Examen final de programmation impérative avancée

ÉNSIIE, semestre 2

mercredi 23 mars 2016

Durée : 1h45.
Tout document personnel autorisé (pas de prêt entre voisins).
Ce sujet comporte 3 exercices indépendants, qui peuvent être traités dans l'ordre voulu.
Il contient 7 pages.
Le barème est donné à titre indicatif, il est susceptible d'être modifié. Le total est sur 20 points.
Certaines questions, précédées par le symbole ($\star$) sont plus difficiles et pourront être traitées à la fin. Il va de soi que toute réponse devra être justifiée.

Exercice 1 : Modularité (8pts)

On souhaite écrire un programme qui prend en ligne de commande une chaîne de caractère et qui affiche à l'envers sur la sortie standard. Par exemple, sur `noel` on affiche `leon`. Pour cela, on procède de la manière suivante : on parcourt la chaîne dans l'ordre en insérant les caractères successifs dans une pile, jusqu'à atteindre `'\0'` ; puis on reparcourt à nouveau la chaîne depuis le début en remplaçant le caractère par celui qu'on enlève en haut de la pile jusqu'à ce que la pile soit vide.

Pour cela, on considère donc trois modules, le module pour les piles vu en cours :

pile.h

```
typedef struct pile_base* pile;

pile vide();
int est_vide(pile);
void push(pile*, int);
int pop(pile*);
```

et deux modules `chaine` et `main`.

Module `chaine`

Une fonction `inverse` qui prend en argument une chaîne de caractères et qui inverse ses lettres en utilisant l'algorithme ci-dessus.

Module `main`

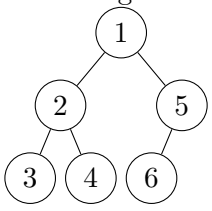
Une fonction `main` qui applique la fonction `inverse` sur le premier argument passé en ligne de commande et qui l'affiche ensuite sur la sortie standard.

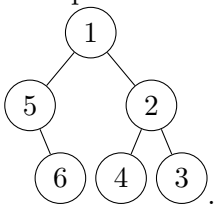
1. Quelles sont les relations de dépendance entre les modules ?
2. Quelle commande faut-il écrire manuellement pour compiler le module `chaine` ?
3. En supposant que tous les modules sont compilés, quelle commande faut-il taper manuellement pour faire l'édition de liens ?
4. On modifie l'interface de `pile`. Quel(s) module(s) faut-il recompiler ?
5. On modifie l'implémentation de `chaine`. Quel(s) module(s) faut-il recompiler ?
6. Écrire l'interface du module `chaine`.
7. Écrire l'implémentation du module `chaine`.
8. Écrire le `Makefile` correspondant au projet. On n'oubliera pas d'écrire une cible pour produire l'exécutable final qu'on appellera `prog`.

Exercice 2 : Arbres binaires d'entiers (4pts)

On rappelle la structure des arbres binaires d'entiers :

```
typedef struct ab* ab;
struct ab {
  ab fg;
  int c;
  ab fd; };
```

1. Une feuille est un nœud dont les deux fils sont vides. Écrire une fonction `nb_feuilles` qui prend en entrée un arbre binaire et qui retourne le nombre de feuilles qu'il contient.
2. Donner la complexité dans le pire des cas de la fonction `nb_feuille` en fonction de la hauteur h de l'arbre.
3. Écrire une fonction `miroir` qui prend en argument un arbre et qui le transforme en son image miroir. Par exemple, 

devient 

.
Indication : comme on ne change pas quel nœud est la racine de l'arbre, on n'est pas obligé de passer l'argument de `miroir` par référence.
4. Donner la complexité dans le pire des cas de la fonction `miroir` en fonction de la hauteur h de l'arbre.

Exercice 3 : Arbres 2–3 (8pts)

Un arbre 2–3 est une généralisation des arbres binaires de recherche dans lesquels les nœuds peuvent contenir de 1 ou 2 associations. On a trois possibilités :

— le nœud est une feuille, donc ne possède aucun fils ; il peut contenir 1 ou 2 associations ;

— le nœud n'est pas une feuille et contient 1 association $k \mapsto v$, il a alors exactement 2 fils ;

dans ce cas, comme dans un arbre binaire de recherche, on suppose que les deux fils sont des arbres 2–3, et que le fils gauche possède des clefs plus petites que k et que le fils droit possède des clefs plus grandes que k ;

— le nœud n'est pas une feuille et contient 2 associations $k_1 \mapsto v_1$ et $k_2 \mapsto v_2$, il a alors exactement 3 fils ;

dans ce cas, on suppose que les trois fils sont des arbres 2–3, et que le fils gauche possède des clefs plus petites que k_1, que le fils du milieu possède des clefs entre k_1 et k_2 et que le fils droit possède des clefs plus grandes que k_2.

De plus, les arbres 2–3 doivent être par construction parfaitement équilibrés, c'est-à-dire que tout chemin partant de la racine vers une feuille doit avoir la même longueur. Autrement dit, pour chaque nœud de l'arbre, les hauteurs de tous ses fils doivent être égales.

Pour simplifier, on ne prendra pas en compte les valeurs associées, et on se contentera de travailler avec les clefs.

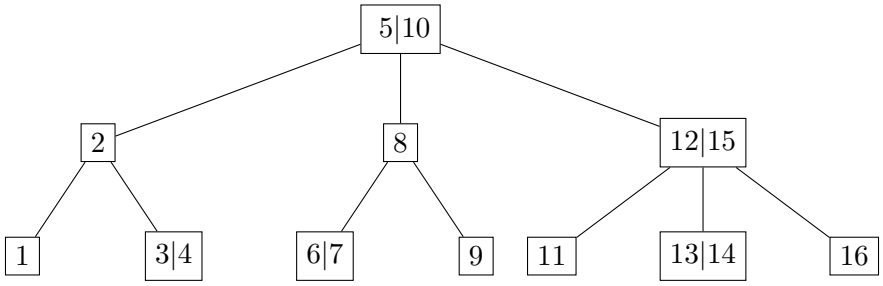


FIGURE 1 – Exemple d'arbre 2–3 contenant comme clefs les entiers de 1 à 16

1. Montrer qu'un arbre 2–3 de hauteur h possède au moins $2^h - 1$ associations.
2. Montrer qu'un arbre 2–3 de hauteur h possède au plus $3^h - 1$ associations.

On propose le type C suivant pour les arbres 2–3 :

```
typedef int clef;

typedef struct arbre23* arbre23;
```

```
struct arbre23 {
  int nbclefs; /* peut valoir 1 ou 2 */
  arbre23 fg;
  clef k1;
  arbre23 fm;  /* uniquement utile si nbclefs == 2 */
  clef k2;     /* uniquement utile si nbclefs == 2 */
  arbre23 fd;
};
```

où `fg` est le fils gauche, `fd` le fils droit, et `fm` le fils du milieu dans le cas où on a deux clefs.

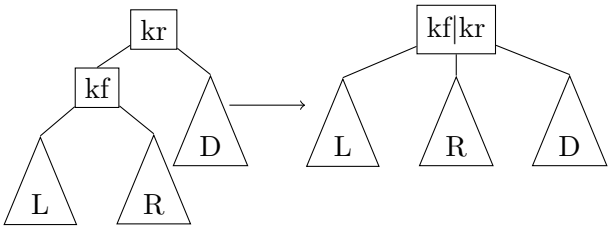
3. Écrire une fonction `rechercher` qui prend en argument un arbre 2–3 et une clef, et qui retourne 1 ssi la clef est présente dans l'arbre 2–3.
4. Donner la complexité dans le pire des cas de `rechercher` en fonction de n le nombre d'associations dans l'arbre 2–3.

L'idée de base de l'insertion est la même que pour les arbres binaires de recherche. On descend dans l'arbre comme pour une recherche jusqu'à arriver à un fils vide où on met un nouveau nœud contenant la clef. Néanmoins, cette opération change la hauteur du père du nouveau nœud. On va donc chercher à remonter le fils en utilisant le fait qu'un nœud peut stocker 2 clefs.

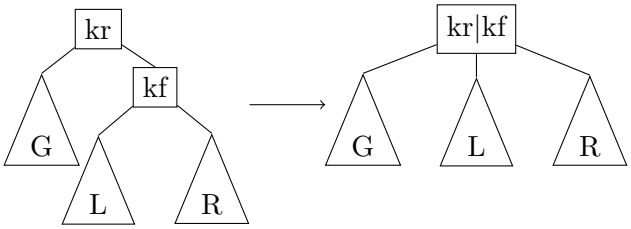
On considère donc une fonction `remonter` qui prend en argument un arbre dont un des fils ne possède qu'une seule clef à la racine et a une hauteur de 1 plus grande que celle de ses frères. Pour rééquilibrer l'arbre :

— Soit la racine ne contient qu'une seule clef : dans ce cas, il suffit de rajouter la clef de la racine du fils dans la deuxième case de la racine du père. On a deux cas suivant quelle clef est la plus petite.

cas 1 :

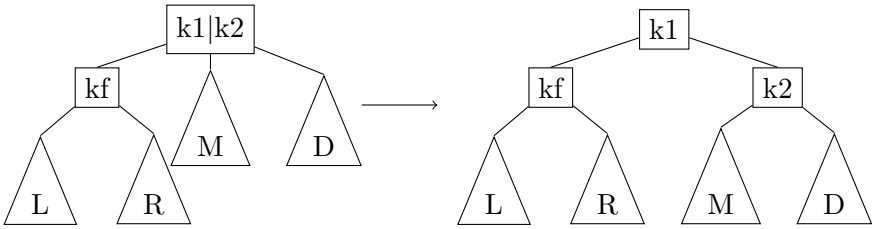


cas 2 :

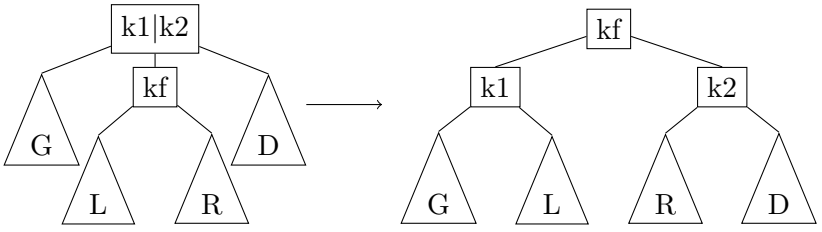


— Soit la racine contient deux clefs : dans ce cas, on stocke les trois clefs (clefs de la racine et clef du fils) dans trois nœuds (un père et ses deux fils), en respectant l'ordre des clefs. On a trois cas en fonction de cet ordre.

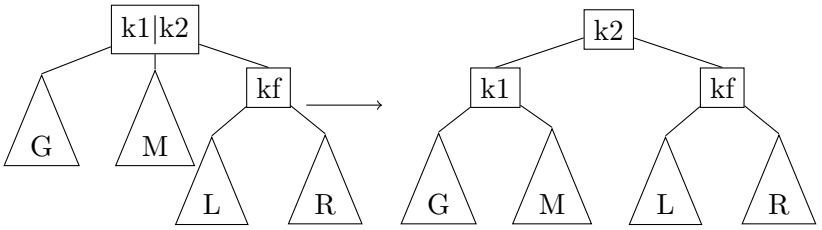
cas 3 :



cas 4 :



cas 5 :



On obtient le code suivant, où `noeud2(fg, k, fd)` crée un nouveau nœud avec une seule clef égale à `k` et comme fils gauche et fils droit `fg` et `fd`

```
/*@ requires \valid(*racine) && \valid(fils);
    requires fils est un des fils de *racine
    requires fils->nbclefs == 1;
    assigns racine, *fils;
    ensures remonte fils au niveau de *racine
            retourne 1 ssi la hauteur de *racine ne diminue pas  */
int remonter(arbre23 fils, arbre23 *racine) {
  if ((*racine)->nbclefs == 1) {
    (*racine)->nbclefs = 2;
    if (fils->k1 <= (*racine)->k1) { /* cas 1 */
      (*racine)->k2 = (*racine)->k1;
      (*racine)->k1 = fils->k1;
      (*racine)->fg = fils->fg;
      (*racine)->fm = fils->fd;
    } else {                         /* cas 2 */
      (*racine)->k2 = fils->k1;
      (*racine)->fm = fils->fg;
      (*racine)->fd = fils->fd;
    };
    free(fils);
    return 0;
  } else {
```

```
    if (fils->k1 <= (*racine)->k1) { /* cas 3 */
      (*racine)->nbclefs = 1;
      (*racine)->fd =
        noeud2((*racine)->fm, (*racine)->k2, (*racine)->fd);
    } else if (fils->k1 >= (*racine)->k2) { /* cas 5 */
      (*racine)->nbclefs = 1;
      (*racine)->fd = (*racine)->fm;
      *racine = noeud2(*racine, (*racine)->k2, fils);
    } else {                          /* cas 4 */
      (*racine)->nbclefs = 1;
      fils->fd = noeud2(fils->fd, (*racine)->k2, (*racine)->fd);
      (*racine)->fd = fils->fg;
      fils->fg = *racine;
      *racine = fils;
    };
    return 1;
  };
}

/*@ requires \valid(a);
    assigns a;
    ensures insere la clef k dans a
            retourne 1 ssi la hauteur de a a augment'e */
int inserer(arbre23* a, clef k) {
  if (!(*a)) {
    *a = noeud2(NULL, k, NULL);
    return 1;  };
  if ((*a)->k1 >= k) {
    int r = inserer(&((*a)->fg), k);
    if (r) return remonter((*a)->fg, a);
    else return 0; }
  if ((*a)->nbclefs == 1) {
    int r = inserer(&((*a)->fd), k);
    if (r) return remonter((*a)->fd, a);
    else return 0; }
  if ((*a)->k2 >= k) {
    int r = inserer(&((*a)->fm), k);
    if (r) return remonter((*a)->fm, a);
    else return 0; }
  else {
    int r = inserer(&((*a)->fd), k);
    if (r) return remonter((*a)->fd, a);
    else return 0; }
}
```

5. Donner l'arbre 2–3 obtenu en insérant dans l'ordre les entiers de 1 à 10, en détaillant chaque étape.
6. Donner la complexité de `remonter` en fonction du nombre de nœuds dans `*racine`.
7. En déduire la complexité dans le pire des cas de `inserer` en fonction du nombre de nœuds dans `*a`.
8. ($\star$) Proposer une fonction pour la suppression qui respecte les invariants des arbres 2–3.

   Indication : On peut procéder comme dans les arbres binaires de recherche et échanger la clef à supprimer avec la plus grande clef du sous-arbre à gauche de la clef à supprimer.

   Indication : La plus grande clef d'un sous-arbre est forcément une feuille.