DEVUIP Service Scolarité	**ANNEE UNIVERSITAIRE 2011/2012** **SESSION D'AUTOMNE** PARCOURS / ETAPE : LIIN300 Code UE : J1IN3001 **Epreuve :** Algorithmique et structures de données 1 **Date :** 3 Janvier **Heure :** 11H00 **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 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 les lignes en pointillé.

☐ Dans une table de hashage chainée, l'exécution d'une recherche s'effectue en moyenne en O(1).

Il y une hypothèse, hachage uniforme, de plus c'est en moyenne O(1+n/m) quand n est le nombre de clé et m la taille de la table.

☒ Il existe 4 parcours dans les arbres binaires.

Les parcours sont hiérarchiques ou préfixe ou infixe ou suffixe.

……

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

Dans une pile, on a accès au sommet de la pile qui par définition est le dernier entré.

……

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

s n'est pas un pointeur c'est n qui l'est. Donc l'accès ce fait par s.n^.

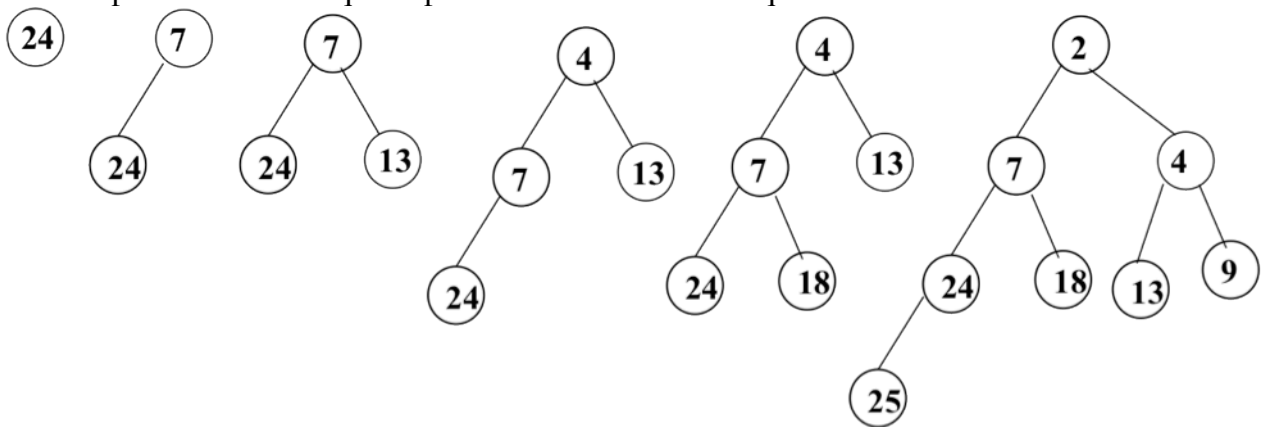
Question 2. Soit la liste de clé A=(24,7,13,4,18,9,2,25).

1 – Ecrire la fonction *supprime* qui supprime un élément dans un tas MIN.

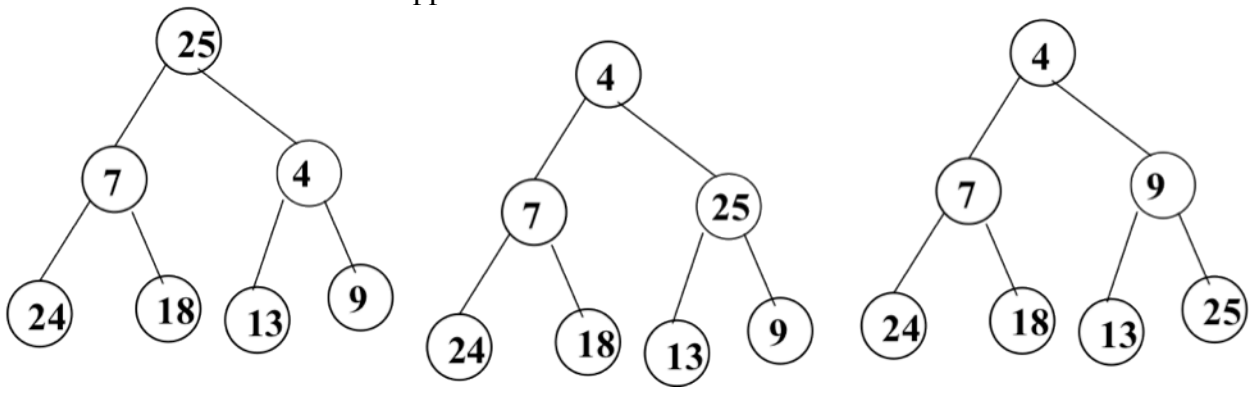
```
fonction supprime(ref T:tas d'entier):vide;
    var r:entier;
    début
        r=valeur(T);
        T.arbre[1]=T.arbre[T.tailleTas];
        T.tailleTas=T.tailleTas-1;
        reorganiseTasDesc(T,1);
    fin
fonction reorganiseTasDes(ref T:tas d'entier,x:sommet):vide;
    var g,d:sommet;
    début
        g= filsGauche(x);
        d= filsDroit(x);
        si g!=NIL alors
            si d!=NIL alors
                si getValeur(T,d)<getValeur(T,g) alors
                    g=d;
```

```
            finsi;
        finsi
        si getValeur(T,x)>getValeur(T,g) alors
            échanger(T.arbre[x],T.arbre[g]);
            reorganiseTasDesc(T,g)
        finsi
    finsi
fin
```

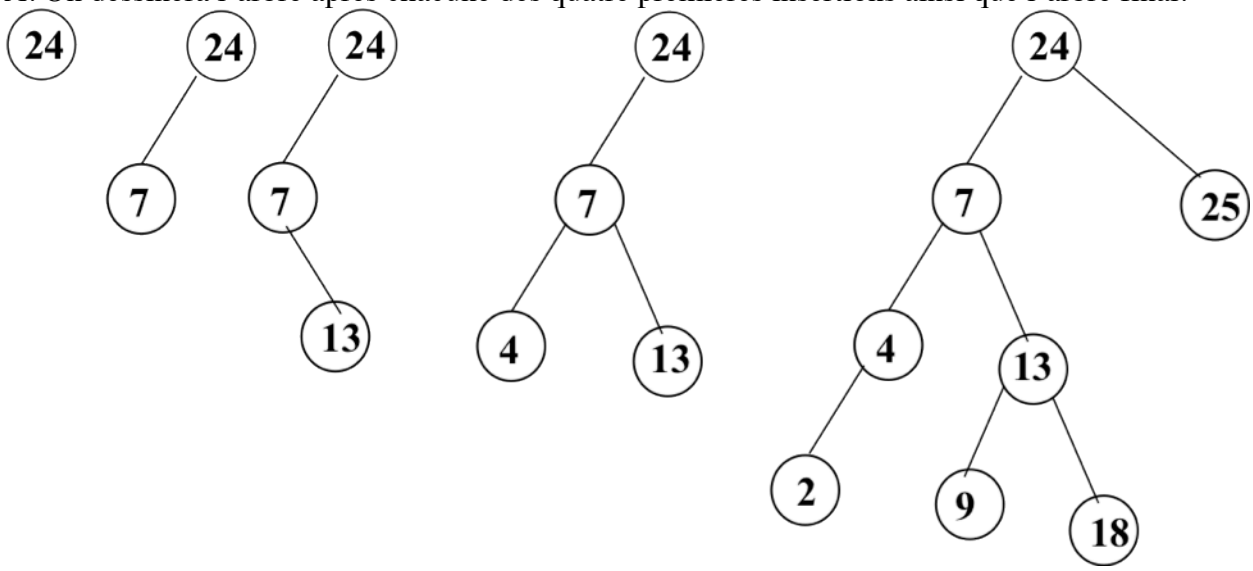
2 - Construire le tas MIN T correspondant à l'insertion consécutive des clés de la liste A. On dessinera l'arbre après chacune des quatre premières insertions ainsi que l'arbre final.



3 - Montrez l'exécution de la suppression sur l'arbre T.



4 – 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.



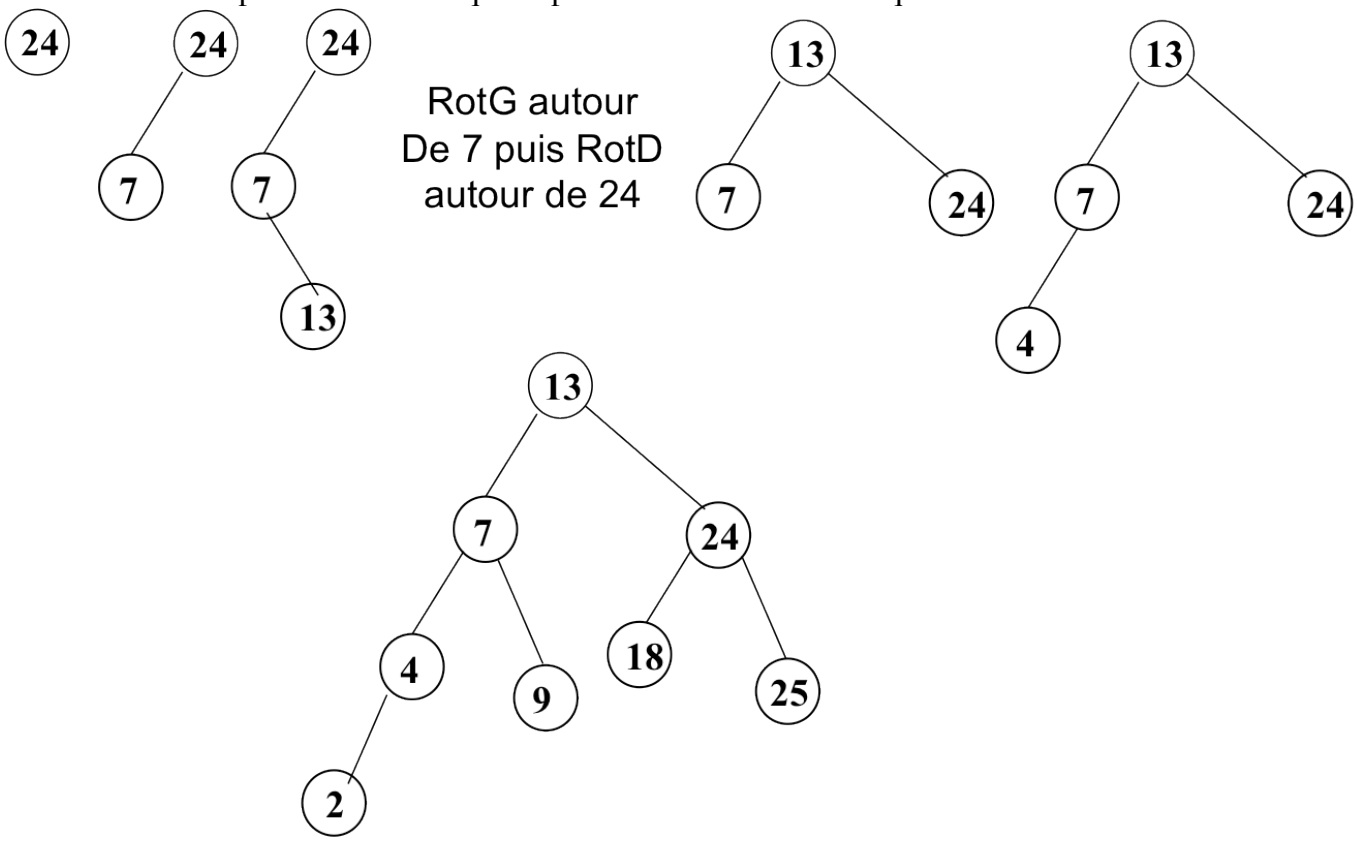
5 – Donner la liste des sommets en ordre préfixe, en ordre infixe et en ordre suffixe.

Préfixe : 24,7,4,2,13,9,18,25

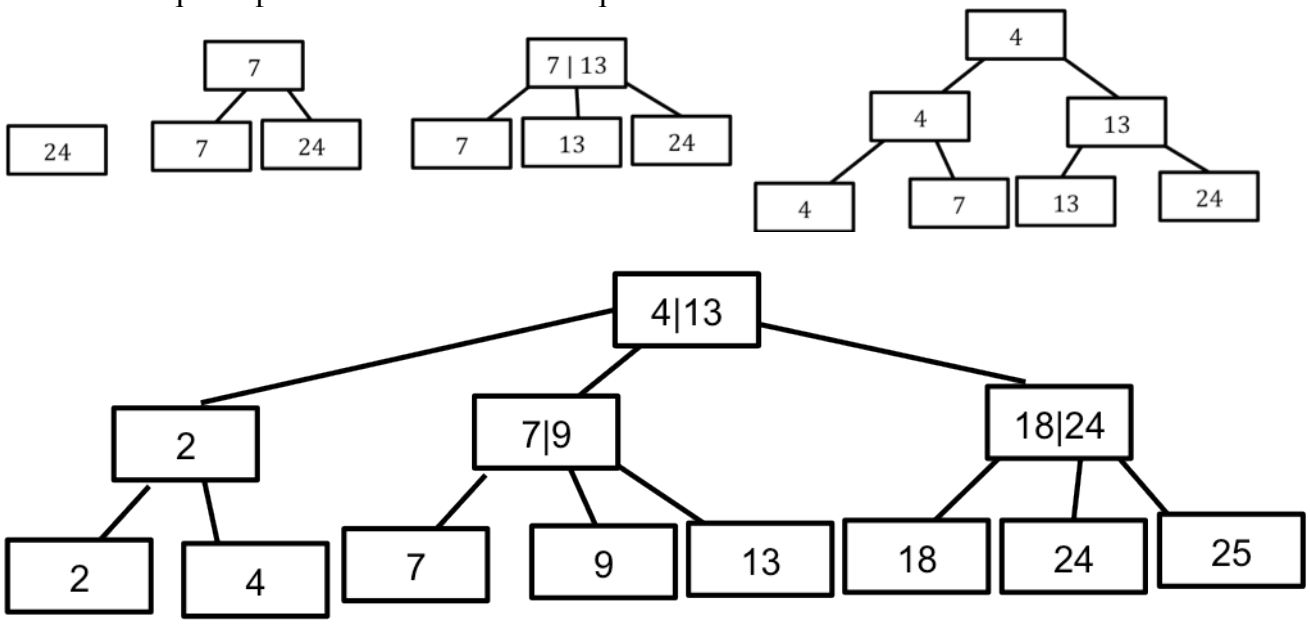
Infixe : 2,4,7,9,13,18,24,25

Suffixe : 2,4,9,18,13,7,25,24

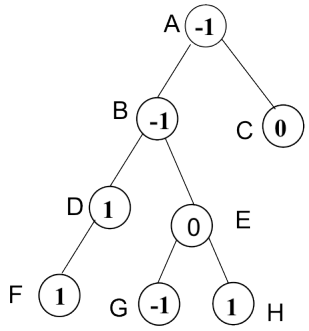
6 – Construire l'arbre binaire de recherche AVL correspondant à l'insertion consécutive de la liste A. On dessinera l'arbre après chacune des quatre premières insertions ainsi que l'arbre final.



7 - 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 un arbre binaire dont les sommets contiennent un élément de A={-1,0,1}. Dans la suite on appellera ce type d'arbre A-arbre. La figure ci-dessous donne un exemple de A-arbre, les lettres représentent les adresses des sommets.



1 – Ecrire la fonction *verifier* qui vérifie qu'un arbre binaire est un A-arbre c'est à dire que les sommets de l'arbre sont correctement étiquetés (i.e.éléments de A).

```
fonction verifier(ref A: sommet):booleen;
    début
        si A=NIL alors
            retourner(vrai)
        sinon
            si valeur(A) dans {-1,0,1} alors
                retourner verifier(filsGauche(A)) et verifier(filsDroit(A))
            sinon
                retourner(faux)
            finsi
        finsi
    fin
```

2 – En utilisant un parcours hiérarchique et les primitives du type abstrait listeSC, écrire la fonction *arbreListes* qui prend pour paramètre un A-arbre et fournit en sortie un tableau T tel que T[i] donne accès à la liste des adresses des sommets ayant pour étiquette i. sur l'exemple T[-1]=(A,B,G), T[0]=(E),T[1]=(D,F,H).

```
fonction arbreListes(ref A: sommet):tableau [-1..1] de listeSC de sommet;
    var T:tableau [-1..1] de listeSC de sommet;
    var P:tableau [-1..1] de pile de sommet;
    var F: file de sommet;
    car s:sommet;
    début
        creerPile(P[-1]); creerPile(P[0]); creerPile(P[1])
        créerFile(F);
        enfiler(F,A);
        tantque !videFile(F) faire
            s=valeur(F);
            empiler(P[valeur(s)],s);
            si filsGauche(s)!=NIL alors
                enfiler(F,filsGauche(s))
            finsi
            si filsDroit(A)!=NIL alors
                enfiler(F,filsDroit(s))
            finsi
            defiler(F)
        fintantque
        detruireFile(F);
        pour i=-1 à 1 faire
            creerListeSC(T[i]);
            tantque !pileVide(P[i]) faire
                insererEnTete(T[i],valeur(P[i]);
                dépiler(P[i])
            fintantque
            detruirePile(P[i]);
```

```
        finPour
        retourner(T);
    fin
```

3 –On dit qu'un sommet s est sympathique s'il a deux fils et si valeur(filsGauche(s))+ valeur(filsDroit(s))= valeur((s). Ecrite la fonction *nbSympa* qui prend pour argument un A-arbre et fournit comme résultat le nombre de nœud sympathique de l'arbre. Dans l'exemple, le résultat est 2.

```
fonction sympa(ref s:sommet):entier;
    si (filsGauche(s)!=NIL et filsDroit(s)!=NIL)
        et (valeur(s)==valeur(filsGauche(s))+valeur(filsDroit(s)))) alors
            retourner 1
sinon
        retourner 0
    finsi

fonction nbSympa(ref A: sommet):entier;
    var n:entier;
    début
        si estFeuille(A) alors
            retourner(0);
        sinon
            n=sympa(A);
            si filsGauche(A)!=NIL alors
                n=n+nbSympa((filsGauche(A));
            finsi
            si filsDroit(A)!=NIL alors
                n=n+nbSympa(filsDroit(A));
            finsi
            retourner(n)
        finsi
    fin
```

Question 4. On souhaite gérer un ensemble de polygones. Chaque polygone est défini par une suite de points. Chaque point est représenté par une paire (abscisse, ordonnée). Il est associé à une chaine de caractère qui est son nom ainsi que deux attributs : la couleur de remplissage du polygone et la couleur de son contour. La couleur est représentée en système RVB (Rouge,Vert,Bleu), soit un triplet d'entiers compris entre 0 et 255 ((0,0,0) reprèsente par exemple le noir). Chaque polygone est de plus muni d'un paramètre « profondeur » qui permet l'affichage. Le polygone de plus faible profondeur est affiché en dernier. 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 sa fonction.

1 – Décrivez la structure de données *couleur* permettant de stocker les couleurs. En donner des primitives.

```
couleur=structure
            R:entier;
            V:entier;
            B:entier;
        finstructure
```

Ce n'est pas un containeur. Les primitives sont get et set sur chaque champs de la structure.

2 – Décrivez la structure de données *polygone* permettant de décrire les informations liées à un polygone. En donner des primitives.

```
point=structure
        x:entier;
        y:entier;
    finstructure
polygone=structure
            nom=tableau[1..256] de caractère;
```

pointPolygone : liste de point;
couleurRemplissage:couleur;
couleurContour:couleur;
profondeur:entire;
finstructure

Un polygone peut être considéré comme un containeur car il est définit par une liste de point et on peut ajouter ou supprimer des points modifiant ainsi la figure. Il a donc 5 primitives: créerPolygone, detruirePolygone, ajouterPolygon (qui ajoute un point au polygone), supprimerPolygone (qui supprime un point du polygone), la cinquième primitive peut par exemple l'accès au nom. ON peut envisager également tous les set et get permettant d'accéder à la structure de données.

3 – Quelles sont les structures de données que l'on peut-on utiliser afin qu'un utilisateur puisse accéder aux paramètre d'un polygone par son nom ? Justifiez. Donnez les avantages et les inconvénients de chacune.

Première solution un hashage sur le nom avec une adresse permettant d'accéder aux caractéristiques du polygone.

Deuxième solution : un dictionnaire maintenu à jour à chaque création d'un polygone. Dans ce cas le pointeur fils gauche du dernier caractère de la chaine pointera vers les données satellites du polygone.

Le type de la donnée satellite sera ^polygone

Un dictionnaire est à la fois plus couteux en mémoire et en temps pour l'adjonction d'un nom que le hashage. Cependant si on veut par exemple envisager de la saisie prédictive seuls les dictionnaires permettront un travail efficace.

Dans les deux cas ce sont des containeurs.

4 – Un utilisateur peut accéder à un polygone pour le faire passer au premier plan et au dernier plan en agissant sur le paramètre profondeur. Proposer une structure de donnée permettant d'accéder facilement au polygone situé en premier plan. Justifiez.

La structure est un tas MIN. En effet, seul les tas MIN permettent d'accéder au plus petit d'un ensemble d'objet en O(1). De plus on veut pouvoir changer la profondeur qui classe les polygones entre eux. C'est le cas dans les tas MIN.

C'est un containeur.

5 – Ecrire la fonction *ajouterPolygone* qui prend comme paramètre un polygone et permet d'insérer le polygone dans toutes les structures de données que vous venez de décrire. On ne demande pas de reimplémenter les primitives vues en cours mais de dire comment on accède à la clé.

Comme vu précédemment il sera plus aidé de manipuler non pas des polygones mais l'adresse de ceux-ci.

On dispose donc de deux containeurs :

- *tas de ^Polygone, qui sera géré en utilisant comme clé pour un polygone P le champ P^.profondeur*
- *dico*

```
fonction ajouterPolygone(D:dico;T:tas de ^polygone; p: ^polygone):vide;
  début
    ajouter(T,p);
    /* dans cette fonction on prendra pour clé p^.profondeur */
    ajouter(D,p);
    /* dans cette fonction on prendra pour mot à ajouter p^.nom */
  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