de données, les tris plus avancés comme le tri fusion ou le tri rapide sont plus appropriés.
- **Itérateurs :** Ils sont essentiels pour parcourir les collections de manière sécurisée et flexible en Java. Ils permettent de travailler avec des collections sans se soucier de leur implémentation sous-jacente.