TP14: Tas et retour du C

MP2I Lycée Pierre de Fermat

Tas binaires

On se propose d'implémenter les tas en utilisant des tableaux, et de les utiliser pour écrire une fonction de tri en place efficace, en $\mathcal{O}(n \log n)$.

On écrira trois fichiers séparés: "tas.h" et "tas.c" serviront à implémenter la structure de tas ainsi que le tri par tas, et "test_tas.c" contiendra les fonctions de test et le main.

Vous trouverez sur Cahier de Prépa une archive contenant ces trois fichiers, partiellement remplis. Le fichier "tas.h" contient l'interface des tas: création, ajout, extraction et tri.

```c
/*** Tas min ***/
typedef struct {
  int* tab; // éléments stockés
  int taille_max; // taille maximale possible du tas
  int taille_actuelle; // nombre actuel d'éléments stockés
} HEAP;

/* Crée et renvoie un tas vide de taille maximale `taille`. */
HEAP* tas_vide(int taille);

/* Ajoute x dans t. t ne doit pas être plein. */
void ajouter(HEAP* t, int x);

/* Renvoie la valeur minimale de t. t doit être non vide. */
int extraire_min(HEAP* t);

/* Trie en place le tableau t de taille n en utilisant un tas. */
void tri_par_tas(int* t, int n);
```

Le fichier "tas.c" contiendra les implémentations de ces fonctions, ainsi que des fonctions auxiliaires utilitaires.

On rappelle que dans l'implémentation par tableau, un tas est représenté par un tableau donnant ses éléments dans l'ordre du parcours en largeur. Alors, le noeud numéro i a pour enfants $2i+1$ et $2i+2$, et comme parent $\lfloor \frac{i-1}{2} \rfloor$. De plus, l'attribut `taille_actuelle` indique la case où se situera la prochaine feuille.

Quelques fonctions pour les tests

Les fonctions de cette partie sont à implémenter dans "test_tas.c", exceptée la dernière.

Question 1. Implémentez une fonction `bool est_tas(int* t, int n)` qui détermine si les n premières cases de t représentent bien un tas.

Question 2. Implémentez une fonction `int* tableau_aleatoire(int n, int a, int b)` renvoyant un tableau de taille n contenant des entiers aléatoires entre a et b inclus.

Question 3. Implémenter une fonction `print_tab(int* t, int n)` affichant les n premières cases d'un tableau, puis une fonction `print_tas(HEAP* h)` affichant un tas.

Question 4. Implémentez une fonction `bool est_trie(int* t, int n)` déterminant si le tableau t de taille n est trié **dans l'ordre décroissant**.

Question 5. Dans "tas.c", écrivez les en-têtes des différentes fonctions du header, et définissez les toutes avec comme seules instructions:

```
1 void tasser(HEAP* t, int i){
2   printf("Pas encore implémentée\n");
3   assert(false);
4 }
```

Cela vous permettra de compiler et d'exécuter votre programme pour tester les fonctions au fur et à mesure que vous les implémentez.

Tas vide et ajout

Les fonctions de cette partie sont à implémenter dans `tas.h` et `tas.c`.

Question 6. Implémentez la fonction `tas_vide`.

On rappelle l'algorithme d'ajout dans un tas:

Algorithme 1 : Ajout dans un tas

Entrée(s) : T un tas, x un élément à insérer
Sortie(s) : x a été inséré dans T, et T reste un tas

$p \leftarrow$ premier noeud de T n'ayant pas 2 enfants;
si p ***n'a pas d'enfant gauche*** **alors**
    Ajouter à T une nouvelle feuille f, enfant gauche de p;
sinon
    `// p n'a pas d'enfant droit`
    Ajouter à T une nouvelle feuille f, enfant droit de p;
tant que f ***n'est pas la racine et*** $f.e <$ ***parent***$(f).e$ **faire**
    Échanger $f.e$ et **parent**$(f).e$;
    $f \leftarrow$ **parent**(f);

La boucle while à la fin de l'algorithme est une procédure appelée **tasser**, et qui permet de placer au bon endroit une étiquette étant **trop loin de la racine** en la rapprochant.

Question 7. Écrire une fonction auxiliaire `tasser` de spécification suivante:

```
1 /* Tasse t en faisant remonter le noeud d'indice i vers la racine pour
2    obtenir un tas
3 Préconditions: - l'arbre dont i est la racine est un tas
4                - l'étiquette du parent de i est inférieure à toutes les
5                  étiquettes de l'arbre enraciné en i, sauf éventuellement
6                  celle de i lui-même
7 Postcondition: t est un tas */
8 void tasser(HEAP* t, int i);
```

Question 8. Implémenter la fonction `ajouter`, et la tester.

Extraction

On rappelle l'algorithme utilisé pour extraire le min d'un tas:

Algorithme 2 : Extraction de la racine

Entrée(s) : T un tas
Sortie(s) : e étiquette de la racine de T. La racine a été supprimée, et T reste un tas

1 $r \leftarrow$ la racine de T ;
2 $e \leftarrow r.e$;
3 $n \leftarrow$ dernière feuille de T ;
4 $r.e \leftarrow n.e$;
5 Supprimer n de T;
6 **tant que** ***l'étiquette de*** r ***est plus grande que celle de l'un de ses enfants*** **faire**
7 $\quad f \leftarrow$ l'enfant de r avec la plus petite étiquette; Échanger $r.e$ et $f.e$;
8 $\quad r \leftarrow f$;
9 **retourner** e

Remarque 1. En pratique, on échangera les cases contenant les étiquettes de r et n pour implémenter les lignes 3 et 4.

A nouveau, la boucle while à la fin de l'algorithme est une procédure classique des tas, appelée **tamiser**. Elle permet de placer au bon endroit une étiquette étant **trop proche de la racine** en l'éloignant et en la faisant descendre vers les feuilles.

Question 9. Écrire une fonction auxiliaire `tamiser` de spécification suivante:

```
1 /* Tamise t en faisant descendre le noeud d'indice i vers les feuilles
2    pour obtenir un tas
3 Précondition: Les deux sous-arbres de i sont des tas
4 Postcondition: t est un tas */
5 void tamiser(HEAP* t, int i);
```

Question 10. Implémenter la fonction `extraire_min` et la tester.

Tri par tas

Le principe du tri par tas est très similaire à celui du tri par sélection. Il s'effectue en deux étapes: transformer le tableau en tas, puis extraire les éléments un à un par ordre croissant. Ce tri aura pour effet de trier le tableau dans l'ordre **décroissant** car il va positionner les éléments du plus petit vers le plus grand en partant de la fin du tableau.

Du tableau au tas Pour trier un tableau A, on utilise ce tableau pour stocker un tas, initialement vide. Puis, on ajoute les éléments de A un par un. Remarquons que pour cela, il nous suffit d'augmenter manuellement la taille du tas, et d'appeler la procédure **tasser**. En effet, si T est un tas utilisant A comme tableau de stockage, et que le tas contient i éléments, alors $A[0..i-1]$ contient les éléments du tas, et alors **tasser**(T, i) va avoir comme effet d'insérer $A[i]$ dans le tas

Par exemple, pour $A = [5, 2, 7, 3, 4, 1]$, si l'on utilise A pour stocker un tas initialement vide, regardons l'état de A au fur et à mesure que l'on tasse le tas:

	État de A	Taille du tas
Initialement	$[5, 2, 7, 3, 4, 1]$	0
Après **tasser**$(A, 0)$	$[5, 2, 7, 3, 4, 1]$	1
Après **tasser**$(A, 1)$	$[2, 5, 7, 3, 4, 1]$	2
Après **tasser**$(A, 2)$	$[2, 5, 7, 3, 4, 1]$	3
Après **tasser**$(A, 3)$	$[2, 3, 7, 5, 4, 1]$	4
Après **tasser**$(A, 4)$	$[2, 3, 7, 5, 4, 1]$	5
Après **tasser**$(A, 5)$	$[1, 3, 2, 5, 4, 7]$	6

A chaque étape, les "Taille du tas" premières cases de A forment un tas.

Ainsi, plutôt que de faire appel à `tas_vide`, la fonction de tri commencera comme suit:

Question 11. Écrire une fonction auxiliaire `HEAP* tasifier(int* t, int n)` qui applique l'étape décrite dans le paragraphe précédent et renvoie un tas stocké dans t contenant tous les éléments initialement présents dans t. Cette fonction sera de la forme suivante:

```
1 HEAP* tasifier(int* t, int n){
2   HEAP* h = malloc(sizeof(HEAP));
3   h->tab = t;
4   h->taille_actuelle = 0;
5   h->taille_max = n;
6   ...
7   return h;
8 }
```

Du tas au tableau La deuxième phase consiste à vider le tas. Remarquons qu'avec la remarque faite dans la partie extraction, la fonction `extraire_min` positionne l'ancienne racine à la place qu'occupait la dernière feuille. Ainsi, si l'on appelle `extraire_min` sur un tas T stocké dans un tableau A, si le tas contient i éléments, alors **extraire_min**(T) met l'élément min de $A[0..i-1]$ à la case $i-1$, puis l'appel suivant à **extraire_min**(T) met l'élément min de $A[0..i-2]$ à la case $i-2$, et ainsi de suite. Par exemple, sur le tas obtenu dans le paragraphe précédent:

	État de A	Taille du tas
Initialement	$[1, 3, 2, 5, 4, 7]$	6
Après **extraire_min**(A)	$[2, 3, 7, 5, 4, 1]$	5
Après **extraire_min**(A)	$[3, 4, 7, 5, 2, 1]$	4
Après **extraire_min**(A)	$[4, 5, 7, 3, 2, 1]$	3
Après **extraire_min**(A)	$[5, 7, 4, 3, 2, 1]$	2
Après **extraire_min**(A)	$[7, 5, 4, 3, 2, 1]$	1

A chaque étape, les "Taille du tas" premières cases forment un tas, et le reste du tableau A contient les plus petits éléments, triés.

Question 12. Implémentez la fonction `tri_par_tas`, et testez-la bien.

Structures arborescentes en C

Dans cette partie, on manipule des arbres en C, gérés avec des pointeurs. Vous trouverez dans l'archive du TP une sous-archive appelée "data.zip". Cette archive contient envrion 400 fichiers textes, qui stockent les informations des membres de l'ordre des chevaliers Jedi. Chaque fichier contient les informations suivantes sur le ou la membre:

- Prénom
- Nom
- Nombre d'apprentis Jedi
- Prénoms et noms des apprentis Jedi

Le nom de chaque fichier est "[prenom]_[nom].txt". On suppose que personne ne porte à la fois le même prénom et le même nom.

On se propose d'utiliser la structure suivante pour stocker les informations d'une ou d'un Jedi:

```
1 struct jedi {
2   char* prenom;
3   char* nom;
4   struct jedi** apprentis; // tableau des apprentis
5   int n; // nombre d'apprentis
6 }
7
8 typedef struct jedi JEDI;
```

Cette structure ressemblera donc à un arbre. Pour les manipuler, il est possible de procéder récursivement comme en OCaml.

On définit la relation "être dans la lignée de" comme la clôture transitive réflexive de la relation "être apprenti de". Ainsi, si A a entraînée B et B a entraînée C, alors C est dans la lignée de A.

Dans l'archive, le fichier "jedi.c" contient une fonction

```
1 bool est_lignee(JEDI* j, char* prenom, char* nom)
```

qui détermine si la personne dont le nom et le prénom sont donnés fait partie de la lignée du jedi j

Question 13. Écrire une fonction `JEDI* load_jedi(char* prenom, char* nom)` qui charge les informations d'un ou d'une Jedi. Cette fonction sera a priori récursive afin de pouvoir également charger les informations des apprentis. Vous pouvez tester votre fonction sur la personne appelée "bohuslava guinvarc", qui doit avoir 2 apprentis, et 1 sous-apprenti.

Question 14. Écrivez une fonction `void print_jedi(JEDI* j)` qui affiche les informations d'un / d'une Jedi ainsi que de ses apprentis, en indentant de la façon suivante:

```
feria lokalis:
        brutus dubblor:
                shal kovandis:
                        [Pas d'apprentis]
                jerome boulanger:
                        [Pas d'apprentis]
        jarvor bistopol:
                [Pas d'apprentis]
```

(Indication: vous écrirez une fonction auxiliaire prenant comme argument le niveau d'indentation actuel.)

Bien que les arbres se prêtent bien aux fonctions récursives, on essaie autant que possible de ne pas les utiliser en C, et de leur préférer les boucles. On rappelle le principe du parcours en profondeur d'un arbre en utilisant une pile:

Algorithme 3 : Parcours en profondeur

Entrée(s) : A un arbre

```
1 P ← pile vide;
2 P.push(A.racine)// empiler
3 tant que P n'est pas vide faire
4     u ← P.pop()// dépiler
      // Traiter u
5     pour v enfant de u faire
6         P.push(v);
```

Question 15. En utilisant un tableau ou une liste chaînée, implémentez une structure de pile permettant de stocker des éléments de type `JEDI*`.

Question 16. En utilisant cette structure, écrire une deuxième version de la fonction de recherche `est_lignee`, non récursive.

Question 17. Implémentez une fonction `char** lignee(JEDI* j, char* prenom, char* nom, int* n)` qui renvoie la liste des Jedi ayant précédé la personne "prenom nom" dans l'arbre des descendants de j, et stocke dans `*n` le nombre de personnes trouvées (i.e. la longueur de la liste renvoyée).

Pour implémenter un parcours en largeur, on utilise le même schéma que pour le parcours en profondeur, mais au lieu d'une pile, on utilise une file.

Question 18. Implémentez une structure de file, et utilisez la pour afficher la liste de tous les Jedi descendant de "laerke santos", dans l'ordre en largeur.

Question 19. Le fichier "liste_fichiers.txt" contient la liste des fichiers de "data/" dans un ordre quelconque. En utilisant ce fichier, construisez l'arbre total de l'ordre des chevaliers, et déterminez qui a fondé l'ordre (i.e. qui n'est l'apprenti de personne).

Avant de vous mettre à programmer, réfléchissez aux différentes structures que vous aurez besoin de mettre en place, et à la stratégie que vous utiliserez.