TD2 : Pointeurs

1 Rappels

La mémoire de l'ordinateur est une suite de bits indifférenciables les uns des autres ; en particulier, une portion de mémoire de 32 bits successifs n'a pas de type intrinsèque. C'est le compilateur qui va la considérer comme un `int` : les types sont créés et gérés par le compilateur. On peut connaître la taille d'un type avec `sizeof` : ainsi `sizeof(int)` donne la place mémoire occupée par un entier.

Le type d'un pointeur `p` conditionne (pour le compilateur) le type qu'il doit assigner à la donnée pointée par `p`. Si `p` est un `float*`, le compilateur considèrera `*p` comme un `float`. Quelques remarques :

- `int * tab ;` et `int tab[] ;` sont équivalents (bien noter qu'on a écrit `int tab[] ;` et non pas `int tab[10] ;`).
- on peut forcer le compilateur à ignorer le type. Par exemple :
  `int *a=new int; double *b; b=(double*)a;`
  est autorisé.
- un tableau double (ou plus) est en fait un tableau de tableaux : `int ** a ;` est un tableaux de `a[i]` qui sont eux-même des tableaux (il faut quand même allouer les `a[i]`.

Pour accéder à l'objet pointé par un pointeur `p`, il faut utiliser `*p`, ou la formulation à base de tableaux : `p[0]`.

- `p+4` est l'adresse du 5-ième objet du tableau pointé par `p`. Donc `p[i]` $\Leftrightarrow$ `*(p+i)`.
- pour une structure `typedef struct toto {int a ;float b ;}toto ;` , et un pointeur sur un `toto` : `toto* p ;`, `p->a` est le champ `a` du `toto` pointé par `p`. Donc `p->a` $\Leftrightarrow$ `(*p).a` .

Déclarer un pointeur n'est pas une réservation de mémoire. `int* p ;` ne fait que déclarer `p`, mais ne réserve en aucun cas un entier sur lequel pointerait `p`. C'est le programme qui doit se charger des réservations de mémoire.

En C, on utilise `void* malloc(int taille)` pour réserver de la mémoire et `free(void* pointeur)` pour la rendre. Par exemple :

```
int *p; p=(int*)malloc(sizeof(int)); // on utilise p...  free(p);
```

déclare `p` et réserve suffisamment de place pour un entier. À la fin, on rend la mémoire réservée.

En C++, `new` réserve de la mémoire et `delete` la rend. Le code précédent devient :

```
int *p; p=new int; // on utilise p...  delete p;
```

En C, on peut aussi agrandir une réservation de mémoire avec `realloc` :

```
int *p; p=(int*)malloc(sizeof(int)); // on utilise p...
p=(int*)realloc(p,3*sizeof(int)); // on utilise p...  free(p);
```

ce qui permet de ne pas écrire explicitement un tableau quand on l'agrandit.

2 Arbres binaires de recherche

Un arbre binaire sera décrit par un ensemble de nœuds reliés par des pointeurs. On utilisera ici la structure :

```
typedef struct arbre { int valeur; arbre* pere; arbre* filsG; arbre*
  filsD; } arbre;
```

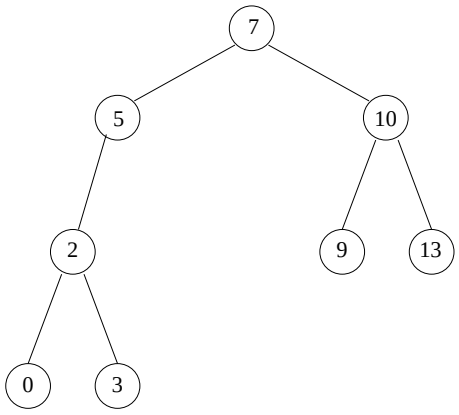
où `filsG` est le fils gauche et `filsD` le fils droit. Par convention, si le nœud n'a pas de fils gauche, on met `filsG=NULL`. Ainsi la racine d'un arbre est le seul nœud tel que `pere=NULL` .

Un arbre binaire de recherche est un arbre binaire dans lequel chaque nœud est étiqueté par une valeur n telle que :

– tous les nœuds du sous-arbre gauche ont une valeur inférieure à n,

– tous les nœuds du sous-arbre droit ont une valeur supérieure à n.

On ne considère ici que des arbres binaires de recherche qui ne contiennent pas deux fois la même valeur.



Un arbre binaire de recherche.

2.1 Recherche dans un ABR

1. Écrire une fonction (récursive) `bool recherche(arbre* a,int x)` qui retourne `true` si `x` est une des valeurs contenues dans l'arbre `a` et `false` sinon.

2. Comment trouver le plus grand élément d'un arbre binaire de recherche ? Ecrire une fonction qui prend en argument un arbre binaire de recherche et qui renvoie la valeur de son plus grand élément.

2.2 Insertion dans un ABR

Pour insérer un élément dans un arbre binaire de recherche, on crée une nouvelle feuille dans l'arbre. On procède de la façon suivante : on parcourt l'arbre récursivement pour déterminer à quel endroit on doit créer la nouvelle feuille, puis on crée un nouveau nœud et on l'insère à cet endroit.

La procédure peut se décrire de la façon suivante :

– si l'arbre est vide (cas de base), on renvoie un arbre constitué d'un seul nœud qui contient la valeur à insérer,
– sinon, trois cas sont possibles : soit l'élément à insérer est inférieur à la valeur de la racine, et on doit insérer l'élément dans le sous-arbre gauche de la racine, soit l'élément est plus grand que la valeur de la racine, et on l'insère dans le sous-arbre droit, soit la valeur de la racine est égale à l'élément à insérer, et on ne fait rien.

1. Donner le résultat de l'insertion des éléments 8, 6 et 4 dans l'arbre binaire de recherche de la figure.

2. Écrire une fonction `arbre* insere(arbre* a,int x)` qui insère une valeur `x` dans un ABR `a`.

2.3 Suppression de valeurs

Pour supprimer un nœud d'un arbre binaire de recherche, il y a deux cas simples : soit le nœud est une feuille, alors on le supprime tout simplement, soit le nœud n'a qu'un fils, on le supprime et on le remplace par son fils.

On a vu que l'élément maximal d'un arbre binaire de recherche n'a pas de fils droit, donc il a au plus un fils.

1. Écrire une fonction qui prend en argument un arbre binaire de recherche et qui renvoie l'arbre privé de son élément maximal (on ne s'occupe pas de la valeur de cet élément).

Pour supprimer un nœud qui a deux fils, on remplace la valeur de ce nœud par la valeur du plus grand élément de son sous-arbre gauche (qu'on supprime de là où il est). On peut également remplacer la valeur du nœud à supprimer par le plus petit élément de son sous-arbre droit.

2. Écrire une fonction qui prend en argument un arbre binaire de recherche et un entier, et qui renvoie l'arbre binaire de recherche privé de cet entier.

3 Matrices

Une matrice est un tableau double. On peut considérer `int ** a ;` comme un tableau de colonnes ou un tableau de lignes. On prend la convention d'un tableau de colonnes. On peut réserver la mémoire pour une matrice de plusieurs manières :

– en réservant un tableau par colonne, plus un tableau de pointeurs sur les colonnes,
– en réservant l'espace mémoire suffisant pour mettre toutes les colonnes les unes à la suite des autres, et un tableau de pointeurs sur les colonnes,
– soit en réservant l'espace pour tous les coefficients et en utilisant une numérotation des coefficients réservés ; par exemple, le $(i*20+j)$-ième coefficient réservé est associé à $matrice[i][j]$.

Écrire les opérations suivantes :

1. Une fonction `int** cree(int n,int m)` qui crée une matrice `n`$\times$ `m` avec des `malloc` et en utilisant la première méthode.

2. Une fonction qui crée une matrice en utilisant la 2-ième méthode.

3. Une fonction `int** double(int** a,int n)` qui double la taille d'une matrice ($n \times n$ devient $2n \times 2n$).

4. Une fonction `void swap_c(int **a,int k,int l,int n,int m)` qui change la colonne k avec la colonne l.

5. Une fonction `void swap_l(int **a,int k,int l,int n,int m)` qui change la ligne k avec la ligne l.

6. Une fonction `int** multiplie(int **a,int **b,int n)` qui multiplie 2 matrices carrées.

7. Une fonction `void transpose(int** a,int n)` qui transpose une matrice carrée (en place).