Algorithmes et structures de données : TD 1 Corrigé

Arbres binaires - Arbres binaires de recherche - Fonctions définies par récurrence - Complexité asymptotique

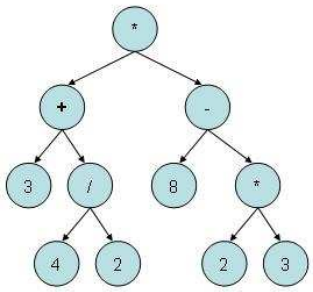
Rappel :
`SetLength(tableau, n)` est de complexité O(n)
`SetLength(tableau, 1)` est de complexité O(1)
`New(element)` est de complexité O(1) quand `element` est d'un type de taille fixe

Exercice 1.1 *Arbres binaires*

Considérer l'arbre suivant :

No	contenu	gauche	droite
1	*	2	3
2	+	4	5
3	-	6	7
4	3	0	0
5	/	8	9
6	8	0	0
7	*	10	11
8	4	0	0
9	2	0	0
10	2	0	0
11	3	0	0

1. Déssiner cet arbre.



2. Quelle est la hauteur de l'arbre ?
   La hauteur de l'arbre est 3.
3. Est-ce que cet arbre est un arbre entier, un arbre parfait (=complet), et/ou un arbre dégénéré ?
   Cet arbre est entier car chaque noued a zero ou deux fils. Par contre, cet arbre est ni parfait ni dégénéré.
4. Afficher cet arbre binaire de la manière préfix, puis infix, et ensuite postfix.
   Premièrement, on va afficher cet arbre sans utiliser des parentheses.

Préfix * + 3 / 4 2 - 8 * 2 3

Infix 3 + 4 / 2 * 8 - 2 * 3

Postfix 3 4 2 / + 8 2 3 * - *

Deuxièmement, on va afficher cet arbre en utilisant des parentheses. Notez que les parenthèses sont important uniquement pour la notation **infix**.

Préfix *(+(3, /(4, 2)) , -(8, *(2, 3)))

Infix ((3+ (4/2)) * (8- (2*3)))

Postfix ((3, (4, 2)/)+, (8, (2, 3)*)-)*

5. Considérer la fonction suivante :

```
procedure afficher( noeud : p_t_noeud )
début
        si NOT(noeud^.gauche = NIL) alors
                afficher(noeud^.gauche)
        si NOT(noeud^.droite = NIL)alors
                afficher(noeud^.droite)
        Write(noeud^.contenu);
fin
```

6. Faites tourner l'appel `afficher(racine);` avec la `racine` de votre arbre dans un tableau. Utiliser une nouvelle colonne `noeud^ .contenu` pour chaque nouvelle variable locale.

noeud^ .contenu 1er appel	noeud^ .contenu 2ème appel	noeud^ .contenu 3ème appel	noeud^ .contenu 4ème appel
*	+	3	/
noeud^ .contenu 5ème appel	**noeud^ .contenu 6ème appel**	**noeud^ .contenu 7ème appel**	**noeud^ .contenu 8ème appel**
4	2	-	8
noeud^ .contenu 9ème appel	**noeud^ .contenu 10ème appel**	**noeud^ .contenu 11ème appel**	
*	2	3	

7. Que fait cette procédure ?
   Cette procédure affiche l'arbre de la manière postfix.
   3 4 2 / + , 8, 2, 3 * -
8. Ecrire une fonction définie par récurrence qui calcule le résultat du terme décrit par cet arbre binaire. (Démarche : Quelle est la condition d'arrêt ? Comment faut-il placer les appels récurrents ?)

```
function calculer ( noeud : p_t_noeud ) : integer;
début
        si (noeud^.contenu = '+') alors
           result := calculer(noeud^.gauche) + calculer(noeud^.droite)
        sinon (noeud^.contenu = '-') alors
           result := calculer(noeud^.gauche) - calculer(noeud^.droite)
        sinon (noeud^.contenu = '*') alors
```

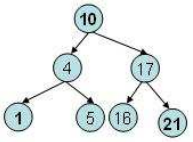
```
            result := calculer(noeud^.gauche) * calculer(noeud^.droite)
         sinon (noeud^.contenu = '/') alors
            result := calculer(noeud^.gauche) / calculer(noeud^.droite)
         sinon
            result := StrToInt(noeud^.contenu);
         fin si
fin
```

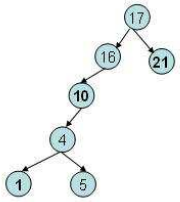
L'évaluation du terme donne le résultat 10, bien evidemmment.

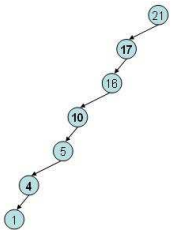
Exercice 1.2 *Arbres binaire de recherche*

Considérer l'ensemble des clés $1, 4, 5, 10, 16, 17, 21$.

1. Dessiner des arbres binaires de recherche de cet ensemble de clés avec une hauteur de 3, puis 5, et ensuite 7.







Exercice 1.3 *Arbres binaire de recherche*

1. Etablir la structure de données `p_t_noeud` pour cet arbre binaire de recherche contenant une clé (`integer`) et deux pointeurs, un pour le sous-arbre gauche, et un pour le sous-arbre droite.

```
type
    p_t_noeud = ^t_noeud;
    t_noeud = RECORD
cle    : integer;
gauche : p_t_noeud;
droite : p_t_noeud;
    END;
```

2. Ecrire une fonction `function max(noeud : p_t_noeud ):integer` qui calcule le maximum de l'abre de recherche.

```
function max(noeud : p_t_noeud) : integer;
var temp : p_t_noeud;
var max : integer;
begin
  max := Low(Integer); { Le plus faible possible }
  temp := noeud;
  while NOT(noeud = NIL) do
  begin
      if noeud^.cle > max then
        max := noeud^.cle ;
      noeud := noeud^.droite;
  end;
  result := max;
end;
```

3. Ecrire une fonction `function min(noeud : p_t_noeud ):integer` qui calcule le minimum de l'abre de recherche.

```
function min(noeud : p_t_noeud) : integer;
var temp : p_t_noeud;
var min : integer;
begin
  min := High(Integer); { Le plus grand possible }
  temp := noeud;
  while NOT(noeud = NIL) do
  begin
      if noeud^.cle < min then
        min := noeud^.cle ;
      noeud := noeud^.droite;
  end;
  result := min;
end;
```