TP 13 : TAD Tas et exemple de tri

Notions : TAD, Arbre, Tris, Complexité, Arbres

1 Notion de tas

1.1 Introduction

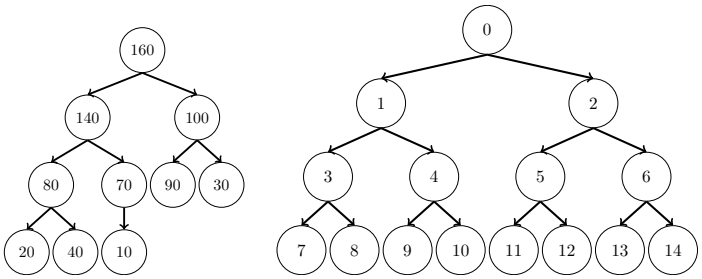
Un tas est un arbre binaire, parfait et partiellement ordonné :

— arbre binaire : un nœud de l'arbre a au plus 2 fils : 85 a pour fils 73 et 36.

— arbre 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.

— 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. C'est la propriété essentielle d'un tas.

La figure 1 donne un exemple de tas. Chaque noeud respecte la propriété essentielle d'un tas, i.e. il est supérieur à ces fils (2,1,0 fils).



Un exemple de tas — Numérotation des noeuds

FIGURE 1 – Un exemple de tas (à gauche) et la numérotation des noeuds (à droite)

1.2 Structure de données utilisée

Nous pouvons bien sûr représenter les arbres avec des structures chaînées récursives comme les listes : un noeud contient une donnée et 2 adresses, une pour le fils gauche et une pour le fils droit.

Dans le cas des arbres parfaits, tous les niveaux de l'arbre sont remplis sauf le dernier. Nous allons pouvoir utiliser une représentation par tableau en remarquant les relations existantes entre les indices d'un tableau et la numérotation des noeuds suivante :

Avec cette numérotation, on peut donc faire le tableau suivant 1 pour représenter le tas figure 1.

Dans le cas des arbres parfaits, cette représentation par tableau présente aussi l'avantage de pouvoir connaitre facilement les pères et les fils d'un noeud. Les relations suivantes relient les indices des noeuds i et ses père et fils :

— $pere(i) = (i-1)/2$

— $fils_{gauche}(i) = 2 * i + 1$

— $fils_{droit}(i) = 2 * (i+1)$

indice	0	1	2	3	4	5	6	7	8	9
valeur	160	140	100	80	70	90	30	50	40	10

TABLE 1 – Représentation de l'arbre précédent dans un tableau

1.3 L'interface du TAD heap

Les données seront du type `element_t` (des entiers dans nos exemples). Les fonctions utiles sur ces éléments ainsi que le type seront définis dans les fichiers `element.h` et `element.c`.

```
#ifndef _ELEMENT
#define _ELEMENT

typedef int element_t;
#define ELEMENT_COMPARE(e1,e2)  (*(e1)>*(e2) ? 1 : *(e1)==*(e2) ? 0 : -1)
#define ELEMENT_EQUAL(e1,e2)  (*(e1)==*(e2))

void element_print (element_t e);
int element_equal(element_t*, element_t*);
int element_compare(element_t*, element_t*);

#endif
```

La structure de données proposée pour le type `heap_t` dans le cadre de ce TP sera une structure (au sens du langage C), comprenant 3 champs :

— Un champ contenant le tableau des valeurs de type `element_t`, qui est alloué dynamiquement

— Un champ contenant le nombre maximal d'éléments du tableau alloué dynamiquement

— Un champ contenant le nombre actuel d'éléments du tas. A la création du tas, ce nombre est zéro (0).

Voici le fichier `heap.h`

```
#ifndef _HEAP
#define _HEAP
#include "element.h"

typedef  struct {
    element_t* data; // Zone de données : Tableau des valeurs du tas
    int max_size;       // Taille maximale du tas
    int number;         // Nombre actuel d'éléments du tas
  } heap_t;

// Crée, alloue dynamiquement le tableau et retourne un tas de m éléments au maximum;
heap_t heap_new(int m);
// Retourne 1 si le tas est vide et 0 sinon. Un tas vide a un nombre actuel nul;
int heap_is_empty(heap_t tas);
// Ajoute un element au tas en respectant la propriété d'ordre partiel; Retourne 1 si réussi et 0
   sinon.
int heap_add(element_t valeur, heap_t* ptas);
// Retourne la racine du tas sans la supprimer
element_t heap_get_max(heap_t tas);
// Supprime la racine du tas et réorganise du tas pour maintenir sa propriété essentielle d'ordre
   partiel. Retourne 1 si réussi et 0 sinon.
int heap_delete_max(heap_t* ptas);
// Vérifie que le tas respecte le propriété d'ordre partiel
int heap_verification(heap_t tas) ;
// Supprime tous les éléments, libère la mémoire et retourne un tas vide
void heap_delete(heap_t* ptas);

// Definition de macros pour accéder aux indices des pères, fils gauche et fils droit.
#define HEAP_FATHER(i) ( ((i)-1)/2)
#define HEAP_LEFTSON(i) ( 2*(i)+1)
#define HEAP_RIGHTSON(i) ( 2*((i)+1))
#endif
```

1.4 Ajout d'un élément au tas

Pour ajouter un élément dans un tas, on l'ajoute à la première place libre de l'arbre, c'est à dire à la fin du tableau des valeurs actuel. Puis, on remonte cet élément vers la racine en l'échangeant avec son père si l'ordre entre le père et le fils n'est pas respecté (le père est plus grand que le fils) On s'arrête soit à la racine, soit quand l'ordre entre le père et le fils est respecté.

Exemple 1 : ajout de 90 au tas

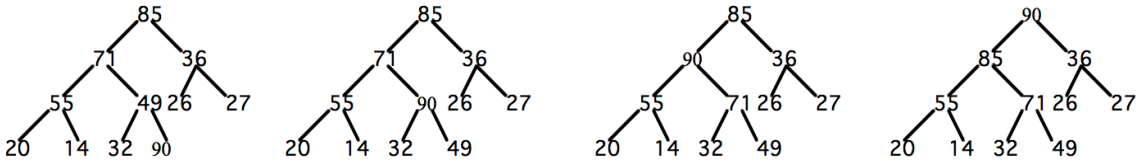


FIGURE 2 – Insertion d'un élément dans un tas

L'exemple figure 2 et la table 1.4 illustre les étapes de l'ajout du nœud de valeur 90 dans le tas initial (tableau). Le tableau des valeurs 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 2 – Insertion d'un élément dans un tas

L'algorithme est donc le suivant :

Algorithm 1 Ajout d'un élément au tas

Entrées: element, Tas de $size$ elements
Sorties: Tas de $size + 1$ elements

```
// Ajout d'un élément à un tas
i=indice du dernier élément+1
mettre l'élément à ajouter dans cette case de tableau
Tant que tant que i n'est pas la racine (i==0) Faire
    j = indice du pere de i
    Si la valeur du noeud j (c'est à dire du pere de i) est plus petite que la valeur du noeud i Alors
        echanger les valeurs des noeuds j et i
        i = j
    Sinon
        Quitter la boucle et fin de la fonction
    Fin Si
Fin Tant que
```

1.5 Suppression de la racine

Le tas permet de trouver facilement le maximum puisque c'est la racine de l'arbre. La plupart des algorithmes récupèrent cette valeur et la supprime du tas. La racine est alors vide et la structure n'est

plus un tas.

Pour réorganiser le tas, on remplace la racine par le dernier élément de l'arbre.

Puis, on descend cet élément vers les feuilles en l'échangeant avec le plus grand de ses deux fils si l'ordre entre le père et le fils n'est pas respecté (le père est plus grand que le fils)

On s'arrête soit sur une feuille (pas de fils), soit quand l'ordre entre le père et le plus grand fils est respecté.

L'exemple figure 3 et table 1.5 illustre les étapes de la suppression du nœud de valeur 85 dans le tas initial figure 1.

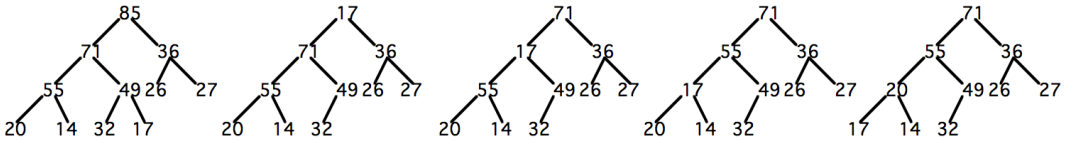


FIGURE 3 – Suppression d'un élément dans un tas

indice	0	1	2	3	4	5	6	7	8	9	10	
valeur	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		Racine = dernier element = 17
Etape 2	71	17	36	55	49	26	27	20	14	32		Echange de 17 et du plus grand de ses fils (71)
Etape 3	71	55	36	17	49	26	27	20	14	32		Echange de 17 et du plus grand de ses fils (55)
Etape 4	71	55	36	20	49	26	27	17	14	32		Echange de 17 et du plus grand de ses fils(20)

TABLE 3 – Suppression d'un élément dans un tas

L'algorithme est donc le suivant :

Algorithm 2 Suppression de la racine du tas

Entrées: Tas de $size$ elements
Sorties: Tas de $size - 1$ elements

```
// Suppression de la racine du tas et réorganisation du tas
i= indice de la racine =0
mettre le dernier élément à la racine
Tant que tant que i n'est pas l'indice d'une feuille Faire
    j = indice du plus grand des fils de i. Attention, i peut n'avoir qu'un seul fils !
    Si la valeur du noeud d'indice j est plus grande que la valeur du noeud i Alors
        echanger les valeurs des noeuds j et i
        i = j
    Sinon
        Quitter la boucle et fin de la fonction
    Fin Si
Fin Tant que
```

2 Application au tri par tas

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.

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

1. Construction du tas à partir d'un tableau de n éléments : on ajoute les éléments du tableau au tas les uns après les autres
2. Tri du tas : le plus grand élément est la racine : c'est donc le dernier du tableau. On met donc la racine dans le dernier élément du tableau, puis on supprime la racine du tas. Et on itère cette opération n fois.

L'algorithme est donc le suivant :

Algorithm 3 Tri avec tas

Entrées: Tableau : tab, Entier : size
Sorties: Tableau trié : tab

```
Créer un tas vide
Pour i=0 à i<size Faire
  Ajouter tab[i] au tas
Fin Pour
Pour i=0 à i<size Faire
  temp = racine du tas
  tab[size-1-i] = temp
  Supprimer la racine du tas
Fin Pour
Libérer le tas
//Le tableau est trié
```

3 Travail à réaliser

3.1 Le TAD `heap_t`

1. Télécharger le fichier `tp13.zip` et le décompresser.
2. Compléter les fonctions de `heap.c` : `heap_new`, `heap_add`, `heap_get_max`, `heap_delete_max`, `heap_delete`.

   Une fonction `int heap_largest_son(heap_t tas, int indice)` qui retourne l'indice du plus grand des deux fils du noeud dont le numéro est `indice` ou `-1` si ce noeud est une feuille pourra être utile.
3. Tester votre tas avec le programme principal proposé dans le fichier `testheap.c`. Vous pouvez comparer les résultats fournis par votre programme à ceux du fichier `ResultatsTestHeap.txt` qui contient la sortie espérée du programme.

3.2 Tri par tas

1. Compléter la fonction du fichier `heapsort.c` : `int heap_sort(element_t *tab, int size);` Cette fonction trie les `size` éléments du tableau `tab` à l'aide d'un tas selon l'algorithme 3 proposé.
2. Ecrire un programme principal qui permet de tester votre fonction de tri simplement.
3. Compiler le fichier `testheapsort.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 première ligne du fichier contient le nombre d'éléments dans un tableau. les lignes suivantes contiennent un tableau par ligne.

3.3 Comparer 2 tris

Voici un exemple de code, disponible dans le fichier `tempstri.c` comparant le tri quicksort (fonction `qsort` de la bibliothèque C), le tri par sélection fourni et votre fonction de tri. On moyenne les temps de calcul pour 10 tirages aléatoires de 10,100,1000,10000,....éléments tirés au hasard, puis triés par le quicksort, le tri par sélection et par votre fonction de tri.

Les temps de calcul sont affichés mais aussi enregistrés dans un fichier texte `tempstri.dat` qui est ensuite tracé à l'écran à l'aide de la commande unix `gnuplot`.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <unistd.h>
#include <sys/times.h>

#include "element.h"
#include "heap.h"
#include "heapsort.h"

#define MAX_DIM_SELECTION 10000

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

/* Tri par selection a titre de comparaison */
void selection_sort(element_t* t, int size) { int i,j,imax;
  for (i=0; i<size; i++) {
    imax=0;
      /* Recherche du maximum de la partie non triee du tableau*/
    for(j=1; j<size-i; j++)
      if (ELEMENT_COMPARE(t+imax,t+j)<0)
        imax=j;
        /* Echange du maxima et du dernier element non encore trie */
    element_t tmp=t[imax];
    t[imax]=t[size-i-1];
    t[size-i-1]=tmp;
  }
}

/*
  Ce programme affiche le temps d'execution du tri qsort de la bilbiotheque C et
  de votre tri par tas en fonction du nombre d'element a trier
*/

int  main(){
  int total_number;       /* Nombre de reels a trier : */
  clock_t avant, apres;
```

```
double temps1, temps2, temps3; // Temps de calcul des tris
FILE* fp;
int *t1, *t2, *t3;
int j,k,l;

/*  Nombre maximal d'elements  */
printf("Affichage et comparaison de votre heap_sort, du quick sort du C et du tri par selection\n");
printf("Au dela de %d elements, le tri par selection ne sera pas teste\n",MAX_DIM_SELECTION);
printf("Quel est le nombre maximal d'element a trie (0..100000000) ?"); fflush(stdout);
scanf("%d", &total_number);
if (total_number<=0) total_number = 1E5;

/* Creation des tableaux a trier : un avec le qsort, un avec votre tri */
if ( (t1=calloc(total_number,sizeof(*t1))) == NULL || (t2=calloc(total_number,sizeof(*t1))) == NULL
 || (t3=calloc(total_number,sizeof(*t1))) == NULL) {
  printf("Allocation impossible\n");
  exit(EXIT_FAILURE);
}

/* Ouverture d'un fichier contenant les resultats */
fp =fopen("tempstri.dat","w");

/* Initialisation du generateur aleatoire */
srandom(getpid());

/*  k est le nombre d'elements a trier, varie entre 1 et total_number */
for (k=1; k<=total_number; k*=10) {
  /* On moyenne les temps de chaque tri sur 10 realisations */
  /* On ne fait pas le tri par selection s'il y a trop d'elements */
  temps1=temps2=0;
  if (k<=MAX_DIM_SELECTION) temps3=0;
  for (j=1; j<10; j++) {
       /* Tirage aleatoire des nombres a trier */
    for (l=0; l<k; l++) t1[l]=random();
        /* Copie dans le deuxieme et troisieme tableau */
    memcpy(t2,t1,k*sizeof(*t1));
    memcpy(t3,t1,k*sizeof(*t1));

        /* Tri par quick sort */
    avant=clock();      qsort(t1,k,sizeof(*t1),compar);      apres=clock();
    temps1+=((double)apres - avant)/CLOCKS_PER_SEC;

      /* Tri par ma fonction heap_sort*/
    avant = clock();      heap_sort(t2,k);      apres = clock();
    temps2+=((double)apres - avant)/CLOCKS_PER_SEC;
    if (memcmp(t1,t2,k*sizeof(*t1))!=0) fprintf(stderr,"Erreur de tri
 ................................\n");

    /* Tri par selection s'il n'y a pas trop d'element */
    if (k<=MAX_DIM_SELECTION) {
      avant = clock();        selection_sort(t3,k);        apres = clock();
      temps3+=((double)apres - avant)/CLOCKS_PER_SEC;
    }
  }

     /* Affichage des temps des 3 tris a l'ecran et dans le fichier */
  if (k<=MAX_DIM_SELECTION) printf("%d\t%lf\t%lf\t%lf\n", k, temps1/10, temps2/10, temps3/10);
  else printf("%d\t%lf\t%lf\n", k, temps1/10, temps2/10);
  printf("Quick sort :%lf Mon tri par tas: %lf\n", temps1, temps2);
  if (fp) {
    if (k<=MAX_DIM_SELECTION) fprintf(fp,"%d\t%lf\t%lf\t%lf\n", k, temps1/10, temps2/10, temps3/10);
    else fprintf(fp,"%d\t%lf\t%lf\n", k, temps1/10, temps2/10);
  }
      /* Verification des tris */
  if (memcmp(t1,t2,k*sizeof(*t1))!=0) fprintf(stderr,"Erreur de tri\n");
}

/* Trace graphique des courbes de temps */
if (fp) {
  fclose(fp);
```

```
    system("gnuplot -p -e \"set logscale x; plot 'tempstri.dat' u 1:2 with line lt 4 title 'quickSort',
    '' u 1:3 with line lt 2 title 'MonTriHeap',  '' u 1:4 with line lt 6 title 'Tri selection'; quit\""
   );
  }

  /* Liberation memoire */
  free(t1); free(t2); free(t3);
  return EXIT_SUCCESS;
}
```

1. Faites varier le nombre d'éléments à trier en modifiant la variable `total_number`. Vous pouvez faire varier cette variable jusqu'à 10^5 ou 10^7 selon vos machines. Le tri par selection est limité par la constante `MAX_DIM_SELECTION` que vous pouvez aussi faire varier 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 ?