DISVE Pôle Licence	**ANNEE UNIVERSITAIRE 2009/2010** **SESSION 1 D'AUTOMNE** **PARCOURS / ETAPE :** CSB3, MHT53 **Code UE :** INF 251 **Epreuve :** Algorithmes et Structures de Données Fondamentaux **Date :** 21 Décembre **Heure :** 8H30 **Durée :** 1H30 Documents : non autorisés Epreuve de M/Mme : DELEST **Vous devez répondre directement sur le sujet qui comporte 10 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 de recherche : 6 points
- Question 3 – Connaissance des structures de données : 6 Points
- Question 4 – Utilisation des structures de données : 6 points

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

☐ Dans un arbre binaire de recherche le minimum est toujours à la racine.

La valeur de la racine est plus grande (resp plus petite) que les valeurs du soius arbre gauche(resp. droit).

☒ Un arbre AVL de taille 15 n'est jamais un tas.

C'est un arbre quasi parfait dont la valeur de la racine est le min ou le max et un AVL est un ABR

☒ La structure de B-arbre permet de diminuer le temps d'accès à un élément par rapport à un AVL.

Toutes les valeurs sont au plus à hauteur $\log_b(n)$ …

☒ Si s est un pointeur vers une structure dont un des champs est n : ^entier, on accède à l'entier par s^**.**n^ ?

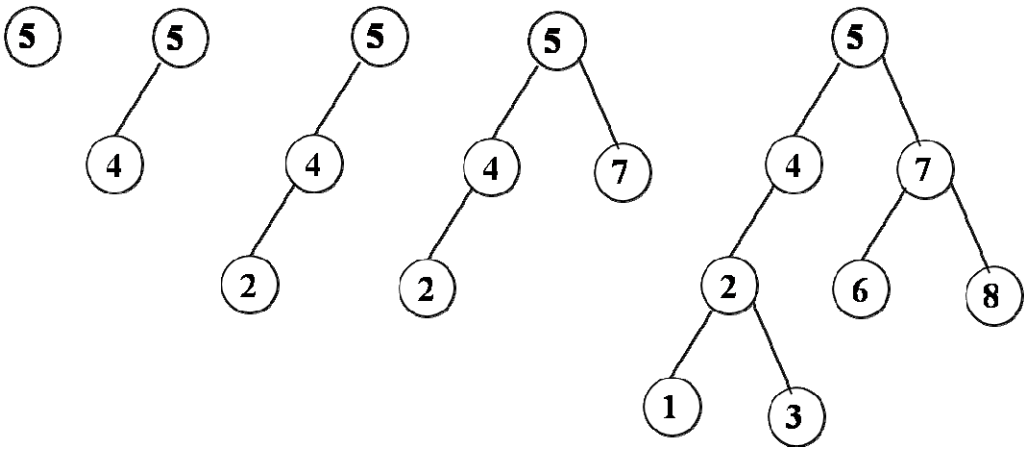
On part de s puis ^ car s est un pointeur puis . car n est un champ et enfin ^ car n est un pointeur

Question 2. Soit la liste de clé A=(5,4,2,7,6,8,3,1).

1 – Ecrire la fonction *ajouter* qui insère un élément dans un arbre binaire de recherche.

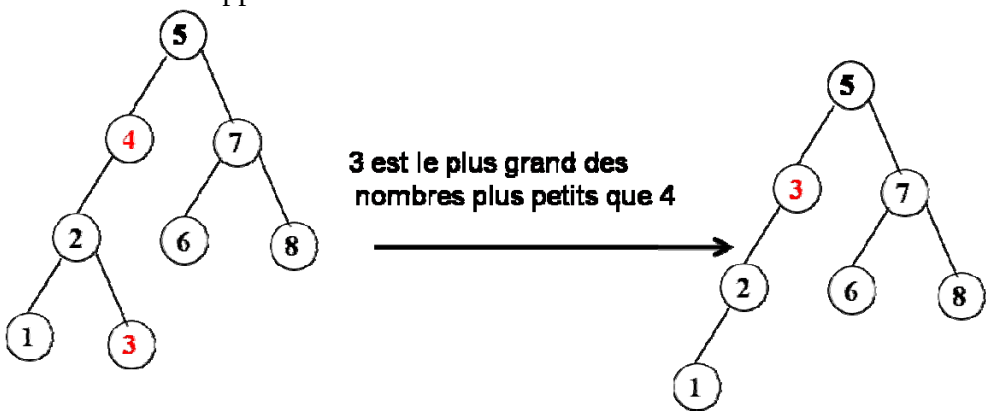
```
fonction ajouter(ref x:sommet, val e:objet):vide;
  var s:sommet;
  début
    si e < valeurSommet(x) alors
      s=filsGauche(x);
      si s==NIL alors
        ajouterFilsGauche(x,e);
      sinon
        ajouter(s,e);
      finsi
    sinon
      s=filsDroit(x);
      si s==NIL alors
        ajouterFilsDroit(x,e);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

2 - Construire l'arbre binaire de recherche 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.

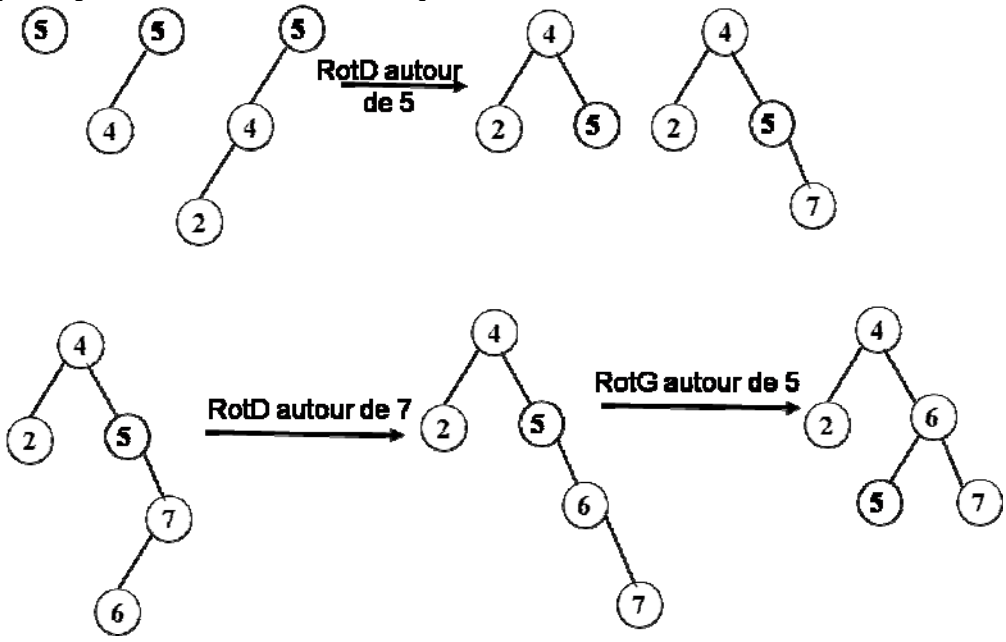


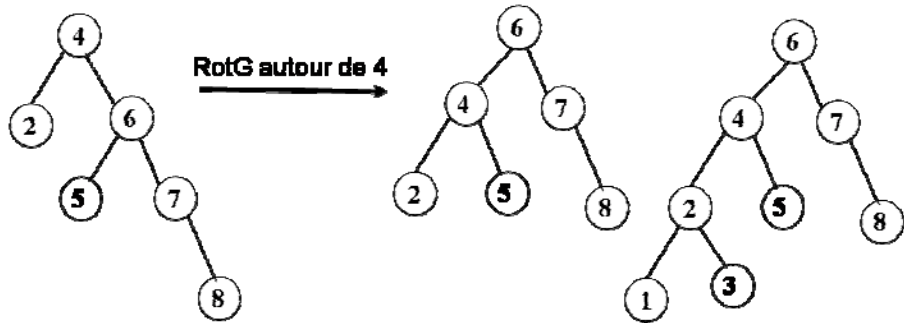
3 – Donnez pour l'arbre B la liste des sommets en ordre préfixe, en ordre infixe et en ordre suffixe.
Ordre préfixe : 5,4,2,1,3,7,6,8
Ordre infixe : 1,2,3,4,5,6,7,8
Ordre suffixe : 1,3,2,4,6,8,7,5

4 - Montrez l'exécution de la suppression de la clé 4 sur l'arbre B.

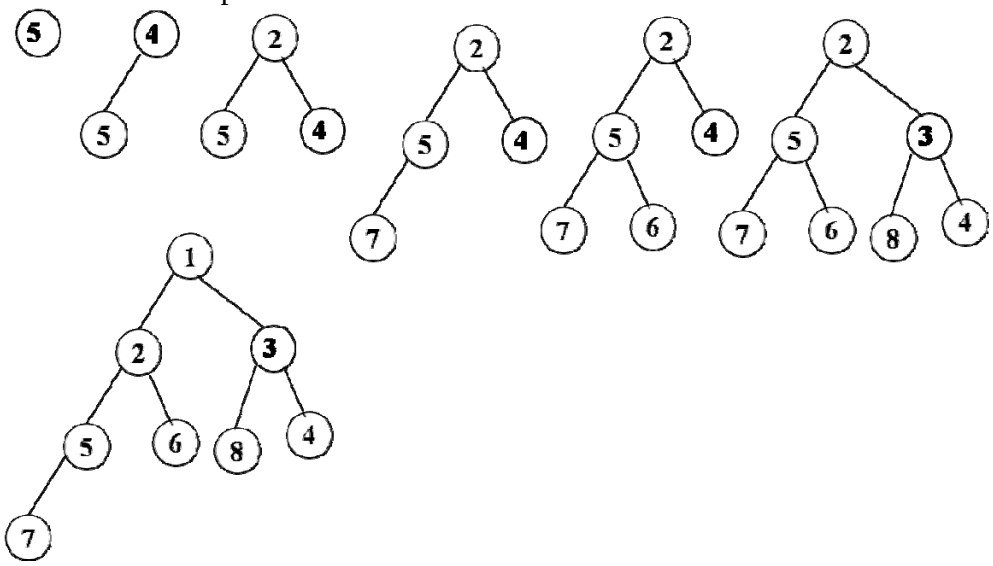


5 – Construire l'AVL correspondant à l'insertion consécutive de la liste A. On dessinera l'arbre après chacune des quatre premières insertions ainsi que l'arbre final.





6 – Construire le tas Min correspondant à l'insertion consécutive de la liste A. On dessinera l'arbre final.



Question 3. Soit une suite de clés dans un tableau T. On considère un tableau de dimension N contenant des entiers tous différents appartenant à l'intervalle [1..N]. Par exemple, le tableau de dimension 8 est donné par T[1]=5, T[2]=8, T[3]=2, T[4]=6, T[5]=3, T[6]=4, T[7]=1, T[8]=7.

1 – Ecrire la fonction *verifier* qui vérifie que les éléments du tableau appartiennent à l'intervalle [1..N] et sont tous différents.

```
fonction verifier(ref T : tableau[1..N]d'entiers) :booleen ;
    var B :booleen ;
    var i : entier ;
    début
        pour i allant de 1à N faire
            B[i]=faux ;
        finpour
        pour i allant de 1à N faire
            si T[i] <1 ou T[i]>N alors
                retourner(faux)
            sinon
                si !B[T[i]] alors
                    B[T[i]]=vrai ;
                sinon
                    retourner faux
                finsi
            finsi
        finpour
        retourner (vrai)
    fin
```

2 – En utilisant les primitives du type abstrait listeSC, écrire la fonction *tableauListe* qui a pour paramètre un tableau d'entier T et fournit en sortie une liste simplement chainée L dans le même ordre que le tableau d'entrée (sur l'exemple, L= (5,8,2,6,3,4,1,7)).

```
fonction tableauListeSC(ref T : tableau[1..N]d'entiers) :listeSC d'entiers ;
    var i : entier ;
```

```
var L:listeSC d'entiers;
début
    creerListeSC(L) ;
    pour i allant de N à 1  faire
        insererEnTete (L,T[i])
    finpour
    retourner(L)
fin
```

3 - Ecrire cette même fonction *tableauListe* en utilisant l'implémentation dynamique à la place des primitives

```
fonction tableauListe(ref T : tableau[1..N]d'entiers) :listeSC d'entiers ;
    var i : entier ;
    var L:listeSC d'entiers;
    var c,tmp:curseur;
    début
        tmp=NIL ;
        pour i allant de N à 1 faire
            new(c) ;
            c^.valeurElement= T[i] ;
            c^.pointeurSuivant=tmp ;
            tmp=c;
        finpour
        L.premier=c ;
        retourner(L)
    fin
```

4 – Ecrire une fonction *creuxDeListe* qui à partir de la liste L simplement chainée fournit la file des valeurs T[j] telles que T[j-1]>T[j] et T[j]<T[j+1] (sur l'exemple [2,3,1[où 2 est la tête de file).

```
fonction creuxDeListe(ref L : listeSC d'entiers) :file d'entier;
    var F :file d'entier ;
    début
        creerFile(F) ;
        si listeVide(L) alors
            retourner(F)
        sinon
            premier(L) ;
            v1=valeur(L) ;
            suivant(L) ;
            si estFinListe(L) alors
                retourner(F)
            sinon
                v2=valeur(L) ;
                suivant(L) ;
                tant que !estFinListe(L) faire
                    si v1>v2 et v2<valeur(L) alors
                        enfiler(F,v2)
                    finsi
                    v1=v2 ;
                    v2=valeur(L)
                    suivant(L)
                fintantque
                retourner(F)
            finsi
        finsi
    fin
```

5 – Ecrire cette même fonction *creuxDeListe* en utilisant la l'implémentation dynamique à la place des primitives. On décrira la structure interne choisit pour la file.

On choisit une file implémentée en allocation dynamique. On peut implémenter la file par une liste simplement chainée ayant deux curseurs pointant respectivement sur le premier et le dernier de la liste.

```
file d'entier=structure
                premier, dernier :^cellule d'entier;
            finstructure
fonction creuxDeListe(ref L : listeSC d'entiers) :file d'entier;
    var F :file d'entier ;
    var c :^cellule d'entier ;
    début
        F.premier=NIL ;
        F.dernier=NIL ;
        si listeVide(L) alors
            retourner(F)
        sinon
            L.clé=L.premier ;
            v1=L.clé^.valeurElement ;
            L.clé=L.clé^.pointeurSuivant ;
            si L.clé==NIL alors
                retourner(F)
            sinon
                v2= L.clé^. valeurElement;
                L.clé=L.clé^.pointeurSuivant ;
                tant que L.clé !=NIL faire
                    si v1>v2 et v2< L.clé^.valeurElement alors
                        new(c) ;
                        c^. valeurElement =v2 ;
                        c^. pointeurSuivant  =NIL ;
                        si F.dernier==NIL alors
                            F.dernier=c ;
                            F.premier=c ;
                        Sinon
                            F.dernier^.pointeurSuivant=c :
                            F.dernier=c ;
                    finsi
                    v1=v2 ;
                    v2= L.clé^. valeurElement ;
                    L.clé=L.clé^.pointeurSuivant ;
                fintantque
                retourner(F)
            finsi
        finsi
    fin
```

6 – Donnez les avantages et les inconvénients des deux implémentations pour les fonctions *tableauListe* et *creuxDeListe.*

Au niveau temps et mémoire, les algorithmes qui utilisent les pointeurs sont plus rapides et utilisent moins de mémoire puisque l'appel à toute fonction demande de générer le contexte donc du temps et de la mémoire ! Les algorithmes écrits avec les primitives sont plus lisibles et donc beaucoup plus « maintenables » et plus compréhensibles. ***Ceci est essentiel.*** *Ainsi, on peut noter qu'écrire la fonction creuxDeListe en implémentation dynamique est bien plus difficile que de l'écrire avec des primitives.*

Question 4. Les internautes utilisent un logiciel pour naviguer sur Internet. Parmi ses fonctions, il y a deux fonctions symbolisées « ⇐ » « ⇒ » qui permettent de naviguer au sein des pages (URL). Par exemple, si l'utilisateur consulte dans l'ordre les pages d'adresses U1,U2,U3,U4, lors de l'affichage de U4, « ⇐ » permet de revenir à U3 et on peut ensuite revenir à U4 par « ⇒ ». Si depuis U3 on visualise une page U5 alors l'accès à U4 est perdu. On rappelle que l'adresse d'une page est une chaine de caractère.

1 - Quelle structure de données peut-on utiliser dans le navigateur pour mémoriser les adresses des pages ? Pourquoi ?

Une pile d'adresse des pages car on souhaite un accès en priorité au plus récent. Les URL sont eds chaines de caractèresque l'on peut stocker dans un tableau.

2 - Quelle structure de données peut-on utiliser dans le navigateur pour réaliser les fonctions« ⇐ » « ⇒ » ? Pourquoi ?

Deux piles P1 et P2 : dans P1 on empile les adresses chaque fois qu'on accède à une nouvelle page.
Si on utilise la fonction « ⇐ » , on dépile la valeur de P1 pour la mettre dans P2. Si on utilise la fonction « ⇒ » , on dépile la valeur de P2 pour la mettre dans P1. Si depuis une page A, on ouvre une page B on vide la pile P2 et on empile B dans P1.

3 - Décrivez le(s) type(s) abstrait(s) correspondant à cette structure et notamment le champ valeur.

```
url=tableau[1..100]de caractères ; /* les adresses*/
contextePage=structure
                P1,P2 :pile de url ;
finstructure
```

4 - Ecrire les primitives d'accès et de modification de ce type.

Cette structure est un containeur, il y a donc 5 primitives à écrire.

```
fonction creerContextePage() :contextePage ;
    var c : contextePage ;
    debut
        creerPile (c.P1) ;
        creerPile(c.P2) ;
        retourner(c)
    fin
fonction detruireContextePage(ref c : contextePage) :vide ;
    debut
        detruirePile (c.P1) ;
        detruirePile (c.P2) ;
    fin
fonction ajouter(ref c : contextePage ;ref u :url) :vide ;
/* on ouvre une nouvelle url */
    var tmp : url ;
    debut
        detruirePile(c.P2) ;
        creerPile(c.P2) ;
        empiler(c.P1,u)
    fin
```

Les fonctions supprimer correspondent aux deux fonctionnalités « ⇐ » « ⇒ »

```
 fonction ⇐ (ref c : contextePage ) :booleen ;
    debut
        si pileVide(c.P1) alors
            retourner(faux)
        sinon
            empiler(c.P2,valeur(c.P1)) ;
            depiler(c.P1) ;
            retourner(vrai)
```

```
        finsi
    fin
fonction ⇒ (ref c : contextePage ) :booleen ;
    debut
        si pileVide(c.P2) alors
            retourner(faux)
        sinon
            empiler(c.P1,valeur(c.P2)) ;
            depiler(c.P2) ;
            retourner(vrai)
        finsi
    fin
```

5 – Une autre fonctionnalité du navigateur consiste à proposer dans la zone de saisie de l'URL, une liste d'URL au fur et à mesure que l'utilisateur écrit des caractères (complétion). Quelle structure de donnée peut permettre de gérer cette fonction ? Pourquoi ? Modifier le type abstrait de la question 3 en conséquence. Donnez les modifications des primitives.

La structure de donnée est un dictionnaire. Dans ce cas il ne faut pas stocker deux fois l'URL. Par exemple, on peut chosir pour valeur de la pile un pointeur vers un sommet du dictionnaire qui permet de retrouver l'URL. Le caractère « \0 » est celui qui doit être pointé puisque en remontant dans l'arbre on trouve l'URL en totalité. La structure de donnée devient :

```
contextePage=structure
                D : dico ;
                P1,P2 :pile de sommet ;
finstructure

fonction creerContextePage() :contextePage ;
    ajout de
        creerDictionnaire(C.D)
fonction detruireContextePage(ref c : contextePage) :vide ;
    ajout de
        detruireDictionnaire(C.D)
fonction ajouter(ref c : contextePage ;ref u :url) :vide ;
    ajout de
        var tmp :sommet ;
        tmp=ajouter(D,u);/* il faut que cette fonction renvoit l'adresse du sommet feulle*/
    modification
        empiler(c.P1,tmp)
    fin
```

Les fonctions fonctionnalités « ⇐ » « ⇒ » ne changent pas.

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;

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 fileVide(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;

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;

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;;

FIN