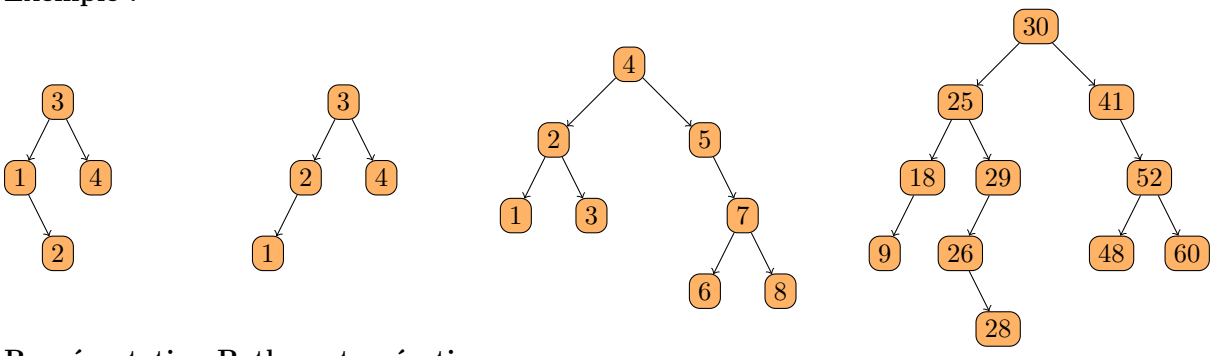
Les arbres binaires de recherche

1 Notion d'arbre binaire de recherche

Un **arbre binaire de recherche** est un arbre binaire où, pour chaque noeud, toutes les valeurs du sous-arbre gauche sont plus petites que la valeur du noeud et toutes les valeurs du sous-arbre droit sont plus grandes que la valeur du noeud.

Exemple :



Représentation Python et opérations
La représentation en Python d'un arbre binaire de recherche est réalisée, comme pour un arbre binaire du chapitre précédent, à l'aide de la classe `Noeud`.
Les fonctions `taille` et `hauteur` restent valable ainsi que les différents parcours. En particulier le parcours infixe permet l'affichage des valeurs d'un arbre de recherche par ordre croissant.
Les fonctions spécifiques des arbres binaires de recherche que nous allons voir vont supposer ou garantir leurs propriétés.

2 Recherche d'un élément

L'intérêt d'un arbre binaire de recherche est l'éfficacité de la recherche d'un élément.

```
1 def appartient(arb, v):
2     '''détermine si la valeur v appartient à l'ABR arb'''
```

Une première condition d'arrêt est le cas où l'arbre est vide.

```
1     if arb is None:
2         return False
```

Si la valeur de la racine est la valeur recherchée `v`, on est également dans un cas de base.

```
1     elif arb.valeur == v:
2         return True
```

Sinon, soit `v` est plus petit que la valeur de la racine et alors, si `v` appartient à l'arbre, `v` appartient au sous-arbre gauche.

```
1     elif v < arb.valeur:
2         return appartient(arb.gauche, v)
```

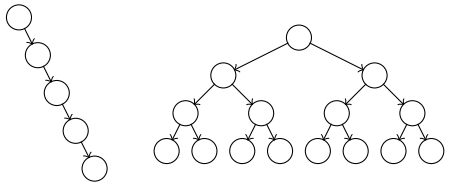
Soit `v` est plus grand que la valeur de la racine et alors, si `v` appartient à l'arbre, `v` appartient au sous-arbre droit.

```
    else:
        return appartient(arb.droite, v)
```

On obtient le programme suivant.

```
def appartient(arb, v):
    '''détermine si la valeur v appartient à l'ABR arb'''
    if arb is None:
        return False
    elif arb.valeur == v:
        return True
    elif v < arb.valeur:
        return appartient(arb.gauche, v)
    else:
        return appartient(arb.droite, v)
```

Cet algorithme réalise un nombre d'opérations au plus égal à sa hauteur. Cette dernière dépend de la forme de l'arbre.
Dans le pire des cas, l'arbre est filiforme et la hauteur est égale au nombre de noeuds. La recherche est alors de complexité linéaire car susceptible de parcourir tous les noeuds de l'arbre comme pour une recherche dans une liste chaînée.
Dans le meilleur des cas, l'arbre est parfait. On élimine alors la moité des éléments à chaque étape comme dans une recherche dichotomique qui est de complexité logarithmique.



3 Ajout d'un élément

Pour construire un arbre binaire de recherche, nous allons utiliser une fonction `ajout`.

```
def ajout(arb, v):
    '''ajoute l'élément v à l'ABR arb et retourne un nouvel ABR'''
```

Le cas de base est l'ajout d'un élément dans un arbre vide.

```
    if arb is None:
        return Noeud(v, None, None)
```

Si la valeur à ajouter `v` est inférieur à la valeur de la racine, on ajoute `v` au sous-arbre gauche. La valeur de la racine reste inchangée, ainsi que le sous-arbre droit.

```
    elif v < arb.valeur:
        return Noeud(v, ajout(arb.gauche, v), arb.droit)
```

Sinon, `v` est supérieur à la valeur de la racine. On ajoute alors `v` au sous-arbre droit, la valeur de la racine reste inchangée, ainsi que le sous-arbre gauche.

```
    else :
        return Noeud(v, arb.gauche, ajout(arb.droit, v))
```

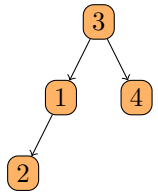
On obtient le programme suivant.

```
def ajout(arb, v):
    '''ajoute l'élément v à l'ABR arb et retourne un nouvel ABR'''
    if arb is None:
        return Noeud(v, None, None)
```

```
    elif v < arb.valeur:
        return Noeud(v, ajout(arb.gauche, v), arb.droit)
    else :
        return Noeud(v, arb.gauche, ajout(arb.droit, v))
```

Si par exemple on exécute les cinq instructions de gauche, on obtient l'arbre représentée à droite :

```
arb = None
arb = ajoute(arb, 3)
arb = ajoute(arb, 1)
arb = ajoute(arb, 2)
arb = ajoute(arb, 4)
```



4 Encapsulation dans un objet

Comme pour les listes chaînées, on peut encapsuler nos arbres binaires de recherche dans une classe ici appelé `ABR`.

```
class ABR:
    '''un arbre binaire de recherche'''
    def __init__(self):
        self.racine = None

    def ajouter(self, v):
        self.racine = ajout(self.racine, v)

    def contient(self, v):
        return appartient(self.racine, v)
```

Cette classe comporte un unique attribut `racine` initialisé à `None` et deux méthodes `ajouter` et `contient` qui correspondent aux fonctions `ajout` et `appartient`. Toutes les autres fonctions comme `taille`, `hauteur`, `parcours_infixe`, etc, sont également à ajouter à cette classe.