IMAC

TP Algorithmique 4

Tas et Codage de Huffman

Le **tas** est une structure binaire basé sur la priorité que l'on peut implémenter à l'aide d'un tableau qu'on appel aussi **file de priorité** (*Priority Queue*).

1 Tas

1.1 Principe

Il s'agit d'une structure binaire comme l'arbre de recherche mais contrairement à lui les éléments les plus grands ne se trouvent pas dans la partie de droite mais dans les niveaux les plus hauts. Chaque noeud est **plus grand que ses enfants**.
Le tas est aussi un **arbre complet**, chaque noeud qui n'est pas une feuille a exactement deux enfants sauf ceux du niveau $h-1$ (l'avant-dernier). Lorsqu'on insère un noeud dans le tas, on doit d'abord compléter le noeud $h-1$ incomplet le plus à droite. C'est cette propriété qui nous permet d'utiliser

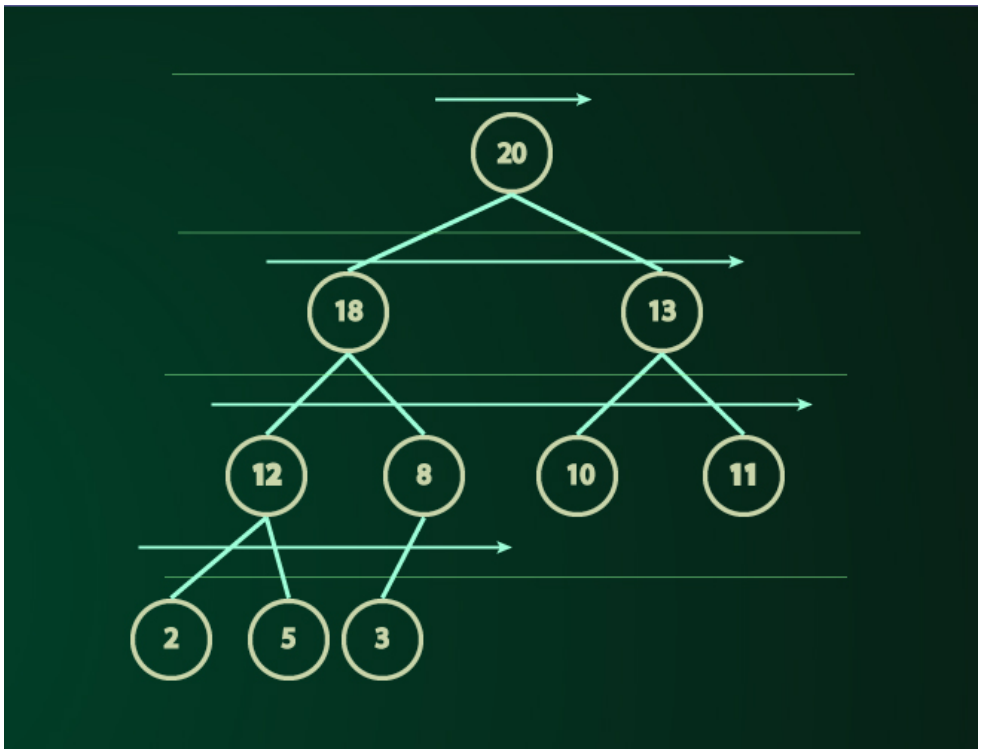

FIGURE 1 – Exemple de tas

un tableau pour représenter cette structure. Un arbre complet peut être lu comme un livre, de haut en bas et pour chaque ligne de gauche à droite. Chaque noeud à l'indice i peut avoir un fils gauche à l'indice $i \times 2 + 1$ et un fils droit $i \times 2 + 2$. Pour l'exemple ci-dessus, on obtient le tableau suivant $[20, 18, 13, 12, 8, 10, 11, 2, 5, 3]$.

1.2 Algorithme

Pour créer un tas, on insère progressivement chaque noeud. Pour chaque noeud à insérer :

Algorithm 1 Insertion d'un noeud dans un tas

heap $\Leftarrow$ Tas de taille n
value $\Leftarrow$ Valeur à insérer
i $\Leftarrow n$
heap[i] $\Leftarrow$ value
Tant que *i* > 0 et *heap[i]* $>$ *heap[* $\frac{i}{2}$ *]* **faire**
    $swap$(*heap[i]*, *heap[* $\frac{i}{2}$ *]*)
    i $\Leftarrow \frac{i}{2}$
fin Tant que

Une variante de cet algorithme permet de mettre à jour un nœud d'un tas pour qu'il reste plus grand que ses enfants.

Algorithm 2 Mis à jour d'un tas à partir d'un noeud ($heapify$)

heap $\Leftarrow$ Tas de taille n
i $\Leftarrow$ indice de la racine
largest $\Leftarrow$ indice de la valeur la plus grande entre le nœud i et ses enfants
Si *largest* $\neq$ *i* **Alors**
    $swap$(*heap[i]*, *heap[largest]*)
    $heapify$(heap, n, largest)
fin Si

Cette algorithme ne fonctionne que si les enfants du nœud en question sont des tas ou des feuilles. A partir de cet algorithme et du postulat que la racine d'un tas est toujours la plus grande valeur, on peut concevoir un tri.

Algorithm 3 Tri par tas

heap $\Leftarrow$ tas créé à partir d'un tableau de n nombres aléatoires
Pour *i* allant de $n - 1$ à 0 **faire**
    $swap$(*heap[0]*, *heap[i]*)
    $heapify$(*heap*, *i*, 0)
fin Pour

2 Codage de Huffman

Le codage de Huffman est un moyen de compresser des données. Un caractère est un entier non-signé codé sur 8 bits (1 octet). Cet entier est un peu comme un identifiant qui permet à un programme de savoir quel caractère il doit afficher. David a proposé un moyen pour identifier un caractère avec moins de 8 bits.
Cette méthode de compression se base sur un arbre binaire qui va servir de dictionnaire. C'est cette arbre qui va savoir quel caractère correspond à quel code.

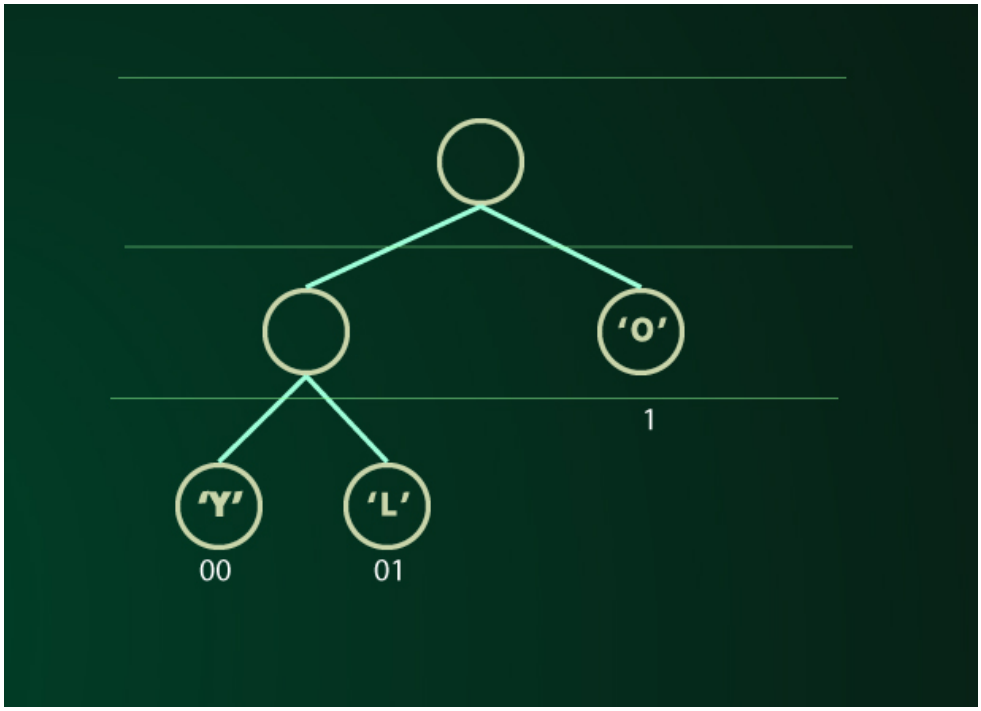

FIGURE 2 – Exemple de dictionnaire de Huffman pour "YOLO"

Prenons un cas simple, "YOLO", ici l'arbre a trois feuilles correspondant chacune à une lettre ayant une occurrence dans notre chaine de caractères. Chaque feuille a un code ("1, "01", "00"), ce code représente le chemin parcouru pour arriver jusqu'à la feuille en question. Pour décoder une chaine codée, on part de la racine et on regarde la composition du code, pour chaque **'1'** rencontré **on choisi l'enfant de droite sinon on choisi celui de gauche**, lorsqu'on tombe sur une feuille on obtient un caractère et on repart de la racine.
Ainsi en utilisant le dictionnaire ci-dessus, le décodage de "001011" donne "YOLO".
Bien sûr, ici on a un cas assez simple où la chaine de caractère crée un dictionnaire avec des codes de 2 bits, mais prenons le pire cas possible où chaque lettre du codage ASCII est représenté, on se retrouve avec **256 feuilles et donc** $log_2(256)$ **niveaux soit 8 niveaux**. Donc dans le pire des cas on se retrouve avec un codage sur 8 bits qui n'offre aucune compression. On peut cependant crée un dictionnaire optimal qui permet de trouver rapidement un caractère avec une occurrence forte. Dans

l'exemple de "YOLO", le 'O' se trouve un niveau au dessus que les autres du fait de son occurrence de 2, ainsi les 'O' de "YOLO" sont codés sur 2 bits plutôt que 4.

3 TP

Implémenter les méthodes de la structure $Heap$ pour construire un tas et implémenter un tri par tas :

- **leftChild**(int $nodeIndex$) : retourne l'index du fils gauche du nœud à l'indice $nodeIndex$.
- **rightChild**(int $nodeIndex$) : retourne l'index du fils droit du nœud à l'indice $nodeIndex$.
- **insertHeapNode**(int $heapSize$, int $value$) : insère un nouveau noeud dans le tas $heap$ tout en gardant la propriété de tas.
- **heapify**(int $heapSize$, int $nodeIndex$) : Si le noeud à l'indice $nodeIndex$ n'est pas supérieur à ses enfants, reconstruit le tas à partir de cette index.
- **buildHeap**(Array $numbers$) : Construit un tas à partir des valeurs de $numbers$ (vous pouvez utiliser soit **insertHeapNode** soit **heapify**)
- **heapSort**() : Construit un tableau trié à partir d'un tas $heap$

Implémenter les fonctions suivantes pour implémenter un codage de Huffman :

- **charFrequences**(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 $data$.
- **huffmanHeap**(Array $frequences$, Heap $heap$) : 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.
- **huffmanDict**(Array $heap$, HuffmanNode* $tree$) : Construit un dictionnaire de Huffman en suivant les règles suivantes :
  - — Chaque feuille décrit un caractère et le nombre d'occurrence de ce caractère.
  - — Chaque parent décrit un caractère nul ('\0') et la somme des occurrences de ses enfants.
  - — L'insertion d'un noeud n répond à l'une de ces situations :
    - — Si $tree$ est un pointer nul, $tree$ pointe vers n.
    - — Si $tree$ est une feuille, $tree$ pointe vers un nouveau noeud ayant pour fils $tree$ et n. L'enfant gauche doit être plus grand que l'enfant droit.
    - — Si n est deux fois plus grand que $tree$, $tree$ pointe vers un nouveau noeud ayant pour fils droit $tree$ et n pour fils gauche.
    - — Si n est plus petit que le fils gauche de $tree$, on l'insère dans le fils droit.
    - — Sinon on l'insère dans le fils gauche.
- **huffmanEncode**(HuffmanNode* $dict$, string $toEncode$) : Retourne la chaine de caractère encodé à partir du dictionnaire $dict$
- **huffmanDecode**(HuffmanNode* $dict$, 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.

3.1 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));
}
```

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.
- Vous pouvez utiliser la méthode createNode(int $value$) pour créer un nouveau nœud.