

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.

Résultat

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

Solutionnaire

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	Commentaire	m[0]	m[1]	m[2]	m[3]	m[4]	m[5]	m[6]
#0	Init. vide	–	–	–	–	–	–	–
#1	Ajout 6 fin (aucun déplacement)	6						
#2	Ajout 7 fin (aucun déplacement)	6	7					
#3	Ajout 2 fin	6	7	2				
#4	Échange	2	7	6				
#5	Ajout 4 fin	2	7	6	4			
#6	Échange	2	4	6	7			
#7	Ajout 8 fin (aucun déplacement)	2	4	6	7	8		
#8	Ajout 3 fin	2	4	6	7	8	3	
#9	Échange	2	4	3	7	8	6	



Correction:

- Chaque erreur enlève 0.3 point jusqu'à 0/1.5.

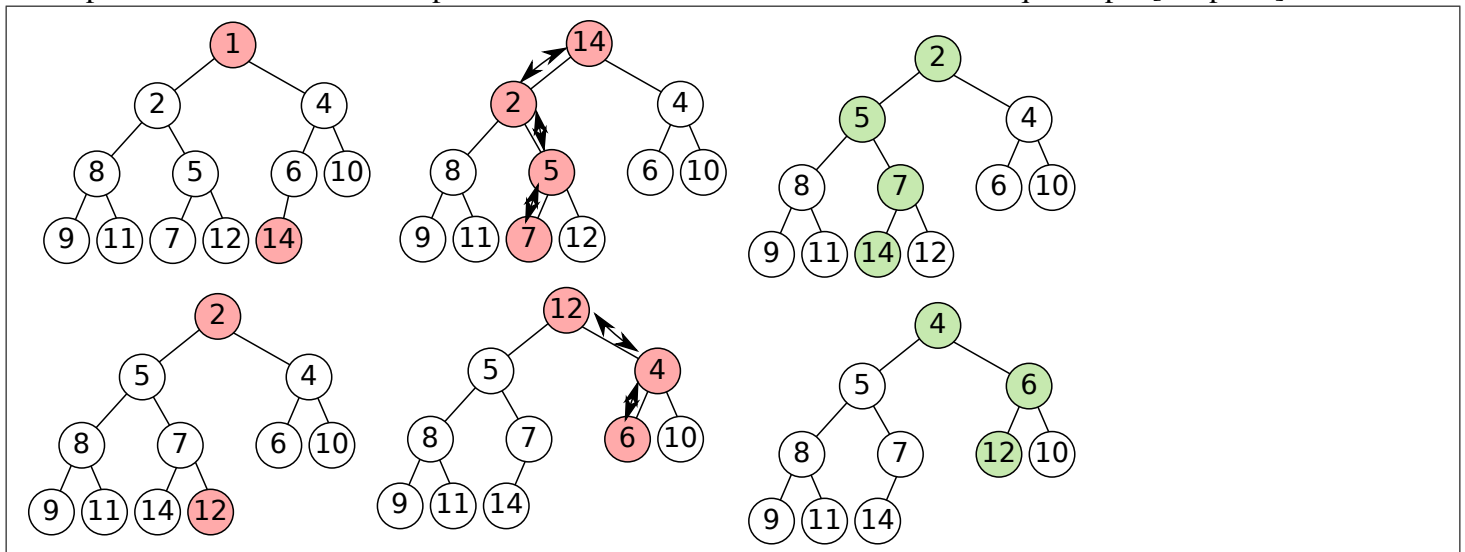
- Types d'erreurs :

(1) ne pas ajouter le nouvelle élément à la toute fin ;

(2) échange manquant ou avec mauvais élément.

- Chaque étape est comparée avec l'état du monceau à l'étape précédente (une erreur au début doit être pénalisée qu'une seule fois).

(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]



Correction:

1.5/1.5 : L'arbre final est correct et on peut voir toutes les étapes.

1.2/1.5 : L'arbre final est correct, quelques étapes ne sont pas claires, mais la réponse démontre une bonne compréhension de l'étudiant.

0.5/1.5 : Seul l'arbre final est dessiné, aucune étape intermédiaire.

1.5 – 0.3 × [nberreursoutapesnonmontres] : Tout autre cas.

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

```


1 #include <list>
2 template <class T> class Monceau<T>{
3     public:
4         void inserer(const T&);
5         T    enleverpremier();
6         bool estVide() const;
7     private:
8         std::list<T> valeurs; };

```

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

Non, c'est un mauvais choix.

Dans un monceau, lors de l'insertion et de l'enlèvement, les opérations de descentes et de remontement ont besoin d'accéder efficacement au parent ou aux enfants d'un noeud à partir de son indice. Une liste ne permet pas d'accès direct (accès aléatoire), ces opérations se font en temps $O(n)$. Un tableau est une meilleure structure car elle permet l'accès direct en temps $O(1)$.

 Correction: 0/1 : Si réponse = «oui», et ce, peu importe la justification.

0.4/1 : Si réponse = «non», mais sans justification.

1/1 : Si réponse = «non» et la justification réfère aux deux faits suivants : (1) l'insertion ou l'enlèvement a besoin de consulter le parent ou enfants à partir d'indices (positions); (2) la liste ne permet pas un accès en temps constant $O(1)$ à l'élément à l'indice i dans une liste.


0.8/1 : Si réponse = «non» et la justification réfère à un seul fait ci-haut.

0.1 à 0.5 point peut être enlevé si la justification contient des contradictions ou des faussetés.

Exemple : «Non, parce qu'on ne peut accéder en temps constant au i -ème élément dans une liste chaînée (Ok jusqu'ici). Il faudrait plutôt utiliser un arbre AVL (-0.5).

(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]


`std::priority_queue`

 Correction: Il faut la réponse exacte. `std::` est optionnel.

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


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

Serveur de noms de domaine (DNS). Dans un serveur DNS, un dictionnaire associe un nom de domaine (string) à un IP (entier).

 Correction: L'application doit être évidente (ex : DNS) ou avoir une brève explication disant où la table de hachage est utile.

(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	$O(1)$	$O(n)$
Retourner la valeur associée à une clé	$O(1)$	$O(n)$


 Correction: 0.25 point / erreur.

Considérez les fichiers source à l'annexe A (page 9).

(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]

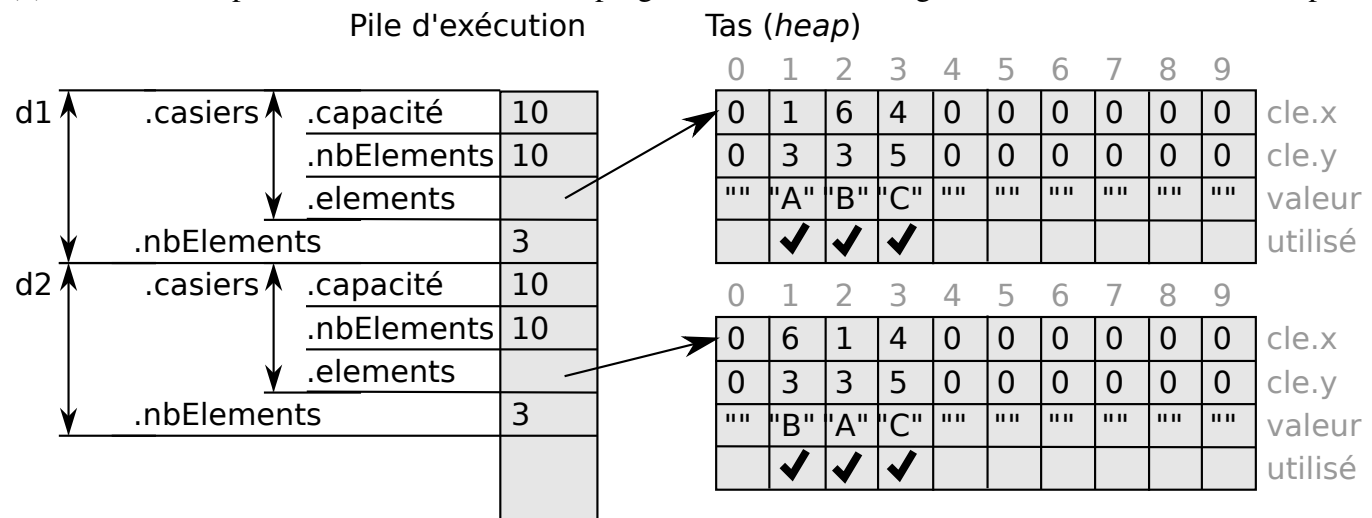
```


1  template <class K, V>
2  V& Dictionnaire::operator[](const K& cle) {
3      int h = cle(); // K::operator int() retourne la valeur de hachage
4      // if(nbEntrees+1 < (int)(0.25 * casiers.taille())) agrandir();
5      while(true) {
6          Casier& c = casiers[h++ % casiers.taille()];
7          if(!c.utilise){ c.utilise=true; c.cle=cle; nbEntrees++; return c.valeur; }
8          if(c.cle==cle) return c.valeur;
9      } // boucle infinie impossible si nbEntrees<casiers.taille()
10 }
```

 Correction: Éléments vérifiées :

- Réduction/compression d'adresse (0.5). Ex : $h = h \text{ modulo } 10$ ou $h = h \text{ modulo } \text{casiers.taille}()$.
- Si casier non utilisé, faire une insertion. Incrémenter nbEntrees (0.2).
- Si casier utilisé, comparer le clé (0.5).
- Retourne référence sur la valeur associée à la clé (0.4).
- Essais linéaires (passage au suivant) si utilisé et clé différente (0.4).

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

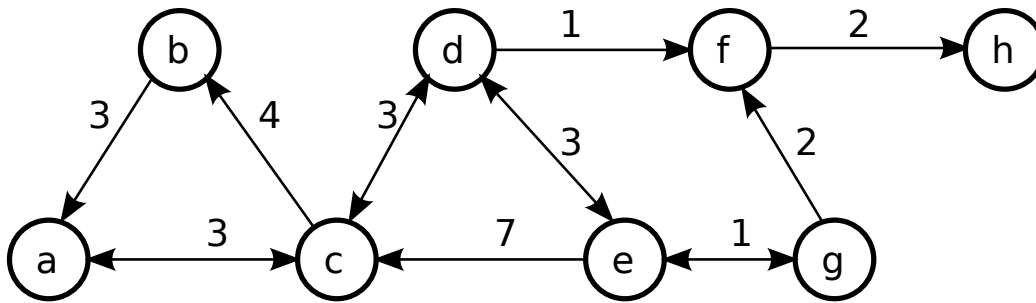


 Correction: Éléments vérifiées :

- Les casiers sont sur le tas (heap) et non la pile d'exécution (0.4) et reliés par deux pointeurs sur la pile d'exécution (0.2).
- Les variables d1.nbEntrees et d2.nbEntrees sont sur la pile =3 (0.2).
- Il y a 3 casiers utilisées pour A, B et C dans d1 et d2 (0.2).
- Les clés sont dans les bons casiers (0.6).
- Les casiers stockent les clés (Foo : :x et Foo : :y) et non la valeur de hachage (0.2).
- Les valeurs bool utilisées sont cohérentes aux casiers utilisées (0.2).
- Les clés des casiers inutilisés sont à (0,0) et les valeurs à "" (0.2).
- D'autres éléments peuvent être vérifiées.

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]

$\langle b, a, c, d, e, g, f, h \rangle$

Correction: 1, 0.5 ou 0

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

$\langle b, a, c, d, e, f, g, h \rangle$

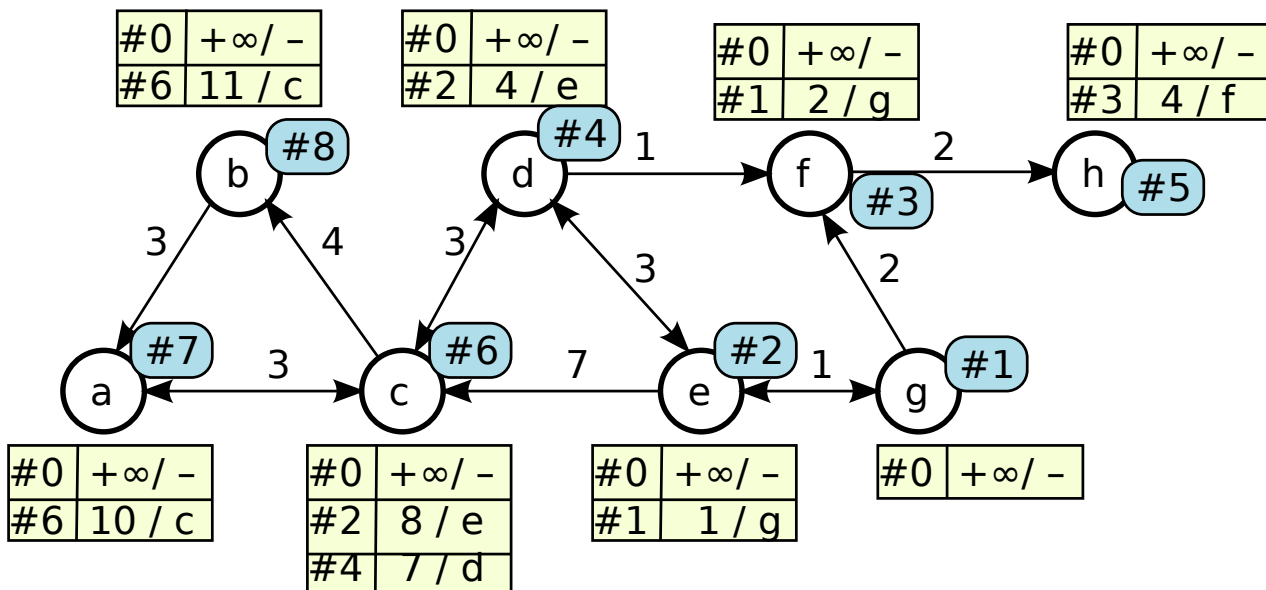
Correction: 1, 0.5 ou 0

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

$\{a, b, c, d, e, g\}, \{f\}, \{h\}$

Correction: 1, 0.5 ou 0

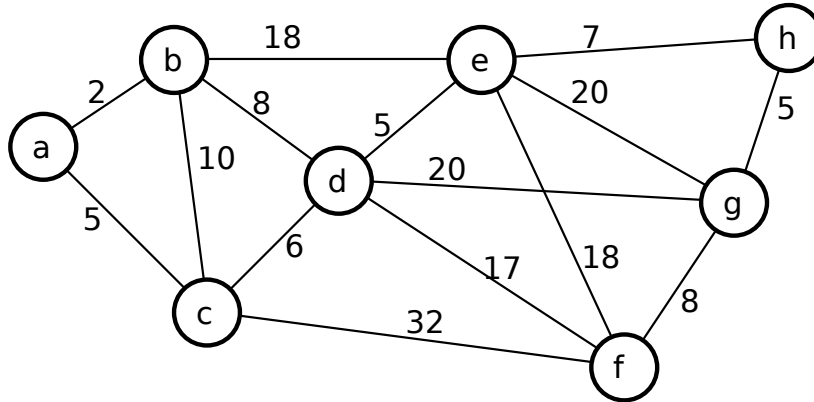
(d) Simulez l'algorithme de Dijkstra pour calculer le plus court chemin de **g** à **c**. Il y a plusieurs façons de présenter votre réponse. L'important est de démontrer votre compréhension de l'algorithme. Les éléments clés à présenter sont l'ordre de visite des sommets et les valeurs *Dist* et *Parent*.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



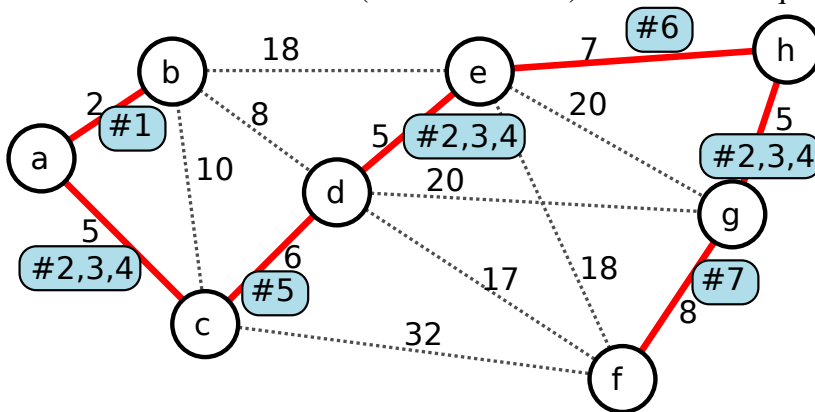
Correction: *Éléments vérifiées :*

- Initialisation valeur *D* à $+\infty$ sur tous les sommets ou uniquement le sommet départ à $+\infty$ ou zéro (0) (0.3).
- Ordre de visite des sommets. La dernière étape peut s'arrêter sur **c**. Il doit donc y avoir au moins 6 étapes (0.5).
- Mise à jour des valeurs *D*/*Parent* à chaque étape quand nouveau meilleur chemin existe (1.2).

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]

Vrai, la solution est unique. En simulant l'algorithme de Kruskal, lorsqu'on a le choix entre plusieurs arêtes, toutes ces arêtes se retrouvent dans la solution. Étapes : #1 choix unique $(a,b,2)$; #2,#3, #4 $(a,c,5)$, $(d,e,5)$, $(d,e,5)$ (peut importe l'ordre, on doit tous les choisir); #5 choix unique $(c,d,6)$; #6 choix unique $(e,h,7)$; #7 choix unique $(g,f,8)$. Il n'arrive aucun cas où qu'une arête candidate est non choisie.

(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]

Non, c'est une mauvaise idée si la carte est trop grande.

Justification courte. La complexité spatiale de Floyd-Warshall est de $O(n^2)$ où n est le nombre de noeuds. La quantité de mémoire requise croît rapidement en fonction du nombre de noeuds.

Exemple. Dans une carte, le nombre de sommets peut être relativement grand. Par exemple, la carte de Montréal fournie avec le TP3 contient environ $n = 81000$ noeuds. Les tables 2D de distances et directions prennent donc $81000^2 = 6.561E + 09$ entrées. Pour la table de distance, chaque entrée consomme `sizeof(float)=4` à `sizeof(float)=8` octets. Donc, $4 \times 6.561E + 09 \approx 26$ Go.

La complexité temporelle est de $O(n^3)$. On pourrait poursuivre l'analyse si c'est vraiment rentable. Il faudrait beaucoup de requêtes (électeurs) pour que ce soit rentable.

5 Résolution d'un problème [4 points]

Lisez la mise en contexte à l'Annexe C (page 11).

Si vous désirez apporter des changements à la représentation de la classe Carte, faites-le ci-dessous.

```
1 class Carte{ /* ... */
2     struct Lieu{
3         bool station;
4         map<string, double> voisins;
5         int nv; // nombre de visites du sommet sur chemin courant
6     };
7     map<string, Lieu> lieux;
8     list<string> chemin; // alternative : passer un chemin par reference en parametre
9 };
```

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

```
1 bool Carte::existeChemin(string o, string d) const{
2     return existeChemin(o, d, 20.0);
3 }
4 bool Carte::existeChemin(string o, string d, double carb){
5     if(carb<0) return false; // manque de carburant : arreter
6     if(o==d) // rendu a destination ?
7         return true; // signaler a l'appelant qu'un chemin faisable existe
8     const Lieu& l = lieux[o]; // lieux.at(o); // ref sur lieux courant
9     if(l.nv>=2) return false; // limiter le nombre de visite a 2
10    l.nv++; // compter le nombre de fois en cours de visite
11    if(l.station) carb=20; // Si le lieu a une station, alors faire le plein
12    // Parcourir les sommets voisins
13    for(map<string,Lieu>::const_iterator i=l.voisins.begin();i!=l.voisins.end();++i)
14        if(existeChemin(i->first, d, carb-i->second))
15            return true; // signaler a l'appelant que chemin faisable existe
16    l.nv--; // decreter le nombre de fois en cours de visite
17    return false;
18 }
```

```
1 list<string> Carte::calculerChemin(string o, string d) const{
2     chemin.clear();
3     calculerChemin(o, d, 20.0);
4     return chemin;
5 }
6 bool Carte::calculerChemin(string o, string d, double carb){
7     if(carb<0) return false; // manque de carburant : arreter
8     if(o==d){ // rendu a destination ?
9         chemin.push_front(d); // commencer construire chemin par la fin
10        return true; // signaler a l'appelant qu'un chemin faisable existe
11    }
12    const Lieu& l = lieux[o]; // ref sur lieux courant
13    if(l.nv>=2) return false; // limiter le nombre de visite a 2
14    l.nv++;
15    if(l.station) carb=20; // Si le lieu a une station, alors faire le plein
16    // Parcourir les sommets voisins
17    for(map<string,Lieu>::const_iterator i=l.voisins.begin();i!=l.voisins.end();++i)
18        if(calculerChemin(i->first, d, carb-i->second)){
19            chemin.push_front(o); // chemin faisable trouve => continuer construire chemin
20            return true; // signaler a l'appelant que chemin faisable existe
21        }
22    l.nv--;
23    return false;
24 }
```


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()`.

```

1  /* dictionnaire.h */
2  template <class K, class V>
3  class Dictionnaire{
4  public:
5      Dictionnaire();
6      V& operator[] (const K& cle);
7      bool operator==(const
          Dictionnaire&) const;
8      int taille() const;
9      // ...
10 private:
11     struct Casier{
12         Casier(){utilise=false;}
13         K cle;
14         V valeur;
15         bool utilise;
16     };
17     //nombre d'entrees cle-valeur
18     int nbEntrees;
19     Tableau<Casier> casiers;
20 };
21
22 template <class K, class V>
23 Dictionnaire<K,V>::Dictionnaire()
24 :casiers(10), nbEntrees(0)
25 {
26 }
```

```

1  /* foo.h */
2  class Foo{
3  public:
4      Foo(int x_=0, int y_=0);
5      Foo(const Foo&);
6      bool operator == (const Foo& f) const;
7      operator int() const;
8  private:
9      int x, y;
10 };
```

```

1  /* foo.cpp */
2  Foo::Foo(int x_, int y_)
3  : x(x_), y(y_) {}
4  Foo::Foo(const Foo& f)
5  : x(f.x), y(f.y) {}
6  bool Foo::operator==(const Foo& f) const
7  {
8      return x==f.x && y==f.y;
9  }
10 // Fonction de hachage
11 Foo::operator int() const{
12     return x*2+y*3;
13 }
```

```

1  /* question2.cpp */
2  #include ...
3  int main(int argc, const char** argv){
4      Dictionnaire<Foo, std::string> d1, d2;
5      d1[Foo(1,3)] = "A";
6      d1[Foo(6,3)] = "B";
7      d1[Foo(4,5)] = "C";
8      d2[Foo(4,5)] = "C";
9      d2[Foo(6,3)] = "B";
10     d2[Foo(1,3)] = "A";
11     // Dessinez representation memoire rendu ici.
12     //bool b = d1==d2;
13     return 0;
14 }
```

Annexe B – Rappel d’algorithmes de graphes

1. RECHERCHEPROFONDEUR($G = (V, E), v \in V$)
2. $v.\text{visité} \leftarrow \text{vrai}$
3. pour toute arête $e \in v.\text{aretesSortantes}()$
4. $w \leftarrow e.\text{arrivee}$
5. si $\neg w.\text{visité}$
6. RechercheProfondeur(G, w)

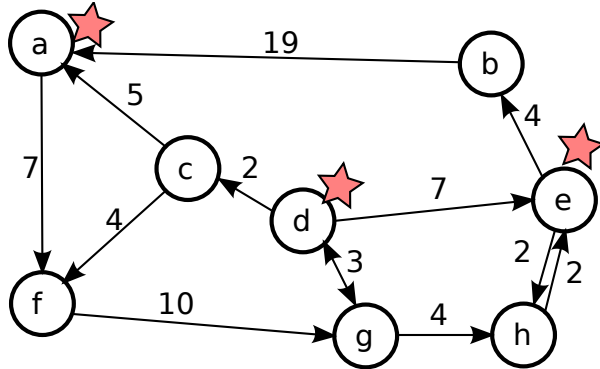
1. DIJKSTRA($G = (V, E), s \in V$)
2. pour tout $v \in V$
3. $\text{distances}[v] \leftarrow +\infty$
4. $\text{parents}[v] \leftarrow \text{indéfini}$
5. $\text{distances}[s] \leftarrow 0$
6. $Q \leftarrow \text{créer FilePrioritaire}(V)$
7. tant que $\neg Q.\text{vide}()$
8. $v \leftarrow v \in Q$ avec la plus petite valeur $\text{distances}[v]$
9. si $\text{distances}[v] = +\infty$ break
10. pour toute arête sortante $e = (v, w)$ depuis le sommet v
11. $d \leftarrow \text{distances}[v] + e.\text{distance}$
12. si $d < \text{distances}[w]$
14. $\text{parents}[w] \leftarrow v$
15. $\text{distances}[w] \leftarrow d$
16. $Q.\text{réduireClé}(w)$
17. retourner $(\text{distances}, \text{parents})$

1. FLOYD_WARSHALL($G = (V, E)$)
2. $\text{distances} \leftarrow \text{créer tableau } |V| \times |V| \text{ initialisé à } +\infty$
3. $\text{directions} \leftarrow \text{créer tableau } |V| \times |V| \text{ initialisé à « ? »}$
4. pour tout $v \in V$
5. $\text{distances}[v][v] \leftarrow 0$
6. $\text{directions}[v][v] \leftarrow v$
7. pour tout $e \in E$
8. $\text{distances}[e.\text{out}][e.\text{in}] \leftarrow e.\text{cout}$
9. $\text{directions}[e.\text{out}][e.\text{in}] \leftarrow e.\text{in}$
10. pour $k = 0$ à $|V| - 1$
11. pour $i = 0$ à $|V| - 1$
12. pour $j = 0$ à $|V| - 1$
13. si $\text{distances}[i][k] + \text{distances}[k][j] < \text{distances}[i][j]$
14. $\text{distances}[i][j] \leftarrow \text{distances}[i][k] + \text{distances}[k][j]$
15. $\text{directions}[i][j] \leftarrow \text{directions}[i][k]$
16. retourner directions

1. KRUSKAL($G = (V, E)$)
2. pour tout sommet $v \in V$
3. $C(v) \leftarrow \text{créer un ensemble } \{v\}$
4. $Q \leftarrow \text{créer FilePrioritaire}<\text{Arête}>(E)$ où la clé est le poids
5. tant que $|E'| < |V| - 1$
6. $e = (v_1, v_2) \leftarrow Q.\text{enleverMinimum}()$
7. si $C(v_1) \neq C(v_2)$
8. ajouter e à E'
9. fusionner $C(v_1)$ et $C(v_2)$ afin que $C(v_1) = C(v_2)$
10. 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 court 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.

```

1 class Carte{
2 public:
3     void ajouterLieu(const string& nom, bool station);
4     void ajouterRoute(const string& n1, const string& n2, double longueur);
5     bool existeChemin(string origine, string destination) const; // 3 points
6     list<string> calculerChemin(string origine, string destination) const; // 4 points
7 private:
8     struct Lieu{
9         bool station; // lieux["a"].station=true; lieux["b"].station=false;
10        map<string, double> voisins; // Ex: lieux["a"].voisins["f"]=7;
11                                   // lieux["c"].voisins["a"]=5; lieux["c"].voisins["f"]=4;
12    };
13    map<string, Lieu> lieux;
14 };

```

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.