Cours 4 : Les arbres binaires

Définition

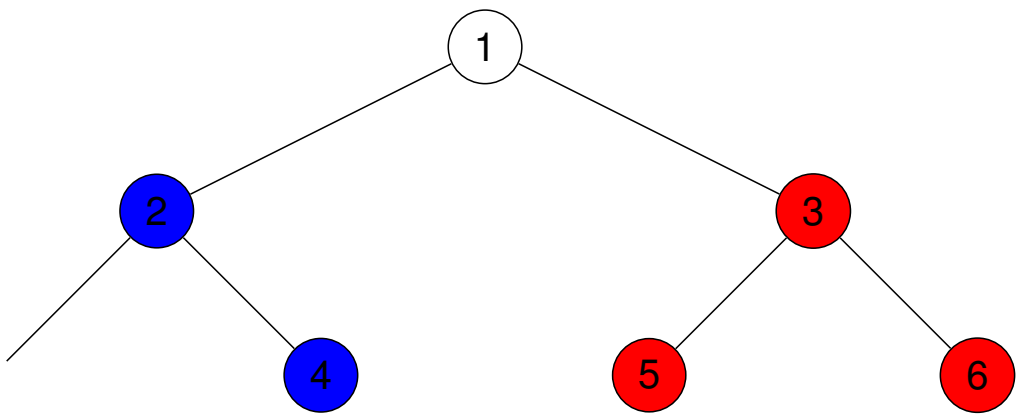
Implémentation

Manipulation

Définition

Arbre Binaire

Un arbre binaire est un arbre qui possède au maximum deux sous-arbres (d'où le binaire)



Deux implémentations possibles

Version itérative

- Un arbre binaire est constitué de sommets
- Chaque sommet possède deux sommets fils

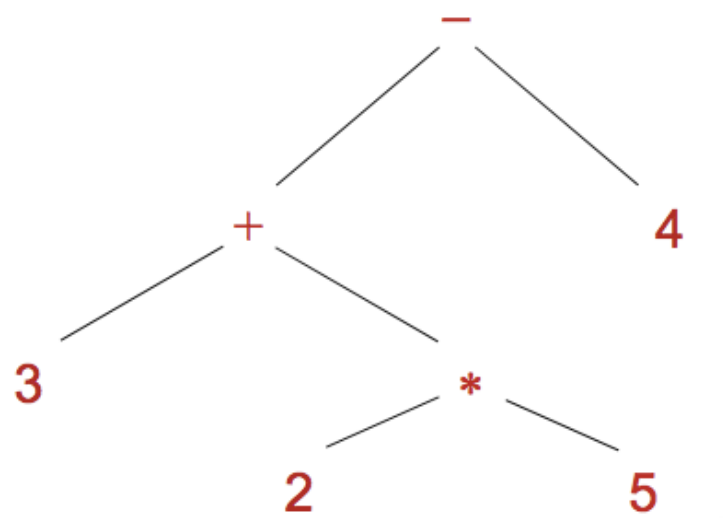
Version récursive

- Un arbre binaire peut être vide
- Un arbre binaire possède un sommet (étiqueté ou pas)
- Un arbre binaire possède deux sous-arbres " fils "

Utilisation

Enormément d'applications, que ce soit dans le domaine informatique ou pas :

- Expressions mathématiques : $3 + 2 \times 5 - 4$



Autres exemples

Résultats d'un tournoi à élimination directe (tennis par exemple)

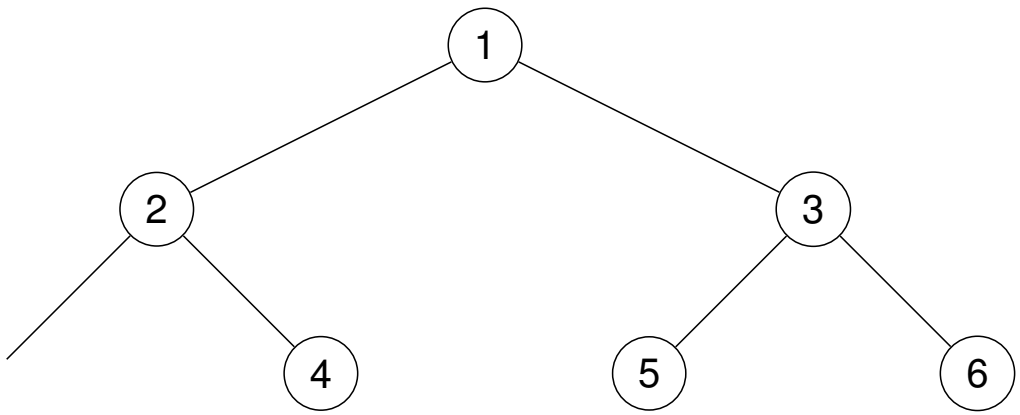
Arborescence de si-alors-sinon (voir les arbres de décision pour les tris vus à la première séance)

Un peu de vocabulaire

Un arbre est constitué de sommets (ou sommets)

Ces sommets sont reliés par des arcs (ou arêtes) orientés : père → fils

Il existe dans tout arbre un sommet qui n'est le point d'arrivée d'aucun arc : c'est le sommet racine



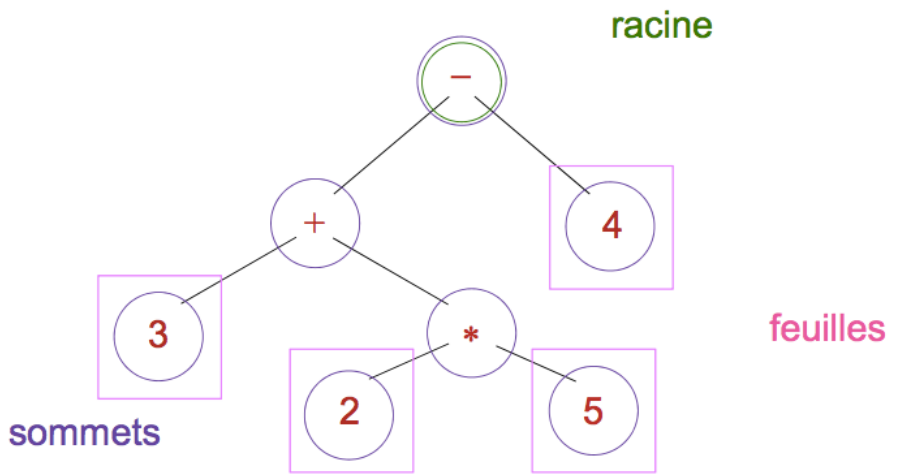
Un peu de vocabulaire (2)

Tout autre sommet est la racine d'un sous-arbre de l'arbre principal

Un sommet qui n'est le point de départ d'aucun arc est appelé feuille

Pour les arbres binaires, on distinguera de façon visuelle le fils gauche du fils droit

Un exemple



Un peu de vocabulaire (3)

La hauteur

La hauteur d'un sommet est la longueur du plus long chemin de ce sommet aux feuilles qui en dépendent plus 1

- C'est le nombre de sommets du chemin
- La **hauteur d'un arbre** est la hauteur de sa racine
- L'arbre vide a une hauteur 0
- L'arbre réduit à une racine étiqueté a une hauteur 1

Un peu de vocabulaire (4)

La profondeur

d'un sommet est le nombre de sommets du chemin qui va de la racine à ce sommet

- La racine d'un arbre est à une profondeur 0
- La profondeur d'un sommet est égale à la profondeur de son père plus 1
- Si un sommet est à une profondeur p, tous ses fils sont à une profondeur p+1

Tous les sommets d'un arbre de même profondeur sont au même niveau

Arbre Binaire en Python

```
1  class Sommet:
2    def __init__(self, val, fg=None, fd=None):
3      self.__val = val
4      self.__fg = fg
5      self.__fd = fd
6    def get(self):
7      return self.__val
8    def fd(self):
9      return self.__fd
10   def fg(slef):
11     return self.__fg
12   def aFG(self):
13     return self.__fg != None
14   def aFD(self):
15     return self.__fd != None
16   def estFeuille(self):
17     return self.__fg == None and self.__fd == None
```

Arbre Binaire en Python

```
1 class ArbreBinaire:
2   def __init__(self):
3     self.__racine = None
```

Premier traitement sur un arbre

L'affichage : Trois façons d'afficher

- Préfixe
- Infixe
- Postfixe

Affichages

Préfixe :

- On affiche la racine, puis le sous-arbre gauche, puis le sous-arbre droit

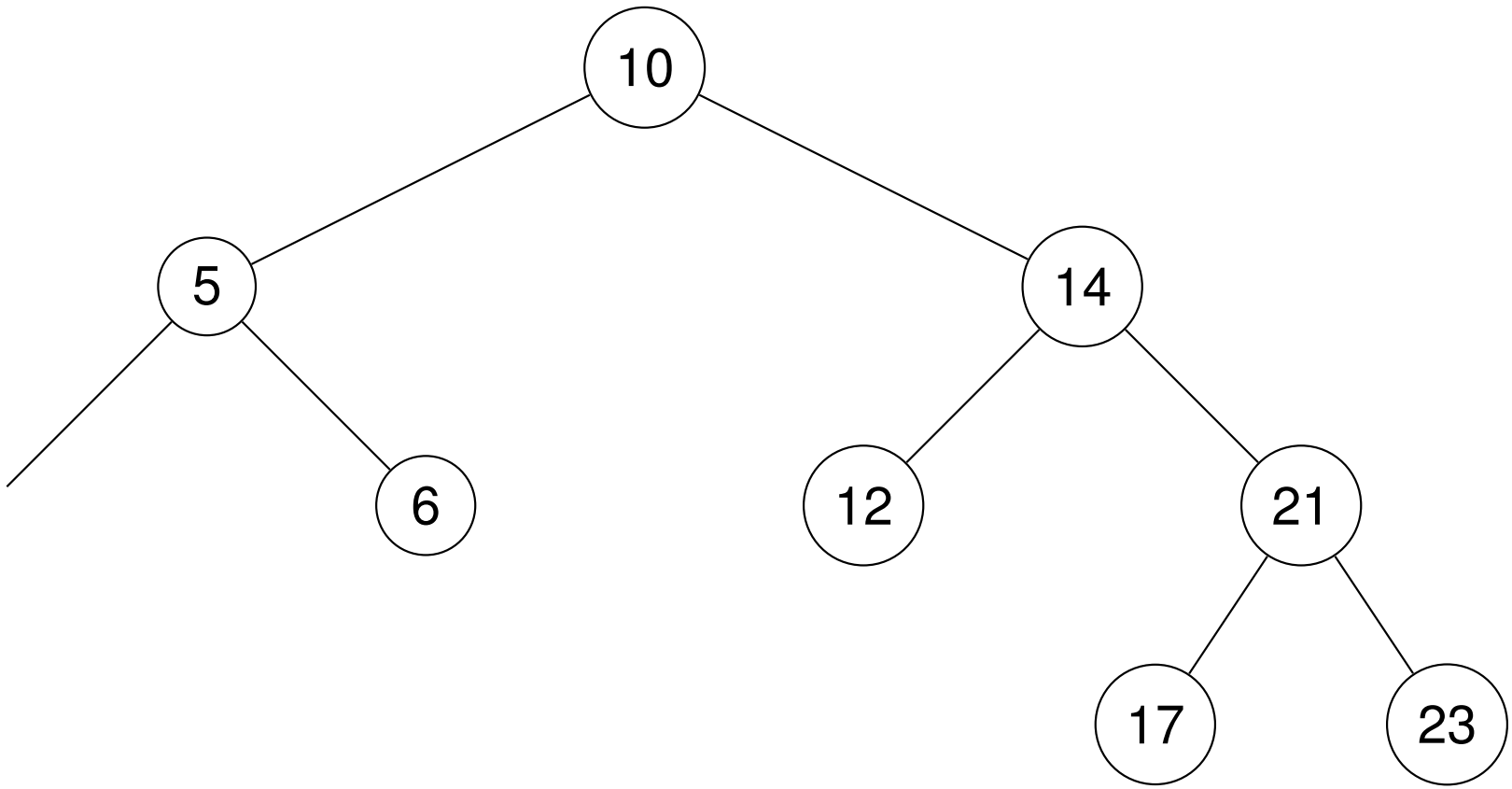
Infixe :

- On affiche le sous-arbre gauche, puis la racine, puis le sous-arbre droit

Postfixe :

- On affiche le sous-arbre gauche, puis le sous-arbre droit, puis la racine

Un exemple



Un exemple (suite)

Préfixe

> (10 (5 (6, ())) (14 (12 () ()) (21 (17 () ()) (23 () ()))))

Infixe

> ((() 5 ()) 6 ()) 10 ((() 12 ()) 14 ((() 17 ()) 21 (() 23 ())))

Postfixe

> ((() (() () 6) 5) ((() () 12) ((() () 17) (() () 23) 21) 14) 10)

Deuxième traitement

Recherche d'un élément

Au moins deux façons de faire :

I **DFS** (Depth-First Search) → recherche en profondeur d'abord

II **BFS** (Breadth-First Search) → recherche en largeur d'abord

DFS en détail

- On cherche à la racine
- S'il n'y est pas :
- On cherche dans le fils gauche
- Puis s'il n'était pas dans le fils gauche, on cherche dans le fils droit

Idée : explorer à fond chaque branche avant de passer à la suivante

On s'arrête quand on a trouvé ou qu'il n'y a plus de branches à explorer

Parcours en profondeur (DFS) - version récursive

Dans la classe `ArbreBinaire`

```
1 def cherche_dfs(self,val):
2     '''
3     Retourne le premier sommet de valeur val,  sinon None
4     '''
5     if not self.est_vide():
6       return self.__racine.chercher_dfs(val)
```

Parcours en profondeur (DFS) - version récursive

Dans la classe Sommet

```
1  def cherche_dfs(self,val):
2      '''
3      Retourne le premier sommet de valeur val,  sinon None
4      '''
5       if self.__val == val:
6          return self
7       if self.aFG():
8          sommet=self.__fg.cherche_dfs(val)
9          if sommet is not None:
10           return sommet
11      if  self.aFD():
12         sommet=self.__fd.cherche_dfs(val)
13         if sommet is not None:
14           return sommet
15      return None
```

Parcours en profondeur (DFS) avec une Pile (version itérative)

Dans la classe `ArbreBinaire`

```
1
2  def cherche_dfs(self,val):
3      '''
4      Retourne le premier sommet de valeur val, sinon None
5      '''
6      assert self is not None
7      pile = Pile()
8      pile.empiler(self.__racine)
9      while not pile.est_vide():
10         sommet = pile.depiler()
11         if sommet.__val == val:
12                 return sommet
13         if sommet.aFD():
14           pile.empiler(sommet.fd())
15         if sommet.aFG():
16           pile.empiler(sommet.fg())
17     return None
```

BFS en Python (version itérative)

Dans la classe `ArbreBinaire`

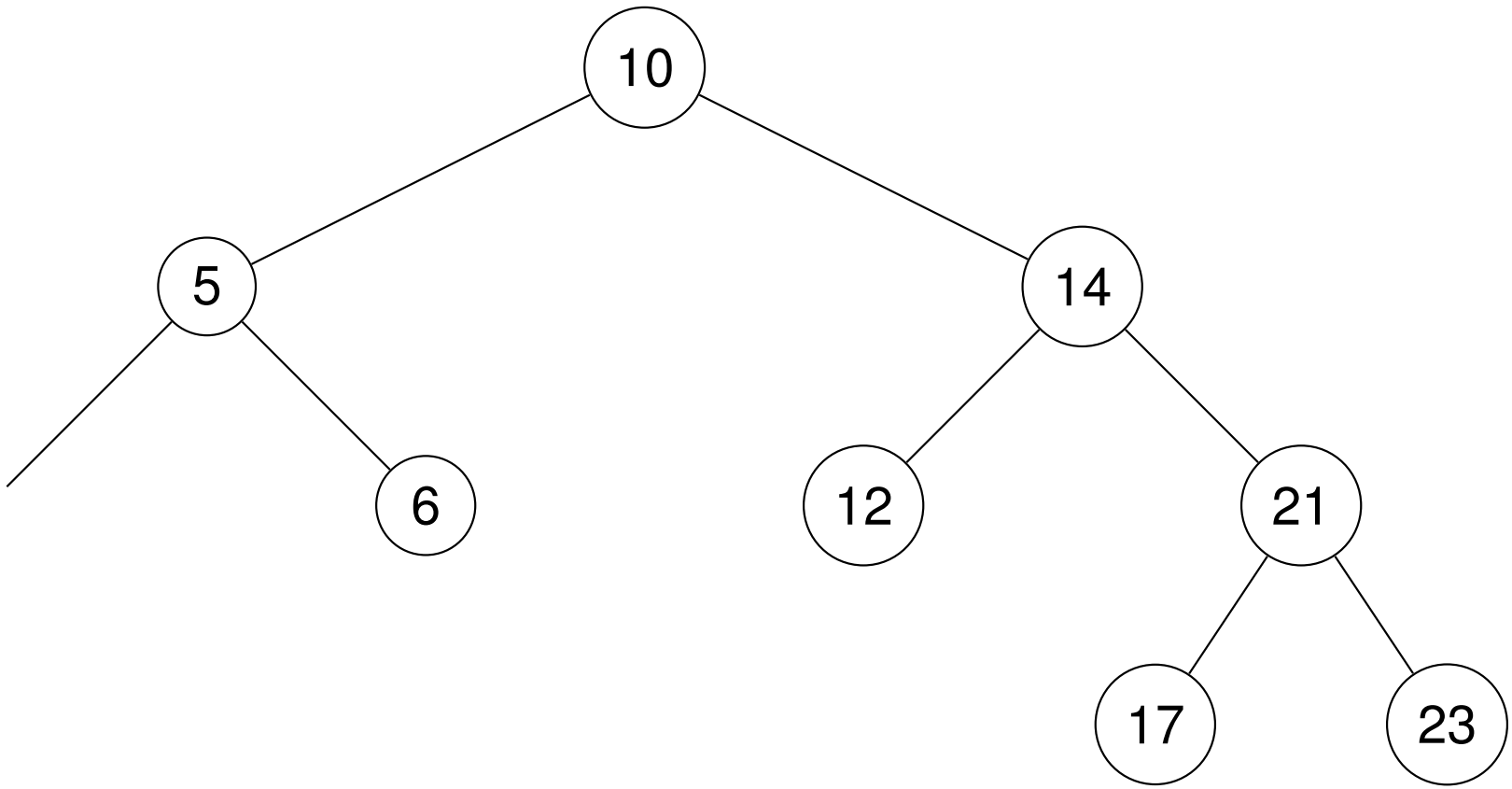
```
1  def cherche_bfs(self, valeur):
2    '''Retourne le noeud de valeur valeur, sinon None'''
3
4    if self.__racine != None:
5      f = File()
6      f.ajouter(self.__racine)
7      while not f.est_vide():
8        sommet = f.supprimer()
9          if sommet.get() == valeur:
10           return sommet
11         if sommet.fg() != None:
12           f.empile(sommet.fg())
13         if sommet.fd() != None:
14           f.ajouter(sommet.fd())
15   return None
```

Réflexions

On reconnaît encore une fois un fonctionnement récursif

Problème : si l'élément est dans le sous arbre droit de la racine, on va quand même explorer tout le sous-arbre de gauche

Exemple



Avec DFS : on cherche 4

Racine = 1 non

- On explore le fils gauche
- Racine = 2 non
- On explore le fils gauche

? Racine = 4 TROUVé

Avec DFS (2) : on cherche 3

Racine = 1 non

- On explore le fils gauche
- Racine = 2 non
- On explore le fils gauche

? Racine = 4 non

? Pas de fils

- Pas de fils droit
- On explore le fils droit
- Racine = 3 TROUVé

BFS en détail

- On cherche à la racine
- Si l'élément n'y est pas :
- On cherche dans les sommets de profondeur 1
- S'il n'est pas dans à la profondeur 1, on cherche à la profondeur 2
- Etc.

Réflexions

→Plutôt un fonctionnement itératif

On va avoir besoin d'une file FIFO (ou autre structure équivalente) pour stocker les sommets à explorer :

- On ne peut pas " sauter " d'un sommet de profondeur p à un autre
- Il faut se souvenir de la liste des sommets à traiter

Fonctionnement de la file

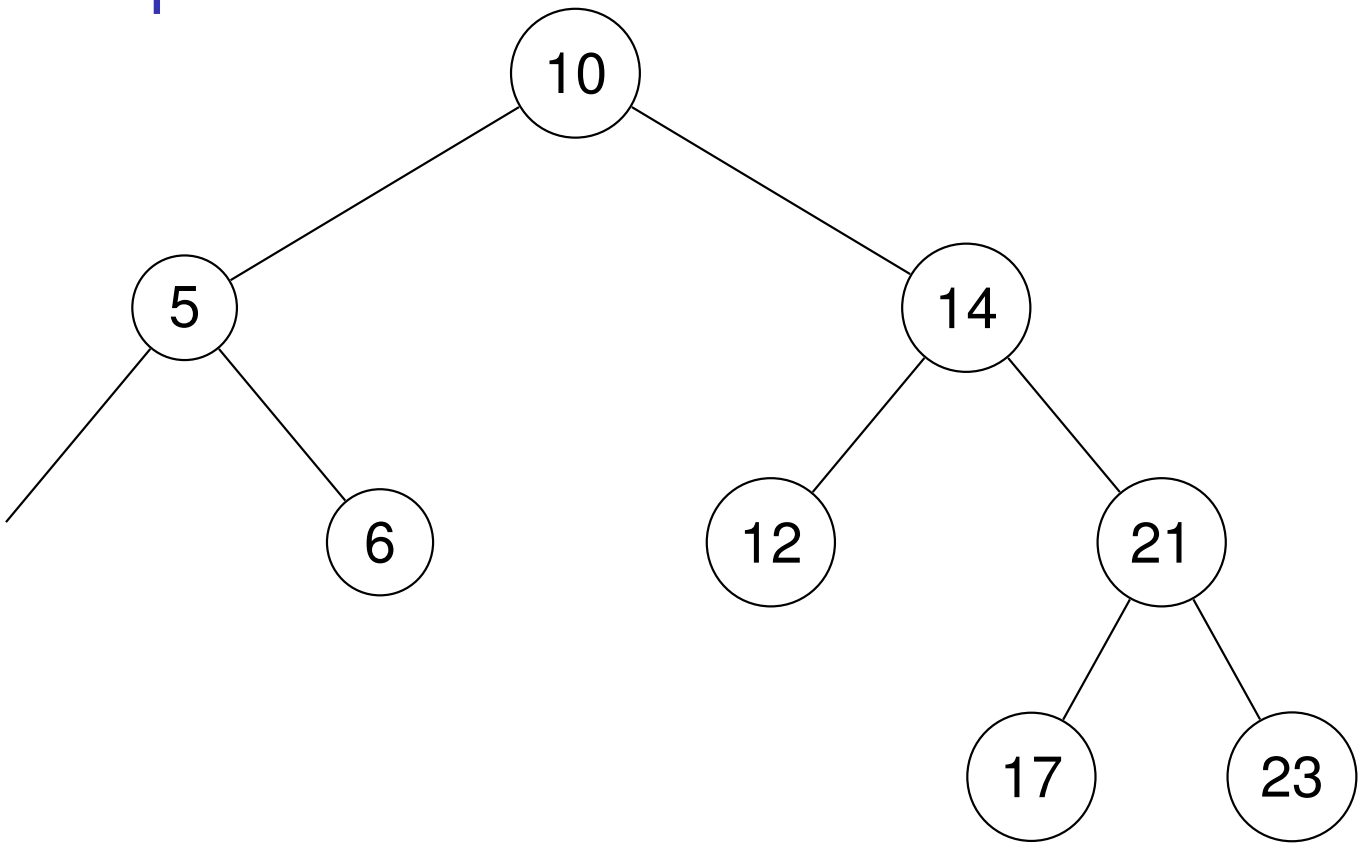
Au début, la file contient l'arbre principal

Si la valeur n'est pas à la racine du premier arbre de la file

- On supprime l'arbre de la file
- On ajoute ses deux sous-arbre en fin de file s'ils ne sont pas vides
- Et on explore l'arbre suivant, qui est le premier de la file

On s'arrête quand on a trouvé ou que la file est vide

Exemples



10
10 5
10 5 6
10 14
10 14 12
10 14 21
10 14 21 17

Parcours en largeur d'un arbre

```
1
2  def cherche(self, val):
3    '''
4    rechercher du premier sommet de valeur val
5    avec DFS
6    '''
7    assert self is not None
8    f = File()
9    f.ajouter(self.__racine)
10   while not f.est_vide():
11     sommet = f.enlever()
12     if sommet.valeur == val:
13       return sommet
14     if sommet.aFG():
15       f.ajouter(sommet.fg())
16     if sommet.aFD():
17       f.ajoute(sommet.fd())
18   return None
```

Avec BFS : on cherche 3

File = [arbre " 1 "]
Valeur de la racine = 1 $\rightarrow$ non

On enlève l'arbre " 1 " et on ajoute ses deux sous-arbres

- File = [sous-arbre gauche de " 1 ", sous-arbre droit de " 1 "]
- On explore le premier : racine = 2 $\rightarrow$ non
- On l'enlève et on ajoute ses sous-arbres
  - File= [sous-arbre droit de " 1 ", sous-arbre gauche de " 2 "]
  - On explore le premier : racine = 3 $\rightarrow$ TROUVé

Avec BFS (2) : on cherche 4

File = [arbre " 1 "]
Valeur de la racine = 1 → non
On enlève l' arbre " 1 " et on ajoute ses deux sous-arbres

- File = [s.-a. gauche de " 1 ", s.-a. droit de " 1 "]
- On explore le premier : racine = 2 → non
- On l'enlève et on ajoute ses deux sous-arbres
  - File = [s.-a. droit de " 1 ", s.-a. gauche de " 2 "]
  - On explore le premier : racine = 3 → non
  - On l'enlève et on ajoute ses deux sous-arbres
    - File = [s.-a. gauche de " 2 ", s.-a. gauche de " 3 ", s.-a. droit de " 3 "]
    - On explore le premier : racine = 4 → TROUVé

Comparatif

Avec DFS, on n'a besoin d'aucun stockage en plus mais on peut s'attarder trop longtemps dans une branche

Avec BFS, on fonctionne par profondeur incrémentale donc on évite le piège mais on a besoin d'un stockage dont la taille augmente à chaque étape

Les ABR

On voit que la recherche dans les arbres binaires est peu aisée

Autre problème : où ajouter un élément ?

Solution : les ABR ou arbres binaires de recherche

À vous

Écrire la fonction pour

1. Compter le nombre de sommets dans un arbre binaire en DFS et BFS

Nombre de sommets d'un arbre

```
1  def nb_sommets_bfs(self) :
2    '''ArbreBinaire -> int
3     retourne le nombre de sommets d'un arbre
4     '''
5    cpt = 0
6    if self.__racine != None:
7      f = File()
8      f.ajoute(self.__racine)
9      while not f.est_vide():
10       sommet = f.supprimer()
11       cpt += 1
12       if sommet.aFG():
13         f.ajouter(sommet.fg())
14       if sommet.aFD():
15         f.ajouter(sommet.fd())
16   return cpt
```

Nombre de sommets d'un arbre

```python
#classe ArbreBinaire
def nb_sommets_dfs(self):
    '''ArbreBinaire -> int
     retourne le nombre de sommets d'un arbre
    '''
    if self.__racine != None:
      return  self.__racine.nb_sommets_dfs()
    return 0
```

```python
#classe Sommet
def nb_sommets_dfs(self):
    '''Sommet -> int
     retourne le nombre de sommets d'un arbre
    '''
    nb1,nb2 = 0,0
    if self.aFG():
      nb1 = self.__fg.nb_sommets_dfs()
    if self.aFD():
      nb2 = self.__fd.nb_sommets_dfs()
    return 1+nb1+nb2
```

Nombre de sommets d'un arbre

```
1  #classe Sommet
2  def nb_sommets_dfs(self):
3      '''Sommet -> int
4       retourne le nombre de sommets d'un arbre
5      '''
6      if self.aFG() and self.aFD():
7        return 1 +  self.__fg.nb_sommets_dfs() + self.__fd.
8      elif self.aFG():
9        return 1 +  self.__fg.nb_sommets_dfs()
10     elif self.aFD():
11       return 1 +  self.__fd.nb_sommets_dfs()
12     else:
13       return 1
```

Nombre de sommets d'un arbre

```
1   def nb_sommets2(self, acc=0):
2     '''ArbreBinaire -> int
3        retourne le nombre de sommets d'un arbre'''
4       if self.aFG():
5         acc = self.__fg.nb_sommets2(acc)
6       if self.aFD():
7         acc = self.__fd.nb_sommets2(acc)
8       return 1+acc
```

À vous

Écrire des fonctions pour

1. calculer la plus grande valeur dans un arbre binaire