
## Pôle Informatique

# STRUCTURE DE DONNEES

| **Rédacteur** | **DE BATZ Christophe (P2014)** |
|---|---|
| Date de rédaction | 11/06/2011 (rédaction en 7h) |
| Relecteur | Nom du relecteur |
| Date de validation | jj/mm/aaaa |
| Responsable du pôle | Nom du responsable du pôle |
| Président(e) | Nom président(e) |

# Contenu

# Introduction

## Structure de Données : qu'est-ce que c'est ?

On appelle Structure de Données (ou « SDD » à l'EFREI) l'étude de structures algorithmiques permettant de contenir des données. Il y a plusieurs structures que l'on étudie à l'EFREI en deuxième année. Par exemple, nous verrons les listes chaînées, les piles et les files, les arbres, les tables de hachages…

## A quoi ça sert ?

Chacune de ces structures a été conçue pour s'adapter à un ou plusieurs cas de programmation. Ainsi on utilisera une file plutôt qu'un arbre pour gérer la mémoire tampon raccrochée au appuies clavier. Pourquoi ? Tout simplement car cette structure se prête mieux à ce cas précis et donc sa gestion en est grandement facilité !

Ces structures ont été inventées dans l'objectif de simplifier la tâche du programmeur. Ainsi, en tant que futur ingénieur EFREI, il est primordial de bien comprendre leur fonctionnement, leur utilisation et leur conception. C'est ce que nous allons essayer dans ce support de cours.

## A propos de ce cours

Voici quelques détails concernant ce support de cours :

- **Auteur du cours :** Christophe de Batz
- **Prérequis :** « Programmation en C » vu en L1 (support également disponible avec Entraide)
- **Difficulté :** Moyenne

Les codes donnés sont en C et sauf mention contraire, sont écris dans la fonction `main()`.

# Petit rappel

## Les pointeurs

Les pointeurs sont un type de variable un peu spécial. En fait, **un pointeur est une variable qui contient l'adresse en mémoire d'une autre variable**. Un pointeur pointe sur une variable. Une variable peut-être pointée par un pointeur. Voici un code d'exemple pour mieux comprendre comment gérer les pointeurs :

```
int a = 2; // création de la variable a
int *p_a = &a; // on met l'adresse de a dans le pointeur p_a

printf(« Contenu de a : %d \n », a); // affiche 2
printf(« Contenu de a : %d \n », *p_a); // affiche 2
printf(« Contenu de p_a : %p \n », p_a); // affiche une adresse
```

# Les listes chaînées

## Définition

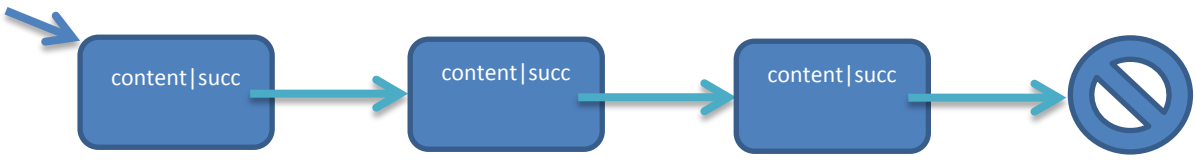
Les listes chaînées constituent la structure de données « de base ». Elles permettent de **relier des éléments entre eux** et, vous l'aurez deviné, elles sont un moyen de ne pas utiliser de tableaux statiques en C : peu pratiques et très gourmands en mémoire.

Les éléments que peut contenir une liste sont totalement dépendant de la volonté du programmeur : cela peut aller du simple entier jusqu'à la structure la plus complexe que vous puissiez imaginer !

Il y a trois sortes de listes chaînées :

- **Liste Simplement Chaînée** (« LSC »)
- **Liste Doublement Chaînée** (« LDC »)
- **Liste Circulaire**

## Illustration (cas d'une LSC)




Comme on le voit, chaque maillon est séparé en deux blocs :

- **content : la contenu du maillon** (un void* par exemple, càd n'importe quel type)
- **succ : le pointeur vers le maillon suivant** de la liste

La liste se finie par un pointeur NULL (symbolisé ici par le signe ⃠).

**Pour une LDC**, c'est la même chose sauf que chaque maillon dispose d'un troisième bloc appelé « prev » et qui est un pointeur vers le maillon précédent cette fois. Sur le schéma précédent, on aurait juste disposé les flèches dans les deux sens.

**Une liste circulaire** est une LSC ou une LDC qui n'a pas de pointeur NULL à la fin. En effet, dans ce cas, le pointeur « succ » du dernier maillon de la liste pointe vers le premier maillon (la boucle est bouclée !).

## Fonctionnement des listes chaînées

### Structure

La structure d'une liste chaînée (LSC) est plutôt simple, la voici :

```
typedef struct listNode {
  int content;
  struct listNode *succ; // , *prev si LDC
} listNode;
typedef listNode *list;
```

La structure est donc composée d'un contenu « content » et d'un lien vers la même structure « succ » qui pointe sur le maillon d'après.

### Parcours

Le principe de parcours des listes est à connaître par cœur (ou à comprendre pourquoi pas !). Il reste très simple :

```
// soit « l » une LSC
while (l != NULL) {
  // action sur chacun des maillon ici
  l = l->succ; // on passe au maillon suivant
}
```

Ce n'est pas plus compliqué et on s'arrête lorsqu'on tombe sur le pointeur NULL ( ). C'est toujours le même principe. Sachez que le « `l != NULL` » peut s'écrire « `l` » directement étant donné qu'il s'agit d'une expression booléenne au final.

### Exemples

Voici plusieurs fonctions de gestion des LDC qui vous serons importante :

1. Afficher chaque élément d'une liste

```
void affiche_liste (list l) { // l : liste d'entiers (int)
  while (l) { // ⇔ while (l != NULL)
    printf(« %d », l->content);
    l = l->succ;
}
```

2. Création d'une LSC à partir d'un tableau statique

```
list liste_depuis_tableau (int *tab, int longueur_tab) {
  if (longueur_tab <= 0)
    return NULL;
  int i = 1; // cas du premier élément
  list l = (list)malloc(sizeof(listNode)); // on créer l'élément
  list h = l; // on sauvegarde la tête de liste
  l->content = tab[0];
  while (i < longueur_tab) { // cas des autres éléments
    l->succ = (list)malloc(sizeof(listNode));
    l->succ->content = tab[i];
    l = l->succ;
    i++;
  }
  l->succ = NULL; // fin de la liste
  return h;
}
```

3. Suppression totale d'une liste en mémoire

```
void supprimer_liste (list l) {
  while (l) {
    list temp = l; // on sauvegarde temporairement le maillon en cours...
    l = l->succ; // puis on itère la liste
    free(temp); // free pour supprimer le maillon en cours
  }
}
```

4. Recherche d'un élément dans une LSC circulaire

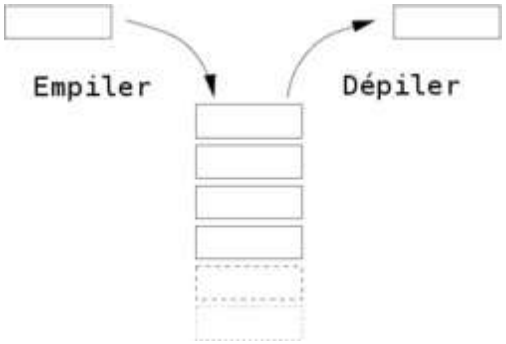
```
short recherche_element_circ (list l, int e) {
  if (l->content == e) // cas de la tête de liste
    return 1;
  list h = l; // on sauvegarde la tête
  l = l->succ;
  while (l != h) { // tant qu'on ne repasse pas par la tête
    if (l->content == e)
      return 1;
    l = l->succ;
  }
  return 0;
}
```

# Les piles et les files

## Les piles

### Définition

Les piles (ou « LIFO » pour « First In First Out ») sont une structure de donnée pouvant se représenter au moyen d'une pile d'assiettes (chaque assiette étant un maillon). Lorsque l'on insère un maillon, celui-ci vient au sommet de la pile. Lorsque l'on enlève le sommet, on enlève le dernier maillon inséré.




### Utilisations

Une pile sera utilisée pour :

- Pile dans les **appels récursifs** d'une fonction
- Pile dans la **remontée des opérations d'un tas**
- Pile pour le **parcours largeur** d'un arbre binaire

### Structure

```
typedef struct StackList {
  struct listNode *top; // on se sert des listes pour modéliser les piles et les files
  int size;
} StackList;
typedef StackList *stack;
```

### Opérations

Les principales opérations seront :

- **`push`** : empile un nouvel élément au sommet de la pile *(solution ci-après)*
- **`pop`** **:** dépile, c'est-à-dire renvoi puis supprime l'élément du sommet de la pile

Autres opérations connexes :

- **`isEmptyStack`** : renvoi vrai si la pile est vide *(solution ci-après)*
- **`getTop`** : renvoi l'élément du sommet de la pile
- **`newEmptyStack`** : créer une nouvelle pile vide *(solution ci-après)*
- **`getSize`** : renvoi le nombre d'élément que contient la pile

#### *Initialisation*

```
stack newEmptyStack (){
  stack s = (stack)malloc(sizeof(StackList)); // creation de la pile
  s->top = NULL; // il n'y a encore aucun sommet
  s->size = 0; // la pile est vide
  return s; // on retourne la pile construite
}
```

#### *Pile vide*

```
short isEmptyStack (stack s){
  if (!s)
    return 0;
  return (s->size > 0) ? 0 : 1;
}
```

#### *Empiler*

```
short push (int value, stack s){
  if (!s)
    return 0;
  list maillon = NULL; // déclaration d'un nouveau maillon de liste
  maillon = (list)malloc(sizeof(ListNode)); // création
  maillon->content = value; // on valorise le maillon
  maillon->succ = s->top;
  s->top = maillon; // le nouveau sommet est le maillon juste créé
  s->size++; // la taille de la pile est incrémentée
  return 1;
}
```

#### *Autres opérations*

Les autres opérations sont facilement faisables ! C'est un peu tout le temps le même principe.

### Les files

#### Définition

Les files (ou « FIFO » pour « First In First Out ») sont un autre moyen de stocker des données dans un ordre précis. Cette fois ci, on peut donner l'image d'une file d'attente à la caisse d'un supermarché pour expliquer le principe de cette structure. En fait, on enfile un élément en queue de file, on défile le premier, c'est-à-dire le plus « vieux » maillon inséré en quelque sorte.

Les opérations sont analogues aux piles :

#### Opérations

Les opérations sont presque les mêmes que pour la pile. Seul l'ordre d'insertion et de suppression change. Pour vous entraîner, refaites les algos vous-même !

## La récursivité

### Définition

La récursivité est une technique de programmation permettant de répéter un bloc d'instruction sans utiliser les boucles. Le récursif est donc une variante intéressante à l'itératif. Les deux éléments essentiels pour concevoir une fonction récursive sont :

- **Un appel récursif à la fonction** dans le bloc de code
- **Une condition d'arrêt** (sinon on part en boucle infinie)

Certaines structures de données sont naturellement mieux gérables en itératif (boucles `for` ou `while`) ; d'autres au contraire, se manie bien plus facilement en récursif : c'est le cas des arbres par exemple !

### Exemples

1. Fonction factorielle

```
int facto (int n) {
  if (n == 0) // condition d'arrêt : lorsque ➔ 0! = 1
    return 1;
  else
    return n * facto(n - 1); // appel récursif à la fonction avec n-1
}
```

2. Fonction puissance ($a^n$)

```
int puiss (int a, int n) {
  if (n == 0)
    return 1;
  else
    return a * puiss(a, n - 1);
}
```

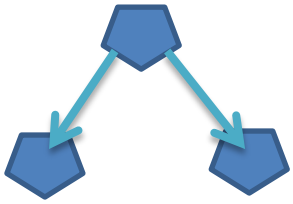
3. Ecriture de n fois une phrase

```
void ecrire (char *texte, int n) {
  if (n == 0)
    return;
  else {
    printf(« %s \n », texte);
    ecrire(texte, n - 1);
  }
}
```

## Les arbres binaires

### Définition

Un arbre est un ensemble de nœuds, organisés de façon hiérarchique, à partir d'un nœud distingué, appelé racine. Dans un arbre binaire, il ne peut y avoir que deux fils au plus pour un nœud père.

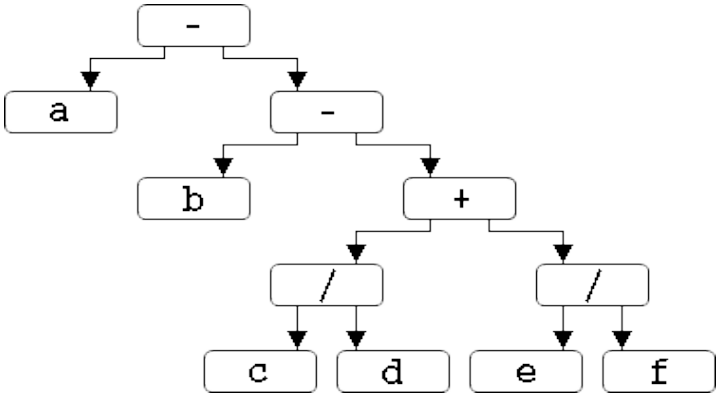


## Utilisation

- Architecture Linux
- Compilation de programme

## Propriétés

- Les arbres se gèrent naturellement de façon **récursive**
- Les **arbres binaires** ne présentent que **2 fils par nœud**
- Un arbre se représente comme ceci : **<Racine, fg, fd>**
- Un nœud possédant deux fils est appelé un **point double**
- Un nœud ne possédant qu'un fils droit est appelé **point simple à droite**
- Un nœud ne possédant qu'un fils gauche est appelé **point simple à gauche**
- Un nœud ne possédant aucun fils est appelé **« feuille »**




- Il existe 2 ordres de parcourt des arbres binaires :
  - **Parcours largeur :** « `-a-b+//cdef` »
    - Parcours itératif à l'aide d'une file
  - **Parcours profondeur :** « `-a-b+/cd/ef` »
    - Parcours récursif facile, mais aussi itératif à l'aide d'une pile
    - **3 ordres de passage :**
      - Pré-ordre (préfixe) : « `-a-b+/cd/ef` »
      - Ordre (infixe) : « `a-b-c/d+e/f` » *(opérations arithmétiques !)*
      - Post-ordre (suffixe) : « `abcd/ef/+--` »

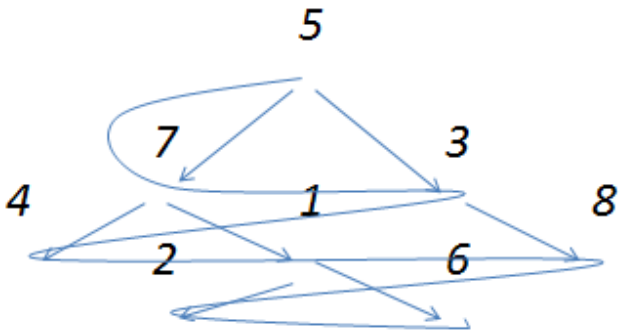
## Structure

Tout comme précédemment, un arbre binaire est une structure chaînée. A chaque nœud on associe une valeur ainsi que deux pointeurs vers les deux fils (pointeur(s) `NULL` si point simple ou feuille).

```
typedef struct treeNode {
  int content; // valeur du nœud
  struct treeNode *sag, *sad; // lien vers sous arbre gauche et sous arbre droit
} treeNode;
typedef treeNode *tree;
```

## Parcours largeur

Il n'y a pas de lien entre les fils donc ce parcours doit être itératif avec une file :

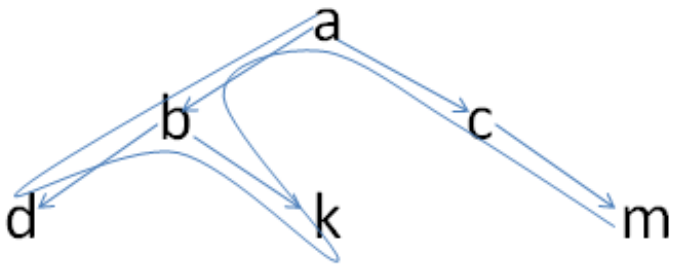



### Algorithme itératif : affichage des nœuds

```
void parcours_largeur (tree t) {
  if (!t) return; // cas de l'arbre vide dès le départ
  queue file = newEmptyQueue(); // nouvelle file
  push(t, file); // on enfile la racine de l'arbre
  while (!isEmptyQueue(file)) { // tant que la file n'est pas vide
    tree e; // nœud temporaire pour l'extraction
    pop(&e, file); // on défile dans le nœud temporaire
    printf(« %s », e->content); // on affiche la valeur du nœud temporaire
    if (t->sag) push(t->sag, file); // si sag, on enfile le sag
    if (t->sad) push(t->sad, file); // idem pour le sad
  }
}
```

## Parcours profondeur

Le parcours peut se faire itérativement mais il est plus simple de le faire récursivement.



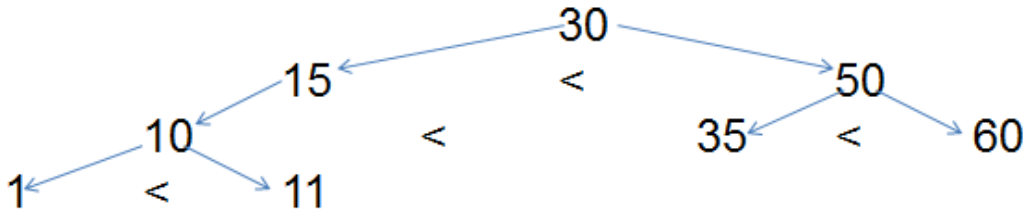

### Algorithme récursif : affichage des nœuds

```
void parcours_profondeur (tree t) {
  if (!t) return; // condition d'arrêt
  printf(« %d », t->content); // affichage en pré-ordre
  parcours_profondeur(t->sag); // appel sur fils gauche
  printf(« %d », t->content); // affichage en ordre
  parcours_profondeur(t->sad); // appel sur fils droit
  printf(« %d », t->content); // affichage en post-ordre
}
```

# Arbres Binaires de Recherche (ABR, BST, ABOH)

## Définition

Un ABR (ou aussi BST pour « Binary Search Tree » ou encore à l'EFREI ABOH pour « Arbre Binaire Ordonné Horizontalement ») est un **arbre binaire muni d'une relation d'ordre récursive entre les racines et leurs sous arbres respectifs**. En fait, à chaque niveau, tous les sous arbres gauches doivent avoir une valeur inférieure strictement à la valeur de la racine. De même pour les sous arbres droits qui doivent eux, avoir une valeur supérieure.




Ici, 1 est bien inférieur à 10 qui est inférieur à 15 et qui est encore inférieur à 30. De même pour 60 qui est supérieur à 50 supérieur à 30. 35 est inférieur à 50 mais supérieur à 30 car il se situe dans le sous arbre droit de 30.

## Relation d'ordre

Lors d'un parcours profondeur en « ordre », on obtient les valeurs rangées dans l'ordre croissant ou décroissant (croissant si sag avant sad et décroissant si sad avant sag).

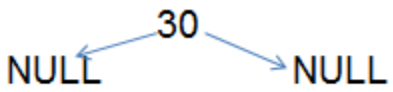
### Recherche dans un ABR

Soit l'ABR ci-dessus. L'algorithme de recherche d'un élément va être beaucoup plus optimisé que dans un simple AB grâce à la relation d'ordre.

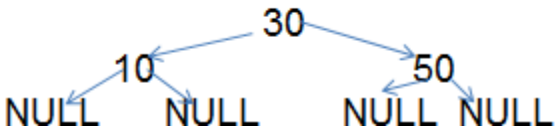
```
short recherche_ABR (tree t, int e) {
  if (!t) return 0; // condition d'arrêt : on a tout parcouru
  if (t->content == e) return 1; // l'élément est trouvé
  else if (t->content < e) return recherche_ABR(t->sad, e);
  else (t->content > e) return recherche_ABR(t->sag, e);
}
```

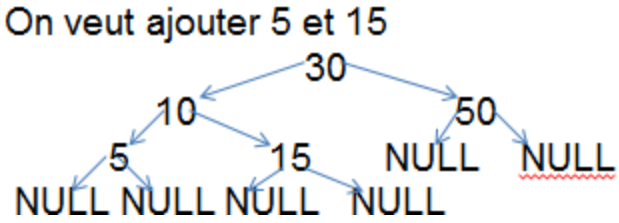
Comme cela, on a moins de sous arbres à parcourir et donc la recherche va plus vite !

### Ajout d'éléments




On veut ajouter 10 et 50

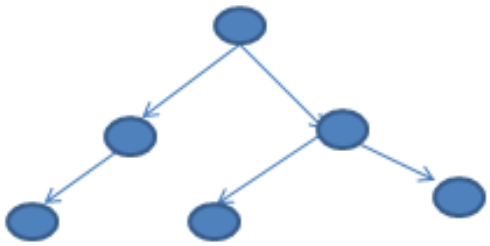




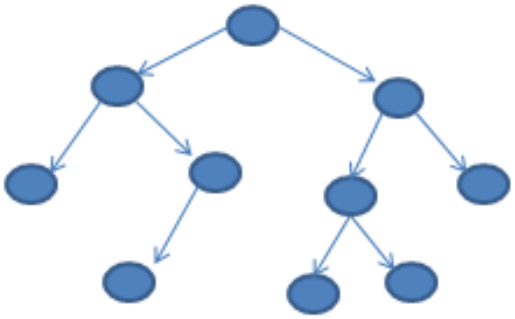

### Arbre équilibré

• **Arbre parfait** : un arbre est dit parfait si, pour tout nœud de l'arbre, la valeur absolue de la différence entre le nombre de nœuds du sad et le nombre de nœuds du sag est inférieure ou égale à 1. C'est-à-dire si $|n_g - n_d| \leq 1$

• **Arbre partiel** : un arbre est dit partiellement équilibré, si, pour tout nœud de l'arbre, la valeur absolue de la différence entre la hauteur du sad et la hauteur du sag est inférieure ou égale à 1.



*Arbre parfait et partiel*



*Arbre non parfait mais partiel*

Les arbres AVL permettent d'éviter les arbres déséquilibrés. En effet, ceux-ci inclus un champ « balance » en plus de « sag », « sad » et « content ». Ce champ va permettre de vérifier que la différence de hauteur de 1 entre le sad et le sag est bien de maximum 1.

## Les arbres AVL

### Définition

Les AVL portent le nom de leurs créateurs (Andelson-Velskii et Landis). Il s'agit d'ABR mais un peu modifiés puisqu'ils portent un nouveau champ « balance » utilisé pour vérifier le déséquilibre en chaque nœud entre le sad et le sag. Ils garantissent donc un arbre partiellement équilibré au final et ils ne contiennent pas de doublons d'élément.

## Structure

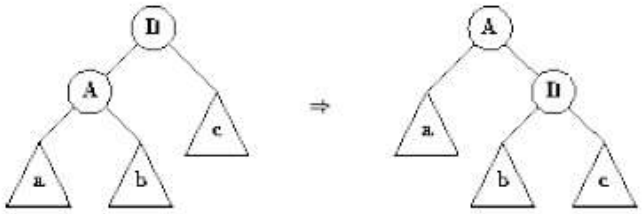
```
typedef struct treeAVL {
  int content, balance; // valeur du nœud et balance (-1, 0 ou 1 en tout nœud)
  struct treeAVL *sag, *sad; // lien vers sous arbre gauche et sous arbre droit
} treeAVL;
typedef treeAVL *avl;
```

## Réorganisation

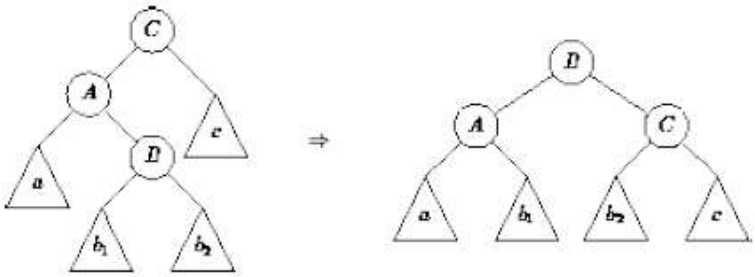
Si, en un nœud, un des sous arbres se retrouve avec une balance inférieure à -1 ou supérieure à 1 alors il y a déséquilibre. On rééquilibre l'AVL à l'aide de rotations.

- Rotation gauche (rg)
- Rotation droite (rd)
- Rotation gauche-droite (rgd)
- Rotation droite-gauche (rdg)
- Rotation droite-droite (rdd)
- Rotation gauche-gauche (rgg)

### Exemples




Rotation droite en A




Double rotation due à un déséquilibre de 2

# Les tables de hachage

## Définition

Une table de hachage est un ensemble de couples clé → valeur permettant de stocker de l'information (valeur) et de la retrouver très facilement (grâce à la clé associée).

## Opérations

Quatre opérations possibles :

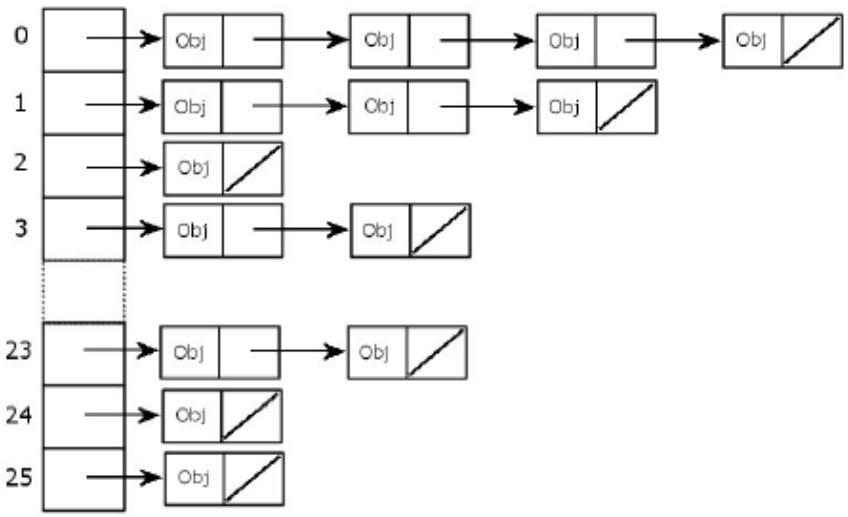
- **Rechercher** un élément
- **Ajouter** un élément
- **Supprimer** un élément
- **Modifier la valeur** d'un élément

## Représentation

Voici un petit schéma pour expliquer comment ces opérations peuvent se faire. Mais avant il faut savoir qu'une table de hachage peut-être représenter sous diverses formes comme :

- Avec des tableaux statiques
- Avec des tableaux dynamiques
- **Avec des listes chaînées**

Nous allons représenter notre table avec des listes chaînées. Pourquoi ? Car cela permet d'économiser de la place en mémoire (lorsque l'on supprimera un élément à un emplacement mémoire du tableau statique, celui-ci sera perdu !).




Le principe est simple : on a **une liste principale** avec chacun des maillons pointant vers le premier maillon d'**une liste secondaire**. La **clé correspond au maillon de la liste principale** qui nous intéresse (dans lequel on désire chercher). Puis **on recherche dans la sous liste correspondante** pour trouver la valeur rechercher, la supprimer, en ajouter une etc !

# Exercices d'application

## Les listes chaînées

**Exercice 1** : Ecrire la fonction permettant de faire la somme des éléments d'une LSC d'entiers.

**Exercice 2** : Ecrire la fonction/procédure permettant de supprimer des occurrences dans une LSC.

**Exercice 3** : Ecrire la fonction/procédure permettant de dupliquer une LSC.

## Les piles et les files

**Exercice 4** : Ecrire le code permettant de simuler la récursivité avec la fonction $a_n$.

**Exercice 5** : Ecrire la fonction du parcours profondeur en itératif.

## La récursivité

**Exercice 6** : Ecrire la fonction récursive décrivant $U_n \Leftrightarrow \begin{cases} U_{n+1} = \frac{1+U_n}{2} \\ U_0 = 0 \end{cases} \quad avec\ n \in \mathbb{N}$

**Exercice 7** : Ecrire la fonction récursive comptant le nombre d'éléments d'une LSC.

## Les arbres binaires

**Exercice 8** : Ecrire la fonction permettant de compter le nombre de feuille d'un arbre binaire.

**Exercice 9** : Ecrire la fonction permettant de mesurer la hauteur d'un arbre binaire.

## Les ABR

**Exercice 10** : Ecrire la fonction permettant de supprimer un élément dans un ABR.

# Corrections

## Les listes chaînées

**Exercice 1 : Ecrire la fonction permettant de faire la somme des éléments d'une LSC d'entiers.**

```
int somme_liste (list l) {
  int somme = 0; // mise du compteur à 0
  while (l) {
    somme += l->content; // on ajoute au compteur la valeur du maillon en cours
    l = l->succ; // on passe au maillon suivant
  }
  return somme; // on retourne la somme des maillons
}
```

**Exercice 2 : Ecrire la fonction/procédure permettant de supprimer des occurrences dans une LSC.**

```
short suppr_dans_liste (list *l, int e) { // fonction
  if (!l || !*l) // pointeur défaillant ou liste vide
    return 0;
  list currentNode = *l, tmpNode = NULL;
  while (currentNode->content == e) { // tant que la valeur est sur la tête
    tmpNode = currentNode; // on stocke le nœud temporairement
    currentNode = currentNode->succ; // on passe au suivant
    free(tmpNode); // on supprime l'adresse stockée
  }
  prevNode = currentNode; // sauvegarde du maillon en cours
  currentNode = currentNode->succ; // on incrémente le maillon en cours
  while (currentNode) { // de même mais pour la suite de la liste
    if (currentNode->content == e) {
      prevNode->succ = currentNode->succ; // lien entre précédent et successeur
      tmpNode = currentNode;
      currentNode = currentNode->succ;
      free(tmpNode);
    } else {
      prevNode = currentNode; // le précédent devient le courant
      currentNode = currentNode->succ; // on incrémente le courant
    }
  } return 1; // suppression faite (ou pas) mais tout s'est bien passé
}
```

**Exercice 3 : Ecrire la fonction/procédure permettant de dupliquer une LSC.**

```
list dupliquer_liste (list l) { // fonction
  if (!l)
    return NULL;
  list newList = NULL, curNodeNewList = NULL, curNodeList = l;
  newList = (list)malloc(sizeof(ListNode)); // cas du premier maillon
  newList->content = curNodeList->content;
  curNodeNewList = newList;

  while (curNodeList->succ) { // on parcourt tant qu'on arrive pas sur le dernier
    curNodeNewList->succ = (list)malloc(sizeof(ListNode)); // création new maillon
    curNodeNewList = curNodeNewList->succ; // on est décalé d'un, on incrémente
    curNodeList = curNodeList->succ;
    curNodeNewList->content = curNodeList->content; // on copie les valeurs
  }
  curNodeNewList->succ = NULL; // à la fin, le suivant est NULL
  return newList; // on retourne la nouvelle liste
}
```

**Exercice 4 : Ecrire le code permettant de simuler la récursivité avec la fonction $a^n$**

```
// structure nécessaire
typedef struct args {
  unsigned a;
  unsigned b;
} args;
```

```
unsigned simuler_puiss (unsigned a, unsigned n) {
  stack *pile = newEmptyStack();
  args pVal;
  unsigned res;
  do { // on insère toutes les valeurs dans la pile
    pVal = (args)malloc(sizeof(args));
    pVal->a = a;
    pVal->b = b;
    push(pVal, pile); n--; } while (n > 0);

  do { // on dépile tout et on applique la règle normale de la puissance
    pop(&pVal, pile);
    if (pVal->b == 0) res = 1; // si coef = 0 alors on termine par 1
    else res *= pVal->a; // sinon, on multiplie par le même chiffre
    free(pVal);
  } while (!isEmptyStack(pile));
  free(pile);
  return res;
}
```

**Exercice 5 : Ecrire la fonction du parcours profondeur en itératif.**

```
void profondeur_iteratif (tree t) {
  if (!t) return;
  stack pile = newEmptyStack();
  push(t, pile);
  while (!isEmptyStack(pile)) {
    tree tmp;
    pop(&tmp, pile);
    printf(« %d », tmp->content);
    if (t->sag) push(t->sag, pile);
    if (t->sad, push(t->sad, pile);
  }
  free(pile); // on supprime la pile de la mémoire
}
```

**Exercice 6 : Ecrire la fonction récursive décrivant** $U_n \Leftrightarrow \begin{cases} U_{n+1} = \frac{1+U_n}{2} \\ U_0 = 0 \end{cases}$ $avec\ n \in \mathbb{N}$

```
int Un (unsigned n) {
  return (n == 0) ? 0 : (1 + Un(n - 1)) / 2;
}
```

**Exercice 7 : Ecrire la fonction récursive comptant le nombre d'éléments d'une LSC.**

```
int compter_liste (list l) {
  if (!l) return 0;
  else return 1 + compter_liste (l->succ);
}
```

**Exercice 8 : Ecrire la fonction permettant de compter le nombre de feuille d'un arbre binaire.**

```
int compter_feuilles (tree t) {
  if (!t) return 0; // arbre vide : fin
  if (!t->sag && !t->sad) // feuille
    return 1 + compter_feuilles(t->sag) + compter_feuilles(t->sad);
  else
    return compter_feuilles(t->sag) + compter_feuilles(t->sad);
}
```

**Exercice 9 : Ecrire la fonction permettant de mesurer la hauteur d'un arbre binaire.**

```
int hauteur_AB (tree t) {
  if (!t) return 0;
  else return 1 + max(hauteur_AB(t->sag), hauteur_AB(t->sad));
}
```

```
// fonction adjointe : maximum de deux entiers
int max (int a, int b) {
  return (a > b) ? a : b;
}
```

**Exercice 10 : Ecrire la fonction permettant de supprimer un élément dans un ABR.**

```
short suppr_ABR (tree t, int e) {
  if (!t) return 0; // arbre vide
  if (e < t->content) return suppr_ABR(t->sag); // plus petit -> sag
  else if (e > t->content) return suppr_ABR(t->sad); // plus grand -> sad
  else { // l'élément a été trouvé
    tree tmp = NULL;
    if (!t->sag && !t->sad) // feuille
      free(t);
    else if (!t->sag && t->sad) { // point simple à droite
      tmp = t;
      t = t->sad;
      free(tmp);
    } else if (t->sag && !t->sag) { // point simple à gauche
      tmp = t;
      t = t->sag;
      free(tmp);
    } else { // point double
      tree minSad = min_ABR(t->sad);
      t->content = minSad->content;
      return suppr_ABR(minSad, minSad->content);
    }
  }
  return 1;
}
```

```
// fonction adjointe
tree min_ABR (tree t) { // retourne le noeud avec la valeur la plus petite
  if (!t) return NULL;
  if (t->sag) return min_ABR(t->sad);
  else return t;
}
```

*That's all Folk !*
*debatz@efrei.fr*