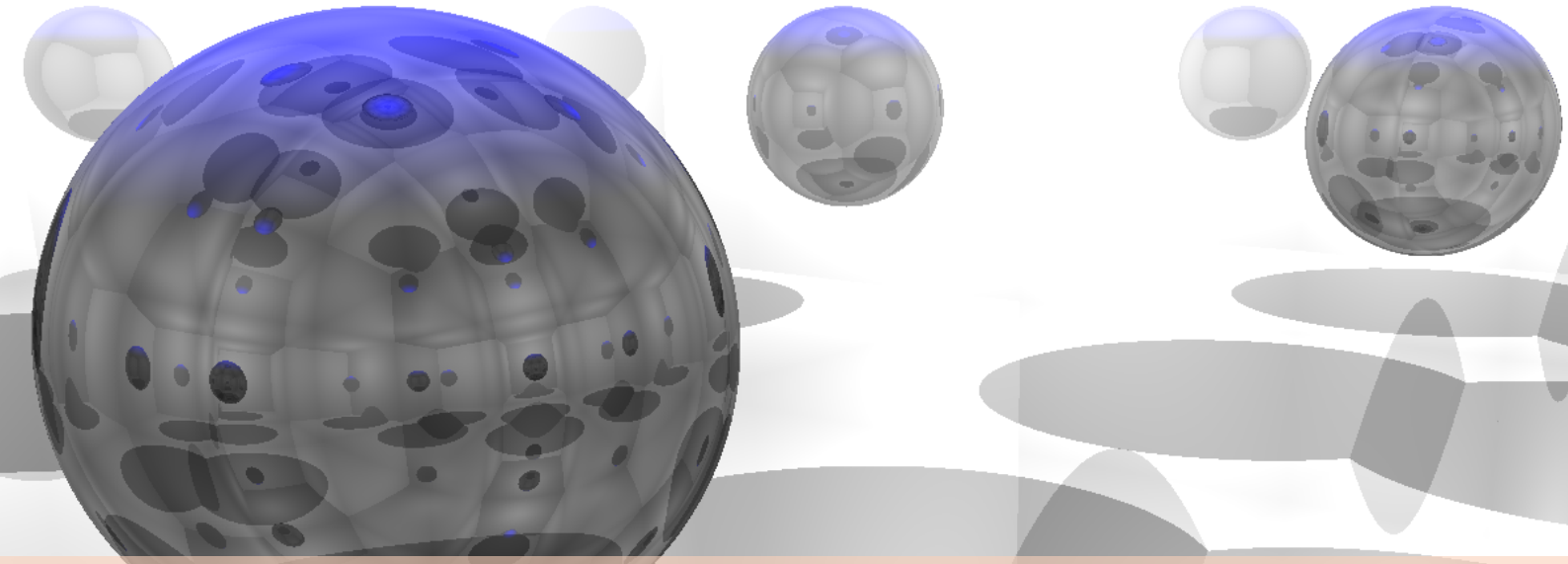


Algorithmique et Structures de Données II

Cours, Travaux dirigés et sujets d'examens

Fethi MGUIS

FACULTÉ DES SCIENCES DE GABÈS

www.fsg.rnu.tn

Mai 2016

Table des matières

I Cours

1 Les enregistrements 11

1.1 Introduction 11

1.2 Définition d'un type "Enregistrement" 11

1.3 Déclaration d'un enregistrement à partir d'un type structuré 12

1.4 Manipulation d'un enregistrement 13

1.4.1 Accès aux champs d'un enregistrement 13

2 Les pointeurs 15

2.1 Introduction 15

2.2 Adresse et valeur d'une variable 15

2.3 Arithmétique des pointeurs 17

2.4 Allocation dynamique 18

2.5 Pointeurs et tableaux 19

2.6 Tableau de pointeurs 20

3 Les listes chaînées 21

3.1 La notion de structure auto-référentielle 21

3.2 Définition d'une liste chaînée 21

3.3 Principales opérations pour une liste simplement chaînée 22
3.3.1 Création d'une liste vide . . . 23
3.3.2 Vérification si une liste est vide . . . 23
3.3.3 Affichage des éléments d'une liste . . . 23
3.3.4 Déterminer la taille d'une liste . . . 23
3.3.5 Recherche d'un élément dans une liste . . . 24
3.3.6 Ajout d'un élément en tête . . . 24
3.3.7 Ajout d'un élément en queue . . . 24
3.3.8 Ajout d'un élément à une position donnée . . . 25
3.3.9 Suppression de l'élément en tête . . . 26
3.3.10 Suppression de l'élément en queue . . . 26
3.3.11 Suppression d'un l'élément donné . . . 27
3.3.12 Suppression de l'élément d'une position donnée . . . 27
3.4 Liste doublement chaînée 28
3.5 Principales opérations pour une liste doublement chaînée 28
3.5.1 Ajout d'un élément en tête . . . 29
3.5.2 Ajout d'un élément en queue . . . 30
3.5.3 Ajout d'un élément à une position donnée . . . 30
3.5.4 Suppression de l'élément en tête . . . 31
3.5.5 Suppression de l'élément en queue . . . 31
3.5.6 Suppression d'un l'élément donné . . . 32
3.5.7 Suppression de l'élément d'une position donnée . . . 32

4 Les piles et les files . . . 35
4.1 Introduction 35
4.2 La notion de pile 35
4.2.1 Caractéristiques . . . 35
4.2.2 Principales opérations . . . 35
4.2.3 Initialisation d'une pile vide . . . 36
4.2.4 Vérification si une pile est vide . . . 36
4.2.5 Ajout d'un élément : Empiler . . . 36
4.2.6 Récupération d'un élément de la pile : Dépiler . . . 36
4.3 La notion de file 37
4.3.1 Caractéristiques . . . 37
4.3.2 Principales opérations . . . 37
4.3.3 Initialisation d'une File vide . . . 37
4.3.4 Vérification si une file est vide . . . 38
4.3.5 Ajout d'un élément : Enfiler . . . 38
4.3.6 Récupération d'un élément de la file : Défiler . . . 38

5 Les arbres . . . 39
5.1 Introduction 39
5.2 Arbre binaire 39
5.2.1 Définitions . . . 39

5.2.2 Principales opérations 40
5.2.3 Création d'un arbre vide 40
5.2.4 Vérification si un arbre est vide 40
5.3 arbre binaire de recherche 43

II **Travaux Dirigés**

TD1 : Les enregistrements 47

TD2 : Les pointeurs 53

TD3 : Les listes chaînées 55

TD4 : Les piles et les files 61

TD5 : Les arbres 63

III **Sujets d'examens**

Table des figures

2.1 (a) Adressage Directe ; (b) Adressage Indirecte. 16
2.2 Exemple d'un tableau de pointeurs. 20

3.1 Structure d'une liste chaînée. 22
3.2 Structure d'une liste doublement chaînée. 28

5.1 Exemple d'un arbre. 39
5.2 Exemple d'un arbre binaire. 40
5.3 Exemple d'un arbre. 43

Cours

1 Les enregistrements 11

1.1 Introduction
1.2 Définition d'un type "Enregistrement"
1.3 Déclaration d'un enregistrement à partir d'un type structuré
1.4 Manipulation d'un enregistrement

2 Les pointeurs 15

2.1 Introduction
2.2 Adresse et valeur d'une variable
2.3 Arithmétique des pointeurs
2.4 Allocation dynamique
2.5 Pointeurs et tableaux
2.6 Tableau de pointeurs

3 Les listes chaînées 21

3.1 La notion de structure auto-référentielle
3.2 Définition d'une liste chaînée
3.3 Principales opérations pour une liste simplement chaînée
3.4 Liste doublement chaînée
3.5 Principales opérations pour une liste doublement chaînée

4 Les piles et les files 35

4.1 Introduction
4.2 La notion de pile
4.3 La notion de file

5 Les arbres 39

5.1 Introduction
5.2 Arbre binaire
5.3 arbre binaire de recherche

1. Les enregistrements

1.1 Introduction

Contrairement aux tableaux qui sont des structures de données dont tous les éléments sont de même type, les enregistrements sont des structures de données dont les éléments peuvent être de différents types et qui se rapportent à la même entité sémantique. Les éléments qui composent un enregistrement sont appelés champs.
Avant de déclarer une variable enregistrement, il faut avoir au préalable définit son type, c'est à dire le nom et le type des champs qui le composent. Le type d'un enregistrement est appelé type structuré. (Les enregistrements sont parfois appelé structures, en analogie avec le langage C).

1.2 Définition d'un type "Enregistrement"

Jusqu'à présent, nous n'avons utilisé que des types primitifs (caractères, entiers, réels, chaînes) et des tableaux de types primitifs. Mais il est possible de créer nos propres types, puis de déclarer des variables ou des tableaux d'éléments de ce type.
Pour ce faire, il faut déclarer un nouveau type, fondé sur d'autres types existants. Après l'avoir défini, on peut dès lors utiliser ce type structuré comme tout autre type normal en déclarant une ou plusieurs variables de ce type. Les variables de type structuré sont appelées enregistrements.
La déclaration des types structurés se fait dans une section spéciale des algorithmes appelée Type, qui précède la section des variables (et succède à la section des constantes). Si l'algorithme comporte des sous-programmes, les types et les constantes sont déclarées en dehors du programme.

Syntaxe

```
Nom_Enregistrement: Enregistrement
    Nom_Champ_1 : Type_Champ_1
    Nom_Champ_2 : Type_Champ_2
    ...
    Nom_Champ_n : Type_Champ_n
Fin Enreg
```

Exemples

```
Personne: Enregistrement
    Nom : Chaîne
    Prénom : Chaîne
    Age : Entier
Fin Enreg
Date: Enregistrement
    Jour : Entier
    Mois : Entier
    Année : Entier
Fin Enreg
Point: Enregistrement
    Num : Entier
    Abscice : Réel
    Ordonné : Réel
Fin Enreg
```

1.3 Déclaration d'un enregistrement à partir d'un type structuré

Une fois qu'on a défini un type structuré, on peut déclarer des variables enregistrements exactement de la même façon que l'on déclare des variables d'un type primitif.

Syntaxe

Nom_Var : Nom_Enregistrement

Exemples

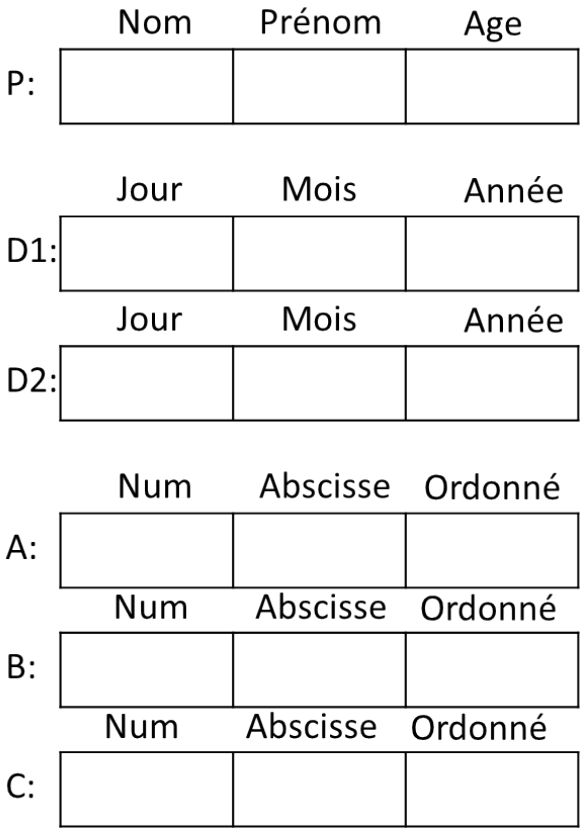
P : Personne

D1, D2 : Date

A, B, C : Point

Représentation

les enregistrements sont composés de plusieurs zones de données, correspondant aux champs.



1.4 Manipulation d'un enregistrement

La manipulation d'un enregistrement se fait à travers ses champs. Comme pour les tableaux, il n'est pas possible de manipuler un enregistrement globalement, sauf pour affecter un enregistrement à un autre de même type (ou le passer en paramètre). Par exemple, pour afficher un enregistrement il faut afficher tous ses champs un par un.

1.4.1 Accès aux champs d'un enregistrement

Alors que les éléments d'un tableau sont par l'intermédiaire de leur indice, les champs d'un enregistrement sont accessibles à travers leur nom, grâce à l'opérateur '**.**'
Nom_Var.Nom_Champ représente la valeur mémorisée dans le champ de l'enregistrement Par exemple, pour accéder à l'âge de la variable P, on utilise l'expression : **P.Age**
Remarque : la lecture d'une telle expression se fait de droit à gauche : l'âge de la personne P.
Attention : le nom d'un champ est **TOUJOURS** précédé du nom de la variable déclarée avec le type enregistrement auquel il appartient. On ne peut pas trouver un nom de champ tout seul, sans indication de la variable.
Les champs d'un enregistrement, tout comme les éléments d'un tableau, sont des variables à qui on peut faire subir les mêmes opérations (affectation, saisie, affichage,...).
Exemple 1 :
Saisie des données concernant deux personnes P1 et P2, puis affichage de la différence d'âge entre ces deux personnes.

2. Les pointeurs

2.1 Introduction

Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée de mots qui sont identifiés de manière unique par un numéro qu'on appelle adresse. Pour retrouver une variable, il suffit donc de connaître l'adresse du mot où elle est stockée ou (s'il s'agit d'une variable qui recouvre plusieurs mots contigus) l'adresse du premier de ces mots. Pour des raisons évidentes de lisibilité, on désigne souvent les variables par des identificateurs et non par leurs adresses. C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire. Parfois, il est très pratique de manipuler une variable par son adresse.

2.2 Adresse et valeur d'une variable

Une variable est caractérisée par :

- Son adresse, c'est à dire l'adresse mémoire à partir de laquelle l'objet est stocké.
- Sa valeur, c'est à dire ce qui est stocké à cette adresse.

Deux modes d'accès :

Adressage direct : Accès au contenu d'une variable par le nom de la variable.

Exemple : Soit A une variable contenant la valeur 10. Avec un mot mémoire de taille 2 octets et sachant que le type int est codé sur 4 octets, la figure 1.1(a) montre comment la variable A peut se présenter en mémoire.

Adressage indirect : Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable.

Exemple : Soit A une variable contenant la valeur 10 et P un pointeur qui contient l'adresse de A(val(A) égale à val(*P)). La figure 1.1(b) montre comment A et P peuvent se présenter en mémoire.

Notion de pointeur :

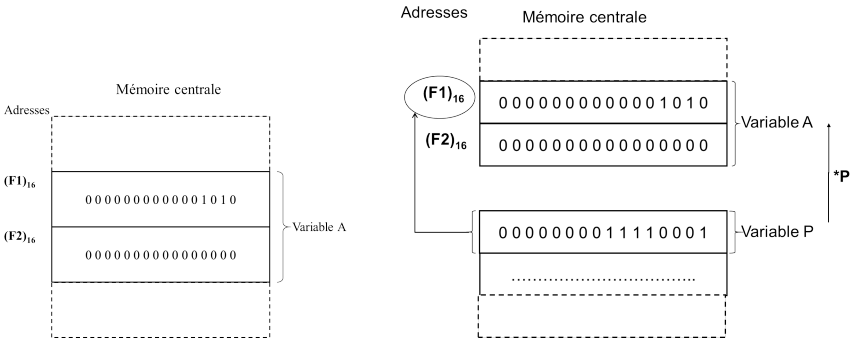


FIGURE 2.1 – (a) Adressage Directe ; (b) Adressage Indirecte.

Un pointeur est une variable dont la valeur est égale à l'adresse d'un autre objet. On déclare un pointeur en algorithmique par l'instruction suivante :

```
nom_du_pointeur: *Type
```

Où :

- **nom_du_pointeur** est l'dentificateur dont la valeur est l'adresse d'un objet de type **"Type"**
- **Type** est le type de l'objet pointé

Exemple :

```
pc:*caractère//pc est un pointeur pointant sur un objet de type caractère.
pe:*entier//pe est un pointeur pointant sur un objet de type entier.
pr:*réel//pr est un pointeur pointant sur un objet de type réel.
```

L'opérateur unaire d'indirection * permet d'accéder directement à la valeur de l'objet pointé. Ainsi si p est un pointeur vers un entier i, *p désigne la valeur de i. Par exemple, l'algorithme suivant :

```
Algorithme Exemple0
Var
  i : Entier
  p : *Entier
Début
  i←3
  p←&i//p contient adresse de i
  Ecrire("Le contenu de la case mémoire pointé par p est :", *p)
  *p←5 //changement de i à travers p
  Ecrire("i =" , i , "*p =", *p)
Fin
```

donne :
contenu de la case mémoire pointé par p est : 3
i = 5 et *p = 5
Dans ce programme, les objets i et *p sont identiques. Ils ont la même valeur. Cela signifie en particulier que toute modification de *p modifie i.

2.3 Arithmétique des pointeurs

La valeur d'un pointeur étant un entier. On peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques. Les seules opérations arithmétiques valides sur les pointeurs sont :

- l'addition d'un entier à un pointeur.
- la soustraction d'un entier à un pointeur.
- la différence de deux pointeurs p et q pointant tous les deux vers des objets de même type. Le résultat est un entier dont la valeur est égale à (p-q)/Taille(type).

Attention ! ! : Notons que la somme de deux pointeurs n'est pas autorisée.
Exemple :
Si k est un entier et p est un pointeur sur un objet de type ***type***. L'expression "p+k" désigne un pointeur sur un objet de type ***type*** dont la valeur est égale à la valeur de p incrémentée de k*sizeof(***type***). Il en va de même pour la soustraction d'un entier à un pointeur et pour les opérateurs d'incrémentation et de décrémentation + + et - -.
Exemple 1 :

```
Algorithme Exemple1
Var
  i : Entier
  p1,p2 : *Entier
Début
  i←3
  p1←&i
  p2←p1
  Ecrire("p1 =" , p1 , "et p2 =", p2)
Fin
```

Exemple 2 :

```
Algorithme Exemple2
Var
  pi : *Entier
  pr : Réel
  pc : Caractère
Début
  *pi←4
  *(pi+1)←5
  *pr←45.7
  pr←pr+1
  *pc←'j'
Fin
```

Remarque : Les opérateurs de comparaison sont également applicables aux pointeurs à condition de comparer des pointeurs qui pointent vers des objets de même type.

2.4 Allocation dynamique

Lorsque l'on déclare une variable caractère, entier, réel, etc. Un nombre de cases mémoire bien défini est réservé pour cette variable. Il n'en est pas de même avec les pointeurs. Avant de manipuler un pointeur, il faut l'initialiser soit par une allocation dynamique soit par une affectation d'adresse p←&i. Sinon, par défaut la valeur du pointeur est égale à une constante symbolique notée NULL.
On peut initialiser un pointeur p par une affectation sur p. Mais il faut d'abord réserver à p un espace mémoire de taille adéquate. L'adresse de cet espace mémoire sera la valeur de p. Cette opération consistant à réserver un espace mémoire pour stocker l'objet pointé s'appelle allocation dynamique. Elle se fait par la fonction **"Allouer"**. Sa syntaxe est la suivante :

```
Ptr$\leftarrow$Allouer(nbr_objets)
```

Cette fonction retourne un pointeur pointant vers "nbr_objets" objets.
Exemples : la fonction **Allouer**
Exemple 1 :

```
Algorithme Exemple1
Var
  pi,pj : *Entier
  pr : Réel
  pc : Caractère
Début
  //réserver 10 octets mémoire, soit la place pour 10 caractères
  pc←Allouer(10)
  //réserver un espace pour 4 entiers
  pi←Allouer(4)
  //réserver un espace pour 6 réels
  pr←Allouer(6)
  //réserver un espace pour 1 entiers
  pj←Allouer()// Par défaut le nombre d'objet est 1
Fin
```

Remarque 1 :
Dans le programme :

```
Algorithme Remarque1
Var
  i : Entier
  p : *Entier
Début
  p←&i
Fin
```

i et *p sont identiques et on n'a pas besoin d'allocation dynamique puisque l'espace mémoire à l'adresse &i est déjà réservé pour un entier.
Enfin, lorsque l'on n'a plus besoin de l'espace mémoire allouée dynamiquement c'est-à-

dire quand on n'utilise plus le pointeur p, il faut libérer cette place en mémoire. Ceci se fait à l'aide de l'instruction Libérer qui a pour syntaxe :

```
Libérer(nom_du_pointeur)
```

Libération de la mémoire : la fonction Libérer

```
Algorithme Exemple1
Var
  pi : *Entier
  pr : Réel
Début
  //réserver un espace pour 4 entiers
  pi←Allouer(4)
  //réserver un espace pour 6 réels
  pr←Allouer(6)
  ...
  //libèrer la place précédemment réservée pour pi
  Libérer(pi)
  //libèrer la place précédemment réservée pour pr
  Libérer(pr)
Fin
```

2.5 Pointeurs et tableaux

Tout tableau est un pointeur constant. Dans la déclaration :

```
tab : Tableau[1..10]d'entier
p : *Entier
p←tab//ou p←&tab[0] adresse du premier élément
```

Les écritures suivantes sont équivalentes : tab[3] ou p[3] ou *(p+3) **tab** est un pointeur constant non modifiable dont la valeur est l'adresse du premier élément du tableau. Autrement dit **tab** a pour valeur &tab[0]. On peut donc utiliser un pointeur initialisé à **tab** pour parcourir les éléments du tableau.
Exemple :

```
tab : Tableau[1..5]d'entier
p : *Entier  i : Entier ...
p←tab
Pour i de 1 à 5 faire
  Ecrire(*p)
  p←p+1
Fin Pour
```

Qui est équivalent à :

```
tab : Tableau[1..5]d'entier
p : *Entier  i : Entier ...
p←tab
Pour i de 1 à 5 faire
| Ecrire(p[i])
Fin Pour
```

Qui est équivalent à :

```
tab : Tableau[1..5]d'entier
p : *Entier ...
p←tab
Pour p de tab à tab+4 faire
| Ecrire(*p)
Fin Pour
```

2.6 Tableau de pointeurs

La déclaration d'un tableau de pointeur se fait comme suit :

```
tab : Tableau[1..N]de *Type
```

C'est la déclaration d'un tableau ***NomTableau*** de N pointeurs sur des données du type ***Type***.
Exemple :

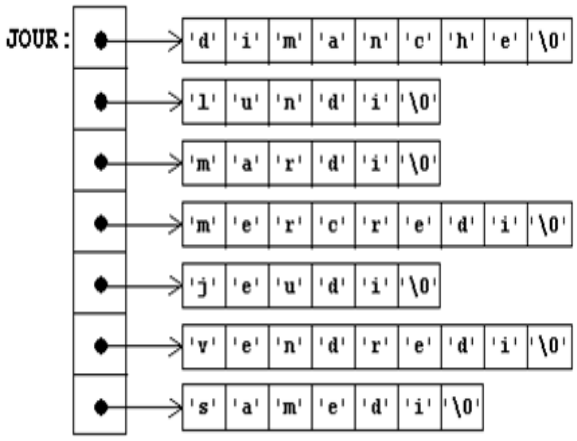


FIGURE 2.2 – Exemple d'un tableau de pointeurs.

3. Les listes chaînées

3.1 La notion de structure auto-référentielle

Une structure auto-référentielle (parfois appelée structure récursive) correspond à une structure dont au moins l'un de ses champs contient un pointeur vers une structure de même type. De cette façon on crée des éléments (appelés cellules) contenant des données, mais, contrairement à un tableau, celles-ci peuvent être éparpillées en mémoire et reliées entre elles par des liens logiques (des pointeurs), c'est-à-dire un ou plusieurs champs dans chaque structure contenant l'adresse d'une ou plusieurs structures de même type.

- Lorsque la structure contient des données et un pointeur vers la structure suivante on parle de liste chaînée
- Lorsque la structure contient des données, un pointeur vers la structure suivante, et un pointeur vers la structure précédente on parle de liste chaînée double
- Lorsque la structure contient des données, un pointeur vers une première structure suivante, et un pointeur vers une seconde, on parle d'arbre binaire

3.2 Définition d'une liste chaînée

Une liste chaînée est un ensemble d'éléments dont chaque élément est une structure comportant des champs contenant des données et un pointeur vers une structure de même type. Ainsi, la structure correspondante à une liste chaînée contenant une chaîne de 15 caractères et un entier peut être définie comme suit :

```
Nom_de_la_structure: Enregistrement
    CH : Chaine[16]
    N : Entier
    Suivant : *Nom_de_la_structure
Fin Enreg
```

En réalité la déclaration de la structure et la récursivité de celle-ci grâce à des pointeurs est nécessaire car cela crée une chaîne d'enregistrements liés par des liens logiques, mais cela n'est pas suffisant. En effet, il est nécessaire de conserver une "trace" du premier enregistrement afin de pouvoir accéder aux autres, c'est pour cela un pointeur vers le premier élément de la liste est indispensable. Ce pointeur est appelé pointeur de tête. D'autre part, étant donné que le dernier enregistrement ne pointe vers rien, il est nécessaire de donner à son pointeur la valeur NULL. Pour créer une liste chaînée, il s'agit dans un premier temps de définir la structure de données, ainsi qu'un enregistrement Liste contenant, au moins, un pointeur vers une structure du type de celle définie précédemment, afin de pointer vers la tête de la liste, c'est-à-dire le premier enregistrement.

```
Cellule: Enregistrement
    CH : Chaine[16]
    Suiv : *Cellule
Fin Enreg
Liste: Enregistrement
    Tete : *Cellule
Fin Enreg
```

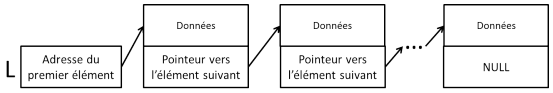


FIGURE 3.1 – Structure d'une liste chaînée.

3.3 Principales opérations pour une liste simplement chaînée

Dans ce paragraphe, nous essayons de présenter les opérations de base dont on a besoin pour travailler avec une liste simplement chaînée. On prend comme exemple une liste simplement chaînées d'entiers. Pour utiliser ces module avec d'autre type de données de la liste, il suffit de remplacer le type entier par le type souhaité. Les structures nécessaires sont les suivantes :

```
Cellule: Enregistrement
    Val : Entier
    Suiv : *Cellule
Fin Enreg
Liste: Enregistrement
    Tete : *Cellule
Fin Enreg
```

Remarque : Une liste simplement chaînée doit être identifiée, au moins par l'adresse de son premier élément au quel on peut ajouter d'autre informations tel que l'adresse du derniers élément, nombre des éléments, etc.

3.3.1 Création d'une liste vide

```
Procédure Init( L : *Liste))
Début
| L→Tete←NULL
Fin
```

Algorithme 1: Création d'une liste vide

3.3.2 Vérification si une liste est vide

```
Fonction Vide( L : Liste)) : Booleen
Début
| Retourner(L.Tete=NULL)
Fin
```

Algorithme 2: Vérification si une liste est vide

3.3.3 Affichage des éléments d'une liste

```
Procédure Afficher( L : Liste))
Var
   p : *Cellule
Début
| p←L.Tete
| Tant que (p<>NULL) faire
| | Ecrire(p→Val)
| | p←p→Suiv
| Fin Tq
Fin
```

Algorithme 3: Affichage des éléments d'une liste

3.3.4 Déterminer la taille d'une liste

```
Fonction Taille( L : Liste)) : Entier
Var
   p : *Cellule  nb : Entier
Début
| nb←0
| p←L.Tete
| Tant que (p<>NULL) faire
| | nb←nb+1
| | p←p→Suiv
| Fin Tq
| Retourner(nb)
Fin
```

Algorithme 4: Taille d'une liste

3.3.5 Recherche d'un élément dans une liste

```
Fonction Recherche( L : Liste ;  X : Entier) : *Cellule
Var
   p : *Cellule
Début
  p←L.Tete
  Tant que (p<>NULL ET p→Val<>X) faire
    p←p→Suiv
  Fin Tq
  Retourner(p)
Fin
```

Algorithme 5: Recherche d'un élément dans une liste

3.3.6 Ajout d'un élément en tête

Une fois la structure et les pointeurs définis, il est possible d'ajouter un premier maillon(cellule) à la liste chaînée, puis de l'affecter au pointeur Tête. Pour cela il est nécessaire :

1. d'allouer la mémoire nécessaire au nouveau maillon grâce à la fonction Allouer.
2. d'assigner au champ "pointeur" du nouveau maillon, la valeur du pointeur vers le maillon de tête.
3. définir le nouveau maillon comme maillon de tête.

```
Procédure Ajout_Tete( L : *Liste ;  X : Entier)
Var
   q : *Cellule
Début
  q←Allouer(1)
  q→Val←X
  q→Suiv←L→Tete
  L→Tete←q
Fin
```

Algorithme 6: Ajout d'un élément en tête d'une liste

3.3.7 Ajout d'un élément en queue

L'ajout d'un élément à la fin de la liste chaînée est similaire, à la différence près qu'il faut définir un pointeur (appelé généralement pointeur courant) afin de parcourir la liste jusqu'à atteindre le dernier maillon (celui dont le pointeur possède la valeur NULL). Les étapes à suivre sont donc :

1. la définition d'un pointeur courant :
2. le parcours de la liste chaînée jusqu'au dernier nœud :
3. l'allocation de mémoire pour le nouvel élément :
4. faire pointer le pointeur courant vers la nouvelle cellule, et la nouvelle cellule vers NULL :

```
Procédure Ajout_Queue( L : *Liste ;  X : Entier)
Var
   p,q : *Cellule
Début
  Si (Vide(*L)=vrai) Alors
  | Ajout_Tete(L,X)
  Sinon
  | p←L→Tete
  | Tant que (p→Suiv<>NULL) faire
  | | p←p→Suiv
  | Fin Tq
  | q←Allouer(1)
  | q→Val←X
  | q→Suiv←NULL
  | p→Suiv←q
  Fin Si
Fin
```

Algorithme 7: Ajout d'un élément en queue d'une liste

3.3.8 Ajout d'un élément à une position donnée

Pour l'ajout à une position donnée, on suppose que la position est valide c'est à dire qu'elle est comprise entre 1 et Taille de la liste +1.

```
Procédure Ajout_Pos( L : *Liste ;  X,k : Entier)
Var
   p,q : *Cellule  i : Entier
Début
  Si (Vide(*L)=vrai OU k=1) Alors
  | Ajout_Tete(L,X)
  Sinon
  | p←L→Tete
  | Tant que (p→Suiv<>NULL et i<=k-2) faire
  | | p←p→Suiv
  | Fin Tq
  | q←Allouer(1)
  | q→Val←X
  | q→Suiv←p→Suiv
  | p→Suiv←q
  Fin Si
Fin
```

Algorithme 8: Ajout d'un élément à une position donnée

3.3.9 Suppression de l'élément en tête

```
Procédure Supp_Tete( L : *Liste)
Var
   q : *Cellule
Début
  Si (Vide(*L)=Faux) Alors
  | q←L→Tete
  | L→Tete←L→Tete→Suiv
  | Liberer(q)
  Fin Si
Fin
```

Algorithme 9: Suppression de l'élément en tête d'une liste

3.3.10 Suppression de l'élément en queue

```
Procédure Supp_Queue( L : *Liste)
Var
   p,q : *Cellule
Début
  Si (Vide(*L)=Faux) Alors
  | Si (L→Tete→Suiv=NULL) Alors
  | | Supp_Tete(L)
  | Sinon
  | | p←L→Tete
  | | Tant que (p→Suiv→Suiv<>NULL) faire
  | | | p←p→Suiv
  | | Fin Tq
  | | q←p→Suiv
  | | p→Suiv←p→Suiv→Suiv
  | | Liberer(q)
  | Fin Si
  Fin Si
Fin
```

Algorithme 10: Suppression de l'élément en queue d'une liste

3.3.11 Suppression d'un l'élément donné

```
Procédure Supp_Elt( L : *Liste ;  X : Entier)
Var
   p,q : *Cellule
Début
 q←Recherche(*L,X)
 Si (q<>NULL) Alors
  p←L→Tete
  Tant que (p→Suiv<>q) faire
   p←p→Suiv
  Fin Tq
  p→Suiv←p→Suiv→Suiv
  Liberer(q)
 Fin Si
Fin
```

Algorithme 11: Suppression d'un élément donné d'une liste

3.3.12 Suppression de l'élément d'une position donnée

Pour l'ajout à une position donnée, on suppose que la position est valide c'est à dire qu'elle est comprise entre 1 et Taille de la liste.

```
Procédure Supp_Pos( L : *Liste)
Var
   p,q : *Cellule  i : Entier
Début
 Si (Vide(*L)=Faux) Alors
  Si (k=1) Alors
   Supp_Tete(L)
  Sinon
   p←L→Tete
   Tant que (p→Suiv→Suiv<>NULL ET i<k-1)
   faire
    p←p→Suiv
   Fin Tq
   q←p→Suiv
   p→Suiv←p→Suiv→Suiv
   Liberer(q)
  Fin Si
 Fin Si
Fin
```

Algorithme 12: Suppression de l'élément à une position donnée dans une liste

3.4 Liste doublement chaînée

Une liste doublement chaînée est basée sur le même principe que la liste chaînée simple, à la différence près qu'elle contient non seulement un pointeur vers le maillon suivant mais aussi un pointeur vers le maillon précédent, permettant de cette façon le parcours de la liste dans les deux sens. Ainsi, la structure correspondante à une liste chaînée double contenant une chaîne de 15 caractères et un entier peut être définie comme suit :

```
Cellule: Enregistrement
    CH : Chaine[16]
    N : Entier
    Suiv : *Cellule
    Prec : *Cellule
Fin Enreg
Liste: Enregistrement
    Tete : *Cellule
Fin Enreg
```

Remarques : Contrairement à une liste simplement chaînée qui doit être identifiée par l'adresse de son premier élément, une liste doublement chaînée peut être représentée par l'adresse de n'importe quel élément de la liste.
On représente généralement cette structure de la manière suivante :

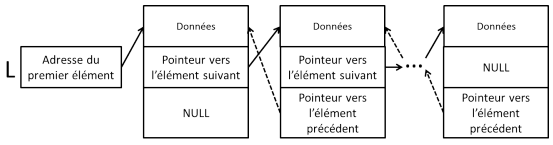


FIGURE 3.2 – Structure d'une liste doublement chaînée.

3.5 Principales opérations pour une liste doublement chaînée

Dans ce paragraphe, nous essayons de présenter les opérations de base dont on a besoin pour travailler avec une liste doublement chaînée. On prend comme exemple une liste doublement chaînées d'entiers. Pour utiliser ces modules avec d'autre types de données de la liste, il suffit de remplacer le type entier par le type souhaité. Les structures nécessaires sont les suivantes :

```
Cellule: Enregistrement
  Val : Entier
  Suiv : *Cellule
  Prec : *Cellule
Fin Enreg
Liste: Enregistrement
  Tete : *Cellule
Fin Enreg
```

Remarques : Dans ce qui suit, on va développer les modules qui présentent des différences par rapport à ceux avec les listes simplement chaînées et qui sont les différents modules d'ajout et de suppression. Le reste des modules sont identiques à ceux avec les listes simplement chaînées.

3.5.1 Ajout d'un élément en tête

```
Procédure Ajout_Tete( L : *Liste ;  X : Entier)
Var
   q : *Cellule
Début
  q←Allouer(1)
  q→Val←X
  q→Suiv←L→Tete
  q→Prec←NULL
  L→Tete→Prec←q
  L→Tete←q
Fin
```

Algorithme 13: Ajout d'un élément en tête d'une liste double

3.5.2 Ajout d'un élément en queue

```
Procédure Ajout_Queue( L : *Liste ;  X : Entier)
Var
   p,q : *Cellule
Début
 Si (Vide(*L)=vrai) Alors
 | Ajout_Tete(L,X)
 Sinon
 | p←L→Tete
 | Tant que (p→Suiv<>NULL) faire
 | | p←p→Suiv
 | Fin Tq
 | q←Allouer(1)
 | q→Val←X
 | q→Suiv←NULL
 | q→Prec←p→Suiv
 | p→Suiv←q
 Fin Si
Fin
```

Algorithme 14: Ajout d'un élément en queue d'une liste double

3.5.3 Ajout d'un élément à une position donnée

Pour l'ajout à une position donnée, on suppose que la position est valide c'est à dire qu'elle est comprise entre 1 et Taille de la liste +1.

```
Procédure Ajout_Pos( L : *Liste ;  X,k : Entier)
Var
   p,q : *Cellule  i : Entier
Début
 Si (Vide(*L)=vrai OU k=1) Alors
 | Ajout_Tete(L,X)
 Sinon
 | p←L→Tete
 | Tant que (p→Suiv<>NULL et i<=k-2) faire
 | | p←p→Suiv
 | Fin Tq
 | q←Allouer(1)
 | q→Val←X
 | q→Suiv←p→Suiv
 | q→Prec←p
 | p→Suiv→Prec←q
 | p→Suiv←q
 Fin Si
Fin
```

Algorithme 15: Ajout d'un élément à une position donnée dans une liste double

3.5.4 Suppression de l'élément en tête

```
Procédure Supp_Tete( L : *Liste)
Var
   q : *Cellule
Début
 Si (Vide(*L)=Faux) Alors
  q←L→Tete
  L→Tete←L→Tete→Suiv
  Si (L→Tete<>NULL) Alors
   L→Tete→Prec←NULL
  Fin Si
  Liberer(q)
 Fin Si
Fin
```

Algorithme 16: Suppression de l'élément en tête d'une liste double

3.5.5 Suppression de l'élément en queue

```
Procédure Supp_Queue( L : *Liste)
Var
   p,q : *Cellule
Début
 Si (Vide(*L)=Faux) Alors
  Si (L→Tete→Suiv=NULL) Alors
   Supp_Tete(L)
  Sinon
   p←L→Tete
   Tant que (p→Suiv→Suiv<>NULL) faire
    p←p→Suiv
   Fin Tq
   q←p→Suiv
   p→Suiv←p→Suiv→Suiv
   Liberer(q)
  Fin Si
 Fin Si
Fin
```

Algorithme 17: Suppression de l'élément en queue d'une liste double

3.5.6 Suppression d'un l'élément donné

```
Procédure Supp_Elt( L : *Liste ;  X : Entier)
Var
   p,q : *Cellule
Début
 q←Recherche(*L,X)
 Si (q<>NULL) Alors
  p←L→Tete
  Tant que (p→Suiv<>q) faire
   p←p→Suiv
  Fin Tq
  Si (p→Suiv→Suiv<>NULL) Alors
   p→Suiv→Suiv→Prec←p
  Fin Si
  p→Suiv←p→Suiv→Suiv
  Liberer(q)
 Fin Si
Fin
```

Algorithme 18: Suppression d'un élément donné d'une liste double

3.5.7 Suppression de l'élément d'une position donnée

Pour l'ajout à une position donnée, on suppose que la position est valide c'est à dire qu'elle est comprise entre 1 et Taille de la liste.

```
Procédure Supp_Pos( L : *Liste ; k : Entier)
Var
   p,q : *Cellule  i : Entier
Début
  Si (Vide(*L)=Faux) Alors
    Si (k=1) Alors
      Supp_Tete(L)
    Sinon
      p←L→Tete
      Tant que (p→Suiv→Suiv<>NULL ET i<k-1)
      faire
        p←p→Suiv
      Fin Tq
      q←p→Suiv
      Si (p→Suiv→Suiv<>NULL) Alors
        p→Suiv→Suiv→Prec←p
      Fin Si
      p→Suiv←p→Suiv→Suiv
      Liberer(q)
    Fin Si
  Fin Si
Fin
```

Algorithme 19: Suppression de l'élément à une position donnée dans une liste double

4. Les piles et les files

4.1 Introduction

Les notions de pile et de file sont deux stratégies de manipulation des structures de données regroupant un ensemble de données tel que les tableaux et les listes chaînées.

4.2 La notion de pile

4.2.1 Caractéristiques

Lorsqu'une structure de données est manipulée selon la stratégie Pile, alors les deux opérations d'ajout et de récupération se font uniquement en tête de la pile, appelé aussi le sommet de la pile. Autrement dit, le seul élément accessible est le premier élément.

4.2.2 Principales opérations

Dans cette section on présente un aperçu sur les principales opération sur une pile. Pour développer ces module, on a choisit de manipuler une liste simplement chaînée selon la stratégies pile.

Structures nécessaires

```
Cellule: Enregistrement
  Val : Entier
  Suiv : *Cellule
Fin Enreg
Pile: Enregistrement
  Sommet : *Cellule
Fin Enreg
```

4.2.3 Initialisation d'une pile vide

```
Procédure Init( P : *Pile))
Début
| P→Sommet←NULL
Fin
```

Algorithme 20: Initialisation d'une pile vide

4.2.4 Vérification si une pile est vide

```
Fonction Vide( P : Pile)) : Booleen
Début
| Retourner(P.Sommet=NULL)
Fin
```

Algorithme 21: Vérification si une pile est vide

4.2.5 Ajout d'un élément : Empiler

L'opération d'ajout d'un élément à une pile est appelée **Empiler**.

```
Procédure Empiler( P : *Pile ;  X : Entier)
Var
   q : *Cellule
Début
| q←Allouer(1)
| q→Val←X
| q→Suiv←P→Sommet
| P→Sommet←q
Fin
```

Algorithme 22: Ajout d'un élément : Empiler

4.2.6 Récupération d'un élément de la pile : Dépiler

L'opération de récupération d'un élément de la pile consiste à récupérer la valeur du sommet en le supprimant de la pile. Cette opération est appelée **Dépiler**. La fonction dépiler ne peut être appelée que lorsque la pile est non vide

```
Fonction Dépiler( P : *Pile) : Entier
Var
   q : *Cellule  X : Entier
Début
  X←P→Sommet→Val
  q←P→Sommet
  P→Sommet←P→Sommet→Suiv
  Liberer(q)
  Retourner(X)
Fin
```

Algorithme 23: Récupération d'un élément de la pile : Dépiler

4.3 La notion de file

4.3.1 Caractéristiques

Lorsqu'une structure de données est manipulée selon la stratégie File, alors l'opération d'ajout n'est possible qu'en queue de la file et la récupération d'un élément n'est possible qu'en tête de la file. Autrement dit, les seuls éléments accessibles sont le premier élément pour la récupération et la queue pour l'ajout. D'où la nécessité de sauvegarder les adresses du premier et dernier élément d'une file.

4.3.2 Principales opérations

Dans cette section on présente un aperçu sur les principales opération sur une file. Pour développer ces module, on a choisit de manipuler une liste simplement chaînée selon la stratégies file.

Structures nécessaires

```
Cellule: Enregistrement
  Val : Entier
  Suiv : *Cellule
Fin Enreg
Pile: Enregistrement
  Tête : *Cellule
  Queue : *Cellule
Fin Enreg
```

4.3.3 Initialisation d'une File vide

```
Procédure Init( F : *File))
Début
  F→Tête←NULL
  F→Queue←NULL
Fin
```

Algorithme 24: Initialisation d'une file vide

4.3.4 Vérification si une file est vide

```
Fonction Vide( F : File)) : Booleen
Début
| Retourner(F.Tête=NULL)
Fin
```

Algorithme 25: Vérification si une file est vide

4.3.5 Ajout d'un élément : Enfiler

L'opération d'ajout d'un élément à une file est appelée **Enfiler**.

```
Procédure Enfiler( F : *File ;  X : Entier)
Var
   q : *Cellule
Début
| q←Allouer(1)
| q→Val←X
| q→Suiv←NULL
| Si (Vide(*F)=Vrai) Alors
| | F→Tête←q
| Sinon
| | P→Queue→Suiv←q
| Fin Si
| P→Queue←q
Fin
```

Algorithme 26: Ajout d'un élément : Enfiler

4.3.6 Récupération d'un élément de la file : Défiler

L'opération de récupération d'un élément de la file consiste à récupérer la valeur la tête en la supprimant de la file. Cette opération est appelée **Défiler**. La fonction dépiler ne peut être appelée que lorsque la file est non vide

```
Fonction Défiler( F : *File) : Entier
Var
   q : *Cellule  X : Entier
Début
| X←F→Tête→Val
| q←F→Tête
| P→Tête←F→Tête→Suiv
| Si (F→Tête=NULL) Alors
| | F→Queue←NULL
| Fin Si
| Liberer(q)
| Retourner(X)
Fin
```

Algorithme 27: Récupération d'un élément de la file : Défiler

5. Les arbres

5.1 Introduction

L'arbre est une structure de données permettant de regrouper un ensemble des éléments dont chacun est précédé par zéro ou un élément et suivi par zéro ou plusieurs éléments. Un élément dans un arbre est appelé **Nœud**. Le premier élément est appelé **Nœud Racine**. Un élément qui ne possède aucun successeur est appelé **Feuille**. Le nombre de niveaux dans un arbre est appelé **Hauteur**. Un exemple d'arbre est présenté par la figure 5.1.

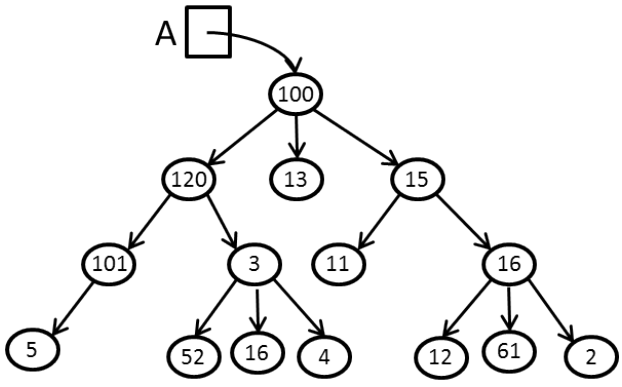


FIGURE 5.1 – Exemple d'un arbre.

5.2 Arbre binaire

5.2.1 Définitions

Un arbre binaire est un arbre dont chaque Nœud possède au maximum deux successeurs. Un exemple d'arbre binaire est présenté par la figure 5.2.

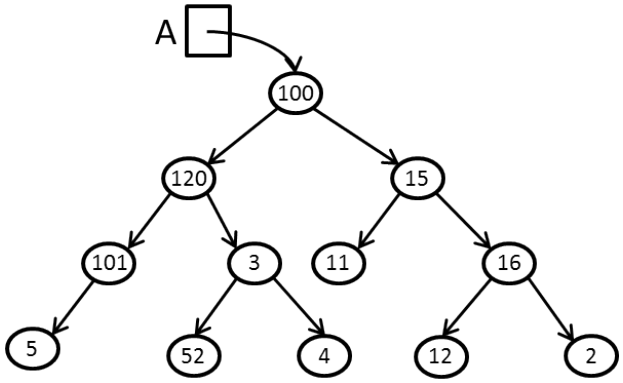


FIGURE 5.2 – Exemple d'un arbre binaire.

5.2.2 Principales opérations

Structures nécessaires

```
Nœud: Enregistrement
  Val : Entier
  FG : *Nœud
  FD : *Nœud
Fin Enreg
ArbreBin: Enregistrement
  Racine : *Nœud
Fin Enreg
```

5.2.3 Création d'un arbre vide

```
Procédure Init( A : *Arbre))
Début
  A→Racine←NULL
Fin
```

Algorithme 28: Création d'un arbre vide

5.2.4 Vérification si un arbre est vide

```
Fonction Vide( A : Arbre)) : Booléen
Début
  Retourner(A.Racine=NULL)
Fin
```

Algorithme 29: Vérification si un arbre est vide

Parcours d'un arbre

On distingues deux type de parcours pour un arbre : Parcours en profondeur et parcours en largeur. Pour le parcours en profondeur, il y a 3 sous-types de parcours : Préfixé, infixé et postfixé. Dans ce qui suit nous détaillons les différents types de parcours et ce pour

afficher les éléments de l'arbre. Pour effectuer n'importe quel autre traitement, il suffit de remplacer l'instruction d'affichage par le traitement souhaité.

Parcours en profondeur préfixé

Parcours en profondeur préfixé consiste à traiter la racine, ensuite les deux fils.

```
Procédure Préfixé( A : Arbre))
Var
   AG,AD : *Arbre
Début
  Si (NON(Vide(A)) Alors
    Ecrire(A→Racine→Val)
    AG→Racine←A→Racine→AG
    AD→Racine←A→Racine→AD    Préfixé(AG)    Pré-
    fixé(AD)
  Fin Si
Fin
```

Algorithme 30: Parcours Préfixé

En appliquant cette procédure sur l'arbre présenté par la figure 5.2, on obtient l'affichage suivant :

100-120-101-5-3-52-4-15-11-16-12-2

Parcours en profondeur infixé

Parcours en profondeur préfixé consiste à traiter le fils gauche, ensuite la racine et en fin le fils droit.

```
Procédure Infixé( A : Arbre))
Var
   AG,AD : *Arbre
Début
  Si (NON(Vide(A)) Alors
    AG→Racine←A→Racine→AG
    AD→Racine←A→Racine→AD          Préfixé(AG)
    Ecrire(A→Racine→Val)
    Préfixé(AD)
  Fin Si
Fin
```

Algorithme 31: Parcours Infixé

En appliquant cette procédure sur l'arbre présenté par la figure 5.2, on obtient l'affichage suivant :

5-101-120-52-3-4-100-11-15-12-16-2

Parcours en profondeur postfixé

Parcours en profondeur postfixé consiste à traiter les deux fils, ensuite la racine.

```
Procédure Préfixé( A : Arbre))
Var
   AG,AD : *Arbre
Début
  Si (NON(Vide(A)) Alors
    AG→Racine←A→Racine→AG
    AD→Racine←A→Racine→AD   Préfixé(AG)   Pré-
    fixé(AD) Ecrire(A→Racine→Val)

  Fin Si
Fin
```

Algorithme 32: Parcours Postfixé

En appliquant cette procédure sur l'arbre présenté par la figure 5.2, on obtient l'affichage suivant :

5-101-52-4-3-120-11-12-2-16-15-100

Parcours en largeur

Le parcours en largeur consiste à parcourir l'arbre par niveau à partir de la racine jusqu'aux les feuilles. Pour ceci, nous utilisons une file de Nœud.

```
Procédure Largeur( A : Arbre)
Var
   F : File  X : Nœud
Début
  Init(&F)
  Si (NON(Vide(A)) Alors
    Enfiler(&F,A→Racine)
    Tant que (Non(Vide(F))) faire
      X←Défiler(&F)
      Ecrire(X.Val)
      Si (X.FG<>NULL) Alors
        Enfiler(&F,*(X.FG))
      Fin Si
      Si (X.FD<>NULL) Alors
        Enfiler(&F,*(X.FD))
      Fin Si
    Fin Tq

  Fin Si
Fin
```

Algorithme 33: Parcours en largeur

En appliquant cette procédure sur l'arbre présenté par la figure 5.2, on obtient l'affichage suivant :

100-120-15-101-3-11-16-5-52-4-12-2

5.3 arbre binaire de recherche

Un arbre binaire de recherche est un arbre binaire dont chaque Nœud vérifie la propriété suivante : Les valeurs de tous les éléments du sous-arbre gauche sont inférieures ou égale à la valeur de la racine et Les valeurs de tous les éléments du sous-arbre droite sont supérieures à la valeur de la racine. Un exemple d'un arbre binaire de recherche est présenté par la figure 5.3.

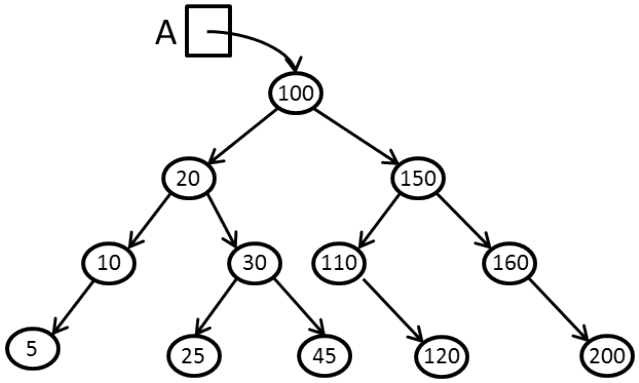


FIGURE 5.3 – Exemple d'un arbre.

II

Travaux Dirigés

TD1 : Les enregistrements 47

TD2 : Les pointeurs 53

TD3 : Les listes chaînées 55

TD4 : Les piles et les files 61

TD5 : Les arbres 63

TD1 : Les enregistrements

Exercice 1

1. Définir une structure "RATIONNEL" sachant qu'un rationnel est caractérisé par son :
   - Numérateur (entier)
   - Dénominateur (entier)
2. On vous demande d'écrire les procédures/fonctions suivantes :
   (a) Une fonction SAISIE_RAT permettant de saisir un rationnel.
   (b) Une procédure SOMME_RAT permettant de calculer et d'afficher la somme de deux nombres rationnels donnés en paramètre.
   (c) Une procédure PRODUIT_RAT permettant de calculer et d'afficher le produit de deux nombres rationnels donnés en paramètre.
   (d) Une procédure INVERSE_RAT permettant d'inverser un nombre rationnel : le dénominateur devient le numérateur et vice versa.
   (e) Une procédure AFFICHE_RAT permettant d'afficher le numérateur et le dénominateur d'un nombre rationnel donné en paramètre.
   (f) Une procédure COMPARAISON_RAT à deux paramètres permettant de comparer et d'afficher si :
      - R1 est supérieur à R2,
      - R1 est inférieur à R2,
      - R1 est égale à R2
3. En utilisant les procédures/fonctions définies ci-dessus, écrire un algorithme principal permettant de :
   - Saisir deux nombre rationnel R1 et R2,
   - Calculer la somme de deux nombres rationnels R1 et R2.
   - Calculer le produit de deux nombres rationnels R1 et R2.
   - inverser et afficher le nombre rationnel R2 avant et après l'inversion.

Exercice 2

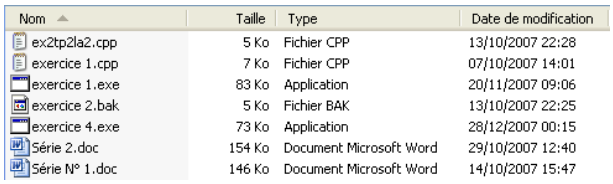
On se propose dans un cet exercice de gérer le stock d'une société qui vend des articles de sport. Sachant que la société a, à sa disposition, 10 articles et un article est une structure possédant :

- un numéro de code (entier)
- un libellé (10 caractères)
- un prix unitaire (réel)
- une quantité en stock (entier)

1. Définir les types Article et société
2. Ecrire une procédure qui ajoute un article à une liste d'articles triés selon l'ordre croissant de leurs codes.
3. Développer une fonction qui cherche un article à travers son code.
4. Ecrire une procédure qui permet de diminuer ou d'augmenter la quantité en stock d'un article par une quantité quelconque. L'article est donné à travers son code.
5. En se basant sur ces modules, développer un algorithme qui affiche et exécute le menu suivant : Recherche d'un article.....Taper 1
   Ajout d'un article.....Taper 2
   Mise à jour de la quantité en stocke.....Taper 3
   Sortie.....Taper 4

Exercice 3

Un fichier est défini par son nom, sa taille, son type et sa date de modification. Soit "OBJETINFO" un type formé des champs indiqués dans l'exemple ci-dessous :



Nom	Taille	Type	Date de modification
ex2tp2la2.cpp	5 Ko	Fichier CPP	13/10/2007 22:28
exercice 1.cpp	7 Ko	Fichier CPP	07/10/2007 14:01
exercice 1.exe	83 Ko	Application	20/11/2007 09:06
exercice 2.bak	5 Ko	Fichier BAK	13/10/2007 22:25
exercice 4.exe	73 Ko	Application	28/12/2007 00:15
Série 2.doc	154 Ko	Document Microsoft Word	29/10/2007 12:40
Série N° 1.doc	146 Ko	Document Microsoft Word	14/10/2007 15:47

1. Définir les structures "DATE" et "OBJETINFO".
2. Ecrire une procédure SAISIE qui permet de charger un tableau T par N éléments de type "OBJETINFO".
3. Ecrire une procédure TROUVE qui permet de chercher et afficher l'objet le plus volumineux dans un tableau T d'OBJETINFO de taille N.
4. Ecrire une procédure INSERER qui permet d'insérer dans un tableau T d'OBJETINFO de taille N, à une position donnée, les informations d'un nouveau élément OBJETINFO.
5. Ecrire une procédure TRI permettant de trier le tableau T dans l'ordre alphabétique selon le champ "nom". (Tri à bulle)
6. Ecrire une procédure AFFICHER qui permet d'afficher les éléments du tableau T.
7. En utilisant les procédures définies ci-dessus, écrire l'algorithme principal permettant de :
   - Saisir la taille du tableau T ($1 < N < 100$)
   - Remplir un tableau T par N éléments de type "OBJETINFO".
   - Afficher l'objet le plus volumineux dans un tableau T d'OBJETINFO.

- Saisir un élément Obj de type OBJETINFO, une position p et insérer le nouvel élément dans T.
- Afficher les éléments du tableau T après l'insertion.
- Afficher les éléments du tableau T après le tri.

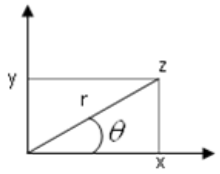
Exercice 4

On donne les fonctions prédéfinies suivantes :

- fonction carre(x : entier) : entier (renvoi la valeur de x au carré)
- fonction rac(x : entier) : entier (renvoi la racine carrée de x)
- fonction sin(x : entier) : entier (renvoi la valeur de sinus de x)
- fonction cos (x : entier) : entier (renvoi la valeur de cosinus de x)
- fonction arctg(x : entier) : entier (renvoi l'arc tangente de x)

Un nombre complexe $z = x + iy$ se compose d'une partie réelle x et d'une partie imaginaire y.

1. Définir la structure "complexe".
2. Ecrire une fonction modul (z : complexe) : réel qui calcule le module d'un nombre complexe z.
3. Ecrire une fonction arg (z : complexe) : réel qui calcule l'argument d'un nombre complexe z.
4. Ecrire une fonction som (z1, z2 : complexe) : complexe qui calcule la somme de deux nombres complexes.
5. Ecrire une fonction prod (z1, z2 : complexe) : complexe qui calcule le produit de deux nombres complexes.
   On définit la forme polaire d'un nombre complexe $z = re^{i\theta}$ (r représente le module de z et représente l'argument de z) par l'enregistrement suivant :
   cpolaire = Enregistrement
   mod : réel
   arg : réel
   Fin



6. Ecrire une fonction Alg2pol (z : complexe) : cpolaire qui transforme un nombre complexe écrit sous forme algébrique en un nombre complexe écrit sous forme polaire.
7. Ecrire un algorithme permettant de :
   - Saisir deux nombres complexes C1 et C2 sous formes algébriques.
   - D'afficher le module et l'argument d'un nombre complexe C1.
   - D'afficher la somme et le produit des 2 nombres complexes
   - D'afficher le nombre complexe C1 de la forme algébrique à la forme polaire.

Exercice 5

Les données relatives à N ouvriers sont rangées dans un tableau d'enregistrements T.(N est un entier pair). Un ouvrier est défini par :

- Nom "nom" : chaîne de 20 caractères au maximum
- Ancienneté "anc" : entier
- Salaire "sal" : réel de 200 à 800 D

1. Définir l'enregistrement "OUVRIER"
2. Écrire une procédure SAISIE permettant de remplir le tableau T par N ouvriers.
   N.B : le salaire est compris entre 200 et 800 D
3. On désire trier ces données par ordre décroissant selon les salaires des ouvriers en appliquant une méthode de tri dont le principe est décrit respectivement comme suit :
   (a) Chercher l'enregistrement ayant le maximum de salaire.
   (b) Ranger cet enregistrement au début du tableau R
   (c) Supprimer cet enregistrement du tableau T
   (d) Chercher l'enregistrement ayant le minimum de salaire
   (e) Ranger cet enregistrement à la fin du tableau R
   (f) Supprimer cet enregistrement du tableau T
   (g) Répéter les six actions précédentes jusqu'à obtenir dans R tous les éléments de T triés.

   On donne la procédure récursive suivante qui supprime un enregistrement d'un tableau T dont la position est P.

```
Procédure SUPPRIMER(T : TAB ; N, p : entier)
Début
  Si (p <= N-1) Alors
    T[p] ←T[p+1]
    SUPPRIMER(T, N, p+1)
  Fin Si
Fin
```

4. Écrire une procédure TRI permettant de trier le tableau T par ordre décroissant des salaires des ouvriers dans un nouvel tableau R en utilisant la procédure "SUPPRIMER"
5. Ecrire une procédure Afficher permettant d'afficher les informations des N ouvriers.
6. Ecrire un algorithme principal permettant de :
   (a) Lire le nombre des ouvriers N (N est pair) ;
   (b) Saisir les informations de ces N ouvriers ;
   (c) Afficher les informations des N ouvriers avant et après le tri.

Exercice 6

Une entreprise contient N employés, chacun est identifié par les champs suivants :

- Nom : chaîne de 30 caractères au maximum
- Téléphone :chaîne de 8 caractères dont les 2 premiers caractères représentent l'indicatif (71, 75, 98, 22, etc)
- Sexe : un caractère
- NbEnfants : entier

- DateEmbauche : struture

On vous demande d'écrire les procédures/fonctions suivantes :

1. Une procédure SAISIE permettant de remplir le tableau TEMP1 par NE employés.
2. Une procédure RECHERCHE1 permettant de chercher et afficher le nombre d'employés de sexe masculin ainsi que celui de sexe féminin.
3. Une procédure RECHERCHE2 permettant de chercher s'il existe un employé recruté avant l'année 2000.
4. Une procédure RECHERCHE3 permettant de chercher et d'afficher les employés ayant plus 3 enfants.
5. Une procédure RECHERCHE4 de chercher et afficher les coordonnées de l'employé ayant le plus grand nombre d'enfants.
6. Une procédure RECHERCHE5 permettant de chercher et afficher les employés ayant des numéros de téléphones palindromes.
7. Une procédure CONSTRUCTION permettant de construire un nouveau tableau TEMP2 comportant tous les enregistrements du tableau TEMP1 dont les numéros commençant par un indicatif IND. (N.B : Si aucun numéro commence par IND n'est trouvé, le message "Aucun numéro n'est trouvé" sera affiché sur l'écran.
8. Un algorithme principal.

Exercice 7

Une entreprise commerciale de produits électroménagers veut mettre en place un système de gestion de sa clientèle. Nous supposons dans cet exercice qu'une entreprise est une collection de 100 clients, dont chacun est caractérisé par ces informations :

- Un nom (Nom) et un prénom (Prenom), dont chacun est une chaîne de 20 lettres ;
- Une adresse (Adresse) qui se compose à son tour des champs suivants :
  - le code postal (Code) (entier) ;
  - la ville de résidence (Ville) (15 caractères) ;
  - le pays (Pays) (15 caractères).
- Un montant d'achats (Montant) du client auprès de l'entreprise (réel).

1. Définir les types de structures Client et Entreprise
2. Ecrire la procédure remplissage (var c : Client) qui remplit les différents champs de la variable c.
3. Ecrire la procédure affiche (c : Client) qui affiche les informations relatives au client c.
4. Ecrire la fonction Achat (E : Entreprise, n : entier) : Client qui permet de retourner le client qui a le plus grand montant d'achat pour une entreprise comportant n clients
5. Ecrire la fonction Tunisie (C : Client) : Booléen qui permet de vérifier si le client c est Tunisien.
6. Ecrire un Algorithme qui permet de :
   - Remplir un tableau de 60 clients,
   - Afficher les informations relatives au client qui a le plus grand montant d'achat, et,
   - Afficher les noms et les prénoms des clients Tunisiens.

TD2 : Les pointeurs

Exercice 1

Soit l'algorithme suivant :

```
Algorithme XXXX
Var
  i, j : Entier
  p, q : *Entier
Début
  i←5
  j←6
  pi←&i
  pj←&j
  écrire(*pi)
  écrire(*pj)
  pj←pi
  écrire(*pi)
  écrire(*pj)
Fin
```

L'algorithme XXXX affiche .....

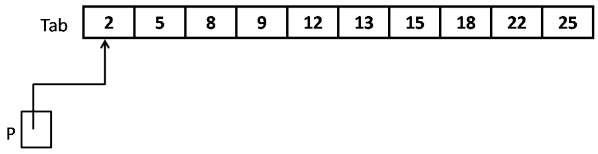
Exercice 2

Remplir le tableau suivant sachant que a, b et c sont des variables de type entier et, p1 et p2 des pointeurs sur entiers.

	a	b	c	p1	p2
Valeurs initiales	10	15	20	NULL	NULL
P1←&a					
P2←&c					
*P1←*P2+1					
P1←P2					
P2←&b					
*P1←*P1-*P2					
*P2←*P2+1					
*P1←*P1**P2					
P1←&a					

Exercice 3

Soit P un pointeur qui pointe sur le tableau Tab :



Quelles valeurs ou adresses fournissent ces expressions :

1. *P+2
2. *(P+2)
3. &P+1
4. &tab[4]-3
5. tab+3
6. &tab[7]-P
7. P+(*P-10)
8. *(P+*(P+8)-tab[7])

Exercice 4

Refaire deux exercices, au choix, du TD6(les tableaux) du premier semestre en utilisant le formalisme pointeur.

Exercice 5

Refaire deux exercices, au choix, du TD1(les structures) en utilisant le formalisme pointeur.

TD3 : Les listes chaînées

Exercice 1

Nous partons de la déclaration ci-après :

```
Type
    Cellule = Enregistrement
        Val : entier
        suiv : *Cellule
    Fin Enreg
    Liste=Enregistrement
        Tête : *Cellule
    Fin Enreg
```

Nous supposons également que nous avons à notre disposition les fonctions et procédures déjà détaillées en cours :

- La procédure **Init** : permettant la création d'une liste vide,
- La procédure **Affiche** : permettant d'afficher les éléments d'une liste,
- La fonction **Taille** : permettant de déterminer la taille d'une liste,
- La fonction **Recherche** : permettant de rechercher la position d'un élément donné,
- La procédure **Ajout_tete** : permettant d'insérer un élément en tête d'une liste,
- La procédure **Ajout_queue** : permettant d'insérer un élément en queue d'une liste,
- La procédure **Ajout_pos** : permettant d'insérer un élément à une position donnée dans une liste,
- La procédure **Supp_tete** : permettant la suppression de l'élément en tête d'une liste,
- La procédure **Supp_queue** : permettant la suppression de l'élément en queue d'une liste,
- La procédure **Supp_elt** : permettant la suppression d'un élément donné,
- La procédure **Supp_pos** : permettant la suppression d'un élément situé à une adresse donnée,

1. Écrire une fonction **nb_occ** permettant de déterminer le nombre d'occurrence d'un entier x dans une liste.
2. Écrire une procédure **Création** qui permet de remplir une liste simplement chaînée par des entiers saisis par l'utilisateur. Un entier est saisi dans une position donnée spécifiée par l'utilisateur. La saisie s'arrête lorsque l'utilisateur répond Non à la question "Voulez vous ajouter un autre entier O/N" (N pour Non, O pour Oui).
3. Écrire une fonction **Déterminer** permettant de retourner le nombre d'éléments distincts d'une liste chaînée donnée.
4. Écrire une procédure **Ajout_Après** permettant d'insérer un élément E donné juste après un élément X s'il existe.
5. Écrire une procédure **Ajout_Avant** permettant d'insérer un élément E donné juste avant un élément X s'il existe.
6. Écrire une procédure **Supp_Après** permettant de supprimer l'élément suivant d'un élément X s'il existe.
7. Écrire une procédure **Supp_Avant** permettant de supprimer l'élément précédent d'un élément X s'il existe.
8. Écrire une procédure **Supp_prem_occ** qui supprime dans une liste la première occurrence d'un élément x donné déjà existant (s'il existe, sinon la procédure ne fait rien).
9. Écrire une procédure **Supp_dern_occ** qui supprime dans une liste la dernière occurrence d'un élément x donné déjà existant (s'il existe, sinon la procédure ne fait rien).
10. Écrire une procédure **Supp_toutes_occ** qui supprime dans une liste toutes les occurrences d'un élément x donné (s'il existe, sinon la procédure ne fait rien).
11. Écrire une fonction **Inverser** permettant d'inverser une liste chaînée Ls dans une liste Ls_Inv.
12. Écrire une fonction **Intersection** qui retourne une troisième liste contenant tous les éléments communs à deux listes données passées en paramètre.
13. Écrire une fonction **Union** qui retourne une troisième liste contenant tous les éléments des deux listes données passées en paramètre.
14. Écrire une procédure **Éclater** qui permet d'éclater la liste L1 d'entiers en deux listes : L2 contenant les éléments pairs et L3 contenant les éléments impairs.
15. Écrire une procédure **Fusion** à trois paramètres L1, L2 et L3 permettant de construire une liste L3 triée par ordre croissant avec les éléments des deux liste L1 et L2 (déjà triées).
16. Écrire une fonction **Compter_pred** qui permet de compter et d'afficher le nombre d'éléments de la liste qui sont supérieurs à leurs prédécesseurs.
17. Écrire une procédure **Tri1** qui permet de trier une liste L1 d'entiers par ordre croissant par l'échange des valeurs contenues des cellules.
18. Écrire une procédure **Tri2** qui permet de trier la liste L1 par ordre croissant par permutation des cellules entièrement (échange des pointeurs)
19. Écrire une procédure **Insertion_triée** qui prend en entrée une liste L qu'on suppose triée en ordre croissant et une valeur val et qui insère val à la bonne place dans la liste (c'est-à-dire de façon à ce que la liste reste triée).

Exercice 2

Le but de cet exercice est d'écrire un algorithme qui permet de mémoriser dans une liste simplement chaînée les informations relatives aux 15 joueurs d'une équipe de rugby et de traiter ses informations.
Un joueur est caractérisé par son nom, son prénom et son poids. Chaque joueur à un numéro de poste (de 1 à 15) qui détermine son rôle dans l'équipe (le numéro 1 est pilier gauche, le numéro 2 est talonneur, etc.).

1. Déclarer les structures de données adéquates permettant de représenter une liste de joueurs.
2. Ecrire une procédure **Créer_liste** qui permet de construire une liste de 15 joueurs
3. Ecrire une fonction **Plus_lourd** permettant de renvoyer l'adresse du joueur le plus lourd de l'équipe.

Exercice 3

Un processus se caractérise par son identificateur et son ordre de priorité (l'ordre de priorité correspond à une valeur entière comprise entre 1 et 5). On se propose de simuler le comportement d'un ordonnanceur de processus qui fonctionne de la manière suivante :

- Tous les processus gérés par l'ordonnanceur sont stockés dans une liste unidirectionnelle (simple chaînage).
- Les processus sont triés dans la liste selon l'ordre de priorité puis l'ordre d'arrivée. Par exemple, un processus P1 qui a un niveau de priorité égal à 1 est plus prioritaire que le processus P2 ayant le niveau de priorité 2 et ainsi de suite. Le processus P1 doit donc apparaître dans la liste avant P2. De plus, si deux processus appartiennent à un même niveau de priorité, celui qui est arrivé le premier doit apparaître avant dans la liste.

1. Proposer une structure algorithmique "processus" permettant de représenter un processus.
2. Définir le type "liste_proc" correspondant à la structure de données permettant de représenter un ensemble de processus (à travers une liste chaînée simple).
3. Écrire une procédure **Ajouter_processus** qui ajoute un processus à la liste à l'emplacement adéquat.
4. Écrire une fonction **Lancer_processus** qui permet de débrancher le processus le plus prioritaire dans la liste L et de retourner son adresse.

Exercice 4

Le codage des polynômes peut être réalisé au moyen d'une liste dynamique, il consiste à stocker dans une liste simplement chaînée ordonnée des monômes. Un monôme est constitué par un coefficient et un degré.

1. Définir les types Monôme et Polynôme
2. Développer les modules suivants :
   (a) La procédure **AffichePoly** qui Permet d'afficher un polynôme.
   (b) La fonction **CopiePolynôme** qui renvoie une copie d'un polynôme passé en paramètre.

(c) La procédure **AjoutMonôme** qui permet d'insérer un monôme dans un polynôme.
(d) La procédure **MultiScalaire** qui permet d'avoir le résultat de la multiplication d'un polynôme par un scalaire.
(e) La procédure **MultiMonôme** qui permet d'avoir le résultat de la multiplication d'un monôme avec un polynôme.
(f) La fonction **SommePoly** qui permet de renvoyer dans un nouveau polynôme le résultat de la somme de deux polynômes.
(g) La fonction **EvaluerPoly** qui permet de retourner la valeur du polynôme pour valeur x donnée.
(h) La fonction **Dérivée** qui permet de renvoyer la dérivée d'un polynôme donné.

Exercice 5

Partie A :

Un club souhaite informatiser la gestion de ses adhérents. Les champs qui constituent la structure d'un adhérent sont les suivants :

- NumLic : de type entier
- NomAd : de type chaîne de caractères
- PréAd : de type chaîne de caractères
- Tab : tableau[10] de tournoi qui présente la liste des tournois auxquels l'adhérent a participé. Un adhérent peut participer à 10 tournois au maximum.

Un tournoi est caractérisé par :

- Code : entier
- NomTour : chaîne de caractères

On voudrait dans cet exercice créer et traiter une liste L simplement chaînée d'adhérents.

1. Définir le type liste_adhérents qui représente une liste simplement chaînée d'adhérents.
2. Ecrire la procédure **Ajouter_début** qui ajoute un adhérent au début d'une liste.
3. Ecrire la fonction **Nombre_joueurs** qui retourne le nombre de joueurs qui ont participé à un tournoi donné.
4. Ecrire une procédure **Supprimer** qui permet de supprimer l'adhérent ayant un numéro de licence donné.

Partie B :

On voudrait changer la structure de la liste de telle façon qu'un adhérent n'a pas un nombre maximum de participations aux tournois. En fait, les participations de chaque adhérent sont représentées par une liste doublement chaînée.

1. Définir les structures nécessaires pour représenter la liste d'adhérents.
2. Refaire la question 3 de la partie A.

Exercice 6

Refaire l'exercice 1 en supposant que la liste est doublement chaînée.

Exercice 7

Refaire l'exercice 1 en supposant que la liste est circulaire.

TD4 : Les piles et les files

Nous partons de la déclaration ci-après :

```
Type
    Cellule = Enregistrement
        Val : entier
        suiv : *Cellule
    Fin Enreg
    Pile=Enregistrement
        Sommet : *Cellule
    Fin Enreg
    File=Enregistrement
        Tête : *Cellule
        Queue : *Cellule
    Fin Enreg
```

Nous supposons également que nous avons à notre disposition les fonctions et procédures déjà détaillées en cours :

- La procédure **Init_Pile** : permettant d'initialiser une pile vide,
- La procédure **Empiler** : permettant d'ajouter un élément en sommet d'une pile,
- La fonction **Dépiler** : permettant d'éliminer le sommet d'une pile en retournant sa valeur.
- La procédure **Init_File** : permettant d'initialiser une file vide,
- La procédure **Enfiler** : permettant d'ajouter un élément en queue d'une file,
- La fonction **Défiler** : permettant d'éliminer la tête d'une file en retournant sa valeur.

Exercice 1

1. Rappeler la définition d'une pile et citer les différences entre "PILE" et "FILE".
2. Définir la structure qui représente une pile d'entiers.

3. Écrire une procédure qui permet d'inverser les éléments d'un tableau d'entiers en utilisant la notion de pile.

Exercice 2

1. Écrire une procédure qui permet d'inverser les éléments d'une pile d'entiers.
2. Écrire une fonction qui retourne une copie d'une pile d'entiers.
3. Écrire une procédure qui permet de placer les éléments pairs d'un tableau avant les éléments impairs.

Exercice 3

On se donne une file d'entiers que l'on voudrait trier avec le plus grand élément en fin de file. Proposer une solution en utilisant 2 piles.

Exercice 4

La notation in-fixée consiste à entourer les opérateurs par leurs opérandes, comme $(a+b)*c$. La notation post-fixée consiste à placer les opérandes devant l'opérateur, comme $a\ b\ +\ c\ *$.
Écrire une fonction qui permet d'évaluer une expression post-fixée valide. Exemple : la fonction renvoie 16 pour l'expression post-fixée $3\ 5\ +\ 2\ *$.

Exercice 5

Soit P1 et P2 deux piles. La pile P1 contient une suite de nombres entiers positifs.
Ecrire un algorithme qui permet de déplacer les entiers de P1 dans P2 de façon à avoir dans P2 tous les nombres pairs au dessus des nombres impairs (utiliser une troisième pile intermédiaire).

TD5 : Les arbres

Nous partons de la déclaration ci-après :

```
Type
    Nœud = Enregistrement
        Val : entier
        FG : *Nœud
        FD : *Nœud
    Fin Enreg
    Arbre=Enregistrement
        Racine : *Nœud
    Fin Enreg
```

Exercice 1

1. Écrire une fonction **Feuille** qui vérifie si un nœud est une feuille ou non.
2. Écrire une fonction **NœudComplet** qui vérifie si un nœud possède exactement 2 fils ou non.

Utilisez ces deux fonctions dans la suite de la série.

Exercice 2

1. Écrire une fonction **Taille** qui retourne le nombre de nœuds d'un arbre binaire.
2. Écrire une fonction **Hauteur** qui retourne la hauteur d'un arbre binaire.
3. Écrire une fonction **NbFeuilles** qui retourne le nombre de feuille d'un arbre binaire.
4. Écrire une fonction **Max** qui retourne la valeur maximale d'un arbre binaire d'entiers.
5. Écrire une fonction **Complet** qui vérifie si un arbre binaire est complet ou non. Un arbre binaire est complet si tous ses nœuds, à part les feuilles, possèdent exactement 2 fils.

Exercice 3

1. Écrire une fonction **Poids** qui calcule la somme des valeurs contenues dans les nœuds d'un arbre binaire.
2. Un arbre binaire de Calder est un arbre tel que pour chaque nœud le poids du sous arbre gauche est égale au poids du sous arbre droit. Définir une fonction **Calder** qui teste si un arbre donné est de Calder ou non.

Exercice 4

1. Écrire les procédures de parcours en profondeurs (préfixe, infixe et postfixe) d'un arbre binaire donné afin d'afficher le contenu de chacun de ses nœuds.
2. Écrire une procédure de parcours en largeur d'un arbre binaire.

Exercice 5

A est un arbre binaire de recherche dont les données sont des entiers.

1. Dessiner A après l'insertion des nombres 100, 20, 30, 150, 110, 10, 25, 45, 160, 200
2. Quelle est la hauteur de A ?
3. Que devient A si on rajoute 5 puis 120 ?
4. Afficher les éléments de A après les 3 parcours en profondeur gauche.
5. Afficher les éléments de A après un parcours en largeur.

Exercice 6

Écrire une procédure qui permet de trier un tableau d'entiers à travers la construction d'un arbre binaire de recherche.

Exercice 7

Écrire les fonctions (ou procédures) qui permettent de :

1. ajouter un nouvel élément à un arbre binaire de recherche.
2. saisir un arbre binaire de recherche de n éléments.
3. vérifier si un arbre binaire est un arbre binaire de recherche ou non.
4. afficher, dans l'ordre décroissant, les éléments d'un arbre binaire de recherche.
5. retourner le maximum d'un arbre binaire de recherche.

Exercice 8

Écrire une fonction qui prend en paramètre un arbre binaire de recherche et un entier. La fonction retourne 1 si l'entier appartient à l'arbre , 0 sinon.

Exercice 9

Pour supprimer un nœud dans un arbre binaire de recherche, on distingue trois cas :

- soit le nœud est une feuille, dans ce cas on le supprime tout simplement

- soit le nœud n'a qu'un seul fils, dans ce cas on remplace ce nœud par son fils
- soit le nœud est complet (possède 2 fils), dans ce cas on remplace ce nœud par le n ?ud qui correspond au plus grand élément de son sous arbre gauche que l'on supprimera ensuite.

Écrire une procédure **SupprimerElément** qui prend en argument un arbre binaire de recherche et un entier, et qui renvoie cet arbre privé de cet entier.

III

Sujets d'examens

Université de Gabès
Faculté des Sciences de Gabès
Département Informatique

A.U: 2014-2015

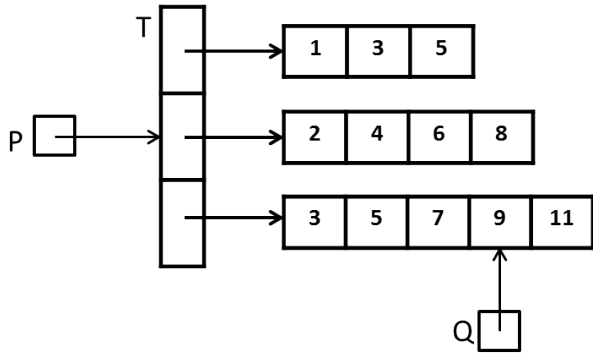
Examen

Section: LFSI1 **Épreuve d':** Algorithmique et structures de données II

Nature de l'épreuve:	DC☑ DS☐ EF☐		Documents:	autorisés	☐	non autorisés	☑
Date de l'épreuve:	01/04/2015		Calculatrice:	autorisée	☐	non autorisée	☐
Durée de l'épreuve:	1H00		Session:	principale	☑	contrôle	☐

Exercise 1 : (10 pts)

Soit T un tableau de 3 adresses, P et Q deux pointeurs.



1. Donner l'algorithme permettant de créer et remplir les structures de données représentées par la figure ci-dessus.
2. Quelles valeurs ou adresses fournissent ces expressions :
   (a) T
   (b) P
   (c) Q
   (d) *T[1]
   (e) **P
   (f) *Q-3
   (g) *(Q-3)
   (h) Q-3
   (i) *P+1
   (j) *(P+1)
   (k) **P+1
   (l) *(*P+1)
   (m) **(P+1)
   (n) *(Q-**(P-1))
   (o) Q+(P-M)

(p) *(*P+(Q-*(P+1)))

Exercise 2 : (10 pts)

N.B :

- Utiliser le formalisme pointeur pour résoudre cet exercice.
- Vous pouvez ajouter les fonctions/procédures que vous jugez nécessaires pour la résolution de l'exercice.

Pour gérer l'emploi du temps d'un établissement, on introduit trois tableaux :

- **TCours**(un tableau de cours) : chaque cours est caractérisé par son code(un entier), et le nombre des étudiants inscrits à ce cours.
- **TSalles**(un tableau de salles) : chaque salle est caractérisée par son numéro(un entier) et sa capacité(nombre de places).
- **TAffectation**(un tableaux des affectations) : Ce tableau permet d'associer à chaque salle, la liste des cours pouvant y être affectés. On ne peut affecter plus de 5 cours à une salle. En plus, on ne peut affecter une salle à un cours que si sa capacité est supérieure ou égale au nombre d'inscrits à ce cours. Chaque élément du tableau TAffectation est composé d'un numéro de salle et les codes des cours affectés à cette salle.

1. Définir les enregistrements nécessaires.
2. Ecrire une procédure Remplir1 permettant de remplir le tableau TCours.
3. Ecrire une procédure Remplir2 permettant de remplir le tableau TSalles.
4. Ecrire une fonction Recherche permettant de retourner le numéro de la salle ayant la capacité minimale qui peut être affectée à un cours donné.
5. Ecrire une procédure Affectation permettant de remplir le tableau TAffectation.
6. Ecrire une procédure Affiche permettant d'afficher l'emploi du temps.
7. Ecrire l'algorithme principale permettant de générer et afficher un emploi du temps.

Bon Travail

Université de Gabès
Faculté des Sciences de Gabès
Département Informatique

A.U: 2015-2016

Examen

Section: LFSI1 **Épreuve d':** Algorithmique et structures de données II

Nature de l'épreuve:	DC☑	DS☐	EF☐	Documents:	autorisés	☐	non autorisés ☑
Date de l'épreuve:	06/04/2016			Calculatrice:	autorisée	☐	non autorisée ☐
Durée de l'épreuve:	01h00			Session:	principale	☑	contrôle ☐

Exercice 1 (9pts)

Un annuaire d'entreprise est composé par "N" entreprises. Chaque entreprise possède un code, un nom, au moins une adresse, et un ensemble d'employés. Chaque adresse est composée d'une ville, un pays et un numéro de téléphone. Chaque employé a un nom, un prénom, un numéro portable et un taux d'absence.

1. Donner les différentes structures de données nécessaires en utilisant la notion des pointeurs. (3 pts)
2. Donner la définition de la procédure qui permet d'ajouter un employé à une entreprise donnée. (1 pt)
3. Donner la définition de la procédure qui permet de supprimer l'employé dont le taux d'absences dépasse le 13%. (1 pt)
4. Donner la définition de la procédure qui permet d'ajouter une entreprise dans la bonne position dans un annuaire sachant que les entreprises sont triées selon leurs codes. (1 pt)
5. Donner la définition de la procédure qui permet de supprimer une entreprise dont le nombre des employés est inférieurs à 5. (1 pt)
6. Donner la procédure qui permet de remplir un annuaire (1 pt)
7. Donner la définition de la procédure qui permet de fusionner deux annuaires sachant que les entreprises sont triées par code. (1 pt)

Exercice 2 (11pts)

Soient les structures suivantes :
$Produit$: $enregistrement$
Nom : $chaine$[20]
$Quantite$: $reel$
$Prix\_Vente\_Entreprise$: $reel$
$Date\_Fin$: $Date$
$Four$: $Fournisseur$
$Finenreg$
Date : enregistrement
Jours : entier
Mois : entier

Année : entier
Fin enreg

Magasin : enregistrement
Prod :* Produit
nb : entier
Fin enreg

Fournisseur : enregistrement
Nom : chaine [20]
Prenom : chaine [20]
Prix_Vente_Fournisseur : réel
Fin enreg

1. Donner les procédures nécessaires permettant d'ajouter un nouveau produit dans un magasin sachant que les produits sont triés selon leur date de fin et que l'ajout d'un nouveau produit implique l'ajout d'un nouveau fournisseur. (3 pts)
2. Donner la fonction permettant de calculer le gain du magasin (1.5 pts)
3. Donner les procédures permettant la suppression du fournisseur qui a le prix de vente le plus élevé sachant que la suppression d'un fournisseur implique la suppression de son produit. (3 pts)
4. Supprimer les produits dont la date de validation est terminée (1.5 pts)
5. Afficher toutes les informations d'un magasin (2 pts)

Bon Travail

Université de Gabès
Faculté des Sciences de Gabès
Département Informatique

A.U: 2015-2016

Examen

Section: LARI1 **Épreuve d':** Algorithmique et structures de données II

Nature de l'épreuve:	DC☑ DS☐ EF☐		Documents:	autorisés	☐	non autorisés	☑
Date de l'épreuve:	05/04/2016		Calculatrice:	autorisée	☐	non autorisée	☐
Durée de l'épreuve:	01h00		Session:	principale	☑	contrôle	☐

Exercice 1 (6 pts)

Soient le tableau t | 1 | 2 | 3 | et les variables :
a, b : entier
p : *entier
Déterminer le résultat d'exécution de chacune des instructions suivantes :
a←10
b←5
t[1]←a
a←(*t)+1
(t + 1)←b
p←t + 2
p←*(p - 1)
p←p - 2
b←*p
t[2]←p - t
t[3]←* (t- *(t+1) + 1)
écrire (a, b, *p, t[1], t[2], t[3])

Exercice 2 (6 pts)

1. Ecrire la fonction contigus (ch : chaine) qui détermine si la chaîne ch donnée en paramètre, comporte deux caractères contigus (successifs) identiques. La fonction retourne 1 s'il existe dans la chaîne ch deux caractères identiques contigus, et 0 sinon.
   Exemples :
   - contigus("masse") a pour résultat 1
   - contigus("escalier") a pour résultat 0
   - contigus("b") a pour résultat 0
2. Ecrire l'algorithme principal VERIF permettant de saisir une chaîne de caractères ch (de longueur maximale 20 caractères), puis détermine et affiche le résultat d'application de la fonction contigus sur ch.

Exercice 3 (8 pts)

On suppose que les informations relatives aux livres gérés dans la bibliothèque de la faculté sont sauvegardées dans un tableau BIBL[100]. Chaque livre est identifié par un code entier (cl), un titre (ti),

le nom de l'auteur (na) et le nombre de fois d'emprunts de ce livre (nem) (ce nombre est incrémenté suite à chaque opération d'emprunt).

1. Développez les modules suivants :
   - la procédure modifliv(...) qui, étant donné le code d'un livre, permet de modifier le livre correspondant dans le tableau BIBL suite à une opération d'emprunt.
   - la procédure affichliv(...) permettant d'afficher les codes et les titres des livres dont le nombre d'emprunts est strictement inférieur à 10.
   - la fonction nbaut(...) permettant de déterminer puis retourner le nombre de livres rédigés par un auteur donné en paramètre.
   - la fonction plusemp(...) permettant de retourner le code du livre le plus demandé par les étudiants de la faculté.
2. Ecrire l'algorithme principal GERER_BIBLIO permettant de :
   - mettre à jour le tableau BIBL suite à l'emprunt d'un livre
   - afficher les codes ainsi que les titres des livres empruntés moins que 10 fois.
   - afficher le nombre de livres rédigés par un auteur donné.
   - afficher le titre du livre le plus demandé dans la bibliothèque.

Bon Travail

Université de Gabès
Faculté des Sciences de Gabès
Département Informatique

A.U: 2014-2015

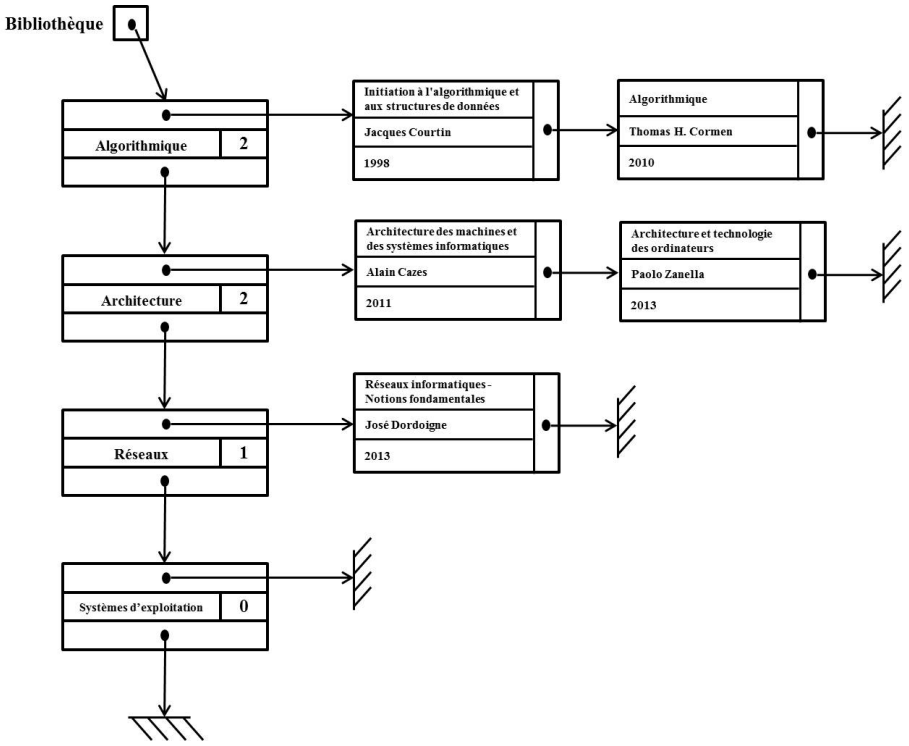
Examen

Section: LFSI1 **Épreuve d':** Algorithmique et structures de données II

Nature de l'épreuve:	DC☐ DS☐ EF☑		Documents:	autorisés	☐	non autorisés	☑
Date de l'épreuve:	25/05/2015		Calculatrice:	autorisée	☐	non autorisée	☐
Durée de l'épreuve:	1H30		Session:	principale	☑	contrôle	☐

Exercice 1 *(10 pts)*

Un étudiant en informatique souhaite représenter sa bibliothèque personnelle en utilisant une structure dynamique. La structure proposée est représentée par la figure suivante :

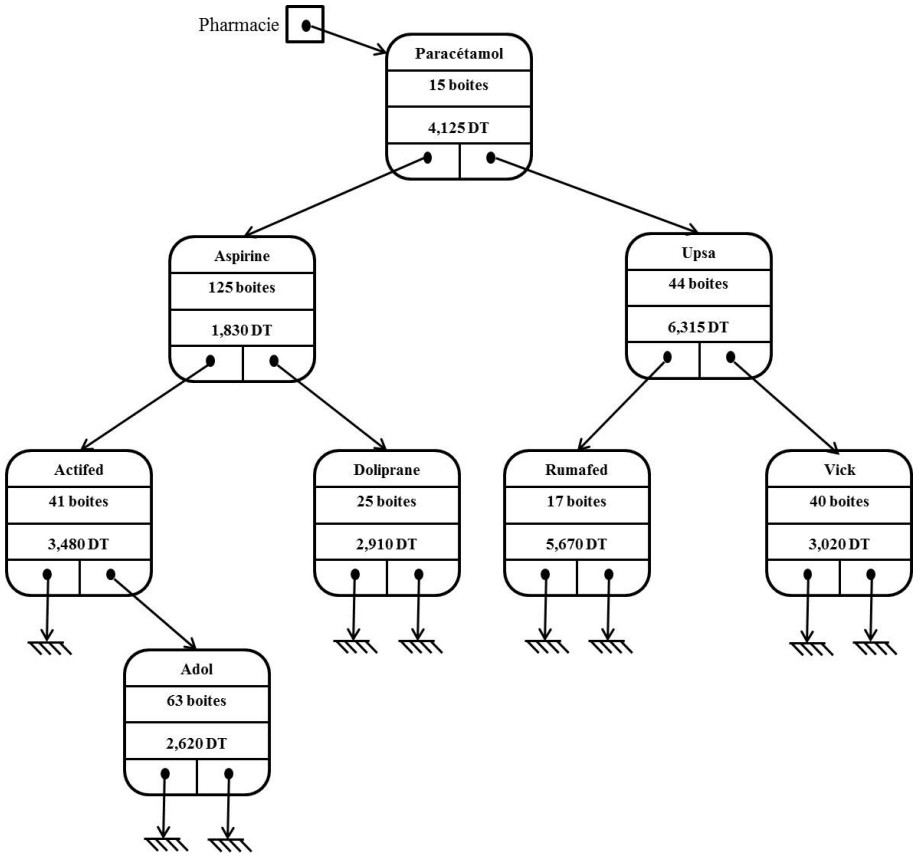


La liste verticale contient les catégories des livres avec le nombre de livre dans chacune, tandis que les listes horizontales contiennent les livres de chaque catégorie. Chaque livre est caractérisé par son titre, son hauteur et son année d'édition. La liste verticale est triée selon les catégories et les listes horizontales sont triées selon l'année d'édition.

1. Définir les structures de données nécessaires à l'implémentation de cette bibliothèque.
2. Ecrire la procédure d'ajout d'une nouvelle catégorie.
3. Ecrire la procédure d'ajout d'un nouveau livre.
4. Ecrire la procédure permettant d'afficher les livres d'une catégorie donnée.
5. Ecrire la fonction permettant de retourner le nombre total de livres dans la bibliothèque.
6. Ecrire la procédure permettant la suppression d'une catégorie avec tous ses livres.

Exercice 2*(10 pts)*

Un pharmacien souhaite traiter les informations concernant son stock de médicaments par ordinateur. Chaque médicament est caractérisé par son libellé, sa quantité en stock et son prix de vente. On vous propose de représenter ces informations à l'aide d'un arbre binaire de recherche basé sur l'ordre alphabétique des libellés des médicaments. Un exemple d'un tel arbre est donné par la figure suivante :



1. Définir les structures de données nécessaires à la représentation de ce stock.
2. Ecrire la procédure **Vente** permettant de retirer, si possible, un nombre donné de boites d'un médicament donné. Si la quantité en stock atteint un seuil min= 5, il faut afficher un message d'alerte.
3. Ecrire la procédure **Achat** permettant d'approvisionner le stock par un nombre donnée d'un médicament donné. Si le médicament ne figure pas dans le stock alors il faut l'ajouter.
4. Ecrire la fonction **PrixStock** permettant de calculer le prix total du stock.
5. Représenter l'arbre donné par la figure ci-dessus après l'ajout de ces deux médicaments :
   - Maxilase ; 23 boites ; 8,670 DT
   - Saifoxyle ; 42 boites ; 6,435 DT

♣ *Bon travail*

Université de Gabès
Faculté des Sciences de Gabès
Département Informatique

A.U: 2015-2016

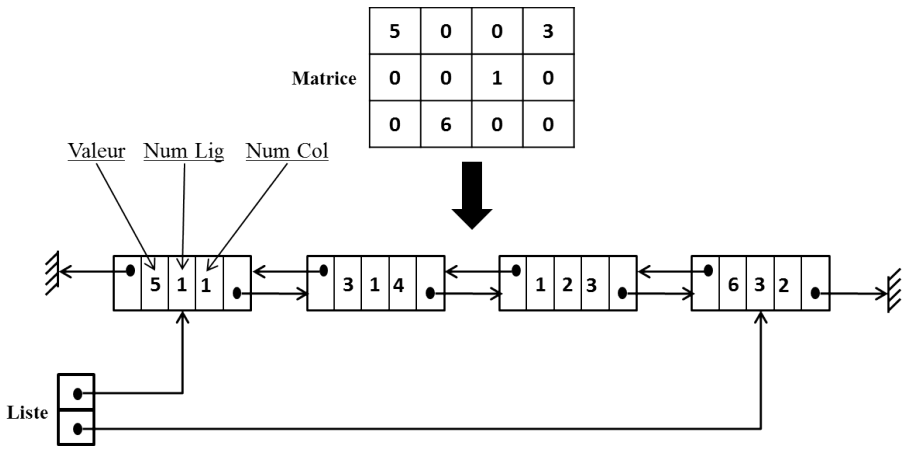
Examen

Section: LFSI1 **Épreuve d':** Algorithmique et structures de données II

Nature de l'épreuve:	DC☐ DS☐ EF☑		Documents:	autorisés	☐	non autorisés	☑
Date de l'épreuve:	23/05/2016		Calculatrice:	autorisée	☐	non autorisée	☐
Durée de l'épreuve:	01h30		Session:	principale	☑	contrôle	☐

Exercice 1 (14pts)

Une matrice Creuse est une matrice dont la plupart des éléments sont null. On souhaite représenter une telle matrice à l'aide d'une liste doublement chaînée comme le montre la figure suivante :



1. Définir les structures nécessaires pour résoudre le problème.
2. Ecrire une procédure permettant de transformer une matrice en une liste.
3. Ecrire les procédures permettant de :
   (a) Modifier la valeur d'un élément non null à une autre valeur non null
   (b) Modifier la valeur d'un élément null à une valeur non null
   (c) Modifier la valeur d'un élément non null à une valeur null
4. Ecrire une fonction permettant de retourner le nombre de lignes d'une matrice représentée par une liste L.
5. Ecrire une fonction permettant de retourner le nombre de colonnes d'une matrice représentée par une liste L.
6. Une matrice diagonale est une matrice carrée (nombre de colonnes égal au nombre de lignes) dont les éléments en dehors de la diagonale principale sont nuls. Les éléments de la diagonale peuvent être ou ne pas être nuls. Ecrire une fonction permettant de vérifier si une matrice, représentée par une liste L, est diagonale ou non.
7. Représenter la liste donnée par la figure ci-dessus après l'exécution des instructions suivantes :

(a) Matrice[2][3]← 4

(b) Matrice[2][4]← 8

(c) Matrice[1][3]← 0

Exercice 2 (6pts)

On souhaite enrichir la structure des arbres binaires de recherche en ajoutant un champ numérique nbg à chaque nœud, qui contiendra le nombre de nœuds de son sous-arbe gauche. Un nœud est donc représenté par 4 éléments [val, nbg, fg, fd]. On suppose que les valeurs d'un arbre de recherche sont toutes distinctes.

1. Dessinez l'arbre de recherche obtenu en insérant dans un arbre vide les valeurs suivantes : 12, 5, 24, 3, 37, 49, 25, 17, 8 et 29, dans cet ordre et en faisant apparaître le champ nbg pour chaque nœud.

2. La procédure permettant d'insérer une nouvelle valeur dans un arbre de recherche doit être modifiée de façon à mettre à jour le champ nbg.

   (a) Ecrire une fonction nbg(a) qui retourne le nombre de nœuds du sous arbre gauche de l'arbre non vide a,

   (b) Ecrire la nouvelle procédure d'insertion.

3. Le rang r(x) d'un élément x figurant dans l'arbre de recherche a est égal au nombre d'éléments de a strictement plus petit que x : le rang du plus petit élément de a est donc égal à 0. Soit $r >= 0$. On souhaite trouver l'élément x de a de rang r. Ecrire une fonction qui prend en entrée un arbre de recherche a et un rang r et retourne l'élément de a de rang r. On suppose que a contient au moins $r + 1$ éléments

4. Ecrire une fonction qui prend en entrée un arbre de recherche a et un élément x et retourne le rang de x dans a. Si x n'existe pas dans l'arbre a, la fonction retourne 0.

Bon Travail

Université de Gabès
Faculté des Sciences de Gabès
Département Informatique

A.U: 2015-2016

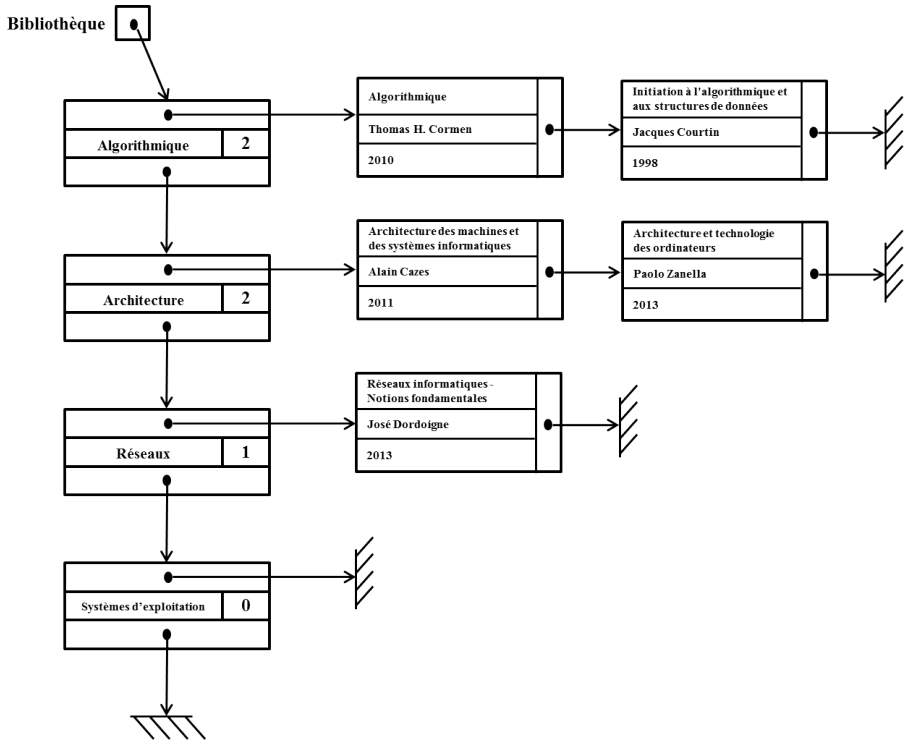
Examen

Section: LARI1 **Épreuve d':** Algorithmique et structures de données II

Nature de l'épreuve:	DC☐ DS☐ EF☑		Documents:	autorisés	☐	non autorisés	☑
Date de l'épreuve:	23/05/2016		Calculatrice:	autorisée	☐	non autorisée	☐
Durée de l'épreuve:	01h30		Session:	principale	☑	contrôle	☐

Exercice 1 (14pts)

Un étudiant en informatique souhaite représenter sa bibliothèque personnelle en utilisant une structure dynamique. La structure proposée est représentée par la figure suivante :



La liste verticale contient les catégories des livres avec le nombre de livre dans chacune, tandis que les listes horizontales contiennent les livres de chaque catégorie. Chaque livre est caractérisé par son titre, son hauteur et son année d'édition. La liste verticale est triée selon les catégories et les listes horizontales sont triées selon l'année d'édition.

1. Définir les structures de données nécessaires à l'implémentation de cette bibliothèque.
2. Ecrire la procédure d'ajout d'une nouvelle catégorie.
3. Ecrire la procédure d'ajout d'un nouveau livre.
4. Ecrire la procédure permettant d'afficher les livres d'une catégorie donnée.
5. Ecrire la fonction permettant de retourner le nombre total de livres dans la bibliothèque.

6. Ecrire la procédure permettant la suppression d'une catégorie avec tous ses livres.

Exercice 2 (6pts)

Soit P = (a1, . . . , an) une pile non vide de n éléments (avec a1 le sommet de la pile). On désire modifier le contenu de la pile P pour qu'elle contienne (a2, . . . , an, a1) (c'est-à-dire le sommet a1 est déplacé au fond de la même pile). Cette opération s'appelle la rotation d'une pile.

1. Définir les structures nécessaires pour résoudre le problème.
2. Ecrire la procédure **Rotation(P : *Pile)**.

Bon Travail

Université de Gabès
Faculté des Sciences de Gabès
Département Informatique

A.U: 2014-2015

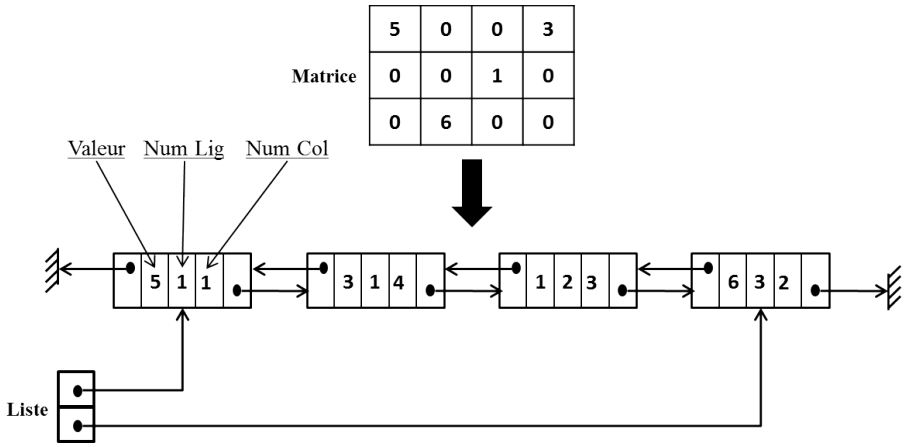
Examen

Section: LFSI1 **Épreuve d':** Algorithmique et structures de données II

Nature de l'épreuve:	DC☐ DS☐ EF☑		Documents:	autorisés	☐	non autorisés	☑
Date de l'épreuve:	17/06/2015		Calculatrice:	autorisée	☐	non autorisée	☐
Durée de l'épreuve:	1H30		Session:	principale	☐	contrôle	☑

Exercice 1 *(10 pts)*

Une matrice Creuse est une matrice dont la plupart des éléments sont null. On souhaite représenter une telle matrice à l'aide d'une liste doublement chaînée comme le montre la figure suivante :



1. Définir les structures nécessaires pour résoudre le problème.
2. Ecrire une procédure permettant de transformer une matrice en une liste.
3. Ecrire les procédures permettant de :
   (a) Modifier la valeur d'un élément non null à une autre valeur non null
   (b) Modifier la valeur d'un élément null à une valeur non null
   (c) Modifier la valeur d'un élément non null à une valeur null
4. Ecrire une fonction permettant de retourner le nombre de lignes d'une matrice représentée par une liste L.
5. Ecrire une fonction permettant de retourner le nombre de colonnes d'une matrice représentée par une liste L.
6. Une matrice diagonale est une matrice carrée (nombre de colonnes égal au nombre de lignes) dont les éléments en dehors de la diagonale principale sont nuls. Les éléments de la diagonale peuvent être ou ne pas être nuls. Ecrire une fonction permettant de vérifier si une matrice, représentée par une liste L, est diagonale ou non.
7. Représenter la liste donnée par la figure ci-dessus après l'exécution des instructions suivantes :

(a) Matrice[2][3]← 4

(b) Matrice[2][4]← 8

(c) Matrice[1][3]← 0

Exercice 2*(4 pts)*

Nous partons de la déclaration ci-après :

```
Type
    Cellule = Enregistrement
        Val : entier
        suiv : *Cellule
    Fin Enreg
    Pile=Enregistrement
        Sommet : *Cellule
    Fin Enreg
```

Nous supposons également que nous avons à notre disposition les fonctions et procédures déjà détaillées en cours :

- La procédure **Init** : permettant d'initialiser une pile vide,
- La procédure **Empiler** : permettant d'ajouter un élément en sommet d'une pile,
- La fonction **Dépiler** : permettant d'éliminer le sommet d'une pile en retournant sa valeur.
- La fonction **Sommet** : permettant de retourner la valeur du sommet d'une pile.
- La fonction **Vide** : permettant de vérifier si une pile est vide ou non.

Utiliser la notion de pile pour :

1. Écrire une fonction qui permet de vérifier si une expression mathématique, donnée sous forme d'une chaîne de caractères, est bien parenthésée ou non.
   Exemples :
   $(a + b - (c * d))$: expression valide
   $a + (b - (c * d)$: expression non valide
2. Réécrire la même fonction dans le cas où on peut trouver des parenthèses ainsi que des crochets dans l'expression mathématique.
   Exemples :
   $[a + b - (c * d)] * a$: expression valide
   $[a + b - (c * d]) * a$: expression non valide

Exercice 3*(4 pts)*

On note n le nombre de nœuds d'un arbre binaire, f son nombre de feuilles et h sa hauteur.

1. Quelle est la hauteur maximale d'un arbre binaire à n nœuds ?
2. Quelle est la hauteur minimale d'un arbre binaire à n nœuds ?
3. Quel est le nombre maximum de feuilles d'un arbre binaire de hauteur h ?
4. Quel est le nombre maximum de nœuds d'un arbre binaire de hauteur h ?
5. Quel est le nombre maximum de feuilles d'un arbre binaire à n nœuds ?

♣ *Bon travail*

Université de Gabès
Faculté des Sciences de Gabès
Département Informatique

A.U: 2015-2016

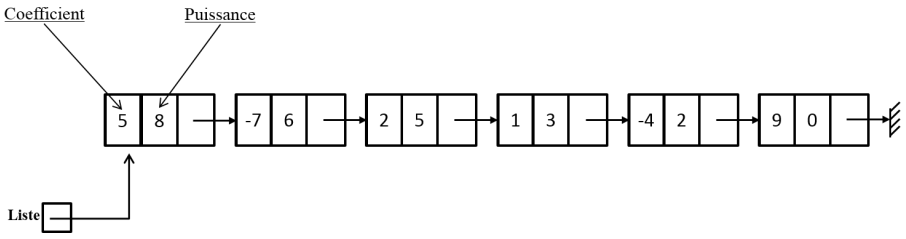
Examen

Section: LFSI1 **Épreuve d':** Algorithmique et structures de données II

Nature de l'épreuve:	DC☐ DS☐ EF☑		Documents:	autorisés	☐	non autorisés	☑
Date de l'épreuve:	15/06/2016		Calculatrice:	autorisée	☐	non autorisée	☐
Durée de l'épreuve:	01h30		Session:	principale	☐	contrôle	☑

Exercice 1 (12 pts)

On souhaite représenter un polynôme à l'aide d'une liste simplement chaînée. Un polynôme est composé d'un ensemble de monômes triés dans l'ordre décroissant selon leurs puissances
Exemple : le polynôme $5x^8 - 7x^6 + 2x^5 + x^3 - 4x^2 + 9$ sera représenté par la liste suivante :

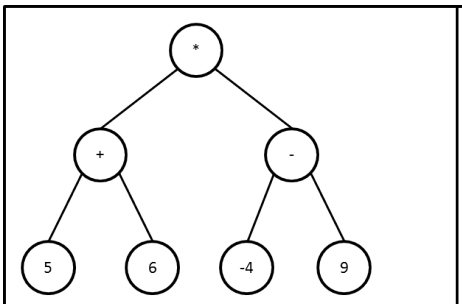


1. Définir les structures nécessaires.
2. Ecrire une procédure permettant de saisir un monôme.
3. Ecrire une fonction permettant de déterminer le degré d'un polynôme.
4. Ecrire une procédure permettant d'ajouter un monôme à la bonne position dans un polynôme.
5. Ecire une fonction permettant d'évaluer un polynôme pour une valeur de x donnée.
6. Ecrire une procédure permettant de déterminer la dérivée d'un polynôme passé en paramètres.
7. Ecrire une procédure permettant de déterminer la somme de deux polynômes passés en paramètres.
8. Ecrire un algorithme principal, utilisant les procédures et les fonctions ci-dessus, pour :
   - Saisir deux polynômes A et B.
   - Déterminer la somme de deux polynômes A et B dans un polynôme C.
   - Déterminer la dérivée du polynôme C dans un polynôme D.
   - Afficher le degré du polynôme D.
   - Saisir un réel x.
   - Caluler et afficher l'image de x par le polynôme D.

Exercice 2 (8 pts)

Un arbre d'expressions est un arbre binaire où les nœuds sont ou bien des opérateurs ou bien des opérandes :

1. En supposant qu'on dispose d'un arbre d'expressions non vide, écrire la procédure **ABR-File(A :Arbre, F :File, O : Entier)** qui permet de parcourir un arbre A de telle sorte à créer l'expression arithmétique qui lui correspond sous forme d'une FILE F. La nature de l'expression demandée est précisé par l'entier O(O=1 : expression préfixée, O=2 : expression infixée, O=3 : expression postfixée). Comme le montre l'exemple suivant :



O	F
Parcours préfixé (O=1)	* + 5 6 - -4 9
Parcours infixé (O=2)	5 + 6 * -4 - 9
Parcours postfixé (O=3)	5 6 + -4 - *

2. Supposons que O=1, donner les différentes étapes qui permettent d'évaluer $F = * + 5\ 6\ -\ -4\ 9$ en utilisant une pile P (à chaque étape préciser le contenu de P et celui de F).

Bon Travail

Université de Gabès
Faculté des Sciences de Gabès
Département Informatique

A.U: 2015-2016

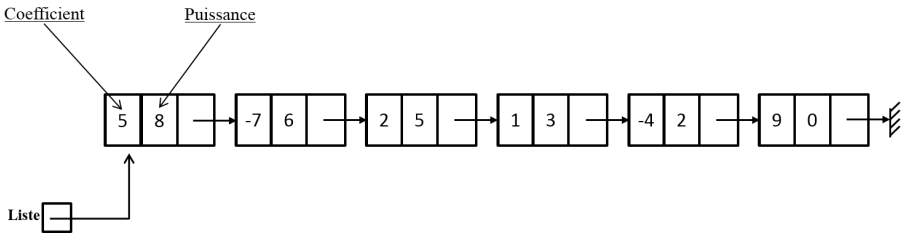
Examen

Section: LARI1 **Épreuve d':** Algorithmique et structures de données II

Nature de l'épreuve:	DC☐ DS☐ EF☑		Documents:	autorisés	☐	non autorisés	☑
Date de l'épreuve:	15/06/2016		Calculatrice:	autorisée	☐	non autorisée	☐
Durée de l'épreuve:	01h30		Session:	principale	☐	contrôle	☑

Exercice 1 (12 pts)

On souhaite représenter un polynôme à l'aide d'une liste simplement chaînée. Un polynôme est composé d'un ensemble de monômes triés dans l'ordre décroissant selon leurs puissances
Exemple : le polynôme $5x^8 - 7x^6 + 2x^5 + x^3 - 4x^2 + 9$ sera représenté par la liste suivante :



1. Définir les structures nécessaires.
2. Ecrire une procédure permettant de saisir un monôme.
3. Ecrire une fonction permettant de déterminer le degré d'un polynôme.
4. Ecrire une procédure permettant d'ajouter un monôme à la bonne position dans un polynôme.
5. Ecire une fonction permettant d'évaluer un polynôme pour une valeur de x donnée.
6. Ecrire une procédure permettant de déterminer la dérivée d'un polynôme passé en paramètres.
7. Ecrire une procédure permettant de déterminer la somme de deux polynômes passés en paramètres.
8. Ecrire un algorithme principal, utilisant les procédures et les fonctions ci-dessus, pour :
   - Saisir deux polynômes A et B.
   - Déterminer la somme de deux polynômes A et B dans un polynôme C.
   - Déterminer la dérivée du polynôme C dans un polynôme D.
   - Afficher le degré du polynôme D.
   - Saisir un réel x.
   - Caluler et afficher l'image de x par le polynôme D.

Exercice 2 (8 pts)

Utiliser la notion de pile pour :

1. Écrire une fonction qui permet de vérifier si une expression mathématique, donnée sous forme d'une chaîne de caractères, est bien parenthésée ou non.
   Exemples :
   $(a + b - (c * d))$: expression valide
   $a + (b - (c * d)$: expression non valide

2. Réécrire la même fonction dans le cas où on peut trouver des parenthèses ainsi que des crochets dans l'expression mathématique.
   Exemples :
   $[a + b - (c * d)] * a$: expression valide
   $[a + b - (c * d]) * a$: expression non valide

Bon Travail