Tableaux dynamiques

1 Définitions

- **Tableau** : structure qui stocke des éléments de même type dans une zone contiguë. L'accès par indice est en temps $O(1)$. La taille d'une instance est le nombre de cases qu'elle contient.
- **Tableau statique** : taille **non modifiable** après création.
  Exemples Java : `int[] a = new int[4];` `int[] b = {1,2,3};`
- **Tableau dynamique** : taille **modifiable** pendant l'exécution (ajouts et suppressions gérés par la structure).
  Exemple en Java : `ArrayList<T>` (voir tutoriel W3Schools).

2 Tableaux en Java : points importants

Opérateur `new`

- `new` permet de créer un tableau (en fait, un nouvel objet).
- Le tableau est initialisé avec des valeurs par défaut : entiers → `0`, booléens → `false`, références → `null`.
- La taille est **figée après création**.
- La libération est automatique (ramasse-miettes).

Validité des indices

- Les indices vont de 0 à `length-1`.
- Accéder à un indice invalide provoque une exception `ArrayIndexOutOfBoundsException`.

Affectation entre tableaux

```java
int[] t1 = {1, 2, 3, 4, 5};
int[] t2 = t1;   // t2 reçoit la même référence que t1
t2[2] = 42;

System.out.println(Arrays.toString(t1));
// [1, 2, 42, 4, 5]
System.out.println(Arrays.toString(t2));
// [1, 2, 42, 4, 5]
```

Conclusion : `t1` et `t2` pointent vers *le même objet tableau*. Modifier l'un entraîne la modification de l'autre.

Passage en paramètre

```java
public class Demo {
    public static void modifier(int[] t) {
        t[0] = 8888;   // on modifie l'élément 0 du même tableau
    }
    public static void main(String[] args) {
        int[] tab = {1, 2, 3};
        modifier(tab);
        System.out.println(tab[0]); // 8888
    }
}
```

Quand on passe un tableau à une fonction, on transmet sa *référence*. Si la fonction modifie le tableau, le changement est visible depuis le programme appelant.

3 Implémentation d'un tableau dynamique

Le langage Java possède la structure `ArrayList<T>` qui permet de manipuler des tableaux dynamiques de manière transparente, sans se soucier de leur redimensionnement si besoin. Ce n'est pas le cas dans tous les langages de programmation.

Dans ce cours, nous allons donc définir notre propre structure de données, appelée `MyArrayList`, qui permet de manipuler des tableaux dynamiques. Cela nous permettra de comprendre le fonctionnement interne d'un tableau dynamique et d'analyser le coût des différentes opérations (*accès, ajout, insertion, recherche, suppression*). Ici, nous utiliserons un tableau d'entiers comme exemple.

Pour représenter un tableau dynamique, il faut :

- un tableau sous-jacent qui contient les éléments ;
- un compteur indiquant le nombre effectif d'éléments stockés.

Nous allons donc définir une classe `MyArrayList` qui contient deux attributs principaux, et dont la capacité initiale, qui fixe la taille du tableau sous-jacent, est fournie par l'utilisateur lors de la création.

```java
public class MyArrayList {
    private int taille;   // nombre d'éléments présents
    private int[] tab;    // tableau interne

    public MyArrayList(int capacite) {
        this.taille = 0;
        this.tab = new int[capacite];
    }

```

```
10      public MyArrayList() {
11          this(100); // capacité par défaut
12      }
13
14      public int getTaille() {
15          return this.taille;
16      }
17  }
```

Remarque : en C, une telle structure est encore plus utile car la bibliothèque standard ne fournit pas d'équivalent à `ArrayList`.

Accès à un élément

L'accès à un élément dans un tableau se fait directement par l'indice. Cette opération est réalisée en temps constant $O(1)$.

Dans notre implémentation, on ajoute la méthode `get` à la classe `MyArrayList` :

```
1  public class MyArrayList {
2
3      public int get(int i) {
4          if (i < 0 || i >= this.taille) {
5              throw new IndexOutOfBoundsException("Indice invalide : " +
                   i);
6          }
7          return this.tab[i];
8      }
9  }
```

Remarque : l'utilisation de `this` rend explicite que `taille` et `tab` sont des attributs de l'objet courant.

Ajout d'un élément à la fin du tableau

Deux cas peuvent se présenter :

- **Le tableau n'est pas plein** : on place l'élément à l'indice `taille`, puis on incrémente `taille` de 1. Cette opération est immédiate et se fait en temps constant $O(1)$.
- **Le tableau est plein** : on crée un nouveau tableau plus grand (par exemple deux fois plus grand), on copie tous les éléments de l'ancien tableau dans le nouveau, puis on ajoute l'élément. Cette opération coûte un temps proportionnel à la taille du tableau, soit $O(\texttt{taille})$.

Remarque : bien que certains ajouts soient coûteux (ceux qui déclenchent une copie complète), la stratégie d'agrandissement par doublement fait que ces copies restent rares. Si l'on ajoute n éléments au total :

- la plupart des ajouts coûtent $O(1)$;
- seuls quelques-uns coûtent plus cher, mais leur coût est « réparti » sur tous les ajouts précédents.

Ainsi, le *coût moyen par ajout* est constant. On dit que l'ajout en fin de tableau dynamique est en **temps constant amorti** $O(1)$.

```java
public class MyArrayList {

    public void ajouteFin(int x) {
        if (this.taille < this.tab.length) {
            this.tab[this.taille] = x;
            this.taille++;
        } else {
            // doublement de la capacité
            int[] nouveauTab = new int[2 * this.tab.length];
            for (int i = 0; i < this.taille; i++) {
                nouveauTab[i] = this.tab[i];
            }
            this.tab = nouveauTab;
            this.tab[this.taille] = x;
            this.taille++;
        }
    }
}
```

Ajout d'un élément à un indice donné

La manière la plus simple consiste à :

- ajouter l'élément à la fin du tableau (coût constant grâce à `ajouteFin`) ;
- puis faire remonter cet élément jusqu'à la position souhaitée en décalant les cases une à une.

Le coût de cette opération dépend du nombre de décalages. Dans le pire des cas (insertion à l'indice 0), il faut décaler tous les éléments existants. L'insertion à un indice donné est donc une opération en **temps linéaire** $O(n)$.

```java
public class MyArrayList {

    public void ajouteIndice(int i, int x) {
        // On ajoute d'abord à la fin (gère aussi le redimensionnement
            si besoin)
        this.ajouteFin(x);

```

```
        // Décalage des éléments vers la droite jusqu'à la position i
        for (int j = this.taille - 1; j > i; j--) {
            int tmp = this.tab[j];
            this.tab[j] = this.tab[j - 1];
            this.tab[j - 1] = tmp;
        }
    }
}
```

Suppression d'un élément à la fin du tableau

La suppression du dernier élément est immédiate : il suffit de décrémenter la variable `taille`. Cette opération ne nécessite aucun décalage et se fait donc en temps constant $O(1)$.

```
public class MyArrayList {

    public void supprimeFin() {
        if (this.taille == 0) {
            throw new IllegalStateException("Tableau vide");
        }
        this.taille--;
    }
}
```

Remarque : l'élément supprimé reste présent dans le tableau sous-jacent, mais il n'est plus considéré comme faisant partie du tableau logique puisque `taille` a été décrémentée.

Suppression d'un élément à un indice donné

Pour supprimer un élément à un indice `i`, il faut décaler tous les éléments suivants d'une position vers la gauche. Le coût de cette opération dépend du nombre de décalages. Dans le pire des cas (suppression à l'indice 0), il faut décaler tous les éléments restants. Cette opération est donc en **temps linéaire** $O(n)$. Enfin, il faut décrémenter `taille` de 1.

```
public class MyArrayList {
    public void supprimeIndice(int i) {
        if (i < 0 || i >= this.taille) {
            throw new IndexOutOfBoundsException("Indice invalide : " +
                i);
        }
        // Décalage vers la gauche
        for (int j = i; j < this.taille - 1; j++) {
            this.tab[j] = this.tab[j + 1];
        }
```

```
        // Décrémenter taille
        this.taille--;
    }
}
```

Recherche d'un élément

Pour rechercher un élément dans un tableau non trié, on parcourt séquentiellement les cases jusqu'à trouver la valeur recherchée. Le coût dépend du nombre de comparaisons effectuées :

- meilleur cas : l'élément est au début ($O(1)$) ;
- pire cas : l'élément est absent ou en dernière position ($O(n)$).

```
public class MyArrayList {

    public boolean recherche(int x) {
        for (int i = 0; i < this.taille; i++) {
            if (this.tab[i] == x) {
                return true;
            }
        }
        return false;
    }
}
```

Remarques :

- rechercher le minimum ou le maximum d'un tableau non trié nécessite également $O(n)$ comparaisons ;
- si le tableau est trié, une recherche dichotomique est possible en $O(\log n)$, mais il faut tenir compte du coût pour trier le tableau (ou pour le maintenir trié).

Récapitulatif de notre classe

Voici une implémentation simplifiée de la classe `MyArrayList`, limitée aux entiers, qui rassemble toutes les opérations étudiées.

```
public class MyArrayList {
    private int taille;     // nombre d'éléments présents
    private int[] tab;      // tableau interne

    // Constructeur avec capacité initiale
    public MyArrayList(int capacite) {
        this.taille = 0;
        this.tab = new int[capacite];
```

```
9       }
10
11      // Constructeur par défaut (capacité 100)
12      public MyArrayList() {
13          this(100);
14      }
15
16      // Retourne le nombre d'éléments présents
17      public int getTaille() {
18          return this.taille;
19      }
20
21      // Accès à l'élément d'indice i
22      public int get(int i) {
23          if (i < 0 || i >= this.taille) {
24              throw new IndexOutOfBoundsException("Indice invalide : " +
                    i);
25          }
26          return this.tab[i];
27      }
28
29      // Ajout à la fin
30      public void ajouteFin(int x) {
31          if (this.taille < this.tab.length) {
32              this.tab[this.taille] = x;
33              this.taille++;
34          } else {
35              // Redimensionnement (double la capacité)
36              int[] nouveauTab = new int[2 * this.tab.length];
37              for (int i = 0; i < this.taille; i++) {
38                  nouveauTab[i] = this.tab[i];
39              }
40              this.tab = nouveauTab;
41              this.tab[this.taille] = x;
42              this.taille++;
43          }
44      }
45
46      // Ajout à un indice donné
47      public void ajouteIndice(int i, int x) {
48          if (i < 0 || i > this.taille) {
49              throw new IndexOutOfBoundsException("Indice invalide : " +
                    i);
```

```
50          }
51          this.ajouteFin(x); // place à la fin (redimensionne si besoin)
52          for (int j = this.taille - 1; j > i; j--) {
53              int tmp = this.tab[j];
54              this.tab[j] = this.tab[j - 1];
55              this.tab[j - 1] = tmp;
56          }
57      }
58
59      // Suppression du dernier élément
60      public void supprimeFin() {
61          if (this.taille == 0) {
62              throw new IllegalStateException("Tableau vide");
63          }
64          this.taille--;
65      }
66
67      // Suppression à un indice donné
68      public void supprimeIndice(int i) {
69          if (i < 0 || i >= this.taille) {
70              throw new IndexOutOfBoundsException("Indice invalide : " +
                    i);
71          }
72          for (int j = i; j < this.taille - 1; j++) {
73              this.tab[j] = this.tab[j + 1];
74          }
75          this.taille--;
76      }
77
78      // Recherche d'une valeur
79      public boolean recherche(int x) {
80          for (int i = 0; i < this.taille; i++) {
81              if (this.tab[i] == x) {
82                  return true;
83              }
84          }
85          return false;
86      }
87  }
```

Conclusion : Tableaux dynamiques

Les tableaux dynamiques présentent plusieurs avantages et limites :

- **Efficacité mémoire** : les éléments sont stockés dans une zone contiguë, ce qui facilite leur gestion.
- **Accès** : l'accès direct par indice est très rapide, en temps constant $O(1)$.
- **Ajout / suppression en fin** : ces opérations sont efficaces ; l'ajout est en temps constant amorti $O(1)$, la suppression en fin est en $O(1)$.
- **Ajout / suppression à un indice donné** : elles nécessitent des décalages, donc coûtent en temps linéaire $O(n)$.
- **Recherche** : dans un tableau non trié, la recherche séquentielle est en $O(n)$. Elle peut être améliorée à $O(\log n)$ si le tableau est trié et qu'on utilise une recherche dichotomique, mais cela suppose un coût supplémentaire pour maintenir l'ordre.