

Conceptions des TAD Collections

Structures statiques, dynamiques et mixtes

Nicolas Delestre



Conceptions TAD Collections v3.4.1

1 / 54

Introduction

Rappels

On a vu un certain nombre de TAD collections :

- les listes
- les listes ordonnées
- les piles
- les files
- les dictionnaires
- ...

Comment les représenter ?



Conceptions TAD Collections v3.4.1

3 / 54

Plan...

1 Introduction

2 Conception préliminaire

- Les opérations → fonctions/procédures
- Utilisation

3 Conception détaillée

- Représentation à l'aide de tableaux
- Représentation à l'aide de structures de données dynamiques
 - Utilisation de la SDD ListeChaine
 - Utilisation de la SDD ArbreBinaire
- Représentation mixte : table de hachage



Conceptions TAD Collections v3.4.1

2 / 54

Conception préliminaire

Contraintes informatiques

- Les TAD sont des représentations mathématiques des types
 - les éléments existent en nombre fini ou infini
 - les opérations (fonctions mathématiques) permettent d'accéder aux éléments et de passer d'un élément à un autre
- La mémoire d'un ordinateur n'est pas infini. Dans le paradigme de la programmation structurée, les opérations :
 - modifient les valeurs des paramètres : procédure avec passage de paramètre en entrée/sortie ou en sortie
 - créent de nouvelles valeurs : fonction ou procédure avec passage de paramètre en sortie
- Il faut faire le bon choix de façon à être efficient

Attention

Une fois que l'on a déterminé comment les représenter on obtient non plus un TAD mais un type de données (l'aspect *abstrait* disparaît)

Conceptions TAD Collections v3.4.1

4 / 54

Pile

- **fonction** pile () : Pile
- **fonction** estVide (unePile : Pile) : **Booleen**
- **procédure** empiler (E/S unePile : Pile, E element : Element)
- **procédure** dépiler (E/S unePile : Pile)
 - |précondition(s) non(estVide(unePile))
- **fonction** obtenirElement (unePile : Pile) : Element
 - |précondition(s) non(estVide(unePile))



File

- **fonction** file () : File
- **fonction** estVide (uneFile : File) : **Booleen**
- **procédure** enfiler (E/S uneFile : File, E element : Element)
- **procédure** défiler (E/S uneFile : File)
 - |précondition(s) non(estVide(uneFile))
- **fonction** obtenirElement (uneFile : File) : Element
 - |précondition(s) non(estVide(uneFile))



Liste

- **fonction** liste () : Liste
- **fonction** estVide (uneListe : Liste) : **Booleen**
- **procédure** insérer (E/S uneListe : Liste, E position : **Naturel**, element : Element)
 - |précondition(s) $1 \leq position \leq longueur(uneListe) + 1$
- **procédure** supprimer (E/S uneListe : Liste, E position : **Naturel**)
 - |précondition(s) $1 \leq position \leq longueur(uneListe)$
- **fonction** obtenirElement (uneListe : Liste, position : **Naturel**) : Element
 - |précondition(s) $1 \leq position \leq longueur(uneListe)$
- **fonction** longueur (uneListe : Liste) : **Naturel**



ListeOrdonnee

- **fonction** listeOrdonnee () : ListeOrdonnee
- **fonction** estVide (uneListe : ListeOrdonnee) : **Booleen**
- **procédure** insérer (E/S uneListe : ListeOrdonnee, E element : Element)
- **procédure** supprimer (E/S uneListe : ListeOrdonnee, E position : **Naturel**)
 - |précondition(s) $1 \leq position \leq longueur(uneListe)$
- **fonction** obtenirElement (uneListe : ListeOrdonnee, position : **Naturel**) : Element
 - |précondition(s) $1 \leq position \leq longueur(uneListe)$
- **fonction** longueur (uneListe : ListeOrdonnee) : **Naturel**



Ensemble

- **fonction** ensemble () : Ensemble
- **procédure** ajouter (E/S unEnsemble : Ensemble, E element : Element)
- **procédure** retirer (E/S unEnsemble : Ensemble, E element : Element)
- **fonction** estPrésent (unEnsemble : Ensemble, element : Element) : **Booleen**
- **fonction** cardinalite (unEnsemble : Ensemble) : **Naturel**
- **fonction** union (e1, e2 : Ensemble) : Ensemble
- **fonction** intersection (e1, e2 : Ensemble) : Ensemble
- **fonction** soustraction (e1, e2 : Ensemble) : Ensemble



Tas

- **fonction** tas () : Tas
- **fonction** estVide (unArbre : Tas) : **Booleen**
- **procédure** insérer (E/S unArbre : Tas, E element : Element)
- **procédure** supprimer (E/S unArbre : Tas, E element : Element)
- **fonction** estPrésent (unArbre : Tas, element : Element) : **Booleen**
- **fonction** obtenirElement (unArbre : Tas) : Element
|précondition(s) non estVide(unArbre)
- **fonction** obtenirFilsGauche (unArbre : Tas) : Tas
|précondition(s) non estVide(unArbre)
- **fonction** obtenirFilsDroit (unArbre : Tas) : Tas
|précondition(s) non estVide(unArbre)



Dictionnaire

- **fonction** dictionnaire () : Dictionnaire
- **procédure** ajouter (E/S unDictionnaire : Dictionnaire, E clef : Clef, element : Valeur)
- **procédure** retirer (E/S unDictionnaire : Dictionnaire, E clef : Clef)
- **fonction** estPresent (unDictionnaire : Dictionnaire, clef : Clef) : **Booleen**
- **fonction** obtenirValeur (unDictionnaire : Dictionnaire, clef : Clef) : Valeur
|précondition(s) estPresent(unDictionnaire, clef)
- **fonction** obtenirClefs (unDictionnaire : Dictionnaire) : Liste<Clef>
- **fonction** obtenirValeurs (unDictionnaire : Dictionnaire) : Liste<Valeur>



Arbre binaire de recherche

- **fonction** aBR () : ABR
- **fonction** estVide (unArbre : ABR) : **Booleen**
- **procédure** insérer (E/S unArbre : ABR, E element : Element)
- **procédure** supprimer (E/S unArbre : ABR, E element : Element)
- **fonction** estPrésent (unArbre : ABR, element : Element) : **Booleen**
- **fonction** obtenirElement (unArbre : ABR) : Element
|précondition(s) non estVide(unArbre)
- **fonction** obtenirFilsGauche (unArbre : ABR) : ABR
|précondition(s) non estVide(unArbre)
- **fonction** obtenirFilsDroit (unArbre : ABR) : ABR
|précondition(s) non estVide(unArbre)



Utilisation

- Lorsque l'on déclare une variable de type collection, on suffixe le nom du type par le type réel des éléments
- Dans les algorithmes de la conception détaillée, si il y a ambiguïté sur les fonctions ou procédures utilisées, on préfixe le nom de la fonction ou de la procédure par le nom du TAD s'y rattachant (utilisation du "." pour séparer ces deux noms)

Exemple

```
a : ArbreBinaire<Entier>
l : Liste<Entier>
...
si ArbreBinaire.estVide(a) alors
  ...
finsi
...
```

Trois types de représentation

Objectif

- L'objectif est de décrire comment sont représentés les TAD collection (Pile, File, Liste, ListeOrdonnée, etc.), c'est-à-dire :
 - **Type** NomType = Représentation
 - Définir les algorithmes des fonctions et procédures vu dans la conception détaillée

Les collections sont des types de données qui "stockent" des éléments de même type

- 1 On peut utiliser des tableaux : structures de données statiques (SDS)
- 2 On peut utiliser les SDD : structures de données dynamiques
- 3 On peut utiliser à la fois des SDS et SDD

Représentation à l'aide de tableaux

- Très souvent ne fonctionne bien qu'avec les TAD collection non hiérarchique (sinon que signifie obtenirFilsGauche?)
- Besoin de stocker les éléments, donc besoin d'un tableau, et donc besoin de savoir combien d'éléments sont significatifs
- Les collections (linéaires) sont donc représentées avec une structure (au sens algorithmique) composée :
 - d'un tableau d'éléments
 - du nombre d'éléments significatifs (un nombre d'éléments significatifs nul représente un ensemble vide)

Attention

Il se peut quelque fois que l'on veuille donner une structure d'arbre à l'organisation des éléments d'un tableau (par exemple pour représenter un tas).

Un premier exemple : la Pile 1 / 2

```
Type Pile = Structure
  lesElements : Tableau[1..MAX] de Element
  nbElements : Naturel
finstructure
fonction pile () : Pile
  Déclaration resultat : Pile
debut
  resultat.nbElements ← 0
  retourner resultat
fin
fonction estVide (unePile : Pile) : Booleen
debut
  retourner unePile.nbElements=0
fin
```

Un premier exemple : la Pile 2 / 2

```

procédure empiler (E/S unePile : Pile, E e : Element)
debut
    unePile.nbElements ← unePile.nbElements+1
    unePile.lesElements[unePile.nbElements] ← e
fin
procédure depiler (E/S unePile : Pile)
    |précondition(s) non estVide(unePile)
debut
    unePile.nbElements ← unePile.nbElements-1
fin
fonction obtenirElement (unePile : Pile) : Element
    |précondition(s) non estVide(unePile)
debut
    retourner unePile.lesElements[unePile.nbElements]
fin

```



Un deuxième exemple : la Liste 1 / 3

```

Type Liste = Structure
    lesElements : Tableau[1..MAX] de Element
    nbElements : Naturel
finstructure
fonction liste () : Liste
    Déclaration resultat : Liste
debut
    resultat.nbElements ← 0
    retourner resultat
fin
fonction estVide (uneListe : Liste) : Booleen
debut
    retourner uneListe.nbElements=0
fin

```



Un deuxième exemple : la Liste 2 / 3

```

procédure insérer (E/S uneListe : Liste, E indice : Naturel, e : Element)
    |précondition(s)  $1 \leq \text{indice} \leq \text{longueur}(\text{uneListe}) + 1$ 
debut
    decalerVersLaDroite(uneListe, indice)
    uneListe.nbElements ← uneListe.nbElements+1
    uneListe.lesElements[indice] ← e
fin
procédure supprimer (E/S uneListe : Liste, E indice : Naturel)
    |précondition(s)  $1 \leq \text{indice} \leq \text{longueur}(\text{uneListe})$ 
debut
    decalerVersLaGauche(uneListe, indice+1)
    uneListe.nbElements ← uneListe.nbElements-1
fin
fonction obtenirElement (uneListe : Liste, indice : Naturel) : Element
    |précondition(s)  $1 \leq \text{indice} \leq \text{longueur}(\text{uneListe})$ 
debut
    retourner uneListe.lesElements[indice]
fin

```



Un deuxième exemple : la Liste 3 / 3

```

fonction longueur (uneListe : Liste) : Naturel
debut
    retourner uneListe.nbElements
fin
procédure decalerVersLaDroite (E/S uneListe : Liste, E indice : Naturel)
    Déclaration i : Naturel
debut
    pour i ← uneListe.nbElements à indice pas de -1 faire
        uneListe.lesElements[i+1] ← uneListe.lesElements[i]
    finpour
fin
procédure decalerVersLaGauche (E/S uneListe : Liste, E indice : Naturel)
    Déclaration i : Naturel
debut
    pour i ← indice à uneListe.nbElements faire
        uneListe.lesElements[i-1] ← uneListe.lesElements[i]
    finpour
fin

```



Avantage / Inconvénient

Avantage

- Simple à programmer

Inconvénients

- Structure statique :
 - Limitée par un nombre d'éléments maximal (MAX)
 - Même si très peu d'éléments stockés, MAX éléments réservés
 - Problème des collections définies récursivement



Utilisation de la SDD ListeChaine 1 / 6

- Les types Pile, File, Liste, ListeOrdonnee peuvent être représentés à l'aide de la SDD ListeChaine ou d'une structure contenant une ListeChaine :

Type XX = ListeChaine

Type XX = Structure

lesElements : ListeChaine

...

finstructure

- Les fonctions/procédures de ces types utilisent donc les fonctions/procédures de la SDD ListeChaine



Utilisation de la SDD ListeChaine 2 / 6

Pile

Type Pile = ListeChaine

Le sommet de la pile est représenté par le premier élément de la liste chaînée :

- estVide → estVide
- empiler → ajouter
- dépiler → supprimerTete
- obtenirElement → obtenirElement



Utilisation de la SDD ListeChaine 3 / 6

File

Type File = Structure

debut : ListeChaine

fin : ListeChaine

finstructure

debut référence la tête de la liste chaînée et *fin* le dernier élément :

- estVide → estVide
- enfiler → une opération qui permet d'« ajouter » un élément au niveau du champ *fin*
- défiler → supprimerTete
- obtenirElement → obtenirElement



Utilisation de la SDD ListeChaine 4 / 6

Liste

Type Liste = Structure

lesElements : ListeChaine

nbElements : **Naturel****finstructure**

Les éléments de la liste sont stockés dans le même ordre que celui de la liste chaînée

- estVide → estVide
- insérer → une opération qui insère un élément à la ième place de la liste chaînée et incrémentation du champ nbElements
- supprimer → une opération qui supprime le ième élément et décrémentation du champ nbElements
- obtenirElement → une opération qui permet d'obtenir le ième élément de la liste chaînée
- longueur → accès au champ nbElements



Utilisation de la SDD ListeChaine 5 / 6

ListeOrdonnee

Type ListeOrdonnee = Structure

lesElements : ListeChaine

nbElements : **Naturel****finstructure**

Les éléments de la liste ordonnée sont stockés dans le même ordre que celui de la liste chaînée

- estVide → estVide
- insérer → une opération qui insère un élément et incrémentation du champ nbElements
- supprimer → une opération qui supprime le ième élément et décrémentation du champ nbElements
- obtenirElement → une opération qui permet d'obtenir le ième élément de la liste chaînée
- longueur → accès au champ nbElements



Utilisation de la SDD ListeChaine 6 / 6

Ensemble

Type Ensemble = Structure

lesElements : ListeChaine

nbElements : **Naturel****finstructure**

Les éléments de l'ensemble sont stockés dans le même ordre que celui de la liste chaînée

- ajouter → ajouter (en vérifiant au préalable que l'élément n'est pas présent)
- retirer → une opération qui supprime un élément et décrémente le champ nbElements si l'élément a bien été supprimé
- estPrésent → une opération qui permet de savoir si un élément est présent dans la liste
- cardinalité → accès au champ nbElements
- union → une opération qui crée un nouvel ensemble en ajoutant les éléments des deux ensembles
- soustraction → une opération qui crée un nouvel ensemble ou sera ajouté les éléments du premier ensemble qui ne sont pas dans l'autre (opération privée)
- intersection → une opération qui crée un nouvel ensemble et qui appelle deux fois une opération qui ajoutera les éléments d'un ensemble qui sont aussi présent dans un autre



Un exemple : Ensemble 1 / 9

ensemble

fonction ensemble () : Ensemble

Déclaration resultat : Ensemble**debut**

resultat.lesElements ← listeChaine()

resultat.nbElements ← 0

retourner resultat**fin**

ajouter

procédure ajouter (E/S e : Ensemble, E el : Element)

debut**si** non Ensemble.estPresent(e,el) **alors**

ListeChaine.ajouter(e.lesElements,el)

e.nbElements ← e.nbElements+1

finsi**fin**

Un exemple : Ensemble 2 / 9

retirer (début)

```

procédure retirerSiPresent (E/S l : ListeChaine, E e : Element, S etaitPresent : Booleen)
  Déclaration temp : ListeChaine
debut
  si estVide(l) alors
    etaitPresent ← FAUX
  sinon
    si obtenirElement(l)=e alors
      supprimerTete(l)
      etaitPresent ← VRAI
    sinon
      temp ← obtenirListeSuivante(l)
      retirerSiPresent(temp,e,etaitPresent)
      fixerListeSuivante(l,temp)
    finsi
  finsi
fin

```



Un exemple : Ensemble 3 / 9

retirer (fin)

```

procédure retirer (E/S e : Ensemble, E el : Element)
  Déclaration etaitPresent : Booleen
debut
  retirerSiPresent(e.lesElements,el,etaitPresent)
  si etaitPresent alors
    e.nbElements ← e.nbElements-1
  finsi
fin

```



Un exemple : Ensemble 4 / 9

estPresent

```

fonction estPresentDansListe (l : ListeChaine, e : Element) : Booleen
debut
  si estVide(l) alors
    retourner FAUX
  sinon
    si obtenirElement(l)=e alors
      retourner VRAI
    sinon
      retourner estPresentDansListe(obtenirListeSuivante(l),e)
    finsi
  finsi
fin
fonction estPresent (e : Ensemble, el : Element) : Booleen
debut
  retourner estPresentDansListe(e.lesElements,el)
fin

```



Un exemple : Ensemble 5 / 9

cardinalite

```

fonction cardinalite (e : Ensemble) : Naturel
debut
  retourner e.nbElements
fin

```



Un exemple : Ensemble 6 / 9

union

```

procédure ajouterElementsListe (E/S e : Ensemble, E l : ListeChaine)
debut
    tant que non ListeChaine.estVide(l) faire
        Ensemble.ajouter(e,obtenirElement(l))
        l ← obtenirListeSuivante(l)
    fintantque
fin
fonction union (e1, e2 : Ensemble) : Ensemble
    Déclaration resultat : Ensemble
debut
    resultat ← ensemble()
    ajouterElementsListe(resultat,e1.lesElements)
    ajouterElementsListe(resultat,e2.lesElements)
    retourner resultat
fin

```



Un exemple : Ensemble 7 / 9

soustraction

```

procédure ajouterElementsListeSiNonPresent (E/S e : Ensemble, E l : ListeChaine, eRef : Ensemble)
debut
    tant que non ListeChaine.estVide(l) faire
        si non Ensemble.estPresent(eRef,obtenirElement(l)) alors
            Ensemble.ajouter(e,obtenirElement(l))
        finsi
        l ← obtenirListeSuivante(l)
    fintantque
fin
fonction soustraction (e1, e2 : Ensemble) : Ensemble
    Déclaration resultat : Ensemble
debut
    resultat ← ensemble()
    ajouterElementsListeSiNonPresent(resultat,e1.lesElements,e2)
    retourner resultat
fin

```



Un exemple : Ensemble 8 / 9

intersection

```

procédure ajouterElementsListeSiPresent (E/S e : Ensemble, E l : ListeChaine, eRef : Ensemble)
debut
    tant que non ListeChaine.estVide(l) faire
        si Ensemble.estPresent(eRef,obtenirElement(l)) alors
            Ensemble.ajouter(e,obtenirElement(l))
        finsi
        l ← obtenirListeSuivante(l)
    fintantque
fin
fonction intersection (e1, e2 : Ensemble) : Ensemble
    Déclaration resultat : Ensemble
debut
    resultat ← ensemble()
    ajouterElementsListeSiPresent(resultat,e1.lesElements,e2)
    retourner resultat
fin

```



Un exemple : Ensemble 9 / 9

Exercice

On aurait pu généraliser les trois dernières procédures privées en une seule (ajouterElementsListeSousCondition) en passant une condition en paramètre.

- Proposez le type de fonction correspondant à cette condition
- Donnez la procédure ajouterElementsListeSousCondition
- Donnez l'algorithme de fonctions union, soustraction et intersection en précisant pour chacun l'algorithme de la fonction qui sera donnée comme paramètre effectif lors de l'appel à la procédure ajouterElementsListeSousCondition



ABR 1 / 6

Arbre Binaire de recherche : ABR

Type ABR = ArbreBinaire

- `estVide` → `estVide`
- `insérer` → *une opération qui insère un élément*
- `supprimer` → *une opération qui supprime un élément*
- `estPresent` → *une opération qui recherche un élément*
- `obtenirElement` → `obtenirElement`
- `obtenirFilsGauche` → `obtenirFilsGauche`
- `obtenirFilsDroit` → `obtenirFilsDroit`



ABR 2 / 6

estPresent

fonction `estPresent` (`a` : ABR, `e` : Element) : Booleen

Déclaration `temp` : ABR

```

debut
  si estVide(a) alors
    retourner FAUX
  sinon
    si e==obtenirElement(a) alors
      retourner VRAI
    sinon
      si e<obtenirElement(a) alors
        retourner estPresent(obtenirFilsGauche(a),e)
      sinon
        retourner estPresent(obtenirFilsDroit(a),e)
    finsi
  finsi
fin

```



ABR 3 / 6

insertion

Principe : parcourt l'arbre jusqu'à arriver sur un arbre vide

procédure `insérer` (`E/S a` : ABR, `E e` : Element)

Déclaration `temp` : ABR

```

debut
  si estVide(a) alors
    a ← ajouterRacine(arbreBinaireRecherche(), arbreBinaireRecherche(), e)
  sinon
    si e≤obtenirElementRacine(a) alors
      temp ← obtenirFilsGauche(a)
      insérer(temp, e)
      fixerFilsGauche(a, temp)
    sinon
      temp ← obtenirFilsDroit(a)
      insérer(temp, e)
      fixerFilsDroit(a, temp)
  finsi
fin

```



ABR 4 / 6

suppression (début)

Principe : Lorsque l'on a trouvé l'élément à supprimer, il y a trois cas :

- 1 L'arbre est une feuille : suppression de l'arbre
- 2 L'arbre a un seul fils : l'arbre devient le fils (et suppression du nœud)
- 3 L'arbre a deux fils : la nouvelle racine devient le plus grand des plus petits (ou le plus petit des plus grands)

fonction `lePlusGrand` (`a` : ABR) : ABR

précondition(s) `non estVide(a)`

```

debut
  si estVide(obtenirFilsDroit(a)) alors
    retourner a
  sinon
    retourner lePlusGrand(obtenirFilsDroit(a))
  finsi
fin

```



ABR 5 / 6

suppression (suite)

procédure supprimer (E/S a : ABR ; E e : Element)

Déclaration nelleValeurRacine : Element
temp, tempG, tempD : ABR

```

debut
  si non estVide(a) alors
    si e < obtenirElement(a) alors
      temp ← obtenirFilsGauche(a)
      supprimer(temp, e)
      fixerFilsGauche(a, temp)
    sinon
      si e > obtenirElement(a) alors
        temp ← obtenirFilsDroit(a)
        supprimer(temp, e)
        fixerFilsDroit(a, temp)
      sinon
        si estVide(obtenirFilsGauche(a)) et estVide(obtenirFilsDroit(a)) alors
          voir cas 1
        sinon
          si estVide(obtenirFilsGauche(a)) ou estVide(obtenirFilsDroit(a)) alors
            voir cas 2
          sinon
            voir cas 3
          fin
        fin
      fin
    fin
  fin

```

ABR 6 / 6

suppression (fin)

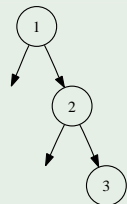
- Cas 1 :
ArbreBinaire.supprimerRacine(a, tempG, tempD)
- Cas 2 :
ArbreBinaire.supprimerRacine(a, tempG, tempD)
si estVide(tempG) **alors**
a ← tempD
sinon
a ← tempG
fin
- Cas 3 :
ArbreBinaire.supprimerRacine(a, tempG, tempD)
nelleValeurRacine ← obtenirElement(lePlusGrand(tempG))
supprimer(tempG, nelleValeurRacine)
a ← ArbreBinaire.ajouterRacine(nelleValeurRacine, tempG, tempD)

Constats

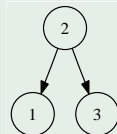
Résultat de l'insertion successive de ...

- 1, 2 et 3
- 2, 1 et 3

Cas 1



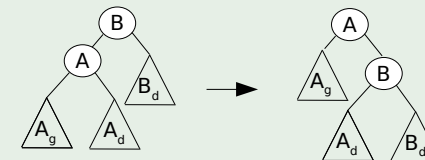
Cas 2



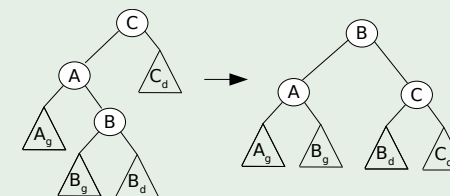
Il faudrait un algorithme qui laisse après insertion l'arbre équilibré :
⇒ Utilisation des algorithmes de simple et double rotations

AVL 1 / 4

Simple Rotation À Droite



Double Rotation À Droite



AVL 2 / 4

insérer (début)

procédure insérer (E/S a : ABR, E e : Element)

Déclaration temp : ABR

debut

si estVide(a) alors

a ← ajouterRacine(arbreBinaireRecherche(), arbreBinaireRecherche(), e)

sinon

si e ≤ obtenirElementRacine(a) alors

temp ← obtenirFilsGauche(a)

insérer(temp, e)

fixerFilsGauche(a, temp)

voir cas 1

sinon

temp ← obtenirFilsDroit(a)

insérer(temp, e)

fixerFilsDroit(a, temp)

voir cas 2

finsi

finsi

fin

AVL 3 / 4

insérer (fin)

- Cas 1 :

si hauteur(obtenirFilsGauche(a)) > hauteur(obtenirFilsDroit(a)) + 1 alors

si hauteur(obtenirFilsGauche(obtenirFilsGauche(a))) ≥

hauteur(obtenirFilsDroit(obtenirFilsGauche(a))) alors

faireSimpleRotationDroite(a)

sinon

faireDoubleRotationDroite(a)

finsi

finsi

- Cas 2 :

si hauteur(obtenirFilsDroit(a)) > hauteur(obtenirFilsGauche(a)) + 1 alors

si hauteur(obtenirFilsGauche(obtenirFilsDroit(a))) ≤

hauteur(obtenirFilsDroit(obtenirFilsDroit(a))) alors

faireSimpleRotationGauche(a)

sinon

faireDoubleRotationGauche(a)

finsi

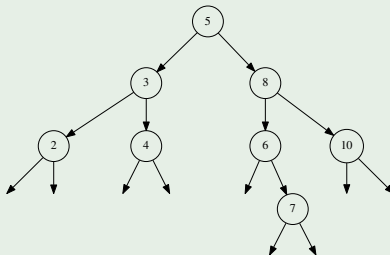
finsi

AVL 4 / 4

Exercice

Donner l'algorithme de la procédure supprimer.

Principe : « La suppression dans un arbre AVL peut se faire par rotations successives du nœud à supprimer jusqu'à une feuille (en choisissant ces rotations de sorte que l'arbre reste équilibré), et ensuite en supprimant cette feuille directement. » (Wikipedia)

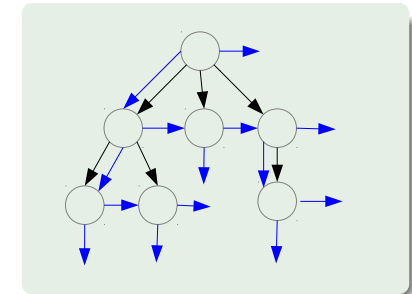


Les arbres n-aires

Deuxième conception

Type Arbre = ArbreBinaire
avec :

- le fils gauche d'un nœud de l'arbre binaire pour représenter le premier fils d'un nœud de l'arbre
- le fils droit d'un nœud de l'arbre binaire pour représenter le prochain frère fils d'un nœud de l'arbre



Exercice

Donnez les algorithmes des fonctions et procédures avec cette représentation

Table de hachage 1 / 3

Table de hashage

- Combinaison des avantages des tableaux (accès aux éléments en $O(1)$) et des listes chaînées (nombre maximal d'éléments variables)
⇒ Tableau de listes chaînées
- Utilisation d'une fonction de hachage et du modulo pour trouver l'indice des valeurs à stocker (à partir de tout ou partie de ces valeurs)
 - La fonction de hachage est une fonction non bijective de *Valeur* (ou une partie de *Valeur*) vers *Naturel* ($[0..MAX_NATUREL]$) tel qu'une petite modification sur valeur donne un naturel très différent
 - Le modulo permet de réduire ce hache à l'espace des indices du tableau



Table de hachage 2 / 3

Exemple d'introduction (suite)

- Stockage de l'âge de personnes (valeur correspond à prénom et âge) avec fonction de hachage h utilisant le prénom :
 - $h(CECILE) = 1 + (3+5+3+9+12+5) \bmod 11 = 5$

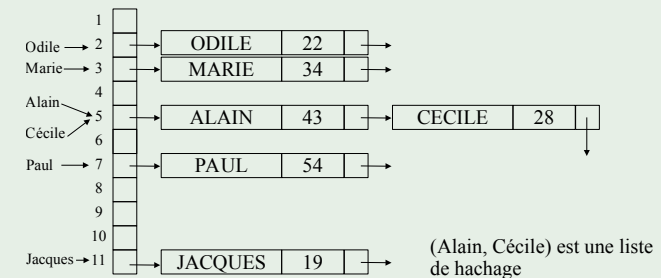


Table de hachage 3 / 3

Les performances d'une table de hachage dépend :

- 1 Du nombre d'éléments que l'on veut stocker au regard de la taille du tableau
- 2 De la qualité de la fonction de hachage :
 - Répartition uniforme (minimise les collisions)
 - Temps de calcul

Exemples de fonction de hashage :

- Extraction :
 - On extrait p bits de la représentation binaire de la (partie de la) valeur
- Compression :
 - On découpe la représentation binaire de la (partie de la) valeur en ss-chaînes d'égales longueurs
 - On additionne ces sous-chaînes avec des "ou exclusifs"
- Division :
 - On calcule le reste de la division de la représentation en naturel de la (partie de la) valeur par m (il est recommandé de prendre m premier)

Exemples de représentation

- TAD Ensemble : les valeurs sont les éléments dans leur entièreté
- TAD Dictionnaire : les valeurs correspondent aux couples (clé, valeur) et la fonction de hachage utilise uniquement la clé



Conclusion 1 / 2

		Conceptions			
		Tableau	Liste chaînée	Arbre binaire	Table de hachage
TAD	Pile	obtenirElement	$\Omega(1) O(1)$	$\Omega(1) O(1)$	
		empiler	$\Omega(1) O(1)$	$\Omega(1) O(1)$	
		depiler	$\Omega(1) O(1)$	$\Omega(1) O(1)$	
	File	obtenirElement	$\Omega(1) O(1)$	$\Omega(1) O(1)$	
		enfiler	$\Omega(1) O(1)$	$\Omega(1) O(1)$	
		defiler	$\Omega(1) O(1)$	$\Omega(1) O(1)$	
	Liste	obtenirElement	$\Omega(1) O(1)$	$\Omega(1) O(n)$	
		insérer	$\Omega(1) O(n)$	$\Omega(1) O(n)$	
		supprimer	$\Omega(1) O(n)$	$\Omega(1) O(n)$	
	Liste Ordonnée	obtenirElement	$\Omega(1) O(1)$	$\Omega(1) O(n)$	
		insérer	$\Omega(1) O(n)$	$\Omega(1) O(n)$	$\Omega(1) O(\log_2(n))$
		supprimer	$\Omega(1) O(n)$	$\Omega(1) O(n)$	$\Omega(1) O(\log_2(n))$
Ensemble	estPresent	$\Omega(1) O(n)$	$\Omega(1) O(n)$	$\Omega(1) O(n)$	$\Omega(1) O(N/K)$
	ajouter	$\Omega(1) O(1)$	$\Omega(1) O(n)$	$\Omega(1) O(\log_2(n))$	$\Omega(1) O(N/K)$
	supprimer	$\Omega(1) O(1)$	$\Omega(1) O(n)$	$\Omega(1) O(\log_2(n))$	$\Omega(1) O(N/K)$

Utilisation des algorithmes d'insertions et suppressions AVL ou Rouge et Noir. Le type des éléments doit donc avoir un ordre total
Les éléments (ou une partie des éléments) doivent être hachable. K représente la taille du tableau. On considère la complexité dans le pire des cas de la fonction de hachage en $O(1)$.



Conclusion 2 / 2

		Conceptions			
		Tableau	Liste chaînée	Arbre binaire	Table de hachage
TAD	Dictionnaire	obtenirElement	$\Omega(1) O(n)$	$\Omega(1) O(n)$	$\Omega(1) O(N/K)$
		ajouter	$\Omega(1) O(n)$	$\Omega(1) O(n)$	$\Omega(1) O(N/K)$
		supprimer	$\Omega(1) O(n)$	$\Omega(1) O(n)$	$\Omega(1) O(N/K)$
	Tas	obtenirMax	$\Omega(1) O(1)$	$\Omega(1) O(1)$	
		ajouter	$\Omega(1) O(\log_2(n))$	$\Omega(1) O(\log_2(n))$	
		supprimer	$\Omega(1) O(\log_2(n))$	$\Omega(1) O(\log_2(n))$	
	Arbre binaire	obtenirElement (racine)		$\Omega(1) O(1)$	
		ajouter (racine)		$\Omega(1) O(1)$	
		supprimer (racine)		$\Omega(1) O(1)$	
	Arbre binaire de recherche	rechercher		$\Omega(1) O(\log_2(n))$	
		insérer		$\Omega(1) O(\log_2(n))$	
		supprimer		$\Omega(1) O(\log_2(n))$	
Arbre n-aire	obtenirElement (racine)			$\Omega(1) O(1)$	
	ajouter (racine)			$\Omega(1) O(1)$	
	supprimer (racine)			$\Omega(1) O(1)$	

Utilisation des algorithmes d'insertions et suppressions AVL ou Rouge et Noir. Le type des éléments doit donc avoir un ordre total
Les éléments (ou une partie des éléments) doivent être hachable. K représente la taille du tableau. On considère la complexité dans le pire des cas de la fonction de hachage en $O(1)$.

