

INF3105 – Structures de données et algorithmes

Examen de mi-session – Hiver 2013

Éric Beaudry
Département d'informatique
Université du Québec à Montréal

Jeudi 7 mars 2013 – 18h00 à 21h00 (3 heures) – Locaux SB-M210 + SB-M230

Instructions

- Aucune documentation permise.
- Les appareils électroniques, incluant les téléphones et 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 6 questions.
- Ne détachez pas les feuilles du questionnaire, à moins de les brocher à nouveau avant la remise.
- Le côté verso peut être utilisé comme brouillon. Des feuilles additionnelles peuvent être demandées au surveillant.
- À l'exception de la question 5, vous devez répondre à l'aide d'un crayon d'une autre couleur que rouge.

Identification

Nom : Solutionnaire

Résultat

Q1 (/5)	Q2 (/5)	Q3 (/5)	Q4 (/4)	Q5 (/2)	Q6 (/4)	TOTAL (/25)

1 Connaissances techniques et C++ (5 points)

```

1  #include <iostream>
2  class Allo{
3  public:
4      Allo(int x_=0) : x(x_){ std::cout << "A" << x << " ";}
5      Allo(const Allo& autre) : x(autre.x){ std::cout << "B" << x << " ";}
6      ~Allo() { std::cout << "C" << x << " ";}
7      int x;
8  };
9  void f(Allo a1, Allo& a2, Allo* a3){
10     a1.x++;    a2.x--;    a3+=1;    (a3->x)+=2000;
11 }
12 int main(){
13     Allo tab[3];
14     Allo a(20);
15     Allo* b = new Allo(5);
16     Allo* c = tab + 1;
17     f(a, b, c);
18 }

```

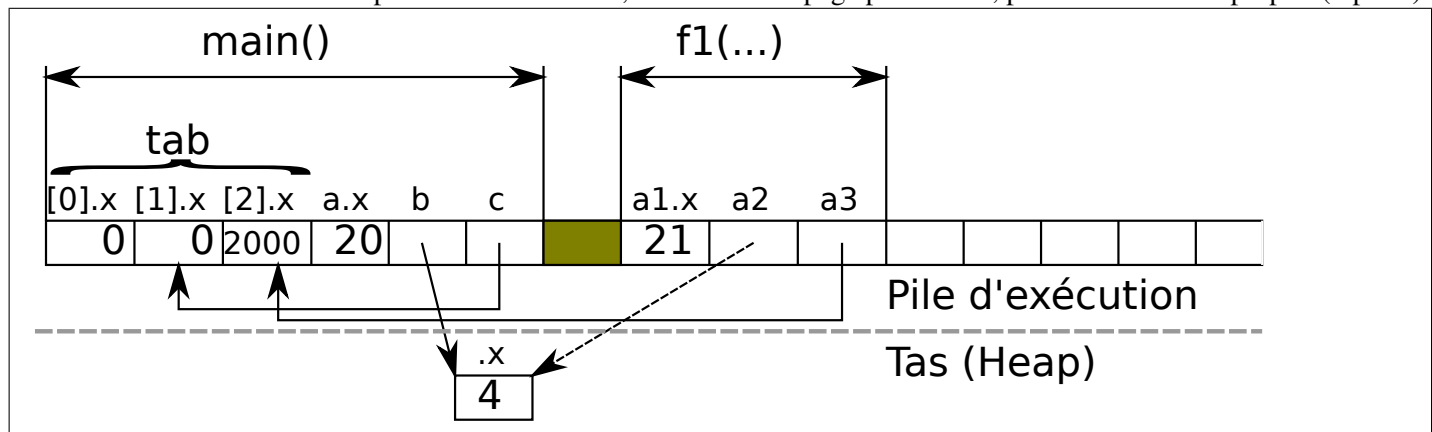
(a) Le programme ci-haut ne peut être compilé correctement en raison d'une erreur à ligne 17. Indiquez la correction minimale nécessaire pour compiler le programme. (1 point)

`f(a, *b, c);`

(b) Une fois corrigé, est-ce que le programme s'exécute correctement (sans terminaison anormale)? Si oui, dites simplement oui. Sinon, indiquez l'erreur et la correction minimale pour éviter une terminaison anormale. (1 point)

Oui.

(c) Dessinez l'état de la pile d'exécution et du tas (*heap*) après l'exécution de la ligne 10 (immédiatement avant le retour de la fonction `f1` appelée à la ligne 17). Soyez aussi précis que possible. Indiquer la zone mémoire occupée par chaque variable. Conseil : commencez par faire un brouillon, au verso de la page précédente, puis transcrivez au propre. (1 point)



(d) Qu'affiche le programme ? (1 point)

A0 A0 A0 A20 A5 B20 C21 C20 C2000 C0 C0

Des permutations dans le bloc «C20 C2000 C0 C0» sont acceptées.

(e) La mémoire est-elle libérée adéquatement à la fin du programme ci-haut ? Si oui, dites simplement oui. Sinon, indiquez la correction minimale (et où l'appliquer) pour libérer la mémoire adéquatement. (1 point)

Non. La mémoire allouée par le `new` de la ligne 15 n'est pas libérée adéquatement. Il faut ajouter : `delete b;` à la tout fin de la fonction `main()`.

2 Analyse algorithmique et optimisation d'une fonction (5 points)

```

1  bool f1(const Tableau<int>& tab){
2      for(int i=0;i<tab.taille();i++)
3          for(int j=i+1;j<tab.taille();j++)
4              for(int k=0;k<tab.taille();k++)
5                  if(tab[i] + tab[j] == tab[k])
6                      return true;
7      return false;
8  }
9  int main(){
10     Tableau<int> tab;
11     tab.ajouter(2); tab.ajouter(3); tab.ajouter(6); tab.ajouter(7); tab.ajouter(10);
12     std::cout << (f1(tab) ? "vrai" : "faux") << std::endl;
13 }

```

(a) Que fait la fonction `f1` ci-haut ? (1 point)

La fonction `f1` vérifie s'il existe de deux nombres dans le tableau `tab` dont la somme se retrouve dans ce même tableau.

(b) Qu'affiche le programme `main` ci-haut ? (1 point)

vrai

(c) Indiquez la complexité temporelle en notation grand O de la fonction `f1`. $n = \text{tab.taille()}$. Justifiez. (1 point)

$O(n^3)$.
La première boucle `for` fait n itérations. Pour chaque i -ème itération de la première boucle `for`, la deuxième boucle `for` fait $n-i-1$ itérations. Donc : $(n-1) + (n-2) + (n-3) + \dots + 1 = (n-1)(n-2)/2 \in O(n^2)$. La troisième boucle `for` fait n itérations. Donc : $O(n^3)$.

(d) Proposez une nouvelle fonction, équivalente à `f1`, mais ayant une complexité temporelle d'un ordre de grandeur moindre. Rappel : vous pouvez supposer l'existence de fonctions et de structures de données raisonnables. (2 points)

Réponse :

```

1  bool f1(const Tableau<int>& tab){
2      Tableau<int> copie = tab; // Copie en O(n)
3      trier(copie); // Tri en O(n log n)
4      for(int i=0;i<tab.taille();i++) // O(n) iterations
5          for(int j=i+1;j<tab.taille();j++) // O(n) iterations
6              if(recherche_dichotomique(copie, tab[i] + tab[j])) // test en O(log n)
7                  return true;
8      return false;
9  }

```

Complexité temporelle de la nouvelle fonction `f1` : $O(n^2 \log n)$.

3 Arbres binaires de recherche (5 points)

```

1  template <class T> class ArbreBinRech{
2      struct Noeud{
3          T contenu; //Rappel : gauche->contenu < contenu < droite->contenu
4          Noeud *gauche, *droite;          };
5      Noeud* racine;
6      public: // ...
7      const T* minimum() const;
8      bool contient(const T& e) const;    };

```

La classe `ArbreBinRech<T>` ci-haut implémente un arbre binaire de recherche qui n'est pas forcément équilibré.

(a) Écrivez une fonction **non récursive** `minimum` qui retourne un pointeur sur l'élément minimal dans un arbre binaire de recherche. Si l'arbre est vide, il faut retourner `NULL`. (1 point)

Réponse :

```

1  template <class T>
2  const T* ArbreBinRech::minimum() const{
3      const Noeud* n = racine;
4      if(n==NULL) return NULL;
5      while(n->gauche!=NULL)
6          n = n->gauche;
7      return &(n->contenu);
8  }

```

(b) Écrivez une fonction `contient` qui vérifie l'existence de l'élément `e` dans un arbre binaire de recherche. La fonction retourne `true` si l'élément existe, sinon `false`. Votre implémentation peut être récursive ou non récursive. (2 points)

Réponse :

```

1  template <class T>
2  bool ArbreBinRech::contient(const T& e) const{
3      const Noeud* n = racine;
4      while(n!=NULL) {
5          if(e < n->contenu) n = n->gauche;
6          else if(n->contenu < e) n = n->droite;
7          else return true;
8      }
9      return false;
10 }

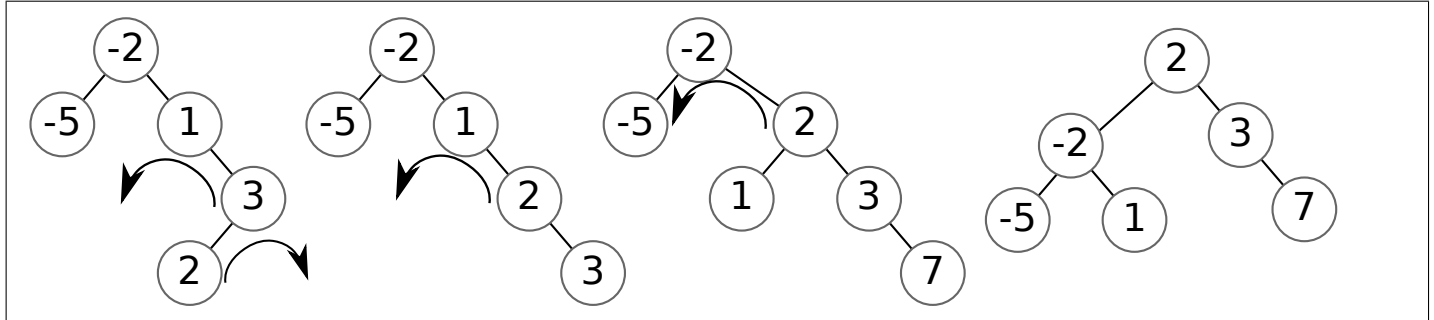
```

(c) Indiquez dans les cases à droite si les énoncés suivants sont vrais ou faux (2 points).

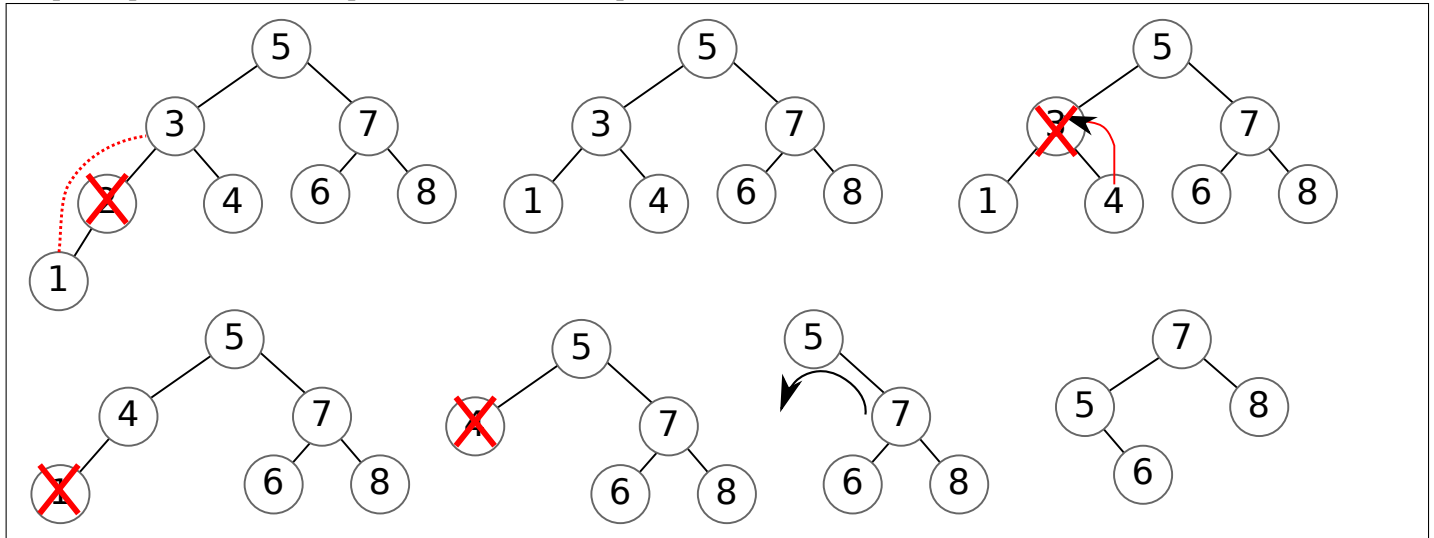
Dans un arbre binaire de recherche non équilibré, l'insertion se fait en temps $O(\log n)$. C'est uniquement la recherche qui dégénère en temps $O(n)$. » Toutes les opérations dégénèrent en temps $O(n)$.	Faux
L'insertion et la suppression sont en pratique plus rapides dans un arbre rouge-noir que dans un arbre AVL. » Bien qu'elles soient du même ordre de grandeur, les insertions/suppressions dans les arbres r-n demandent généralement (en pratique) moins de réorganisations que dans les arbres AVL.	Vrai
La recherche est en pratique plus rapide dans un arbre rouge-noir que dans un arbre AVL. » Hauteur maximale d'un arbre AVL : $1.44 \log n - 0.328$; rouge-noir : $2 \log n$. Donc, l'arbre AVL est généralement moins haut en pratique, donc plus rapide pour les recherches.	Faux
La copie d'un arbre binaire de recherche se fait en temps d'au moins $O(n \log n)$. La raison est qu'on doit inévitablement faire n insertions, chacune coûtant $O(\log n)$. » Il est possible de copier un arbre en temps linéaire en recopiant sa structure exacte.	Faux

4 Arbres AVL (4 points)

(a) Insérez les nombres -2, 1, -5, 3, 2 et 7 dans un arbre AVL initialement vide. À chaque insertion, ajoutez un nouveau noeud à l'arbre courant. À chaque fois qu'une rotation est requise : (1) dessinez une flèche pour montrer la rotation requise ; (2) dessinez un nouvel arbre pour montrer le résultat ; (3) le nouvel arbre devient l'arbre courant pour la prochaine étape. (2 points)



(b) Supprimez les éléments 2, 3, 1 et 4 dans l'arbre AVL suivant. Indiquez toutes les opérations. Dessinez un arbre après chaque étape (incluant les étapes intermédiaires). (2 points)

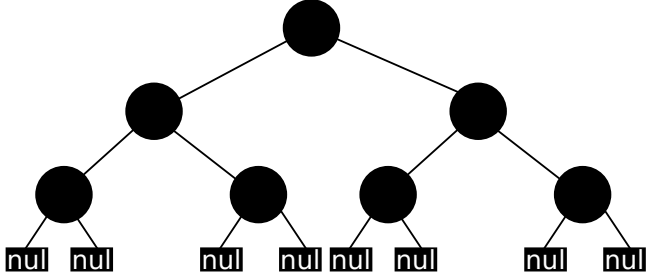
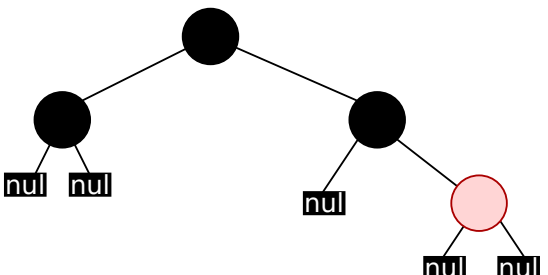


5 Arbres rouge-noir (2 points)

À titre de rappel, revoici les principales caractéristiques d'un arbre rouge-noir :

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.

<p>Nombre maximal de nœuds (1 point)</p>  <p>Quatre autres solutions possibles. Ex : il est possible de colorier tout le 2e ou 3e niveau en rouge.</p>	<p>Nombre minimal de nœuds (1 point)</p>  <p>Trois autres solutions possibles : le nœud rouge doit être un enfant d'un des deux nœuds noirs au 2e niveau.</p>
---	---

6 Résolution d'un problème (4 points)

Vous devez écrire un programme qui détermine et affiche la cooccurrence de deux mots la plus fréquente dans un texte envoyé dans l'entrée standard. Une cooccurrence de deux mots est définie comme une paire de mots qui apparaissent dans une même phrase. Dans le texte ci-bas, la cooccurrence la plus fréquente est la paire («livres», «bibliothèque»).

Plusieurs livres sur les structures de données sont disponibles à la bibliothèque . Dans une bibliothèque , on retrouve beaucoup de livres . Marc a écrit des livres que je n ' ai pas pu trouver à la bibliothèque . Le cours INF3105 porte sur les structures de données .

Pour faciliter le *parsing*, des espaces blancs et des retours de ligne séparent les mots et les symboles de ponctuation. Une phrase se termine par une chaîne ".". Un mot est toute chaîne de 3 caractères et plus. Il n'est pas requis de gérer les familles de mots (ex. : livres et livre). Vous pouvez vous inspirer du code suivant pour lire des phrases.

```

1  while(cin){
2      string mot;
3      do{      cin >> mot;
4          if(mot.length() >= 3) { /* L'objet mot est un mot valide. */
5      }while(cin && mot != "."); // Tant que la phrase n'est pas finie par un point
6  }
```

Problème simplifié (maximum de 3 points). Si vous jugez que le problème ci-haut est trop difficile, vous pouvez résoudre un problème simplifié. Cependant, vous aurez au maximum 3 points sur quatre. Le problème simplifié consiste à déterminer le mot le plus fréquent dans le texte en entrée. Les mots de deux caractères ou moins doivent être ignorés.

Conseil : avant de commencer à répondre à la question, prenez le temps de bien relire les instructions sur la première page relativement aux questions demandant l'écriture de code.

Solution complète

```

1  #include <arbremap.h>
2  #include <string>
3  #include <tableau.h>
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9      ArbreMap<string, ArbreMap<string, int> > compteurs;
10     int nbMax=0;
11     string cpf_mot1, cpf_mot2;
12     while(cin){
13         Tableau<string> phrase;
14         string mot;
15         do{
16             cin >> mot;
17             if(mot.length() >= 3)
18                 phrase.ajouter(mot);
19         }while(cin && mot != ".");
20
21         for(int i=0;i<phrase.taille();i++)
22             for(int j=i+1;j<phrase.taille();j++){
23                 const string& mot1 = phrase[i];
24                 const string& mot2 = phrase[j];
25                 int n=0;
26                 if(mot1<mot2) n=compteurs[mot1][mot2]++;
27                 else if(mot2<mot1) n=compteurs[mot2][mot1]++;
28                 if(n>nbMax){
29                     cpf_mot1=mot1;
30                     cpf_mot2=mot2;
31                     nbMax=n;
32                 }
33             }
34     }
35     cout << cpf_mot1 << ", " << cpf_mot2 << endl;
36     return 0;
37 }

```

Analyse de l'algorithme. (optionnelle)

Taille du problème : n mots en entrée, m phrases, $k \approx \frac{n}{m}$ mots par phrase et l mots distincts.

Complexité temporelle : $O(n + mk^2 \log l)$. Complexité spatiale : $O(l^2)$.

Solution au problème simplifié : voir les diapositives sur l'ArbreMap de la semaine #7, et plus spécifiquement l'Exercice 1.

/****** Fin de l'examen ! *****/