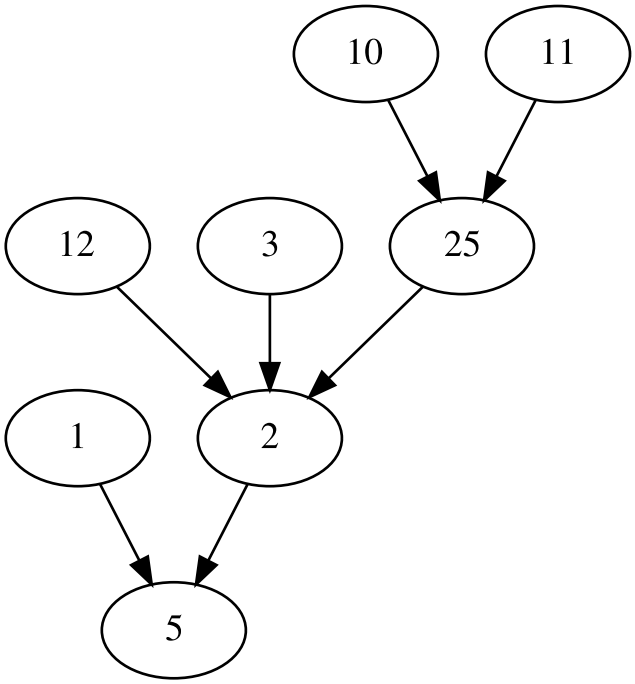
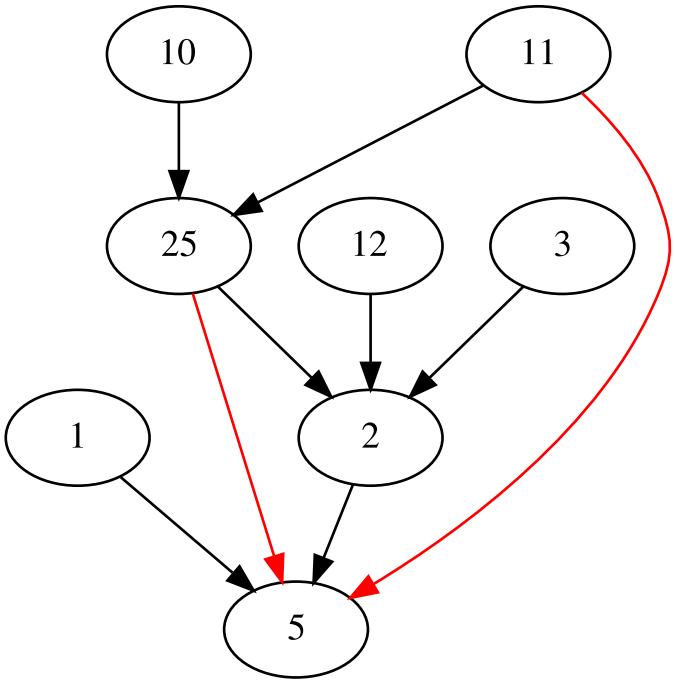
Cours 3 Sophie Laplante 29/09/14


```
def root(u):
    while Pere[u] != u:
        u = Pere[u]
    return u
```



Ex: root(11)




```
def root2(u):
    while Pere[u] != u:
        Pere[u] = root2(Pere[u])
    return Pere[u]
```

Compléxité et correction d'un algo

1. *Étude de cas*: ABR

1.1. Notations

Étant donné un algo `A` `count_A(x) = # étapes élémentaires` effectu"es lorsqu'on lance `A` sur un donnée `x` `cout_A(n) = max{cout_A(x) : x de taille n}`

Coût moye que prend A sur un x trié au hasard parmi les x de taille n. C'est pour obtenir une notion de coût qui est moins sensible aux cas extrêmes.

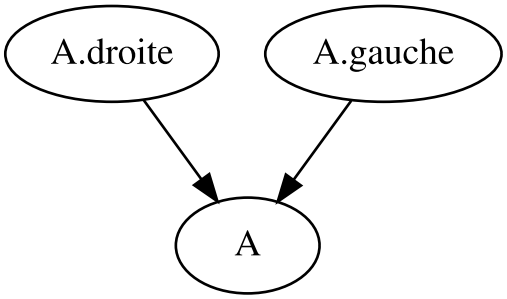
Ex: Tri rapide : x tableau déja trié : O(n) comparaison. En moyenne : O(n*log(n)) x tableau trié dans l'ordre inverse : $O(n^2)$

2. Les ABR:

Definition des ABR sert à: 1. Consevoir l'algo 2. prouver sa complexité 3. démontrer la correction

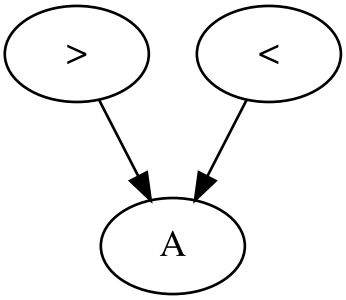
Définition

1 .Un ABR `A` est soit *Un arbre vide* Une racine `A` avec 2 sous-arbres `A.droite` et `A.gauche`



1. Un arbre binaire *étiquetté* a des valeurs à chaque noeud `A.val`
2. **La propriété ABR** Pour chaque noeud d'un ABR, `A` `A.val` *> v Pour toute valeur v dans* `A.gauche` ***ET*** `A.val` < v Pour toute valeur v dans `A.droite`

On suppose toutes les valeurs distinctes



Pour l'implémentation des ABR: (*Python*) Une classe avec 3 attributs

```
A.val
A.gauche # ABR à gauche
A.droite # ABR à droite
```

```
def trouver_ABR(A, v):

    if A.val == c:
        return A
    if A.val > v:
        return trouver_ABR(A.gauche, v) if A.gauche else None
    if A.val < v:
        return trouver_ABR(A.droite, v) if A.droite else None
```

On va montrer

Prop 1 L'algo est correct Preuve Prop 1 :