



**People’s Democratic Republic of Algeria**

**Ministry of Higher Education and Scientific Research**

**University of Algiers 1 Benyoucef BENKHEDDA**

**Faculty of Science**

**Department of Computer Science**

**Specialization: Network and Embedded Systems**

**1st year master**

(Teacher: Dr. **Abbas**)

**Raport**

**Theme**

|  |
| --- |
| Automated Backup and Restoration with Bash and C |

**By:**

Hicham SAIDI

Rafik BOUAICHA

2024/2025

# Used Libraries:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include "backup.h”

#include "option.h"

## Backup Library:

Contain all the backup and restoration functions, with other functions that are used by them:

* **Commend executer:**

Take a bash command and execute it if it was structured right, and return an error message if not.

void execute\_command(const char\* command);

**x**

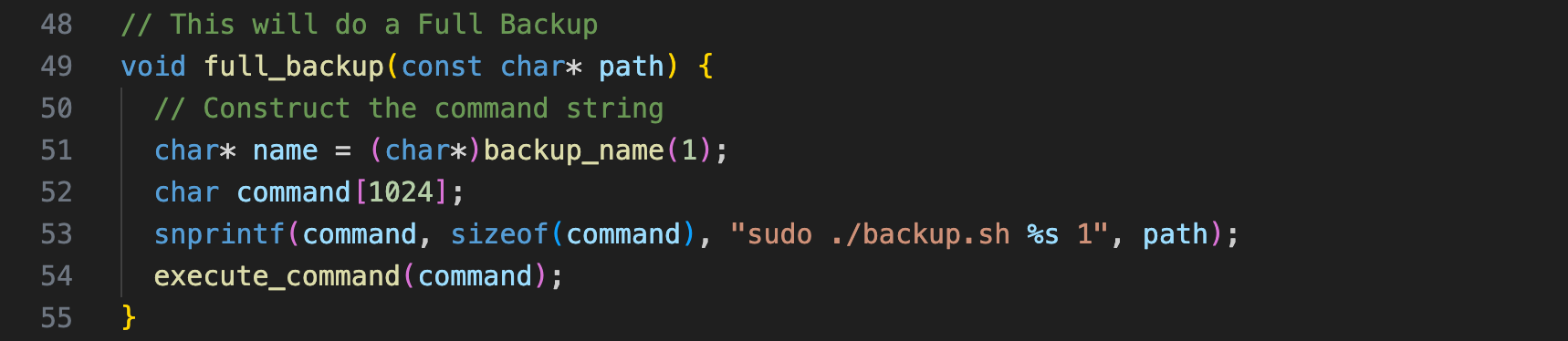
* **Bash launcher:**

Give to backup.sh the permission to be executed by running this command “chmod +x ./backup.sh”, like that we can use it in the command line and give it some parameters that are explain in the bash section of this report.

void bash\_lancher(const char\* command);

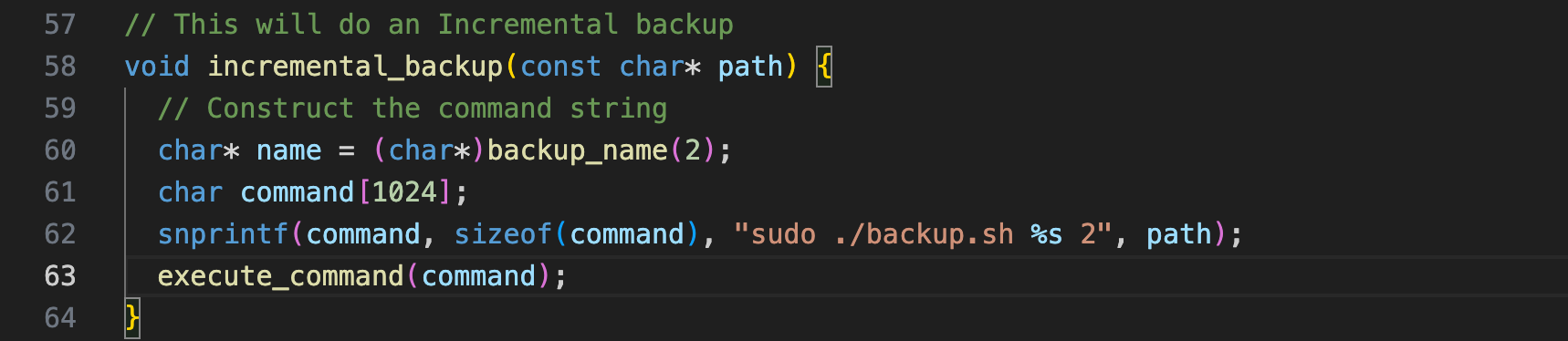
* **Full back up:**

The full\_backup function do a full back up, it accepts one parameter, path, which is a pointer to a character’s list that represent the path to the database or directory need to be backed up.



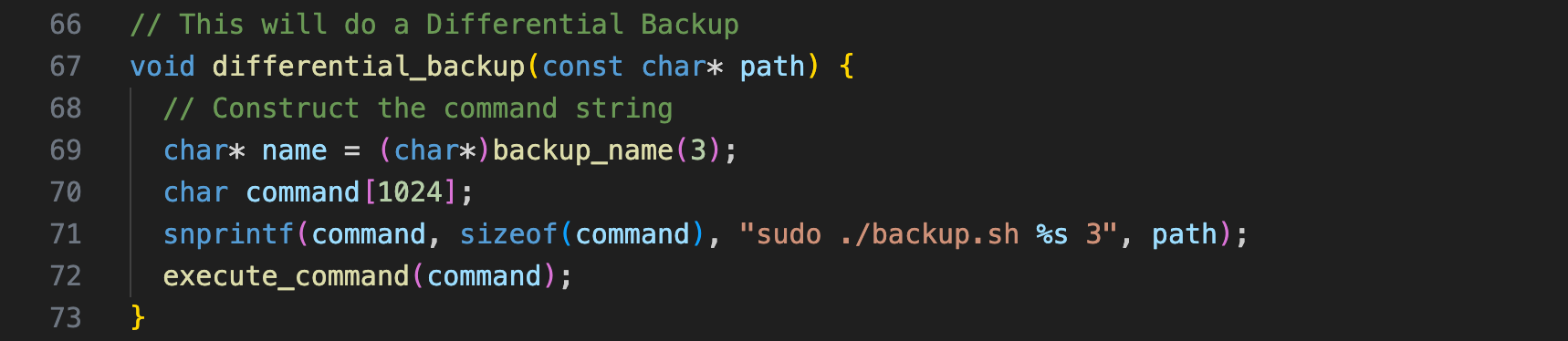
* **Incremental back up:**

The incremental\_backup function do an incremental back up, it accept one parameter, path, which is a pointer to a characters list that represent the path to the database or directory need to be backed up.



* **Differential back up:**

The differentail\_backup function does a differential back up, it accepts one parameter, path, which is a pointer to a characters list that represent the path to the database or directory need to be backed up.

****

## Option Library:

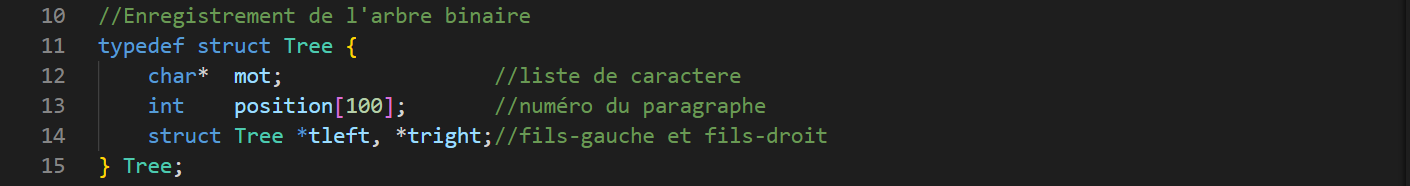
Contain all the functions that allow the user to control what to do a backup, restoration, exiting the program:

|  |  |
| --- | --- |
| void choose\_option(); | Allow the user to choose what option he need by selecting a number from 0 to 2:   1. For exiting 2. Back up 3. Restoration |
| char\* choose\_path(); | Take an input from the user and return a pointer of a list of characters that represent the path: |
| void choose\_backup(const char\* path); | Allow the user to choose backup type to use, change the path, or exiting by selecting a number from 0 to 4:   1. For exiting. 2. Full back up. 3. Incremental back up. 4. Differential back up. 5. Change path. |
| void choose\_restoration(const char\* path); |  |

1. **Création de la structure :**

Elle est composée de 4 champs :

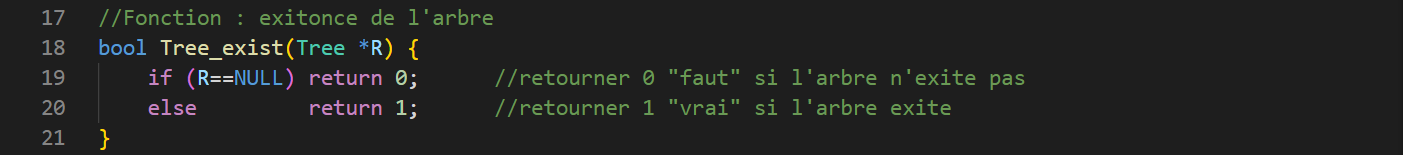
* Le champ « mot »  : liste de caractères.
* Le champ « position »: c’est-à-dire l’index.
* Le champ « tleft »  : le fils gauche du nœud.
* Le champ « tright »  : le fils droit du nœud.

****

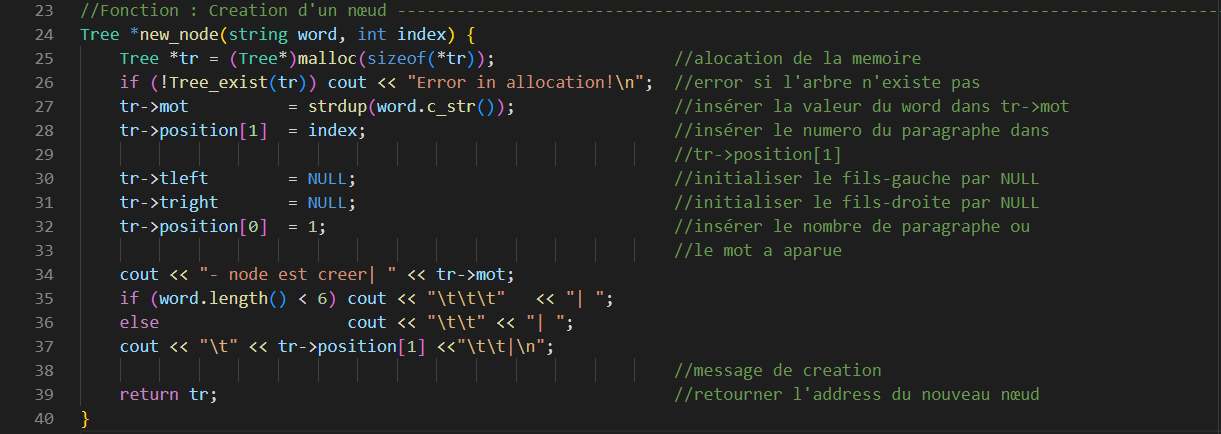
1. **Les fonctions créer et utiliser :**

**Supposition :**

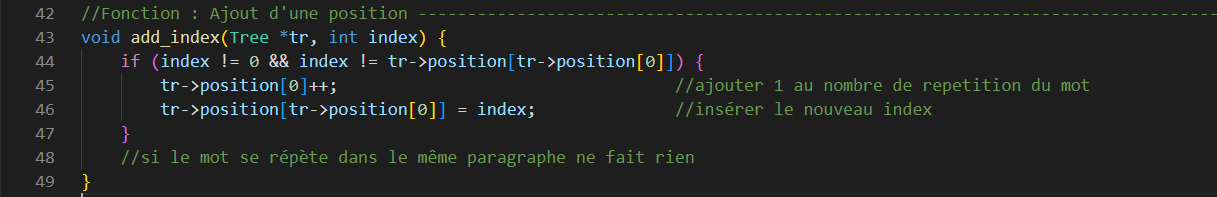
* Le fichier texte « .txt ».
* Chaque phrase se termine par un point.
* Ignorer (./,/ ?/ !) lors de l’insertion du mot dans l’arbre.
  1. **Existence d’un arbre :**
* **Fonction**.
* **Les entrés :** L’adresse de la racine de l’arbre (tr).
* **Les sorties :** boolien.
* **Le fonctionnement :** retourner (0 «faux»/ 1 «vrai») si l’arbre (n’existe pas/existe).



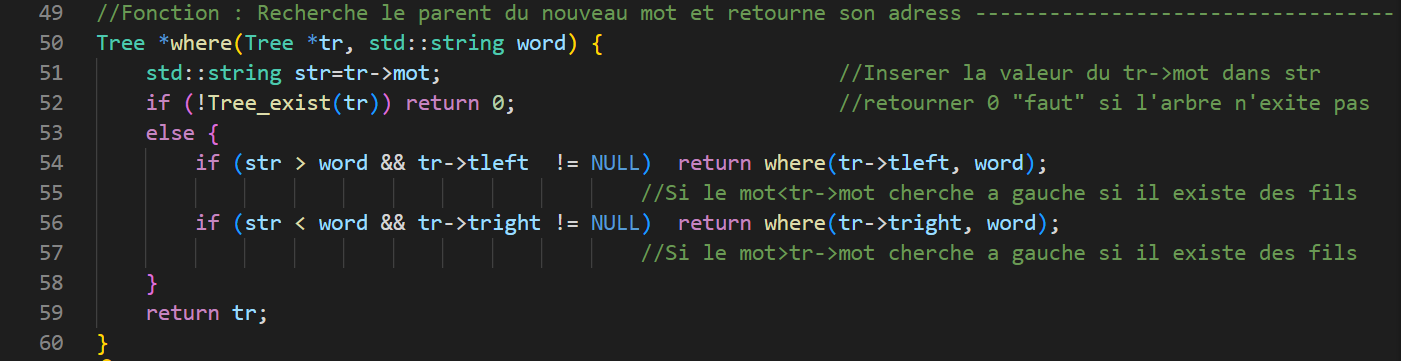
* 1. **Création d’un nouveau nœud (ajout d’un mot) :**
* **Fonction**.
* **Les entrés :** le mot (std::string word), le index (int index).
* **Les sorties :** retourner l’adresse du nouveau nœud criée.
* **Le fonctionnement :** créer un nouveau nœud contient le mot (word) et affiche un message contient le mot et sa première apparition,



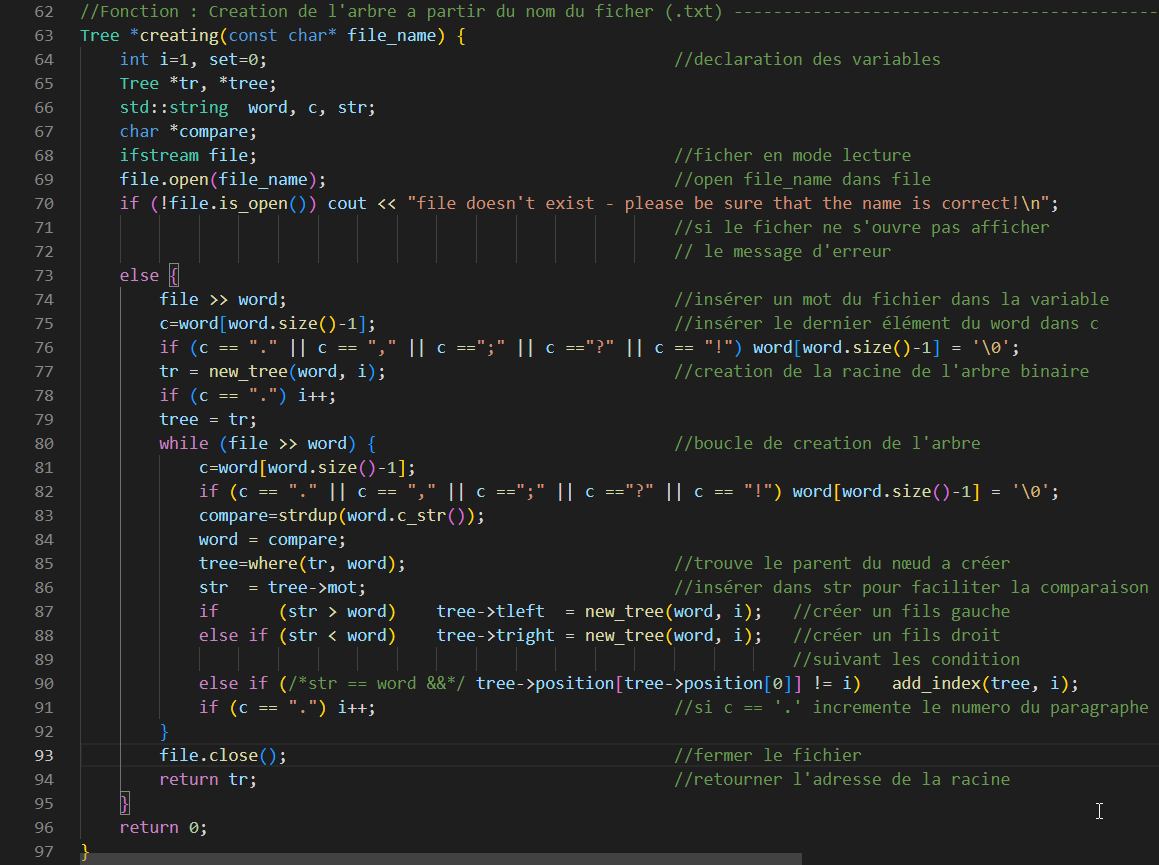
* 1. **Ajout d’une position :**
* **Procédure.**
* **Les entrés :** L’adresse du nœud (Tree \*tr), le nouveau index (int index).
* **Les sorties : /.**
* **Le fonctionnement :** Incrémente le nombre de fois d’apparition du mot dans les différents paragraphes du texte (pas dans le même paragraphe).



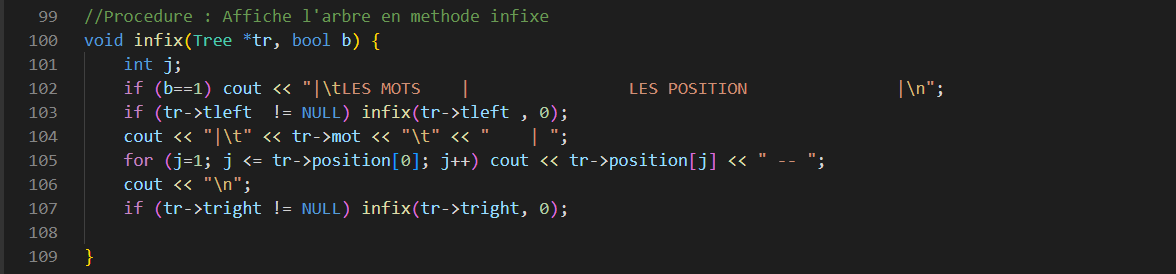
* 1. **Recherche de l’adresse du mot ou le parent du mot :**
* **Fonction.**
* **Les entrés :** L’adresse de la racine de l’arbre (Tree \*tr), le mot (std::string word).
* **Les sorties :** adresse d’un nœud.
* **Le fonctionnement :** retourne l’adresse du nœud ou le mot se trouve s’il existe sinon i il retourne l’adresse du nœud qui doit être le parent du mot donner s’il n’existe pas.



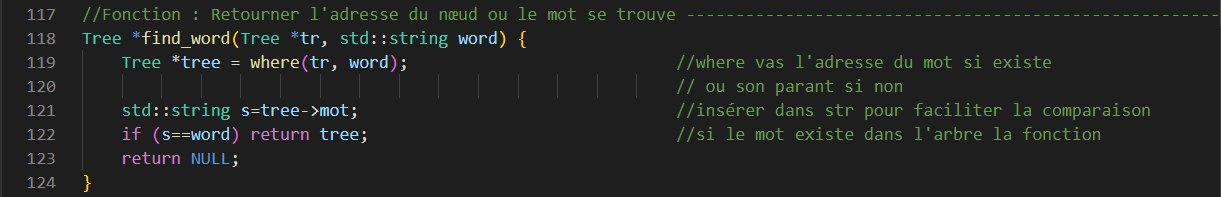
* 1. **Création de l’arbre binaire de recherche :**
* **Fonction.**
* **Les entrés :** le nom du fichier
* **Les sorties :** l’adresse de la racine de l’arbre binaire de la recherche.
* **Le fonctionnement :** ouvrir le fichier « txt.txt » et créer l’arbre binaire de rechercheà partir de ces mots en utilisant les fonctions précédentes.



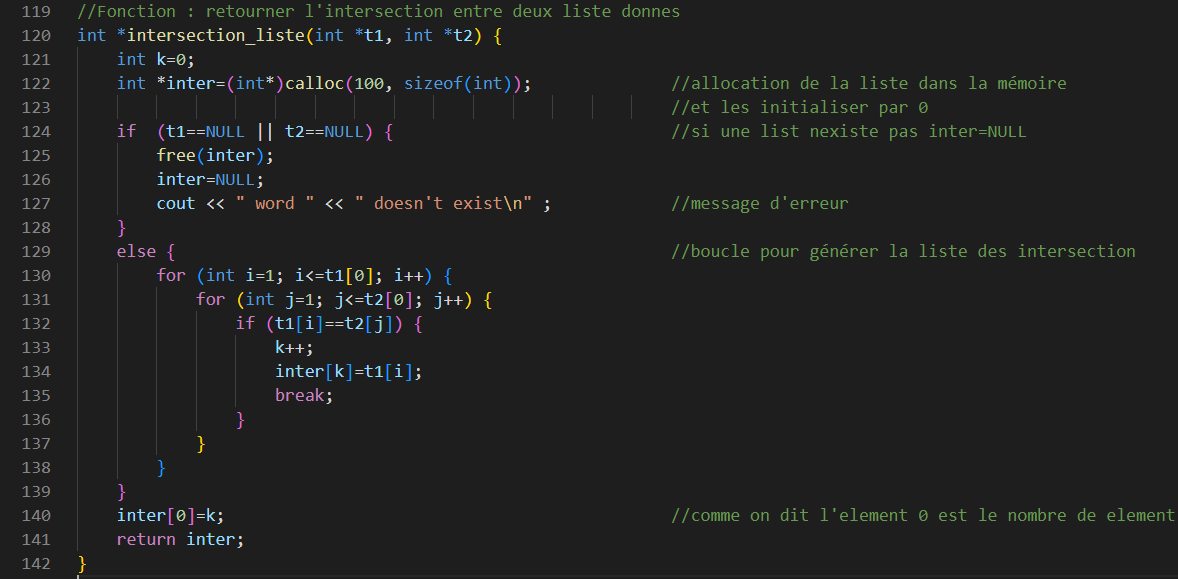
* 1. **Procédure d’affichage (infixe) :**
* **Procédure.**
* **Les entrés :** La racine (Tree \*tr), une variable aide à organiser l’affichage (bool b).
* **Les sorties :** / .
* **Le fonctionnement :** affiche l’arbre binaire en méthode infixe et le variable b (boolien) est utilisé pour afficher le titre de tableau une seule fois.



* 1. **Recherche d’un mot :**
* **Fonction.**
* **Les entrés :** La racine (Tree \*tr), le mot a trouver (std::string word).
* **Les sorties :** L’adresse du nœud ou le mot se trouve.
* **Le fonctionnement :** il test le résultat de la fonction « where » si le mot existe (retourner l’adresse trouver) si non (retourner NULL).



* 1. **Intersection entre deux listes données :**
* **Fonction.**
* **Les entrés :** deux listes d’entier (int \*t1, int \*t2).
* **Les sorties :** L’adresse de la liste d’intersection des positions (un pointeur sur un entier).



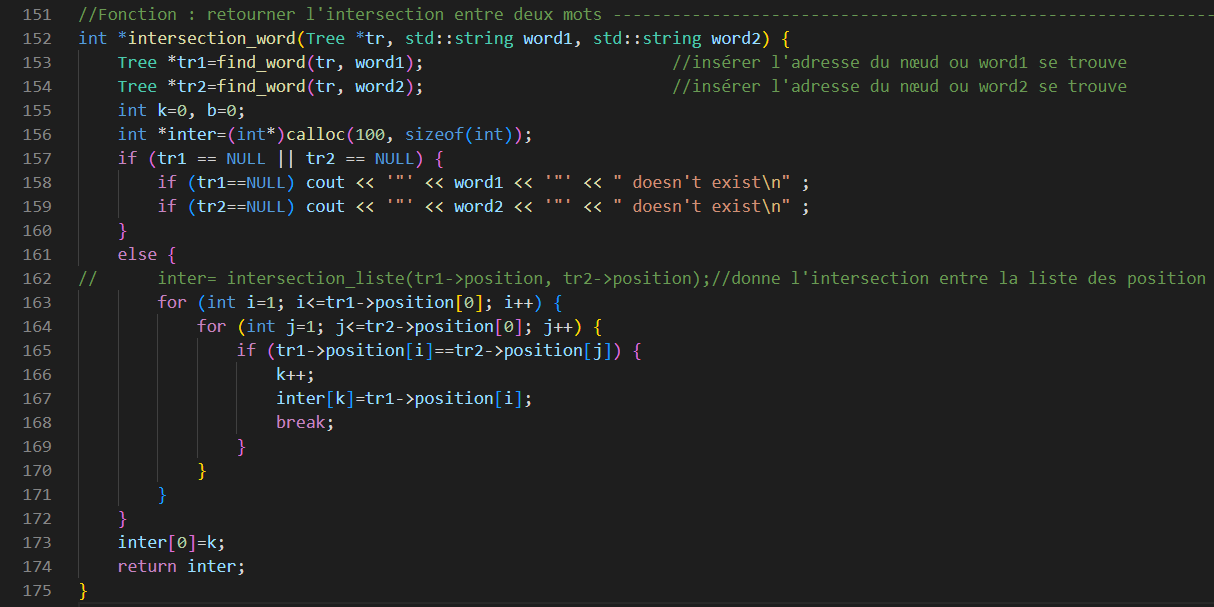
* 1. **Intersection entre la liste des positions entre deux mots donner :**
* **Fonction.**
* **Les entrées :** pointeur sur la racine de l’arbre (Tree \*tr), deux mots (std::string word1,

std::string word2).

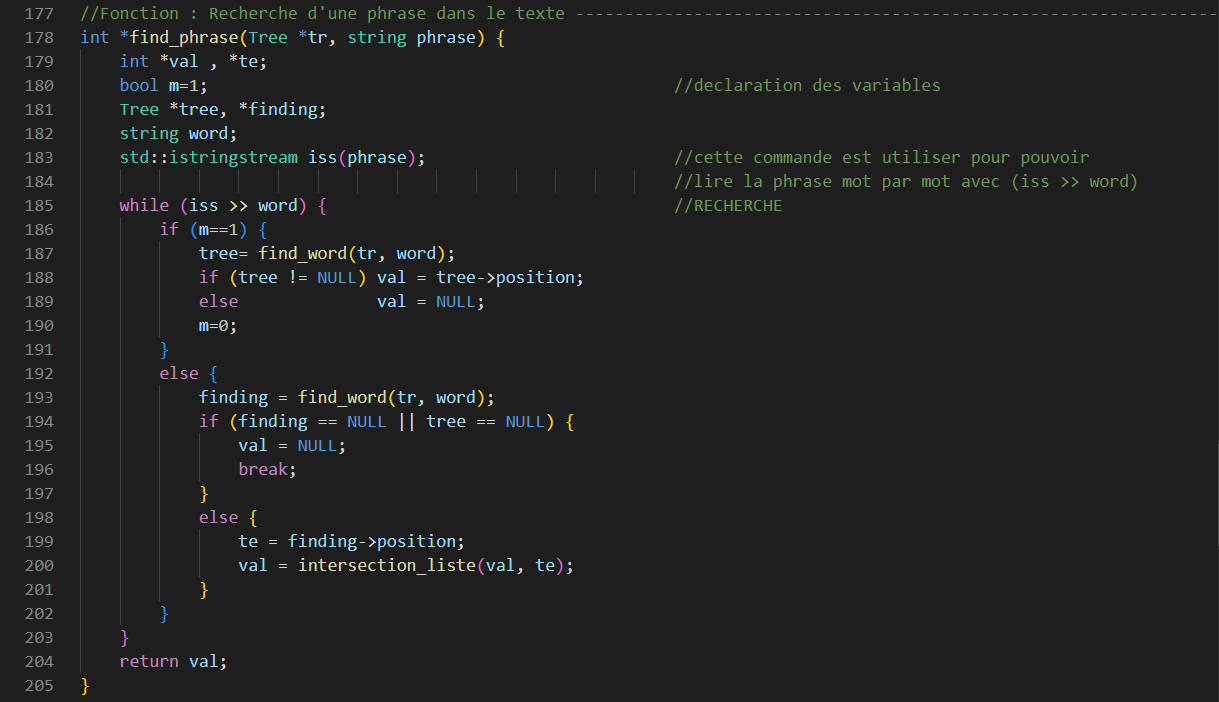
* **Les sorties :** L’adresse (premier élément) de la liste d’intersection de position

des deux mots.

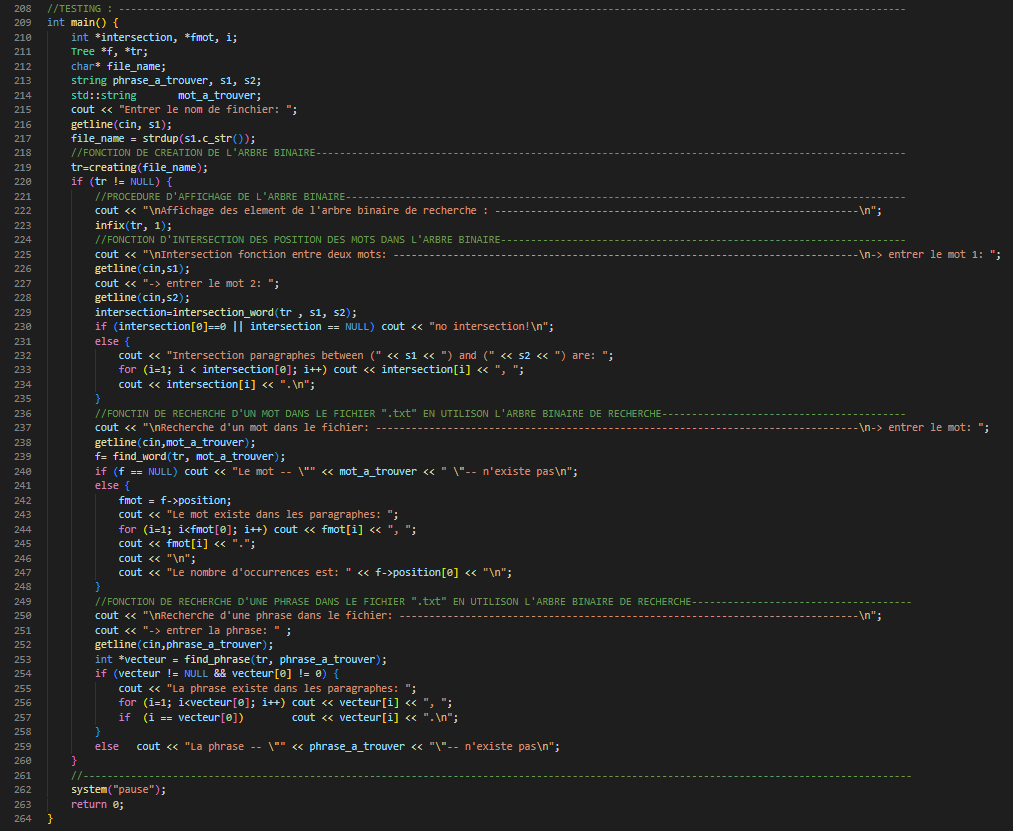
* **Le fonctionnement :** retourner la liste d’intersection de position des deux mots si les deux mots existent (si un ou les deux mots n’existent pas un message s’affichera).



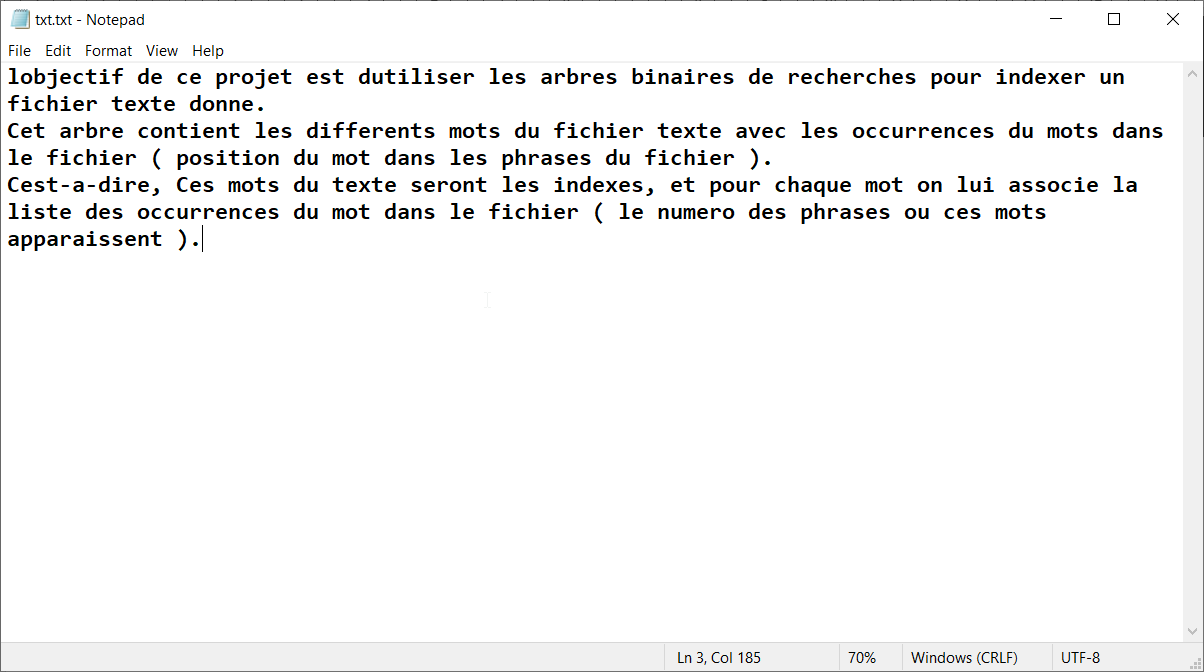
* 1. **Recherche d’une phrase:**
* **Fonction.**
* **Les entrés :** La racine (Tree \*tr), le mot a trouver (std::string word).
* **Les sorties :** List des intersections des mots de la phrase.
* **Le fonctionnement :** trouve les paragraphes où se trouvent les mots de la phrase sans pondre en considération l’ordre des mots.



1. **Test du code (exécution) :**
   1. **Code du test :**



* **Tester le programme avec le fichier texte suivant**

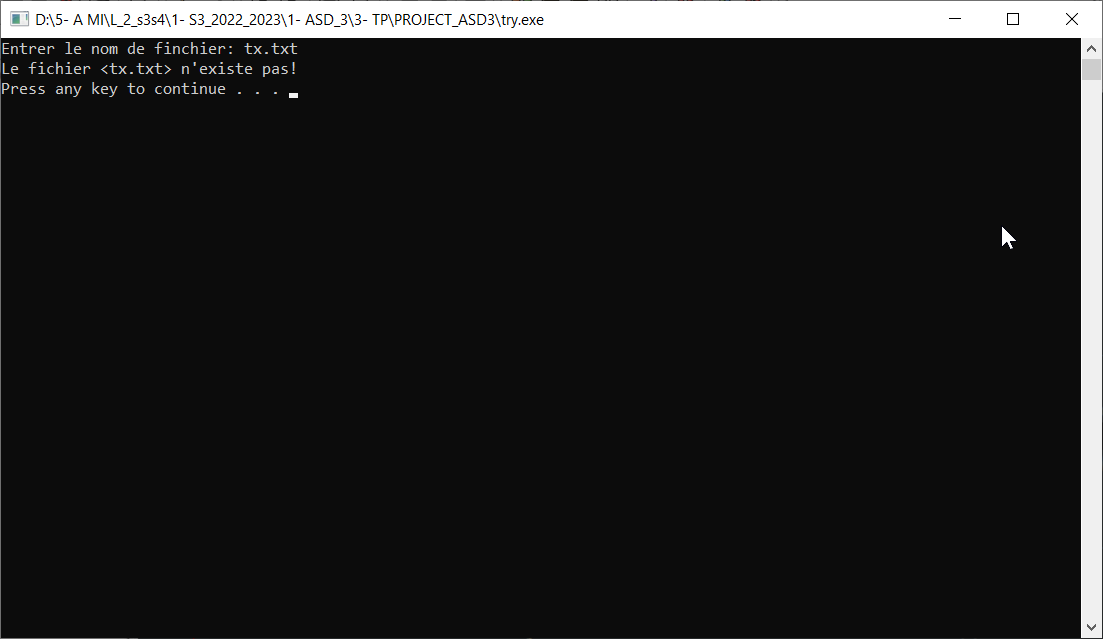


L’objectif de ce projet est d’utiliser les arbres binaires de recherches pour indexer un fichier texte donne. Cet arbre contient les différents mots du fichier texte avec les occurrences du mot dans le fichier (position du mot dans les phrases du fichier).

C’est-à-dire, Ces mots du texte seront les indexes, et pour chaque mot on lui associe la liste des occurrences du mot dans le fichier (le numéro des phrases ou ces mots apparaissent).

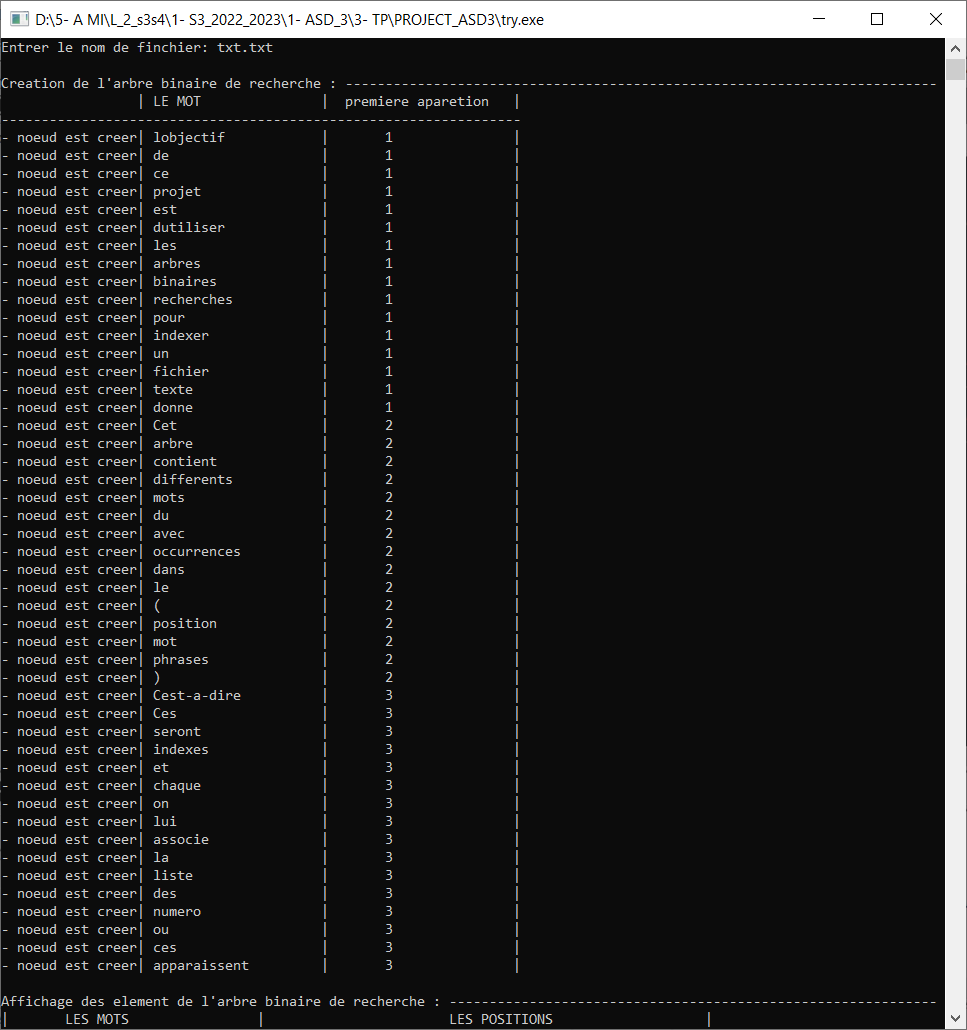
1. Ecrire le nom du fichier avec « .txt » :

* Si le fichier existe aller à l’étape suivante.
* Si le fichier n’existe pas un message d’erreur s’affichera.



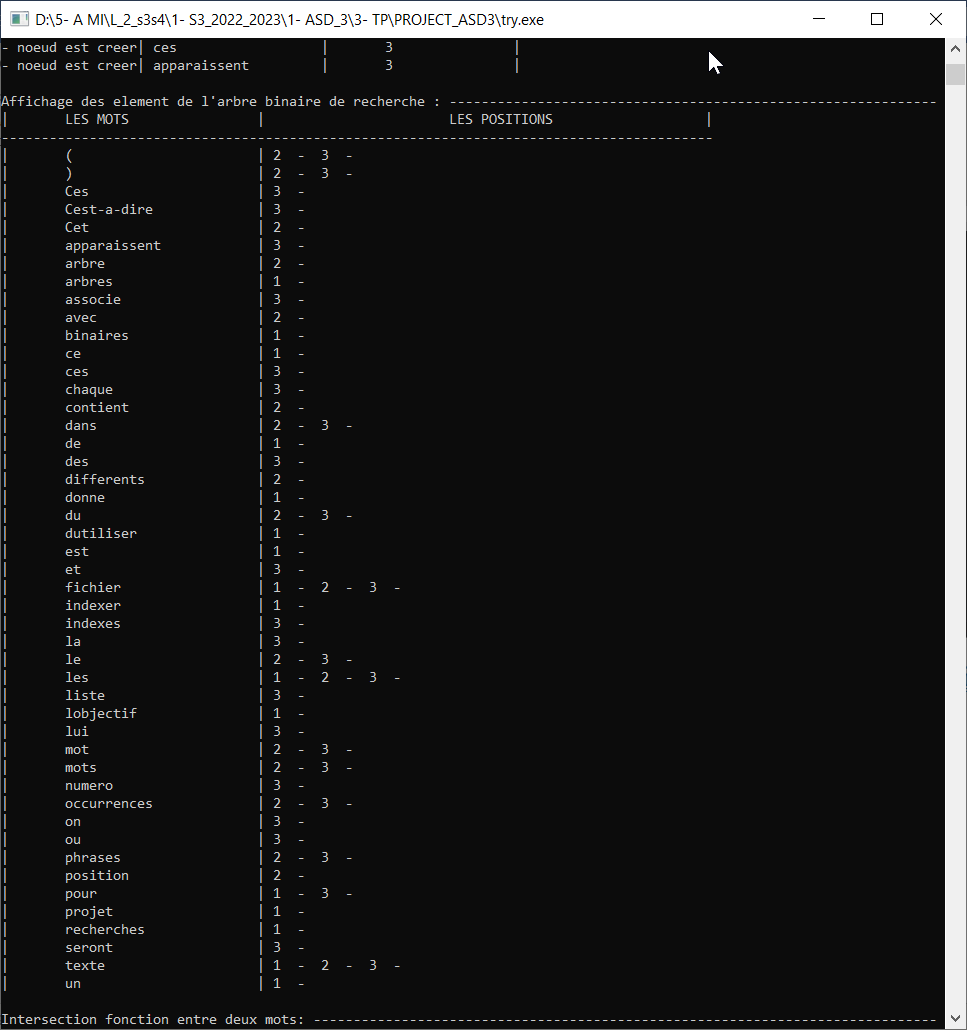
1. Création de l’arbre binaire de recherche : Le nom du fichier donné est « txt.txt ».

* Le porograme affichera les nœuds crés et leurs contenus initials ( les mots avec l’index du premier paragraphe apparessent dans le fichier sous forme de tableau ).



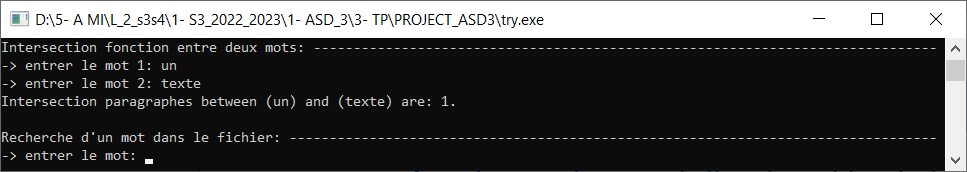
1. Affichage de l’arbre binaire en mode « infixe » :

* Le porograme affichera les mots avec leur liste de position sous forme d’un tableau.

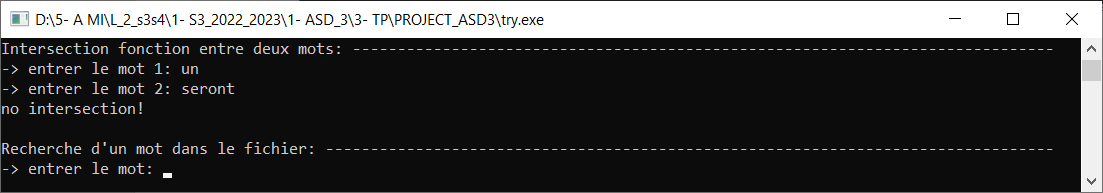


1. L’intercession entre la liste de position de deux mots :

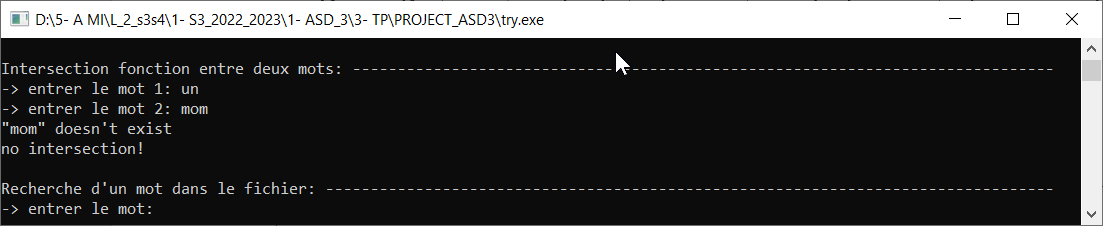
* Cas d’existance des deux mots et de l’intersections des liste des positions:



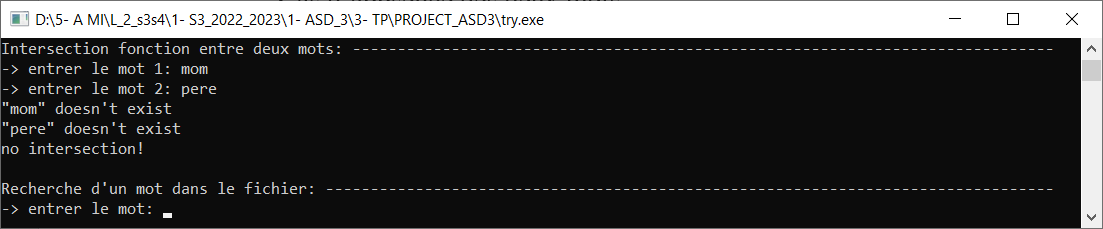
* Cas d’existonce des deux mots et absence de l’intersection :



* Cas d’absence d’un mot :

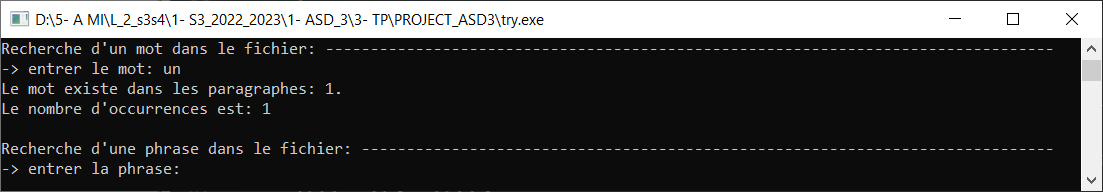


* Cas d’absence des deux mots :

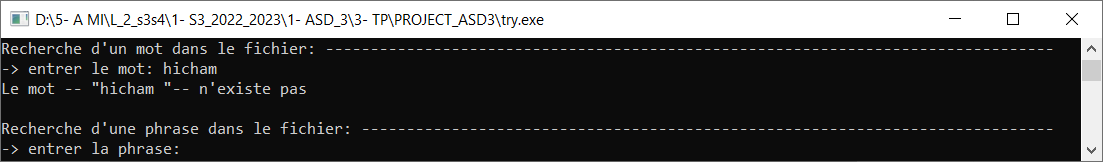


1. Recherche d’un mot :

* Si le mot existe : la liste des positions (les index du mot) s’affiche avec le nembre d’occurance.

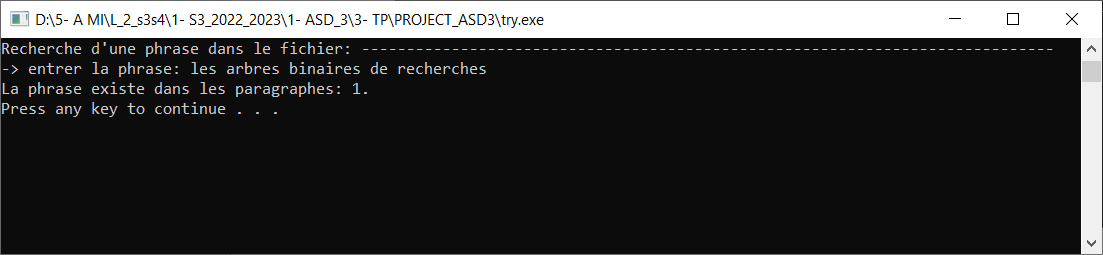


* Si le mot n’exite pas : un message d’erreur s’affiche.

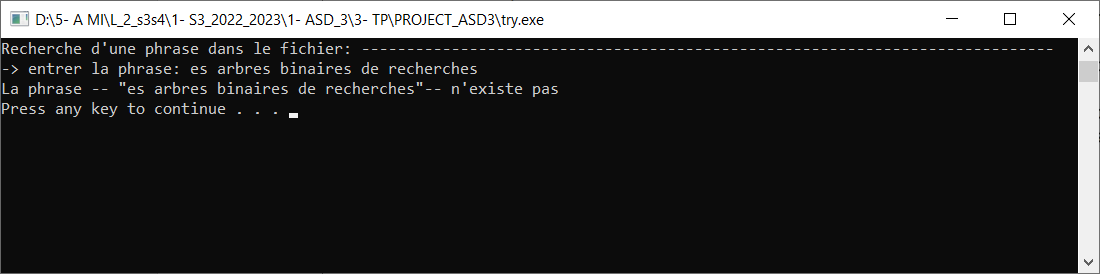


1. Recherche d’une phrase :

* Si la phrase existe : la liste des intersections des mots de la phrase s’affiche.



* Si la phrase n’existe pas : un message d’erreur s’affiche.



1. **Le code :**

|  |
| --- |
| 1. #include<iostream> 2. #include<stdio.h> 3. #include<stdlib.h> 4. #include<stdbool.h> 5. #include<string.h> 6. #include<fstream> 7. #include<sstream> 8. using namespace std; 9. //Enregistrement de l'arbre binaire -------------------------------------------------------------------------------------------------------- 10. typedef struct Tree { 11. char\*  mot;                                             //liste de caractère 12. int    position[100];                                   //numéro du paragraphe 13. struct Tree \*tleft, \*tright;                            //fils-gauche et fils-droit 14. } Tree; 15. //Fonction : existence de l'arbre  ---------------------------------------------------------------------------------------------------------- 16. bool Tree\_exist(Tree \*R) { 17. if (R==NULL) return 0;                                  //retourner 0 "faut" si l'arbre n'existe pas 18. else         return 1;                                  //retourner 1 "vrai" si l'arbre existe 19. } 20. //Fonction : Creation d'un nœud ------------------------------------------------------------------------------------------------------------ 21. Tree \*new\_node(string word, int index) { 22. Tree \*tr = (Tree\*)malloc(sizeof(\*tr));                  //allocation de la memoire 23. if (!Tree\_exist(tr)) cout << "Error in allocation!\n";  //erreur si l'arbre n'existe pas 24. tr->mot          = strdup(word.c\_str());                //insérer la valeur du word dans tr->mot 25. tr->position[1]  = index;                               //insérer le numéro du paragraphe dans 26. //tr->position[1] 27. tr->tleft        = NULL;                                //initialiser le fils-gauche par NULL 28. tr->tright       = NULL;                                //initialiser le fils-droite par NULL 29. tr->position[0]  = 1;                                   //insérer le nombre de paragraphe ou 30. //le mot a apparue 31. cout << "- noeud est creer| " << tr->mot; 32. if (word.length() < 5) cout << "\t\t\t"   << "| "; 33. else                   cout << "\t\t" << "| "; 34. cout << "\t" << tr->position[1] <<"\t\t|\n"; 35. //message de création 36. return tr;                                              //retourner l'adresse du nouveau nœud 37. } 38. //Fonction : Ajout d'une position ---------------------------------------------------------------------------------------------------------- 39. void add\_index(Tree \*tr, int index) { 40. if (index != 0 && index != tr->position[tr->position[0]]) { 41. tr->position[0]++;                                  //ajouter 1 au nombre de répétition du mot 42. tr->position[tr->position[0]] = index;              //insérer le nouveau index 43. } 44. //si le mot se répète dans le même paragraphe ne fait rien 45. } 46. //Fonction : Recherche le parent du nouveau mot et retourne son adresse--------------------------------------------------------------------- 47. Tree \*where(Tree \*tr, std::string word) { 48. std::string str=tr->mot;                                //insérer la valeur du tr->mot dans str 49. if (!Tree\_exist(tr)) return 0;                          //retourner 0 "faut" si l'arbre n'existe pas 50. else { 51. if (str > word && tr->tleft  != NULL)  return where(tr->tleft, word); 52. //si le mot<tr->mot chercher à gauche s'il existe des fils 53. if (str < word && tr->tright != NULL)  return where(tr->tright, word); 54. //si le mot>tr->mot cherche à gauche s'il existe des fils 55. } 56. return tr; 57. } 58. //Fonction : Création de l'arbre à partir du nom du ficher (.txt) -------------------------------------------------------------------------- 59. Tree \*creating(const char\* file\_name) { 60. int i=1, set=0;                                         //déclaration des variables 61. Tree \*tr, \*tree; 62. std::string  word, c, str; 63. char \*compare; 64. ifstream file;                                          //ficher en mode lecture 65. file.open(file\_name);                                   //ouvrir le fichier dont le nom est dans file\_name 66. if (!file.is\_open()) cout << "Le fichier <" << file\_name << "> n'existe pas!\n"; 67. //si le ficher ne s'ouvre pas afficher 68. // le message d'erreur 69. else { 70. cout << "\nCreation de l'arbre binaire de recherche : --------------------------------------------------------------------------\n"; 71. cout << "                 | LE MOT  \t\t|  premiere aparetion   |\n-----------------------------------------------------------------\n"; 72. file >> word;                                       //insérer un mot du fichier dans la variable 73. c=word[word.size()-1];                              //insérer le dernier élément du word dans c 74. if (c == "." || c == "," || c ==";" || c =="?" || c == "!") word[word.size()-1] = '\0'; 75. tr = new\_node(word, i);                             //création de la racine de l'arbre binaire 76. if (c == ".") i++; 77. tree = tr; 78. while (file >> word) {                              //boucle de création de l'arbre 79. c=word[word.size()-1]; 80. if (c == "." || c == "," || c ==";" || c =="?" || c == "!") word[word.size()-1] = '\0'; 81. compare=strdup(word.c\_str()); 82. word = compare; 83. tree=where(tr, word);                           //trouve le parent du nœud à créer 84. str  = tree->mot;                               //insérer dans str pour faciliter la comparaison 85. if      (str > word)    tree->tleft  = new\_node(word, i);   //créer un fils gauche 86. else if (str < word)    tree->tright = new\_node(word, i);   //créer un fils droit 87. //suivant les conditions 88. else if (/\*str == word &&\*/ tree->position[tree->position[0]] != i)   add\_index(tree, i); 89. if (c == ".") i++;                              //si c == '.' incrémente le numéro du paragraphe 90. } 91. file.close();                                       //fermer le fichier 92. return tr;                                          //retourner l'adresse de la racine 93. } 94. return 0; 95. } 96. //Procedure : Affiche l'arbre en méthode infixe -------------------------------------------------------------------------------------------- 97. void infix(Tree \*tr, bool b) { 98. int j; 99. string s; 100. if (b==1) cout << "|\tLES MOTS\t\t|\t\t\tLES POSITIONS\t\t\t|\n-----------------------------------------------------------------------------------------\n"; 101. if (tr->tleft  != NULL) infix(tr->tleft , 0); 102. cout << "|\t" << tr->mot; 103. s = tr->mot; 104. if (s.length() < 8) cout << "\t\t\t"   << "| "; 105. else               cout << "\t\t" << "| "; 106. for (j=1; j <= tr->position[0]; j++) cout << tr->position[j] << "  -  "; 107. cout << "\n"; 108. if (tr->tright != NULL) infix(tr->tright, 0); 109. } 110. //Fonction : Retourner l'adresse du nœud où le mot se trouve ------------------------------------------------------------------------------- 111. Tree \*find\_word(Tree \*tr, std::string word) { 112. Tree \*tree = where(tr, word);                           //where retourne l'adresse du mot s’il existe 113. // ou son parent s’il n’existe pas 114. std::string s=tree->mot;                                //insérer dans str pour faciliter la comparaison 115. if (s==word) return tree;                               //si le mot existe dans l'arbre la fonction 116. return NULL; 117. } 118. //Fonction : retourner l'intersection entre deux liste donnes ------------------------------------------------------------------------------ 119. int \*intersection\_liste(int \*t1, int \*t2) { 120. int k=0; 121. int \*inter=(int\*)calloc(100, sizeof(int));              //allocation de la liste dans la mémoire 122. //et les initialiser par 0 123. if  (t1== NULL || t2== NULL || t1[0]==0 || t2[0]==0) {  //si une liste n'existe pas inter=NULL 124. cout << " word " << " doesn't exist\n" ;            //message d'erreur 125. } 126. else {                                                  //boucle pour générer la liste des intersections 128. for (int i=1; i<=t1[0]; i++) { 129. for (int j=1; j<=t2[0]; j++) { 130. if (t1[i]==t2[j]) { 131. k++; 132. inter[k]=t1[i]; 133. break; 134. } 135. } 136. } 137. } 138. inter[0]=k;                                             //comme on dit l'élément 0 est le nombre des éléments 139. return inter; 140. //  return inter; 141. } 142. //Fonction : retourner l'intersection entre deux mots -------------------------------------------------------------------------------------- 143. int \*intersection\_word(Tree \*tr, std::string word1, std::string word2) { 144. Tree \*tr1=find\_word(tr, word1);                         //insérer l'adresse du nœud ou word1 se trouve 145. Tree \*tr2=find\_word(tr, word2);                         //insérer l'adresse du nœud ou word2 se trouve 146. int k=0, b=0; 147. int \*inter=(int\*)calloc(100, sizeof(int)); 148. if (tr1 == NULL || tr2 == NULL) { 149. if (tr1==NULL) cout << '"' << word1 << '"' << " doesn't exist\n" ; 150. if (tr2==NULL) cout << '"' << word2 << '"' << " doesn't exist\n" ; 151. } 152. else { 153. //      inter= intersection\_liste(tr1->position, tr2->position);//donne l'intersection entre la liste des positions 154. for (int i=1; i<=tr1->position[0]; i++) { 155. for (int j=1; j<=tr2->position[0]; j++) { 156. if (tr1->position[i]==tr2->position[j]) { 157. k++; 158. inter[k]=tr1->position[i]; 159. break; 160. } 161. } 162. } 163. } 164. inter[0]=k; 165. return inter; 166. } 167. //Fonction : Recherche d'une phrase dans le texte ------------------------------------------------------------------------------------------ 168. int \*find\_phrase(Tree \*tr, string phrase) { 169. int \*val , \*te; 170. bool m=1;                                               //déclaration des variables 171. Tree \*tree, \*finding; 172. string word; 173. std::istringstream iss(phrase);                         //cette commande est utilisée pour pouvoir 174. //lire la phrase mot par mot avec (iss >> word) 175. while (iss >> word) {                                   //RECHERCHE 176. if (m==1) { 177. tree= find\_word(tr, word); 178. if (tree != NULL) val = tree->position; 179. else              val = NULL; 180. m=0; 181. } 182. else { 183. finding = find\_word(tr, word); 184. if (finding == NULL || tree == NULL) { 185. val = NULL; 186. break; 187. } 188. else { 189. te = finding->position; 190. val = intersection\_liste(val, te); 191. } 192. } 193. } 194. return val; 195. } 196. //TESTING : ------------------------------------------------------------------------------------------------------------------------------------ 197. int main() { 198. int \*intersection, \*fmot, i; 199. Tree \*f, \*tr; 200. char\* file\_name; 201. string phrase\_a\_trouver, s1, s2; 202. std::string       mot\_a\_trouver; 203. cout << "Entrer le nom de finchier: "; 204. getline(cin, s1); 205. file\_name = strdup(s1.c\_str()); 206. //FONCTION DE CREATION DE L'ARBRE BINAIRE--------------------------------------------------------------------------------------------------- 207. tr=creating(file\_name); 208. if (tr != NULL) { 209. //PROCEDURE D'AFFICHAGE DE L'ARBRE BINAIRE---------------------------------------------------------------------------------------------- 210. cout << "\nAffichage des element de l'arbre binaire de recherche : -------------------------------------------------------------\n"; 211. infix(tr, 1); 212. //FONCTION D'INTERSECTION DES POSITIONS DES MOTS DANS L'ARBRE BINAIRE-------------------------------------------------------------------- 213. cout << "\nIntersection fonction entre deux mots: ------------------------------------------------------------------------------\n-> entrer le mot 1: "; 214. getline(cin,s1); 215. cout << "-> entrer le mot 2: "; 216. getline(cin,s2); 217. intersection=intersection\_word(tr , s1, s2); 218. if (intersection[0]==0 || intersection == NULL) cout << "no intersection!\n"; 219. else { 220. cout << "Intersection paragraphes between (" << s1 << ") and (" << s2 << ") are: "; 221. for (i=1; i < intersection[0]; i++) cout << intersection[i] << ", "; 222. cout << intersection[i] << ".\n"; 223. } 224. //FONCTIN DE RECHERCHE D'UN MOT DANS LE FICHIER ".txt" EN UTILISON L'ARBRE BINAIRE DE RECHERCHE----------------------------------------- 225. cout << "\nRecherche d'un mot dans le fichier: ---------------------------------------------------------------------------------\n-> entrer le mot: "; 226. getline(cin,mot\_a\_trouver); 227. f= find\_word(tr, mot\_a\_trouver); 228. if (f == NULL) cout << "Le mot -- \"" << mot\_a\_trouver << " \"-- n'existe pas\n"; 229. else { 230. fmot = f->position; 231. cout << "Le mot existe dans les paragraphes: "; 232. for (i=1; i<fmot[0]; i++) cout << fmot[i] << ", "; 233. cout << fmot[i] << "."; 234. cout << "\n"; 235. cout << "Le nombre d'occurrences est: " << f->position[0] << ".\n"; 236. } 237. //FONCTION DE RECHERCHE D'UNE PHRASE DANS LE FICHIER ".txt" EN UTILISON L'ARBRE BINAIRE DE RECHERCHE------------------------------------- 238. cout << "\nRecherche d'une phrase dans le fichier: -----------------------------------------------------------------------------\n"; 239. cout << "-> entrer la phrase: " ; 240. getline(cin,phrase\_a\_trouver); 241. int \*vecteur = find\_phrase(tr, phrase\_a\_trouver); 242. if (vecteur != NULL && vecteur[0] != 0) { 243. cout << "La phrase existe dans les paragraphes: "; 244. for (i=1; i<vecteur[0]; i++) cout << vecteur[i] << ", "; 245. if  (i == vecteur[0])        cout << vecteur[i] << ".\n"; 246. } 247. else   cout << "La phrase -- \"" << phrase\_a\_trouver << "\"-- n'existe pas\n"; 248. } 249. //------------------------------------------------------------------------------------------------------------------------------------------- 250. system("pause"); 251. return 0; 252. } |