TP d'algorithmique & structures de données C. Bentz/D. Porumbel

Séance de TP n°1 du 07/11/2024 : Tris et Tas

Exercice 1 : tas min

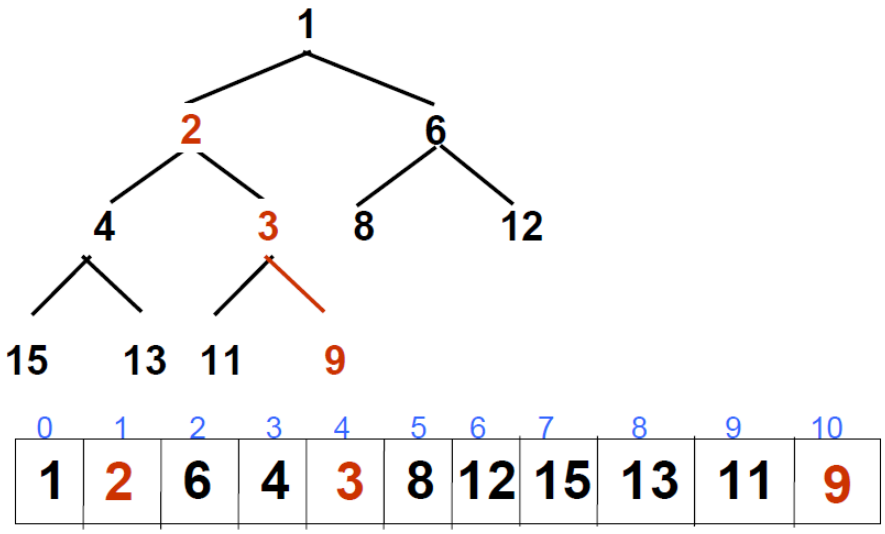
Un tas binaire minimum, ou simplement tas min, est une structure de données aux propriétés hautement utiles pour un certain nombre d'applications. Formellement, un tas min est un arbre binaire dit **parfait** dans lequel la valeur stockée en chaque nœud est inférieure ou égale à celle stockée en chacun de ses fils (s'ils existent). Cette définition implique notamment que la plus petite valeur stockée dans un tas min se trouve à la racine de ce tas.

Pour rappel, un arbre binaire parfait est soit un arbre binaire **complet** (c'est-à-dire un arbre binaire dans lequel les nœuds de tous les niveaux sauf le dernier ont exactement 2 fils, et les nœuds du dernier niveau sont des feuilles), soit un arbre binaire dans lequel tous les niveaux sauf le dernier forment un arbre binaire complet, et dans ce cas les nœuds (feuilles) du dernier niveau sont tous regroupés sur la gauche (on dit aussi "calés à gauche").

Un arbre binaire parfait qui a $h+1$ niveaux (où h est sa hauteur, c'est-à-dire la longueur, en nombre d'arêtes, de la plus longue chaîne reliant la racine à une feuille) possède donc entre $1 + 2^1 + 2^2 + \cdots + 2^{h-1} + 1 = 2^h$ nœuds (cas d'un arbre binaire complet de hauteur $h-1$, plus 1 feuille sur le dernier niveau) et $2^h - 1 + 2^h = 2^{h+1} - 1$ nœuds (cas d'un arbre binaire complet de hauteur h, donc qui possède $h+1$ niveaux). En d'autres termes, dans un arbre binaire parfait à n nœuds, on a toujours $h = \Theta(\log n)$.

Une autre conséquence de cette définition est que, pour stocker un arbre binaire parfait (par exemple, un tas min) à n nœuds, qui est une **structure arborescente**, on peut en réalité utiliser un tableau de taille n (donc sans perte de place), qui est pourtant une **structure linéaire**, en stockant chacun des nœuds dans une case du tableau. En général, on "balaye" les nœuds de l'arbre niveau par niveau (en commençant par la racine, puis en descendant vers les feuilles), de gauche à droite, et on stocke les nœuds de l'arbre dans le tableau associé au fur et à mesure qu'on les rencontre.

À titre d'exemple, la figure qui suit montre un tas min d'entiers de hauteur $h = 3$ contenant 11 valeurs (la plus petite étant 1, qui est donc stockée à la racine), ainsi que le tableau d'entiers de taille 11 associé (les 4 feuilles du dernier niveau, calées à gauche, y sont stockées dans les 4 dernières cases) :



Avec cette représentation, on peut remarquer que, si la valeur d'un nœud est stockée en case d'indice i (pour i entre 0 et $n-1$), alors la valeur de son père est stockée en case d'indice $\lfloor \frac{i-1}{2} \rfloor$, et celles de ses fils gauche et droite (s'ils existent) sont stockées en cases d'indice $2i+1$ et $2i+2$, respectivement.

On veut implémenter en C les opérations suivantes dans des tas min qu'on représentera par des tableaux d'entiers : l'accès à la plus petite valeur stockée dans le tas, l'insertion d'une valeur dans le tas, et la suppression de la plus petite valeur du tas. En cours de route, on montrera que chaque opération peut être implémentée en $O(h)$, et donc (d'après la page précédente) en $O(\log n)$, où n est le nombre de valeurs stockées dans le tas.

Travail demandé :

1. Définir en C un type structuré `tas` : un tel tas sera représenté par un tableau d'entiers, mais on aura également besoin de la taille de ce tableau (nombre maximum de valeurs que peut contenir le tas), et du nombre de valeurs qui y sont effectivement stockées à tout instant.

   Ce type sera défini dans un fichier d'en-tête à part (qu'on appellera `tas.h`, par exemple), tout comme les prototypes des procédures et fonctions qui seront implémentées dans cet exercice.

2. Implémenter en C les fonctions de base définies sur les tas : une fonction pour créer un nouveau tas (elle prend en paramètre le nombre maximum de valeurs que peut contenir ce tas), une fonction qui renvoie la plus petite valeur stockée dans le tas, une fonction qui renvoie `true` si le tas est plein (plus de place disponible pour y stocker de nouvelles valeurs) et `false` sinon, et enfin une fonction qui renvoie `true` si le tas est vide (aucune valeur stockée dedans) et `false` sinon.

On pourra également, de façon temporaire, implémenter une procédure permettant d'afficher l'ensemble des valeurs stockées dans le tas (en particulier pour faciliter le débogage du code, le cas échéant).

3. Implémenter en C une procédure permettant d'insérer une valeur donnée dans un tas min donné. Cette procédure prendra en paramètres la valeur à insérer ainsi qu'une référence vers le tas dans lequel insérer cette valeur. Le principe de l'insertion d'une valeur `val` dans un tas min est le suivant : si on note `ins` le plus petit indice (du tableau associé au tas) qui ne contient pas de nœud (dans la suite, on désignera cet emplacement vide particulier comme "la première feuille disponible"), alors on commence par insérer la valeur `val` à cet emplacement.

   Le tas contient alors une valeur supplémentaire, et cette valeur `val` devient donc "la dernière feuille" de l'arbre. De cette façon-là, l'arbre binaire obtenu est toujours un arbre binaire parfait. En revanche, ce n'est plus nécessairement un tas min. La suite de cette procédure consiste alors, par des échanges de valeurs successifs (qui maintiennent la structure d'arbre binaire parfait), à transformer cet arbre en tas.

   Plus précisement, si la valeur `val` est plus grande que celle de son père, alors l'arbre est déjà un tas. Sinon, on va échanger la valeur `val` avec celle de son père : on dit que `val` "remonte", et que la valeur de son père "redescend". On recommence cette opération (consistant à remonter la valeur `val` et à redescendre celle de son père) jusqu'à ce que la valeur `val` ne soit pas strictement plus petite que celle de son père (s'il existe), ou bien que la valeur `val` soit remontée à la racine (auquel cas `val` est en réalité la plus petite valeur du tas ainsi obtenu).

4. Implémenter en C une procédure permettant de supprimer la plus petite valeur d'un tas min donné. Cette procédure prendra en paramètre une référence vers le tas dans lequel on souhaite effectuer cette suppression. Le principe de la suppression de la plus petite valeur d'un tas min est similaire à celui de l'insertion, excepté que dans ce cas-là on parcourt le tas dans l'autre sens (de la racine vers les feuilles).

   Plus précisément, dans le cas de la suppression de la valeur minimum du tas, on commence par stocker à la racine de l'arbre la valeur `val_fin` qui était auparavant stockée à la fin du tableau, c'est-à-dire dans la dernière feuille. De cette façon-là, le tas contient une valeur de moins (celle qui était à la racine, et qu'on a maintenant écrasée et remplacée par `val_fin`), et l'arbre binaire obtenu est toujours un arbre binaire parfait. En revanche, ce n'est plus nécessairement un tas min. Comme dans le cas de l'insertion, la suite de cette procédure consiste alors, par des échanges de valeurs successifs (qui maintiennent la structure d'arbre binaire parfait), à transformer cet arbre en tas.

Plus précisement, si la valeur `val_fin` est plus petite que celles de ses deux fils (s'ils existent), alors l'arbre est déjà un tas. Sinon, on va échanger la valeur `val_fin` avec **celle du plus petit de ses 2 fils**. On recommence cette opération (consistant à redescendre la valeur `val_fin` et à remonter celle du plus petit de ses 2 fils) jusqu'à ce que la valeur `val_fin` ne soit pas strictement plus grande que celle d'un de ses 2 fils (s'il(s) existe(ent)), ou bien que la valeur `val_fin` soit redescendue dans une feuille de l'arbre.

5. On justifiera en détails dans les commentaires du code le fait que les opérations d'insertion et de suppression de la valeur minimum implémentées dans les 2 questions précédentes sont correctes (c'est-à-dire que l'arbre obtenu à la fin est bien un tas) et s'exécutent en temps $O(h)$, et donc en temps $O(\log n)$, dans un tas de hauteur h à n éléments.

Indications :

- Pour échanger la valeur stockée dans le nœud qui se trouve à l'indice i avec celle de son père (resp. d'un de ses fils), on pourra utiliser les indices $\lfloor \frac{i-1}{2} \rfloor$ (resp. $2i+1$ ou $2i+2$, selon le cas).

- Pour rappel, pour accéder en C au champ `ch` d'une variable d'un type structuré, référencée par un pointeur `p`, on peut utiliser la notation `p->ch` à la place de `(*p).ch` ; les 2 notations sont donc équivalentes.

- Si on souhaite inclure un fichier d'en-tête `toto.h` dans un autre fichier (soit un fichier source, soit un autre fichier d'en-tête), on peut utiliser les mots-clés `#ifndef` et `#endif` (destinés au pré-processeur) ainsi, pour éviter que les multiples inclusions de fichiers d'en-tête en cascade ne mènent au cas où un même fichier d'en-tête est inclus plusieurs fois :

```
#ifndef TOTO_H
#define TOTO_H
#include <toto.h>
#endif
```

  En utilisant ces quelques lignes, on définira une constante `TOTO_H` si elle n'est pas encore définie, et on inclura alors également le fichier d'en-tête `toto.h` juste après. Toutes les fois suivantes où ces lignes seront rencontrées dans d'autres fichiers, la constante `TOTO_H` sera déjà définie, et les 2 lignes suivantes (y compris celle consistant à inclure le fichier d'en-tête `toto.h`) seront alors ignorées, comme souhaité.

- Durant toute cette séance, on utilisera systématiquement un fichier *Makefile* pour compiler correctement tous les fichiers qui doivent l'être. En outre, on inclura tous les fichiers d'en-tête standard nécessaires : `stdio.h` (`printf, scanf, etc.`), `stdlib.h` (`malloc, free, etc.`), `stdbool.h` (`true, false, etc.`) et `stddef.h` (`sizeof, NULL, etc.`).

Exercice 2 : tri par tas ("heap sort")

À l'aide de la structure de tas min définie dans l'exercice précédent, et des opérations associées, on peut obtenir un algorithme de tri très efficace appelé **tri par tas**. Cet algorithme a une complexité en temps au pire cas de $O(n \log n)$ pour trier n entiers, ce qui en fait l'algorithme de tri par comparaisons le plus efficace (au même titre que l'algorithme de **tri fusion**).

Informellement, le principe de cet algorithme est le suivant :

- Créer un tas min pouvant contenir un nombre maximum de valeurs égal à la taille du tableau d'entiers à trier, puis insérer dans ce tas les valeurs contenues dans ce tableau à trier, une par une ;
- Tant que ce tas n'est pas vide, récupérer la plus petite valeur qu'il contient, la stocker dans le tableau d'entiers triés par ordre croissant en construction, puis supprimer cette valeur du tas.

Travail demandé :

1. Implémenter en C cet algorithme de tri, sous la forme d'une procédure qui modifie directement le tableau d'entiers passé en paramètre.
2. Écrire un `main` pour tester cet algorithme. Au moment de l'appel du programme, l'utilisateur fournira en paramètres les entiers du tableau à trier, séparés par des espaces. Ces entiers seront stockés dans un tableau d'entiers pour lequel on allouera de la mémoire via la fonction `malloc`. Pour ce faire, on pourra utiliser la fonction C prédéfinie `atoi`, qui se trouve dans le fichier d'en-tête `stdlib.h`, et qui renvoie un entier dont la valeur est celle de la chaîne de caractères passée en paramètre (par exemple, on renverra l'entier 2 si le paramètre vaut "2").
3. On justifiera brièvement dans les commentaires du code le fait que cet algorithme permet de trier n entiers en temps $O(n \log n)$ au pire cas.

Exercice 3 : tri rapide ("quick sort")

On souhaite ici implémenter un autre algorithme de tri, dit **tri rapide**. Le principe de cet algorithme récursif est le suivant : on choisit (il y a différentes façons de le faire) une valeur dans le tableau d'entiers qu'on souhaite trier, qui devient alors le **pivot**. On partitionne ensuite les autres valeurs du tableau à trier, de façon à ce que toutes celles qui sont inférieures au pivot soient à gauche de ce dernier (mais pas forcément triées entre elles), et que toutes celles qui sont supérieures au pivot soient à droite de ce dernier (mais pas forcément triées entre elles). De cette façon, on est sûr que le pivot est à la bonne place par rapport à toutes les autres valeurs du tableau à trier.

Puis, on fait un appel récursif à l'algorithme de tri rapide sur la partie du tableau à trier qui se trouve à gauche du pivot, et un autre appel récursif sur la partie du tableau à trier qui se trouve à droite du pivot.

On arrête la récursion si ce qu'il reste du tableau à trier ne contient qu'une seule valeur (partie du tableau ne contenant qu'une seule case).

Pour effectuer la partition du tableau une fois le pivot choisi, on peut informellement procéder de la façon suivante :

- Stocker le pivot tout à droite du tableau ;
- Définir un indice `ppg` (pour *premier plus grand*), égal initialement à l'indice tout à gauche, qui doit vérifier les 2 invariants de boucle suivants, à chaque itération `i` (pour `i` allant de l'indice tout à gauche à l'indice juste avant celui du pivot) : (*i*) les valeurs plus petites que le pivot sont à gauche de l'indice `ppg-1` (en particulier, si `ppg` $>$ `i`, toute valeur à gauche de l'indice `i` inclus est plus petite que le pivot) et (*ii*) les valeurs plus grandes que le pivot sont entre les indices `ppg` et `i` ;
- À la fin de la boucle sur `i`, échanger la valeur qui se trouve à l'indice `ppg` (qui est donc une valeur plus grande que le pivot, par hypothèse) avec celle du pivot. Il pourra également être utile de renvoyer à l'algorithme récursif l'indice `ppg`, où se trouve à présent le pivot.

Afin de garantir les invariants de boucle (*i*) et (*ii*), il suffit, à chaque itération `i`, de différencier 2 cas : soit la valeur qui est à l'indice `i` est plus grande que le pivot (auquel cas il n'y a rien à faire), soit cette valeur est plus petite que le pivot (auquel cas il faut échanger les valeurs qui sont aux indices `i` et `ppg`, puis mettre à jour `ppg` en l'incrémentant).

Travail demandé :

1. Justifier en détails dans les commentaires les raisons pour lesquelles les 2 invariants de boucle (*i*) et (*ii*) sont ainsi vérifiés.
2. Implémenter en C l'algorithme de tri rapide, sous la forme d'une procédure récursive qui modifie directement le tableau d'entiers passé en paramètre. On laissera la possibilité de choisir le pivot de différentes façons, en fonction de la valeur d'un paramètre entier, le pivot pouvant être : (*a*) la valeur tout à droite, (*b*) la valeur tout à gauche, (*c*) la valeur médiane entre celle tout à gauche, celle tout à droite, et celle au milieu, et (*d*) une valeur choisie aléatoirement dans le tableau.
3. Mettre à jour le `main` de l'exercice précédent : on ajoutera la possibilité pour l'utilisateur de spécifier l'algorithme de tri à utiliser (tri par tas ou tri rapide, pour l'instant), au moment de l'appel du programme.

4. On peut montrer (si l'on suppose que tous les tableaux d'entiers de même taille sont des données équiprobables pour l'algorithme) que cet algorithme trie n entiers en temps $O(n \log n)$ *en moyenne*. On justifiera en revanche en détail dans les commentaires du code le fait que, dans le pire cas, cet algorithme trie n entiers en temps $O(n^2)$.

Indication :

- Pour générer un nombre entier pseudo-aléatoire compris entre 0 et $M-1$ pour un certain M, on peut utiliser la fonction prédéfinie `rand()` (qui renvoie une valeur entière pseudo-aléatoire comprise entre 0 et une certaine constante `RAND_MAX`, et qui se trouve dans le fichier d'en-tête `stdlib.h`), puis appliquer un modulo M au résultat renvoyé. Pour initialiser la séquence de nombres pseudo-aléatoires, on préconise en général d'utiliser une valeur plus ou moins aléatoire, comme par exemple l'heure exacte (en secondes), passée en paramètre à la procédure `srand()` (qui se trouve également dans le fichier d'en-tête `stdlib.h`), de la façon suivante : `srand(time(NULL))` (où `time` est une fonction prédéfinie qui se trouve dans le fichier d'en-tête `time.h`).

Exercice 4 : tri fusion ("merge sort")

On souhaite ici implémenter un autre algorithme de tri récursif, différent du tri rapide : le **tri fusion**. Le principe de cet algorithme récursif est le suivant : on coupe le tableau à trier en 2 parties égales (à 1 case près, le cas échéant), puis on fait un appel récursif à l'algorithme de tri fusion sur la moitié du tableau à trier qui se trouve à gauche du milieu, et un autre appel récursif sur la moitié du tableau à trier qui se trouve à droite du milieu.

Une fois que ces 2 appels récursifs se terminent, on *fusionne* les 2 moitiés de tableau ainsi triées en un unique tableau trié, à l'aide d'une procédure qui est en réalité le cœur de l'algorithme. Comme dans le cas du tri rapide, on arrête la récursion si ce qu'il reste du tableau à trier ne contient qu'une seule valeur (partie du tableau ne contenant qu'une seule case).

Informellement, le principe de la procédure qui permet de fusionner 2 moitiés de tableau triées en un unique tableau trié est le suivant :

- Stocker ces 2 moitiés dans un tableau temporaire, dans le même ordre d'apparition des valeurs (moitié gauche puis moitié droite) ;

- Définir 3 indices `i`, `j` et `k`, dont les rôles respectifs seront les suivants : `i` (resp. `j`) sera initialisé à l'indice tout à gauche de la moitié gauche (resp. droite), et sera incrémenté au fur et à mesure du déroulement de la procédure, et `k` sera initialisé à l'indice tout à gauche du tableau unique résultant de la fusion des 2 moitiés, qu'il parcourra ensuite ;

- Tant qu'il reste des valeurs à parcourir dans la moitié gauche **et** dans la moitié droite, comparer les valeurs qui sont à l'indice `i` de la moitié gauche et à l'indice `j` de la moitié droite, et stocker la plus petite des 2 à l'indice `k` du tableau unique résultant de la fusion des 2 moitiés, puis incrémenter `k` et, selon le cas, soit `i`, soit `j` ;

- Stocker à la fin du tableau unique résultant de la fusion des 2 moitiés les valeurs restantes dans la seule moitié (gauche ou droite) au sein de laquelle il reste des valeurs à parcourir à l'issue de la boucle précédente.

Travail demandé :

1. Implémenter en C la procédure qui fusionne en un tableau unique 2 moitiés de tableau triées. On justifiera en détail dans les commentaires du code le fait que cette fusion est correcte à l'issue de l'exécution de cette procédure, ainsi que le fait que cette procédure s'exécute en temps $O(T)$, où T est la taille du tableau unique résultant de cette fusion.

2. Implémenter en C l'algorithme récursif de tri fusion, à l'aide de la procédure implémentée à la question précédente, et mettre à jour le `main` de l'exercice précédent, en ajoutant la possibilité pour l'utilisateur de choisir d'utiliser le tri fusion au moment de l'appel du programme.

3. Justifier en détails dans les commentaires du code les raisons pour lesquelles cet algorithme permet de trier n entiers en temps $O(n \log n)$ au pire cas. On pourra pour cela remarquer, en expliquant pourquoi, que tout appel récursif effectué sur un tableau de taille $\frac{n}{2^p}$ s'exécute en temps $O(\frac{n}{2^p})$, et qu'il y a $O(2^p)$ tels appels, pour $p \in \{0, \ldots, \lceil \log_2 n \rceil\}$.