TP n°23 - Tas et files de priorité

Exercice 1 (Implémentation des arbres binaires complet par des tableaux).

1. Redonner la définition d'un arbre binaire complet. Dessiner un arbre complet à 10 nœuds avec des étiquettes aléatoires comprises entre 1 et 99.
2. Expliquer comment un arbre binaire complet peut avantageusement être stocké dans un tableau. Dessiner le stockage de l'arbre ci-dessus dans un tableau.
3. On stocke dans la case i du tableau l'étiquette d'un nœud. Quelle est l'indice de la case contenant l'étiquette de son père ?
4. Quelle sont les indices des cases contenant les étiquettes de ses fils ? Quelle précaution doit-on prendre avant d'aller chercher les étiquettes des fils ?

Définition 1 (Tas)

Un **tas** t (*heap* en anglais) est :

- soit l'arbre vide,
- soit un **arbre binaire complet** étiqueté dont les étiquettes sont à valeurs dans un ensemble $(E, \preceq)$ **totalement ordonné**.

De plus, si cet arbre est non vide, alors $t = N(\ell, x, r)$ et $x \in E$ est **supérieur ou égal** aux étiquettes de tous les nœuds de ℓ et de r pour la relation d'ordre $\preceq$ sur E.

Nous allons implémenter les primitives de manipulation d'un tas :

- `heap_create` (constructeur)
- `heap_free` (destructeur)
- `heap_insert` (transformateur, insertion d'une donnée)
- `heap_remove` (transformateur, suppression d'une donnée)

Pour ce TP, **nous nous contenterons d'implémenter des tas à étiquettes entières**. Mais bien entendu, on peut stocker des objets de type (clé, valeur), tant que les clés appartiennent à un ensemble totalement ordonné. Dans ce cas, toutes les opérations de comparaison se font uniquement sur les clés.

Toutes les fonctions de ce TP seront codées dans un fichier `NOM_heap.c`.

Exercice 2 (Création et insertion dans un tas).

1. Proposer un type implémentant une structure de données de tas avec un tableau. On nommera `heap` cette structure de données.
2. Écrire une fonction `heap_create` qui initialise un tas vide et renvoie un objet de type `heap *`.
3. Écrire une fonction `heap_free` qui libère tout l'espace mémoire alloué au stockage d'un tas.
4. Écrire une fonction `heap_print` qui affiche les étiquettes du tas dans l'ordre de stockage du tableau.
5. Écrire une fonction **récursive** `heap_print_infixe` qui affiche les étiquettes du tas dans l'ordre infixe.
6. Le principe de l'insertion d'un nœud d'étiquette v dans un tas est simple : on crée un nouveau nœud avec cette étiquette v tout en bas à droite du tas, puis on fait remonter cette étiquette tant que la propriété de tas n'est pas rétablie.
   a. Construire le tas obtenu en insérant successivement les valeurs suivantes, dans cet ordre : $8, 7, 2, 10, 1, 5, 3, 4$.
   b. Justifier en quelques phrases pourquoi la propriété est respectée avec cet algorithme de remontée.
   c. Écrire une fonction `bool heap_is_full(heap *t)` permettant de vérifier si la structure de données est pleine.
   d. Écrire une fonction `void move_up(heap *t, int i, elt_type v)` qui place l'étiquette v dans la case d'indice i du tableau du tas, et qui remonte cette valeur dans le tas jusqu'à ce que la propriété de tas soit rétablie.
   e. En déduire une fonction `void heap_insert(heap *t, elt_type v)` qui insère un nouveau nœud d'étiquette v dans un tas.
   f. Écrire une fonction `heap *heap_create_from_tab(elt_type *tab, int n)` qui renvoie un tas obtenu par insertion successive des valeurs du tableau `tab` à `n` éléments. On veillera à limiter les allocations mémoires : le tableau des valeurs n'est alloué et stocké qu'une seule fois ! Pour cela, on pourra rajouter dans la structure de données du tas un booléen `is_tab_allocated_outside` qui indique si le tableau a été alloué pour un constructeur ou s'il a été alloué à l'extérieur et fourni en paramètres d'entrée. Adapter la fonction `heap_free` en conséquence.
   g. Tester votre fonction d'insertion sur le cas détaillé sur papier à la question ci-dessus, où l'on a inséré dans l'ordre $8, 7, 2, 10, 1, 5, 3, 4$.

Exercice 3 (Suppression dans un tas et tamisage).

Pour supprimer un nœud dans un tas, on va remplacer son étiquette par celle du nœud tout en bas à droite du tas, puis faire descendre cette étiquette dans l'arbre jusqu'à ce que la propriété de tas soit rétablie, en n'oubliant pas de diminuer le nombre de nœuds du tas de 1. Cette opération est parfois poétiquement appelée *tamisage* car les petits nombres passent à travers le tas pour retomber en bas, comme lorsque l'on tamise de la terre.

1. Écrire une fonction `elt_type move_down(heap *t, int i, elt_type v)` qui remplace l'étiquette stockée à l'indice i par v et effectue le tamisage. La fonction renvoie la valeur v qui a été remplacée.
2. Pour supprimer un nœud dans un tas, on va remplacer son étiquette par celle du nœud tout en bas à droite du tas, puis tamiser cette valeur jusqu'à ce que la propriété de tas soit rétablie, en n'oubliant pas de diminuer le nombre de nœuds du tas de 1.
3. Écrire une fonction `elt_type heap_remove(heap *t, int i)` qui supprime l'étiquette stockée à l'indice i du tas tout en maintenant la propriété de tas. La fonction renvoie la valeur de l'étiquette qui a été supprimée.
4. Tester votre fonction en supprimant la racine (10) du tas utilisé pour tester la fonction d'insertion à l'exercice précédent.

Définition 2 (File de priorité)

Une file de priorité (*priority queue* en anglais, ou *pqueue*) est une structure de données abstraite dans laquelle on stocke des données issues d'un ensemble $(E, \preceq)$ totalement ordonné, et que l'on muni uniquement de deux primitives de manipulation :

— une fonction `pqueue_enqueue` qui ajoute un élément dans la file de priorité, avec une étiquette indiquant sa priorité : plus l'étiquette est grande vis-à-vis de la relation d'ordre $\preceq$, plus l'élément est prioritaire ;

— une fonction `pqueue_dequeue` qui retire de la file l'élément de priorité maximale.

Les files de priorités ont de nombreuses applications et apparaissent dans de nombreux algorithmes : on peut par exemple les utiliser dans la mise en œuvre de l'algorithme de Dijkstra ou de l'algorithme de Kruskal [1].

Exercice 4 (Tas et files de priorité).

1. Que peut-on dire de la racine d'un tas ? Expliquez pourquoi la structure de données de file de priorité peut être implémentée par un tas. Créer un alias `pqueue` sur le type tas crée au premier exercice.
2. En déduire immédiatement une implémentation de la fonction `void pqueue_enqueue(pqueue *q)` utilisant une fonction déjà implémentée.
3. Pour implémenter la primitive `pqueue_dequeue`, quel nœud du tas doit être supprimé ? Implémenter cette primitive en utilisant une fonction déjà implémentée.

Une structure de données de type file de priorité peut également être utilisée pour trier des données : on met tous les éléments à trier dans une file de priorité pour ensuite tous les extraire du plus grand au plus petit pour la relation d'ordre $\preceq$.

Définition 3 (Tri en place)

On dit qu'un tri est **en place** s'il ne nécessite aucune sauvegarde de toute ou partie de la structure de donnée initiale pour s'exécuter correctement. Le tri est effectué dans la structure de donnée initiale elle-même sans recours à un stockage externe.

Définition 4 (Tri stable)

On dit qu'un tri est **stable** s'il préserve l'ordre de deux éléments ayant la même clé.
Cette notion n'a réellement d'intérêt que dans le cas où l'on trie un série d'éléments de type (clé, valeur) par rapport à une relation d'ordre sur les clés.

Exercice 5 (Tri par tas *heapsort*).

1. Dérouler l'algorithme de tri par tas sur la série $8, 7, 2, 10, 1, 5, 3, 4$, en dessinant l'état de la file de priorité (arbre et stockage tableau) à toutes les étapes intermédiaires.
2. Implémenter une fonction de tri par tas `void *heap_sort(elt_type *tab, int n)`
3. La tester d'abord sur de petits tableaux statiques, puis sur des tableaux générés aléatoirement.
4. Dans quel ordre sont triés vos tableaux ? Comment faire changer facilement cela ?
5. Étudier la complexité temporelle, puis la complexité spatiale de cet algorithme de tri.
6. Ce tri est-il un tri en place ? Ce tri est-il stable ?

1. Qui sera vu en deuxième année.