INF3105 – Structures de données et algorithmes
Été 2016 – Examen final

Éric Beaudry
Département d'informatique
Université du Québec à Montréal

Jeudi 28 juillet 2016 – 13h30 à 16h30 (3 heures) – Locaux PK-R220 et PK-R250

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. L'examen dure 3 heures et contient 6 questions et vaut 25 % de la session.
7. Ne détachez pas les feuilles du questionnaire, à l'exception des annexes à la fin.
8. Le côté verso peut être utilisé comme brouillon. Des feuilles additionnelles peuvent être demandées.

Identification

Nom : _____

Code permanent : _____

Signature : _____

Résultat

Q1		/ 20
Q2		/ 16
Q3		/ 14
Q4		/ 15
Q5		/ 20
Q6		/ 15
Total		/ 100

1 Monceaux (*Heaps*) [20 points]

(a) Insérez les entiers 7, 8, 1, 5, 9, 10 et 5 dans un monceau initialement vide. Dessinez sous forme d'arbre l'état du monceau après chaque insertion. Les étapes intermédiaires ne sont pas demandées. [10 points]

(b) Codez la fonction `Monceau::enleverMin` qui enlève et retourne le minimum d'un monceau. Contrairement à celle présentée en classe et dans les notes de cours, **votre solution ne doit pas être récursive**. Vous pouvez supposer l'existence des fonctions telles que `void Tableau::enleverDernier()`, `T& Tableau::operator[](int)` et `int Tableau::taille()`. La représentation `Monceau` contient un seul objet : `Tableau<T> tab`. [10 points]

```
1  template <class T> T Monceau::enleverMin() {
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25                                                    }
```

2 Arbres binaires de recherche *vs* Tables de hachage [16 points]

Voici deux fonctions similaires. Celle de gauche utilise un conteneur `map` basé sur un arbre rouge-noir. Celle de droite utilise un conteneur `unordered_map` basé sur une table de hachage. Supposez que le conteneur `unordered_map` utilise une liste simplement chaînée comme structure externe pour la gestion des collisions.

```
1  int main1(){
2    map<string,int> compteurs;
3    while(cin){
4      string mot;
5      cin >> mot;
6      compteurs[mot]++;  }
7    map<string, int>::iterator
8     iter = compteurs.begin();
9    for(;iter!=compteurs.end();++iter)
10     cout << iter->first << " : "
11          << iter->second << endl;
12 }
```

```
1  int main2(){
2    unordered_map<string,int> compteurs;
3    while(cin){
4      string mot;
5      cin >> mot;
6      compteurs[mot]++;  }
7    unordered_map<string, int>::iterator
8      iter = compteurs.begin();
9    for(;iter!=compteurs.end();++iter)
10     cout << iter->first << " : "
11          << iter->second << endl;
12 }
```

(a) Quelle est la complexité temporelle de chacune de ces deux fonctions ? Considérez le **cas moyen** et le **pire cas**. Exprimez l'ordre de grandeur en fonction des paramètres n et m où n est le nombre de mots dans l'entrée et m est le nombre de mots différents. [4 points]

main1 / cas moyen :

main1 / pire cas :

main2 / cas moyen :

main2 / pire cas :

(b) Expliquez intuitivement ce que serait un **pire cas** pour la fonction `main2`. [4 points]

(c) Quelle fonction entre `main1` et `main2` recommanderiez-vous ? Justifiez. [4 points]

(d) Selon vous, les deux programmes produisent-ils exactement la même sortie ? Expliquez. [4 points]

3 Table de hachage (*Hashtable*) [14 points]

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

(a) Codez `Dictionnaire<T>::operator[]`. [7 points]

```
1  template <class K, class V> V& Dictionnaire::operator[](const K& cle) {
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20                                                                        }
```

(b) Dessinez la représentation en mémoire de l'objet `agenda` rendu à la ligne 44. [7 points]

4 Graphes / Représentation et simulation d'algorithmes [15 points]

(a) Dessinez la représentation en mémoire du graphe G. [5 points]

(b) Écrivez l'ordre de visite des sommets lors d'un parcours «recherche en ___» à partir du sommet d. Supposez que les arêtes sortantes d'un sommet sont énumérées dans le même ordre qu'elles ont été ajoutées. [4 points]

profondeur : $\langle d,$	largeur : $\langle d,$

(c) Écrivez le résultat de l'algorithme **Floyd-Warshall**. Les étapes intermédiaires ne sont pas demandées. Dans le tableau de gauche, écrivez les distances. Dans le tableau de droite, écrivez les directions. [4 points]

	destination				
origine	a	b	c	d	e
a	0				
b		0			
c			0		
d				0	
e					0

	destination				
origine	a	b	c	d	e
a	–				
b		–			
c			–		
d				–	
e					–

(d) Le graphe G est-il fortement connexe ? Justifiez en vous référant uniquement à votre réponse en (c). [2 points]

5 Graphes / Analyse de la complexité [20 points]

Voici une fonction implémentant l'algorithme Prim-Jarnik. La représentation de `Graphe` est à l'Annexe B.

```
1  Graphe Graphe::prim_jarnik() const{
2      Graphe r;
3      r.sommets[sommets.begin()->first]; // ajout sommet départ
4      while(true){
5          string v, w;
6          double z = std::numeric_limits<double>::infinity();
7          map<string, list<Arete> >::iterator i;
8          for(i=r.sommets.begin();i!=r.sommets.end();++i){
9              const list<Arete>& as = sommets.at(i->first); // [i->first]
10             for(list<Arete>::const_iterator j=as.begin();j!=as.end();++j)
11             if(r.sommets.find(j->arrivee)==r.sommets.end() && j->poids<z){
12                 v = i->first;
13                 w = j->arrivee;
14                 z = j->poids;
15             }
16         }
17         if(std::numeric_limits<double>::infinity()==z)
18             break;
19         r.sommets[w]; // ajout sommet
20         r.sommets[v].push_back(Arete(w, z));
21     }
22     return r;
23 }
```

Analysez la complexité temporelle de l'implémentation ci-haut. Exprimez votre réponse en notation grand O. Supposez $n = |V|$, c'est-à-dire le nombre de sommets, et $m = |E|$, c'est-à-dire le nombre d'arêtes. Il est suggéré d'annoter le code ci-haut en faisant l'analyse ligne par ligne.

Analyse :

Réponse :

6 Graphes / Résolution d'un problème [15 points]

Lisez la problématique à l'Annexe C. Expliquez comment implémenter la fonction `Graphe::suggerer_lieu_rencontre(string l1, string l2)` qui suggère le lieu du café «le plus près» pour deux personnes situées à `l1` et `l2`. Soyez aussi précis que possible. Vos explications doivent être suffisantes pour coder la solution. Vous pouvez expliquer comment utiliser et/ou adapter des structures de données et des algorithmes vus dans le cours.

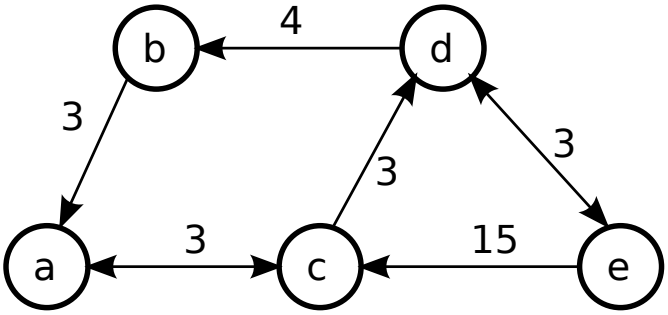
Annexe A pour la Question 3

À noter que le code a été condensé pour rentrer sur une seule page.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  class Date{
5      int annee, mois, jour;
6    public:
7      Date(int a=1900, int m=1, int j=0)  : annee(a), mois(m), jour(j) { }
8      int hash() const; // retourne la valeur "hash"
9      bool operator==(const Date&) const;
10 };
11 int Date::hash() const {
12   return annee%10 + mois/2 + jour; //rappel : mois/2 donne un int
13 }
14 template <class K, class V>
15 class Dictionnaire{
16     struct Entree{
17         Entree(const K& c) : cle(c), valeur(), suivante(NULL){ }
18         K cle;
19         V valeur;
20         Entree* suivante;
21     };
22     Entree** casiers; // pointe sur un tableau de pointeurs d'entrées
23     int nc; // nombre de casiers
24   public:
25     Dictionnaire();
26     ~Dictionnaire();
27     V& operator[](const K&);
28 };
29 template <class K, class V> Dictionnaire<K,V>::Dictionnaire() : nc(11){
30     casiers = new Entree*[nc];
31     for(int i=0;i<nc;i++)
32         casiers[i] = NULL;
33 }
34 int main(){
35     Dictionnaire<Date, std::string> agenda;
36     agenda[Date(1776, 7,  4)] = "USA";     // (6+7/2+4)%11=2
37     agenda[Date(1789, 7, 16)] = "France";  // (9+7/2+16)%11=6
38     agenda[Date(1867, 7,  1)] = "Canada";  // (7+7/2+1)%11=0
39     agenda[Date(1954, 6,  7)] = "Turing";  // (4+6/2+7)%11=3
40     agenda[Date(2016, 5,  2)] = "INF3105"; // (6+5/2+5)%11=2
41     agenda[Date(2016, 6, 23)] = "Examen1"; // (6+6/2+23)%11=10
42     agenda[Date(2016, 7, 28)] = "Examen2"; // (6+7/2+28)%11=4
43     agenda[Date(2016, 8,  4)] = "TP3";     // (6+8/2+4)%11=3
44     /****** ligne 44 pour la sous-question 3b. ******/
45     return 0;
46 }
```

Annexe B pour les questions 4 et 5

Ci-bas à gauche, le graphe $G = (V,E)$ où $V = \{a,b,c,d,e\}$ et $E = \{(a,c,3),(b,a,3),(c,a,3),(c,d,3),(d,b,4),(d,e,3),(e,c,15),(e,d,3)\}$. Ce graphe est chargé dans un objet de type `Graphe` défini ci-bas à droite. Rappels : `map` est dictionnaire basé sur un arbre rouge-noir ; `list` est une liste doublement chaînée. Supposez que les arêtes ont été ajoutées en ordre lexicographique (même ordre que l'énumération de E ci-haut).

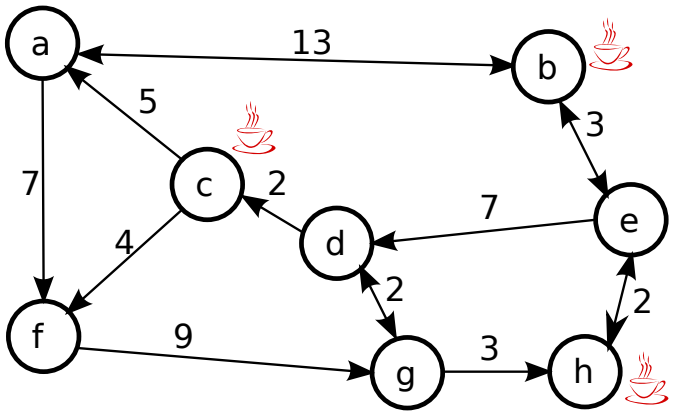



```
1  class Graphe{
2    struct Arete{
3      Arete(string, double);
4      string arrivee; // sommet d'arrivée
5      double poids;
6    };
7    map<string, list<Arete> > sommets;
8  };
```

Annexe C pour la Question 6

Une entreprise, qui opère un site web de rencontres, a conclu des ententes avec des cafés. Lorsque deux membres du site désirent se rencontrer en personne, pour prendre un verre ou un dessert, le site de rencontres leur suggère le café «le plus près». Le café «le plus près» est le café qui permet aux deux membres de se rencontrer le plus tôt possible. Il s'agit du café qui minimise le temps de déplacement du membre le plus éloigné (en temps de déplacement).

À titre d'exemple, voici ci-dessous une carte d'une région fictive. Le poids des arêtes indique la distance, en unités de temps, séparant deux nœuds. Sur la carte, il y a 3 cafés partenaires situés aux sommets b, c et h.




```
1  class Graphe{
2    struct Sommet{
3      map<string,int> as;//nom-->poids
4      bool cafe;//indique présence café
5    };
6    map<string, Sommet> sommets;
7  public:
8    string suggerer_lieu_rencontre
9      (string l1, string l2) const;
10 };
```

Exemple 1. Deux membres X et Y, respectivement situés aux nœuds d et f, désirent se rencontrer. Le café «le plus près», qui permet la rencontre le plus tôt possible, est celui au nœud h. Le membre X initialement à d peut se rendre à h en 5 unités de temps. Le membre Y initialement à f peut se rendre à h en 12 unités de temps. La rencontre est donc possible après $\max(5,12) = 12$ unités de temps. Bien que X soit à 2 unités de temps du café au nœud d, une rencontre au café d n'est pas possible avant 13 unités de temps, soit la longueur du chemin $\langle f,g,d,c\rangle$ pour Y. Le café situé au nœud b permet au plus tôt une rencontre après 17 unités de temps.

Exemple 2. Deux membres V et W, respectivement situés aux nœuds c et b, désirent se rencontrer. Le café «le plus près», qui permet la rencontre le plus tôt possible, est celui au nœud c. Cette rencontre peut avoir lieu après 12 unités de temps.