INF3105 – Structures de données et algorithmes
Examen final – Automne 2013

Éric Beaudry
Département d'informatique
Université du Québec à Montréal

Jeudi 12 décembre 2013 – 13h30 à 16h30 (3 heures) – Locaux SH-3620 + SH-3320

Instructions

- Aucune documentation n'est permise.
- 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 caractéristiques 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 ;
  - le respect exact de la syntaxe de C++ n'est pas sujet à la correction.
- 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 4 questions.
- À l'exception de l'annexe à la fin du questionnaire, ne détachez aucune feuille.
- Le côté verso peut être utilisé comme brouillon.

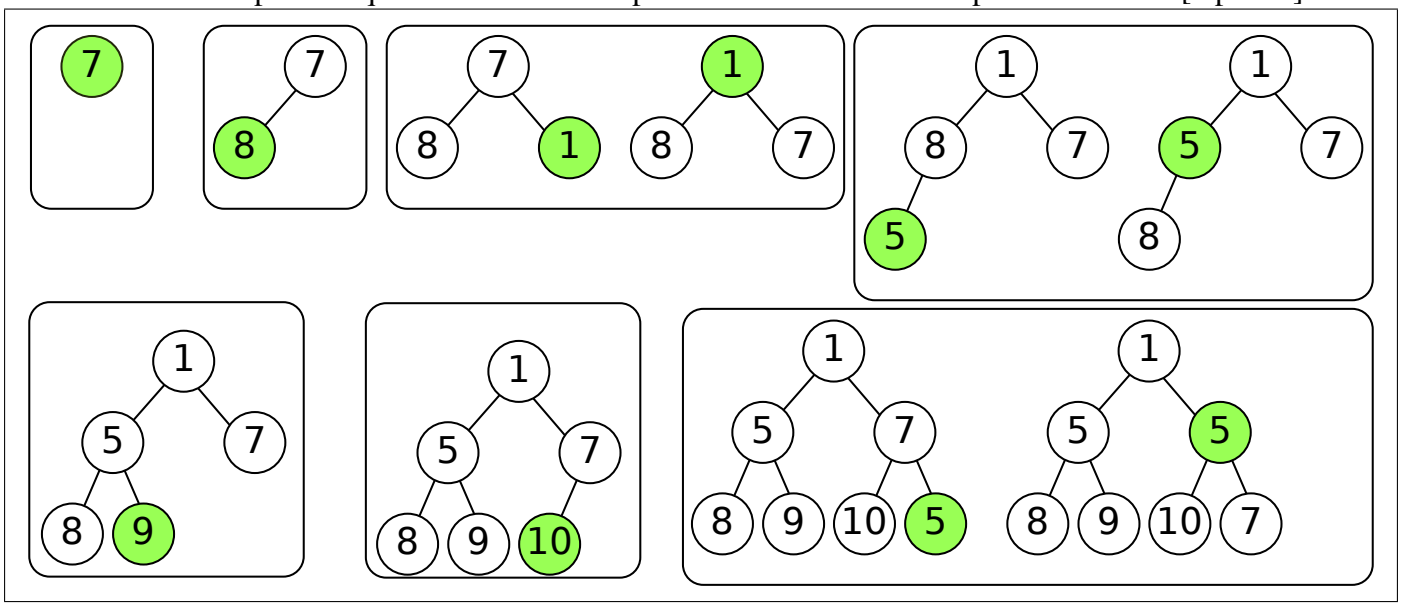
Résultat

Q1		/ 5
Q2		/ 6
Q3		/ 9
Q4		/ 5
Total		/ 25

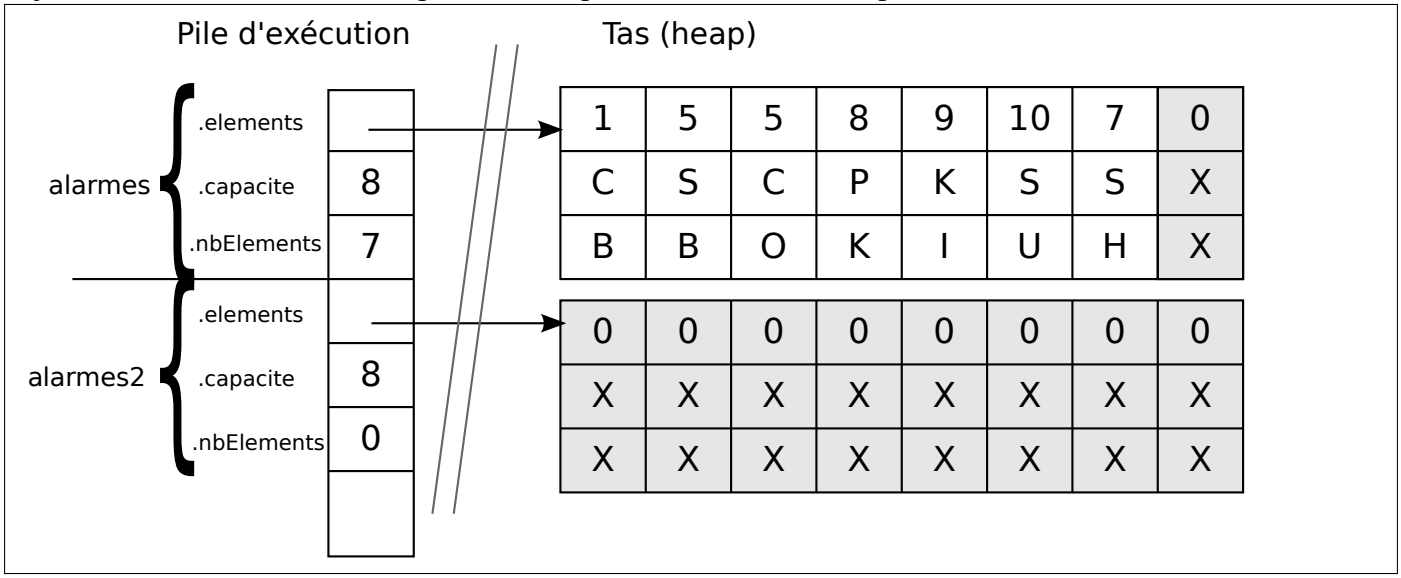
Solutionnaire

1 Monceaux (*Heaps*) et connaissances techniques de C++ [5 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. [2 points]



(b) Référez-vous au code fourni à l'Annexe A (vous pouvez détacher la page 10). Dessinez la représentation en mémoire du programme `main.cpp` rendu à la ligne 32. Soyez aussi précis que possible. Montrez clairement ce qui est sur la pile d'exécution et ce qui est sur le tas (*heap*). Remarquez que les temps des objects `Alarme` sont les mêmes qu'à la sous-question (a) ci-haut ! [3 points]



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

(a) Qu'est-ce qu'une collision dans une table de hachage ? Expliquez en vos propres mots. [1 point]

Une collision survient lorsque deux clés différentes ont la même adresse dispersée réduite. En d'autres mots, pour deux clés k_1 et k_2, on a $hash(k_1) \mod n = hash(k_2) \mod n$, où *hash* est la fonction de hachage et n est le nombre de casiers (*buckets*). Dans ce cas, les 2 clés arrivent dans le même casier. Les collisions doivent être gérées par une stratégie de gestion des collisions.

(b) Quelle est la complexité temporelle de l'insertion dans une table de hachage ? Supposez la gestion de collisions au moyen d'une liste chaînée externe. Supposez n=nombre d'éléments. [2 points]

Cas moyen (0.5 point) : $O(1)$

Pire cas (0.5 point) : $O(n)$

Décrivez le pire cas : Le pire cas survient lorsque toutes les clés sont en collision dans le même casier.

(c) Insérez dans l'ordre les clés 12, 72, 31, 170, 10, 25, 59 et 99 dans une table de hachage. Cette table contient 10 casiers (*buckets*) et n'est jamais redimensionnée. Chaque casier ne peut contenir qu'au plus une clé. Les collisions doivent être gérées à l'aide de la résolution linéaire. [1 point]

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
170	31	12	72	10	25	99	–	–	59

(d) Supposez qu'un de vos collègues souhaite améliorer les performances de son TP3 (Planification de déplacements sur l'île de Montréal) / partie A (génération du chemin le plus court). Ce dernier dit qu'il a utilisé les conteneurs `std::set` et `std::map` (arbres rouge-noir). Il songe maintenant à utiliser les conteneurs `std::unordered_set` et `std::unordered_map` (tables de hachage). Croyez-vous que cela peut améliorer «significativement» les performances de son programme `tp3a` ? Justifiez. [2 points]

Non. La complexité du TP3a est la somme de la complexité de :

— la lecture de la carte en mémoire ;

— la lecture et le traitement des requêtes avec l'algorithme de Dijkstra.

Complexité de lecture de la carte. Lire les lieux peut se faire en $O(n)$. Cependant, si on stock les noms de lieu dans un `map<string, xxxxx>`, cela se fera en $O(n \log n)$. Ensuite, lors de la lecture des routes, il faut la chercher dans un `map<string, xxxxx>` pour chaque nom de lieu d'une route. Ainsi, lire m arêtes (segments de route) coûte $O(m \log n)$. Donc, la lecture de la carte coûte $O((m+n)\log n)$. Généralement $m > n$, donc $O(m \log n)$.

Complexité du traitement des requêtes. La complexité de l'algorithme de Dijkstra est d'au moins $O(m \log n)$ (selon l'implémentation), où n est le nombre de lieux (sommets) et m le nombre d'arêtes (segments de route). Chaque requête coûte aussi $O(n)$ (si on n'a pas de *kd-tree*) pour trouver le lieux le plus près de chaque coordonnées d'origine et de destination. Donc, $O(n + m \log n) = O(m \log n)$ pour chaque requête. Si on a r requêtes : $O(r \cdot m \log n)$.

Donc, la complexité globale du TP3a est de $O(m \log n + r \cdot m \log n) = O(r \cdot m \log n)$.

L'unique endroit où les tables de hachages peuvent être utiles est le dictionnaire des noms de lieux. Donc, on peut espérer réduire le temps de lecture à $O(n+m) = O(m)$. Toutefois, la complexité de Dijkstra n'en sera pas réduite et continuera à dominer le temps de calcul. La complexité globale du TP3a restera de $O(r \cdot m \log n)$.

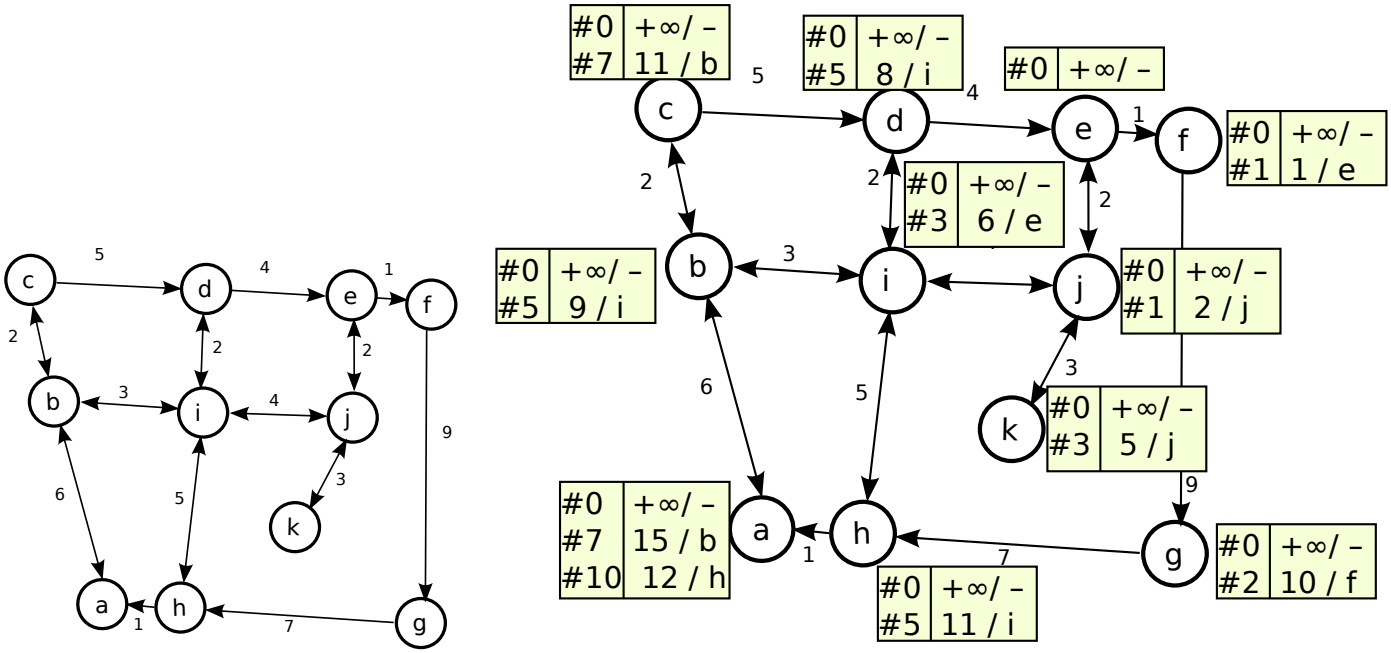
En conclusion, le programme sera certes plus rapide, mais pas «significativement» plus rapide.

* * *

Des explications plus brèves sont aussi acceptées.

3 Graphes [9 points]

Aux sous-questions (a) à (d), considérez le graphe ci-dessous. En (a) et (b), les arêtes sortantes d'un sommet 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 **f**. [1 point]

f, g, h, a, b, c, d, e, j, i, k

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

i, b, d, h, j, a, c, e, k, f, g

(c) Simulez l'algorithme de Dijkstra pour calculer le plus court chemin de e à a. 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]

	choix	a	b	c	d	e	f	g	h	i	j	k	File prioritaire
#0	(init)	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	e
#1	e	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0	$1/e$	$+\infty$	$+\infty$	$+\infty$	$2/e$	$+\infty$	f, j
#2	f	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0	$1/e$	$10/f$	$+\infty$	$+\infty$	$2/e$	$+\infty$	j,g
#3	j	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0	$1/e$	$10/f$	$+\infty$	$6/j$	$2/e$	$5/j$	k,i,g
#4	k	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0	$1/e$	$10/f$	$+\infty$	$6/j$	$2/e$	$5/j$	i,g
#5	i	$+\infty$	$9/i$	$+\infty$	$8/i$	0	$1/e$	$10/f$	$11/i$	$6/j$	$2/e$	$5/j$	d,b,g,h
#6	d	$+\infty$	$9/i$	$+\infty$	$8/i$	0	$1/e$	$10/f$	$11/i$	$6/j$	$2/e$	$5/j$	b,g,h
#7	b	$15/b$	$9/i$	$11/b$	$8/i$	0	$1/e$	$10/f$	$11/i$	$6/j$	$2/e$	$5/j$	g,c,h,a
#8	g	$15/h$	$9/i$	$11/b$	$8/i$	0	$1/e$	$10/f$	$11/i$	$6/j$	$2/e$	$5/j$	c,h,a
#9	c	$15/h$	$9/i$	$11/b$	$8/i$	0	$1/e$	$10/f$	$11/i$	$6/j$	$2/e$	$5/j$	h,a
#10	h	$12/h$	$9/i$	$11/b$	$8/i$	0	$1/e$	$10/f$	$11/i$	$6/j$	$2/e$	$5/j$	a
#11	**a**	$12/h$	$9/i$	$11/b$	$8/i$	0	$1/e$	$10/f$	$11/i$	$6/j$	$2/e$	$5/j$	–

Chemin : $\langle (e,j),(j,i),(i,h),(h,a) \rangle$.

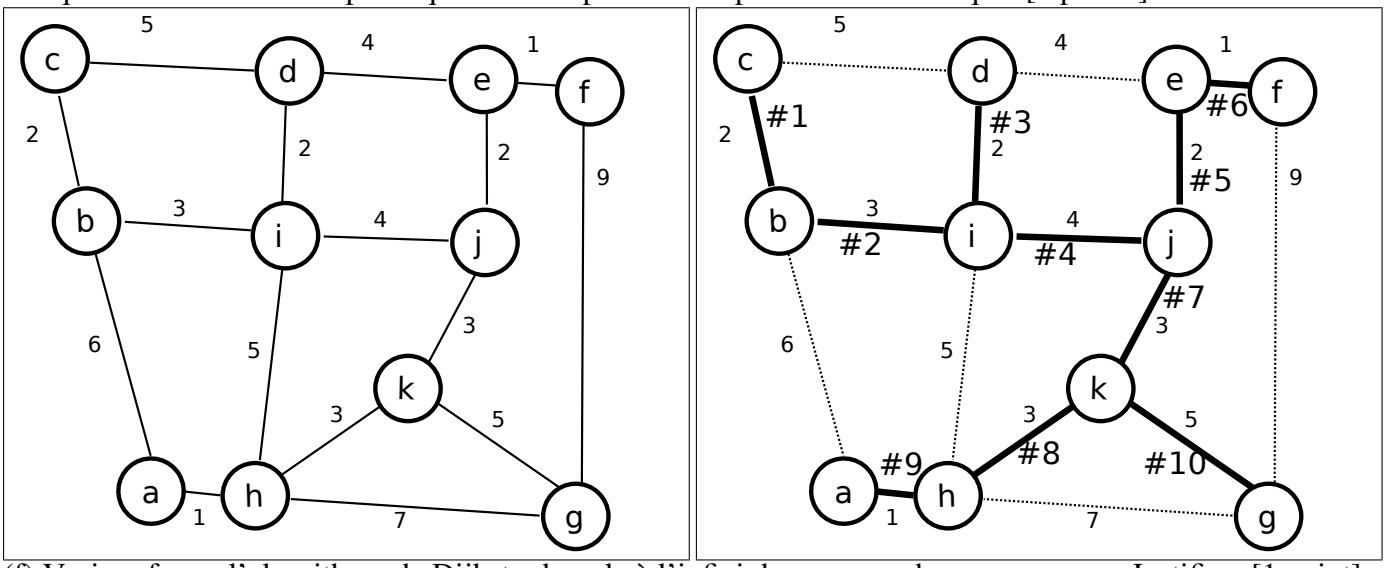
(d) Dans un graphe, le plus court chemin reliant une paire de sommets n'est pas nécessairement unique. Par exemple, dans le graphe à la page précédente, il existe deux chemins optimaux de b à e : $\langle (b,i),(i,j),(j,e)\rangle$ et $\langle (b,i),(i,d),(d,e)\rangle$. Expliquez comment adapter l'algorithme de Dijkstra pour vérifier l'unicité du chemin le plus court. En d'autres mots, l'algorithme doit calculer un booléen qui doit être mis à `true` si et seulement si la solution optimale est unique. [2 points]

Dans l'algorithme de Dijkstra, quand on visite un nœud x, on itère sur les arêtes sortantes $e = (x,y)$. Pour chaque arête sortante $e = (x,y)$, on vérifie si on a trouvé un meilleur chemin vers le sommet d'arrivée y. Cela se teste généralement avec une inégalité stricte `if(D[x]+poids(e)<D[y])`. Pour vérifier si une solution est non unique, on peut faire un test supplémentaire avec un test d'égalité : `if(D[x]+poids(e)<D[y]){noter un seul parent pour` y $\cdots$`}`**`if(D[x]+poids(e)==D[y]) { noter multiple parents pour`** y**`}`**. En d'autres mots, pour se rendre à y, passer par x ou par l'actuel sommet $P[y]$ est équivalent. Enfin, une solution est unique si et seulement chaque sommet sur le chemin n'a qu'un seul parent possible.

* * *

Des explications plus brèves peuvent aussi être acceptées.

(e) Calculez l'arbre de recouvrement minimal du graphe suivant en utilisant l'algorithme de Prim-Jarnik. Indiquez clairement les étapes requises. La réponse n'est pas forcément unique. [2 points]



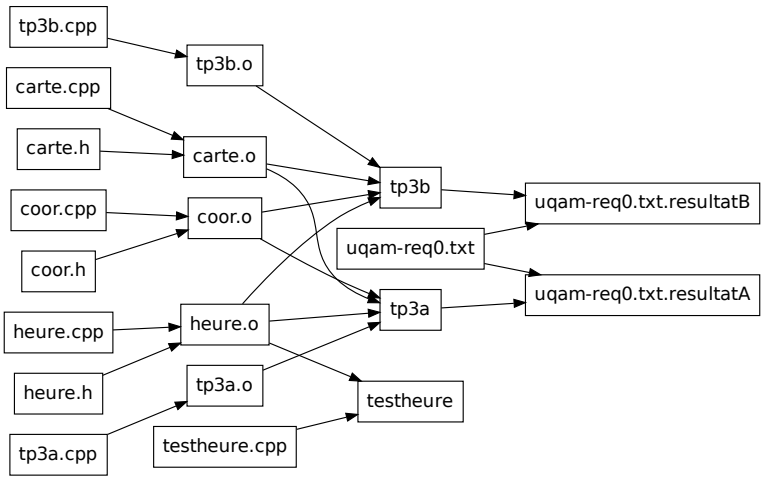
(f) Vrai ou faux : l'algorithme de Dijkstra boucle à l'infini dans un graphe non connexe. Justifiez. [1 point]

Faux. L'algorithme de Dijkstra n'a aucun problème avec les graphes non connexes. Dans le pire cas, l'algorithme de Dijkstra visitera toute la composante connexe accessible depuis le nœuds de départ. Si un nœud est inaccessible, sa valeur D restera à $+\infty$. L'algorithme peut aisément arrêter dès qu'il rencontre une valeur D à $+\infty$. Si les sommets ne sont pas tous insérés dans la file prioritaire dès le départ, aucun sommet d'inaccessible (donc hors de la composante connexe contenant le sommet de départ) ne sera ajouté dans la file prioritaire. Pour conclure, l'algorithme ne peut boucler à l'infini.

4 Résolution d'un problème [5 points]

Un fichier `Makefile` décrit comment construire un projet à partir de ses fichiers sources. Pour simplifier le problème, considérons une syntaxe simplifiée de `Makefile`. Un fichier `Makefile` spécifie une liste de cibles (fichiers) à construire. Chaque cible est spécifiée sur deux lignes. Sur la première ligne, on retrouve le nom de fichier de la cible, un deux-points (`:`) et la liste des fichiers requis (dépendances). Sur la deuxième ligne, débutant par une tabulation, on retrouve la commande à exécuter pour construire la cible.
Voici ci-bas à gauche un exemple de fichier `Makefile` pour le TP3. Un `Makefile` peut être représenté à l'aide d'un graphe orienté, où les sommets sont des fichiers et les arêtes expriment les relations de dépendance et/ou d'ordonnancement. La figure ci-dessous à droite montre le graphe pour le `Makefile` à sa gauche (le sens des arcs exprime des relation d'ordonnancement, l'inverse de dépendance).

```
1  tp3a : carte.o coor.o heure.o tp3a.o
2     g++ −o tp3a carte.o coor.o heure.o tp3a.o # c1
3  tp3b : carte.o coor.o heure.o tp3b.o
4     g++ −o tp3b carte.o coor.o heure.o tp3b.o # c2
5  tp3a.o : tp3a.cpp
6     g++ −o tp3a.o tp3a.cpp # c3
7  tp3b.o : tp3b.cpp
8     g++ −o tp3b.o tp3b.cpp # c4
9  carte.o : carte.h carte.cpp
10    g++ −o carte.o carte.cpp # c5
11 coor.o : coor.h coor.cpp
12    g++ −o coor.o coor.cpp # c6
13 heure.o : heure.h heure.cpp
14    g++ −o heure.o heure.cpp # c7
15 uqam−req0.txt.resultatA : tp3a uqam−req0.txt
16    ./tp3a ... uqam−req0.txt>uqam−req0.txt.resultatA # c8
17 uqam−req0.txt.resultatB : tp3b uqam−req0.txt
18    ./tp3b ... uqam−req0.txt>uqam−req0.txt.resultatB # c9
19 testheure : testheure.cpp heure.o
20    g++ −o testheure testheure.cpp heure.o #c10
```



On vous demande d'écrire un programme `make`. Lisez la sous-question (b) avant de répondre à (a).
(a) Complétez la représentation de la classe Makefile pour stocker le contenu d'un `Makefile`. [2 points]

```
1  class Makefile{
2    public:
3      list<string> getSequenceCommandes() const; // option (b1)
4      list<list<string> > getSequenceGroupesCommandes(int n) const; // option (b2)
5    private: // Completez uniquement la representation
6      struct Cible{
7        string commande;  // la commande pour constuire la cible
8        set<string> dependances; // cibles devant etre construites AVANT cette cible
9        set<string> successeurs;  // cibles neccessitant cette cible
10       mutable bool b1_construite; // La cible a ete construite ?
11       mutable enum {Non, Attente, Oui} b1_etat; // La cible a ete construite ?
12       mutalbe int b2_ndr; // nombre de dependances restantes avant d'etre eligible.
13       mutalbe int b2_priorite; // priorite=distance maximale avec une cible finale.
14     };
15     map<string, cible> cibles;
16   friend istream& operator >> (istream&, Makefile&);
17 };
```

(b) Choisissez une seule fonction entre (b1) et (b2). Implémentez-la en C++ ou en pseudo-code. [3 points]

(b1) [maximum 1.5 point] La fonction `list<string> Makefile::getSequenceCommandes()` retourne une séquence correctement ordonnée de commandes à exécuter pour construire le projet. Par exemple, $\langle c3, c4, c5, c6, c7, c1, c2, c8, c9, c10\rangle$ est une liste correcte pour l'exemple, où c3="g++ -o tp3a.o tp3a.cpp", c4="g++ -o tp3b.o tp3b.cpp", etc. Indice : parcours en profondeur.

L'astuce est de faire une recherche en profondeur. Avant d'exécuter une commande pour construire une cible, on doit construire les cibles dépendantes avant. Donc, on parcourt récursivement les dépendances.

```
1  list<string> Makefile::getSequenceCommandes() const{
2    list<string> commandes;
3    for(map<string, Cible>::const_iterator i=cibles.begin();i!=cibles.end();++i)
4      i->second.b1_construite = false;
5    for(map<string, Cible>::const_iterator i=cibles.begin();i!=cibles.end();++i)
6      getSequenceCommandes(i->second, commandes);
7    return commandes;
8  }
9  // Fonction recursive privee effectuant une sorte de parcours en profondeur
10 void Makefile::getSequenceCommandes(const Cible& cible, list<string>& commandes)
       const{
11   if(cible.b1_construite) return; // Deja construite, pas besoin de continuer.
12   if(cible.commande.empty()) return; // Un fichier source fourni (aucune cmd).
13   for(set<string>::const_iterator
       i=cible.dependances.begin();i!=cible.dependances.end();++i)
14     getSequenceCommandes(cibles.at(*i), commandes); // appel en profondeur
15   commandes.push_back(cible.commande);
16   cible.b1_construite = true; // marquer construite
17 }
```

Si on souhaite détecter les cycles, on peut introduire un état de calcul :

```
1  list<string> Makefile::getSequenceCommandes() const{
2    list<string> commandes;
3    for(map<string, Cible>::const_iterator i=cibles.begin();i!=cibles.end();++i)
4      i->second.b1_etat = Cible::Non;
5    for(map<string, Cible>::const_iterator i=cibles.begin();i!=cibles.end();++i)
6      getSequenceCommandes(i->second, commandes);
7    return commandes;
8  }
9  void Makefile::getSequenceCommandes(const Cible& cible, list<string>& commandes)
       const{
10   if(cible.b1_etat==Cible::Oui) return; // Deja construite
11   if(cible.b1_etat==Cible::Attente) return; // Detection cycle!
12   if(cible.commande.empty()) return; // Un fichier source fourni (aucune cmd).
13   cible.b1_etat=Cible::Attente;
14   for(set<string>::const_iterator
       i=cible.dependances.begin();i!=cible.dependances.end();++i)
15     getSequenceCommandesV2(cibles.at(*i), commandes); // appel en profondeur
16   commandes.push_back(cible.commande);
17   cible.b1_construite = Cible::Oui; // marquer construite
18 }
```

(b2) [maximum 3 points] Si on dispose de plusieurs unités de calcul (processeurs ou cœurs), certaines cibles peuvent être construites en parallèle. Pour simplifier le problème, on suppose que toutes les commandes ont la même durée. La fonction `list<list<string> > Makefile::getSequenceGroupesCommandes(int n)` retourne une séquence correctement ordonnée de groupes de commandes à exécuter pour construire le projet sur une machine disposant de n unités de calcul. Chaque groupe contient i commande(s) à exécuter en parallèle tel que $1 \leq i \leq n$. Dans l'exemple, si $n = 2$, alors la fonction pourrait retourner $\langle\langle c3, c4\rangle, \langle c5, c7\rangle, \langle c6, c10\rangle, \langle c1, c2\rangle, \langle c8, c9\rangle\rangle$. Les commandes c3 et c4 s'exécutent en parallèle, ensuite c5 et c7, etc.

La solution la plus simple consiste à maintenir un compteur qui compte le nombre de dépendances restantes à constuire. Dès que ce compteur tombe à zéro, cela signifie que la cible est maintenant prête à être construite. Cette technique nécessite des arêtes bidirectionnelles indiquant les relations de dépendances et d'ordonnancement. Cependant, cette technique ne donne pas toujours un résultat optimal, mais ça fonctionne relativement bien.

```
1  list<list<string> > Makefile::getSequenceGroupesCommandes(int n) const{
2    queue<string> pretes; // cibles pretes a construire
3    for(map<string, Cible>::const_iterator i=cibles.begin();i!=cibles.end();++i)
4      if((i->second.b2_ndr = i->second.dependances.size())==0)
5        pretes.push(i->first);
6    list<list<string> > groupes; // le resultat a retourner
7    while(!pretes.empty()){
8      list<string> commandes; // commandes du present groupe
9      list<string> nouvpretes;
10     while(!pretes.empty() && commandes.size()<n){
11       const Cible& cible = cibles.at(pretes.front());
12       pretes.pop();
13       if(!cible.commande.empty()) commandes.push_back(cible.commande);
14       set<string>::const_iterator i=cible.successeurs.begin();
15       for(;i!=cible.successeurs.end();++i)
16         if(--(cibles.at(*i).b2_ndr)==0)
17           if(cible.commande.empty())
18             pretes.push(*i);
19           else nouvpretes.push_back(*i);
20     }
21     for(list<string>::iterator i=nouvpretes.begin();i!=nouvpretes.end();++i)
22        pretes.push(*i);
23     groupes.push_back(commandes);
24   }
25   return groupes;
26 }
```

Pour améliorer le résultat, on doit commencer par construire les cibles sur ce qu'on appelle le **chemin critique**. Le chemin critique est le plus long chemin de dépendances. On commence par calculer une priorité pour chaque cible, la priorité étant la distance maximale avec une cible finale.

```
1  struct RefCible{
2    RefCible(int p=0,string n="") : priorite(p), nom(n){}
3    int priorite;
4    string nom;
5    bool operator<(const RefCible& o)const{return priorite<o.priorite;}
6  };
7  list<list<string> > Makefile::getSequenceGroupesCommandesV2(int n) const{
8    list<string> commandes;
9    for(map<string, Cible>::const_iterator i=cibles.begin();i!=cibles.end();++i)
10     i->second.b2_priorite = -1;
11   for(map<string, Cible>::const_iterator i=cibles.begin();i!=cibles.end();++i)
12     calculerPriorite(i->second);
13   priority_queue<RefCible> pretes;
14   for(map<string, Cible>::const_iterator i=cibles.begin();i!=cibles.end();++i)
15     if((i->second.b2_ndr=i->second.dependances.size())==0)
16       pretes.push(RefCible(i->second.b2_priorite, i->first));
17   list<list<string> > groupes; // le resultat a retourner
18   while(!pretes.empty()){
19     list<string> commandes, nouvpretes;
20     while(!pretes.empty() && commandes.size()<n){
21       const Cible& cible = cibles.at(pretes.top().nom);
22       pretes.pop();
23       if(!cible.commande.empty()) commandes.push_back(cible.commande);
24       for(set<string>::iterator i=cible.successeurs.begin();i!=cible.successeurs.end();++i)
25         if(--(cibles.at(*i).b2_ndr)==0)
26           if(cible.commande.empty())
27             pretes.push(RefCible(cibles.at(*i).b2_priorite, *i));
28           else nouvpretes.push_back(*i);
29     }
30     for(list<string>::iterator i=nouvpretes.begin();i!=nouvpretes.end();++i)
31        pretes.push(RefCible(cibles.at(*i).b2_priorite, *i));
32     groupes.push_back(commandes);
33   }
34   return groupes;
35 }
36 int Makefile::calculerPriorite(const Cible& c) const{
37   if(c.b2_priorite==-2) exit(1); // dependance cyclique
38   c.b2_priorite = -2;
39   int nouvellepriorite=0;
40   for(set<string>::const_iterator i=c.successeurs.begin();i!=c.successeurs.end();++i){
41     int priorite_i = calculerPriorite(cibles.at(*i));
42     if(priorite_i+1>nouvellepriorite)
43       nouvellepriorite = priorite_i + 1;
44   }
45   c.b2_priorite = nouvellepriorite;
46   return nouvellepriorite;
47 }
```

Annexe A pour la Question 1

Cette page peut être détachée.

```
/*** tableau.h ***/
template <class T> class Tableau{
public:
  Tableau(int capacite_initiale=8);
  Tableau(const Tableau&);
  ~Tableau();

  void ajouter(const T& element);
  bool vide() const;
  void vider();
  int  taille() const;
  T& operator[] (int index);
  const T& operator[] (int index)
   const;
  Tableau<T>& operator=(const
   Tableau<T>&);
  bool operator==(const
   Tableau<T>&) const;

private:
  T* elements;
  int capacite;
  int nbElements;
};
// ...
```

```
/*** monceau.h ***/
#include "tableau.h"
template <class T> class Monceau {
  public:
    void inserer(const T&);
    void enleverMinimum();
    void enleverMinimum(T& sortie);
    const T& minimum() const;
    bool estVide() const;

  private:
    Tableau<T> valeurs;
    void remonter(int indice);
    void descendre(int indice);
};
```

```
/*** main.cpp ***/
#include "monceau.h"
class Alarme{
public:
 Alarme(int t=0, const char* c="XX");
 Alarme(const Alarme&);
 bool operator<(const Alarme&)const;
private:
 int temps; // "date" de l'alarme
 char[2] code;
};
Alarme::Alarme(int t, const char* c)
: temps(t) {
 strncpy(code,c,2); //Voir [1] ci-bas
}
Alarme::Alarme(const Alarme& a)
: temps(a.temps) {
   strncpy(code,a.code,2); }
bool Alarme::operator<(const Alarme&
   a)const{
  return temps < a.temps;
}

int main(){
  Monceau<Alarme> alarmes;
  Monceau<Alarme> alarmes2;
  alarmes.inserer(Alarme(7, "SH"));
  alarmes.inserer(Alarme(8, "PK"));
  alarmes.inserer(Alarme(1, "CB"));
  alarmes.inserer(Alarme(5, "SB"));
  alarmes.inserer(Alarme(9, "KI"));
  alarmes.inserer(Alarme(10, "SU"));
  alarmes.inserer(Alarme(5, "CO"));
  // Dessinnez l'etat rendu ici
  alarmes2 = alarmes;
  while(!alarmes.estVide()){
    // ...
    alarmes.enleverMinimum();
  }
  return 0;
}
```

[1] La fonction **C** `strncpy(char* dest, const char* src, int n)` copie jusqu'à `n` caractères de la chaîne `src` vers la chaîne `dest`.