DEVUIP Service Scolarité	**ANNEE UNIVERSITAIRE 2012/2013** **DEUXIEME SESSION** PARCOURS / ETAPE : LIIN300 Code UE : J1IN3001 **Epreuve :** Algorithmique et structures de données 1 **Date :** **Heure :** **Durée :** 1H30 Documents : non autorisés Epreuve de Mme : DELEST **Vous devez répondre directement sur le sujet qui comporte 8 pages. Insérez ensuite votre réponse dans une copie d'examen comportant tous les renseignements administratifs**	

Indiquez votre code **d'anonymat** : N° : .

La notation tiendra compte de la clarté de l'écriture des réponses.

Barème indicatif

- Question 1 – Connaissances générales : 2 points
- Question 2 – Containeur liés aux arbres binaires : 6 points
- Question 3 – Connaissance des structures de données : 3 Points
- Question 4 – Connaissance des structures de données : 3 Points
- Question 5 – Utilisation des structures de données : 6 points -

Question 1. Cochez les affirmations correctes et quelle que soit la réponse justifiez sur les lignes en pointillé.

☒ Un arbre binaire de recherche est de type containeur.

Oui car il stocke des valeurs et on a bien les primitives propres aux containeurs.

☐ Une file de priorité est de type file.

Non, une file de priorité est un contaneur de type arbre binaire

☒ La structure de pile permet l'accès au dernier élément entré dans le containeur.

Oui, car uniquement le sommet de la pile est accessible en O(1).

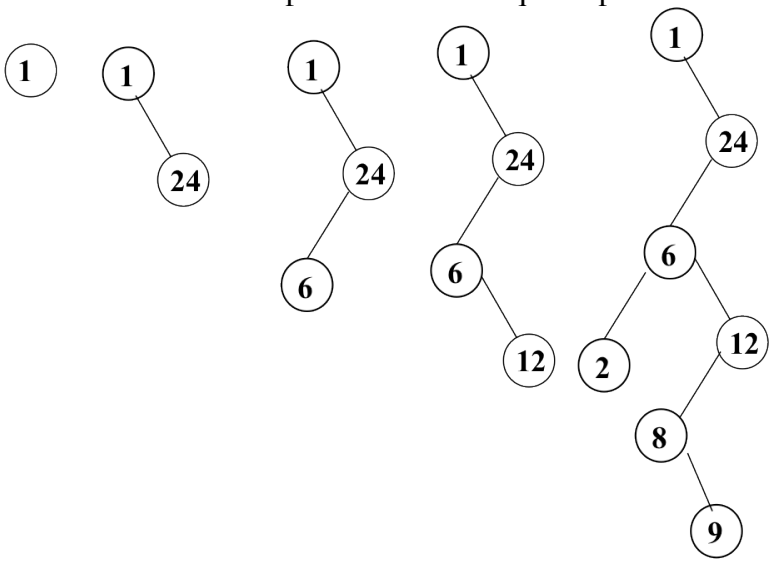
……

☒ Si s est une structure contenant un champs p qui est un pointeur vers une structure contenant un champs k entier, on accède à l'entier par s.p^.k ?

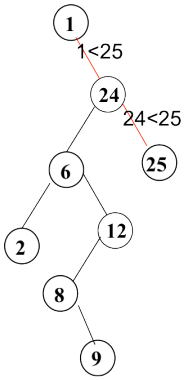
Oui, S désigne la structure comme p est un champ s.p désigne le pointeur, s.p^ désigne donc la structure pointée par p. Enfin k est un des champs de cette dernière structure donc on ajoute « .k »

Question 2. Soit la liste de clé A=(1,24,6,12,8,9,2).

1 - Construire l'arbre binaire de recherche T correspondant à l'insertion consécutive des clés de la liste A. On dessinera l'arbre après chacune des quatre premières insertions ainsi que l'arbre final.



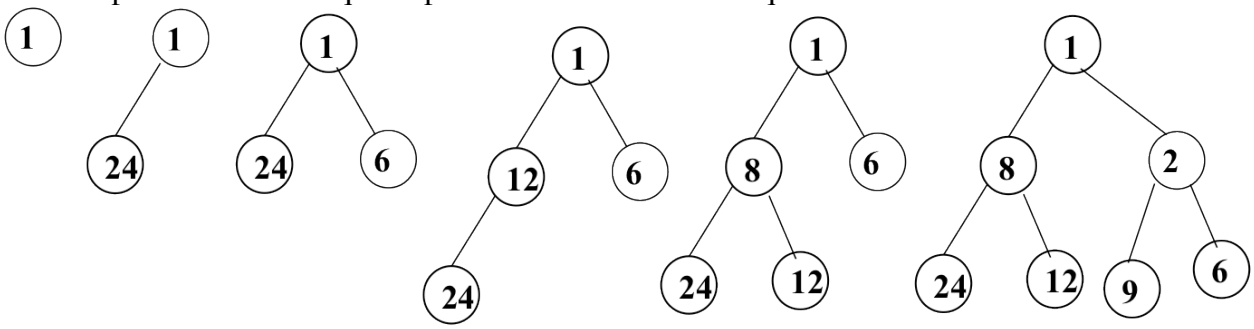
2 - Montrez l'exécution de l'ajout de la clé 25 sur l'arbre T.



2 - Donner la fonction *ajouter* qui ajoute un élément dans un arbre binaire de recherche.

```
fonction ajouter(ref x:sommet, val e:objet):vide;
    var s:sommet;
    début
        si e < getValeur(x) alors
            s=filsGauche(x);
            si s==NIL alors
                ajouterFilsGauche(x,e);
            sinon
                ajouter(s,e);
            finsi
        sinon
            s=filsDroit(x);
            si s==NIL alors
                ajouterFilsDroit(x,e);
            sinon
                ajouter(s,e);
            finsi
        finsi
    fin
```

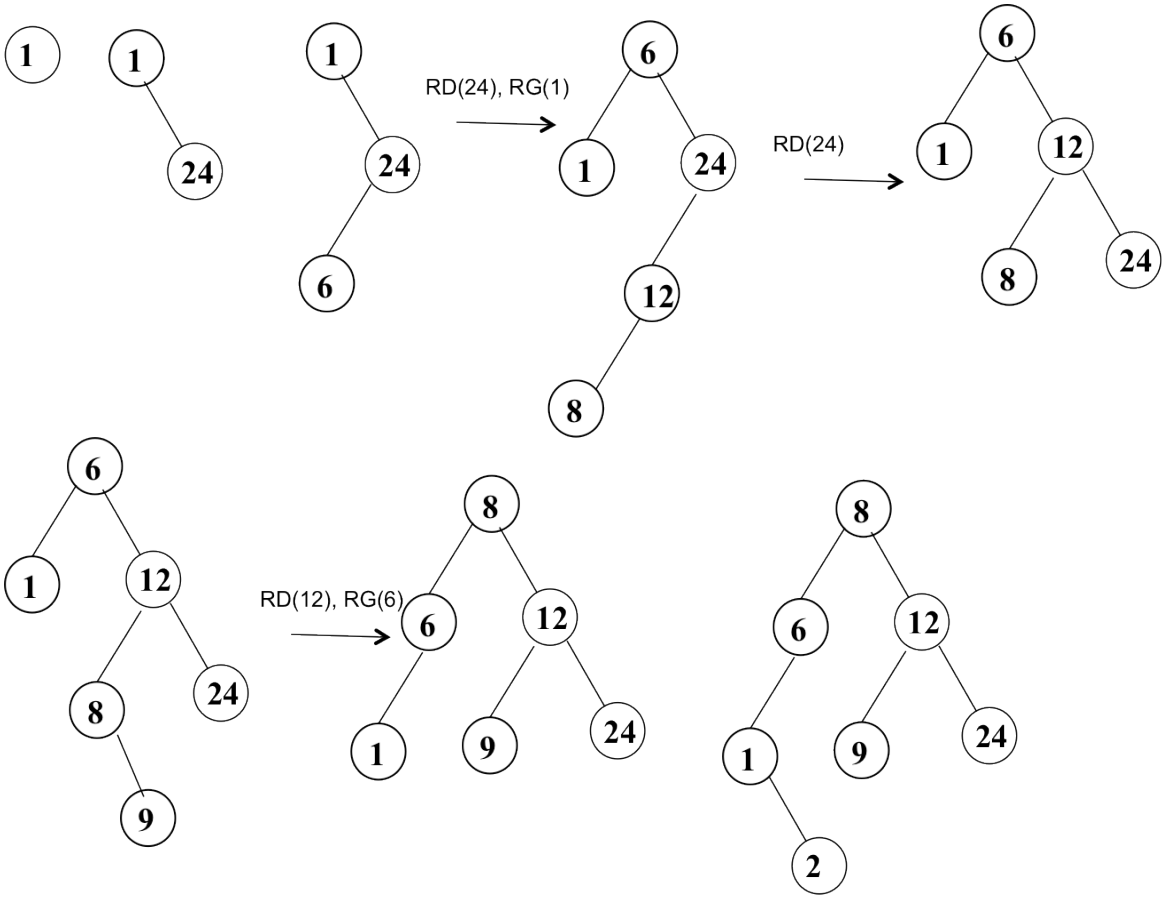
3 – Construire le tasMin B correspondant à l'insertion consécutive des clés de la liste A. On dessinera l'arbre après chacune des quatre premières insertions ainsi que l'arbre final.



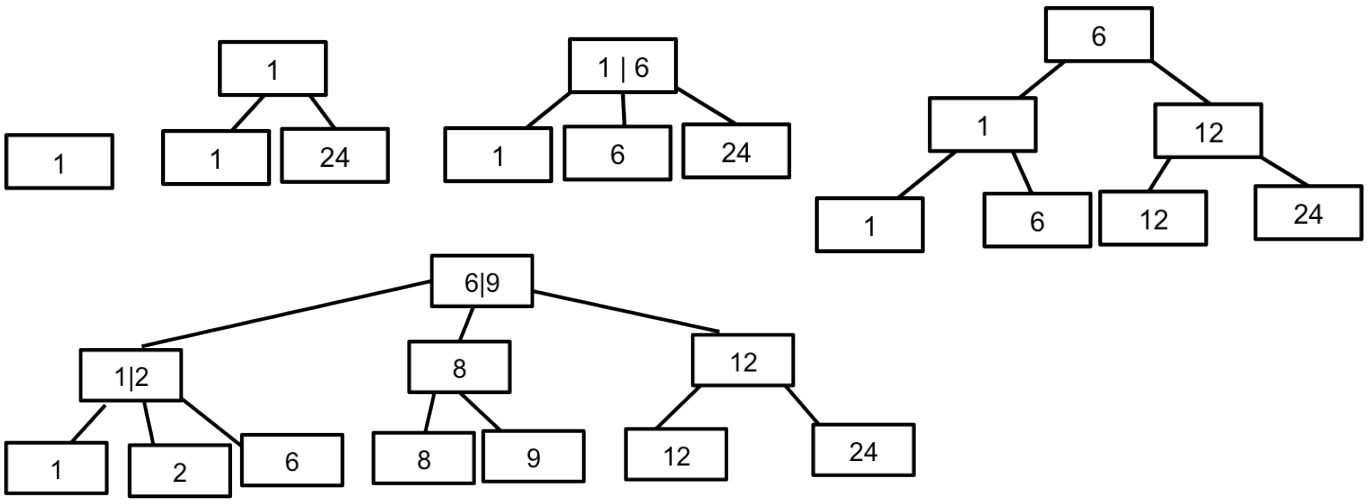
4 – Donner la liste des sommets en ordre préfixe, en ordre infixe et en ordre suffixe de l'arbre B.

Préfixe :1,8,24,12,2,9,6
Infixe : 24,8,12, 1,9,2,6
Suffixe : 24,12,8,9,6,2,1

5 – Construire l'arbre binaire de recherche AVL correspondant à l'insertion consécutive de la liste A. On dessinera l'arbre après chacune des quatre premières insertions ainsi que l'arbre final.



6 - Construire le 2-3 arbre correspondant à l'insertion consécutive de la liste A. On dessinera l'arbre après chacune des quatre premières insertions ainsi que l'arbre final.



Question 3. Soit une suite de clés de l'ensemble {0,1} stockées dans une liste L. Par exemple L=(0, 0,1,0,0,1,1,1).

1 – Ecrire la fonction booléenne *verifier* qui retourne vrai quand dans la liste L, il y a autant de valeurs 0 que de valeurs 1. Elle retourne faux dans le cas contraire. On utilisera les primitives du type abstrait listeSC.

```
fonction verifier(ref L:listeSC d'entiers):booleen;
    var cpt:entier;
    début
        cpt=0;
        debutListe(L);
        tantque !finListe(L) faire
            si valeur(L)!=0 alors
                cpt0=cpt0-1
            sinon
                cpt0=cpt0+1
            finsi
            suivant(L)
        fintantque
        retourner (cpt=0)
    fin
```

2 – On souhaite disposer d'une fonction *complete* qui complète la liste L en fin de liste après que la fonction *verifier* retourne faux. Quel type de liste choisir pour que chaque élément soit examiné une seule fois?

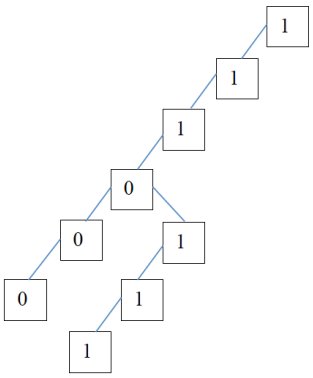
Il faut une liste doublement chainée sinon la fonction complete devra repositionner la clé sur le dernier élément (et donc balayer la liste) pour ajouter le nombre de 0 ou 1 nécessaire. Soit on inclus el code de la fonction vérifier dans la fonction complete soit la fonction vérifier renvoie le nombre de 0 ou 1 manquant donc un entier. Si c'est entier est négatif, ce sera le nombre de zéro sinon ce sera le nombre de 1.

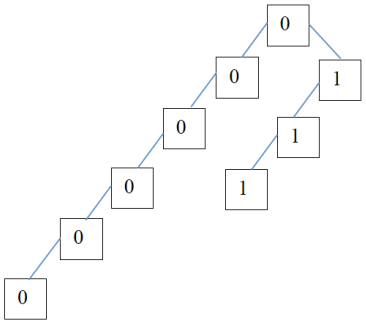
3 – Ecrire la fonction *complete* en utilisant le type abstrait listeDC.

```
fonction complete(ref L:listeDC d'entiers ; ref cpt :entier):vide;
    debut
        si cpt>alors
            completeBis(L,cpt,0)
        sinon
            completeBis(L,-cpt,1)
        finsi
    fin
fonction completeBis(ref L:listeDC d'entiers ; ref cpt,val :entier):vide;
    début
        finListe(L);
        tantque cpt>0 faire
            insererApres(L, val);
            cpt=cpt-1;
            suivant(L)
        fintantque
    fin
```

4 – On suppose ces clés stockées dans un arbre binaire de recherche. Tous les arbres auront seulement deux formes caractéristiques. Lesquelles ? Donnez un exemple dans chaque cas.

Les arbres auront au plus deux branches. Si l'arbre à deux branches, il y aura un seul nœud étiqueté 0 qui a un fils gauche 0 et un fils droit 1. Tous les autres nœuds n'auront pas de fils à droite pas de fils à droite 0





5 – Ecrire la fonction *sommetZero* qui à partir d'un tel arbre, fournit le sous arbre des sommets de valeur 0.

```
fonction sommetZero(ref A:arbreBinaire d'entier): sommet;
    var s:sommet;
    debut
        s=pere(A);
        tantque valeur(s)!=0 faire
            s=filsGauche(s)
        fintantque
        retourner(s)
    fin
```

Question 4. Ecrire une fonction qui prend en entrée une pile d'entiers et qui fournit en sortie une file telle que tous les nombres pairs se retrouvent en tête de la file. La pile de départ reste inchangée. Par exemple, la pile [1,3,4,5,2,6,8[de sommet de pile 8, fournit en sortie en]8,6,2,4,1,3,5 [de tête de file 8.

```
fonction pileFile(val P: pile d'entier): file d'enties;
    var F: file d'entier ;
    var PC: pile d'entier ;
    début
        creerFile(F);
        creerPile(PC);
        tantque !pileVide(P) faire
            si estPair(valeur(P)) alors
                enfiler(F,valeur(P));
            finsi
            empiler(PC,valeur(P));
            depiler(P);
        fintantque
        tantque !pileVide(PC) faire
            si estImpair(valeur(PC)) alors
                enfiler(F,valeur(PC));
            finsi
            empiler(P,valeur(PC));
            depiler(PC);
        fintantque
        retourner(F)
    fin
```

Question 5. On souhaite gérer un ensemble de photos rectangulaires. Chaque photo est décrite par un ensemble d'informations : les dimensions du rectangle, une chaine de caractère donnant le chemin d'accès, une date de prise de vue (jour, mois, année), un index qui la classifie suivant trois catégories (famille, travail, loisir). Dans la suite, on décrira précisément chaque structure de donnée. On précisera s'il s'agit ou non d'un containeur. On ne demande pas d'écrire les primitives mais uniquement de donner l'en-tête de la primitive et son rôle.

1 – Décrivez la structure de données *date* permettant de stocker une date. En donner les primitives.

date=structure
jour,mois,an :entier
finstructure
Ce n'est pas un containeur. Les primitives sont des get et set.
fonction getX(val D :date) :entier
fonction setX(ref D :date ;val v :entier) :vide
où X est l'un des champs jour,mois,an.

2 – Décrivez la structure de données *photo* permettant de décrire les informations liées à une photo. En donner les primitives.

dimension=structure
largeur,longueur :réel
finstructure
photo=structure
di :dimension ;
c :chaine de caractères ;
da :date ;
*classe :entier ;/*0=famille,1=travail,2=loisir*/*
finstructure
Ce n'est pas un containeur. Les primitives sont des get et set.
fonction getdi(val P :photo) :dimension ;
fonction setdi(ref P :photo;val d :dimension) :vide
fonction getc(val P :photo) :chaine de caractère;
fonction setc(ref P :photo;val d :chaine de caractère) :vide
fonction getda(val P :photo) :date;
fonction setda(ref P :photo;val d :date) :vide
fonction getclasse(val P :photo) :entier;
fonction setclasse(ref P :photo;val d :entier) :vide

3 – Quelle est la structure de donnée que l'on peut utiliser pour l'ensemble des photos d'un album si on souhaite répondre en temps constant à la question « quelle est la photo prise la plus récemment ». Justifiez. Décrivez la clé d'accès.

Un tas min permet de placer un des éléments de clé minimum en tête de l'arbre. Il suffit donc de prendre comme clé d'accès la date. On peut transformer une date en entier aisément. Il n'y a donc pas besoin d'implémenter une fonction de comparaison de dates. Dans tous les cas, on ajoute un champ à chaque sommet qui est soit de type photo soit un pointeur vers une cellule de type photo.

4 – Décrire la structure de données *albumPhoto* qui permet de stocker un ensemble de photos en prenant en compte les questions précédentes.

albumPhoto= tasMin de cellules ;
cellule=structure
clé :entier ;
p :photo ;
finstructure

5 – Ecrire la fonction *ajouterPhoto* qui prend comme paramètre une variable de type *photo* et permet d'ajouter la photo dans une variable de type *albumPhoto*. On ne demande pas d'implémenter les primitives vues en cours mais de les utiliser.

On suppose existant
fonction dateToEntier(d :date)entier ;
On suppose également que la fonction ajouter prend en argument l'objet à stocker ainsi que la clé
fonction ajouter(ref T:tasMin de cellules, val v:objet ;val c :entier):vide;
On a alors

```
fonction ajouterPhoto(ref A :albumPhoto ; val p :photo) :vide ;
    val n :entier ;
    début
        n=dateToEntier(p.da) ;
        ajouter(A, p,n)
    fin
```

Listes simplement chainées (listeSC)

fonction valeur(val L:liste d'objet):objet;
fonction debutListe(val L:liste d'objet) :vide ;
fonction suivant(val L:liste d'objet) :vide ;
fonction listeVide(val L:liste d'objet): booleen;
fonction créerListe(ref L:liste d'objet):vide;
fonction insérerAprès(ref L:liste d'objet; val x:objet;):vide;
fonction insérerEnTete(ref L:liste d'objet val x:objet):vide;
fonction supprimerAprès(ref L:liste d'objet):vide;
fonction supprimerEnTete(ref L:liste d'objet):vide;
fonction detruireListe(ref L:liste d'objet):vide;

Listes doublement chainées (listeDC)

fonction finListe(val L:liste d'objet):vide;
fonction précédent(val L::liste d'objet): vide;

Piles

fonction valeur(ref P:pile de objet):objet;
fonction pileVide(ref P:pile de objet):booléen;
fonction créerPile(P:pile de objet) :vide
fonction empiler(ref P:pile de objet;val x:objet):vide;
fonction dépiler(ref P:pile de objet):vide;
fonction detruirePile(ref P:pile de objet):vide;

Files

fonction valeur(ref F:file de objet):objet;
fonction fileVide(ref F:file de objet):booléen;
fonction créerFile(F:file de objet);vide;
fonction enfiler(ref F:file de objet;val x:objet):vide;
fonction défiler (ref F:file de objet):vide;
fonction detruireFile(ref F:file de objet):vide;

Arbres binaires

fonction getValeur(val S:sommet):objet;
fonction filsGauche(val S:sommet):sommet;
fonction filsDroit(val S:sommet):sommet;
fonction pere(val S:sommet):sommet;
fonction setValeur(ref S:sommet;val x:objet):vide;
fonction ajouterFilsGauche(ref S:sommet,val x:objet):vide;
fonction ajouterFilsDroit(ref S:sommet,x:objet):vide;
fonction supprimerFilsGauche(ref S:sommet):vide;
fonction supprimerFilsDroit(ref S:sommet):vide;
fonction detruireSommet(ref S:sommet):vide;
fonction créerArbreBinaire(val Racine:objet):sommet;
fonction detruireArbreBinaire(val Racine:objet):sommet;

Arbres planaires

fonction valeur(val S:sommetArbrePlanaire):objet;
fonction premierFils(val S:sommetArbrePlanaire):sommetArbrePlanaire;
fonction frere(val S:sommetArbrePlanaire):sommetArbrePlanaire;
fonction pere(val S:sommetArbrePlanaire):sommetArbrePlanaire;
fonction créerArborescence(val racine:objet):sommetArbrePlanaire;
fonction ajouterFils(ref S:sommetArbrePlanaire,val x:objet):vide;
fonction supprimerSommet(ref S:sommetArbrePlanaire):vide;
fonction detruireArborescence(val racine:objet):sommetArbrePlanaire;

Arbres binaire de recherche

fonction ajouter(ref A:arbreBinaire d'objets, val v:objet):vide;
fonction supprimer(val A: arbreBinaire d'objets, val v:objet):vide;

Tas

fonction valeur(ref T:tas d'objet): objet;
fonction ajouter(ref T:tas de objet, val v:objet):vide;
fonction supprimer(val T:tas de objet):vide;
fonction creerTas(ref T:tas,val:v:objet):vide;
fonction detruireTas(ref T:tas):vide;

File de priorité

fonction changeValeur(ref T:tas d'objet,val s:sommet,val v:objet):vide;

Dictionnaire

fonction appartient(ref d:dictionnaire,val M::mot):booléen;
fonction creerDictionnaire(ref d: dictionnaire):vide ;
fonction ajouter(ref d:dictionnaire,val M::mot):vide;
fonction supprimer(ref d:dictionnaire,val M:mot):vide;
fonction detruireDictionnaire(ref d:dictionnaire):vide;

Table de Hachage

fonction chercher(ref T:tableHash de clés, val v:clé):curseur;
fonction créerTableHachage(ref T: tableHash de clé, ref h:fonction):vide;
fonction ajouter(ref T:tableHash de clé,val x:clé):booleen;
fonction supprimer((ref T:tableHash de clé,val x:clé):vide;
fonction detruireTableHachage(ref T: tableHash de clé, ref h:fonction):vide;

FIN