

INF3105 – Structures de données et algorithmes

Été 2015 – Examen de mi-session

Éric Beaudry
Département d'informatique
Université du Québec à Montréal

Vendredi 26 juin 2015 – 17h30 à 20h30 (3 heures) – Locaux PK-6605 et PK-6610

Instructions

- Aucune documentation n'est permise, excepté l'aide-mémoire C++ (feuille recto verso).
- 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 ;
 - 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, contient 7 questions et vaut 20 % de la session.
- Ne détachez pas les feuilles du questionnaire, à moins de les brocher à nouveau avant la remise.
- Dans l'entête de l'actuelle page, si vous encerclez le numéro de local où vous vous trouvez présentement, vous aurez un point boni pour avoir lu les instructions.
- Le côté verso peut être utilisé comme brouillon. Des feuilles additionnelles peuvent être demandées au surveillant.

Identification

Nom : Solutionnaire

Résultat

Q1		/ 16
Q2		/ 20
Q3		/ 12
Q4		/ 18
Q5		/ 06
Q6		/ 08
Q7		/ 20
Total		/ 100

1 Connaissances techniques et C++ [16 points]

Pour répondre à cette question, référez-vous au programme fourni à l'Annexe A (page 8). Notez que ce programme se compile sans erreur. Contrairement au TP1, la fonction `distance` n'est pas une fonction membre de la classe `Coordonnee`, mais plutôt une fonction `friend` (amie). Pour les sous-questions (b) et (d), considérez que le programme reçoit la chaîne suivante :

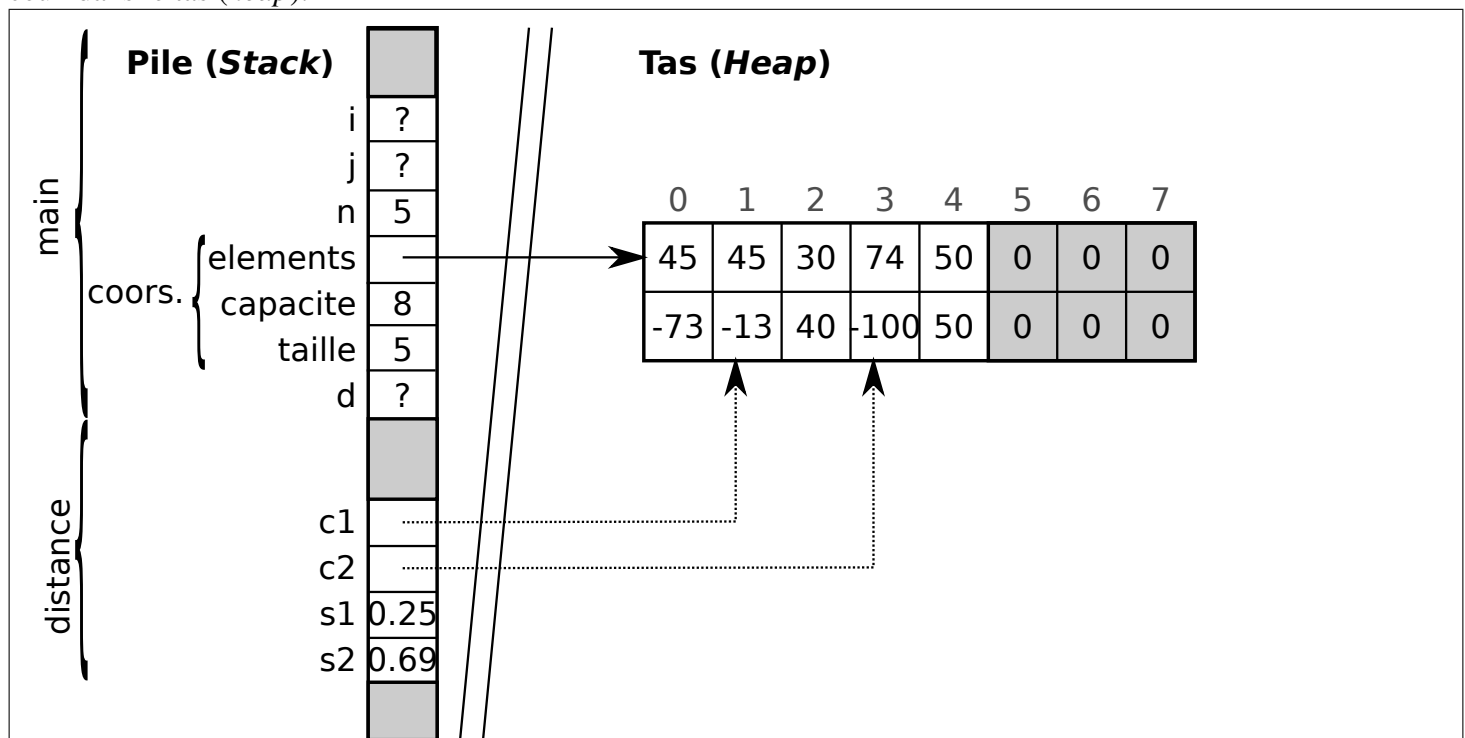
«5 (45,-73) (45,-13) (30, 40) (74,-100) (50, 50)».

(a) [4 points] Que fait ce programme ?

Ce programme :

- (1) lit un entier n ; (2) lit n coordonnées ;
- (3) parcourt les $n(n-1)$ paires de coordonnées :
- (3a) affiche "Eureka !" chaque fois d'une paire est à une distance de 25 ou moins.

(b) [4 points] On met un point d'arrêt sur la ligne 28. Dessinez l'état de la mémoire du programme immédiatement après la première exécution de la ligne 27. Montrez clairement les objets sur la pile d'exécution (*stack*) et ceux dans le tas (*heap*).



Notes : `i`, `j` et `d` n'ont pas encore de valeur. La coordonnée `c` peut être ignorée. Les valeurs exactes de `s1` et `s2` ne sont pas vérifiées.

(c) [4 points] Expliquez tout ce que fait la ligne 14. Cette ligne contient le mot clé `new`. Soyez aussi précis que possible dans votre réponse. Mettez en évidence les différentes étapes et leur ordre.

- (1) Demande au gestionnaire de mémoire d'allouer un bloc de capacité `* sizeof(Coordonnee)`.
- (2) Appelle le constructeur `Coordonnee::Coordonnee()` sur les toutes les objets du tableau natif alloué (`nb=capacite`). Ce constructeur initialise `latitude` et `longitude` à 0.0 (valeurs par défaut des 2 paramètres).
- (3) Assigne le pointeur `elements` à l'adresse du bloc qui a été alloué.

(d) [4 points] Dans le programme, il y a deux occurrences du mot clé `new`. La première dans le constructeur de `Tableau` et la deuxième dans la fonction `ajouter`. Combien d'octets au total ont été alloués dynamiquement par ces deux occurrences de `new` durant toute la vie du programme ? Considérez que la fonction `Tableau<T>::ajouter` double la capacité à chaque fois qu'un agrandissement est requis.

Le tableau a une capacité initiale de 1 est est doublé 3 fois jusqu'à capacité=8.

Donc 4 allocations. Taille des allocations : $1 + 2 + 4 + 8 = 15$.

Nombres d'octets : $15 * \text{sizeof}(\text{Coordonnee}) = 15 * 2 * \text{sizeof}(\text{double}) = 15 * 2 * 8 = 240$ octets.

3 Arbres binaires de recherches [12 points]

Dans le cours, plusieurs ébauches d'implémentations d'arbres binaires de recherche ont été présentées. Par exemple, la classe `ArbreAVL` présentée contient qu'un pointeur racine de type `Noeud`. Calculer la taille d'un arbre (nombre d'éléments dans un arbre) coûte $O(n)$ en temps. Pour éviter de parcourir tous les noeuds de l'arbre, une solution très simple est de conserver un compteur dans l'objet arbre. Ce compteur est incrémenté et décrémenté respectivement après chaque insertion et enlèvement. Ainsi, obtenir la taille de l'arbre se fait en temps constant.

Dans l'ébauche de classe `ArbreBinRech` ci-droite, on a décidé d'aller un peu plus loin en conservant la taille de tous les sous-arbres. Ainsi, la variable `taille` de la structure `Noeud` contient le nombre de noeuds du sous-arbre. Lors d'une insertion ou d'un enlèvement, la fonction `recalculertaille` est appelée sur tous les noeuds sur le chemin de l'opération et ceux impliqués dans une rotation.

```

1  template <class T>
2  class ArbreBR {
3  public:
4      int taille() const;
5      const T& get(int i) const;
6      const T& medianne() const
7          { return get(taille()/2); }
8  private:
9      struct Noeud{
10         T contenu;
11         Noeud *gauche, *droite;
12         void recalculertaille();
13         int taille;
14     };
15     Noeud* racine;
16 };

```

Écrivez le code des fonctions suivantes. Rappel : l'efficacité est un critère de correction.

```

1  template <class T> int ArbreBR<T>::taille() const{
2      return racine==NULL ? 0 : racine->taille; //si racine==NULL, l'arbre est vide (0)
3  }

```

```

1  template <class T> void ArbreBR<T>::Noeud::recalculertaille() const{
2      taille = 1; // On compte le noeud courant (forcément this!=NULL)
3      if(gauche!=NULL) taille += gauche->taille; // Additionne à gauche
4      if(droite!=NULL) taille += droite->taille; // Additionne à droite
5  }

```

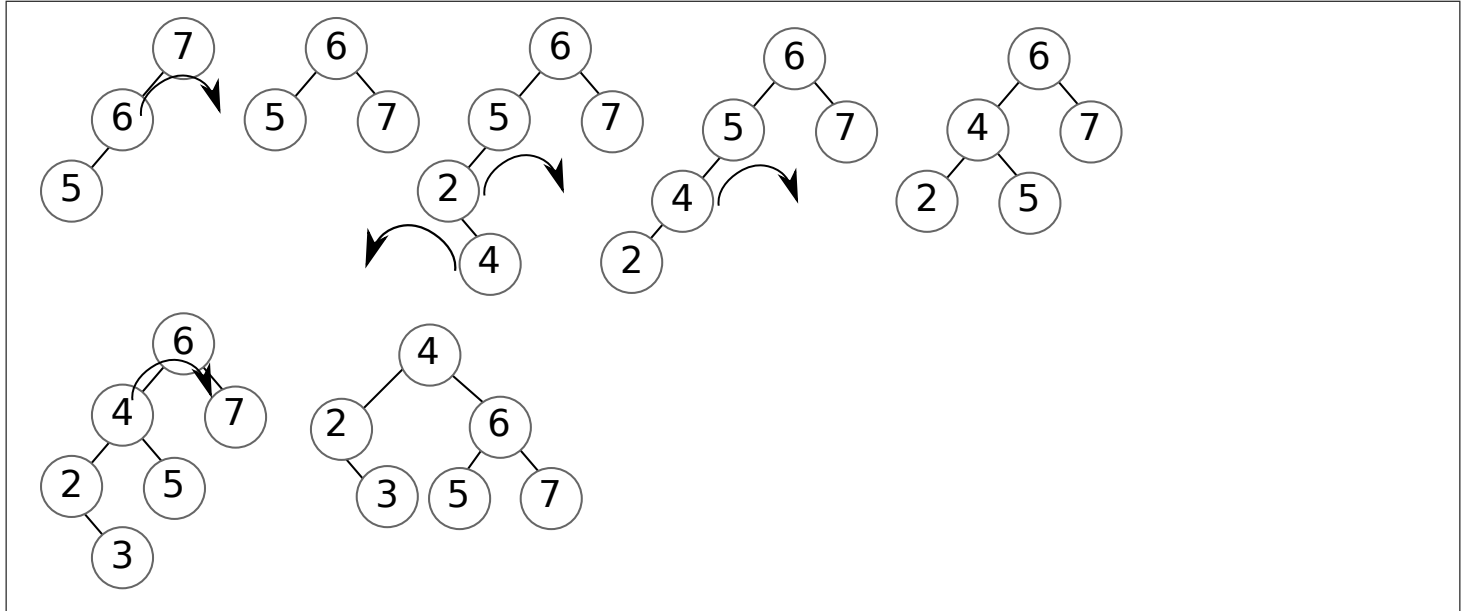
```

1  //Retourne une réf sur l'élément à la position i (le i-ème dans un parcours en inordre)
2  template <class T> const T& ArbreBR<T>::get(int i) const{
3      assert(i>=0 && i<taille()); // Idée : On doit passer par dessus i noeuds.
4      Noeud* n = racine; // Recherche binaire à partir de la racine.
5      while(true){
6          if(n->gauche!=NULL){ // Il y a des noeuds à gauche
7              if(i<n->gauche->taille){ // si i-ème noeud est à gauche
8                  n = n->gauche;
9                  continue;
10             }else // sinon on passe par dessus le sous-arbre de gauche
11                 i -= n->gauche->taille;
12         }
13         if(i==0) // s'il ne reste aucun noeud à passer par dessus
14             return n->contenu; // alors, on a trouvé le bon noeud
15         i--; // on passe par dessus le noeud courant (un noeud de moins à passer)
16         n = n->droite; // le noeud recherche est forcément à droite
17     }
18 }

```

4 Arbres AVL [18 points]

(a) [6 points] Insérez les nombres 7, 6, 5, 2, 4 et 3 dans un arbre AVL initialement vide. Montrez clairement les différentes étapes. Lorsqu'une rotation est requise, dessinez une flèche et redessinez le nouvel arbre résultant.



(b) [4 points] Combien de noeuds un arbre AVL de hauteur 6 peut-il contenir au ... ? Justifiez à l'aide de calculs.

Minimum : 20 noeuds.

$n(0) = 0$; $n(1) = 1$; $n(h) = n(h-1) + n(h-2) + 1$; $n(2) = 1 + 0 + 1 = 2$; $n(3) = 2 + 1 + 1 = 4$; $n(4) = 4 + 2 + 1 = 7$; $n(5) = 7 + 4 + 1 = 12$; $n(6) = 12 + 7 + 1 = 20$;

Maximum : 63 noeuds.

$2^6 - 1 = 63$

(c) [5 points] Dans un arbre AVL, quelle est la complexité temporelle des opérations suivantes :

recherche :

$O(\log n)$

insertion :

$O(\log n)$

enlèvement :

$O(\log n)$

rotation g-d :

$O(1)$

trouver le minimum :

$O(\log n)$

(d) [3 points] Écrivez le code de la fonction qui retourne la hauteur de l'arbre.

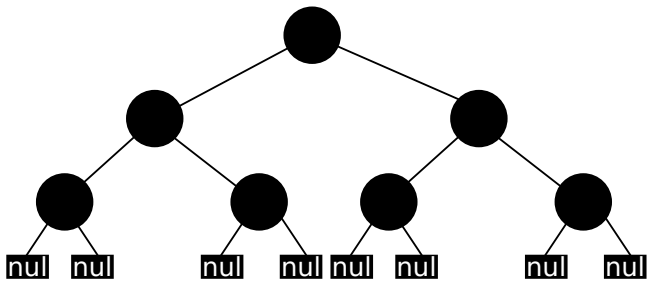
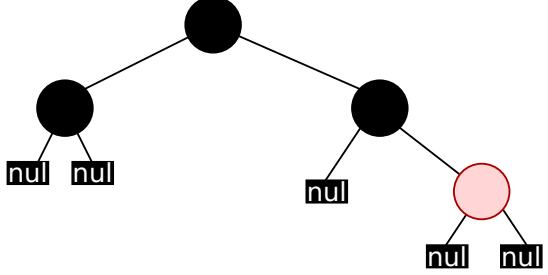
```
1 template <class T> int ArbreAVL<T>::hauteur() const{ // version inefficace
2     return hauteur(racine); }
3 template <class T> int ArbreAVL<T>::hauteur(const Noeud* n) const{
4     if(n==NULL) return 0;
5     return max(hauteur(n->gauche), hauteur(n->droite)) + 1;
6 }
```

```
1 template <class T> int ArbreAVL<T>::hauteur() const{ // version efficace
2     int h = 0;
3     Noeud* n = racine;
4     while(n!=NULL){
5         h++;
6         if(n->equilibre==1) // equilibre=hauteurgauche - hauteurdroite
7             n=n->gauche;
8         else
9             n=n->droite;
10    }
11    return h;
12 }
```

5 Arbres rouge-noir [6 points]

À titre de rappel, les principales caractéristiques d'un arbre rouge-noir sont : (1) la racine est noire ; (2) le nœud parent d'un nœud rouge doit être noir (on ne peut pas avoir deux nœuds rouges de suite sur un chemin) ; (3) les feuilles sont appelées « sentinelles », ne stockent aucun élément et sont considérées comme des nœuds noirs ; (4) toutes les sentinelles sont à une « profondeur noire » égale, la profondeur noire étant définie par le nombre de nœuds noirs sur le chemin.

Dessinez 2 arbres rouge-noir d'une hauteur de 3, le premier ayant un nombre maximum de nœuds, et le deuxième ayant un nombre minimal de nœuds. Considérez les directives suivantes : (1) si vous n'avez pas de crayon rouge, dessinez des cercles pleins pour les nœuds noirs et des cercles vides pour les nœuds rouges ; (2) dessinez de petits carrés pour représenter les sentinelles ; (3) pour le calcul de la hauteur de l'arbre, considérez uniquement les nœuds réguliers (les sentinelles ne sont pas considérées) ; (4) il n'est pas nécessaire de mettre des nombres dans les nœuds.

Nombre maximal de nœuds	Nombre minimal de nœuds
 <p style="color: red; margin-top: 10px;">Quatre autres solutions possibles. Ex : il est possible de colorier tout le 2e ou 3e niveau en rouge.</p>	 <p style="color: red; margin-top: 10px;">Trois autres solutions possibles : le nœud rouge doit être un enfant d'un des deux nœuds noirs au 2e niveau.</p>

6 Questions générales [8 points]

(a) [4 points] Vrai ou Faux : Dans un arbre binaire de recherche non équilibré, l'insertion se fait en temps $O(\log n)$. C'est uniquement la recherche qui dégénère en temps $O(n)$. Justifiez brièvement.

Faux. Quand l'arbre n'est pas équilibré, toutes les opérations dégénèrent en $O(n)$.

(b) [4 points] Vous devez stocker n entiers de type `int` dans une structure. On vous demande de choisir parmi les deux structures suivantes : une liste simplement chaînée (`Liste<int>`), ou un tableau linéaire (`Tableau<int>`). Notez que le nombre n ne peut être connu à l'avance. Le nombre n ne sera connu qu'après la dernière insertion. Quelle structure choisissiez-vous ? Le critère devant guider votre choix est la quantité de mémoire utilisée. Supposez que les pointeurs ont la même taille que les entiers de type `int`. Justifiez votre réponse.

Les deux choix sont à peu près équivalents, mais le Tableau demeure un meilleur choix.

Dans une liste simplement chaînée, une cellule contient un `int` et un pointeur, ce qui fait $2n$ entiers et pointeurs. Si on ajoute le pointeur dans l'objet `Liste`, on obtient $2n + 1$.

Dans un objet `Tableau` linéaire, il faut considérer que le tableau alloué peut avoir une capacité plus grande que requise. Le meilleur cas est lorsque n est une puissance de 2. Dans ce cas, on a exactement n entiers. Si on ajoute les 3 objets d'un `Tableau` (pointeur, capacité et taille), alors on obtient $n + 3$ dans le meilleur cas. Le pire cas est lorsque n égale une puissance de 2 + 1. Dans ce cas, on a $2(n - 1) = 2n - 2$ entiers. Si on ajoute les 3 objets d'un `Tableau` (pointeur, capacité et taille), alors on obtient $2n + 1$ dans le pire cas.

En conclusion, le `Tableau` est un meilleur choix dans tous les cas, sauf dans le pire cas où le `Tableau` est la `Liste` sont équivalents. Le `Tableau` est donc globalement un meilleur choix.

7 Résolution d'un problème - Distribution de brosses à dents (20 points)

Un dentiste donne une brosse à dents à ses clients à chaque visite. Le dentiste demande à chaque client de choisir une couleur pour sa brosse. Toutefois, le dentiste refuse un choix de couleur si un autre membre de la même famille du client possède une brosse de la même couleur. Deux clients sont considérés être dans la même famille s'ils ont le même numéro de téléphone. Un client jette sa vieille brosse lorsqu'il reçoit la nouvelle brosse. Un client peut choisir la même couleur que sa brosse précédente. Le dentiste a écrit une ébauche d'un programme C++. Complétez ce programme. Hypothèse raisonnable : les noms des personnes sont uniques.

```
1 #include <set> // std::set est similaire à notre ArbreAVL (mais arbre rouge-noir).
2 #include <map> // std::set est similaire à notre dictionnaire ArbreMap.
3 #include <string>
4 #include <iostream>
5 using namespace std;
6
7 int main(){
8     map<string, set<string> > famille; // tel --> ensemble de couleurs prises
9     map<string, string> clients; // nom --> couleur prise
10    //Pour enlever l'hypothèse des noms uniques :
11    //map<string, map<string, string> clients; // tel --> nom --> couleur prise
12    while(cin){
13        string nom, tel, couleur;
14        cout << "Entrez votre nom et le numéro de téléphone de votre famille: ";
15        cin >> nom >> tel;
16        famille[tel].erase(clients[nom]);
17        //famille[tel].erase(clients[tel][nom]); // Pour enlever l'hypothèse des noms
18        while(couleur.empty()){ // empty retourne vrai si ==""
19            cout << "Choisissez une couleur: ";
20            cin >> couleur;
21            bool ok; // <== mettre ok=true si la couleur est disponible
22            ok = famille[tel].find(couleur)==famille[tel].end();
23            if(ok){
24                cout << "OK\n";
25            }else{
26                cout << "Désolé, cette couleur est déjà prise\n";
27                couleur = ""; // force un nouveau choix de couleur
28            }
29        }
30        famille[tel].insert(couleur);
31        clients[nom] = couleur;
32        //clients[tel][nom] = couleur; //Pour enlever l'hypothèse des noms uniques
33    } //fin while(true)
34 }
```

Annexe A pour la Question 1

Cette page et les suivantes peuvent être détachées. Notez que le code a été allégé pour rentrer sur une page.

```

1  /* question1.cpp */
2  template <class T> class Tableau {
3      public:
4          Tableau(int capacite_initiale=1); // =1 important pour Q1(d)
5          ~Tableau();
6          void ajouter(const T& element); // ajouter à la fin
7          T& operator[] (int index);
8      private:
9          T* elements;
10         int capacite, taille;
11 };
12 template <class T> Tableau<T>::Tableau(int capacite_initiale)
13 : capacite(capacite_initiale), taille(0)
14 { elements = new T[capacite]; } // <== LIGNE 14
15 /* ... */
16 class Coordonnee {
17     public:
18         Coordonnee(double latitude_=0, double longitude_=0);
19     private:
20         double latitude;
21         double longitude;
22     friend double distance(const Coordonnee& c1, const Coordonnee& c2);
23     /*...*/
24 };
25 double distance(const Coordonnee& c1, const Coordonnee& c2){
26     double s1 = sin((c2.latitude-c1.latitude)/2);
27     double s2 = sin((c2.longitude-c1.longitude)/2); // <== LIGNE 27
28     return 2*RAYONTERRE * asin(sqrt(s1*s1 + cos(c1.latitude)*cos(c2.latitude)*s2*s2));
29 }
30 int main(){
31     int i, j, n;
32     std::cin >> n;
33     Tableau<Coordonnee> coors;
34     for(i=0;i<n;i++){
35         Coordonnee c;
36         std::cin >> c;
37         coors.ajouter(c);
38     }
39     double d = distance(coors[1], coors[3]);
40     for(i=0;i<n;i++)
41         for(j=i+1;j<n;j++)
42             if(distance(coors[i], coors[j]) <= 25.0)
43                 std::cout << "Eureka!" << std::endl;
44     return 0;
45 }
```


Annexe B pour la Question 2

```

1  int table[6][256]; // table de transitions : [état][caractère] -> nouvel état
2  bool alphanumeriques[256]; // tableau global
3  void initTable(){
4      for(int c=0;c<256;c++){
5          alphanumeriques[c] = false;
6          for(int e=0;e<7;e++){
7              table[e][c] = -1;
8          }
9      for(int c='a';c<='z';c++) alphanumeriques[c] = true;
10     for(int c='A';c<='Z';c++) alphanumeriques[c] = true;
11     for(int c='0';c<='9';c++) alphanumeriques[c] = true;
12     for(int c=0;c<256;c++){
13         if(alphanumeriques[c]){
14             table[0][c] = 1;           table[1][c] = 1;
15             table[2][c] = 3;
16             table[3][c] = 3;
17             table[4][c] = 5;           table[5][c] = 5;
18         }
19         table[1]['.'] = 0;
20         table[1]['@'] = 2;
21         table[3]['.'] = 4;
22         table[5]['.'] = 4;
23     }
24 int main(int argc, char** argv){
25     initTable();
26     string contenu = lireFichier(argv[1]); // lit le fichier argv[1] passé en argument
27     //string contenu="a aa aaa a@a.com b.e@uqam.ca deux@uqam.,
28     trois@quatre.cinq;six..sept@hu.it"; // <== Exemple
29     ArbreAVL<string> resultat;
30     int n = contenu.size();
31     for(int i=0;i<n;i++){
32         int debut=i, fin=i, etat=0;
33         while(etat!=-1 && i<n){
34             etat = table[etat][contenu[i]]; // contenu[i] retourne le i-ème caractère
35             i++;
36             if(etat==5) fin=i;
37         }
38         if(fin>debut){
39             resultat.inserer(contenu.substr(debut, fin-debut)); //substr=sous-chaine
40             i=fin-1;
41         }else
42             i=fin;
43     }
44     for(ArbreAVL<string>::Iterateur i=resultat.debut();i;i++)
45         cout << *i << endl;
46     return 0;
47 }
```