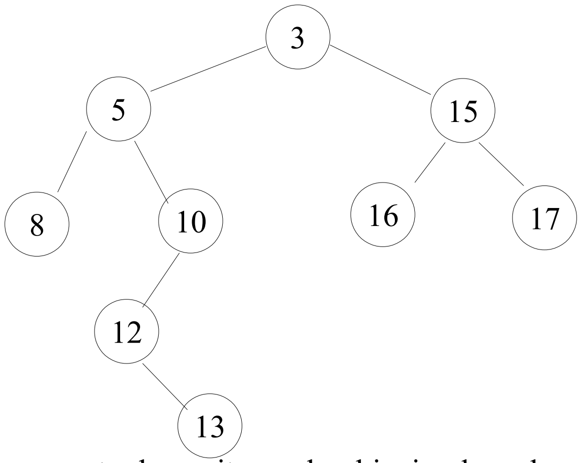
TD ARBRE BINAIRE

Q1. Construire tous les arbres binaires de recherche sur E= { 1 ,2 , 3 }

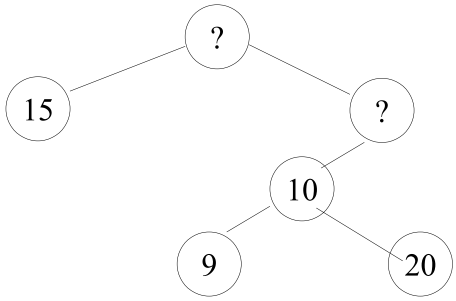
Q2. L'arbre de la figure ci dessous est il un arbre binaire de recherche sur son ensemble de clés ?



Q3. Redistribuer les clés pour que cet arbre soit un arbre binaire de recherche sur E de la même forme. On notera cet arbre A1.

Q4. Insérer dans l'arbre A1 les clés 4,7 et 14

Q5. Est il possible de compléter le schèma suivant pour obtenir un arbre binaire de recherche ?



Q6.1 Montrer que si un noeud d'un arbre binaire de recherche a deux fils, alors, dans la relation d'ordre sur E, son successeur appartient obligatoirement au sous arbre droit de x.

Q6.2 Montrer que si un noeud d'un arbre binaire de recherche a deux fils, alors, dans la relation d'ordre sur E, son successeur n'a pas de fils gauche et son prédecesseur n'a pas de fils droit.

Q7.Reprendre l'arbre binaire de recherche obtenu en Q4. Supprimer la clé 13.

Q8. Ecrire une fonction en pseudo code permettant de trouver le noeud de clé minimale d'un arbre binaire de recherche. Calculer sa complexité.

Q9. Proposer une procèdure itérative d'insertion d'une clé dans un arbre binaire de recherche. Calculer sa complexité. Votre procèdure retournera 0 si l'insertion s'est bien effectuée, -1 si la clé était déjà existante.

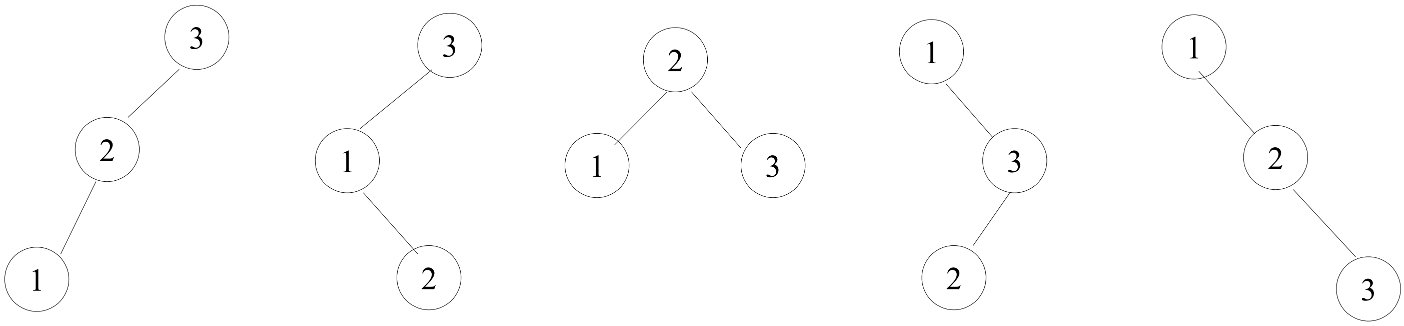
Q10. Proposer une procèdure récursive d'insertion d'une clé dans un arbre binaire de recherche. Calculer sa complexité.

Q11. Ecrire une fonction itérative qui permet de détacher le successeur d'un noeud x possédant un fils droit. La fonction doit retourner le noeud détaché. Pour faire ceci, vous ferez un dessin avec les 4 cas possibles pour le successeur de x.

Q12. Proposer une procèdure itérative de suppression d'une clé dans un arbre binaire de recherche. Calculer sa complexité. Votre procèdure retournera 0 si la suppression s'est bien effectuée, -1 si la clé n'a pas été trouvée.

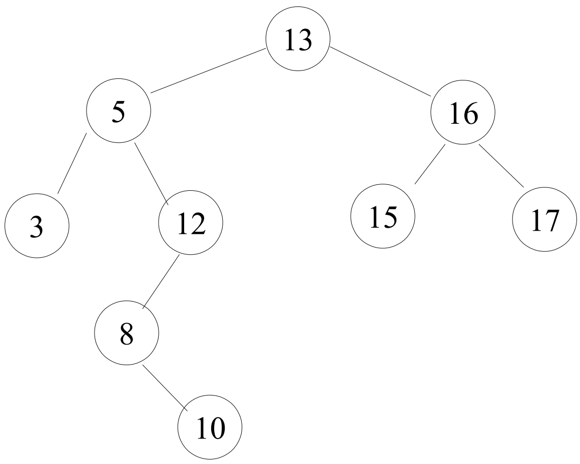
Correction exercice 1

Q1. Cinq arbres binaires de recherche sont possibles sur E :



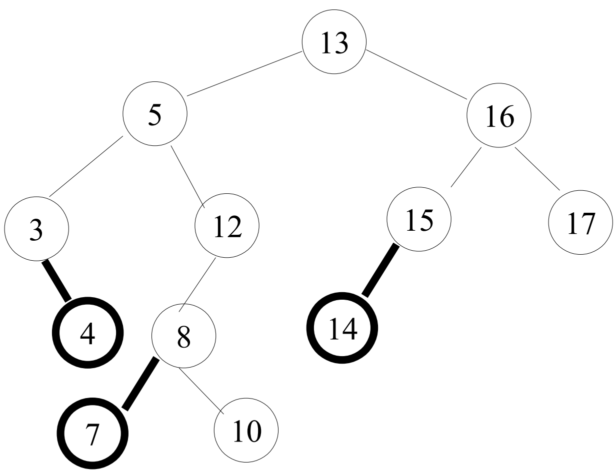
Q2. L'arbre A n'est pas un arbre binaire de recherche (la racine a une clé plus petite que son fils gauche , de même pour les noeuds de clés 5,10 et 15)

Q3. Voici l'arbre obtenu



Quelle est votre méthode pour trouver cette solution ?

Q4. Voici l'arbre obtenu



Q5. Non. Pourquoi ?

Q6.1 Soit x un noeud de A ayant deux fils : y à gauche et z à droite.

Soit **s** le successeur de x, montrons maintenant que **s** est présent dans le sous arbre droit de x.

s peut être dans :

- le sous arbre droit
- le sous arbre gauche
- le reste de l'arbre

s ne peut pas être dans le sous arbre gauche : celui ci contient uniquement des clés qui sont strictement inférieures à x.

Montrons **s** ne peut pas être dans le reste de l'arbre par l'absurde. Considérons que s est dans le reste de l'arbre, il existe alors un ancêtre **u** commun à **s** et **x,** et s et x sont dans des sous arbres différents par rapport à u.
Trois cas sont possibles :
1/ s est à gauche de u et x à droite de u : alors cle[s]<cle[u]<cle[x] : s ne peut pas être successeur
2/s est à droite de u et x à gauche de u: alors cle[x]<cle[u]<cle[s] : s ne peut pas être successeur, car u est meilleur candidat
3/s est confondu avec u :
    3.1/ x est à droite de u=s : alors cle[s]<cle[x] : s ne peut pas être successeur
    3.2/x est à gauche de u=s : cle[x]<cle[s] , le fils droit de x est un membre du sous arbre gauche de u , donc cle[fd[x]]<cle[s] , cle[x]<cle[fd[x]], donc fd[x] est un meilleur candidat de s

s ne peut donc pas être dans le reste de l'arbre, donc s est dans le sous arbre droit.

Q6.2 Montrons tout d'abord que si un noeud d'un arbre binaire de recherche a deux fils,
alors, dans la relation d'ordre sur E, son successeur n'a pas de fils gauche

Procédons par l'absurde : considérons un noeud a , qui a deux fils ,
D'apres le 6.1, on sait que son successeur est dans le sous arbre droit
Nous notons son successeur b
Supposons que son successeur ait un fils gauche , notons le c

alors la cle de c est plus grande que la clé de a (car c est dans le sous arbre droit de a)
la clé de c est plus petite que la clé de b (car c est le fils gauche de b)

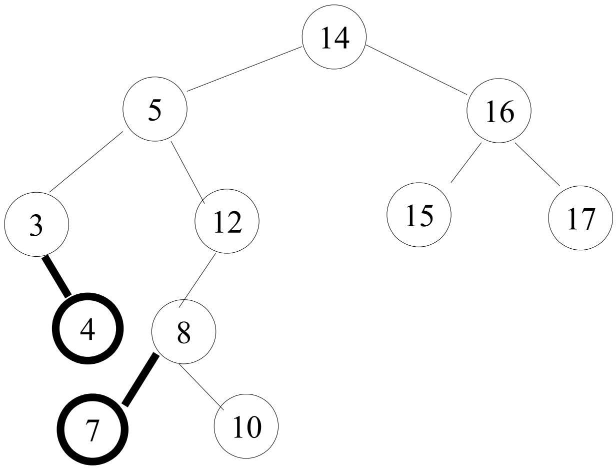
donc cle[a]<cle[c]<cle[b]
ce qui n'est pas possible, car b est le successeur de a , donc on a forcement
cle[a]<cle[b]<cle[c]

Nous avons donc montrer que que si un noeud d'un arbre binaire de recherche a deux fils,
alors, dans la relation d'ordre sur E, son successeur n'a pas de fils gauche

Il faut faire le même raisonnement pour

si un noeud d'un arbre binaire de recherche a deux fils, alors, dans la relation d'ordre sur E,
son prédecesseur n'a pas de fils droit.

Q7. Voici l'arbre obtenu



Quelle est la méthode utilisée ?

Q8. La fonction suivante retourne le noeud de clé minimale d'un arbre binaire de recherche

```
MIMIMUM(A)
x ← racine[A]
si x = NIL
     retourner NIL
finsi
tant que fg[x] différent de NIL alors
     x ← fg[x]
fin tant que
retourner x
```

A à chaque itération de la boucle la hauteur de **x** décroit d'au moins **un** et donc le nombre d'itérations de cette boucle est bornée par la hauteur de l'arbre.

La complexité de cette fonction est O(h) avec h la hauteur de l'arbre.

Q9.

```
INSERTION(A,val)

// Cas de l'arbre vide
si racine[A] = NIL alors
      z ← NOUVEL_ELEMENT
      cle[z] ← val
      racine[A] ←  z
      retourner 0
fin si

y ← NIL           // Correspond au coup précédent
x ← racine[A]     // Correspond au coup suivant

tant que x différent de NIL
      y ← x
      si cle[x] = val
            retourner -1
      sinon si val <cle[x] alors
            x ← fg[x]
      sinon
            x ← fd[x]
      finsi
fin tant que

// A ce point , on a trouve l'élément auquel on peut rajouter 1 fils :  c'est y
// On crée donc l'élément et on le met du bon coté
```

```
z ← NOUVEL_ELEMENT
cle[z] ← val
si val < cle[y] alors
      fg[y] ← z
sinon
      fd[y] ← z
finsi

retourner 0
```

La complexité est proportionnelle à la hauteur de l'arbre, car à chaque itération de la boucle la hauteur de **x** décroit d'au moins **un** et donc le nombre d'itérations de cette boucle est bornée par la hauteur de l'arbre.

Complexité : O(h), avec **h** la hauteur de l'arbre

Q10. Une première solution

```
INSERTION_RECURSIF(A,val)

// Cas de l'arbre vide
si racine[A] = NIL alors
      z ← NOUVEL_ELEMENT
      cle[z] ← val
      racine[A] ←  z
      retourner 0
fin si

retourner INSERTION_IN(racine[A],val)

INSERTION_IN(x,val)
      si cle[x] = val
            retourner -1
      sinon si val <cle[x] alors
            si fg[x] différent de NIL
                  retourner INSERTION_IN(fg[x],val)
            sinon
                  z ← NOUVEL_ELEMENT
                  cle[z] ← val
                  fg[x] ←  z
                  retourner 0
            finsi
      sinon
            si fd[x] différent de NIL
                  retourner INSERTION_IN(fd[x],val)
            sinon
                  z ← NOUVEL_ELEMENT
                  cle[z] ← val
                  fd[x] ←  z
                  retourner 0
            finsi
      finsi
```

Une autre solution

```
INSERTION_RECURSIF(A,val)

(rx,x) ← INSERTION_IN(racine[A],val)
racine[A] ← x
retourner rx

// Prend en entrée le nœud courant et la valeur à insérer
// Retourne un couple (code erreur , nœud de l'arbre)
INSERTION_IN(x,val)
      si x = NIL
            z ← NOUVEL_ELEMENT
            cle[z] ← val
```

```
            retourner (0,z)
      sinon si cle[x] = val
            retourner (-1,x)
      sinon si val <cle[x] alors
            (rx,fg[x]) ← INSERTION_IN(fg[x],val)
            retourner (rx,x)
      sinon
            (rx,fd[x]) ← INSERTION_IN(fd[x],val)
            retourner (rx,x)
      finsi
```

La complexité est proportionnelle à la hauteur de l'arbre, car à chaque appel de la fonction récursive la hauteur de **x** décroit d'au moins **un** et donc le nombre d'appels récurisf est bornée par la hauteur de l'arbre.

Complexité : O(h), avec **h** la hauteur de l'arbre

Q11.

```
// Prend en entrée un nœud x de l'arbre qui doit avoir un fils droit
// Retourne le successeur de x , et celui ci a été détaché de l'arbre
DETACHE_SUCCESSEUR(x)

// On recherche d'abord le minimum dans le sous arbre de droit de x
// pere_min sera le pere de min
(min,pere_min) ← MIN2(fd[x],x)

// Si le minimum est le fils droit de son père
si fd[pere_min] egal min alors
      fd[pere_min]← fd[min]
sinon
      fg[pere_min]← fd[min]
finsi

fd[min]← NIL
retourner min


// Cette fonction retrouve l'élément  minimum parmi les fils de w
// w doit être non nul
// Cette fonction retourne l'élement minimum et aussi le père de l'élement minimum
MIN2(w,pere_w)
tant que fg[w] différent de NIL alors
      pere_w ← w
      w ← fg[w]
fin tant que
retourner (w,pere_w)
```

Q12.

```
SUPPRESSION(A,val)

// Cas de l'arbre vide
si racine[A] = NIL alors
      retourner -1
fin si

y ← NIL          // Correspond au coup précédent (père de x)
x ← racine[A]    // Correspond au coup suivant

tant que x différent de NIL
      si cle[x] = val
            SUPPRESS(x,y,A)
            retourner 0
      sinon si val <cle[x] alors
            y ← x
            x ← fg[x]
      sinon
            y ← x
```

```
            x ← fd[x]
      finsi
fin tant que
retourner -1

// Réalise la suppression du nœud x dans A, pere_x est le pere de x
SUPPRESS(x,pere_x,A)
// Si x a 0 ou 1 fils
si fd[x] = NIL OU fg[x] = NIL
      SUPPRESS01(x,pere_x,A)
// Si x a 2 fils
sinon
      y ← DETACHE_SUCCESSEUR(x)
      // On copie les valeurs de y dans x
      cle[x] ← cle[y]
finsi

// Réalise la suppression du nœud x dans A, pere_x est le pere de x
// le fils droit de x doit être  NIL
SUPPRESS01(x,pere_x,A)
// Recherche du fils de x
si fd[x] = NIL
      fils = fg[x]
sinon
      fils = fd[x]
finsi

// Si on souhaite supprimer la racine
si pere_x égal NIL
      racine[A]← fils
sinon
      // Si x est le fils droit de son père
      si fd[pere_x] egal x alors
            fd[pere_x]← fils
      sinon
            fg[pere_x]← fils
      finsi
finsi
```