TP Algorithmique et Optimisation Discrète

Un *arbre binaire de recherche*[1] (*ABR*) est un arbre binaire tel que la valeur en tout nœud est strictement supérieure aux valeurs des nœuds de son sous arbre gauche et inférieure ou égale à celles de son sous arbre droit.

Les ABR sont une structure de données utilisée pour implémenter un dictionnaire, c'est à dire un ensemble totalement ordonné d'éléments. La complexité en pire cas de la recherche d'un élément dans un ABR est la profondeur de l'ABR.

Le but de ce TP est de générer un programme implémentant un ABR qui minimise le nombre moyen de comparaisons lors d'une opération de recherche ; un tel ABR est dit *optimal*.

Plus précisément, soit un ensemble E ordonné de n éléments $e_0 < e_1 < \ldots < e_{n-1}$ avec, pour $0 \leq i < n$, une probabilité p_i de rechercher l'élément e_i. Il s'agit de construire un ABR dont la profondeur moyenne est

$$\min_{A \in \{\text{ABR stockant } E\}} \sum_{i=0}^{n-1} p_i \times \Delta_A(e_i)$$

où $\Delta_A(e_i)$ vaut 1+ la profondeur de e_i dans A (par convention, la profondeur de la racine est 0). La figure 1 illustre les 5 configurations possibles pour un ABR optimal à trois éléments.

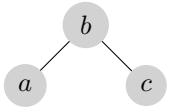
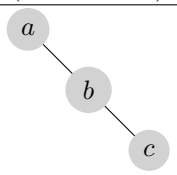
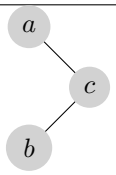
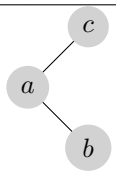
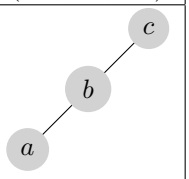
(p_a, p_b, p_c)	(0.2, 0.4, 0.4)	(0.5, 0.3, 0.2)	(0.6, 0.2, 0.2)	(0.3, 0.2, 0.5)	(0.1, 0.3, 0.6)
ABR optimal					
Nombre moyen de comparaisons	1.6	1.7	1.6	1.7	1.5

FIGURE 1 – Exemples d'ABR optimaux avec trois éléments $a < b < c$ de probabilités respectives p_a, p_b, p_c.

1 Équation de Bellman

Question 1 2 points Justifier que tout sous-arbre d'un ABR optimal est un ABR optimal. En déduire une modélisation sous forme d'équation de Bellman en précisant les conditions aux bords.

Question 2 1 point Quels sont, en notation Θ, l'espace mémoire et le nombre d'opérations requis par un programme (sans calcul redondant) qui calculerait un ABR optimal à partir de cette formulation récursive ?

2 Représentation en C d'un ABR et recherche générique

Dans toute la suite, les n éléments du dictionnaire $\{e_i, 0 \leq i < n\}$ sont supposés ordonnés par ordre croissant. On suppose disposer d'une fonction C qui prend en entrée un entier i ($0 \leq i < n$) et retourne l'adresse de l'élément e_i ; la figure 2 donne un exemple. Cette fonction a le type : `void* (*e)(int i)`.
On impose la représentation C suivante pour un ABR optimal associé à un ensemble trié de cardinal n (cf figure 3 pour des exemples) :

```
1        static int BSTroot ;
2        static int BSTtree [ ][ 2 ] = { ... } ;
```

avec la sémantique suivante en notant `e` la fonction d'accès aux éléments :

— `BSTroot` : entier dans $\{0, \ldots, n-1\}$ qui code la racine de l'ABR. Ainsi, `e(BSTroot)` est la valeur associée à la racine de l'ABR.

1. Binary Search tree (BST) en anglais

— `BSTtree` : tableau de n couples d'entiers qui codent les pointeurs de l'ABR : `BSTtree[ k ]` représente le nœud de l'ABR associé à la valeur `e( k )`. Ainsi `BSTtree[k][0]` (resp. `BSTtree[k][1]`) est un entier i qui vaut : soit -1 si `BSTtree[ k ])` n'a pas de fils gauche (resp. droit) ; soit $i \in \{0, \ldots, n-1\}$ tel que `BSTtree[ i ]` est le nœud racine du sous-arbre gauche (resp. droit) qui contient donc des éléments plus petits (resp. plus grands) que `e(k)`.

```
1  void* e( int i ) {
2     static char tab = { 'a', 'b', 'c' } ;
3     assert (i >=0) && (i < 3) ;
4     return (void*) &(tab[i]) ;
5  }
```

FIGURE 2 – Exemple de fonction d'accès aux éléments d'un ensemble de 3 éléments {'a','b','c'}.

— L'ABR à gauche de la figure 1 est représenté par :

```
1          static int BSTroot = 1;
2          static int BSTtree[3][2] = { {-1, -1}, { 0, 2}, {-1, -1} } ;
```

— L'ABR à droite de la figure 1 est représenté par :

```
1          static int BSTroot = 2;
2          static int BSTtree[3][2] = { {-1, -1}, { 0, -1}, {1, -1} } ;
```

FIGURE 3 – Exemples de représentation C d'ABR à 3 éléments

Algorithme générique de recherche dans un ABR

Un intérêt de la représentation précédente en C est de pouvoir facilement être utilisée par un algorithme de recherche générique.

Listing 1 – Un exemple d'algorithme générique de recherche dans un ABR (optimal)

```
1  /**
2   * \fn int BSTfind( void* (*e)(int), int (*cmp)(const void *, const void *), void* x)
3   * \brief recherche la valeur pointée par x dans e en utilisant un ABR interne.
4   * \param e fonction d'accès à l'ensemble; e(k) retourne le k-ième élément dans l'
        ordre croissant défini par cmp.
5   * \param cmp fonction de comparaison des valeurs pointées; cmp(a,b) retourne un
        entier <0, nul ou >0 ssi respectivement a<b, a==b, ou a>b.
6   * \param x pointeur vers la valeur à rechercher.
7   * \return i tel que ( cmp( e(i), x) == 0 ) s'il existe; -1 sinon.
8   * L'implementation de BSTfind utilise un ABR optimal défini par les variables
        statiques BSTroot et BSTtree.
9   */
10 int BSTfind( void* (*e)(int), const void* x, int (*cmp)(const void*, const void*))
11 {
12    int k = BSTroot ;
13    while (k != -1)
14    {
15      int cmp = compar( x, e(k) ) ;
16      if (cmp < 0) k = BSTtree[ k ][ 0 ] ;
17      else if (cmp > 0) k = BSTtree[ k ][ 1 ] ;
18      else return k ;
19    }
20    return -1 ;
21 }
```

Par exemple, l'algorithme `BSTfind` ci-dessus retourne l'indice de l'élément cherché si celui-ci existe dans le tableau et -1 sinon. Comme dans la fonction `qsort` de la librairie standard C, la fonction de comparaison :

`int (*compar)(const void *, const void *)`

prend en paramètre des pointeurs sur les deux éléments à comparer ; elle doit retourner un entier inférieur, égal ou supérieur à 0 si son premier argument est considéré être respectivement inférieur, égal ou supérieur au second.
Noter que le type `void*` désigne en C un pointeur générique qui peut être converti en un pointeur vers n'importe quel type (par cast). Les comparaisons sont donc effectuées via des pointeurs. La figure 4 donne des exemples d'une telle fonction de comparaison et de tableaux triés associés.

Listing 2 – Un tableau de `double` trié

```
int cmp(const void* a, const void* b)
{
  const double precision = 1e-6;
  if ( *(double*)a < *(double*)b
          *(1 - precision) ) return -1;
  if ( *(double*)a > *(double*)b i
          * (1 + precision) ) return 1;
  return 0 ;
}

double tab[3] = { 1.0,  1.58, 3.14 } ;
void* accestab( int i )
{
   assert((i >=0) && (i < 3));
   return (void*) &(tab[i]) ;
}

/** Exemple d'appels **/
cmp( &(tab[1]), &(tab[2]) );
cmp( accestab(1), accestab(2) );
double x=7 ;
BSTfind(accestab, (void*)&x, cmp);
```

Listing 3 – Un tableau de `char*` trié

```
int cmpstr(const void* a, const void* b)
{
  return strncmp((char*)a,(char*)b, 80);
}

void* e( int i )
{
   static char* t[]={ "AOD", "Ensimag",
          "University Grenoble-Alpes" };
   assert((i >=0) && (i < 3));
   return (void*) &(tab[i][0]);
}

/** Exemple d'appels **/
cmp_double( e(1), e(2) );
char mot[256];
strncpy(mot, "AOD", 200);
BSTfind(e, (void*)mot, cmpstr);
BSTfind(e, (void*)"Ensimag", cmpstr);
```

FIGURE 4 – Exemples d'utilisation d'ABR optimaux.

3 Programme et rapport demandés

On demande d'implanter un programme `compileBST` avec la spécification suivante : Le langage de programmation est laissé libre. Vous pouvez écrire ce programme dans le langage de votre choix, par exemple en C ou C++, ou encore Java ou Python [2] mais doit implémenter

`compileBST` nF

n est le nombre d'éléments dans le dictionnaire.

F est un fichier texte contenant n entiers positifs $o_O, \ldots, o_{n-1}$: o_i est le nombre d'accès à l'élément i. Ainsi la probabilité d'accès à l'élément $s[i]$ est $\frac{o_i}{\sum_{k=0}^{n-1} o_k}$.

Cette commande écrit sur la sortie standard un programme C qui représente un ABR optimal pour le dictionnaire sous la forme (cf figure 3 pour des exemples) :

```
static int BSTroot ;
static int BSTtree [ ][ 2 ] = { ... } ;
```

Le choix du langage de programmation est laissé libre.

2. Selon les fonctions externes utilisées (en particulier en Python et Java), la performance expérimentale pourra éventuellement s'avérer plus difficile à justifier.

Le rendu final doit être déposé sous teide avant le 10/11/2015 ; votre rendu *doit* être un répertoire nommé `TPAOD` que vous rendrez sous forme compressé dans une archive [3] `tgz` qui devra s'appeler `TPAOD.tgz` : le répertoire `TPAOD` (cf contenu en fin d'énoncé) incluant notamment un `Makefile`, les sources pour construire l'exécutable `bin/compileBST` et sa documentation `doc/index.html` ainsi qu'un rapport de 4 pages maximum `rapport/rapport.pdf`.

Question 3 2 points Evaluation des sources rendus. Tout programme dont le makefile ne permet pas de générer l'exécutable `bin/compileBST` recevra une note 0 (par exemple en cas d'erreur de compilation).

Question 4 8 points Rapport Ce rapport doit suivre le modèle fourni ; il décrit les choix d'implémentation et répond dans l'ordre aux questions suivantes :

1. (1 point) explication brève du principe de votre programme en précisant la méthode implantée (récursive, itérative) ;
2. (2 points) analyse du coût théorique de votre programme en fonction du nombre n d'éléments dans le dictionnaire :
   - nombre d'opérations en pire cas ;
   - place mémoire requise ;
   - analyse des défauts de localité sur le modèle CO (argumenter en s'appuyant sur le programme) ;
3. (2 points) compte rendu d'expérimentation :
   (a) la description synthétique de la machine et des conditions dans lesquelles les mesures ont été effectuées (pour permettre la reproductibilité des mesures) ;
   (b) un tableau donnant les temps d'exécution mesurés pour chaque benchmark indiqué (temps minimum, maximum et moyen sur 5 exécutions) ;
   (c) une réponse justifiée à la question : les temps mesurés correspondent-ils à votre analyse théorique (nombre d'opérations et défauts de cache) ?
4. (1 point) pour la qualité globale du rapport (présentation, concision et clarté de l'argumentation).

Question 5 5 points Évaluation automatique : un programme automatique testera la correction des programmes. Tous les tests échoués recevront automatiquement la note 0 (par exemple si le nom ou la spécification du programme ne sont pas respectés, ou si l'ABR généré n'est pas optimal, si le programme `bin/compileBST` n'est pas généré, semble ne pas terminer ou se plante à l'exécution, etc)

4 Informations

Informations sur le rendu Le rendu final doit être déposé avant le 10/11/2016 à 23h59 sur teide. Ce rendu `TPAOD.tgz` doit contenir un répertoire `TPAOD` incluant :

- un Makefile
- les sources pour créer l'exécutable `bin/computeABROpt`
- la documentation du programme `doc/index`
- un rapport de 4 pages maximum répondant aux questions `rapport/rapport.pdf`. La qualité globale du rapport est notée sur 1 point.

Une évaluation automatique sera réalisée sur le programme pour vérifier son fonctionnement.

3. cette archive peut être générée par la commande Unix : `tar cvfz TPAOD.tgz TPAOD`