VAISSIERE Florian
L3 Informatique Spécialité Ingénierie Informatique
Réalisé le 30 Septembre 2018.
Semestre 5.

Projet – Codage Huffman

I-Structure

Le projet nécessite deux structures, la structure nœud et la structure arbre.

La structure nœud est la base de tout le projet car c'est d'elle d'où démarre la création du codage Huffman. En effet elle se compose d'un caractère alphabétique, d'un entier qui correspond au nombre d'occurrence du caractère, et de deux pointeurs qui indique la présence de nœuds suivant.

```
struct noeud
{
    char alphabet;
    int val;
    struct noeud* fils_droit;
    struct noeud* fils_gauche;
};
```

La structure arbre découle de la structure nœud et permet de faire un appel au premier nœud de l'arbre afin de ne mémoriser le tout premier nœud. Elle se compose d'un unique pointeur qui indique le nœud racine ou le tout premier nœud de l'arbre.

```
struct arbre
{
    nd* racine;
};
```

II-Fonctions

1-Fonctions de création

Chaque structure de donnée possède sa fonction de création qui alloue la mémoire nécessaire.

Création d'un nœud :

```
/*
*   Function creer_noeud
*   Create node.
*   Parameters:
*   <int val>   :(int)   Value we need to create new node
*   <char letter>   :(char)     Symbol we need to found
*/
nd creer_noeud(int val, char letter)
{
    nd res = (nd)malloc(sizeof(struct noeud));
    res->val            = val;
    res->alphabet       = letter;
    res->fils_droit     = NULL;
    res->fils_gauche    = NULL;
    return res;
}
```

Création d'un arbre :

```
/*
*   Function creer_arbre
*   Create tree.
*   Parameters:
*   <nd node>   :(noeud)    Value of first node
*/
ab creer_arbre(nd node)
{
    ab arb      = (ab)malloc(sizeof(struct arbre));
    nd* racine  = node;
    return arb;
}
```

2-Fonctions usuelles

Ces fonctions correspondent à la l'affichage et à la libération de la mémoire ou à la recherche d'un nœud.

Dans nœud :

```c
/*
*   Function afficher_tout
*   Print all nodes
*   Parameters:
*   <nd n>  :(noeud) Struct of node
*/
void afficher_tout_noeud(nd n)
{
    nd courant = n;
    if (courant == NULL)
    {
        printf("Noeud vide \n");
    } else {
        printf("Cle = %d Lettre = %s\n", courant->val, courant->alphabet);
        if(courant->fils_gauche)
        {
            afficher_tout_noeud(courant->fils_gauche);
        }

        if(courant->fils_droit)
        {
            afficher_tout_noeud(courant->fils_droit);
        }
    }
}

/*
*   Function detruire_tout
*   Delete all nodes
*   Parameters:
*   <nd* n> :(noeud*) Pointer on node struct
*/
void detruire_tout_noeud(nd* n)
{
    if((*n) != NULL)
    {
        if((*n)->fils_gauche != NULL)
        {
            detruire_tout_noeud((&((*n)->fils_gauche)));
        }
        if((*n)->fils_droit != NULL)
        {
            detruire_tout_noeud((&((*n)->fils_droit)));
        }

        free (*n);
        (*n) = NULL;
    }
}
```

Dans arbre :

```c
/*
*   Function detruire_arbre
*   Delete tree.
*   Parameters:
*   <ab arbre>  :(arbre)    Struct of tree
*/
void detruire_arbre(ab arbre)
{
    if(arbre != NULL)
    {
        detruire_tout_noeud(arbre->racine);

        free (arbre->racine);
        (arbre->racine) = NULL;
        free (arbre);
        (arbre) = NULL;
    }
}
```

```c
/*
*   Function trouver_dans_arbre
*   Delete tree.
*   Parameters:
*   <ab arbre>  :(arbre)    Struct of tree
*/
int trouver_dans_arbre(ab arbre, int val)
{
    nd courant = arbre->racine;
    if (arbre == NULL)
    {
        printf("Arbre vide \n");
    }
    else if (courant == NULL)
    {
        printf("Noeud vide \n");
    } else {
        if(courant->fils_droit->val > val)
        {
            if (courant->fils_gauche->val != val)
            {
                courant = courant->fils_gauche;
            } else {
                return 1;
            }
        }
        else
        {
            if (courant->fils_droit->val != val)
            {
                courant = courant->fils_droit;
            } else {
                return 1;
            }
        }
    }
}
```

3-Fonctions d'Huffman

Ces fonctions gèrent le calcul du préfixe, le calcul de la fréquence de chaques occurrences, la création du code de chaques lettres, la compression et la décompression du texte.

Ces fonctions s'appellent entre elles donc si l'on souhaite compresser un texte il suffit de créer l'arbre binaire en appelant la fonction *assembler_arbre* qui prend en paramètre la fréquence et génère un arbre binaire pour le texte et ensuite appeler la fonction compresse en passant l'arbre généré ainsi que les fichiers de lecture et d'écriture afin de pouvoir observer l'encodage du texte. Pour décompresser c'est la même procédure sauf que vu que l'on a déjà l'arbre nous n'avons pas besoin de répéter la création de l'arbre.

```c
int main(int argc, char* argv[])
{
    FILE* fichierL = NULL;
    FILE* fichierE = NULL;

    char choix = fscanf(stdin, "Voulez vous utiliser un fichier ou l'entré stand
    if(choix == "s")
    {
        fichierL = fopen("lecture.txt", "w");
        char compresse = fscanf(stdin, "Que désirez-vous compresser?");
        fprintf(fichierL, "%c\n", compresse);
        fclose(fichierL);
    }

    ab arbre = assembler_arbre(frequence(fichierL));
    compresse(arbre, fichierL, fichierE);
    decompresse(arbre, fichierL, fichierE);

    return 0;
}
```

Dans ce *main* comme on peut le voir on peut aussi écrire directement depuis le terminal sinon c'est une lecture de fichier qui aura lieu.

Conclusion :

La plus grande difficulté fut d'assembler l'arbre binaire et de respecter les poids des nœuds intermédiaires ainsi que la généricité.