

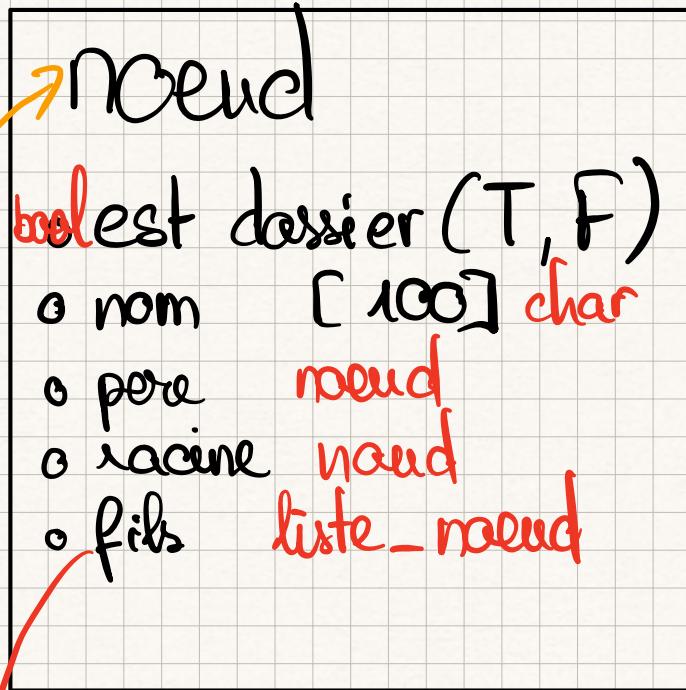
Pourquoi 1 seul P.C

Comment se répartir les tâches ?

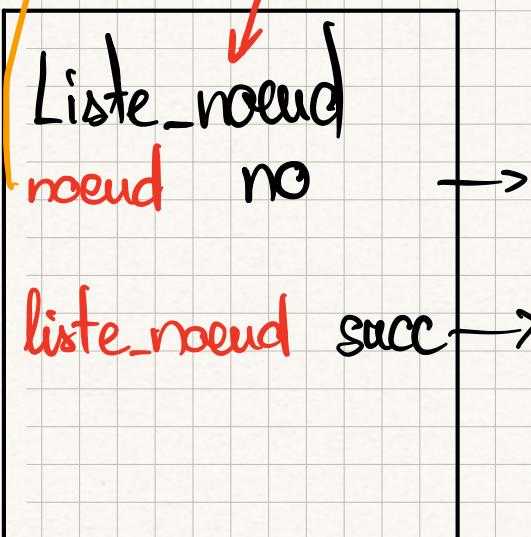
PCP pas à 100%

x test6 seg fault

Struct



Dossier ou fichier



Liste pour représenter les enfants
d'un noeud
donc chaque enfant est un noeud
et il a un lien vers le suivant

Malloc() :

malloc alloue la mémoire d'une taille spécifique
il connaît pas le type du data

Les fonctions :

initArbre() : initialise un arbre en allouant la mémoire pour la racine
Cette fonction a pour but d'initialiser le 1er noeud qui est un dossier

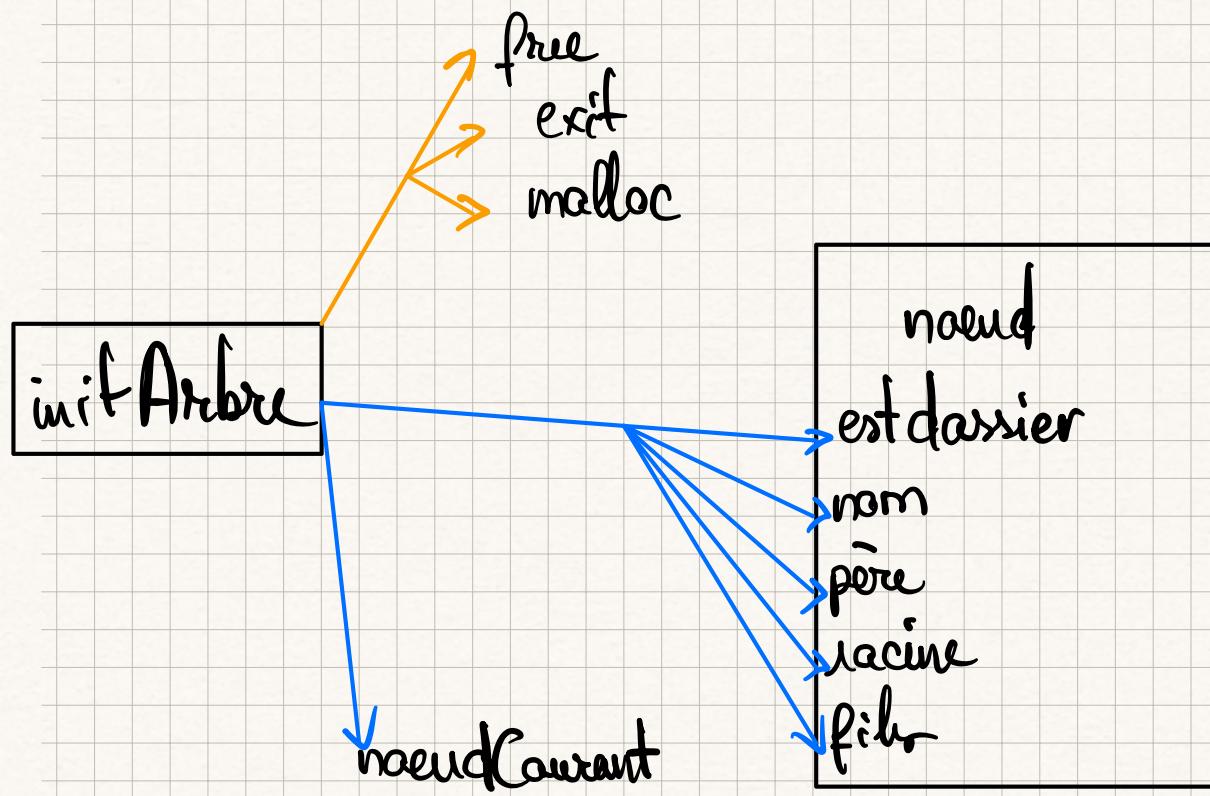
Son père est lui-même

Sa racine est lui-même

Il a pas de fils pour l'instant

Il a pas de nom

On se place nœuds même sur la racine
noeudCourant = racine;



noeud *GeeNoeud (const char *nom, noeud *père,
bool estDossier) :

Cette fonction crée un nouveau noeud avec
un nom, un pointeur vers son père et un bool
pour indiquer que ça fait un dossier

→ Ça alloue la mémoire pour le nouveau noeud

→ n->estDossier : utilise le champ est-Dossier

→ On donne le n->père = père donc

le père de n sera père dans l'argument

n->racine = TrouverRacine(n);

→ appelle à la racine pour pointer le champ racine

n->fils = Null, il a sans fils au départ

if (stolen(nom) < 100 && nom[0] != "10") {

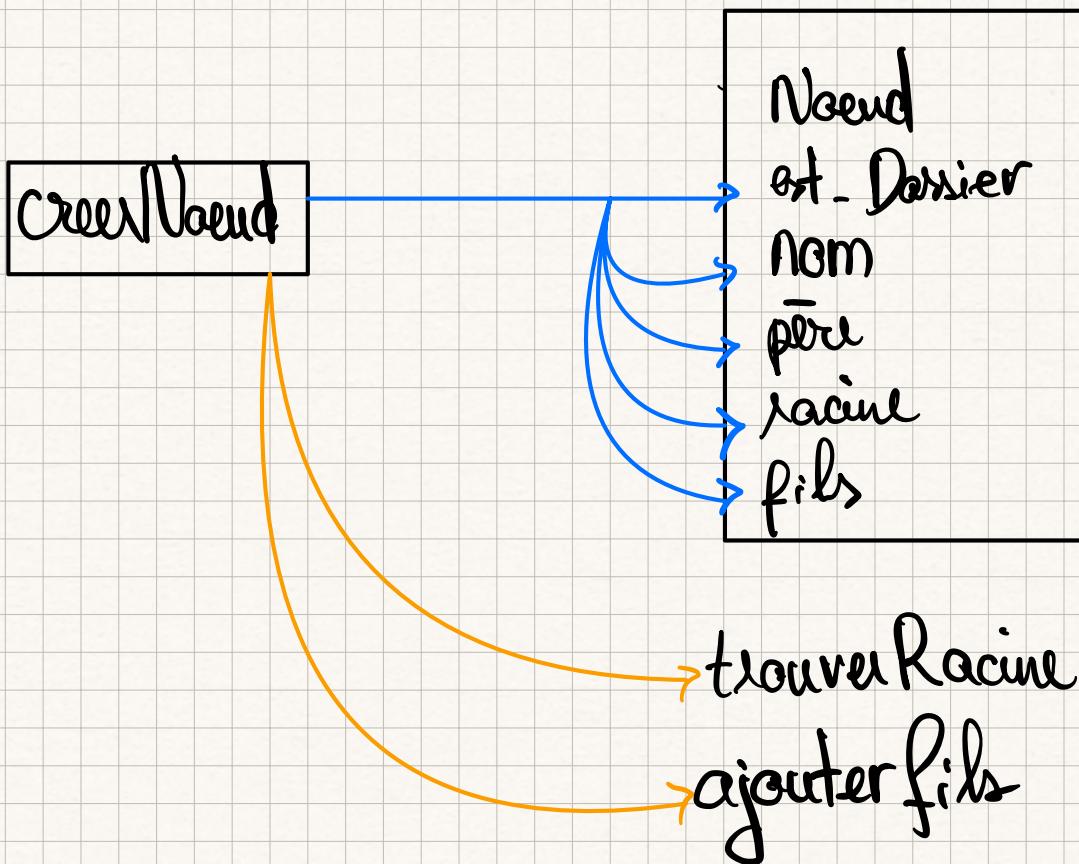
Vérifie que la longueur est < 100 et que ça
ne commence pas par un vide

ajouterFils(père, n) : ajoute le nouveau noeud
comme fils de son parent

const char * nom: la fonction va pas modifier les données aux quelles le pointeur "nom" pointe.

char * nom : C'est typique car une chaîne de caractère est généralement représenté comme un tableau et tableau est essentiellement un pointeur vers le premier élément du tableau

=> C'est un pointeur à une chaîne de caractère qu'on modifie par



bool ajouterFils(nœud *parent, nœud *n)

Cette fonction ajoute un nœud enfant à un nœud parent donné

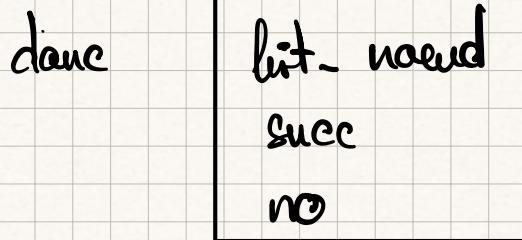
Elle renvoie true si tout est ok

→ if (!parent → est dossier || parent == Null) { return false; }

(Ca vérifie si le nœud parent est un dossier ou null)

→ liste-nœud *ln = initListeNœud(enfant);

(Ca crée une nouvelle instance de liste-nœud pour le nœud enfant)

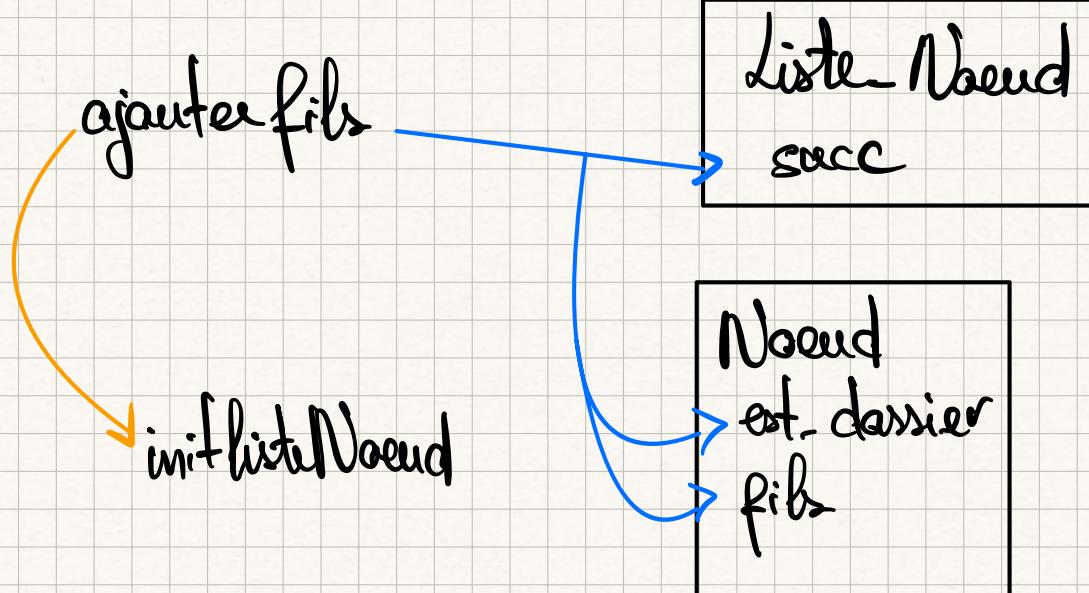


→ if (parent → fils == Null) { parent → fils = ln; }

vérifie si le parent a déjà un enfant, Si il a pas, on assigne le nouveau nœud enfant à la tête des enfants du parent

→ else {

Si il a déjà un fils, Il va parcourir la liste des enfants du parent jusqu'à ce qu'il atteigne le dernier, ca ajoute à la fin de la liste



init Liste_Noeud (noeud *n):

Elle initialise une nouvelle liste_noeud pour un noeud donné

→ On alloue d'abord de la mémoire pour liste

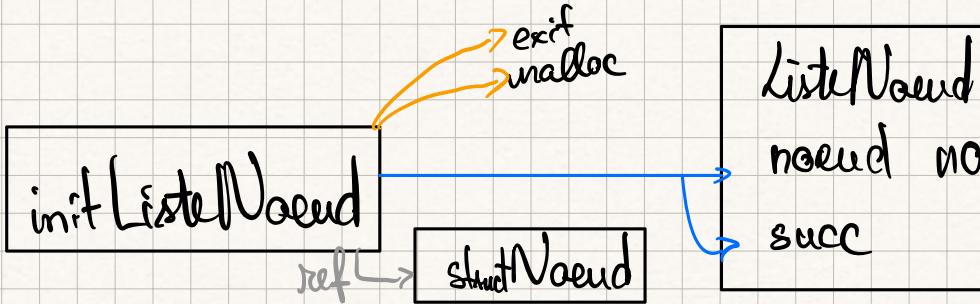
if (liste == Null || n == Null)

On vérifie si l'allocation a réussi et si le noeud est pas Null.

liste → no = no;

liste → succ = Null;

→ Ça assigne le noeud à l'attribut "no" de la liste et initialise liste → succ à Null



Trouver Racine (nœud n)

Trouve la racine de l'arbre à partir d'un nœud donné

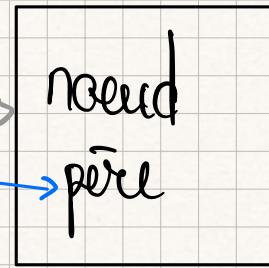
→ if ($n \rightarrow \text{père} = n$) {

Si le nœud en argument est égal à lui-même, donc c'est la racine. On retourne lui-même

else {

→ Si le nœud n'est pas sa propre racine, la fonction s'appelle récursivement en passant le père du nœud courant. On renvoie

trouverRacine



noeud TrouverNoeud (const char * path)

Cette fonction recherche un noeud spécifique à partir d'un chemin donné

typ int capacity = 10;

char ** pathfolders = (char **) malloc(capacity * sizeof(char *));

char * : un pointeur vers un char et peut être utilisé pour stocker une chaîne de caractère (tableau char)

char ** : est un pointeur vers un pointeur de "char" pour stocker un tableau de chaîne de caractère (un tableau de "char *") → Contient plusieurs chaînes de caractère

Dans un tab de pointeur vers des "char"

Chaque pointeur de char peut pointer vers une chaîne de caractère distincte

→ char *pathCopy → Copier le chemin d'accès

Bloc stetok :

Ici, on a la fonction est divisée en chaîne de caractère en plusieurs morceaux, ici token

Ici le délimiteur est "/"

Dans /home/user/documents ⇒ home, user, doc

On les stocke ensuite dans pathfolders
while boucle, ça continue jusqu'à ce que
stok() retourne Null

Ici on a une capacité à 10 donc 10 éléments
si ça dépasse la capacité, on double la capacité

On redouble sa mémoire en utilisant realloc()

C'est ça marche :

les anciennes données stockées dans le tab pathFolders
sont copiées dans la nouvelle zone de mémoire

Puis → if (path[0] == '\0')

Ca vérifie le 1er caractère du chemin

Si ça commence par '/' → chemin absolu
⇒ La recherche commence à partir de la racine de l'arbre

Sinon, c'est un chemin relatif ou ça commence à partir le nœud courant

Bloc While (pathFolders[i] != NULL)

Ca parcourt tous les tokens

if (strcmp (pathFolders[i], "..") == 0) {

Cela signifie que le nœud donné en argument
doit être mis à jour pour remonter au nœud
parent. Avant tout, on vérifie quand même si le
nœud actuel n'est pas déjà la racine

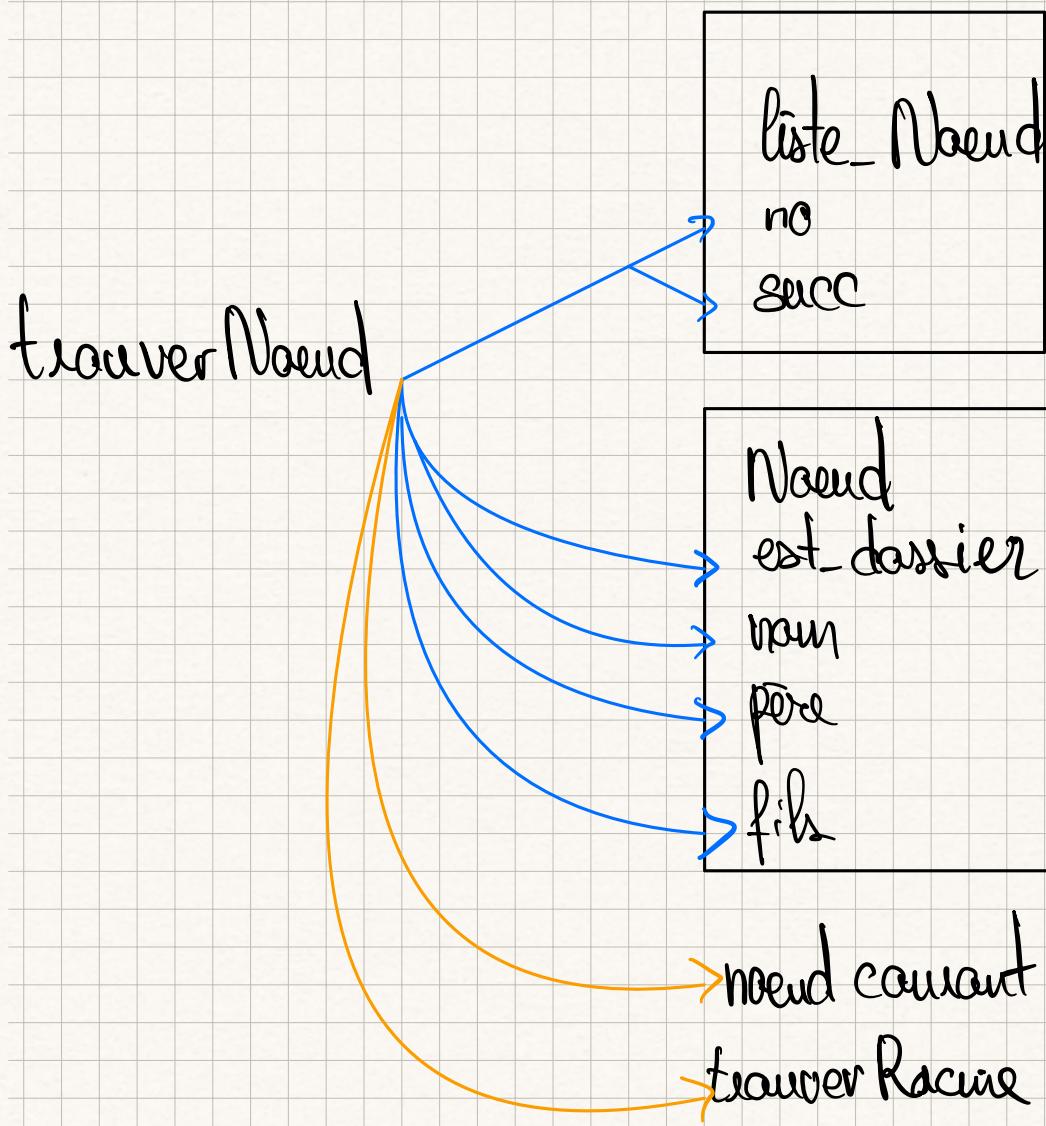
de cas : ".":

le nœud actuel reste le même

Si on

Si le token correspond au nom d'un nœud dans la liste des fils du nœud actuel, le nœud est mis à jour avec le nœud correspondant

On les libère à la fin. On libère donc chaque token



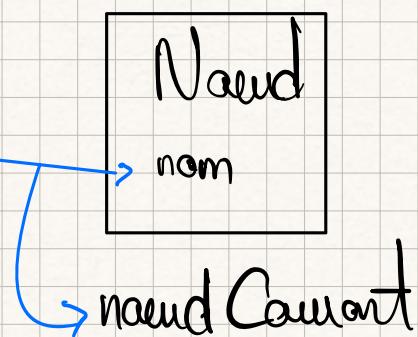
ChangerDossier (noeud *n)

Elle change de répertoire /noeud courant à celui passé en argument 'n'

Si n est Null \rightarrow failed

Si non on met à jour le noeudCourant à n

changerDossier



BougerNoeud (noeud *n , noeud *nouveauPere)

Tout d'abord, on vérifie avec une série de condition

Si une des conditions est vraie, déplacement impossible

On prend la liste des fils du noeud parent

Si la liste est null \rightarrow déplacer impossible

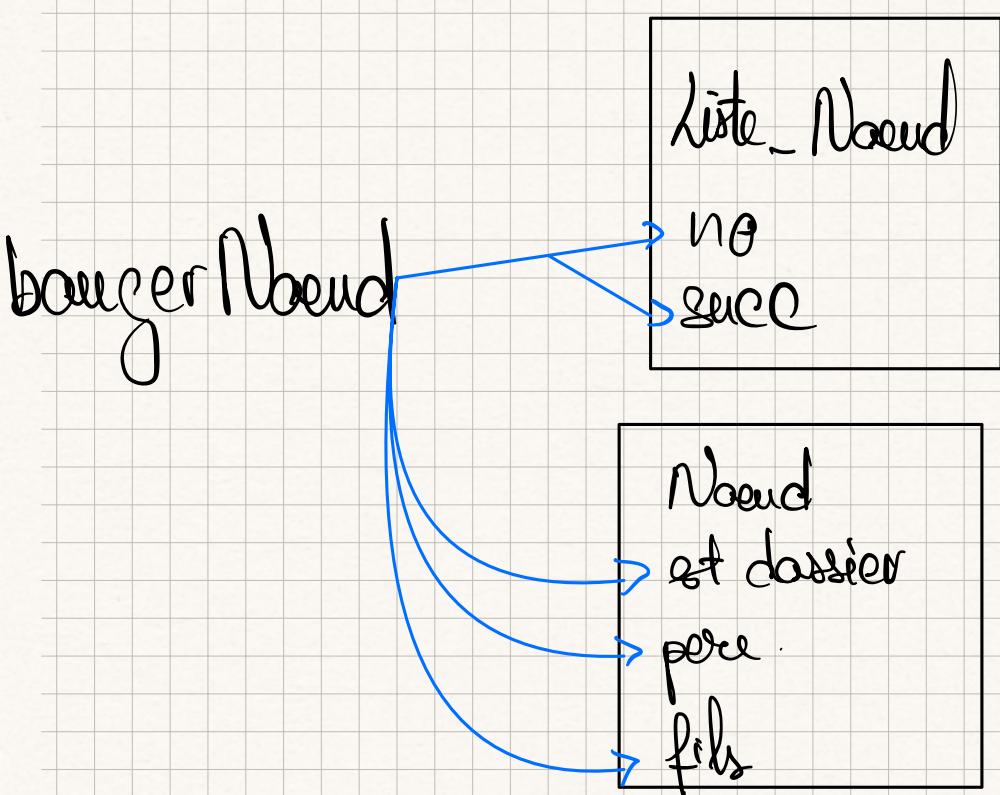
On obtient la liste des fils du parent actuel du noeud à déplacer

Si la liste est null → déplacer impossible

Puis on retire n de la liste de son parent actuel
Puis boucle while {} trouve le dernier élément de la liste du nouveau parent

On alloue de la mémoire pour un nouveau nœud dans la liste des fils du nouveau parent

Puis on attache nœud n au nouveau parent
On met à jour son parent



CopierNoeud (noeud *n, noeud *nouveau)

On vérifie plusieurs conditions avant de procéder

Si n est un dossier, la fonction parcourt ses fils avec celle-ci

On récupère ses fils dans la liste

Pour chaque fils, elle crée une copie du noeud en utilisant la fonction creerNoeud

Si c'est un dossier, appel récursivement copierNoeud

Copier et Gérer

effectuer la copie et création d'un nœud

Tant d'abord on vérifie si n existe

On alloue ensuite la mémoire pour copier le chemin
puis on copie pour modifier sans affecter le chemin

On cherche d'abord le chemin dans TrouverNœud, si
ça existe pas on peut faire la copie

Elle cherche le dernier caractère '/' dans copiepath
par exemple cp /Rep1/Rep11 Rep12
ici lastSlash est Null

Etant donné que lastSlash n'est pas Null

⇒ '/' a été trouvé dans copiepath

⇒ un chemin de dossier ou fichier a été spécifié

⇒ la fonction cherche le nœud correspondant au dossier parent

Si le slash est NULL, le nouveau nœud doit être
créé dans dossier courant

On remplace le dernier slash = '10'

⇒ Ca crée 2 chaîne de caractère :

chemin parent et chemin fichier au dossier

Donc ça permet de transformer

/home/user/file → /home/user

On restitue la slash → on doit représenter le chemin complet

Si Il n'y pas de Slash ' / ', chemin contient pas de répertoires ⇒ Dans ce cas parent = NoeudCourant

Création nouveau noeud

si lastSlash est Null
⇒ c'est donc le chemin cp /Rep1

Même nom que l'origine

Sinon

Le nom se trouve dernier slash
dans nom=lastSlash + 1

Supprimer (noeud n)

On vérifie d'abord si $n == \text{noeudCourant}$

On vérifie ensuite si $n == \text{Null}$: False

On parcourt ensuite pour supprimer tous ses enfants

On vérifie ensuite si n.y père est pas lui-même. Si c'est égal, donc on est à la racine