Arbres

Jérémie Cabessa

Laboratoire DAVID, UVSQ

Introduction

- Les **arbres** sont des structures de données très utiles en informatique.
- Ils permettent de gérer des données arborescentes: structures de fichiers, opérations arithmétiques, etc.
- Ils permettent des algorithmes de recherche accélérés.

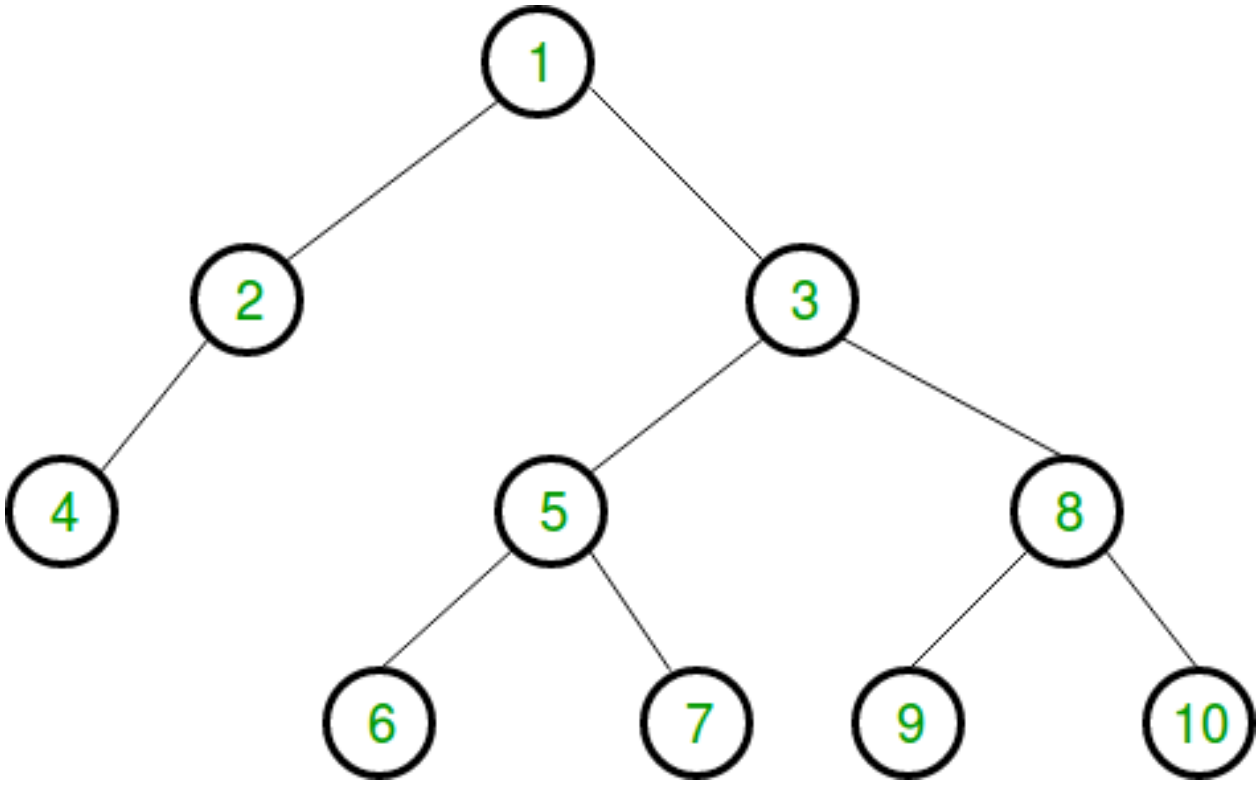
Introduction

- Les **arbres** sont des structures de données très utiles en informatique.
- Ils permettent de gérer des données arborescentes: structures de fichiers, opérations arithmétiques, etc.
- Ils permettent des algorithmes de recherche accélérés.

Introduction

- Les **arbres** sont des structures de données très utiles en informatique.
- Ils permettent de gérer des données arborescentes: structures de fichiers, opérations arithmétiques, etc.
- Ils permettent des algorithmes de recherche accélérés.

Arbre



Arbre

- Un **arbre** est la généralisation d'une liste chaînée.
- Un arbre peut être vu comme un objet *noeud* qui pointe vers plusieurs objets, eux-mêmes de la classe *noeud*.
- Si le noeud A pointe vers B, one dit que A est un *parent* de B, ou que B est un *fils* de A.

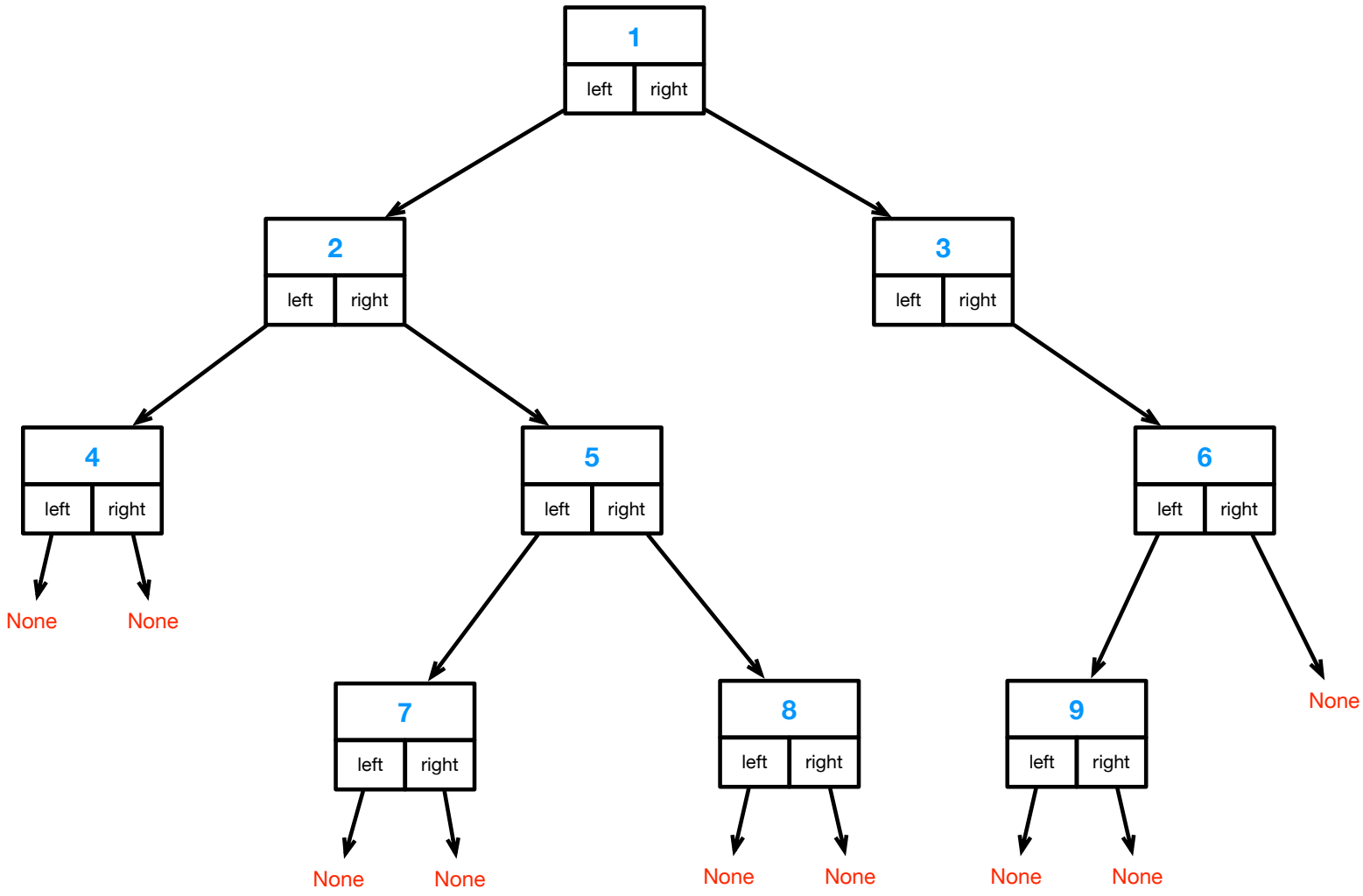
Arbre

- Un **arbre** est la généralisation d'une liste chaînée.
- Un arbre peut être vu comme un objet *noeud* qui pointe vers plusieurs objets, eux-mêmes de la classe *noeud*.
- Si le noeud A pointe vers B, one dit que A est un *parent* de B, ou que B est un *fils* de A.

Arbre

- Un **arbre** est la généralisation d'une liste chaînée.
- Un arbre peut être vu comme un objet *noeud* qui pointe vers plusieurs objets, eux-mêmes de la classe *noeud*.
- Si le noeud A pointe vers B, one dit que A est un *parent* de B, ou que B est un *fils* de A.

Arbre


1
left
right
2
left
right
3
left
right
4
left
right
5
left
right
6
left
right
None
None
7
left
right
8
left
right
9
left
right
None
None
None
None
None
None
None

Arbres binaires

```
class ArbreBinaire:

    def init(self, valeur, fg=None, fd=None):

        self.valeur = valeur
        self.fg = fg
        self.fd = fd
```

Porprétés

- De par leurs structures récursives, plusieurs propriétés des arbres binaires peuvent être établies facilement par des algorithmes récursifs:
  - déterminer le nombre de noeuds de l'arbre
  - déterminer la hauteur de l'arbre
  - rechercher la valeur maximale dans un arbre
  - etc.

Porprétés

- De par leurs structures récursives, plusieurs propriétés des arbres binaires peuvent être établies facilement par des algorithmes récursifs:
  - déterminer le nombre de noeuds de l'arbre
  - déterminer la hauteur de l'arbre
  - rechercher la valeur maximale dans un arbre
  - etc.

Porprétés

- De par leurs structures récursives, plusieurs propriétés des arbres binaires peuvent être établies facilement par des algorithmes récursifs:
  - déterminer le nombre de noeuds de l'arbre
  - déterminer la hauteur de l'arbre
  - rechercher la valeur maximale dans un arbre
  - etc.

Porprétés

- De par leurs structures récursives, plusieurs propriétés des arbres binaires peuvent être établies facilement par des algorithmes récursifs:
  - déterminer le nombre de noeuds de l'arbre
  - déterminer la hauteur de l'arbre
  - rechercher la valeur maximale dans un arbre
  - etc.

Porprétés

- De par leurs structures récursives, plusieurs propriétés des arbres binaires peuvent être établies facilement par des algorithmes récursifs:
  - déterminer le nombre de noeuds de l'arbre
  - déterminer la hauteur de l'arbre
  - rechercher la valeur maximale dans un arbre
  - etc.

Nombre de noeuds

```
def nbNoeuds(arbre):
    if arbre is None:
        return 0
    else:
        return 1 + nbNoeuds(arbre.fg)
                 + nbNoeuds(arbre.fd)
```

Hauteur

```
def hauteur(arbre):
    if arbre is None:
        return 0
    else:
        return 1 + max(hauteur(arbre.fg),
                       hauteur(arbre.fd))
```

Maximum

```
def rechercherMax(arbre):
    if arbre is None:
        return 0
    else:
        max(arbre.valeur, rechercherMax(arbre.fg),
                          rechercherMax(arbre.fd))
```

Porprétés

Lemma

Un arbre binaire de hauteur h possède au plus 2^h feuilles et $2^{h+1} - 1$ noeuds.

Preuve: Par récurrence sur h.

1. On montre que le lemme est vrai pour $h = 0$.
2. Soit $n \geq 0$. On montre que si le lemme est vrai pour $h = n$ (hypothèse de récurrence), alors il est vrai pour $h = n + 1$.

On en conclut que le lemme est vrai pour tout $h \geq 0$.

Parcours

- Il y a plusieurs manières de parcourir des arbres binaires.
  - Le parcours préfixe.
  - Le parcours suffixe.
  - Le parcours infixe.
  - Le parcours en largeur.

Parcours

- Il y a plusieurs manières de parcourir des arbres binaires.
- Le parcours préfixe.
- Le parcours suffixe.
- Le parcours infixe.
- Le parcours en largeur.

Parcours

- Il y a plusieurs manières de parcourir des arbres binaires.
- Le parcours préfixe.
- Le parcours suffixe.
- Le parcours infixe.
- Le parcours en largeur.

Parcours

- Il y a plusieurs manières de parcourir des arbres binaires.
- Le parcours préfixe.
- Le parcours suffixe.
- Le parcours infixe.
- Le parcours en largeur.

Parcours

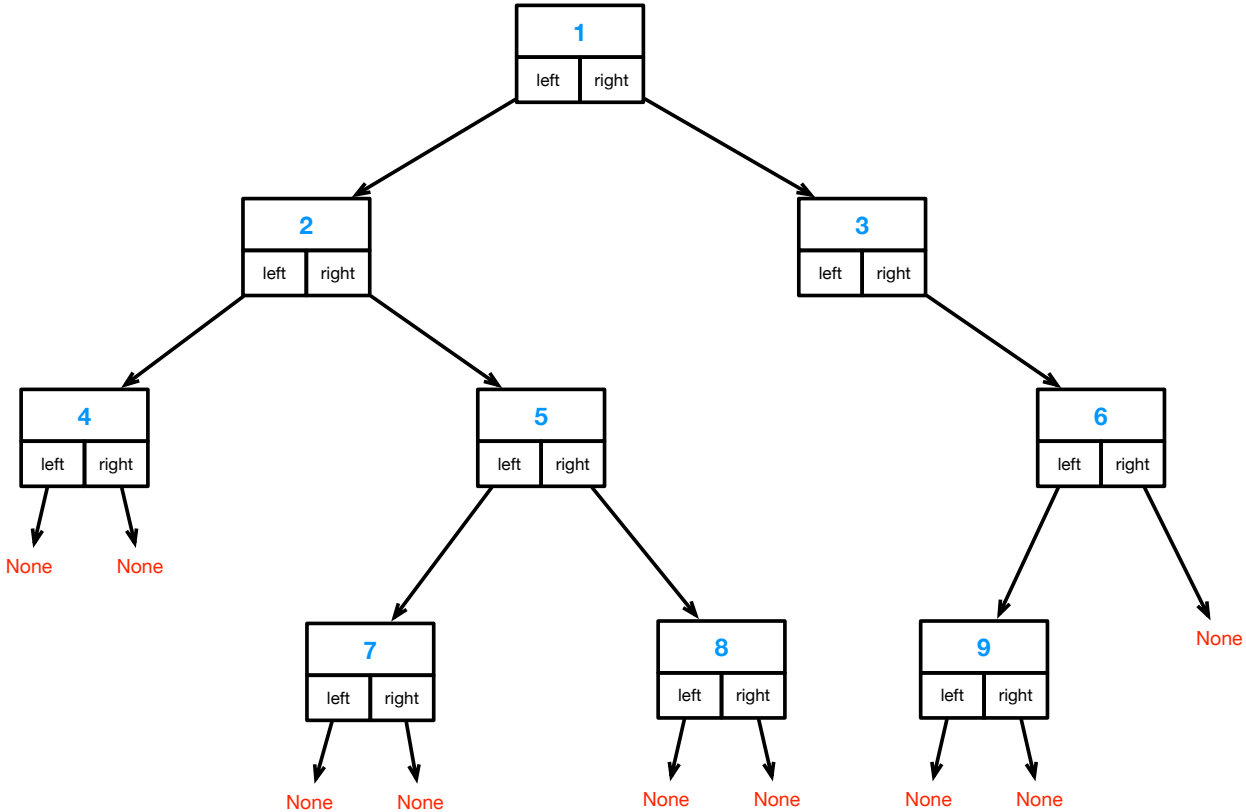
- Il y a plusieurs manières de parcourir des arbres binaires.
- Le parcours préfixe.
- Le parcours suffixe.
- Le parcours infixe.
- Le parcours en largeur.

Parcours préfixe

- Chaque noeud est visité, puis son fils gauche, puis son fils droit, le tout récursivement.

```
def parcoursPrefixe(arbre):
    if arbre is not None:
        print(arbre.valeur, end =  )
        parcoursPrefixe(arbre.fg)
        parcoursPrefixe(arbre.fd)
```

Parcours préfixe



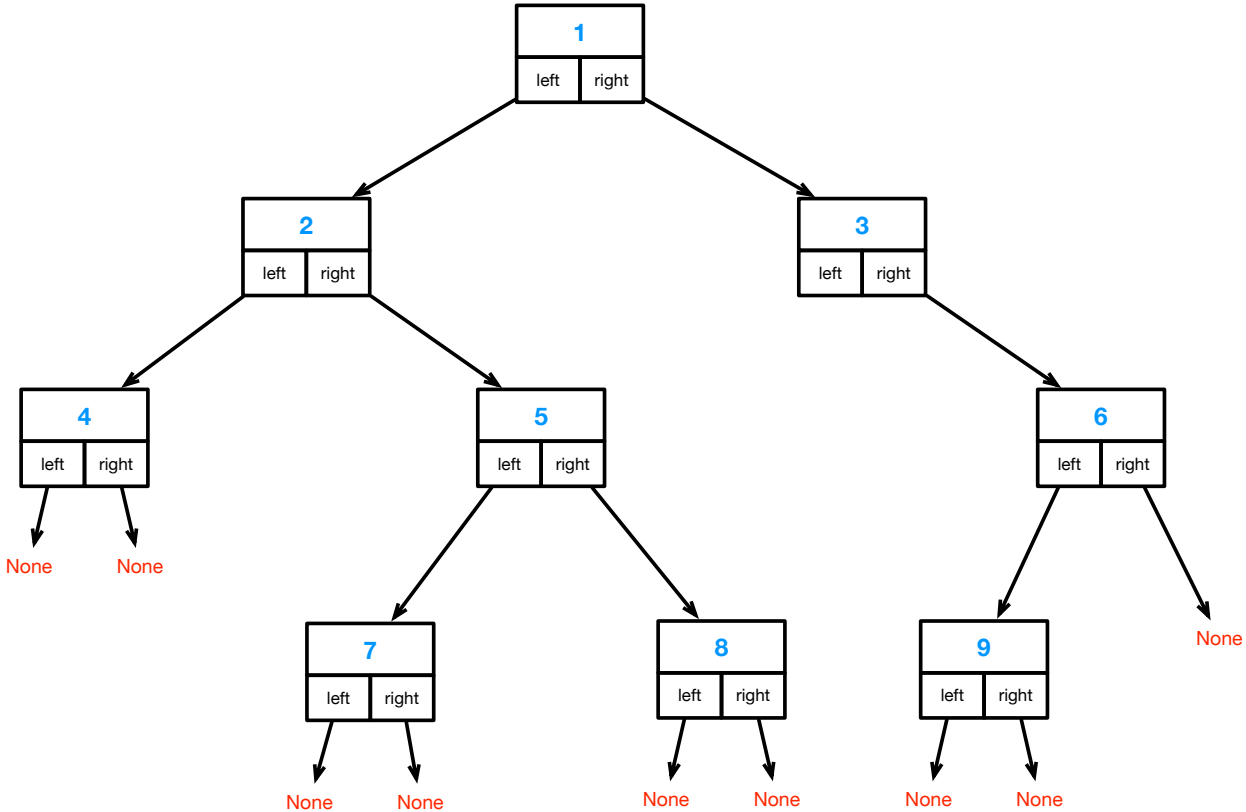
$$[1, 2, 4, 5, 7, 8, 3, 6, 9]$$

Parcours suffixe

- Pour chaque noeud, on visite son fils gauche, puis son fils droit, puis lui-même, le tout récursivement.

```
def parcoursSuffixe(arbre):
    if arbre is not None:
        parcoursSuffixe(arbre.fg)
        parcoursSuffixe(arbre.fd)
        print(arbre.valeur, end =  )
```

Parcours suffixe



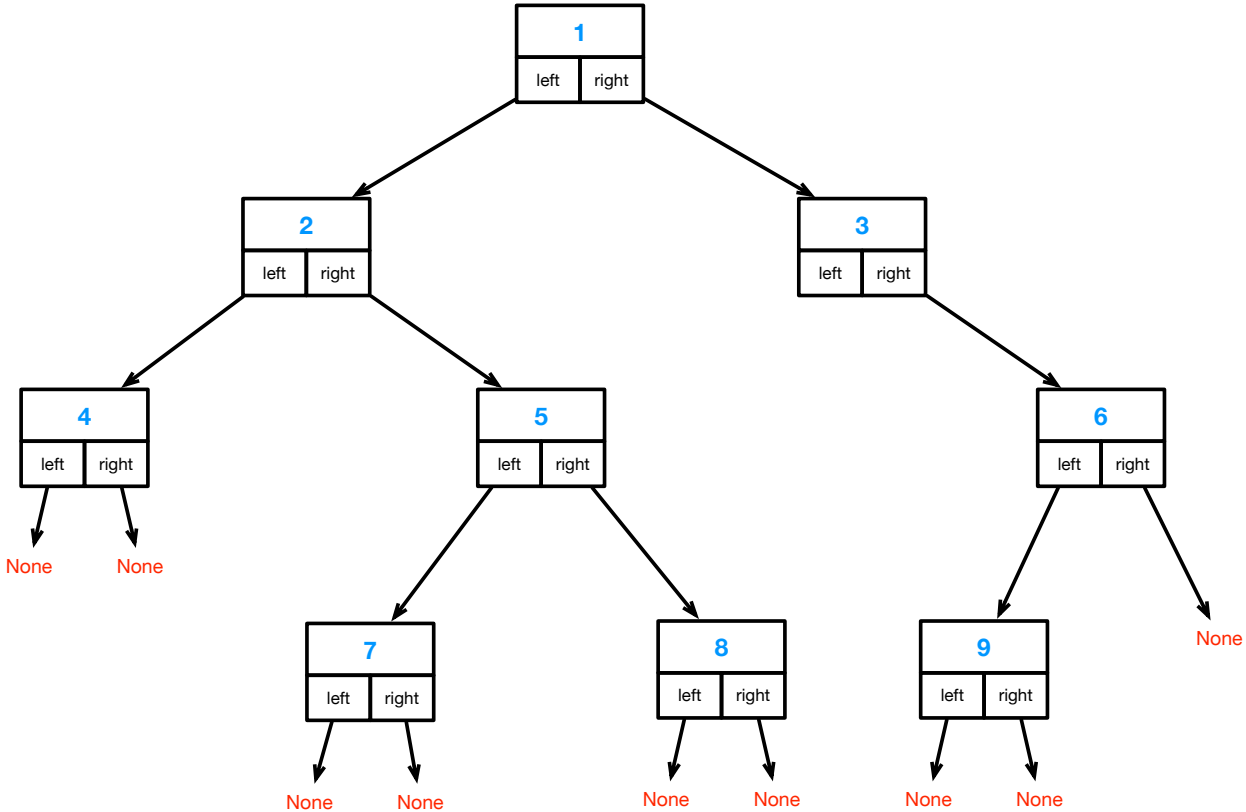
$$[4, 7, 8, 5, 2, 9, 6, 3, 1]$$

Parcours infixe

- Pour chaque noeud, on visite son fils gauche, puis lui-même, puis son fils droit, le tout récursivement.

```
def parcoursInfixe(arbre):
    if arbre is not None:
        parcoursInfixe(arbre.fg)
        print(arbre.valeur, end =  )
        parcoursInfixe(arbre.fd)
```

Parcours infixe



$$[4, 2, 7, 5, 8, 1, 3, 9, 6]$$

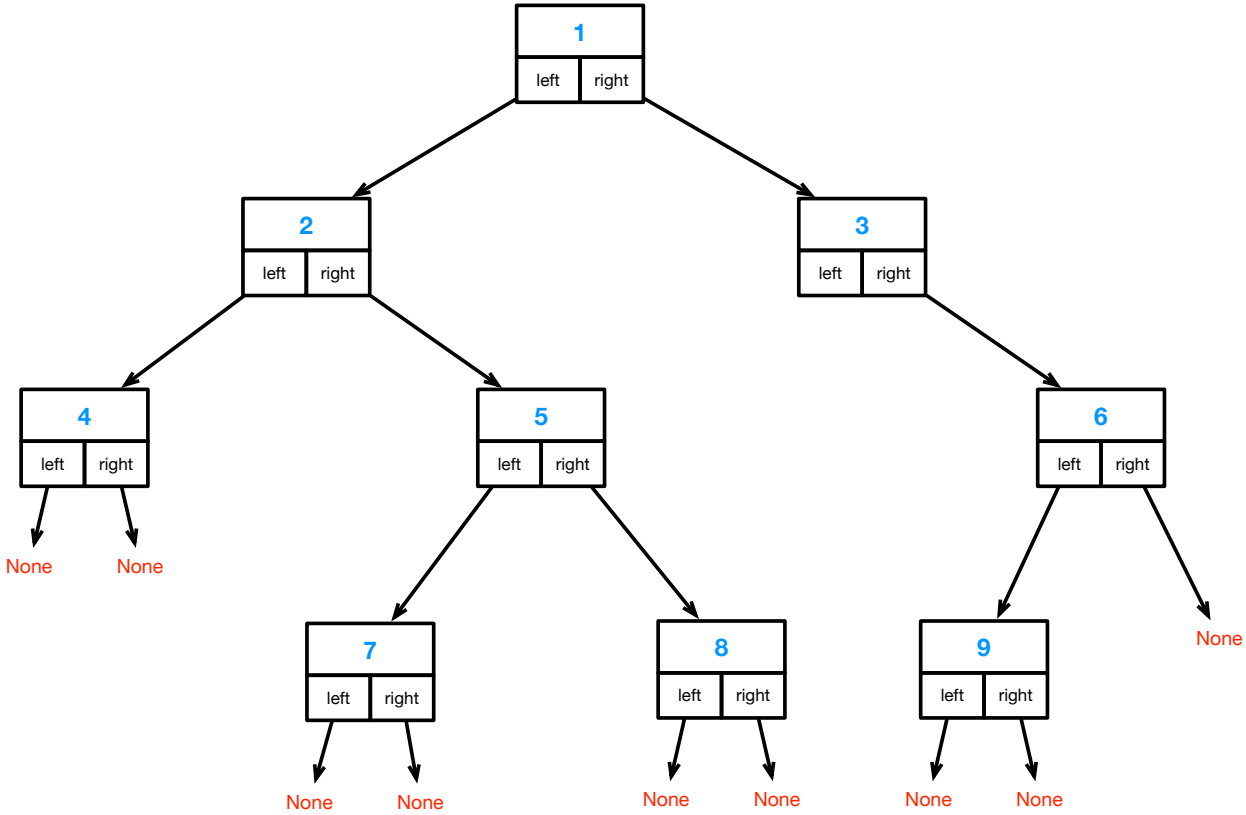
Parcours en largeur

- Algorithme non-récursif: on utilise une file (FIFO) qui, initialement, contient l'arbre à visiter. À chaque étape, on visite le premier élément de la file (pop), avant d'y ajouter ses fils gauche et droit.

```
def parcoursLargeur(arbre):
    file = [arbre]
    while file != []:
        courant = file.pop(0)    FIFO
        print(courant.valeur, end =  )

        if courant.fg is not None:
            file.append(courant.fg)
        if courant.fd is not None:
            file.append(courant.fd)
```

Parcours en largeur



$$[1, 2, 3, 4, 5, 6, 7, 8, 9]$$

Arbres binaires de recherche (ABR)

- Un **arbre binaire de rercherche (ABR)** est une structure de donnée utilisée pour représenter un *ensemble dynamique*, i.e., de taille variable.
- Less ABR possèdent une bonne complexité en moyenne pour les opérations `prédécesseur`, `successeur`, `rechercher`, `minimum`, `maximum`, `insérer` et `supprimer`.
- Un ABR peut donc servir aussi bien de dictionnaire que de file de priorités.

Arbres binaires de recherche (ABR)

- Un **arbre binaire de rercherche (ABR)** est une structure de donnée utilisée pour représenter un *ensemble dynamique*, i.e., de taille variable.
- Less ABR possèdent une bonne complexité en moyenne pour les opérations `prédécesseur`, `successeur`, `rechercher`, `minimum`, `maximum`, `insérer` et `supprimer`.
- Un ABR peut donc servir aussi bien de dictionnaire que de file de priorités.

Arbres binaires de recherche (ABR)

- Un **arbre binaire de rercherche (ABR)** est une structure de donnée utilisée pour représenter un *ensemble dynamique*, i.e., de taille variable.
- Less ABR possèdent une bonne complexité en moyenne pour les opérations `prédécesseur`, `successeur`, `rechercher`, `minimum`, `maximum`, `insérer` et `supprimer`.
- Un ABR peut donc servir aussi bien de dictionnaire que de file de priorités.

Arbres binaires de recherche (ABR)

- Formellement, un **arbre binaire de recherche (ABR)** est un arbre binaire qui satisfait la *propriété d'arbre binaire de recherche* suivante:
  - si y est un noeud du sous-arbre gauche de x alors $valeur(y) \leq valeur(x)$.
  - si y est un noeud du sous-arbre droit de x alors $valeur(y) >= valeur(x)$.
- **Remarque:** attention, on parle du sous-arbre entier et pas seulement des fils gauche et droit.

Arbres binaires de recherche (ABR)

- Formellement, un **arbre binaire de recherche (ABR)** est un arbre binaire qui satisfait la *propriété d'arbre binaire de recherche* suivante:
- si y est un noeud du sous-arbre gauche de x alors $valeur(y) \leq valeur(x)$.
- si y est un noeud du sous-arbre droit de x alors $valeur(y) >= valeur(x)$.
- **Remarque:** attention, on parle du sous-arbre entier et pas seulement des fils gauche et droit.

Arbres binaires de recherche (ABR)

- Formellement, un **arbre binaire de recherche (ABR)** est un arbre binaire qui satisfait la *propriété d'arbre binaire de recherche* suivante:

- si y est un noeud du sous-arbre gauche de x alors $valeur(y) \leq valeur(x)$.
- si y est un noeud du sous-arbre droit de x alors $valeur(y) >= valeur(x)$.

- **Remarque:** attention, on parle du sous-arbre entier et pas seulement des fils gauche et droit.

Arbres binaires de recherche (ABR)

- Formellement, un **arbre binaire de recherche (ABR)** est un arbre binaire qui satisfait la *propriété d'arbre binaire de recherche* suivante:

  - si y est un noeud du sous-arbre gauche de x alors $valeur(y) \leq valeur(x)$.
  - si y est un noeud du sous-arbre droit de x alors $valeur(y) >= valeur(x)$.

- **Remarque:** attention, on parle du sous-arbre entier et pas seulement des fils gauche et droit.

Arbres binaires de recherche (ABR)

- Suivant l'ordre dans lequel les clés ont été insérées, on peut obtenir différents arbres binaires...

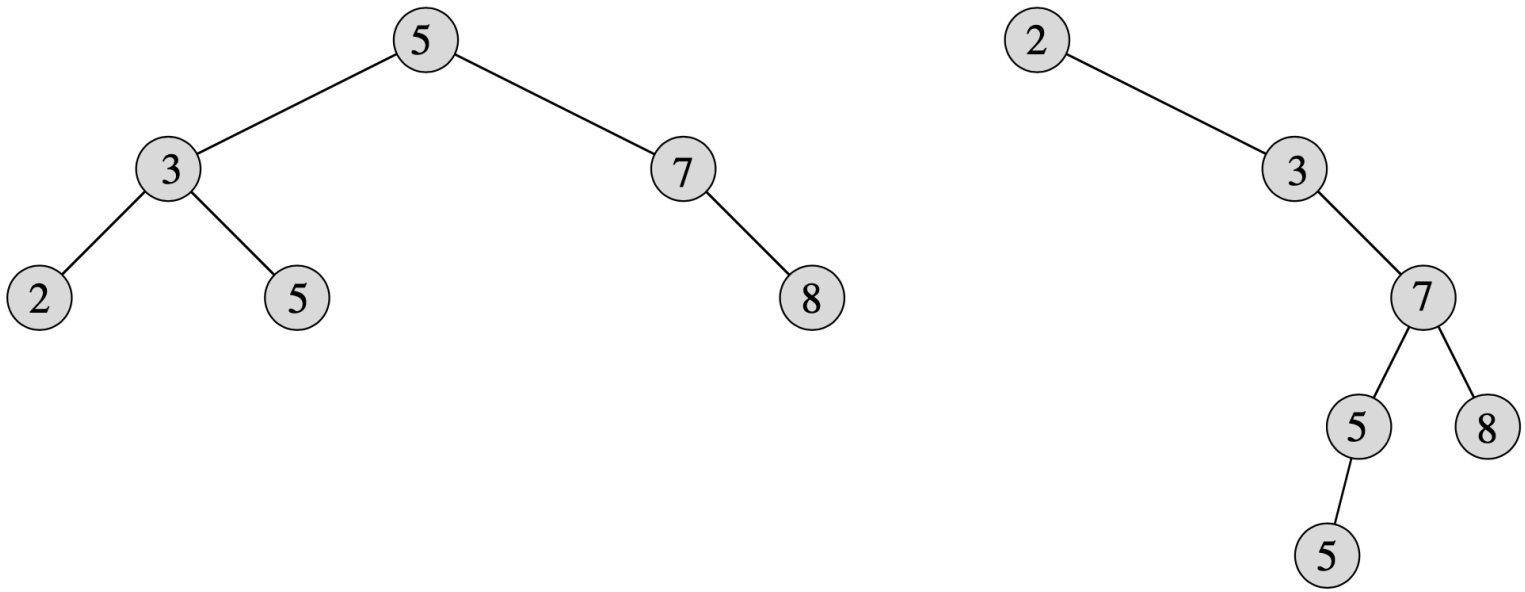


Figure: Deux ABR qui représentent le même jeu de données

Arbres binaires de recherche (ABR)

- Dans un ABR, la complexité d'une opération dépend généralement de la profondeur de l'arbre.
- Ainsi, plus l'arbre est compact, plus il est efficace; plus l'arbre est dégénéré, moins il est efficace.

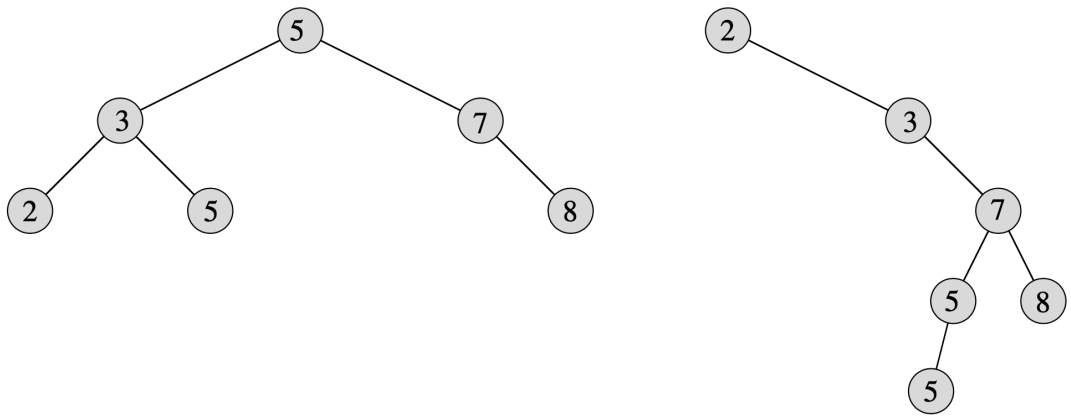


Figure: Deux ABR qui représentent le même jeu de données

Arbres binaires de recherche (ABR)

- Dans un ABR, la complexité d'une opération dépend généralement de la profondeur de l'arbre.
- Ainsi, plus l'arbre est compact, plus il est efficace; plus l'arbre est dégénéré, moins il est efficace.

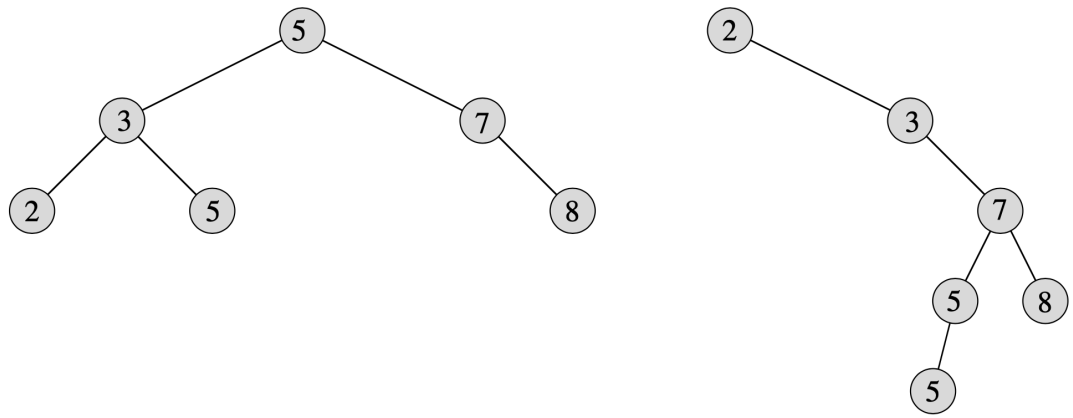


Figure: Deux ABR qui représentent le même jeu de données

Pacours infixe

- Le *parcours infixe* d'un ABR imprimera ses clés dans l'ordre croissant.

```
def parcoursInfixe(arbre):
    if arbre is not None:
        parcoursInfixe(arbre.fg)
        print(arbre.valeur, end =  )
        parcoursInfixe(arbre.fd)
```

- Ceci découle de la propriété d'arbre binaire de recherche.

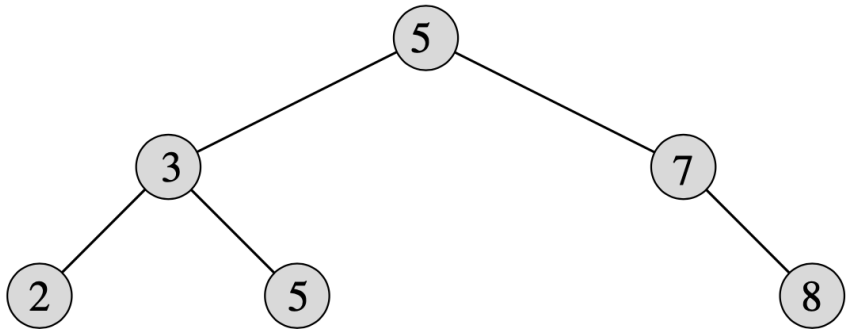
Pacours infixe

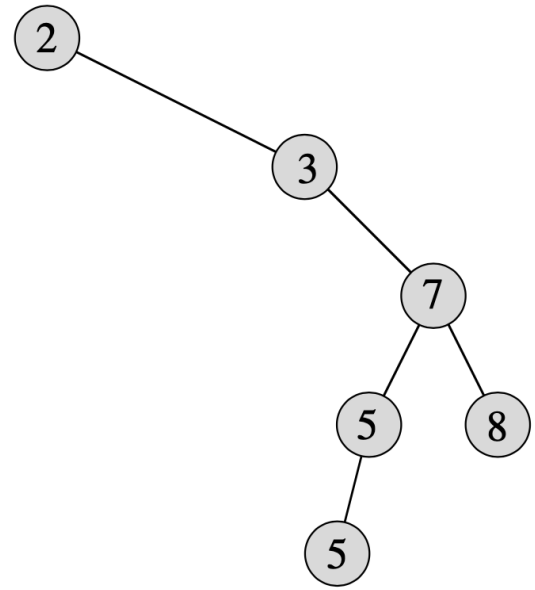
- Le *parcours infixe* d'un ABR imprimera ses clés dans l'ordre croissant.

```
def parcoursInfixe(arbre):
    if arbre is not None:
        parcoursInfixe(arbre.fg)
        print(arbre.valeur, end =  )
        parcoursInfixe(arbre.fd)
```

- Ceci découle de la propriété d'arbre binaire de recherche.

Pacours infixe





- Parcours infixe de ces arbres: $2, 3, 5, 5, 7, 8$

Pacours infixe

Proposition

Si x est la racine d'un sous-arbre à n nœuds, alors la procédure parcours_infixe(x) *prend un temps* $\Theta(n)$.

Preuve: On montre par récurrence que $T(n) = (c+d)n + c$, pour tout $n \in \mathbb{N}$.

Insérer un élément

- Insérer une clé dans un ABR:
  - On va insérer une nouvelle feuille dans l'arbre.
  - On descend dans l'arbre et on place la clé sur un fils gauche ou fils droit vide, là où elle devrait se trouver.

Insérer un élément

- Insérer une clé dans un ABR:
- On va insérer une nouvelle feuille dans l'arbre.
- On descend dans l'arbre et on place la clé sur un fils gauche ou fils droit vide, là où elle devrait se trouver.

Insérer un élément

- Insérer une clé dans un ABR:
- On va insérer une nouvelle feuille dans l'arbre.
- On descend dans l'arbre et on place la clé sur un fils gauche ou fils droit vide, là où elle devrait se trouver.

Insérer un élément

```
def insertion(abr, val):

    if abr is None:
        return ABR(val)

    elif val < abr.valeur:
        abr.fg = insertion(abr.fg, val)

    else:
        abr.fd = insertion(abr.fd, val)

    return abr
```

Rechercher un élément

- Rechercher une clé dans un ABR:
  - On descend dans l'arbre, en suivant les fils gauches ou les fils droits, en fonction de si la valeur à rechercher est plus petite ou plus grande que celle du fils en question, respectivement.

Rechercher un élément

- Rechercher une clé dans un ABR:
- On descend dans l'arbre, en suivant les fils gauches ou les fils droits, en fonction de si la valeur à rechercher est plus petite ou plus grande que celle du fils en question, respectivement.

Rechercher un élément

```
def rechercher(abr, val):

    if abr is None:
        return False

    elif abr.valeur == val:
        return True

    else:
        if val < abr.valeur:
            return rechercher(abr.fg, val)
        else:
            return rechercher(abr.fd, val)
```

Recherche du minimum

- Rechercher la clé minimale dans un ABR:
  - On descend dans la feuille la plus à gauche.

```
def minimum(abr):
    if abr is None:
        return None
    while abr.fg is not None:
        abr = abr.fg
    return abr.valeur
```

Recherche du minimum

- Rechercher la clé minimale dans un ABR:
- On descend dans la feuille la plus à gauche.

```
def minimum(abr):

    if abr is None:
        return None

    while abr.fg is not None:
        abr = abr.fg

    return abr.valeur
```

Recherche du minimum

- Rechercher la clé minimale dans un ABR:
- On descend dans la feuille la plus à gauche.

```
def minimum(abr):

    if abr is None:
        return None

    while abr.fg is not None:
        abr = abr.fg

    return abr.valeur
```

Recherche du minimum

- Rechercher la clé minimale dans un ABR:
- On descend dans la feuille la plus à gauche.

```
def minimum(abr):

    if abr is None:
        return None

    while abr.fg is not None:
        abr = abr.fg

    return abr.valeur
```

Recherche du maximum

- Rechercher la clé maximale dans un ABR:
  - On descend dans la feuille la plus à droite.

```
def maximum(abr):
    if abr is None:
        return None
    while abr.fd is not None:
        abr = abr.fd
    return abr.valeur
```

Recherche du maximum

- Rechercher la clé maximale dans un ABR:
- On descend dans la feuille la plus à droite.

```
def maximum(abr):

    if abr is None:
        return None

    while abr.fd is not None:
        abr = abr.fd

    return abr.valeur
```

Recherche du maximum

- Rechercher la clé maximale dans un ABR:
- On descend dans la feuille la plus à droite.

```
def maximum(abr):

    if abr is None:
        return None

    while abr.fd is not None:
        abr = abr.fd

    return abr.valeur
```

Recherche du maximum

- Rechercher la clé maximale dans un ABR:
- On descend dans la feuille la plus à droite.

```
def maximum(abr):

    if abr is None:
        return None

    while abr.fd is not None:
        abr = abr.fd

    return abr.valeur
```

Recherche du successeur

- Rechercher le successeur d'une valeur `val` (s'il existe) dans un ABR:
  - Une implémentation classique de cette fonction requiert un attribut "parent" dans la class ABR...
  - On remédie à ce problème en ajoutant un paramètre `succ` qui garde en mémoire le parent courant du fils qu'on visite.
  - **Idée générale:** `succ(ABR, n)` est la feuille la plus à gauche du fils droit de n.

Recherche du successeur

- Rechercher le successeur d'une valeur `val` (s'il existe) dans un ABR:
- Une implémentation classique de cette fonction requiert un attribut "parent" dans la class ABR...
- On remédie à ce problème en ajoutant un paramètre `succ` qui garde en mémoire le parent courant du fils qu'on visite.
- **Idée générale:** `succ(ABR, n)` est la feuille la plus à gauche du fils droit de n.

Recherche du successeur

- Rechercher le successeur d'une valeur `val` (s'il existe) dans un ABR:
- Une implémentation classique de cette fonction requiert un attribut "parent" dans la class ABR...
- On remédie à ce problème en ajoutant un paramètre `succ` qui garde en mémoire le parent courant du fils qu'on visite.
- **Idée générale:** `succ(ABR, n)` est la feuille la plus à gauche du fils droit de n.

Recherche du successeur

- Rechercher le successeur d'une valeur `val` (s'il existe) dans un ABR:
- Une implémentation classique de cette fonction requiert un attribut "parent" dans la class ABR...
- On remédie à ce problème en ajoutant un paramètre `succ` qui garde en mémoire le parent courant du fils qu'on visite.
- **Idée générale:** `succ(ABR, n)` est la feuille la plus à gauche du fils droit de n.

Recherche du successeur

1. Si le fils visité est `None`, on retourne le parent courant.
2. Si un noeud `abr` de valeur `val` existe, et si `abr` possède un fils droit, alors le successeur est le min de ce fils droit.
3. Si `val < abr.valeur`, on cherche récursivement dans le fils gauche de `abr` tout en gardant en mémoire la racine de `abr` (le parent courant).
4. Si `val > abr.valeur`, on cherche récursivement dans le fils droit de `abr`

Recherche du successeur

1. Si le fils visité est `None`, on retourne le parent courant.
2. Si un noeud `abr` de valeur `val` existe, et si `abr` possède un fils droit, alors le successeur est le min de ce fils droit.
3. Si `val < abr.valeur`, on cherche récursivement dans le fils gauche de `abr` tout en gardant en mémoire la racine de `abr` (le parent courant).
4. Si `val > abr.valeur`, on cherche récursivement dans le fils droit de `abr`

Recherche du successeur

1. Si le fils visité est `None`, on retourne le parent courant.
2. Si un noeud `abr` de valeur `val` existe, et si `abr` possède un fils droit, alors le successeur est le min de ce fils droit.
3. Si `val < abr.valeur`, on cherche récursivement dans le fils gauche de `abr` tout en gardant en mémoire la racine de `abr` (le parent courant).
4. Si `val > abr.valeur`, on cherche récursivement dans le fils droit de `abr`

Recherche du successeur

1. Si le fils visité est `None`, on retourne le parent courant.
2. Si un noeud `abr` de valeur `val` existe, et si `abr` possède un fils droit, alors le successeur est le min de ce fils droit.
3. Si `val < abr.valeur`, on cherche récursivement dans le fils gauche de `abr` tout en gardant en mémoire la racine de `abr` (le parent courant).
4. Si `val > abr.valeur`, on cherche récursivement dans le fils droit de `abr`

Recherche du successeur

```
def successeur(abr, val, succ=None):

    if abr is None:
        try:
            return succ.valeur
        except:
            return None

    if abr.valeur == val and abr.fd:
        return minimum(abr.fd)

    elif val < abr.valeur:
        succ = abr
        return successeur(abr.fg, val, succ)

    else:
        return successeur(abr.fd, val, succ)
```

Recherche du prédécesseur

- Rechercher le prédécesseur d'une valeur `val` (s'il existe) dans un ABR:
  - Une implémentation classique de cette fonction requiert un attribut "parent" dans la class ABR...
  - On remédie à ce problème en ajoutant un paramètre `pred` qui garde en mémoire le parent courant du fils qu'on visite.
  - **Idée générale:** `pred(ABR, n)` est la feuille la plus à droite du fils gauche de n.

Recherche du prédécesseur

- Rechercher le prédécesseur d'une valeur `val` (s'il existe) dans un ABR:
- Une implémentation classique de cette fonction requiert un attribut "parent" dans la class ABR...
- On remédie à ce problème en ajoutant un paramètre `pred` qui garde en mémoire le parent courant du fils qu'on visite.
- **Idée générale:** `pred(ABR, n)` est la feuille la plus à droite du fils gauche de n.

Recherche du prédécesseur

- Rechercher le prédécesseur d'une valeur `val` (s'il existe) dans un ABR:
- Une implémentation classique de cette fonction requiert un attribut "parent" dans la class ABR...
- On remédie à ce problème en ajoutant un paramètre `pred` qui garde en mémoire le parent courant du fils qu'on visite.
- **Idée générale:** `pred(ABR, n)` est la feuille la plus à droite du fils gauche de n.

Recherche du prédécesseur

- Rechercher le prédécesseur d'une valeur `val` (s'il existe) dans un ABR:
- Une implémentation classique de cette fonction requiert un attribut "parent" dans la class ABR...
- On remédie à ce problème en ajoutant un paramètre `pred` qui garde en mémoire le parent courant du fils qu'on visite.
- **Idée générale:** `pred(ABR, n)` est la feuille la plus à droite du fils gauche de n.

Recherche du prédécesseur

1. Si le fils visité est `None`, on retourne le parent courant.
2. Si un noeud `abr` de valeur `val` existe, et si `abr` possède un fils gauche, alors le prédécesseur est le max de ce fils gauche.
3. Si `val > abr.valeur`, on cherche récursivement dans le fils droit de `abr` tout en gardant en mémoire la racine de `abr` (le parent courant).
4. Si `val < abr.valeur`, on cherche récursivement dans le fils gauche de `abr`

Recherche du prédécesseur

1. Si le fils visité est `None`, on retourne le parent courant.
2. Si un noeud `abr` de valeur `val` existe, et si `abr` possède un fils gauche, alors le prédécesseur est le max de ce fils gauche.
3. Si `val > abr.valeur`, on cherche récursivement dans le fils droit de `abr` tout en gardant en mémoire la racine de `abr` (le parent courant).
4. Si `val < abr.valeur`, on cherche récursivement dans le fils gauche de `abr`

Recherche du prédécesseur

1. Si le fils visité est `None`, on retourne le parent courant.
2. Si un noeud `abr` de valeur `val` existe, et si `abr` possède un fils gauche, alors le prédécesseur est le max de ce fils gauche.
3. Si `val > abr.valeur`, on cherche récursivement dans le fils droit de `abr` tout en gardant en mémoire la racine de `abr` (le parent courant).
4. Si `val < abr.valeur`, on cherche récursivement dans le fils gauche de `abr`

Recherche du prédécesseur

1. Si le fils visité est `None`, on retourne le parent courant.
2. Si un noeud `abr` de valeur `val` existe, et si `abr` possède un fils gauche, alors le prédécesseur est le max de ce fils gauche.
3. Si `val > abr.valeur`, on cherche récursivement dans le fils droit de `abr` tout en gardant en mémoire la racine de `abr` (le parent courant).
4. Si `val < abr.valeur`, on cherche récursivement dans le fils gauche de `abr`

Recherche du prédécesseur

```
def predecesseur(abr, val, pred=None):

    if abr is None:
        try:
            return pred.valeur
        except:
            return None

    if abr.valeur == val and abr.fg:
        return maximum(abr.fg)

    elif val > abr.valeur:
        pred = abr
        return predecesseur(abr.fd, val, pred)

    else:
        return predecesseur(abr.fg, val, pred)
```

Pacours infixe

Proposition

Les procédures rechercher, minumum, maximum, successeur, prédécesseur, insérer *et* supprimer *sont en* $O(h) = O(\log n)$, *où* h *est la hauteur de l'arbre et* n *son nombre de noeuds.*