



Université Blida 1  
Faculté des Sciences  
Département d’Informatique  
Master IL (Ingénierie Logiciel)  
Semestre 1



# TP 2 : MANIPULER LES ARBRES EQUILIBRÉS

Module : Algorithme avancé  
Professeur : 1 Mme AROUSSI  
[\(s\\_aroussi@esi.dz\)](mailto:s_aroussi@esi.dz)  
Anné : 2025 - 2026

08

## OUR GROUPE :

**SEKKAL Wassila**

**ESCHROUGUI Yousra**

**Hadj Mohammed douae**

**Fares yasmine**

**Galleze nada**

**Lezoul imene**



# PLAN DE PRESENTATION

- 01 Introduction
- 02 Environnement utilisé
- 03 Les bibliothèques
- 04 Structures de données et Opérations
- 05 Mode d'emploi
- 06 Résultats
- 07 Analyse critique et Conclusion

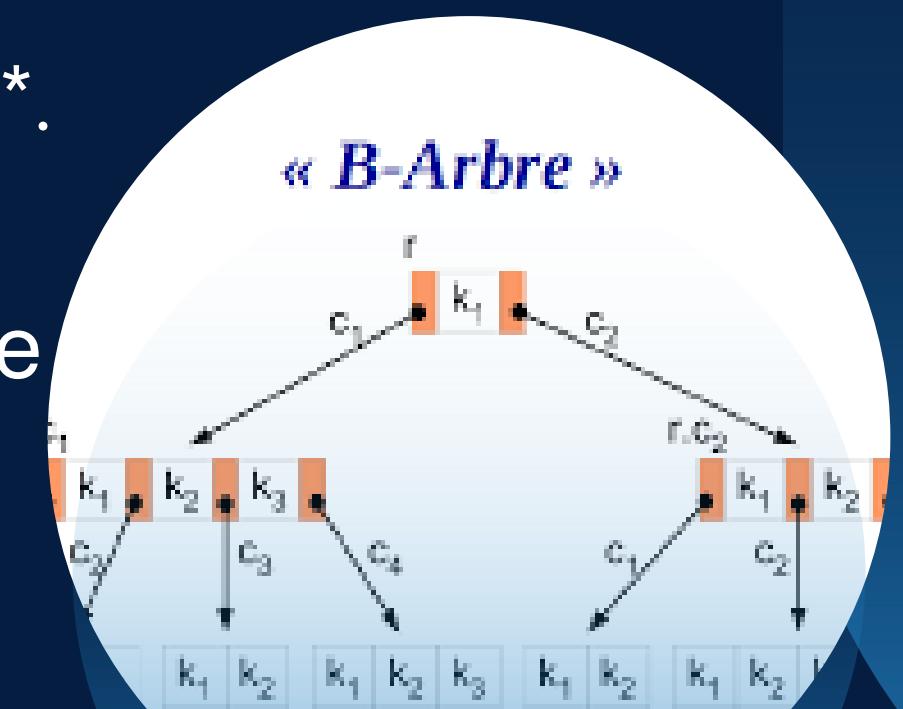
01

# INTRODUCTION

Les arbres équilibrés sont essentiels dans les structures de données et les bases de données. Ils garantissent une gestion efficace des opérations de recherche, d'insertion et de suppression. Les B-arbres et B-arbres\* sont parmi les plus utilisés pour optimiser l'accès aux données et sont des arbres de recherche équilibrés (AMR).

Ce TP vise à :

- Comprendre la structure et les propriétés des B-arbres et B-arbre\*.
- Implémenter les opérations : recherche, insertion et suppression.
- Visualiser le fonctionnement des arbres à l'aide d'une bibliothèque graphique.



# 01 INTRODUCTION

Comparaison entre B-arbre et B-arbre\* :

Critère	B-arbre	B-arbre*
<b>Définition</b>	Un <b>B-arbre</b> est un <b>arbre de recherche multibranche équilibré (AMR équilibré)</b> d'ordre $m$ ( $m$ est impair, i.e. $m = 2d + 1$ , $d \geq 1$ ) utilisé pour stocker des données de manière organisée.	Un <b>B-arbre*</b> est une <b>version améliorée du B-arbre</b> , c'est un <b>arbre de recherche multibranche équilibré (AMR équilibré)</b> d'ordre $m$ utilisé pour stocker les données de façon plus efficace.
<b>Feuilles</b>	stocker des données de manière organisée.	Toutes les feuilles se trouvent au même niveau.
<b>Nombre de clés par nœud</b>	<ul style="list-style-type: none"> <li>Racine : <math>1 \leq k \leq m-1</math></li> <li>Nœud non racine : <math>\lceil (1/2) \times m \rceil \leq k \leq m</math></li> </ul>	<ul style="list-style-type: none"> <li>Racine : <math>1 \leq k \leq m-1</math></li> <li>Nœud non racine : <math>\lceil (2/3) \times m \rceil \leq k \leq m</math></li> </ul>
<b>Racine</b>	Si ce n'est pas une feuille, elle possède au moins deux enfants.	Si ce n'est pas une feuille, elle possède au moins deux enfants.
<b>Taux de remplissage</b>	Entre 50 % et 100 %.	Entre 67 % et 100 %, grâce à un mécanisme de <b>redistribution</b> avant tout éclatement.

01

# INTRODUCTION

Comparaison entre B-arbre et B'-arbre\* :

<b>Ordre m</b>	Généralement impair pour avoir une clé médiane unique lors de l'éclatement.	Peut être pair ou impair (pas de contrainte particulière).
<b>Objectif</b>	Accélérer les opérations de <b>recherche, insertion et suppression</b> tout en maintenant l'équilibre de la structure.	<b>Réduire le nombre d'éclatements et optimiser l'utilisation de l'espace mémoire</b> , tout en gardant de bonnes performances pour les opérations de recherche, d'insertion et de suppression.

# 01 Compréhension de l'arbre B-arbre\* :

<b>Insertion</b>	Lorsqu'un nœud est plein, il éclate directement en deux (split en 2).	<ul style="list-style-type: none"><li><b>Racine</b> : lorsqu'un nœud est plein, il éclate directement en deux (split en 2).</li><li><b>Nœud non racine</b> : avant d'éclater, le nœud tente une redistribution des clés avec un frère. Si ce n'est pas possible → éclatement en trois nœuds (split en 3).</li></ul>
<b>Suppression</b>	appliquer rééquilibrage : <ul style="list-style-type: none"><li><b>Cas 1 (Emprunt)</b> : emprunter une clé d'un frère avec plus de <math>d</math> clés.</li><li><b>Cas 2 (Fusion)</b> : fusionner le nœud avec un frère avec moins de <math>d</math> clés.</li></ul>	<ul style="list-style-type: none"><li><b>Redistribution</b> : si un frère a plus de clés que <math>k_{min}</math>, redistribuer les clés entre les nœuds.</li><li><b>Fusion</b> : si redistribution impossible, fusionner nœuds + clé du père.</li><li>Si fusion viole <math>k_{max}</math>, appliquer <b>éclatement en 3</b></li></ul>

## 02 ENVIRONNEMENT UTILISÉ

### ► ENVIRONNEMENT MATÉRIEL :

- Processeur : Intel(R)core(TM)i7-5600U CPU @ 2.60GHZ
- RAM: 4.00 GB
- Carte Graphique : Intel(R) HD Graphics 5500

### ► ENVIRONNEMENT LOGICIEL :

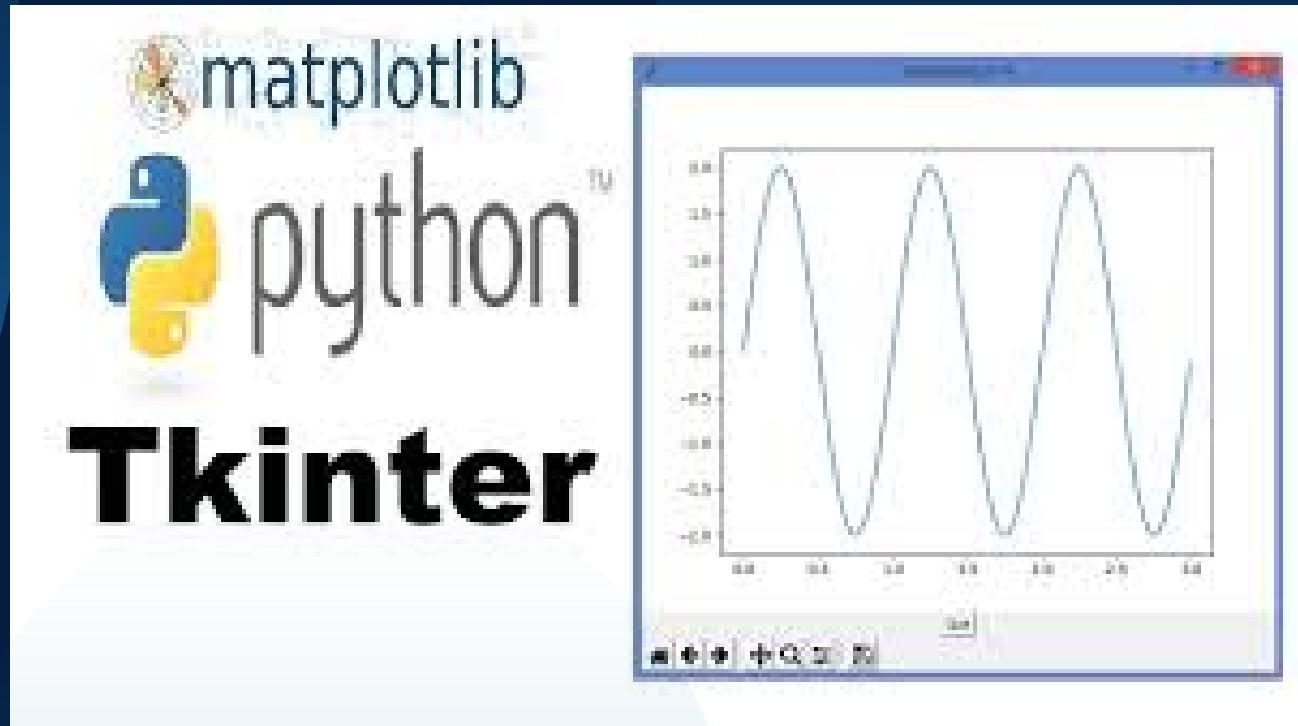
- Os utilisé : Windows 10 PRO
- Language :Python “3.12.3”
- Environnement de développement: Visual Studio Code
- Bibliothèque : Tkinter , Matplotlib , uuid, typing, math

### ► ASSISTANCE DU CHATBOT IA :

ChatGPT (OpenAI) a contribué à hauteur d'environ 60 % d'assistance (correction, amélioration du code et visualisation graphique)



# 03 LES BIBLIOTHÈQUES



## Tkinter: Définition et rôle

C'est un bibliothèque standard de Python pour la création d'interfaces graphiques (GUI).

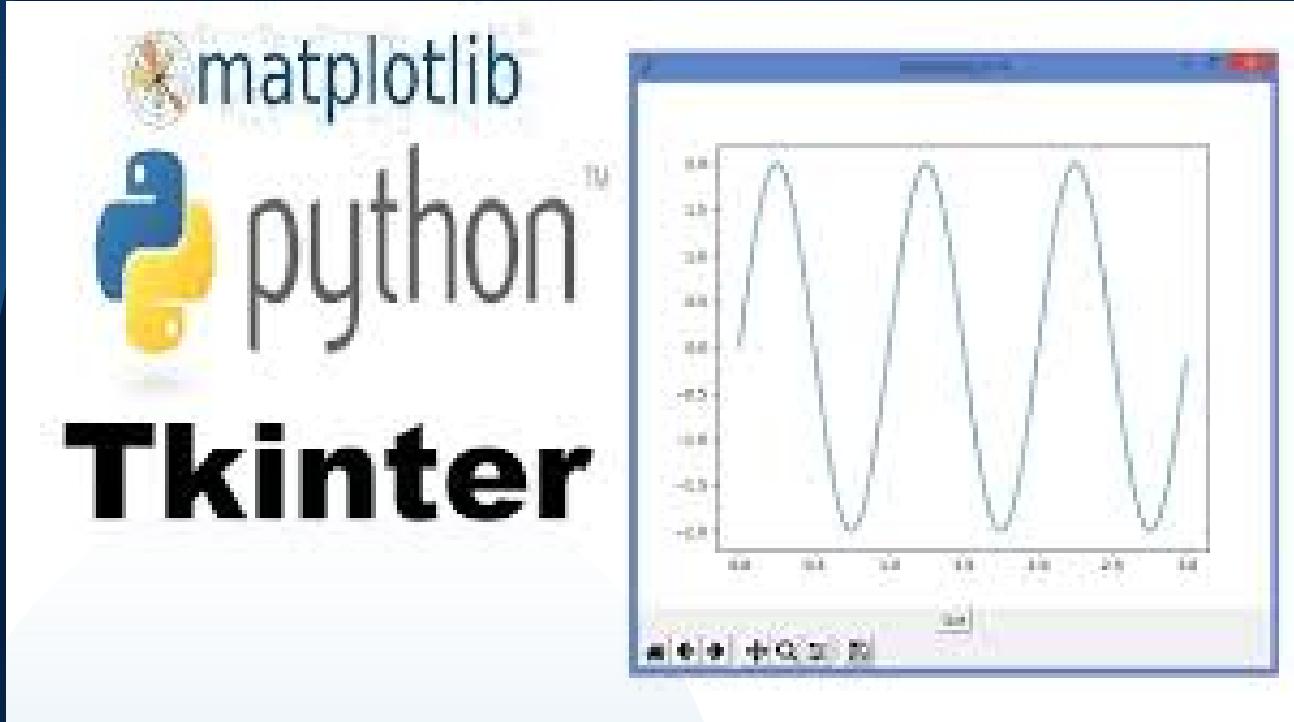
## Matplotlib: Définition et rôle

C'est un bibliothèque de traçage utilisée pour la visualisation graphique des graphes et arbres.

## Typing : Définition et rôle

C'est un bibliothèque pour indiquer les types de données dans le code (comme List, Optional, etc.).

# 03 LES BIBLIOTHÈQUES



## Uuid : Définition et rôle

C'est un bibliothèque permet de générer des identifiants uniques pour différencier les objets.

## Math : Définition et rôle

C'est un bibliothèque contient des fonctions mathématiques comme ceil, sqrt, sin, etc.

## ► Le principe d'équilibrage de B-arbre\* :

- Principe d'équilibrage :

Un B-arbre\* conserve son équilibre après chaque opération d'insertion ou de suppression.

Contrairement à un B-arbre , où un nœud plein éclate immédiatement, un B-arbre\* tente d'abord de redistribuer les clés entre les nœuds frères afin de minimiser les éclatements. Cette redistribution se fait de manière à ce que chaque nœud soit rempli au moins aux  $\frac{2}{3}$  de sa capacité, ce qui permet d'optimiser l'utilisation de la mémoire et de réduire la hauteur de l'arbre.

Voici un exemple illustratif montrant cette différence dans une série de d'opérations :

## ► Le principe d'équilibrage de B-arbre\*:

- Exemple d'équilibrage après insertion

Cas 1: Si un nœud devient plein → on tente une redistribution 2/3 avec un frère

Données initiales :

$$L = [25, 60, 35, 10, 5, 20, 65, 45, 70, 40, 50]$$

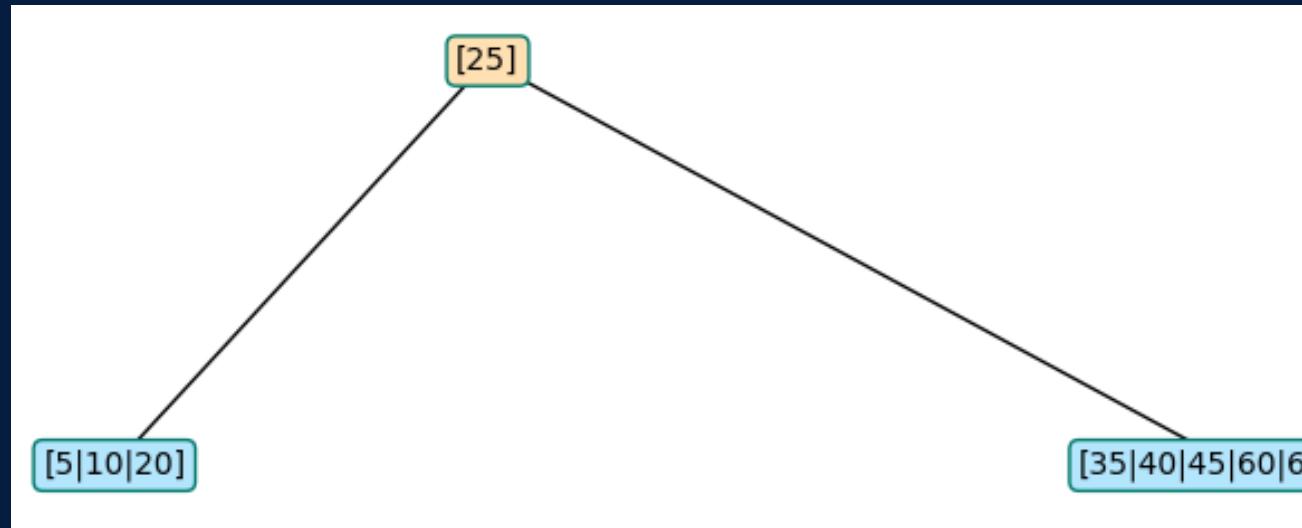
$$\text{Ordre } m = 7$$

$$K_{\max} = m-1 = 6$$

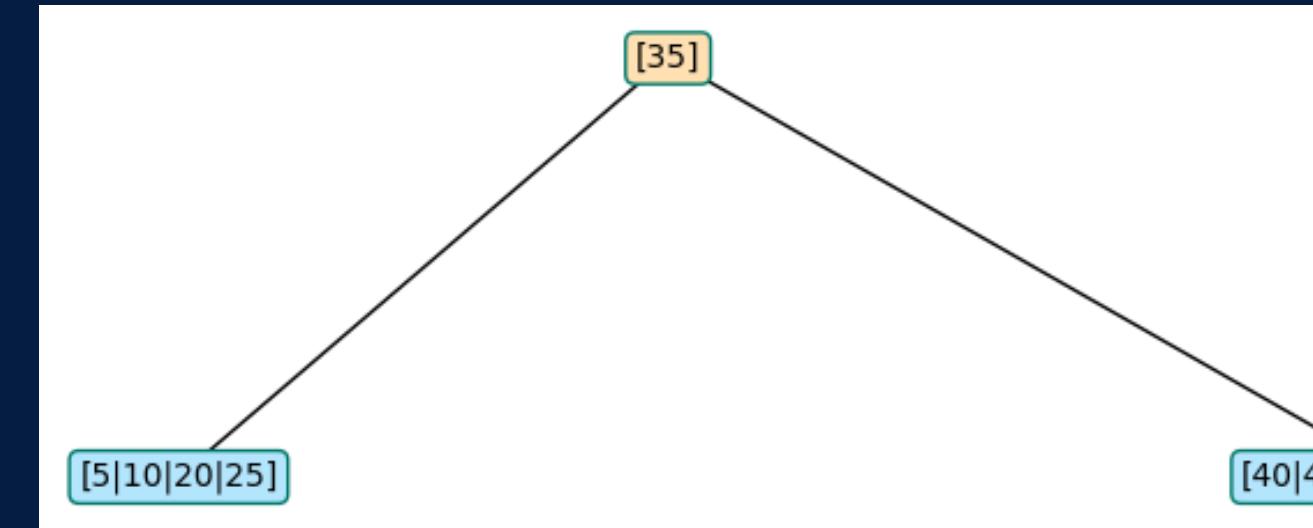
$$K_{\min} = 2*m/3 = 4$$

## ► Le principe d'équilibrage de B-arbre\*:

Avant redistribution 2/3 :



Après redistribution 2/3 :



Cas 2 : Si la redistribution 2/3 est impossible → on effectue un écletement en 3.

Données initiales :

$$L = [25, 60, 35, 10, 5, 20, 65, 45, 70, 40]$$

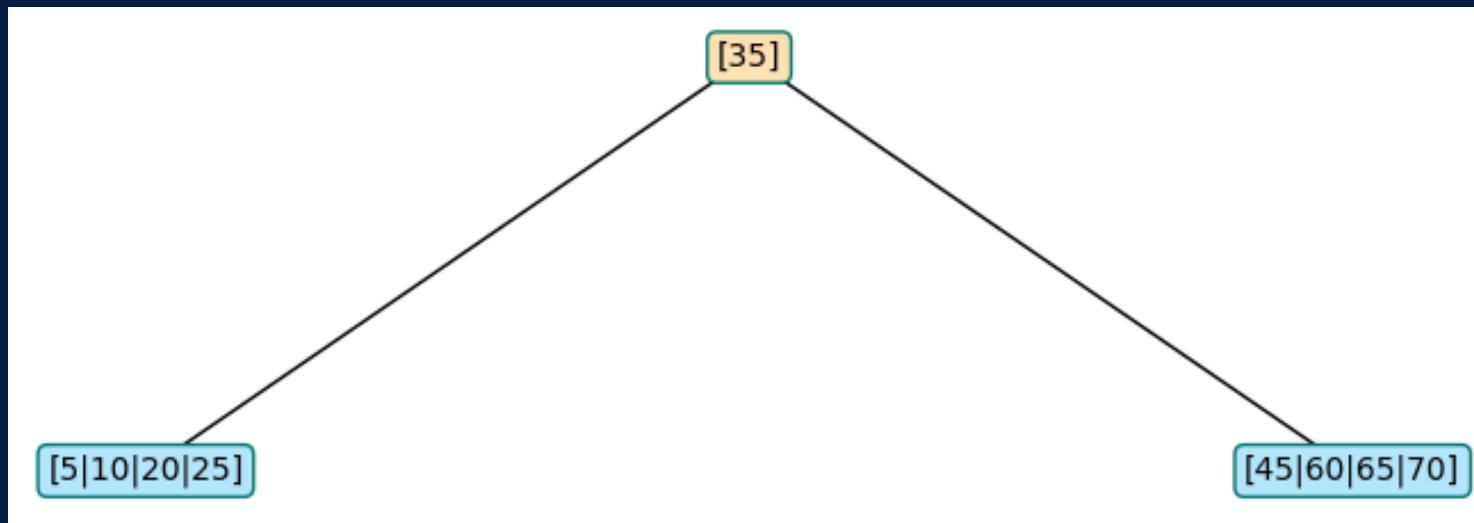
$$\text{Ordre } m = 5$$

$$K_{\max} = m-1 = 4$$

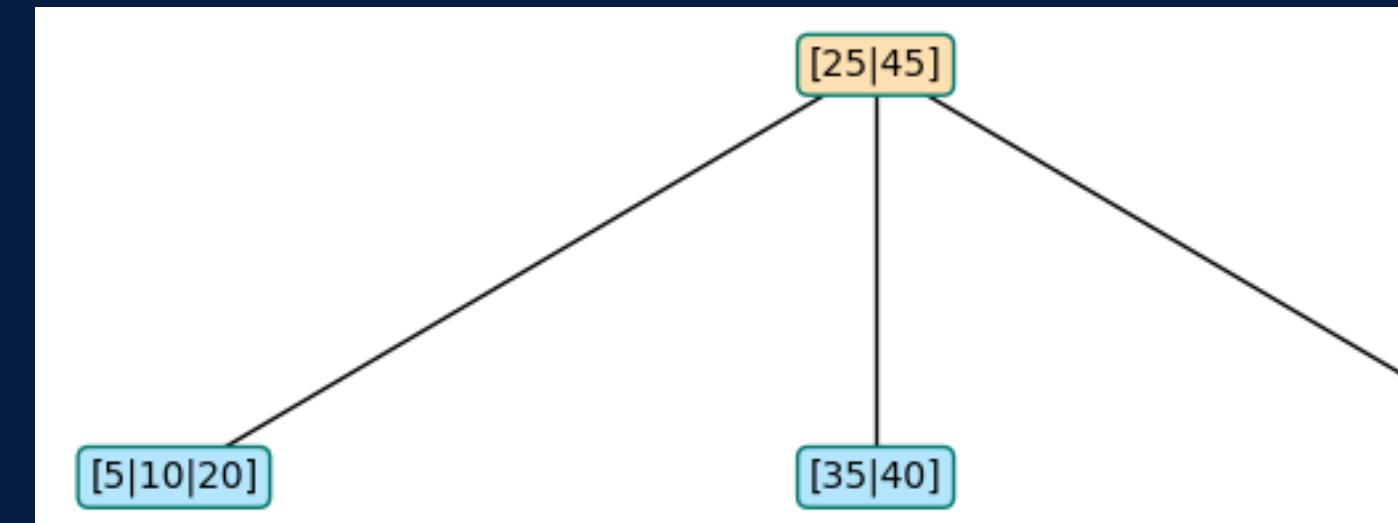
$$K_{\min} = 2*m/3 = 3$$

## ► Le principe d'équilibrage de B-arbre\*:

Avant écletement en 3 :



Après écletement en 3 :



- Exemple d'équilibrage après suppression

Cas 1: Redistribution 2/3 après suppression

Données initiales :

L1= [25, 60, 35, 10, 5, 20, 65, 45, 70, 40, 50, 55, 30, 15, 22, 62, 64, 4, 8]

L2 =[15, 20, 10]

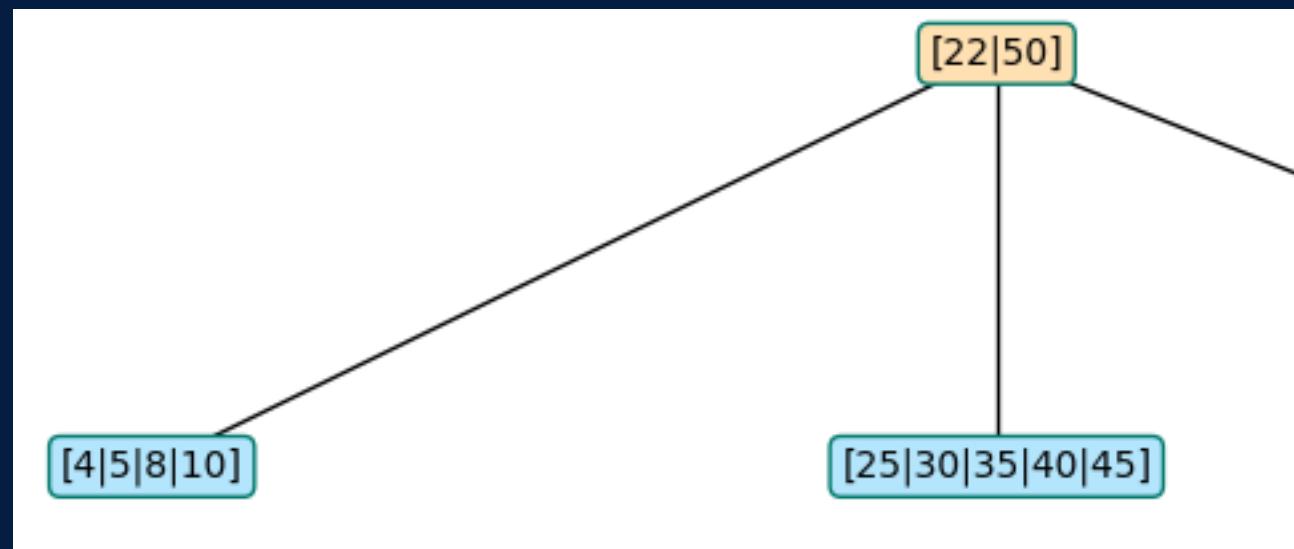
ordre = 7

Kmax = 6

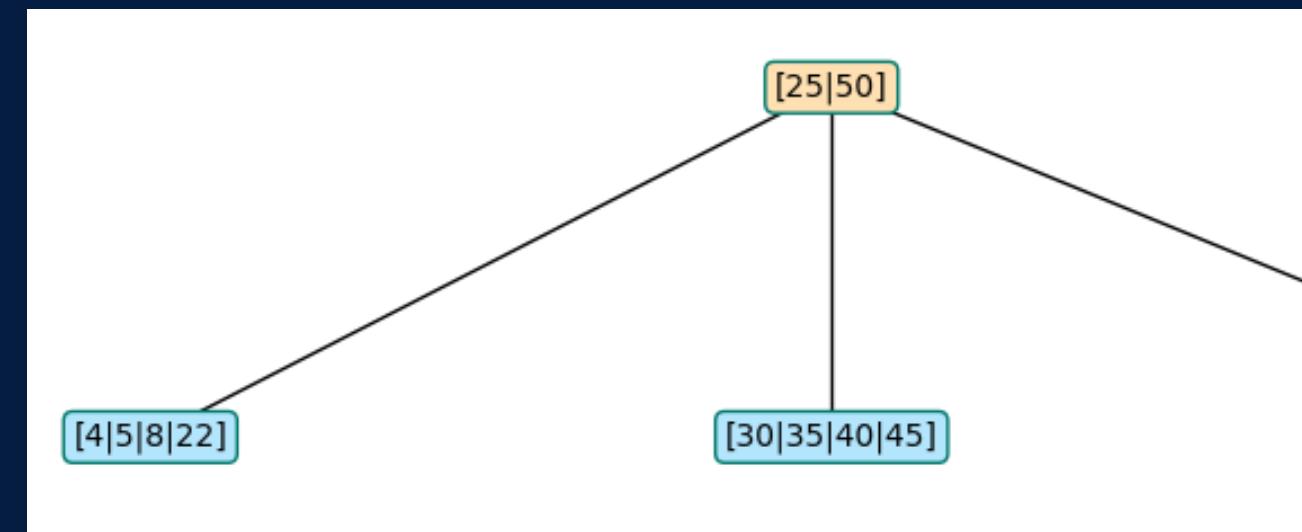
Kmin = 4

## ► Le principe d'équilibrage de B-arbre\*:

Avant suppression de 10 :



Après redistribution 2/3:



Cas 2 : Fusion après suppression

Données initiales :

L1= [25, 60, 35, 10, 5, 20, 65, 45, 70, 40, 50, 55]

L2 =[20]

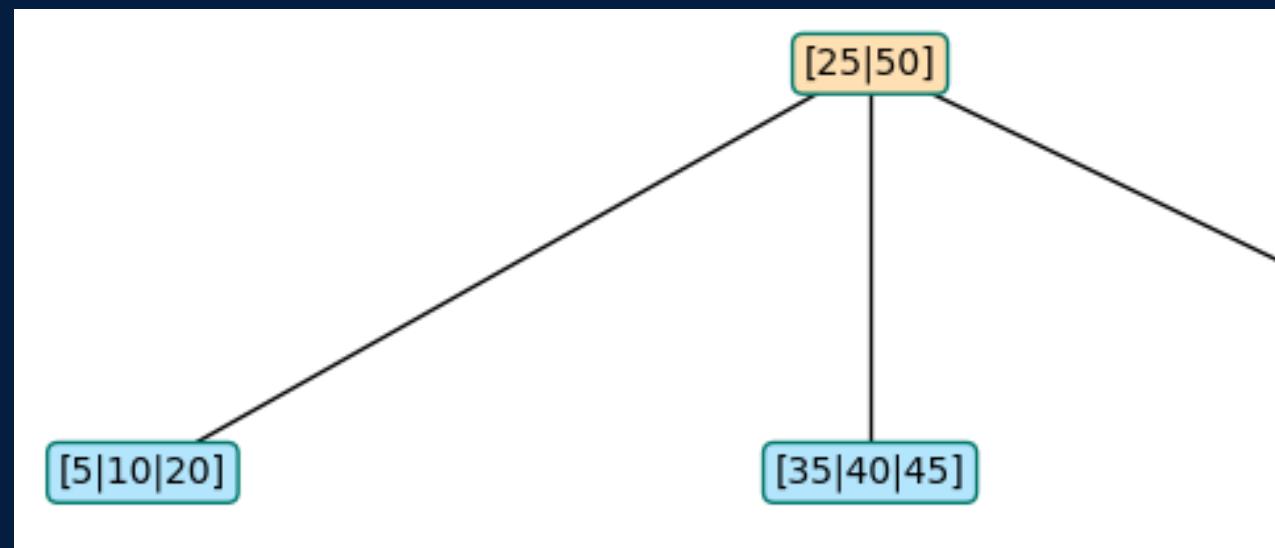
ordre = 5

Kmax = 4

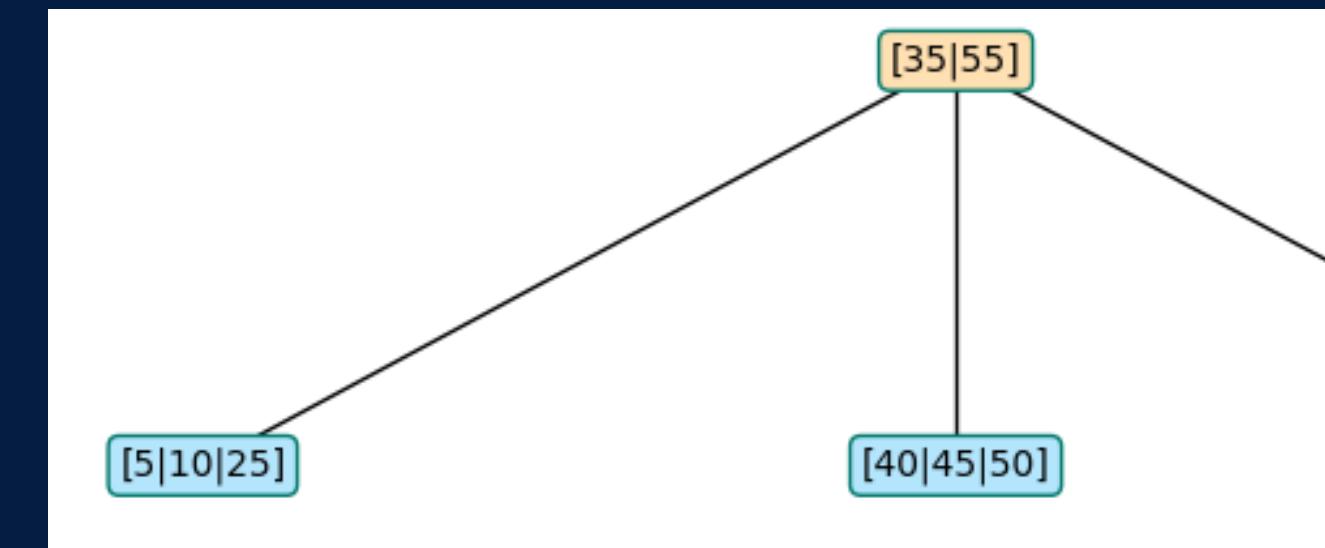
Kmin = 3

## ► Le principe d'équilibrage de B-arbre\*:

Avant suppression de 20 :



Après Fusion:



## ► La structure des données utilisée pour les types d'arbres :

- Structure du B-arbre : La structure principale est définie par deux classes :

La structure d'un nœud :

```
class BTreenode:  
    def __init__(self, leaf: bool = True):  
        self.leaf: bool = leaf  
        self.keys: List[int] = []  
        self.children: List['BTreenode'] = []  
        self.parent: Optional['BTreenode'] = None  
        self._id = str(uuid.uuid4())[:8]  
  
    def __repr__(self):  
        return f"Node(keys={self.keys}, leaf={self.leaf})"
```

La classe principale :

```
__init__(self, order: int = 7):  
    if order < 3:  
        raise ValueError("Order must be >= 3")  
    self.order = order  
    l = (order - 1) // 2          # Degré minimal  
    max_keys = 2 * self.d         # Nombre max de clés par nœud  
    min_keys = self.d             # Nombre min de clés  
    root = BTreenode(leaf=True)
```

## ► La structure des données utilisée pour les types d'arbres :

- Structure du B-arbre\* : La structure principale est définie par deux classes :

La structure d'un nœud :

```
class BStarNode:  
    def __init__(self, leaf: bool = True):  
        self.leaf: bool = leaf  
        self.keys: List[int] = []  
        self.children: List[BStarNode] = []  
  
    def is_full(self, m: int) -> bool:  
        return len(self.keys) >= m  
  
    def __repr__(self):  
        return f"BStarNode(leaf={self.leaf}, keys={self.keys})"
```

La classe principale :

```
class BStarTree:  
    def __init__(self, order: int = 7):  
        if order < 3:  
            raise ValueError("Order must be >= 3")  
        self.m = order  
        self.kmin = round(2 * (self.m - 1) / 3)  
        self.kmax = self.m - 1  
        self.root = BStarNode(leaf=True)
```

Chaque nœud du B-arbre\* contient entre  $(2/3)*m$  et  $m-1$  clés, assurant un remplissage minimal de 66 %

## ► Les algorithmes des opérations principales de B-arbre :

- Recherche :

```
def search(self, k: int, node: Optional[BTreeNode] = None) -> Optional[Tuple[BTreeNode, int]]:  
    if node is None:  
        node = self.root  
    i = 0  
    while i < len(node.keys) and k > node.keys[i]:  
        i += 1  
    if i < len(node.keys) and node.keys[i] == k:  
        return node, i  
    if node.leaf:  
        return None  
    return self.search(k, node.children[i])
```

► Les algorithmes des opérations principales de B-arbre :

- Insertion : Insertion dans la racine

```
def insert(self, key: int):  
    node = self.root  
    self._insert_non_full(node, key)  
  
    if len(self.root.keys) > self.max_keys:  
        new_root = BTreenode(leaf=False)  
        new_root.children.append(self.root)
```

## ► Les algorithmes des opérations principales de B-arbre :

- Insertion : Insertion dans un nœud non plein

```
def _insert_non_full(self, node: BTreenode, key: int):  
    if node.leaf:  
        i = 0  
        while i < len(node.keys) and key > node.keys[i]:  
            i += 1  
        node.keys.insert(i, key)  
  
    if len(node.keys) > self.max_keys:  
        self.handle_split(node)
```

## ► Les algorithmes des opérations principales de B-arbre :

- Insertion : Eclatement d'un nœud plein

```
def _split_child(self, parent: BTTreeNode, i: int):  
    full = parent.children[i]  
    mid = len(full.keys) // 2  
    median = full.keys[mid]  
  
    new_node = BTTreeNode(leaf=full.leaf)  
    new_node.parent = parent  
  
    new_node.keys = full.keys[mid + 1:]  
    full.keys = full.keys[:mid]
```

## ► Les algorithmes des opérations principales de B-arbre :

- Suppression : Fonction principale

```
def delete(self, k: int):  
    if not self.root:  
        return  
  
    self._delete_internal(self.root, k)  
  
    if self.root and len(self.root.keys) == 0:  
        if self.root.leaf:  
            self.root = None
```

## ► Les algorithmes des opérations principales de B-arbre :

- Suppression : Fonction interne

```
def _delete_internal(self, node: BTTreeNode, k: int):  
    """  
        # Étape 1 : trouver l'indice où la clé k se trouve (ou devrait se trouver)  
        idx = self._find_key_index(node, k)  
  
        # Étape 2 : la clé k est présente dans ce nœud  
        if idx < len(node.keys) and node.keys[idx] == k:  
  
            # Cas 1 : si le nœud est une feuille → suppression directe  
            if node.leaf:  
                node.keys.pop(idx)  
                return  
  
            # Cas 2 : si c'est un nœud interne  
            else:  
                ...  
    """
```

## ► Les algorithmes des opérations principales de B-arbre :

- Suppression : Descente dans le sous-arbre

```
else:  
    # Si c'est une feuille → rien à supprimer  
    if node.leaf:  
        return  
    # Détermination du fils correct à explorer  
    flag = (idx == len(node.keys))  
    child = node.children[idx]  
    # Avant de descendre : vérifier si le fils a le nombre minimal de clés  
    if len(child.keys) == self.min_keys:  
        # Cas 1 : emprunter une clé au frère gauche  
        if idx > 0 and len(node.children[idx - 1].keys) > self.min_keys:  
            self._borrow_from_prev(node, idx)  
        # Cas 2 : emprunter une clé au frère droit  
        elif idx < len(node.children) - 1 and len(node.children[idx + 1].keys) > self.min_keys:  
            self._borrow_from_next(node, idx)  
        # Cas 3 : fusionner avec un frère (si aucun emprunt possible)  
    else:
```

## ► Les algorithmes des opérations principales de B-arbre :

- Suppression : Outils de suppression : Elles permettent de trouver la position, le prédecesseur ou le successeur d'une clé

```
def _find_key_index(self, node: BTreenode, k: int) -> int:  
    idx = 0  
  
    while idx < len(node.keys) and node.keys[idx] < k:  
        idx += 1  
  
    return idx  
  
  
def _get_predecessor(self, node: BTreenode) -> int:  
    cur = node  
  
    while not cur.leaf:  
        cur = cur.children[-1]  
  
    return cur.keys[-1]
```

## ► Les algorithmes des opérations principales de B-arbre :

- Suppression : Rééquilibrage : Emprunt et Fusion

```
def _borrow_from_prev(self, parent: BTTreeNode, idx: int):
    # Emprunt d'une clé du frère gauche pour équilibrer le fils
    child = parent.children[idx]
    sibling = parent.children[idx - 1]
    child.keys.insert(0, parent.keys[idx - 1])
    if not sibling.leaf:
        child.children.insert(0, sibling.children.pop())
    parent.keys[idx - 1] = sibling.keys.pop()

def _borrow_from_next(self, parent: BTTreeNode, idx: int):
    # Emprunt d'une clé du frère droit pour équilibrer le fils
    child = parent.children[idx]
    sibling = parent.children[idx + 1]
    child.keys.append(parent.keys[idx])
    if not sibling.leaf:
        child.children.append(sibling.children.pop(0))
    parent.keys[idx] = sibling.keys.pop(0)
```

## ▶ Les algorithmes des opérations principales de B-arbre\* :

- Recherche : même principe d'un B-arbre

**Copier le code** 

`python`

## ► Les algorithmes des opérations principales de B-arbre\* :

- Insertion : Insertion dans la racine

```
def insert(self, key: int):
    print(f"\n--- INSERT {key} ---")
    root = self.root

    if not root.keys:
        bisect.insort(root.keys, key)
        return

    if root.is_full(self.m):
        if root.leaf:
            old_root = self.root
            new_root = BStarNode(leaf=False)
            new_root.children.append(old_root)
            self.root = new_root
            self._split_child(new_root, 0)
        return
```

```
t = root.children[0]
ht = root.children[1] if len(root.children) >
left and len(left.keys) < self.kmax:
self._redistribute(root, 0)
if right and len(right.keys) < self.kmax:
self._redistribute(root, 0)
else:
self._three_way_split_strict(root, 0)
root_nonfull(self.root, key)
```

## ► Les algorithmes des opérations principales de B-arbre\* :

- Insertion : Insertion dans un nœud non plein

```
def _insert_nonfull(self, node: BStarNode, key: int, parent: Optional[BStarNode] = None, idx_in_parent: int = -1):
    if node.leaf:
        bisect.insort(node.keys, key)

    if len(node.keys) > self.kmax:
        if parent is None:
            old_root = self.root
            new_root = BStarNode(leaf=False)
            new_root.children.append(old_root)
            self.root = new_root
            self._split_child(new_root, 0)
        else:
            left_sib = parent.children[idx_in_parent - 1] if idx_in_parent - 1 >= 0 else None
            right_sib = parent.children[idx_in_parent + 1] if idx_in_parent + 1 < len(parent.children) else None

            if left_sib and len(left_sib.keys) < self.kmin:
                self._redistribute(parent, idx_in_parent - 1)
            elif right_sib and len(right_sib.keys) < self.kmin:
                self._redistribute(parent, idx_in_parent)
            elif left_sib and len(left_sib.keys) < self.kmax:
                self._redistribute(parent, idx_in_parent - 1)
            elif right_sib and len(right_sib.keys) < self.kmax:
                self._redistribute(parent, idx_in_parent)
            else:
                if left_sib:
                    self._three_way_split_strict(parent, idx_in_parent - 1)
                elif right_sib:
                    self._three_way_split_strict(parent, idx_in_parent)

    return
```

```
.bisect_left(node.keys, key)
de.children[i]

ld.is_full(self.m):
insert_nonfull(child, key, node, i)

ib = node.children[i - 1] if i - 1 >= 0 else None
sib = node.children[i + 1] if i + 1 < len(node.childre

t_sib and len(left_sib.keys) < self.kmax:
lf._redistribute(node, i - 1)
ight_sib and len(right_sib.keys) < self.kmax:
lf._redistribute(node, i)

lf._three_way_split_strict(node, i if right_sib else i
insert_nonfull(node, key, parent, idx_in_parent)
```

► Les algorithmes des opérations principales de B-arbre\* :

- Insertion : Redistribution 2/3

```
def _redistribute(self, parent: BStarNode, idx: int):  
    L = parent.children[idx]  
    R = parent.children[idx + 1]  
  
    if len(L.keys) < self.kmin or len(L.keys) < len(R.keys):  
        sep = parent.keys[idx]  
        moved_up = R.keys.pop(0)  
        L.keys.append(sep)  
        parent.keys[idx] = moved_up
```

## ► Les algorithmes des opérations principales de B-arbre\* :

- Insertion : Eclatement en 3

```
def _three_way_split_strict(self, parent: BStarNode, idx_parent: int, new_key=None):  
    L = parent.children[idx_parent]  
    R = parent.children[idx_parent + 1]  
    sep = parent.keys[idx_parent]  
  
    merged = sorted(L.keys + [sep] + R.keys + ([new_key] if new_key else []))  
  
    n = len(merged)  
    up1_idx = n // 3  
    up2_idx = (2 * n) // 3  
    up1 = merged[up1_idx]  
    up2 = merged[up2_idx]  
  
    left_keys = merged[:up1_idx]  
    mid_keys = merged[up1_idx + 1:up2_idx]  
    right_keys = merged[up2_idx + 1:]
```

```
parent.keys[idx_parent:idx_parent+1] = [up1, up2]  
  
left = BStarNode(leaf=L.leaf)  
middle = BStarNode(leaf=L.leaf)  
right = BStarNode(leaf=R.leaf)  
  
left.keys = left_keys  
middle.keys = mid_keys  
right.keys = right_keys  
  
parent.children[idx_parent] = left  
parent.children[idx_parent + 1] = middle  
parent.children.insert(idx_parent + 2, right)
```

► Les algorithmes des opérations principales de B-arbre\* :

- Suppression : Fonction principale

```
def delete(self, key: int):
    print(f"\n==== DELETE {key} ====")
    if self.root is None or not self.root.keys:
        return
    self._delete_internal(self.root, key)

    if not self.root.keys and not self.root.leaf:
```

## ► Les algorithmes des opérations principales de B-arbre\* :

- Suppression : Fonction interne

```
def _delete_internal(self, node: BStarNode, key: int, parent: Optional[BStarNode] = None, idx_in_parent: int = -1):  
    i = bisect.bisect_left(node.keys, key)  
    child = None  
  
    if node.leaf:  
        if i < len(node.keys) and node.keys[i] == key:  
            node.keys.pop(i)  
        else:  
            return  
  
    elif i < len(node.keys) and node.keys[i] == key:  
        pred_node = node.children[i]  
        while not pred_node.leaf:  
            pred_node = pred_node.children[-1]  
        predecessor = pred_node.keys[-1]  
        node.keys[i] = predecessor  
        child = node.children[i]
```

## ► Les algorithmes des opérations principales de B-arbre\* :

- Suppression : Rééquilibrage (redistribution + fusion)

```
def _fix_underflow(self, parent: BStarNode, idx: int):  
    child = parent.children[idx]  
    left_sib = parent.children[idx - 1] if idx - 1 >= 0 else None  
    right_sib = parent.children[idx + 1] if idx + 1 < len(parent.children) else None  
  
    if left_sib and len(left_sib.keys) > self.kmin:  
        self._redistribute(parent, idx - 1)  
    elif right_sib and len(right_sib.keys) > self.kmin:  
        self._redistribute(parent, idx)  
    else:
```

► Les algorithmes des opérations principales de B-arbre\* :

- Suppression : Fusion

```
def _merge_nodes(self, parent: BStarNode, idx: int):  
    left = parent.children[idx]  
    right = parent.children[idx + 1]  
    sep = parent.keys.pop(idx)  
  
    left.keys.append(sep)  
    left.keys.extend(right.keys)
```



## 05 MODE D'EMPLOI

### Les installations requises:

- Language de développement : Python (version 3.13.9)  
-lien : <https://www.python.org/downloads/>
- Environnement de développement : Visual Studio Code (version 1.90)  
-lien: <https://code.visualstudio.com/>
- Bibliothèque de graphes :  
Matplotlib (version 3.10.0)  
-lien : <https://matplotlib.org/>  
Tkinter  
-lien :  
<https://docs.python.org/3/library/tkinter.html>



## LE FORMAT DES DONNÉES D'ENTRÉE :

- Liste de clés séparées par des virgules : Par exemple  $L= [25, 60, 35, 10, 5, 20, 65, 45, 70, 40, 50, 55, 30, 15, 22, 62, 64, 4, 8 ]$
- Actions disponibles dans l'interface :
  - Choisir entre **B-arbre** et **B-arbre\*** .
  - Créer l'arbre.
  - Insérer, Supprimer, Rechercher.
  - Affichage graphique automatique .



**les commandes ou scripts à exécuter :**

- ▶ **1- INSTALLER LES BIBLIOTHÈQUES NÉCESSAIRES**
- ▶ **2-OUVRIR LE TERMINAL ET SE PLACER DANS LE DOSSIER CONTENANT LE SCRIPT PRINCIPAL**
- ▶ **3. EXÉCUTER LE SCRIPT PRINCIPAL :PYTHON APPLICATION\_GRAPHE.PY**
- ▶ **4. UTILISER L'APPLICATION**

## LES RÉSULTATS ATTENDUS :



Comprendre la structure et les propriétés des B-arbres et B-arbres\*.



Implémenter les opérations recherche, insertion et suppression.



Visualiser le fonctionnement des arbres à l'aide d'une bibliothèque graphique.

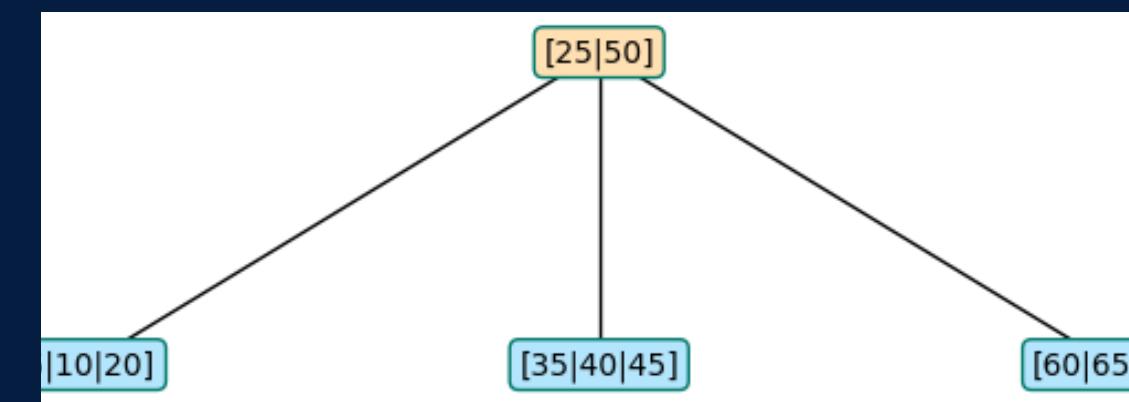
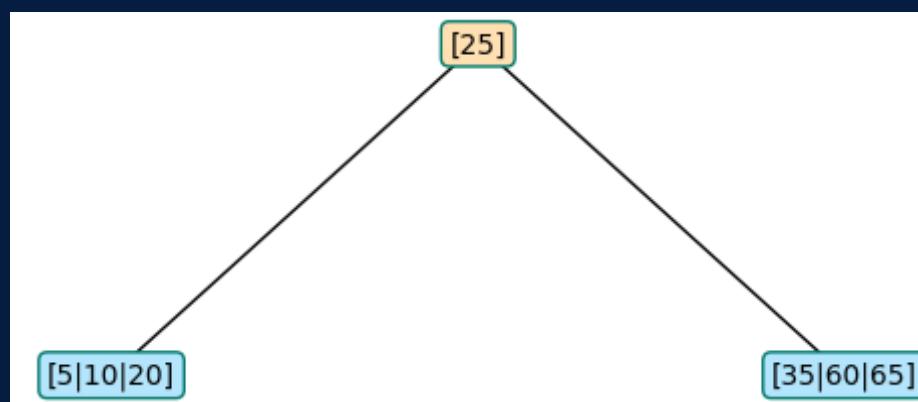
## ► Exécution des exercices de la série 1 du TD :

- Créez l'arbre R: à partir des valeurs  $L1 = [25, 60, 35, 10, 5, 20, 65, 45, 70, 40, 50, 55, 30, 15, 22, 62, 64, 4, 8]$ 
  - B-arbre d'ordre 7 :

Nous créons les clés de L1 jusqu'à la clé 65, lorsque le nœud est plein  $K_{max} = 6$ , puis nous appliquons split

Nous continuons à insérer les clés jusqu'à la clé 50 du côté droit.

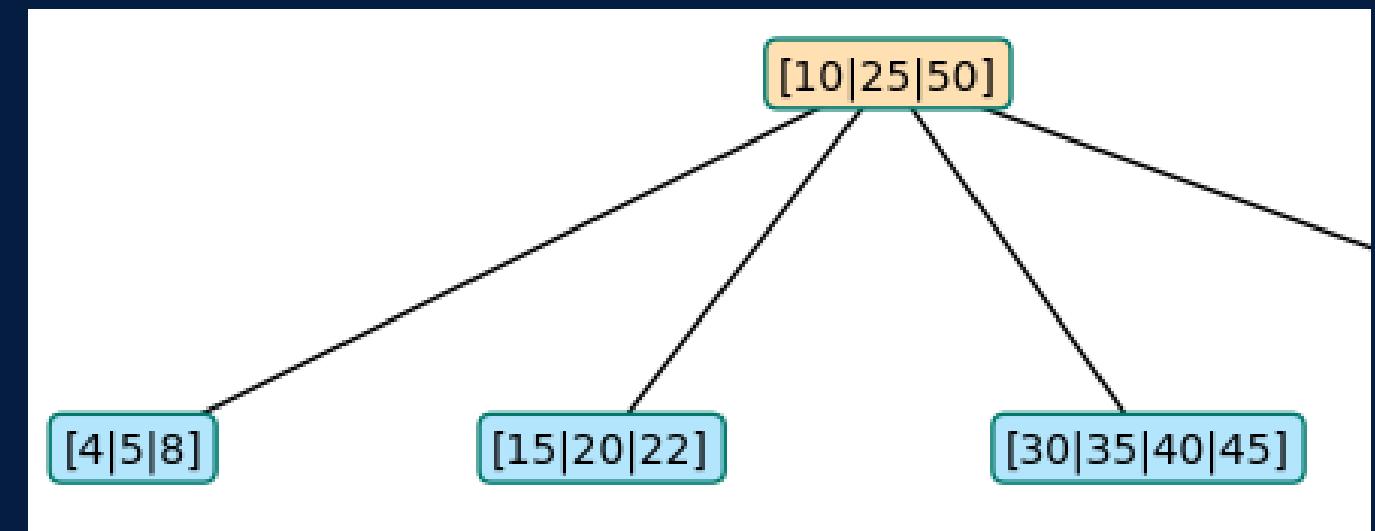
Lorsque le nœud devient plein, une division (éclatement en 2) est effectuée : la clé 50 remonte à la racine, et les autres clés sont réparties de gauche à droite entre les nœuds enfants.



## ► Exécution des exercices de la série 1 du TD :

Après l'insertion successive des clés restantes, le nœud gauche devient plein.

Une division classique est alors effectuée : la clé 10 remonte à la racine, ce qui donne une nouvelle racine [10, 25, 50] avec cinq sous-arbres équilibrés.



## ► Exécution des exercices de la série 1 du TD :

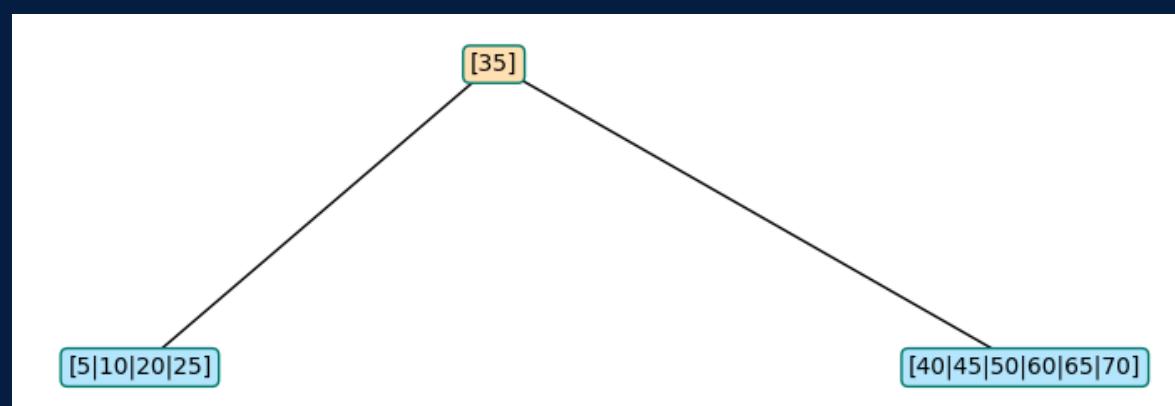
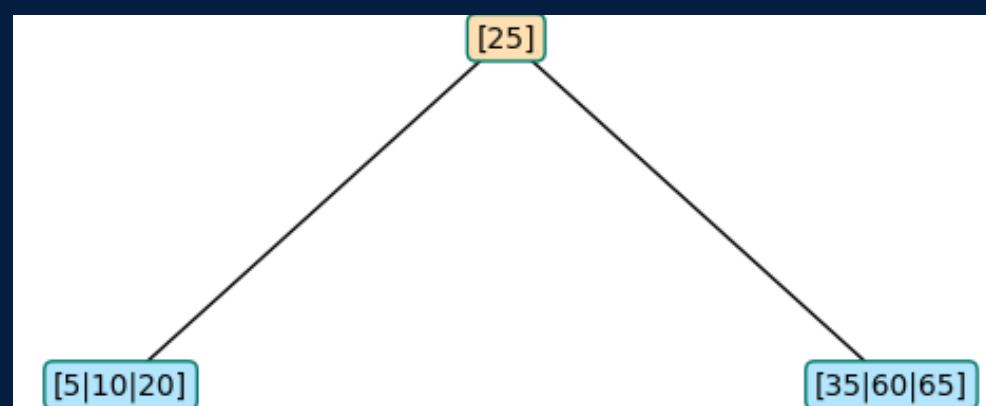
- B-arbre\* d'ordre 7 :

Après plusieurs insertions successives (5, 10, 20, 25, 35, 60, 65), le nœud racine devient plein.

Une division classique est effectuée avec la clé médiane 25, qui remonte pour devenir la nouvelle racine.

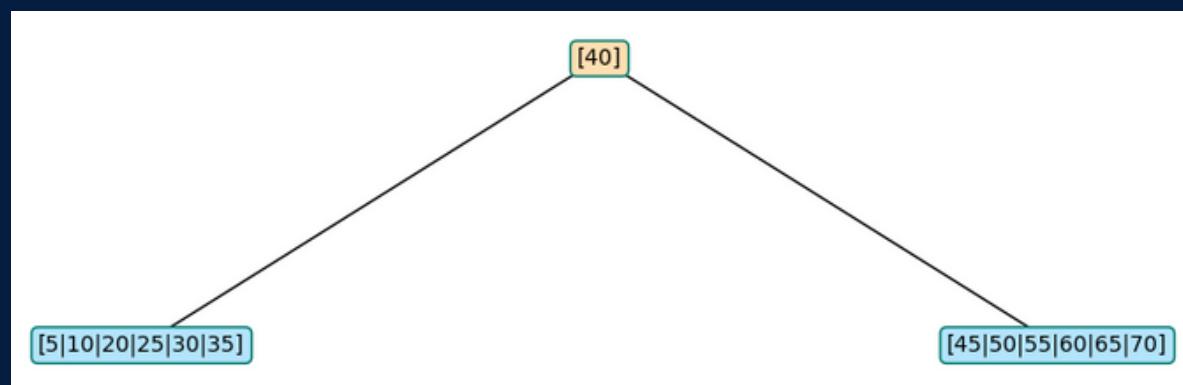
L'arbre est alors scindé en deux sous-arbres équilibrés

Après l'insertion de plusieurs clés supplémentaires (70, puis 40, 50), le nœud de droite devient surchargé. Une redistribution 2/3 vers la gauche

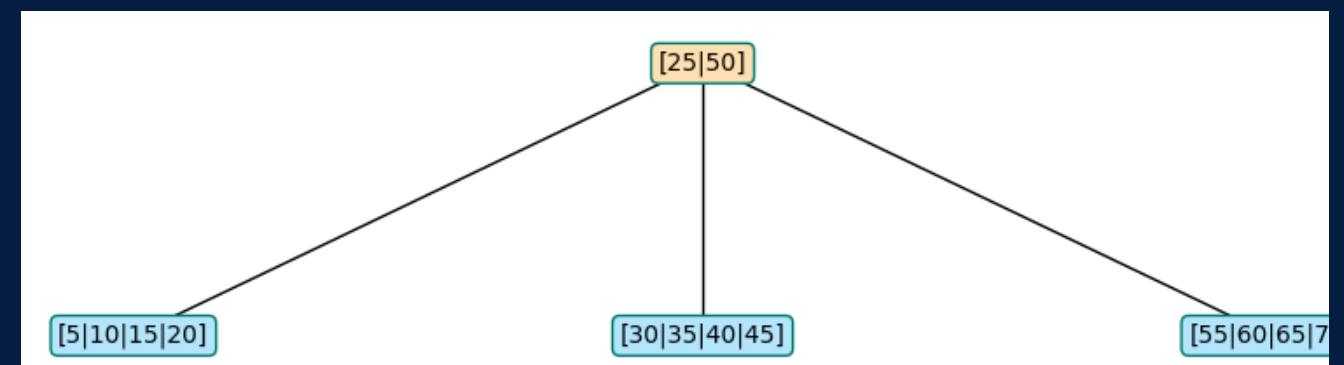


## ► Exécution des exercices de la série 1 du TD :

Après l'insertion des nouvelles clés (50, puis 30), nous appliquons redistribution .



Après l'insertion de la clé 15 nous appliquons éclatement en 3.



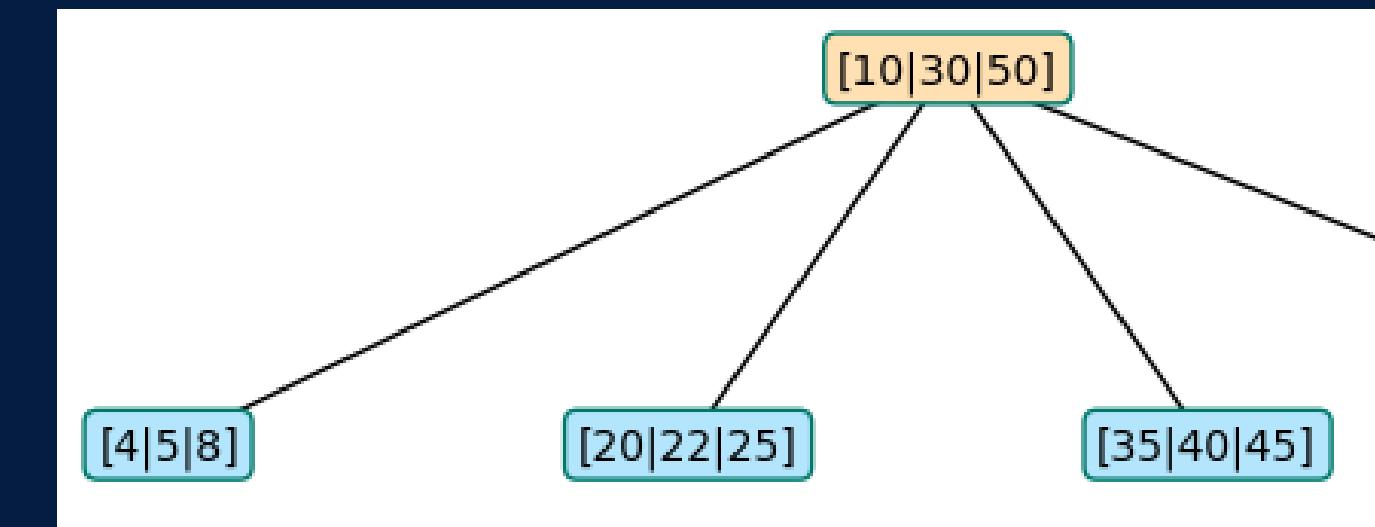
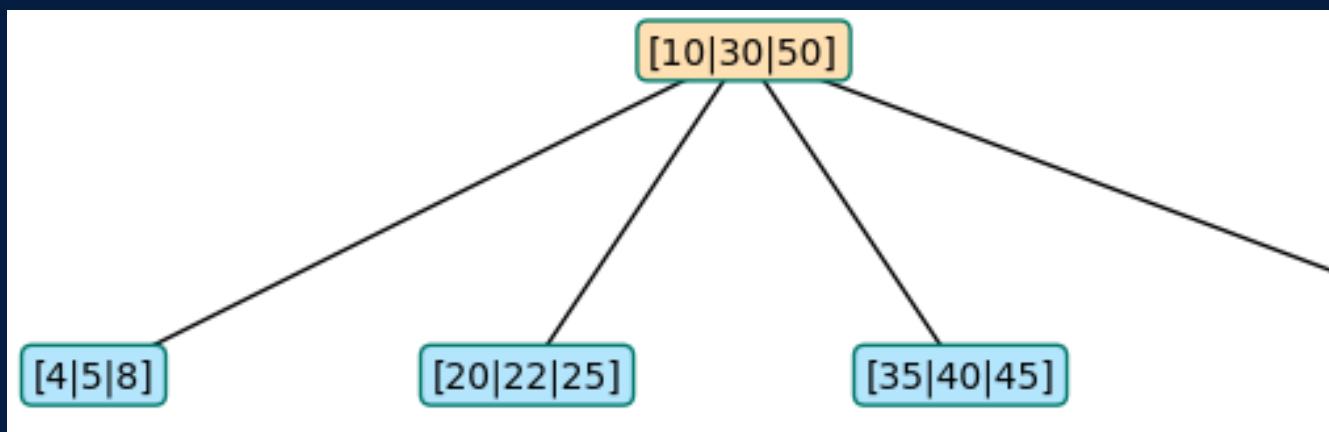
## ► Exécution des exercices de la série 1 du TD :

- Suppressions de l'arbre R : à partir des valeurs  $L2 = [15, 70, 50, 35, 60, 25]$

- B-arbre d'ordre 7 :

suppression de 15 : Action : suppression directe →  
la feuille devient [20, 22].

suppression de 70 :  
Action : suppression directe → [55, 60, 62, 64, 65].

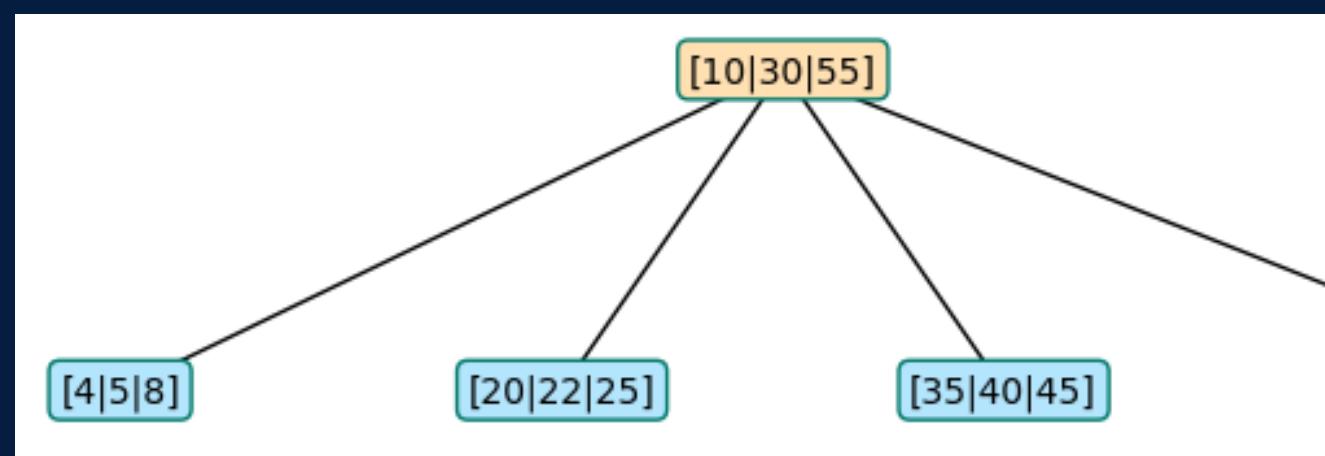


## ► Exécution des exercices de la série 1 du TD :

Suppression de 50 : Action : on remplace 50 par son successeur trouvé dans la feuille suivante [55, 60, 62, 64, 65], le successeur est 55, Ensuite, on supprime 50 de la feuille.

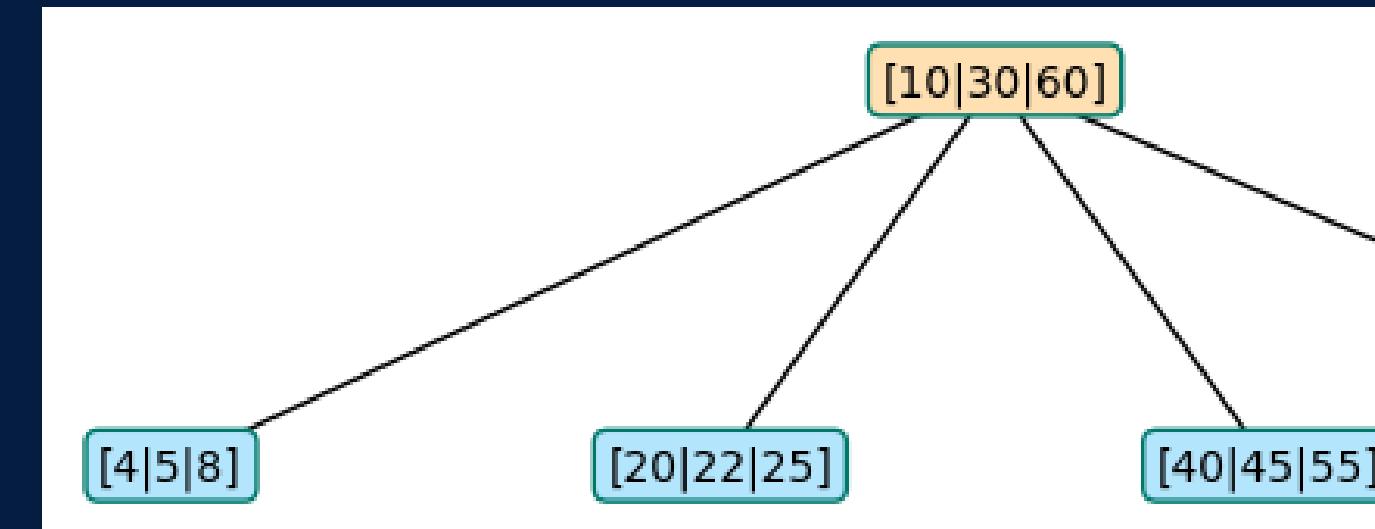
Résultat : pas de fusion ni d'emprunt.

Cas remplacement par le successeur au niveau d'une feuille



suppression de 35 :

Action : suppression directe → [40, 45].

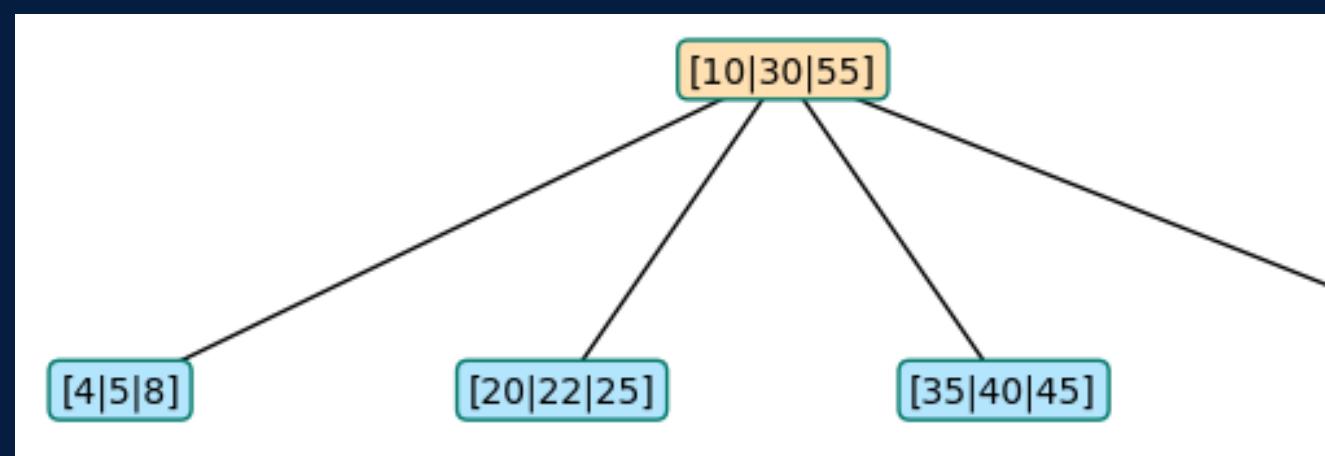


## ► Exécution des exercices de la série 1 du TD :

Suppression de 50 : Action : on remplace 50 par son successeur trouvé dans la feuille suivante [55, 60, 62, 64, 65], le successeur est 55, Ensuite, on supprime 50 de la feuille.

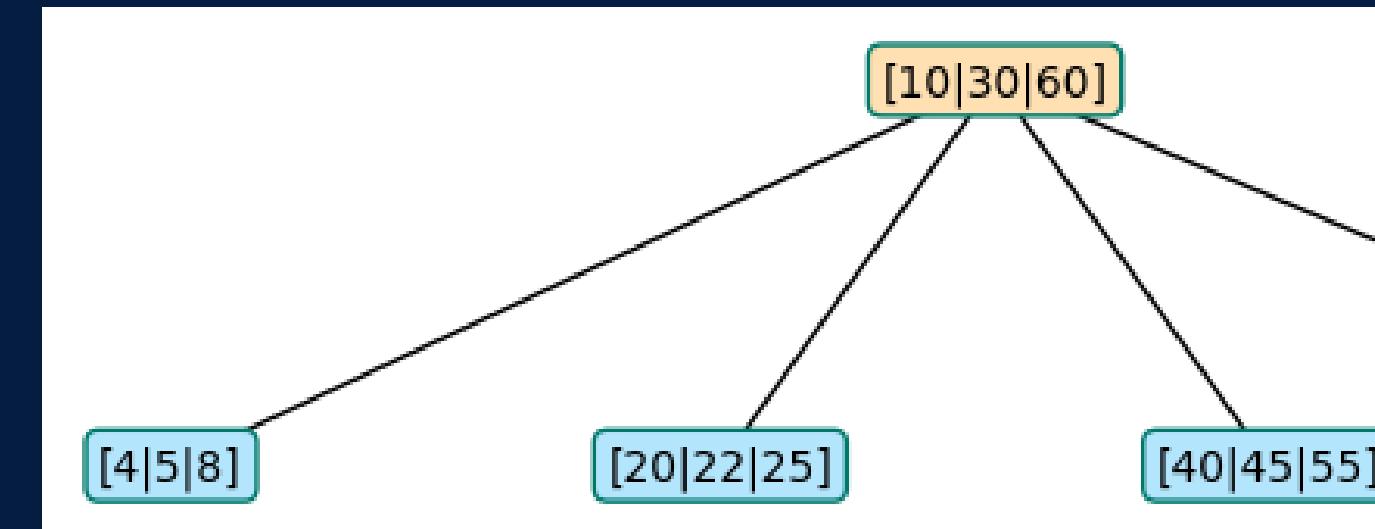
Résultat : pas de fusion ni d'emprunt.

Cas remplacement par le successeur au niveau d'une feuille



suppression de 35 :

Action : suppression directe → [40, 45].





## LES difficultés rencontrées

- Gestion correcte de la redistribution dans le B-arbre\* .
- Calcul automatique des positions pour l'affichage centré.



## Limites

- Pas d'animation (insertions instantanées)
- Redistribution simplifiée (2/3 exacte non visualisée)



## Points positifs de la solution

- Interface graphique intuitive.
- Visualisation claire et automatique.
- Code structuré, réutilisable et extensible.

MERCI  
POUR  
VOTRE  
ATTENTION

