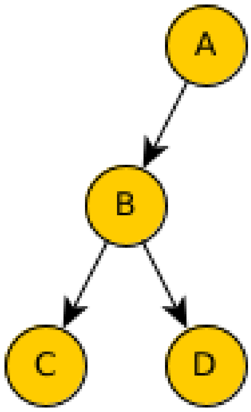
1 Algos arbre binaires

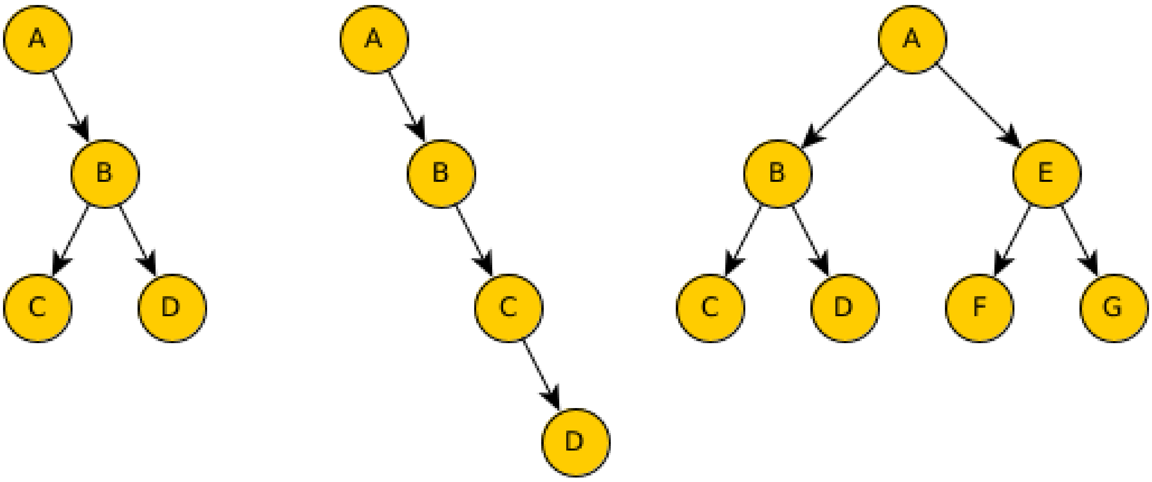
```
class Node:
    def __init__(self, left, value, right):
        self.value = value
        self.left = left
        self.right = right
```




```
a = Node(Node(Node(None,"C",None),
              "B",
              Node(None,"D",None)),
         "A",
         None)
```

```
a = Node(
        Node(
            Node(
                None,
                "C",
                None),
            "B",
            Node(
                None,
                "D",
                None)),
    "A",
    None)
```

1.1 Arbres exemples




```
arbre1 = Node(
    None,
    "A",
    Node(
        Node(None, "C", None),
        "B",
        Node(None, "D", None)
    )
)

arbre2 = \
Node(
    None,
    "A",
    Node(
        None,
        "B",
        Node(
            None,
            "C",
            Node(
                None,
                "D",
                None
                )
        )
```

```
    )
)

arbre3 = \
Node(
    Node(
        Node(None, "C", None),
        "B",
        Node(None, "D", None)
    ),
    "A",
    Node(
        Node(None, "F", None),
        "E",
        Node(None, "G", None)
    )
)
```

2 Algos sur les arbres

2.1 Calcul de la taille

```python
# Algo et application aux exemples
def taille(arbre):
    if arbre is None:
        return 0
    return 1+taille(arbre.left)+taille(arbre.right)

print("Taille arbre 1 : ", taille(arbre1))
print("Taille arbre 2 : ", taille(arbre2))
print("Taille arbre 3 : ", taille(arbre3))
```

```
Taille arbre 1 :  4
Taille arbre 2 :  4
Taille arbre 3 :  7
```

2.2 Calcul de la hauteur

```python
# Algo et application aux exemples
def hauteur(arbre):
    if arbre is None:
```

```python
        return 0
    return 1 + max(hauteur(arbre.left), hauteur(arbre.right))

print("Hauteur arbre 1", hauteur(arbre1))
print("Hauteur arbre 2", hauteur(arbre2))
print("Hauteur arbre 3", hauteur(arbre3))
```

```
Hauteur arbre 1 3
Hauteur arbre 2 4
Hauteur arbre 3 3
```

2.3 Parcours préfixes, infixes, postfixes

```python
# Préfixe
def prefixe(arbre):
    if arbre is None:
        return
    print(arbre.value, end=" ")
    prefixe(arbre.left)
    prefixe(arbre.right)

prefixe(arbre1); print()
prefixe(arbre2); print()
prefixe(arbre3); print()
```

```
A B C D
A B C D
A B C D E F G
```

```python
# Infixe
def infixe(arbre):
    if arbre is None:
        return
    infixe(arbre.left)
    print(arbre.value, end=" ")
    infixe(arbre.right)

infixe(arbre1); print()
infixe(arbre2); print()
infixe(arbre3); print()
```

```
A C B D
A B C D
C B D A F E G
```

```python
# Post
def postfixe(arbre):
    if arbre is None:
        return
    postfixe(arbre.left)
    postfixe(arbre.right)
    print(arbre.value, end=" ")

postfixe(arbre1); print()
postfixe(arbre2); print()
postfixe(arbre3); print()
```

```
C D B A
D C B A
C D B F G E A
```

2.4 Parcours en largeur

```python
class File:
    def __init__(self):
        self.file = list()
    def enfile(self, valeur):
        self.file.append(valeur)
    def defile(self):
        assert len(self.file) > 0, "La file est vide"
        return self.file.pop(0)
    def nbelts(self):
        return len(self.file)

# Algo et application aux exemples

# VARIANTE 1 : on met les noeuds vides dans la file
def parcours_largeur(arbre):
    f = File()
    f.enfile(arbre)
    while f.nbelts() > 0:
        cur = f.defile()
        if cur is None:
            continue
```

```
        print(cur.value, end=" ")
        f.enfile(cur.left)
        f.enfile(cur.right)
    print()
parcours_largeur(arbre1)
parcours_largeur(arbre2)
parcours_largeur(arbre3)
```

```
A B C D
A B C D
A B E C D F G
```

```
# VARIANTE : on ne met pas les noeuds vides dans la file
def parcours_largeur(arbre):
    f = File()
    f.enfile(arbre)
    while f.nbelts() > 0:
        cur = f.defile()
        print(cur.value, end=" ")
        if cur.left is not None:
            f.enfile(cur.left)
        if cur.right is not None:
            f.enfile(cur.right)
    print()
parcours_largeur(arbre1)
parcours_largeur(arbre2)
parcours_largeur(arbre3)
```

```
A B C D
A B C D
A B E C D F G
```

3 Exercices

3.1 Exercice 1

Ecrire une fonction `affiche(arbre)` qui imprime un arbre sous la forme suivante

:

- si l'arbre est vide, on n'imprime rien
- pour un noeud, on imprime successivement
  - une parenthèse ouvrante
  - son sous-arbre gauche (récursivement)
  - sa valeur
  - son sous-arbre droit (récursivement)
  - une parenthèse fermante

Par exemple, l'arbre a du début doit afficher `(((C)B(D))A)`

```
def affiche(arbre):
    if arbre is None:
        print("", end="")
        return
    print("(", end="")
    affiche(arbre.left)
    print(arbre.value, end="")
    affiche(arbre.right)
    print(")", end="")

affiche(a); print()
affiche(arbre1); print()
affiche(arbre2); print()
affiche(arbre3); print()
```

```
(((C)B(D))A)
(A((C)B(D)))
(A(B(C(D))))
(((C)B(D))A((F)E(G)))
```

3.2 Exercice 2

Dessiner l'arbre binaire pour lequel le programme précédent produit la sortie `(A((B)C))`. De manière générale, expliquer comment retrouver la forme de l'arbre dont l'affichage est donné.

```
  A
 / \
   C
  / \
 B
```

```
         /  \
```

On navigue en profondeur dans les parenthèses, en partant de la racine (une seule parenthèse) et en dessinant au fur et à mesure ce qu'il y a à gauche et à droite, récursivement.

3.3 Exercice 3

Donner 4 arbres de taille 3, tous différents, pour lesquels le parcours infixe affiche `123`.

On peut dessiner la forme des arbres en premier :

```
    x         x         x        x            x
   / \       /         /          \            \
  x   x     x         x            x            x
           /           \            \          /
          x             x            x        x
```

et ensuite les "remplir" avec les valeurs dans l'ordre qu'il faut pour que ça donne 123 en infixe :

```
    2         3         3        1            1
   / \       /         /          \            \
  1   3     2         1            2            3
           /           \            \          /
          1             2            3        2
```

3.4 Exercice 4

Ecrire une fonction `indente` qui affiche un arbre de manière indentée, en affichant un tiret pour les sous-arbres vides.

Exemple de sortie du programme pour l'arbre de la figure \label{fig:pyrepr1} (chaque point · représente un espace)

```
A
·B
··C
···-
···-
```

```
..D
...-
...-
.-

# Il faut faire **DESCENDRE** l'information de profondeur aux enfants =>
↪  argument
def indente(arbre, prof=0):
    if arbre is None:
        print("."*prof+"-")
        return
    print("."*prof+arbre.value)
    indente(arbre.left, prof+1)
    indente(arbre.right, prof+1)

indente(a)
print("---------------")
indente(arbre1)
print("---------------")
indente(arbre2)
print("---------------")
indente(arbre3)

A
.B
..C
...-
...-
..D
...-
...-
.-
---------------
A
.-
.B
..C
...-
...-
..D
```

```
...-
...-
---------------
A
.-
.B
..-
..C
...-
...D
....-
....-
---------------
A
.B
..C
...-
...-
..D
...-
...-
.E
..F
...-
...-
..G
...-
...-
```

3.5 Exercice 5

- Ecrire une fonction `parfait(h:int)` qui prend en argument un nombre entier h supérieur ou égal à zéro et qui renvoie un arbre binaire parfait de hauteur h
- Ecrire une fonction `peigne_gauche(h:int)` qui prend en argument un nombre entier h supérieur ou égal à zéro et qui renvoie un peigne gauche (tous les sous-arbres droits sont vides) de hauteur h
- Ecrire une fonction `est_peigne_gauche(arbre)` qui renvoie `True` si `arbre` est un peigne

gauche.

```python
# Un arbre parfait a une racine, et chacun de ses sous arbres gauches et
 ↪  droites est aussi un sous-arbre
# parfait de hauteur h-1 => définition et implémentation récursives
def parfait(h):
    if h==0:
        return None
    return Node(parfait(h-1), h, parfait(h-1))
affiche(parfait(3))
print()
affiche(parfait(5))
```

```
(((1)2(1))3((1)2(1)))
(((((1)2(1))3((1)2(1)))4(((1)2(1))3((1)2(1))))5((((1)2(1))3((1)2(1)))4(((1)2(1))
```

```python
# Un peigne gauche a une racine, son sous arbre droit est vide, et son sous
 ↪  arbre gauche est aussi
# un peigne gauche de hauteur h-1 => définition et implémentation récursives
def peigne_gauche(h):
    if h == 0:
        return None
    return Node(peigne_gauche(h-1), h, None )

affiche(peigne_gauche(5))
```

```
(((((1)2)3)4)5)
```

```python
# Un arbre est un peigne gauche si pour chaque noeud, son sous-arbre droit
 ↪  est vide et son
# sous-arbre gauche est aussi un peigne gauche => définition et
 ↪  implémentation récursive
# Cas de base : un arbre vide est un peigne_gauche
def est_peigne_gauche(arbre):
    if arbre is None:
        return True
    return (est_peigne_gauche(arbre.left)) and (arbre.right is None)

print(est_peigne_gauche(arbre1))
print(est_peigne_gauche(arbre2))
print(est_peigne_gauche(arbre3))
print(est_peigne_gauche(peigne_gauche(3)))
```

```
False
False
False
True
```

3.6 Exercice 6

Déterminer quel type de parcours est effectué si dans la fonction `parcours_largeur` on remplace la file (FIFO) par une pile (LIFO).

Quel avantage peut-il y avoir par rapport aux méthodes de parcours récursifs ?

```python
class Pile:
    def __init__(self):
        self.pile = list()
    def enpile(self, valeur):
        self.pile.append(valeur)
    def depile(self):
        assert len(self.pile) > 0, "La pile est vide"
        return self.pile.pop()
    def nbelts(self):
        return len(self.pile)

def parcours_avec_pile(arbre):
    p = Pile()
    p.enpile(arbre)
    while p.nbelts() > 0:
        cur = p.depile()
        print(cur.value, end=" ")
        if cur.left is not None:
            p.enpile(cur.left)
        if cur.right is not None:
            p.enpile(cur.right)
    print()

parcours_avec_pile(arbre1)
parcours_avec_pile(arbre2)
parcours_avec_pile(arbre3)
```

```
A B D C
```

A B C D
A E G F B D C

C'est un parcours en **profondeur**. Avantage : la profondeur des arbres qu'on peut traiter n'est pas limitée par la limite de la taille de la pile d'appel (recursion error), on peut faire des arbres de profondeur > 1000 ou 2000 sans pb.