Travail de synthèse
Date de remise : 18 mai 2018 23h55
Ce travail peut être fait individuellement ou en équipe de 2

Conception et programmation orientée objet II
420-401-AL
Les Arbres

1 Objectifs

— Gestion de la mémoire
— Création d'objets
— Manipulation d'une structure de données en arbre
— Héritage
— Analyse de performance

2 Présentation du problème

L'objet de ce travail est l'implantation des fonctionalités de base sur une structure de données récursive : la structure en arbre, et plus particulièrement l'arbre lexicographique. Nous utilisons ici cette structure pour construire un dictionnaire. Nous voulons être capable d'y rechercher un mot de manière efficace. Par exemple, la figure 1 montre un arbre binaire de recherche correspondant aux mots *don*, *donc*, *dons*, *dont*, *faim*, *fais*, *fac*, *face* et *fou*, alors que la figure 2 montre l'arbre lexicographique pour les mêmes mots. On notera que dans un arbre lexicographique, les mots ne sont pas représentés par les nœuds, mais par les chemins depuis la racine.

Un arbre est un arbre lexicographique si les conditions suivantes sont vérifiées :
— à chaque nœud est associée une lettre
— chaque chemin de la racine vers une feuille correspond à un mot
— un préfixe commun à plusieurs mots de l'arbre n'apparaît qu'une fois dans l'arbre
— la racine est conventionnelle et ne porte aucune information significative
— les fils de la racine sont des nœuds qui portent les premières lettres des mots, triées par ordre alphabétique ; il en de même pour tous les niveaux inférieures

À des fins de simplification, seules les 26 lettres de l'alphabet, minuscules ou majuscules mais sans accents, sont permises.

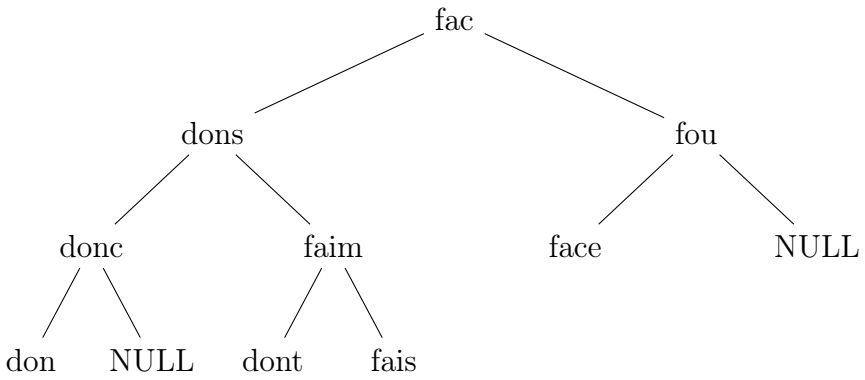

FIGURE 1 – Arbre binaire de recherche

3 Votre travail

Vous devez construire un arbre lexicographique pour une liste de mots et ensuite faire quelques comparaisons de performance avec une approche basée sur un arbre binaire de recherche.

1. Commencez d'abord par faire fonctionner et analyser le petit programme **C** en annexe. Il vous permettra de comprendre comment un arbre lexicographique est construit et parcouru. Vous pourrez réutiliser ce code si vous voulez, mais vous devrez apporter les modifications nécessaires pour le rendre pleinement orienté objet, de telle sorte que vous aurez une classe **Arbre** dans laquelle vous allez inclure les données et fonctions membres nécessaires décrites plus loin (voir quelques remarques à ce sujet dans le *main*).
2. Assurez-vous de ne laisser aucune trace du **C** dans votre programme : pas de *assert*, *printf* et *malloc*, utilisez des *string* au lieu de tableaux de *char* et le conteneur STL *stack* au lien de la pile maison. (5 points)
3. Créez une classe **Arbre** à partir de laquelle vous dériverez les classes **ArbreLex** et **ArbreBin**. Fournissez deux constructeurs pour chacune des deux classes filles : un vide et un prenant un vecteur de mots qui seront insérés dans l'arbre. (10 points)
4. Réécrivez les fonctions **insere_dans_dict** et **parcours_le_dict** comme méthodes. (5 points)
5. Écrivez la méthode **effaceMot** permettant d'effacer un mot passé en argument. (10 points)
6. Écrivez la méthode **bool estVide** pour savoir si un arbre est vide. (5 points)
7. Écrivez la méthode **void effaceArbre** pour effacer le cntenu d'un arbre. (5 points)
8. Écrire une méthode **Lecture** prenant en argument un nom de fichier et qui insère les mots du fichier dans l'arbre. On suppose un mot par ligne. (5 points)
9. Écrire une méthode **Sauvegarde** prenant en argument un nom de fichier et qui écrit tous les mots de l'arbre en ordre alphabétique. (5 points)

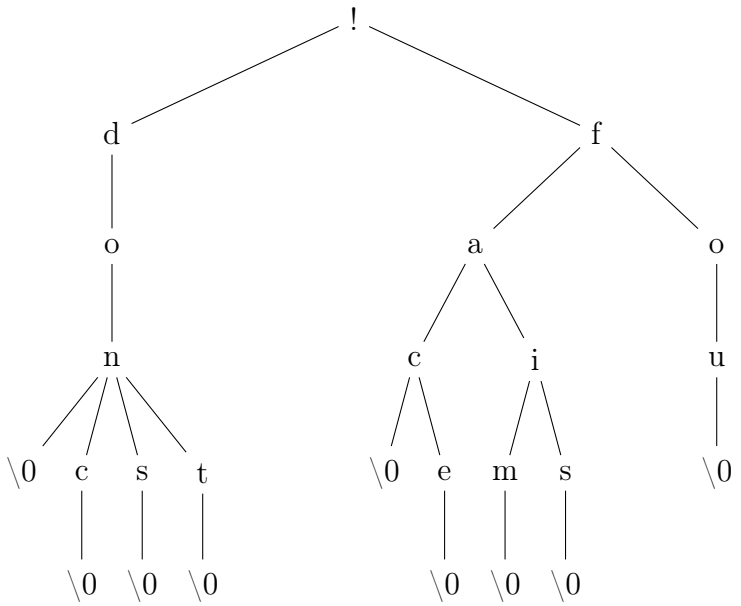

FIGURE 2 – Arbre lexicographique

10. Écrire une méthode **Appartient** pour savoir si un mot appartient à l'arbre. (5 points)
11. Écrire une méthode **AfficherMotsLongueur** qui affiche tous les mots de l'arbre ayant une longueur donnée. (5 points)
12. Écrire une méthode **AfficherMotsPrefixe** qui affiche tous les mots de l'arbre ayant un préfixe commun fourni en argument. (5 points)
13. Évaluez la complexité temporelle des méthodes **Appartient** , **AfficherMotsLongueur** et **AfficherMotsPrefixe**. Faites la comparaison entre les deux types d'arbres. (5 points)
14. Évaluez la complexité temporelle de vos fonction d'insertion. En déduire la complexité globale de construction des arbres lexicographique et binaire. Proposer (sans implanter) des variantes pour la représentation afin de diminuer la complexité. (15 points)

4 Barême de correction

Qualité du code	**/15**
commentaires (commentaires utiles et clairs)	/5
clareté du code (nom des variables, indentation...)	/10
Fonctionnement et analyse de complexité	**/85**

5 Remise

La remise sera faite dans l'environnement MOODLE, dans la section prévue à cet effet, avant le 18 mai 2018 à 23h55. Remettez votre analyse de complexité (questions 13 et 14) dans un fichier à part.

Annexe

```c
#include<stdio.h>
#include<assert.h>
#include<stdlib.h>
typedef struct noeud {
    char lettre;
    struct noeud *fils, *frere;
} UN_NOEUD, *PTRdeNOEUD;

static struct {
    char lettres[80];
    int n;
} pile;

PTRdeNOEUD noeud(char lettre, PTRdeNOEUD fils, PTRdeNOEUD frere) {
    PTRdeNOEUD p = (PTRdeNOEUD)(malloc(sizeof(UN_NOEUD)));
    assert(p != NULL);
    p->lettre = lettre;
    p->fils = fils;
    p->frere = frere;
    return p;
}
int insere_dans_dict(const char mot[], PTRdeNOEUD ascendant) {
    PTRdeNOEUD pr, pc;
    int i;
    for(i = 0; ; i++) {
        pr = NULL;
        pc = ascendant->fils;
        while (pc != NULL && pc->lettre < mot[i]) {
            pr = pc;
            pc = pc->frere;
        }
        if (pc != NULL && pc->lettre == mot[i]) {
            if (mot[i] == '\0')
                return 1; /* le mot existait */
            ascendant = pc;
        }
        else {
            pc = noeud(mot[i], NULL, pc);
            if (pr != NULL)
                pr->frere = pc;
            else
                ascendant->fils = pc;
            while (mot[i] != '\0') {
                pc->fils = noeud(mot[++i], NULL, NULL);
                pc = pc->fils;
            }
            return 0; /* le mot est nouveau */
        }
    }
```

```
}

void parcours_le_dict(PTRdeNOEUD arbre) {
    PTRdeNOEUD p;
    pile.lettres[pile.n++] = arbre->lettre;
    if (arbre->lettre == '\0')
        printf("%s\n", pile.lettres + 1);
    else
        for (p = arbre->fils; p != NULL; p = p->frere)
            parcours_le_dict(p);
    pile.n--;
}

int main(int argc, char *argv[]) {
    //ArbreLex a('!'); ou ArbreBin a(NULL); en C++
    PTRdeNOEUD racine = noeud('!', NULL, NULL);
    insere_dans_dict("face", racine);
    insere_dans_dict("fac", racine);
    // a.insere_dans_dict("donc"); en C++
    insere_dans_dict("donc", racine);
    insere_dans_dict("dons", racine);
    insere_dans_dict("fou", racine);
    pile.n = 0;
    //a.parcours_le_dict(); en C++
    parcours_le_dict(racine);
    insere_dans_dict("dont", racine);
    insere_dans_dict("faim", racine);
    insere_dans_dict("fais", racine);
    insere_dans_dict("don", racine);
    pile.n = 0;
    parcours_le_dict(racine);
}
```