INF3105 – Structures de données et algorithmes
Examen final – Hiver 2014

Éric Beaudry
Département d'informatique
Université du Québec à Montréal

Jeudi 24 avril 2014 – 18h00 à 21h00 (3 heures) – Local SH-3420

Instructions

- Aucune documentation n'est permise. Quelques algorithmes sont fournis à l'annexe B.
- 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 à condition de donner le code pour chacune d'elles ;
  - 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 et contient 5 questions.
- Ne détachez pas les feuilles du questionnaire, à l'exception des annexes à la fin.
- Le côté verso peut être utilisé comme brouillon. Des feuilles additionnelles peuvent être demandées.

Identification

Nom : ______________________________

Code permanent : ______________________________

Signature : ______________________________

Résultat

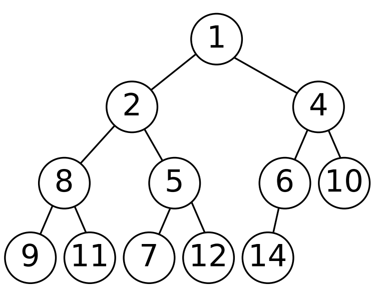
Q1		/ 5
Q2		/ 6
Q3		/ 5
Q4		/ 5
Q5		/ 4
Total		/ 25

1 Monceau (*Heap*) [5 points]

(a) Simulez l'insertion des nombres 6, 7, 2, 4, 8 et 3 dans un monceau. Dans le tableau ci-dessous, utilisez une ligne par étape. Une étape est l'ajout d'un nouvel élément ou l'échange de deux éléments. Le monceau est initialement vide. [1.5 point]

Étape	m[0]	m[1]	m[2]	m[3]	m[4]	m[5]	m[6]	m[7]	m[8]	m[9]	m[10]	m[11]
#0	–	–	–	–	–	–	–	–	–	–	–	–
#1												
#2												
#3												
#4												
#5												
#6												
#7												
#8												
#9												

(b) Soit le monceau suivant sous forme d'arbre. Simulez deux fois l'enlèvement de l'élément minimal. Montrez les étapes intermédiaires. Il n'est pas nécessaire de redessiner tout l'arbre à chaque étape. [1.5 point]



(c) Un collègue vous présente sa classe `Monceau` ci-dessous. Rappel : la structure `std::list` implémente une liste doublement chaînée.

```
#include <list>
template <class T>
class Monceau<T>{
  public:
    void inserer(const T&);
    T    enleverpremier();
    bool estVide() const;
  private:
    std::list<T> valeurs;
};
```

Selon vous, le choix de représentation est-il adéquat ? Justifiez votre réponse. [1 point]

(d) Dans la bibliothèque standard de C++ (anciennement dans la *Standard Template Library (STL)*), quelle est la structure de données qui implémente un monceau ? Indice : son usage était recommandé pour le TP3. [1 point]

2 Table de hachage (*Hashtable*) [6 points]

(a) Nommez un exemple d'application où l'usage de tables de hachage est pertinent. [1 point]

(b) Complétez le tableau suivant en indiquant la complexité temporelle des opérations dans une table de hachage contenant n entrées (paires clé-valeur). Utilisez la notation grand O. Utilisez une gestion de collisions à «adressage ouvert» (par opposition à l'usage d'une structure externe) avec «essais linéaires» (*linear probing*). [1 point]

Opération	Cas moyen	Pire Cas
Insertion d'une nouvelle entrée clé-valeur		
Retourner la valeur associée à une clé		

Considérez les fichiers source à l'annexe A (page 8).

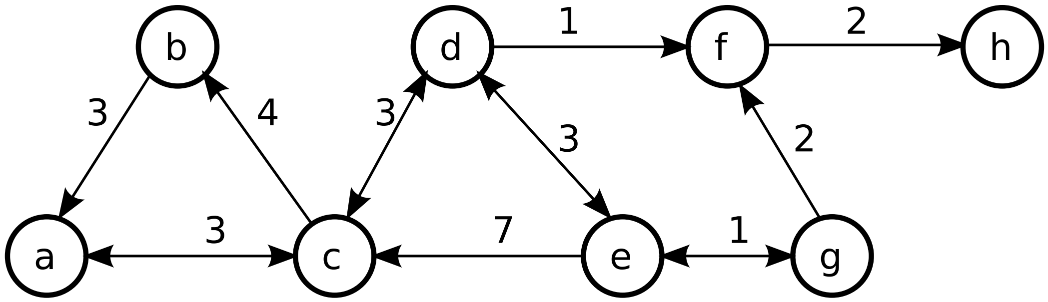
(c) Implémentez l'opérateur `[]`. Utilisez une gestion de collisions à «adressage ouvert» (par opposition à l'usage d'une structure externe) avec «essais linéaires» (*linear probing*). Vous n'avez pas à gérer le redimensionnement lorsqu'un taux de chargement (*load factor*) maximal est atteint. [2 points]

```
template <class K, V>
V& Dictionnaire::operator[](const K& cle){
  int h = cle(); // K::operator int() retourne la valeur de hachage











}
```

(d) Dessinez la représentation en mémoire du programme rendue à la ligne 10 de la fonction `main`. [2 points]

3 Graphes 1 [5 points]

Considérez le graphe ci-dessous. Pour les sous-questions (a) et (b), les arêtes sortantes doivent être parcourues en ordre alphabétique de leur sommet d'arrivée.



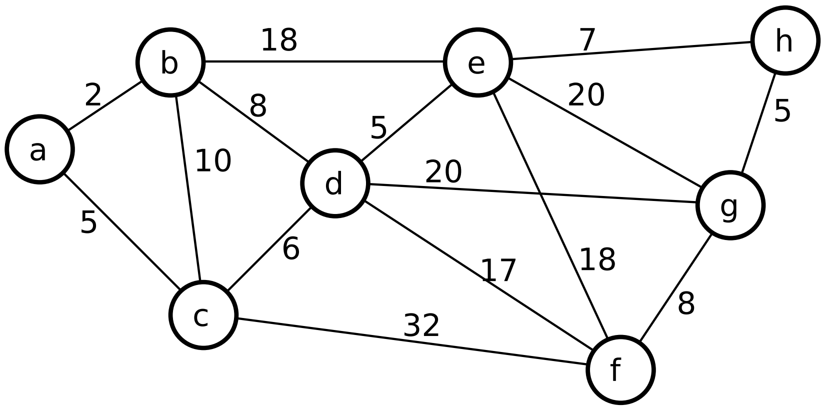
(a) Écrivez l'ordre de visite des sommets d'une recherche en **profondeur** à partir du sommet **b**. [1 point]

(b) Écrivez l'ordre de visite des sommets d'une recherche en **largeur** à partir du sommet **b**. [1 point]

(c) Énumérez les composantes fortement connexes du graphe. [1 point]

(d) Simulez l'algorithme de Dijkstra pour calculer le plus court chemin de **g** à **c**. Il y a plusieurs façons de présenter votre réponse. L'important est de démontrer votre compréhension de l'algorithme. Les éléments clés à présenter sont l'ordre de visite des sommets et les valeurs *Dist* et *Parent*. [2 points]

4 Graphes 2 [5 points]



(a) Simulez l'algorithme Kruskal pour retrouver un arbre de recouvrement minimal pour le graphe ci-haut. Montrez clairement la trace (ordre des arêtes). Lisez la sous-question (b) avant de répondre. [2 points]

(b) Vrai ou faux : la solution en (a) est unique. Justifiez en référant à votre trace en (a). [1 point]

(c) Selon vous, est-ce une bonne idée d'utiliser l'algorithme Floyd-Warshall plutôt que l'algorithme Dijkstra pour le TP3 ? Justifiez votre réponse en étant aussi précis que possible. [2 points]

5 Résolution d'un problème [4 points]

Lisez la mise en contexte à l'Annexe C (page 10).
Si vous désirez apporter des changements à la représentation de la classe `Carte`, faites-le ci-dessous.

```
class Carte{ /* ... */
    struct Lieu{
      bool station;
      map<string, double> voisins;

    };
    map<string, Lieu> lieux;

/* ... */ };
```

Choisissez la fonction `existeChemin` (max 3 points) ou `calculerChemin` (max 4 points) et écrivez-la.

```
______________ Carte::__________Chemin(string o, string d){
```

Annexe A – Pour la Question 2

Cette page peut être détachée. La classe générique `Dictionnaire<K,V>` implémente une table de hachage. La variable `Dictionnaire<K,V>::nbEntrees` contient le nombre d'entrées insérées dans le dictionnaire, à ne pas confondre avec le nombre de casiers (*buckets*). Le constructeur `Tableau<K,V>::Tableau(int t)` initialise un tableau de taille `t` en utilisant le constructeur `T::T()`.

```
/*  dictionnaire.h */
template <class K, class V>
class Dictionnaire{
public:
  Dictionnaire();
  V& operator[] (const K& cle);
  bool operator==(const
    Dictionnaire&) const;
  int taille() const;
  // ...
private:
  struct Casier{
    Casier(){utilise=false;}
    K cle;
    V valeur;
    bool utilise;
  };
  //nombre d'entrees cle-valeur
  int nbEntrees;
  Tableau<Casier> casiers;
};

template <class K, class V>
Dictionnaire<K,V>::Dictionnaire()
:casiers(10), nbEntrees(0)
{
}
```

```
/* foo.h */
class Foo{
public:
  Foo(int x_=0, int y_=0);
  Foo(const Foo&);
  bool operator == (const Foo& f) const;
  operator int() const;
private:
  int x, y;
};
```

```
/* foo.cpp */
Foo::Foo(int x_, int y_)
: x(x_), y(y_) {}
Foo::Foo(const Foo& f)
: x(f.x), y(f.y) {}
bool Foo::operator==(const Foo& f) const
{
  return x==f.x && y==f.y;
}
// Fonction de hachage
Foo::operator int() const{
  return x*2+y*3;
}
```

```
/*  question2.cpp */
#include ...
int main(int argc, const char** argv){
  Dictionnaire<Foo, std::string> d1, d2;
  d1[Foo(1,3)] = "A";
  d1[Foo(6,3)] = "B";
  d1[Foo(4,5)] = "C";
  d2[Foo(4,5)] = "C";
  d2[Foo(6,3)] = "B";
  d2[Foo(1,3)] = "A";
  // Dessinez representation memoire rendu ici.
  //bool b = d1==d2;
  return 0;
}
```

Annexe B – Rappel d'algorithmes de graphes

```
RECHERCHEPROFONDEUR(G = (V,E), v ∈ V)
  v.visité ← vrai
  pour toute arête e ∈ v.aretesSortantes()
    w ← e.arrivee
    si ¬w.visité
      RechercheProfondeur(G, w)
```

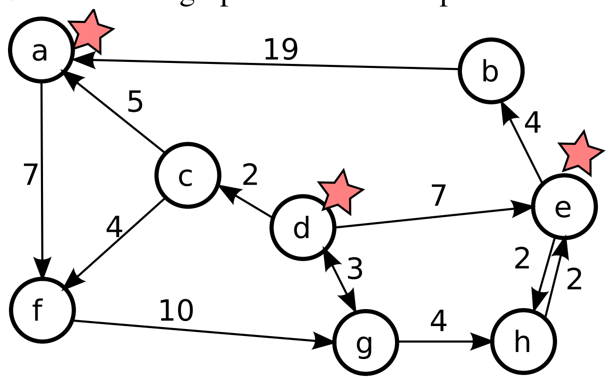
```
DIJKSTRA(G = (V,E), s ∈ V)
  pour tout v ∈ V
    distances[v] ← +∞
    parents[v] ← indéfini
  distances[s] ← 0
  Q ← créer FilePrioritaire(V)
  tant que ¬Q.vide()
    v ← v ∈ Q avec la plus petite valeur distances[v]
    si distances[v] = +∞ break
    pour toute arête sortante e = (v,w) depuis le sommet v
      d ← distances[v] + e.distance
      si d < distances[w]
        parents[w] ← v
        distances[w] ← d
        Q.réduireClé(w)
  retourner (distances, parents)
```

```
FLOYD_WARSHALL(G = (V,E))
  distances ← créer tableau |V| × |V| initialisé à +∞
  directions ← créer tableau |V| × |V| initialisé à « ?»
  pour tout v ∈ V
    distances[v][v] ← 0
    directions[v][v] ← v
  pour tout e ∈ E
    distances[e.out][e.in] ← e.cout
    directions[e.out][e.in] ← e.in
  pour k = 0 à |V| − 1
    pour i = 0 à |V| − 1
      pour j = 0 à |V| − 1
        si distances[i][k] + distances[k][j] < distances[i][j]
          distances[i][j] ← distances[i][k] + distances[k][j]
          directions[i][j] ← directions[i][k]
  retourner directions
```

```
KRUSKAL(G = (V,E))
  pour tout sommet v ∈ V
    C(v) ← créer un ensemble {v}
  Q ← créer FilePrioritaire<Arête>(E) où la clé est le poids
  tant que |E′| < |V| − 1
    e = (v1, v2) ← Q.enleverMinimum()
    si C(v1) ≠ C(v2)
      ajouter e à E′
      fusionner C(v1) et C(v2) afin que C(v1) = C(v2)
  retourner G′ = (V,E′)
```

Annexe C – Pour Question 5

Considérer le graphe ci-dessous représentant une carte.



L'algorithme de Dijkstra permet de trouver un chemin le plus court entre une paire de sommets dans un graphe. Par exemple, le chemin le plus court reliant a à e est le chemin $\langle a, f, g, h, e\rangle$. Le chemin le plus cours reliant a à h est le chemin $\langle a, f, g, h\rangle$.

À la question 5, ce problème est légèrement complexifié. On y ajoute une contrainte, soit la quantité de carburant dans le réservoir du véhicule. On suppose que le véhicule a un réservoir d'une capacité de 20 unités de carburant. Le véhicule part toujours avec son réservoir plein. Le véhicule consomme une unité de carburant par unité de distance. Le véhicule peut faire le plein aux stations de recharge marquées par une étoile (a, d, e). Le temps de remplissage n'est pas considéré.

Dans ce nouveau contexte, le chemin $\langle a, f, g, h, e\rangle$ n'est plus un **chemin faisable**, car le véhicule manquera une unité de carburant sur le tronçon (g, h). Le meilleur chemin pour se rendre de a à e est maintenant $\langle a, f, g, d, e\rangle$. Dans certains cas, il faut passer jusqu'à deux fois sur un même sommet. Par exemple, le meilleur chemin pour se rendre de a à h est maintenant $\langle a, f, g, d, g, h\rangle$. Il n'existe aucun chemin faisable reliant e à a.

À la question 5, on vous demande d'écrire une fonction pour vérifier l'existence ou pour calculer un **chemin faisable**. Notez qu'un chemin faisable n'est pas nécessairement optimal (le plus court possible). Votre fonction `calculerChemin` n'a pas besoin de générer un chemin optimal. Comme point de départ, on vous donne la déclaration de la classe `Carte` suivante.

```
class Carte{
public:
 void ajouterLieu(const string& nom, bool station);
 void ajouterRoute(const string& n1, const string& n2, double longueur);
 bool existeChemin(string origine, string destination) const; // 3 points
 list<string> calculerChemin(string origine, string destination) const; // 4 points
private:
 struct Lieu{
   bool station; // lieux["a"].station=true; lieux["b"].station=false;
   map<string, double> voisins;//Ex: lieux["a"].voisins["f"]=7;
                               //lieux["c"].voisins["a"]=5;lieux["c"].voisins["f"]=4;
 };
 map<string, Lieu> lieux;
};
```

Suggestion : adaptez l'algorithme du parcours «recherche en profondeur» en permettant de visiter jusqu'à 2 fois un même sommet. L'algorithme «recherche en profondeur» s'implémente naturellement de façon récursive.

La fonction `existeChemin` doit retourner vrai ssi il existe un chemin faisable de l'origine à la destination. La fonction `calculerChemin` retourne une liste de noms de lieux d'un chemin faisable. Si aucun chemin n'existe, il suffit de retourner une liste vide.