INF3105 – Structures de données et algorithmes
Été 2022 – Examen de mi-session

Éric Beaudry
Département d'informatique
Université du Québec à Montréal

Mardi 28 juin 2022 – 13h30 à 16h30 (3 heures) – Local PK-1780

Instructions

1. Aucune documentation n'est permise, excepté l'aide-mémoire C++ (feuille recto verso).
2. Les appareils électroniques, incluant les téléphones et les calculatrices, sont strictement interdits.
3. Répondez directement sur le questionnaire à l'intérieur des endroits appropriés.
4. Pour les questions demandant l'écriture de code :
   — le fonctionnement correct, l'efficacité (temps et mémoire), la clarté, la simplicité du code et la robustesse sont des critères de correction à considérer ;
   — vous pouvez scinder votre solution en plusieurs fonctions ;
   — vous pouvez supposer l'existence de fonctions et de structures de données raisonnables.
5. **Aucune question ne sera répondue durant l'examen**. Si vous croyez qu'une erreur ou qu'une ambigüité s'est glissée dans le questionnaire, indiquez clairement la supposition que vous avez retenue pour répondre à la question.
6. Vous pouvez détachez les annexes à la fin du questionnaire. Évitez de détacher les autres feuilles.

Résultat

Q1		/ 20
Q2		/ 20
Q3		/ 20
Q4		/ 10
Q5		/ 10
Q6		/ 20
Total		/ 100

Solutionnaire

1 C++ [20 points]

(a) Qu'affiche le programme A (voir l'Annexe A à la page 9) ? [6 points]

```
A6 A1 A9 D69 C9 C1 C9
A3 A7 M A1 A2 M P C7 C3
```

(b) Ce programme contient une fuite de mémoire (*memory leak*). Combien d'octets sont laissés sur le tas (*heap*) à la fin du programme ? Supposez `sizeof(int)=4`. [4 points]

La ligne 39 alloue un bloc mémoire avec l'opérateur `new` pour un objet de type `M`. Ce bloc n'est jamais libéré avec l'opérateur `delete`. À la fin du programme, cet objet reste donc sur le tas (*heap*).
Un objet `M` est composé de deux objets `A`. Chaque objet `A` est composé d'un objet `int`. Donc : `sizeof(M)` $2 \times 1 \times 4 = 8$ octets.
Avec un ordinateur sous Linux, on peut vérifier avec l'outil Valgrind :

```
HEAP SUMMARY:
in use at exit: 8 bytes in 1 blocks
...
LEAK SUMMARY:
definitely lost: 8 bytes in 1 blocks.
```

(c) Comment faudrait-il corriger cette fuite de mémoire ? [4 points]

À la fin de la fonction `f2`, à insérer entre les lignes 39 et 40, il faut faire :

```
delete m2;
```

(d) Pourquoi retrouve-t-on «`return *this;`» à la fin d'opérateur = ? [3 points]

En cas d'utiliser le résultat de l'effectation.
Exemple1 :

```
a = b = c;  // Ici a sera affecté au résultat de b=c. Autrement dit, le compilateur l'interprète comme :
a.operateor=(b.operateor=(c));
```

Pour fonctionner, il faut que `b.operateor=(c)` retourne une référence sur lui-même, soit `*this`. Étant donné que les objects deviennent équivalents, retourner le paramètre pourrait une option. Mais, il est plus logique de retourner `*this`.
Exemple2 : `f3(a=b);  //` où la fonction `f3` prendrait en paramètre le résultat de `a=b`, soit `a`.

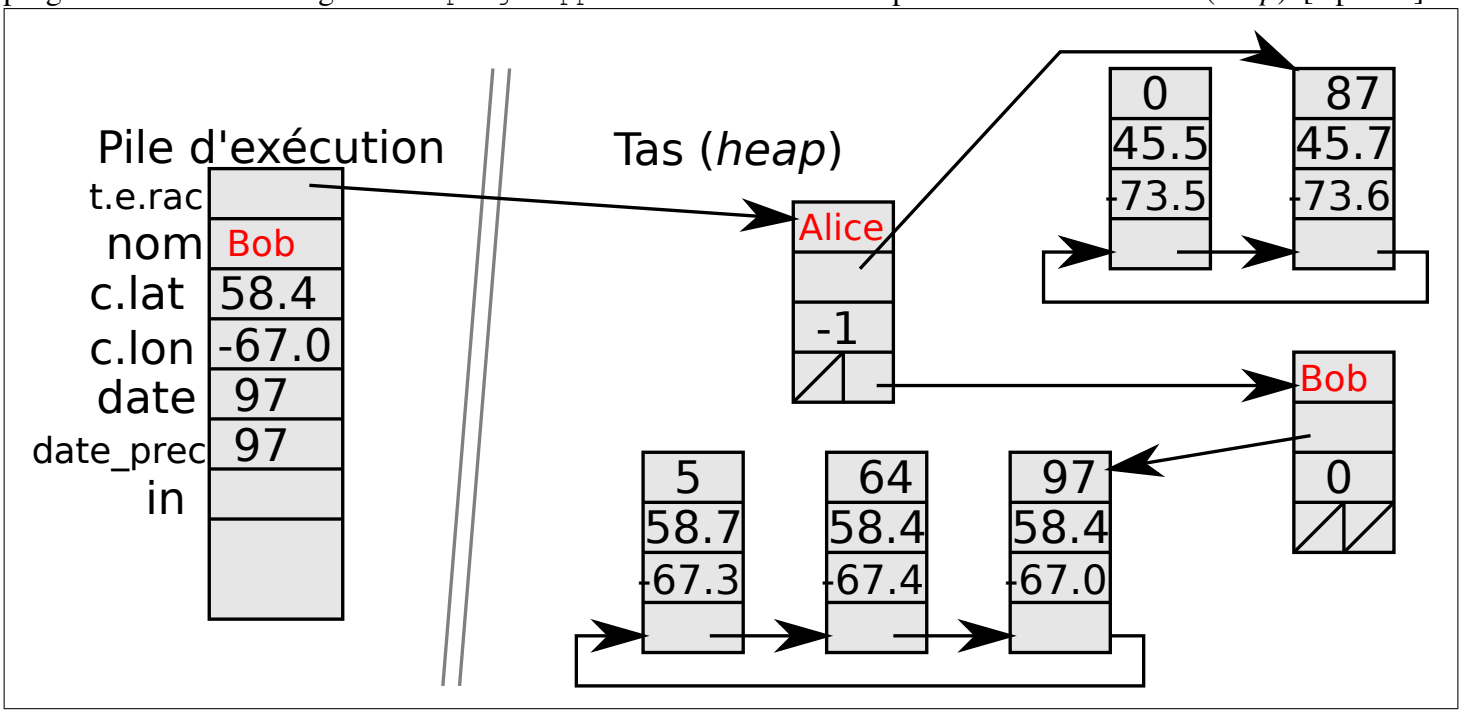
(e) L'opérateur `M::operator=(const M&)` est déclaré, **mais non défini**. Est-ce grave ? Commentez. [3 points]

Non. En C++, il est possible de déclarer l'existance de fonctions sans les définir. À la compilation, en particulier lors de la phase d'édition des liens, on vérifie que toutes les fonctions utilisées sont bien définie. Ici, l'opérateur `M::operator=(const M&)` n'étant jamais utilisé (appelé), il n'y a donc aucun problème.

2 Représentation, Analyse et Codage [20 points]

Référez-vous à l'Annexe B (page 10).

(a) On exécute la commande `./progB traces.txt < test.txt`. Dessinez la représentation en mémoire du programme rendue à la ligne 16 de `progB.cpp`. Montrez clairement la pile d'exécution et le tas (*heap*). [5 points]



(b) Quelle est la complexité temporelle du programme B ? Utilisez la notation grand O. Supposez k noms différents de personne et n traces (lignes) dans `traces.txt` et k requêtes dans `test.txt`. Détaillez. [5 points]

`progB.cpp` / lignes 5 à 16 :

Ces lignes concernent la lecture des traces. La boucle `while` (ligne 10) fera n itérations. À la ligne 13, l'opérateur`[]` dans `traces[nom]` fait une recherche dans le dictionnaire. Le dictionnaire pouvant contenir jusqu'à k noms différents, chaque recherche coûte $O(\log k)$. Toujours à la ligne 13, chaque appel à la fonction `Liste::ajouter` coûte $O(1)$. On obtient : $n \times (\log k + 1) = n \log k$.

Réponse : $O(n \log k)$.

`progB.cpp` / lignes 16 à 22 :

Puisqu'il y a k requêtes, la bouche `while` (ligne 17) fait k itérations. Comme précédemment, on peut établir que la partie «`traces[nom]`» de la ligne 20 coûte $O(\| \log \|)$.

La fonction `Trace::vitesseMoyenne` nécessite de calculer la distance totale d'une trace. Pour cela, il faut itérer sur toutes la liste `l`. Itérer sur une liste se fait en temps linéaire. On peut supposer que chaque liste a en moyenne n/k traces. Ainsi, pour les k itérations sur les k objets `Trace`, cela coûtera $O(n)$ en tout.

On obtient ainsi $O((k \times \log k) + n)$. Il est difficile d'établir si $k \log k >< n$. On peut établir que $k \leq n$, car dans le pire cas, chaque trace sera avec une personne différente.

Réponse à $O(k \log k + n)$

`progB.cpp` / total :

On garde le coût le plus grand ou la sommes des deux :

$O(n \log k)$ + $O(k \log k + n) = O(n \log k)$

Réponse : $O(n \log k)$

(c) Complétez la fonction suivante. [5 points]

```
1  double Trace::vitesseMoyenne() const {
2      double disttolpar = 0; // distance totale parcourrue
3      Liste<Obs>::iterateur it1 = l.debut();
4      double datedebut = (*it1).date; // ou =l[it1].date;
5      Liste<Obs>::iterateur it2 = it1 + 1; // ou it2=it1; ++it2;
6      while(i2){ // n itérations où n=taille de la liste l
7          disttolpar += (*it1).coor.distance((*it2).coor); // O(1)
8          it1 = it2; // O(1)
9          ++it2 // O(1)
10     }
11     double datefin = (*it1).date; // ou =l[it1].date;
12     double dureetotale = datefin - datedebut;
13     return disttolpar / dureetotale;
14
15     // Cout total: // O(n) où n=taille de la liste l
16 }
```

(d) Supposons qu'on désire lever l'hypothèse d'un ordre chronologique (`assert` ligne 10). Expliquez les modifications que vous feriez à `util.h` et/ou `util.cpp`. [5 points]

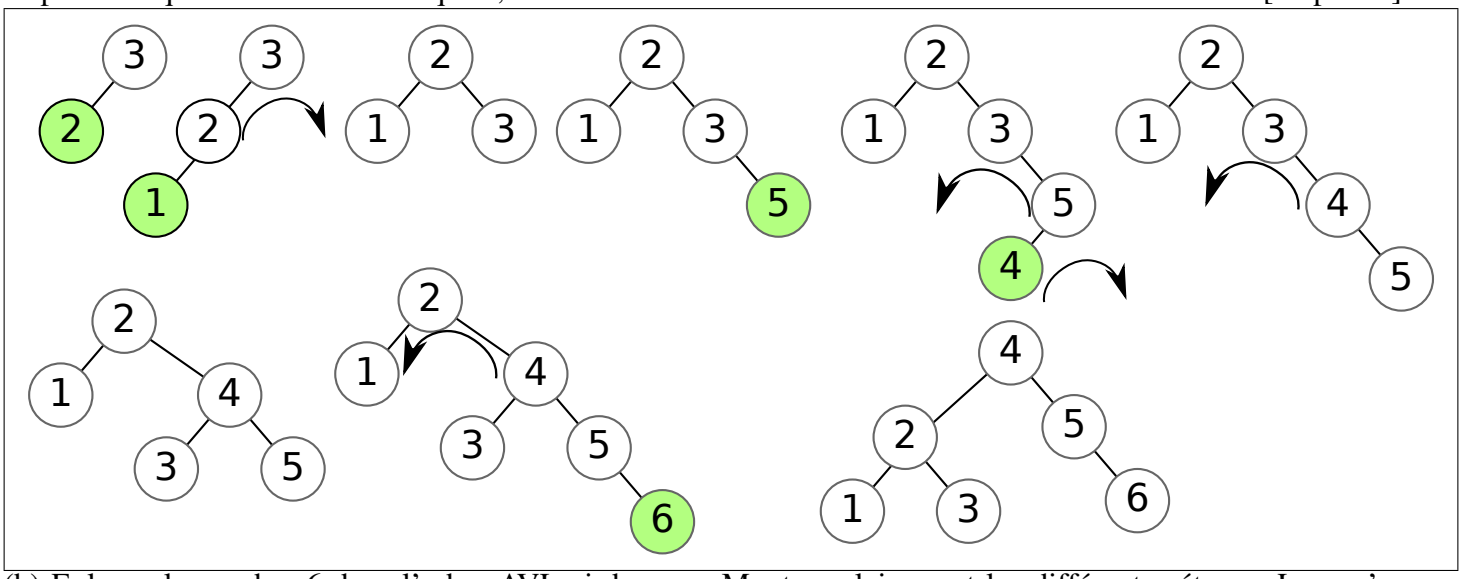
Il y a deux façons de faire.

1. La première façon consiste à utiliser un arbre binaire de recherche. Cela ressemble à l'exercice 2 sur les températures dans les diapositives de ArbreMap (08-arbresmap.pdf). Les clés sont les dates, les valeurs sont les coordonnées. On a plus besoin de la `struct Obs`.
On remplace la ligne 29 de `util`.h par : `ArbreMap<double, Coor> l;`
Dans `util.cpp`, la ligne 9 (fonction `ajouter`) devient : `l[d]=c;`
Il sera qu'à adapter les autres fonctions `vitesseMax` et `vitesseMoyenne`.

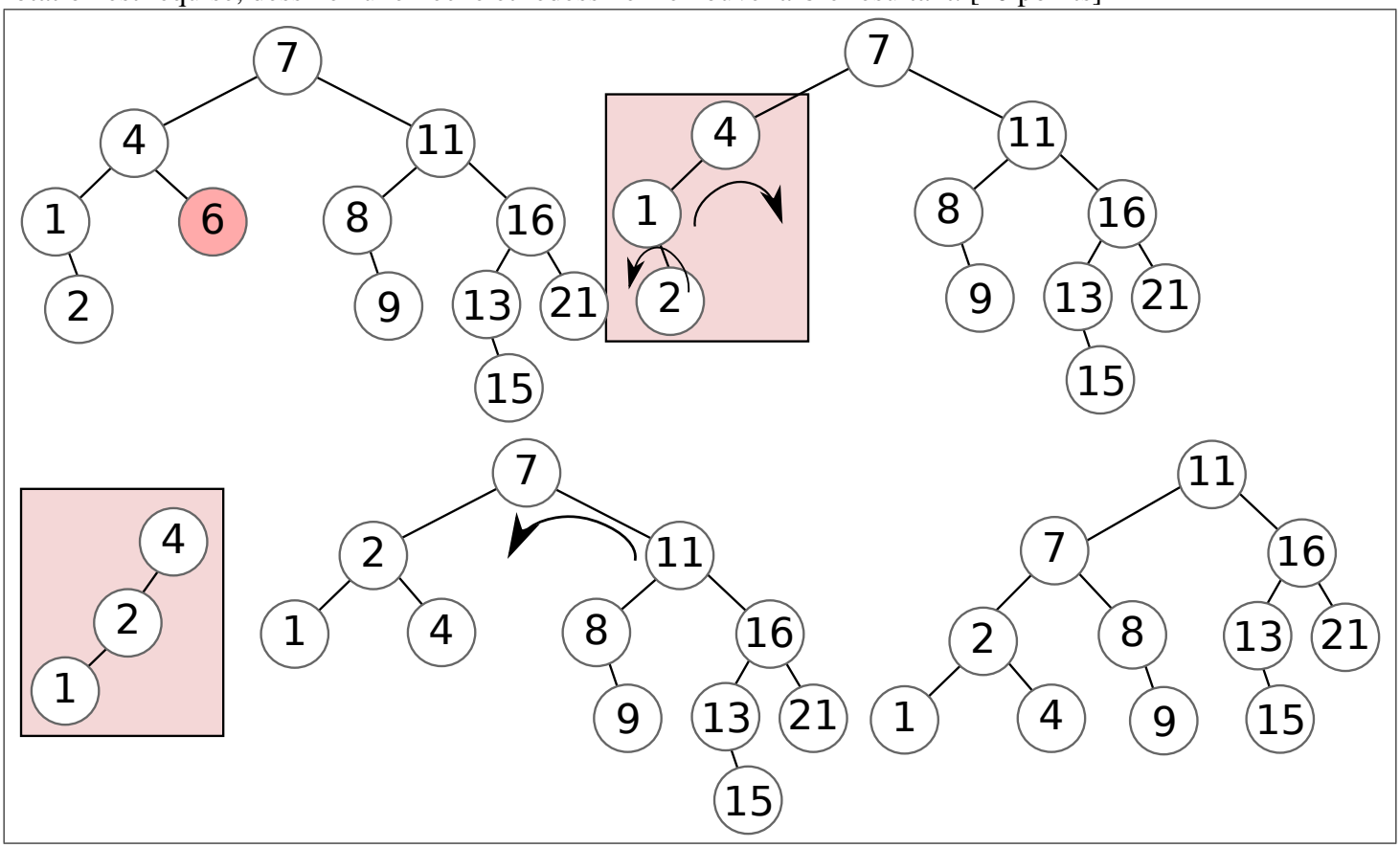
2. Si l'utilisation de la classe `Trace` est toujours dans un contexte ressemblant à `progB.cpp`, où la lecture de toutes les traces est entièrement faite avant les requêtes, on pourrait plutôt utiliser des tableaux. À la fin de la lecture, il suffirait de trier ces tableaux en ordre de date.

3 Arbres AVL [20 points]

(a) Insérez les nombres 3, 2, 1, 5, 4 et 6 dans un arbre AVL initialement vide. Montrez clairement les différentes étapes. Lorsqu'une rotation est requise, dessinez une flèche et redessinez le nouvel arbre résultant. [10 points]



(b) Enlevez le nombre 6 dans l'arbre AVL ci-dessous. Montrez clairement les différentes étapes. Lorsqu'une rotation est requise, dessinez une flèche et redessinez le nouvel arbre résultant. [10 points]

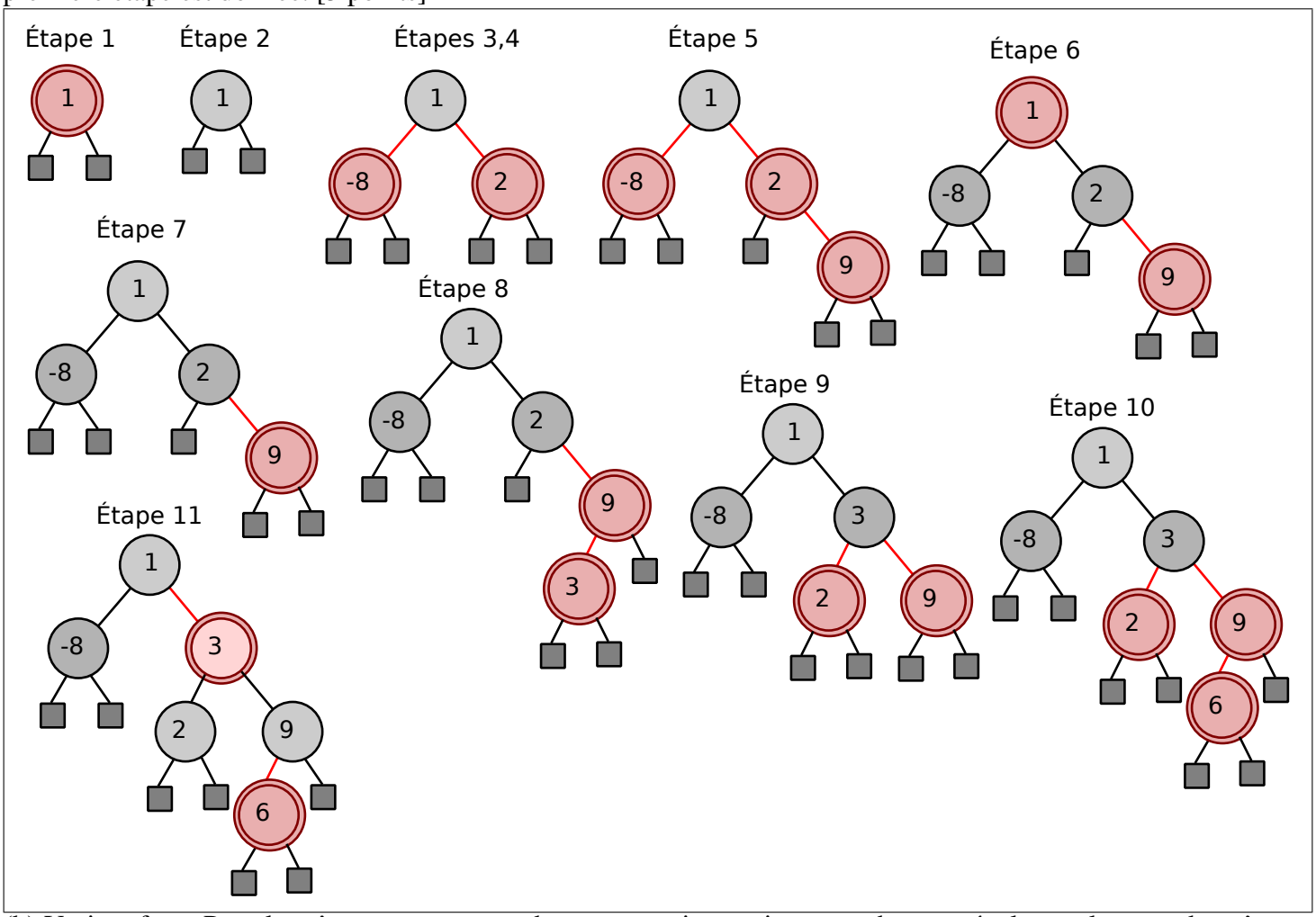


4 Arbres Rouge-Noir [10 points]

À titre de rappel, revoici les principales caractéristiques d'un arbre rouge-noir :

- la racine est noire ;
- le nœud parent d'un nœud rouge doit être noir (pas deux nœuds rouges de suite sur un chemin) ;
- une « sentinelle » est une feuille qui ne stocke aucun élément et est considérés un nœud noir ;
- toutes les sentinelles sont à une « profondeur noire » égale, la profondeur noire étant définie par le nombre de nœuds noirs sur le chemin (à ne pas confondre avec la hauteur de l'arbre qui elle ne tient pas compte des sentinelles).

(a) Insérez les nombres 1, -8, 2, 9, 3 et 6 dans un arbre rouge-noir initialement vide. Lorsqu'une réorganisation/-recoloration de nœud(s) est requise, redessinez l'arbre afin de bien montrer les étapes intermédiaires. Considérez les directives suivantes : (1) si vous n'avez pas de crayon rouge, dessinez un cercle simple pour un nœud noir et un cercle double pour un nœud rouge ; (2) dessinez de petits carrés pleins pour représenter les sentinelles. La première étape est donnée. [5 points]



(b) Vrai ou faux. Pour le même contenu, un arbre rouge-noir a toujours une hauteur égale ou plus grande qu'un arbre AVL. À noter que les sentinelles ne sont pas comptées pour la hauteur d'un arbre. Justifiez. [5 points]

Faux. Il est facile de trouver un conte-exemple.
[Mettre un contre-exemple ici]
Ce qui est vrai, c'est que le pire cas de hauteur d'un arbre RN est plus haut que le pire cas d'arbre AVL. Autrement dit, AVL permet un meilleur équilibre, car plus stricte.

5 Arbres binaires de recherche [10 points]

La classe `ArbreBinRech<T>` à l'Annexe C implémente un arbre binaire de recherche abstrait.
(a) Codez la fonction `maximum` qui retourne une référence sur l'élément **maximum** de l'arbre. Votre fonction doit être **non récursive**. (5 points)

```
1  template <class T>
2  const T& ArbreBinRech<T>::maximum const {
3     const Noeud* n = racine;
4     while(n->droite != nullptr)
5         n = n->droite;
6     return n->contenu;
7
8  // Cette fonction plantera si l'arbre est vide.
9  // Il est raisonnable de supposer qu'on n'appelera jamais maximum si l'arbre est vide.
10 }
```

(b) Écrivez la fonction `hauteur` qui retourne la hauteur de l'arbre. Notez que la classe `ArbreBinRech<T>::Noeud` n'a pas d'attribut `hauteur` ou `equilibre`. (5 points)

```
1  template <class T>
2  const int ArbreBinRech<T>::hauteur() const {
3     return hauteur(racine);
4  }
5  template <class T>
6  const int ArbreBinRech<T>::hauteur(const Noeud* n) const {
7     if(n==nullptr)
8         return 0;
9     int hg = hauteur(n->gauche);
10    int hd = hauteur(n->droite);
11    return 1 + max(hg, hd);
12 }
```

6 Résolution d'un problème – Présences aux laboratoires (20 points)

Lisez l'Annexe D.
Vous devez écrire un programme qui affiche pour chaque étudiant le nombre de laboratoires auxquels il a été présent. Pour qu'un étudiant soit considéré comme avoir été présent à un laboratoire, il doit avoir été connecté pendant au moins 90 minutes, c'est-à-dire que la commande `who` doit l'avoir identifié au moins 6 fois pour la même date. Vous pouvez écrire votre solution directement dans le squelette suivant ou au verso de cette feuille. Utilisez les structures vues dans le cours.

```
1  #include <iostream>
2  #include <sstream>
3  #include <string>
4  using namespace std;
5  int main(){
6    // Deux dictionnaires avec clé=code_MS
7    ArbreMap<string, int> nbLabs;//compteur de nb de labs présents
8    ArbreMap<string, int> nbFoisWho;//compteur du nb de fois identifiés par who
9    string dateprecedente; // pour detécter le changement de date
10   while(cin){
11     string ligne, date, heure;
12     getline(cin, ligne);
13     stringstream ss(ligne); // un stringstream est un flux C++ sur un objet string
14     ss >> date >> heure;
15
16     if(date!=dateprecedente){ // détection d'un changement de date
17         dateprecedente = date; // retenir la dernière date
18         nbFoisWho.vider();  // remettre les compteurs à zéro
19     }
20
21     while(ss){
22       string code_ms;
23       ss >> code_ms;
24       if(++nbFoisWho[code_ms]==6) // on incrémente le compteur..
25           ++nbLabs[code_ms];  // si atteint 6, nouvelle présence
26       nbLabs[code_ms]; //option: force ajout code_ms au cas aucune prés. lab
27
28     }
29
30
31
32   }
33   cout << "Code_MS\tNombre_labs" << endl;
34   for(ArbreMap<string, int>::Iterateur iter=nbLabs.debut();iter;++iter)
35     cout << iter.cle() << '\t' << iter.valeur() << endl;
36
37   return 0;
38 }
```

Annexe A

```
1  /************** Programme A *****************/
2  #include <iostream>
3  using namespace std;
4
5  class A{
6  public:
7    A(int i=1) : x(i) {cout << "A" << i << " ";}
8    A(const A& a) : x(a.x) {cout << "B" << x << " ";}
9    ~A() {cout << "C" << x << " ";}
10   A& operator=(const A& a){
11     cout << "D" << x << a.x << " ";
12     x = a.x;
13     return *this;
14   }
15 private:
16   int x;
17 };
18
19 class M{
20  public:
21    M(int i, int j) : a1(i), a2(j){
22      cout << "M "; }
23    M(const M& m) : a1(m.a1){
24      a2 = m.a2;
25      cout << "N "; }
26    ~M() {cout << "P ";}
27    M& operator=(const M&);
28  private:
29    A a1, a2;
30
31 };
32
33 void f1(){
34   A a1(6), a2, a3(9);
35   a1 = a3;
36 }
37 void f2(){
38   M m1(3, 7);
39   M* m2 = new M(1, 2);
40 }
41
42 int main(){
43   f1();
44   cout << endl;
45   f2();
46   cout << endl;
47 }
```

Annexe B

Le programme suivant permet de collecter des traces, un peu comme pour le TP2. Hypothèse : on suppose que les traces arrivent en ordre chronologique (ligne 10 progB.cpp).

```
1  /* util.h */
2  #include <iostream>
3  #include <string>
4  #include "liste.h"
5
6  using namespace std;
7
8  class Coor{ // Coordonnées
9  public:
10   double distance(const Coor&) const;
11   //...
12 private:
13   double lat, lon;
14 };
15
16 struct Obs{ // Observation
17   int  date; // en secondes
18   Coor coor;
19 };
20
21 class Trace{
22 public:
23   void ajouter(int date, Coor c);
24   const std::string& getNom const (){
25     return nom; }
26   double vitesseMax() const ();
27   double vitesseMoyenne() const ();
28 private:
29   Liste<Obs> l;
30   std::string nom;
31 };
```

```
1  /** util.cpp */
2  #include "util.h"
3
4  const std::string& Trace::getNom
     const (){
5     return nom;
6  }
7
8  void Trace::ajouter(int d, Coor c){
9    l.inserer_fin(Obs{d, coor});
10 }
11 // ...
```

```
1  /* progB.cpp */
2  #include <fstream>
3  #include "util.h"
4  int main(int argc, char** argv){
5    ArbreMap<string, Trace> traces;
6    string nom;
7    Coor c;
8    int date, date_prec=0;
9    ifstream in(argv[1]); //traces.txt
10   while(!in.eof()){
11     in >> nom >> date >> c;
12     assert(d>=date_prec);//hypothèse
13     traces[nom].ajouter(date, c);
14     date_prec = date;
15   }
16   /** Dessinez représentaiton **/
17   while(cin){
18     cin >> nom;
19     cout <<
20       traces[nom].vitesseMax()
21       << endl;
22   }
23   return 0;
24 }
```

Pour compiler :

```
g++ -o progB util.cpp progB.cpp
```

Exemple de fichier d'entrée `traces.txt` :

```
1 alice 0   (45.5,-73.5)
2 bob 5     (58.7,-67.3)
3 bob 64    (58.4,-67.4)
4 alice 87  (45.7,-73.6)
5 bob 97    (58.4,-67.0)
```

Exemple de fichier de requêtes `test.txt` :

```
1 alice
2 bob
```

Pour exécuter :

```
./progB traces.txt < test.txt
```

Annexe C

```
1 template <class T> class ArbreBinRech{
2     struct Noeud{
3         T contenu;  //Rappel : gauche->contenu < contenu < droite->contenu
4         Noeud *gauche, *droite;         };
5     Noeud* racine;
6   public: // ...
7     const T& maximum() const;
8     int hauteur() const;  };
```

Annexe D

Un professeur s'interroge à savoir si les étudiants qui se présentent aux laboratoires d'un cours obtiennent de meilleurs résultats. Au lieu de demander aux démonstrateurs de noter manuellement les présences, le professeur exécute automatiquement la commande `who` à toutes les 15 minutes sur les serveurs et stations durant les périodes de laboratoire. La commande `who` permet de récupérer la liste des usagers actuellement connectés. Les données sont collectées dans un seul fichier. Sur chaque ligne, on retrouve la date et l'heure à laquelle la commande `who` a été exécutée, ainsi que la liste des étudiants (codes MS) connectés à cet instant précis. Vous pouvez supposer que le fichier est ordonné chronologiquement. Voici un exemple de fichier.

```
1 20130501 17:30 ABCD01020304 WZYZ03030310 HFKQ05829704 ...
2 20130501 17:45 ABCD01020304 JDNW21128601 WZYZ03030310 HFKQ05829704 ...
3 20130501 18:00 ABCD01020304 JDNW21128601 HFKQ05829704 ...
4 ...
5 ...
6 20130508 17:30 WZYZ03030310 WZYZ03030310 JDNW21128601 ...
7 ...
8 ...
```

Vous devez écrire un programme qui affiche pour chaque étudiant le nombre de laboratoires auxquels il a été présent. Pour qu'un étudiant soit considéré comme avoir été présent à un laboratoire, il doit avoir été connecté pendant au moins 90 minutes, c'est-à-dire que la commande `who` doit l'avoir identifié au moins 6 fois pour la même date. Vous pouvez écrire votre solution directement dans le squelette suivant ou au verso de cette feuille. Utilisez les structures vues dans le cours.