Notions de structures de données

Une **structure de données** est une manière d'organiser les données pour les traiter plus facilement. Une structure de données implémente concrètement un type abstrait.

En organisant d'une certaine manière les données, on permet faire un traitement automatique de ces dernières plus efficacement et rapidement.

L'étude des structures de données fait suite à celle de l'algorithmique. Nous utilisons dans les algorithmes classiques une structure de données de type tableau. Cette structure de données pose un problème : elle occupe la mémoire en fonction de sa déclaration. Ce qui signifie que si on ignore combien l'utilisateur va nécessiter d'indice, il va falloir réserver le tableau le plus grand possible afin d'être sûr qu'il puisse contenir autant d'élément que l'utilisateur souhaitera. Il serait plus judicieux de réserver l'espace au fur et à mesure de la demande. C'est ce que l'on appelle la gestion dynamique de la mémoire.

Les tableaux sont génériques : c'est à dire qu'au moment de sa déclaration, tous ses éléments sont d'un type donné.

Les tableaux sont homogènes : c'est à dire qu'un tableau donné ne peut contenir qu'un seul type d'élément (celui précisé dans la déclaration du tableau). Nous pourrions avoir besoin d'un tableau dont certain élément serait des entiers, d'autres des chaînes, d'autres encore des booléens. Il s'agit là de la problématique de l'hétérogénéité.

Ainsi il ne faut pas confondre la notion de généricité avec celle d'hétérogénéité. La première désigne la capacité à contenir des éléments d'un même type que l'on peut choisir en le fixant au moment de la déclaration. La seconde désigne la capacité à contenir des éléments de types différents.

Supposons que nous souhaitons utiliser des forme d'information qui ne font pas partis de type de bases : les listes chaînées, les matrices, les arbres, etc. Comment créer les éléments nécessaires au stockage de telle information quand un simple tableau ou un simple enregistrement ne suffisent plus ?

Dynamité, généricité, hétérogénéité et la création de nouveaux types non-élémentaires sont autant de problèmes à prendre en compte.

Le fait d'utiliser une structure de données appropriée à un traitement informatique peut également faire baisser de manière significative la complexité d'une application informatique et ainsi contribuer à diminuer le taux d'erreurs. Différentes structures de données existent pour des données différentes ou répondant à des contraintes algorithmiques différentes :

- Structures finies :
    - constantes,
    - variables,
    - enregistrements,

- Structures indexées :
    - tableaux,
    - vecteurs ;
- Structures récursives :
    - listes,
    - arbres,
    - piles,
    - files,
    - graphes.

Notion de récursivité

La récursivité est une méthode de description d'algorithmes qui permet à une procédure de s'appeler elle-même (directement ou indirectement). Une notion est récursive si elle se contient elle-même en partie, ou si elle est partiellement définie à partir d'elle-même.
L'expression d'algorithmes sous forme récursive permet des descriptions concises et naturelles. Le principe est d'utiliser, pour décrire l'algorithme sur une donnée **D,** l'algorithme lui-même appliqué à un ou plusieurs sous-ensembles de **D,** jusqu'à ce que le traitement puisse s'effectuer sans nouvelle décomposition. Dans une procédure récursive, il y a deux notions à retenir :
• la procédure s'appelle elle-même : on recommence avec de nouvelles données.
• il y a un test de fin : dans ce cas, il n'y a pas d'appel récursif. Il est souvent préférable d'indiquer le test de fin des appels récursifs en début de procédure.

Exemples de fonctions récursives

1 - factorielle

La factorielle d'un nombre n donné est le produit des nombres entiers inférieurs ou égaux à ce nombre n. Cette définition peut se noter de différentes façons.
Une première façon consiste à donner des exemples et à essayer de généraliser.
0! = 1
1! = 1
2! = 1 x 2
3! = 1 x 2 x3
n! = 1 si n = 0
n! = 1 * 2 * ... * (n-1) * n si n > 0
Une deuxième notation plus rigoureuse fait appel à la récurrence.
n! = 1 si n = 0
n! = n * (n-1)! si n > 0

n! se définit en fonction d'elle-même (n-1)!

Programme en C

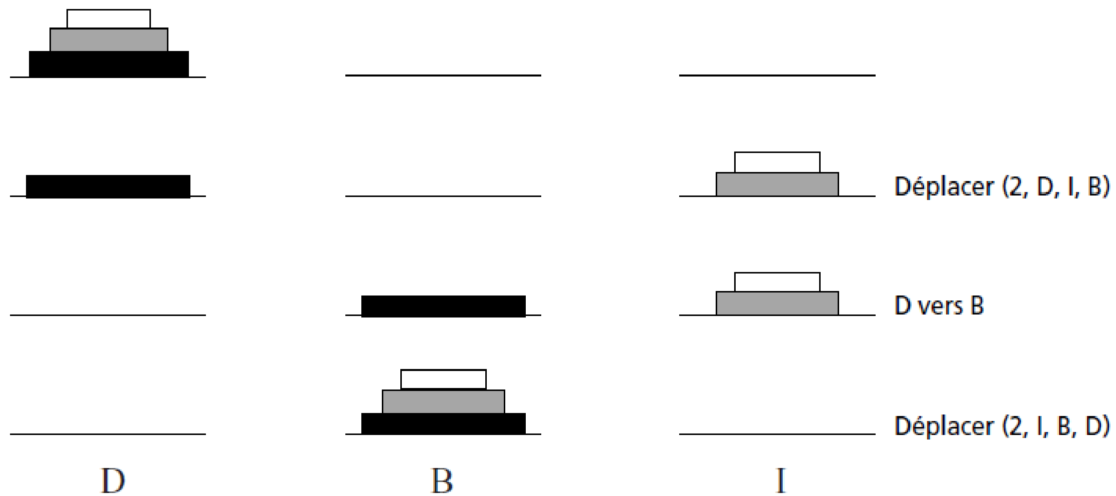
```
/*Calcul récursif de la factorielle d'un entier n >= 0 */
#include <stdio.h>
long factorielle (int n) {
if (n == 0) {
return 1;
} else {
long y = factorielle (n-1);
return n*y;
}
}
void main () {
printf ("Entier dont on veut la factorielle (n<=14) ? ");
int n; scanf ("%d", &n);
printf ("Factorielle %d est : %ld\n", n, factorielle (n));

}
```

2 - Tours de Hanoi

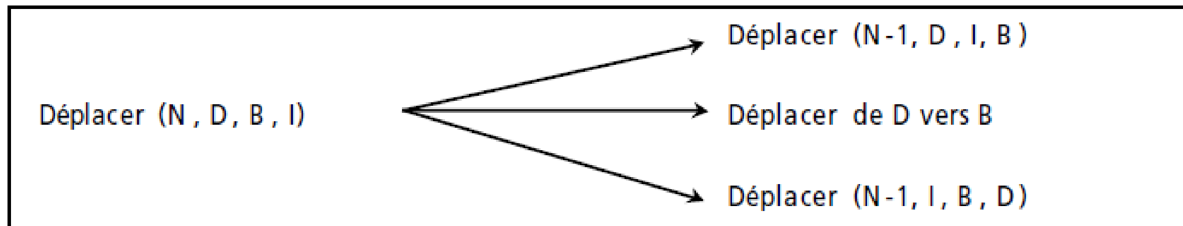
Les « Tours de Hanoi » est un jeu où il s'agit de déplacer un par un des disques superposés de diamètre décroissant d'un socle de départ D sur un socle de but B, en utilisant éventuellement un socle intermédiaire I. Un disque ne peut se trouver au-dessus d'un disque plus petit que lui.

Le schéma de la Figure suivante montre le raisonnement mettant en évidence le caractère récursif de l'algorithme à suivre. Pour déplacer un disque de D vers B, le socle I (intermédiaire) est inutile. Pour déplacer 2 disques, il faut transférer celui qui est au sommet sur le socle I, déplacer le disque reposant sur le socle D vers B, et ramener le disque du socle I au sommet de B. Pour déplacer 3 disques, il faut déplacer les 2 disques en sommet de D vers I (en utilisant B comme intermédiaire), ensuite déplacer le disque reposant sur le socle de D vers B, et ramener les 2 disques mis de côté sur I, en sommet de B. Pour déplacer 3 disques, on utilise 2 fois la méthode permettant de déplacer 2 disques sans entrer dans le détail de ces mouvements, ce qui met en évidence la récursivité.



Algorithme

D'une manière générale comme l'indique la Figure 7, pour déplacer N disques de D vers B, il faut déplacer les N-1 disques de D vers I en utilisant B comme intermédiaire, déplacer le disque restant de D vers B, ramener les N-1 disques de I vers B en utilisant D comme intermédiaire.




```
/* tours de Hanoi. Exemple de programme avec deux appels récursifs */
#include <stdio.h>
// déplacer n disques du socle depart vers le socle but
// en utilisant le socle intermédiaire.
void deplacer (int n, char depart, char but, char intermediaire) {
if (n > 0) {
deplacer (n-1, depart, intermediaire, but);
printf ("Déplacer un disque de %c --> %c\n", depart, but);
deplacer (n-1, intermediaire, but, depart);
}
}
void main () {
printf ("Nombre de disques à déplacer ? ");
int n; scanf ("%d", &n);deplacer (n, 'A', 'B', 'C');

}
```

Analyse de la complexité des algorithmes

L'**analyse de la complexité d'un algorithme** consiste en l'étude formelle de la quantité de ressources (par exemple de temps ou d'espace) nécessaire à l'exécution de cet algorithme. Celle-ci ne doit pas être confondue avec la théorie de la complexité, qui elle étudie la difficulté intrinsèque des problèmes, et ne se focalise pas sur un algorithme en particulier.

Différentes approches

L'approche la plus classique est donc de calculer le temps de calcul dans le pire des cas.

Il existe au moins trois alternatives à l'analyse de la complexité dans le pire des cas. La **complexité en moyenne** des algorithmes, à partir d'une répartition probabiliste des tailles de données, tente d'évaluer le temps moyen que l'on peut attendre de l'évaluation d'un algorithme sur une donnée d'une certaine taille. La **complexité amortie des structures de données** consiste à déterminer le coût de suites d'opérations. **L'analyse lisse d'algorithme**, plus récente, se veut plus proche des situations réelles en calculant la complexité dans le pire des cas sur des instances légèrement bruitées.

Exemple de la recherche dans une liste triée

Supposons que le problème posé soit de trouver un nom dans un annuaire téléphonique qui consiste en une liste triée alphabétiquement. On peut s'y prendre de plusieurs façons différentes. En voici deux :

1. Recherche linéaire : parcourir les pages dans l'ordre (alphabétique) jusqu'à trouver le nom cherché.
2. Recherche dichotomique : ouvrir l'annuaire au milieu, si le nom qui s'y trouve est plus loin alphabétiquement que le nom cherché, regarder avant, sinon, regarder après. Refaire l'opération qui consiste à couper les demi-annuaires (puis les quarts d'annuaires, puis les huitièmes d'annuaires, etc.) jusqu'à trouver le nom cherché.

Pour chacune de ces méthodes il existe un pire des cas et un meilleur des cas. Prenons la méthode 1 :

- Le meilleur des cas est celui où le nom est le premier dans l'annuaire, le nom est alors trouvé instantanément.
- Le pire des cas est celui où le nom est le dernier dans l'annuaire, le nom est alors trouvé après avoir parcouru tous les noms.

Problématique

- Comment stocker des données en mémoire en prenant de la mémoire de façon dynamique, en fonction du besoin. Ceci afin d'éviter les dépassements de mémoire et de ne pas mobiliser des ressources machines (parfois précieuses) inutilement.
- Comment stocker en mémoire une donnée si aucun type n'est intégré dans le langage ? Les langages ne peuvent intégrer toutes les structures de données possibles. Il faut parfois les implémenter soit-même.
- Comment, au sein d'une structure, gérer sa généricité, son hétérogénéité.