TP2 : Arbres

Rappels :
- Les TPs de tout le semestre seront réalisés en `C++`. Le but n'est pas d'utiliser toute la puissance du `C++`, mais simplement quelques outils qui nous faciliteront la tâche, en particulier la fonction `cout` à la place de `printf`, et la structure de donnnées `vector` pour faire des tableaux dynamiques.
- Les fichiers à télécharger sont disponibles à http://pagesperso.g-scop.fr/~pastorl/ (ou bien, tapez *Lucas Pastor* dans Google, trouvez sa page web, et descendez jusqu'à *Teaching*).
- Les questions marquées d'une ou plusieurs étoiles sont plus compliquées. Elles ne sont à traiter que lorsque le reste est fait.
- **Toutes vos fonctions doivent être testées au fur et à mesure.**

Ce TP porte sur les arbres dits "d'informaticiens", c'est-à-dire des arbres ayant une racine, que l'on représente en haut, puis éventuellement des nœuds internes, et enfin des feuilles, que l'on représente en bas. Pour ce TP, nous allons nous restreindre aux arbres binaires (chaque nœud peut avoir 0, 1 ou 2 fils). Pour cela, nous utiliserons la structure suivante en `C++` :

```
struct ArbreB{
    ArbreB(string et) {etiquette=et; fg=NULL; fd=NULL;}
    ArbreB(string et, ArbreB* filsGauche, ArbreB* filsDroit) {etiquette=et; fg=filsGauche; fd=filsDroit;}
    string etiquette;
    ArbreB* fg; /* pointeur vers le fils gauche; NULL si vide */
    ArbreB* fd; /* pointeur vers le fils droit; NULL si vide */
};
```

Ainsi, on définit un nouveau type `ArbreB`, qui a trois composantes (c'est ce que l'on lit dans les trois dernière lignes de l'accolade) : un `string` nommé `etiquette`, et deux pointeurs vers des `ArbreB`, nommés repsectivement `fg` (pour le fils gauche) et `fd` (pour le fils droit). Si le fils gauche est vide (c'est-à-dire, la racine n'a pas de fils gauche), on prend la convention que `fg` vaut le pointeur nul `NULL`, et de même si le fils droit est vide. Les deux premières lignes dans l'accolade sont des constructeurs, on verra dans l'Exercice 1 comment s'en servir. Une fois que l'on a réussi à avoir une variable `A` de type `ArbreB*` (un pointeur vers un `ArbreB`), on peut accéder à ses composantes par l'opérateur `"->"` : on tape `A->etiquette` pour l'étiquette, `A->fg` pour le fils gauche, et `A->fd` pour le fils droit.

Exercice 1 : Mise en route

Téléchargez les fichiers `Arbres.cpp`, `affichage.h` et `affichage.cpp`. Compilez-les à l'aide de la commande `g++ affichage.cpp Arbres.cpp -o arbres` puis testez avec `./arbres`.

Lisez maintenant le code de `Arbres.cpp` pour comprendre comment sont construits les arbres. On observe en particulier que l'on a deux constructeurs :

- Le premier `ArbreB(string et)` qui
- Le deuxième `ArbreB(string et, ArbreB* filsGauche, ArbreB* filsDroit` qui

De plus, le mot-clé `new` permet d'allouer automatiquement la mémoire nécéssaire à la création d'un `ArbreB`, et renvoie un pointeur sur l'`ArbreB` nouvellement créé.

Une fois qu'un arbre est construit, on peut le modifier :
- En faisant `A1->etiquette = "cc";`, on remplace l'étiquette de la racine de `A1` par `"cc"`.
- En faisant `A2->fg = <description du nouvel arbre>`, on remplace le fils gauche de A2 (qui était pour l'instant vide, c'est-à-dire égal à `NULL`) par un nouvel arbre correspondant à la description.

Il faut remarquer que l'on peut utiliser plusieurs fois l'opérateur flèche `->` successivement : ainsi `A2->fg->fd->etiquette` renvoie (vérifiez bien avec l'affichage que c'est logique !).

Maintenant, *à vous de jouer !*

1. Modifiez `A2` pour que le nœud étiquetté `s3` reçoive comme fils gauche l'arbre `A1`, et que le nœud étiquetté par `u` reçoive comme fils gauche l'arbre `B1`. Ré-affichez ensuite `A2`.
2. Affichez le sous-arbre enraciné au nœud d'étiquette `u` grâce à la commande `affiche_arbre(A2-> ...);` (les pointillés sont bien sûrs à compléter).

3. Supprimez dans `A2` le sous-arbre enraciné au nœud d'étiquette `u`. Ré-affichez ensuite `A2`.

Exercice 2 : Générer un arbre grâce à une fonction

Il existe une deuxième façon de créer un arbre : au lieu de le créer entièrement dans le main, on va appeler une fonction. Décommenter les lignes `ArbreB* F=arbre_formule();` et `affiche_arbre(F, "F");` et `libere(F)` (en bas) puis comparer l'affichage de `F` au contenu de la fonction `arbre_formule`. Dans tous les cas, il faut libérer l'espace alloué aux arbres avant de sortir du main. Pour cela, on utilise la fonction `libere` qui libère l'arbre `A` et tous ses sous-arbres (NB : elle utilise la fonction `free`, que vous connaissez probablement).

À vous !

1. Écrivez une fonction `arbre_formule2` ne prenant aucun paramètre et générant un arbre dont :
   - La racine est étiquettée `"*"`,
   - Le fils gauche est un arbre créé en appelant `arbre_formule`,
   - Le fils droit est un arbre à 3 nœuds correspondant à la formule $4y$.
2. Écrivez une fonction `arbre_complet_a(int n)` générant un arbre complet de hauteur `n`, dont toutes les étiquettes sont `"a"`. On rappelle qu'un arbre complet est un arbre dont tous les nœuds à distance inférieure ou égale à `n-1` de la racine ont exactement 2 fils, et dont tous les nœuds à distance exactement `n` de la racine ont 0 fils.
   Note : dans le `main`, vous pouvez donc appeler cette fonction pour créer un arbre complet puis modifier ensuite les étiquettes à la main.
3. (*) Écrivez une fonction `arbre_complet(int n)` générant un arbre complet de hauteur `n`, dont les étiquettes suivent la règle suivante :
   - La racine a pour étiquette `"r"`.
   - Soit s un nœud qui n'est pas la racine, p son père et $prefixe$ l'étiquette de p. Alors l'étiquette de s est $prefixe$ suivi de 0 si s est le fils gauche de p, et $prefixe$ suivi de 1 si s est le fils droit de p.

   Ainsi, dans l' arbre complet de hauteur au moins 4, le nœud d'étiquette `r1001` est atteint en partant de la racine et en suivant le chemin descendant suivant : fils droit, fils gauche, fils gauche, fils droit. Vous pourrez vous aider d'une fonction auxiliaire `arbre_complet_aux(int n, string prefixe)`.
4. (*) Écrivez une fonction `arbre_from_tableau(vector<string> &tab)` qui construit un arbre de la manière suivante : si `tab` n'a qu'une case, alors l'arbre est une feuille dont l'étiquette est le contenu de la case `tab[0]` ; sinon :
   - La racine reçoit comme étiquette le contenu de la case `milieu` de `tab`, avec `milieu = size/2` (où `size` doit être la taille de `tab`),
   - la première moitié de `tab` (cases `0` à `milieu-1`) est utilisée pour construire récursivement le fils gauche de la racine,
   - la deuxième moitié de `tab` (cases `milieu+1` à `size-1`) est utilisée pour construire récursivement le fils droit de la racine.

   Note : Pour cela, on peut créer à partir d'un vector `t`, un autre vector `partie` contenant une copie des cases `i` à `j` de `t` grâce à la commande suivante : `vector<string> partie (t.begin()+i, t.begin()+j+1);`
   (si `j` est la dernière case de `t`, on peut aussi utiliser `t.end()` à la place de `t.begin()+j+1` .)

Exercice 3 : Jouons avec les arbres !

1. Écrivez une fonction `feuille(ArbreB* A)` qui renvoie un booléen valant `true` si l'arbre `A` est une feuille, `false` sinon. *Note : un nœud est soit une feuille s'il n'a aucun fils, soit un nœud interne s'il a un ou deux fils.*
2. Écrivez une fonction `profondeur(ArbreB* A)` qui renvoie un entier valant la profondeur de l'arbre `A`. *Note : la profondeur est la distance entre la racine et la feuille la plus éloignée.*
3. Écrivez une fonction `nb_feuilles(ArbreB* A)` qui renvoie un entier valant le nombre de feuilles de l'arbre `A`.
4. Écrivez une fonction `nb_noeuds_internes(ArbreB* A)` qui renvoie un entier valant le nombre de nœuds internes de l'arbre `A`.
5. Écrivez une fonction `concatene_feuilles(ArbreB* A)` qui renvoie un `string` composé de la concaténation de toutes les étiquettes des feuilles (mais pas celles des nœuds internes). *Note : on peut concaténer deux `string a` et `b` grâce à l'opérateur `+` : `a+b`.*
6. Écrivez une fonction `concatene_etiquettes(ArbreB* A)` qui renvoie un `string` composé de la concaténation de toutes les étiquettes des nœuds de l'arbre.

Exercice 4 : Parcours préfixe, infixe et postfixe

1. Ecrivez une fonction `string parcours_infixe(ArbreB* A)` qui renvoie le parcours infixe de l'arbre A. Testez-la sur `F`, vous devez obtenir `3 * x + 5 - x * 2 * x + 1`.

2. Ecrivez une fonction `string parcours_prefixe(ArbreB* A)` qui renvoie le parcours préfixe de l'arbre A. Testez-la sur `F`, vous devez obtenir `- + * 3 x 5 * x + * 2 x 1`.
3. Ecrivez une fonction `string parcours_postfixe(ArbreB* A)` qui renvoie le parcours postfixe de l'arbre A. Testez-la sur `F`, vous devez obtenir `3 x * 5 + x 2 x * 1 + * -`.

Exercice 5 : Parcours en largeur

Écrivez une fonction `string parcours_largeur(ArbreB* A)` qui renvoie le parcours en largeur de l'arbre A. Il vous est nécessaire d'utiliser une file. Heureusement, cette structure de données est déjà implémentée dans la librairie standard du C++ sous le nom `queue`. Pour savoir comment l'utiliser, consultez la documentation en ligne :

http://www.cplusplus.com/reference/queue/queue/ (ou tapez `queue C++` dans Google)

Testez votre fonction sur `A2`, vous devez obtenir `r s1 t s2 s3 v cc a b`.

Exercice 6 : Arbre binaire de recherche

Un *arbre binaire de recherche* (*ABR*) est une structure de données très utile. Il s'agit d'un arbre dont les étiquettes sont des entiers, avec la propriété suivante : si un nœud N a pour étiquette k, alors toutes les étiquettes apparaissant dans le sous-arbre fils gauche de N sont des entiers inférieurs ou égaux à k ; et toutes les étiquettes apparaissant dans le sous-arbre fils droit de N sont des entiers supérieurs ou égaux à k. Ainsi, si l'on vous donne un entier d, vous pouvez très rapidement trouver dans l'arbre le nœud d'étiquette d, ou répondre qu'il n'y en a pas. On supposera pour simplifier que les étiquettes contenues dans un même ABR sont toutes différentes.

1. Écrivez une structure `ABR` en reprenant celle de `ArbreB`, mais cette fois-ci, les étiquettes doivent être de type `int`. *Remarque : dans la définition de la structure, on ne tient pas compte de la condition sur la répartition des étiquettes dans un ABR. Par contre, dans toutes les fonctions à venir, on supposera que les `ABR*` passés en argument ont bien cette propriété d'ABR, et on fera attention à ce que les `ABR*` renvoyés la respecte également.*
2. Écrivez une fonction `ABR_to_arbre(ABR* A)` qui crée un `ArbreB` obtenu à partir de l'`ABR A` en convertissant les étiquettes de `A` en `string`. *Note : on pourra utiliser la fonction `to_string` en ajoutant l'option de compilation -std=c++0x.*
3. Déduisez de la question précédente une fonction `affiche_ABR(ABR* A, string nom)` qui affiche l'`ABR* A` dans la console.
4. Écrivez une fonction `void insere(int clef, ABR* A)` qui modifie l'`ABR` pointé par `A` en ajoutant une feuille d'étiquette `clef`. Attention, la feuille doit être placée au bon endroit pour que `A` conserve la propriété d'être un ABR.
5. (*) Écrivez une fonction `ABR* ABR_from_tableau_trie(vector<int> &tab)` qui prend en entrée un tableau `tab` d'entiers triés par ordre croissant, et qui renvoie un pointeur sur un `ABR` nouvellement créé dont les étiquettes sont précisément les entiers apparaissant dans `tab`. *Indice : c'est le moment de relire la question 4 de l'exercice 2 sur `arbre_from_tableau`.*
6. Écrivez une fonction `cherche_noeud(int clef, ABR* A)` qui renvoie un pointeur (type `ABR*`) sur le nœud d'étiquette `clef` dans `A`, s'il existe (pointeur `NULL` sinon).
7. En reprenant une partie de la fonction `cherche_noeud`, écrivez une fonction `chemin_dans_ABR(int clef, ABR* A)` qui renvoie un `string` correspondant au chemin descendant de la racine au nœud d'étiquette `clef` dans `A`, avec la convention suivante : `0` pour fils gauche et `1` pour fils droit. Par exemple, le `string 00110` correspond au nœud `a5` que l'on atteint depuis la racine `r` en suivant chemin `a1-a2-a3-a4-a5` avec : `a1` est le fils gauche de `r`, `a2` est le fils gauche de `a1`, `a3` est le fils droit de `a2`, `a4` est le fils droit de `a3`, et `a5` est le fils gauche de `a4`.
   Si aucun nœud ne porte pour étiquette `clef`, on s'autorise à renvoyer n'importe quel `string`.
8. Écrivez une fonction `noeud_min(ABR* A)` qui renvoie un pointeur (type `ABR*`) vers le nœud d'étiquette minimale dans `A`.
9. (**) Écrivez `supprime(int clef, ABR* A)` qui modifie `A` en enlevant le nœud d'étiquette `clef` (on supposera pour simplifier qu'un tel nœud existe toujours lorsque la fonction est appelée). Pour supprimer le nœud z, on distinguera trois cas :
   (a) Soit le nœud z est une feuille, dans ce cas on le supprimer directement sans problème ;
   (b) Soit le nœud z a un seul fils f, dans ce cas on supprime z en raccrochant f à l'ancien père de z ;
   (c) Soit le nœud z a deux fils, dans ce cas on appelle y le nœud d'étiquette minimale dans le sous-arbre droit de z (observez qu'y n'a donc pas de fils gauche) ; puis on recopie l'étiquette de y à la place de l'étiquette de z ; et enfin on supprime y de manière assez simple (cf cas précédents) car y a au plus un fils.
   Note : il vous sera peut-être utile d'utiliser une fonction auxiliaire renvoyant un pointeur sur le père d'un nœud donné en paramètre.

Exercice 7 : Convertir un arbre ternaire en arbre binaire

1. En s'inspirant de la définition de `ArbreB`, définissez une structure de données `ArbreT` pour les arbres ternaires (ternaire signifie que chaque nœud peut avoir 0, 1, 2 ou 3 fils).
2. Écrivez une fonction `ArbreT_to_ArbreB(ArbreT* A)` qui renvoie un pointeur (type `ArbreB*`) vers un arbre binaire `B` obtenu en transformant l'arbre ternaire `A` en arbre binaire, selon l'algorithme vu en cours. *Petit rappel : un nœud d'étiquette a dans `A`, de premier fils b et de prochain frère c, correspond dans `B` à un nœud d'étiquette a ayant pour fils gauche b et pour fils droit c.*