Support de cours Structures de données avancées

Dr. ASSIE Brou Ida

Table des matières

Chapitre 1 : Généralités sur les structures de données

La programmation est utilisée pour résoudre les problèmes qui amènent souvent à analyser, à lire ou à mémoriser des données. Pour cela, un algorithme à travers des petites opérations est utilisé pour effectuer des actions simples. Pour réduire le temps de calcul (c'est-à-dire, le temps pour rendre un résultat) et l'espace en mémoire (le nombre d'informations à stocker pour faire les calculs), il est préférable de les organiser, selon le but à atteindre, dans un objet de type **structure de données**.

Ce chapitre fera donc l'objet de la généralité sur les structures de données.

I- Intérêt de l'utilisation des structures de données

Pour montrer l'importance des structures de données, considérons le cas suivant :

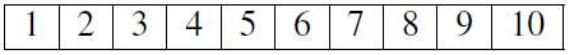
Les numéros dans les annuaires téléphoniques sont présentés par nom, par profession, par numéro téléphonique, par rue ou une combinaison quelconque de ces classements. À chaque usage, correspond une structure d'annuaire appropriée.

En organisant d'une certaine manière les données, on permet un traitement automatique de ces dernières plus efficace et plus rapide. D'où l'intérêt d'utiliser une structure de données appropriée.

II- Définition d'une structure de données

Une structure de données est un ensemble organisé de données reliées logiquement pour faciliter leur traitement. C'est aussi un moyen particulier d'organiser les données dans un ordinateur afin qu'elles puissent être utilisées efficacement.

Par exemple, nous pouvons stocker une liste d'éléments ayant le même type de données à l'aide de la structure de données « Tableau ».



III- Différents types de structures de données

Il existe différents types de structures de données pour des données différentes ou répondant à des contraintes algorithmiques différentes :

- **Tableaux** : pour stocker des éléments homogènes à des emplacements contigus. La taille d'un tableau peut être fournie avant de stocker des données.
- **Enregistrements** : sont des structures de données dont les éléments peuvent être de type différent, contrairement aux tableaux qui sont des structures de données dont tous les éléments sont de même type.
- **Fichiers** : sont un ensemble d'enregistrements qui sont tous de même type. Un fichier peut être conservé sur un support externe à la mémoire centrale.
- **Piles et files** : Les piles et les files sont des ensembles dynamiques pour lesquels l'élément à supprimer *via* l'opération SUPPRIMER est défini par la nature intrinsèque de l'ensemble.
- **Listes chainées, Arbres binaires, Graphes, Tas, Hachage, ...**

Chapitre 2 : Les enregistrements

Les tableaux sont les structures de données les plus connus. Tous les éléments sont homogènes et dans certaines applications, ces types de données ne répondent pas à nos besoins car nous voulons regrouper des données de différents type différents. Par exemple, le nom et l'âge d'une centaine de personnes. Il est donc utile de pouvoir rassembler des éléments de type différents. On parle alors d'enregistrement ou de structure.

I- Définition d'un enregistrement

Un enregistrement est une structure de données qui permet de rassembler sous un même nom des données de types différents (structure hétérogène).

L'avantage d'un enregitrement est de regrouper des informations sémantiquement liées pour éviter de multiplier les variables et bien faire apparaitre la logique du traitement.

Exemple : Fiche d'un étudiant de UPB

- Nom : chaine de caractères
- Age : entier
- Sexe : typé énumératif (masculin ou féminin)
- Matricule : Entier

II- Déclaration d'un enregistrement

La syntaxe générale de la déclaration d'un enregistrement :

Type

nom_Enregistrement = **Enregistrement**

nomduchamp1 : type1;
nomuchamp2 : type2;
...

Fin Enregistrement ;

Exemple :

```
Type
   t_personne =  enregistrement
      nom : chaine de caractères;
      age : entier;
    Fin ;
```

Le type enregistrement peut ensuite être utilisé pour définir des variables comme suit :

```
var
   joueur1, joueur2 : t_personne;
   tabpersonne : tableau [1..100] de t_personne;
```

III- Utilisation d'un type enregistrement

Pour donner une valeur à un "champ" de l'enregistrement, on écrit un point « . » entre le nom de la variable et le nom du champ :

« nomdevariable ». « nomduchamp» ← expression

Pour utiliser la valeur d'un champ dans une expression, c'est pareil, on utilise un « . » entre le nom de la variable et le nom du champ. Par exemple, avec le type et la variable déclarés ci-dessus, on a :

```
Ecrire ("Entrez votre nom.");
Lire( joueur1.nom);
```

IV- Exercice d'application

Dans cet exercice, nous allons reprendre le schéma de l'enregistrement vu en cours et concernant des étudiants.

Nous avons au plus 500 étudiants et pour chaque étudiant nous voulons avoir les informations suivantes :

- Le matricule ,
- le nom et prénom,

- le groupe,
- les notes en tp, en projet, sa moyenne et son rang.

1- Quelle est la structure de données convenable pour ce problème ?
2- Construire 2 modules : le premier qui permet de rentrer toutes les informations concernant un étudiant et toutes ses notes en algorithmique, et le second (MOYALGO) qui calcule sa moyenne.

Chapitre 3 : Les pointeurs

Toute variable manipulée dans un programme est stockée en mémoire centrale de l'ordinateur (cf. au cours d'architecture des ordinateurs). La mémoire peut être assimilée à une armoire de rangement ou à un tableau dont chaque élément est identifié par une adresse ou un numéro. Ainsi, pour retrouver une variable en mémoire de l'ordinateur, il faut connaitre l'élément adresse dans lequel elle est stockée. C'est le compilateur qui assure cette fonction en reliant l'identifcateur de la variable à son adresse mémoire. Il serait donc plus intéressant de décrire une variable non plus par son identificateur mais par son adresse mémoire. D'où, le recours au type d'objets les **pointeurs**.

I- Notion de pointeur

Un pointeur est un objet or une Lvalue dont la valeur est égale à l'adresse d'un autre objet. On appelle Lvalue (left value) toute expression du langage pouvant être placé à gauche d'un opérateur d'affectation. Une Lvalue est caractérisée par :

- son adresse : l'adresse mémoire à partir de laquelle l'objet est stocké ;
- sa valeur : ce qui est stocké à cette adresse.

Exemple : Exemple d'une Lvalue

int i ; i= 1; j=i ;

Interpretation :

Si le compilateur a placé la variable i à l'adresse 4831830000 et j à l'adresse 4831830004, alors, on a :

Lvalue	Adresse	Valeur
i	4831830000	1
j	4831830004	1

L'opérateur unaire « & » permet d'accéder à l'adresse d'une Lvalue. Pour l'exemple ci-dessus, l'accès à l'adresse de la variable i est donné par l'instruction suivante :

Afficher ('l'adresse de la variable i est', &i) ;

L'écran va présenter :

L'adresse de la variable i est 4831830000

II- Déclaration d'un pointeur

On déclare un pointeur par l'instruction :

type *nom-du-pointeur;

« type » est le type de l'élément pointé.

Exemple : Déclaration d'un pointeur p sur un entier

```
i n t i =3;
i n t ∗p ;
p = &i ;// p reçoit l'adresse de la variable i
```

III- Accès à la valeur d'un élément pointé

L'accès à la valeur d'un élément pointé par un pointeur se fait par l'utilisation d'un nouvel opérateur unaire « * » comme suit :

Exemple : En reprenant l'exemple précédant de la déclaration d'un pointeur p sur un entier, l'accès à la valeur de la variable i par le pointeur p est donné par :

```
i n t i =3;
i n t ∗p ;
p = &i ;// p reçoit l'adresse de la variable i
Afficher ('l'adresse de la variable i est', *p) ;
```

IV- Allocation dynamique des pointeurs

La valeur par defaut d'un pointeur est la constante symbolique notée NULL. La valeur de cette constante vaut 0.

Avant toute utilisation d'un pointeur, il faut l'initialiser soit par :

- l'affectation d'une valeur nulle à un pointeur : p = NULL ;
- l'affectation de l'adresse d'une autre variable (lvalue) : p = &i ;
- l'allocation dynamique d'un nouvel espace-mémoire.

- **Notion d'allocation dynamique :** L'allocation dynamique est l'opération qui consiste à réserver un espace-mémoire d'une taille définie.

Tout espace-mémoire alloué dynamiquement doit obligatoirement être désalloué, sinon il y a un problème de fuite mémoire.

V- Exercice d'application

Chapitre 4 : Les piles

Une pile est une structure de données qui stocke de manière ordonnée des éléments, mais rend accessible uniquement un seul d'entre eux, appelé le **sommet de la pile**. Quant on ajoute un élément à la pile, celui-ci devient le sommet de la pile. Quant on retire un élément de la pile, on retire toujours le sommet, et le dernier élément ajouté avant lui devient alors le sommet de la pile. Autrement dit, le dernier élément ajouté dans la pile est le premier élément à en être retiré. On dit que la pile met en oeuvre le principe LIFO (*Last In, First Out*).

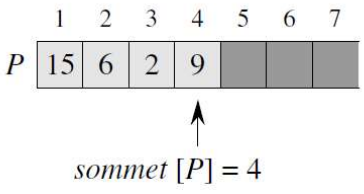
Il existe deux (02) manières de représenter une pile :

- Un tableau,
- Une liste chainée (nous y reviendrons au chapitre 4).

I- Modélisation par un tableau

L'implémantation d'une structure de données pile peut se faire par un tableau, c'est-à-dire, elle consiste à utiliser un tableau pour représenter la pile. Un tableau est une structure de données permettant de stocker des données de même type. Il est caractérisé par sa taille, le type des éléments qu'il contient et son nom.

Avec un tableau P, il est donc possible d'implémenter une pile d'au plus n éléments si la taille de celui-ci vaut n. Le tableau possède un attribut appelé *sommet*[P] qui indexe l'élément le plus récemment inséré. La pile est constituée des éléments P[1], ..., P[*sommet*[P]], où P[1] est l'élément situé à la base de la pile et P[*sommet*[P]] est l'élément situé au sommet. La figure suivante représente une pile par cette modélisation :



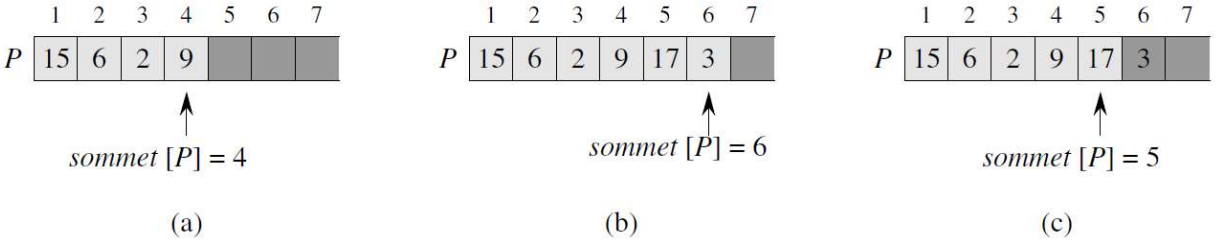
Il s'agit de l'implémentation d'une pile P via un tableau. Les éléments de la pile apparaissent uniquement aux positions en gris clair. La pile P contient 4 éléments dont l'élément sommet est 9.

II- Opérations sur une pile

Le type PILE sera utilisé pour représenter une pile au sens général sans se soucier de sa modélisation. Les opérations sur une pile sont :

- INITIALISERPILE(*P*)
- PILEVIDE(*P*)
- EMPILER(*P, x*)
- DÉPILER(*P*)

La figure suivante présente ces opérations :



➢ **Interpretation** : **(a)** La pile *P* contient 4 éléments. L'élément sommet est 9.
(b) L'état de la pile *P* après les appels EMPILER(*P,* 17) et EMPILER(*P,* 3).
(c) L'état de la pile *P* après que l'appel DÉPILER(*P*) a retourné 3, qui est l'élément le plus récemment empilé.

1. InitialiserPile (P)

L'opération INITIALISERPILE(*P*) permet d'initialiser la pile, c'est-à-dire, elle fait en sorte que la pile soit vide. L'algorithme suivant permet d'implementer cette opération :

```
INITIALISERPILE (P)
DEBUT
    sommet[P] = 0 ;
FIN
```

1. PileVide (P)

L'opération PILEVIDE(*P*) est de type booléen et permet d'indiquer si la pile est vide. L'algorithme suivant permet d'implementer cette opération :

```
PILEVIDE(P)
    Si sommet[P] = 0  Alors
        retourner VRAI
    Sinon
        retourner FAUX
```

2. Empiler(*P*,x)

Cette opération est de type booléen et permet d'empiler l'élément x au sommet de la pile P. VRAI est retournée si l'opération a réussi.

L'algorithme suivant permet d'implementer cette opération :

```
EMPILER(P, x)
    sommet[P] ← sommet[P] + 1
    P[sommet[P]] ← x
```

3. Depiler(*P*,x)

Cette opération est de type booléen et permet de dépiler l'élément au sommet de la pile P. Cela revient à récupérer la valeur de l'élément au sommet avant de le supprimer de cette dernière. VRAI est retournée si la pile n'est pas vide.

L'algorithme suivant permet d'implementer cette opération :

```
DEPILER(P)
  Si PILE-VIDE(P) Alors
      erreur « débordement négatif »
  Sinon
     sommet[P] ← sommet[P] – 1
     retourner P[sommet[P] + 1]
```

III- Exercices d'application

Chapitre 5 : Les files

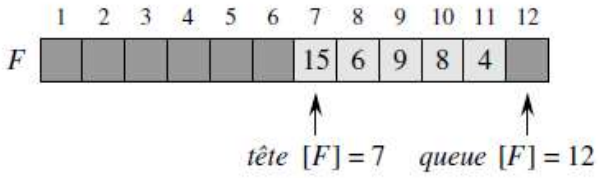
Une file est une structure de données qui stocke de manière ordonnée des éléments, mais rend accessible uniquement un seul d'entre eux, appelé la **tête de la file**. La file comporte une **tête** et une **queue.** Quant on ajoute un élément à la file, c'est à la queue de la file que celui-ci est ajouté. Quant on retire un élément de la file, on retire toujours la tête de la file. Autrement dit, le premier élément ajouté dans la file est le premier élément à en être retiré. On dit que la pile met en oeuvre le principe FIFO (*First In, First Out*).

Il existe deux (02) manières de représenter une file :

- Un tableau,
- Une liste chainée (nous y reviendrons au chapitre 6).

I- Modélisation par un tableau

L'implémantation d'une structure de données file peut se faire par un tableau, c'est-à-dire, elle consiste à utiliser un tableau pour représenter la file. L'ajout d'un élément se fait à la suite du dernier élément du tableau. Le retrait d'un élément de la file se fera en enlevant le premier élément du tableau. Il faut donc deux indices pour ce tableau, le premier qui indique le premier élément de la file et le deuxième qui indique la fin de la file. La figure suivante représente une file d'attente par cette modélisation :



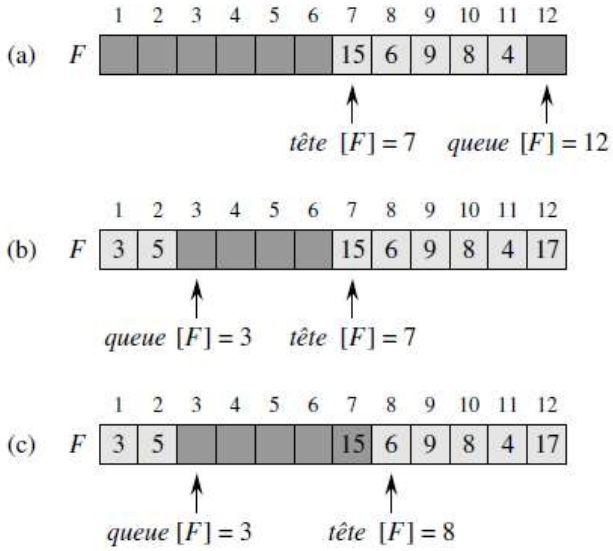
La file implémentée à l'aide d'un tableau $F[1 . . 12]$ contient 5 éléments, aux emplacements $F[7 . . 11]$. Les éléments de la file apparaissent uniquement aux positions en gris clair.

II- Opérations sur une file

Le type FILE sera utilisé pour représenter une file au sens général sans se soucier de sa modélisation. Les opérations sur une pile sont :

- INITIALISERFILE(*F*)
- FILEVIDE(*F*)
- TAILLEFILE(F)
- ENFILER(F, *x*)
- DÉFILER(F)

La figure suivante présente ces opérations :



- **<u>Interpretation</u>** : **(a)** La file contient 5 éléments, aux emplacements *F*[7 . . 11].
  (b) La configuration de la file après les appels ENFILER(*F,* 17), ENFILER(*F,* 3) et ENFILER(*F,* 5).
  (c) La configuration de la file après l'appel DÉFILER(*F*) qui retourne la valeur de clé 15 précédemment en tête de file. La nouvelle tête à la clé 6.

4. InitialiserFile (F)

L'opération INITIALISERFILE(*F*) permet d'initialiser la file, c'est-à-dire, elle fait en sorte que la file soit vide :

```
INITIALISERFILE(F)
Debut
        tete[F] =0 ;
        queue[F]=0 ;
Fin
```

5. FileVide (F)

L'opération FILEVIDE(F) est de type booléen et permet d'indiquer si la file est vide. L'algorithme suivant permet d'implementer cette opération :

```
FONCTION FILEVIDE(P) : BOOLEEN
DEBUT
SI tete[F] = 0 ALORS
    retourner(VRAI)
SINON
    retourner(FAUX)
FIN
```

6. TailleFile(F)

La fonction *Taille*(f) permet de calculer la taille de la FILE.

```
FONCTION TAILLEFILE(F) : ENTIER
DEBUT
SI FileVide (F) ALORS
    Retourner 0
SINON
    SI tete[F] ≤ queue[F] ALORS
        retourner (queue[F] - queue[F]+1)
    SINON
        retourner (queue[F] + queue[F]+1)
```

FIN

FIN

7. EnFiler(*F*,x)

Cette opération est de type booléen et permet d'enfiler l'élément x à la queue de la file F si celle-ci n'est pas pleine.

8. DeFiler(F)

Cette opération permet de défiler le premier élément de la tête de la file si celle-ci n'est pas vide.

III- Exercices d'application

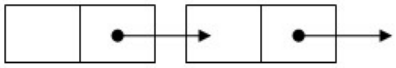
Chapitre 6 : Listes chainées

Une liste chaînée est une structure de données dans laquelle les objets sont ordonnés linéairement. L'ordre d'une liste chaînée est déterminé par un pointeur dans chaque objet. Il exite 2 tpes de liste chainée :

- Les listes simplement chainées
- Les listes doublement chainées

I- Les listes simplement chainées

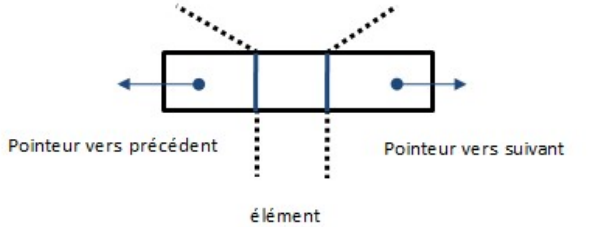
Chaque élément d'une **liste simplement chaînée** L est un objet comportant un champ clé et un champ pointeurs : succ. Étant donné un élément x de la liste, $succ[x]$ pointe sur son successeur dans la liste chaînée.

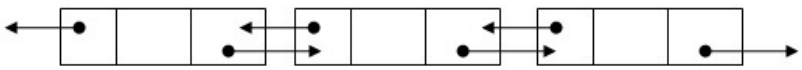


Dans les listes simplement chaînées, on parcourt la chaîne uniquement du début vers la fin.

II- Les listes doublement chainées

Chaque élément d'une **liste doublement chaînée** L est un objet comportant un champ clé et deux autres champs pointeurs : succ et préd.





Étant donné un élément x de la liste, $succ[x]$ pointe sur son successeur dans la liste chaînée et $préd[x]$ pointe sur son prédécesseur.

Si *préd*[*x*] = NIL, l'élément *x* n'a pas de prédécesseur et est donc le premier élément, aussi appelé ***tête*** de liste.

Si *succ*[*x*] = NIL, l'élément *x* n'a pas de successeur et est donc le dernier élément, aussi appelé ***queue*** de liste. Un attribut *tête*[*L*] pointe sur le premier élément de la liste.

Si *tête*[*L*] = NIL, la liste est vide.

III- Exercice d'applications