Références croisées sur un ensemble de fichier C++

1 Spécifications

Le but du programme est de permettre de retrouver les emplacements d'identifcateurs dans une collection de fichiers. L'emplacement de chaque occurence sera désigné par le nom du fichier ainsi que le numero de ligne où elle a été rencontrée. On ne considèrera pas la position sur la ligne.

Dans le cas où un identifacteur apparaitrait plusieurs fois sur une même ligne, nous avons pris la décision d'afficher la ligne conserné une fois par occurence. Etant donné qu'un même identificateur ne devrait pas être présent de trop nombreuses fois sur une même ligne, nous avons jugé que cette présentation ne nuirait pas à la lisibilité de notre programme.

1.1 Définition du vocabulaire

- **ligne** : On considèrera une ligne comme une suite de caractère terminé par un retour chariot
- **Identificateur** : C'est un mot sensible à la casse composé uniquement de caractères alphanumériques et du caractère '_'. Les identificateurs présents dans des commentaires ou des chaines littérales ne seront pas pris en compte.
- **Déliminateur** : C'est un unique caractère qui représente une séparation entre deux mots
- **Référence croisée** : Une référence croisée est une occurence d'un identificateur

1.2 Spécifications des options

$$tp\_stl\ [-e]\ [-kfichier\_mot\_clef]\ [nomfichier]+$$

-e : Permet d'inverser le comportement par défaut du programme. Elle exclue de la référence croisée tous les mots clefs
-k fichier_mot_clef : Permet de spécifier au programme une liste de mot clef à rechercher pour la référence croisée
nomfichier : Chemin d'un fichier où chercher les références des identifacateurs

1.3 Identification des classes

1.3.1 Structure pour stocker les mots clefs

Besoins utilisateurs : Les mots clefs sont stockés dans un fichier. Pour en faciliter l'accès dans le programme nous souhaitons les charger en mémoire. Ce

pose alors la question du choix de la structure de donnée pour les contenir ?

Nous avons besoins d'une structure permettant de nous dire facilement si un mot clef est présent ou non dans la structure de données. Un mot clef ne peut être présent qu'une seule fois dans la structure, car les doublons n'ont pas de sens dans ce problème. De plus l'ordre n'a que peu d'importance.

Nous avons donc besoin uniquement de connaitre rapidement la présence ou non d'un mot clef.

Analyse d'une liste : Comme nous l'avons vu dans nos besoins, seul l'accès en lecture nous intérèsse. Une liste est une structure plate, c'est à dire qu'il n'y a qu'une direction possible. Dans le pire des cas il nous faut donc parcourir tous les éléments de la liste pour savoir si le mot clef est présent. La complexité d'une telle structure pour la recherche est donc linéaire (O(n)).

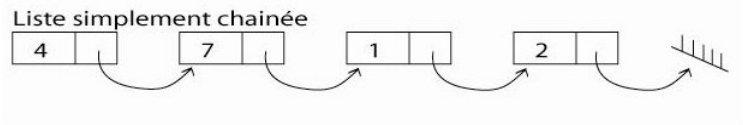

FIGURE 1 – Liste chainée

Analyse d'un arbre : Un arbre est une structure possédant plusieurs directions. De plus si l'on utilise un arbre binaire le choix de la direction est déterminé par le noeud courant et la valeur que l'on cherche.

Dans le pire des cas, si l'arbre n'est pas isométrique il faut parcourir tous les noeuds pour savoir si la clef est présente ou non. Par contre dans les cas moyen avec un arbre équilibré la recherche se fait en log(n).

Choix de la structure : Une structure en arbre nous semble plus adapté car nous souhaitons favoriser la recherche. Dans les cas moyens la liste reste en compléxité linéaire tandis qu'un arbre est de compléxité logarithmique.

Nous avons donc décidé de stocker la liste des mots clefs dans une stucture sous forme d'arbre.

1.4 Structure pour stocker les identifiants

Besoins : Chaque identificateur présents dans la collection de fichier(s) doit être stocké en mémoire avec la liste de ses occurences. Un identificateur a donc une clef (lui même) et une valeur (sa liste d'occurence). Nous souhaitons optimiser l'insertion et la lecture des valeurs des identificateurs. On se pose alors la question du choix de la stucture de donnée a adopter.

Choix d'un arbre rouge et noir :

choix d'une table de hash :

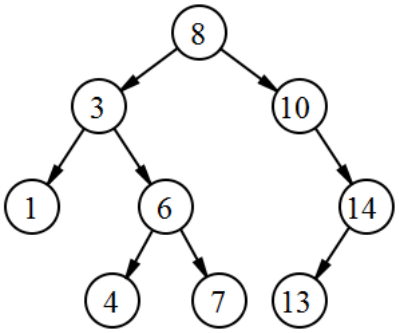

FIGURE 2 – Arbre binaire

1.5 Plan des tests fonctionnels