INF3105 – Structures de données et algorithmes
Automne 2014 – Examen de mi-session

Éric Beaudry
Département d'informatique
Université du Québec à Montréal

Mardi 21 octobre 2014 – 13h30 à 16h30 (3 heures) – Locaux SB-R440 et A-2770

Instructions

- Aucune documentation n'est permise, excepté l'aide-mémoire C++ (feuille recto verso).
- Les appareils électroniques, incluant les téléphones et les calculatrices, sont strictement interdits.
- Répondez directement sur le questionnaire à l'intérieur des endroits appropriés.
- Pour les questions demandant l'écriture de code :
  - le fonctionnement correct, la robustesse, la clarté, l'efficacité (temps et mémoire) et la simplicité du code 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 ;
- **Aucune question ne sera répondue durant l'examen**. Si vous croyez qu'une erreur ou qu'une ambiguïté s'est glissée dans le questionnaire, indiquez clairement la supposition que vous avez retenue pour répondre à la question.
- L'examen dure 3 heures, contient 5 questions et vaut 20 % de la session.
- Ne détachez pas les feuilles du questionnaire, à moins de les brocher à nouveau avant la remise.
- Dans l'entête de l'actuelle page, si vous encerclez le numéro de local où vous vous trouvez présentement, vous aurez un point boni pour avoir lu les instructions.
- Le côté verso peut être utilisé comme brouillon. Des feuilles additionnelles peuvent être demandées au surveillant.

Identification

Nom : Solutionnaire

Résultat

Q1		/ 20
Q2		/ 20
Q3		/ 28
Q4		/ 12
Q5		/ 20
Total		/ 100

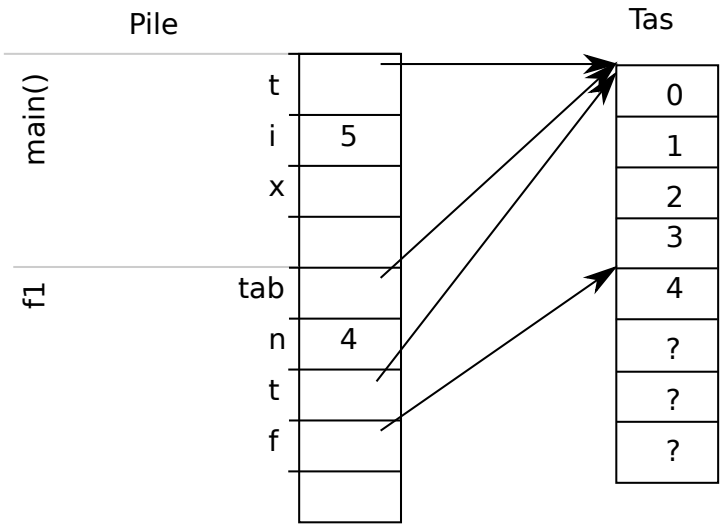
1 Connaissances techniques et C++ [20 points]

Pour répondre à cette question, référez-vous au code fourni à l'Annexe A (page 8).

(a) Que fait la ligne 10 du programme `question1a.cpp` (`int* t = new int[8];`) ? [3 points]

(1) Alloue automatiquement sur la pile d'exécution un pointeur t.
(2) Alloue dynamiquement sur le tas (*heap*) un bloc contigu de 8 entiers. Les 8 entiers sont non initialisés et conservent leur valeur courante.
(3) Enfin, on affecte à `t` l'adresse du début du bloc alloué.

(b) Dessinez la représentation en mémoire de ce programme après avoir exécuté la ligne 3 de la fonction `f1`. Montrez les objets des fonctions `main` et `f1` sur la pile (*stack*) et ceux sur le tas (*heap*). [5 points]



Remarque : les variables `i` et `x` pourraient ne pas être montrées, car elles sont à l'extérieur de la porté (*scope*).

(c) Qu'affiche le programme `question1a.cpp` ? [3 points]

1 3 5 7 5 ? ? ? : 16
Les ? dépendent de ce qui avait en mémoire avant. Si vous expérimentez, vous devriez obtenir trois zéros aux ?, car les systèmes d'exploitation modernent peuvent initialiser à zéro tout l'espace mémoire utilisateur.

(d) Ce programme libère-t-il la mémoire correctement ? Si oui, dites simplement oui. Si non, écrivez la modification minimale pour corriger ce problème. [3 points]

Non.
Il faut ajouter « `delete[] t;` » entre les lignes 19 et 20.

(e) Alice et Bob ont respectivement écrit deux programmes `q1e-alice.cpp` et `q1e-bob.cpp`. Ces 2 programmes **affichent**-ils le même résultat ? Expliquez brièvement deux différences entre ces deux programmes. [3 points]

Oui, ils **affichent** le même résultats et font pratiquement la même chose.
La principale différence est la façon d'allouer l'objet `Pile p`. Alice le fait par allocation dynamique sur le tas alors que Bob le fait par allocation automatique. Une seconde différence est au niveau du passage de paramètre de la fonction `foo`. Alice passe une adresse (le pointeur `p`) tandis que Bob fait un passage par copie.

(f) Alice et Bob sont convaincus d'avoir une meilleure solution que l'autre. Selon vous, qui a raison ? Justifiez. [3 points]

Bob a la meilleure solution. Premièrement, la solution de Bob est plus simple. L'objet `Pile p` est fait par allocation automatique sur la pile d'exécution. Les appels au constructeur (ligne 3) et au destructeur (à la fin de `main`) sont fait automatiquement. De plus, rien de justifie à Alice d'allouer des pile dynamiquement. L'allocation dynamique est généralement nécessaire pour allouer des objets de taille variable ou trop volumineux pour être stockés sur la pile d'exécution. Ici, ce n'est pas nécessaire. Un objet `Pile` ne contient qu'un pointeur `sommet` et a donc une taille fixe et très peu volumineux. La représentation interne de `Pile` (une chaine de cellules) est déjà basée sur de l'allocation automatique. Ici, il n'y a pas raison d'ajouter un 2e niveau d'indirection.

2 Complexité algorithmique [20 points]

(a) Simplifiez les ordres de grandeur suivants (en notation grand O). [8 points]

$O(2)$	$O(1)$	$O(4n\log n + 8m\log n)$	$O((n+m)\log n)$
$O(2n \times n/8)$	$O(n^2)$	$O((4n+3)\times(12n-7))$	$O(n^2)$
$O(2^n + n^3 + 9n^2)$	$O(2^n)$	$O(4n + 8n\log(7n) + 75)$	$O(n\log n)$
$O((42n^2+n+8)(n+8))$	$O(n^3)$	$O(n! + 8n^3 + n^2)$	$O(n!)$

Pour les sous-questions suivantes, référez-vous au code fourni à l'Annexe B (page 9).

(b) Que fait le programme à l'Annexe B (`q2.cpp`) ? [4 points]

1. Le programme lit un nuage de points.
2. Le programme compte le nombre de triangles équilatéraux (3 côtés de la même longueur).
3. Le programme affiche le nombre de triangles équilatéraux.

(c) Quelle est la complexité temporelle de la fonction `q2` notation grand O ? Justifiez brièvement. [4 points]

$O(n^3)$. La fonction `q2` a trois boucles `for` faisant respectivement tout près de n itérations, où n est la taille du tableau `t`, soit le nombre de nombres lus. Donc $O(n^3)$.

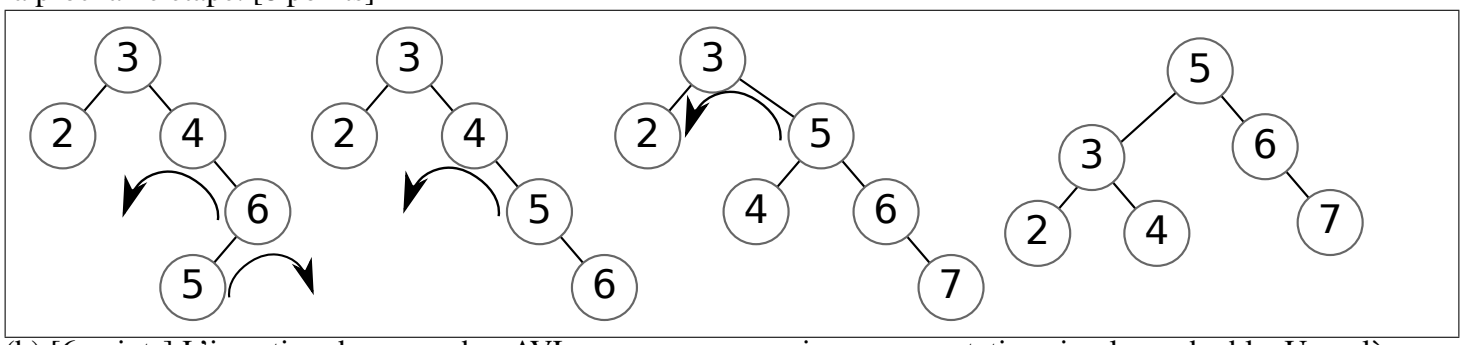
(d) Il est possible d'améliorer la fonction `q2` pour réduire sa complexité temporelle. Expliquez (en français) comment améliorer la fonction `q2` **OU** codez (C++ ou pseudo-code) une version améliorée. Écrivez aussi la complexité temporelle de cette nouvelle version. [4 points]

```
1  struct DistanceIndex{
2    DistanceIndex(double d=0, int i=0):distance(d),index(i){}
3    double dist;   int index;
4    bool operator <(const DistanceIndex& o) const{return dist<o.dist;}
5  };
6  int q2(const Tableau<Point>& t){
7    int nb=0;
8    for(int i=0;i<t.taille();i++){ // O(n) iterations
9      Tableau<DistanceIndex> distances;
10     for(int j=i+1;j<t.taille();j++) // O(n) iterations
11       distances.ajouter(DistanceIndex(t[i].distance(t[j]),i));
12     trier(distances); // O(n log n)
13     for(int j=0;j<distances.taille();j++) // O(n) iterations
14       for(int k=j+1;k<distances.taille()&&distances[k].dist==distances[j].dist;k++)
15         if(t[distances[j].index].distance(t[distances[k].index])==distances[j].dist)
16           nb++;
17   }
18   return nb;
19 }
```

Explications : (1) garder une boucle `for i` pour choisir le premier point (ligne 8) ; (2) calculer la distance entre le point i et tous les autres points (lignes 10-11) ; (3) trier ces distances (ligne 12) ; (4) parcourir les distances triées pour choisir un deuxième point j (ligne 13) ; (5) itérer sur les points consécutifs à j et de distance égale (ligne 14) pour détecter les triangles isocèles. (5) lorsqu'on trouve un triangle isocèle, il reste à tester s'il est aussi équilatéral (ligne 15) et le compter si oui (ligne 16).
Complexité : $O(n^2 \log n)$ (à condition d'avoir un nombre limité de triangles isocèles)

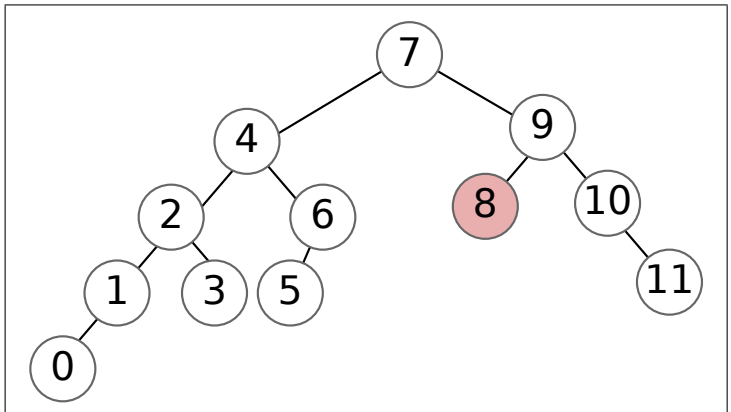
3 Arbres AVL [28 points]

(a) Insérez les nombres 3, 4, 2, 6, 5 et 7 dans un arbre AVL initialement vide. À chaque insertion, ajoutez un nouveau nœud à l'arbre courant. Lorsqu'une rotation est requise : (1) dessinez une flèche pour montrer la rotation requise ; (2) dessinez un nouvel arbre pour montrer le résultat ; (3) le nouvel arbre devient l'arbre courant pour la prochaine étape. [8 points]

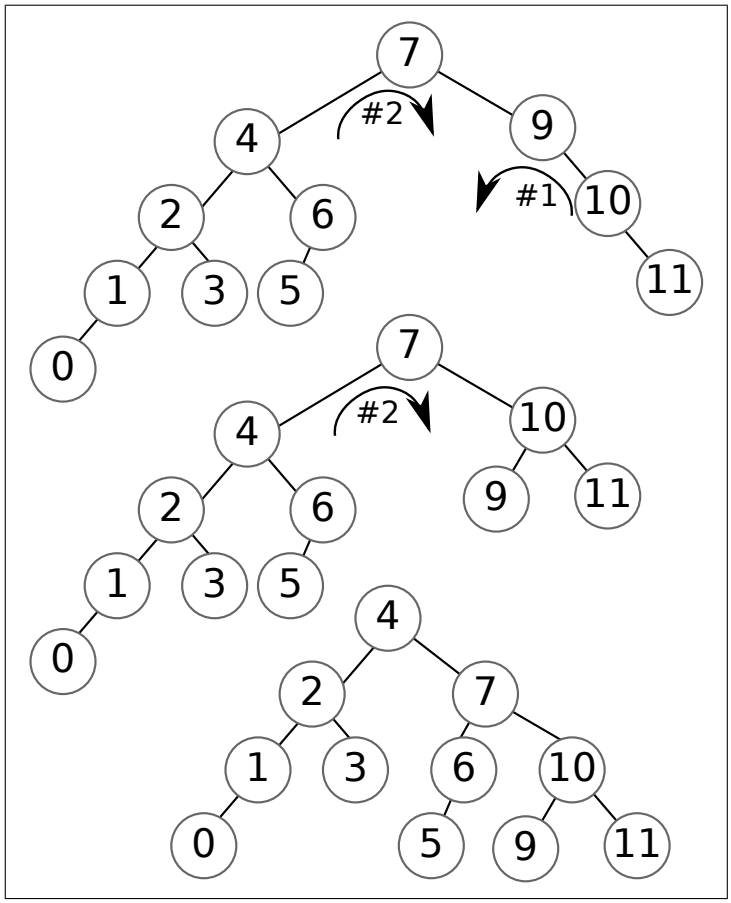


(b) [6 points] L'insertion dans un arbre AVL provoque au maximum une rotation simple ou double. Un enlèvement peut provoquer une cascade de rotations simples ou doubles jusqu'à la racine. La présente sous-question vise à montrer un tel exemple. Étapes :

1. Dans le rectangle à gauche, dessinez un arbre AVL équilibré qui nécessite au moins deux rotations (simples et/ou doubles) suite à l'enlèvement d'un élément précis. Mettez des nombres dans les nœuds.
2. Indiquez clairement quel élément provoque une cascade de rotations lors de son enlèvement.
3. Montrez le résultat final de l'enlèvement dans le rectangle à droite. Indiquez clairement les rotations.



L'enlèvement de 8 provoque une cascade de rotations.



(c) **Sans utiliser d'itérateurs** d'arbre AVL, codez une fonction qui retourne une liste chainée en **désordre** (du plus grand au plus petit) les éléments contenus dans un arbre. [4 points]

```
1  template <class T> Liste<T> ArbreAVL<T>::convertirEnListeInv() const{
2      Liste<T> resultat;
3      convertirEnListeInv(racine, resultat);
4      return resultat;
5  }
6  template <class T>  // parcours en inordre
7  void ArbreAVL<T>::convertirEnListeInv(Noeud* n, Liste<T>& resultat) const{
8      if(n==NULL)
9          return;
10     convertirEnListeInv(n->gauche, resultat);
11     resultat.inserer_debut(n->contenu);
12     convertirEnListeInv(n->droite, resultat);
13 }
```

(d) On insère les nombres 1 à 100 dans un Arbre AVL dans un ordre aléatoire. Le nombre 25 peut-il être à la racine ? Répondez simplement par oui ou non. [2 points]

Oui.

(e) La présente question est liée à la sous-question précédente (d). Quels nombres peuvent se retrouver à la racine d'un arbre AVL contenant les nombres 1 à 100 ? Justifiez brièvement en utilisant l'espace alloué. [4 points]

Réponse : Les nombres 21 à 80 inclusivement.
Justification : Pour trouver la plus petite racine possible, il faut déterminer le nombre minimal de nœuds dans le sous-arbre de gauche. Pour cela, on a besoin du nombre de nœuds minimal en fonction de la hauteur d'un arbre AVL. Un arbre AVL d'hauteur h, le nombre minimal de noeuds est de : $nbmin(h) = nbmin(h-1) + nbmin(h-2) + 1$. Il faut aussi vérifier si le sous-arbre de droite peut contenir tous les nœuds restants. Pour cela, on a besoin du nombre maximal de noeuds en fonction de la hauteur : $nbmax(h) = 2^h - 1$ On construit ainsi ce tableau.

h	0	1	2	3	4	5	**6**	**7**	8	9	10
nb min	0	1	2	4	7	12	**20**	33	54	88	143
nb max	0	1	3	7	15	31	63	**127**	255	511	1023

La plus petite hauteur possible d'un arbre de 100 nœuds est de 7. Donc, le plus petit sous-arbre de gauche a une hauteur de 6 et contient au minimum 20 nœuds. **Ainsi, la plus petite racine possible serait le nombre 21.** Il reste à savoir si on peut mettre les nœuds 22 à 100 (79 nœuds) dans le sous arbre de droite. Le sous-arbre de droite peut avoir une hauteur de 6 ou 7. C'est possible de stocker 79 noeuds à partir d'une hauteur de 7. L'arbre AVL a donc finalement une hauteur de 8.
Pour trouver le plus grand nombre possible, on peut faire un raisonnement similaire en considérant 20 nœuds à droite (nombre 81 à 100). **Donc le plus grand nombre possible à la racine est 80**.

(f) Combien d'octets sont nécessaires pour stocker en mémoire un arbre AVL dans lequel on a inséré les nombres 1 à 100 ? Supposez `sizeof(int)=4` et `sizeof(X*)=4`. Supposez que l'arbre AVL est implémenté tel que présenté en classe et dans les notes de cours. Justifiez votre réponse. [4 points]

La classe `ArbreAVL::noeud` contient un contenu (4 octets), deux pointeurs `gauche` et `droite` (4 octets chacun), et un indice d'équilibre (4 octets). Donc : $4+2*4+4 = 16$ octets par nœuds. Multiplier par 100 nœuds, ça fait 1600 octets.
À cela on peut ajouter le pointeur racine qui nécessite 4 octets. Donc, 1604 octets en tout.

4 Arbres rouge-noir [12 points]

(a) Proposez une **représentation** pour implémenter un arbre rouge-noir. Ajoutez le code directement dans le squelette ci-droit. Votre représentation doit être suffisante pour implémenter toutes les opérations usuelles d'une arbre binaire de recherche, soit la recherche, l'insertion et l'enlèvement. Un rappel des principales caractéristiques d'un arbre rouge-noir est disponible à l'Annexe C (page 9). [4 points]

```
1  template <class T> class ArbreRN {
2    struct Noeud{
3      T contenu;
4      Noeud* gauche;
5      Noeud* droite;
6      bool rouge; //=vrai ssi rouge
7    };
8    Noeud* racine;
9    ...
10 public:
11   ArbreRN();
12   ...
13   int fonctionB() const;
14 };
```

(b) Complétez la solution de la fonction `fonctionB` qui retourne le nombre maximal de nœuds pouvant être insérés dans l'arbre sans provoquer une réorganisation ou un recoloriage de nœuds existants. [4 points]

Intuition : La procédure d'insertion dans un arbre R-N commence par insérer un nouveau noeud rouge. Ensuite on vérifie si des contraintes ont été violées, et les résoudre le cas échéant. Pour causer aucune réorganisation et aucun recoloriage de nœuds existants, **il faut insérer sous un noeud noir**. Ainsi, l'astuce consiste à compter le nombre de pointeur NULL (sentinelle) sous les nœuds noirs. Cela s'implémente bien avec une fonction récursive.

```
1  template <class T> int ArbreRN<T>::fonctionB() const{
2      return fonctionB(racine);
3  }
4  template <class T> int ArbreRN<T>::fonctionB(const Noeud* n) const{
5      if(n==NULL) return racine==NULL ? 1 : 0; // cas special si arbre vide==>1
6      int nb=fonctionB(n->gauche) + fonctionB(n->droite);
7      if(!n->rouge){
8          if(n->gauche==NULL) nb++;
9          if(n->droite==NULL) nb++;
10     }
11     return nb;
12 }
```

(c) Quel est le nombre minimal de nœuds dans un arbre rouge-noir de hauteur 8 ? Justifiez votre réponse. Vous pouvez donner le nombre exact [4 points] ou une approximation [3 points].

Approximation : $2^{8/2} = 16$ nœuds. Justification : On sait que la hauteur maximal d'un arbre rouge-noir est de $h = 2\log_2 n$. Autrement dit, un arbre rouge-noir contient au moins $n = 2^{h/2}$ nœuds.
Nombre exact : 30 nœuds. Justification : Le plus long chemin racine-feuille a 8 nœuds et alterne forcément noir et rouge. La «profondeur noire» est donc de $\lceil 8/2 \rceil = 4$. Chacun de ces nœuds sur ce chemin doit avoir un sous-arbre. Pour minimiser le nombre de nœuds dans ces sous-arbres, ils doivent avoir aucun rouge. Ils sont donc des arbres remplis de nœuds noirs.
La racine de l'arbre et son enfant rouge auront donc deux sous-arbres plein d'une hauteur de 3. Ensuite, les 3e et 4e nœuds sur le chemin le plus long auront deux sous-arbres plein d'une hauteur de 2. Ensuite, les 5e et 6e nœuds sur le chemin le plus long auront deux sous-arbres plein d'une hauteur de 1. Ensuite, les 7e et 8e nœuds sur le chemin le plus long auront deux sous-arbres plein d'une hauteur de 0.
Total : $8+2\cdot 7+2\cdot 3+2\cdot 1+2\cdot 0 = 30$

5 Résolution d'un problème – Analyse des résultats (20 points)

On souhaite vérifier si les étudiants obtiennent des notes comparables dans les cours INF3105 et INF4230. Vous trouverez à l'Annexe D (page 10) deux fichiers de notes `INF3105.txt` et `INF4230.txt` en exemple. Chaque ligne contient le code permanent d'un étudiant et sa note obtenue. Écrivez un programme qui produit la matrice d'analyse décrite à l'Annexe D. Les étudiants n'ayant suivi qu'un seul des deux cours doivent être ignorés. Votre solution doit utiliser au moins un dictionnaire basé sur un arbres binaires de recherche (`ArbreMap` ou `map`).

```
1  #include <fstream>
2  #include <iostream>
3  #include <string>
4  #include <map>    // ou "arbremap.h"
5  #include <vector> // ou "tableau.h"
6  using namespace std;
7
8  void lireNotes(map<string, vector<char> >& notes, const char* fichier){
9    ifstream is(fichier);
10   while(is){
11     string codeperm;
12     char note='?';
13     is >> codeperm >> note;
14     notes[codeperm].push_back(note); // .ajouter(note)
15   }
16 }
17
18 int main(){
19   map<string, vector<char> > notes; // codepermanent --> notes obtenues
20   lireNotes(notes, "INF3105.txt");
21   lireNotes(notes, "INF4230.txt");
22   map<char, map<char, int> > matrice;
23   for(map<string, vector<char> >::iterator i=notes.begin();i!=notes.end();++i)
24     if((i->second).size()==2) // si exactement 2 notes
25       matrice[(i->second)[0]][(i->second)[1]]++;
26   for(char note3105='A';note3105<='E';note3105++){
27     for(char note4230='A';note4230<='E';note4230++)
28       cout << '\t' << matrice[note3105][note4230];
29     cout << endl;
30   }
31   return 0;
32 }
```

Annexe A pour la Question 1

Cette page peut être détachée. À noter que le code a été allégé (ex. : ~~#include~~) pour rentrer sur une page.

```cpp
/* question1a.cpp */
void f1(int* tab, int n){
  int* t = tab, *f=tab+n;
  while(t<f) *(t++) *= 2;
}
void f2(int& a, int& b){
  a += b++;
}
int main(){
  int* t = new int[8];
  for(int i=0;i<5;i++)
    t[i]=i;
  f1(t, 4);
  int x = 0;
  for(int i=0;i<5;i++)
    f2(x, t[i]);
  for(int i=0;i<8;i++)
    cout <<" " << t[i];
  cout<<" : "<<x<<endl;
  return 0;
}
```

```cpp
/* pile.h */
template <class T>
class Pile //Tout est correctement implemente
{
public:
  Pile();
  Pile(const Pile&);
  ~Pile();
  void empiler(const T&);
  T depiler();
  bool vide() const;
  const Pile<T>& operator=(const Pile<T>&);

private:
  struct Cellule{
    Cellule(const T& e, Cellule* c);
    T contenu;
    Cellule* suivante;
  };
  Cellule* sommet;
};
```

```cpp
/* q1e-alice.cpp */
int foo(Pile<int>* pile){
  Pile<int>* c = new Pile<int>(*pile);
  int s = 0;
  while(!c->vide())
    s += c->depiler();
  delete c;
  return s;
}
int main(){
  Pile<int>* p = new Pile<int>();
  while(cin){
    int i=0;
    cin >> i;
    p->empiler(i);
  }
  cout << foo(p) << endl;
  /* ... */
  delete p;
  return 0;
}
```

```cpp
/* q1e-bob.cpp */
int foo(Pile<int> pile){

  int s = 0;
  while(!pile.vide())
    s += pile.depiler();

  return s;
}
int main(){
  Pile<int> p;
  while(cin){
    int i=0;
    cin >> i;
    p.empiler(i);
  }
  cout << foo(p) << endl;
  /* ... */

  return 0;
}
```

Annexe B pour la Question 2

Cette page peut être détachée.

```
1  /** q2.cpp pour la Question 2 **/
2  #include <iostream>
3  #include <tableau.h>
4  #include <point.h>
5  using namespace std;
6
7  int q2(const Tableau<Point>& t){
8    int nb=0;
9    for(int i=0;i<t.taille();i++)
10     for(int j=i+1;j<t.taille();j++)
11       for(int k=j+1;k<t.taille();k++){
12         double d1, d2, d3;
13         d1 = t[i].distance(t[j]);
14         d2 = t[i].distance(t[k]);
15         d3 = t[j].distance(t[k]);
16         if(d1==d2 && d2==d3)
17           nb++;
18       }
19   return nb;
20 }
21 int main(){
22   Tableau<Point> nuage;
23   while(cin){
24     Point p;
25     cin >> p >> std::ws;
26     nuage.ajouter(p);
27   }
28   cout << q2(nuage) << endl;
29   return 0;
30 }
```

Annexe C pour la Question 4

Principales caractéristiques d'un arbre rouge noir.

- la racine est noire ;
- le nœud parent d'un nœud rouge doit être noir (on ne peut pas avoir deux nœuds rouges de suite) ;
- les feuilles sont appelées « sentinelles », ne stockent aucun élément et sont des nœuds noirs ;
- 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).

Annexe D pour la Question 5

Fichier INF3105.txt :

```
1  IIII11050520    A
2  JJJJ11563388    A
3  MMMM26506769    A
4  CCCC04104447    B
5  GGGG10613710    B
6  AAAA02024750    B
7  KKKK15597411    B
8  LLLL14543156    B
9  VVVV22001013    B
10 BBBB18599738    C
11 DDDD10550203    C
12 EEEE10057851    C
13 FFFF04031148    C
14 HHHH01555089    C
15 NNNN13568130    C
16 WWWW25596464    C
```

Fichier INF4230.txt :

```
1  IIII11050520    A
2  MMMM26506769    A
3  GGGG10613710    A
4  KKKK15597411    A
5  LLLL14543156    A
6  NNNN13568130    A
7  AAAA02024750    B
8  JJJJ11563388    B
9  BBBB18599738    B
10 DDDD10550203    B
11 HHHH01555089    B
12 XXXX06016863    B
13 YYYY09539502    B
14 ZZZZ06022620    B
15 CCCC04104447    D
16 EEEE10057851    D
17 FFFF04031148    D
```

À noter que les données dans cette page sont purement aléatoires et ne reflètent pas la réalité !

Matrice à produire en sortie (en format texte simplifié) :

	A	B	C	D	E
A	2	1	0	0	0
B	3	1	0	1	0
C	1	3	0	2	0
D	0	0	0	0	0
E	0	0	0	0	0

Dans la matrice ci-dessus, les lignes et les colonnes correspondent respectivement aux notes obtenues en INF3105 et INF4230. Le nombre dans une case ligne l et colonne c indiquent le nombre d'étudiants qui a obtenu la note l en INF3105 et la note c en INF4230.

Exemple de code pour lire un ficher de notes et afficher une matrice.

```
1  void lireNotes(){
2      ifstream f1("INF3105.txt");
3      while(f1){
4          string code;
5          char note='?';
6          f1 >> code >> note;
7      }
8  }
9  void afficherMatrice(){
10    for(char note3105='A';note3105<='E';note3105++){
11      for(char note4230='A';note4230<='E';note4230++)
12        std::cout << '\t' << 1;
13      std::cout << std::endl;
14    }
15 }
```