

Travaux Dirigés sur Machine n°6 — Tas, tri et Dijkstra — 2x2h noté

Tout le nécessaire est contenu dans l'archive `TDM6.tgz` (fichiers de base, de test et sorties attendues).

Attention, le TDM est en deux séances et il faut remonter quelque chose à la fin de chaque séance :

- la première remise doit contenir le code complet pour le tas et
- la seconde remise doit tout contenir (dont le CR)

Entre les deux séances, un code pour le module `heap` sera fourni afin d'aborder l'algorithme de Dijkstra lors de la seconde séance.

1 Structure de tas

Un tas est une structure de données qui stocke un ensemble d'éléments comparables et qui permet de :

- extraire le minimum en temps logarithmique,
- ajouter un élément en temps logarithmique et
- mettre à jour la position d'un élément en temps logarithmique.

L'idée principale est de travailler avec un arbre équilibré où chaque nœud contient une valeur plus grande que celle de son père. La valeur minimale est donc à la racine. De plus, il n'y a pas de *trou* dans l'arbre : tous les niveaux sont complets, sauf le dernier où toutes les feuilles sont à gauche. Cela est illustré sur la Fig. 1(a).

L'idée complémentaire est de replier cet arbre dans un tableau (Fig. 1(b)) ce qui permet une navigation simple. Les nœuds sont rangés par profondeur dans le tableau : d'abord la racine, puis ses fils, puis les fils de ses fils... à chaque fois le niveau est rangé de gauche à droite.

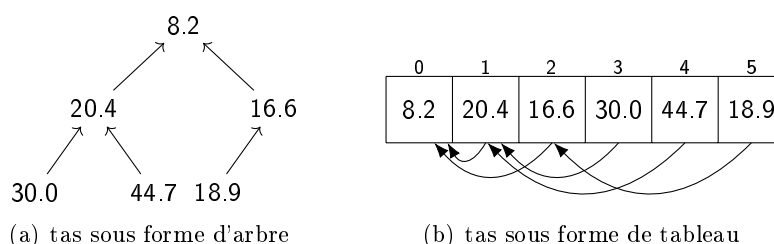


FIGURE 1 – Structure de tas.

Quand on ajoute une valeur à un tas, elle est ajoutée à gauche dans la partie libre du dernier niveau, soit la première case vide du tableau. Ensuite, tant qu'elle est plus petite que celle du père, les deux valeurs sont échangées et on remonte dans l'arbre.

Quand on enlève la première valeur, on la remplace par celle la plus à droite dans le dernier niveau, c.-à-d. la dernière valeur du tableau. À chaque fois on regarde ses deux fils, tant qu'il y en a un qui contient une valeur plus petite, on l'échange avec la plus petite des trois et on descend.

Répondre dans le compte-rendu aux questions suivantes :

- 1 Que deviennent l'arbre et le tableau de la Fig. 1 quand on ajoute la valeur 12.2 puis la valeur 7.3 ?
- 2 Que deviennent l'arbre et le tableau de la Fig. 1 (après les ajouts de 1) quand on retire le minimum ?
- 3 Expliquer pourquoi l'extraction du minimum est en temps logarithmique alors que connaître sa valeur est en temps constant.
- 4 Quelle est l'expression mathématique donnant le numéro de la case du tableau stockant le fils gauche d'un nœud stocké dans la case i ? De même pour le fils droit. De même pour le père.

Écrire le code des méthodes dans `heap.hpp` (à tester avec `test_heap`). Les méthodes les plus élémentaires sont définies dans la classe, les autres en dehors. Il faut utiliser au maximum les méthodes élémentaires afin de simplifier l'écriture de la version avancée (`heap_id` plus bas).

Le code utilise des `template` (patrons) bien qu'ils n'aient pas été vus en cours. Cela ne devrait pas poser de problème car toutes les déclarations attenantes sont données. À l'intérieur des patrons, il faut considérer les

paramètres (des types ici) comme définis et utilisables (comme des arguments d’une fonction qui n’ont de valeur que lors d’un appel). Une des particularités des **template** est que leur code doit être disponible à la compilation (avant l’édition de liens) et donc être présent dans le `.hpp`.

Illustration simple : tri

Le programme de test réalise un tri en mettant tout dans un tas puis en retirant tout (dans l’ordre). Il est essayé sur deux types pour les données.

- 5 Expliquer dans le compte-rendu la complexité de ce tri.

2 Algorithme de Dijkstra

Il s’agit de l’algorithme classique de *recherche du plus court chemin*. Il n’est pas rappelé ici, mais il se trouve sur internet, dans les manuels classiques ou dans votre cours de réseau.

2.1 Graphes

Un graphe est un ensemble de sommets (**vertex/vertices**) reliés par des arêtes (**edges**). Chaque arête a une distance. Cela se représente sur un dessin comme sur la Fig.2.

La classe **graph** est fournie en entier à l’exception de la méthode `print_dijkstra` qui calcule un plus court chemin par l’algorithme de Dijkstra et l’affiche.

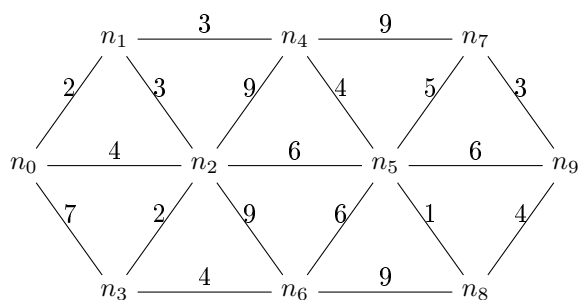


FIGURE 2 – Exemple de graphe avec des distances.

- 6 Donner dans le compte-rendu le plus court chemin de n_0 à n_9 .

Avant de programmer, il faut être clair sur les différentes structures de données annexes nécessaires à chaque niveau. En effet, si les concepts sont simples, leur implantation n’est pas forcément évidente et il y a beaucoup d’indirections. Pour garder des complexités logarithmiques, il faut des tableaux annexes pour retrouver la place dans le tas ou d’autres informations pour ne pas avoir à parcourir en entier la moindre structure.

2.2 Aménagement du tas

Il faut pouvoir intervenir sur n’importe quelle valeur contenue dans le tas et demander au tas de la repositionner correctement (en la montant ou en la descendant). La classe `heap_id` permet cela en rajoutant un tableau `id_to_pos` permettant de lier les éléments à leur position dans le tas. Elle utilise également un autre tableau pour gérer les `id` déjà attribués à des éléments et les `id` non utilisés.

Ainsi à chaque fois que l’on fait un `push`, un `unsigned int` est retourné. Celui-ci sert d’identifiant `id` pour l’élément rajouté dans le tas. Attention, cette valeur ne bouge pas alors que la position du nœud dans le tableau varie. Pour toute opération sur le tas `push`, `pop`... il faut faire attention à bien mettre à jour `id_to_pos` par rapport aux nouvelles positions des nœuds dans le tas.

La classe `heap_id` utilise donc le tableau pour faire la correspondance entre cet `id` et la position dans le tableau. De même, elle utilise un autre tableau pour gérer les `id` non utilisés.

- 7 Expliquer dans le compte-rendu le sur-coût de complexité lié à la mise en place de ces tableaux. Donner des exemples de sur-coût sur des méthodes s’il n’y avait pas ces tableaux.

8

Expliquer dans le compte-rendu ce qui doit être fait en plus pour vérifier que la structure est valide.

Écrire le code des méthodes dans `heap_id` (à tester avec `test_heap_id`). Attention, certaines peuvent faire l'objet de simple copier-coller de `heap`, d'autres non !

2.3 Implantation de la méthode `print_dijkstra`

Écrire le code de la méthode `print_dijkstra` dans `graph.cpp` (à tester avec `test_graph`). Il peut y avoir encore un tableau à faire pour passer entre les données du graphe et celle du tas.

Attention c'est de loin le code le plus complexe du TDM et c'est le moins documenté...