INF3105 – Structures de données et algorithmes
Été 2016 – Examen de mi-session

Éric Beaudry
Département d'informatique
Université du Québec à Montréal

Jeudi 23 juin 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 :
   — le fonctionnement correct, l'efficacité (temps et mémoire), la clarté, la simplicité du code 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, contient 7 questions et vaut 20 % de la session.
7. Ne détachez pas les feuilles du questionnaire, à moins de les brocher à nouveau avant la remise.
8. 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.

Identification

Nom : Solutionnaire

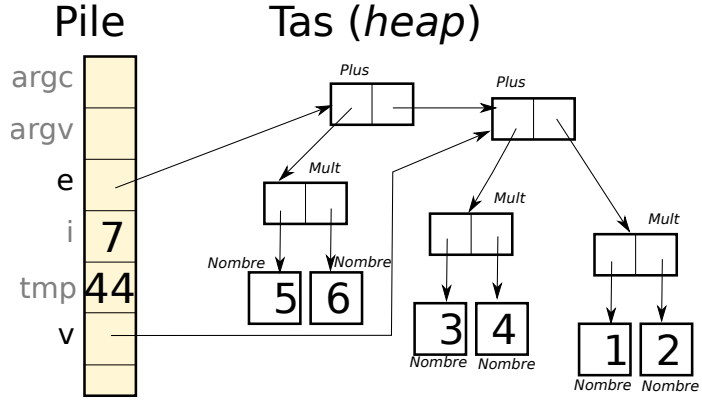
Résultat

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

1 Connaissances techniques et C++ [16 points]

Pour répondre à cette question, référez-vous au programme fourni à l'Annexe A (page 9). Notez que ce programme se compile sans erreur.

(a) [4 points] On met un point d'arrêt sur la ligne 42. Dessinez l'état de la mémoire du programme rendu à ce point d'arrêt. Montrez clairement les objets sur la pile d'exécution (*stack*) et ceux dans le tas (*heap*). La classe `Expression` a une fonction virtuelle `evaluer`. Cela s'apparente à une méthode `abstract` en Java.



Les variables `argc` et `argv` peuvent ne pas être présentées. Les variables `i` et `tmp` peuvent ne pas apparaître, car elles ne sont plus dans la portée courante (*current scope*). Dans la réalité, tous les objets de type `Expression`, ou héritant de, auront un attribut spécial suplémentaire `vtable` à l'interne. Cet attribut pointe sur une table d'adresses de fonctions virtuelles. Il y a une table pour la classe `Expression` et une table par classe héritée de.

(b) [4 points] Qu'affiche le programme ?

44

(c) [4 points] Ce programme libère-t-il la mémoire correctement ? Si **oui**, expliquez comment et quand tous les objets de type `Mult` sont libérés. Si **non**, indiquez combien d'objets sont «perdus» en mémoire. Expliquez brièvement comment vous pourriez corriger le problème.

Il y a 10 objets «perdus» : 6 de type `Nombre`, 3 de type `Mult` et 1 de type `Plus`.
Notez que le nombre d'octets «perdus» n'étaient pas demandé.
Pour résoudre le problème, il faut déclarer et définir des destructeurs sur les classes `Mult` et `Plus`. Ces derniers doivent faire :

```
delete e1;
delete e2;
```

(d) [4 points] Expliquez dans vos propres mots ce qu'est une **déclaration** et une **définition** en C++.

Déclaration : Une déclaration annonce l'existance de quelque chose, comme une fonction, une classe ou une variable.

Définition : Une définition définie ce que fait une chose précise. Par exemple, la définition d'une fonction, c'est le code qui lui est associé.

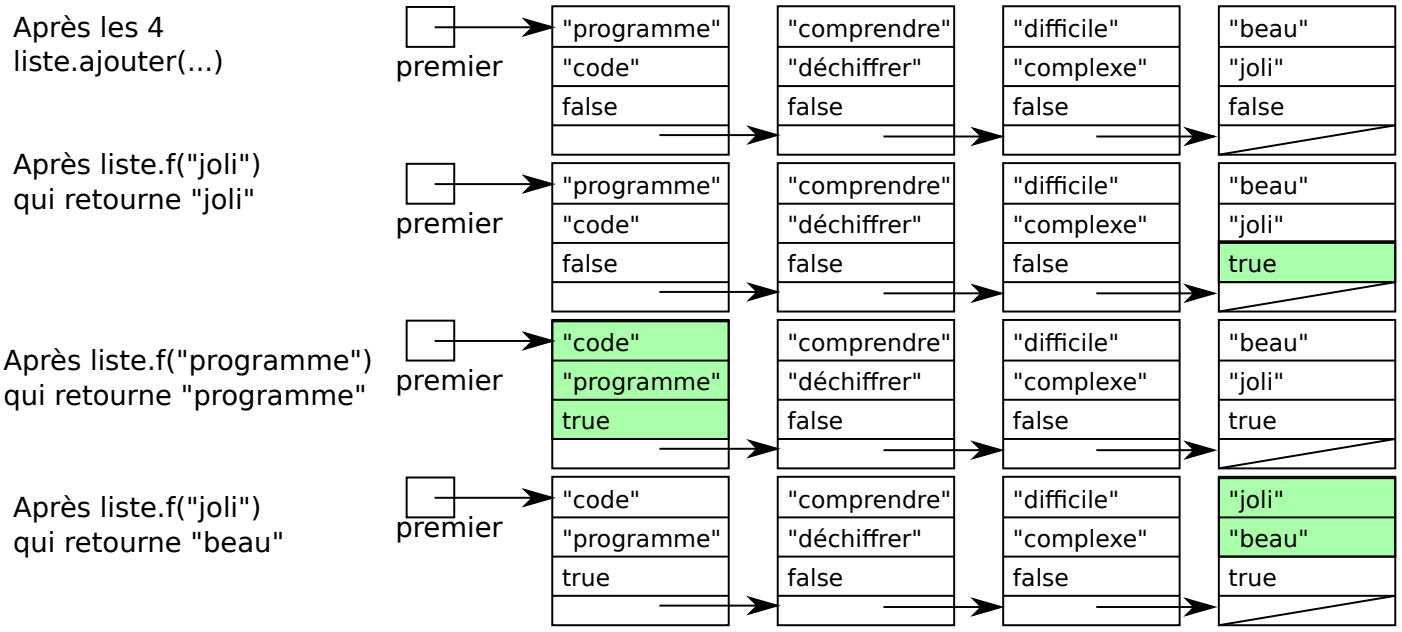
2 Complexité algorithmique et Anayse de code [20 points]

(a) Simplifiez les ordres de grandeur suivants (en notation grand O). [8 points]

$O(7)$	$O(1)$	$O(4n\log n + 8k\log n)$	$O((n+k)\log n)$
$O(2n \times n/8)$	$O(n^2)$	$O((4n+3)\times(12n-7))$	$O(n^2)$
$O(2^n + n^3 + 9n^2)$	$O(2^n)$	$O(4n + 8n\log(7n) + 75)$	$O(n\log n)$
$O((42n^2+n+8)(k+8))$	$O(n^2k)$	$O(n! + 8n^3 + n^2)$	$O(n!)$

Référez-vous au programme fourni à l'Annexe B (page 10) pour les sous-questions suivantes.

Le programme utilise une liste de paires de synonymes pour alterner les synonymes. Le programme évite d'utiliser 2 fois un même mot lorsque qu'un synonyme existe.



(b) [5 points] Qu'affiche le programme si on lui entre la chaîne : «

```
Ce joli programme semblera difficile à comprendre .
Avec l' indice « synonyme », ce joli programme sera moins complexe à comprendre .
Le code aurait été moins complexe à comprendre avec un beau résumé .
```

» ? Au lieu de tout récrire, vous pouvez simplement encercler les mots qui changent et par quoi ils sont remplacés.

Ce joli programme semblera difficile à comprendre . Avec l' indice « synonyme », ce beau code sera moins complexe à déchiffrer . Le programme aurait été moins difficile à comprendre avec un joli résumé .

(c) [4 points] Supposons qu'on remplace les lignes 5 à 8 dans `main.cpp` par une lecture dans un fichier `s.txt` contenant k paires de mots. Quelle est la complexité temporelle du programme ? Utilisez la notation grand O. Supposez que n est le nombre de mots en entrée et k est la taille de la liste. Justifiez.

Lire les synonymes coûte $O(k)$. La boucle `while` à la ligne 9 de `main.cpp` itère n fois. Chaque itération génère un appel à la fonction `ListeS::f`. Dans le pire cas, cette fonction itère au complet la liste des k paires de synonymes. Donc : $O(k+nk)$. Après simplification : $O(nk)$

(d) [3 points] Écrivez le code du destructeur de `ListeS`. La mémoire doit être libérée correctement.

```
1 ListeS::~ListeS(){
2     while(premier!=NULL){
3         S* s = premier->suivant;
4         delete premier;
5         premier = s;
6     }
7 }
```

3 Arbres binaires de recherches [16 points]

Lisez l'Annexe C (page 10) et complétez le code des fonctions suivantes. Attention, les 2 premières fonctions à coder sont avec la classe `ArbreBR1` et les 2 dernières avec la classe `ArbreBR2`.

```
1  template <class T> int ArbreBR1<T>::taille() const{
2      return taille(racine); // calculer taille du sous-arbre engendré par la racine
3  }
4  template <class T> int ArbreBR1<T>::taille(const Noeud* n) const{
5      // si l'arbre (le sous-arbre engendré par n) est vide, donc zéro (0)
6      if(n==NULL) return 0;
7      // la taille d'un arbre = somme de la taille des 2 sous-arbres gauche et doite + 1
8      return taille(n->gauche) + taille(n->droite) + 1;
9  }
```

```
1  template <class T> int ArbreBR1<T>::compter(const T& min, const T& max) const{
2      return compter(racine, min, max);
3  }
4  // Méthode simple en O(n) : presque identique au calcul de la taille
5  template <class T>
6  int ArbreBR1<T>::compter(const Noeud* n, const T& min, const T& max) const{
7      // si l'arbre (sous-arbre engendré par n) est vide, donc zéro (0)
8      if(n==NULL) return 0;
9      // si le noeud courant est dans [min,max] nb=1, sinon nb=0
10     int nb = min <= n->contenu  &&  n->contenu <= max   ? 1 : 0;
11     // compter à gauche et à droite
12     return compter(n->gauche, min, max) + compter(n->droite, min, max) + nb;
13 }
14
15 // ou
16
17 // Méthode efficace en O(k + log n)
18 template <class T>
19 int ArbreBR1<T>::compter(const Noeud* n, const T& min, const T& max) const{
20     // si l'arbre (sous-arbre engendré par n) est vide, donc zéro (0)
21     if(n==NULL) return 0;
22     // si le noeud courant est dans [min,max] nb=1, sinon nb=0
23     int nb = min<=n->contenu && n->contenu<=max ? 1 : 0;
24     if(min < n->contenu) // compter à gauche ssi il peut y avoir des noeuds >=min
25         nb += compter(n->gauche, min, max);
26     if(max > n->contenu) // compter à droite ssi il peut y avoir des noeuds <=max
27         nb += compter(n->droite, min, max);
28     return nb;
29 }
```

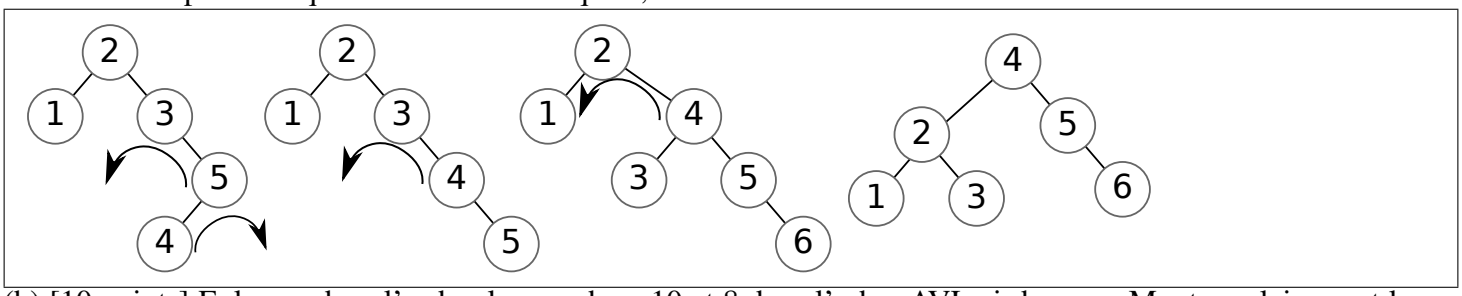
```
1  template <class T> void ArbreBR2<T>::inserer(const T& e) {
2      inserer(racine, e);
3  }
4  template <class T> bool ArbreBR2<T>::inserer(Noeud*& n, const T& e) {
5      if(n==NULL){ // si on ne trouve pas e dans l'arbre, on l'ajoute dans une feuille
6          n = new Noeud(e); // on suppose que le constructeur Noeud::Noeud met taille=1
7          return true; // indique d'un nouveau noeud a été créé
8      }
9      if(n->contenu == e) // si existe déjà, rien faire (ou écraser {n->contenu=e;})
10         return false;
11     // le ? dans la ligne permet de décider si on descend à gauche ou à droite
12     if(inserer(e < n->contenu ? n->gauche : n->droite, e)){ // insérer récursivement
13         n->taille++;  // si une insertion a eu lieu, on incrémente taille
14         return true;
15     }else
16         return false;
17 }
```

La fonction `ArbreBR2<T>::compter` en temps $O(\log n)$ est la partie la plus difficile de l'examen. L'astuce consiste à trouver la position indicée des bornes min et max dans l'arbre. Ensuite, il suffit de faire la différence des positions de min et max, plus 1, et on aura le nombre de noeuds compis entre min et max.
Le calcul de la position d'un élément e peut se faire avec une fonction `position`. Cette fonction recherche l'élément e dans l'arbre. Durant la recherche, on incrémente un compteur p qui compte le nombre de noeuds plus petits ou égal à e. Quand on descend à gauche, on ne touche pas à p. Quand on on descend à droite, c'est qu'on «saute» le noeud courant et son sous-arbre de gauche. Ainsi, p est incrémenté de la taille du sous-arbre de gauche plus 1. Quand on trouve un noeud contenant e, on saute seulement son sous-arbre de gauche. Si le noeud n'est pas trouvé, on retranche 1 si on ne veut pas la position du noeud suivant (pour borne max).

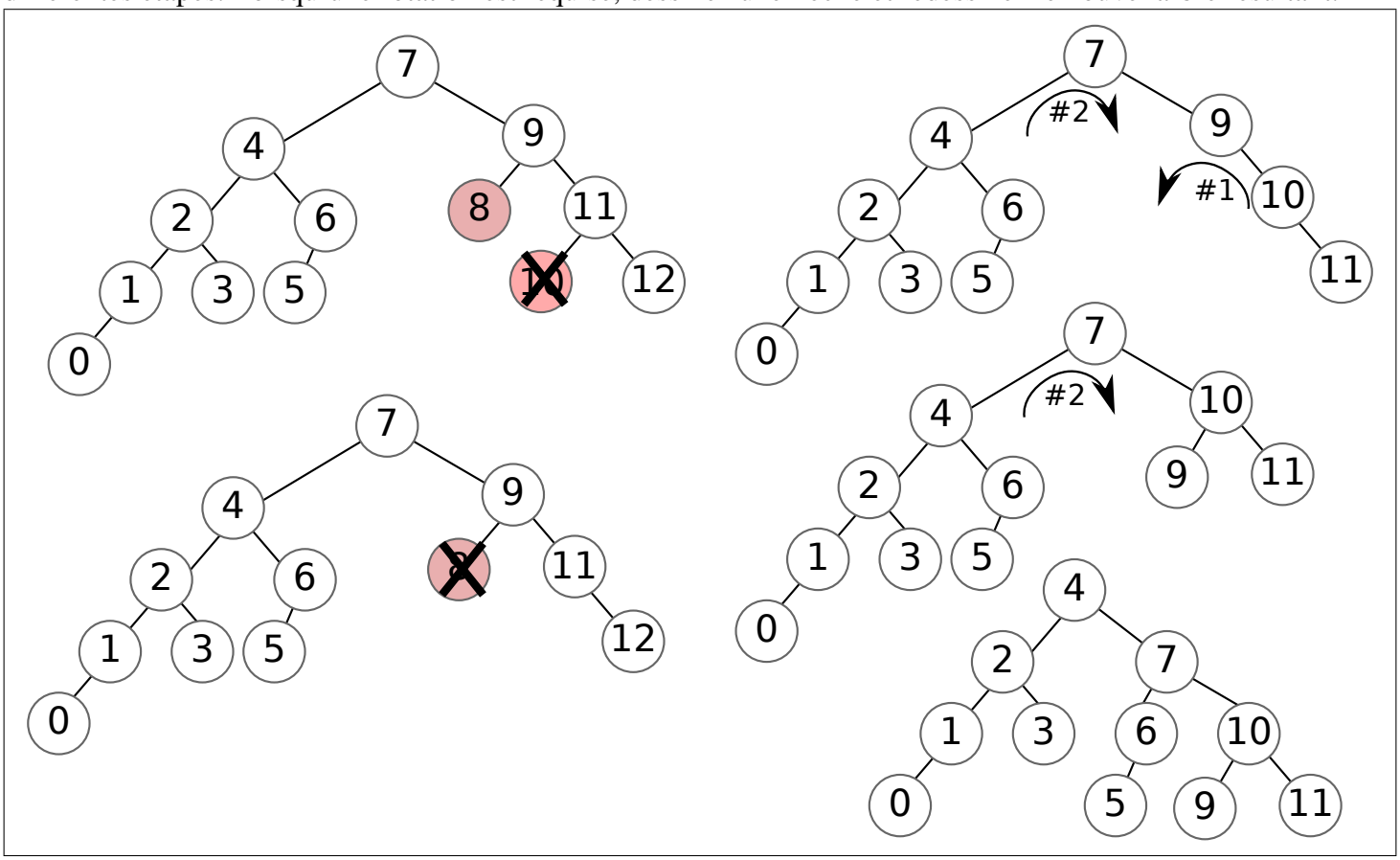
```
1  template <class T> int ArbreBR2<T>::compter(const T& min, const T& max) const{
2      return position(max,false) - position(min,true) + 1;
3  }
4  template <class T>
5  int ArbreBR2<T>::position(const T& e, bool suivantsinontrouve) const{
6      const Noeud* n = racine;
7      int p = 0; // position : compteur de noeuds <= e
8      while(n){ // tant que n!=NULL
9          if(e < n->contenu) // test si à gauche
10             n = n->gauche; // rien de spécial, juste descendre à gauche
11         else{ // cas e>=n->contenu
12             if(n->gauche)
13                 p+= n->gauche->taille;  // on saute le sous-arbre de gauche
14             if(e==n->contenu) return p; // si on trouve le noeud
15             p++; // cas e strictement > que n->contenu
16             n = n->droite; // descendre à droite
17         }
18     }
19     return suivantsinontrouve ? p : p-1 ;
20 }
```

4 Arbres AVL [20 points]

(a) [10 points] Insérez les nombres 2, 3, 1, 5, 4 et 6 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.

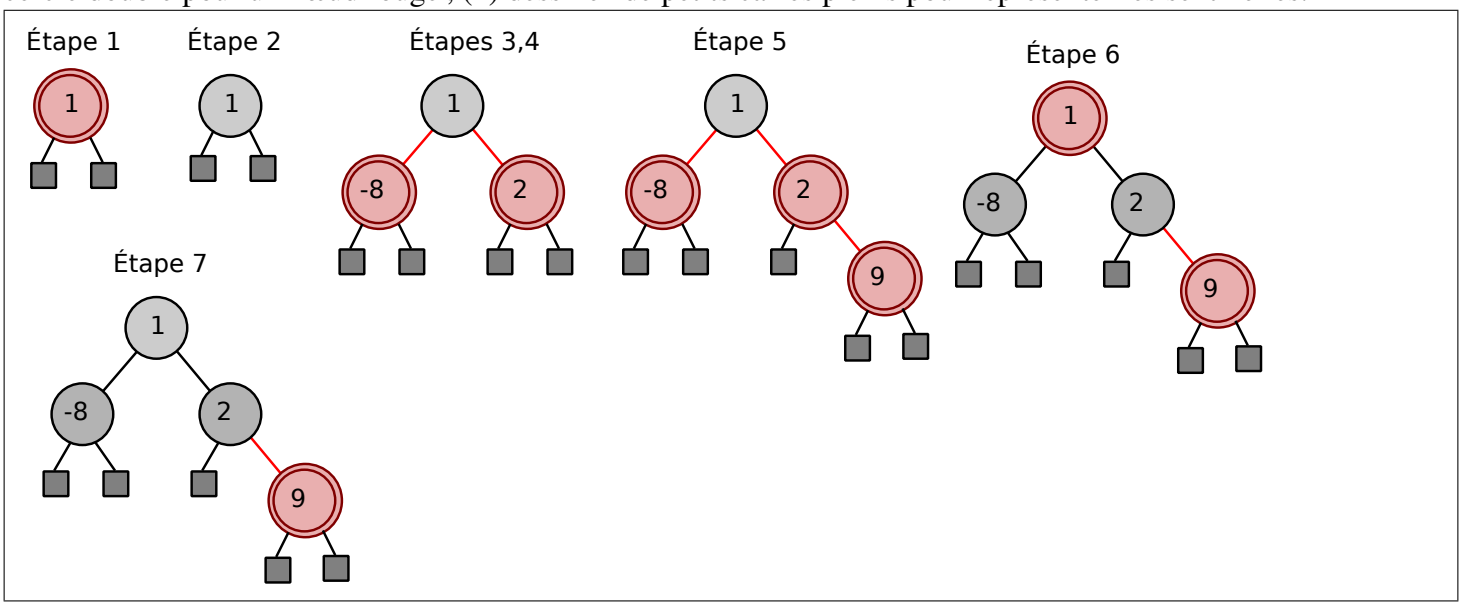


(b) [10 points] Enlevez, dans l'ordre, les nombres 10 et 8 dans l'arbre AVL ci-dessous. Montrez clairement les différentes étapes. Lorsqu'une rotation est requise, dessinez une flèche et redessinez le nouvel arbre résultant.



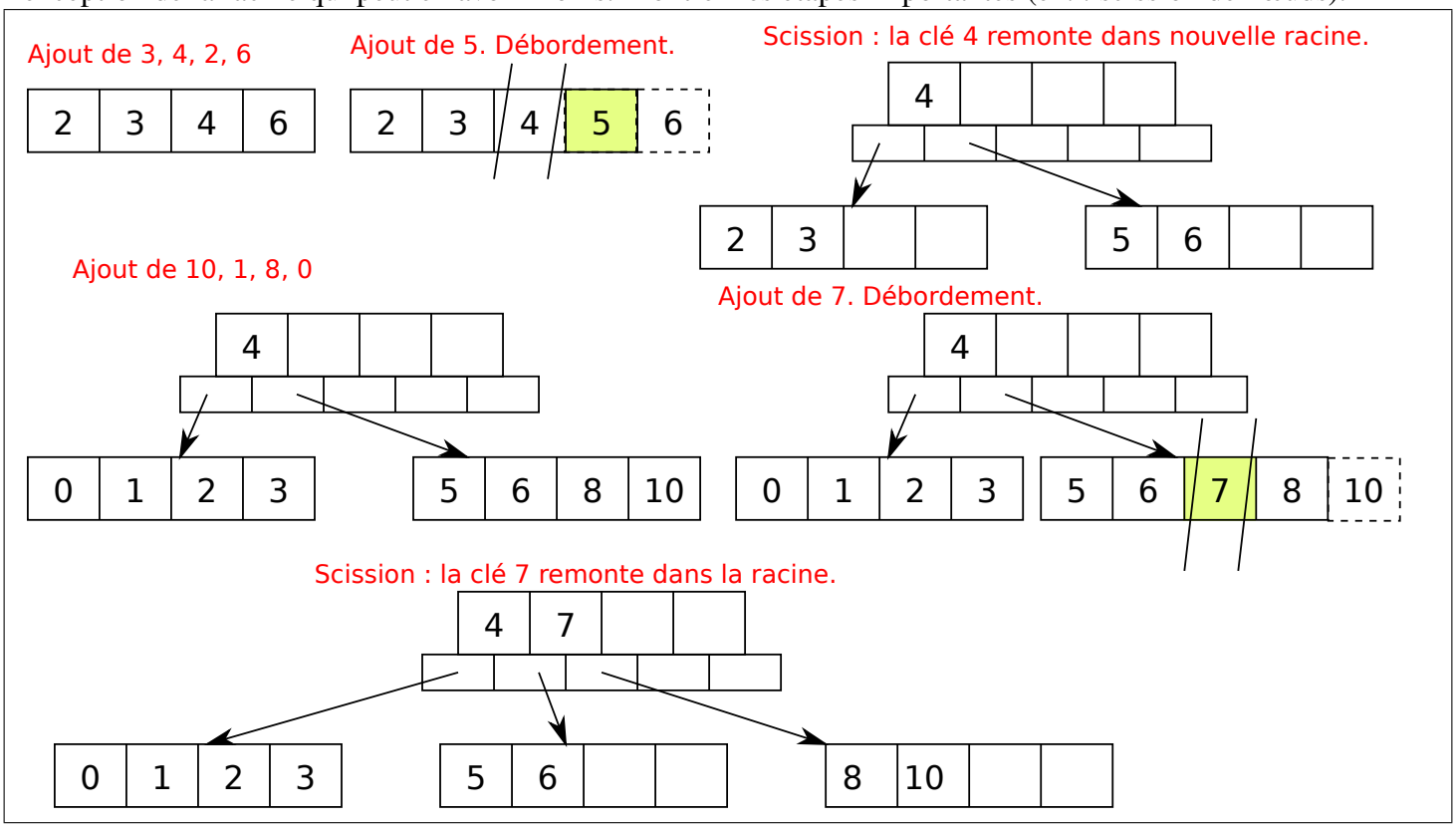
5 Arbres rouge-noir [6 points]

Insérez les nombres 1, -8, 2 et 9 dans un arbre rouge-noir initialement vide. Lorsqu'une réorganisation/recoloration de nœud(s) est requise, redessinez l'arbre afin de bien montrer les étapes intermédiaires. Considérez les directives suivantes : (1) si vous n'avez pas de crayon rouge, dessinez un cercle simple pour un nœud noir et un cercle double pour un nœud rouge ; (2) dessinez de petits carrés pleins pour représenter les sentinelles.



6 Arbres B [6 points]

Insérez les éléments 3, 4, 2, 6, 5, 10, 1, 8, 0 et 7 dans un arbre B initialement vide. L'arbre B doit être de degré 5 : les nœuds intérieurs (tous les nœuds sauf les feuilles) ont au plus 5 enfants ; les nœuds ont au moins 2 clés à l'exception de la racine qui peut en avoir moins. Montrez les étapes importantes (ex. : scission de nœuds).



7 Résolution d'un problème - Optimisation Annexe B [16 points]

Référez-vous à nouveau au programme fourni à l'Annexe B (page 10).
(a) La représentation de la classe `ListeS` ne permet pas une implémentation efficace de la fonction `f`. Proposez une nouvelle implémentation de la `ListeS` permettant une implémentation plus efficace de la fonction `f`. Montrez la nouvelle représentation et la nouvelle définition des fonctions `ajouter` et `f`. Votre solution doit utiliser des arbres binaires de recherche (ex. : ensemble `ArbreAVL` et/ou dictionnaire `ArbreMap`).

```
1  class ListeS{
2    public:
3      const string& f(const string& m);
4      void ajouter(const string& m1, const string& m2);
5    private:
6      ArbreMap<string, string> synonymes; // mot --> son synonyme
7      ArbreAVL<string> derniers;          // derniers mots utilisés (à ne pas répéter)
8  };
9  void ListeS::ajouter(const string& m1, const string& m2){
10     synonymes[m1] = m2;  // exemple : "beau" --> joli
11     synonymes[m2] = m1;  // exemple : "joli" --> beau
12 }
13 const string& ListeS::f(const string& m){
14     if(!synonymes.contient(m)) return m; // si m n'a pas de synonyme --> retourner m
15     if(derniers.enlever(m)){ // si m est dans les derniers mots utilisés
16         derniers.inserer(synonymes[m]); // noter son synonyme dans les dernier
17         return synonymes[m]; // retourner le synonyme
18     }
19     // m n'est pas dans les derniers, donc m est ok
20     derniers.inserer(m); // noter m dans les derniers
21     derniers.enlever(synonymes[m]); // enlever le synonyme des derniers
22     return m; // retourner m
23 }
```

(b) Analysez à nouveau la complexité temporelle du programme `main.cpp`, mais cette fois-ci en considérant votre nouvelle classe à la sous-question précédente. Utilisez la notation grand O. Supposez n mots lus et k paires de mots dans l'objet `liste`.

Lire les synonymes coûte $O(k)$. La boucle `while` à la ligne 9 de `main.cpp` itère n fois. Chaque itération génère un appel à la fonction `ListeS::f`. Les recherches dans `derniers` et `synonymes` coûteront au pire $O(\log k)$. Donc : $O(k + n \log k)$.
Aussi acceptée : $O(n \log k)$ si on suppose $k < n$.

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  #include <iostream>
3  class Expression{
4    public:
5      virtual double evaluer() const = 0;
6  };
7  class Plus : public Expression{
8      Expression* e1, *e2;
9    public:
10     Plus(Expression* e1_, Expression* e2_) : e1(e1_), e2(e2_){}
11     double evaluer() const;
12 };
13 class Mult : public Expression{
14     Expression* e1, *e2;
15   public:
16     Mult(Expression* e1_, Expression* e2_) : e1(e1_), e2(e2_){}
17     virtual double evaluer() const;
18 };
19 class Nombre : public Expression{
20     double n;
21   public:
22     Nombre(double n_) : n(n_){}
23     virtual double evaluer() const;
24 };
25 double Plus::evaluer() const{
26     return e1->evaluer() + e2->evaluer();
27 }
28 double Mult::evaluer() const{
29     return e1->evaluer() * e2->evaluer();
30 }
31 double Nombre::evaluer() const{
32     return n;
33 }
34 int main(int argc, char** argv){
35     Expression* e = NULL;
36     for(int i=1;i<=5;i+=2){
37         Expression* tmp = new Mult(new Nombre(i+0.0), new Nombre(i+1.0));
38         e = e==NULL ? tmp : new Plus(e, tmp);
39     }
40     double v = e->evaluer();
41     std::cout <<  v << std::endl;
42     /*** Point d'arrêt ici ***/
43     delete e;
44     return 0;
45 }
```

Annexe B pour les Questions 2 et 7

```
1  /* listes.h */
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  class ListeS{
7  public:
8    ListeS() : premier(NULL) {}
9    ~ListeS()  { /* ... */ }
10   const string& f(const string& m);
11   void ajouter(const string& m1,
12                const string& m2);
13
14 private:
15   struct S{
16     string a, b;
17     bool u;
18     void inverser();
19     S* suivant;
20   };
21   S* premier;
22 };
23
24 void ListeS::S::inverser(){
25     string t = b;
26     b = a;
27     a = t;
28 }
```

```
1  /* listes.cpp */
2  void ListeS::ajouter(const string& m1,
3                       const string& m2)
4  {
5      S* ancien = premier;
6      premier = new S();
7      premier->a=m1;  premier->b=m2;
8      premier->u=false;
9      premier->suivant=ancien;
10 }
11 const string& ListeS::f(const string& m){
12     S* s = premier;
13     while(s!=NULL){
14         if(m == s->a){
15             s->inverser();
16             s->u=true;
17             return m;
18         }
19         if(m == s->b){
20             if(s->u)
21                 s->inverser();
22             s->u=true;
23             return s->b;
24         }
25         s = s->suivant;
26     }
27     return m;
28 }
```

```
1  /* main.cpp */
2  #include "listes.h"
3  int main(int argc, char**){
4      ListeS liste;
5      liste.ajouter("beau", "joli");
6      liste.ajouter("difficile", "complexe");
7      liste.ajouter("comprendre", "déchiffrer");
8      liste.ajouter("programme", "code");
9      while(cin){
10         string s;
11         cin >> s;
12         cout << liste.f(s) << " ";
13     }
14     cout << endl;
15     return 0;
16 }
```

Annexe C pour la Question 3

Ci-dessous, vous trouverez deux implémentations d'arbres binaires de recherche (ABR). Ici, on suppose que l'ABR est équilibré, mais on fait abstraction de comment l'équilibre est maintenu.
La première implémentation (`ArbreBR1` à gauche) correspond à celle présentée en classe et dans les notes de cours. Essentiellement, un nœud d'`ArbreBR1` contient 3 attributs : le contenu de type `T` et deux pointeurs de `Noeud` nommés gauche et droite. Calculer la taille d'un objet de type `ArbreBR1` nécessite un parcours en entier de l'arbre, ce qui coûte $O(n)$ en temps (n=le nombre de nœuds). Compter le nombre de noeuds compris dans un intervalle `[min, max]` peut également se faire facilement en temps $O(n)$. Avec un peu plus d'efforts, il y a moyen de faire mieux, soit en temps $O(\log n + k)$ où k est le nombre de nœuds compris dans cet intervalle. L'astuce consiste à visiter récursivement l'arbre en évitant la visites de certains sous-arbres. On évite de visiter le sous-arbre de gauche d'un nœud lorsque son contenu est plus grand ou égal à la borne `min`. De façon similaire, on évite de visiter le sous-arbre de droite d'un nœud lorsque son contenu est plus petit ou égal à la borne `max`.
La deuxième implémentation (`ArbreBR2` à droite) ajoute un attribut `taille` dans la `struct Noeud`. Ainsi, dans chaque nœud, on conserve dans un attribut la taille du sous-arbre engendré par le nœud. En exploitant bien cet attribut, on peut arriver à compter efficacement le nombre de noeuds compris dans un intervalle `[min, max]` en temps $O(\log n)$. L'attribut `taille` dans la `struct Noeud` doit être maintenu à jour lors de nouvelles insertions.

```
1  template <class T> class ArbreBR1 {
2  public:
3    void inserer(const T&);
4    int compter(const T& min,
5                const T& max) const;
6    int taille() const;
7  private:
8    struct Noeud{
9      T contenu;
10     Noeud *gauche, *droite;
11   };
12   Noeud* racine;
13 };
```

```
1  template <class T> class ArbreBR2 {
2  public:
3    void inserer(const T&);
4    int compter(const T& min,
5                const T& max) const;
6    int taille() const;
7  private:
8    struct Noeud{
9      T contenu;
10     Noeud *gauche, *droite;
11     int taille;
12   };
13   Noeud* racine;
14 };
```

Ci-bas, voici un exemple d'arbre de type `ArbreBR2` contenant les nombres 2, 3, 5, 6, 8 et 12. Un appel à `ArbreBR1::compter(3, 10)` ou `ArbreBR2::compter(3, 10)` doit retourner 4.