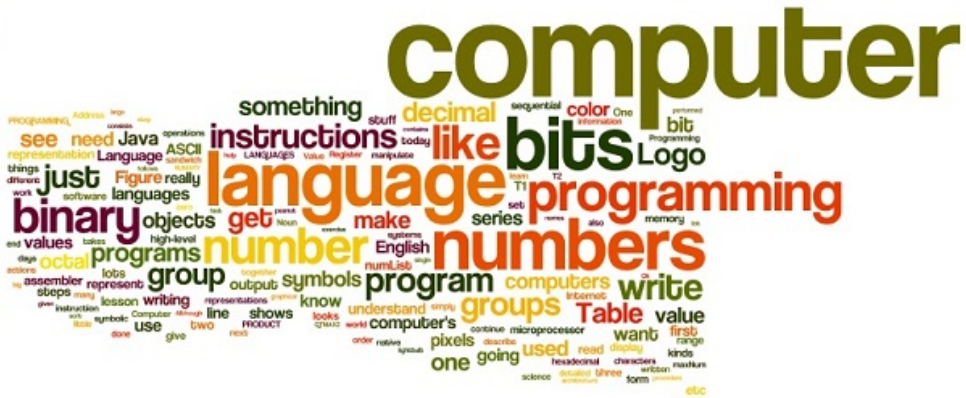
université de Hassiba ben bouali - Chlef
faculté des sciences exactes et de l'informatique
Département de l'Informatique

# Algorithmique et Structures de Données 2




Niveau L2
Année Académique 2020-2021
M-A. Tahraoui

# Table des Matières

# Chapitre 1

# Introduction

## Philosophie de ASD II

Il s'agit de sensibiliser les étudiants au fait que les algorithmes et les structures de données sont des technologies: posséder la meilleure machine du marché ne suffit pas pour obtenir les meilleurs résultats, il faut également utiliser des algorithmes performants pour le problème donné, sachant que les algorithmes performants peuvent s'appuyer sur des structures de données particulières. Dans ce cours, le fil conducteur est celui des structures de données, en allant des plus basiques (types numériques) aux plus évoluées (arbres). On aborde aussi les graphes. Les structures de données sont présentées le plus généralement possible : on commence par la norme algorithmique avant de voir les particularités (et éventuellement les limites) d'une implantation en C. On donne au fur et à mesure des exemples d'algorithmes qui s'appuient sur les structures étudiées.

Les TP se font sous Linux et sans gros environnement de développement: il s'agit de montrer aux étudiants que l'on peut programmer avec simplement un éditeur de texte (gedit en l'occurrence) et un compilateur en ligne de commande (gcc). Cela leur permet également d'écrire leurs propres « Makefiles » et donc de comprendre les fichiers que génère un environnement de développement.

## Public

Ce cours s'adresse aux étudiants inscrits en deuxième année informatique. Sur le plan pédagogique, ASD II est la suite directe de ASD I, un cours de première année dans laquelle les étudiants ont découvert les bases de l'algorithmique

et de la programmation en C. Vous le verrez lors du premier TP : beaucoup ont un peu oublié la syntaxe du C, bien qu'elle soit rappelée lors des premiers cours magistraux. En ASD II, les étudiants ont en fait appris à programmer avec un mélange de C et de C++ : ils ont programmé en procédural et non en objet (pas de classes), mais ils ont utilisé cin/cout pour les entrées-sorties, et surtout les références pour le passage de paramètres en mode donnée- résultat. En ASD II, ils vont apprendre à programmer en C « pur ». Autre nouveauté : en ASD I, les étudiants ont programmé sous Windows, avec l'environnement de développement « Dev-C++ ». En ASD II, ils vont devoir travailler sous Linux, et utiliser simplement un éditeur de texte et un terminal. L'objectif est de démystifier ce qu'est un code source, et également de leur faire comprendre la différence entre compilation et exécution.

## Référence


- Gilles Brassard et Paul Bratley, Fundamentals of Algorithms, Prentice-Hall, 1996.
- T.H. Cormen, C.E. Leiserson, R.L. Rivest et C. Stein, Introduction to algorithms (3rd edition), MIT Press, 2009.
- T.H. Cormen, C.E. Leiserson, R.L. Rivest et C. Stein, Algorithmique (3ieme édition), Dunod, 2010

# Chapitre 2

# Notion de complexité algorithmique

## Sommaire



## 2.1 Introduction

Le propos de la complexité algorithmique est de mesurer la difficulté d'un problème à l'aune de l'efficacité des algorithmes pour le résoudre. Dans ce

premier chapitre, nous allons définir formellement ce que nous entendons par problèmes et algorithmes.

## 2.2 Analyse des Algorithmes

Analyser un algorithme est devenu synonyme de prévoir les ressources nécessaires à cet algorithme. Parfois, les ressources à prévoir sont la mémoire, la largeur de bande d'une communication ou le processeur ; mais, le plus souvent, c'est le temps de calcul qui nous intéresse. En général, en analysant plusieurs algorithmes susceptibles de résoudre le problème, on arrive aisément à identifier le plus efficace. Ce type d'ana- lyse peut révéler plus d'un candidat viable, mais parvient généralement à éliminer les algorithmes inférieurs.

## 2.3 But d'un calcul de complexité

L'objectif d'un calcul de complexité algorithmique est de pouvoir comparer l'efficacité d'algorithmes résolvant le même problème. Dans une situation donnée, cela permet donc d'établir lequel des algorithmes disponibles est le plus optimal.

Si nous devons par exemple trier une liste de nombres, est-il préférable d'utiliser un tri fusion ou un tri à bulles ?
Ce type de question est primordial, car pour des données volumineuses la différence entre les durées d'exécution de deux algorithmes ayant la même finalité peut être de l'ordre de plusieurs jours.

Pour faire cela nous chercherons à estimer la quantité de ressources utilisée lors de l'exécution d'un algorithme. Les règles que nous utiliserons pour comparer et évaluer les algorithmes devront respecter certaines contraintes très naturelles. On requerra principalement qu'elles ne soient pas tributaires des qualités d'une machine ou d'un choix de technologie.
En particulier, cela signifiera que ces règles seront indépendantes des facteurs suivants:

1. du langage de programmation utilisé pour l'implémentation.
2. du processeur de l'ordinateur sur lequel sera exécuté le code.
3. de l'éventuel compilateur employé.

Nous allons donc effectuer des calculs sur l'algorithme en lui même, dans sa version "papier". Les résultats de ces calculs fourniront une estimation du temps d'exécution de l'algorithme, et de la taille mémoire occupée lors de son fonctionnement.

## 2.4 Les deux types de complexité

On distinguera deux sortes de complexité, selon que l'on s'intéresse au temps d'exécution ou à l'espace mémoire occupé.

### 2.4.1 Complexité en temps

Réaliser un calcul de complexité en temps revient à décompter le nombre d'opérations élémentaires (affectation, calcul arithmétique ou logique, comparaison,...) effectuées par l'algorithme.
Pour rendre ce calcul réalisable, on émettra l'hypothèse que toutes les opérations élémentaires sont à égalité de coût. En pratique ce n'est pas tout à fait exact mais cette approximation est cependant raisonnable.
On pourra donc estimer que le temps d'exécution de l'algorithme est proportionnel au nombre d'opérations élémentaires.

### 2.4.2 Complexité en espace

La complexité en espace est quand à elle la taille de la mémoire nécessaire pour stocker les différentes structures de données utilisées lors de l'exécution de l'algorithme.
On s'intéressera peu à cette problématique dans la suite du cours, on se contentera juste de l'évoquer de temps à autre.

## 2.5 La complexité temporelle

La complexité (temporelle) d'un algorithme est le nombre d'opérations élémentaires (affectations, comparaisons, opérations arithmétiques) effectuées par un algorithme. Ce nombre s'exprime en fonction de la taille n des données.

Par example, imaginons par exemple que l'on effectue une recherche séquentielle d'un élément dans une liste non triée. Le principe de l'algorithme est simple, on parcourt un par un les éléments jusqu'à trouver, ou pas, celui recherché. Ce parcours peut s'arrêter dès le début si le premier élément est "le bon". Mais on peut également être amené à parcourir la liste en entier si l'élément cherché est en dernière position, ou même n'y figure pas. Le nombre d'opération élémentaires effectuées dépend donc non seulement de la taille de la liste, mais également de la répartition de ses valeurs.

Cette remarque nous conduit à préciser un peu notre définition de la complexité en temps. En toute rigueur, on devra en effet distinguer trois formes de complexité en temps:

1. **la complexité dans le meilleur des cas:** c'est la situation la plus favorable, qui correspond par exemple à la recherche d'un élément situé à la première postion d'une liste, ou encore au tri d'une liste déjà triée.

2. **la complexité dans le pire des cas:** c'est la situation la plus défavorable, qui correspond par exemple à la recherche d'un élément dans une liste alors qu'il n'y figure pas, ou encore au tri par ordre croissant d'une liste triée par ordre décroissant.

3. **la complexité en moyenne:** on suppose là que les données sont réparties selon une certaine loi de probabilités.

On calculera le plus souvent la complexité dans le pire des cas, car elle est la plus pertinente. Il vaut mieux en effet toujours envisager le pire.

### 2.5.1 Exemple: Cas de la recherche d'un élément dans un tableau

Soit l'algorithme naïf suivant :
fonction appartient(x,T[],n: entier): boolean
pour (i de 0 a n) faire
si (x=T[i]) return (true); fsi;
finpour
return (false);
Fin

Soit T(n) le nombre d'instruction élémentaire qui dépend de la taille du tableau, voici des exemples d'instructions élémentaires:

1. écrire un caractère à l'écran;

2. lire un caractère au clavier;

3. affecter une valeur atomique (un caractère, un entier, un réel, ...) à une variable.

4. réaliser une opération arithmétique;

Dans la fonction *appartient*, nous avons les instructions élémentaires suivantes:

1. l'affectation d'une valeur à une variable entière;

2. la comparaison de deux valeurs entières (avec $<$ et $==$);

3. l'incrément d'une valeur entière;

4. le renvoi d'une valeur booléenne.

l'analyse de la complexité au pire de cas de la fonction appartient est la suivante:

| déroulement de l'algorithme | nombre d'exaction |
|---|---|
| i de 0 | 1 |
| i$\leq n$ | n+1 |
| i=i+1 | 2n |
| si (x=T[i]) | n |
| return | 1 |

on obtient $T(n) = 3 + 4n$

### 2.5.2 Mesure asymptotique

La complexité est une mesure du comportement asymptotique de l'algorithme. Que veut dire ce mot compliqué ? Il veut dire " quand l'entrée devient très grande ". "entrée" désigne ici la quantification des conditions de départ de l'algorithme. Il y a deux conséquences (qui sont en fait liées aux fondements mathématiques de la notion de complexité, qui ne seront pas abordés ici). D'une part, les temps constants ne sont pas pris en compte. On appelle "**temps constants**" les délais qui ne dépendent pas de l'entrée.

D'autre part, les "facteurs multiplicatifs constants" ne sont pas non plus pris en compte: la mesure de la complexité ne fait pas la différence entre un algorithme qui effectue (en fonction de $N$) $N$, $2 * N$ ou $157 * N$ opérations.

Pourquoi cette décision ? Considérez les deux algorithmes suivants, dépendant de $N$:

faire N fois l'opération A

faire N fois (l'opération B puis l'opération C)

Dans le premier cas, on fait N fois l'opération A, et dans le deuxième cas on fait au total N fois l'opération B, et N fois l'opération C. En admettant que ces deux algorithmes résolvent le même problème, et que toutes les opérations sont prises en compte pour la mesure de la complexité, le premier algorithme fait $N$ opérations et le deuxième $2N$. Mais est-ce que l'on peut dire lequel est le plus rapide? Pas du tout, car cela dépend des temps mis par les trois opérations: peut-être que $B$ et $C$ sont tous les deux quatre fois plus rapides que $A$, et que globalement c'est donc l'algorithme en $2N$ opérations qui est le plus rapide.

Plus généralement, les facteurs multiplicatifs n'ont pas forcément d'influence sur l'efficacité d'un algorithme, et ne sont donc pas pris en compte dans la mesure de la complexité. Cela permet aussi de répondre à notre question: si deux programmeurs ont deux ordinateurs, et que l'un est plus rapide que l'autre, il sera par exemple 3 fois plus rapide en moyenne; ce facteur constant sera négligé, et les deux programmeurs peuvent donc comparer la complexité de leurs algorithmes sans problème.

### 2.5.3 Notation "grand O"

On a vu que la complexité ne prenait en compte qu'un ordre de grandeur du nombre d'opérations (on néglige des choses). Pour représenter cette approximation on utilise une notation spécifique, **la notation $O$.**

La notation $O$ est comme un grand sac, qui permet de ranger ensemble des nombres d'opérations différents, mais qui ont le même ordre de grandeur. Par exemple, des algorithmes effectuant environ $N$ opérations, $2N + 5$ opérations ou $N/2$ opérations ont tous la même complexité: on la note $O(N)$. De même, un algorithme en $(2N^2 + 3N + 5)$ opérations aura une complexité de $O(N^2)$): on néglige les termes $3N$ et 5 qui sont de plus petits degrés que $2N^2$, donc croissent moins vite.

### 2.5.4 Exemple

soient les 6 algorithmes de la Figure **??**, quelle est la complexité de chacun de ces algorithmes ?

1. **Algo1:** nous avons trois boucles imbriquées, chacune allant jusqu'à $N$, soit $N*N*N$ exécutions de la boucle la plus interne, donc une complexité de $O(N^3)$.

2. **Algo2:** Vous avez peut-être envie de répondre $O(N + 1)$? En fait on dira toujours que la complexité est de $O(N)$, car le temps d'exécution est toujours à peu près proportionnel à $N$. Dès que la valeur de $N$ est suffisamment grande, ajouter 1 devient négligeable, donc on l'ignore, c'est une complexité de $O(N)$.

3. **Algo3:** la boucle interne de la première partie va s'exécuter $N^2$ fois, et la boucle de la deuxième partie va s'exécuter $N$ fois. On pourrait donc penser que la complexité est en $O(N2 + N)$, mais ici encore, pour une valeur suffisamment grande de $N$, $N$ devient négligeable par rapport

**Algo1**
```
total = 0
Pour compteur1 allant de 1 à N
    Pour compteur2 allant de 1 à N
        Pour compteur3 allant de 1 à N
            total = total + 1
```

**Algo 2**
```
Pour compteur1 allant de 1 à N + 1
    ne rien faire
```

**Algo 3**
```
total = 0
Pour compteur1 allant de 1 à N
    Pour compteur2 allant de 1 à N
        total = total + 1
Pour compteur3 allant de 1 à N
    total = total + 1
```

**Algo 4**
```
total = 0
Pour compteur1 allant de 1 à N
    Pour compteur2 allant de 1 à P
        total = total + 1
```

**Algo 5**
```
total = 0
Pour compteur1 allant de 1 à N * 2
    Pour compteur2 allant de 1 à N
        total = total + 1
Pour compteur3 allant de 1 à P
    total = total + 1
Pour compteur4 allant de 1 à 10
    total = total + 1
```

**Algo 6**
```
total = 0
donnees est un tableau de N valeurs
entières
Pour compteur1 allant de 1 à N
    valeur = donnee[compteur1]
    Pour compteur2 allant de 1 à valeur
        total = total + 1
```

Figure 2.1: Exercice

à $N^2$, et le temps d'exécution est à peu près proportionnel à $N^2$. La complexité de l'algorithme est donc de $O(N^2)$.

4. **Algo4:** Dans ce cas, on a au total $N * P$ itérations, ce qui donne donc une complexité de $O(N * P)$.

5. **Algo5:** le nombre total d'itérations est de $2 * N^2 + P + 10$. A partir d'une certaine valeur de $N$ ou de $P$, 10 devient négligeable, et n'est donc pas considéré pour la complexité. De même, la constante multiplicative 2 n'est pas conservée. On pourrait avoir envie de dire que pour une certaine valeur de $N$, $P$ devient négligeable devant $N^2$. On ne peut cependant faire aucune supposition sur la valeur de $P$, qui peut être bien plus grand que $N^2$ dans certains cas. On conserve donc $P$ dans la complexité obtenue: $O(N^2 + P)$.

6. **Algo6:** le nombre d'itérations dépend cette fois du contenu de chacune des cases du tableau. Le nombre de cases du tableau étant défini par une variable, on ne peut pas simplement lister une par une toutes ces valeurs dans le calcul de la complexité. Le seul moyen est ici de définir une nouvelle variable $P$, représentant la valeur maximale pouvant être stockée dans le tableau. Une fois cette variable définie, la complexité peut alors être définie comme $O(N * P)$.

### 2.5.5 Les principales classes de complexité

La théorie de la complexité des problèmes permet de classer les problèmes algorithmiques par difficulté selon diverses mesures. Certaines classes de complexité sont définies par le temps de calcul, par exemple les problèmes de la classe P sont ceux pour lequel il existe un algorithme dont la complexité en temps est bornée par un polynôme.

| **Nom** | **grand O** | **Exemple de T(N)** | **Exemple d'algorithmes** |
|---|---|---|---|
| Constant | $O(1)$ | 10 | |
| Logarithmique | $O(logn))$ | $logn$, $log(n^2)$ | Recherche dichotomique |
| Linéaire | $O(n)$ | n | Recherche séquentielle |
| Pseudo-linéaire | $O(nlogn)$ | $nlogn$ | Tri fusion |
| Polynomial | $2^{O(logn)} = poly(n))$ | $n, N * p, n^10$ | multiplication matricielle. |
| Exponentiel | $2^{O(n)}$ | $2^n, 10^n$ | fonction récursive de fibonacci |

## 2.6 Conclusion

Nous avons vu que le temps d'exécution d'un programme dépendait de nombreuses choses, principalement:

- Du type et de la fréquence du microprocesseur utilisé.
- Du langage utilisé et du compilateur choisi, ainsi que de ses réglages.
- Des données d'entrée De la complexité en temps de l'algorithme

Ces nombreux paramètres font qu'on ne peut obtenir qu'une approximation grossière du temps d'exécution, pour des valeurs d'entrée données. Le calcul de la complexité en temps nous fournit cependant un bon moyen pour comparer plusieurs algorithmes entre eux, avant même de les avoir implémentés et testés.

Pour déterminer la complexité d'un algorithme, plusieurs étapes sont nécessaires :

1. Déterminer de quelles variables le temps de calcul dépend. La complexité sera exprimée comme une fonction de ces variables.
2. Etablir une première version de la formule, en considérant toutes les boucles. La complexité d'une boucle est généralement égale à la complexité de l'algorithme exécuté à l'intérieur de la boucle, multipliée par le nombre d'itérations de la boucles.

3. Eliminer tout ce qui ne change pas la proportionnalité, ou devient négligeable pour des valeurs suffisamment grandes des variables : constantes additives, multiplicatives, etc.

# Chapitre 3

# Listes chaînées, Les Piles et Les Files

## Sommaire

## 3.1 Qu'est ce qu'une liste chaînée?

Une liste chaînée est simplement une liste d'objet de même type dans laquelle chaque élément contient :

- Des informations relatives au fonctionnement de l'application.par exemple des noms et prénoms de personnes avec adresses et numéros de téléphone pour un carnet d'adresse.
- L'adresse de l'élément suivant ou une marque de fin s'il n'y a pas de suivant. C'est ce lien via l'adresse de l'élément suivant contenue dans l'élément précédent qui fait la "chaîne" et permet de retrouver chaque élément de la liste.

Donc, chaque élément "pointe" sur l'élément suivant, c'est à dire possède le moyen d'y accéder et une liste chaînée peut se représenter ainsi:

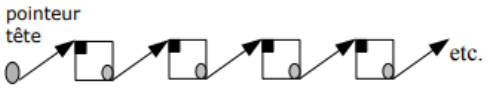



Figure 3.1: Structure d'une liste chaînée

Chaque carré correspond à un maillon de la chaîne. Le petit carré noir en haut à gauche correspond à l'adresse en mémoire de l'élément. Le petit rond en bas à droite correspond au pointeur qui "pointe" sur le suivant, c'est à dire qui contient l'adresse de l'élément suivant. Au départ il y a le pointeur de tête qui contient l'adresse du premier élément c'est à dire l'adresse de la chaîne.

## 3.2 Trois types de listes chaînées

1. **Liste simple:** La liste chaînée simple permet de circuler que dans un seul sens, c'est ce modèle:

2. **Liste symétrique ou doublement chaînée:** Avec le modèle double chaque élément possède l'adresse du suivant et du précédent ou des marques de fin s'il n'y en a pas. Il est alors possible de parcourir la chaîne dans les deux sens:

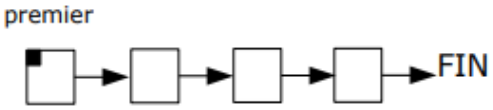



Figure 3.2: Liste simple

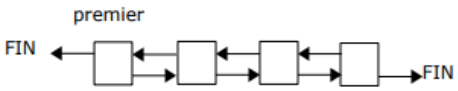



Figure 3.3: Liste doublement chaînée

3. **Liste circulaire simple:** Dans une liste circulaire simple le dernier prend l'adresse du premier et la circulation est prévue dans un seul sens:

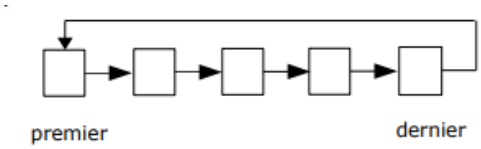



Figure 3.4: Liste circulaire simple

## 3.3 Les actions sur une liste chaînée

Les actions sont en général toujours les mêmes et toutes ne sont pas toujours nécessaires, en gros il s'agit d'écrire des fonctions pour:

1. **Liste vide:** savoir si une liste est vide ou pas.
2. **Parcourir:** passer chaque élément en revue dans l'ordre du début vers la fin.
3. **Supprimer:** enlever un élément de la liste.
4. **Insérer:** ajouter un maillon quelque part dans la chaîne.

5. **Détruire une liste:** désallouer tous les maillons de la liste.

6. **Copier une liste:** cloner de la chaîne.

7. **Sauvegarder une liste:** copier la liste sur fichier ou récupérer la liste dans le programme.

## 3.4 Implémenter une liste simple en dynamique

Chaque élément de la liste est une structure qui contient des informations pour l'application (ici deux variables juste pour faire des tests) et un pointeur sur une structure de même type :

### Structure de donnée d'un maillon

```
typedef struct Element Element;
struct Element
{
    int val;
    Element *suivant;
};
```

En plus de la structure qu'on vient de créer (que l'on dupliquera autant de fois qu'il y a d'éléments), nous allons avoir besoin d'une autre structure pour contrôler l'ensemble de la liste chaînée. Elle aura la forme suivante:

```
typedef struct Liste Liste;
struct Liste
{
    Element *premier;
};
```

Cette structure **Liste** contient un pointeur vers le premier élément de la liste. En effet, il faut conserver l'adresse du premier élément pour savoir où commence la liste. Si on connaît le premier élément, on peut retrouver tous les autres en «sautant» d'élément en élément à l'aide des pointeurs suivant.

Notre schéma est presque complet. Il manque une dernière chose : on aimerait retenir le dernier élément de la liste. En effet, il faudra bien arrêter de parcourir la liste à un moment donné. Avec quoi pourrait-on signifier à notre programme « Stop, ceci est le dernier élément » ?

Il serait possible d'ajouter dans la **structure Liste** un pointeur vers le dernier Element. Toutefois, il y a encore plus simple: il suffit de faire pointer le dernier élément de la liste vers **NULL**, c'est-à-dire de mettre son pointeur

suivant à NULL. Cela nous permet de réaliser un schéma enfin complet de notre structure de liste chaînée.

## 3.5 Les fonctions de gestion de la liste

Nous avons créé deux structures qui permettent de gérer une liste chaînée :

- Element, qui correspond à un élément de la liste et que l'on peut dupliquer autant de fois que nécessaire ;
- Liste, qui contrôle l'ensemble de la liste. Nous n'en aurons besoin qu'en un seul exemplaire.

C'est bien, mais il manque encore l'essentiel: les fonctions qui vont manipuler la liste chaînée. En effet, on ne va pas modifier « à la main » le contenu des structures à chaque fois qu'on en a besoin ! Il est plus sage et plus propre de passer par des fonctions qui automatisent le travail. Encore faut-il les créer.

A première vue, on a besoin de fonctions pour:

- Initialiser la liste ;
- Ajouter un élément ;
- Supprimer un élément ;
- Afficher le contenu de la liste ;
- Supprimer la liste entière.

On pourrait créer d'autres fonctions (par exemple pour calculer la taille de la liste) mais elles sont moins indispensables. Nous allons ici nous concentrer sur celles que je viens de vous énumérer, ce qui nous fera déjà une bonne base. Je vous inviterai ensuite à réaliser d'autres fonctions pour vous entraîner une fois que vous aurez bien compris le principe.

### 3.5.1 Initialiser la liste

La fonction d'initialisation est la toute première que l'on doit appeler. Elle crée la structure de contrôle et le premier élément de la liste.

On vous propose la fonction ci-dessous, que nous commenterons juste après, bien entendu :

```
1
2 void initialisation(Liste *l)
3 {
4     liste->premier=NULL;
5 }
```

### 3.5.2 Ajouter un élément

Ici, les choses se compliquent un peu. Où va-t-on ajouter un nouvel élément ? Au début de la liste, à la fin, au milieu ?

La réponse est qu'on a le choix. Libre à nous de décider ce que nous faisons. .

#### Ajouter un élément au début de la liste

Nous devons créer une fonction capable d'insérer un nouvel élément en début de liste. Pour nous mettre en situation, imaginons un cas semblable à la figure . suivante: la liste est composée de trois éléments et on souhaite en ajouter un nouveau au début.

Il va falloir adapter le pointeur premier de la liste ainsi que le pointeur suivant de notre nouvel élément pour « insérer » correctement celui-ci dans la liste. Je vous propose pour cela ce code source que nous analyserons juste après:

```
1  void ajouterDebut(Liste *liste, int nvNombre)
2  {
3      /* Cr ation du nouvel element */
4      Element *nouveau = malloc(sizeof(*nouveau));
5      if (liste == NULL || nouveau == NULL)
6      {
7          exit(EXIT_FAILURE);
8      }
9      nouveau->nombre = nvNombre;
10
11     /* Insertion de l'element au debut de la liste */
12     nouveau->suivant = liste->premier;
13     liste->premier = nouveau;
14 }
```

La fonction ajouterDebut() prend en paramètre l'élément de contrôle liste (qui contient l'adresse du premier élément) et le nombre à stocker dans le nouvel élément que l'on va créer.

Dans un premier temps, on alloue l'espace nécessaire au stockage du nouvel élément et on y place le nouveau nombre nvNombre. Il reste alors une étape délicate: l'insertion du nouvel élément dans la liste chaînée.

Nous avons ici choisi pour simplifier d'insérer l'élément en début de liste. Pour mettre à jour correctement les pointeurs, nous devons procéder dans cet ordre précis:

1. faire pointer notre nouvel élément vers son futur successeur, qui est l'actuel premier élément de la liste ;

2. faire pointer le pointeur premier vers notre nouvel élément.

Cela aura pour effet d'insérer correctement notre nouvel élément dans la liste chaînée (Figure. suivante)!

### Ajouter un élément à la fin de la liste

Cette fois-ci, c'est un peu plus compliqué. Il nous faut tout d'abord créer un nouvel élément, lui assigner sa valeur, et mettre l'adresse de l'élément suivant à **NULL**. En effet,, comme cet élément va terminer la liste nous devons signaler qu'il n'y a plus d'élément suivant. Ensuite, il faut faire pointer le dernier élément de liste originale sur le nouvel élément que nous venons de créer. Pour ce faire, il faut créer un pointeur temporaire sur element qui va se déplacer d'élément en élément, et regarder si cet élément est le dernier de la liste. Un élément sera forcément le dernier de la liste si NULL est assigné à son champ **suivant**.

```
1  void ajouterEnFin(Liste *liste, int valeur)
2  {
3   if(liste->premier == NULL)
4      {
5          ajouterDebut (l,valeur);
6      }
7    else {
8      element* nouvelElement = malloc(sizeof(element));
9      nouvelElement->val = valeur;
10     nouvelElement->suivant = NULL;
11
12
13         /* Sinon, on parcourt la liste a l'aide d'un pointeur
       temporaire et on
14         indique que le dernier element de la liste est relie au
       nouvel element */
15         element* temp=liste->premier;
16         while(temp->suivant != NULL)
17         {
18             temp = temp->suivant;
19         }
20         temp->suivant = nouvelElement;
21     }
22 }
```

### Insertion d'un élément en milieu de liste

Actuellement, nous ne pouvons ajouter des éléments qu'au début de la liste, ce qui est généralement suffisant. Si toutefois on veut pouvoir ajouter un

élément au milieu, il faut créer une fonction spécifique qui prend un paramètre supplémentaire : l'adresse de celui qui précédera notre nouvel élément dans la liste. Votre fonction va parcourir la liste chaînée jusqu'à tomber sur l'élément indiqué. Elle y insérera le petit nouveau juste après. (voir TD3).

### 3.5.3 Supprimer un élément

De même que pour l'insertion, nous allons ici nous concentrer sur la suppression du premier et dernier élément de la liste. Il est techniquement possible de supprimer un au milieu de la liste, ce sera d'ailleurs un des exercices de TD3

La suppression ne pose pas de difficulté supplémentaire. Il faut cependant bien adapter les pointeurs de la liste dans le bon ordre pour ne « perdre » aucune information.

#### Supprimer un élément en tête

Il s'agit là de supprimer le premier élément de la liste. Pour ce faire, il nous faudra utiliser la fonction free que vous connaissez certainement. Si la liste n'est pas vide, on stocke l'adresse du premier élément de la liste après suppression (i.e. l'adresse du 2eme élément de la liste originale), on supprime le premier élément, et on renvoie la nouvelle liste. Attention quand même à ne pas libérer le premier élément avant d'avoir stocké l'adresse du second, sans quoi il sera impossible de la récupérer.

```
List* supprimerElementEnTete(List* liste)
{
    if(liste != NULL)
    {
        /* Si la liste est non vide, on se pr pare    renvoyer l
    'adresse de
        l'element en 2eme position */
        element* aRenvoyer = liste->suivant;
        /* On lib re le premier   lment   */
        free(liste);
        /* On retourne le nouveau d but de la liste */
        return aRenvoyer;
    }
    else
    {
        return NULL;
    }
}
```

#### Supprimer un élément en fin de liste

Cette fois-ci, il va falloir parcourir la liste jusqu'à son dernier élément, indiquer que l'avant-dernier élément va devenir le dernier de la liste et libérer le dernier élément pour enfin retourner le pointeur sur le premier élément de la liste d'origine

```
Liste* supprimerElementEnFin(Liste* liste)
{
    /* Si la liste est vide, on retourne NULL */
    if(liste == NULL)
        return NULL;

    /* Si la liste contient un seul element */
    if(liste->suivant == NULL)
    {
        /* On le libere et on retourne NULL (la liste est
   maintenant vide) */
        free(liste);
        return NULL;
    }

    /* Si la liste contient au moins deux elements */
    element* tmp = liste;
    element* ptmp = liste;
    /* Tant qu'on n'est pas au dernier element */
    while(tmp->suivant != NULL)
    {
        /* ptmp stock l'adresse de tmp */
        ptmp = tmp;
        tmp = tmp->suivant;
    }
    ptmp->suivant = NULL;
    free(tmp);
    return liste;
}
```

### 3.5.4 Rechercher un élément dans une liste

Le but du jeu cette fois est de renvoyer l'adresse du premier élément trouvé ayant une certaine valeur. Si aucun élément n'est trouvé, on renverra NULL. L'intérêt est de pouvoir, une fois le premier élément trouvé, chercher la prochaine occurrence en recherchant à partir de elementTrouve->suivant. On parcourt donc la liste jusqu'au bout, et dès qu'on trouve un élément qui correspond à ce que l'on recherche, on renvoie son adresse.

```
Liste* rechercherElement(Liste* liste, int valeur)
```

```
{
    element *tmp=liste;
    /* Tant que l'on n'est pas au bout de la liste */
    while(tmp != NULL)
    {
        if(tmp->val == valeur)
        {
            return tmp;
        }
        tmp = tmp->suivant;
    }
    return NULL;
}
```

### 3.5.5 Compter le nombre d'occurrences d'une valeur

Pour ce faire, nous allons utiliser la fonction précédente permettant de rechercher un élément. On cherche une première occurrence : si on la trouve, alors on continue la recherche à partir de l'élément suivant, et ce tant qu'il reste des occurrences de la valeur recherchée. Il est aussi possible d'écrire cette fonction sans utiliser la précédente bien entendu, en parcourant l'ensemble de la liste avec un compteur que l'on incrémente à chaque fois que l'on passe sur un élément ayant la valeur recherchée. Cette fonction n'est pas beaucoup plus compliquée, mais il est intéressant d'un point de vue algorithmique de réutiliser des fonctions pour simplifier nos codes.

```
int nombreccurences(List* liste, int valeur)
{
    int i = 0;

    /* Si la liste est vide, on renvoie 0 */
    if(liste == NULL)
        return 0;

    /* Sinon, tant qu'il y a encore un element ayant la val =
    valeur */
    Liste* tmp=liste
    while((tmp = rechercherElement(liste, valeur)) != NULL)
    {

        tmp = liste->suivant;
        i++;
    }
    /* Et on retourne le nombre d'occurrences */
    return i;
}
```

## 3.6 Les piles

Nous avons découvert avec les listes chaînées un nouveau moyen plus souple que les tableaux pour stocker des données. Ces listes sont particulièrement flexibles car on peut insérer et supprimer des données à n'importe quel endroit, à n'importe quel moment.

Les piles et les files que nous allons découvrir ici sont deux variantes un peu particulières des listes chaînées. Elles permettent de contrôler la manière dont sont ajoutés les nouveaux éléments. Cette fois, on ne va plus insérer de nouveaux éléments au milieu de la liste mais seulement au début ou à la fin.

Les piles et les files sont très utiles pour des programmes qui doivent traiter des données qui arrivent au fur et à mesure. Nous allons voir en détails leur fonctionnement dans cette section.

Les piles et les files sont très similaires, mais révèlent néanmoins une subtile différence que vous allez rapidement reconnaître. Nous allons dans un premier temps découvrir les piles qui vont d'ailleurs beaucoup vous rappeler les listes chaînées, à quelques mots de vocabulaire près.

Globalement, cette section sera simple pour vous si vous avez compris le fonctionnement des listes chaînées. Si ce n'est pas le cas, retournez d'abord au chapitre précédent car nous allons en avoir besoin.

### 3.6.1 Définition

Imaginez une pile de pièces (fig. suivante). Vous pouvez ajouter des pièces une à une en haut de la pile, mais aussi en enlever depuis le haut de la pile. Il est en revanche impossible d'enlever une pièce depuis le bas de la pile.

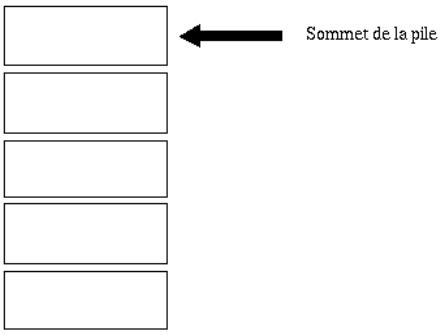



Figure 3.5: Structure d'une Pile

### 3.6.2 Fonctionnement des piles

Le principe des piles en programmation est de stocker des données au fur et à mesure les unes au-dessus des autres pour pouvoir les récupérer plus tard. Par exemple, imaginons une pile de nombres entiers de type **int**. Si j'ajoute un élément (on parle d'empilage), il sera placé au-dessus.

Le plus intéressant est sans conteste l'opération qui consiste à extraire les nombres de la pile. On parle de dépilage. On récupère les données une à une, en commençant par la dernière qui vient d'être posée tout en haut de la pile. On enlève les données au fur et à mesure, jusqu'à la dernière tout en bas de la pile.

On dit que c'est un algorithme LIFO, ce qui signifie « Last In First Out ». Traduction : « Le dernier élément qui a été ajouté est le premier à sortir ».

Les éléments de la pile sont reliés entre eux à la manière d'une liste chaînée. Ils possèdent un pointeur vers l'élément suivant et ne sont donc pas forcément placés côte à côte en mémoire. Le dernier élément (tout en bas de la pile) doit pointer vers NULL pour indiquer qu'on a... touché le fond.

### 3.6.3 Création d'un système de pile

Maintenant que nous connaissons le principe de fonctionnement des piles, essayons d'en construire une. Comme pour les listes chaînées, il n'existe pas de système de pile intégré au langage C. Il faut donc le créer nous-mêmes. Chaque élément de la pile aura une structure identique à celle d'une liste chaînée:

```
typedef struct Element Element;
struct Element
{
    int nombre;
    Element *suivant;
};
```

La structure de contrôle contiendra l'adresse du premier élément de la pile, celui qui se trouve tout en haut :

```
typedef struct Pile Pile;
struct Pile
{
    Element *premier;
};
```

Nous aurons besoin en tout et pour tout des fonctions suivantes :

- Empilage d'un élément.

- Dépilage d'un élément.

Vous noterez que, contrairement aux listes chaînées, on ne parle pas d'ajout ni de suppression. On parle d'empilage et de dépilage car ces opérations sont limitées à un élément précis, comme on l'a vu. Ainsi, on ne peut ajouter et retirer un élément qu'en haut de la pile.

### Empiler un élément

Notre fonction empiler doit prendre en paramètre la structure de contrôle de la pile (de type Pile) ainsi que le nouveau nombre à stocker. On rappelle que nous stockons ici des **int**, mais rien ne vous empêche d'adapter ces exemples avec un autre type de données. On peut stocker n'importe quoi : des **double**, des **char**, des **chaînes**, des **tableaux** ou même d'autres **structures**!

```
1  void empiler(Pile *pile, int nvNombre)
2  {
3      Element *nouveau = malloc(sizeof(*nouveau));
4      if (pile == NULL || nouveau == NULL)
5      {
6          exit(EXIT_FAILURE);
7      }
8
9      nouveau->nombre = nvNombre;
10     nouveau->suivant = pile->premier;
11     pile->premier = nouveau;
12 }
```

L'ajout se fait en début de pile car, comme on l'a vu, il est impossible de le faire au milieu d'une pile. C'est le principe même de son fonctionnement, on ajoute toujours par le haut. De ce fait, contrairement aux listes chaînées, on ne doit pas créer de fonction pour insérer un élément au milieu de la pile. Seule la fonction empiler permet d'ajouter un élément

### Dépiler un élément

Le rôle de la fonction de dépilage est de supprimer l'élément tout en haut de la pile, ça, vous vous en doutiez. Mais elle doit aussi retourner l'élément qu'elle dépile, c'est-à-dire dans notre cas le nombre qui était stocké en haut de la pile.

C'est comme cela que l'on accède aux éléments d'une pile : en les enlevant un à un. On ne parcourt pas la pile pour aller y chercher le second ou le troisième élément. On demande toujours à récupérer le premier.

La fonction *depiler* va donc retourner un int correspondant au nombre qui se trouvait en tête de pile:

```
int depiler(Pile *pile)
{
    if (pile == NULL)
    {
        exit(EXIT_FAILURE);
    }

    int nombreDepile = 0;
    Element *elementDepile = pile->premier;

    if (pile != NULL && pile->premier != NULL)
    {
        nombreDepile = elementDepile->nombre;
        pile->premier = elementDepile->suivant;
        free(elementDepile);
    }

    return nombreDepile;
}
```

On récupère le nombre en tête de pile pour le renvoyer à la fin de la fonction. On modifie l'adresse du premier élément de la pile, puisque celui-ci change. Enfin, bien entendu, on supprime l'ancienne tête de pile grâce à *free*.

## 3.7 Les Files

Les files ressemblent assez aux piles, si ce n'est qu'elles fonctionnent dans le sens inverse!

### 3.7.1 Fonctionnement des files

Dans ce système, les éléments s'entassent les uns à la suite des autres. Le premier qu'on fait sortir de la file est le premier à être arrivé. On parle ici d'algorithme FIFO (First In First Out),c'est-à-dire « Le premier qui arrive est le premier à sortir ».

En programmation, les files sont utiles pour mettre en attente des informations dans l'ordre dans lequel elles sont arrivées. Par exemple, dans un logiciel de chat (type messagerie instantanée), si vous recevez trois messages à peu de temps d'intervalle, vous les enfilez les uns à la suite des autres en mémoire. Vous vous occupez alors du premier message arrivé pour l'afficher à l'écran, puis vous passez au second, et ainsi de suite.

### 3.7.2 Création d'un système de file

Le système de file va ressembler à peu de choses près aux piles. Il y a seulement quelques petites subtilités étant donné que les éléments sortent de la file dans un autre sens, mais rien d'insurmontable si vous avez compris les piles.

Nous allons créer une structure Element et une structure de contrôle File:

```
typedef struct Element Element;
struct Element
{
    int nombre;
    Element *suivant;
};

typedef struct File File;
struct File
{
    Element *premier;
    Element *dernier
};
```

#### Enfiler Un élément

La fonction qui ajoute un élément à la file est appelée fonction « d'enfilage ». Il y a deux cas à gérer :

soit la file est vide, dans ce cas on doit juste créer la file en faisant pointer premier et dernier élément vers le nouvel élément créé;

soit la file n'est pas vide, dans ce cas, On rajoutera notre nouvel élément après le dernier.

Voici comment on peut faire dans la pratique :

```
void enfiler(File *file, int nvNombre)
{
    Element *nouveau = malloc(sizeof(*nouveau));
    if (file == NULL || nouveau == NULL)
    {
        exit(EXIT_FAILURE);
    }

    nouveau->nombre = nvNombre;
    nouveau->suivant = NULL;

    if (file->premier != NULL)
    {
        file->dernier->suivant = nouveau;
         file->dernier = nouveau
    }
```

```
    else
    {
        file->premier = nouveau;
        file->dernier = nouveau
    }
}
```

### Défiler un élément

Le défilage ressemble étrangement au dépilage. Étant donné qu'on possède un pointeur vers le premier élément de la file, il nous suffit de l'enlever et de renvoyer sa valeur.

```
int defiler(File *file)
{
    if (file == NULL)
    {
        exit(EXIT_FAILURE);
    }

    int nombreDefile = 0;
    if (file->premier != NULL)
    {
        Element *elementDefile = file->premier;

        nombreDefile = elementDefile->nombre;
        file->premier = elementDefile->suivant;
        free(elementDefile);
       if(file->premier==NULL)    file->dernier=NULL;

    }

    return nombreDefile;
}
```

## 3.8 Exercice

1. Ecrire une fonction qui renvoie le nombre d'éléments d'une liste chaînée.

2. Ecrire une fonction qui renvoie le nombre d'éléments d'une liste chaînée ayant une valeur donnée(champ Info).

3. Ecrire une fonction qui vérifie si une liste chaînée est triée par valeurs croissantes du champ Info.

4. Ecrire une fonction Conceptualistes qui concatène deux listes. La fonction reçoit deux listes et transforme la première en la concaténation des deux listes. Cette fonction ne doit pas créer de nouvelles cellules: elle modifie directement les listes passées en paramètres.

5. Ecrire une fonction qui recopie une liste chaînée en conservant l'ordre des éléments. En utilisant les Piles, Ecrire une fonction qui inverse l'ordre du chaînage des cellules d'une liste.

6. Ecrire un algorithme pour déplacer les entiers de P1 dans une pile P2 de façon à avoir dans P2 tous les nombres pairs en dessous des nombres impairs.

7. Ecrire un algorithme pour copier dans P2 les nombres pairs contenus dans P1 . Le contenu de P1 aprés exécution de l'algorithme doit etre identique à celui avant execution. Les nombres pairs dans P2 doivent etre dans l'ordre ou ils apparaissent dans P1 .

# Chapitre 4

# Algorithme de tri

## Sommaire



## 4.1 Introduction

On désigne par "tri" l'opération consistant à ordonner un ensemble d'éléments selon une relation d'ordre prédéfinie.

Les algorithmes de tri ont une grande importance pratique. Ils sont fondamentaux dans certains domaines, comme l'informatique de gestion où l'on tri de manière quasi-systématique des données avant de les utiliser.

L'étude du tri est également intéressante en elle-même car il s'agit sans doute du domaine de l'algorithmique qui a été le plus étudié et qui a conduit à des résultats remarquables sur la construction d'algorithmes et l'étude de leur complexité. Dans le reste de ce chapitre, on suppose que les éléments à trier sont des entiers, mais les algorithmes présentés sont valables pour n'importe quel type d'éléments, pourvu qu'il soit muni d'un ordre total. On suppose aussi qu'on trie des tableaux, dans l'ordre croissant.

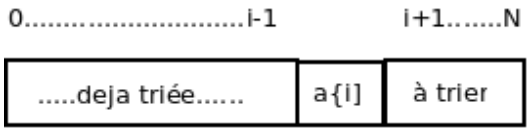
## 4.2 Tri par insertion

Le tri par insertion est sans doute le plus naturel. Il consiste à insérer successivement chaque élément dans l'ensemble des éléments déjà triés. C'est souvent ce que l'on fait quand on trie un jeu de cartes ou un paquet de copies. Le tri par insertion d'un tableau **a** s'effectue en place, c'est-à-dire qu'il ne demande pas d'autre tableau que celui que l'on trie. Son coût en mémoire est donc constant si on ne compte pas la place occupée par les données. Il consiste à insérer successivement chaque élément a[i] dans la portion du tableau a[0:i] déjà triée. Illustrons cette idée sur un tableau de cinq entiers contenant initialement [5,2,3,1,4]. Au départ, a[0:0] = 5 est déjà trié.

1. On insère 2 dans a[0:1]. On obtient [2 5 3 1 4].
2. On insère 3 dans a[0:2]. On obtient [2 3 5 1 4].
3. On insère 1 dans a[0:3]. On obtient [1 2 3 5 4].
4. On insère 4 dans a[0:4]. On obtient [1 2 3 4 5].

### 4.2.1 Réalisation

De manière générale, chaque étape du tri par insertion correspond à la situation suivante :




On commence par une boucle for pour parcourir le tableau :

```
1 for (i=1;i<N;i++)
```

Pour insérer l'élément a[i] à la bonne place, on utilise alors une boucle while qui décale vers la droite les éléments tant qu'ils sont supérieurs à a[i] :

```
temp = t[i];
pos= i;
 while((pos > 0) && (t[pos-1] > temp))
  { t[pos]= t[pos-1];
    pos= pos -1;
}
```

Une fois sorti de la boucle, il reste à positionner a[i] à sa place :

```
t[pos] = temp;
```

Le code complet est donné programme ci-dessous.

```
void triInsertion(int t[], int N)
{ int i, pos, temp;
for(i=1;i<N;i++)
{

temp = t[i];
pos= i;
 while((pos > 0) && (t[pos-1] > temp))
  {
    t[pos]= t[pos-1];
    pos= pos -1;
  }

}
}
```

### 4.2.2 Complexité

On note que la fonction tri insertion effectue exactement le même nombre de comparaisons et d'affectations. Lorsque la boucle while insère l'élément a[i] à la position $i-k$, elle effectue $k+1$ comparaisons. Au mieux, $k$ vaut 0 et au pire, $k$ vaut $i$, avec $i$ qui varie de 1 à $N-1$, ce qui donne au final le tableau suivant:

| **Opération** | **Meilleur cas** | **Moyenne** | **Pire cas** |
|---|---|---|---|
| comparaisons | $N$ | $N^2/4$ | $N^2/4$ |
| affectations | $N$ | $N^2/4$ | $N^2/4$ |

Table 4.1: Complexité de l'algorithime tri par insertion.

## 4.3 Tri par sélection

Le principe du tri par sélection/échange (ou tri par extraction) est d'aller chercher le plus petit élément du vecteur pour le mettre en premier, puis de

repartir du second élément et d'aller chercher le plus petit élément du vecteur pour le mettre en second, etc..

### 4.3.1 Réalisation

Pour un tableau de $n$ éléments et à partir de l'indice 0: A chaque étape $i$:

1. On recherche parmi les $t[i], ..t[n]$ le plus petit élément qui doit être positionné à la place $i$.
2. Supposons cet élément à $ind_{min}$ $t[ind_{min}]$ et $t[i]$ sont échangés.
3. i = i+1, retourner en 1.

De cette façon, au cours du tri les éléments de 0 à i-1 sont tous ordonnés et bien placés. Le code complet est donné programme ci-dessous.

```
void triSelection(int t[ ], int max)
   { int ind_min, i, j;
   int temp;
   i= 0;
   while(i < max)
   { ind_min= i;
    for(j = i+1; j < max; j++)
    if(t[j] < t[ind_min])
    ind_min= j;
    temp= t[i];
    t[i]= t[ind_min];
    t[ind_min]= temp;
    i= i+ 1;
 }
 }
```

### 4.3.2 Complexité (Exercice)

- Dérouler à la main l'algorithme de tri par sélection sur le tableau [15,4,2,9,55,16,0,1].
- Quelle est la complexité de cet algorithme ?, Expliquer.

## 4.4 Tri Bulle

Le principe du tri à bulles (bubble sort ou sinking sort) est de comparer deux à deux les éléments a[i] et a[i+1] consécutifs d'un tableau et d'effecteur une permutation si a[i] > a[i+1]. On continue de trier jusqu'à ce qu'il n'y ait plus de permutation.

### 4.4.1 Réalisation

Pour un tableau de n éléments„au début max=n. A la fin de chaque étape la partie du tableau de t[max+1,.. n] est constituée d'éléments bien placés et tous supérieurs aux éléments de t[0,.. max-1].

- Pour les éléments i, de 0 à max-1 comparer et permuter successivement (si nécessaire) les éléments t[i] et t[i+1].
- Faire max = max -1, (fin de l'étape) recommencer en 1 Arrêt de l'algorithme lorsque max = 0.

il est claire qu'on peut d'arrêter l'algorithme si à une étape il n'y a pas eu de changement. Le code complet est donné programme ci-dessous.

```
1
2  void triBulle(int t[ ], int max)
3  { int i, temp;
4  while(max > 0)
5  { i = 0;
6  while(i < max -1)
7  { if( t[i] > t[i+1])
8  { temp = t[i];
9  t[i] = t[i+1];
10 t[i+1] = temp;
11 }
12 i = i+1;
13 }
14 max= max -1;
15 }
16 }
```

### 4.4.2 Complexité (Exercice)

1. Dérouler à la main l'algorithme de tri à bulles sur le tableau [15,4,2,9,55,16,0,1].
2. Quelle est la complexité de cet algorithme ?, Expliquer.

## 4.5 Tri Fusion

Le tri fusion applique le principe diviser pour régner. Il partage les éléments à trier en deux parties de même taille, sans chercher à comparer leurs éléments. Une fois les deux parties triées récursivement, il les fusionne, d'où le nom de tri fusion. Ainsi on évite le pire cas du tri rapide où les deux parties sont de tailles disproportionnées.

Illustrons cette idée sur un tableau de 8 entiers contenant initialement [7 6 3 5 4 2 1 8].

1. Pour trier a[0:7], on trie a[0:3] et a[4:7]: |7|6|3|5|4|2|1|8|.
2. Pour trier a[0:4], on trie a[0:1] et a[2:3]. |7|6|3|5|x|x|x|x|.
3. Pour trier a[0:1], on trie a[0:0] et a[1:1]. |7|6|x|x|x|x|x|x|.
4. On fusionne a[0:1] et a[1:2]. |**6**|**7**|x|x|x|x|x|x|.
5. Pour trier a[2:3], on trie a[2:2] et a[3:3]. |x|x|3|5|x|x|x|x|.
6. On fusionne a[2:3] et a[3:4]. |x|x|**3**|**5**|x|x|x|x|.
7. On fusionne a[0:1] et a[2:3]. |**3**|**5**|**6**|**7**|x|x|x|x|
8. Pour trier a[4:7], on trie a[4:5] et a[6:7]. |x|x|x|x|4|2|1|8|
9. Pour trier a[4:5], on trie a[4:4] et a[5:5]. |x|x|x|x|4|2|x|x|
10. On fusionne a[4:4] et a[5:5]. |x|x|x|x|**2**|**4**|x|x|
11. Pour trier a[6:7], on trie a[6:6] et a[7:7]. |x|x|x|x|x|x|1|8|
12. On fusionne a[6:6] et a[7:7]. |x|x|x|x|x|x|**1**|**8**|
13. On fusionne a[4:5] et a[6:7]. |x|x|x|x|**1**|**2**|**4**|**8**|
14. On fusionne a[0:3] et a[4:7]. |**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|.

### 4.5.1 Réalisation

On va chercher à réaliser le tri fusion d'un tableau en place, en délimitant la portion à trier par deux indices gauche et droite. Pour le partage, il suffit de calculer l'indice médian $m = (g + d)/2$ On trie alors récursivement les deux parties délimitées par gauche et m d'une part, m et d d'autre part. Il reste à effectuer la fusion . Il s'avère extrêmement difficile de la réaliser en place. Le plus simple est d'utiliser un second tableau, alloué une et une seule fois au début du tri.

Le pseudo-code complet est donné programme ci-dessous.

```
1
2
3 PROCEDURE tri_fusion ( TABLEAU a[1:n])
4 FAIRE
5     SI TABLEAU EST VIDE RENVOYER TABLEAU
```

```
6     gauche = partie_gauche de TABLEAU (a1)
7     droite = partie_droite de TABLEAU (a2)
8     gauche = tri_fusion gauche (a1)
9     droite = tri_fusion droite (a2)
10    renvoyer fusion gauche droite
11    POUR i VARIANT DE 1 A n - 1 - passage FAIRE
12        SI a[i] > a[i+1] ALORS
13            echanger a[i] ET a[i+1]
14            permut      VRAI
15        FIN SI
16    FIN POUR
17    passage      passage + 1
18 FIN PROCEDURE
```

**Exercice:** Ecrire Un programme en langage C qui permet d'effectuer le tri par fusion.

### 4.5.2 Complexité

Si on note C(N) (resp. f(N)) le nombre total de comparaisons effectuées par triFusion pour trier un tableau de longueur N, on a l'équation de récurrence suivante:

$$C(N) = 2C(N/2) + f(N)$$

En effet, les deux appels récursifs se font sur deux segments de même longueur $N/2$. Dans le meilleur des cas, la fonction fusion n'examine que les éléments de l'un des deux segments car ils sont tous plus petits que ceux de l'autre segment. Dans ce cas, $f(N) = N/2$ et donc $C(N) \sim \frac{1}{2}NlogN$. Dans le pire des cas, tous les éléments sont examinés par fusion et donc $f(N) = N-1$ , d'où $C(N) \sim NlogN$.

Le nombre d'affectations est le même dans tous les cas : $N$ affectations dans la fonction fusion (chaque élément est copié de $a1$ vers $a2$). Si on note $A(N)$ le nombre total d'affectations pour trier un tableau de longueur $N$, on a donc :

$$A(N) = 2A(N/2) + N$$

d'où un total de $2NlogN$ affectations.

| **Opération** | **Meilleur cas** | **Moyenne** | **Pire cas** |
|---|---|---|---|
| comparaisons | $1/2NlogN$ | $NlogN$ | $NlogN$ |
| affectations | $1/2NlogN$ | $2NlogN$ | $2NlogN$ |

Table 4.2: Complexité de l'algorithime tri par fusion.

On note que, dans tous les cas, la complexité du tri fusion est la même. Cette complexité est optimale.

# Chapitre 5

# Les Arbres

## Sommaire

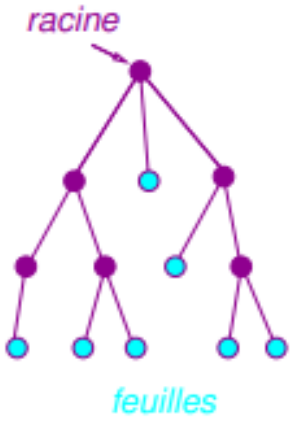


## 5.1 Présentation

Les arbre sont très utilisés en informatique, d'une part parce que les informations sont souvent hiérarchisées, et peuvent être représentées naturellement sous une forme arborescente, et d'autre part, parce que les structures de données arborescentes permettent de stocker des données volumineuses de façon que leur accès soit efficace.

## 5.2 Définitions et terminologie

Un arbre est un ensemble organisé de nœuds dans lequel chaque nœud a un père et un seul, sauf un nœud que l'on appelle la racine. Si le nœud $p$ est le




père du nœud $f$, nous dirons que $f$ est un fils de $p$, et si le nœud $p$ n'a pas de fils nous dirons que c'est une feuille. Chaque nœud porte une étiquette ou valeur ou clé. On a l'habitude, lorsqu'on dessine un arbre, de le représenter avec la tête en bas, c'est-à-dire que la racine est tout en haut, et les nœuds fils sont représentés en-dessous du nœud père.

Un nœud est défini par son étiquette et ses sous-arbres. On peut donc représenter un arbre par un n-uplet $[e, a_1, ..., a_k]$ dans lequel $e$ est l'étiquette portée par le nœud, et $[a_1, ..., a_k]$ sont ses sous-arbres.

On distingue les arbres binaires des arbres généraux. Leur particularité est que les fils sont singularisés : chaque nœud a un fils gauche et un fils droit. L'un comme l'autre peut être un arbre vide. L'écriture d'un arbre s'en trouve modifiée, puisqu'un nœud a toujours deux fils.

On utilise pour les arbres une terminologie inspirée des liens de parenté:

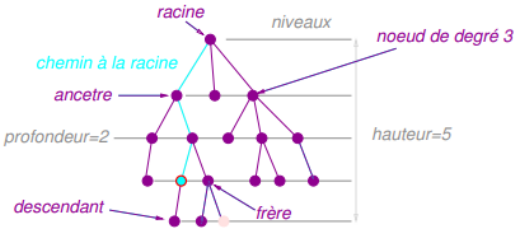
1. Les descendants d'un nœud $p$ sont les nœuds qui apparaissent dans ses sous-arbres.

2. Un ancêtre d'un nœud $p$ est soit son père, soit un ancêtre de son père,

3. Le chemin qui relie un nœud à la racine est constitué de tous ses ancêtre (c'est-à-dire de son père et des nœuds du chemin qui relie son père à la racine).

4. Un frère d'un nœud $p$ est un fils du père de $p$, et qui n'est pas $p$.

Les nœuds d'un arbre se répartissent par niveaux: le premier niveau (par convention ce sera le niveau 0) contient la racine seulement, le deuxième niveau




contient les deux fils de la racine,..., les nœuds du niveau $k$ sont les fils des nœuds du niveau $k-1$,....
La hauteur d'un arbre est le nombre de niveaux de ses nœuds. C'est donc aussi le nombre de nœuds qui jalonnent la branche la plus longue. Attention, la définition de la hauteur varie en fonction des auteurs. Pour certains la hauteur d'un arbre contenant un seul nœud est 0.

## 5.3 Arbre binaire

Après avoir défini de façon théorique ce qu'est un arbre en général, nous allons nous intéresser plus particulièrement aux arbres binaires, qui sont une forme simple d'arbre possédant de nombreuses applications en informatique.

**Arbre binaire :** arbre dans lequel un nœud peut avoir 0, 1 ou 2 fils.

Un arbre binaire peut être décomposé en trois parties :

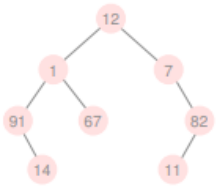
1. Sa racine.
2. Son sous-arbre gauche.
3. Son sous-arbre droit.

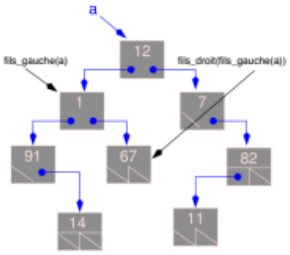
De même, chacun des sous-arbres peut être lui aussi décomposé de la même manière. Le type abstrait arbre binaire est basé sur cette décomposition récursive.

### 5.3.1 Type abstrait

Comme pour les piles et les files de données, le type abstrait proposé ici se compose d'un ensemble d'opérations et d'un ensemble de contraintes qui leur sont appliquées.

**Opérations**




- **creeArbre:** créer un arbre vide
- **estVide:** déterminer si l'arbre est vide
- **racine:** renvoyer la valeur située à la racine de l'arbre
- **filsGauche:** renvoyer le sous-arbre gauche d'un nœud
- **filsDroit:** renvoyer le sous-arbre droit d'un nœud
- **enracine:** créer un arbre à partir d'une valeur et de deux sous-arbres

Les opérations permettant de créer un arbre et de tester s'il est vide sont similaire à celles déjà définies pour les piles et les files. Les opérations racine, **filsGauche** et **filsDroit** sont des opérations d'accès, elles permettent de récupérer les trois constituants de l'arbre et d'y naviguer. a noter que racine renvoie la valeur associée à la racine de l'arbre (i.e. son label ou sa clé), alors que **filsGauche** et **filsDroit** renvoient ses deux sous-arbres, qui sont eux-mêmes du type arbre. On voit déjà qu'on aura une structure de données récursive, comme c'était déjà le cas avec les listes chaînées.

L'opération enracine permet de construire un arbre: elle crée un nouveau nœud en associant une racine et deux sous-arbres. Elle renvoie le nouvel arbre obtenu.

### 5.3.2 Implémentation par pointeur

Les listes, que l'on avait utilisées pour implémenter les types abstraits piles et files, ne sont pas adaptés type abstrait arbre binaire. Nous allons définir une structure de données spécifiquement dans ce but. Dans notre structure de données, chaque nœud doit contenir les informations suivantes :

- Une valeur représentant son étiquette.
- Un pointeur vers son fils gauche.

- Un pointeur vers son fils droit.

Supposons, sans perte de généralité, que l'on veut manipuler un arbre dont les nœuds contiennent des entiers int. Alors, on utilisera la structure suivante :

```
typedef struct s_noeud
{ int valeur ;
struct s_noeud *gauche;
struct s_noeud *droit;
} noeud;
typedef noeud* arbre;
```

Le premier type est une structure récursive appelée nœud. Elle contient bien la valeur entière et deux pointeurs vers les sous-arbres gauche et droite. À noter l'utilisation d'un nom *snoeud* pour la structure elle-même, comme on l'avait fait pour définir les éléments d'une liste chaînée dans le chapitre précédent. Le second type est seulement un nouveau nom arbre associé à un pointeur sur un nœud. Concrètement, cela signifie que les types struct *s-noeud**, *noeud** et arbre sont synonymes. Un arbre vide sera représenté par un pointeur **NULL**.

*Création Pour la création, on déclare une variable de type arbre et on l'initialise à NULL:

```
arbre cree_arbre()
{ return NULL;
}
```

Comme on l'a vu, l'arbre sera vide s'il vaut NULL:

```
int est_vide(arbre a)
{ return (a == NULL);
}
```

*Accès L'accès à la racine est direct: si l'arbre n'est pas vide, on peut directement renvoyer sa valeur. Sinon, l'opération n'est pas définie et on renvoie un code d'erreur. Soit **int racine(arbre a, int *r)** la fonction qui permet d'accéder à la racine de l'arbre a. La fonction transmet la valeur par adresse grâce au paramètre r. La fonction renvoie -1 en cas d'erreur (si l'arbre est vide) ou 0 sinon.

```
int racine(arbre a, int *r)
{ int erreur = 0;
if(est_vide(a))
erreur = -1;
else
*r = a->valeur;
return erreur;
}
```

L'accès aux fils est symétrique: soit arbre **filsgauche(arbre a, arbre *f)** la fonction qui permet d'accéder au fils gauche de a. La fonction transmet le nœud par adresse grâce à la variable f. La fonction renvoie -1 en cas d'erreur (si l'arbre est vide) ou 0 sinon.

```
int fils_gauche(arbre a, arbre *f)
{ int erreur = 0;
if(est_vide(a))
erreur = -1;
else
*f = a->gauche;
return erreur;
}
```

La fonction arbre **filsdroit(arbre a, arbre *f)** est obtenue simplement en remplaçant le mot gauche (indiqué en gras) par droit dans le code source ci-dessus.

### Insertion

L'enracinement consiste à créer un nouvel arbre à partir de deux sous-arbres et d'un nœud racine. Soit arbre **enracine(int v, arbre a1, arbre a2)** la fonction qui crée et renvoie un arbre dont la racine est n et dont les sous-arbres gauche et droit sont respectivement a1 et a2.

```
arbre enracine(int v, arbre a1, arbre a2)
{ noeud *n;
if((n = (noeud *) malloc(sizeof(noeud))) != NULL)
{ n->valeur = v;
n->gauche = a1;
n->droit = a2;
}
return n;
}
```

## 5.3.3 Parcours d'un arbre binaire

Le principe est simple, pour parcourir l'arbre a, on parcourt récursivement son sous-arbre gauche, puis son sous arbre droit. Ainsi le sous-arbre gauche sera exploré dans sa totalité avant de commencer l'exploration du sous-arbre droit.

```
void Parcours(ARBRE a)
{
if (a != NULL)
{
/* traitement avant */
```

```
6  Parcours (a->fg);
7  /* traitement entre */
8  Parcours (a->fd);
9  /* traitement apres */
10 }
11 }
```

Dans ce schéma l'exploration descend d'abord visiter les nœuds les plus à gauche: on parle de parcours en profondeur d'abord. On distingue le parcours **préfixe**, le parcours **infixe** et le parcours **postfixe**. La différence entre ces parcours tient uniquement à l'ordre dans lequel sont traités les nœuds du sous-arbre gauche, le nœud courant, et les nœuds du sous-arbre droit.

## Parcours préfixe:

La valeur du nœud courant est traitée avant les valeurs figurant dans ses sous-arbres. Si le traitement consiste simplement à afficher la valeur du nœud, le parcours du l'arbre ci-dessous produirait l'affichage 12 1 91 67 7 82 61.

```
1  void ParcoursPrefixe(ARBRE a)
2  {
3  if (a != NULL)
4  {
5  TraiterLaValeur (a->val);
6  ParcoursPrefixe (a->fg);
7  ParcoursPrefixe (a->fd);
8  }
9  }
```

## Parcours Infixe:

la valeur du nœud courant est traitée après les valeurs figurant dans son sous-arbre gauche et avant les valeurs figurant dans son sous-arbre droit.

```
1
2  void ParcoursInfixe(ARBRE a)
3  {
4  if (a != NULL)
5  {
6  ParcoursInfixe (a->fg);
7  TraiterLaValeur (a->val);
8  ParcoursInfixe (a->fd);
9  }
10 }
```

### Parcours postfixe:

la valeur du nœud courant est traitée après les valeurs de ses sous-arbres.

## 5.4 Arbre binaire de recherche

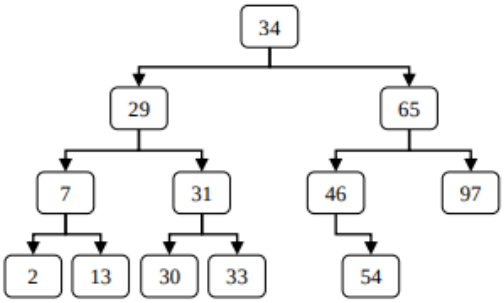
Un arbre binaire de recherche est un type spécifique d'arbre binaire, tel que pour tout nœud qui n'est pas une feuille:

Tout nœud appartenant au sous-arbre de gauche (s'il existe) possède une étiquette dont la valeur est inférieure à celle de N.

- Tout nœud appartenant au sous-arbre de droite (s'il existe) possède une étiquette dont la valeur est supérieure à celle de N.
- Les deux sous-arbres sont eux-mêmes des arbres binaires de recherche. Donc, les valeurs contenues dans l'arbre ne sont pas réparties comme on veut: elles doivent suivre un certain nombre de règles qui permettent de les organiser. Ces règles rendent l'implémentation plus compliquée, mais en contrepartie permettent un accès plus rapide au contenu de l'arbre.

Toutes les valeurs à gauche de la racine 34 lui sont strictement inférieures. Toutes les valeurs à droite lui sont strictement supérieures. De la même façon, le sous-arbre dont la racine est 29 est lui-même un arbre de recherche : toutes les valeurs à sa gauche sont inférieures à 29, et toutes celles à sa droite lui sont supérieure.

l faut noter que comme pour les tris (Chapitre 2), le concept d'arbre de recherche est défini indépendamment de la relation d'ordre utilisée pour comparer les clés des nœuds. Autrement dit, on peut utiliser une autre relation d'ordre que celle choisie dans l'exemple ci-dessus, sans pour autant devoir modifier le type abstrait arbre de recherche.



### 5.4.1 Opérations

- **cherche:** rechercher une valeur dans l'arbre.
- **minimum/maximum:** obtenir le minimum/maximum de l'arbre.
- **predecesseur/successeur:** obtenir le prédécesseur/successeur d'un nœud.
- **insere:** insérer une valeur au niveau des feuilles, au bon endroit.
- **supprime:** supprimer une valeur de l'arbre.

L'opération recherche permet de déterminer si l'arbre contient une valeur en particulier. Les opérations minimum et maximum renvoie les valeurs respectivement les plus petite et grande. Les opérations predecesseur et successeur renvoie non pas des valeurs, mais des nœuds. Il s'agit des nœuds précédant ou suivant directement un nœud donné. Cet ordre est relatif à la relation utilisée pour organiser les nœuds dans l'arbre.

L'opération insère permet de rajouter une nouvelle valeur dans l'arbre. Celle-ci sera insérée à l'endroit approprié, de manière à conserver les propriétés caractéristiques d'un arbre de recherche. Même chose pour supprime, qui est l'opération réciproque consistant à retirer une valeur déjà contenue dans l'arbre.

### Recherche

Soit int recherche(arbre a, int v) la fonction récursive qui recherche la valeur v dans l'arbre a. La fonction renvoie 1 si v apparaît dans l'arbre, et 0 sinon.

```
1
2  int recherche(arbre a, int v)
3  { int resultat=0,valeur;
4  arbre g,d;
5  if(!est_vide(a))
6  { racine(a,&valeur);
7  if(valeur==v)
8  resultat = 1;
9  else
10 { if(valeur>v)
11 { fils_gauche(a,&g);
12 resultat = recherche(g, v);
13 }
14 else
15 { fils_droit(a,&d);
16 resultat = recherche(d, v);
17 }
```

```
}
}
return resultat;
}
}
```

La complexité temporelle de cette fonction dépend de la hauteur de l'arbre :

- Pour un arbre équilibré contenant N nœuds, cette fonction a une complexité en *O(log n)* , car on effectue une recherche dichotomique (cf. les TP pour plus de détails sur la recherche dichotomique).
- Par contre, si l'arbre est dégénéré, la complexité est en *O(n)*(car on se ramène à une recherche dans une liste chaînée).

## Minimum et maximum

Soit la fonction void minimum(arbre a, arbre *m) qui renvoie via le paramètre m le sous-arbre dont la racine a la valeur minimale dans l'arbre a. La fonction renvoie -1 en cas d'erreur et 0 en cas de succès.

```

int minimum(arbre a, arbre *m)
{ int erreur=0;
 arbre g;

if(est_vide(a))
erreur=-1;
else
{ fils_gauche(a,&g);
if (est_vide(g))
*m=a;
else
minimum(g,m);
}
return erreur;
}
```

Soit la fonction void maximum(arbre a, arbre *m) qui renvoie via le paramètre m le sous-arbre dont la racine a la valeur maximale dans l'arbre a. La fonction renvoie -1 en cas d'erreur et 0 en cas de succès.

```
int maximum(arbre a, arbre *m)
{ int erreur=0;
 arbre d;

if(est_vide(a))
```

```
6  erreur=-1;
7  else
8  { fils_droit(a,&d);
9  if (est_vide(d))
10 *m=a;
11 else
12 maximum(d,m);
13 }
14 return erreur;
15 }
```

## Successeur et prédécesseur

Rappelons que dans un arbre binaire de recherche, on parle de prédécesseur d'un nœud N pour le nœud dont la valeur est maximale parmi les nœuds de valeur inférieure à celle de N. De même, on parle de successeur pour le nœud dont la valeur est minimale parmi les nœuds de valeur supérieure à celle de N.

Soit la fonction int predecesseur(arbre a, arbre *p) qui renvoie dans p le sousarbre dont la racine contient la valeur précédant l'étiquette de a dans l'arbre. La fonction renvoie -1 en cas d'erreur et 0 en cas de succès.

```
1
2  int predecesseur(arbre a, arbre *p)
3  { int erreur=0;
4   arbre g;
5  if(est_vide(a))
6  erreur=-1;
7  else
8  { fils_gauche(a,&g);
9  if(est_vide(g))
10 erreur=-1;
11 else
12 maximum(g,p);
13 }
14 return erreur;
15 }
```

Soit la fonction int successeur(arbre a, arbre *p) qui renvoie dans p le sousarbre dont la racine contient la valeur succédant à l'étiquette de a dans l'arbre. La fonction renvoie -1 en cas d'erreur et 0 en cas de succès.

```
1
2  int successeur(arbre a, arbre *s)
3  { int erreur=0;
4   arbre d;
5  if(est_vide(a))
```

```
erreur=-1;
else
{ fils_droit(a,&d);
if(est_vide(d))
erreur=-1;
else
minimum(d,s);
}
return erreur;
}
```

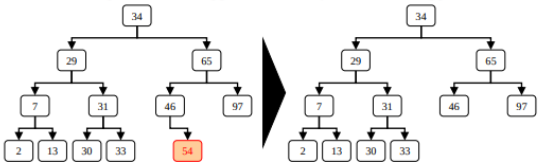
## Insertion

Soit int insertion(arbre *a, int v) la fonction qui insère au bon endroit la valeur v dans l'arbre a. La fonction renvoie 0 si l'insertion s'est bien passée, et -1 en cas d'échec (i.e. si la valeur est déjà dans l'arbre).

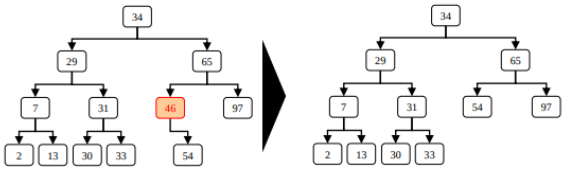
```


int insere(arbre *a, int v)
{ int valeur,erreur=0;
arbre g,d;
if(est_vide(*a))
*a = enracine(v,cree_arbre(),cree_arbre());
else
{ racine(*a,&valeur);
fils_gauche(*a,&g);
fils_droit(*a,&d);
if(v<valeur)
{ insere(&g, v);
*a = enracine(valeur,g,d);
}
else
{ if(valeur<v)
{ insere(&d, v);
*a = enracine(valeur,g,d);
}
else
 erreur=-1;
}
}
return erreur;
}
```

La complexité temporelle de la fonction est la même que celle de recherche, puisque la méthode pour rechercher le bon endroit d'insertion est la même, et que l'insertion elle-même est effectuée en temps constant.

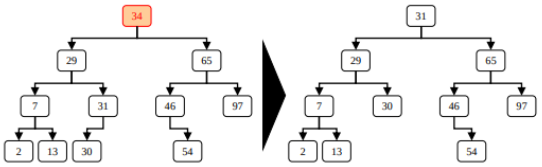
- Si le nœud que l'on veut supprimer est une *feuille* : on le supprime *directement*.




- Si le nœud possède *un seul fils* : le nœud est *remplacé* par son fils.




- Si le nœud possède *deux fils* : le nœud est remplacé par son *prédécesseur* ou son successeur.




## Suppression

La suppression dans un arbre binaire de recherche est plus complexe que l'insertion. Comme spécifié dans le type abstrait, plusieurs cas sont possibles :

Soit int supression(arbre *a, int v) la fonction qui supprime la valeur v dans l'arbre a. La fonction renvoie 0 si l'insertion s'est bien passée, et 1 en cas d'échec (si la valeur n'a pas été trouvée dans l'arbre).

```
1
2  int supprime(arbre *a, int v)
3  { int erreur=0,valeur,valtemp;
4  arbre g,d,temp;
5  if(est_vide(*a))
6  erreur=-1;
7  else
8  { racine(*a,&valeur);
9  fils_gauche(*a,&g);
10 fils_droit(*a,&d);
11 if(v<valeur)
12 { supprime(&g,v);
13 *a = enracine(valeur,g,d);
14 }
15 else if(v>valeur)
16 { supprime(&d,v);
17 *a = enracine(valeur,g,d);
18 }
19 else
```

```
{ if(est_vide(g))
{ if(est_vide(d))
{ free(*a);
*a=cree_arbre();
}
else
{ temp=*a;
*a = d;
free(temp);
}
}
else
{ if(est_vide(d))
{ temp=*a;
*a = g;
free(temp);
}
else
{ predecesseur(*a,&temp);
racine(temp,&valtemp);
supprime(&g, valtemp);
*a = enracine(valtemp,g,d);
}
}
}
}
return erreur;
}
```

La complexité de la fonction est la même que pour celle de recherche.

## 5.5 Exercice

- Écrivez une fonction arbre genereArbreRech(int n) qui génère aléatoirement un arbre binaire de recherche contenant n nœuds.
- Donner une version itérative de la procédure INSERER .
- On peut trier un ensemble donné de n nombres en commençant par construire un arbre binaire de recherche contenant ces nombres (en répétant INSERER pour insérer les nombres un à un), puis en imprimant les nombres via un parcours infixe de l'arbre. Quels sont les temps d'exécution de cet algorithme de tri, dans le pire et dans le meilleur des cas ?

Université Hassiba Benbouali de Chlef
Faculté des Sciences
département d'informatique

ASD II
Niveau: Licence 2
2020-2021

# TD0: Rappel



1. Ecrire un sous-programme récursif qui calcule la somme des n premiers carrés. Par exemple, si n vaut 3, ce sous-programme calculera $1^2 + 2^2 + 3^2$. Ce sous programme n'est défini que pour un n supérieur à 0.

2. Ecrire une fonction récursive qui calcule les valeurs de la série de Fibonacci, définie par :
   - $u_0 = 0$
   - $u_1 = 1$
   - $u_n = u_{n-1} + u_{n-2}$

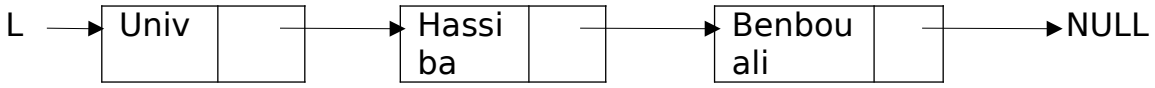
   Ecrivez cette fonction sous forme itérative et sous forme récursive. Laquelle des deux variantes est préférable ici ?

3. On veut représenter une liste de mots en utilisant les listes chainées, où chaque élément de la liste est définit par un mot (chaine de caractères) et l'adresse de l'élément suivant.
   - Ecrire la déclaration de cette structure.
   - Ecrire un sous-programme qui prend en argument une liste de mots et renvoi le nombre des éléments contenus dans cette liste.
   - Ecrire un sous-programme qui prend en argument une liste de mots et un mot M puis insère un nouvel élément (dont on demande à l'utilisateur de saisir le contenu) après l'élément qui contient le mot M dans la liste.

Exemple :

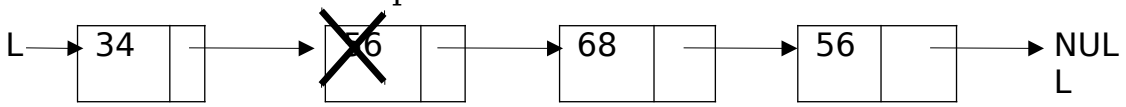
on veut insérer un nouvel élément dans la liste L après l'élément qui contient « Univ », l'utilisateur va saisir Chlef par exemple :

Avant l'insertion :

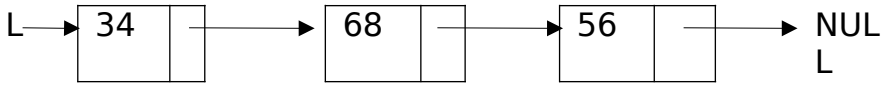



Après l'insertion :


4. Ecrire un sous-programme qui prend en argument une liste simple d'entiers et un entier X, et qui supprime de la liste la première occurrence de l'élément qui contient X.




Supprimer la première occurrence de X = 56 :






5. A l'aide de la notion de Pile on veut écrire un programme qui permet de lire au clavier une suite de caractères qui contient des parenthèses ou crochets : ( ) [ ] et vérifie que l'écriture est cohérente ou non.

Exemple :

- ( A + T [ [ R ] ) : n'est pas cohérente,
- ( A D ( [ ] ) ) : est cohérente.

En générale : il faut que le nombre des symboles (parenthèses ou crochets) ouverts soit égal au nombre du même symbole fermé.

6. A l'aide de la notion de File on veut représenter une salle d'attente, elle s'ouvre à 08h00, un client arrive toutes les 03 minutes, il est servie en 05 minutes, la salle peut contenir au maximum N personnes (capacité de la salle).

Ecrire un programme qui demande la capacité de la salle puis calcule et affiche à quelle heure la salle soit pleine.

Université Hassiba Benbouali de Chlef
Faculté des Sciences
Département d'informatique

ASD II
Niveau: Licence 2
2020-2021

# TD1: Complexité

**Exercice 1**

On note par T(n), le nombre d'instructions élémentaires en fonction de la taille des données n Pour chacun des fonctions T(n) suivant, déterminer sa complexité asymptotique dans la notation Grand-O. Par exemple: $T(n) = 3n \in O(n)$.

1. $T1(n) = 6n^3 + 10n^2 + 5n + 2$
2. $T2(n) = 3 \log_2 n + 4$
3. $T3(n) = 2n + 6n^2 + 7n$
4. $T4(n) = 7k + 2$ avec $(k \leq 100)$
5. $T4(n) = 4 \log_2 n + n$
6. $T5(n) = 2 \log_{10} k + kn^2$ $(k \leq 50)$

**Exercice 2**

Donnez la complexité des algorithmes suivants (n est une variable entière initialisée).

```
Algorithme a1
Var
        s : réel ;
        i, n : entier ;
Début
        s  ←  0 ;
        i  ←  0 ;
        Tant que i  ≤  n faire
                s  ←  s + 1 ;
                i  ←  i + 1 ;
        Fin Tant que ;
Fin.
```

```
Algorithme a2
Var
        s : réel ;
        i, j, n : entier ;
Début
        s  ←  0 ;
        i  ←  0 ;
        Tant que i  ≤  n faire
                j  ←  0 ;
                Tant que j  ≤  n faire
                        s  ←  s + 1 ;
                        j  ←  j + 1 ;
                Fin Tant que ;
                i  ←  i + 1 ;
        Fin Tant que ;
Fin.
```

```
Algorithme a3
Var
        s : réel ; i, j, k, n : entier ;
Début
        s  ←  0 ; i  ←  0 ;
        Tant que i  ≤  n faire
                j  ←  0 ;
                Tant que j  ≤  n faire
                        k  ←  0 ;
                        Tant que k  ≤  n faire
                                s  ←  s + 1 ;
                                j  ←  j + 1 ;
                        Fin Tant que ;
                        s  ←  s + 1 ;
                        j  ←  j + 1 ;
                Fin Tant que ;
                i  ←  i + 1 ;
        Fin Tant que ;
Fin.
```

```
Algorithme a4
Var
        s : réel ;
        i, j, n : entier ;
Début
        s  ←  0 ;
        i  ←  0 ;
        Tant que i  ≤  n faire
                j  ←  0 ;
                Tant que j  ≤  n * n faire
                        s  ←  s + 1 ;
                        j  ←  j + 1 ;
                Fin Tant que ;
                i  ←  i + 1 ;
        Fin Tant que ;
Fin.
```

**Exercice 3**

On considère trois tris élémentaires : le tri sélection, le tri par insertion, et le tri bulle. On considère pour un tableau les deux cas extrêmes où le tableau est déjà trié (dans l'ordre croissant), et celui où il est trié dans l'ordre décroissant. Décrire avec précision le comportement et la complexité de chacun des algorithmes dans ces deux cas. Quelles conséquences peut-on en tirer ?

**Exercice 4 (Concours Formation doctorale, université Tlemcen 2017)**

que calcule l'algorithme suivant ? (justifiez votre réponse)

```
variables N,M,res : entiers naturels
début
entrer (N,M) ;
res=0 ;
tanque (N!=0) faire
        si (N mod 2!=0) alors
            res=res+M ;
        fsi
        N=N div 2 ;
        M=M*2 ;
FinTanque
afficher(res) ;
fin
```

Quelle est la complexité de l'algorithme précédent

**Exercice 5 (Concours Formation doctorale, université Oran 1, 2017)**

Soit une matrice Mat d'entiers à n lignes et m colonnes (n,m≤100). sa particularité est que si on la parcourt par ligne, la séquence des éléments rencontrés est strictement croissante. Par exemple, pour une matrice Mat (n=3 et m=3), ses éléments pourraient être :

| | | |
|---|---|---|
| 3 | 8 | 10 |
| 17 | 24 | 27 |
| 29 | 39 | 5 |

Supposons que certaines lignes de la matrice peuvent ne être triées :

1. Ecrire une fonction ***Verif_ligne*** qui renvoi vrai si une ligne de la matrice ne respecte pas le tri sous indiqué
2. Ecrire une procedure Tri_ins de tri par insertion et faire appel à cette dernière dans une procédure **tri_mat** pour effectuer le tri des lignes non ordonnés.
3. Donner la complexité de la procédure **Tri_ins**

**Exercice 6**

On considère, pour effectuer la recherche d'un élément dans un tableau, la recherche séquentielle et la recherche dichotomique. On s'intéresse à leur complexité temporelle. Pour cela, considérer un tableau ayant mille éléments (version trié, et version non trié). Pour chaque algorithme, et pour chaque version du tableau, combien de comparaisons sont à effectuer pour :

- Trouver un élément qui y figure ?
- Trouver un élément qui n'y figure pas ?
- Quels sont les cas où le tableau est parcouru complètement et les cas où un parcours partiel est suffisant ?
- Conclure en donnant la complexité temporelle pour chaque algorithme.

# Exercice facultatif

**Exercice 1 (Concours Formation doctorale, USTHB, 2014)**

Soit E un ensemble d'entiers strictement positifs E= $\{e_1, \ldots, e_n\}$ tel que $1 \leq e_i \leq k$.
Exemple k=20 et E = {1, 7, 11, 14, 17}

1. Donner une représentation de E qui permet une recherche en O(1) et écrire l'algorithme de recherche d'une valeur donnée x avec $1 \leq x \leq k$.
2. Ecrire les algorithmes de recherche du successeur de x ainsi que du prédécesseur de x. Donner les complexités respectives.

**Exercice 2** Calculez la complexité du code suivant (vous pouvez supposer que la fonction ecrire() est O(1), et que les variable n,m,x ontété initialisée) :

```
Algorithme b1
Var
        i, j, m, n : entier ;
Début
        i  ←  1 ;
        m  ←  n ;
        Tant que i  ≤  n faire
                j  ←  1 ;
                Tant que j  ≤  m faire
                        Ecrire (j) ;
                        j  ←  j +1 ;
                        i  ←  i + 1 ;
                Fin Tant que ;
                i  ←  i + 1 ;
        Fin Tant que ;
Fin.
```

```
Algorithme b2
Var
        i, j, m, n, r, x, k : entier ;
Début
        r  ←  0 ;
        i  ←  1 ;
        Tant que i  ≤  n faire
                j  ←  1 ;
                Tant que j  ≤  m faire
                        r  ←  2 * (i + j) + r ;
                        j  ←  j +1 ;
                Fin Tant que ;
                k  ←  1 ;
                Tant que k  ≤  x faire
                        r  ←  r * k ;
                        k  ←  k +1 ;
                Fin Tant que ;
                Ecrire (r) ;
                i  ←  i + 1 ;
        Fin Tant que ;
Fin.
```

Université Hassiba Benbouali de Chlef
Faculté des Sciences
Département d'informatique

ASD II
Niveau: Licence 2
2020-2021

# TD3: Liste Chaînée, File, Pile

**Exercice 1**

L'objectif de ce TD est d'écrire une nouvelle implantation de liste chaînée, différente de celle vue en cours, mais ayant la même interface (mêmes services proposés aux utilisateurs du module). L'implantation que vous allez écrire est celle d'une liste doublement chaînée, dans laquelle:

1. chaque cellule contient un pointeur sur la cellule suivante et un pointeur sur la cellule précédente,
2. la structure Liste contient un pointeur sur la première cellule et un pointeur sur la dernière cellule.

Ainsi, il est possible de parcourir la liste dans les deux sens.

Un autre intérêt de cette implantation est que l'insertion d'un élément en fin de liste (ajoutEnQueue) peut se faire en temps constant, c'est-à-dire en un nombre d'opérations qui ne dépend pas du nombre d'éléments dans la liste. Etait-ce le cas avec l'implantation vue en cours ? Pourquoi ?

La structure suivante code des listes doublement chaînées avec maillon vide.

```
struct sCellule
{
  int info;
  struct sCellule *suivant;
  struct sCellule *precedent;
};
typedef struct sCellule Cellule;
```

```
struct sListe
{
  Cellule *prem;
  Cellule *last;
};
typedef struct sListe Liste;
```

Écrivez les primitives(en Langage C):

| **Fonction** | **Precondition / Postcondition** |
|---|---|
| **void initialise(Liste * l);** | **Precondition** : l non prealablement initialisee<br>**Postcondition** : la liste l initialisee est vide |
| **Int estVide(const Liste * l);** | **Precondition** : l prealablement initialisee et manipulee uniquement a travers les operations du module.<br>**Résultat** : 1 si l est vide, 0 sinon |
| **unsigned int nbElements(const Liste * l);** | **Precondition** : l initialisee et manipulee uniquement a travers les operations du module<br>**Résultat** : nombre d'elements contenus dans la liste |
| **void ajoutEnTete(int e,Liste * l);** | **Precondition** : l et e initialises et manipules uniquement a travers les operations de leurs modules respectifs.<br>**Postcondition** : e est ajoute en tete de l |
| **void ajoutEnQueue(int e, Liste * l);** | **Precondition** : l et e initialises et manipules uniquement a travers les operations de leurs modules respectifs<br>**Postcondition** : e est ajoute en fin de la liste l |
| **void suppressionEnTete(Liste * l);** | **Precondition** : l n'est pas vide<br>**Postcondition** : la liste l perd son premier element |
| **void insererElement(int e, Liste * l, unsigned int position);** | **Preconditions** : l initialisee et manipulee uniquement a travers les operations du module, 0 <= position <= nbElements(l)<br>Postconditions : une copie independante de e est inseree de sorte a ce qu'elle occupe la position indiquee dans la liste l (les positions etant numerotees a partir de 0). |

**Exercice 2**

Le but de cet exercice est d'écrire (en langage algorithmique) une procédure qui inverse une file d'entiers qui lui est passée en paramètre. On demande de ne pas utiliser de tableau ou de liste de travail pour effectuer l'inversion, mais d'utiliser plutôt une pile. Il existe en effet une méthode très simple pour inverser une file en utilisant une pile.

**Exercice 3**

Un problème fréquent pour les compilateurs et les traitements de textes est de déterminer si les parenthèses d'une chaîne de caractères sont balancées et proprement incluses les unes dans les autres. On désire donc écrire une fonction qui teste la validité du parenthésage d'une expression :

1. on considère que les expressions suivantes sont valides : "()", "[([bonjour+]essai)7plus- ];"
2. alors que les suivantes ne le sont pas : "(", ")(", "4(essai]".

Notre but est donc d'évaluer la validité d'une expression en ne considérant que ses parenthèses et ses crochets. On suppose que l'expression à tester est dans une chaîne de caractères, dont on peut ignorer tous les caractères autres que '(', '[', ']' et ')'. Écrire la fonction valide(ch : chaîne de caractères) : booléen qui renvoie vrai si l'expression passée en paramètre est valide, faux sinon.

Université Hassiba Benbouali de Chlef
Faculté des Sciences
Département d'informatique

ASD II
Niveau: Licence 2
2020-2021

# TD3: Arbre

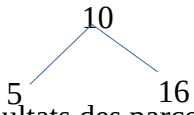
**Exercice 1**

**Q1:** Dessiner l'arbre binaire de recherche (**ABR**) qui contient:

- 10, 1, 4, 2, 12, 5, 6, 3, 14, 8 ,9 , 15, 11, 7, 13.
- 8, 4, 12, 2, 6, 10, 13, 1, 5, 7, 9, 11, 15, 14.

**Q2:** Lequel de ces deux **ABRs** est le plus efficace.

**Q3:** Ajouter les éléments 2, 4, 8, 20, 18, 25, 1 à l'**ABR** représenté sur la figure ci-dessous.




**Q4:** Donnez les résultats des parcours infixe, postfixe et préfixe de

**Q5:** Quelle est la particularité du parcours infixe?

**Q6:** Dessinez les **ABRs** que l'on obtient après avoir supprimer 1, puis 16 et enfin 10.

**Q7:** Quel est le nombre de nœuds dans un arbre binaire complet.

**Exercice 2:** Ecrire en langage C (fonctions ou procédures) réalisant :

- L'affichage des valeurs d'un ABR par ordre croissant.
- La recherche de la valeur a dans un ABR.
- La recherche du maximum des valeurs portées par les nœuds d'un ABR.
- La recherche du minimum des valeurs portées par les nœuds d'un ABR.
- L'insertion de la valeur a dans un ABR.

**Exercice 3**

Ecrire en langage C la fonction qui détermine la hauteur d'un arbre binaire .

**Exercice 4**

Ecrire en langage C la procédure itérative du parcours préfixé dans un arbre binaire.

**Exercice 5**

Ecrire en langage algorithmique la procédure de suppression d'un élément dans un arbre binaire de recherche.

## Exercices Facultatifs

**Exercice 1 (Concours Formation Doctorale, Université d'Oran 1 2016/2017)**

**Exercice 3 : (7 points)**

Soit un arbre binaire de recherche B représenté par chaînage.

1- Ecrire une fonction récursive ***Court_Chemin*** qui permet de calculer la longueur du plus court chemin dans l'arbre B.

2- Ecrire une fonction itérative ***Min_ABR*** qui retourne un pointeur sur le nœud contenant la valeur minimale de B.

3- Ecrire une procédure récursive ***Affiche_Dec*** qui affiche les valeurs de B dans l'ordre décroissant.

## Exercice 2+3 (Concours Formation Doctorale, Université d'Oran 1 2017/2018)

Université Oran 1
République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Faculté des sciences exactes et appliquées

**Concours Formation Doctorale**
**Algorithmiques et Structures de Données**

**Durée : 1h30**
**03/11/2018**

**Exercice N°1 : 6 points (Traiter toutes les questions en langage C)**
1- Ecrire une fonction récursive 'détruit-arbre' qui libère la mémoire occupée par tous les nœuds d'un arbre binaire.
2- Ecrire une fonction récursive 'nb-nœud' qui calcule le nombre de nœuds d'un arbre binaire.
3-Ecrire une fonction récursive 'Affich-ABR' qui affiche les valeurs des nœuds d'un arbre binaire de recherche par ordre croissant (choisir le bon parcours de l'arbre).
4- Ecrire une fonction récursive 'inser' qui ajoute une valeur dans l'arbre binaire de recherche. (ce nouveau nœud doit être placé correctement dans l'arbre binaire de recherche)

**Exercice N° 2: 8 Points**
Soit une liste simplement chainée d'entiers donnée par deux pointeurs D et F sur le premier et le dernier élément.
1- Ecrire une procédure permettant d'insérer un élément en fin de liste. Calculer sa complexité.
2- En utilisant la procédure définie en 1 écrire une procédure récursive permettant de construire deux listes L1 et L2 (simplement chainées accessibles par deux pointeurs) à partir d'un arbre binaire de recherche B et d'une valeur entière val telle que :
L1 contient toutes les valeurs Entières de B qui sont inférieurs à val.
L2 contient toutes les valeurs Entières de B qui sont supérieurs à val.
3- Calculer sa complexité.

## Exercice 4 (Concours Formation Doctorale, Université de Tlemcen 2017/2018)

Exercice 3 : (6 points)

3.1 Dans ce qui suit, on notera N le nombre de nœuds d'un arbre binaire et H sa hauteur.

a. Quelle est la hauteur maximale d'un arbre à N nœuds ?
b. Quel est le nombre maximal de feuilles d'un arbre de hauteur H ?
c. Quel est le nombre maximal de nœuds d'un arbre de hauteur H ?
d. Quelle est la hauteur minimale d'un arbre de N nœuds ?

3.2 Considérons une liste doublement chainée **l1**, et une liste doublement chainée circulaire **l2** sachant que les deux listes sont triées dans un ordre croissant. Calculer intuitivement la complexité (pire cas) des opérations suivantes :

a. Trouver le plus grand élément dans chacune des deux listes ?
b. Inverser chacune des deux listes ?

Supposons que l'on a une fonction **Nœud* position(Nœud ****racine***, **int** ***pos***)** permettant de retourner un pointeur vers l'élément de la liste se trouvant à la position ***pos***, et que la taille de la liste **l2** est **N**. On veut appliquer une recherche pour vérifier si un élément appartient à **l2** en utilisant la fonction précédente.

c. Quelle serait la complexité (pire cas) de cette recherche ?

Université Hassiba Benbouali de Chlef
Faculté des Sciences
Département d'informatique

ASD II
Niveau: Licence 2
2020-2021

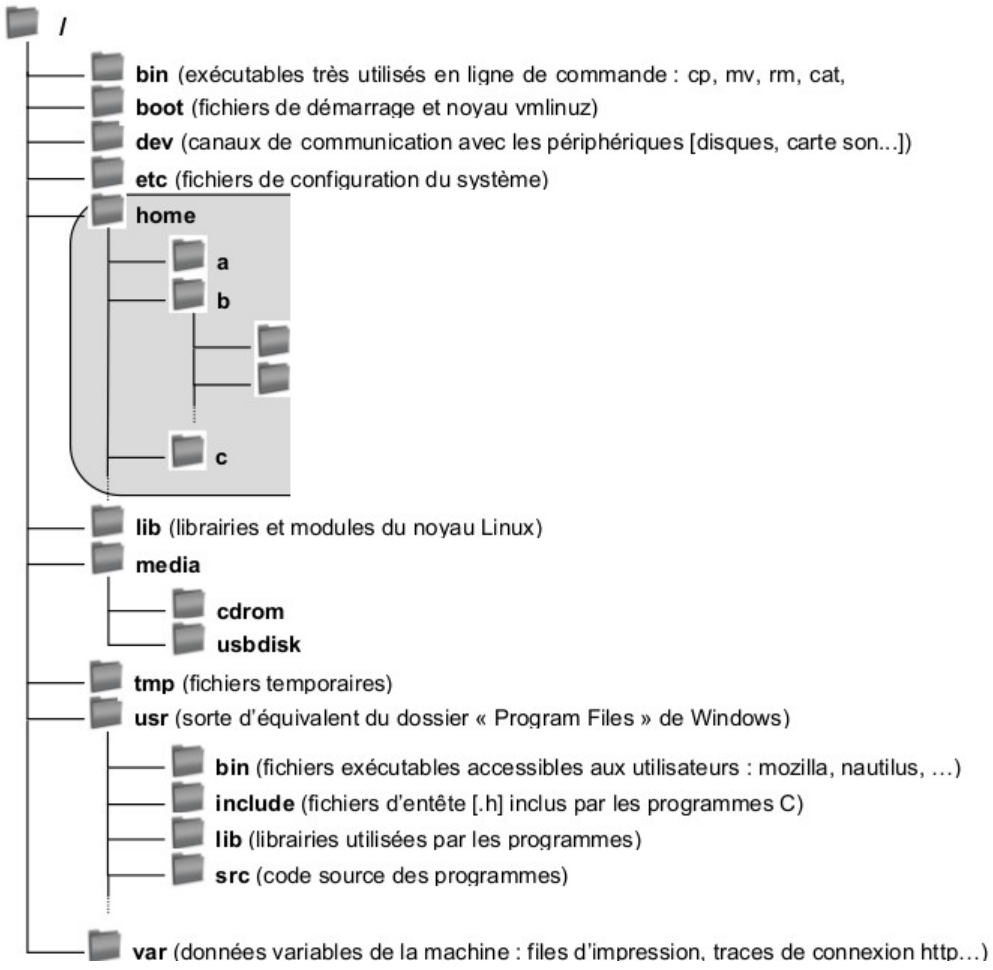
# TP1: Première compilation avec gcc sous Linux

**Objectifs:**

- Découvrir l'environnement Linux.
- Expérimenter avec la ligne de commande.
- Compiler avec la gcc

## 1. Familiarisation avec le système de fichiers Linux

Redémarrez l'ordinateur sous Linux (Ubuntu), Pour utiliser au mieux son compte Linux, il est nécessaire de connaître quelques notions basiques sur le système de fichiers Linux. La racine du système de fichier (l'équivalent du « C :\ » d'un Windows non partitionné) est le répertoire « / ». Voici un schéma des principaux répertoires que contient ce dossier racine.



**2. Création d'une arborescence ASD 2 dans votre répertoire personnel**

Vous allez créer l'arborescence suivante dans votre répertoire personnel, en n'utilisant que la ligne de commande :

1. Allez dans le terminal, puis vérifiez que vous êtes dans votre répertoire personnel en tapant la commande **pwd** . Si vous n'y êtes pas, retournez-y en tapant **cd** (cela signifie « change directory », et si l'on ne précise pas de répertoire de destination, on va par défaut dans le répertoire personnel).
2. créez le répertoire **ASD2-TP2018** en tapant la commande suivante (respectez bien l'espace après **mkdir**, mais n'en mettez pas dans le nom du répertoire) : **mkdir ASD2-TP2018.**
3. vérifiez que ce nouveau répertoire apparaît dans le répertoire courant, en tapant **ls**
4. allez dans le répertoire créé en tapant la commande : **cd ASD2-TP2018.**
5. créez le répertoire TP1 en tapant **mkdir TP1**

Vérifiez que vous retrouvez bien les dossiers créés en explorant votre dossier personnel en mode graphique.

**Exercice 1 : Programmer sans environnement de développement**

En L1, vous avez programmé en C/C++ à l'aide de **Dev-C++,** qui est un « environnement de développement intégré » regroupant un éditeur de texte, un compilateur, des outils automatiques de fabrication, et un débogueur. Il en existe d'autres, par exemple **CodeBlocks**. Cependant, il n'est pas indispensable d'utiliser ce genre d'environnement intégré pour programmer : on peut aussi utiliser un éditeur de texte et lancer la compilation et l'exécution en ligne de commande, depuis le terminal. C'est ce que nous allons faire dans ce TP. Cela vous permettra, par la suite, de mieux comprendre ce que fait un environnement de développement lorsque vous cliquez sur « Compiler », par exemple. Cela vous permettra aussi de mieux comprendre les mystérieux fichiers « **Makefile** » utilisés par ces environnements, car vous en aurez fait vous-même (mais nous verrons cela plus tard...).

Nous allons utiliser l'éditeur de texte « **gedit** ». Pour cela, allez dans le terminal, puis:

- Vérifiez que vous êtes dans le répertoire **TP1** en tapant **pwd** .
- Lancez **gedit** en tapant **gedit hello.c &** . Comme le fichier **hello.c** n'existe pas encore, **gedit**
- crée un fichier vide que vous allez pouvoir remplir.
- Tapez le code suivant dans gedit :

```
#include <stdio.h>
int main()
{
printf("Hello world !\n");
return 0;
}
```

Nous allons maintenant compiler ce code à l'aide de gcc. Il s'agit du principal compilateur C libre. Pour compiler, retournez dans le terminal (sans fermer gedit) et tapez :

**gcc –g -Wall -ansi -pedantic -o hello.out hello.c**

Tapez **man gcc** pour comprendre ce que signifient les différents éléments de cette commande et compléter le tableau suivant (aide : une fois la documentation affichée, tapez **/mot** pour rechercher un mot, **n** pour passer à l'occurrence suivante, **q** pour sortir).

| | |
|---|---|
| gcc | |
| -g | |
| -Wall | |

| | |
|---|---|
| -ansi -pedantic | |
| -o hello.out | |
| hello.c | |

Toujours dans le terminal, appuyez plusieurs fois sur la flèche en haut du clavier. Que se passe-t-il ? A quoi cela peut-il servir ?

| |
|---|
| |

Corrigez les erreurs de compilation éventuelles en modifiant votre code source dans gedit, en sauvegardant et en recompilant. Recommencez jusqu'à ne plus avoir aucune erreur. Vous pouvez ensuite exécuter votre programme en tapant dans le terminal : **./hello.out**

Université Hassiba Benbouali de Chlef
Faculté des Sciences
Département d'informatique

ASD II
Niveau: Licence 2
2020-2021

# TP2 :

**Commencez par créer un répertoire TP1 à l'intérieur de votre répertoire ASD**

1. Créez un fichier exo1.c avec gedit.
2. Ecrivez-y l'entête et le code de la procédure suivante :
   *void **remplirTableauAvecEntiersAleatoires**(int * tab, int taille, int valeurMax);*
   **Preconditions** : tab est un tableau pouvant contenir "taille" entiers.
   **Postconditions** : tab est rempli avec des nombres entiers aleatoires compris entre 0 (inclus) et ValeurMax (exclue). */
   **Aide** : en incluant le fichier stdlib.h, on peut utiliser la fonction rand(), qui renvoie un int compris entre 0 et une constante appelée RAND_MAX. Ainsi, la formule **rand()/((double)RAND_MAX + 1)** fournit un double dans [0.0, 1.0[. Ilvous reste à trouver comment, partant de cela, arriver à un int entre 0 et valeurMax.

3. Ecrivez à présent un « main » qui demande à l'utilisateur quelle taille de tableau il souhaite, qui réserve la mémoire pour ce tableau et qui le remplit avec des nombres entiers aléatoires compris entre 0 et 1000000.

4. Ajoutez à votre fichier l'entête et le code d'une procédure de tri par sélection du minimum.

5. Ajoutez à votre fichier l'entête et le code d'une procédure de tri par insertion.

6. Complétez votre « main » pour qu'il demande à l'utilisateur quel algorithme de tri il souhaite utiliser (en tapant par exemple « 0 » pour le tri par sélection, et « 1 » pour le tri par insertion). L'algorithme choisi sera appelé pour trier le tableau de nombres aléatoires.

7. Ajoutez à votre « main » le minutage du temps d'exécution du tri.
   **Aide :** incluez le fichier time.h et aidez-vous de la page web suivante : http://www.codecogs.com/reference/computing/c/time.h/clock.php ( Annexe)

8. Exécutez plusieurs fois votre programme et utilisez un tableur pour tracer les deux courbes du temps d'exécution en fonction de la taille du tableau. Enregistrez votre tableur au format xls.

**Annexe:** http://www.codecogs.com/reference/computing/c/time.h/clock.php

## Interface

```
#include <time.h>
    clock_t   clock (void)
```

## Description

The **clock** function determines the amount of processor time used since the invocation of the calling process, measured in *CLOCKS_PER_SEC* of a second.

The code below counts the number of seconds that passed while running some for loop.

### Example - Determine processor time used

Workings

```
#include <stdio.h>
#include <time.h>
#include <math.h>
int main()
{
  clock_t start = clock();
  for (long i = 0; i < 100000000; ++i)
    exp(log((double)i));
  clock_t finish = clock();
  printf("It took %d seconds to execute the for loop.\n",
  (finish - start) / CLOCKS_PER_SEC);
  return 0;
}
```

Solution

**Output:**

```
It took 23 seconds to execute the for loop.
```

Université Hassiba Benbouali de Chlef
Faculté des Sciences
Département d'informatique

ASD II
Niveau: Licence 2
2020-2021

# TP3: Liste doublement Chaînée

**Exercice 1**

Commencez par créer un répertoire TP3 à l'intérieur de votre répertoire ASD2, en utilisant les lignes de commandes vues au TP1 (cd, mkdir, ls...).
L'objectif de ce TP est d'écrire une nouvelle implantation de liste chaînée, différente de celle vue en cours, mais ayant la même interface (mêmes services proposés aux utilisateurs du module). L'implantation que vous allez écrire est celle d'une liste doublement chaînée, dans laquelle:

1. chaque cellule contient un pointeur sur la cellule suivante et un pointeur sur la cellule précédente,
2. la structure Liste contient un pointeur sur la première cellule et un pointeur sur la dernière cellule.

Ainsi, il est possible de parcourir la liste dans les deux sens.

Un autre intérêt de cette implantation est que l'insertion d'un élément en fin de liste (ajoutEnQueue) peut se faire en temps constant, c'est-à-dire en un nombre d'opérations qui ne dépend pas du nombre d'éléments dans la liste. Etait-ce le cas avec l'implantation vue en cours ? Pourquoi ?

Récupérez sur le site de de ASD2 les fichiers Liste.h et Liste.c. Enregistrez les dans votre répertoire Ecrivez-y les #include requis, puis le code de la procédure d'initialisation d'une liste (uniquement cette procédure-ci pour l'instant !). Enregistrez le fichier mais ne le fermez pas, vous y reviendrez plus tard. Créez un nouveau fichier main.c, et écrivez-y un programme principal minimal, qui teste la procédure d'initialisation.

Implantez progressivement les autres fonctions et procédures décrites dans le .h. Attention : vous ne devez PAS écrire toutes les procédures d'un coup. Testez vos sous-programmes au fur et à mesure, en les appelant dans le main. Vérifiez que vous procédures fonctionnent aussi dans les cas particuliers : liste vide, liste ne contenant qu'un seul élément, etc.

Université Hassiba Benbouali de Chlef
Faculté des Sciences
Département d'informatique

ASD II
Niveau: Licence 2
2020-2021

# TP4: arbre

1. Récupérez sur le site de ASD2 les fichiers arbre.h et arbre.c. Créez un nouveau fichier main.c et écrivez-y un « main() » vide pour l'instant. Complétez le fichier arbre.c en y ajoutant les définitions des fonctions et procédures suivantes :

- initialiserArbre
- insererElement
- rechercheElement
- hauteur
- testamentArbre

**Remarque**: Pour certains de ces sous-programmes, vous devrez utiliser une fonction ou procédure auxiliaire travaillant sur un sous-arbre, et prenant donc comme paramètre supplémentaire l'adresse du nœud dans lequel le sous-arbre est enraciné. Ces sous- programmes auxiliaires restent internes au module Arbre : ils n'apparaissent donc pas dans le .h, et sont précédés du mot-clé **static** dans le .c.

Dans le fichier main.c, complétez le main() pour qu'il insère dans un arbre binaire de recherche 255 entiers aléatoires compris entre 1 et 100000, puis qu'il calcule la hauteur maximale de l'arbre et sa profondeur moyenne. Le programme devra bien entendu se terminer proprement, c'est-à-dire en vidant l'arbre. Testez que tout fonctionne bien, en utilisant notamment la procédure d'affichage d'arbre qui vous est fournie. Vérifiez que vous n'obtenez pas le même arbre si vous exécutez deux fois le programme.
**Aide** : pour le tirage de nombres aléatoires, vous pouvez vous inspirer du petit programme suivant :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#define VALEUR_MAX 100000
int main()
{
int alea1, alea2;
/* Initialisation du générateur, à ne faire qu'une fois
dans le programme. En l'initialisant avec l'heure
à laquelle l'exécution est lancée, on aura des valeurs
différentes à chaque exécution, ce qui est souhaitable ! */
srand((unsigned) time(NULL));
/* Tirage de deux nombres aléatoires compris entre 0 et
VALEUR_MAX incluse */
alea1 = rand()%(VALEUR_MAX + 1);
alea2 = rand()%(VALEUR_MAX + 1);
printf("%d %d\n", alea1, alea2);
return 0;
}
```

3. Complétez le main pour qu'il recherche 100 nombres aléatoires entre 0 et 100000 dans l'arbre (bien évidemment, parmi ces 100 nombres, certains seront effectivement présents dans l'arbre et d'autres non). En plus de la hauteur maximale et de la profondeur moyenne de l'arbre, le main devra afficher le nombre moyen de nœuds visités par opération de recherche.