Organisation et Exploitation des Données

Code cours : OEDB1

1. Introduction

Le principe de base d'une structure de données est de stocker en mémoire des données. On pourra effectuer des opérations sur cette structure (Mise à jour, Suppression, Insertion, ...).

1.1 Complexité temporelle et spatiale

Les opérations sont caractérisées par leur complexité :

- Complexité Temporelle : Scalabilité de l'algorithme en fonction de la taille du problème
  - Best Case : meilleur cas pour l'algorithme
  - Worst Case : pire cas pour l'algorithme
- Complexité en espace : Utilisation de l'espace mémoire

1.2 Les types de variables

Les variables sont caractérisées par leur *type*, il existe des types supportées nativement par la plus part des langages de programmation. Ces types sont appelés *types primitifs*. On y trouves

- **Entiers** : Taille variable. Il stocke le résultat exact des opérations
- **Réel** : La précision est variable en fonction du nombre de décimales stockées (erreurs d'arrondi)
- **Booléen** : Vrai ou Faux
- **Caractère** : Stockage suivant le code ASCII ou EBCDIC
- **Pointeurs** : Adresse en mémoire désignant un objet

1.3 Les structures de données

Une **structure de donnée** (ou *collection*) est une structure logique destinée à contenir des données organisées de manière à simplifier le traitement.

Un des avantages à l'utilisation de structures de données est de permettre un traitement **plus rapide** et **plus efficace** . Mais surtout de **diminuer la complexité d'une application informatique ainsi que le taux d'erreurs.**

2. Tableaux

Un tableau est une structure de donnée les plus anciennes que l'on retrouve dans les premiers langages de programmation évolués.

L'espace mémoire des tableaux est gérée de manière statique et la longueur d'un tableau est fixe.

Chaque cellule d'un tableau peut contenir soit un type primitif, soit une autre structure (ex: un autre tableau).

Les tableaux sont :

- ***Homogène*** : tous les éléments sont du même type
- ***A accès direct*** via l'indice, et grâce au fait que les éléments d'un tableau soient stockées de manière contiguë dans la mémoire.

2.1 Types de tableaux

2.1.1 Tableaux à une dimension

Ces tableaux sont ordonnées de manière consécutive.

Insertion ($O(n)$)

Pour ajouter de nouvelles données, il suffit d'ajouter celle-ci à la fin du tableau.

Trouver un élément ($O(n)$)

Pour trouver un élément dans le tableau il faut parcourir chaque élément qui le compose.

Suppression ($O(1)$/$O(n)$)

La suppression d'un élément peut être rapide si l'ordre n'a pas d'importance. Il suffit de déplacer le dernier élément contenu dans le tableau pour combler le trou laissé.

Dans le cas où l'ordre aurait de l'importance (ex: arrivée de patients dans la salle d'attente), il faudra décaler tout le tableau.

2.1.2 Tableaux à une dimension trié

Le tableau est ordonné suivant une relation d'ordre

Insertion ($O(?)$)

Pour ajouter de nouvelles données, il faudra décaler tout le tableau d'une unité pour y mettre le nouvel élément.

Trouver un élément ($O(?)$)

Trouver un élément dans le tableau est bien plus rapide car on peut utiliser cette relation pour rechercher plus rapidement. (ex : tri dichotomique)

Suppression ($O(n)$)

La suppression nécessite de déplacer tout le tableau à partir du point ou il à été supprimé

2.1.3 Tableaux à 2 dimensions

Les tableaux multidimensionnels sont des tableaux contenant des tableaux. On peut donc accéder aux données par le biais d'un double index (premier et second tableau).

Le premier indice désigne généralement la ligne et le second la colonne.

2.2 Algorithmes de recherche

2.2.1 Tableaux non-trié

Complexité temporelle : $O(n)$

si recherche fructueuse : (nb+1)/2

```
 o————————————————————————————————o ↓ valeurs, nb
 | rechercher dans tableau non trié |
 o————————————————————————————————o
┌─── *
| obtenir cléLue
| o————————————————o ↓ valeurs, nb, cléLue
| | indRecherché |
| o————————————————o ↓ iValeur
|┌── if( iValeur == nb )
|| sortir "clé inexistante"
|├── else
|| sortir "clé trouvée dans la cellule d'indice ", iValeur
|└──
└──────────


o————————————————o ↓ valeurs, nb, cléLue
| indRecherché |
o————————————————o ↓ iValeur
┌─── *
| iValeur = 0
|╔══ do while (iValeur < nb and valeurs[iValeur].clé ≠ cléLue)
|‖ iValeur ++
|╚══
└──────────
```

2.2.2 Recherche séquentielle dans un tableau trié

Complexité temporelle ($O(n)$)

```
o————————————————————————————————o ↓ valeurs, nb
| rechercher dans tableau trié |
o————————————————————————————————o
┌─── *
| obtenir cléLue
| o————————————————o ↓ valeurs, nb, cléLue
| | indRecherché |
| o————————————————o ↓ iValeur
|┌── if( iValeur == nb or cléLue ≠ valeurs[iValeur].clé )
|| sortir "clé inexistante"
|├── else
|| sortir "clé trouvée dans la cellule d'indice ", iValeur
|└──
└──────────
o————————————————o ↓ valeurs, nb, cléLue
| indRecherché |
o————————————————o ↓ iValeur
┌─── *
| iValeur = 0
|╔══ do while (iValeur < nb and valeurs[iValeur].clé < cléLue)
|‖ iValeur ++
|╚══
└──────────
```

recherche fructueuse/infructueuse : $(nb + 1)/2$

2.2.3 Recherche dichotomique dans un tableau trié

Complexité temporelle : $O(log(n))$

```
o──────────────────────────o ↓ valeurs, nb
| recherche dichotomique |
o──────────────────────────o
┌─── *
| obtenir cléLue
| bInf = 0
| bSup = nb - 1
|
| iMilieu= [(bInf + bSup) /2]ENT
|
|╔══ do while (bInf ≤ bSup and valeurs[iMilieu].clé ≠ cléLue)
|||
|||┌── if ( cléLue < valeurs[iMilieu].clé ))
|||| bSup = iMilieu - 1
|||├── else
|||| bInf = iMilieu + 1
|||└──
||| iMilieu= [(bInf + bSup) /2]ENT
|||
|╚══
|┌── if( bInf > bSup )
|| sortir "clé inexistante"
|├── else
|| sortir "clé trouvée dans la cellule d'indice ", iMilieu
|└──
└──────────
```

2.3 Bloc Logique

Lorsque dans un tableau trié suivant un champ déterminé, les cellules successives contiennent la même valeur de ce champ, on à un bloc logique.

La gestion des blocs logiques s'effectue par le biais de zones. Il y a généralement 3 zones. (l'initialisation, le traitement et la clôture). Dans le cas de blocs logiques, une boucle principale à pour objectif de parcourir tous les éléments du tableau. Une autre boucle imbriquée dans la première aura la même condition que la première mais ajoutera une dimension de catégorie principale à cette dernière(on parcours tous les élément d'une catégorie). Si on prends le cas de (catégorie, sous-catégorie et Soussous Catégorie) on 3 couches et le nombre de boucles nécessaires à cette opération sera 3+1 = 4 (les 3 catégories et la boucle qui parcours tous les éléments). Chaque boucle imbriquée à pour objectif de vérifier que la catégorie en cours est bien celle de élément analysé.

Un bloc logique est un regroupement de cellules dans un tableau partageant les mêmes caractéristiques (ex: regroupement de lieux par ville, puis par quartiers).

```
o──────────────────────────────────o ↓ hôtels, nbHôtels
| Afficher statistiques Hôtels |
```

```
o──────────────────────────────o
┌─── *
| // Initialisation générale
| nbMinHôtelsCatégorie = HV
| iHôtel = 0
|
|╔══ do while ( iHôtel < nbHôtels )
|‖
|‖ // Initialisation catégorie
|‖ catégorieEnCours = hôtels[iHôtel].catégorie
|‖ sortir catégorieEnCours
|‖ nbHôtelsCatégorie = 0
|‖ nbMaxHotelsVille = 0
|‖
|‖╔══ do while ( iHôtel < nbHôtels and
|‖‖ catégorieEnCours == hôtels [iHôtel].catégorie )
|‖‖
|‖‖ // Initialisation ville
|‖‖ villeEnCours = hôtels [iHôtel].ville
|‖‖ sortir villeEnCours
|‖‖ nbHôtelsVille = 0
|‖‖
|‖‖╔══ do while ( iHôtel < nbHôtels and
|‖‖‖ catégorieEnCours == hôtels [iHôtel].catégorie and
|‖‖‖ villeEnCours == hôtels [iHôtel].ville )
|‖‖‖
|‖‖‖ // Traitement hôtel
|‖‖‖ nbHôtelsVille ++
|‖‖‖ iHôtel ++
|‖‖╚══
|‖‖ // Clôture ville
|‖‖ sortir nbHôtelsVille
|‖‖ nbHôtelsCatégorie += nbHôtelsVille
|‖‖┌── if (nbHôtelsVille > nbMaxHotelsVille )
|‖‖| nbMaxHotelsVille = nbHôtelsVille
|‖‖| nomMaxVille = villeEnCours
|‖‖└──
|‖╚══
|‖ // Clôture catégorie
|‖┌── if (nbHôtelsCatégorie < nbMinHôtelsCatégorie )
|‖| nbMinHôtelsCatégorie = nbHôtelsCatégorie
|‖| categMin = catégorieEnCours
|‖└──
|‖ sortir nomMaxVille
|╚══
| // clôture générale
| sortir categMin
└──────────
```

3. Listes chainées

Une **liste** est une structure de donnée homogène constituée d'éléments ordonnées linéairement et chainées entre eux.

On accède à la liste en désignant le premier chainon par le biais d'un pointeur (généralement appelé *pDébut*)

Un **chainon** contient **de l'information** et **un pointeur** vers élément suivant dans la liste (ou NULL si il n'y en a pas).

La gestion de la **mémoire** de la liste chainée est gérée de manière **dynamique** et **sans déplacement des chainons**.

La *liste chainée* peut aussi être **circulaire** alors le dernier élément pointe vers le premier.

Il existe aussi des **listes doublement chainées** qui ont un pointeur vers l'élément suivant et l'élément précédent.

3.1 Algorithmes

3.1.1 Recherche dans une liste simple non triée

```
o──────────────────────────────────o ↓ pDébut
| rechercher dans liste non triée |
o──────────────────────────────────o
┌─── *
| obtenir donnéeLue
|
| o─────────────o ↓ pDébut, donnéeLue
| | pRecherché |
| o─────────────o ↓ p
|
|┌── if (p == NULL)
|| sortir " Donnée absente de la liste "
|├── else
|| sortir " Donnée trouvée : ", p → donnée
|└──
└──────────
o─────────────o ↓ pDébut, donnéeRecherchée
| pRecherché |
o─────────────o ↓ p
┌─── *
| p = pDébut
|╔══ do while (p ≠ NULL and p → donnée ≠ donnéeRecherchée)
||  p = p → pSuiv
|╚══
└──────────
```

3.1.2 Recherche dans une liste simple triée

```
o──────────────────────────────────o ↓ pDébut
| rechercher dans liste triée |
o──────────────────────────────────o
┌─── *
| obtenir donnéeLue
|
| o─────────────o ↓ pDébut, donnéeLue
| | pRecherché |
| o─────────────o ↓ p
```

```
|
|┌── if (p == NULL or donnéeLue < p → donnée)
|| sortir " Donnée absente de la liste "
|├── else
|| sortir " Donnée trouvée : ", p → donnée
|└──
└──────────

o──────────o ↓ pDébut, donnéeRecherchée
| pRecherché |
o──────────o ↓ p
┌─── *
| p = pDébut
|╔══ do while (p ≠ NULL and p → donnée < donnéeRecherchée)
|╟ p = p → pSuiv
|╙──
└──────────
```

3.1.3 Insertion d'un nouveau chainon dans une liste simple

insertion en début de liste

```
o──────────────────────o ↓ pDébut, donnéeLue
| insertion début de liste |
o──────────────────────o ↓ pDébut, message
┌─── *
| pNouv = adresse mémoire nouveau chainon
|┌── if ( pNouv == NULL )
|| message = " Mémoire insuffisante "
|├── else
|| pNouv → donnée = donnéeLue
|| pNouv→ pSuiv = pDébut
|| pDébut = pNouv
|| message = " Ajout effectué "
|└──
└──────────
```

Insertion en fin de liste

```
o──────────────────────o ↓ pDébut, donnéeLue,
| insertion fin de liste |
o──────────────────────o ↓ pDébut, message
┌─── *
| pNouv = adresse mémoire nouveau chainon
|┌── if ( pNouv == NULL )
|| message = " Mémoire insuffisante "
|├── else
|| p = pDébut
||╔══ do while (p ≠ NULL)
|||╟ pPrécédent = p
|||╟ p = p → pSuiv
||╙──
|| pNouv → donnée = donnéeLue
```

```
||
||┌── if (pDébut == NULL)
||| pDébut = pNouv
||├── else
||| pPrécédent → pSuiv = pNouv
||└──
|| pNouv→ pSuiv = NULL
|└──
| message = " Ajout effectué "
└──────────
```

Insertion dans une liste triée

```
o──────────────────────────────────o ↓ pDébut, donnéeLue
| insertion dans une liste triée |
o──────────────────────────────────o ↓ pDébut, message
┌─── *
| pNouv = adresse mémoire nouveau chainon
|┌── if (pNouv == NULL)
|| message = "mémoire insuffisante"
|├── else
|| p = pDébut
||╔══ do while (p ≠ NULL and donnéeLue > p → donnée)
|||‖ pPrécédent = p
|||‖ p = p → pSuiv
||╚══
|| pNouv → donnée = donnéeLue
||
||┌── if (p == pDébut) // ajout début de liste ou liste vide
||| pNouv→ pSuiv = pDébut
||| pDébut = pNouv
|||
||├── else
|||┌── if (p == NULL) // ajout en fin de liste
|||| pPrécédent → pSuiv = pNouv
|||| pNouv → pSuiv = NULL
||||
|||├── else // ajout milieu de liste
|||| pPrécédent → pSuiv = pNouv
|||| pNouv → pSuiv = p
|||└──
||└──
|| message = « Ajout effectué »
|└──
└──────────
```

3.1.4 Suppression d'un chainon dans une liste simple

```
o──────────────o ↓ donnéeRecherchée, pDébut
| suppression |
o──────────────o ↓ pDébut, message
┌─── *
|
| p = pDébut
```

```
|╔══ do while (p ≠ NULL and donnéeRecherchée> p → donnée)
|‖ pPrécédent = p
|‖ p = p → pSuiv
|╚══
|
|
|┌── if (p == NULL or donnéeRecherchée < p → donnée)
||
|| message = " Erreur: donnée inexistante "
||
|├── else
||
||┌── if (p == pDébut) // suppression du premier chainon
||| pDébut = pDébut → pSuiv
||├── else
||| pPrécédent → pSuiv = p → pSuiv
||└──
|| message = " Suppression effectuée "
|└──
| libérer la mémoire pointée par p
└──────────
```

3.2 Comparaison entre un tableau et une liste

	Avantages	Inconvénients
Listes	**grand nombre d'opérations** (ajout, supp.) et **allocation dynamique** de mémoire	un accès nécessite de **parcourir tous les chainons** et **plus espace** utilisé (pointeurs)
Tableaux	**accès direct** avec l'indice et **allocation mémoire plus faible**	(ajout, supp.) **nécessite décalage** et **allocation statique**

4. Les Piles

Les **piles** et les **files** sont des structures de données dont les éléments sont ordonnés linéairement mais qui ne permettent l'accès qu'à un seul élément à la fois.

Les **piles** se basent sur le principe du LIFO (Last In First Out) où le dernier élément à être rentré sera le premier à en sortir. Cette structure est beaucoup utilisé en informatique pour gérer les opérations à effectuer par un processeur.

Le seul élément de la pile est donc celui qui est situé au sommet de la pile.

En général, une pile possède les opérations suivantes :

- initialiser la pile
- vérifier si la pile est vide
- obtenir la valeur du haut de la pile
- pop : permets de retirer le premier élément situé au sommet de la pile
- push : permets d'ajouter un élément au sommet de celle-ci

4.1.1 Représentation sous la forme d'un tableau

La pile contient alors un nombre MAX d'éléments qui correspond à la taille du tableau. On stocke alors le sommet en stockant l'indice ou se situe l'élément courant.

4.1.2 Représentation sous la forme d'une liste chainée simple

La taille de la pile est alors de taille dynamique et variable. On stocke le sommet de la pile via un pointeur vers le début de la liste chainée.

4.2 Algorithmes utilisant un tableau

4.2.1 Initialisation

```
o————————————————o
| initialisation |
o————————————————o ↓ sommet
┌─── *
| sommet = 0
└──────────
```

4.2.2 Empiler (Push)

```
o—————————o ↓ pile, sommet, donnéeNouvelle
| empiler |
o—————————o ↓ pile, sommet, message
┌─── *
| message = " "
|┌── if (sommet < MAX)
|| pile[sommet] = donnéeNouvelle
|| sommet ++
|├── else
|| message = "la pile est pleine"
|└──
└──────────
```

4.2.3 Dépiler (Pop)

```
o——————————o ↓ pile, sommet
| dépiler |
o——————————o ↓ pile, sommet, donnée, message
┌─── *
| message = " "
|┌── if (sommet > 0)
|| sommet --
|| donnée= pile[sommet]
|├── else
|| message = "erreur: la pile est vide"
|| donnée= " "
|└──
└──────────
```

4.3 Algorithmes utilisant une liste chaînée simple

4.3.1 Initialisation

```
o——————————————o
| initialisation |
o——————————————o ↓ sommet
┌─── *
| sommet = null
└──────────
```

4.3.2 Empiler (Push)

```
o——————————o ↓ sommet, donnéeNouvelle
| empiler |
o——————————o ↓ sommet, message
┌─── *
| message = " "
| pNouveau = adresse mémoire nouveau chaînon
|┌── if (pNouveau ≠ null)
|| pNouveau → donnée = donnéeNouvelle
|| pNouveau → pSuiv = sommet
|| sommet = pNouveau
|├── else
|| message = "plus de place mémoire"
|└──
└──────────
```

4.3.3 Dépiler (Pop)

```
o——————————o ↓ sommet
| dépiler |
o——————————o ↓ sommet, donnée, message
┌─── *
| message = " "
```

```
|┌── if(sommet == null)
|| message = "erreur, la pile est vide"
|| donnée = " "
|├── else
|| donnée = sommet → donnée
|| pSauvé = sommet
|| sommet = sommet → pSuiv
|| libérer la place pointée par pSauvé
|└──
└──────────
```

5. Les files

Les **piles** et les **files** sont des structures de données dont les éléments sont ordonnés linéairement mais qui ne permettent l'accès qu'à un seul élément à la fois.

Les **files** se basent sur le principe du FIFO (First In First Out) on peut le comparer à une fille à la caisse d'un super marché, le premier entré dans la file sera le premier à en sortir.

Une file possède une **tête** et une **queue**.

On peut opérer diverses opérations sur les files :

- initialiser la file
- Vérifier si la file est vide
- accéder à sa tête et obtenir la valeur de celle-ci
- défiler : supprimer l'élément de la tête
- enfiler : ajouter un élément à la fin de la file

5.1 Représentation d'une file

5.1.1 Représentation sous la forme d'un tableau à simple indice

Dans le cas de la représentation de la file sous la forme d'un tableau. Le nombre maximum d'éléments que l'on peut stocker est équivalent à la taille du tableau.

Il existe 2 méthodes pour gérer le décalage des éléments au cours du temps :

Gestion circulaire

On débute à l'indice 0 et l'on avance. Lorsque l'on arrive en bout du tableau, on recommence à enfiler à partir de 0.

Cette méthode permets d'éviter le décalage de la deuxième méthode

Gestion par décalage

Cette méthode consiste à toujours prendre l'élément à l'indice 0 de la file et ensuite de décaler tous les autres pour combler le vide.

5.1.2 Liste Chainée Simple

Le pointeur de tête est l'adresse du premier chainon et le pointeur Queue est le pointeur de l'adresse du dernier chainon

5.2 Algorithmes utilisant un tableau

5.2.1 Initialisation

```
o————————————————o
| initialisation |
o————————————————o ↓ (tête), queue
┌─── *
| (tête = 0)
| queue = 0
└──────────
```

5.2.2 Enfiler

```
o—————————o ↓ file, queue, donnéeNouvelle
| enfiler |
o—————————o ↓ file, queue, message
┌─── *
| message = " "
|┌── if (queue < MAX)
|| file[queue] = donnéeNouvelle
|| queue ++
|├── else
|| message = "la file est pleine"
|└──
└──────────
```

5.2.3 Défiler

```
o—————————o ↓ file, (tête), queue
| défiler |
o—————————o ↓ file, queue, donnée, message
┌─── *
| message = " "
|┌── if (tête == queue) // ou if( queue == 0)
|| message = "erreur, la file est vide"
|| donnée = " "
|├── else
|| donnée = file[tête] // ou donnée = file[0]
|| o——————————o ↓ file, queue
|| | décalage |
|| o——————————o ↓ file, queue
|└──
└──────────

o——————————o ↓ file, queue
| décalage |
o——————————o ↓ file, queue
┌─── *
| ind = 0
|╔══ do while (ind < queue - 1)
|‖ file [ind] = file [ind + 1]
```

```
|‖ ind ++
|╚══
| queue --
└──────────
```

5.3 Algorithmes utilisant une liste chainée simple

5.3.1 Initialisation

```
o────────────────o
| initialisation |
o────────────────o ↓ tête, queue
┌─── *
| tête = queue = null
└──────────
```

5.3.2 Enfiler

```
o─────────o ↓ tête, queue, donnéeNouvelle
| enfiler |
o─────────o ↓ tête, queue, message
┌─── *
| message = " "
| pNouveau = adresse mémoire nouveau chaînon
|┌── if (pNouveau ≠ null)
|| pNouveau → donnée = donnéeNouvelle
|| pNouveau → pSuiv = null
||
||┌── if (tête == null) // ajout dans une file vide
||| tête = queue = pNouveau
||├── else
||| queue → pSuiv = pNouveau
||| queue = pNouveau
||└──
|├── else
|| message = "plus de place mémoire"
|└──
└──────────
```

5.3.3 Defiler

```
o─────────o ↓ tête, queue
| défiler |
o─────────o ↓ tête, queue, donnée, message
┌─── *
| message = " "
|┌── if (tête == null)
|| message = "erreur, la file est vide"
|| donnée = " "
|├── else
|| donnée = tête → donnée
```

```
|| pSauvé = tête
|| tête = tête→ pSuiv
|| libérer la place pointée par pSauvé
||┌── if (tête == null) // la file ne contenait qu'un seul chaînon
||| queue = null
||└──
|└──
└──────────
```

6. Les Arbres

Un arbre est une structure de donnée non-linéaire dans laquelle les informations sont retenues dans ce que l'on appelle un nœud.

Tout nœud de l'arbre est la racine d'un sous-arbre constitué par sa descendance et lui-même. Un arbre est donc une structure récursive.

> terminologie :
>
> - **racine** : sommet de l'arbre. c'est un nœud qui ne possède pas de *père*
> - **nœud intérieur** : nœud qui possède un *père* (sauf racine) et au moins un *fils*
> - **Feuille** : nœud qui termine l'arborescence. il s'agit d'un nœud sans fils

Un arbre est qualifié de connexe car la séquence des pères partant d'un nœud vers la racine est toujours unique (on à toujours qu'un père).

- **Chemin** : Suite (Unique) à d'arcs à parcourir entre 2 nœuds
- **Longueur d'un chemin** : nombre d'arcs d'un chemin
- **Niveau** : longueur du chemin depuis la racine jusqu'à ce nœud
- **Hauteur** : longueur du plus long chemin depuis ce nœud jusqu'à une feuille
- **Hauteur d'un arbre** : hauteur de sa racine
- **Profondeur** : combien d'arcs y a t-il pour remonter à la racine
- **Degré extérieur d'un nœud** : nombre de sous-Arbres d'un nœud
- **Degré d'un Arbre** : degré extérieur maximum des nœuds de l'arbre
- **Un arbre ordonné** : si il existe un ordre au sein de ses sous-arbres

6.1 Arbre Binaire

Un **Arbre Ordonné** de **degré extérieur 2**

[pGauche, data,pDroit]

6.1.1 Les Parcours

Parcours Pre-Order

Racine, Gauche, Droit

Parcours Suffixe, Post-Fixe, Post-Order

Gauche, Droit, Racine

Parcours Infixe ou in-Order

Gauche, Racine, Droit

6.2 Arbre Binaire de Recherche (ABR)

Arbre binaire qui possède une clé unique et qui possède une forme d'ordre Arbre Gauche < clé < Arbre Droit.

pGauche|clé|data|pDroit

6.3 Algorithmes

6.3.1 Recherche dans un arbre binaire de recherche

```
o──────────────────────o ↓ racine
| rechercher dans arbre |
o──────────────────────o
┌─── *
| obtenir cléLue
| o──────────────────o ↓ racine, cléLue
| | pNoeudRecherché |
| o──────────────────o ↓ pNoeud, père
|┌── if( pNoeud == null)
|| sortir "clé absente de l'arbre"
|├── else
|| sortir "clé trouvée:", pNoeud → donnée
|└──
└──────────
```

```
o──────────────────o ↓ racine, cléLue
| pNoeudRecherché |
o──────────────────o ↓ pNoeud, père
┌─── *
| père = null
| pNoeud = racine
|╔══ do while (pNoeud ≠ null and cléLue ≠ pNoeud → clé)
|‖ père = pNoeud
|‖┌── if(cléLue < pNoeud → clé)
|‖| pNoeud = pNoeud → pGauche
|‖├── else
|‖| pNoeud = pNoeud → pDroit
|‖└──
|╚══
└──────────
```

> Remarquons que le module pNoeudRecherché renvoie la variable père. Cette variable n'est pas nécessaire dans le cas d'une recherche mais sera indispensable lorsque cette recherche sera suivie d'un ajout ou d'une suppression de nœud.

```
o──────────────────o ↓ racine, cléLue, donnéeLue
| ajout dans ABR |
o──────────────────o ↓ racine, message
┌─── *
| message = " "
| o──────────────────o ↓ racine, cléLue
| | pNoeudRecherché |
| o──────────────────o ↓ pNoeud, père
```

```
|┌── if (pNoeud ≠ null)
|| message = "erreur, la clé est déjà présente dans l'arbre"
|├── else
|| pNoeudNouv = adresse mémoire nouveau nœud
||┌── if (pNoeudNouv == null)
||| message = "Plus de place mémoire"
||├── else
||| // garnir le nouveau noeud
||| pNoeudNouv → clé = cléLue
||| pNoeudNouv → donnée = donnéeLue
||| pNoeudNouv → pGauche = null
||| pNoeudNouv → pDroit = null
||| // attacher le noeud à l'arbre
|||┌── if(racine == null) // arbre vide
|||| racine = pNoeudNouv
|||├── else
||||┌── if(cléLue < père → clé)
||||| père → pGauche = pNoeudNouv
||||├── else
||||| père → pDroit = pNoeudNouv
||||└──
|||└──
||└──
|└──
└──────────
```

```
o──────────────────────────────────────o ↓ racine, cléLue
| suppression d'un nœud par déplacement |
o──────────────────────────────────────o ↓ racine, message
┌─── *
| o────────────────o ↓ racine, cléLue
| | pNoeudRecherché |
| o────────────────o ↓ pNoeud, père
| // voir point 5.5.3.
|
|┌── if (pNoeud == null)
||
|| message = "clé absente de l'arbre"
||
|├── else
||
||┌── if (pNoeud → pDroit == null) // le noeud n'a pas de fils droit
|||
|||┌── if(pNoeud == racine)
|||| racine = racine → pGauche
|||├── else
||||┌── if(cléLue < père → clé)
||||| père → pGauche = pNoeud → pGauche
||||├── else
||||| père → pDroit = pNoeud → pGauche
||||└──
|||└──
||├── else // le noeud a un fils droit
|||
|||┌── if(pNoeud == racine)
|||| racine = racine → pDroit
|||├── else
```

```
|||||┌── if(cléLue < père → clé)
|||||| père → pGauche = pNoeud → pDroit
|||||├── else
|||||| père → pDroit = pNoeud → pDroit
|||||└──
||||└──
|||
||||┌── if( pNoeud → pGauche ≠ null)
|||| pNoeudAccroche = pNoeud → pDroit
||||╔══ do while (pNoeudAccroche → pGauche ≠ null)
|||||‖ pNoeudAccroche = pNoeudAccroche → pGauche
||||╚══
|||| // pNoeudAccroche est le plus petit noeud du sous-arbre droit
||||
|||| pNoeudAccroche → pGauche = pNoeud → pGauche
||||└──
||└──
|| libérer la mémoire pointée par pNoeud
|└──
|
└──────────
```

Prefixe

```
o──────────────────────────o ↓ racine
| parcours itératif préfixe |
o──────────────────────────o
┌─── *
| o────────────────────o
| | initialisationPile |
| o────────────────────o ↓ sommet
| pNoeud = racine
|
|╔══ do while (pNoeud ≠ null OR "pile non vide")
|‖
|‖╔══ do while (pNoeud ≠ null)
|‖‖ o──────────────o ↓ pNoeud
|‖‖ | traiterNoeud |
|‖‖ o──────────────o
|‖‖ o──────────o ↓ sommet, pNoeud
|‖‖ | empiler |
|‖‖ o──────────o ↓ sommet
|‖‖
|‖‖ pNoeud = pNoeud → pGauche
|‖╚══
|‖ o──────────o ↓ sommet
|‖ | dépiler |
|‖ o──────────o ↓ sommet, pNoeud
|‖
|‖ pNoeud = pNoeud → pDroit
|╚══
└──────────
```

```
o──────────────────────────o ↓ racine
```

```
| parcours récursif préfixe |
o————————————————————————o
avec
o————————————————————————o ↓ pNoeud
| parcours récursif préfixe |
o————————————————————————o
┌─── *
|┌── if (pNoeud ≠ null)
|| o————————————————————o ↓ pNoeud
|| | traitement du nœud |
|| o————————————————————o
|| o————————————————————o ↓ pNoeud→ pGauche
|| | parcours récursif préfixe |
|| o————————————————————o
|| o————————————————————o ↓ pNoeud→ pDroit
|| | parcours récursif préfixe |
|| o————————————————————o
|└──
└──────────
```

Infixe

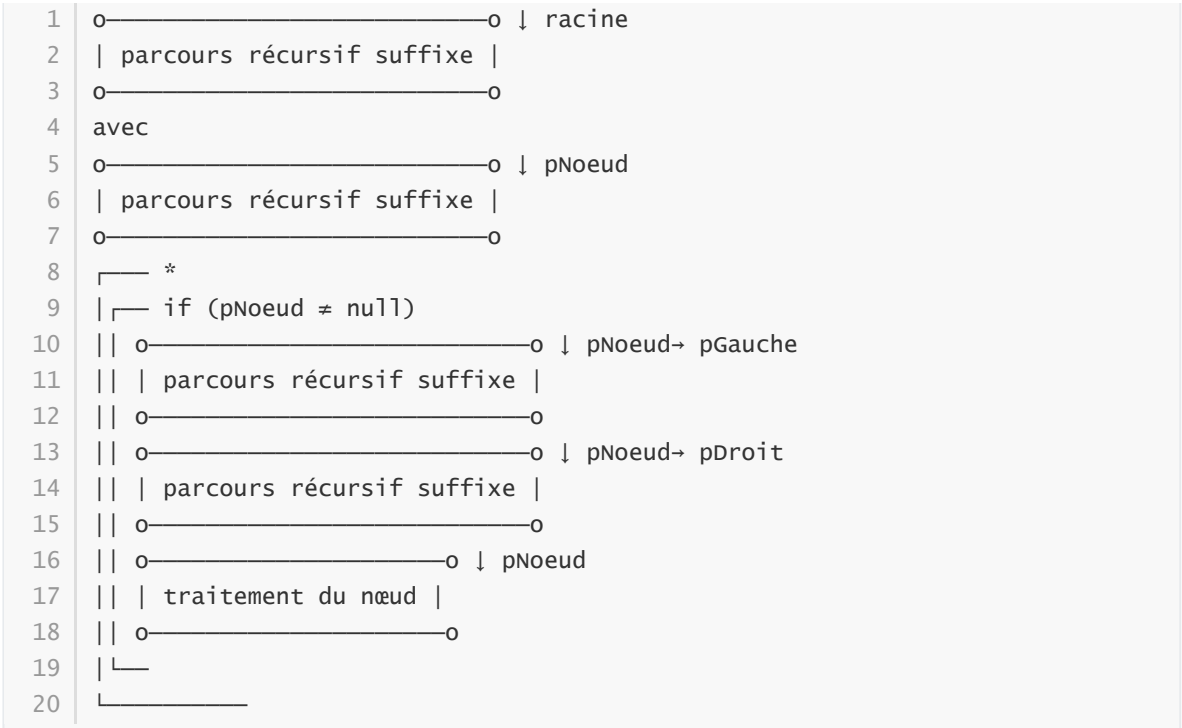
```
o————————————————————————o ↓ racine
| parcours itératif infixe |
o————————————————————————o
┌─── *
| o————————————————————o
| | initialisationPile |
| o————————————————————o ↓ sommet
| pNoeud = racine
|
|╔══ do while (pNoeud ≠ null OR "pile non vide")
||
||╔══ do while (pNoeud ≠ null)
|||| o——————————o ↓ sommet, pNoeud
|||| | empiler |
|||| o——————————o ↓ sommet
||||
|||| pNoeud = pNoeud → pGauche
||╚══
|| o——————————o ↓ sommet
|| | dépiler |
|| o——————————o ↓ sommet, pNoeud
|| o————————————————o ↓ pNoeud
|| | traiterNoeud |
|| o————————————————o
|| pNoeud = pNoeud → pDroit
|╚══
└──────────
```

```
o————————————————————————o ↓ racine
| parcours récursif infixe |
o————————————————————————o
avec
o————————————————————————o ↓ pNoeud
```

```
| parcours récursif infixe |
o————————————————————————o
┌─── *
|┌── if (pNoeud ≠ null)
|| o————————————————————————o ↓ pNoeud→ pGauche
|| | parcours récursif infixe |
|| o————————————————————————o
|| o————————————————————o ↓ pNoeud
|| | traitement du nœud |
|| o————————————————————o
|| o————————————————————————o ↓ pNoeud→ pDroit
|| | parcours récursif infixe |
|| o————————————————————————o
|└──
└──────────
```

Suffixe

```
o————————————————————————o ↓ racine
| parcours itératif suffixe |
o————————————————————————o
┌─── *
| o————————————————————o
| | initialisationPile |
| o————————————————————o ↓ sommet
| pNoeud = racine
|
|╔══ do while (pNoeud ≠ null OR "pile non vide")
|‖
|‖╔══ do while (pNoeud ≠ null)
|‖‖ o——————————o ↓ sommet, pNoeud, passage = 1
|‖‖ | empiler |
|‖‖ o——————————o ↓ sommet
|‖‖
|‖‖ pNoeud = pNoeud → pGauche
|‖└──
|‖ o——————————o ↓ sommet
|‖ | dépiler |
|‖ o——————————o ↓ sommet, pNoeud, passage
|‖┌── if (passage == 1)
|‖| passage ++
|‖| o——————————o ↓ sommet, pNoeud, passage
|‖| | empiler |
|‖| o——————————o ↓ sommet
|‖|
|‖| pNoeud = pNoeud → pDroit
|‖├── else
|‖| o————————————o ↓ pNoeud
|‖| | traiterNoeud |
|‖| o————————————o
|‖| pNoeud = null
|‖└──
|╚──
└──────────
```


o——————————————————o ↓ racine
| parcours récursif suffixe |
o——————————————————o
avec
o——————————————————o ↓ pNoeud
| parcours récursif suffixe |
o——————————————————o
┌── *
|┌── if (pNoeud ≠ null)
|| o——————————————————o ↓ pNoeud→ pGauche
|| | parcours récursif suffixe |
|| o——————————————————o
|| o——————————————————o ↓ pNoeud→ pDroit
|| | parcours récursif suffixe |
|| o——————————————————o
|| o——————————————————o ↓ pNoeud
|| | traitement du nœud |
|| o——————————————————o
|└──
└──────────