Préface

Ce polycopié a été préparé au centre Universitaire de Relizane, il traite les structures de données complexes. Ce polycopié rassemble les cours et travaux dirigés (avec corrigés) du module Algorithmique et structures de données (ASD) de département d'informatique, Institut des sciences et techniques. Le module s'intègre dans la deuxième année de Licence d'Informatique.

Dans ce polycopié, vous allez trouvez des leçons détaillées, bien illustrées par des exemples, qui peuvent servir aussi bien à l'étudiant pour mieux assimiler les nouveaux concepts de la programmation des structures de données complexes, que pour l'enseignant, qui peut se baser sur les thèmes traités pour préparer un cours complet et progressif.

L'objectif principal de ce travail est de simplifier autant que possible , l'utilisation et l'exploitation de la puissance offerte par les structures de données telles que les piles, les files , les arbres... etc.

Ce polycopié est structuré d'une manière à respecter le caractère pédagogique et progressif des cours.

Le premier cours traite la notion des procédure et fonctions (les sous programmes).

Les procédures et fonctions sont des sortes de sous-programmes écrits avant le programme principal mais appelés depuis ce programme principal, d'une autre procédure ou même d'une autre fonction. Le nom d'une procédure ou d'une fonction est un identifiant qui ne de doit pas excéder 127 caractères et ne pas contenir d'accent.

Le second cours traite la notion de récursivité.

En informatique et en logique, une fonction ou plus généralement un algorithme qui contient un ou des appel(s) à lui-même est dit récursif. Ce

procédé est employé dans la conception d'algorithme basée sur le paradigme diviser pour régner.

Deux fonctions peuvent s'appeler l'une l'autre, on parle alors de récursivité croisée.

La définition de certaines structures de données, comme les listes, les arbres est récursive. Par exemple un arbre binaire est soit fait deux arbres binaires sur lequel on enracine un nœud, soit est un arbre binaire vide. La récursivité est un point délicat dans l'enseignement de l'informatique, car son appropriation par l'apprenant demande une dose d'abstraction.

Le troisième cours traite les structures de données séquentielles piles et files.

Les piles (stack) et les files (queue) constituent des structures de données. Elles vont, comme leur nom l'indique, nous permettre de stocker diverses données, comme pourrait le faire un tableau.

Une pile permet de réaliser ce que l'on nomme une LIFO (Last In First Out), ce qui signifie en clair que les derniers éléments à être ajoutés à la pile seront les premiers à être récupérés. Il est possible de comparer cela à une pile d'assiettes. Lorsqu'on ajoute une assiette en haut de la pile, on retire toujours en premier celle qui se trouve en haut de la pile, c'est-à-dire celle qui a été ajoutée en dernier, sinon tout le reste s'écroule.

Une file, quant à elle, permet de réaliser une FIFO (First In First Out), ce qui veut dire que les premiers éléments ajoutés à la file seront aussi les premiers à être récupérés.

Les piles peuvent être utilisées dans des algorithmes d'évaluation d'expressions mathématiques. Les files peuvent être utilisées afin de mémoriser des données en attente de traitement.

Le quatrième cours traite les structures hiérarchiques arbres, l'un des concepts algorithmiques les plus importants de l'informatique. Les arbres servent à représenter un ensemble de données structurées hiérarchiquement.

Un arbre est une structure non linéaire, c'est un graphe sans cycle où chaque nœud a au plus un prédécesseur.

Les cours de ce polycopié sont suivie d'une partie exercice pour mettre l'accent sur les points forts de chaque cours.

L' auteur peut être contacté par courrier électronique à l'adresse suivante : yachbakhadidja@yahoo.fr

Table des matières

Liste des figures

Liste des tableaux

Les procédures et les fonctions

1

Dans ce chapitre, nous présentons les différentes définitions sur les procédures, les fonctions et la différence entre les deux notions l'objectif de ce chapitre est de Comprendre la notion de division de problème, Connaitre les fonctions et les procédures ainsi que la différence entre les deux, Appliquer le concept de programmation structurée lors de l'écriture des algorithmes.

1.1 Introduction

Une application, surtout si elle est longue, a toutes les chances de devoir procéder aux mêmes traitements, ou à des traitements similaires, à plusieurs endroits de son déroulement. Par exemple, la saisie d'une réponse par oui ou par non (et le contrôle qu'elle implique), peuvent être répétés dix fois à des moments différents de la même application, pour dix questions différentes.

La manière la plus évidente, mais aussi la moins habile, de programmer ce genre de choses, c'est bien entendu de répéter le code correspondant autant de fois que nécessaire. Apparemment, on ne se casse pas la tête : quand il faut que la machine interroge l'utilisateur, on recopie les lignes de codes voulues en ne changeant que le nécessaire, et roule Raoul. Mais en procédant de cette manière, la pire qui soit, on se prépare des lendemains qui déchantent... D'abord, parce que si la structure d'un programme écrit de cette manière peut paraître simple, elle est en réalité inutilement lourdingue. Elle contient des répétitions, et pour peu que le programme soit joufflu, il peut devenir parfaitement illisible. Or, le fait d'être facilement modifiable donc lisible, y compris - et surtout - par ceux qui ne l'ont pas écrit est un critère essentiel pour un programme informatique ! Dès que l'on programme non pour soi-même, mais dans le cadre d'une organisation (entreprise ou autre), cette nécessité se fait sentir de manière aiguë. L'ignorer, c'est donc forcément grave.

En plus, à un autre niveau, une telle structure pose des problèmes considérables de maintenance : car en cas de modification du code, il va falloir traquer toutes les apparitions plus ou moins identiques de ce code pour faire convenablement la modification ! Et si l'on en oublie une, patatras, on a laissé un bug.

Il faut donc opter pour une autre stratégie, qui consiste à séparer ce traitement du corps du programme et à regrouper les instructions qui le composent en un module séparé. Il ne restera alors plus qu'à appeler ce groupe d'instructions (qui n'existe donc désormais qu'en un exemplaire unique) à chaque fois qu'on en a besoin. Ainsi, la lisibilité est assurée ; le programme devient modulaire, et il suffit de faire une seule modification au bon endroit, pour que cette modification prenne effet dans la totalité de l'application.

Le corps du programme s'appelle alors la procédure principale, et ces groupes d'instructions auxquels on a recours s'appellent des fonctions et des sous-procédures (nous verrons un peu plus loin la différence entre ces deux termes).

1.2 Problème

Dès qu'on commence à écrire des programmes sophistiqués, il devient difficile d'avoir une vision globale sur son fonctionnement.

Difficulté de trouver des erreurs

Solution : décomposer le problème en sous problèmes

– Trouver une solution à chacun

– La solution partielle donne lieu à un sous-programme

1.3 Programmation procédurale

Dans cette section, nous présentons les notions de bases de la programmation procédurale.

1.3.1 Principe

Il s'agit d'écrire des programmes en utilisant des sous-programmes.

1.3.2 Forme générale d'un programme

La forme générale d'un programme utilisant un sous programme est comme suit :

Programme P

Sous-programme SP1

.....

Sous-programme SPn

FinP.

1.4 Procédures et Fonctions

En algorithmique, on distingue deux types de sous-programmes

- Les procédures
- Les fonctions

1.4.1 Fonctions : structure

Une fonction est un sous-programme qui :

- A un nom
- Peut avoir des paramètres
- Qui retourne une valeur d'un certain type
- Qui peut avoir besoin de variables
- Qui est composé d'instructions

1.4.2 Fonctions : déclaration

Pour déclarer une fonction , nous devont respecter la syntaxe suivante :

Fonction nomf (<paramètres>) : type
Déclaration des variables
Début
instructions
nomf ← expression
Fin fonction

1.4.3 Fonctions : Exemple

Fonction qui retourne le carré d'un entier :

Fonction carré (n : entier) : entier ;
Début
*carré ← n * n ;*
fin fonction

1.4.4 Fonction : utilisation dans un algorithme

Il faut intégrer le code de la fonction dans l'algorithme pour pouvoir l'appeler par la suite. Pour l'exemple précédent l'algorithme est comme suit :

Algorithme ex1 ;

Variable i, j : entier ;

Fonction carré(n : entier) : entier ;

Début

*carré ← n * n ;*

fin fonction ;

Début

Lire (i) ;

Ecrire(carré(i)) ; // Appel de la Fonction

Fin

1.4.5 Fonctions : A retenir

- Une fonction retourne toujours une valeur
- Une fonction NomF contient toujours une instruction de la forme :

  NomF ← Expression
- Il ne faut jamais utiliser d'instructions de la forme :

  f(paramètres) ← expression
- En général, l'utilisation d'une fonction se fait :
  - Soit par une affectation : v ← f(paramètres)
  - Soit dans l'écriture : Ecrire (f(paramètres))

1.4.6 Procédures : définition

- Une procédure est un sous-programme qui ne retourne pas de valeur
- C'est donc un type particulier de fonction
- En général, une procédure modifie la valeur de ses paramètres

1.4.7 Procédures : structure

Tout comme les fonctions, une procédure est un sous-programme qui :

- A un nom
- Peut avoir des paramètres
- Qui peut avoir besoin de variables
- Qui est composé d'instructions

1.4.8 Procédures : déclaration

Il faut restecter la syntaxe générale d'une procédure , cette syntaxe est comme suit :

Procédure nomf (<paramètres>)

Déclaration des variables

Début

instructions

Fin procédure

Procédures : exemple

Une procédure qui ajoute 2 à un entier.

Procédure aug2(n : entier) ;

Début

n ← n+2 ;

Fin Procédure

1.4.9 Procédures : dans les algorithmes

Après l' écriture de la procédure il faut l'intégrer dans l'algorithme pour pouvoir l'utiliser.

Écrire un algorithme qui :

– Lit un entier positif n puis

– Affiche tous les nombres impairs inférieurs à n

Algorithme ex2 ;

Variable i,n : entier ;

Procédure Aug2(..) ;

.....

Fin Procédure ;

Début

Lire(n) ;

i ← 1 ; Tant que i ≤ n

Ecrire(i) ;

aug2(i) ; // appel de la procédure

Fin TantQue ;

Fin

1.4.10 Procédures : A retenir

- Une procédure ne retourne pas de valeur
- Il est donc faux de l'affecter à une variable
- Ne pas écrire : j ←aug2(i)

1.5 Différence entre fonction et procédure

Les notions de procédure et de fonction sont très proches. La principale différence est qu'une procédure réalise un traitement et est assimilable à une instruction alors qu'une fonction (calcule et) renvoie une information et correspond donc à une expression. Un appel à une fonction peut apparaître là où une expression peut être utilisée (à droite de l'affectation, dans une expression, dans une condition, etc.)

1.6 Intérêt des sous-programmes

L'utilisation des sous-programmes (procédures et fonctions) est intéressante à plus d'un titre. Nous en précisons quelques uns ci-dessous.

- Structuration de l'algorithme : en l'absence de procédure et de fonction, l'algorithme est monolithique : les instructions sont toutes écrites les unes à la suite des autres et ce ne sont que les commentaires, traces du raffinages, qui le structurent et permettent de le comprendre. Chaque procédure ou fonction correspond à une sous-partie (un sous-programme) de l'algorithme initial. Il permet donc de le découper en « morceaux » et donc de le structurer en entités relativement indépendantes.
- Compréhensibilité de l'algorithme : la structuration permet d'améliorer la compréhension de l'algorithme, car il est découpé en procédures et fonctions de taille réduite qui sont donc plus facile à comprendre. Les algorithmes ne font donc plus nécessairement appel aux instructions élémentaires mais à des instructions de plus haut niveau les procédures et les fonctions qui, de part leur nom significatif, en améliore la compréhension.

– Factorisation du code : les sous-programmes permettent de factoriser des parties de l'algorithme.

  dont l'objectif était de faire tourner à gauche le robot. La manière de faire tourner à gauche le robot n'est donnée qu'à un seul endroit.

– Mise au point : dès qu'un sous-programme est écrit, il peut (et doit) être testé. Ainsi, le programme est testé petits bouts par petits bouts (sous-programme après sous-programme). Les erreurs et surtout leur origine sont alors beaucoup plus facilement identifiées que si l'ensemble du l'algorithme était testé d'un seul coup. Ceci est également vrai lors de la modification de sous-programmes (maintenance corrective ou évolutive).

– Amélioration de la maintenance : comme la compréhension du programme, la maintenance est automatiquement améliorée, car il sera plus facile d'identifier les parties de l'algorithme à modifier et d'en évaluer l'impact. L'idéal est bien entendu que la modification puisse être limitée à un petit nombre de sous-programmes. La factorisation améliore également notablement la maintenance. En effet, si une modification est faite dans un sous-programme, elle sera automatiquement prise en compte dans toutes les autres parties qui utilisent ce sous-programme. Si le code avait été dupliqué, il aurait fallu le changer à tous les endroits. Dans le même ordre d'idée, les sous-programmes permettent au programmeur de définir ses propres instructions et expressions. Ainsi, s'appuyant sur ses instructions, il peut écrire des algorithmes indépendants des instructions élémentaires du processeur.

La récursivité

2

Dans ce chapitre, nous présentons la notion de récursivité, le principe de la récursivité, la différence entre la solution itérative et récursive,..

.

2.1 Introduction

En programmation, de nombreux problémes résolus par répétition de taches : certains langages (comme Pascal, C) munis de structures de contrôles répétitives : boucles pour et tant que, mais certains problémes se résolvent simplement en résolvant des problémes identiques : C'est la récursivité.

La programmation récursive est une technique de programmation qui remplace les instructions de boucle (tant que , pour, etc.) par des appels de fonction.

La récursivité est un domaine très intéressant de l'informatique, un peu abstrait, mais très élégant, elle permet de résoudre certains problémes d'une maniéré très rapide, alors que si on devait les résoudre d'une manière itérative, il nous faudrait beaucoup plus de temps et de structures de données intermédiaires.

2.2 Définition

Un programme récursif c'est un programme qui s'appelle lui-même ou une fonction qui est définie par rapport à elle-même

La récursivité est une démarche qui fait référence à l'objet même de la démarche à un moment du processus. En d'autres termes, c'est la propriété de pouvoir appliquer une même règle plusieurs fois en elle-même, Ainsi, les cas suivants constituent des cas concrets de récursivité :

- Décrire un processus dépendant de données en faisant appel à ce même processus sur d'autres données plus « simples » ;
- Montrer une image contenant des images similaires ;
- Définir un concept en invoquant le même concept ;
- Écrire un algorithme qui invoque lui-même ;
- Définir une structure à partir de l'une au moins de ses sous-structures ;
- faire pointer un article de wikipédia vers lui-même ou vers un article qui, par une succession de pointeurs, pointe vers l'article dont on est partie.

2.3 Structure d'un programme récursif

un programme récursif a la structure suivante :

P

-

-

Appel à P

-

-

Si nous prenons comme exemple une procédure récursive , la structure de cette procédure est comme suit :

Procedure P

Début

<instructions>

Appel à P

<instructions>

Fin

Exemple 1 : Calcul de factoriel « n » ;

n !=n*(n-1)*(n-2)* ? ? ? ?..*3*2*1.

0 !=1

1 !=1

Fact (n)= n*(n-1)*(n-2)* ? ? ? ?..*3*2*1.

Cas général : Fact (n)= n*Fact (n-1) .

Cas particuliers : Fact (0)=1 ; Fact (1)=1

Algorithme 1 : Fact-Réc ;

/ entrée : un nombre n*/*

/ sortie : la factorielle de n*/*

Début

Si (n<=1) alors

Retourner 1

Sinon

Retourner n Fact-Rec(n-1) ;*

Fin

Déroulement : exécution :

Fact-Rec (4)= ?

n=4

Retourne 4* Fact-Rec (3)

n=3

Retourne 3* Fact-Rec (2)

n=2

Retourne 2* Fact-Rec (1)

n=1

Retourne 1

Fact-Rec (4)= 4*3*2*1

Remarque

La gestion de la mémoire se fait par pile.

Règles

– Ne jamais ré appliquer l'algorithme sur des données plus grandes.

– Toujours effectuer un Test de Terminaison.

Factorielle : version itérative :

Algorithme Fact-Iter ;

Début

/ entrée : un nombre n*/*

/ sortie : la factorielle de n*/*

f ← 1 ;

Pour i de 2 à n faire

*f ← f*i ;*

Fin pour ;

Fin

Déroulement : exécution :

Fact-Iter(0)= ? ; Fact-Iter(1)= ?

i=2

f ← 1*2

i=3

f ← 1*2*3

i=4

f ← 1*2*3*4

2.3.1 Solution récursive ou itérative ?

- Cela dépend de l'application
- Du coût en temps d'exécution et en place mémoire.
- Factorielle : équivalence des solutions itérative et récursive.
- Ce n'est pas toujours le cas.

2.4 Notions de pile d'exécution et de point terminal

Dans cette section , nous définissons les notions de pile d'exécution en récursivité et point terminal.

2.4.1 Définition (Pile d'exécution)

La Pile d'exécution du programme en cours est un emplacement mémoire destiné à mémoriser les paramètres, les variables locales ainsi que les adresses de retour des fonctions en cours d'exécution. Elle fonctionne selon le principe LIFO (Last-In-First-Out) : dernier entré premier sorti.

Attention ! La pile à une taille fixée, une mauvaise utilisation de la récursivité peut entraîner un débordement de pile.

2.4.2 Point terminal

Comme dans le cas d'une boucle, il faut un cas d'arrêt où l'on ne fait pas d'appel récursif. C'est le point terminal

***Procédure** récursive (paramètres)*

si (TEST D'ARRET)

Instructions du point d'arrêt

Sinon

Instructions

Récursive(paramètres changés) ; // appel récursif

Instructions

Fin

On constate que les paramètres de l'appel récursif changent ; en effet, à chaque appel, l'ordinateur stocke dans la pile les variables locales ; le fait de ne rien changer dans les paramètres ferait que l'ordinateur effectuerait un appel infini à cette procédure, ce qui se traduirait en réalité par un débordement de pile, et d'arrêt de l'exécution de la procédure en cours. Grâce à ces changements, tôt ou tard l'ordinateur rencontrera un ensemble de paramètres vérifiant le test d'arrêt, et donc à ce moment la procédure récursive aura atteint le "fond" (point terminal). Ensuite les paramètres ainsi que les variables locales sont désempilées au fur et à mesure qu'on remonte les niveaux.

2.5 Ecrire un algorithme récursif

Écrire un algorithme récursif réalisant un certain traitement T sur des données D.

1. Décomposer le traitement T en sous traitements de même nature mais sur des données plus petites.
2. Trouver la condition d'arrêt.
3. Tester éventuellement sur un exemple.
4. Ecrire l'algorithme.

Les structures Piles et Files

3

Ce chapitre présente les structures séquentielles Pile et File, le principe de chaque structure, comment utiliser ces deux structures en programmation....

3.1 Introduction

Les piles et files sont deux structures de données très utilisées dans la pratique, elles servent essentiellement à stocker de façon temporaire des éléments sur lesquels, on effectue un grand nombre d'insertions et de suppressions.

3.2 Les piles

Une pile est une liste ordonnée d'éléments où les insertions et les suppressions d'éléments ont lieu à une seule et même extrémité de la liste appelée « Sommet de la pile ». Une pile est appelée aussi LIFO pour Last In First Out « Le dernier qui entre est le premier qui sort ».

La Figure (3.1) illustre la structure pile.

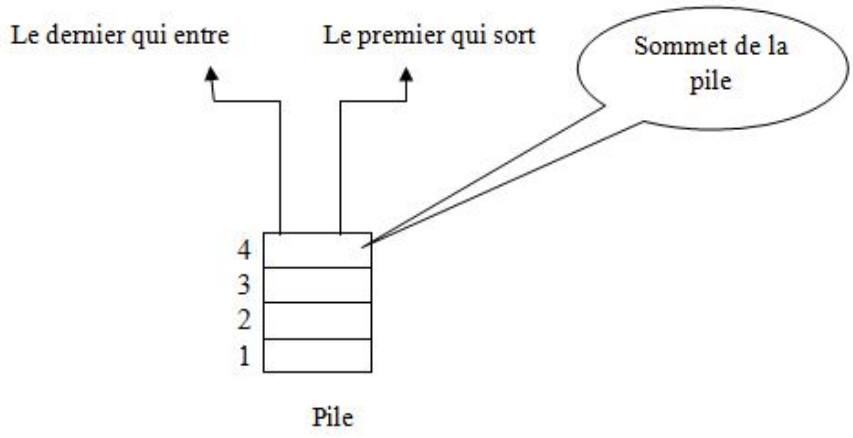

Figure 3.1 – *Une pile*

3.2.1 Principe

Une pile est une donnée à accès restreint, ce qui signifie que son traitement se limite à quelques manipulations. Tout nouvel élément est ajouté en haut de la pile, le retrait d'un élément fournie ce qui a été posé en dernier.

La figure (3.2) suivante illustre un exemple d'ajout et de suppression dans une pile.

Les piles sont particulièrement adaptées aux processus qui nécessitent la mise e attente de données pour un traitement ultérieur. C'est le cas du calcul d'une expression arithmétique en notation polonaise inversée, ou les données

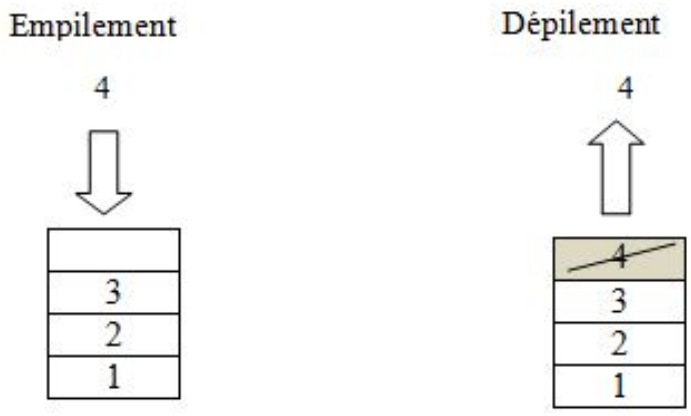

FIGURE 3.2 – *Exemple d'empilement et de dépilement d'un élément de la pile*

apparaissent avant l'opérateur. Les piles sont également un support privilégié pour rendre itératif un traitement récursif. Les valeurs correspondant aux appels récursifs sont empilées, et leur dépilement simule la remontée des appels.

3.2.2 Les primitives

Un programme utilisant une pile n'a pas besoin de connaitre l'implémentation réelle des données. Il se contente de les empiler ou de les dépiler. Il illustre des sous programmes qui implémentent les traitements de base, appelés Primitives de gestion. Les principales primitives sont :

- **Empiler () :** cette procédure ajoute la donnée passée en argument en haut de la pile.
- **Dépiler () :** cette fonction, sans argument, retourne l ?élément extrait de dessus de la pile.
- **Pilevide()** : cette fonction retourne la valeur VRAI si la pile est vide , et FAUX dans le cas contraire.
- **Initialiserpile () :** cette procédure initialise la pile.

Ces primitives gèrent la pile sous forme de tableaux ou de listes chainées. La représentation la plus simple d'une pile est le tableau. Si le nombre maximal d'éléments contenus dans la pile est connu à l'avance, il est facile de définir la taille du tableau. Sinon, il faut définir un tableau suffisamment grand, ou employer un tableau dynamique.

Procédure Initialiserpile ()

Début

nbelement = 0 ;

Fin

Procédure Empiler (C)

Début

Déclaration paramètres C en caractères ;

Pile [nbelement] = C ;

nbelement = nbelement + 1 ;

Fin

Fonction Dépiler ()

Début

Déclaration paramètres C en caractères ;

nbelement = nbelement = 1 ;

C = Pile [nbelement] ;

Retourner C ;

Fin

Fonction PileVide ()

Début

Si (nbelement=0) alors

Retourner VRAI

Sinon

Retourner FAUX ;

FinSi ;

Fin

Attention

Il faut toujours s'assurer que la pile n'est pas vide ou encore qu'elle n'est pas saturée, sinon des erreurs, telles que « pile vide » ou « débordement » , seront générées.

3.3 Les files

Une file est une liste ordonnée d'éléments dans laquelle toutes les insertions se font à une extrémité, et toutes les suppressions se font à l'autre extrémité.

Une file est aussi désignée par FIFO pour First In First Out (le premier entrant est le premier sortant).

La figure (3.3) suivante illustre la structure file.

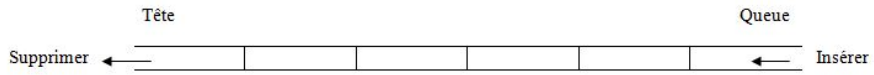

Figure 3.3 – *La file*

3.3.1 Exemples

Une file d'attente devant un guichet est une illustration réelle des files. Les personnes viennent faire la chaine (la file) selon leur arrivée au guichet. La première personne qui arrive, est la première à être servie.

La structure de la file est souvent utilisée dans le fonctionnement de l'ordinateur, par exemple lors de l'impression de fichiers sur une imprimante. On suppose qu'il y a plusieurs commandes d'impression (plusieurs documents à imprimer), les fichiers à imprimer se mettent en file d'attente, de façon à ce que le premier fichier envoyé soit le premier à être imprimé.

3.3.2 Principe

Une file est aussi une donnée à accès restreint, elle se comporte de la même manière que la pile, à la différence que le premier élément entré est le premier sortie. La gestion d'une file se résume à enfiler un élément (on l'ajoute à la fin de la file), ou à défiler (on retire le premier élément de la file).

La figure (3.4) illustre le principe de fonctionnement de la file.

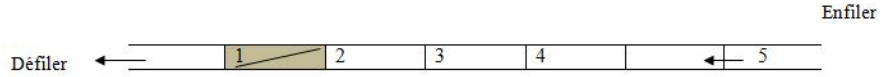

Figure 3.4 – *Le principe de la file*

Les files sont moins présentes dans les algorithmes, car leur champ d'application est moins étendu que celui des piles.

3.3.3 Les primitives

Dans cette section, nous présentons les primitives de base utilisées pour manipuler les files.

La procédure Initialiserfile () : positionne les indices de début de file et de fin de file.

La procédure Enfiler () : ajoute le nouvel élément dans une case vide du tableau et fait progresser l'indice de fin de liste.

La fonction Défiler () : retourne l'élément indiqué par début de liste et fait progresser cet indice.

La fonction filevide () : retourne une valeur booléenne qui indique si la file est vide ou non.

Procédure Initialiserfile ()

Début

DebutFile = 0 ;

FinFile = 0 ;

Fin

Procédure Emfiler (C)

Début

Déclaration paramètres C en caractères ;

File [FinFile] = C ;

FinFile = FinFile + 1 ;

Fin

Fonction Défiler ()

Début

Déclaration paramètres C en caractères ;

C = File [DebutFile] ;

DebutFile = DebutFile + 1 ;

Retourner C ;

Fin

Fonction PileVide ()

Début

Si (DebutFile = FinFile) alors

Retourner VRAI

Sinon

Retourner FAUX ;

FinSi ;

Fin

3.4 Questions/Réponses

1. Quels sont les points Communs entre les piles et les files ?
   - Ce sont deux listes ordonnées d'éléments.
   - Elles ont la même représentation physique (en mémoire).
   - Représentation par tableaux
   - Représentation par listes chainées.
2. Quels sont les points de différence entre les piles et les files ?
   - Dans le cas d'une pile l'ajout et la suppression d'éléments se font uniquement au sommet de la pile.
   - Dans le cas d'une file l'ajout se fait à la fin de la file(en queue de la file) et la suppression se fait au début de la file (en tête de liste).
3. Quels sont les cas d'erreurs ?
   - Empiler sur une pile pleine cause une erreur de débordement (overflow), même chose pour la file.
   - Supprimer un élément d'une pile (file) vide cause une erreur : pile ou file vide.

4 Les structures hiérarchiques (Arbres)

Sommaire

Le présent chapitre est consacré aux arbres, l'un des concepts algorithmiques les plus importants de l'informatique. Les arbres servent à représenter un ensemble de données structurées hiérarchiquement.

4.1 Définitions

Un arbre est une structure dynamique d'éléments appelés aussi parfois « sommet » ou « Nœud ». Ses nœuds sont organisés d'une manières hiérarchique (Père, Fils , Petit-fils, ?). Les nœuds sont reliés par des « Arcs » tel que chaque Nœud (à part la racine) a exactement un arc pointant vers lui. La racine est un NŒud particulier car il n'a pas de prédécesseur. Les feuilles sont les nœuds sans successeurs. En fin, chaque nŒud est composé de données et de pointeurs.

Un arbre est composé d'un nœud particulier appelé racine ; de plusieurs nœuds intermédiaires possédant chacun un et un seul nœud appelé père, et des nœuds possédant éventuellement un ou plusieurs fils. Les nœuds qui ne possèdent pas de fils sont appelés feuilles. La Figure (4.1) résume de façon générale la structure générale d'un arbre.

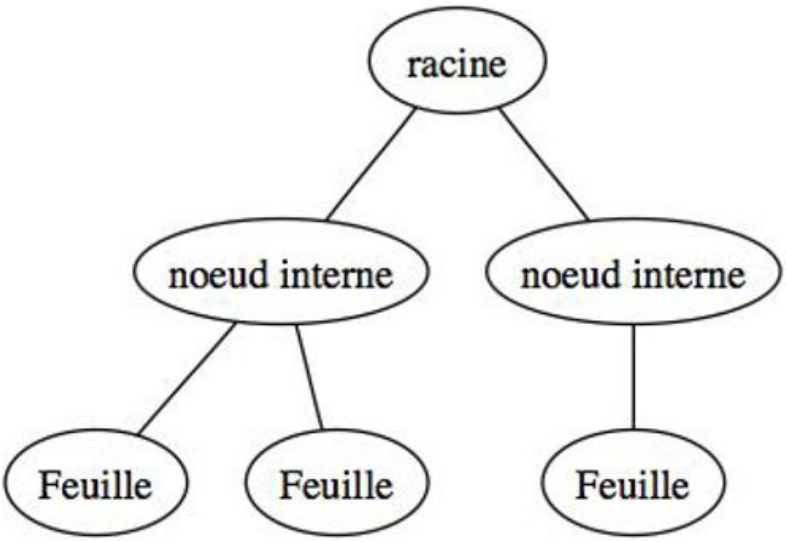

Figure 4.1 – *Description des différentes composantes d'un arbre*

4.2 Terminologie de base

Dans cette section les définitions des concepts de base des arbres sont représentées.

4.2.1 Arité d'un arbre

L'arité de l'arbre est le nombre de fils qu'il possède. Un arbre dont les nœuds ne comporteront qu'au maximum n fils sera d'arité n. On parlera

alors d'arbre n-aire. Il existe un cas particulièrement utilisé : c'est l'arbre binaire. Dans un tel arbre, les nœuds ont au maximum 2 fils. On parlera alors de fils gauche et de fils droit pour les nœuds constituant ce type d'arbre. L'arité n'impose pas le nombre minimum de fils, il s'agit d'un maximum, ainsi un arbre d'arité 3 pourra avoir des nœuds qui ont 0,1,2 ou 3 fils, mais en tout cas pas plus. On appelle degré d'un nœud, le nombre de fils que possède ce nœud.

4.2.2 Taille et hauteur d'un arbre

On appelle la taille d'un arbre, le nombre de nœud interne qui le compose. C'est à dire le nombre nœud total moins le nombre de feuille de l'arbre. On appelle également la profondeur d'un nœud la distance en terme de nœud par rapport à l'origine. Par convention, la racine est de profondeur 0.

Dans l'exemple suivant (Figure (4.2)), le nœud F est de profondeur 2 et le nœud H est de profondeur 3.

La hauteur de l'arbre est alors la profondeur maximale de ses nœuds. C'est à dire la profondeur à laquelle il faut descendre dans l'arbre pour trouver le nœud le plus loin de la racine. On peut aussi définir la hauteur de manière récursive : la hauteur d'un arbre est le maximum des hauteurs de ses fils. C'est à partir de cette définition que nous pourrons exprimer un algorithme de calcul de la hauteur de l'arbre. La hauteur d'un arbre est très importante. En effet, c'est un repère de performance. La plupart des algorithmes que nous verrons dans la suite ont une complexité qui dépend de la hauteur de l'arbre. Ainsi plus l'arbre aura une hauteur élevée, plus l'algorithme mettra de temps à s'exécuter.

FIGURE 4.2 – *Exemple d'un arbre*

Exemple :

- A,B,C„D,E,F,G,H est l'ensemble de nœuds de l'arbre.
- Le nœud A est appelé la racine de l'arbre ;
- Le nœud E , F, G et H sont appelés les feuilles de l'arbre.
- La hauteur d'un arbre est le plus long chemin qui mène de la racine à une feuille =3.

TABLE 4.1 – *Représentation d'un nœud*

Valeur	**Ag**	**Ag**

4.3 Les arbres binaires

4.3.1 Définitions

Un arbre binaire est un cas particulier des arbres.

Un arbre binaire est un arbre où chaque nœud possède au plus deux fils : le fils droit et le fils gauche.

Même si un nœud possède un seul fils, il peut être un fils gauche ou un fils droit.

Un arbre binaire A est :

– Soit l'arbre vide, noté Φ .

– Soit un triplet (Ag, r, Ad) où :

– r est un nœud, appelé la racine de A.

– Ag est un arbre binaire, appelé le sous-arbre gauche de A.

– Ad est un arbre binaire, appelé le sous-arbre droit de A.

Exemple : l'arbre binaire de l'expression arithmétique (a+b)*(c-d)(Figure (4.3)).

4.3.2 Implémentation

Un nœud est une structure (ou un enregistrement) qui contient au minimum trois champs : un champ contenant l'élément du nœud, c'est l'information qui est importante. Cette information peut être un entier, une chaîne de caractère ou tout autre chose que l'on désire stocker. Les deux autres champs sont le fils gauche et le fils droit du nœud. Ces des fils sont en fait des arbres, on les appelle généralement les sous arbres gauches et les sous arbres droit du nœud. Chaque nœud possède la représentation du Tableau (4.1) :

De part cette définition, un arbre ne pourra donc être qu'un pointeur sur un nœud. Voici donc une manière d'implémenter un arbre binaire :

Type Arbre = pointeur (Nœd)

Nœd = Structure

Valeur = TElement

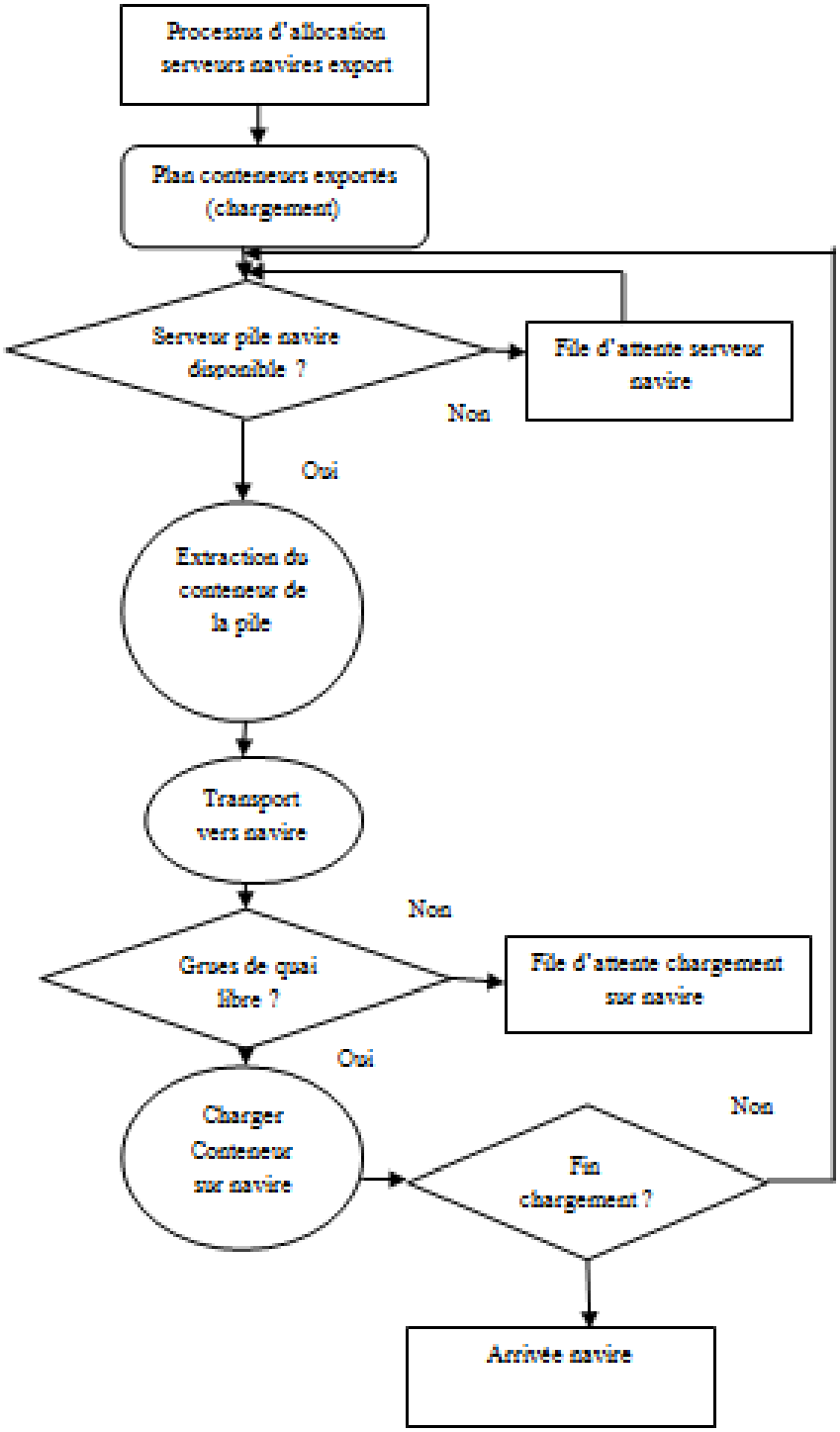

FIGURE 4.3 – *L'arbre binaire de l'expression arithmétique (a+b)*(c-d)*

Ag= Arbre (Fils Gauche) ;

Ad= Arbre (Fils Droit) ;

Fin.

On remplacera le type TElement pas type ou la structure de données que l'on veut utiliser comme entité significative des nøuds de l'arbre. De par cette définition, on peut donc aisément constater que l'arbre vide sera représenté par la constante NULL.

Exemple : représentation chainée de l'expression (a+b)*(c-d) (Figure (4.4)).

FIGURE 4.4 – *Représentation chainée de l'expression (a+b)*(c-d)*

Lors de l'arrivée des conteneurs dangereux aux zones de stockages, on aborde le problème de stockage des conteneurs. Le résultat final c'est l'affectation des conteneurs à des emplacements disponibles dans la pile de façon optimale.

4.4 Parcours d'arbres

Un parcours d'arbres est un algorithme qui permet de visiter chacun des nœuds de cet arbre. Nous distinguerons deux types de parcours : le parcours en profondeur et le parcours en largeur. Le parcours en profondeur permet d'explorer l'arbre en explorant jusqu'au bout une branche pour passer à la suivante. Le parcours en largeur permet d'explorer l'arbre niveau par niveau. C'est à dire que l'on va parcourir tous les nœuds du niveau 1 puis ceux du niveau deux et ainsi de suite jusqu'à l'exploration de tous les nœuds.

4.4.1 Parcours en profondeur

On définit trois parcours en profondeur privilégiés qui sont :

- Le parcours préfixé (pré-ordre),
- Le parcours infixe,
- Le parcours postfixé (post-ordre).

De manière formelle : Les parcours préfixé (infixé, suffixé) sont définis comme suit :

- Si A est l'arbre vide, alors pref(A) = inf(A) = suf(A) =Φ ;

– Si A = (Ag ; r ;Ad), et e(r) est le nom de r, alors : pref(A) = e(r)pref(Ag)pref(Ad) inf(A) = inf(Ag)e(r)inf(Ad) suf(A) = suf(Ag)suf(Ad)e(r)

Exemple : Soit l'arbre (Figure (4.5)).

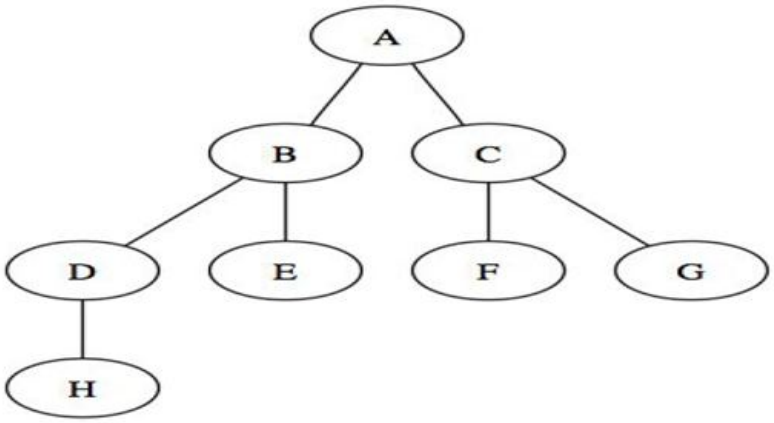

FIGURE 4.5 – *Exemple de parcours d'un arbre*

– **Préfixe :** ABDHJECFGH
– **Infixé :** JHDBEAFCIG
– **Postfixé :** JHDEBFIGCA

Le parcours préfixé (pré-ordre)

Le parcours préfixé consiste à effectuer les opérations suivantes :

– Traitement de la racine ;
– Parcours du sous-arbre gauche ; NGD (Noeud, Gauche, Droite).
– Parcours du sous-arbre droit.

La procédure de parcours préfixé est comme suit :

Procédure parcoursprofprefixe(A : arbre)

Si non EstVide(A) alors

Traiterracine(A) ;

parcoursprofprefixe(FilsGauche(A)) ;

parcoursprofprefixe(FilsDroit(A)) ;

Fin si ;

Fin ;

Le parcours infixé

Le parcours infixé consiste à effectuer les opérations suivantes :

- Le parcours du sous-arbre gauche ;
- Traitement de la racine ; GND (Gauche, Noeud, Droite).
- Parcours du sous arbre-droit.

La procédure de parcours infixé est comme suit :

Procédure parcoursprofinfixe(A : arbre) ;

Si non EstVide(A) alors

parcoursprofinfixe(FilsGauche(A)) ;

traiterracine(A) ;

parcoursprofinfixe(FilsDroit(A)) ;

Fin si ;

Fin ;

Parcours postfixé ou suffixé

Le parcours postfixé consiste à effectuer les opérations suivantes :

- Parcours du sous-arbre gauche ;
- Parcours du sous-arbre droit ; GDN (Gauche, Droite, Noeud).
- Traitement de la racine .

La procédure de ce type de parcours en profondeur est comme suit :

Procédure parcoursprofsuffixe(T : arbre)

Si non EstVide(T) alors

parcoursprofsuffixe(FilsGauche(T)) ;

parcoursprofsuffixe(FilsDroit(T)) ;

traiterracine(T) ;

Fin si ;

Fin

4.4.2 Parcours en largeur (ou par niveau)

Nous allons aborder un type de parcours un peu plus compliqué, c'est le parcours en largeur. Il s'agit d'un parcours dans lequel, on traite les nœuds un par un sur un même niveau. On passe ensuite sur le niveau suivant, et ainsi de suite.

Le parcours en largeur de l'arbre précédant est : ABCDEFGHIJ.

Une méthode pour réaliser un parcours en largeur consiste à utiliser une structure de données de type file d'attente.

Le principe est le suivant :

Lorsque nous somme sur un nœud nous traitons ce nœud (par exemple nous l'affichons) puis nous mettons les fils gauche et droit non vides de ce nœud dans la file d'attente, puis nous traitons le prochain nœud de la file d'attente.

Au début, la file d'attente ne contient rien, nous y plaçons donc la racine de l'arbre que nous voulons traiter. L'algorithme s'arrête lorsque la file d'attente est vide. En effet, lorsque la file d'attente est vide, cela veut dire qu'aucun des nœuds parcourus précédemment n'avait de sous arbre gauche ni de sous arbre droit. Par conséquent, on a donc bien parcouru tous les nœuds de l'arbre.

Application : arbres d'expressions

Une expression arithmétique peut-être représentée de trois façons différentes : infixée, postfixée et préfixée.

Exemple : l'expression arithmétique a+b ;

- Exp. Préfixée : + a b
- Exp. Infixée : a + b
- Exp. Postfixée : a b +

Les expressions post-fixées et préfixées ne nécessitent pas de parenthèses pour être évaluées. Les expressions infixées doivent être transformées en expressions postfixées ou préfixées pour pouvoir être évaluées par les langages de programmation.

4.5 Les fonctions de base sur la manipulation des arbres

Afin de faciliter notre manipulation des arbres, nous allons créer quelques fonctions :

- **La fonction qui détermine si un arbre est vide**

  Fonction EstVide (T : Arbre) renvoie un booléen

  Si T== Null alors renvoyer Vrai

Sinon

renvoyer faux ;

FinSi ;

Fin ;

– **Les fonctions qui vont nous permettre de récupérer le fils gauche ainsi que le fils droit d'un arbre**

Il faut faire attention à un problème : le cas où l'arbre est vide. En effet, dans ce cas, il n'existe pas de sous arbre gauche ni de sous arbre droit. Pour régler ce problème nous décidons arbitrairement de renvoyer l'arbre vide comme fils d'un arbre vide.

Fonction FilsGauche (T : Arbre) renvoie un arbre

Si EstVide (T) alors renvoyer Arbre Vide

Sinon

renvoyer SousArbreGauche ;

FinSi ;

Fin ;

La fonction qui retourne le fils droit sera codée de la même manière mais tout simplement au lieu de renvoyer le fils gauche, nous renvoyons le fils droit.

– **La fonction qui détermine si le nœud est une feuille**

Fonction EstUneFeuille (T : Arbre) renvoie booléen

Si EstVide (T) alors renvoyer faux

Sinon

Si EstVide (FilsGauche (T)) et EstVide (FilsDroit (T)) alors

renvoyer vrai ;

FinSi ;

Fin ;

4.6 Arbres binaires de recherche

La recherche d'un élément dans un tableau contenant des données ordonnées était très efficace : recherche binaire, en temps logarithmique. Cependant, l'insertion et la suppression d'un élément dans un tableau est très inefficace. Inversement, dans le cas d'une liste chaînée, l'insertion et la sup-

pression sont très efficaces, mais la recherche d'un élément se fait par une procédure séquentielle. Comment combiner les deux avantages (recherche efficace, et insertion/suppression efficace) : Arbre binaire de recherche.

4.6.1 Définition

Un arbre binaire de recherche est un arbre binaire ayant les propriétés suivantes (définition récursive) :

1. Tous les nœuds du sous arbre gauche de la racine ont des valeurs inférieures ou égales à la valeur de la racine.
2. Tous les nœuds du sous arbre droit de la racine ont des valeurs strictement supérieures à la valeur de la racine.
3. Les sous-arbre gauche et droit sont eux mêmes des arbres binaires de recherche.

Dans la suite, on travaillera avec un arbre binaire représenté de façon chaînée par la représentation fils gauche, fils droit. Pour un nœud N :

- N.G (resp. N.D) représente la racine du sous-arbre gauche (resp. droit) de N (ou l'arbre vide si ce sous arbre gauche (resp. droit) est vide).
- N.Valeur représente la valeur contenue dans le nœud N.

Exemples Les clés sont des chiffres (Figure (4.6)).

FIGURE 4.6 – *Exemple d'un arbre binaire de recherche*

4.6.2 Adjonction d'un élément

Dans un arbre binaire de recherche , l'opération de l'adjonction se décompose en deux étapes :

- Une étape de recherche, qui renvoie des informations sur la place où doit être inséré le nouvel élément ;
- Une deuxième étape de l'adjonction proprement dite.

Insertion aux feuilles

Dans cette méthode, on rattache un nouveau nœud à une feuille de l'arbre de recherche. L'ajout est précédé d'une phase de recherche de la place

d'insertion pour qu'après insertion la propriété d'arbre binaire de recherche soit respectée.

La procédure récursive est donnée par :

Procédure Ajoutefeuille(X, A) ;

Rôle : Adjonction d'un élément X aux feuilles d'un arbre binaire de recherche A

Paramètre Donnée

X : <type de base> la valeur à ajouter

Paramètre Donnée/Résultat

A : ABR on travaille sur l'arbre A à modifier

Début

Si A = Vide

Alors

A ← Créer ABR()

A.Valeur ← X

A.G ← Vide

A.D ← Vide

Sinon

Si X ← A.Valeur

Alors Ajoutefeuille(X,A.G)

Sinon Ajoutefeuille(X,A.D) ;

Fin Si ;

Fin Si ;

Fin ;

Adjonction à la racine

Pour pouvoir mettre une valeur X à la racine de l'arbre de recherche, il faut que toutes les valeurs précédemment contenues dans l'arbre de recherche soient séparées en deux arbres de recherche :

– un arbre qui sera le sous-arbre gauche de la racine et qui contiendra des valeurs inférieures ou égales à X,

On considère deux étapes :

– Une première étape qui consiste à couper l'arbre binaire de recherche en deux sous-arbres binaires de recherche.
– Une deuxième étape qui consiste à rassembler les nouveaux sous-arbres crées autour de la nouvelle racine X.

Première étape

La procédure de coupure de l'arbre binaire de recherche initial en deux sous-arbres binaires de recherche est donnée par :

Procédure Couper(X, A, G, D)

Rôle : coupure de l'ABR A, selon l'élément X, en deux ABR G et D ;

G contient tous les éléments de A inférieurs ou égaux à X, et D contient tous les éléments de A

strictement supérieurs à X.

Paramètre Donnée

X : <type de base> la valeur autour coupe

Paramètre Donnée/Résultat

A : ABR ABR à couper

Paramètres Résultat

G, D : ABR deux nouveaux ABRs

Début

Si A = Vide

Alors

G ← Vide ;

D ← Vide

Sinon

Si X = A.Valeur Alors

D ← A ;

Couper(X, A.G, G, D.G)

Sinon

G ← A ;

Couper(X, A.D, G.D, D) ;

Fin Si ;

Fin Si ;

Fin ;

Deuxième étape

La procédure de rassemblement des deux ABRs autour de la racine X est donnée par :

Rôle : la procédure réalise l'adjonction de l'élément X à la racine de l'arbre binaire de recherche A ; elle utilise la procédure de coupure d'un arbre binaire de recherche

Paramètre Donnée

X : <type de base> la valeur à insérer

Paramètre Donnée/Résultat

A : ABR ABR dans lequel on insère X

Variable

R : ABR de type ABR

Début

R ← Créer ABR() ;

R.Valeur ← X ;

Couper(X, A, R.G, R.D) ;

A ← R ;

Fin ;

4.7 Suppression d'un élément

Dans un arbre binaire de recherche, l'opération de la suppression se décompose en deux étapes :

- En une étape de recherche ;
- Une deuxième étape de suppression qui dépend de la place de l'élément X dans l'arbre binaire de recherche.

Pour cette suppression, plusieurs cas sont possibles :

1. Si le nœud contenant la valeur est sans fils, on peut supprimer le nœud directement.
2. Si le nœud contenant la valeur possède un fils, on peut remplacer le nœud par son fils, et on obtient directement un arbre binaire de recherche.
3. Si le nœud contenant la valeur possède deux fils, plusieurs solutions sont possibles.

Dans ce cas, remplacer le nœud à supprimer par la feuille du sous-arbre gauche contenant la valeur maximale, ou par la feuille du sous-arbre droit contenant la valeur minimale.

Fonction de suppression de l'élément Max dans arbre binaire de recherche :

Fonction Sup Max(A) : <type de base>

Rôle : cette fonction supprime le noeud contenant l ?élément maximum dans un ABR A non vide ; En résultat, elle renvoie l ?élément Max de A, qui est privé de ce maximum.

Paramètre Donnée/Résultat

A : ABR ABR : on cherche (privé) de Max

Variable

Max : <type de base> Max élément à chercher

Début

Si A.D = Vide Alors

Max $\leftarrow$ *A.Valeur ;*

A $\leftarrow$ *A.G ;*

Retourner (Max)

Sinon

Retourner (Sup Max(A.D)) ;

Finsi ;

Fin ;

Exercices avec solutions 5

Sommaire

Ce tte partie, est consacrée aux exercices avec solutions sur les différentes notions traitées dans ce polycopié.

5.1 Exercices sur les fonctions et les procédures avec solutions

Dans cette section, nous donnons quelques exercices avec solutions sur les procédures et les fonctions.

Exercice 01

Écrire une fonction max qui retourne le maximum entre deux entier a et b.

Intégrer cette fonction dans l'algorithme.

Solution

Algorithme maximum ;
Var x, y : entier ;
fonction max (a,b : entier) : entier ;
Début
Si (a > b) alors
max = a
sinon max = b ;
Finsi ;
FinFonction ;
Début
Ecrire (donner la valeur du premier nombre) ;
Lire (x) ;
Ecrire (donner la valeur du deuxieme nombre) ;
Lire (y) ;
Ecrire (max(x,y)) ; //Appel de la fonction//
Fin.

Exercice 02

Écrire une fonction qui permet de savoir si un entier est divisible par un autre.

Solution

Fonction logiqueDivise (a : entier, b : entier) : Booléen ;
Début
Si (a mod b = 0) alors
Retourner vrai
Sinon
Retourner faux ;
Finsi ;
Fin ;

Exercice 03

Écrire une fonction qui retourne la somme des entiers pairs inférieurs ou égaux à un entier n donné.

Solution

Fonction SommePairs (n : entier) : entier ;
somme : entier ;
Début

somme=0 ;
Pour i de 1 à n Faire
Si i Mod 2= 0 alors
somme= somme+i ;
Ecrire (somme) ;
FinSi ;
Fin ;

Exercice 04

Écrire la procédure qui échange les valeurs de deux entiers. Intégrer cette procédure dans l'algorithme.

Solution

Algorithme permutation ;
Var a, b : réel ;
Procédure Echange (x : réel, y : réel) ;
variables z : réel ;
Debut
z=x ;
x =y ;
y =z ;
Fin ;
Début
Ecrire (donner la valeur du premier nombre) ;
Lire (a) ;
Ecrire (donner la valeur du deuxième nombre) ;
Lire (b) ;
Ecrire (Echange(a,b)) ; //Appel de la procédure//
Fin.

Exercice 05

Écrire une procédure qui reçoit un nombre réel, comme paramètre, teste s'il est négatif, positif ou nul et affiche le résultat à l'écran. Intégrer cette procédure dans l'algorithme.

Solution

Algorithme determination signe ;
Var X : réel ;
Procedure signe(a :réel) ;
Debut
Si (a>0) alors
Ecrire (le nombre est positif)
Sinon
Si (a<0) alors
Ecrire (le nombre est negatif)
Sinon
Ecrire (le nombre est nul) ;
Finsi ;
Finsi ;
Fin ;
Debut

Ecrire(Entrer un nombre réel) ;
Lire(x) ;
Ecrire (Signe(X)) ; / Appel de la procédure*/*
Fin.

Exercice 06

Écrire un algorithme utilisant une procédure qui calcule une somme de 100 nombres.

Solution

Algorithme essai ;
Variables I, S : entier ;
Procedure Somme ;
Debut
S = 0 ;
Pour i de 1 à 100 Faire
S = S + i ;
FinPour ;
Ecrire (La somme des 100 premiers nombres est, S) ;
Fin ;
Debut
Somme ;
Fin.

5.2 Exercices sur la récursivité

Cette section présente quelques exercices sur la notion de récursivité avec des solutions.

Exercice 01

Écrire une procédure récursive qui permet d'afficher la valeur binaire d'un entier n.

Solution

Procédure binaire (n : entier) ;
Si (n<>0) alors
binaire (n/2) ;
Ecrire (n mod 2) ;
Finsi ;
Fin ;

Exercice 02

En remarquant que $n^2=(n-1)^2+2n-1$;
Écrire une fonction Récursive qui calcule le carré d'un entier positif.

Solution

Fonction carre(n :entier) :entier ;
Début
si (n = 0) Retourner 0 ;
Sinon
*Retourner carre(n - 1) + 2 * n - 1 ;*

Fin

Exercice 03

Écrire une fonction récursive qui calcule le pgcd de deux nombres entiers positifs.

On utilise pgcd(a, b) = a si b = 0 et pgcd(a, b) = pgcd(b, a mod b).

Solution

Fonction pgcd(a :entier, b :entier) :entier ;
Debut
Si (a < b) alors
Retourner pgcd(b, a) ;
Sinon
Si (b = 0)
alors retourner a ;
Sinon
Retourner pgcd(b, a / b) ;
Fin

Exercice 04

Écrire une fonction récursive qui détermine si une chaine de caractères est un palindrome ou non.

Solution

Fonction palindrome (s :chaine de caractères) : booleen ;
var taille : entier ;
Début
taille=s.lenght ;
si (taille(s) = 0 ou taille(s) =1)
palindrome(s)=VRAI
Sinon
Si (caractère(s,0) = caractère(s,taille(s)-1) palindrome(s) = palindrome(sousChaine(s,1, taille(s)-1)
Sinon
palindrome(s) = FAUX ;
Finsi ;
Finsi ;
Fin

Exercice 05

Écrire une fonction récursive qui calcule la somme des n premiers entiers.

Solution

Fonction somme (n : entier) : entier ;
Debut
Si n=0 alors somme =0
Sinon
Somme=n+somme (n-1) ;
Finsi ;
Fin ;

5.3 Exercices sur les piles et les files

Dans cette section, nous présentons des exercices corrigés sur les structures séquentielles : pile et file.

Exercice 01

1. En s'inspirant de la structure de la pile vue en cours, illustrer le résultat de chacune des opérations de la séquence : Empiler(4), Empiler(1), Empiler(3), Depiler(),Empiler(8), Depiler() sur une pile initialement vide (le tableau contient 6 cases).
2. En s'inspirant de la structure de la file vue en cours, illustrer le résultat de chacune des opérations de la séquence : Enfiler(4), Enfiler(1), Enfiler(3), Defiler(), Enfiler(8), Defiler().

Solution

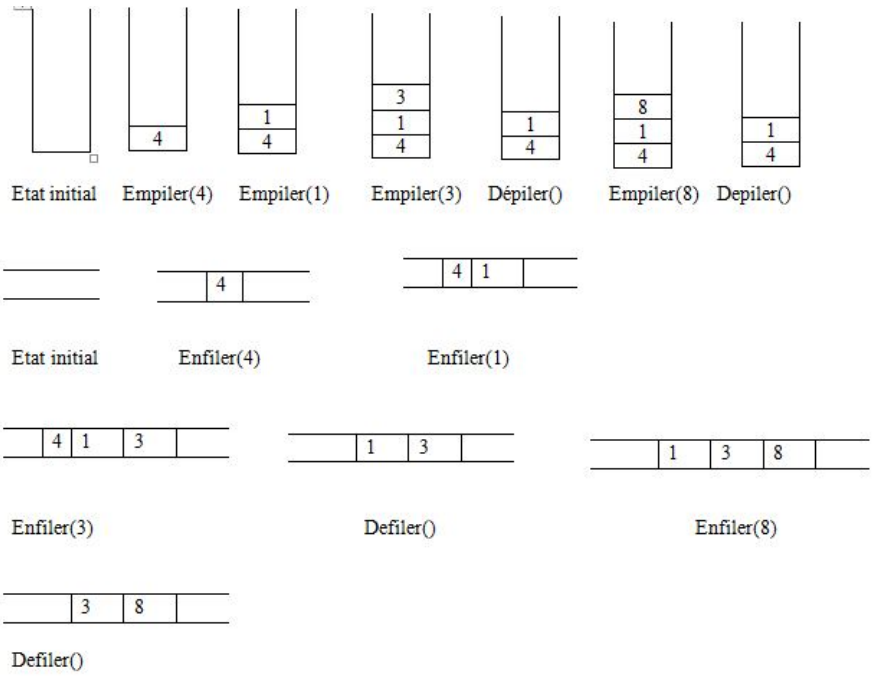

Figure 5.1 – *Les différents états de la pile et de la file*

Exercice 02

Écrire un programme qui inverse une chaine de caractères en utilisant la structure de pile et les primitives vues en cours.

Solution

Algorithme inverse
Var Caract : chaine de caractères ;
Début
Carat = '' ;
Initialiser-pile() ;
Ecrire (Entrer une chaine de caractères) ;
Tant que (Carat ≠ fin-de-ligne()) faire
Lire (Carat) ;

TABLE 5.1 – *Représentation d'un nœud*

C	A	G	B	E	I	D	F	Z

Empiler (Carat) ;
Fin tant que ;
Tant que (non-pile-vide()) faire
Ecrire (Depiler()) ;
Fin tant que ;
Fin.

Exercice 03

Écrire une procédure qui inverse le contenu d'une pile en utilisant une file.

Solution

Procedure inverser(Pile) ;
var F :File ;
DEBUT
f <- fileVide() ;
TANT QUE (NON estVide(p)) FAIRE
DEBUT
f <- ajouter(f, sommet(p)) ;
FIN ;
TANT QUE (NON estVide(file)) FAIRE
DEBUT
p <- empiler(p, dernier(f)) ;
FIN ;
RETOURNER p ;
FIN

5.4 Exercices sur les arbres

Cette dernière section est consacrée aux exercices avec solutions sur les structures hiérarchiques arbres.

Exercice 01

Voici une liste aléatoire de 9 clés Tableau (5.1) :Notez que vous pouvez faire cet exercice en prenant une autre liste aléatoire ! ; évidemment, il y a peu de chances que vous obteniez le même résultat.

1. Rappelez les propriétés des arbres binaires de recherche.
2. Rappelez ce qu'est l'opération d'adjonction aux feuilles.
3. Construire l'arbre binaire de recherche par adjonction des valeurs aux feuilles, dans l'ordre de la liste. On s'attachera particulièrement à expliquer le raisonnement.
4. Ajouter la valeur « H, K, R » dans l'arbre de recherche obtenu.

Solution

1. Un arbre binaire de recherche est un arbre binaire ayant les propriétés suivantes (définition récursive) :
   - Tous les nœeuds du sous arbre gauche de la racine ont des valeurs inférieures ou égales à la valeur de la racine.
   - Tous les nœeuds du sous arbre droit de la racine ont des valeurs strictement supérieures à la valeur de la racine.
   - Les sous-arbre gauche et droit sont eux mêmes des arbres binaires de recherche.
2. Dans un ABR, l'opération de l'adjonction se décompose en deux étapes :
   - Une étape de recherche, qui renvoie des informations sur la place où doit être inséré le nouvel élément ;
   - Une deuxième étape de l'adjonction proprement dite.
3.

Exercice 02

Écrire une fonction Taille(x) prenant un arbre binaire et rendant le nombre de ses éléments.

Solution
Fonction Taille (X : Arbre) : entier ;
Début
Si (Estvide(X)) alors
Retourner 0
Sinon
Retournet 1+ Taille (Gauche (X))+ Taille (Drtoi(X)) ;
Finsi ;
Fin

Bibliographie

Ce document a été préparé à l'aide de l'éditeur de texte GNU Emacs et du logiciel de composition typographique LaTeX 2_ε.