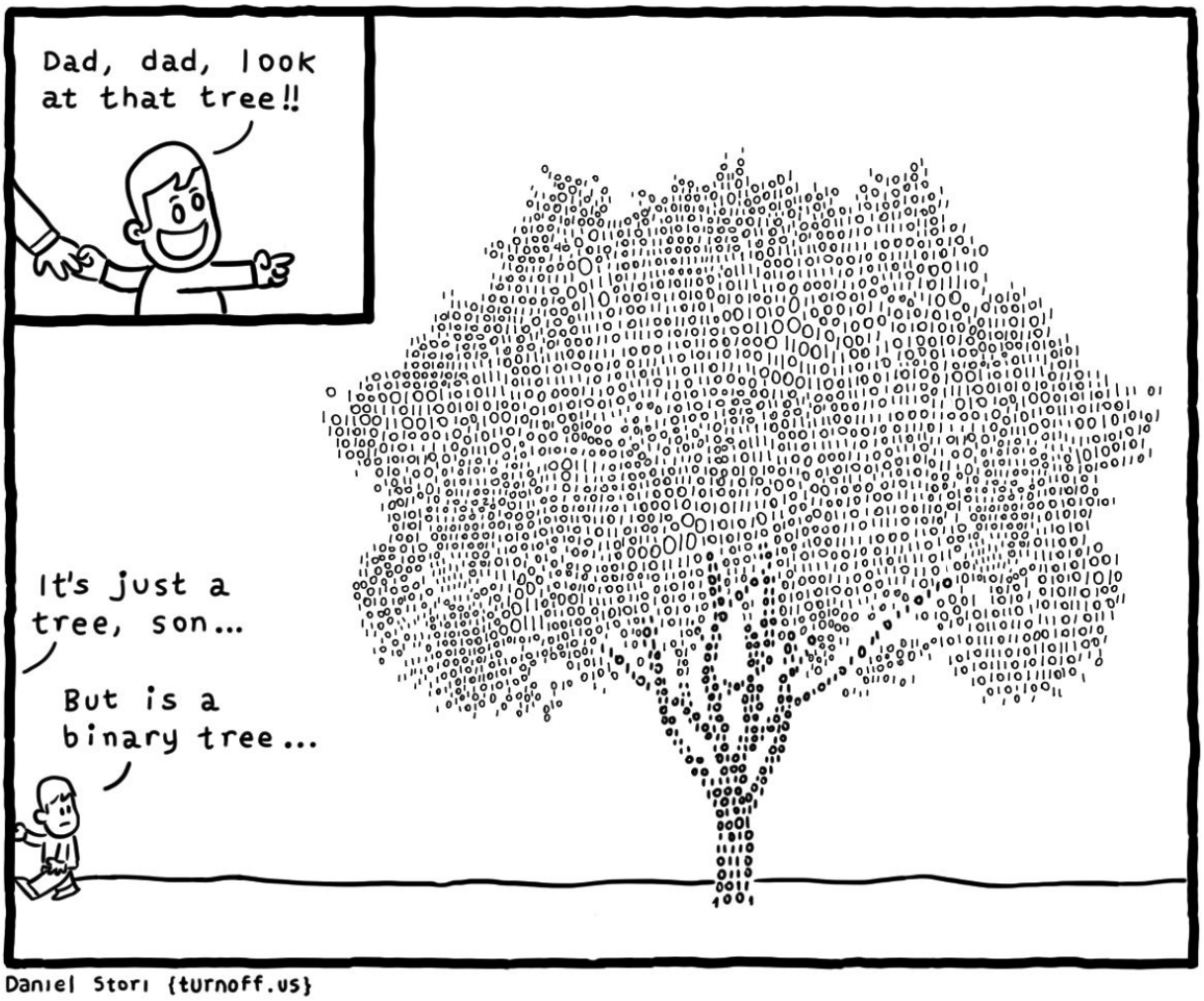
Université de Versailles St-Quentin en Yvelines

IN505 Langages avancés

Cryptage et décryptage en C++

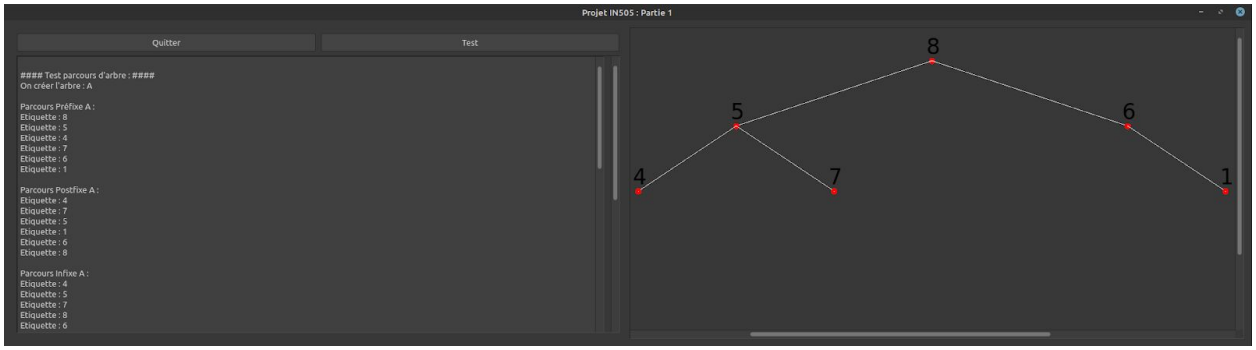


Sommaire

1. Introduction

Ce projet à pour but l'implémentation d'un système de codage et décodage basé sur l'algorithme de Huffman, et la programmation d'une interface graphique qui nous permettra de représenter l'arbre (suite à ajouter à la fin du projet.)

Nous avons décidé de faire l'interface graphique avec QT5.



Capture d'écran de notre interface graphique

2. Logiciels et langages utilisés

Pour ce projet nous utilisons :

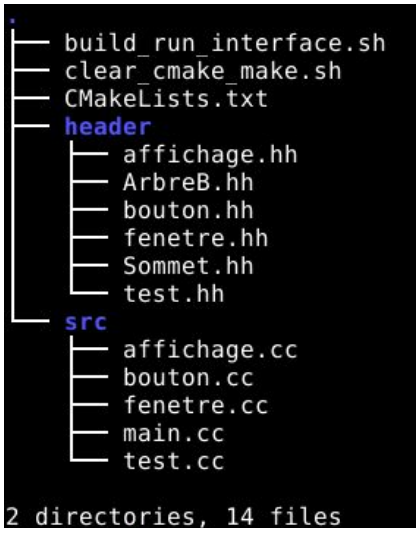
- cmake pour la compilation du projet
- Qt5 pour l'interface graphique
- GNU Compiler Collection (GCC)

Dans le cas où la compilation ne serait pas possible, il faudra installer les dépendances ci-dessus, ou exécuter le script d'installation nommé :

dependances_install.sh via le terminal : **$ sh dependances_install.sh**

3. Partie 1 : Structures abstraites de données

3.1 Arborescence du projet



La partie 1 est séparé en deux dossiers :

- **src** (le dossier contenant les sources, fichiers .cc)
- **header** (dossier contenant les headers, fichiers .hh)

Elle contient aussi à sa racine trois fichiers :

- **CMakeLists.txt** (fichier de création des dépendances pour la compilation de Qt)
- **build_run_interface.sh** (exécution du CMakeLists.txt puis compilation et exécution du projet)
- **clear_cmake_make.sh** (suppression des fichiers engendrés par l'exécution de cmake).

3.2 Compilation et exécution du projet

Pour compiler et exécuter le projet, on se placera dans le dossier **Partie1** puis on exécutera le script **build_run_interface.sh** via le terminal : **$ sh build_run_interface.sh**

Ce script exécutera le fichier CMakeList.txt via $cmake qui créera les dépendance et un makefile qui sera exécuté par la suite du script.

Pour supprimer les fichiers généré par cmake il est possible d'utiliser le script **clear_cmake_make.sh** via le terminal : **$ sh clear_cmake_make.sh**

Ce script supprime toutes les dépendances et les fichiers engendrés par le script précédent.

3.3 Listing des méthodes de calcul de l'arbre

3.3.1 Classe ArbreB :

Attributs de la classe :

- `Sommet<T> *_racine` : correspond à la racine de l'arbre
- `Sommet<T> *_sCourant` : correspond au sommet courant de l'arbre
- `_nbr_sommet` : correspond au nombre de sommets de l'arbre

Méthodes de la classe :

1) Création de l'arbre et copie d'arbre

Pour créer un arbre on peut utiliser le constructeur par défaut ou le constructeur par copie.

`ArbreB()` : Met la racine à null, le sommet courant à null et met la valeur de _nbr_sommet à 0.

`Sommet<T> *copie(Sommet<T> *sommet)` : Permet de copier un sommet avec tous ses fils à partir du sommet passé en paramètre et renvoie la racine du sous arbre copié.

`ArbreB(const ArbreB<T> &arbre)` : Permet de faire une copie de l'arbre mis en paramètre en utilisant la méthode copie.

`ArbreB<T> &operator=(const ArbreB<T> &arbre)` : Permet de faire une copie de l'arbre passé en paramètre dans l'arbre courant en utilisant la méthode copie.

2) Les getters et les setters

`int getEtiquette() const` : Renvoie l'étiquette du sommet courant.

`void setEtiquette(const T &val)` : Permet de modifier la valeur de l'étiquette du sommet courant par la valeur passée en paramètre.

`int getNbrSommet() const` : Permet d'obtenir le nombre de sommets de l'arbre

`void nbrSommet(Sommet<T> *&sommetRacine, int *&val)` : Méthode private permet d'incrémenter la valeur du paramètre val en fonction du nombre de sommets de l'arbre on passera la racine en paramètre pour sommetRacine pour pouvoir commencer à compter au sommet de l'arbre.

`int profondeur(Sommet<T> *&racine)` : Méthode private permettant de retourner la profondeur de l'arbre. Le calcul de la profondeur s'effectue à partir du sommet passé en paramètre.

`int get_profondeur()` : Renvoie le résultat retourné par la méthode profondeur à la quel on a passé la racine de l'arbre en paramètre.

3) Les méthodes permettant l'ajout de sommet à l'arbre :

`void ajoutG(const T &val)` : Permet d'ajouter au sommet courant un fils gauche avec pour étiquette la valeur passée en paramètre.

`ArbreB<T> &operator<<(const T &val)` : Surcharge de l'opérateur <<, on va appeler la méthode ajoutG avec comme paramètre val.

`void ajoutD(const T &val)` : Permet d'ajouter au sommet courant un fils droit avec pour étiquette la valeur passée en paramètre.

`ArbreB<T> &operator>>(const T &val)` : Surcharge de l'opérateur >>, on va appeler la méthode ajoutD avec comme paramètre val.

`void ajout(const T &val)` : Méthode privée permettant l'ajout d'un sommet à l'arbre, en fonction du poids de son étiquette. La méthode prend en paramètre la valeur à affecter au sommet.

`void ajoutAuto(const T &val)` : Avant l'appel de la méthode ajout à lequel on passe en paramètre le paramètre val, on utilise la méthode remonte_racine.

4) Les méthodes permettant de se déplacer dans l'arbre :

`void remonter_racine()` : Permet de mettre le sommet courant sur la racine

`bool deplacementG()` : Permet de déplacer le sommet courant sur son fils gauche

`bool deplacementD()` : Permet de déplacer le sommet courant sur son fils droit

5) Les méthodes de parcours de l'arbre :

`void prefixe(Sommet<T> *&sommetRacine)` : Méthode private permettant d'effectuer un parcours préfixe à partir du sommet passé en paramètre.

`void parcours_prefixe()` : Fais appel à la méthode préfixe en lui passant en paramètre la racine de l'arbre.

`void infixe(Sommet<T> *&sommetRacine)` : Méthode private permettant d'effectuer un parcours infixe à partir du sommet passé en paramètre.

`void parcours_infixe()` : Fais appel à la méthode infixe en lui passant en paramètre la racine de l'arbre.

`void postfixe(Sommet<T> *&sommetRacine)` : Méthode private permettant d'effectuer un parcours postfixe à partir du sommet passé en paramètre.

`void parcours_postfixe()` : Fais appel à la méthode postfixe en lui passant en paramètre la racine de l'arbre.

6) Les méthodes permettant la suppression :

`void supprimer(Sommet<T> *&racine)` : Cette méthode est private et aide à la suppression totale de l'arbre. Elle est notamment utilisée dans la méthode tout_supprimer et dans le destructeur. Elle prend en paramètre un sommet.

`void suppression_un_fils(Sommet<T> *&sommet)` : Cette méthode est private et permet de supprimer un sommet possédant un fils. Le sommet supprimé est le sommet passé en paramètre qui correspond au sommet courant dans l'arbre. Cette méthode est utilisée dans la méthode supprimer_sommet.

`void suppression_deux_fils(Sommet<T> *&sommet)` : Cette méthode est private et a le même fonctionnement que la méthode suppresion_un_fils sauf que cette fois le sommet qu'on souhaite supprimer possède deux fils.

`void supprimer_feuille()` : permet de supprimer le sommet courant si c'est une feuille.

`void supprimer_sommet()` : permet de supprimer n'importe quel sommet en utilisant les méthodes supprimer_feuille, suppression_un_fils et suppression_deux_fils.

`void tout_supprimer()` : permet de supprimer tous les sommets d'un arbre en utilisant la méthode supprimer.

`~ArbreB()` : En utilisant la méthode supprimer il va supprimer chaque sommet de l'arbre.

`bool estVide()` : Retourne vrai si la racine et le sommet courant sont vides sinon retourne faux.

7) Méthode de recherche

`bool recherche(Sommet<T> *&sommetRacine, const T &val, bool &existe)` : Méthode private permettant de savoir si une étiquette présente dans l'arbre retourne vraie si la valeur passée en paramètre correspond à une étiquette de l'arbre, retourne faux sinon.

`bool estPresent(const T &val)` : Utilise la méthode recherche en lui donnant en paramètre la valeur val, le racine de l'arbre est un boolean à faux, on return le résultat de recherche

`void recherchePere(Sommet<T> *&courant, Sommet<T> *&recherche, Sommet<T> *&pere)` : Méthode private permettant d'obtenir le père du sommet courant. Pour cela on passe en paramètre le sommet courant correspondant au sommet pour lequel on va commencer la recherche, on donne le sommet recherche correspondant au sommet à qui on recherche le père, et le sommet père qui une fois la recherche effectuée prendra la position du sommet père du sommet recherché.

8) Méthode de décomposition et fusion :

`void decomposition(ArbreB<T> &arbre)` : Permet de décomposer un arbre dont la partie décomposée sera affecté à l'arbre passé en paramètre et supprimé de l'arbre courant.

`ArbreB<T> &operator+=(ArbreB<T> &arbre)` : permet de fusionner l'arbre courant avec l'arbre passé en paramètre. La fusion des deux arbres sera affecté à l'arbre courant

9) La méthode d'écriture de fichier

`static void ecrire_fichier(const std::string &&fichier, const std::string &log)` : Méthode private permettant d'écrire dans un fichier dont le nom est stocké dans le paramètre fichier et un message de type string contenu dans le paramètre log.

`static void ecrire_log(const std::string &log)` : Utilise la méthode ecrire_fichier en lui passant en paramètre le nom du fichier log.txt et le message à écrire dans le fichier.

`void ecrire_arbre(const std::string &info)` : Méthode private permettant d'écrire dans un fichier l'arbre textuel.

`void ecrire_arbre_interface(const std::string &info)` : Méthode private permettant d'écrire les informations nécessaires à la construction de l'arbre dans l'interface graphique dans ce fichier on stockera G pour descendre à gauche dans l'arbre D pour descendre à droite RD et RG pour remonter gauche et droite et l'étiquette du sommet.

`void ecrire_fichier_arbre(Sommet<T> *&racine)` : Méthode private permettant de parcourir l'arbre à partir du sommet en paramètre, on utilisera la méthode ecrire_arbre_interface afin de passer les paramètres de construction de l'arbre.

`void arbreInterface()` : Permet d'appeler la méthode ecrire_fichier_arbre en lui passant la racine de l'arbre en paramètre.

10) La méthode d'affichage de l'arbre textuellement

`void affichage_arbre(Sommet<T> *&racine, int decalage)` : Méthode private permettant de dessiner un arbre textuellement dans le terminal de côte et d'écrire cet arbre dans un fichier texte pour pouvoir également l'écrire dans l'interface graphique

3.3.2 Classe Sommet :

- `Sommet<T> *_filsG` : correspond au fils gauche du sommet
- `Sommet<T> *_filsD` : correspond au fils droit du sommet
- `T _etiquette` : l'étiquette prise par le sommet
- `Sommet(const T &etiquette)` : _filsG et _filsD sont mis à null et _etiquette prend la valeur du paramètre

Tout est en private la destruction des objets Sommet est fait dans le destructeur de ArbreB.

On définit Sommet comme étant friend de la classe ArbreB ce qui nous permettra de manipuler facilement les sommets dans l'arbre.

3.4 Listing des fonctions de test

3.4.1 test.hh :

`void testParcoursArbre()` : On crée un arbre avec plusieurs sommets dedans et on teste les différents types de parcours préfixe, postfixe et infixe.

`void testSuppresionArbre()` : On va créer un arbre vide regarder s'il est bien vide ensuite nous allons ajouter des sommets à cet arbre tester s'il est vide puis le supprimer et tester à nouveau s'il est bien vide.

`void testCopieArbre()` : On va créer un arbre avec plusieurs sommets, on va créer deux nouveaux arbres, le premier on va faire une copie de l'arbre en utilisant l'opérateur = et le second d'arbre nous utiliserons le constructeur de copie.

`void testFusionArbre()` : Nous allons créer deux arbres avec des sommets que nous allons fusionner en utilisant l'opérateur +=

`void testEtiquette()` : Nous allons créer un arbre avec plusieurs sommets on va modifier la valeur d'un des sommets et tests plusieurs valeurs pour savoir si elles correspondent à des étiquettes de notre arbre.

`void testDecompositionArbre()` : On va créer un arbre A avec plusieurs sommets on va choisir un sommet pour faire la décomposition.

`void testAjoutAuto()` : On va créer un arbre et utiliser la méthode ajoutAuto pour remplir l'arbre afin d'en faire un Arbre Binaire de Recherche (ABR).

`void testSuppressionSommet()` : On va créer un arbre avec plusieurs sommets on va supprimer l'arbre en supprimant une feuille, un sommet avec un fils et un sommet avec deux fils jusqu'à avoir un arbre vide

`void testDessinArbre()` : On va créer l'arbre qu'on affichera dans la partie graphique.

3.5 Listing des méthodes de l'interface

3.5.1 Classe Affiche :

Méthodes de la classe :

`Affichage()` : Constructeur de la classe Affichage

`~Affichage()` : Destructeur de la classe Affichage

`QSize sizeHint() const` : Cette méthode permet de gérer la taille de la zone de dessin de l'arbre

`int lecture_fichier_arbre()` : Cette classe va lire le contenu du fichier arbre_interface.txt et appeler les méthodes de dessin de l'arbre à partir du contenu du fichier texte.

`void affiche_ligne(int tmpX, int tmpY, int X, int Y)` : Permet d'afficher les arêtes de l'arbre.

`void affichage_noeud(std::string s, int abs_X, int ord_Y, int taille_C)` : Permet d'afficher les sommets et les étiquettes de l'arbre

`void paintEvent(QPaintEvent *event)` : Permet l'affichage de l'arbre

3.5.2 Classe Bouton :

Méthodes de la classe :

`Bouton()` : Le constructeur permet de créer et de gérer les boutons Quitter et Test.

`~Bouton()` : Destructeur de la classe bouton

`aff_test()` : Permet de lire les informations du fichier log.txt contenant le résultat de tous les tests et de leurs affichage lorsqu'on clique sur le bouton Test.

3.5.3 Classe Fenêtre :

Méthodes de la classe :

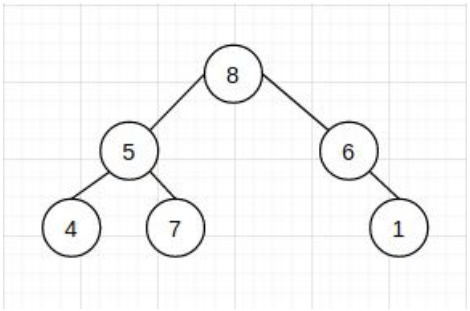
`Fenetre()` : Permet de gérer l'apparence de la fenêtre graphique

`~Fenetre()` : Destructeur de la classe Fenetre

3.6 Schématisation

3.6.1 Construction d'un arbre

Pour créer l'arbre suivant :

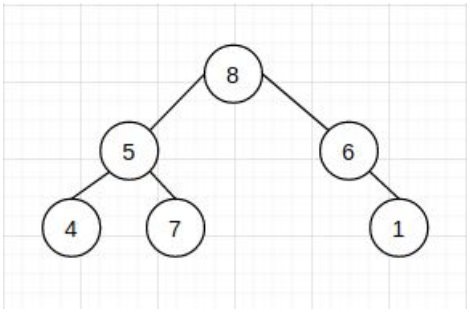


Nous pouvons faire :

```
Arbre<int> arbre; // Permet de créer l'arbre

arbre.ajoutG(8); // Ici on ajoute un sommet à gauche comme l'arbre est vide cet ajout va assigner le sommet à la racine avec l'étiquette 8

arbre << 5 << 4; // Autre façon d'ajouter un sommet à gauche

arbre.remonter_racine(); // Permet de mettre le sommet courant sur la racine

arbre.ajoutD(6); // Permet d'ajouter un sommet à droite avec l'étiquette 6

arbre >> 1; // Autre façon d'ajouter un sommet à droite avec pour étiquette 1

arbre.remonter_racine();

arbre.deplacementG(); // Permet de déplacer le sommet courant sur son fils gauche s'il existe

arbre >> 7;
```

3.6.2 Schéma parcours d'un arbre

On crée l'arbre suivant :



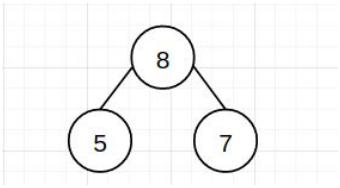
Le parcours préfixe nous donne : 8, 5, 4, 7, 6, 1

Le parcours postfixe nous donne : 4, 7, 5, 1, 6, 8

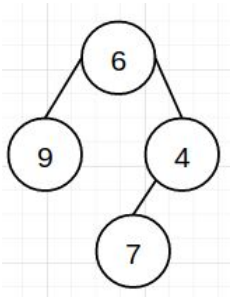
Le parcours infixe nous donne : 4, 5, 7, 8, 6, 1

3.6.3 Schéma fusion de deux arbres

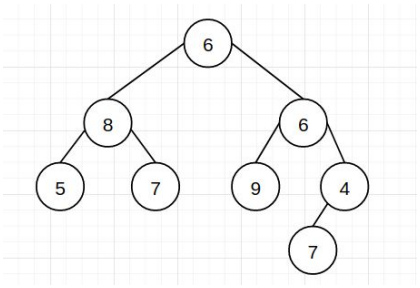
On crée un arbre A :



On crée l'arbre B :

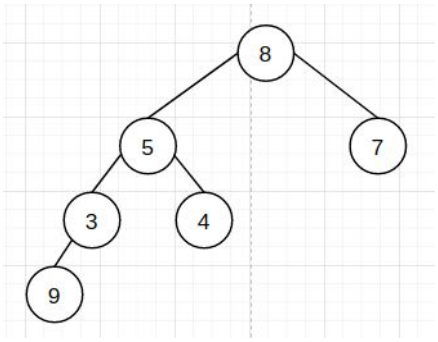


La fusion des deux nous donne :



3.6.4 Schéma de décomposition d'un arbre

On crée l'arbre A :

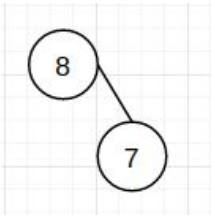


On décompose l'arbre à partir du sommet 5.

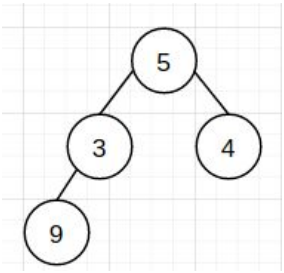
Pour cela on utilise la méthode `decomposition(ArbreB<T> &arbre)` de cette façon :

`A.decomposition(B);` ici B est un arbre vide.

L'arbre A devient :

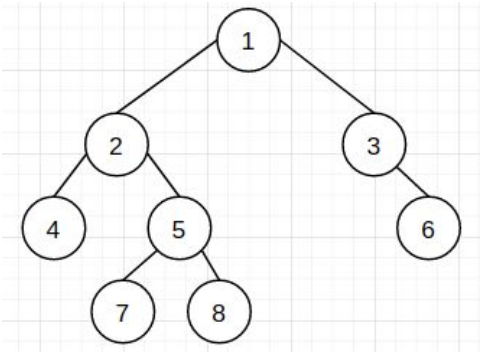


L'arbre B contient :



3.6.5 Schéma suppression de sommet

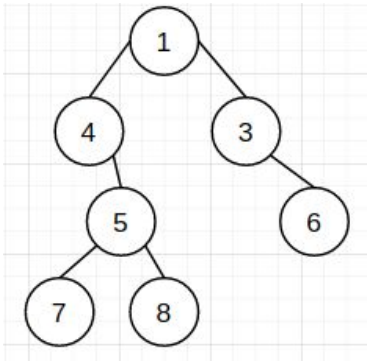
On crée l'arbre :



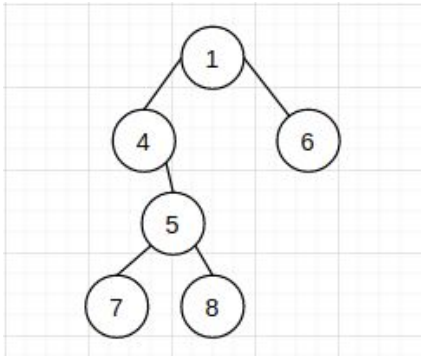
Dans cet exemple la suppression des sommets se fera avec la méthode :

`supprimer_sommet()`

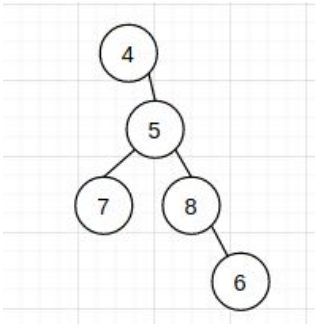
On supprime le sommet 2 on obtient :



On supprime le sommet 3 on obtient :

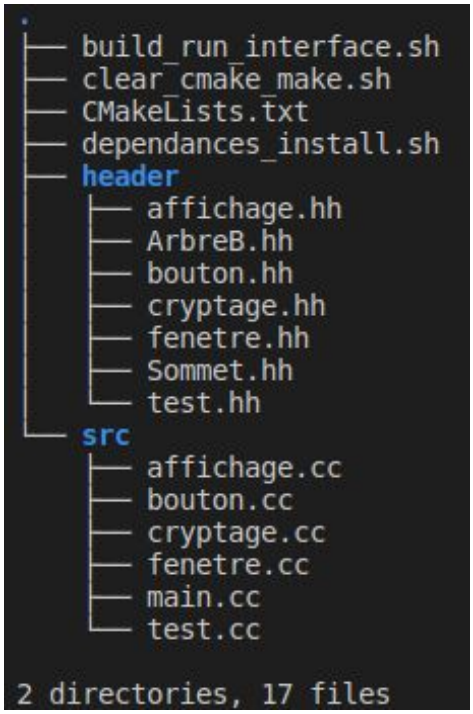


On supprime le sommet 1 on obtient :



4. Partie 2 : Cryptage

4.1 Nouvelle Arborescence du projet



La partie 2 est composée des deux mêmes dossiers :

- **src**
- **header**

Et contient les mêmes quatre fichiers à sa racine :

- **CMakeLists.txt**
- **build_run_interface.sh**
- **clear_cmake_make.sh**
- **dependances_install.sh**

Il a 2 fichiers supplémentaires :

- **cryptage.cc**
- **cryptage.hh**

Qui permettent de faire tourner le codage de Huffman.

Ils seront expliqués en détail par la suite.

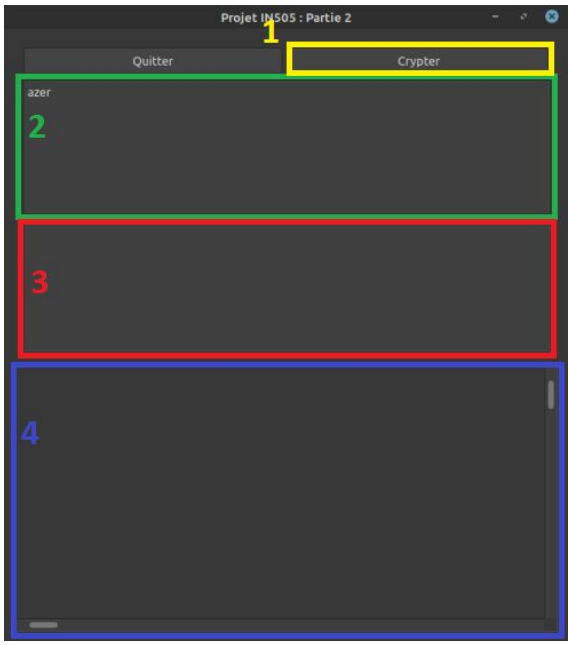
4.2 Compilation et exécution du projet

La compilation et l'exécution du projet restent inchangés, on se placera dans le dossier **Partie2** puis on exécutera le script **build_run_interface.sh** via le terminal : **$ sh build_run_interface.sh**

Pour supprimer les fichiers généré par cmake il est toujours possible d'utiliser le script **clear_cmake_make.sh** via le terminal : **$ sh clear_cmake_make.sh**

4.3 Fonctionnement de l'application

Voici l'interface graphique :

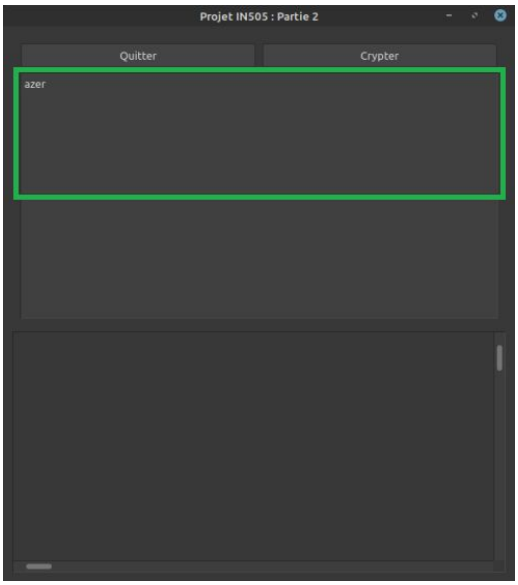


Il s'agit de l'interface graphique de la partie 1 que nous avons améliorée.

Elle est composée de deux nouvelles zones, une pour l'écriture du texte à crypter (zone encadrée en **vert 2**) et la zone 1(zone encadrée en **jaune 1**) est une transformation du bouton "test" en un bouton "Crypter" . Les zones encadrés **3** et **4** sont respectivement les zones d'écriture du code de Huffman et les zones d'affichage de l'arbre.

Il y a aussi un changement du placement de l'affichage de l'arbre (zone **4**) et de l'affichage du texte (zone **3**) pour des raisons de clarté (représentation d'un arbre qui contiendrait beaucoup de caractères).

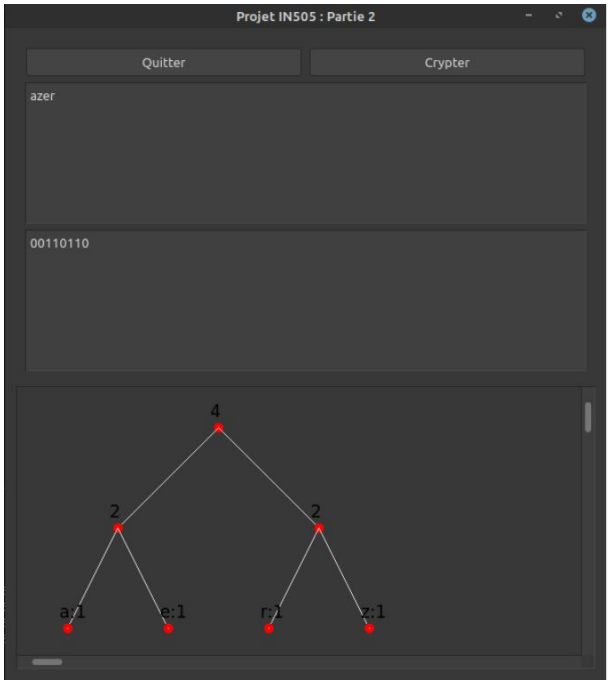
Pour utiliser cette interface, on commence par écrire un texte (ici :"azer") dans la zone de texte (zone **2**) :



Puis on appuie sur le bouton "**Crypter**" (zone **1**) :

Le programme va donc exécuter un codage de Huffman et afficher le résultat dans la zone **3** (ici **00110110**).

L'arbre binaire résultant de ce codage sera dessiné dans la zone **4** avec les occurrences, et les caractères qui leurs sont associées.



Après avoir "crypté" un texte une première fois, on peut changer le texte de la zone **2** et utiliser à nouveau le bouton "**Crypter**" pour obtenir un nouvel arbre et un nouveau code de Huffman, et ce à l'infini.

Le bouton "**Quitter"** comme son nom l'indique permet de quitter l'application.

Attention, le programme ne prend en compte que les caractères de la norme **ASCII**. Par conséquent les accents et autres caractères spéciaux externes à cette norme déclencheront un message d'erreur et empêcheront l'exécution du "cryptage".

4.4 Ajout de nouvelles méthodes et modifications

4.4.1 Classe ArbreB :

`char getLettre()` : Retourne la lettre du sommet courant.

`void ajoutR(const T &val, const char &lettre)` : permet de créer une racine en choisissant son étiquette et sa lettre via les paramètres de la méthode.

`void chemin(Sommet<T> *&racine, std::string &parcours, std::map<char, std::string> &map)` : Méthode privée permettant de remplir le paramètre map avec la lettre et son codage en parcourant l'arbre à partir du sommet passé en paramètre. Le paramètre parcours sert à remplir la map avec le codage correspondant à chaque lettre.

`std::map<char, std::string> codage()` : Cette méthode fait appel à la méthode **chemin** qui va remplir la map avec les lettres et leurs codages et va ensuite retourner la map complétée.

4.4.2 Classe Sommet :

- Ajout d'un nouveau membre `char lettre` nous permettant d'assigner une lettre à un Sommet cette lettre correspond à un caractère du texte à encoder.
- Ajout d'un nouveau constructeur permettant de setup l'étiquette correspondante aux occurrences.

`Sommet(const T &etiquette, const char &lettre)` : Constructeur permettant de setup l'étiquette correspondante à l'occurrence et la lettre correspondante à un des caractères du mot.

Par défaut l'étiquette est à 0 et la lettre est à \0.

4.5 Listing des fonctions de cryptage

4.5.1 cryptage.hh

`std::map<char, int> calcul_occurence(std::string &mon_texte)` : Permet de calculer les occurrences du string passé en paramètre de la fonction. Cette méthode nous retourne une map ayant pour clé la lettre et pour valeur l'occurrence de la lettre.

`std::vector<ArbreB<int>> creation_racines(std::map<char, int> &map)` : Permet de créer un vecteur d'ArbreB contenant les arbres avec une simple racine pour chaque caractère avec l'occurrence de cette lettre.

`ArbreB<int> la_plus_petite(std::vector<ArbreB<int>> &vec_arbre)` : Prend en paramètre un vecteur composé d'ArbreB<int> et qui retourne l'arbre ayant la plus petite étiquette à la racine, et supprime cet arbre du vecteur.

`ArbreB<int> fusion_racines(std::vector<ArbreB<int>> &vec_arbre)` : Prend en paramètre un vecteur composé d'ArbreB<int> et qui retourne l'arbre de Huffman.

`std::string codage_texte(std::string &texte, std::map<char, std::string> &map)` : Prend en paramètres le texte à coder, une map avec les lettres et leurs codages, et retourne un string avec le texte codé.

4.6 Modification de l'interface graphique

4.6.1 classe affiche

`void Affichage::settings_according_depth()` : Permet d'ajuster la position de l'arbre dans l'interface graphique en fonction de la profondeur de l'arbre en modifiant les valeurs `ecart_feuille`, `ecart_ordonnee` et `racineX` pour proposer une lecture optimale de l'interface à l'utilisateur.

4.6.2 classe bouton

`QTextEdit *codage` : Ajout d'un nouveau QTextEdit qui nous servira à afficher le texte codé.

Le bouton `Test` a été remplacé par le bouton `Crypter`

Surpression de la `QScrollArea` du `QTextEdit texte` qui étaient inutiles.

`void Bouton::crypter()` : Cette méthode est utilisée lorsque l'on clique sur le bouton `Crypter`. Elle permet de récupérer le texte dans la QTextEdit, d'utiliser les différentes méthodes permettant de construire l'arbre d'Huffman et de coder les lettres et le texte. Affiche le résultat dans le `QTextEdit *codage` et dessine l'arbre d'Huffman correspondant.

4.6.3 classe fenetre

Modification du positionnement des Widgets pour une meilleure lisibilité de l'interface.

Les widgets sont désormais empilés au lieu d'être côte à côte.

4.7 Information sur l'affichage de l'arbre

L'affichage de l'arbre est fonctionnel, mais selon la profondeur, il est plus ou moins lisible. En effet, l'arbre augmente en taille de manière exponentielle et donc à partir d'une profondeur de 11, son affichage demande une très grande largeur pour être visible. L'affichage est donc clair et concis pour des arbres ayant une profondeur de 10 ou moins. Au-delà, l'écart entre les feuilles devient infime et donc il devient illisible. Néanmoins, le shell donne un aperçu des différentes occurrences et l'arbre est quand même dessiné pour donner une idée générale.