TD No3

Exercice 1 arbres binaires

Question 1 : recherche

Ecrire un algorithme de **recherche** retournant l'adresse du noeud contenant une valeur e passée en argument et l'arbre vide dans le cas où l'élément n'est pas dans l'arbre.

Eléments de correction:

sans ordre sur les éléments de l'arbre, il n'est pas possible de prévoir à l'avance le sous-arbre où chercher l'élément. Ainsi, si l'on ne le trouve pas dans l'un des sous-arbres, il faut aller voir s'il n'est pas dans l'autre.

```
arbre recherche(int e, arbre a){
  if (estVide(a))
     return arbreVide();
  else if (a->valeur == e)
    return a
  else {
    arbre atmp=recherche(e, a->gauche);
    if (estVide(atmp))
      atmp=recherche(e, a->droit);
    return atmp;
  }
 }
```

Question 2 plus petit élément d'un arbre binaire

Ecrire un algorithme retournant l'adresse du noeud contenant la plus petites valeur d'un arbre binaire non vide.

Eléments de correction:

Même problème que pour la recherche: il faut regarder dans le sous-arbre gauche et dans le sous-abre droit. L'algorithme compare la valeur de la racine avec le min du sag,avec le min du sad et retourne la plus petite des trois valeurs.

D'un point de vue programmation, écrire une expression de la forme « a->qqc » suppose que a ne soit pas nul. A chaque fois que l'on écrit une telle expression, il faut se demander si a peut être nul.

```
arbre min(arbre a){
  int m,tmp;
  assert(! EstVide(a));
  m = a-> valeur;
  if (! estVide(a->gauche)){
    tmp=min(a->gauche);
    if (tmp < m)
      m=tmp;
  }
  if (! estVide(a->droit)){
    tmp=min(a->droit);
    if (tmp < m)
      m=tmp;
  }
  return m;
 }
```

IUP2-Miage-ASR Structures de données 2004-2005

Exercice 2 arbres binaires de recherche

Question 1 : exemples d'arbres binaires de recherche

Donnez tous les arbres binaires de recherche pouvant contenir les valeurs suivantes : 3; 6; 2; 7

Eléments de correction:

Racine 2: 5 arbres

Racine 3: 2 arbres

Racine 6: 2 arbres

Racine 7: 5 arbres

Question 2 : exemple d'arbre binaire de recherche

Dessinez l'arbre binaire de recherche obtenu par ajout successif aux feuilles des éléments suivants: 10;12;7;14;16;15;4;9;8;5;3;2

Supprimez les éléments 15; 9 et 7 en utilisant l'algorithme vu en cours.

Elément de correction:

on a à supprimer :

- *une feuille (15)*
- *un noeud interne au sens large ayant un seul fils*
- *un noeud interne ayant deux sous-arbres (dans ce cas, on échange la valeur avec celle du min de son sad ou celle du max de son sag)*

Ajoutez 9 à la racine de l'arbre obtenu.

Eléments de correction:

on applique l'algorithme de coupure sur cet exemple.

Question 3 : appartenance/ajout

Ecrivez une version itérative de l'algorithme indiquant si une valeur e appartient à un arbre binaire de recherche a. Faire de même pour l'algorithme ajout un élément e aux feuilles d'un arbre a.

Eléments de correction:

```
int appartientIter(int e, arbre a)
{
  while ((!estVide(a)) && (e != a->valeur)){
    if (e <a->valeur)
      a=a->gauche;
    else
      a=a->droit;
  }
```

```
  // 2 cas de sortie: soit a est vide, soit on a trouvé e
  if (estVide(a))
    return 0;
  else return 1;
}
```

Question 4 : plus petit élément d'un arbre binaire de recherche non vide

Ecrire une version récursive et une version itérative d'un algorithme retournant la plus petite valeur d'un arbre binaire de recherche non vide.

Eléments de correction:

l'ordre sur les éléments nous dispense de chercher dans les deux sous-arbres. En fait, on a même mieux, l'ordre sur les éléments nous permet de trouver le plus petit élément sans aucune comparaison d'éléments: le plus petit élément, c'est celui qui est le plus à gauche.

```
int min(arbre a)
{
  assert( ! estVide(a));
  if (estVide(a->gauche))
    return a->valeur;
  else return min(a->gauche);
}
// version itérative
int minIter(arbre a)
{
  assert( ! estVide(a));
  while ( ! estVide(a->gauche))
    a=a->gauche;
  return a->valeur;
}
```

Question 5 : successeur d'un élément donné

Le successeur d'un élément est la plus petite valeur qui lui est immédiatement supérieure. Ainsi, le successeur de 10 dans l'arbre de la question 1 est 12. Un parcours infixe d'un arbre binaire de recherche permet d'obtenir les éléments dans l'ordre croissant. En utilisant cette propriété, écrire un algorithme permettant d'obtenir l'adresse du noeud contenant le successeur d'un élément donné de l'arbre et arbreVide si l'élément n'a pas de successeur. Quelle est la complexité au pire de votre algorithme en nombre de comparaisons d'éléments ?

Eléments de correction:

*Il faut parcourir l'arbre et s'arrêter juste après avoir rencontré l'élément **e**. Pour cela, on va déterminer une valeur « trouve » qui vaudra: 0 quand on n'a pas encore rencontré **e**, 1 quand on vient de recontrer **e** et 2 quand on a trouvé le successeur de e.*

D'un point de vue complexité, c'est celle d'un parcours infixe qui s'arrête dès que l'élément cherché est trouvé. La complexité au pire est donc celle du parcours infixe : complexité linéaire par rapport au nombre d'éléments de l'arbre.

```
////////////////////////////////////////////////////////////////////////////
```



```
//  Nom :       successeurParcoursInfixeAux
//
//  Arguments : un element e, un arbre a
//
//
//  retourne :  res le successeur de e dans a (passé par valeur)
//              trouve qui vaut :
//                    + 2 si on a déjà trouvé le successeur
//                    + 1: si l'élément précédemment vu est e
//                    + 0: dans les autres cas
//
void successeurParcoursInfixeAux(int e, arbre a, int * res, int * trouve){
  if (! estVide(a)){
    if (! estVide(a->gauche))
      successeurParcoursInfixeAux(e, a->gauche, res, trouve);
    if (*trouve == 1){
      *res = a-> valeur;
      *trouve=2;
    }else if (a->valeur == e)
      *trouve = 1;
    if ((*trouve != 2) && (! estVide(a->droit)))
        successeurParcoursInfixeAux(e, a->droit, res, trouve);
  }
}

////////////////////////////////////////////////////////////////////////////
//  Nom :       successeurParcoursInfixe
//
//  Arguments : un element e et un arbre a
//
//  retourne :  le successeur de e dans a
//
//  l'élément e doit appartenir à a et y avoir un successeur
int successeurParcoursInfixe(int e, arbre a){
  int res=-1, t=0;
  successeurParcoursInfixeAux(e,a,&res, &t);
  assert(t == 2);
  return res;
}
```

Question 6 : successeur d'un élément donné

Dans cette question (et seulement dans cette question), on suppose que le type noeud contient aussi un lien vers le père d'un noeud qui vaudra arbreVide() pour la racine de l'arbre.

- Ajoutez ces liens sur le dessin d'arbre de la question 1

- Définisssez les types noeuds et arbre correspondants
- adaptez l'algorithme d'ajout aux feuilles à cette nouvelle définition des types.
- Proposez un algorithme permettant de déterminer l'adresse du noeud contenant le successeur d'un élément de l'arbre plus efficace que celui proposé en question 2. On pourra envisager deux cas de figures illustrés par le successeur de 7 et celui de 9 dans l'arbre donné en question 1. Quelle est la complexité au pire (en nombre de comparaisons d'éléments) de votre algorithme.

Eléments de correction:

Pour permettre les parcours d'un noeud jusqu'à la racine, on prendra comme convention que la racine de l'arbre a l'arbre vide comme père. On pourra écrire des choses de la forme:

```
while (! estVide(a->pere)){
  traitement sur le noeud;
  a=a->pere;
}
```

La contrepartie est que cette propriété ne sera pas valable pour un sous-arbre car son champ père n'est pas nul. On pourra remplacer ce critère d'arrêt par une comparaison avec la racine de l'arbre (qu'il aura alors fallu préserver):

```
// a désigne l'adresse du noeud courant
// racine désigne l'adresse de la racine de l'arbre
while (a != racine){
  traitement sur le noeud;
  a=a->pere;
}
```

Pour tenir compte de la modification de structure, on va

- *modifier la fonction cree_noeud de façon à ce qu'elle aie un argument supplémentaire contenant le père du noeud à créer*
- *faire en sorte que la fonction d'ajout aux feuille aie le père du noeud courant comme argument : cette fonction sera une fonction auxiliaire qui fera le travail*
- *la fonction ajoute_feuille appellera la fonction ajoute_feuilleAux avec les paramètres ad hoc (notamment le père de la racine de l'arbre).*

Algo d'ajout aux feuilles:

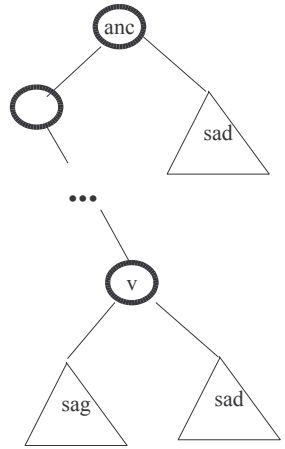
```
arbre cree_noeud(int e, arbre fg, arbre, arbre p)
{
  arbre a=malloc(sizeof(noeud));
  assert(a != NULL);
  a->valeur=e;
  a->gauche=fg;
  a->droit=fd;
  a->pere=p;
  return a;
}
arbre ajoute_feuilleAux(int e, arbre a, arbre p)
{
```



```
  if (estVide(a))
    a=cree_noeud(e,NULL, NULL,p);
  else
    if (e<a->valeur)
      a->gauche=ajoute_feuilleAux(e,a->gauche,p);
    else if (e> a->valeur)
      a->droit=ajoute_feuilleAux(e,a->droit,p);
  return a;
}
arbre ajoute_feuille(int e, arbre a){
  if (estVide(a))
    return ajoute_feuilleAux(e, a, arbreVide());
  else return ajoute_feuilleAux(e, a, a->pere);
}
```

Successeur: nous allons tirer partie du fait que l'on a affaire à des arbres binaires de recherche. On peut montrer que les éléments plus grands que la valeur v d'un noeud se définissent récursivement de la façon suivante :

- *les éléments de son sous-arbre droit*
- *en remontant vers la racine: la valeur anc du premier noeud où l'on remonte par la gauche (i.e.: v appartient au sous-arbre gauche de anc) et toutes les valeurs supérieurs à anc.*



Les valeurs du sous arbre droit de v sont inférieures à anc car elles sont dans le sous-arbre gauche de anc. Le successeur de v est donc :

- *si le noeud a un sous-arbre droit, le successeur est le min de ce sous-arbre.*
- *Si le noeud n'a pas de sous-arbre droit, on remonte jusqu'à trouver un noeud où on arrive par la gauche. Le successeur est la valeur de ce noeud.*

L'algorithme suivant utilise cette propriété:

ràf

Question 7 : code positionnel d'un élément dans l'arbre

A tout élément de l'arbre, on peut associer une chaîne de 0 et de 1 indiquant le chemin de la racine à l'élément dans l'arbre de la façon suivante :

- la chaîne se lit de gauche à droite
- la racine a 1 comme code
- si on rencontre un 0, on va sur le sous-arbre droit
- si on rencontre un 1, on va sur le sous-arbre gauche
- quand la chaîne est vide, on doit être arrivé à l'élément

Ainsi, dans l'arbre de la question 1, le code positionnel de 8 est 1010.

Donnez le code positionnel de 5 dans l'arbre de la question 1.

Ecrire un algorithme permettant de trouver le code positionnel d'un élément donné. Inversement, écrire un algorithme retournant l'adresse du noeud désigné par un code positionnel donné.

Elément de correction:

Pour stocker le code d'un élément, nous allons supposer que nous disposons d'une structure de données linéaire permettant un ajout et une suppression à la fin, un affichage des éléments du début à la fin.

Ràf: donner le TAD correspondant

le code C correspondant :

```
////////////////////////////////////////////////////////////////////////////
//  Nom :       construitCode
//
//  Arguments : une valeur e, un arbre a
//
//  retourne :  un tableau t (passé par adresse) qui contiendra le code de
//              l'élément dans a
//              si l'élément n'est pas dans a, le code est celui qu'il
//              aurait après insertion aux feuilles
void construitCode(int e, arbre a, tableau * t){
  if ( ! estVide(a)){
    if (e < a-> valeur){
      tableau_ajouterFin(0,t);
      construitCode(e, a->gauche, t);
    }
    else if (a->valeur < e){
      tableau_ajouterFin(1,t);
      construitCode(e, a->droit, t);
    }
  }
}
```



```
////////////////////////////////////////////////////////////////////////////
//  Nom :       afficheCode
//
//  Arguments : un tableau t contenant le code d'un élément
//
//  Fait:        affiche le contenu de t du début à la fin
//               rien si t est vide
//               l'affichage est suivi d'un saut de ligne
void afficheCode(tableau t){
  int cb;
  printf("1"); // la racine
  for (cb=0; cb < tableau_taille(t); cb++)
    printf("%i",tableau_ieme(cb,t));
  printf("\n");
}

////////////////////////////////////////////////////////////////////////////
//  Nom :       NoeudCode
//
//  Arguments : un arbre a, un tableau t contenant le code d'un élément de
l'arbre
//
//  retourne :  l'adresse du noeud de l'arbre ayant t comme code
//              si aucun élément n'a t comme code, retourne l'arbre vide.
arbre noeudCode(arbre a, tableau t){
  int cb=0;
  while ((! estVide(a)) && (cb <tableau_taille(t))){
    if (tableau_ieme(cb,t)==0)
      a=a->gauche;
    else a=a->droit;
    cb ++;
  }
  return a;
}

////////////////////////////////////////////////////////////////////////////
//  Nom :       valeurCode
//
//  Arguments : un arbre a, un tableau t contenant le code d'un élément de
//              l'arbre
//
//  retourne :  la valeur de l'élément de a ayant t comme code
//
//  pre-condition: il doit y avoir un élément ayant le code t
```

```
int valeurCode(arbre a, tableau t){
  arbre n=noeudCode(a,t);
  assert( ! estVide(n));
  return n->valeur;
}
```

Question 8 : successeur d'un élément donné

Proposez un algorithme tirant partie de la question 7 et permettant de déterminer l'adresse du noeud contenant le successeur d'un élément donné dans le cas où les noeuds ne contiennent pas de lien vers leur père. Quelle est sa complexité au pire en nombre de comparaisons d'éléments ?

Eléments de correction:

La complexité est en O(h) car on fait un parcours pour trouver le code de l'élément et, suivant le cas:

- *soit on trouve le min du sag (ce qui revient à finir la hauteur de l'arbre commencée en cherchant le code de l'élément),*
- *soit on parcourt le code jusqu'à trouver un 1 (remontée par la gauche, cf question 6) puis le code résultant par la droite pour trouver l'élément correspondant de l'arbre. Au pire, on fait 3 * hauteur(a) parcours*

Dans tous les cas, on est en O(h).

Dans le second cas, il est possible de mémoriser le dernier élément situé en haut d'un sous arbre gauche ce qui éviterait le parcours du code et la recherche de l'élément correspondant. Cela suppose de ne pas utiliser construitCode mais de faire un travail adapté dans successeur. Cette amélioration qui ne change pas l'ordre de grandeur de la complexité est laissé en exercice.

```
int successeur(int e, arbre a){
  tableau t=tableau_tableauVide();
  arbre ae;
  construitCode(e, a, &t);
  ae=noeudCode(a, t);
  assert(! estVide(ae));
  if (! estVide(ae->droit))
    return min(ae->droit);
  else {
    while((! tableau_estVide(t)) && (tableau_ieme(tableau_taille(t)-1, t)==1))
      tableau_supprimerFin(&t);;
    if (tableau_taille(t) >0)
      tableau_supprimerFin(&t);
    return valeurCode(a,t);
  }
}
```