Programmation Impérative 2 - L2 Informatique

TP4 - Arbres

Exercice I -

Implémenter une structure d'arbre binaire pouvant stocker un caractère sur chaque nœud. On implémentera également les fonctions suivantes:

- `int est_vide()`
- `arbre creer_arbre_vide()`
- `arbre creer_noeud(char c)`
- `arbre creer_arbre(char c,arbre filsgauche,arbre filsdroit)`
- `arbre inserer_fils_gauche(arbre a,arbre filsgauche)`
- `arbre inserer_fils_droit(arbre a,arbre filsdroit)`
- `void afficher_arbre(arbre a)`,
- `void affiche_arbre_graphique(arbre a)`.

La fonction `inserer_fils_gauche` insère `filsgauche` comme fils gauche de `a`, la fonction `inserer_fils_droit` insere `filsdroit` comme fils gauche de `a`. L'arbre vide pourra être défini par `NULL`. La fonction `afficher_arbre` effectue un affichage sur la sortie du terminal de type parcours en profondeur, la fonction `afficher_arbre_graphique` affiche l'arbre sur une sortie graphique à l'aide de la fonction MLV.

Exercice II - On souhaite maintenant utiliser cette structure d'arbre pour effectuer des évaluations d'expressions arithmétiques codées en notation postfixée (ou polonaise) inversée. Par exemple, *3 * (4 + 7)* peut s'écrire en notation postfixée inversée sous la forme ** 3 + 4 7*. Autre exemple: le calcul *((1 + 2) * 4) + 3* pourra se représenter, soit sous la forme *+ 3 * 4 + 2 1*, soit sous la forme *+ * + 1 2 4 3* (on utilise donc une représentation inversée par rapport au tp précédent). On supposera que les chiffres sont uniquement entre 0 et 9.

On commence par écrire une fonction qui crée un arbre binaire représentant une expression arithmétique codée sous forme postfixée inversée: `arbre creer_arbre_expression(char * expression, int * position)`.
L'algorithme est le suivant, on parcours la chaine `expression` de gauche à droite, si le caractère à la position courante:

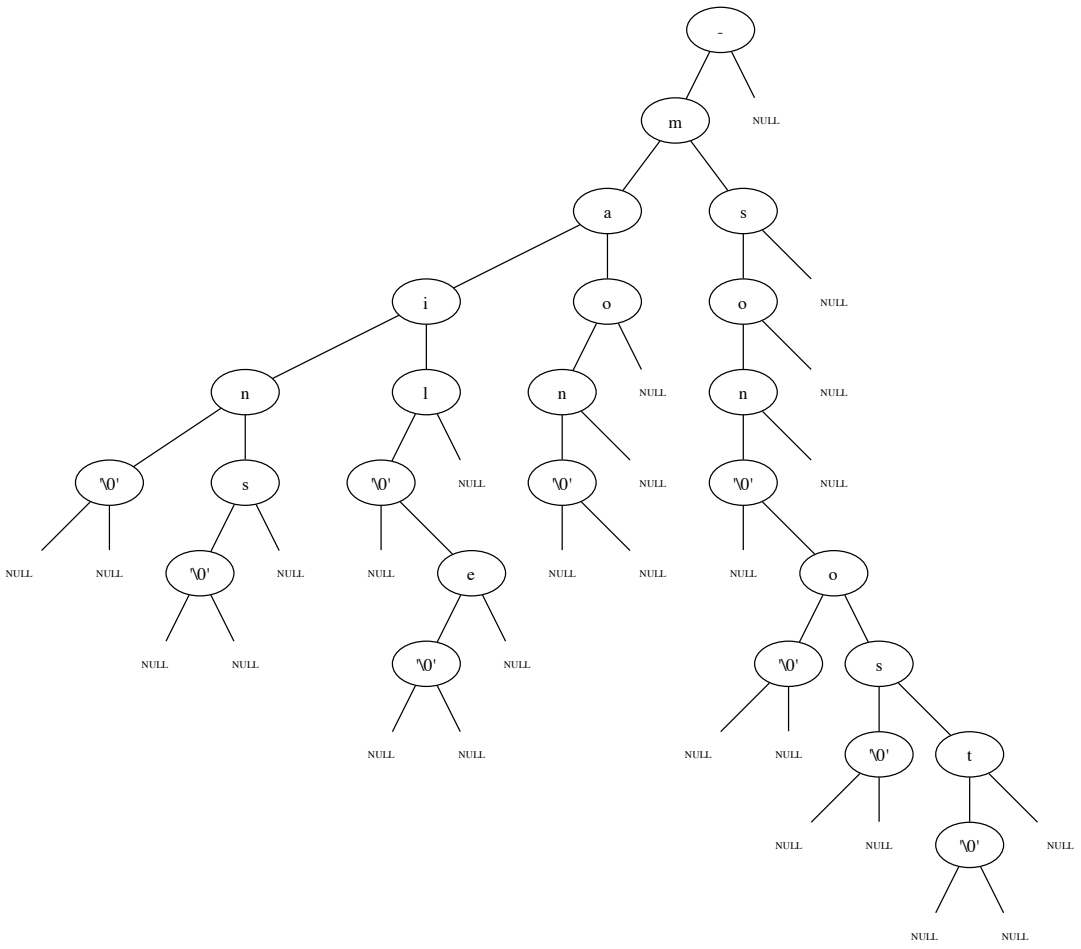
- est un opérateur, alors il faut créer un arbre contenant ce caractère pour racine, augmenter `*position` de 1, puis appeler la fonction récursivement pour créer le sous-arbre gauche, puis le sous-arbre droit. L'argument `position` doit contenir la position courante dans la chaîne, il doit donc être mis à jour exactement une fois dans chaque appel de `creer_arbre_expression`. La fonction renvoie ensuite le sous-arbre créé.
- est un chiffre, alors il faut créer un noeud pour ce chiffre, le noeud créé représentera alors une feuille de l'arbre, augmenter `*position` de 1, puis renvoyer l'arbre correspondant à ce nœud.

1. Ecrire la fonction `creer_arbre_expression`.
2. Tester cette fonction à l'aide d'un programme permettant d'entrer des expressions sur la ligne de commande puis affichant les arbres;
3. Ecrire une fonction permettant d'évaluer l'expression arithmétique: `int eval(arbre a)`. Effectuez plusieurs tests à l'aide du programme précédent.

Exercice III - On souhaite implanter un dictionnaire de mots sous forme d'arbre binaire. Chaque nœud d'un arbre représente un caractère et les fils gauche et droit s'interprètent de la manière suivante:

- le fils gauche désigne des lettres qui peuvent être mises à la suite du caractère courant pour former des mots.
- le fils droit désigne des lettres qui peuvent remplacer le caractère du nœud courant.

La figure suivante donne un exemple d'arbre représentant les mots *main, mais, mal, male, mon, son, sono, sons, sont.* La fin d'un mot est indiquée par le symbole de fin de chaîne `'\0'`.



.

Figure 1: Exemple d'arbre représentant un dictionnaire, la racine de l'arbre est définie à l'aide d'un symbole spécial.

En utilisant la structure d'abre binaire créée précédemment, proposée une implantation de dictionnaire. Vous implémenterez les fonctions suivantes:

- `int existe(dictionnaire d, char * mot)` qui renvoie 1 si la chaine `mot` existe dans le dictionnaire et 0 sinon.

- `void affiche_dico(dictionnaire d)` qui affiche l'ensemble des mots présents dans le dictionnaire (un par ligne). Pour cette fonction, vous pouvez vous aider d'une fonction utilisant un tableau auxiliaire et un indice de position pour stocker la chaine construite à chaque étape.
- `int insere_dico(dictionnaire d,char * mot)` qui insere un nouveau mot dans le dictionnaire et qui renvoie 1 si tout s'est bien passé et 0 sinon. Pour vous aider, vous pouvez définir une fonction auxiliaire utilisant un (ou plusieurs arguments) supplémentaire comme un indice indiquant la position courante du caractère traité dans la chaine.
- Ecrire un petit programme qui prend des mots sur la ligne de commande et les range dans le dictionnaire.
- Ecrire une extension avec une interface graphique créée à partir de la librairie MLV.

Note: `dictionnaire` pouvant être défini à l'aide d'un `typedef` comme un `arbre`.

Note : les 3 derniers points précédents, ainsi que ce qui suit, sont en bonus.
Ensuite, une fois que tout ceci a été effectué, vous pouvez écrire des fonctions de sauvegarde et de récupération du dictionnaire en implémentant les fonctions suivantes:

- `void sauvegarde_dico(dictionnaire d, char * fichier)`: qui enregistre les mots du dictionnaire dans le fichier ayant pour nom `fichier`, un par ligne (vous pouvez vous aider du principe de la fonction d'affichage).
- `void recupere_dico(dictionnaire d,char * fichier)`: qui récupère les mots écrits dans `fichier` et les insère dans le dictionnaire (en utilisant la fonction d'insertion).

Pour ceux qui sont plus à l'aise:

- vous pouvez modifier la fonction d'insertion pour qu'elle fasse des insertions respectant l'ordre alphabétique.
- Vous pouvez implanter un dictionnaire à l'aide d'arbre n-aire (et non plus binaires). Pour cela, vous pouvez définir un nœud d'un arbre à l'aide d'un caractère `char`, d'un tableau de pointeurs sur des arbres représentant les fils et d'un entier représentant la taille du tableau - cette dernière pouvant être fixe dans un premier temps (la mettre à 26 par exemple, ou alors refaire des allocations si besoin). Il faut ensuite écrire toutes les fonctions précédentes avec cette nouvelle structure.
  Quels sont les avantages et inconvénients de chacune des 2 implantations ?