Interface PM
IFT2905
Fuille de notes

Franz Girardin

21 février 2024

Table des matières

Section 1

Introduction aux structures de données

1.1 Caractéristique génériques de 𝕊𝔻

Comment décrire une structure de données

- ▷ **Linéaire ou *non* linéaire**
  - ► P. ex. `graph` vs. `arrays`
- ▷ Homogène ou non homogène
  - ► Indique si les données sont mê `type`
- ▷ Statique ou *dynamique*
  - ► P. ex. taille modifiable ou `fixe`.

Note :

Plusieurs problèmes ont une complexité telle que les 𝕊𝔻 **base** sont inappropriées, d'où la nécessité des 𝕊𝔻 **abstraites**, plus adéquates.

𝕊𝔻 Abstaites Elles `généralise` les types de données de base et permettent l'émergence de concept sophistiqués d'**abstraction**, d'**encapsulation** et de typage. Elle possèdent **5 caractéristiques** fondamentales (𝕋𝕌𝕆ℙ𝔸) :

- ▷ Type
- ▷ Utilise
- ▷ Opérations
- ▷ Procédures
- ▷ Axiomes

1.2 Tableau

Stocke `E` dans emplacement de mémoire contigus.

- ▷ De longueur fixe ou variable
  - ► Autrefois : adresse brutes
  - ► Maintenant : N `arrays` (indexation)
  - ► Maintenant : `records` (champs)

1.3 Pile

La propriété fondamentale d'une pile est que le seul éléement accessible et celui se trouvant au sommet de la pile. La **pile** suit le principe **LIFO** : *last in, first out.*

1.4 File

La **file** suit le principe **FIFO** : *first in, first out.* Elle possède, entre autres, les opération enfiler *enqueue* et défiler *dequeue.*

1.5 Liste

Une file est le résultat de l'*ordonancement* d'un nombre dénombrable de données où la même valeur peut apparaître plusieurs fois.

- ▷ **Implémentation**
  - ► Liste chaînée `Linked List`
  - ► Liste doubleemnt chaînée `Linked List`
  - ► Liste circulaire
  - ► Tableau `array`

Note :

Une liste chaînée stocke un ensemble d'éléments de façon linéaire. Chaque élément ou nœud d'une liste chaînée contient un **élément de données** ainsi qu'une **référence**, ou lien, vers l'élément suivant de la liste.

1.6 Arbre

Un arbre stocke un ensemble d'éléments sous une forme **hiérarchique abstraite**. Chaque nœud est relié aux autres et peut contenir plusieurs sous-valeurs appelées enfants. Il s'agit d'un `graphe` avec **3 particularité fondamentales** :

- ▷ Acyclique
- ▷ Connexe
- ▷ Possède une seule racine

1.7 File de priorité

Possède les mêmes propriété de base qu'un file, sauf que chaque élément a en plus un poids de priorité qui détermine l'ordre global des éléments.

1.8 Graphe

Un graphe stocke un ensemble d'éléments de façon non linéaire. Il se compose d'un ensemble fini de nœuds, appelés **sommets**, et de lignes, les arêtes, qui relient les sommets entre eux. Les graphes permettent notamment de représenter des systèmes réels, comme des réseaux informatiques.

Théorème des quatre couleurs Il existe une 4-coloration de n'importe qu'elle pays qui a été découpé sur une carte.

Section 2

Pile

Exemples d'applications

- ▷ Historique de pages visitées
- ▷ Annuler une séquence de texte dans un éditeur
- ▷ Chaîne de méthodes appelées d'un fonction complexe

5 Opération fondamentales

- ▷ Créer un pile
- ▷ Empiler
- ▷ Dépiler
- ▷ Regarder le sommet
- ▷ Obtenir le nombre d'éléments

```
elements[0...n-1] <-- tableau // tableau taille n
// top <-- 0 pour assigner 0 Ã  top on fait :

push(E) :
element[top] <-- E
top <-- top + 1 // modifie l'index du sommet


//  Pour obtenir l'E au sommet, on fait :

pop() :
retourne element[top]
top <-- top -1 //decremente la valeur du sommet
x <-- element[top + 1] // enregistre val sommet
element[top + 1] <-- null
```

Conventions d'écriture

- ▷ **IN**fixé : `gauche racine droite`
  - ► $5 - 6\times7 \leftrightarrow (5-6)\times7$
- ▷ **POST**fixé : `gauche droite racine`
  - ► $56-7\times \leftrightarrow (56-)\ 7\times$
- ▷ **PRÉ**fixé : `racine gauche droite`
  - ► $\times-567\times \leftrightarrow \times(-56)\ 7$

Avantage de RPN

- ▷ Aucune ambiguité
- ▷ Parenthèses sont superflus
- ▷ Suit la méthodologie calcul des ordinateurs
- ▷ Peut être implémentée par une pile

Méthodtologie RPN pour `parse` une opératon

1. Emplier chaque **opérande** qu'on rencontre dans `valStack`
2. Process quand on rencontre un **opérateur** :
   - ▷ `pop` les deux dernières valeurs de la `valStack`
   - ▷ Effectuer l'opération
   - ▷ `push` le résultat dans la pile

```
doOp()
x = valStack.pop()
y = valStack.pop()
op = opStack.pop()
valStack.push(y op x)

repeatOps(reOp)
tant que (valStack.size > 1) et prec(refOp) < prec(opStack.top()) faire
   doOp()
 fin tant que

EvalExpr()
// Entree : flux de jetons representant expression arithmÃ©tique
// Sortie : valeur calculee de l'expression

tant que il y a un autre jeton z faire :
   si z est un nombre alors valStack.push(z)
   sinon
       repeatOps(z)
       opstack.push(z)
 fin tant que

 // Noter que $ est l'opÃ©rateur ayant la plus petite precedence
 repeatOps($)
 return valStack.top()
```

Section 3

Listes

Définition 1

Un **type de données abstrait** (ADT) est une abstraction qui définit les *opérations* sur les *données* (qui sont physiquement stockées dans une structure de données)

Les ADT spécifient :

- ▷ Données
- ▷ Opération sur les données
- ▷ Cond. d'erreurs associées aux ops.

6 Opération fondamentales des listes

- ▷ Créer une liste
- ▷ Ajouter un `E` à la liste
- ▷ Retirer un `E` de la liste
- ▷ Vérifier la taille de la liste
- ▷ Remplacer un élément par un autre
- ▷ Retrouver un élément

Rappel
Les k premiers termes d'une **série arithmétique** :

$$S_n = \sum_{k=0}^{n} k = \frac{n(n+1)}{2}$$
$$r = a_n - a_{n-1} \ \forall\, n \geq 2$$
$$a_1 + (n-1)r \ \ \forall\, n \geq 1$$

La k premiers termes d'une **série géométrique**

$$S_n = \sum_{k=0}^{n} 2k = \frac{1-2^{n+1}}{1-2} = 2^{n+1} - 1$$
$$S_n = \frac{a(r^{n+1}-1)}{r-1}$$
$$r = \frac{a_n}{a_{n-1}} \ \forall\, n \geq 2$$
$$a_1 + a_1 r^{n-1} \ \ \forall\, n \geq 1$$

Opération d'insertion Soit un tableau `A` de n éléments implémenté par un `array` indexé de `0` à `n - 1`. Et soit n la valeur de l'index pointant vers la prochaine **case vide**. Les caractéristiques de l'**opération d'insertion** sont les suivantes :

- ▷ `add(i, x)`
  - ▶ **Meilleur temps** : `i = n`
  - ▷ Déplace uniquement `n`
  - ▷ `n <-- n + 1`
  - ▷ **Temps constant** $O(1)$
- ▷ `add(i, x)`
  - ▶ **Pire temps** : `i = 0`
  - ▷ Déplace à droite toute valeur d'index $i \geq 0$
  - ▷ `pour Chaque i<= dex <= n faire`
    `A[dex+1] <-- A[dex]`
  - ▷ **Temps linéaire** $O(n)$

En moyenne, le temps pour chaque opération de déplacement est

$$\frac{1}{n}\sum_{i=0}^{n} i = \frac{n(n+1)}{2n} = \frac{n+1}{2} \implies O(n)$$

Opération de suppression Soit un tableau `A` de n éléments implémenté par un `array` indexé de `0` à `n - 1`. Et soit n la valeur de l'index pointant vers la prochaine **case vide**. Les caractéristiques de l'**opération de suppresion** sont les suivantes :

- ▷ `rem(i, x)`
  - ▶ **Meilleur temps** : `i = n -1`
  - ▷ Déplace uniquement `null`
  - ▷ `n <-- n - 1`
  - ▷ `A[n] <-- null`
  - ▷ **Temps constant** $O(1)$
- ▷ `add(i, x)`
  - ▶ **Pire temps** : `i = 0`
  - ▷ Déplace à gauche toute valeur d'index concernée
  - ▷ `pour Chaque i<= dex <= n - A[dex] <-- A[dex + 1]`
  - ▷ **Temps linéaire** $O(n)$

En moyenne, le temps pour chaque opération de déplacement est

$$\frac{1}{n}\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2n} = \frac{n-1}{2} \implies O(n)$$

Performance

- — Complexité spaciale de la crétion du tableau : $O(n)$
- — L'idexation d'un `E` : $O(1)$
- — Opération `add` et `remove` : $O(n)$

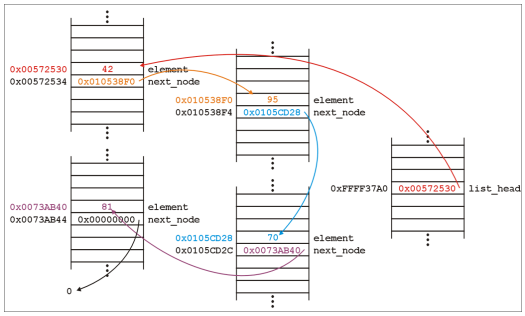
Agradir un tableau dynamique

- ▷ Crée un nouveau `tab` de `size + 1`
- ▷ Copier les n premiers éléments
- ▷ Ajouter le nouvel élément à l'index $n-1$ (Optionnel) du nouveau `tab`
- ▷ Réaffecter la `ref` de l'ancient `tab` au nouveau `tab`

```
Algorithme add(x)
// Si aucune E supprime, n aura
// meme valeur que size
si n = tab1.size() faire :
  Creer nouveau tab de taille voulue
  pour chaque 0 <= dex n - 1 faire :
      tab2[i] <-- tab1[i]
  tab1 <-- tab2
  tab1[n] <-- x
  n <-- n + 1
```

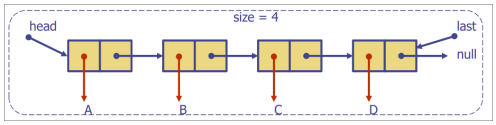
3.1 Liste chaîné

La distinction fondamentale entre une liste simple et une liste chaînée est, dans le cas de la liste chaîné, l'existence d'une référence **indiquant l'emplacement mémoire** du prochain élément de la liste. Et, puisque les `E` ne sont pas contigus en mémoire, la taille d'une liste chaînée n'a pas unique limite que la quantité de mémoire disponible sur l'ordinateur.



Définition 2 Liste S. chaînée

Une **liste simplement chaînée** est une structure de données concrète constituée d'une *séquence de nœuds* à partir d'une référence vers le **noeud de tête**, où chaque nœud contient deux références : vers un `E` et vers le **nœud suivant**. On gardera aussi une référence sur le dernier nœud et un entier pour le nombre d'éléments dans la liste.




```
initialisation(varElement, varNext)
element <-- varElement // Assigne E
next <-- varNext  // specifie proch
```

```
Class LinkList(List)
initialisation()
head <-- null
next <-- null
size <-- 0

fonction len()
   retourner size
```

```
fonction str()
  Si isEmpty()
    retourner "[] (size = 0)"
  Si non :
  PP <- "["
  curr <- head
  tant que curr != null faire
    // (concat)
    PP <- PP + curr.next + " "
    curr <- curr.next

  Fin tant que

  PP <- PP + "]"
  PP <- PP + "(size = " + size +")"

fonction isEmpty()
    retourner size = 0
```

3.2 Liste doublement chaînée

Définition 3 Liste D. chaînée

Une **liste doublement chaînée** est une structure de données constituée d'une *séquence de nœuds* à partir de références vers les nœuds de tête et de queue, où chaque nœud contient **trois références** : vers un `E` , vers le **nœud suivant** et vers le **nœud précédent**. On gardera aussi un entier pour le nombre d'éléments dans la liste.

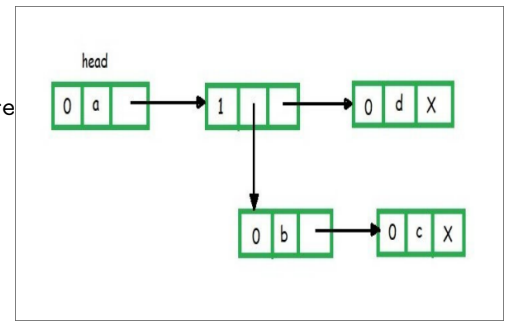
```
Class DoublyLinkedNode()

// Methode generale d'initialisation
initialisation(varE, varPrev, varNext)
  element <-- varE
  prev <-- varPrev
  next <-- varNext

// Initialisation pour
// engendrer la fonction str()
```

```
initialisation()
  head <-- DoublyLinkedNote(null, null,
  tail <-- DoublyLinkedNode(null, null,
  head.next <-- tail
  tail.prev <-- head
  size <-- 0
```

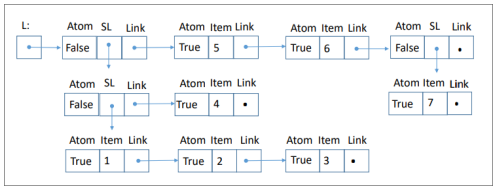
```
fonction len()
  retourner size

fonction str()
  Si isEmpty()
    retourner
    "[] (size = 0)"
  Sinon :
    PP <-- "["
    curr <-- head.next

    tant que curr.next != tail faire
      PP <-- PP + curr.element + ""
      curr <-- curr.next
    Fin tant que

    PP <-- PP + "]" +
    "size = (" + size + ")"
  Fin si

fonction isEmpty()
  retourner size = 0
```

Note :

On utilise **0** pour représenter un atome et **1** pour représenter une sous-liste (Parfois T/F)



Liste généralisée 2



Section 4

Liste généralisée

Définition 4 Liste G.

Liste dans laquelle les éléments peuvent être des sous-listes Si un E de la liste **n'est pas** une sous-liste, il est dit atomique. Les liste généralisées peuvent être représentées par des représentations séquentielles ou chaînées.

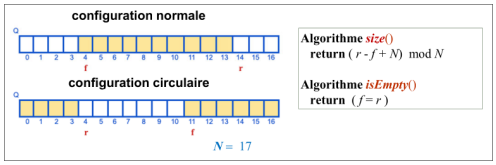
```
(a_1, a_2,
  (b_1,
    (c_1, c_2),
  b_3),
a_4,
  (d_1, d_2)
a_6)
```

Section 5

File

Pour l'implémentation, on utilise un tableau circulaire de taille N, ainsi que *deux variables* qui gardent trace de l'**avant** et l'**arrière** : f et r.

- ▷ f index directement sur la valeur
- ▷ r index 1ere case vide après valeur.




```
// Algorithme size()
retourne (r - f + N) mod N

// Algorithme isEmpty()
retourn (f = r)


// Algorithme Enqueue(x)
Si size() = N - 1 faire :
  retourne Erreur
Sinon :
  Q[r] <-- x
  r <-- (r+1) mod N


// Algorithme Dequeue()
Si Q iseEmpty() faire :
  retourne erreur
Sinon :
  x <-- Q[f]
  f <-- (f + 1) mod N
  retourne x
```

Variante

- ▷ f index directement sur la valeur
- ▷ r index directement sur case fin.

```
 // Algorithme size()
retourne ((r - f + N) mod N) + 1

// Algorithme isEmpty()
retourne (f = (r + 1) mod N)

// Algorithme Enqueue(x)
Si ((r + 1) mod N) = f alors
  retourne Erreur // La file est pleine
Sinon :
  Q[r] <-- x
  r <-- (r + 1) mod N

// Algorithme Dequeue()
Si isEmpty() alors
  retourne erreur
Sinon :
  x <-- Q[f]
  f <-- (f + 1) mod N
  retourne x
```