Arbre binaire de recherche

Christophe Viroulaud

Terminale - NSI

Algo 10

Les arbres binaires, les tas imposent des contraintes aux structures arborescentes. Il en résulte des objets avec des propriétés très utiles. Par exemple, la complexité du tri par tas est $O(n) = n.log(n)$.

Comment obtenir une méthode de recherche efficace avec les arbres ?

Sommaire

Arbre binaire de recherche - définition

On impose une contrainte à chaque nœud d'un arbre binaire :

- les valeurs du sous-arbre gauche sont plus petites que celle du nœud,
- les valeurs du sous-arbre droit sont plus grandes que celle du nœud.

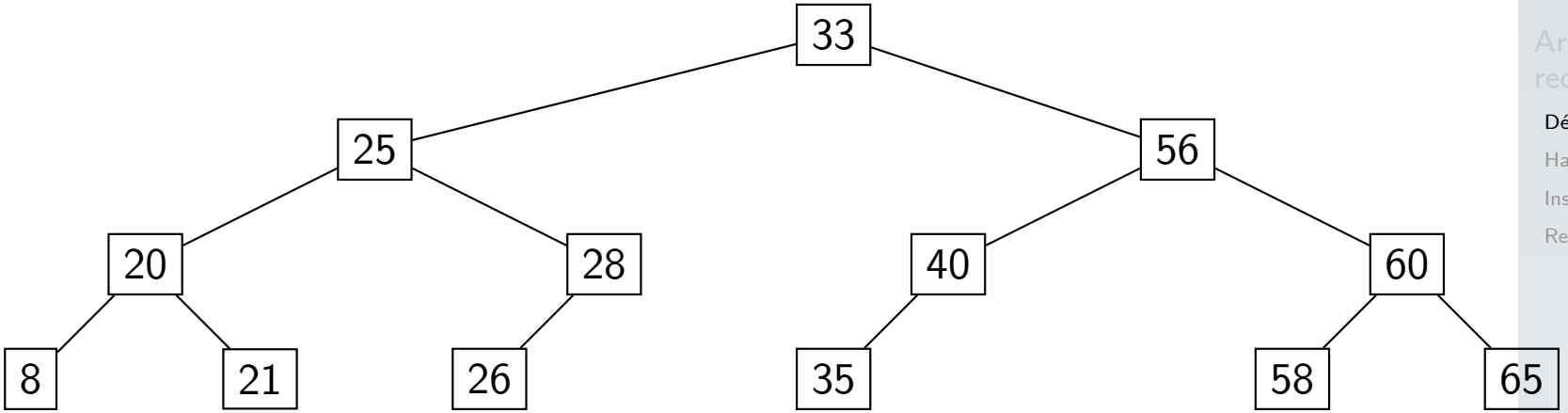


FIGURE 1 – Un Arbre Binaire de Recherche (*ABR*)

Remarque

On suppose que chaque valeur n'apparaît qu'une seule fois dans l'arbre.

Activité 1 :

1. Placer les valeurs 23, 27, 54, 55 dans l'ABR.
2. Où se trouve la plus grande valeur ? La plus petite ?
3. Effectuer un parcours infixe de l'arbre. Que remarque-t-on ?

Correction

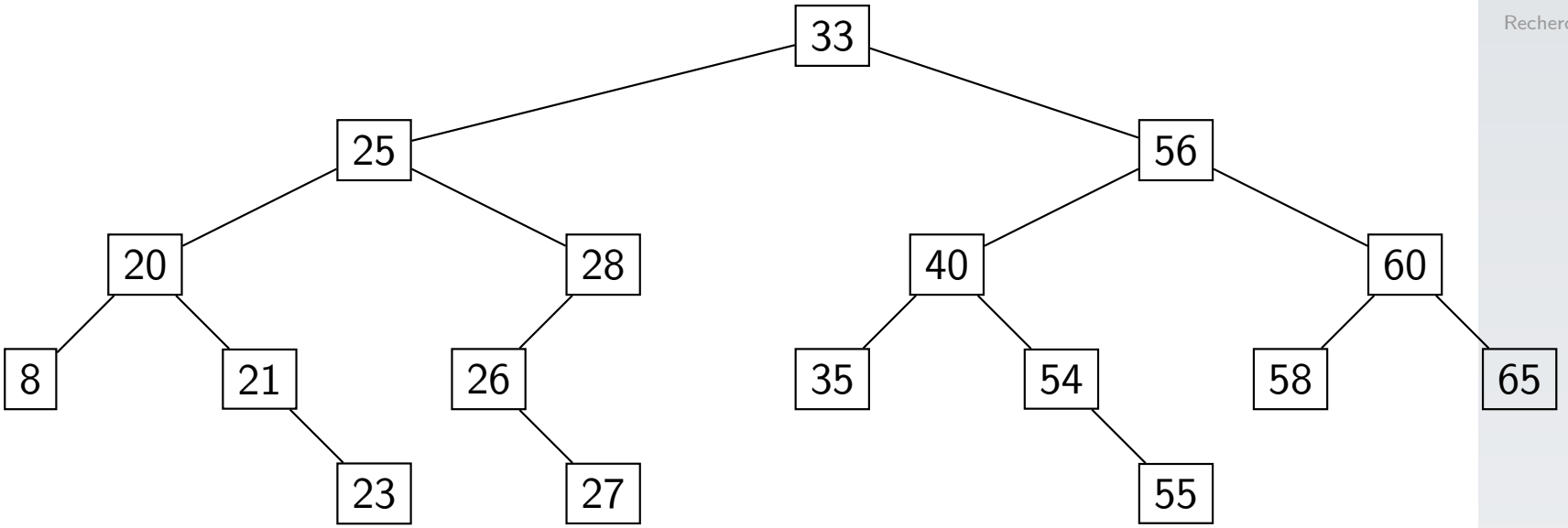


FIGURE 2 – Activité 1

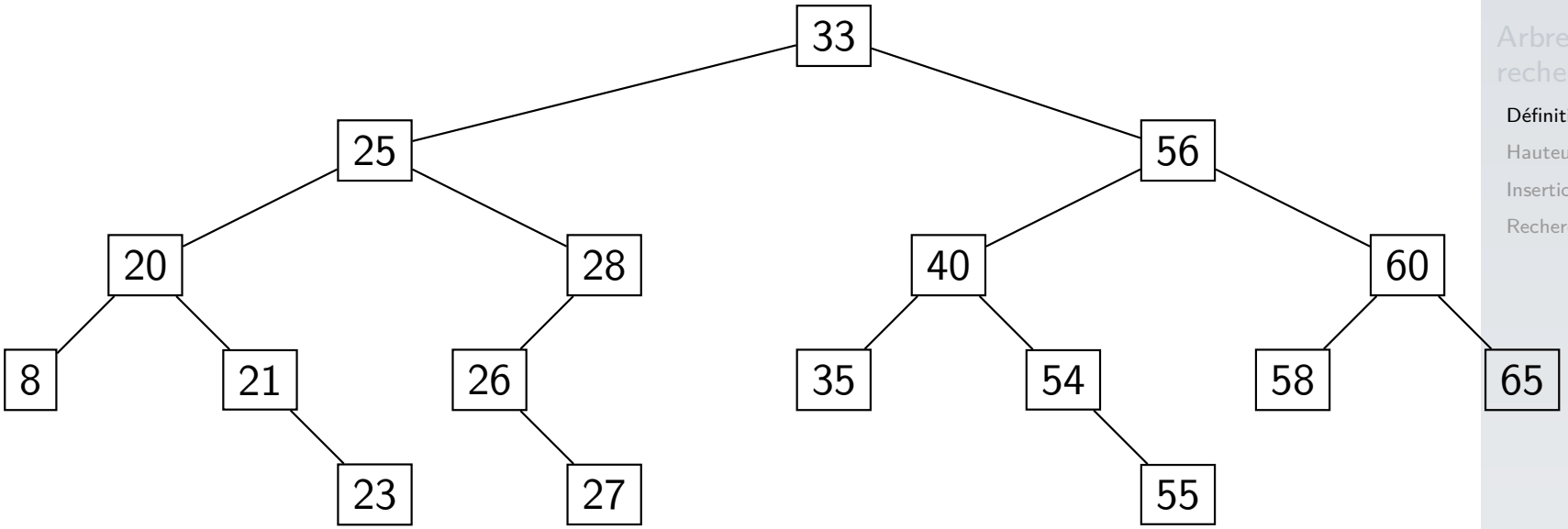
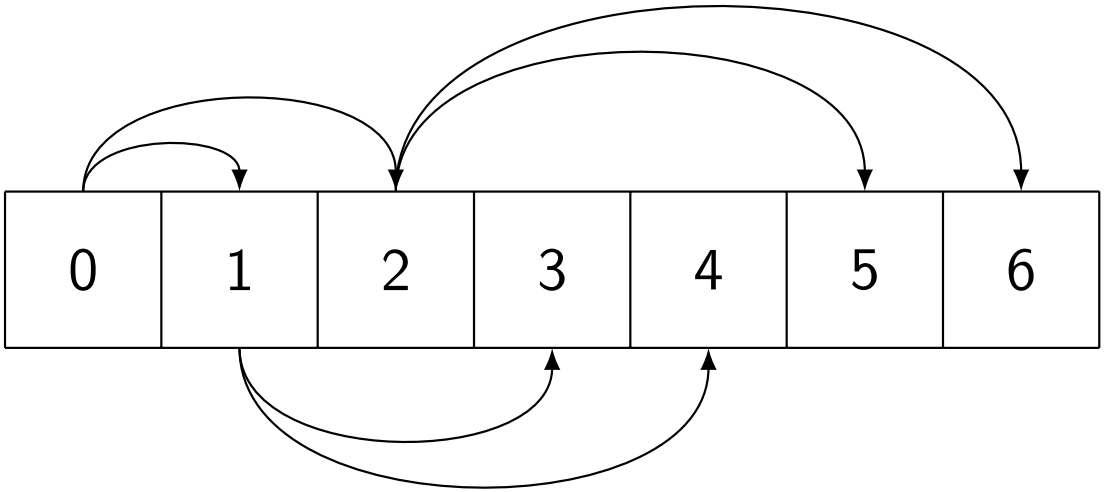


FIGURE 3 – Activité 1

- La plus petite valeur est dans le nœud le plus à gauche.
- La plus grande valeur est dans le nœud le plus à droite.
- parcours infixe : 8 - 20 - 21 - 23 - 25 - 26 - 27 - 28 - 33 - 35 - 40 - 54 - 55 - 56 - 58 - 60 - 65



Activité 2 : Il est judicieux d'utiliser un tableau pour représenter un arbre binaire de recherche.
Construire par compréhension un tableau **`arbre`** rempli de cent 0. On considérera dans la suite, que la valeur 0 représente un nœud vide.

Correction

```
1 arbre = [0 for _ in range(100)]
```

Sommaire

Hauteur

$$h + 1 \leq n \leq 2^{h+1} - 1$$

Code 1 – Propriété des arbres binaires

On peut également montrer que :

$$\log_2(n+1) - 1 \leq h \leq n - 1$$

À retenir

Dans un arbre binaire **équilibré** :

$$h = \log_2(n)$$

Hors programme

On peut rééquilibrer un arbre en effectuant des **rotations**.

Sommaire

Insertion

Pour insérer un élément dans un arbre binaire de recherche :

- On part de la racine.
- On descend dans le sous-arbre gauche si l'élément est inférieur à la racine.
- On descend dans le sous-arbre droit si l'élément est supérieur à la racine.

On applique récursivement cet algorithme jusqu'à une feuille.

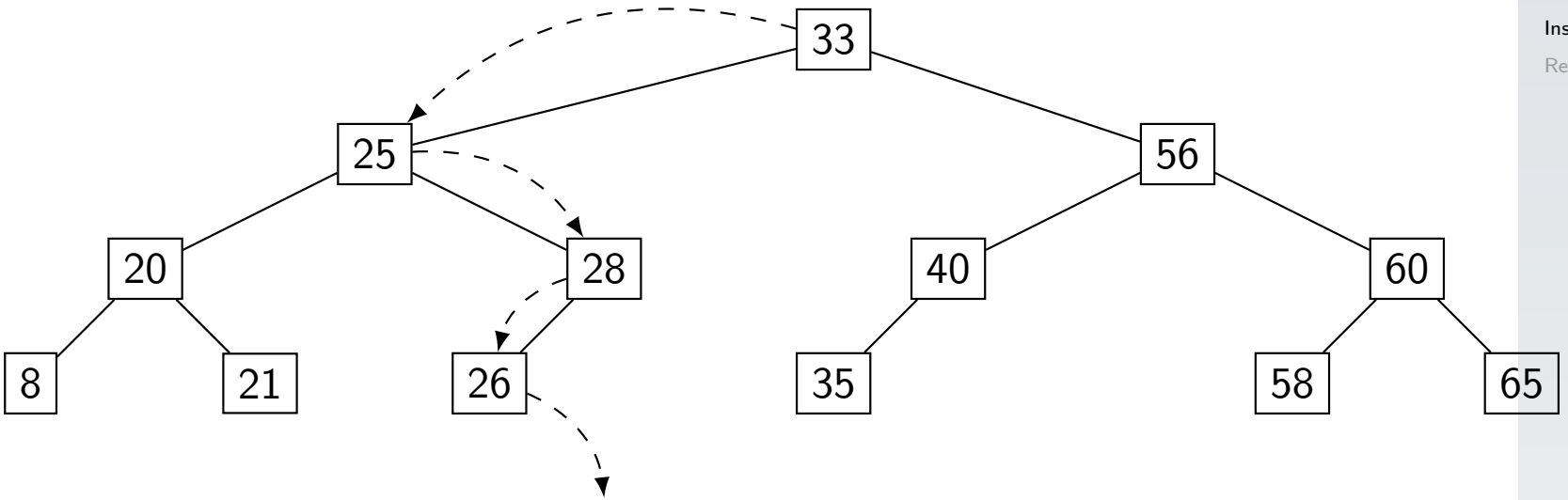


FIGURE 4 – Ajouter 27

Activité 3 :

1. Quelle est la complexité temporelle de cet algorithme ?
2. Écrire la fonction récursive `inserer(val: int, abr: list, i_pere: int)` $\rightarrow$ `None` qui insère `val` dans l'arbre de recherche `abr` représenté par un tableau.
3. Insérer dans l'ordre les valeurs : 33, 56, 25, 20, 28, 40, 21, 8, 26, 60, 35, 58, 65.

Correction


```python
def inserer(val: int, abr: list, i_pere: int) -> None:
    if abr[i_pere] == 0:  # cellule vide: cas limite
        abr[i_pere] = val
    elif val < abr[i_pere]:
        inserer(val, abr, 2*i_pere+1)  # gauche
    else:
        inserer(val, abr, 2*i_pere+2)  # droite
```

À retenir

L'insertion a une complexité linéaire $O(n)$. On parcourt au maximum la hauteur de l'arbre.

```
1 inserer(33, arbre, 0)
2 inserer(56, arbre, 0)
3 inserer(25, arbre, 0)
4 ...
```

Remarque

Selon l'ordre d'ajout, l'arbre produit ne sera pas le même.

Sommaire

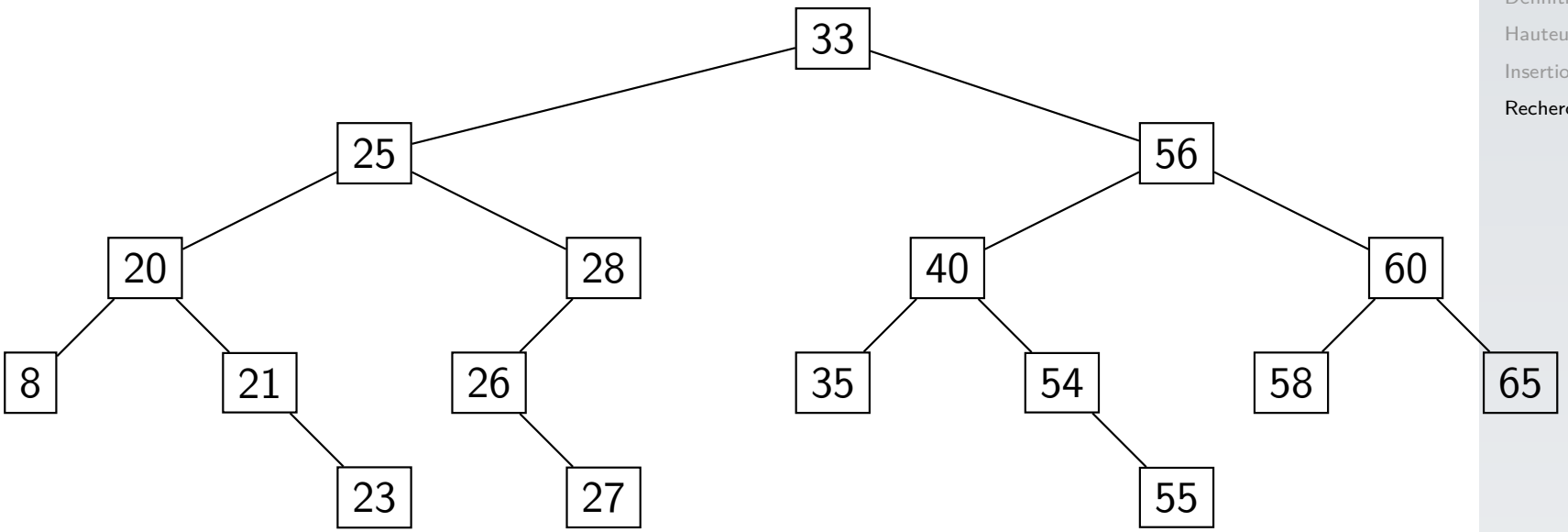
Recherche

Pour rechercher un élément dans un arbre binaire de recherche :

- On part de la racine.
- On descend dans le sous-arbre gauche si l'élément est inférieur à la racine.
- On descend dans le sous-arbre droit si l'élément est supérieur à la racine.

On applique récursivement cet algorithme jusqu'à trouver l'élément ou bien arriver à une feuille.

```
1 [33, 25, 56, 20, 28, 40, 60, 8, 21, 26, 35, 58, 65]
```



Activité 4 :

1. Quelle la complexité temporelle dans le pire des cas de la recherche d'un élément dans le tableau ?
2. Que devient cette complexité pour un arbre binaire de recherche ?

Correction

- Dans un tableau la recherche a une complexité **linéaire,** $O(n)$.
- Dans un ABR la recherche a une complexité **logarithmique,** $O(\log_2(n))$ dans le pire des cas (c'est à dire quand on ne trouve pas l'élément).

Activité 5 : Écrire la fonction récursive `rechercher(val: int, abr: list, i_pere: int)` $\rightarrow$ `bool` qui cherche si `val` est présent dans l'arbre `abr`.

Correction


```python
def rechercher(val: int, abr: list, i_pere: int) -> bool:
    if abr[i_pere] == 0: # non trouvé
        return False
    elif abr[i_pere] == val: # trouvé
        return True
    elif val < abr[i_pere]: # gauche
        rechercher(val, abr, 2*i_pere+1)
    else: # droit
        rechercher(val, abr, 2*i_pere+2)
```