IMAC
INGÉNIEUR
ESIPE

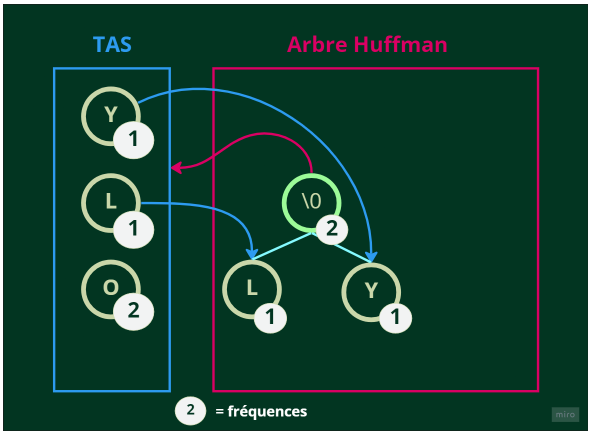
Algorithmique Avancé

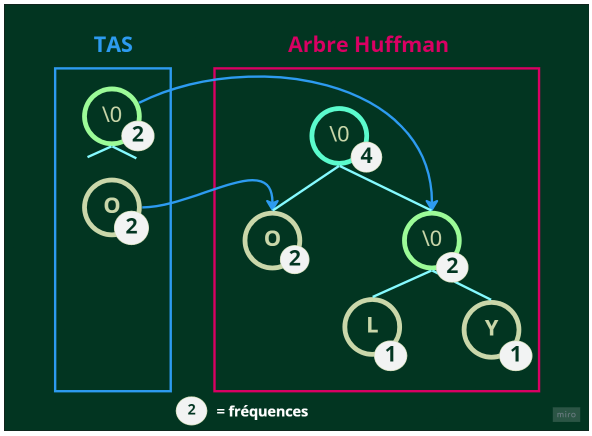
TP 4

Tas et Codage de Huffman

1 TP - Exercice 2

Pour l'exercice 2, vous allez implémenter un codage de Huffman. C'est à dire encoder une chaine de caractères pour obtenir un code composé de '0' et de '1'. Puis faire le processus inverse, c'est à dire à partir de ce code, réobtenir la chaine originale.
Pour ce faire vous allez devoir créer un arbre de Huffman et pour en obtenir un efficient, vous allez utiliser un Tas, un Tas-minimum plus précisément, qui va classer les caractères de la chaine selon leur occurrences, en ayant celui avec le moins de fréquence en tête de Tas. Avec ce tas, vous allez prendre 2 par 2 les éléments les plus petits pour en faire des sous-arbre avec un nouveau noeud parent qui va être réinséré dans le tas. On répète le processus jusqu'à ne plus avoir de couple de caractères dans le tas.





Pour y arriver, implémentez un certains nombres de fonctions décrites plus bas.
Pour faire fonctionner ces fonctions vous aurez besoin d'une structure *HuffmanHeap* et *HuffmanNode*.
HuffmanNode est un noeud d'arbre possédant les attributs $character$, $value$ et $code$ qui sont respectivement le caractère représentant le noeud, la fréquence d'apparition de ce caractères et le code produit par le dictionnaire.
HuffmanHeap est un tas de *HuffmanNode* utilisant la fréquence d'un caractères pour les ordonner.

- **processCharFrequences**(string $data$, Array $frequences$) : Rempli chaque case i de $frequences$ avec le nombre d'apparition du caractère correspondant au code ASCII i dans la chaine de caractère $data$.
- **insertHeapNode**(HuffmanHeap $heap$, int $heapSize$, HuffmanNode* $node$) : Insère un nouveau HuffmanNode dans le tas-minimum en utilisant la $frequence$ comme priorité.
- **buildHuffmanHeap**(Array $frequences$, HuffmanHeap $heap$) : Construit un tas $heap$ minimum à partir des fréquences d'apparition non nulles de caractères. Un tas minimum est un tas qui donne la priorité aux valeurs les plus basses $\rightarrow$ chaque nœud est plus petit que ses fils.
- **heapify**(HuffmanHeap $heap$, int $heapSize$, int $nodeIndex$) : répare un tas en partant de l'index donné.
- **extractMinNode**(HuffmanHeap $heap$, int $heapSize$) : récupère la plus petite valeur (la tête normalement), l'enlève du tas et répare le tas après l'extraction.
- **makeHuffmanSubTree**(HuffmanNode* $right$, HuffmanNode* $left$) : Construit un sous arbre de huffman à partir de deux nodes.
- **buildHuffmanTree**(HuffmanHeap $heap$, HuffmanNode* $tree$) : Construit un dictionnaire de Huffman.
- **processCodes**(HuffmanNode $tree$) : Détermine et définit les codes de toutes les feuilles de $tree$
- **huffmanEncode**(string $toEncode$, HuffmanNode* $huffmanTree$) : Retourne la chaine de caractère encodé à partir des codes se trouvant dans $characters$. $characters$ est un tableau de HuffmanNode*, chaque indice i correspond au code ASCII i.
- **huffmanDecode**(HuffmanNode* $huffmanTreeRoot$, string $toDecode$) : Retourne la chaine de caractère décodé partir du dictionnaire $dict$

Si vous êtes chaud patate, renvoyez la chaine de caractère encodé sous forme binaire plutôt que sous forme de caractère. Le but étant de compresser vous devez utilisez un octet plus stocker plusieurs caractères. N'hésitez à appeler votre chargé de TD préféré pour avoir plus d'informations (parce que la flemme d'expliquer par écrit). Vous pouvez utiliser le langage que vous souhaitez.

1.1 C++

Le dossier *Algorithme_TP4/TP* contient un dossier *C++*. Vous trouverez dans ce dossier des fichiers *exo<i>.pro* à ouvrir avec *QtCreator*, chacun de ces fichiers projets sont associés à un fichier *exo<i>.cpp* à compléter pour implémenter les différentes fonctions ci-dessus.
L'exercice *exo1.cpp* implémente une structure *Heap* possédant les différentes méthodes d'un tas à implémenter.
Cette structure est une spécialisation de *Array*, il possède donc les mêmes fonctions d'accès que lui.

```
class Heap : public Array {
```

```
  void print(); // declaration de la methode print de Heap

}

void Heap::print() // corps de la methode print de Heap
{
  for (i=0; i < this->size(); ++i)
    printf("%d ", this->get(i));
}

void Heap::clear() // corps de la methode clear de Heap
{
  for (i=0; i < this->size(); ++i)
    this->set(i, 0);
}
```

HuffmanHeap est un tas qui plutôt de stocker des entier, stocker des HuffmanNode.
HuffmanNode est un noeud, comme *BinaryTree* dans le TP3, il possède un enfant gauche $left$ et droit $right$ du même type que lui, ces deux enfants peuvent donc utiliser les mêmes méthodes que *HuffmanNode*.

```
struct HuffmanNode {

  HuffmanNode* left;
  HuffmanNode* right;
  int value;
  char character;
  string code;

  void print()
  {
    if (this->left != nullptr)
      printf("left: %d with code: %s\n", this->left->value, this->left->code);
    if (this->right != nullptr)
      printf("right: %d with code: %s\n", this->right->value, this->right->code
          );
    printf("this: %d\n", this->value);
  }

}
```

Notes :

- Dans une fonction C_{++}, si le type d'un paramètre est accompagné d'un '&' alors il s'agit d'un paramètre entré/sortie. La modification du paramètre se répercute en dehors de la fonction.
- La fonction $huffmanDict$ a pour paramètre un $HuffmanNode * \&$, il s'agit d'un pointer dont vous pouvez modifier l'adresse vers laquelle il pointe.
- Lorsque vous faites appel à $this$ dans une méthode d'une structure (ou d'une classe), vous pouvez accéder aux attributs de la structure en question, comme dans l'exemple ci-dessus.