# TREEDIR
# Projet de Langage C

**Licence Informatique - Université Paris Cité**

**2023**

*Attention :* Prenez le temps de lire attentivement TOUT le document et ce dans ses moindres détails. Relisez plusieurs fois, même en groupe. Ne vous lancez pas immédiatement dans la réalisation. Essayez d'abord de vous abstraire des problèmes techniques.

## Introduction

Le but de ce projet est de faire un programme qui simule les instructions système de manipulation de l'arborescence des dossiers/fichiers. Le programme prendra en entrée un fichier contenant une liste d'instructions et devra simuler ces instructions en manipulant un 'arbre' stocké en mémoire. Le programme se déplacera dans l'arborescence, pourra créer des dossiers/fichiers, déplacer, copier ou supprimer des dossiers/fichiers.

La structure de données manipulée par le programme sera imposée et des fichiers d'entrée test vous seront fournis.

## Structure de données

Comme nous l'avons dit, le programme devra manipuler un 'arbre' représentant l'organisation des dossiers/fichiers d'un système. Les noeuds de l'arbre représenteront soit des dossiers, soit des fichiers. Pour représenter cet arbre, nous vous imposons l'utilisation des structures de données suivantes.

```
struct noeud;
struct liste_noeud;

struct noeud{
  bool est_dossier;
  char nom[100];
  struct noeud *pere;
  struct noeud *racine;""
  struct liste_noeud *fils;
};

struct liste_noeud{
  struct noeud *no;
  struct liste_noeud *succ;
};

typedef struct noeud noeud;

typedef struct liste_noeud liste_noeud;
```

Chaque noeud de l'arbre sera représenté par un objet de type `struct noeud` tel que :

- le champ `est_dossier` est vrai si le noeud représente un dossier et faux si il représente un fichier. **Pour le noeud racine, ce champ sera égal à** `true`.
- le champ `nom` est une chaîne de caractères d'au plus 99 caractères contenant le nom du dossier/fichier. **Pour le noeud racine, ce champ sera égal à la chaîne vide** `""`.
- le champ `pere` est un pointeur vers le noeud père du noeud, si le noeud n'a pas de père car c'est le noeud racine alors le pointeur pointera vers le noeud lui mème.
- le champ `racine` est un pointeur vers le noeud racine de l'arbre.
- le champ `fils` est un pointeur vers une liste simplement chaînée des fils du noeud. Si le noeud n'a pas de fils, ce pointeur vaut `NULL`. **Les noeuds représentant des fichiers n'ont pas de fils**.

Ainsi le type `struct liste_noeud` est utilisé pour faire une liste simplement chaînée de noeuds dont le dernier élément pointe sur `NULL`.

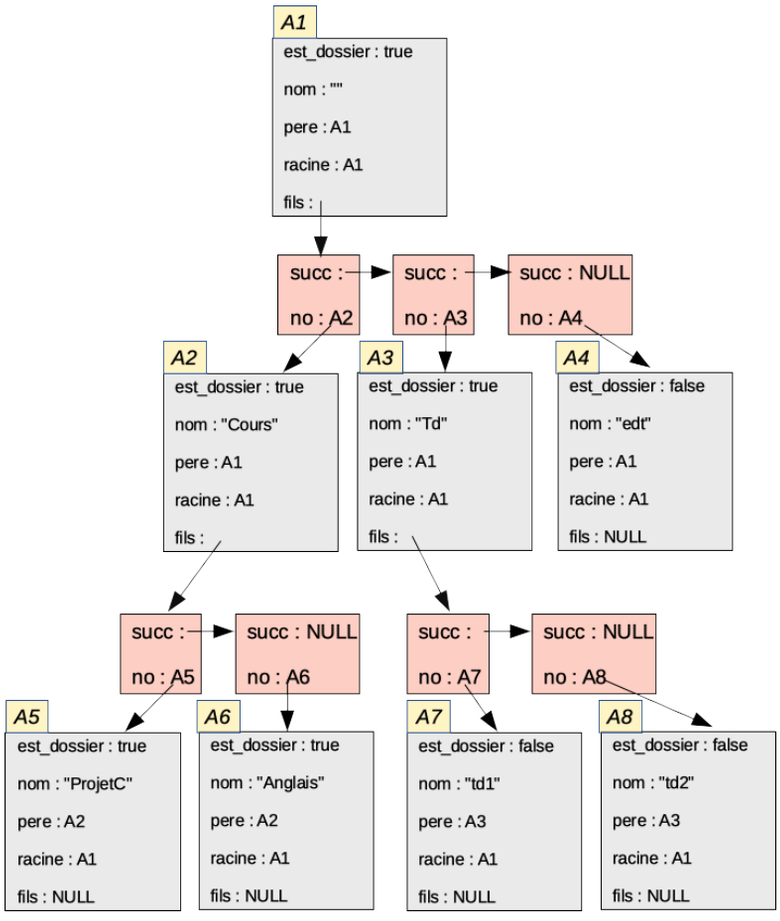


FIGURE 1 – Exemple de représentation d'un arbre

Par exemple, mettons qu'à la racine on est deux dossiers `Cours` et `Td` et un fichier `edt`, que dans le dossier `Cours` on est deux dossiers `ProjetC` et `Anglais` et dans le dossier `Td` on est deux fichiers `td1` et `td2`. La figure 1 donne une représentation graphique d'un tel arbre. Sur ce schéma, les indications A1,...,A8 indiquent les adresses de chaque noeud. Pour les listes simplement chaînées, nous les avons représentées graphiquement sans indiquer les adresses des éléments de la liste.

## Liste des instructions

Comme nous l'avons dit, le programme prendra en entrée un fichier contenant sur chaque ligne une instruction à réaliser sur l'arbre manipulé. Au début l'arbre ne contiendra que le noeud racine sans fils, dont le le champ `nom` vaut toujours `""`. À tout instant votre programme se trouvera dans un noeud de l'arbre (il s'agira toujours d'un noeud correspondant à un dossier) et au début, il se trouve donc dans le noeud racine.

Avant de donner la liste des instructions possibles, nous devons parler de chemin. À chaque noeud dans l'arbre est associé un chemin qui est une concaténation de noms (de fichiers/dossiers) et de symbole `/` servant à distinguer les différents noeuds de l'arbre. Par exemple, le chemin associé au noeud à l'adresse A5 dans l'exemple de la figure 1 est `"/Cours/ProjetC"` et le chemin associé au noeud à l'adresse A8 est `"/Td/td2"`. On supposera que le chemin associé au noeud racine est `"/"`.

Dans le contexte qui suit, un *chemin d'adresse* sera pour nous toujours non-vide (c'est-à-dire) qu'il contiendra au moins un nom et ne pourra pas être égal à `"/"` ni à la chaîne vide `""`. On supposera également qu'il finira par un nom (c'est à dire qu'il ne peut pas finir par `"/"`). En revanche il pourra être **relatif** ou **absolu**. Un chemin absolu commencera toujours par `"/"` alors qu'un chemin relatif commencera par un nom. Ainsi `"/Cours/ProjetC"` est un chemin absolu et `"Cours/ProjetC"` est un chemin relatif. Le chemin absolu indique un chemin **en partant du noeud racine** alors qu'un chemin relatif indique un chemin en partant du noeud où se trouve le programme. Par exemple, si le programme se trouve au noeud racine alors `"/Cours/ProjetC"` et `"Cours/ProjetC"` indiquent les mêmes noeuds. Par contre si il se trouve au noeud A2 alors `"/Cours/ProjetC"` et `"ProjetC"` indiquent le même noeud.

Nous détaillons maintenant les instructions et leur syntaxe.

1. `ls`
   Cette instruction affiche sur le terminal la liste des noms des fils du noeud courant. Par exemple, si le programme se trouve dans le noeud d'adresse A2 de la figure 1, alors on affichera :

```
ProjetC
Anglais
```

2. `cd chem`
   où `chem` est un chemin d'adresse. Cette instruction déplace le programme au noeud indiqué par `chem`. Si ce noeud n'existe pas ou si ce n'est pas un dossier, le programme s'arrête en affichant une erreur. Par exemple si le programme se trouve dans le noeud d'adresse A2 alors `cd /Td` l'amènera au noeud A3, et `cd /Td/td1` affichera une erreur dans le terminal.
3. `cd`
   Cette instruction ramène le programme à la racine.
4. `cd ..`
   Cette instruction ramène le programme au père du noeud où il se trouve. Pour rappel, le père du noeud racine est le noeud racine.
5. `pwd`
   Cette instruction affiche sur le terminal le chemin du noeud où se trouve le programme.
6. `mkdir nom`
   Cette instruction crée un nouveau noeud **dossier** fils du noeud courant qui portera le nom `nom`. Attention, `nom` ne peut pas être vide et ne doit contenir que des caractères alpha-numériques et ne doit pas avoir plus de 99 caractères. De plus, si le noeud courant a déjà un fils qui porte le même nom, votre programme s'arrête et affiche un message d'erreur.
7. `touch nom`
   Cette instruction crée un nouveau noeud **fichier** fils du noeud courant qui portera le nom `nom`. Attention, `nom` ne peut pas être vide et ne doit contenir que des caractères alpha-numériques et ne doit pas avoir plus de 99 caractères. De plus, si le noeud courant a déjà un fils qui porte le même nom, votre programme s'arrête et affiche un message d'erreur.

char ** ParsePath (char *, int) {

10 = capacity → *Pcapacity

PathCopy = path  size of capacity, si on dépasse la taille du tableau → Realloc

delim = "/"

8. `rm chem`
   Cette instruction détruit le noeud (ainsi que tous ses fils, et les fils de ses fils, etc) indiqué par le chemin d'adresse `chem`. Si il n'y a pas de noeud avec un tel chemin, alors votre programme s'arrête et affiche un message d'erreur. De plus, si `chem` est un chemin absolu qui indique un noeud parent du noeud où se trouve le programme, alors votre programme s'arrête et affiche un message d'erreur. Par exemple, si votre programme est dans le noeud d'adresse A5 du schéma de la figure 1, alors ces deux instructions feront une erreur `rm /Cours` et `rm /Cours/ProjetC` car elles détruiront le noeud où se trouve le programme. Attention, lorsque l'on efface un noeud fils d'un autre noeud, il faut mettre à jour la liste chaînée du père.

9. `cp chem1 chem2`
   Cette instruction copie le noeud (ainsi que ces fils, et les fils de ses fils, etc) indiqué par le chemin d'adresse `chem1` et le met au chemin d'adresse `chem2`. Si il n'y a pas de noeud au chemin d'adresse `chem1`, le programme s'arrête et envoie un message d'erreur. Le noeud copié sera renommé par le dernier nom du chemin associé au noeud indiqué par `chem2`. Attention, il faut aussi que le chemin indiqué par `chem2` auquel on retire le dernier nom indique un chemin vers un noeud **dossier** existant et que ce noeud n'ait pas de fils ayant pour nom le dernier nom du chemin du noeud indiqué par `chem2`. De plus, il ne faut pas que le noeud indiqué par le chemin `chem2` auquel on retire le dernier nom, soit un noeud du sous-arbre partant du noeud indiqué par `chem1`. Si ces conditions ne sont pas respectées, votre programme s'arrête et affiche un message d'erreur. Par exemple, si l'on est dans le noeud racine de l'arbre représenté à la figure 1, alors `cp Cours /Td/Coursbis` est valide et a pour effet de copier tout le sous-arbre partant du noeud `Cours` comme fils du noeud à l'adresse A3 et le nom du noeud copié A3 deviendra `Coursbis`. Par contre `cp Cours /Td/td1/Coursbis` n'est pas valide car le noeud au chemin `/Td/td1` est un fichier. De la même façon, `cp /Cours/Anglais /Td/td1`, n'est pas valide car le noeud situé au chemin `/Td` a déjà un fils `td1`. Finalement, `cp /Cours /Cours/Anglais/Copie` génère également une erreur car le noeud indiqué par le chemin `/Cours/Anglais` se trouve dans le sous-arbre du noeud à l'adresse A2 indiqué par le chemin `/Cours`.

10. `mv chem1 chem2`
    Cette instruction revient dans notre contexte à faire les deux instructions `cp chem1 chem2` puis `rm chem1`. Il est toutefois recommandé de programmer cette instruction sans faire la copie. Juste cela, vous permet de connaître les règles appliqués pour que le déplacement soit autorisé.

11. `print`
    Cette instruction permet d'afficher le contenu de l'arbre. Le choix est libre de la façon dont vous souhaitez l'afficher, mais il faut absolument que chaque noeud soit affiché avec le nom de son père et la liste de ses fils. Un exemple possible d'affichage dans le terminal si l'on fait cette instruction pour l'arbre de la Figure 1 est (où `D` indique un dossier et `F` un fichier) :

```
Noeud / (D), 3 fils: Cours (D), Td (D), edt (F)
Noeud Cours (D), pere : /, 2 fils : ProjetC (D), Anglais (D)
Noeud Td (D), pere : /, 2 fils : td1 (F), td2 (F)
Noeud ProjetC (D), pere : Cours, 0 fils
Noeud Anglais (D), pere : Cours, 0 fils
Noeud td1 (F), pere : Td, 0 fils
Noeud td2 (F), pere : Td, 0 fils
```

## Exemples

Voilà une liste d'instructions permettant d'obtenir l'arbre représenté à la Figure 1 :

```
mkdir Cours
cd Cours
```

CP : création nv noeud : Au lieu d'ajouter le noeud source à la destination. On crée le même type avec le même propriété.
Ca évite les problème si le noeud source est référencé d'ailleurs

On parcourt ensuite les enfants du noeud src les copier un par un en utilisant CopyNodeHelper

On vérifie quand même si Src et DstNode est un dossier

MV : il localise les noeuds Source

il vérifie ensuite si le noeud source et le noeud de destination existent bien et si le noeud de destination est 1 dossier

Mais Problème est que l'élément qu'on déplace sera pas supprimé de son emplace d'origine

Donc on a 2 instances du même éléments dans l'arbre

=> Solu : On doit supprimer le noeud source de la liste des enfants de son parent d'origine

Comme l'énoncé:

1, On va supprimer de supprimer le noeud source de la liste des enfants de son parent avant de l'ajouter à la liste des enfants du noeud de destination.

2, On doit vérifier que le dossier de destination n'est pas un sous-dossier du dossier source

Donc:

On fait une fonction IsDescendant qui vérifie si un noeud est un descendant d'un autre noeud

et

RemoveChild qui supprime un noeud enfant de la liste des enfants de son parent

puis MoveNode qui vérifie si le noeud source et le noeud de destination existent, si le noeud de Dest est un dossier et si le noeud de Dest n'est pas un descendant du noeud source

Si tout est ok, MoveNodeHelper déplace le noeud vers le noeud de DST

TrouverNoeud :

Elle va analyser le chemin

Elle divise le chemin en Noms de repertoires et fichiers individuels

Elle divise le chemin en nom de répertoire en utilisant strchr

(strchr qui renvoie le nouveau string et prend le string en argument

strchr("Hello World!", 'W');

=> "World!"

Donc strchr trouver le prochain caractère '/'

Si c'est trouvé, ca remplace temporairement ce caractère par un caractère nul "\0"

Si ya pas "/", on est à la fin du chemin

La fonction recherche le noeud enfant avec le noeud correspond à la partie restante du chemin

Sinon Null

```
mkdir ProjetC
mkdir Anglais
cd
touch edt
cp Cours /Td
rm /Td/ProjetC
rm /Td/Anglais
cd Td
touch td1
touch td2
```

Si on complète ensuite ces instructions par :

```
cp /Cours/ProjetC CopieProjet
cd
mv Td /Cours/Td
```

On obtient l'arbre dessiné à la Figure 2 (cf dernière page).

# Réalisation

Vous devez programmer en C un programme qui à l'appel attendra un nom de fichier dans lequel se trouveront des listes d'instructions et qui exécutera ces instructions. Par exemple, si votre programme s'appelle `treedir` pour l'appeler on fera `treedir instrs.txt` où `instrs.txt` est un nom de fichier avec une liste d'instructions (une instruction par ligne).

Il est impératif de respecter scrupuleusement la spécification fournie dans le sujet et de traiter les cas d'erreur. Toute violation sera jugée très défavorablement !

On vous fournira des exemples de fichiers d'instructions sur lesquels votre projet devra fonctionner mais nous attendrons également que vous fournissiez vos propres fichiers tests.

Vous serez également évalués sur la façon dont votre programme libère correctement la mémoire dynamiquement allouée.

Nous mettrons en place un forum sur Discord pour que vous puissiez communiquer entre vous et avec nous, par exemple pour faire part d'imprécisions dans le sujet. **Toute demande sur le projet devra passer par le forum.** Prenez garde à ne pas vous fier à la rumeur, la seule source d'information fiable sera le forum sur Discord ! Au moindre doute, n'hésitez pas à y poster votre question !

La communication verbale entre groupes est non seulement autorisée mais encouragée, cependant il est **strictement interdit** d'échanger du code ; ceci serait considéré comme plagiat et par conséquent jugé sévèrement. Les discussions doivent donc seulement porter sur le fonctionnement du programme et son interprétation ; il vaut donc mieux éviter de donner trop d'indications sur la façon de coder les fonctionnalités.

Vos programmes devront nécessairement pouvoir être exécutés sur les machines des salles de TP. Toute solution ne respectant pas ce critère sera jugée invalide.

Votre projet devra bien entendu être robuste (c'est à dire sans erreur) et devra être capable de gérer des fichiers erronés sans planter, mais il pourra dans ce cas s'arrêter en signalant l'erreur dans le fichier fourni.

La réalisation du projet se fera de préférence par groupe de **deux** étudiants. Et bien entendu, chacun dans un groupe devra travailler et il n'est pas exclu que des étudiants d'un même groupe n'aient pas la même note au final.

La composition des groupes devra être envoyée par mail à `sangnier@irif.fr` avant **le Vendredi 31 Mars 2023 23h59**. Toute personne n'ayant pas soumis de groupe avant cette date prend le risque de ne pas avoir de note au projet.

Pour le rendu, votre code devra être structuré en plusieurs fichiers `.c` et `.h` et un `Makefile` sera attendu. Vous n'aurez pas de rapport à écrire mais il faudra un fichier `readme.txt` expliquant comment compiler et exécuter votre projet et le cas échéant détaillant les éventuels problèmes dont souffre votre projet.

Le rendu se fera via la plateforme Moodle et des informations sur le rendu (qui aura lieu quelques jours avant la soutenance) et la soutenance (ordre de passage et instructions) seront aussi fournies plus tard. **Notez seulement que les soutenances auront lieu après les examens**.

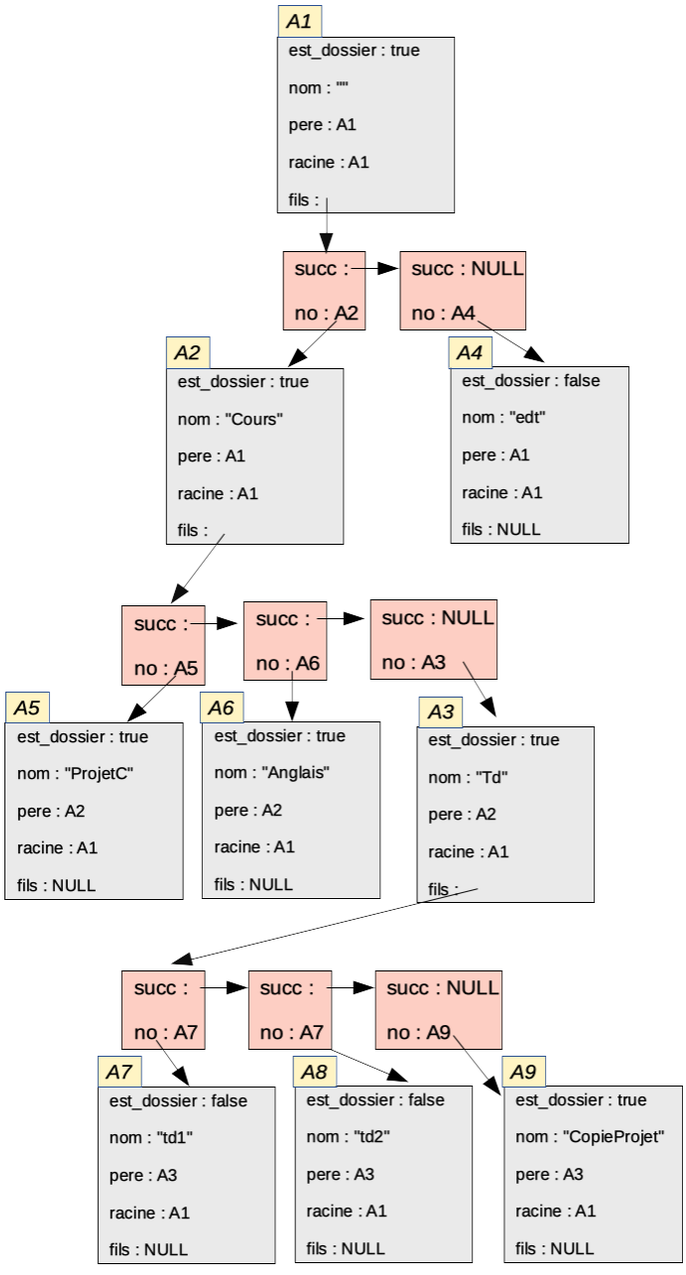


FIGURE 2 – Autre exemple de représentation d'un arbre

test 1 : OK ✓

test 2 : OK ✓

test 3 : OK ✓

test 4 : NON Il comprend pas cp Rep11 /Rep2

test 5 : NON

test 6 : Segmentation

test 7 : Rep998 ✓

test erreur 1 : Mettre existe pas d'erreur

test 2 : Mettre existe