Algorithmiques et Structures de données 01 Cours + TDs + TPs

Version 1.0.0

Mr. Ismail HADJADJ

Octobre 2013

Remarque :
Ce document n'est pas révisé, vous pouvez trouver des erreurs d'orthographes, de saisies et autres.

Table des matières

Chapitre 1

Généralités

1.1 Introduction

pour résoudre un problème donnée par l'informatique l'utilisateur de l'ordinateur doit mettre au point un programme et l'exécution par la machine
Un programme est une succession logique et ordonnée d'instructions
Pour écrire un programme il faut :

- Bien connaître le problème ;
- S'avoir le découper logiquement en un ensemble d'opérations élémentaires(actions) ;
- Connaître un langue de programmation.

problème $\longrightarrow$ Algorithme $\longrightarrow$ programme

1.2 Algorithme(Définition)

Un algorithme est une "Spécification d'un schéma de calcule sous forme d'une suite finie d'opération élémentaire à un enchaînement déterminé".
Ou encore :
La description des étapes à suivre pour réaliser un travail.

Un **programme** est (donc) la description d'un algorithme dans un langage de programmation.
Remarque :
Un algorithme est indépendant du langage de programmation(Donc la machine).

1.3 Structure d'un algorithme :

La structure d'un algorithme est donc :

Algorithme 1: NomDeLAlgorithme

Const : const1,cont2,....=
Var : var1,var2... :type

Procédure P1(....)

début
...
...
...
fin

Fonction f1(....) :type

début
...
...
...
fin

début
...
$P1(...)$
$var1 \leftarrow f1(...)$
...
fin

1.4 Sous-Programmes

Une **fonction** est un algorithme indépendant, l'appel de la fonction déclenche l'exécution de son bloc d'instructions. Une fonction se termine entourant ou non une valeur.
Un procédure est une fonction qui retourne vide.

1.4.1 Passage de paramètre

Par valeur

On dit qu'un procédure fait un passage par valeur, s'il ne change pas la valeur de variable passé.

Par référence

Il s'agit de modification du valeur du variable passé.

Exemples

Procédure P1(x :entier)

```
Var : i :entier
début
    pour i=1 à 10 faire
        x ← x * x
    finpour
fin
```

Procédure P1(var x :entier)

```
Var : i :entier
début
    pour i=1 à 10 faire
        x ← x * x
    finpour
fin
```

1.5 L'indécidabilité de terminaison

Supposons que le problème de l'arrêt soit décidable
$\Rightarrow \exists$ une fonction B qui décide si un programme s'arrête
$\Rightarrow B(X : programme) = true$
$B(X2 : programme) = false$

Algorithme 2: X

```
Var : y :entier
début
    y ← 1
    Ecrire(y)
fin
```

Algorithme 3: X2

```
Var : y :entier
début
    y ← 1
    tant que y>0 faire
        y ← y + 1
    fintq
    Ecrire(y)
fin
```

Proposons l'algorithme :

Algorithme 4: C

début
  tant que *B(C)* **faire**
  fintq
  Ecrire(y)
fin

Si C boucle indifiniment, alors $B(C) = false \Rightarrow C$ termine
Sinon $B(C) = true \Rightarrow C$ boucle infiniment
Alors Contradiction

Notez bien :
Dans ce cours , On étudiera seulement des problèmes pour lesquels il existe des algorithmes décidables.

1.6 Conception d'un algorithme

1.6.1 Analyse descendent

Décomposer le problème en sous problèmes.

1.6.2 Analyse ascendante

Utiliser les fonctions, primitives, outils,.. dont on dispose, les assembler pour en faire un truc qui résoudre notre problème.

1.6.3 mélange des deux

On fait une analyse descendante tout en ayant à l'esprit les module bien conçus qui existent déjà.

1.7 Conclusion

dans ce cours on a introduire des notions de base liées à l'algorithmique.

1.8 Exercices

Chapitre 2

Complexité d'Algorithmes

2.1 Introduction

Il existe souvent plusieurs algorithmes permettant de résoudre un même problème. Exemple : les algorithmes de tri. Le choix du meilleur algorithme implique une analyse de ses performances. En général, le critère le plus important est celui du temps nécessaire à son exécution. Celui ci dépend le plus souvent de la quantité de données à traiter. Par exemple, le temps nécessaire pour trier un ensemble d'objets dépend du nombre d'objets.

2.2 Qualité d'un algorithme

2.2.1 Qualité d'écriture

un algorithme doit être structuré, indenté, modulaire, avec des commentaires pertinents, etc. Il faut pouvoir comprendre la structure d'un coup d'oeil rapide, et pouvoir aussi revenir dessus 6 mois plus tard et le comprendre encore.

2.2.2 Terminaison

le résultat doit être atteint en un nombre fini d'étapes. Il ne faut donc pas de boucles infinies, il faut étudier tous les cas possibles de données.

2.2.3 Validité

le résultat doit répondre au problème demande. Attention, un jeu d'essais ne prouve **jamais** qu'un programme est correct. Il peut seulement prouver qu'il est faux.

2.2.4 Performance

Etude du coût (complexité) en temps et en mémoire. On s'intéresse dans ce cours qu'au complexité temporelle.

2.3 Complexité d'un algorithme

La complexité d'un algorithme est une estimation du nombre d'opérations de base effectuées par l'algorithme en fonction de la taille des données en entrées de l'algorithme.

La Complexitéd'un algorithme c'est pour :

1. Evaluer les ressources(mémoire CPU)utiles,
2. Comparer deux algorithmes pour le même problème,
3. donner un borne sur ce qui est effectivement possible de résoudre. On considère aujourd'hui qu'on peut réaliser en temps raisonnable de 2^{60} opérations, quand à la mémoire est de l'ordre de 10^{10} octets.

2.4 Mésure de complexité

Pour mesurer la complexité temporelle d'un algorithme, on s'intéresse plutôt aux opérations le plus coûteuses pour le problème de calcul particulier ;

- Racine carrée, Log, Exp, Addition réelle ;
- Comparaisons dans le cas des tris ...

et on calcule le nombre d'opérations fondamentales exécutées par l'algorithme.

Le temps de l'exécution dépend de la taille de l'entrée. On veut considère seulement la taille essentielle de l'entrée. Cela peut être par exemple :

- le nombre d'éléments combinatoires dans l'entrée,
- le nombre de bits pour représenter l'entrée,
- ... etc ...

2.5 Complexité moyenne et en pire des cas

Soi n la taille des données du problème et $T(n)$ le temps d'exécution de l'algorithme. On distingue :

2.5.1 Temps le plus mouvais(en pire des cas)

$T_{max}(n)$ qui correspond au temps maximum pris par l'algorithme pour un problème de taille n ;

2.5.2 Temps moyenne

$T_{moy}(n)$ temps moyen d'exécution sur de données de taille n (Supposition sur la distribution de données).

Soit A un Algorithme
D_n l'ensemble des entrées de taille n
$I \in D_n$ une entrée

1. $Cout_A(i)$ = nombre d'opération fondamentales exécutées par A sur I.
2. La complexité de A en pire des cas :

$$T_{max}(n) = Max(Cout_A(i)); I \in D_n$$

3. la complexité de A en moyenne :

$$T_{moy}(n) = \sum_{i \in D} Pr[i] * Cout_A[i]$$

Exemple

Recherche séquentielle :

Fonction RechercheS(tab :Tableau de n éléments ;x :élément) :entier

Var : j :entier
début
  $j \leftarrow 1$
  tant que $(j \leq n) et (tab[j] <> x)$ **faire**
    $j \leftarrow j + 1$
  fintq
  si $j > n$ **alors**
    $RechercheS \leftarrow -1$
  sinon
    $RechercheS \leftarrow j$
  finsi
fin

Comlexité en pire des cas :

$$T_{max}(RS) = n$$

Complexité moyenne :
-$Pr[x \in tab] = q$
- Si $x \in tab$ alors tous les places son équiprobables pour $1 \leq i \leq n$ soit :

$$I_i = x \in tab$$

et

$$I_{n+10} = x \notin tab$$

On a :
$Pr[I_i] = q/n$ pour $1 \leq i \leq n$ et $Cout_RS(I_i) = i$
et
$Pr[I_{n+1}] = 1 - q$ et $Cout_RS(I_{n+1}) = n$

$$\begin{aligned} T_{moy}(RS) &= \sum_{j=1}^{n+1} Pr[I_j] * Cout_RS(I_j) \\ &= \sum_{i=1}^{n} q/n(i) + (1-q)n \\ &= q/n \sum_{i=1}^{n} i + (1-q)n \\ &= q/n * n(n+1)/2 + (1-q)n \\ &= (1 - q/2)n + q/2 \end{aligned}$$

-Si $q = 1$ alors $T_{moy}(RS) = (n+1)/2$
-Si $q = 1/2$ alors $T_{moy}(RS) = (3n+1)/4$

2.6 Notation utilisées

Il faut comparer les taux d'accroissement de différentes fonctions qui mesurent les performances d'un programme.

2.6.1 Notation "o"

On dit que $f(x) = o(g(x))$ pour $x \to \infty$ si

$$\lim_{\longrightarrow} \frac{f(x)}{g(x)} = 0$$

$\lim_{x\to\infty}$ Ce que veut dire que f croît plus lentement que g quand x est très grand. Par exemple :
$x^2 = o(x^5)$
$\sin(x) = o(x)$
$14.709\sqrt{x} = o(x/2 + 7\cos(x))$
$23\log(x) = x^{0.002}$

2.6.2 Notation "O"

On dit que$f(x) = O(g(x))$ s'il existe k et x_0 tels que :

$$\forall x > x_0 \Rightarrow f(x) < kg(x)$$

La notation o est plus précise que O, mais O est plus facile à calculer et suffisant. Par exemple :
$\sin(x) = O(x)$
$\sin(x) = O(1)$

2.7 Un classement des fonctions

Problèmes rationnellement facile :

Groupe01 : $\log(\log(x)), \log(x), \log^2(x)$
Groupe02 : $x^{0.2}, x, x^2, x^{15}\log(x)$

Problèmes difficiles :

Groupe 03 :$e^{\sqrt{x}}, 1.03^x, 2^x$
Groupe 04 :$x!, x^x, x^{x^2}$

2.8 Exercices

Rappele :Notation "O"
On dit que$f(x) = O(g(x))$ s'il existe k et x_0 tels que :

$$\forall x > x_0 \Rightarrow f(x) < kg(x)$$

2.8.1 Exercice 01

1. Parmi les fonctions suivantes, quelles sont celles qui ont le même ordre de grandeur ?
   - $f1(n) = 4n^3 + n$
   - $f2(n) = n^2 + log_2(n)$
   - $f3(n) = n^2 \times log_3(n) + 6n^3$
   - $f4(n) = \dfrac{n(n^2+1)}{2}$
2. En utilisant la définition de O montrer que $f(n) + g(n) = O(max(f(n); g(n)))$
3. En utilisant la définition de O montrer que $3n^2 + 4n + 6 = O(n^2)$

2.8.2 Exercice 02

Considérer les algorithmes suivantes avec un temps d'exécution $T(n)$ pour une longueur de données n. Déterminer leur complexités asymptotiques respectives, et les classez par ordre de grandeur croissant.

Algorithme A1 $T(n) = 3n + 2$
Algorithme A2 $T(n) = 6$
Algorithme A3 $T(n) = 4n2 + n + 2$
Algorithme A4

Exécuter A1 ;
Exécuter A2 ;
Exécuter A3 ;

Algorithme A5

Algorithme 5: A5

```
début
    pour i ← 1 à n faire
        Exécuter A3
    finpour
    Exécuter A1
fin
```

Algorithme A6

Algorithme 6: A6

début
  pour $i \leftarrow 1$ **à** 5 **faire**
    Exécuter A1
  finpour
fin

2.8.3 Exercice 03

Evaluer les complexités des fonctions et des programmes suivants en donnant le nombre d'instructions réalisées.

Algorithme 7: A1

Var : a,b :entier
début
  $a \leftarrow 0$
  $b \leftarrow 0$
  $Affiche r(a)$
  $Affiche r(b)$
fin

Fonction somme(n :entier) :entier

Var : i,res :entier
début
  $res \leftarrow 0$
  pour $i \leftarrow 1$ **à** n **faire**
    $res \leftarrow res + i$
  finpour
  $somme \leftarrow res$
fin

Fonction f1(n :entier) :entier

Var : i :entier
début
  pour $i \leftarrow 1$ **à** n **faire**
    $somme(n)$
  finpour
fin

Fonction f2(n :entier) :entier

Var : i :entier
début
  pour $i \leftarrow 1$ **à** n **faire**
    $somme(i)$
  finpour
fin

```
Fonction f3(n :entier) :entier
  Var : i,j,k :entier
  début
  |  pour i ← 1 à n faire
  |  |  pour j ← i à n faire
  |  |  |  pour i ← 1 à j faire
  |  |  |  |  inst()
  |  |  |  finpour
  |  |  finpour
  |  finpour
  fin
```

2.8.4 Exercice 04

Calculer la complexité de la fonction récursive :

```
Fonction Fib(n :entier) :entier
```

vue en cours.

2.8.5 Exercice 05

É tudiez le nombre d'additions réalisées par les fonctions suivantes dans le meilleur cas, le pire cas, puis dans le cas moyen en supposant que les tests ont une probabilité de $\frac{1}{2}$ d'être vrai.

```
Fonction f1(t :tab de n er
  Var : s :entier
  début
  |  s ← 0
  |  pour i ← 1 à n faire
  |  |  si t[i] > a alors
  |  |  |  s ← s + t[i]
  |  |  finsi
  |  finpour
  |  f1 ← s
  fin
```

```
Fonction f1(a,b :entier) :entier
  Var : s :entier
  début
  |  s ← 0
  |  si a > b alors
  |  |  pour i ← 1 à n faire
  |  |  |  s ← s + a
  |  |  finpour
  |  sinon
  |  |  s ← s + b
  |  finsi
  |  f1 ← s
  fin
```

Chapitre 3

La récursivité

3.1 Introduction

la récurrence ("induction") est un outil mathématique essentiel. Tout informaticien se doit de la maîtriser. Heureusement, la récurrence est une technique simple, en dépit de sa puissance.
La récursivité est une notion très utilisée en programmation, et qui permet l'expression d'algorithmes concis, faciles à écrire et à comprendre. La récursivité peut toujours être remplacée par son équivalent sous forme d'itérations, mais au détriment d'algorithmes plus complexes surtout lorsque les structures de données à traiter sont elles-mêmes de nature récursive.

3.2 Définition

- Un algorithme (fonction, procédure) est dite récursif lorsqu'il s'appelle lui-même, c-à-d., définition (son code) contient un appel à lui-même.
- Un algorithme qui n'est pas récursif est dit itératif.

Notez Bien

- On peut toujours transformer un algorithme récursif en algorithme itératif.

la forme générale d'un algorithme récursive :

Algorithme 8: P(...)

```
début
    si fin() alors
        ....
        //pas d'appel récursif
    sinon
        ...
        P(...) //l'algorithme P appelle lui-même une

        ... //ou plusieurs fois

    finsi
fin
```

3.3 Exemples

3.3.1 Exemple1

Calcul de la factorielle d'un nombre.
Définition itérative :
$n! = F(n) = n * (n-1) * (n-2) * ... * 2 * 1$
Soit en algorithmique :

Algorithme 9: factorielle(n :entier) :entier

```
Var : i,f :entier
début
    f ← 1
    pour i=1 à n faire
        f ← f * i
    finpour
    factorielle ← f
fin
```

Définition récursive :
$F(0) = 1$
$F(n) = n * F(n-1)$; Soit en algorithmique :

Algorithme 10: factRecursive(n :entier) :entier

```
début
    si n = 0 alors
        factorielle ← 1
    sinon
        factorielle ← factorielle(n − 1) * n
    finsi
fin
```

3.3.2 Exemple2 :

La suite des nombres de Fibonacci se définit comme suit :
$f0 = 0$
$f_1 = 1$
$f_n = f_{n-1} + f_{n-2}$ si $n > 1$
On peut formuler cette suite sous forme de fonction (n >= 0) :
$fibonacci(n) = n$ si $n <= 1$
$fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)$ si $n > 1$
soit en Algorithmique :

Algorithme 11: fibonacciRec(n :entier) :entier

```
début
    si n <= 1 alors
        fibonacci ← n
    sinon
        fibonacci ← fibonacci(n − 1) + fibonacci(n − 2)
    finsi
fin
```

3.4 Règles

1. Tout algorithme récursif doit distinguer plusieurs cas, dont l'un au moins ne doit pas comporter d'appel récursif.
   - Les cas non récursifs d'un algorithme récursif sont appelés cas de base.
   - Les conditions que doivent satisfaire les données dans ces cas de base sont appelées conditions de terminaison.
2. Tout appel récursif doit se faire avec des données plus proches de données satisfaisant une condition de terminaison.

3.5 Récursivité croisée

Dans la récursivité croisée, les appels récursifs sont provoqués par l'exécution d'autres procédure ou fonction.

Procédure P1(x)

début
  ...
  $Q(f(x))$
  ...
fin

Procédure Q1(x)

début
  ...
  $P(g(x))$
  ...
fin

3.5.1 Exemple : Paire ou impaire

On peut formuler les notions récursives suivant :
- Le 0 est paire et non impaire.
- Un nombre n est paire si n-1 est impaire.
- Un nombre n est impaire si n-1 est paire.

Soit en algorithmique :

Fonction Paire(n :entier) :entier

début
  si $n = 0$ **alors**
    $Paire \leftarrow vrai$
  sinon
    $Paire \leftarrow ImPaire(n-1)$
  finsi
fin

FIGURE 3.1 – Position de départ des tours d'Hanoi

Fonction ImPaire(n :entier) :entier

début
  si $n = 0$ **alors**
    $ImPaire \leftarrow false$
  sinon
    $ImPaire \leftarrow Paire(n-1)$
  finsi
fin

3.6 Problème(les tours d'Hanoï)

Le titre du problème vient de l'histoire racontée habituellement, qui est celle de moines bouddhistes en Asie du sud-est qui égrènent le temps en transférant des disques, tous de tailles différentes, sur un jeu de trois piquets (voir figure 1). Dans cette figure il n'y a que cinq disques, mais la tradition veut que les moines jouent avec 64. L'histoire est une invention du dix-neuvième siècle, Lucas la plaçantà Bénarès (en Inde). On ne sait pas comment elle s'est restituée à Hanoï . . .

Le jeu consiste à transférer la pile de disques du piquet A vers le piquet B, en utilisant C comme piquet de manoeuvre, tout en respectant les deux règles suivantes :

– un disque ne peut pas être posé sur plus petit que lui,
– on ne déplace qu'un disque à la fois.

La solution la plus simple vient de la réponse à la question suivante : "si je savais transférer n-l disques, saurais-je transférer n ?". La réponse est oui, car autrement nous n'aurions pas posé la question. Ainsi, pour transférer n disques de A à B, on commence par le transfert de n-l disques de A à C, suivi du déplacement du dernier disque (le plus gros) de A à B, suivi du transfert des n-l disques de C à B.

Cela donne la procédure :

Procédure hanoi(n :entier ; a,b :entier)

début
  si $n = 1$ **alors**
    $deplacement(a, b)$
  sinon
    $hanoi(n-1, a, 6-a-b)$
    $deplacement(a, b)$
    $hanoi(n-1, 6-a-b, b)$
  finsi
fin

Dans cette procédure, nous avons supposé que les piquets sont numérotés 1,2,3. Déplacer un disque se fait par la procédure déplacement. Pour $n = l$ le déplacement est immédiat, autrement on applique l'algorithme décrit ci-dessus, 6-a-b étant le numéro du troisième piquet (6=1+2+3, donc en soustrayant deux des trois possibilités de 6 on obtient la troisième).

3.7 Conclusion

Certains problèmes peuvent être résolus plus logiquement en utilisant la récursivité. Les programmes sont plus compacts, plus faciles à écrire et à comprendre. Son usage est naturel quand le problème à traiter peut se décomposer en deux ou plus sous-problèmes identiques au problème initial mais avec des valeurs de paramètres différentes. Refuser la récursivité dans ce dernier cas oblige l'utilisateur à gérer lui-même une pile des différentes valeurs des variables, ce que le système fait automatiquement lors de l'utilisation de la récursivité.

3.8 Exercices

3.8.1 Exercice 01

Ecrire une fonction récursive qui calcule le carré d'un entier $n^2 = n \times n$:
Fonction carré (n : Entier) : Entier
On pourra utiliser la relation suivante : $(n+1)^2 = n^2 + 2n + 1$.

3.8.2 Exercice 02

Ecrire une fonction récursive qui prend en argument une chaîne de caractères et qui renvoie un booléen indiquant s'il s'agit d'un palindrome ou non.

3.8.3 Exercice 03

On définit la fonction suivante :

Fonction McCarty(n : Entier) : Entier

début
  si $n > 100$ **alors**
    $McCarthy \leftarrow n - 10$
  sinon
    $McCarthy \leftarrow McCarthy(McCarthy(n + 11))$
  finsi
fin

1. Quelle est la valeur de McCarthy(n) pour n > 100 ?
2. Calculer McCarthy(98), McCarthy(99) et McCarthy(100).
3. En déduire la valeur de McCarthy(n) pour n - 100. Expliquer.

3.8.4 Exercice 04

Ecrire une procédure récursive qui représente le déplacement d'un cavalier sur un échiquier à partir de la position (X_0, Y_0).

3.8.5 Exercice 05

Soit une fonction continue f définie sur un intervalle $[a; b]$. On cherche à trouver un zéro de f, c'est-à-dire un réel $x \in [a; b]$ tel que $f(x) = 0$. Si la fonction admet plusieurs zéros, n'importe lequel fera l'affaire. S'il n'y en a pas, il faudra le signaler.

Dans le cas où $f(a).f(b) < 0$, on est sûr de la présence d'un zéro. Lorsque $f(a).f(b) > 0$, il faut rechercher un sous-intervalle $[\alpha; \beta]$, tel que $f(\alpha).f(\beta) < 0$.

L'algorithme procède par dichotomie, c'est-à-dire qu'il va diviser l'intervalle de recherche en deux moitiés à chaque étape. Si l'un des deux nouveaux intervalles, par exemple $[\alpha; \beta]$, est tel que $f(\alpha).f(\beta) < 0$, on sait qu'il contient un zéro puisque la fonction est continue : on poursuivra alors la recherche dans cet intervalle.

En revanche, si les deux demi intervalles sont tels que f a le même signe aux deux extrémités, la solution, si elle existe, sera dans l'un ou l'autre de ces deux demi intervalles. Dans ce cas, on prendra arbitrairement l'un des deux demi intervalles pour continuer la recherche ; en cas d'échec on reprendra le deuxième demi intervalle qui avait été provisoirement négligé.

écrivez de façon récursive l'algorithme de recherche d'un zéro, à ε près, de la fonction f.

3.9 TP

L'objective de ce TP et d'expérimenter la récursivité et la complexité des algorithme

Notez bien :

– Un **rapport(3 page max) et code source** doit être remis avant le :.....

3.9.1 Recherche un élément dans un tableau

La recherche compare les différents éléments d'un tableau à la clef reherchée.

exemple :

La valeur recherchée est 7.

La case colorée est l'élément en cours de traitement.

8	5	2	7	1	3	0	4
8	**5**	2	7	1	3	0	4
8	5	**2**	7	1	3	0	4
8	5	2	**7**	1	3	0	4

TABLE 3.1 – recherche dans un tableau

La valeur a été trouvée ; la fonction retourne la position de cette valeur, qui dans notre cas est 4.

3.9.2 Recherche Séquentielle

La recherche séquentielle consiste à parcourir le tableau un par un jusqu'à trouver le clef.

1. Ecrivez un sous-programme pascale pour résoudre le problème de la recherche séquentielle.

   Fonction RechercheS(A :tableau, e :entier) : Entier

2. Donnez un exemple de programme principale qui utilisent cette fonction.
3. Si le tableau est ordonnée ; modifiez la fonction pour supporter ce changement.
4. Calculer le nombre de comparaison pour un recherche d'un élément à la fin du tableau de 64 éléments. Qu'est ce que représente ce nombre.

3.9.3 Recherche dichotomique

Cet algorithme opère sur un ensemble ordonné et se sert de l'ordre pour diriger la recherche. Le mot **dichotomie** vient du grec qui signifie : **couper en deux**.

La recherche dichotomique dans un tableau trié repose sur le principe suivant : si la recherche s'effectue entre les indices b_i et b_s on compare la valeur recherchée x avec la valeur qui est à égale distance des bornes b_i et b_s , soit à l'indice $m = \dfrac{b_i + b_s}{2}$

– Si x est égale à la valeur en m, alors la recherche s'arrête positivement, sinon
– Si x est inférieur à la valeur en m alors on continue la recherche entre les indices b_i et $m-1$ sinon
– On continue la recherche entre les indices $m+1$ et b_s.

1. Calculer $log_2(64)$.
2. Ecrire une fonction *RechDicho*

   Fonction RechDicho(A :tableau, e :entier) : Entier

   qui a pour paramètre un tableau que l'on suppose trié, un entier x et qui renvoie l'indice de x si x est dans le tableau, -1 sinon, selon le principe énoncé ci-dessus.

3. Donnez un exemple de programme principale qui utilisent cette fonction.
4. CalculerCalculer le nombre de comparaison pour un recherche d'un élément à la fin du tableau de 64 éléments. Qu'est ce que représente ce nombre.

3.9.4 Complexité

1. En déduire la complexité C_1 du fonction *RechercheS*.
2. En déduire la complexité C_2 du fonction *RechDicho*.
3. Calculer $\lim \frac{C_1}{C_2}$.
4. Qu'est ce que on peut déduire.

Chapitre 4

Structure de données

4.1 Types de données abstraits (TDA)

Un TDA est un ensemble de données organisé de sorte que les spécifications des objets et des opérations sur ces objets (interface) soient séparées de la représentation interne des objets et de de la mise en oeuvre des opérations. Exemple de TDA : le type entier muni des opérations $+$;- ;* ;% ; $=$; $>$; $<$;$<=$;$>=$;$==$ est un TDA.

Une mise en oeuvre d'un TDA est la structure de données particulière et la définition des opérations primitives dans un langage particulier.

Les avantages des TDA sont :
- prise en compte de types complexes.
- séparation des services et du codage. L'utilisateur d'un TDA n'a pas besoin de connaître les détails du codage.
- écriture de programmes modulaires.

4.2 Définition d'un type abstrait

Un type abstrait est décrit par sa signature qui comprend :
- une déclaration des ensembles définis et utilisés ;
- une description fonctionnelle des opérations ;
- une description axiomatique de la sémantique des opérations.

4.2.1 Exemple

Déclaration :

$$EntierNaturel.0 \in EntierNaturel$$

Description fonctionnelle :
$succ : EntierNaturel \to EntierNaturel$
$+ : EntierNaturel + EntierNaturel \to EntierNaturel$
$\times : EntierNaturel \times EntierNaturel \to EntierNaturel$
Description axiomatique

1. $\forall x \in EntierNaturel; \exists x'; succ(x) = x'$
2. $\forall x; x_0 \in EntierNaturel; x \neq x_0 succ(x) \neq succ(x_0)$
3. $\nexists x \in EntierNaturel; succ(x) = 0$

4. $\forall x \in EntierNaturel; x + 0 = x$
5. $\forall x; y \in EntierNaturel; x + succ(y) = succ(x + y)$
6. $\forall x \in EntierNaturel; x \times 0 = 0$
7. $\forall x; y \in EntierNaturel; x \times succ(y) = x + xy$

4.3 L'implantation D'un Type Abstrait

L'implantation est la façon dont le type abstrait est programmé dans un langage particulier.

Pour un type abstrait donné, plusieurs implantations possibles peuvent être développées.Le choix d'implantation du type abstrait variera selon l'utilisation qui en est faite et aura une influence sur la complexité des opérations.

4.4 Utilisation de type abstrait

Puisque la définition d'un type abstrait est indépendante de toute implantation particulière, l'utilisation du type abstrait devra se faire exclusivement par l'intermédiaire des opérations qui lui sont associées et en aucun cas en tenant compte de son implantation. D'ailleurs certains langages de programmation peuvent vous l'imposer, mais ce n'est malheureusement pas le cas de tous les langages de programmation et c'est alors au programmeur de faire preuve de rigueur !

Les en-têtes des fonctions et des procédures du type abstrait et les affirmations qui définissent leur rôle représentent l'interface entre l'utilisateur et le type abstrait. Ceci permet évidemment de manipuler le type abstrait sans même que son implantation soit définie, mais aussi de rendre son utilisation indépendante vis-à-vis de tout changement d'implantation.

Chapitre 5

Structures linéaires-Les Listes

5.1 Introduction

Les structures linéaires sont un des modèles de données les plus élémentaires et utilisés dans les programmes informatiques. Elles organisent les données sous forme de séquence non ordonnée d'éléments accessibles de façon séquentielle. Tout élément d'une séquence, sauf le dernier, possède un successeur. Une séquence s constituée de n éléments sera dénotée comme suit :
$s =< e_1, e_2, e_3, \ldots, e_n >$
et la séquence vide :
$s =<>$

Les opérations d'ajout et de suppression d'éléments sont les opérations de base des structures linéaires. Selon la façon dont procèdent ces opérations, nous distinguerons différentes sortes de structures linéaires. Les listes autorisent des ajouts et des suppressions d'éléments n'importe où dans la séquence, alors que les piles et les files ne les permettent qu'aux extrémités. On considère que les piles et les files sont des formes particulières de liste linéaire. Dans ce chapitre, nous commencerons par présenter la forme générale, puis nous étudierons les trois formes particulières de liste.

5.2 Définition

La liste définit une forme générale de séquence. Une liste est une séquence finie d'éléments repérés selon leur rang. S'il n'y a pas de relation d'ordre sur l'ensemble des éléments de la séquence, il en existe une sur le rang. Le rang du premier élément est 1, le rang du second est 2, et ainsi de suite. L'ajout et la suppression d'un élément peut se faire à n'importe quel rang valide de la liste.

5.3 Définition abstraite

5.3.1 Ensembles

Liste est l'ensemble des listes linéaires non ordonnées dont les éléments appartiennent à un ensemble ε quelconque. L'ensemble des entiers représente le rang des éléments. La constante *listevide* est la liste vide.
Liste utilise ε, naturel et entier
$listevide \in Liste$

5.3.2 Définition abstraite

Le type abstrait Liste définit les quatre opérations de base suivantes :
$longueur : Liste \leftarrow naturel$
$i^{eme} : Liste \times entier \leftarrow \varepsilon$
$supprimer : Liste \times entier \leftarrow Liste$
$ajouter : Liste \times entier \times \varepsilon \leftarrow Liste$
L'opération longueur renvoie le nombre d'éléments de la liste. L'opération i^{eme} retourne l'élément d'un rang donné. Enfin, supprimer (resp. ajouter) supprime (resp. ajoute) un élément à un rang donné.

5.3.3 Définition axiomatique

Les axiomes suivants décrivent les quatre opérations applicables sur les listes. La longueur d'une liste vide est égale à zéro. L'ajout d'un élément dans la liste augmente sa longueur de un, et sa suppression la réduit de un.
$\forall l \in Liste; et \forall e \in \varepsilon$
$longueur(listevide) = 0$
$\forall r; 1 \leq r < longueur(l); longueur(supprimer(l; r)) = longueur(l) - 1$
$\forall r; 1 \leq r \leq longueur(l) + 1; longueur(ajouter(l, r, e)) = longueur(l) + 1$

L'opération i^{eme} renvoie l'élément de rang r, et n'est définie que si le rang est valide.
$(4)\forall r; r < 1 et r > longueur(l); \nexists e; e = i^{eme}(l, r)$

L'opération supprimer retire un élément qui appartient à la liste, c'est-à-dire dont le rang est compris entre un et la longueur de la liste. Les axiomes suivants indiquent que le rang des éléments à droite de l'élément supprimé est décrémenté de un.
$\forall r; 1 \leq r \leq longueur(l) et 1 \leq i < r, i^{eme}(supprimer(l, r), i) = i^{eme}(l, i)$
$\forall r; 1 \leq r \leq longueur(l) et r \leq i \leq longueur(l) - 1; i^{eme}(supprimer(l; r); i) = i^{eme}(l; i + 1)$
$\forall r; r < 1 et r > longueur(l); \nexists l'; l' = supprimer(l, r)$

L'opération ajouter insère un élément à un rang compris entre un et la longueur de la liste plus un. Le rang des éléments à la droite du rang d'insertion est incrémenté de un. $\forall r; 1 \leq r \leq longueur(l) + 1 et 1 \leq i < r; i^{eme}(ajouter(l, r, e), i) = i^{eme}(l, i)$
$\forall r; 1 \leq r \leq longueur(l) + 1 et r = i; i^{eme}(ajouter(l; r; e); i) = e$
$\forall r; 1 \leq r \leq longueur(l) + 1 et r < i \leq longueur(l) + 1; i^{eme}((ajouter, r, e), i) = i^{eme}(l, i - 1)$
$\forall r; r < 1 et r > longueur(l) + 1; @l'; l' = ajouter(l, r, e)$

5.4 Implimentation

5.4.1 Implémentation contiguë(Utilisation d'un tableau)

La méthode qui vient en premier à l'esprit, lorsqu'on mémorise les éléments d'une liste dans un tableau, est de conserver systématiquement le premier élément à la première place du tableau, et de ne faire varier qu'un indice de fin de liste.
L'encodage des listes (version itérative) utilise des tableaux :

- Une liste est représentée par un tableau de taille n et un entier qui représente la taille de la liste ;
- La liste vide est donc un tableau de taille n (comme toutes les autres) dont les valeurs n'ont pas d'importance et dont la taille est 0 .

la définition de la structure *liste* est donc :
type liste=**enregistrement**

```
tab:tableau[1..n]d'élément
lg:entier
```

fin

L'algorithme de l'opération *ime* est très simple, puisque le tableau permet un accès direct à l'élément de rang r. La complexité de cet algorithme est donc $O(1)$.

Fonction ieme(l :liste,r :entier) :élément

Var :
début
  si *(r<=l.lg)& (r>=1)* **alors**
    $ieme \leftarrow l.tab[r]$
  finsi
fin

L'opération de suppression d'un élément de la liste provoque un décalage des éléments qui se situent à droite du rang de suppression. Pour une liste de n éléments, la complexité de cette opération est $O(n)$, et l'algorithme qui la décrit est le suivant :

Procédure supprimer(var l :liste ;r :entier)

Var : i :entier
début
  si *(r<=l.lg)& (r>=1)* **alors**
    pour $i \leftarrow r$ **à** *l.lg-1* **faire**
      $l.tab[i] \leftarrow l.tab[i+1]$
    finpour
  finsi
  $l.lg \leftarrow l.lg - 1$
fin

L'opération d'ajout d'un élément e au rang r consiste à décaler d'une position vers la droite tous les éléments à partir du rang r. Le nouvel élément est inséré au rang r. le procédure *ajouter* doit vérifier si le tableau *tab* dispose d'une place libre avant d'ajouter un nouvel élément. Comme pour l'opération de suppression, la complexité de cet algorithme est $O(n)$. L'algorithme est le suivant :

Procédure ajouter(var l :liste ;e :élément ;r :entier)

Var : i :entier
début
  si *(r<=l.lg+1)& (r>=1)* **alors**
    pour $i \leftarrow l.lg$ **à** *r-2* **faire**
      $l.tab[i+1] \leftarrow l.tab[i]$
    finpour
  finsi
  $l.tab[r] \leftarrow e$
  $l.lg \leftarrow l.lg + 1$
fin

Il est important de remarquer que la suppression de l'élément de tête est très coûteuse puisqu'elle provoque un décalage de tous les éléments de la liste. Ainsi pour des raisons d'efficacité, il sera préférable de gérer le tableau de façon circulaire.

La gestion circulaire du tableau se fait à l'aide de deux indices : un indice de *tte* qui désigne le premier élément de la liste, et un indice de *queue* qui indique l'emplacement libre après le dernier élément de la liste.

la définition de la structure *liste* est donc :

type liste=**enregistrement**

```
tab:tableau[1..n]d'élément
lg:entier
tete,queue:entier
```

fin

Les indices de tête ou de queue sont incrémentés ou décrémentés de un à chaque ajout ou suppression. Lorsqu'on incrémente un indice en fin de tableau, sa prochaine valeur est alors l'indice du premier composant du tableau. De même, lorsqu'on décrémente un indice en début de tableau, sa prochaine valeur est alors l'indice du dernier composant du tableau.

les procedures et fonctions sont écrits comme suite :

Fonction ieme(l :liste,r :entier) :élément

Var :
début
  si *(r<=l.lg)& (r>=1)* **alors**
    $ieme \leftarrow l.tab[((l.tete + r - 2) mod\ n) + 1]$
  finsi
fin

```
Procédure suprimmer(var l :liste ;r :entier)
Var : i :entier
début
    si (r<=l.lg)& (r>=1) alors
        si r==l.lg alors
            si l.queue = 1 alors
                l.queue ← n
            sinon
                l.queue ← l.queue − 1
            finsi
        sinon
            si r==1 alors
                si l.tete = n alors
                    l.tete ← 1
                sinon
                    l.tete ← l.tete + 1
                finsi
            sinon
                pour i ← l.tete + r à l.tete + l.lg − 1 faire
                    l.tab[(i − 2 mod n) + 1] ← l.tab[(i − 1 mod n) + 1]
                finpour
                si l.queue = 1 alors
                    l.queue ← n
                sinon
                    l.queue ← l.queue − 1
                finsi
            finsi
        finsi
        l.lg ← l.lg − 1
    finsi
fin
```

Procédure ajouter(var l :liste ;e :élément ;r :entier)

```
Var : i :entier
début
    si (r<=l.lg+1)& (r>=1) alors
        si r==l.lg+1 alors
            l.tab[l.queue] ← e
            si l.queue = n alors
                l.queue ← 1
            sinon
                l.queue ← l.queue + 1
            finsi
        sinon
            si r==1 alors
                si l.tete = 1 alors
                    l.tete ← n
                sinon
                    l.tete ← l.tete − 1
                finsi
                l.tab[l.tete] ← e
            sinon
                pour i ← l.tete + l.lg à l.tete+r-2 faire
                    l.tab[(i − 1 mod n) + 1] ← l.tab[(i − 2 mod n) + 1]
                finpour
                l.tab[l.tete + r − 1] ← e
            finsi
        finsi
        l.lg ← l.lg + 1
    finsi
fin
```

5.4.2 Implémentation chaînées(Utilisation d'une structure chaînée)

Une structure chaînée est une structure dynamique formée de noeuds reliés par des liens.
Pour représenter une liste. les noeuds sont représentés par des boîtes rectangulaires,et les liens par des flèches.

L'encodage des listes (version chaînées) utilise des pointeurs :

type cellule=**enregistrement**

```
Info:élément
suivant:^ cellule
```

fin

type liste=↑cellule

Avec cette structure chaînée, les opérations *longueur*, *ime*, *supprimer*, et *ajouter* nécessitent toutes un parcours séquentiel de la liste et possèdent donc une complexité égale à $O(n)$.

Fonction longueur(l :liste) :entier

```
Var : i :entier
début
    i ← 0
    tant que l<>nil faire
        i ← i + 1
        l ← l ↑ .suivant
    fintq
    longueur ← i
fin
```

On peut écrire une version récursive de cette fonction comme suit :

Fonction longueur(l :liste) :entier

```
début
    si l=nil alors
        longueur ← 0
    sinon
        longueur ← 1 + longueur(l ↑ .suivant)
    finsi
fin
```

On atteint le noeud de rang r en appliquant $r-1$ fois l'opération *Suivant* à partir de la tête de liste. L'algorithme de l'opération ième s'écrit :

Fonction ieme(l :liste ;r :entier) :élément

```
Var : i :entier
début
    si (r<=longueur(l))& (r>=1) alors
        pour i ← 1 à r − 1 faire
            l ← l ↑ .suivant
        finpour
        ieme ← l ↑ .info
    finsi
fin
```

Toujour on peur écrire des algorithmes récursive :

Fonction ieme(l :liste ;r :entier) :élément

```
début
    si (r<=longueur(l))& (r>=1) alors
        si r==1 alors
            ieme ← l ↑ .info
        sinon
            ieme ← ieme(l ↑ .suivant, r − 1)
        finsi
    finsi
fin
```

La suppression d'un élément de rang r consiste à affecter au lien qui le désigne la valeur de son lien suivant. Notez que si r est égal à un, il faut modifier la tête de liste.

Procédure suprimmer(var l :liste ;r :entier)

```
Var : i :integer
t,q :liste
début
    si (r<=longueur(l))& (r>=1) alors
        si r=1 alors
            l ← l ↑ .suivant
        sinon
            t ← l
            q ← nil
            pour i ← 1 à r − 1 faire
                q ← t
                t ← t ↑ .suivant
            finpour
            q ↑ .suivant ← t ↑ .suivant
        finsi
    finsi
fin
```

On remarque que pour supprimer un élément de rang r, il suffit de supprimer l'élément de rang $r-1$ de la sous-liste qui commence par le suivant de la tête. alors on peut traduire ceci par l'algorithme :

Procédure suprimmer(var l :liste ;r :entier)

```
Var :
début
    si (r<=longueur(l))& (r>=1) alors
        si r=1 alors
            l ← l ↑ .suivant
        sinon
            supprimer(l ↑ .suivant, r − 1)
        finsi
    finsi
fin
```

L'ajout d'un élément e au rang r consiste à créer un nouveau noeud c initialisé à la valeur e, puis à relier le noeud de rang $r-1$ à c, et enfin à relier le noeud c au noeud de rang r. Si l'élément est ajouté en queue de liste, son suivant est la valeur *nil*. Comme précédemment, si $r = 1$, il faut modifier la tête de liste.

Procédure ajouter(var l :liste ;e :élément ;r :entier)

```
Var : i :integer
t,q :liste
c :↑cellule
début
    si (r<=longueur(l))& (r>=1) alors
        new(c)
        c↑.info ← e
        si r=1 alors
            c↑.suivant ← l
            l ← c
        sinon
            t ← l
            q ← nil
            pour i ← 1 à r − 1 faire
                q ← t
                t ← t↑.suivant
            finpour
            c↑.suivant ← t
            q↑.suivant ← c
        finsi
    finsi
fin
```

Comme dans la procédure *supprimer* la procédure *ajouter* peut se traduire en sa version itérative suivante :

Procédure ajouter(var l :liste ;e :élément ;r :entier)

```
Var : c :↑cellule
début
    si (r<=longueur(l))& (r>=1) alors
        si r=1 alors
            new(c)
            c↑.info ← e
            c↑.suivant ← l
            l ← c
        sinon
            ajouter(l↑.suivant, e, r − 1)
        finsi
    finsi
fin
```

5.5 Exercices

5.5.1 Exercice 01 : *Pointeurs*

Considérer l'algorithme suivant :
si a se trouve à #300540 et b à #202100

Algorithme 12: Pointeurs

Var : a,b,c :entier
p_x, p_y, p_z :↑entier
début
  $a \leftarrow 4$;
  $b \leftarrow 12$;
  $c \leftarrow 23$;
  $p\_x \leftarrow Addr(a)$
  $p\_y \leftarrow Addr(b)$
  $p\_z \leftarrow p\_y$
  $p\_x \uparrow \leftarrow p\_x \uparrow +2$
  $p\_y \uparrow \leftarrow p\_y \uparrow +1$
  $c \leftarrow c + 3$
fin

- Faites tourner cet algorithme dans un tableau (de 6 colonnes bien sur).

5.5.2 Exercice 02 : *Pointeurs*

Considérons le programme suivant :

Algorithme 13: Pointeurs2

Var : A :tableau[1..9]d'entier
p :↑entier
début
  $A \leftarrow \{12, 23, 34, 45, 56, 67, 78, 89, 90\}$
  $p \leftarrow A$
fin

Quelles valeurs ou adresses fournissent ces expressions ?

1. p↑+ 2
2. (p + 2)↑
3. p + 1
4. Addr(A[4]) - 3
5. A + 3
6. Addr(A[7]) - p
7. p + (p↑- 10)
8. (p + (p + 8)↑- A[7])↑

5.5.3 Exercice 03 : *les listes*

L'encodage des listes vu en cours (version itérative) utilise des tableaux :
- Une liste est représentée par un tableau de taille n et un entier qui représente la taille de la liste ;
- La liste vide est donc un tableau de taille n (comme toutes les autres) dont les valeurs n'ont pas d'importance et dont la taille est 0 .

la définition de la structure *liste* est donc :
type liste=**enregistrement**

```
tab:tableau[1..n]d'élément
lg:entier
```

fin
Écrivez les fonctions suivantes :
- *liste*() qui renvoie la liste vide ;
- *vide*(l) qui renvoie **vrai** si la liste l est vide et **faux** sinon ;
- *cons*(r, l) qui renvoie la liste commençant par l'élément r suivi de la queue l ;
- *tete*(l) qui renvoie le premier élément de la liste l ;
- *queue*(l) qui renvoie la queue de la liste l.

(ces fonctions sont toutes très simples à écrire, il ne faut pas plus de 3 lignes pour chacune)

5.5.4 Exercice 04 :*les listes*

Écrivez les fonctions un peu plus avancées suivantes :
- *occ*(x, l) qui compte le nombre dŠoccurrences de x dans la liste l ;
- *concatener*(l, l') qui fusionne le sdeux liste l et l' ;
- *recherche*(x, l) qui renvoi le rang de l'élément x de la lsite l s'il existe , et -1 sinon ;

5.5.5 Exercice 05 :*les listes*

Récrivez les fonctions des exercices 03 et 04 dans le cas d'une liste circulaire.

5.5.6 Exercice 06 :*Algorithmes de base*

Ecrire les algorithmes de base suivants sur les listes linéaires chaînées :

1. Suppression par valeur.
2. Inversement d'une liste.
   - En créant une nouvelle liste
   - Sans créer une nouvelle liste (En Inversant le chainage des mêmes maillons)
3. Tri par la méthode des bulles
4. Recherche de l'élément qui a le plus grand nombre d'occurrences

5.5.7 Exercice 07 :*Un peu avancés*

Soit une liste linéaire chaînée contenant des nombres entiers et dont la tête est l :

1. Ecrire la fonction *tete*(l) qui retourne la valeur du premier élément de la liste.
2. Ecrire la fonction $sans_tete(l)$ qui retourne la liste sans le premier élément.
3. Ecrire la fonction *cons*(l, r) qui retourne une liste dont le premier élément est l'élément du rang r et le reste est la liste l.

4. Ecrire la fonction $triee(L)$ qui retourne vrai si la liste l est triée dans l'ordre croissant et faux sinon.
5. Ecrire la fonction $fusion(l1, l2)$ qui prend deux listes triées dans l'ordre croissant $l1$ et $l2$ et retourne une liste triée, dans le même ordre, contenant les deux listes et cela en utilisant les fonctions précédentes.

5.5.8 Exercice 08 : *Différence de deux listes linéaires chaînées*

Soient $l1$ et $l2$ deux listes linéaires chaînées . Ecrire la procédure qui permet de construire la liste $l = l1 - l2$ contenant tous les éléments appartenant à $l1$ et n'appartenant pas à $l2$.

5.5.9 Exercice 09 : *Réaliser le chaînage arrière d'une liste doublement chaînée*

Concevoir un sous-algorithme qui réalise le chaînage arrière d'une liste doublement chaînée dont seul le chaînage avant a été effectué.

5.6 TP : Dictionnaire

L'objective de ce TP est de vous formalisez avec l'implémentation des listes chaînées afin de les utilisées pour résoudre des problèmes.

Notez bien :

- La date de remise de ce TP **(Un rapport de 3 pages max + sources)** est fixée le : .....
- Il est recommander d'utiliser le compilateur Dev-c++ ;

5.6.1 Problème

Pour gérer un dictionnaire. nous avons besoin d'effectuer les opérations suivantes :

- lecture d'un ensemble de mots et de leur définition d'un fichier, et les stockés dans une structure expliquée en détail plus bas ;
- trier les mots dans l'ordre alphabétique ;
- insérer un nouveau mot dans un dictionnaire dans l'ordre ;
- supprimer un mot du dictionnaire ;
- chercher un mot dans le dictionnaire ;
- afficher le mots s'il est trouver.

Pour ce TP, un fichier texte contenant les mots et leurs définitions sera fourni.

Un programme exemple sera aussi donné pour montrer comment lire un fichier texte ligne par ligne. Vous devez adapter ce programme dans votre méthode de lecture.

5.6.2 Implémentation

Pour stocker le dictionnaire, on vous demande d'utiliser la structures de données Liste ; notre liste contient un maillon qui contient deux chaîne de caractère(l'un pour le mot et l'autre pour sa définition) et un pointeur au mot suivant

$tte \rightarrow$ [mot1][définition1][-] $\rightarrow$ [mot2][définition2][-] $\rightarrow nil$

On désire dans ce TP de créer une bibliothèque *listes.h* qui contient la définition de la structure liste linéaire simplement chaînée (Vue en cours) et les fonctions et procédure suivants :

- *liste creer*() : Le constructeur d'une liste vide ;
- *intest_vide*(*liste l*) : Qui teste si la liste est vide ou non ;
- *void lire*(*String nom*) : Ce procédure permet de lire toutes les lignes d'un fichier (dont le nom est spécifier par le paramètre) contenant un mot dans chaque deux lignes, et ajouter les mots et les définitions à la fin de liste correspondante dans le dictionnaire. Dans le fichier, le mot est mis dans le premier ligne , et ensuite on a un String pour sa définition dans le deuxième ligne.
- *void trier*() : Ce procédure permet de trier la liste ;
- *liste ajouter*(*String mot*, *String definition*) ;
- *liste supprimer*(*String mot*, *String definition*) ;
- *String recherche*(*String mot*) : Doit retourner la définition (un String) correspondent au mot. Si le mot n'existe pas dans le dictionnaire, elle retourne *NULL*. ;
- *void afficher*() : IL affiche les mots et les définitions dans l'ordre. Le format d'impression est : une ligne pour un mot. Vous devez utiliser la récursion dans cette méthode.

Écrivez en fin une application en utilisant les opération définit ultérieurement.

Chapitre 6

Les Piles

6.1 Introduction

Une pile est une séquence d'éléments accessibles par une seule extrémité appelée *sommet*. Toutes les opérations définies sur les piles s'appliquent à cette extrémité. L'élément situé au sommet s'appelle le *sommet de pile*.

L'ajout et la suppression d'éléments en sommet de pile suivent le modèle dernier entré Ű premier sorti (LIFO). Les piles sont des structures fondamentales, et leur emploi dans les programmes informatiques est très fréquent. Nous avons déjà vu que le mécanisme d'appel des sous-programmes suit ce modèle de pile. Les logiciels qui proposent une fonction **undo** servent également d'une pile pour défaire, en ordre inverse, les dernières actions effectuées par l'utilisateur. Les piles sont également nécessaires pour évaluer des expressions postfixées.

6.2 Définition abstraite

6.2.1 Ensembles

Pile est l'ensemble des piles dont les éléments appartiennent à un ensemble E quelconque. Les opérations sur les piles seront les mêmes quelle que soit la nature des éléments manipulés. La constante *pilevide* représente une pile vide.

Pile **utilise** E et *booleen*
$pilevide \in Pile$

6.2.2 Description fonctionnelle

Quatre opérations abstraites sont définies sur le type Pile :
empiler : *Pile* X $E \rightarrow Pile$
depiler : Pile $\rightarrow Pile$
sommet : Pile $\rightarrow E$
$est - vide?$: Pile $\rightarrow booleen$

Le rôle de l'opération *empiler* est d'ajouter un élément en sommet de pile, celui de *dpiler* de supprimer le sommet de pile et celui de *sommet* de renvoyer l'élément en sommet de pile. Enfin, l'opération $est - vide?$ indique si une pile est vide ou pas.

6.2.3 Description axiomatique

La sémantique des fonctions précédentes est définie formellement par les axiomes suivants :

$\forall p \in Pile\,; \forall e \in E$
(1) $est-vide?(pilevide) = vrai$
(2) $est-vide?(empiler(p;e)) = faux$
(3) $depiler(empiler(p;e)) = p$
(4) $sommet(empiler(p;e)) = e$
(5) $\nexists p; p = depiler(pilevide)$
(6) $\nexists e; e = sommet(pilevide)$

Notez que ce sont les axiomes (3) et (4) qui définissent le comportement LIFO de la pile. Les opérations dépiler et sommet sont des fonctions partielles, et les axiomes (5) et (6) précisent leur domaine de définition ; ces deux opérations ne sont pas définies sur une pile vide.

6.3 implémentation

La complexité des opérations de pile est $O(1)$ quelle que soit l'implantation choisie, tableau ou structure chaînée. L'implantation doit donc assurer un accès direct au sommet de la pile.

6.3.1 Utilisation des tableaux (Implémentation Contiguë)

L'encodage des piles (version itérative) utilise des tableaux :
- Une *Pile* est représentée par un tableau de taille n et un entier qui représente l'indice de sommet de la pile ;
- *pilevide* est donc un tableau de taille n (comme toutes les autres) dont les valeurs n'ont pas d'importance et dont l'indice de sommet est 0 .

la définition de la structure pile est donc :

type pile=**enregistrement**

```
tab:tableau[1..n]d'élément
sommet:entier
```

fin

Les algorithmes des opérations de pile sont très simples. L'opération *sommet* consiste à retourner l'élément de queue, alors que *depiler* et *empiler* consistent, respectivement, à supprimer et à ajouter en queue.

6.3.2 Utilisation des pointeurs (Implémentation chaînée)

L'encodage des piles (version chaînées) utilise des pointeurs :
type cellule=**enregistrement**

```
Info:élément
suivant:^ cellule
```

fin
type pile=↑cellule
Les algorithmes des opérations de pile sont les suivant :

6.4 Exercices

6.4.1 Exercice 02 : *Algo. de base sur les Piles*

Ecriver les sous-algorithmes suivants :

1. Suppression de l'ième élément ;
2. Recherche d'un élément ;
3. Concaténer deux piles ;
4. Eclater une pile en deux : l'une pour les éléments paire, et l'autre pour les éléments impaire.

6.4.2 Exercice 03 : *Un peu avancés*

En utilisant une pile, écrivez un sous-algorithme qui vérifie si un texte lu sur l'entrée standard est correctement parenthésé. Il s'agit de vérifier si à chaque parenthéseur fermant rencontré },],) ou correspond son parenthéseur ouvrant },[ou (.

Le sous-algorithme écrit sur la sortie standard **Vrai** ou **Faux** selon que le texte est correctement parenthésé ou pas.

6.4.3 Exercice 03 : *Inverser pile et file*

Concevoir deux sous-algorithmes, qui créent respectivement :
– la file inverse d'une file ;
– la pile inverse d'une pile.
Ces deux sous-algorithmes doivent restituer leur entrée inchangée.

Chapitre 7

Les Files

7.1 Introduction

Les files définissent le modèle premier entré premier sorti (FIFO). Les éléments sont insérés dans la séquence par une des extrémités et en sont extraits par l'autre. Ce modèle correspond à la file d'attente que l'on rencontre bien souvent face à un guichet dans les bureaux de poste, ou à une caisse de supermarché la veille du week-end. à tout moment, seul le premier client de la file accède au guichet ou à la caisse.

Le modèle de file est très utilisé en informatique. On le retrouve dans de nombreuses situations, comme, par exemple, dans la file d'attente d'un gestionnaire d'impression d'un système d'exploitation.

7.2 Définition abstraite

7.2.1 Ensembles

File est l'ensemble des files dont les éléments appartiennent à l'ensemble E, et la constante $filevide$ représente une file vide.

File **utilise** E et *booleen*

$fillevide \in File$

7.2.2 Description fonctionnelle

Quatre opérations sont définies sur le type File :

$enfiler$: $File$ X $E \rightarrow File$

$defiler$: $File \rightarrow File$

$dernier$: $File \rightarrow E$

$est-vide?$: $File \rightarrow booleen$

l'Opération $enfiler$ a pour rôle d'ajouter un élément en queue de file, et l'opération $defiler$ supprime l'élément en tête de file. *dernier* retourne le dernier élément de la file et $est-vide?$ indique si une file est vide ou pas. Notez que les signatures de ces opérations sont, au mot ń$File$ż près, identiques à celles des opérations du type abstrait $Pile$. Ce sont bien les axiomes qui vont différencier ces deux types abstraits.

7.2.3 Description axiomatique

Ce sont en particulier, les axiomes (3) et (4), d'une part, et (5) et (6) d'autre part, qui distinguent le comportement de la file de celui de la pile. Ils indiquent clairement qu'un élément est ajouté par une extrémité de la file, et qu'il est accessible par l'autre extrémité.

$\forall f \in File\,;\, \forall e \in E$
(1) $est-vide?(filevide) = vrai$
(2) $est-vide?(enfiler(f;e)) = faux$
(3) $est-vide?(f) \Rightarrow dernier(enfiler(f;e)) = e$
(4) $non\ est-vide?(f) \Rightarrow dernier(enfiler(f;e)) = dernier(f)$
(5) $est-vide?(f) \Rightarrow defiler(enfiler(f;e)) = filevide$
(6) $nonest-vide?(f) \Rightarrow defiler(enfiler(f;e)) = enfiler(defiler(f);e)$
(7) $\nexists f; f = defiler(filevide)$
(8) $\nexists e; e = dernier(filevide)$

7.3 implantation

7.3.1 Utilisation des tableaux (Implémentation Contiguë)

au contraire de celle de *Pile*, l'implantation de l'interface *File* (version itérative) utilise des tableaux circulaire Pour que ces opérations gardent une complexité égale à $O(1)$:

- Une *File* est représentée par un tableau de taille n et deux entier qui représentent, respectivement l'indice de premier et dernier élément de la file ;
- *filevide* est donc un tableau de taille n (comme toutes les autres) dont les valeurs n'ont pas d'importance et dont les indices de premier et dernier élément est 0 .

la définition de la structure file est donc :

type file=**enregistrement**

```
tab:tableau[1..n]d'élément
premier, dernier:entier
```

fin

7.4 Exercices

7.4.1 Exercice 01 :

- Que peut bien vouloir dire FIFO ?

7.4.2 Exercice 02 : *Algo. de base sur les Files*

Ecriver les sous-algorithmes suivants :

1. Suppression de l'ième élément ;
2. Recherche d'un élément ;
3. Concaténer deux files ;
4. Éclater une file en deux : l'une pour les éléments paire, et l'autre pour les éléments impaire.

7.4.3 Exercice 03 : *Un peu avancés*

- Montrez comment implémenter une pile à l'aide de deux files.
- Montrez comment implémenter une file à l'aide de deux piles.

7.4.4 Exercice 03 : *Inverser une file*

Concevoir deux sous-algorithmes, qui créent respectivement :
– la file inverse d'une file ;
– la file inverse d'une file en utilisant une file auxiliaire.