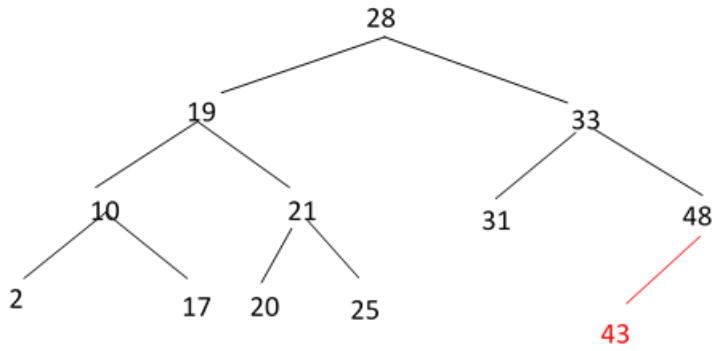
Corrigé TD3

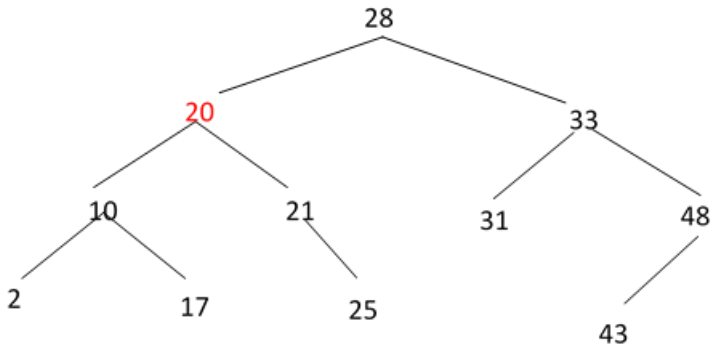
Habiba Drias

Exercice 3.2

1) insérer le nombre 43 dans l'arbre.



2) Ensuite supprimer le nombre 19 de l'arbre.



3) Ecrire un algorithme de suppression d'un élément d'un arbre binaire de recherche.

L'algorithme passe par 4 étapes qui sont :

1. Recherche de la valeur *x* dans l'arbre.
2. Recherche de la valeur minimale du sous-arbre droit du nœud contenant *x* ou bien de la valeur maximale du sous-arbre gauche du nœud contenant *x*. Soit y, cette valeur.
3. Suppression du nœud contenant *y*.
4. Remplacer *x* par *y*.

Algorithme Suppression d'un élément d'un arbre binaire de recherche
Programme principal
var *racine : ↑élément ; a : entier ;*
début *suppression (a) ;* ***fin***

procédure *suppression ;*
entrée *: racine :↑élément; x :***entier** *;*
sortie *: l'arbre binaire de recherche sans le nœud contenant x ;*
var *nœud, p, prédécesseur, parent :↑élément ; existe :* ***booléen*** *; y :* ***entier*** *;*
début *nœud := racine ;*

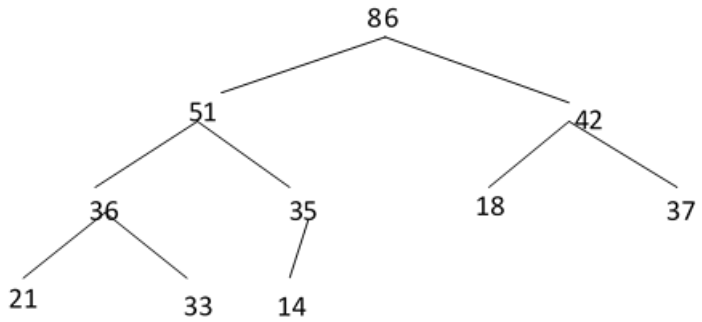
```
existe := faux ;
tant que (noeud ≠ nil) et non existe  faire
   (* recherche de la clé x dans l'arbre *)
        si (x = nœud↑.clé) alors existe := vrai
        sinon début prédecesseur := nœud ;
                    si (x < nœud↑.clé) alors nœud := nœud↑.fgauche
                   sinon nœud := nœud↑.fdroit ;
                fin ;
si (nœud = nil) alors signaler ('x n'existe pas dans l'arbre')
sinon si (x = nœud↑.clé) alors
    début
          p := nœud↑.fdroit ;
        (* recherche de la clé minimale du sous-arbre droit *)
         si (p ≠ nil) alors tant que (p↑.fgauche ≠ nil) faire
                 début   parent := p ;
                          p := p↑.fgauche ;
                 fin ;
        (* recherche de la clé maximale du sous-arbre gauche *)
        sinon début p := nœud↑.fgauche ;
                   si (p ≠ nil) alors tant que (p↑.fdroit ≠ nil) faire
                       début    parent := p ;
                                p := p↑.fdroit ;
                       fin ;
         sinon (* le nœud contenant x n'a pas de successeur alors
                  suppression de ce nœud *)
                        si (nœud = racine) alors racine := nil
                        sinon prédecesseur↑.fdroit := nil ;
              fin ;
        si (nœud↑.fgauche ≠ nil) ou (nœud↑.fdroit ≠ nil) alors
                 début  (* remplacer x par y *)
                y := p↑.clé ;
                        nœud↑.clé := y ;
                        (* supprimer le nœud contenant y *)
                si (p↑.fgauche = nil) alors
                              parent↑.fgauche := p↑.fdroit
                        sinon parent↑.fdroit := p↑.fgauche ;
                 fin
        fin
fin ;
```

4) Calculer la complexité de l'algorithme.

La recherche de l'élément *x* à supprimer se fait en $O(log_2(n))=p$ où *p est la profondeur de l'arbre et n le nombre de nœuds. L'opération de recherche de l'élément du sous-arbre droit qui a la plus petite valeur pour remplacer x s'effectue aussi en* $O(log_2(n))$*, de même pour la recherche de la clé maximale du sous-arbre gauche. La complexité est donc* $O(log_2(n))$.

Exercice 3.9

Considérer l'arbre binaire suivant :



1) Quelles sont les propriétés de cette structure ?

Comme les clés associées aux nœuds internes de l'arbre sont supérieures à celles associées aux fils, la structure est un **tas**.

2) Supprimer '51' de la structure tout en préservant ses propriétés.

Pour supprimer '51'de l'arbre, nous procédons comme suit :

a) Déterminer la structure de tableau équivalente
b) Supprimer '51' du tableau
c) Construire le tas à partir du tableau obtenu

Ce qui donne :

a) tableau

86	51	42	36	35	18	37	21	33	14

b) suppression de '51' du tableau

86	42	36	35	18	37	21	33	14

c) construction du tas

86	42	36	35	18	37	21	33	14
86	42	37	35	18	36	21	33	14

3) Ecrire un algorithme de suppression d'un élément de la structure tout en préservant ses propriétés.

Algorithme :

1. obtenir la structure du tas sous forme de tableau
2. Supprimer l'élément du tableau
3. Construire le tas à partir du tableau obtenu en 2.

Exercice 3.10

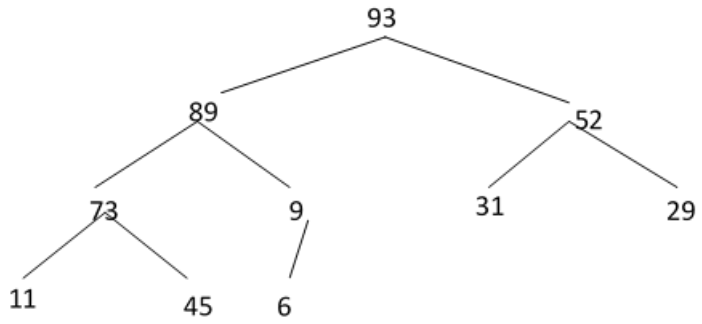
1) **Construction du tas**

Les étapes des différentes itérations et plus précisément des différentes permutations de la construction du tas sont montrées ci-dessous :

11	73	29	45	6	31	52	89	93	9

				9					6
			93					45	
		52				29			
	93		73						
			89				73		
93	11								
	89		11						
93	89	52	73	9	31	29	11	45	6

Le tas est schématisé comme suit :



2) **Recherche d'un élément dans un tas.**

```
procédure recherche-tas ;
entrée : T : tas [1..n] ; x : entier ;
sortie : position de x dans T ;

var trouve : booléen ;
    i : 1..n+1 ;
début
        trouve := faux ;
        i := 1 ;
        tant que (i <= n) et (non trouve) faire
            début
                si (T[i] = x) alors trouve := vrai ;
                i := i+1 ;
            fin ;
        si trouve alors retourner(i-1)
            sinon afficher ('x n'existe pas dans T) ;
fin ;
```

appel de la procédure : recherche-tas (90,1..n) ;

complexité :

Comme le vecteur n'est pas trié, au pire cas le vecteur est parcouru en entier à la recherche de l'élément. La complexité est donc en O(n).

3) **Insertion d'un élément dans un tas**

L'idée est d'insérer l'élément à la fin du vecteur puis d'appeler la procédure construire-tas.

procédure *insérer-tas (x :**entier** ; 1..n) ;*
entrée : *T : tas [1..n] ; x : **entier** ;*
sortie : *T : tas [1..n+1] ;*

début
T[n+1] :=x ;
Construire-tas (1..n+1) ;
fin

Complexité :

Comme toutes les branches du tas sont triées, la procédure construire-tas va toucher uniquement la branche qui contient l'élément à insérer pour l'insérer à la bonne place sur la branche correspondante. La complexité est donc O($\log_2 n$).

Exemple : Illustration de l'insertion de 90 dans le tas construit précédemment.

93	89	52	73	9	31	29	11	45	6	90
				90						9
93	90	52	73	89	31	29	11	45	6	9