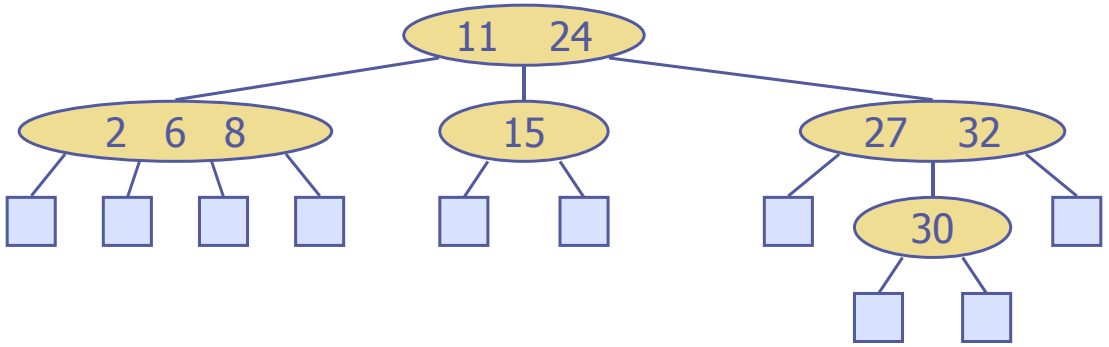
Arbres de recherche généralisés et Arbres (2,4)

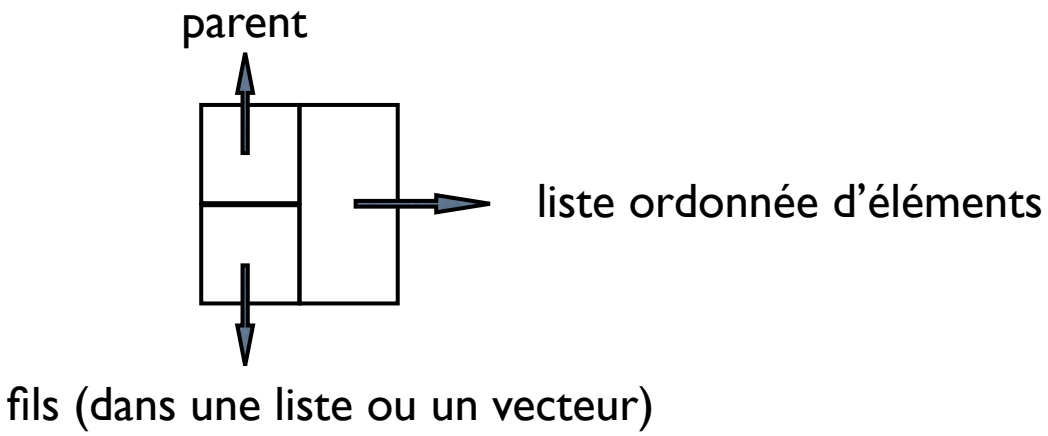
Arbres de recherche généralisés

- Un arbre de recherche généralisé est un arbre ordonné ayant les propriétés suivantes:
  - Chaque noeud interne a au moins deux enfants et garde en mémoire d-1 éléments (k_i, v_i) , où d est le nombre d'enfants
  - Pour chaque noeud interne gardant en mémoire les clés $k_1, k_2, \ldots, k_{d-1}$ et ayant pour enfants les noeuds $n_1, n_2, \ldots, n_d$ on a
    - les clés dans le sous-arbre de racine n_1 sont plus petites que k_1
    - les clés dans le sous-arbre de racine n_i sont plus petites que k_i et plus grande ou égale à k_{i-1} ($i = 2, \ldots, d-1$)
    - les clés dans le sous-arbre de racine n_d sont plus grandes ou égales à k_{d-1}



Implémentation (structure chaînée)

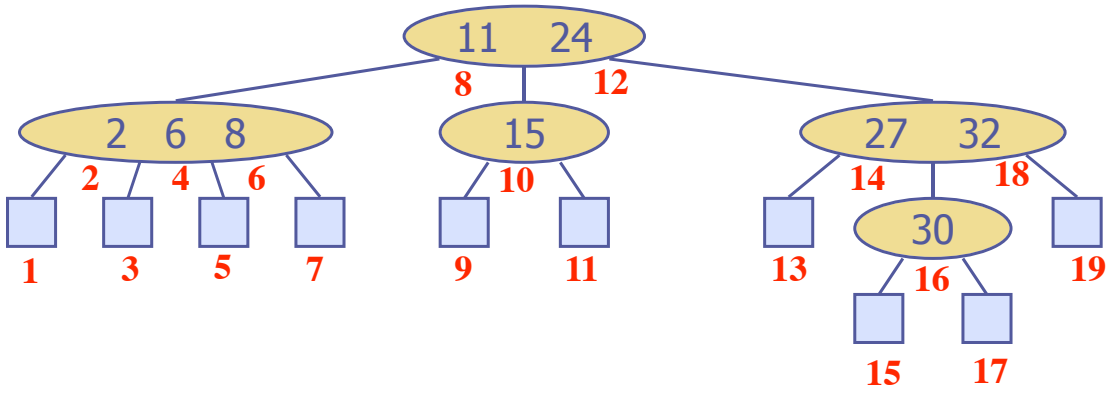
- Un noeud de la structure chaînée représentant le noeud d'un arbre de recherche généralisé est de cette forme



- Exemple au tableau

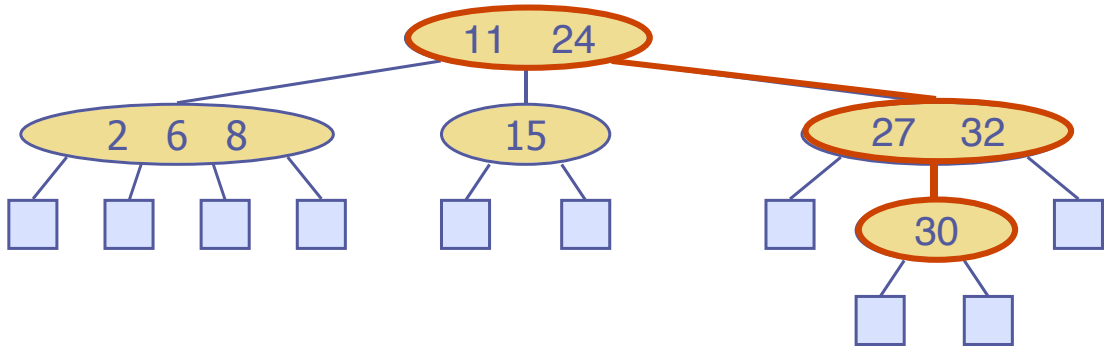
Parcours symétrique d'un arbre de recherche généralisé

- On peut généraliser la notion de parcours symétrique d'un arbre binaire aux arbres de recherche généralisés
  - On visite l'élément (k_i, v_i) d'un noeud n entre les visites récursives des sous-arbres de n de racines n_i et n_{i+1}
  - Un parcours symétrique d'un arbre de recherche généralisé visite les clés selon un ordre croissant



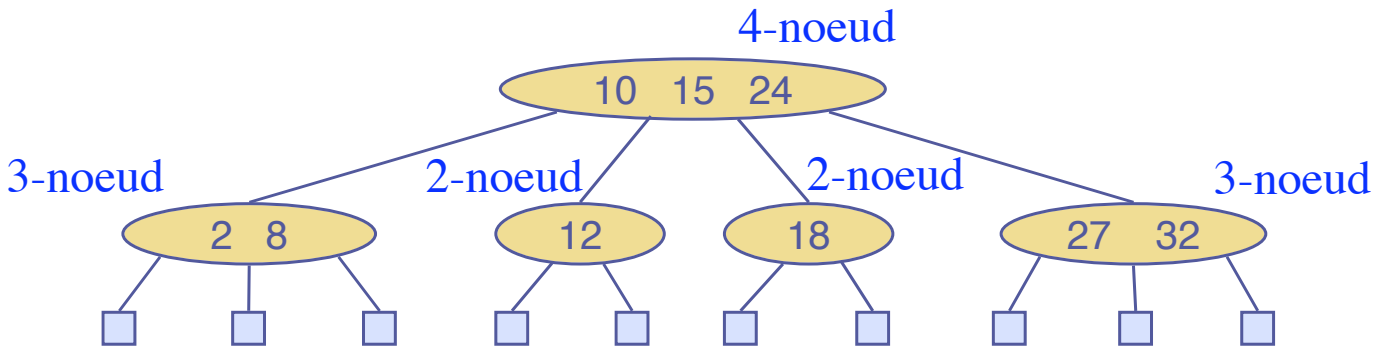
Chercher dans un arbre de recherche généralisé

- Similaire à la recherche dans un arbre binaire de recherche
- À chaque noeud interne d'enfants $n_1, n_2, \ldots, n_d$ et de clés $k_1, k_2, \ldots, k_{d-1}$
  - Si $k = k_i \ \ (i = 1, \ldots, d-1)$: la recherche se termine et on retourne v_i
  - Si $k < k_1$, on continue la recherche dans le fils n_1
  - Si $k_{i-1} \leq k < k_i \ (i = 2, \ldots, d-1)$, on continue la recherche dans le fils n_i
  - Si $k > k_{d-1}$, on continue la recherche dans le fils n_d
- Si on atteint un noeud externe, on retourne NULL
- Exemple: Chercher 30



Arbres (2,4)

- Un arbre (2,4) est un arbre de recherche généralisé ayant les propriétés suivantes:
  - Nombre d'enfants: tout noeud interne a au plus 4 enfants
  - Propriété de profondeur: tous les noeuds externes ont la même profondeur
- Donc, dépendant du nombre d'enfants, un noeud interne d'un arbre (2,4) est appelé un 2-noeud, un 3-noeud ou un 4-noeud.



Hauteur d'un arbre (2,4)

- La hauteur d'un arbre (2,4) gardant en mémoire n éléments est en O(log n)
  - Soit h, la hauteur d'un arbre (2,4) gardant en mémoire n éléments
  - Étant donné les propriétés des arbres (2,4), on a au moins 2^i éléments de profondeur $i = 0, \ldots, h-1$ et aucun élément de profondeur h, on a donc

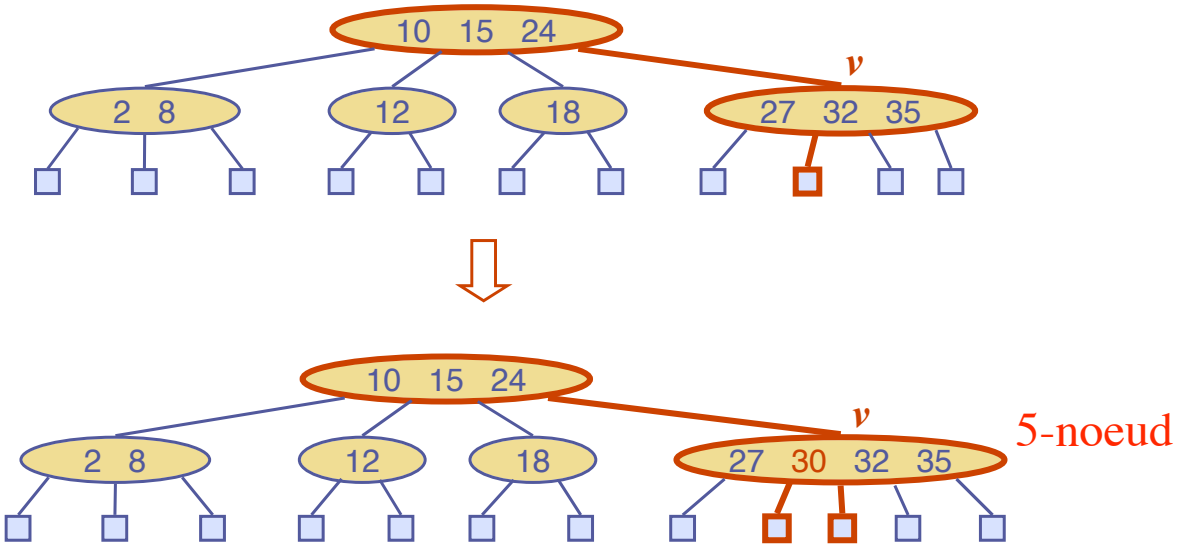
$$n \geq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$$

  - On a donc $h \leq \log(n+1)$

- La recherche dans un arbre (2,4) se fait donc en O(log n)

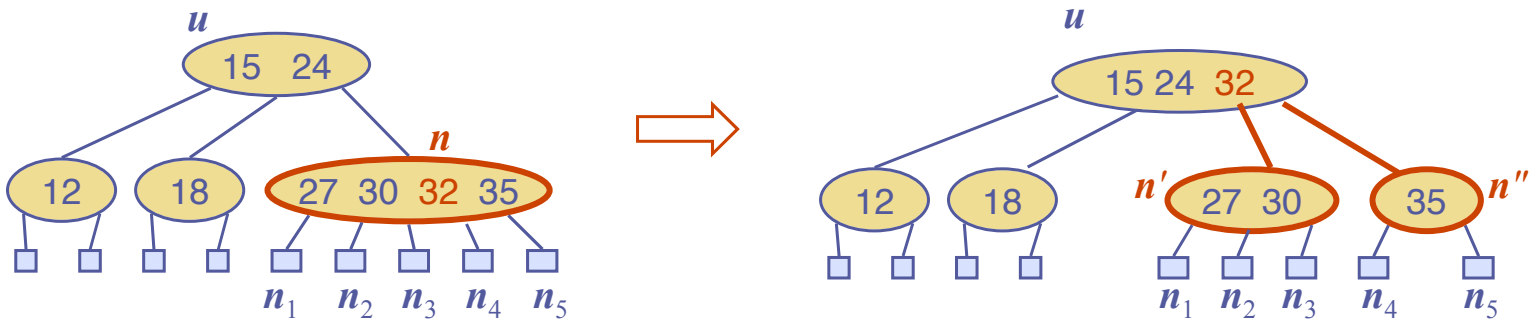
Insérer dans un arbre (2,4)

- On insère un nouvel élément (k,v) dans le parent n de la feuille atteinte lors de notre recherche pour k
  - La propriété de profondeur est préservée
  - Insérer k dans n peut causer un débordement (i.e que le noeud n devient un 5-noeud)
- Exemple: Insérer 30 cause un débordement



Débordement et fractionnement

- On "répare" le débordement d'un 5-noeud n avec une opération de fractionnement:
  - Soit $n_1, \ldots, n_5$ les enfants de n, et $k_1, \ldots, k_4$ les clés de n
  - Le noeud n est remplacé par deux noeuds n' et n''
    - n' est un 3-noeud de clés k_1, k_2 et d'enfants n_1, n_2, n_3
    - n'' est 2-noeud de clé k_4 et d'enfants n_4, n_5
  - La clé k_3 est insérée dans le parent u du noeud n (une nouvelle racine peut être créée)
- Le débordement peut se propager dans u



Complexité en temps d'une insertion

Algorithme *insérer ((k,v), T)*

Étape 1. On exécute l'algorithme chercher(***k***), jusqu'à ce qu'on atteigne une feuille. Soit ***n*** le parent de cette feuille.

Étape 2. On met (***k***, v) dans ***n***

Étape 3. **Tant que** ***Déborde***(***n***)

si ***estRacine***(***n***)

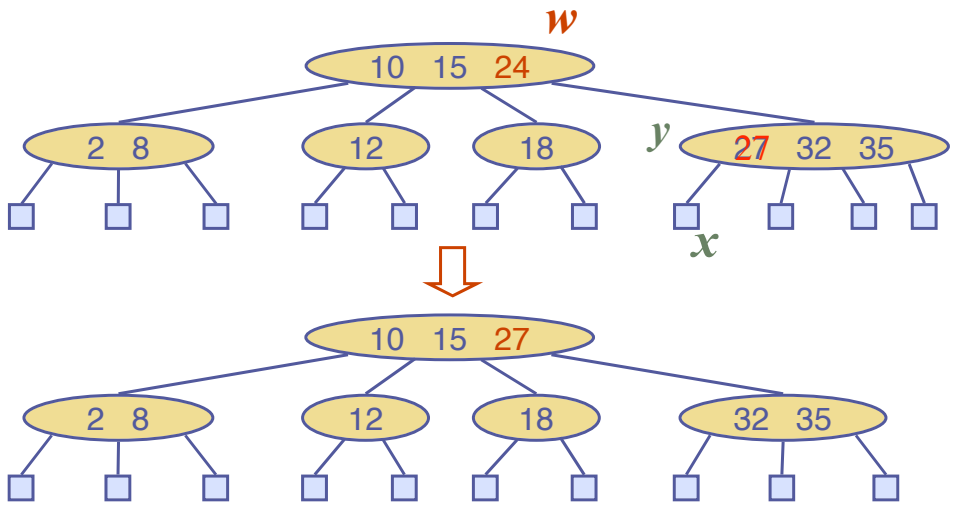
créer une nouvelle racine au-dessus de ***n***

n ← ***fractionne***(***n***)

- Soit T un arbre (2,4) contenant n éléments:
  - l'arbre T a une hauteur en $O(\log n)$
  - et donc, l'étape 1 prend un temps $O(\log n)$
  - l'étape 2 prend un temps $O(1)$
  - l'étape 3 prend un temps $O(\log n)$ étant donné que chaque fractionnement se fait en temps $O(1)$ et qu'on doit exécuter dans le pire des cas $O(\log n)$ fractionnements
- Insérer un élément dans un arbre (2,4) prend un temps $O(\log n)$

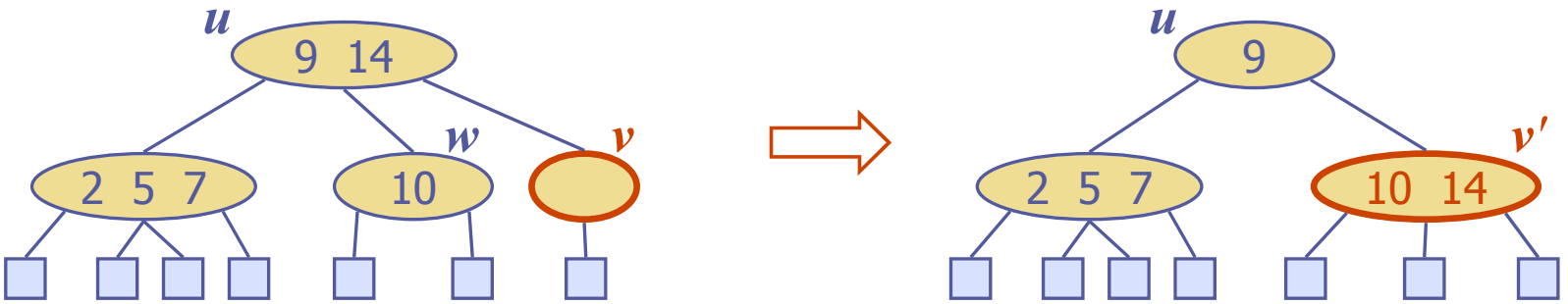
Supprimer dans un arbre (2,4)

- Si le noeud interne w contenant l'élément à enlever a au moins un enfant qui est une feuille, on enlève cette feuille et w et on retourne la valeur de l'élément.
- Si tous les enfants de w sont internes, on trouve le noeud y contenant le successeur, selon l'ordre symétrique, de la clé à enlever. On remplace l'élément à enlever par ce successeur et on enlève une feuille de y. On retourne la valeur de l'élément enlevé.
- Exemple: enlever un élément de clé 24



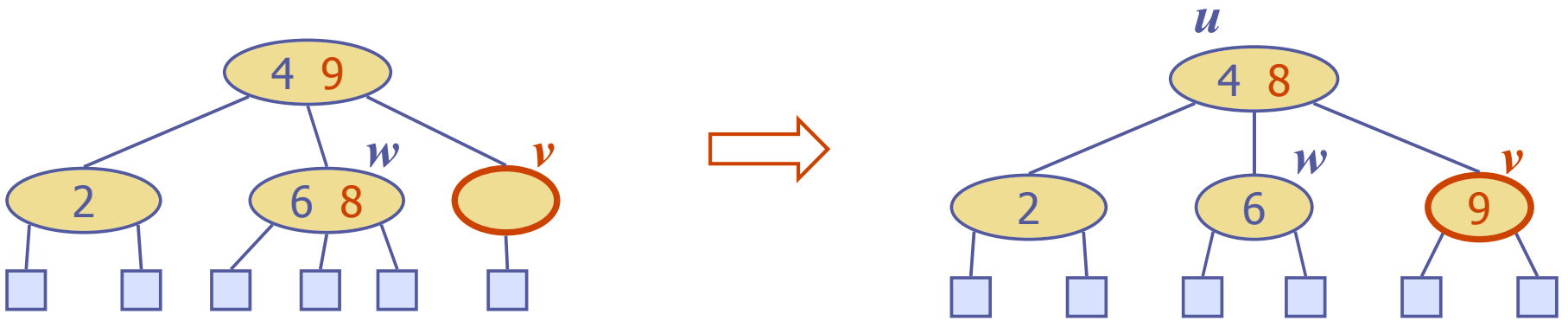
Noeud interne vide et fusion

- Enlever une clé d'un noeud interne v peut rendre ce noeud vide, i.e que v devient un 1-noeud (noeud avec un enfant et aucune clé)
- Pour remédier à cette situation, où un noeud interne v de parent u devient vide, on considère deux cas:
- Premier cas: le noeud (frère) adjacant à v est un 2-noeud w
  - Opération de fusion: on fusionne v et w en un noeud v', et on bouge un élément du parent u dans v'
  - Après la fusion, il est possible que le parent u devienne vide



Noeud interne vide et transfert

- Deuxième cas: le noeud (frère) adjacant à v est un 3-noeud ou un 4-noeud w
  - Opération de transfert:
    1) on transfert un enfant de w à v
    2) on transfert un élément de u à v
    3) on transfert un élément de w à u
  - Après un transfert, aucun autre noeud devient vide



Complexité en temps d'une suppression

- Soit T un arbre (2,4) contenant n éléments
  - l'arbre T a une hauteur en $O(\log n)$
- Lorsqu'on exécute une opération de suppression
  - On visite $O(\log n)$ noeuds pour trouver le noeud dans lequel on va supprimer un élément
  - Si un noeud interne devient vide après la suppression, on exécute au plus une série de $O(\log n)$ fusions et ensuite au plus un transfert
  - Chaque fusion et transfert s'exécute en $O(1)$
- Donc, supprimer un élément dans un arbre (2,4) prend un temps $O(\log n)$