INF3105 – Structures de données et algorithmes
Été 2024 – Examen final

Jaël Champagne Gareau
Département d'informatique
Université du Québec à Montréal

Mardi 6 août 2024 – 13h30 à 16h30 (3 heures) – Local PK-1705

Instructions

1. Aucune documentation n'est permise, excepté l'aide-mémoire C++ disponible à la dernière page.
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.
7. Dans l'entête de l'actuelle page, si vous encerclez le numéro de local où a lieu le cours, vous aurez un point boni pour avoir lu les instructions.
8. Vous devez répondre à l'aide d'un crayon non rouge.

Identification

Nom : _____

Code permanent : _____

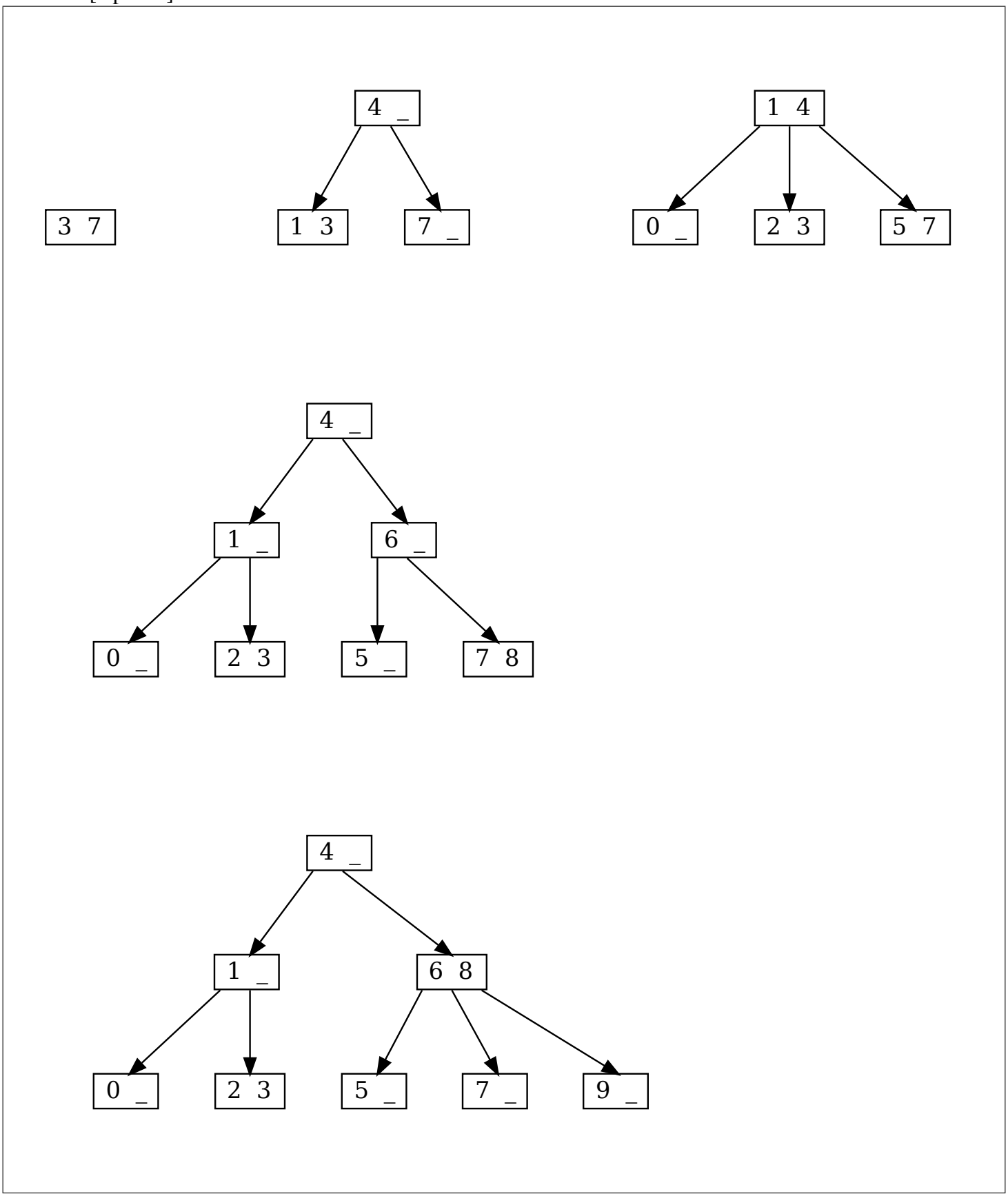
Signature : _____

Résultat

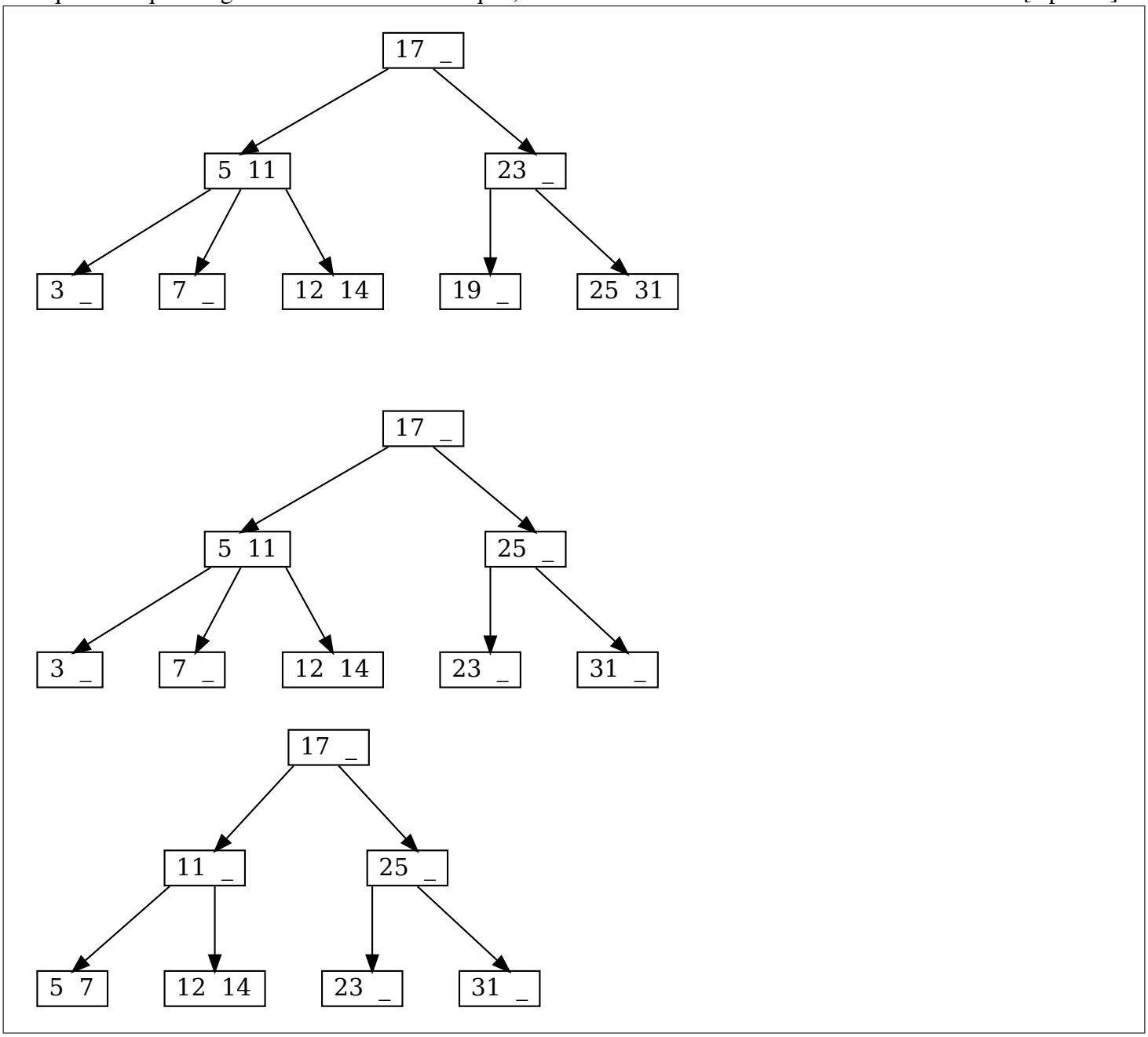
Q1		/ 20
Q2		/ 20
Q3		/ 10
Q4		/ 15
Q5		/ 15
Q6		/ 20
Total		/ 100

1 Arbres B [20 points]

(a) Insérez les nombres 3, 7, 4, 1, 0, 5, 2, 6, 8, 9 (dans cet ordre) dans un arbre B d'ordre 3 (un nœud a au plus trois enfants) initialement vide. Lorsqu'un rééquilibrage est requis, dessinez une flèche et redessinez l'arbre résultant. [8 points]



(b) Enlevez les nombres 2, 19, 3 (dans cet ordre) de l'arbre B d'ordre 3 ci-dessous.
Lorsqu'un rééquilibrage/fusion de nœuds est requis, dessinez une flèche et redessinez l'arbre résultant. [8 points]



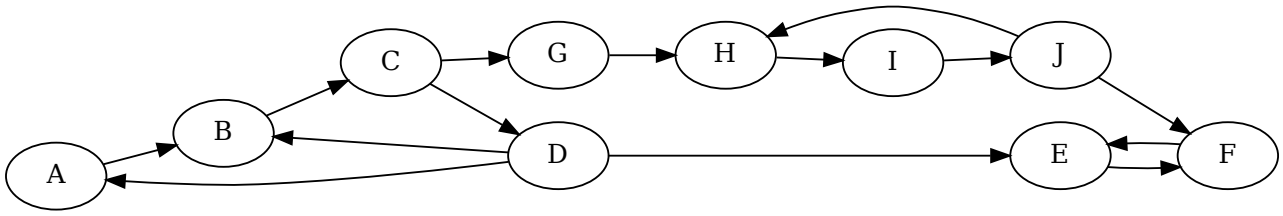
(c) Dans un arbre B d'ordre 9, on insère 500 éléments dans un ordre quelconque.
Quel est la hauteur minimale et maximale de l'arbre résultant ?
Rappel : dans le cours, nous utilisons la définition selon laquelle la hauteur d'un arbre vide est -1.

Hauteur minimale [2 points] : $h_{\min} = 2$

Hauteur maximale [2 points] : $h_{\max} = 4$

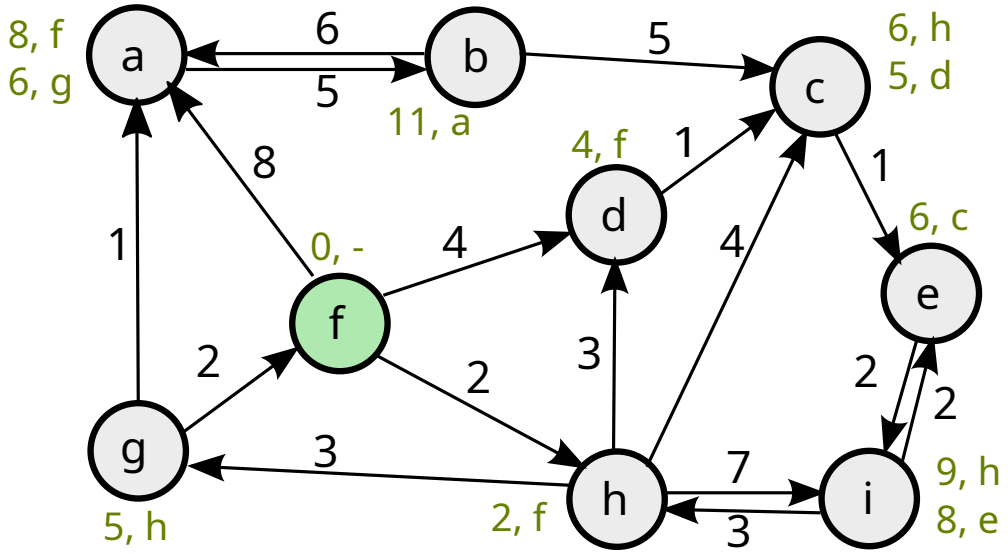
2 Graphes [20 points]

(a) Soit le graphe suivant. Complétez les quatres questions suivantes. Lors d'un parcours, lorsqu'un sommet a plusieurs voisins, explorez ceux-ci en ordre alphabétique. [8 points]



(i) parcours profondeur préordre (partant de D) : D, A, B, C, G, H, I, J, F, E
(ii) parcours profondeur postordre (partant de A) : F, E, D, J, I, H, G, C, B, A
(iii) parcours largeur (partant de C) : C, D, G, A, B, E, H, F, I, J
(iv) composantes fortement connexes : {A, B, C, D}, {E, F}, {G}, {H, I, J}

(b) Simulez l'algorithme de Dijkstra en partant du nœud f. À côté de chaque nœud du graphe, écrivez sa distance à f et son parent. Écrivez aussi l'ordre dans lequel les nœuds sont extraits de la file prioritaire. Lorsque deux sommets ont la même priorité, extrayez celui qui a été inséré dans le monceau en premier. [8 points]



Ordre de traitement des sommets : f, h, d, g, c, a, e, i, b

(c) Expliquez deux avantages d'utiliser l'algorithme Floyd-Warshall par rapport à faire n fois un des autres algorithmes de calcul de plus courts chemins vus au cours. [4 points]

Rapidité : L'algorithme de Floyd-Warshall calcule tous les plus courts chemins en $\mathcal{O}(n^3)$, alors que faire n fois l'algorithme de Dijkstra prend $\mathcal{O}(nm \log n)$ (avec un monceau binaire) ou $\mathcal{O}(n^2 \log n + nm)$ (avec un monceau de Fibonacci). Si le graphe est dense, la stratégie avec Dijkstra prend donc $\mathcal{O}(n^3 \log n)$ ou $\mathcal{O}(n^3)$. Bien qu'avec un monceau de Fibonacci, la complexité soit la même qu'avec Floyd-Warshall, en pratique Floyd-Warshall est plus rapide car la mémoire est accédée de manière plus locale et car la constante cachée derrière les opérations en $\mathcal{O}(1)$ du monceau de Fibonacci est grande.
Robustesse : Contrairement à l'algorithme de Dijkstra, Floyd-Warshall fonctionne avec des poids négatifs (on peut faire fonctionner l'algorithme de Dijkstra avec des poids négatifs, mais la complexité temporelle devient alors exponentielle).
Extraction du chemin : L'algorithme de Floyd-Warshall permet d'obtenir le premier sommet à visiter sur un plus court chemin en une seule opération ($\mathcal{O}(1)$), alors que Dijkstra nécessite de parcourir le chemin en remontant les parents, ce qui prend $\mathcal{O}(k)$, où k est la longueur du chemin.
D'autres réponses étaient aussi acceptées.

3 Codage [10 points]

(a) Vous devez compléter la fonction `parcoursProfondeurIteratif` de la classe `Graphe` à l'annexe A, qui effectue un parcours en profondeur et affiche les nœuds visités en préordre. Contrairement à la fonction vue en classe, qui était récursive, votre fonction doit être itérative. Vous pouvez utiliser toutes les structures de données vues en classe, si nécessaire, et supposer que les fichiers d'entête nécessaires sont déjà inclus. [8 points]

```
1  #include <stack>
2  using namespace std;
3
4  template<class S, class A>
5  void Graphe<S, A>::parcoursProfondeurIteratif(const S& sommetDepart) {
6      const int idSommetDepart = indices[sommetDepart];
7      stack<int> pile;
8      pile.push(idSommetDepart);
9      while (!pile.empty()) {
10         const int idActuel = pile.top();
11         const Sommet& sommetActuel = sommets[idActuel];
12         pile.pop();
13         if (sommetActuel.visite)
14             continue;
15
16         sommetActuel.visite = true;
17         cout << sommetActuel.s << " ";
18         for (const auto& [idVoisin, coutArc] : sommetActuel.arcsSortants)
19             pile.push(idVoisin);
20     }
21 }
```

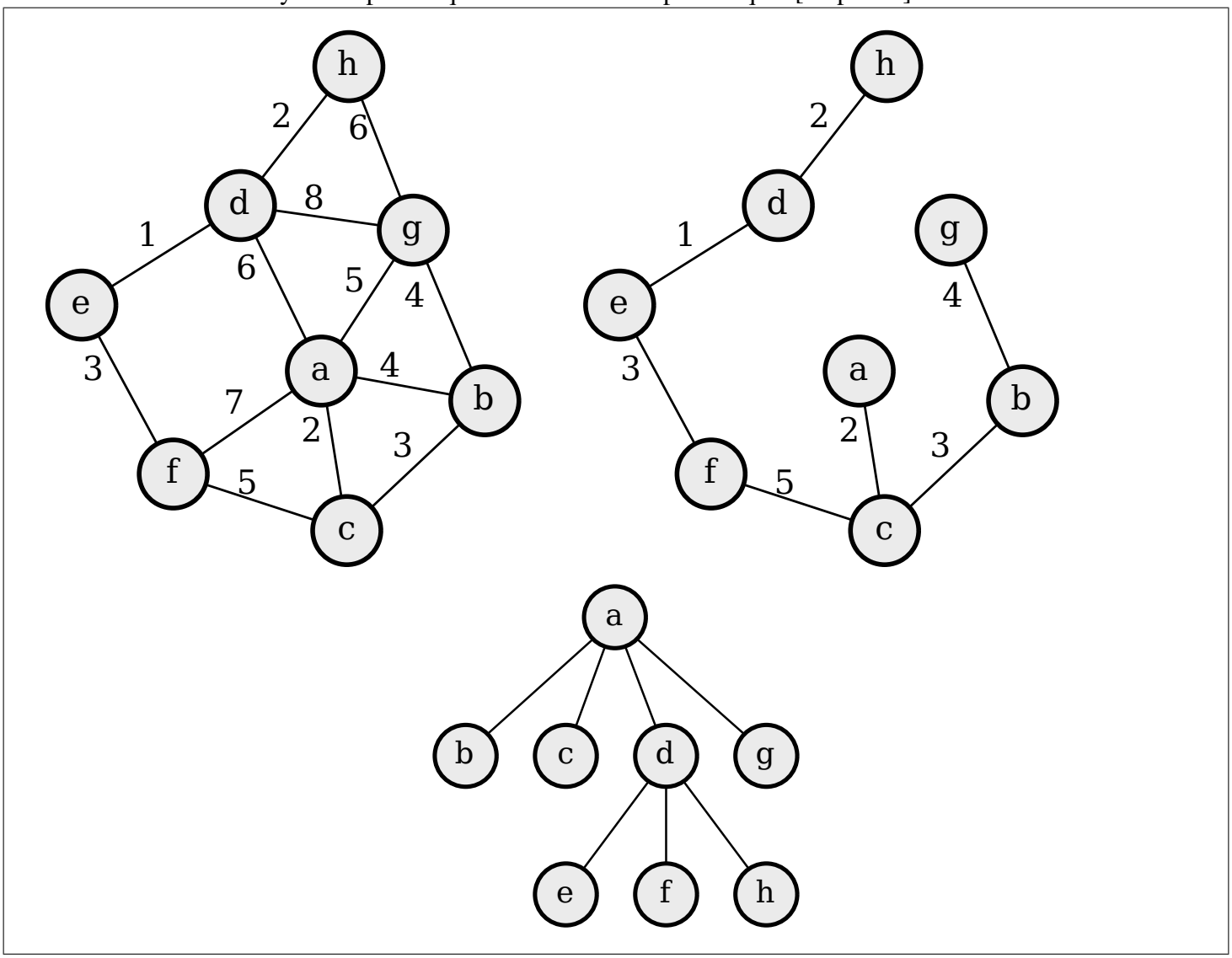
(b) Quelle est la complexité temporelle de votre fonction `parcoursProfondeurIteratif` ? Note : considérez bien votre implémentation et la représentation de graphe en annexe. La complexité n'est pas nécessairement la même que celle du parcours en profondeur vue en classe. [2 points] : $\mathcal{O}(n+m)$

4 Arbres de recouvrement minimal [15 points]

(a) Expliquez de la manière la plus détaillée possible le fonctionnement de l'algorithme Prim-Jarnik. Entre-autres, expliquez les structures de données nécessaires et comment elles sont utilisées. De plus, décrivez la complexité asymptotique de l'algorithme. [5 points]

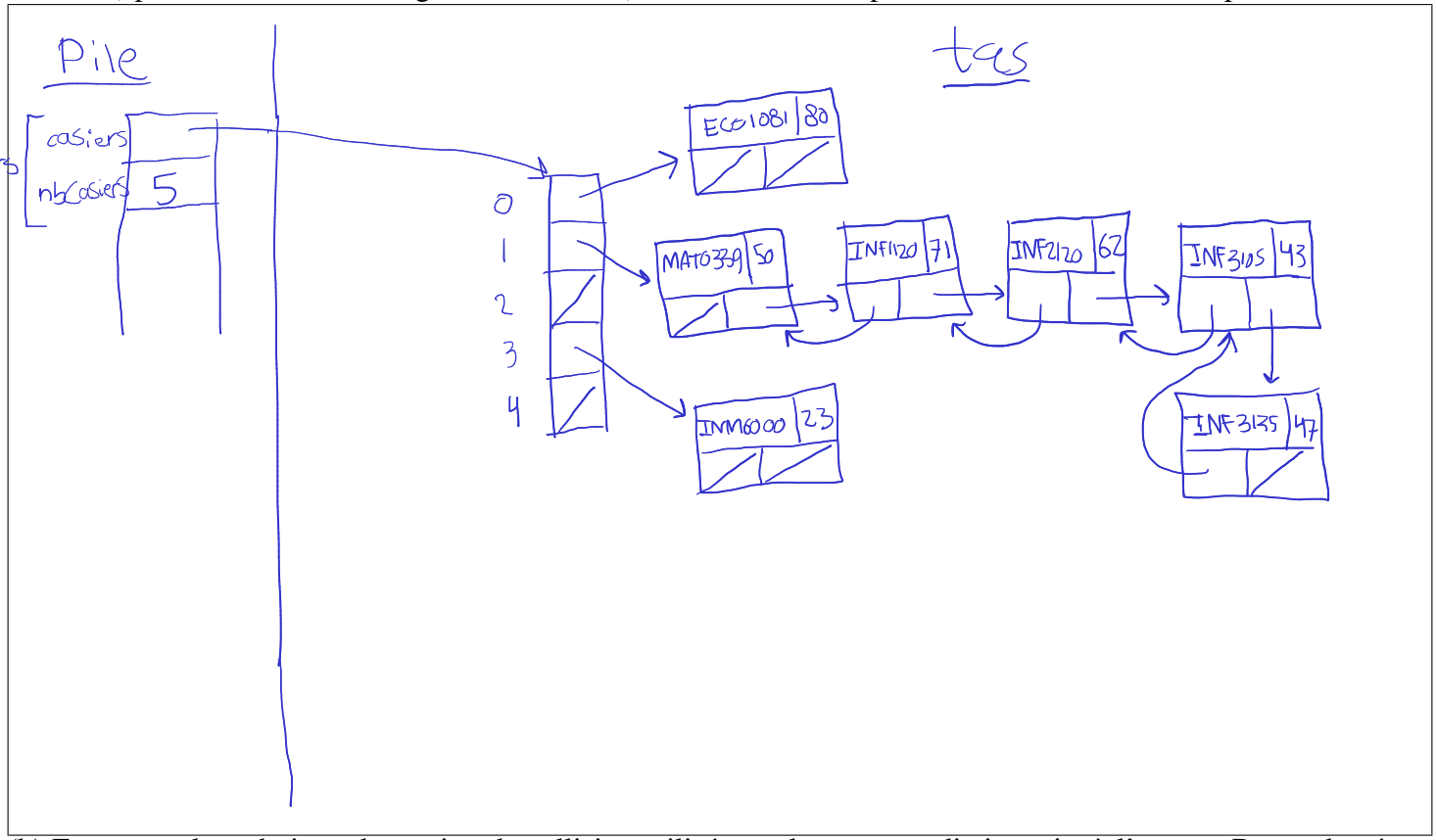
Prim-Jarnik est un algorithme qui permet de trouver un arbre de recouvrement minimal d'un graphe connexe non-orienté. L'algorithme utilise une file prioritaire qui contient les sommets voisins de ceux déjà ajoutés à l'arbre de recouvrement, mais qui ne sont pas encore eux-mêmes dans l'arbre de recouvrement. Deux tableaux sont utilisés : (1) pour stocker la longueur de l'arête la moins coûteuse ayant été découverte reliant chaque sommet à l'arbre de recouvrement, et (2) l'arête correspondante. Tout comme l'algorithme de Dijkstra, Prim-Jarnik extrait le sommet ayant la plus petite longueur de la file prioritaire, met à jour les longueurs des sommets voisins, et ajoute l'arête correspondante à l'arbre de recouvrement. La complexité temporelle est $\mathcal{O}(m \log n)$, où n est le nombre de sommets et m le nombre d'arêtes.

(b) Simulez l'algorithme de Kruskal sur le graphe suivant. Indiquez sur le graphe les arêtes sélectionnées. Dessinez également l'état final de la structure Union-Find (ensemble disjoint) après exécution de Kruskal. Supposez que l'implémentation du Union-Find utilise l'optimisation de taille (lors de l'union, l'arbre le plus petit se connecte sur l'arbre le plus grand) mais n'utilise pas la compression de chemins. Supposez également que le tri des arêtes met les arêtes ayant un poids équivalent en ordre alphabétique. [10 points]



5 Tables de hachage [15 points]

(a) Faites un dessin-mémoire du programme `main.cpp` à l'annexe B, juste au moment avant que le programme termine (après avoir exécuté la ligne 25 du fichier). Identifiez bien la pile d'exécution et le tas. [5 points]



(b) Est-ce que la technique de gestion de collision utilisée par la structure dictionnaire à l'annexe B est adaptée au programme `main.cpp` ? Dans un cas comme dans l'autre, justifiez votre réponse. [4 points]

Non, la stratégie utilisée (gestion de collisions par structure externe à l'aide de listes chaînées) est inefficace pour le programme `main.cpp`. En effet, le programme insère des noms de cours, qui commencent pour la plupart par INF, et la fonction de hachage n'utilise que les trois premiers caractères. Ainsi, la plupart des éléments tombent dans la même case, et la complexité temporelle des opérations dans le dictionnaire peuvent dégénérer en temps $\mathcal{O}(n)$.

(c) Nommez une autre technique de gestion de collision qui permettrait d'avoir une meilleure complexité temporelle que celle utilisée actuellement par le dictionnaire. Si l'autre technique nécessite des hypothèses/contraintes supplémentaires, mentionnez les. Justifiez votre choix. [2 points]

Une meilleure stratégie de gestion de collision qui peut être utilisée dans le cas actuel est la gestion par structure externe à l'aide d'arbre binaire de recherche (ex. : arbre AVL ou arbre rouge-noir). Cette stratégie permet à la table de hachage d'avoir une complexité temporelle dans le pire cas de $\mathcal{O}(\log n)$ par accès. Cette stratégie nécessite comme contrainte supplémentaire d'avoir un opérateur de comparaison (`operator<`) dans le type stockée dans le dictionnaire. Puisque dans notre cas les objets stockées sont des chaines de caractères, cette contrainte est respectée.

(d) En supposant que le programme insère les n cours offerts par le département d'informatique dans le dictionnaire, quelle sera la complexité du programme en fonction de n avec : [4 points]

(i) La technique de gestion de collision initiale ?

$\mathcal{O}(n^2)$

(ii) La technique proposée en (c) ?

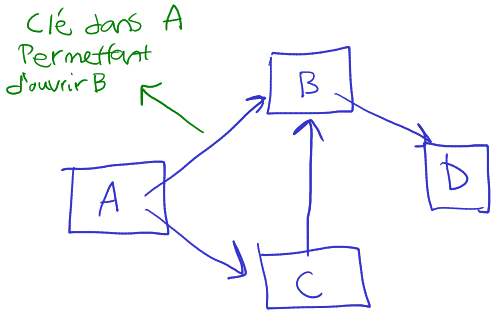
$\mathcal{O}(n \log n)$

6 Résolution de problèmes [20 points]

Vous avez une collection de n boîtes, et m clés. Chaque clé permet de déverouiller une boîte. Chaque boîte peut être déverouillée par zéro, une ou plusieurs clés. Il y a deux façons de déverouiller une boîte : (1) en utilisant une clé ; (2) en utilisant un marteau pour briser la serrure. Un collègue malintentionné a répartis les clés dans les boîtes et a verrouillé chacune d'elles. Certaines boîtes peuvent être vides, et d'autres peuvent contenir plusieurs clés. Grâce à votre système de sécurité qui a enregistré votre collègue, vous savez exactement quelles clés sont dans quelles boîtes. Vous voulez récupérer toutes vos clés, mais en utilisant votre marteau le moins possible (pour éviter de briser les serrures inutilement : elles sont en or !).

(a) Expliquez comment modéliser ce problème à l'aide d'un graphe. N'hésitez pas à illustrer visuellement votre explication à l'aide d'un exemple. Les marteaux ne sont pas considérés pour cette sous-question. [6 points]

On peut modéliser le problème par un graphe orienté où chaque sommet représente une boîte, et chaque arc représente une clé. Un arc partant d'une boîte u et allant vers une boîte v modélise le fait que la boîte u contient une clé qui permet d'ouvrir la serrure de la boîte v.



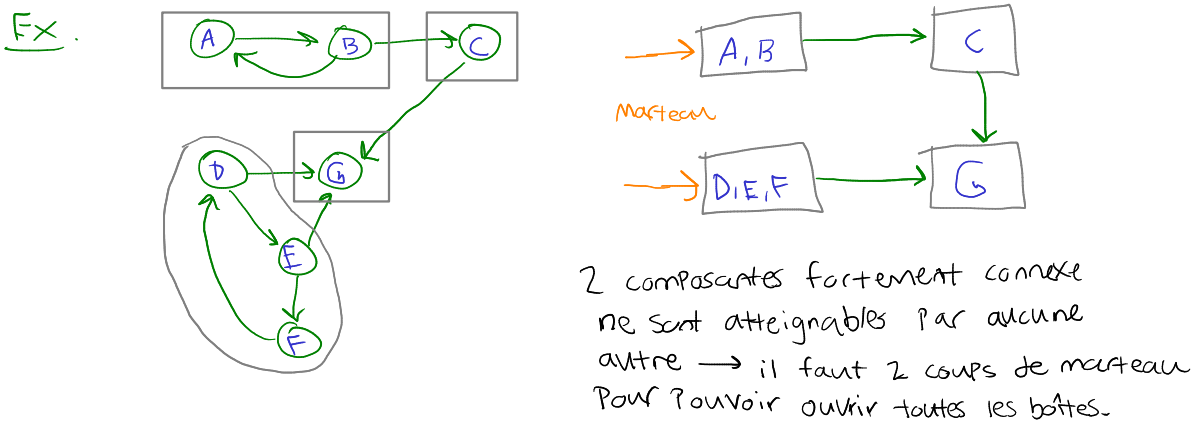
(b) Comme toutes les clés sont dans des boîtes verrouillées, vous devez briser au moins une serrure. Étant donné une boîte sur laquelle vous utilisez initialement votre marteau, décrivez un algorithme qui détermine si vous pouvez récupérer toutes les clés sans briser d'autres serrures. Vous pouvez décrire en phrases l'algorithme que vous utiliseriez pour résoudre ce problème, en autant que la description soit assez clair pour qu'une personne puisse implémenter votre stratégie. Décrivez la complexité temporelle de votre solution. [6 points]

On peut effectuer une recherche en largeur ou en profondeur à partir de la boîte sur laquelle on utilise initialement le marteau. Si, suite à cette recherche, tous les sommets du graphe sont visités, cela veut dire qu'il a été possible d'ouvrir toutes les boîtes (et donc, de récupérer toutes les clés) sans avoir à briser d'autres serrures. La complexité temporelle de cette solution est $\mathcal{O}(n+m)$.

(c) Supposons maintenant que, contrairement à la sous-question précédente, on ne fixe pas une boîte sur laquelle on utilise initialement le marteau. De plus, on permet d'utiliser le marteau k fois. Décrivez un algorithme qui permet de déterminer si vous pouvez récupérer toutes les clés en utilisant au plus k coups de marteau. Décrivez la complexité temporelle de votre nouvelle solution. [8 points]

Stratégie simple, mais inefficace : On peut ajouter un nouveau sommet au graphe, et ajouter k arcs partant de ce sommet vers d'autres sommets. Ensuite, on peut effectuer une recherche en largeur ou en profondeur à partir de ce sommet et vérifier que tous les sommets sont visités (comme dans la sous-question précédente). Cependant, il faut faire cela pour chaque combinaison de k sommets sur lesquels on utilise le marteau. Puisque le nombre de combinaisons possibles est $\frac{n!}{k!(n-k)!}$, la complexité temporelle de cette solution est $\mathcal{O}(\frac{n!}{k!(n-k)!}(n+m)) \approx \mathcal{O}(n!(n+m))$.

Stratégie efficace : On peut utiliser l'algorithme de Tarjan pour trouver les composantes fortement connexes du graphe. Ensuite, on peut vérifier combien de ces composantes fortement connexes n'ont aucune autre composante fortement connexe qui les atteint. Si le nombre de ces composantes fortement connexes est inférieur ou égal à k, alors on peut récupérer toutes les clés en utilisant au plus k coups de marteau. La stratégie a une complexité temporelle de $\mathcal{O}(n+m)$.



Annexe A : Représentation de graphe (Question 3)

```
1  template <class S, class A>
2  class Graphe {
3      public:
4          struct Sommet {
5              Sommet(const S& s_) : s(s_), visite(false) {}
6              // ...
7
8              S s;
9              map<int, A> arcsSortants;
10             mutable bool visite;
11         };
12
13         // ...
14         void parcoursProfondeurIteratif(const S& s) const; // Q3
15         // ...
16     private:
17         map<S, int> indices;
18         vector<Sommet> sommets;
19 };
```

Annexe B : Représentation de dictionnaire (Question 5)

Fichier `dictionnaire.h` :

```
1  #include <list>
2  template<class K, class V>
3  class Dictionnaire {
4   public:
5    Dictionnaire(int capacite) : nbCasiers(capacite) {
6      casiers = new std::list<Casier>[nbCasiers];
7    }
8    ~Dictionnaire() { delete[] casiers; }
9    V& operator[](const K&);
10   const V& operator[](const K&) const;
11
12  private:
13   struct Casier {
14     Casier();
15     K cle;
16     V valeur;
17   };
18
19   // Gestion de collisions par structure externe
20   std::list<Casier>* casiers;
21   int nbCasiers;
22 };
```

Fichier `main.cpp` :

```cpp
#include <string>
#include "dictionnaire.h"

class NomCours {
    public:
        NomCours(const std::string& nom_) : nom(nom_) {}
        bool operator==(const NomCours& autre) const {
            return nom == autre.nom;
        }
        int hash() const {
            return (nom[0] - 'A') + (nom[1] - 'A') + (nom[2] - 'A');
        }
    private:
        std::string nom;
};

int main() {
    Dictionnaire<NomCours, int> cours(5);
    cours[NomCours("MAT0339")] = 50;
    cours[NomCours("INF1120")] = 71;
    cours[NomCours("INF2120")] = 62;
    cours[NomCours("ECO1081")] = 80;
    cours[NomCours("INF3105")] = 43;
    cours[NomCours("INF3135")] = 47;
    cours[NomCours("INM6000")] = 23;
}
```

/**** Aide-mémoire sur C++ pour les examens d'INF3105 ****/

1 Types, déclarations, initialisations et opérateurs

Type (taille octets) : `bool` (1), `char` (1), `short` (2), `int` (4), `long` (8), `float` (4), `double` (8), `void*` (8).
Opérateurs arithmétiques : `*`, `/`, `+`, `-`.
Opérateurs logiques : `&&`, `||` (ou).
Opérateurs d'affectation sans/avec opération : `=`, `+=`, `++`, `-=`, `-`, `*=`, `/=`, `,|=`, `,&=`, etc.

```
int a;    // a est non initialisé
int b();  // b est initialisé à 0
int c=2;  // c est initialisé à 2
int d(3); // d est initialisé à 3
a = b++;  // post-incrementation  (a=0, b=1)
a = ++c;  // pre-incrementation   (a=3, c=3)
```

2 Entrées et sorties

```
#include <iostream>
using namespace std; /* std:: est le namespace standard */
int main(){
    int a, b;
    cin >> a >> b; // std::cin >> a >> b;
    std::cout << "La somme est : " << (a + b) << std::endl;
}
```

3 Pointeurs, déférencement, arithmétique des pointeurs et références

- Un pointeur est un objet (une variable) qui contient une adresse mémoire (un entier) de X bits (selon plateforme).
- On peut manipuler un pointeur de façon similaire à un entier (incrémenter, additionner, etc.).
- Le déférencement d'un pointeur doit être explicite à l'aide de l'opérateur de déférencement `*`.
- Une référence est un objet (une variable) qui contient une adresse mémoire, mais se manipule comme un objet.
- Le déférencement d'une référence est implicite. On ne peut manipuler l'adresse mémoire contenu dans une référence.

```
int x; // déclare un entier x
int* p; // déclare un pointeur d'entier p
int* q = &x; // q prend pour valeur l'adresse mémoire de x
p = q; // copie l'adresse mémoire contenu dans q vers p
int y = *q; // accède au contenu pointé par q, c.-à-d. x.
int tab[4]; // tab est un pointeur sur le premier élément d'un bloc de 4 entiers
p = tab + 2; // p = (l'adresse tab) + (2*sizeof(int)). Idem p=&(tab[2]).
p++; // p pointe sur l'entier suivant, c'est-à-dire tab[3]
cout << tab[2]; // équivaut : cout << *(tab+2).
int& r = x; // crée une référence r sur x
r = 3; // affecte 3 à l'entier référé par r, c.-à-d x=3
```

4 Instructions de contrôle d'exécution et blocs d'énoncés

- `if(condition) enonce; [else enonce;]`
- `for(expression_init;expression_condition;expression_increment) enonce;`
- `while(condition) enonce;`
- `do enonce; while(condition);`
- `condition ? expression_sivrai : expression_sifaux;`
- `break` permet d'arrêter et de sortir de la bouche en cours.
- `continue` permet d'arrêter l'itération en cours et d'aller directement à la prochaine itération.
- Les accolades permettent de définir des blocs : `{ enonce1; enonce2; ...; enonceN;}`.

5 Classes, constructeurs, destructeurs, fonctions et appels de fonctions

```
// Déclaration (point.h)
#include <iostream>
using namespace std;
class Point {
 public:
  Point();
  Point(double x_,
        double y_);
  ~Point();

  double dist(const
    Point& p) const;

 private:
  double x, y;

friend istream& operator
    >> (istream& is,
    Point& p);
friend ostream& operator
    << (ostream& os, const
    Point& p);
};
```

```
// Définition (point.cpp)
#include <math.h>
#include "point.h"
Point::Point(){}
Point::Point(float x_, float y_)
 : x(x_), y(y_)
{ /* Alternative: x=x_; y=y_; */ }
Point::~Point(){}

float Point::dist(const Point& p) const {
  float dx = x - p.x, dy = y - p.y;
  return sqrt(dx * dx + dy * dy);
}
istream& operator>>(istream& is,Point& p) {
  char po, vir, pf;
  is >> po >> p.x >> vir >> p.y >> pf;
  return is;
}
ostream& operator<<(ostream& os, const
    Point& p) {
  os << "(" << p.x << "," << p.y << ")";
  return os;
}
```

```
// Utilisation (main.cpp)
#include "point.h"
#include <math.h>

void f1(
Point a // par valeur (copie),
Point* b // par pointeur,
Point& c // par référence,
const Point& d // réf. const
)
{
  float d1 = b->dist(a);
  float d2 = d.dist(c);
  *b = c = d; // p2 = p3 = p4
}

int main() {
  Point p1(0,0), p2(3,4), p3;
  float d = p1.dist(p2);
  Point* p4 = new Point();
  f1(p1, &p2, p3, *p4);
  delete p4;
}
```

Un constructeur (dans l'ordre) :

1. appelle le constructeur de la ou des classes héritées ;
2. appelle le constructeur de chaque variable d'instance ;
3. exécute le code dans le corps du constructeur.

Un destructeur (dans l'ordre) :

1. exécute le code dans le corps du destructeur.
2. appelle le destructeur de chaque variable d'instance ;
3. appelle le destructeur de la ou des classes héritées ;

6 Allocation de la mémoire automatique et dynamique

```
int main(){
  Point p1(0,0), p2(10,10), p3; // alloue automatiquement sur la pile d'exécution
  float d = p1.dist(p2);
  Point* p = new Point();    Point* points = new Point[10];
  delete p;                  delete[] points;
}
```

7 Classes génériques

```
template <class T> class Tableau {
  T*  elements;
  int capacite, nbElements;
public:
  Tableau(int capacite_initiale = 4);
  ~Tableau();
  int taille() const {return nbElements;}
  void ajouter(const T& item);
  T& operator[] (int index);
  const T& operator[] (int index) const;
};
```

```
template<class T> Tableau<T>::Tableau(int ci)
  : capacite(ci), nbElements(0),
    elements(new T[capacite]) {}
template<class T> Tableau<T>::~Tableau() {
  delete[] elements; // elements = nullptr;
}
template <class T>
T& Tableau<T>::operator[] (int index) {
  assert(index < nbElements);
  return elements[index];
}
```