

## Séances 1 et 2 : table de codage, et construction de l'arbre de Huffman

L'algorithme de construction de l'arbre de Huffman est donnée ci-dessous :

- Étape 0 : chaque valeur d'octet à coder, et son nombre d'occurrences, est représentée par un arbre, composé au début d'un seul nœud. Les arbres sont triés dans une liste chaînée par ordre croissant du nombre d'occurrences.
- Étape 1 : Les deux premiers nœuds de la liste, c'est à dire les arbres représentant le moins d'occurrences, sont rattachés à un nouveau nœud en tant que nœuds fils. Le nombre d'occurrences du nouveau nœud père est la somme du nombre d'occurrences de ses fils.
- Étape 2 : Le nœud père est inséré dans la liste triée.
- Les étapes 1 et 2 sont ré-itérées jusqu'à l'obtention d'un arbre unique.

La figure 2 montre une itération de la construction de l'arbre.

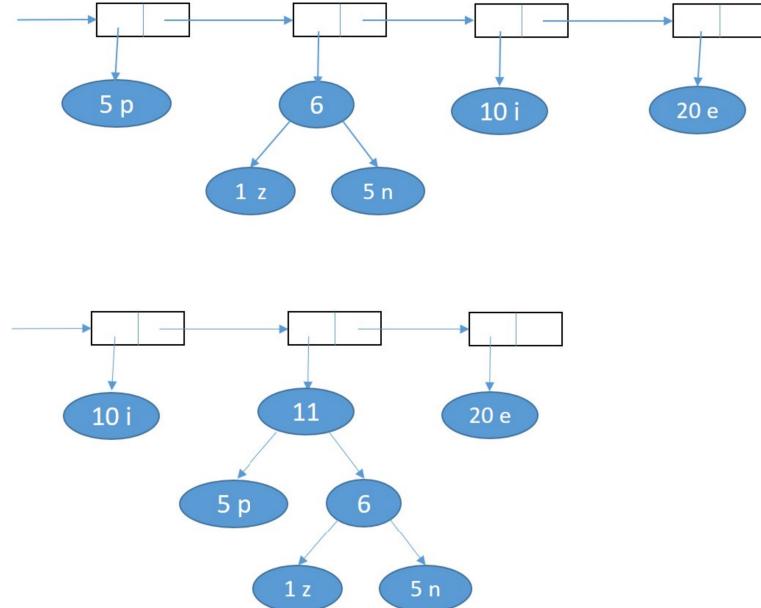


FIGURE 2 – Une étape de la construction de l'arbre.

L'arbre obtenu à la fin du processus est représenté figure 3. Les feuilles sont associées aux valeurs d'octet à coder. Le code correspondant est construit à partir du chemin qui relit la racine de l'arbre à la feuille : un passage par le fils gauche se traduit par un 0, et un passage par le fils droit par un 1.

Notez que, dans le codage obtenu, les préfixes d'un code ne peuvent pas correspondre au code complet associé à une autre valeur.

**Travail demandé** Ce premier module logiciel construit l'arbre des codes d'Huffman et la table de codage des octets. L'interface publique du module est donnée ci-dessous.

La table de codage des octets est un tableau de 256 éléments ; chaque élément définit la valeur d'octet associé (champ `byte`), le nombre d'occurrences de la valeur (champ

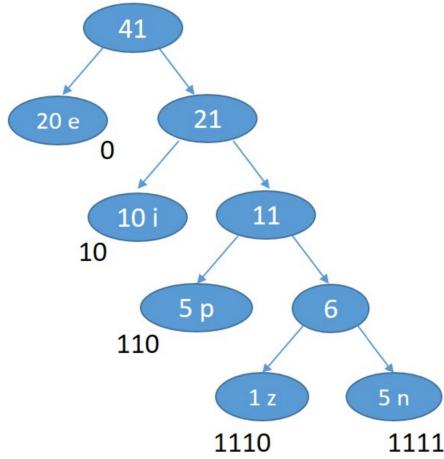


FIGURE 3 – L’arbre d’Huffman.

`occurrence`), et après construction du code, le code d’Huffman correspondant (`huffmanCode`) et sa taille en bits (`nbBits`).

À la fin de sa construction, les feuilles de l’arbre pointent vers les éléments de la table de codage (voir la figure 4).

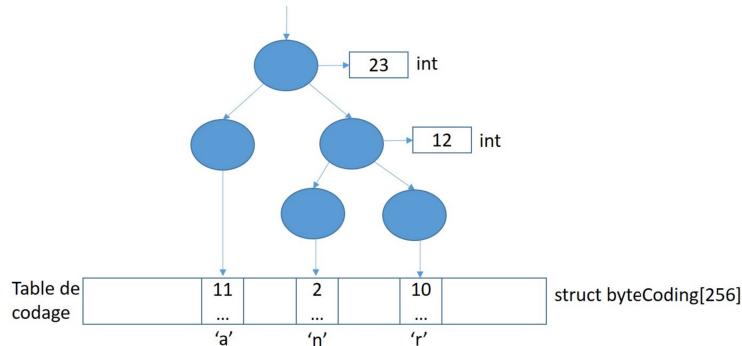


FIGURE 4 – Relation arbre-table de codage.

## Étapes du travail

1. Définir la structure `tree_node`.
2. Écrire les fonctions listées dans l’interface publique, en respectant les prototypes indiqués. L’implantation de listes chaînées nécessaire à la construction s’appuiera sur un module `liste` qui a déjà été développé (fichier objet et entête sont disponibles sur la page moodle du projet). L’interface du module `liste` est reproduite ci-dessous.
3. Écrire un test unitaire du module. Dans votre test, vous devrez entre autre vérifier la propriété sur les préfixes de codes énoncée ci-dessus.

```

// codage des valeurs d'octet
struct byteCoding {
    int occurrence;
    t_byte byte;
    unsigned int huffmanCode;
    unsigned int nbBits;
};

// arbre de huffman
struct tree_node {
    // a completer
};

// gestion du tableau de codage des octets
void tree_resetByteOccurrence(struct byteCoding indexedCodeTable[256] ;
    // remet a zero le nombre d'occurrences des octets
void tree_resetByteCoding(struct byteCoding indexedCodeTable[256]) ;
    // remet a zero les codes octet
void tree_countByteOccurrence( const t_byte * buffer ,
    struct byteCoding indexedCodeTable[256]);
    // compte de nombre d'occurrences des octets dans un tampon

void tree_displayByteOccurrence(struct byteCoding indexedCodeTable[256]);
    // affiche la table de codage des valeurs d'octet pour debug

/* gestion de l'arbre de Huffman */
struct tree_node * tree_createNode(struct tree_node * father , int * value) ;
    // creation d'un noeud
struct tree_node * tree_createCodingNode (struct tree_node * left ,
    struct tree_node * right);
    // creation d'un noeud de codage et liaison avec ses fils
struct tree_node * tree_create (struct byteCoding * indexedCodeTable);
    // creation de l'arbre
void tree_destroy( struct tree_node * root );
    // destruction d'un arbre et liberation des donnees
    // (sauf au niveau des feuilles)
void tree_buildHuffmanCode(struct tree_node * root , int level , int code) ;
    // construit les codes de huffman en parcourant l'arbre
void tree_display( struct tree_node * root , int level ) ;
    // affichage de l'arbre

```

**Module Liste** Le module liste gère une liste simplement chaînée. Les données enregistrées dans les nœuds sont des pointeurs génériques (`void *`). La gestion mémoire des objets pointés n'est prise en charge dans le module.

Le module apporte les fonctionnalités suivantes :

- création d'une liste,
- lecture et écriture de la donnée d'un nœud,
- insertion d'une donnée en tête de liste,
- ajout d'une donnée en queue de liste,
- suppression d'une donnée en tête de la liste,
- suppression de la première instance d'une donnée dans la liste,
- insertion d'une donnée dans une liste ordonnée (une fonction fournie en paramètre)

est appelée pour comparer deux données),  
— destruction d'une liste complète.  
L'interface publique du module est donnée ci-dessous.

```
typedef struct _list_node s_node;
s_node * list_create(void);
    // creation d'une nouvelle liste vide
void * list_get_data(s_node * node);
void list_set_data(s_node * node, void * data);
    // lire ou ecrire la donnee d'un noeud
s_node * list_insert(s_node * head, void * data);
    // creation et insertion d'un noeud en tete de liste
    // retourne la tete de liste
s_node * list_append(s_node * head, void * data);
    // creation et ajout d'un noeud en queue de liste
    // retourne la tete de liste
s_node * list_orderedAppend(s_node ** head,
                            int (*compare)(s_node *, void *),
                            void *param);
    // ajout d'un noeud dans une liste ordonnee
    // selon le resultat de "compare" ;
    // si la donner existe deja, elle n'est pas ajoutee
s_node * list_remove(s_node * head, void * data);
    // suppression de la premiere instance d'une
    // donnee dans la liste, retourne la tete de liste
s_node * list_headRemove(s_node * head);
    // suppression de la tete de liste
    // retourne la nouvelle tete de liste
void list_destroy(s_node * head);
    // destruction d'une liste
    // (La libération des données n'est pas prise en charge)
s_node * list_next(s_node * list);
    // noeud suivant de la liste
```