

TD12 : Arbre binaire

Exercice 1 : Parcours préfixé itératif d'un arbre binaire

Ecrire en langage C++ la procédure membre **itérative** du parcours **préfixé** dans un arbre binaire qui affiche les éléments de l'arbre.

L'astuce consiste à utiliser une pile pour garder en mémoire les nœuds en attente de traitement.

```
void Arbre::parcoursPrefixeIteratif () const {
    if (adRacine != NULL) {
        Pile p;
        Noeud * n;
        p.empiler(adRacine);
        while (!p.estVide()) {
            n = p.consulterSommet();
            p.depiler();
            afficherElementA(n->info);
            if (n->fd != NULL) p.empiler(n->fd);
            if (n->fg != NULL) p.empiler(n->fg);
            /* On met le fils gauche en dernier sur la pile,
               pour qu'il soit le premier traité au passage suivant */
        }
    }
}
```

Exercice 2 : Suppression dans un ABR

Ecrire en langage algorithmique la procédure de suppression d'un élément dans un arbre binaire de recherche.

Dans cette solution, on supprime la première occurrence rencontrée de l'élément.

La marche à suivre dépend du nombre de fils du nœud concerné (cf. CM).

(0) Si l'élément n'est pas dans l'arbre, la procédure n'a aucun effet.

(a) Si n n'a pas de fils (c'est une feuille), il suffit de le supprimer.

(b) Si n n'a qu'un seul fils, on court-circuite n (on attache le fils de n au père de n), puis on supprime n.

(c) Si n a deux fils, on le remplace par son plus proche successeur (le nœud le plus à gauche du sous-arbre droit).

Plus précisément, on copie l'élément du plus proche successeur de n dans n, puis on supprime le nœud du plus proche successeur. Remarque : on pourrait aussi remplacer n par son prédécesseur le plus proche (le nœud le plus à droite du sous-arbre gauche).

SOLUTION 1 (la plus compacte) : On utilise un « lien sur lien sur Noeud » (adresse d'une adresse de Noeud) pour mémoriser l'adresse de la case mémoire qui contient l'adresse du nœud contenant e (ie. l'adresse de fd ou fg du nœud parent).

Procédure supprimerElementABR (e : ElementA, a : Arbre)

Précondition : a est un ABR non vide,

Postcondition : la première occurrence de e dans a est supprimée de l'arbre, a est toujours un ABR

Paramètres en mode donnée : e

Paramètres en mode donnée-résultat : a

Variables locales :

n, successeur : lien sur Noeud

pere, pere_successeur : lien sur lien sur Noeud

trouve : booléen

Début

{On commence par rechercher le Noeud qui contient l'élément, et on mémorise au passage l'adresse de la case mémoire qui contient l'adresse de ce Noeud (ie. l'adresse de fd ou fg du noeud parent)}

$n \leftarrow a.adRacine$

$pere \leftarrow \&a.adRacine$

$trouve \leftarrow faux$

Tant que ($n \neq NIL$) et non(trouve) Faire

Si estEgal($n \uparrow.info$, e) Alors

$trouve \leftarrow vrai$

Sinon

Si estInferieur(e, $n \uparrow.info$) Alors

$pere \leftarrow \&(n \uparrow.fg)$ *{adresse du champ fg du Noeud n}*

$n \leftarrow n \uparrow.fg$

Sinon

$pere \leftarrow \&(n \uparrow.fd)$

$n \leftarrow n \uparrow.fd$

FinSi

FinSi

FinTantQue

Si (trouve) Alors *{si on a trouvé l'élément, pas de sinon car cas(0) pas d'effet}*

Si ($n \uparrow.fg = NIL$) et ($n \uparrow.fd = NIL$) Alors *{cas (a) : pas de fils, on supprime}*

$pere \uparrow \leftarrow NIL$

libérer n

Sinon

Si ($n \uparrow.fg \neq NIL$) et ($n \uparrow.fd = NIL$) Alors *{cas (b1) : 1 fg, on court-circuite et supprime}*

$pere \uparrow \leftarrow n \uparrow.fg$

libérer n

Sinon

Si ($n \uparrow.fg = NIL$) et ($n \uparrow.fd \neq NIL$) Alors *{cas (b2) : 1 fd, on court-circuite et supprime}*

$pere \uparrow \leftarrow n \uparrow.fd$

libérer n

Sinon

{cas (c) : deux fils. On recherche le plus proche successeur de n = noeud, le plus à gauche du sous-arbre droit}

$successeur \leftarrow n \uparrow.fd$

$pere_successeur \leftarrow \&(n \uparrow.fd)$

Tant que ($successeur \uparrow.fg \neq NIL$) Faire

$pere_successeur \leftarrow \&(successeur \uparrow.fg)$

$successeur \leftarrow successeur \uparrow.fg$

FinTantQue

$n \uparrow.info \leftarrow successeur \uparrow.info$ *{pas besoin de changer fg et fd, juste l'info}*

$pere_successeur \uparrow \leftarrow successeur \uparrow.fd$ *{on court-circuite le successeur}*

libérer successeur

FinSi

FinSi

FinSi

FinSi

Fin supprimerElementABR

SOLUTION 2 : Plus longue, mais sans lien sur lien. A la place, on mémorise l'adresse du Noeud parent (lien « simple ») et un booléen pour savoir si l'élément à supprimer se trouve dans le fils gauche ou dans le fils droit de ce noeud. On doit aussi traiter différemment la racine, c'est-à-dire les cas où l'on doit modifier a.adRacine et non fd ou fg.

Procédure supprimerElementABR(e : ElementA, a : Arbre)

Précondition : a est un ABR non vide,

Postcondition : la première occurrence de e dans a est supprimée de l'arbre, a est toujours un ABR

Paramètres en mode donnée : e

Paramètres en mode donnée-résultat : a

Variables locales :

n, nouveau_n : lien sur Noeud

pere, nouveau_pere : lien sur Noeud

trouve, explorer_gauche, e_dans_racine : booléen

Début

{On commence par rechercher le Noeud qui contient l'élément et on mémorise l'adresse du noeud parent}

n ← a.adRacine

pere ← NIL

trouve ← faux

Tant que (n ≠ NIL) et non(trouve) Faire

Si estEgal(n↑.info, e) Alors

trouve ← vrai

Sinon

pere ← n

Si estInferieur(e, n↑.info) Alors

explorer_gauche ← vrai

n ← n↑.fg

Sinon

explorer_gauche ← faux

n ← n↑.fd

FinSi

FinSi

FinTantQue

Si (trouve) Alors *{si on a trouvé l'élément}*

Si (n↑.fg = NIL) et (n↑.fd = NIL) Alors *{cas (a) : n est une feuille}*

Si (n = a.adRacine) Alors a.adRacine ← NIL

Sinon *{on vient de la gauche ou de la droite}*

Si (explorer_gauche) Alors pere↑.fg ← NIL Sinon pere↑.fd ← NIL FinSi

FinSi

libérer n

Sinon

Si (n↑.fg ≠ NIL) et (n↑.fd = NIL) Alors *{cas (b1) : n a seulement un fils gauche}*

Si (n = a.adRacine) Alors a.adRacine ← n↑.fg

```

Sinon
  Si (explorer_gauche) Alors pere↑.fg ← n↑.fg Sinon pere↑.fd ← n↑.fg FinSi
FinSi
libérer n
Sinon
  Si (n↑.fg = NIL) et (n↑.fd ≠ NIL) Alors {cas (b2) : n a seulement un fils droit}
    Si (n = a.adRacine) Alors a.adRacine ← n↑.fd
    Sinon
      Si (explorer_gauche) Alors pere↑.fg ← n↑.fd Sinon pere↑.fd ← n↑.fd FinSi
    FinSi
  libérer n
Sinon
  {cas (c) : n a deux fils. On recherche le plus proche successeur de n = noeud,
  le plus à gauche du sous-arbre droit}
  successeur ← n↑.fd
  pere_successeur ← n
  Tant que (successeur↑.fg ≠ NIL) Faire
    pere_successeur ← successeur
    successeur ← successeur↑.fg
  FinTantQue
  n↑.info ← successeur↑.info {pas besoin de changer fg et fd}
  Si (pere_successeur = n) Alors
    {le sous-arbre droit était réduit à une seule feuille, on n'est pas rentré dans le
    TantQue, il faut changer le fd de n}
    pere_successeur↑.fd ← successeur↑.fd
  Sinon
    {on est passé au moins une fois dans le TantQue, c'est le fg d'un des noeuds
    du sous-arbre droit qu'il faut changer}
    pere_successeur↑.fg ← successeur↑.fd {on court-circuite le successeur}
  FinSi
  libérer successeur
FinSi
FinSi
FinSi
FinSi

```

Fin supprimerElementABR

SOLUTION 3 : Version récursive, avec lien en mode donnée-résultat. Dans cette version, une fonction récursive est utilisée pour se déplacer sur l'élément à supprimer ainsi que pour se déplacer (et supprimer) le plus proche successeur. Pour simplifier l'écriture nous utiliserons une fonction supplémentaire qui retourne le noeud contenant le minimum d'un (sous-)arbre.

Fonction minApartirDeNoeud(n : lien sur Noeud) : lien sur Noeud

Précondition : le noeud pointé par n existe

Postcondition : l'arbre commençant à n est inchangé

Résultat : le lien sur le noeud contenant la valeur minimum de l'arbre commençant à n

Paramètres en mode donnée : n

Variables locales :

minNoeud : lien sur Noeud

Début

minNoeud \leftarrow n

Tant que (minNoeud \uparrow .fg \neq NIL) faire {recherche du min dans l'arbre = parcourir les fg}

minNoeud \leftarrow minNoeud \uparrow .fg

Fin TantQue

retourner minNoeud

Fin minApartirDeNoeud

Procédure supprimerElementABR(e : ElementA, a : Arbre)

Postcondition : la première occurrence de e dans a est supprimée de l'arbre, a est toujours un ABR

Paramètres en mode donnée : e

Paramètres en mode donnée-résultat : a

Variables locales :

Début

supprimerElementABRApartirDeNoeud(e,a.adRacine) {appel à la fonction récursive de suppression}

Fin supprimerElementABR

Procédure supprimerElementABRApartirDeNoeud (e : ElementA, n : lien sur Noeud)

Postcondition : la première occurrence de e à partir de n est supprimée de l'arbre, n contient le nouveau sous-arbre qui est toujours un sous-arbre d'un ABR

Paramètres en mode donnée : e

Paramètres en mode donnée-résultat : n

Variables locales :

aDetruire, successeur : lien sur Noeud

valeurEchangee : ElementA

Début

Si (n \neq NIL) Alors {rien à faire si le noeud n'existe pas}

Si (n \uparrow .info > e) Alors supprimerElementABRApartirDeNoeud(e, n \uparrow .fg) {on cherche à gauche}

Sinon Si (n \uparrow .info < e) Alors supprimerElementABRApartirDeNoeud(e, n \uparrow .fd) {on cherche à droite}

Sinon

{on a trouvé le noeud à supprimer}

{on sauvegarde le noeud à détruire}

aDetruire \leftarrow n

{on regarde les différents cas, en fonction des fils existants}

Si (n \uparrow .fg = NIL) Alors {pas de fg, on court-circuite le fd}

n \leftarrow n \uparrow .fd

libérer aDetruire

Sinon Si (n \uparrow .fd = NIL) Alors {pas de fd, on court-circuite le fg}

n \leftarrow n \uparrow .fg

libérer aDetruire

Sinon {le noeud a deux fils}

{on cherche son plus proche successeur, c'est-à-dire le minimum du sous-arbre droit}

successeur \leftarrow minApartirDeNoeud(n \uparrow .fd)

{on sauvegarde la valeur à échanger}

valeurEchangee \leftarrow successeur \uparrow .info

{on supprime le successeur grâce à la procédure de suppression => appel récursif}

supprimerElementABRApartirDeNoeud(valeurEchangee,n)
{on échange les valeurs}
n↑.info ← valeurEchangee
Fin supprimerElementABR