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 ou la proposition d'une solution :
   — le fonctionnement correct, l'efficacité (temps et mémoire), la clarté, la simplicité 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.

Résultat

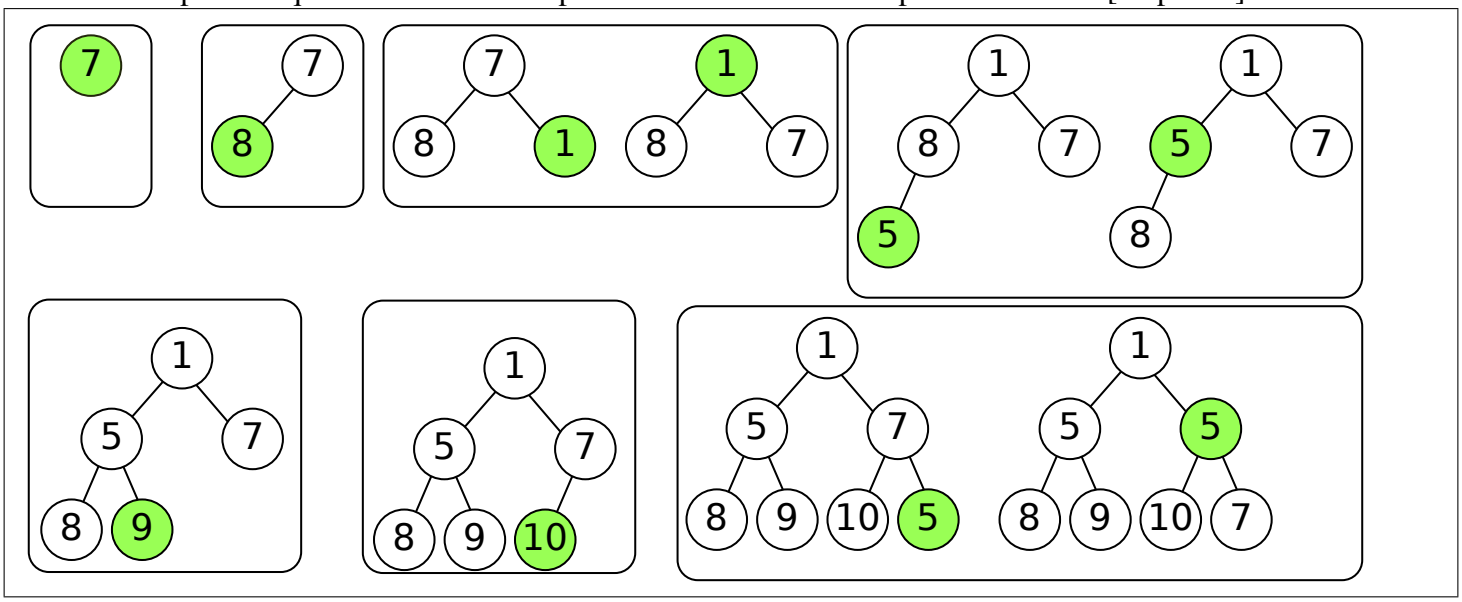
Q1		/ 20
Q2		/ 16
Q3		/ 14
Q4		/ 15
Q5		/ 20
Q6		/ 15
Total		/ 100

Identification

Nom, Prénom : **Solutionnaire**

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      T min = tab[0];  // Garder une copie du plus petit, à la racine
3      tab[0] = tab[tab.taille()-1]; // Mettre le dernier à la racine
4      tab.enleverDernier();         // Réduire taille du tableau
5      int p = 0;                    // p : l'indice de l'élément à descendre
6      while(true){                   // while(2*p+1<tab.taille)
7          int e = 2*p+1;            // position le premier enfant
8          if(e>=tab.taille())       // le premier enfant existe-t-il ?
9              break;                // Si inexistant, p est une feuille
10         if(e+1<tab.taille()       // Si le deuxième enfant existe
11            && tab[e+1]<tab[e])    //   et qu'il est plus petit que le premier
12             e++;                  // alors l'enfant à considérer est le 2e
13         if(tab[p]<=tab[e])        // Avons-nous terminé ?
14             break;                // oui si on a pas à descendre
15         echanger(tab[e], tab[p]); // Procéder à l'échange
16         p = e;                    // Faudra continuer avec e
17     }
18     return min;                    // Retourner le min gardé en copie
19 }
```

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 : $O(n \log m)$

main1 / pire cas : $O(n \log m)$

main2 / cas moyen : $O(n)$

main2 / pire cas : $O(nm)$

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

Le pire cas survient lorsque les m mots différents ont la même adresse réduite (ex. : valeur de hachage modulo taille du tableau). Dans un tel cas, toutes les entrées seront en collision dans le même casier (*bucket*). Ainsi ce casier sera associée à une liste chaînée de longueur m. Chaque opération de recherche (ex. : `operator[]` dégénéreront en temps $O(m)$. Ainsi, pour faire n insertions, il en coûtera $O(nm)$.
Dire que le pire cas est lorsque tous les mots sont différents, donc $m = n$, avec la même valeur de hachage est aussi accepté. Dans ce cas, on peut écrire $O(nm)$ ou $O(n^2)$.

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

Recommandation : `main2`. La fonction `main2` offre un meilleur temps moyen. Le pire cas avec la fonction `main2` est raisonnablement extrêmement rare en pratique. À l'exception d'une attaque ciblée, où les mots seraitent soigneusement choisis dans le but de créer volontairement des collisions, une telle dégénérescence est pratiquement impossible, même si théoriquement possible. Recommandation alternative : `main1` lorsqu'on veut avoir une garantie absolue contre le pire cas.

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

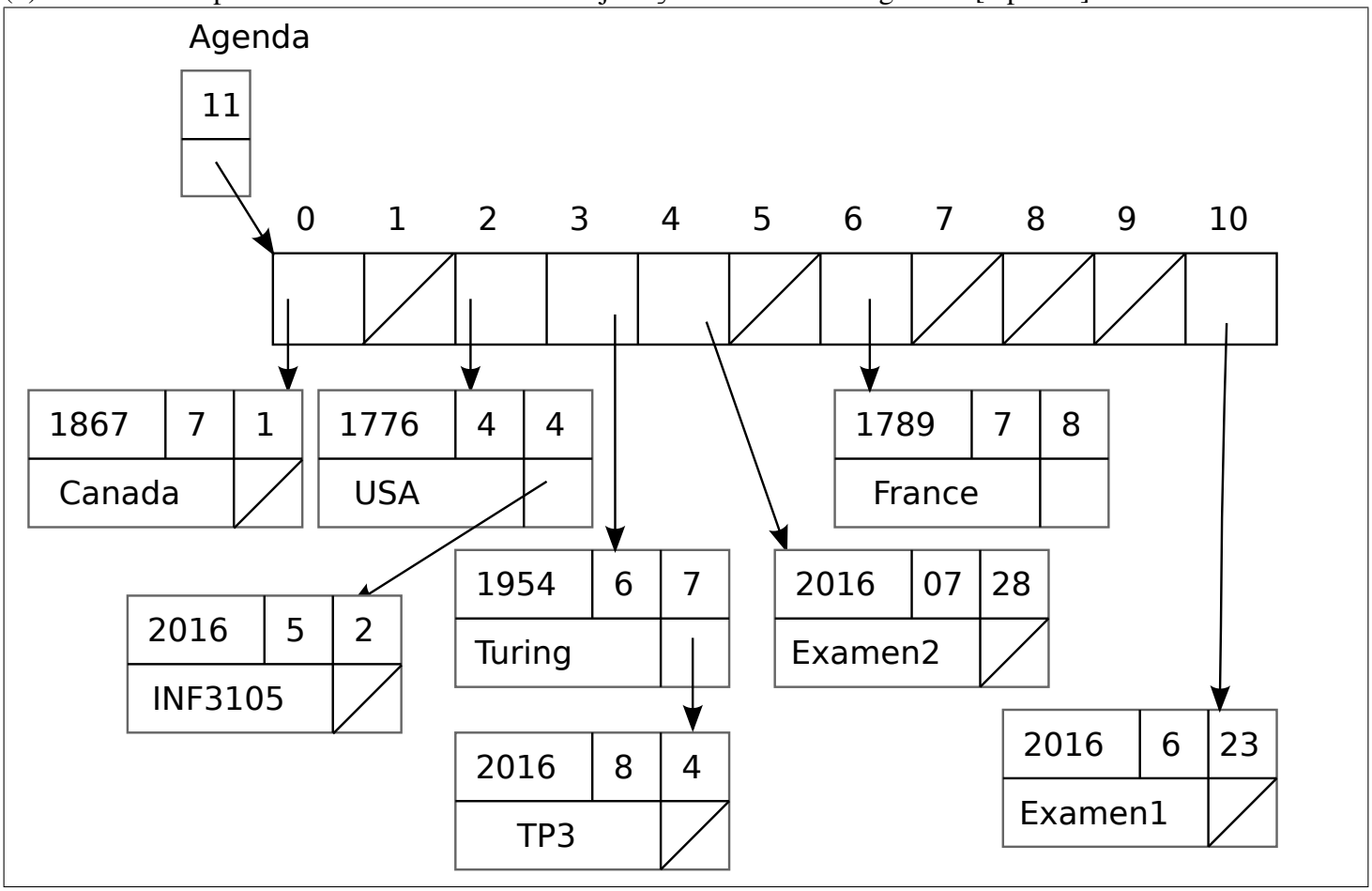
Non, les sorties ne sont pas parfaitement identiques. L'unique différence sera l'ordre dans lequel les mots, avec leur fréquence, seront affichés. Dans `main1`, avec un `map`, les mots seront triés en ordre alphabétique. Dans `main2`, avec un `unordered_map`, les mots seront dans un ordre indéfini. On peut voir cet ordre comme un ordre pseudo-aléatoire qui dépend des mots, de leur ordre d'apparition, de la fonction de hachage, du nombre de casiers (pouvant être variable), etc.

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]

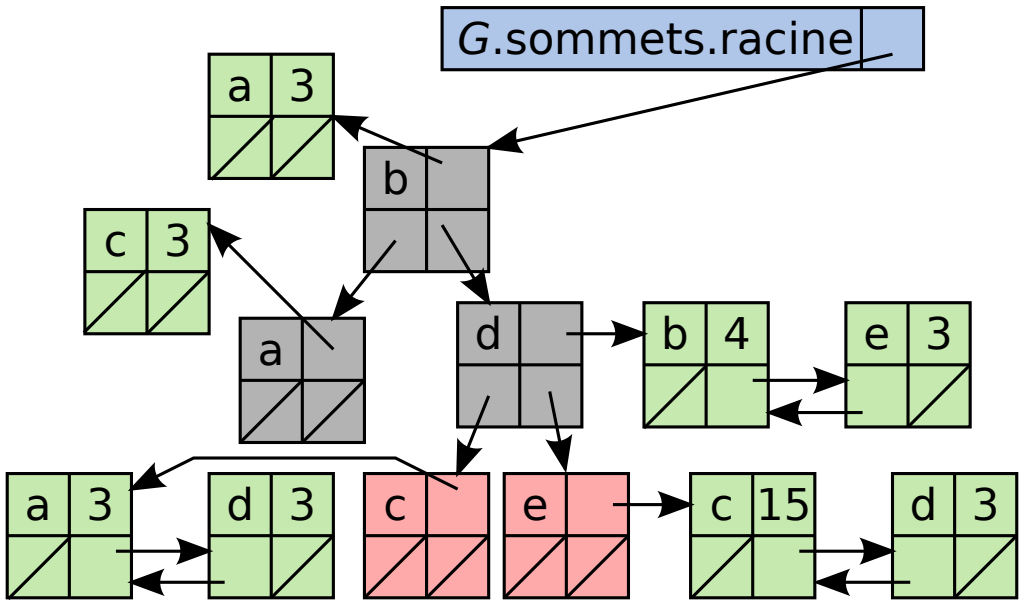
```
1  template <class K, class V> V& Dictionnaire::operator[](const K& cle) {
2      Entree** e;  // pointeur sur un pointeur d'Entrée
3      e = casiers + cle.hash() % taille; //pointeur sur le casier contenant un pointeur
4      // Rechercher la clé dans la liste chainée d'entrées de type Entree
5      while(*e!=NULL){
6          if((*e)->cle == cle) // test de la clé
7              return (*e)->valeur; // si bonne clé, retourner référence sur la valeur
8          e = &((*e)->suivante); // sinon, passer à l'entrée suivante
9      }
10     *e = new Entree(cle); // si *e==NULL ==> insérer nouvelle clé
11     return (*e)->valeur;  // enfin, retourner réf sur valeur de nouvelle entrée
12 }
```

(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]



Idéalement, l'arbre devrait être structuré ainsi pour refléter l'ordre d'insertion des sommets. Les détails sur la représentation rouge-noir (couleur des noeuds ou pointeurs) ne sont pas corrigés. Les listes doivent être des listes doublement chaînées, tel que mentionné dans l'Annexe B.

(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 : $\langle d, b, a, c, e \rangle$

largeur : $\langle d, b, e, a, c \rangle$

(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	10	3	6	9
b	3	0	6	9	12
c	3	7	0	3	6
d	7	4	10	0	3
e	10	7	13	3	0

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

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

Oui, le graphe G est fortement connexe. S'il n'était pas fortement connexe, il serait resté au moins une valeur $+\infty$ dans la matrice de distances après l'algorithme Floyd-Warshall.

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;  // O(1)
3      r.sommets[sommets.begin()->first]; // O(log n) pour begin
4      while(true){ // Il y aura au maximum n-1 itérations, donc O(n) itérations
5          string v, w; // O(1)
6          double z = std::numeric_limits<double>::infinity(); // O(1)
7          map<string, list<Arete> >::iterator i; // O(1)
8          for(i=r.sommets.begin();i!=r.sommets.end();++i){ // 1, ..., n-1 itérations
9              const list<Arete>& as = sommets.at(i->first); // O(log n)
10             for(list<Arete>::const_iterator j=as.begin();j!=as.end();++j)//O(m)/Ligne4
11                 if(r.sommets.find(j->arrivee)==r.sommets.end() &&j->poids<z){//O(log n)
12                     v = i->first; // O(1)
13                     w = j->arrivee; // O(1)
14                     z = j->poids; // O(1)
15                 }
16         }
17         if(std::numeric_limits<double>::infinity()==z) // O(1)
18             break;  // O(1)
19         r.sommets[w].push_back(Arete(v, z)); // O(log n) pour [] et O(1) pour push_back
20         r.sommets[v].push_back(Arete(w, z)); // idem
21     }
22     return r; // variale, mais au maximum O(n+m)
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 :

Ligne 4 : Chaque itération de la boucle `while` choisi un sommet à connecter dans le graphe résultat. La boucle fera donc au plus $n-1$ itérations. Ce maximum est atteint lorsque le graphe est connexe.

Ligne 8 : la boucle `for` fera respectivement 1, 2, 3, ..., $n-1$ itérations pour chaque itération de la boucle `while` à la ligne 4.

Ligne 9 : la recherche (`at comme []`) coûte $O(\log n)$ à chaque itération. Globalement, pour la fonction (toutes les itérations de lignes 4 et 8), on obtient : $O(n^2 \log n)$.

Ligne 10 : la boucle `for` itère sur les arêtes sortantes du sommet courant de la ligne 8. Le nombre d'arêtes est variable. En moyenne, il y m/n arêtes sortantes par sommet. La moyenne peut être considérée, car globalement le 2e `for` itérera sur au plus m arêtes par itération de la boucle `while`. Puisque la première boucle `for` (ligne 8) fait 1, 2, 3, ..., $n-1$ itérations, la deuxième boucle fera respectivement m/n, $2m/n$, $3m/n$, $\cdots$, $(n-1)m/n$ itérations pour chaque itération de la boucle `while` à la ligne 4. Ainsi, on obtient en moyenne $O(m/n)$ itérations de la boucle `for` de la ligne 10.

Ligne 11 : la fonction `find` implique une recherche, donc $O(\log n)$. Globalement, pour toute la fonction : $O(m \cdot n \cdot \log n)$.

Les autres lignes sont négligeables.

Coût global : $O(n^2 \log n + m \cdot n \cdot \log n)$. Puisque $m > n$, on peut simplifier.

Réponse : $O(m \cdot n \cdot \log n)$

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.

Approche naïve et peu efficace [jusqu'à 10/15]. L'idée consiste à faire une boucle qui itère sur tous les cafés. Pour chaque café c_i, on lance deux fois l'algorithme de Dijkstra, la première pour calculer $distance(l1, c_i)$ et la deuxième pour calculer $distance(l2, c_i)$. On garde le temps minimal pour le point de rencontre : $t_i = \max(distance(l1, c_i), distance(l2, c_i))$ Enfin, on garde le café c_i ayant le plus petit t_i. Complexité temporelle théorique de Dijkstra (meilleure implémentation possible) : $O(m \log n)$. Pour k cafés, on obtient : $O(k \cdot m \log n)$.

Approche moyennement efficace [jusqu'à 13/15]. L'algorithme de Dijkstra peut calculer toutes les distance à partir d'une origine vers tous les noeuds de destination possible (*point-to-multipoint*). Ainsi, on lancer Dijkstra seulement 2 fois (plutôt que $2k$ fois), depuis $l1$ et $l2$, pour découvrir tous les cafés accessibles depuis ces deux points de départ. Complexité temporelle : en $O(m \log n)$. Ensuite, il suffit d'itérer sur tous les cafés pour trouver celui qui minimise $\max(distance(l1, c), distance(l2, c))$. Cela demande seulement k itérations qui sont négligeables par rapport à $O(m \log n)$.

Approche efficace sans précalculs [jusqu'à 15/15]. L'approche moyenne sera coûteuse en temps si les lieux $l1$ et $l2$ sont relativement proches comparativement à la taille de la carte. Dans l'application visée, il est raisonnable de croire que cela sera très fréquent. Imaginez un site de rencontres qui opère à la grandeur du Québec ou même sur toute l'Amérique du nord. Les rencontres sont généralement entre deux membres vivants dans la même région. Il est rare de suggérer un point de rencontre à deux membres demeurant respectivement en Gaspésie et dans l'ouest du Québec !
Diverses stratégies peuvent être utilisées pour éviter de visiter tous les noeuds de la carte et de parcourir toutes les arêtes. La difficulté ici est de savoir quand arrêter l'algorithme de Dijkstra. Comme la destination (quel café) est inconnue à l'avance, on peut pas arrêter l'algorithme à une destination fixe. L'astuce consiste à modifier l'algorithme de Dijkstra pour faire deux recherches en parallèle, l'une depuis $l1$ et l'autre depuis $l2$. On modifie la file prioritaire de Dijkstra pour avoir 2 files prioritaires (une par sommet de départ $l1$ et $l2$). Une seule boucle `while` itère tant que nécessaire. À chaque itération, on pige dans la file prioritaire dont le *top* a la valeur D la plus petite. Cela permet de faire progresser deux instances de algorithme de Dijkstra simultanément. (Les deux files prioritaires pourraient aussi être combinées en une seule.) L'algorithme termine lorsqu'un noeud, possédant un café, a été visité par les 2 instances, c'est-à-dire à partir de $l1$ et $l2$. Le premier café trouvé à partir de $l1$ et $l2$ sera le café à suggérer. Coût : $O(m' \log n')$ où n' et m' sont respectivement le nombre de sommets et d'arêtes dans le sous-graphe exploré autour de $l1$ et $l2$. Le coût est significativement plus faible, car $n' << n$ et $m' << m$ lorsque $l1$, $l2$ et le café suggéré sont relativement proches.

Approche très efficace avec précalculs [jusqu'à 15/15]. Comme mentionné dans l'approche moyennement efficace, l'algorithme de Dijkstra peut calculer toutes les distances à partir d'une origine vers tous les noeuds (*point-to-multipoint*). En parcourant les arêtes entrantes plutôt que sortantes, l'inverse est aussi possible (*multipoint-to-point*). Ainsi, on peut précalculer toutes les distances vers tous les cafés depuis tous les noeuds de la carte. Pour k cafés, cela coûtera $O(k \cdot m \log n)$ en temps. Les tables coûteront $O(k \cdot n)$ en mémoire. Une fois ces calculs effectués, il sera possible de déterminer en temps $O(k)$ le café le plus près à recommander. Et avec un peu d'efforts, en utilisant des structures de données avancées non vues dans le cours (ex. : arbre k-D), il serait possible de réduire à $O(\log k + \log n)$.
Si l'investissement initial est trop important en temps et/ou mémoire, il est possible de mettre une limite à l'algorithme de Dijkstra pour visiter les noeuds pas plus loin qu'une distance maximal (ex. : 100 km). Le coût est significativement plus faible : $O(k \cdot m' \log n')$ en temps et $O(k \cdot n')$ en mémoire.

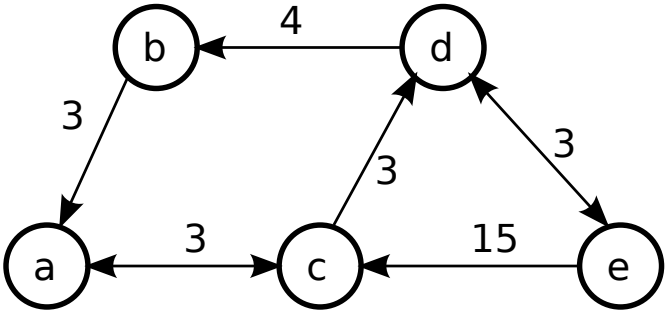
Annexe A pour la Question 3

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).

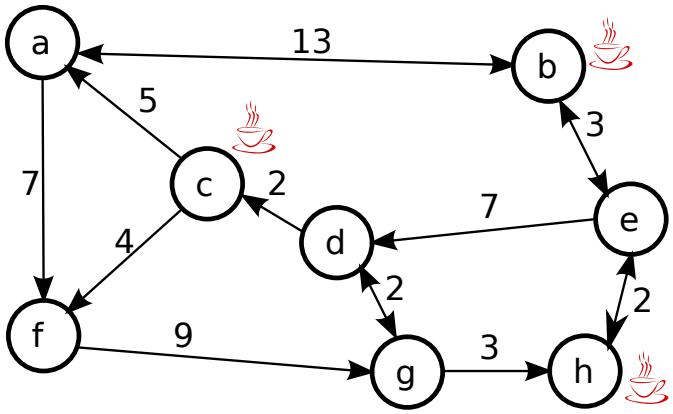



```
1  class Graphe{
2    struct Arete{
3      Arete(string, double);
4      string arrivee; // sommet d'arrivée
5      double poids;
6    };
7    map<string, list<Arete> > sommets;
8  };
```

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 c, une rencontre au café c 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.