TP 13 : tri par Tas

Notions : Tris, Complexité, Arbres

1 Introduction

Le tri est l'opération consistant à ordonner un ensemble d'éléments en fonctions de clés sur lesquelles est définie une relation d'ordre. Les algorithmes de tri sont fondamentaux dans de nombreux domaines, en particulier la gestion où les données sont presque toujours triées selon un critère avant d'être traitées. La notion de complexité permet de donner un sens numérique à l'idée intuitive de coût d'un algorithme, en temps de calcul et en place mémoire. Il est cependant important de différencier la complexité théorique, qui donne des ordres de grandeurs de ces coûts sans être liée à une machine, et la complexité pratique, qui dépend d'une machine et de la manière dont l'algorithme est programmé. La littérature évoque généralement la complexité théorique i.e. le nombre de comparaisons effectuées sur les clés et le nombre de permutations effectuées sur les données dans le cas d'un tri. Nous illustrerons aussi la complexité pratique par le temps d'exécution des fonctions. Le tri interne est un tri opérant sur des données présentes en mémoire, tandis que le tri externe travaille sur des données appartenant à des fichiers. Les tris internes seuls seront exposés, et trois catégories se distinguent en fonction de leur complexité théorique (n est le nombre d'éléments à trier) :

— Les tris triviaux sont en $O(n^2)$: tri bulle, tri par sélection, tri par insertion, ...

— Les tris en $O(n^x)$ avec $1 < x < 2$: tri shell

— Les tris en $O(nlog(n))$: Tri par tas (Heap Sort), tri rapide (Quick Sort), tri MergeSort.

2 Principe du tri par tas

Ce tri est assez simple à programmer et est de complexité en o(n log n), le situant parmi les tris performants. Cette méthode de tri utilise une structure de données sous-jacente qui est le tas (heap en anglais). Un tas est un arbre binaire (un nœud de l'arbre a 2 fils : 85 a 73 et 36 pour fils) parfait (tous les niveaux de l'arbre sont complètement remplis, sauf peut-être le dernier, où tous les nœuds sont le plus à gauche possible) qui est de plus partiellement ordonné : en tout nœud de l'arbre, la valeur du nœud père est supérieure à la valeur de chacun de ses nœuds fils. Par contre, les deux fils ne sont pas ordonnés entre eux. La figure 1 donne un exemple de tas.

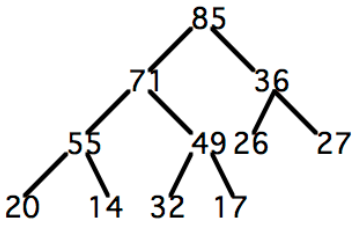

FIGURE 1 – Un exemple de tas

Trois propriétés sont utilisées :

1. le tas est représenté par un tableau. Les deux fils sont obtenus grâce à la relation suivante : si on examine le nœud i, les deux fils de ce nœud i sont donnés par les éléments d'indice 2*(i+1) et 2*(i+1)-1.
2. Réciproquement, un noeud d'indice i dans le tableau a pour père le noeud d'indice (i-1)/2
3. Dans un tas non vide, la valeur maximale est toujours située à la racine.

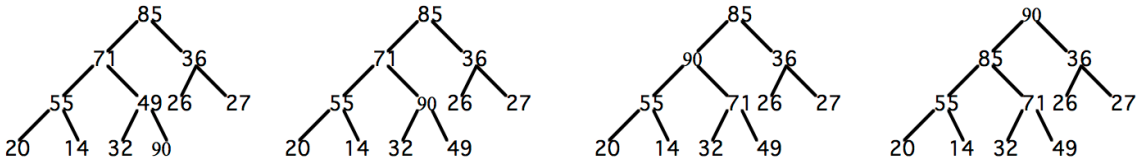

FIGURE 2 – Insertion d'un élément dans un tas

2.1 Ajout d'un élément à un tas de n éléments

Pour ajouter un élément à un tas de n éléments (donc le dernier élément du tableau est d'indice n-1), on place ce nouvel élément à l'indice n. Ensuite, on doit vérifier la propriété du tas, c'est à dire que le père, d'indice (n-1)/2 dans le tableau doit être supérieur à l'élément ajouté. Si c'est le cas, c'est un tas et on s'arrête. Sinon, on échange les éléments d'indices n et (n-1)/2. Et on itère entre le père de (n-1)/2 qui est d'indice ((n-1)/2-1)/2 et l'élément échangé. On arrête bien sûr le processus itératif à la racine. Exemple 1 : ajout de 90 au tas

L'exemple figure 2 et table ?? illustre les étapes de l'ajout du nœud de valeur 90 dans le tas initial (tableau) de la figure 2. Le tableau devient successivement :

indice	0	1	2	3	4	5	6	7	8	9	10	Commentaire
	85	71	36	55	49	26	27	20	14	32		Tableau initial
Etape 1	85	71	36	55	49	26	27	20	14	32	90	échange de 90 et de son pere (49)
Etape 2	85	71	36	55	90	26	27	20	14	32	49	échange de 90 et de son pere (71)
Etape 3	85	90	36	55	71	26	27	20	14	32	49	échange de 90 et de son pere (85)
Etape 4	90	85	36	55	71	26	27	20	14	32	49	tas final

TABLE 1 – Insertion d'un élément dans un tas

2.2 Suppression de la racine d'un tas de n éléments

L'exemple figure 3 et table ?? illustre les étapes de la suppression du nœud de valeur 85 dans le tas initial figure 1.

Quand on supprime la première valeur (85), on l'échange avec la dernière valeur du tas (17). On doit maintenant se débrouiller pour que les n-1 premiers éléments du tableau soient un tas. Il faut, à partir de la nouvelle racine (17) rétablir les propriétés du tas : la valeur qui vient d'être changée doit respecter la relation fondamentale : le père est supérieur à ses deux fils. Après avoir mis 17 à la racine, il faut donc, comme cette relation n'est pas vérifiée, échanger la valeur 17 d'indice 0 et le plus grand de ses fils d'indices 1 et 2 (ici le nœud 71 d'indice 1). On itère ensuite cette opération entre le nœud venant d'être modifié (le nœud d'indice 1 qui vaut maintenant 17 et ses fils d'indice 3 et 4). Il faut alors échanger 17 et 55. On procède ainsi jusqu'au dernier niveau (un nœud d'indice supérieur ou égal à n/2 : le nœud 20 d'indice 7 ici).

Notez que la valeur 85 est maintenant à l'indice n-1 dans le tableau, qui forme un tas entre les indices 0 et n-2. Elle n'apparaît pas dans l'arbre représentant le tas.

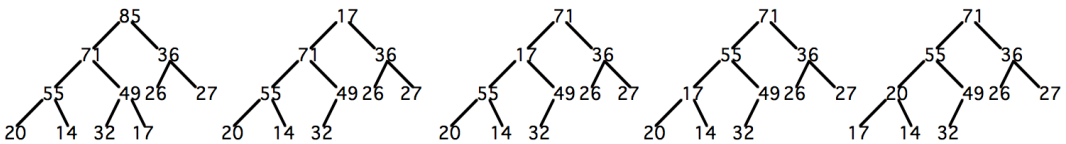

FIGURE 3 – Suppression d'un élément dans un tas

indice	0	1	2	3	4	5	6	7	8	9	10	Commentaire
	85	71	36	55	49	26	27	20	14	32	17	Tableau initial
Etape 1	17	71	36	55	49	26	27	20	14	32	85	Echange de 17 et du plus grand de ses fils
Etape 2	71	17	36	55	49	26	27	20	14	32	85	Echange de 17 et du plus grand de ses fils
Etape 3	71	55	36	17	49	26	27	20	14	32	85	Echange de 17 et du plus grand de ses fils
Etape 4	71	55	36	20	49	26	27	17	14	32	85	Echange de 17 et du plus grand de ses fils

TABLE 2 – Suppresion d'un élément dans un tas

3 Tri

Pour effectuer un tri, il faut 2 étapes : la construction du tas, puis le tri.

3.1 Construction du tas à partir d'un tableau de n éléments

On construit un tas à partir du tableau en commençant par un tas ne comportant qu'un seul nombre : le premier nombre du tableau. Le tas initial est donc le tableau limité à son premier élément. On ajoute alors le deuxième nombre du tableau avec la fonction d'ajout dans un tas et le tableau forme alors un tas entre les éléments d'indice 0 et 1. Puis on ajoute le troisième pour former un tas entre les indices 0 et 2, etc...

3.2 Tri par ordre croissant du tas de n éléments

Pour effectuer le tri, on supprime le premier élément qui est le plus grand du tas (donc du tableau) en l'échangeant avec le nième élément du tas (le dernier) en utilisant la fonction suppression dans un tas (§1.2). On a alors le plus grand élément en position n-1 et la partie d'indice 0..n-2 du tableau forme encore un tas. On recommence en supprimant le premier par échange avec celui d'indice n-2, ... Le résultat est un tableau dont les éléments sont ordonnés par ordre croissant.

Attention : ne pas confondre le nombre d'éléments du tableau et le nombre d'éléments du tableau qui forment un tas. Ce sont deux informations différentes.

4 Travail à réaliser

4.1 Les fonctions à écrire pour réaliser ce tri sont donc les suivantes

Nous supposons ici que nous avons un tableau contenant `n` éléments, qui sont dans un ordre quelconque au début.

1. `void augmenteTas(double* tas, int i)` : ajoute le nombre dont l'indice est `i` au tas `tas` existant. L'élément à ajouter est déjà dans le tableau `tas`. Avant exécution de cette fonction, ce tableau forme déjà un tas de i éléments entre les indices 0 et i-1. Après exécution de cette fonction, ce tableau forme un tas de i+1 éléments entre les indices 0 et i. Le tableau contient cependant n éléments.
2. `void constructionTas(double* tas, int n)` qui établit un tas à partir du tableau de n éléments. Avant l'exécution de cette fonction, le tableau `tas` contient n éléments non organisés en tas. Après exécution, les éléments du tableau `tas` ont été réorganisés pour former un tas entre les indices 0 et n-1. Utilisez la fonction précédente en ajoutant les éléments les uns après les autres.
3. `void descendreTas(double* tas, int i)` : Cette fonction fait descendre le premier élément (la racine du tas, à l'indice 0) à une place compatible avec la notion de tas. Avant l'appel, les éléments entre les indices 1 et i-1 respectent les conditions d'un tas, mais pas l'élément d'indice 0. Après l'exécution de cette fonction, les éléments d'indice 0 à i-1 forment un tas. Le tableau contient cependant de n éléments.

4. `void suppressionTas(double* tas, int i)` : échange le premier nombre du tas `tas` de i éléments avec le dernier (indice i-1) et réorganise le tas entre les indices 0 et i-2 ensuite en utilisant la fonction précédente. Avant l'appel de cette fonction, le paramètre `tas` est un tas de i éléments (tableau). Après exécution, le paramètre `tas` est un tas de i-1 éléments, et l'ancienne racine est maintenant à l'indice i-1 dans le tableau `tas`. Le tableau contient cependant de n éléments.
5. `void triTas(double* tas, int n)`, qui trie le tableau `tas` par la méthode décrite ci dessus. La fonction construit un tas, puis ordonne les éléments selon la méthode décrite. Après exécution, le paramètre `tas` est tableau trié par ordre croissant de n éléments.

4.2 Programme principal

1. Ecrire un programme qui tri un tableau de nombres entrés au clavier. Vérifier sur des exemples simples.
2. Télécharger le fichier `testTas.c` qui lit un fichier contenant des tableaux, tri ces tableaux et les affiche. Il compare aussi le résultat avec le tri quicksort de la bibliothèque C, ce qui permet de savoir si le résultat est correct.

   3 fichiers de données de test sont fournis : test1.dat (tableaux de 8 nombres), test2.dat (9 nombres), test3.dat (10 nombres). Ajouter des tableaux pour ajouter vos propres tests. La premiere ligne du fichier contient le nombre d'elements dans un tableau. les lignes suivantes contiennent un tableau par ligne.
3. Modifier le programme précédent pour trier un tableau de nombres, alloué dynamiquement, dont la dimension est entrée au clavier et les valeurs tirées au hasard grace à la fonction `int rand()`

4.3 Comparer 2 tris

Voici un exemple de code, disponible dans le fichier `tempstri.c` comparant le tri quicksort (fonction `qsort` de la bibliothèque C) à votre fonction de tri. On relève les temps de calcul pour 10,20,30..90,100,200,300, ..1000,2000,....éléments tirés au hasard, puis triés par le quicksort et par votre fonction de tri. Les temps de calcul sont affichés mais aussi enregistrés dans un fichier texte `tempstri.dat`. Chaque ligne de ce fichier contient 3 nombre : le nombre d'éléments, le temps du QuickSort, le temps de votre tri.

```
#include <time.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

/* fonction de comparaison utilisee pour le quick sort */
int compar( const void* ax, const void * ay) {
  return( *((int *)ax)- *((int *)ay));
}

int  main(){
  int lo;       /* Nombre de reels a trier : 10E lo */
  clock_t avant, apres;
  double temps1, temps2;
  FILE* fp;
  int *t1, *t2;
  int i,j,k,l;

  /*  Nombre d'octaves  */
  lo = 6;
  /* Creation des tableaux a trier */
  if ( (t1=calloc(pow(10,lo+1),sizeof(*t1))) == NULL) {printf("Allocation impossible\n"); exit(1); }
  if ( (t2=calloc(pow(10,lo+1),sizeof(*t1))) == NULL) {printf("Allocation impossible\n"); exit(1); }
  /* Ouverture d'un fichier contenant les resultats */
  fp =fopen("tempstri.dat","w");
  /* Initialisation du generateur aleatoire */
  srand(getpid());
```

```
  /* Premiere boucle sur le nombre d'octave */
  for (k=10,i=0; i<lo; k*=10,i++)
    for (j=1; j<10; j++) {
  /* j*k est le nombre d'elements a trier */
    /* Tirage aleatoire des nombres a trier */
      for (l=0; l<j*k; l++) t1[l]=rand()%100;
    /* Copie dans le deuxieme tableau */
      memcpy(t2,t1,j*k*sizeof(*t1));
      avant = clock();
        /* Tri par quick sort */
      qsort(t1,j*k,sizeof(*t1),compar);
      apres = clock();
      temps1=(double)(apres - avant)/CLOCKS_PER_SEC;
        /* Tri par ma fonction */
      avant = clock(); tri(t2,j*k); apres = clock();
      temps2=(double)(apres - avant)/CLOCKS_PER_SEC;
        /* Affichage des temps des 2 tris */
      printf("%d\t%lf\t%lf\n", j*k, temps1, temps2);
      if (fp) fprintf(fp,"%d\t%lf\t%lf\n", j*k, temps1, temps2);
      /* Comparaison des resultats des 2 tris : Erreur si resultats differents */
      if (memcmp(t1,t2,j*k*sizeof(*t1))!=0)
        printf("Erreur de tri .....................................");
    }
  if (fp) fclose(fp);
  /* Liberation memoire */
  free(t1); free(t2);
  return EXIT_SUCCESS;
}
```

Pour tracer la courbe d'evolution des tris en fonction du nombre d'elements à partir des données qui sont enregistrées dans le fichier `tempstri.dat` après l'execution du programme `tempstri`, vous pouvez utiliser le logiciel `gnuplot`. Ce logiciel permet de tracer des courbes 2D, 3D de fonctions numériques ou de données, d'ajuster des courbes sur des données expérimentales, etc.. (voir http://www.gnuplot.info).

Il faut alors lancer la commande linux `gnuplot` qui vous affiche l'invite de commande suivante `gnuplot>`.

Les commandes gnuplot pour afficher l'evolution des temps sont :

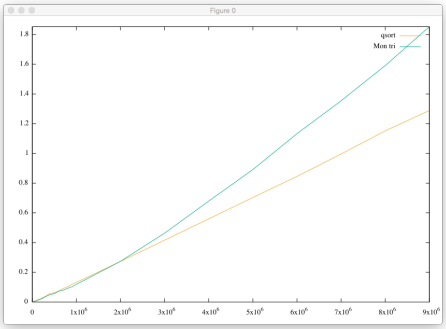

FIGURE 4 – Visualisation des temps des tris

```
gnuplot> plot 'tempstri.dat' u 1:2 with lines lt
4 title "qsort", "" u 1:3 with lines lt 2 title "Mon tri"
```

1. Faites varier le nombre d'éléments à trier en modifiant la variable `lo`. Cette variable est le logarithme du nombre maximal de données I.e le nombre de données maximal est 10^{lo+1}. Vous pouvez faire varier cette variable jusqu'à 6 ou 7 selon vos machines.

   Quel est le tri le plus rapide ?

2. Les nombres sont générés aléatoirement entre les valeurs 0 et RAND_MAX, qui vaut 2147483647 et est définie dans `stdlib.h`. 2 nombres tirés aléatoirement ont peu de probabilité d'être égaux pour quelques millions de tirage et nos tests comportent donc principalement des nombres différents. Pour créer des tableaux avec des nombres identiques, modifier le tirage par

   `for (l=0; l<j*k; l++) t1[l]=random()%100`.

   Dans ce cas, de nombreuses occurrences du meme nombre seront présentes. Compiler, executer, tracer les courbes.

   Quel est le tri le plus rapide ? Pourquoi ?