FACULTÉ DES SCIENCES

Formulaire pour reprographie d'examen

Le questionnaire d'examen doit parvenir au Centre de reprographie 5 jours ouvrables avant la date de l'examen.

INFORMATIONS POUR LE SECRÉTARIAT DE LA FACULTÉ
(Attacher à chaque questionnaire)

☐ **INTRA**
☒ **FINAL**

Sigle : IFT 339

Titre : Structures de données

Professeur : Jean Goulet

Date et heure de l'examen : Mercredi 15 avril à 9 h

Nombre de pages : 12 en cahier S.V.P.

Nombre d'étudiants : ~~45~~ +2 pour AGES (45)

☐ **Je désire prendre possession des questionnaires d'examen la veille de l'activité et je me rends responsable de leur mise en sécurité.**

Signature : _____

L'étudiant devra répondre

☐ dans un cahier d'examen
☒ sur le questionnaire Impression sous forme de cahier

Veuillez inclure

_____ feuilles blanches additionnelles
_____ feuilles de papier graphique

Je consens à ce que deux (2) copies du questionnaire de cet examen soient remises à l'AGES (Association générale des étudiants en sciences) après la fin de la période des examens.

☒ **OUI** ☐ **NON**

Signature : _____

Département d'informatique
Examen final
Hiver 2015

IFT339

Structures de données

mercredi 15 avril 2015
9 h à 12 h
Jean Goulet

Cet examen comprend quatre questions. Lisez d'abord tout l'examen, puis répondez directement sur le questionnaire. Il y a en général plus d'espace que nécessaire pour vos réponses. L'examen sera corrigé sur 100, puis ramené à 40. Donnez le code demandé en C++. Essayez d'écrire le plus lisiblement possible. Quand on code avec des pointeurs, c'est toujours une bonne idée de commenter le code suffisamment pour que le correcteur comprenne ce que vous vouliez faire... et il faut utiliser avec parcimonie les doubles déréférences de pointeurs!

Ne débrochez pas ce questionnaire. Comme documentation, vous avez droit au **Résumé C++** jaune plus une feuille 8 ½ x 11 de notes personnelles. Pas de calculatrice. Il y a du papier supplémentaire pour vos brouillons.

Identifiez-vous lisiblement ci-dessous :

Nom :	
Prénom :	
Matricule :	
Signature :	

1		/20
2		/20
3		/40
4		/20
total		/100

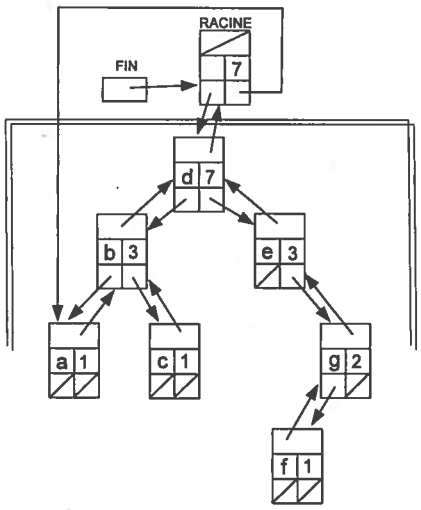
Question 1 – Implantation d'un arbre binaire de recherche en poids (20 points)

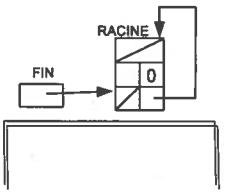
On sait qu'un arbre équilibré en poids est un peu moins efficace qu'un arbre AVL ou un Rouge et noir car on ne peut y faire une insertion ou une élimination en temps O(1) amorti. En effet, il faut toujours remonter jusqu'à la racine pour mettre les poids à jour. En revanche, un tel arbre offre la possibilité de localiser en temps O(log n) le ième élément d'un ensemble. En effet, on peut savoir si l'élément recherché est dans la partie gauche de l'arbre en examinant le poids de ce sous-arbre. Si ce poids est de 81 par exemple, c'est qu'il contient les éléments numéro 0 à 80. Le 81^{e} est la racine, et les éléments de numéros supérieurs à 81 sont dans le sous-arbre de droite.

Avec cette information, vous pouvez coder une fonction qui prend un tel numéro en entrée et qui retourne une référence constante à l'élément de ce numéro. On peut même utiliser l'opérateur [] pour simuler un vecteur. Ce n'est pas aussi efficace qu'un vecteur, où cette opération se fait en temps O(1), mais tout de même O(log n).

Voici le code de base de l'arbre équilibré en poids, où on a ajouté cette fonction publique, ainsi qu'une fonction privée qui détermine le poids du nœud identifié par un pointeur (cela évite beaucoup de if dans l'autre fonction). Codez ces deux fonctions. En gros, c'est le même code que nous avons utilisé au labo 4.

```
template <typename TYPE>
class WBT{
public:
    class iterator;
    friend class iterator;
private:
    struct noeud{
       noeud* PARENT;
       TYPE CONTENU;
       size_t POIDS;
       noeud* GAUCHE,*PARENT;
};
    noeud RACINE;
    noeud* FIN;
    void reequilibrer(noeud*&);
    void rotation_gauche_droite(noeud*&);
    void rotation_droite_gauche(noeud*&);
    //NOUVELLE FONCTION PRIVEE
    size_t poids(noeud*)const;
public:
    WBT();
    ~WBT(){clear();}
    WBT(const WBT&);
    WBT& operator=(const WBT&);
    void swap(WBT&);

    size_t size()const{return RACINE.POIDS;}
    bool empty()const{return RACINE.POIDS==0;}
    void clear();

    iterator find(const TYPE&)const;
    std::pair<iterator,bool> insert(const TYPE&);
    size_t erase(const TYPE&);
    iterator erase(iterator);

    //NOUVELLE FONCTION PUBLIQUE
    const TYPE& operator[](size_t)const;

    //fonction d'iteration
    iterator begin()const{return iterator(RACINE.DROITE);}
    iterator end()const{return iterator(FIN);}

    //fonction de mise au point
    void afficher()const;
};
```





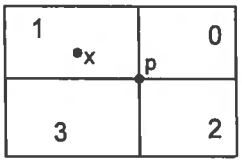
```
template <typename TYPE>
size_t WBT<TYPE>::poids(noeud* p)const{
```

```
template <typename TYPE>
const TYPE& operator[](size_t numero)const{
```

Question 2 – Construire un Quad-tree (20 points)

Dans le troisième travail pratique, vous avez codé une fonction "localiser" qui permet de trouver le point le plus proche d'une paire de coordonnées. Vous avez utilisé un Quad-tree pour cette opération, que je vous avais moi-même construit. Mais vous auriez pu facilement le faire vous-même. Si je vous fournis un fichier qui contient toutes les informations du graphe sauf les "zones QT" de chaque point, vous pouvez écrire un petit programme qui les calcule et les écrit dans le fichier. En effet, tout comme on peut lire où on veut dans un fichier binaire, on peut aussi aller écrire où on veut. Dans notre cas, puisque les valeurs des quatre zones de chaque point sont dans la partie fixe, nous n'avons jamais besoin d'aller jouer dans la partie variable. Nous pouvons sans problème aller modifier ces zones pour chaque point sans rien changer au reste.

Voici le code de base dont vous aurez besoin pour faire ce travail. C'est la même classe graphe que nous avons utilisée, un peu épurée pour n'y inclure que les éléments pertinents à ce problème. On vous donne aussi la fonction construire_QT. Voici l'algorithme suivi: le point 0 est toujours la racine de l'arbre. On insère d'abord tous les numéros des autres points dans un unordered_set<uint32_t> pour ensuite les traiter un par un. Pourquoi un tel conteneur? C'est une façon de les obtenir dans un ordre aléatoire, ce qui nous permet de croire qu'ils seront bien répartis et que l'arbre résultant sera assez équilibré. En effet, en itérant sur cet ensemble, on aura les numéros de nœuds en désordre. Ensuite, il suffit d'insérer chacun des points. Pour chaque point, la procédure est la même que pour l'insertion dans un arbre binaire. On démarre à la racine. On détermine dans quelle zone on doit aller à partir de là. S'il n'y a aucun point dans cette zone (la valeur QT pour cette zone est à 0), on peut y mettre le numéro du point insérer. Sinon, il faut poursuivre avec ce point jusqu'à ce qu'on arrive à une zone vide. Avant de tenter d'accéder à un nœud, il faut s'assurer que les données du nœud sont dans le map en utilisant la fonction lire_noeud. Une fois modifiée la valeur d'une zone QT d'un nœud, il faut aller réécrire les valeurs QT de ce nœud dans le fichier avec la fonction reecrire_QT. On pourrait alors gérer la possibilité que le map lesNoeuds devienne trop gros en effaçant les données de ce nœud. Mais on ne s'occupera pas de cela ici. Votre travail ici ne consiste qu'à écrire la fonction inserer_QT, qui reçoit un numéro de nœud et doit aller le placer dans la zone QT appropriée d'un nœud qui a déjà été inséré. En guise de rappel, vous avez ici la numérotation utilisée pour les zones.




```
class graphe{
private:
   struct noeud{  //description de toutes les composantes d'un nœud
      float LATITUDE, LONGITUDE;
      vector<uint32_t> zones_QT;     //les 4 valeurs pour ce nœud
      ...                            //autres valeurs pas utiles ici
      };
   map<uint32_t,noeud> lesNoeuds;    //les noeuds deja lus
   uint32_t nbNOEUDS;                //le nombre total de nœuds du graphe
   ifstream DATA;                    //le flot d'entree
   uint32_t DEBUT;                   //adresse de debut de la partie fixe
   void lire_noeud(uint32_t);
   void reecrire_QT(uint32_t);       //ne pas coder cette fonction
   void inserer_QT(uint32_T);        //CODEZ CETTE FONCTION!
   ...
public:
   graphe(string);                                  //constructeur
   ~graphe();                                       //destructeur
   uint32_t size()const;                            //nombre de noeuds dans le graphe
   void afficher_noeud(uint32_t);
   void construire_QT();
   ...
};
```

```
void graphe::constuire_QT(){
   lire_noeud(0);
   unorderd_set<uint32_t> S;
   for(uint32_t i=1;i<size();++i)
      S.insert(i);
   for(auto j=S.begin();j!=S.end();++j){
      uint32_t noeud_a_inserer=*j;;
      lire_noeud(noeud_a_inserer);
      inserer_QT(noeud_a_inserer);
      }
   }


void graphe::inserer_QT(uint32_t numero){
```

Question 3 – L'équilibre des arbres de recherche (40 points)

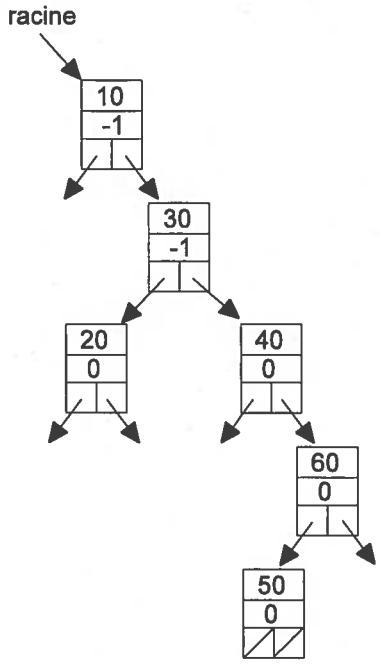
Dans chacun des cas ci-dessous effectuez l'opération demandée sur la structure qui vous est donnée et dessinez l'arbre résultant. S'il y a lieu, montrez les étapes intermédiaires. Avant de commencer un problème, essayez d'identifier le plus d'information possible sur les sous-arbres que vous ne voyez pas. **Rappels:**

Dans un AVL, l'indice est la hauteur du sous-arbre de gauche moins la hauteur du sous-arbre de droite.

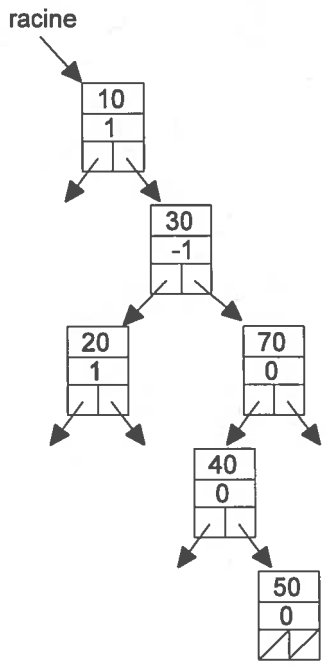
Dans un arbre équilibré en poids, on ne s'occupe pas de l'équilibre d'un nœud dont le poids est inférieur ou égal à 3, mais on doit rééquilibrer un nœud dont un des sous-arbres a un poids strictement supérieur à trois fois celui de l'autre. On doit faire une double rotation si une rotation unique cause un déséquilibre de l'autre côté du nœud.

Dans les arbres rouges et noirs, les arcs rouges sont représentés par des traits doubles et les pointeurs nuls sont des arcs noirs.

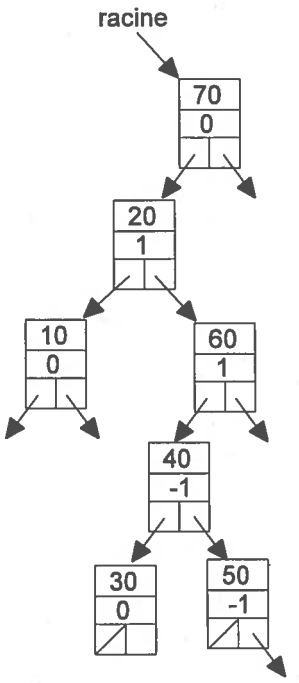
A- Ajoutez 45 dans l'arbre AVL suivant:



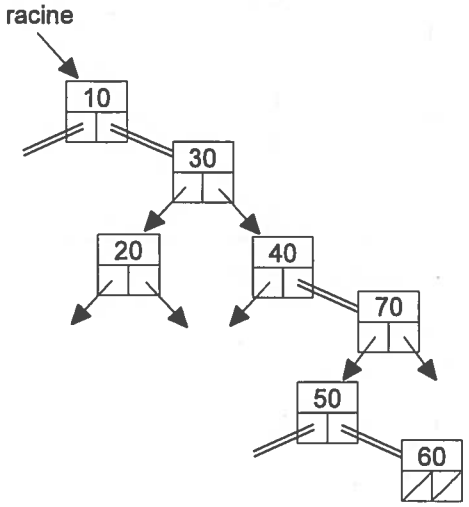
B- Ajoutez 45 dans l'arbre AVL suivant:



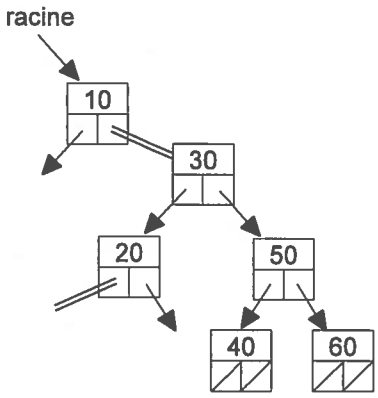
C- Enlevez 20 dans l'arbre AVL suivant:



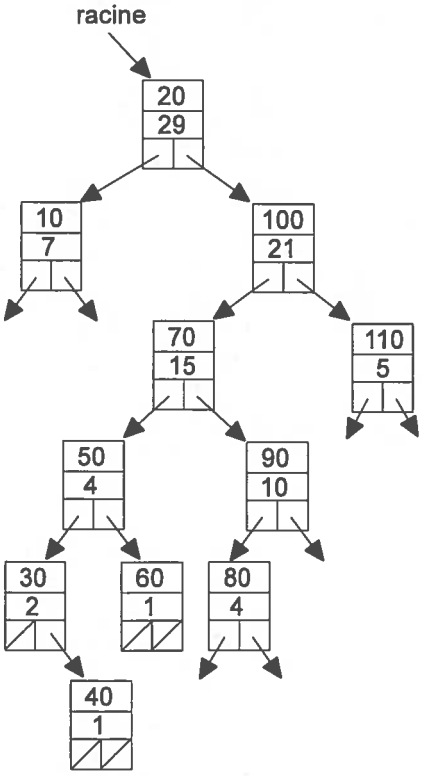
D- Ajoutez 55 dans l'arbre rouge et noir suivant:



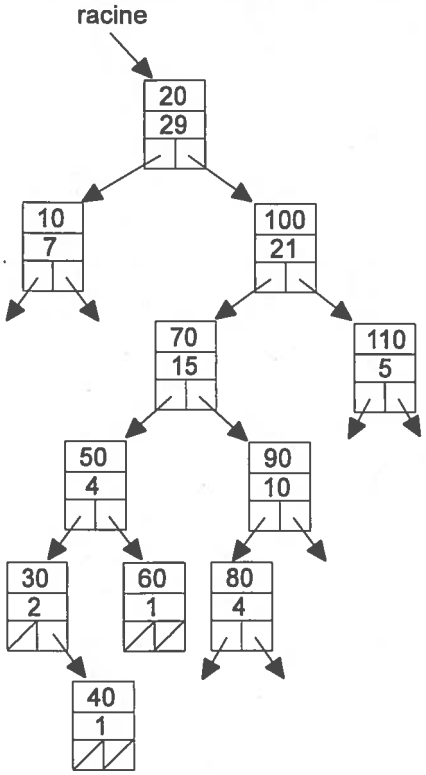
E- Enlever 60 dans l'arbre rouge et noir suivant:



F- Ajouter 65 dans l'arbre équilibré en poids suivant:



G- Enlevez 20 dans l'arbre équilibré en poids suivant:



H- Ajoutez 35 dans le monceau suivant:

44 33 39 17 26 12 28 12 8 4 2 4 2 5 19 5

Question 4 – Quelques éléments techniques ou théoriques divers (20 points)

A- Complétez le tableau suivant, qui donne le nombre minimal d'éléments que peut contenir un arbre équilibré selon les règles de la question 3. La première colonne du tableau est la hauteur de l'arbre, la deuxième est le nombre minimal d'éléments pour un arbre AVL de cette hauteur, la troisième le nombre minimal d'éléments pour un arbre rouge et noir (ARN) de cette hauteur, la quatrième le nombre minimal pour un arbre équilibré en poids (WBT) de cette hauteur. Quelques valeurs du tableau vous sont données à titre d'exemples.

h	AVL	ARN	WBT
0	0	0	0
1	1	1	1
2	2	2	2
3			
4			
5			
6	20	14	12
7	33	22	17

Pour justifier vos réponses, dessinez un exemple de chacun des neufs arbres (juste la structure, pas besoin de mettre des valeurs dans les nœuds).

B Convertissez la valeur -38.12 dans un float

Convertir d'abord 38.12 en base 2 avec environ 25 chiffres significatifs (de façon à pouvoir procéder correctement à l'arrondi

_____ . _____

Exprimez cette valeur sous forme de nombre normalisé multiplié par une puissance de 2:

1 . _____

_____

x 2

Donnez maintenant les bits des trois composantes du float (le tout devrait donner 32 bits):

le signe: ____

la caractéristique: _____

la mantisse:

_____

C- Ordre de croissance de la complexité algorithmique

Donnez l'ordre de croissance du **pire cas** de la complexité des fonctions suivantes de la bibliothèque normalisée de C++. **Justifiez succinctement.**

La fonction find de la class map _____

La fonction find de la class unordered_map _____

La fonction operator++ des iterateurs de map _____

La fonction insert avec indice de la classe map _____

Fin de l'examen (APRÈS la dernière question!)
(Utilisez les feuilles fournies à part pour vos brouillons)