ALGORITHMES ET STRUCTURES DE DONNÉES

IFT-2008/GLO-2100

Chapitre 7 :Les structures arborescentes et les monceaux

Thierry Eude, Kim Rioux-Paradis

Département d'informatique et de génie logiciel

UNIVERSITÉ LAVAL

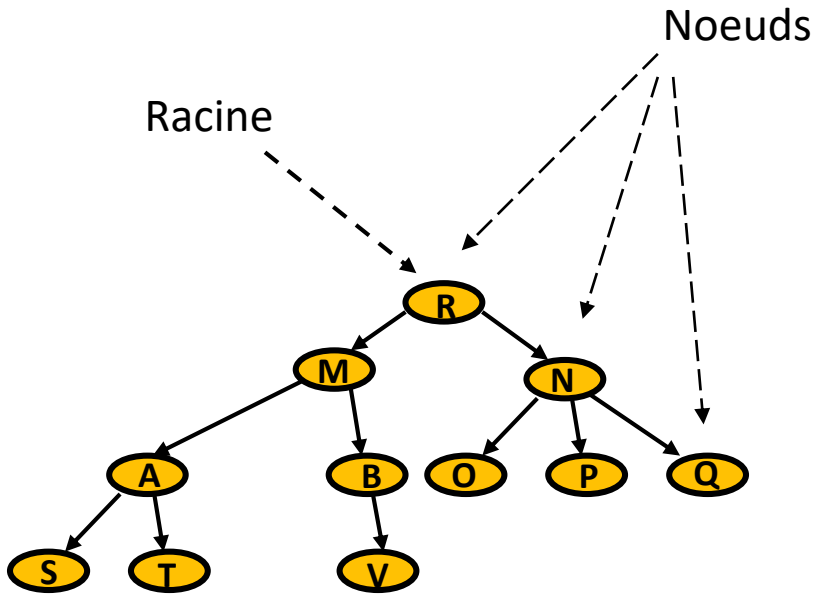
Plan

- Terminologie des arbres
- Parcours d'arbres
  - pré-ordre, post-ordre, et en-ordre
- Implémentation des arbres dans un tableau
  - l'implémentation par chaînage sera vu au chapitre suivant
- Monceaux (tas)
  - tri par tas («heap sort»)

Qu’est-ce qu’un arbre?

- Définition récursive: Un arbre est un noeud racine pointant sur des arbres (qui, eux-mêmes, sont des nœuds racine pointant vers d'autres arbres).

- Typiquement, chaque nœud possède une information sous la forme d’une clé ou d’une paire (clé, valeur).



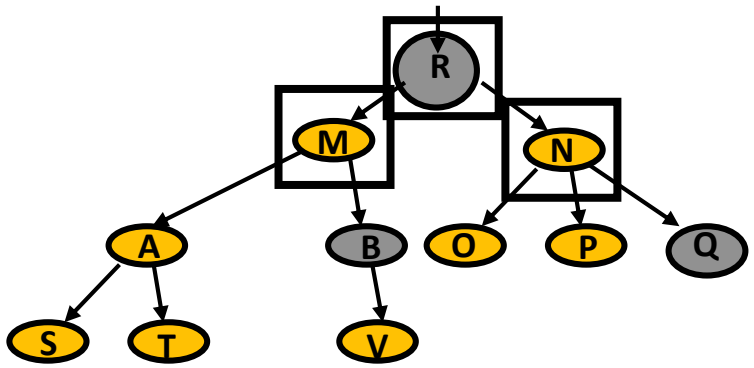
Terminologie des arbres

- **Parent d'un nœud** : Le nœud immédiatement prédécesseur.

Parent(B) = M

Parent(R) = Nil

Parent(Q) = N



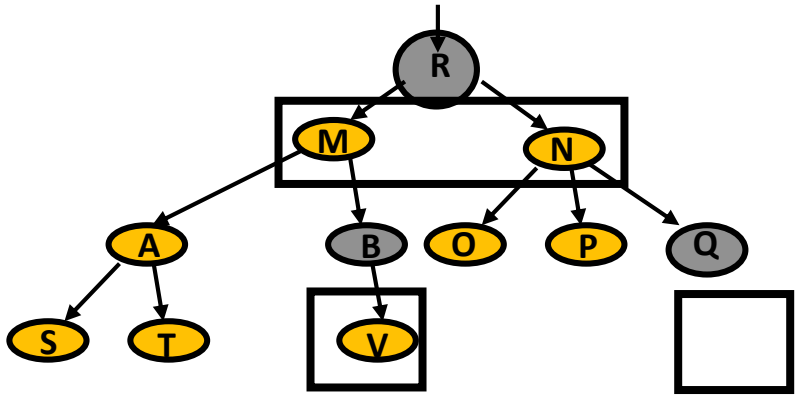
Terminologie des arbres

- Parent d'un nœud : Le nœud immédiatement prédécesseur.
- **Enfants d'un noeud** : Les nœuds immédiatement successeurs du nœud.

Enfants(B) = {V}

Enfants(R) = {M,N}

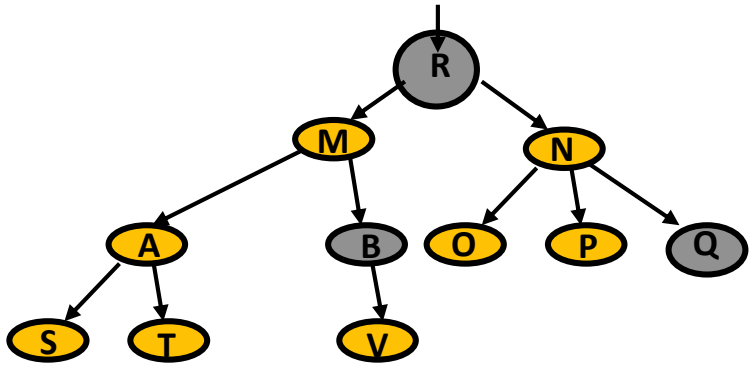
Enfants(Q) = {}



Terminologie des arbres

- Parent d'un nœud : Le nœud immédiatement prédécesseur.
- Enfants d'un noeud : Les nœuds immédiatement successeurs du nœud.
- **Racine** : Le nœud qui n'a pas de prédécesseur.

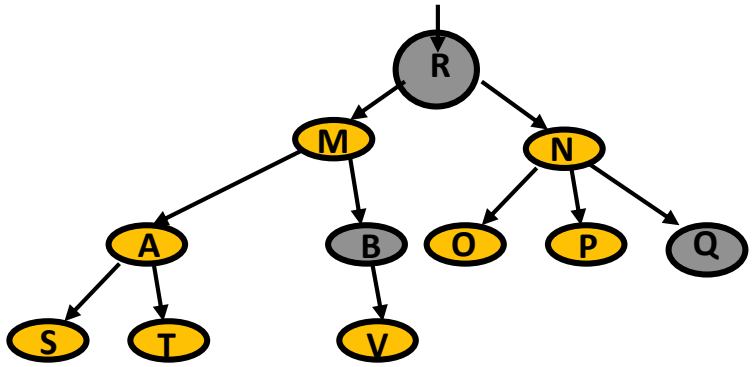
Racine(Arbre) = R



Terminologie des arbres

- Parent d'un nœud : Le nœud immédiatement prédécesseur.
- Enfants d'un noeud : Les nœuds immédiatement successeurs du nœud.
- Racine : Le nœud qui n'a pas de prédécesseur.
- **Feuille** : Un nœud qui n'a pas d'enfants.

Feuilles(Arbre) = {S,T,V,O,P,Q}



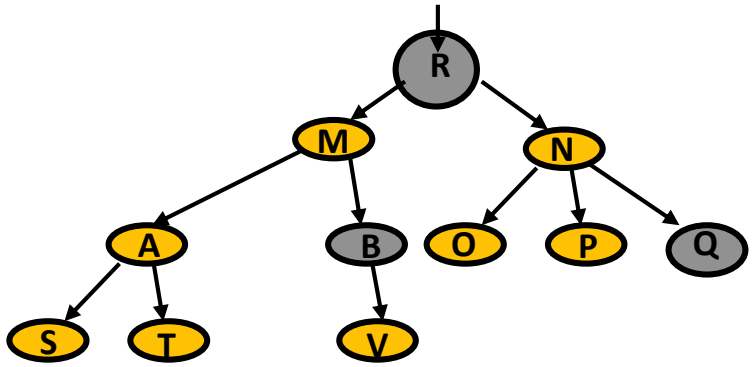
Terminologie des arbres

- Parent d'un nœud : Le nœud immédiatement prédécesseur.
- Enfants d'un noeud : Les nœuds immédiatement successeurs du nœud.
- Racine : Le nœud qui n'a pas de prédécesseur.
- Feuille : Un nœud qui n'a pas d'enfants.
- **Ancêtres d'un nœud** : Tous les nœuds prédécesseurs jusqu'à la racine.

Ancêtres(B) = {M,R}

Ancêtres(R) = {}

Ancêtres(Q) = {N,R}



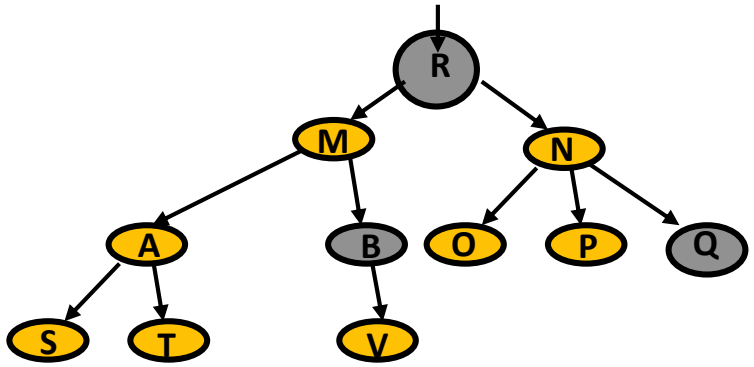
Terminologie des arbres

- **Descendants d'un nœud** : Tous les nœuds successeurs jusqu'aux feuilles accessibles par ce nœud.

Descendants(B) = {V}

Descendants(R) = {M,N,…Q,S,T,V}

Descendants(Q) = {}



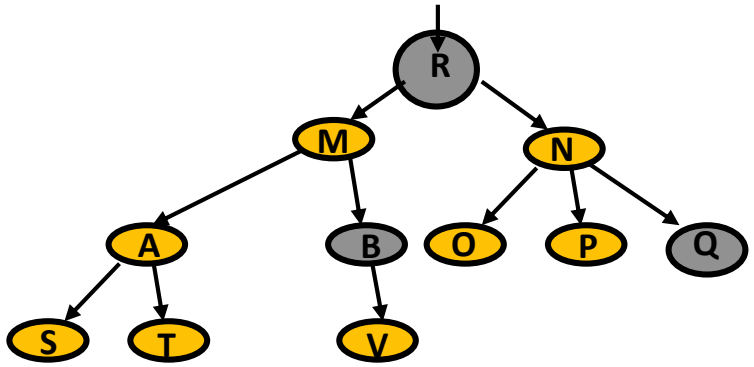
Terminologie des arbres

- Descendants d'un nœud : Tous les nœuds successeurs jusqu'aux feuilles accessibles par ce nœud.
- **Hauteur d'un nœud** : Longueur du chemin le plus long pour atteindre une feuille.
- **Hauteur de l'arbre** : La hauteur du noeud racine.

Hauteur(B) = 1

Hauteur(R) = 3 (Hauteur de l'arbre)

Hauteur(Q) = 0



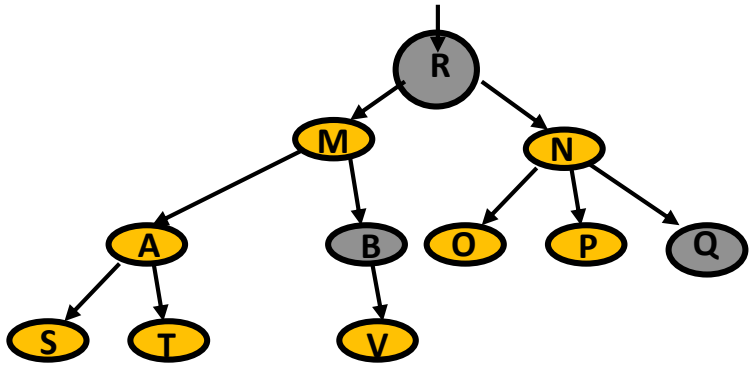
Terminologie des arbres

- Descendants d'un nœud : Tous les nœuds successeurs jusqu'aux feuilles accessibles par ce nœud.
- Hauteur d'un nœud : Longueur du chemin le plus long pour atteindre une feuille.
- Hauteur de l'arbre : La hauteur du noeud racine.
- **Niveau ou profondeur d'un nœud** : Longueur du chemin à partir de la racine.

Niveau(B) = 2

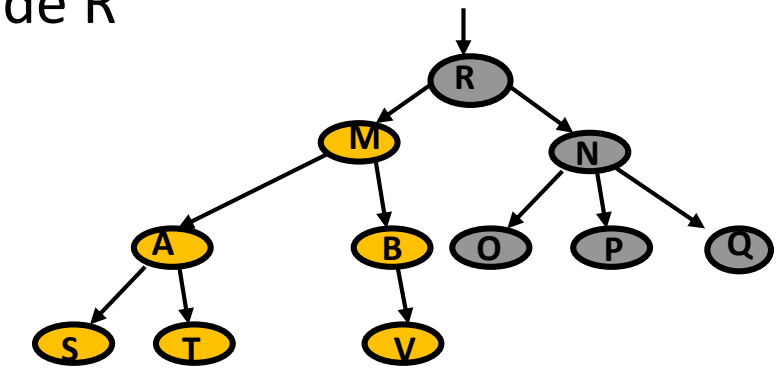
Niveau(R) = 0

Niveau(Q) = 2



Terminologie des arbres

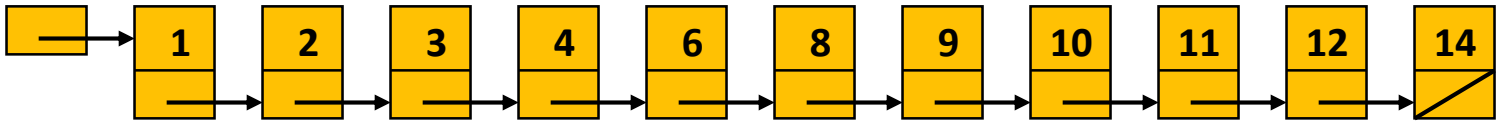
- **Sous-arbre de racine M**
  - arbre constitué de M et de ses descendants.
  - sous-arbre aussi appelé le sous-arbre gauche de R



- **Degré d'un nœud** : nombre d'enfants que possède ce nœud.
- **Degré d'un arbre** : degré le plus élevé de ses noeuds.

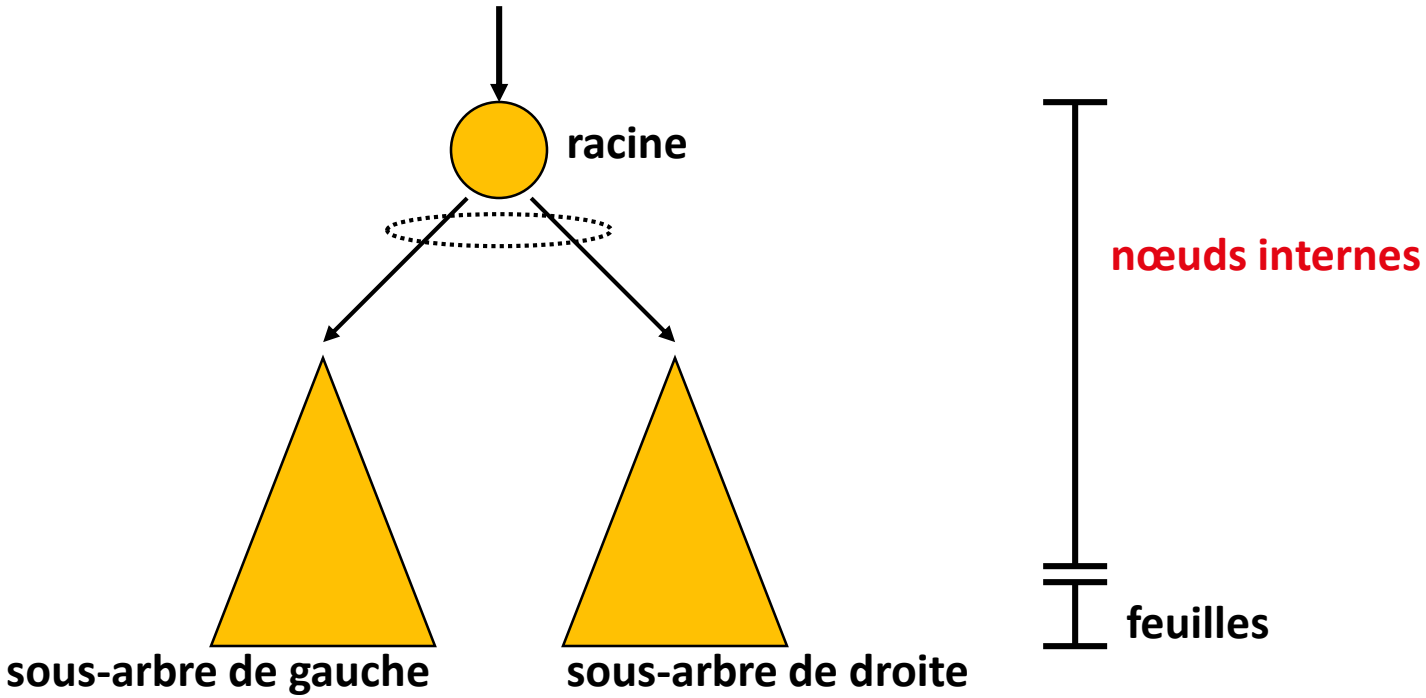
Liste simplement chaînée

- Une liste simplement chaînée est un arbre de degré 1.
  - arbre dégénéré



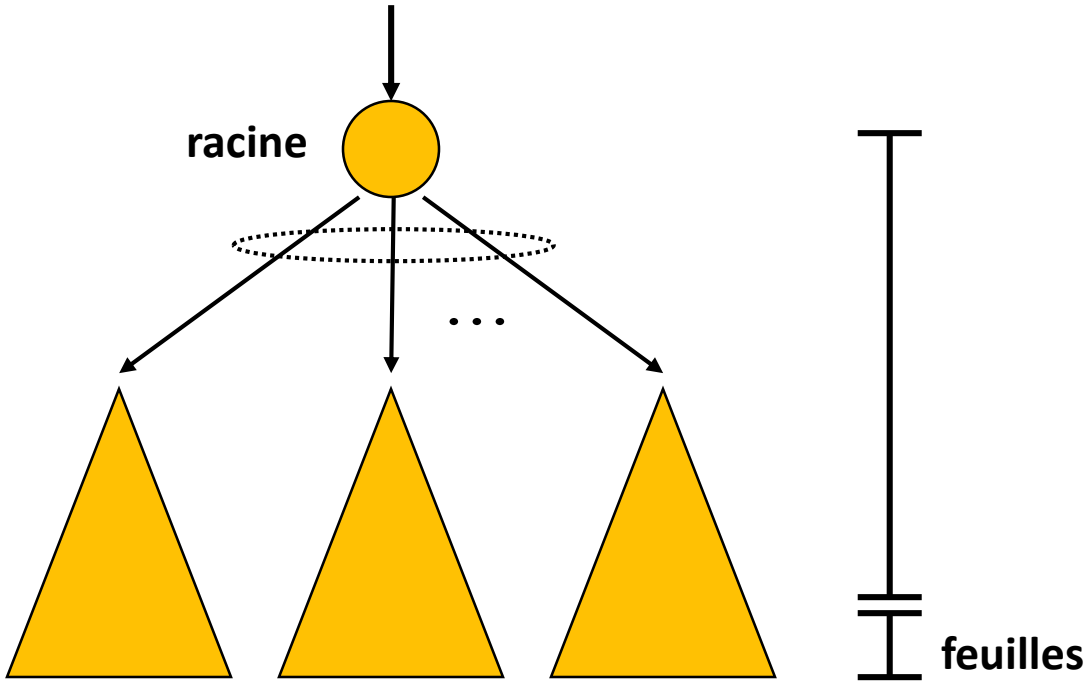
Arbre **binaire**

arbre de **degré 2**



Arbre **n-aire**

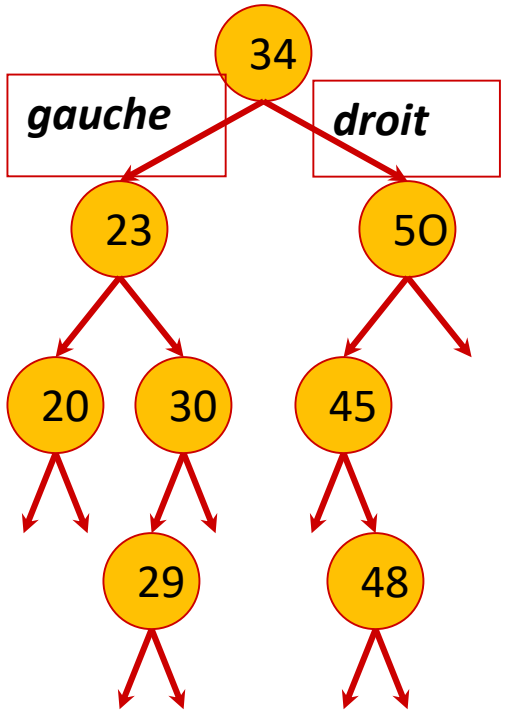
arbre de **degré n**



Les arbres binaires de recherche

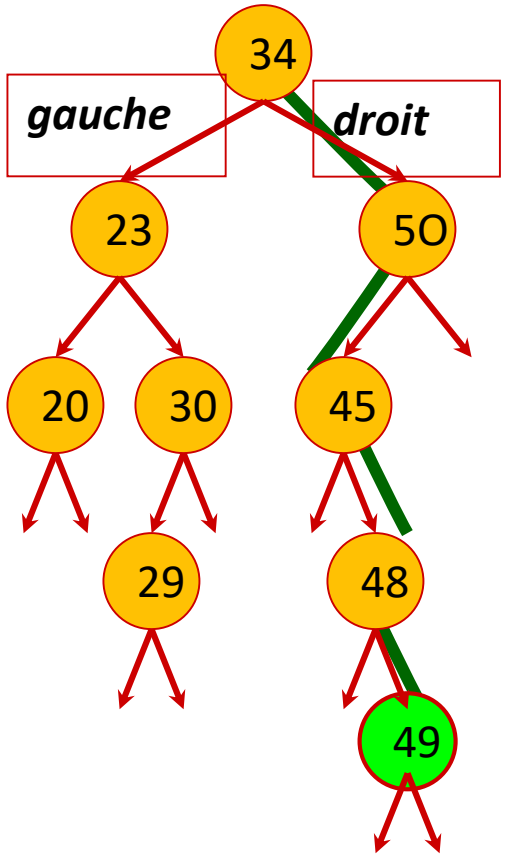
- Si chaque nœud possède une clé dont les valeurs satisfont la propriété suivante:
  - Pour tout noeud de l'arbre la valeur de sa clé est:
    - ✓ > aux clés de tous les noeuds de son sous-arbre gauche.
    - ✓ < aux clés de tous les nœuds de son sous-arbre droit.
  - Toutes les clés doivent donc être distinctes.
  - Tout ajout ou suppression de nœud doit donc maintenir cette propriété vraie.

Les arbres binaires de recherche



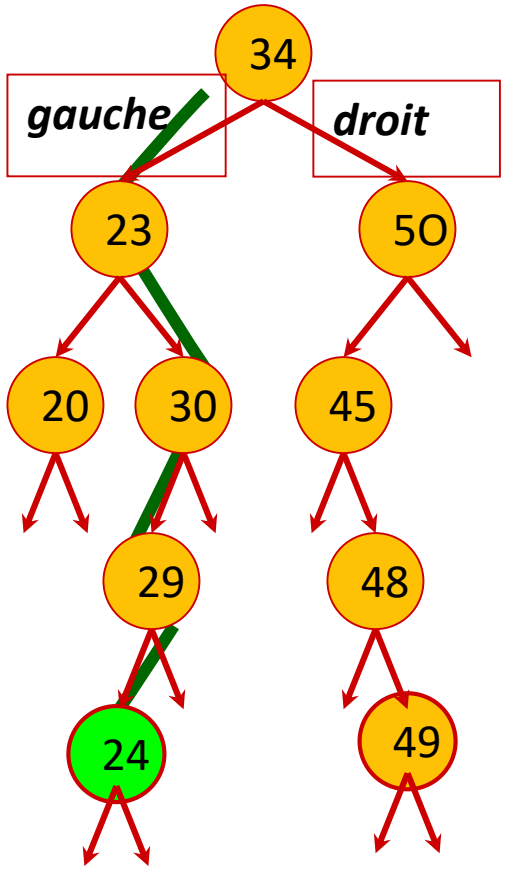
Les arbres binaires de recherche

Remarque :
Tout ajout se fait par une feuille.
Ajout de la valeur **49**



Les arbres binaires de recherche

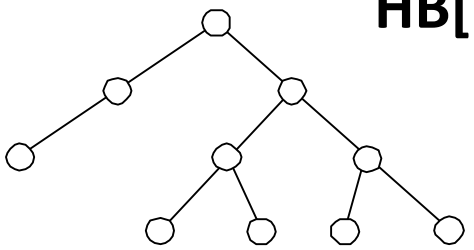
Remarque :
Tout ajout se fait par une feuille.
Ajout de la valeur **24**



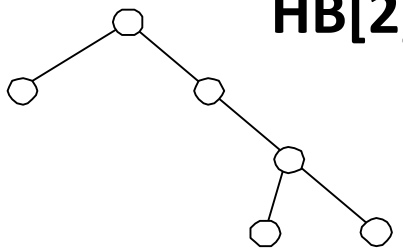
Le facteur d'équilibre

- Un arbre binaire est dit **équilibré** lorsque, pour tout nœud, la valeur absolue de la différence entre les hauteurs de ses sous-arbres gauche et droit est ≤ 1.
- **Facteur d'équilibre** (Height-Balanced, HB[k]) :
  - ➢ valeur maximale de cette différence de hauteur parmi tous les nœuds de l'arbre.

HB[1]



HB[2]

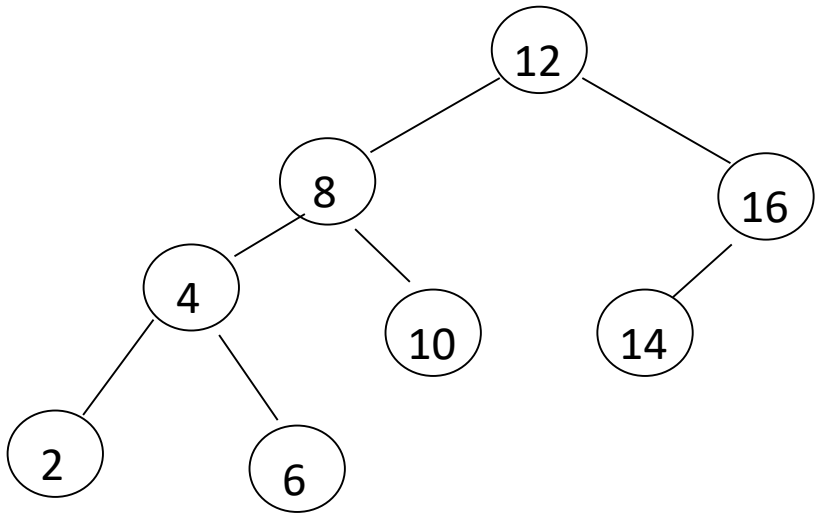


Le facteur d'équilibre

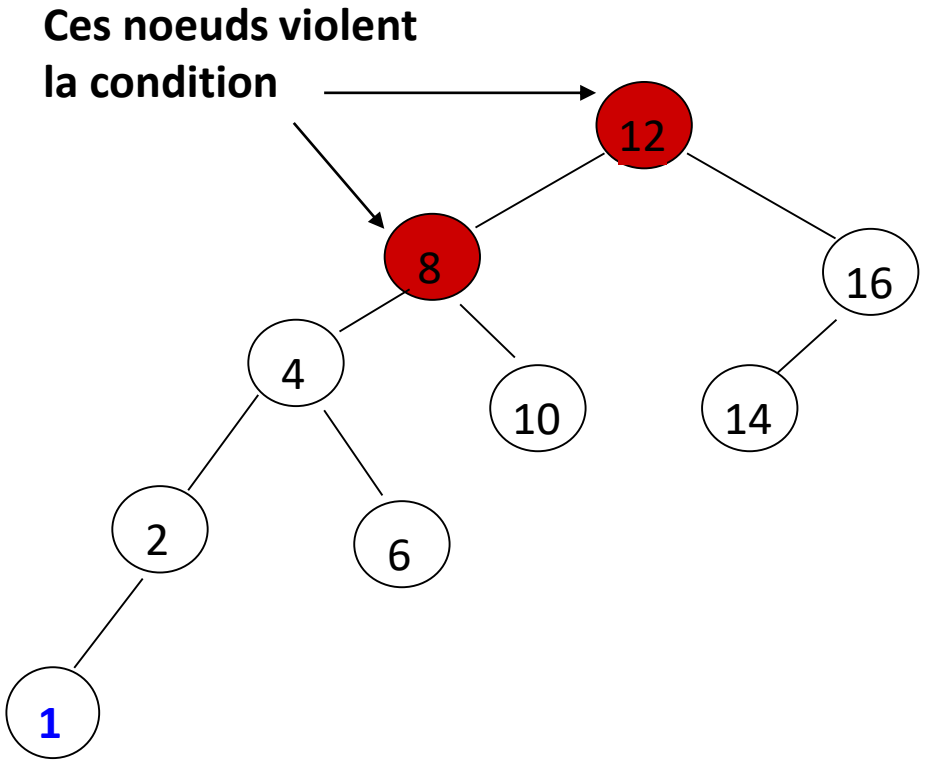
- Arbre AVL (Adelson–Velsky et Landis, 1962)
  - Arbre HB[1] maintenu à l'équilibre grâce à des opérations de rotations effectuées lors des insertions et suppressions (chapitre suivant).

- Un arbre équilibré HB[1] de n noeuds possède une hauteur en $O(Log\ n)$.
  - recherche d'un élément : en temps $O(Log\ n)$.
  - même complexité que la recherche dichotomique dans un tableau trié (voir chapitre 1).

Arbres AVL: exemple



Un arbre AVL



Après l'ajout de 1, ce n'est plus un arbre AVL

Synthèse

- Définitions
  - Récursive
  - Parent d’un nœud
  - Enfant(s) d’un nœud
  - Racine
  - Feuille
  - Ancêtres d’un nœud
  - Descendants d’un nœud
  - Hauteur d’un nœud
  - Hauteur de l’arbre
  - Niveau ou profondeur d’un nœud
  - sous-arbre
  - Degré d’un nœud
  - degré d'un arbre
- Arbre dégénéré
- Arbre binaire
- Arbre n-aire
- Arbres de recherche
- Facteur d’équilibre HB[]
- Arbre AVL

Parcours d'arbre

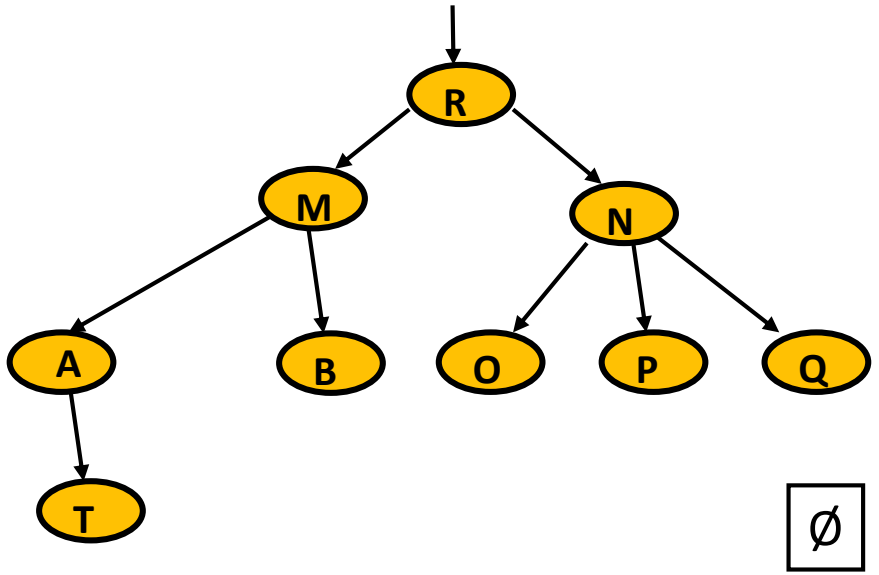
- Fréquemment, nous devons parcourir (ou visiter) tous les nœuds d'un arbre. Typiquement, lorsqu'un nœud est visité:
  - On affiche une clé, une valeur, ou un mot
  - On met à jour une valeur, un objet, etc…
- La méthode de parcours utilisée nous définit un itérateur pour l'arbre
  - définit l'ordre dans lequel seront visités les nœuds.
- On peut utiliser les parcours en profondeur ou en largeur définis pour les graphes (aussi définis pour les arbres).

Parcours d’arbre

- Habituellement trois ordres de visite (définis uniquement pour les arbres):
  - En pré-ordre (donne la même chose qu’un PP ou DFS)
  - En post-ordre
  - En-ordre (symétrique)

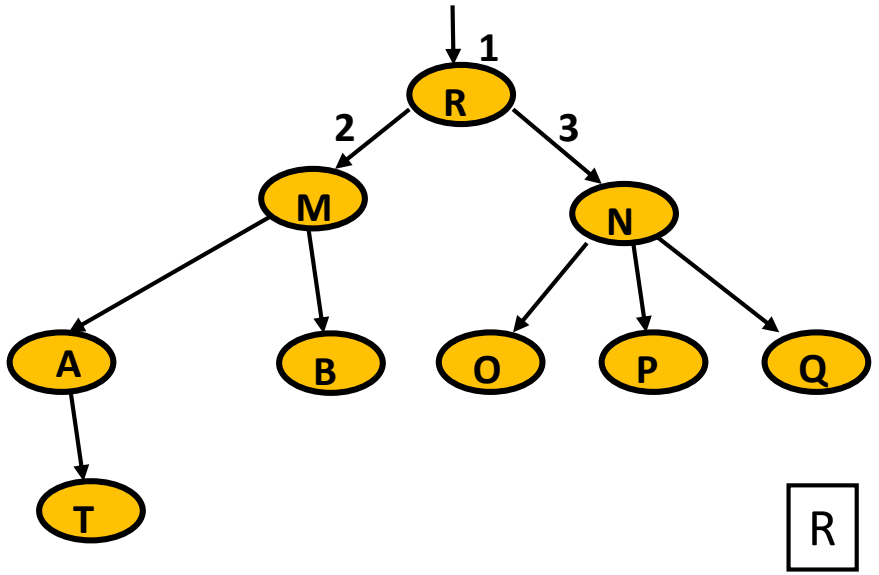
Itérateurs d'arbre

- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - ➢ Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



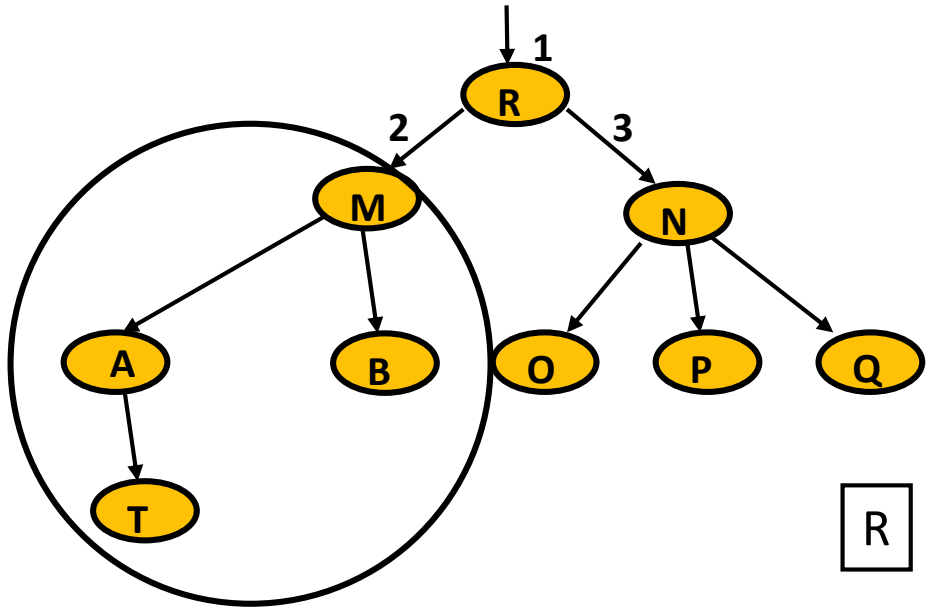
Itérateurs d'arbre

- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



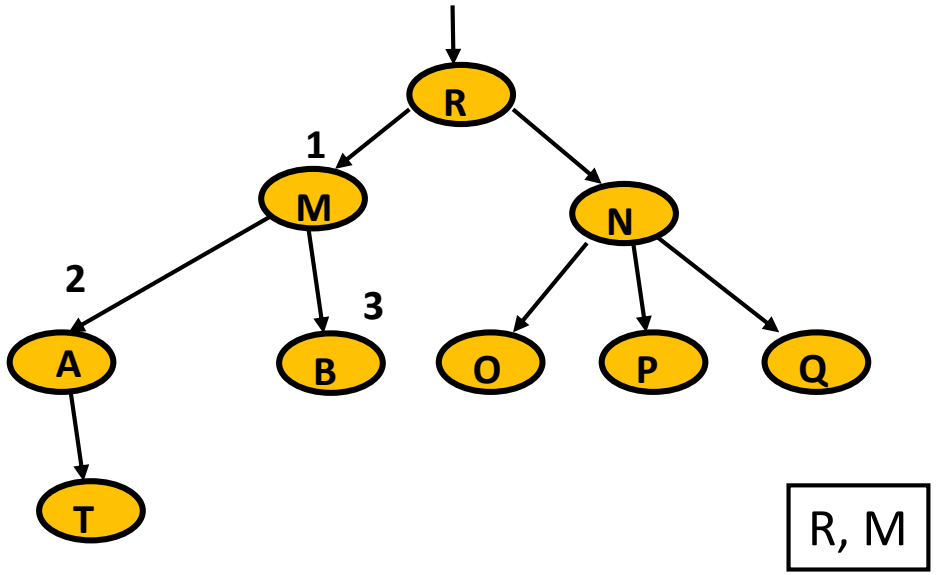
Itérateurs d'arbre

- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



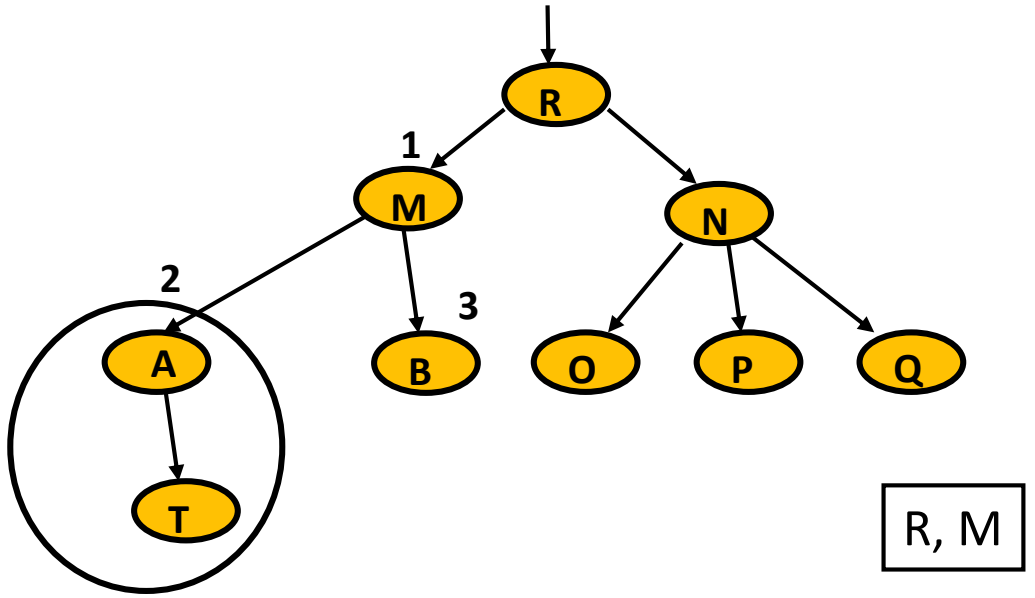
Itérateurs d'arbre

- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - ➢ Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



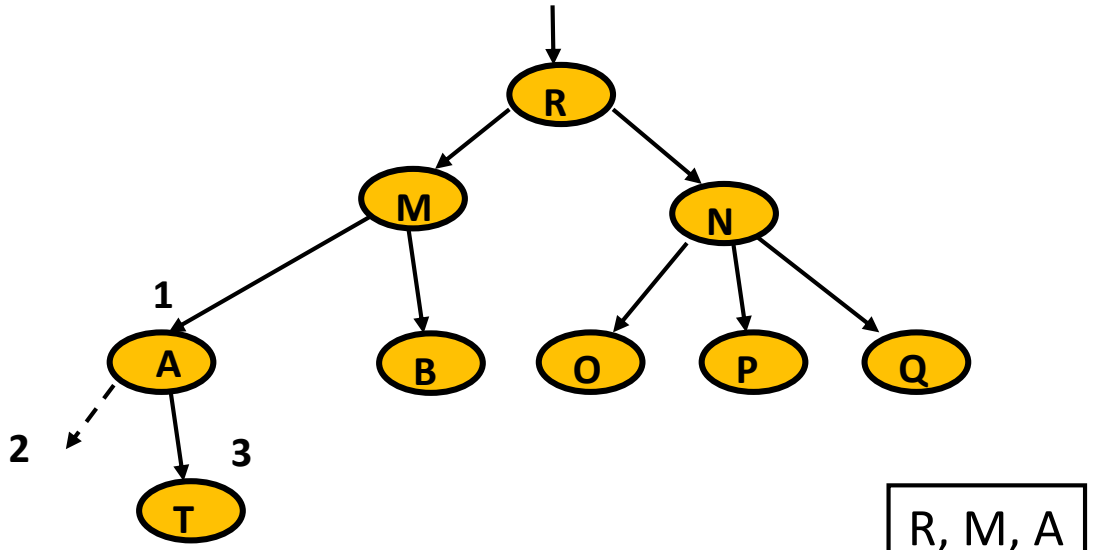
Itérateurs d'arbre

- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - ➢ Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



Itérateurs d'arbre

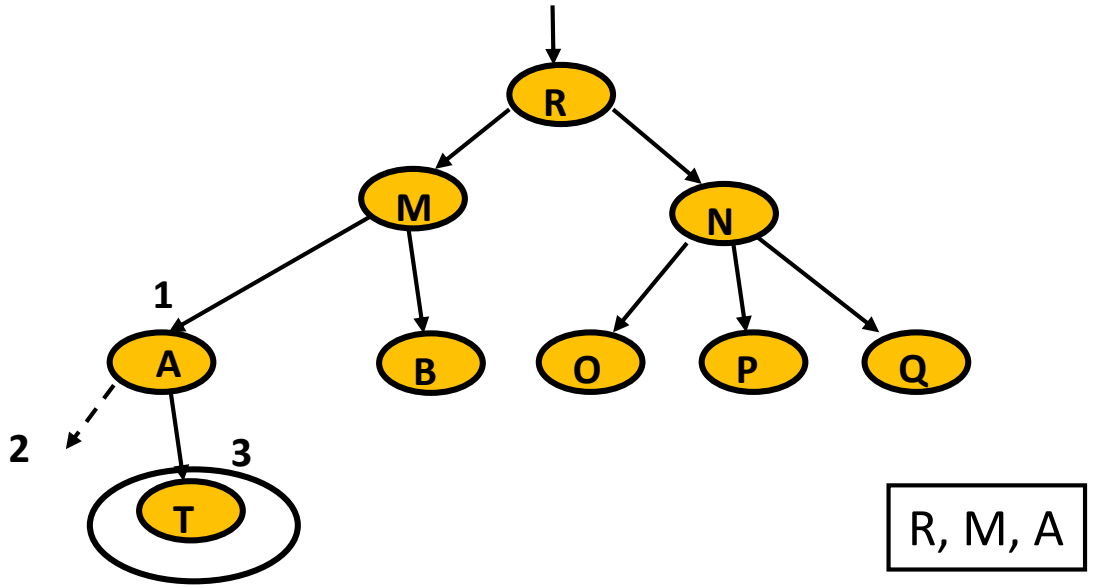
- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - ➢ Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



R, M, A

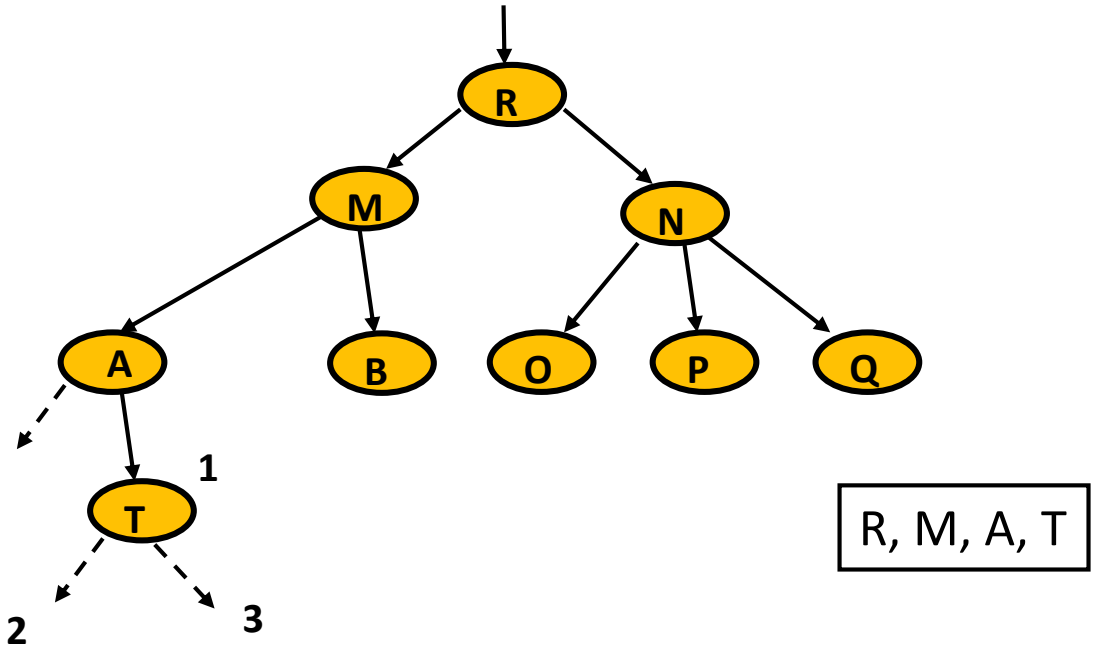
Itérateurs d'arbre

- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - ➢ Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



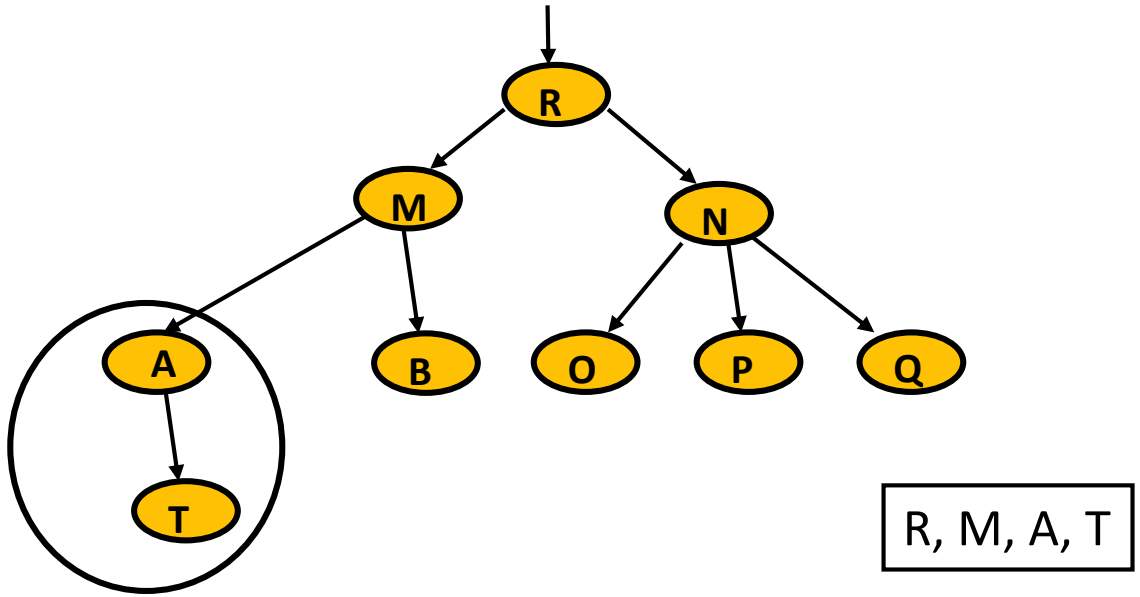
Itérateurs d'arbre

- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



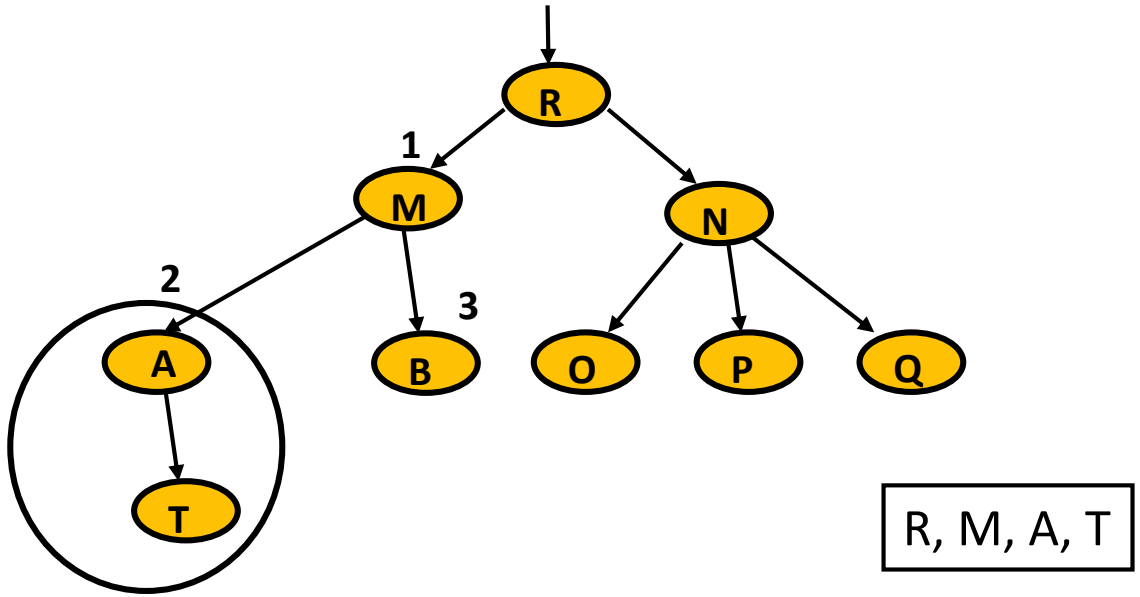
Itérateurs d'arbre

- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



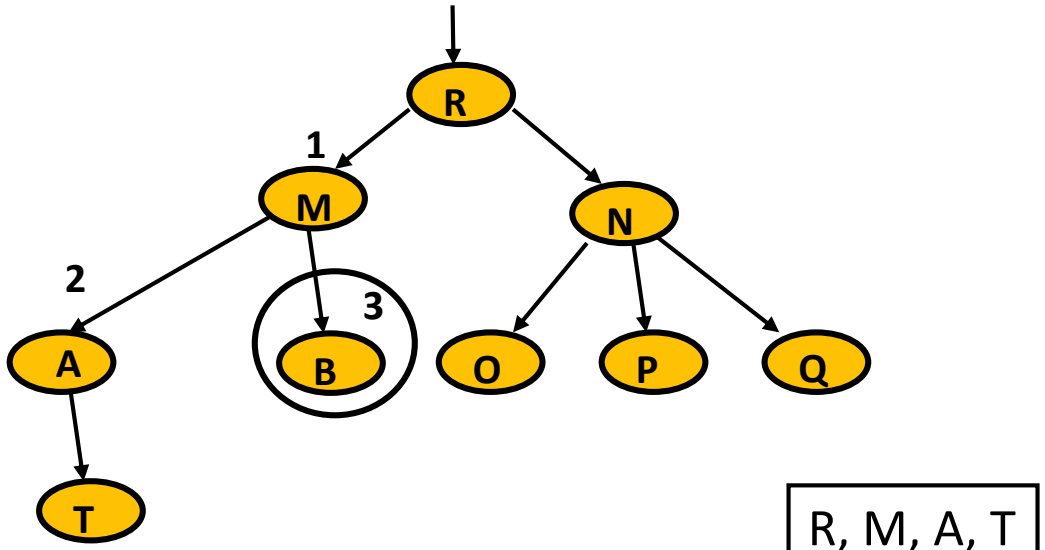
Itérateurs d'arbre

- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - ➢ Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



Itérateurs d'arbre

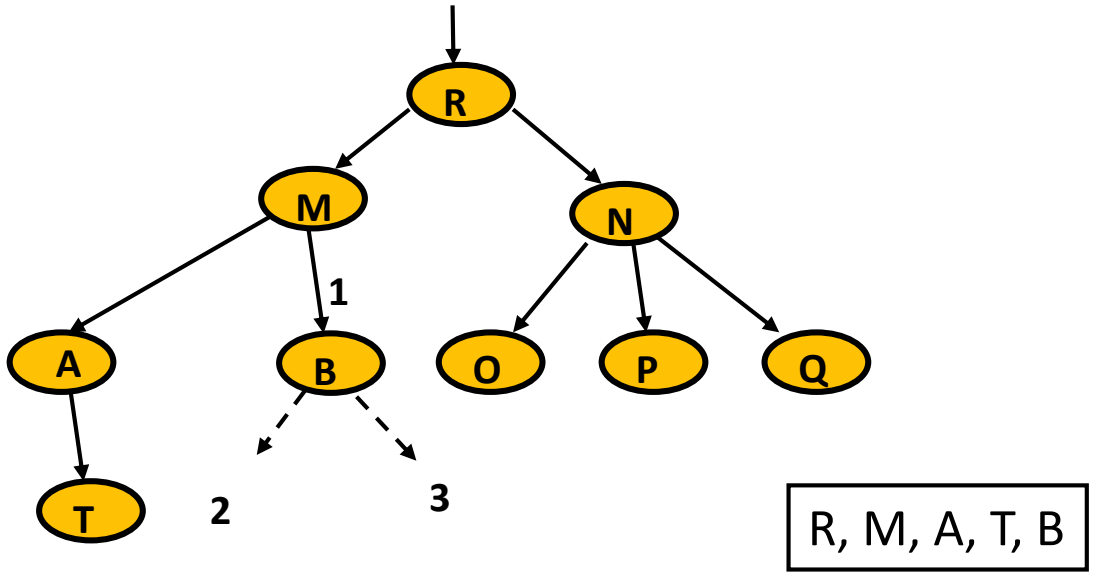
- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - ➢ Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



R, M, A, T

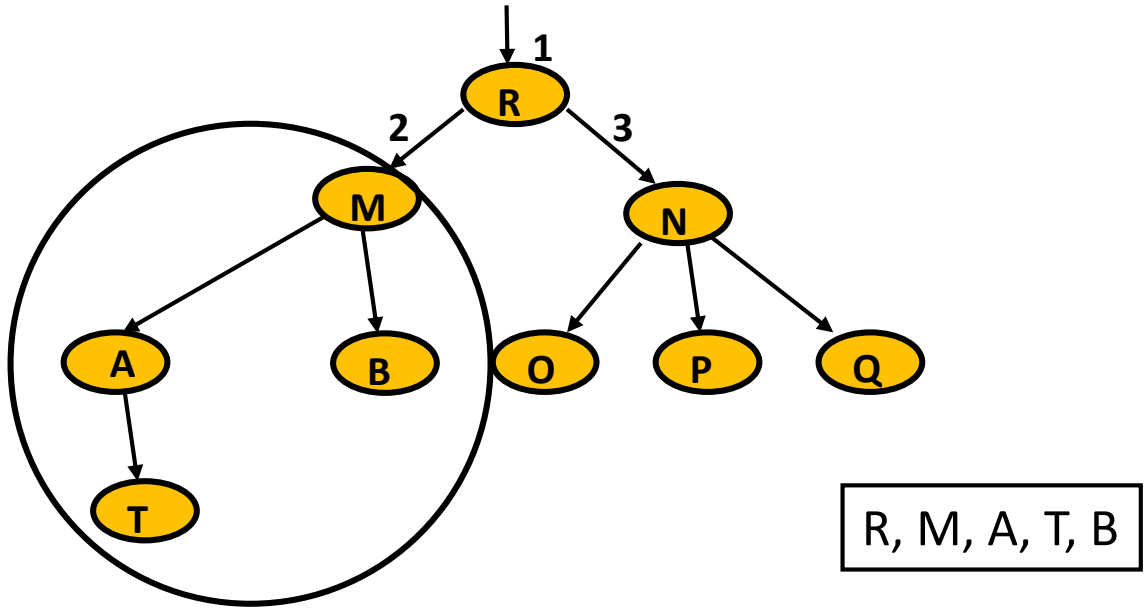
Itérateurs d'arbre

- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



Itérateurs d'arbre

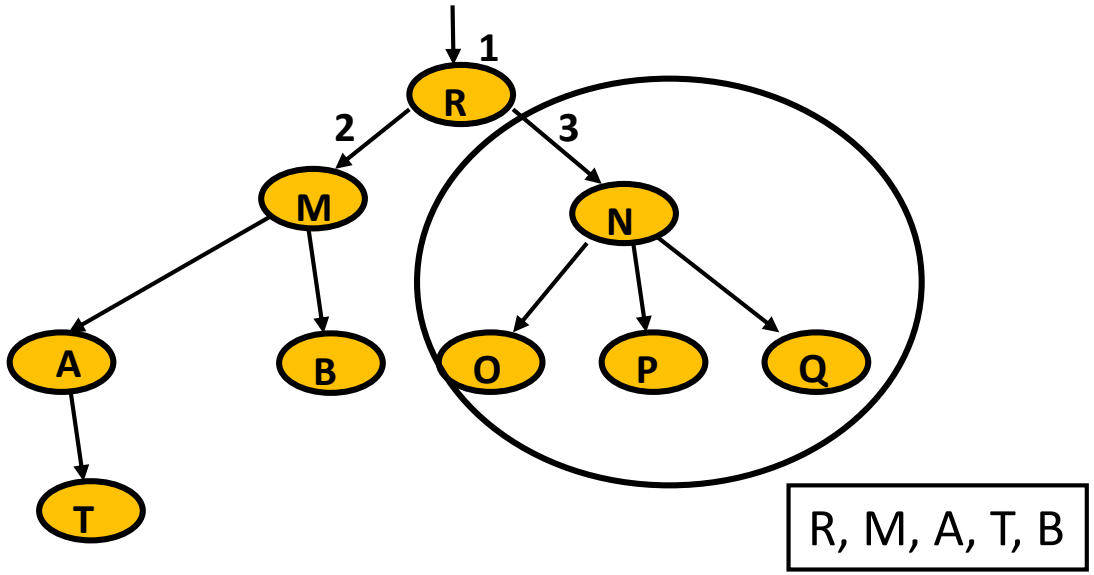
- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - ➢ Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



R, M, A, T, B

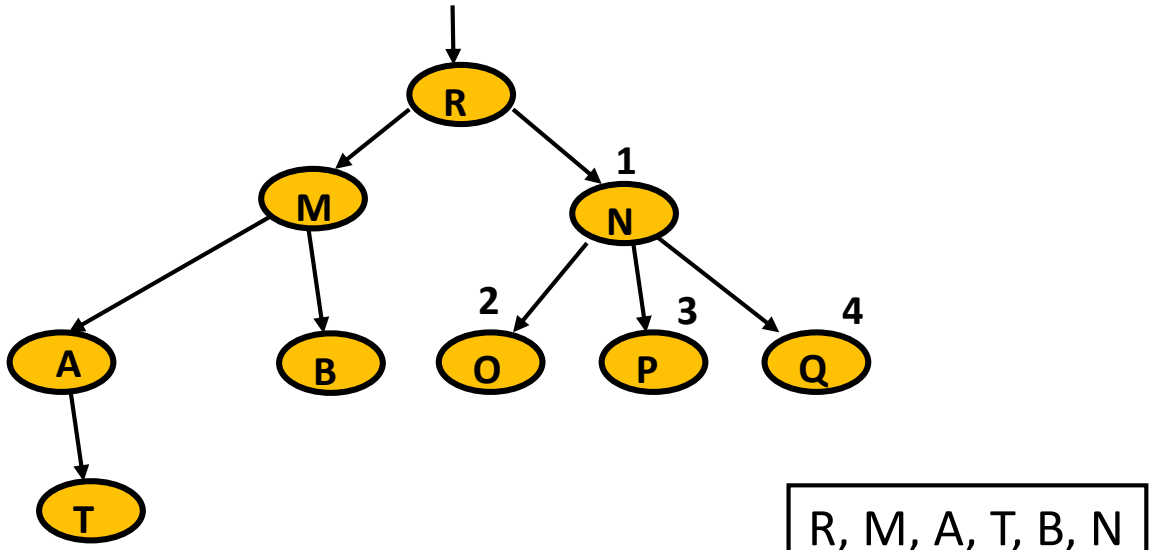
Itérateurs d'arbre

- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



Itérateurs d'arbre

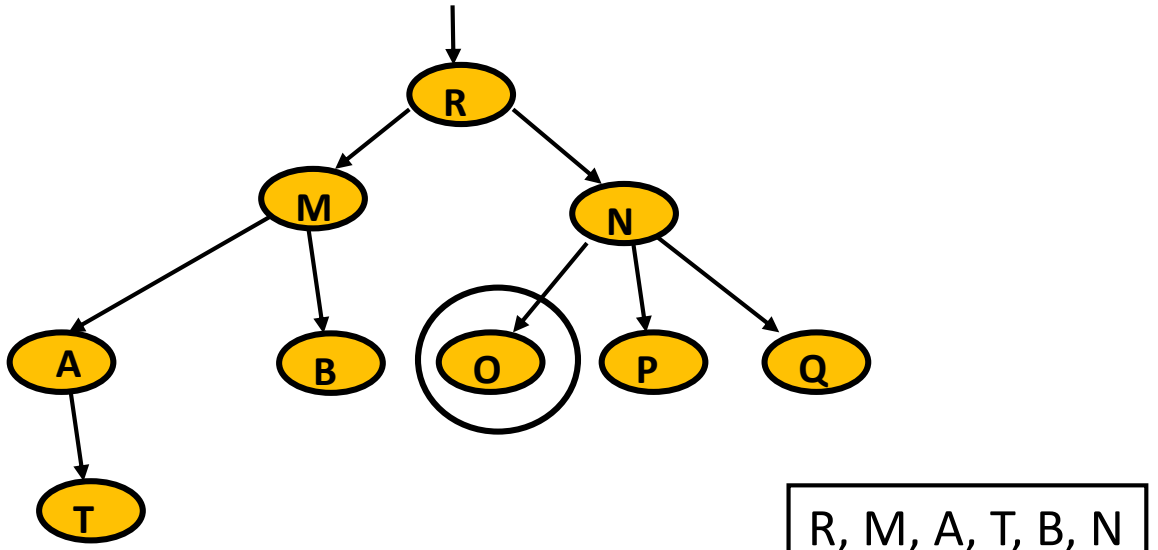
- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - ➢ Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



R, M, A, T, B, N

Itérateurs d'arbre

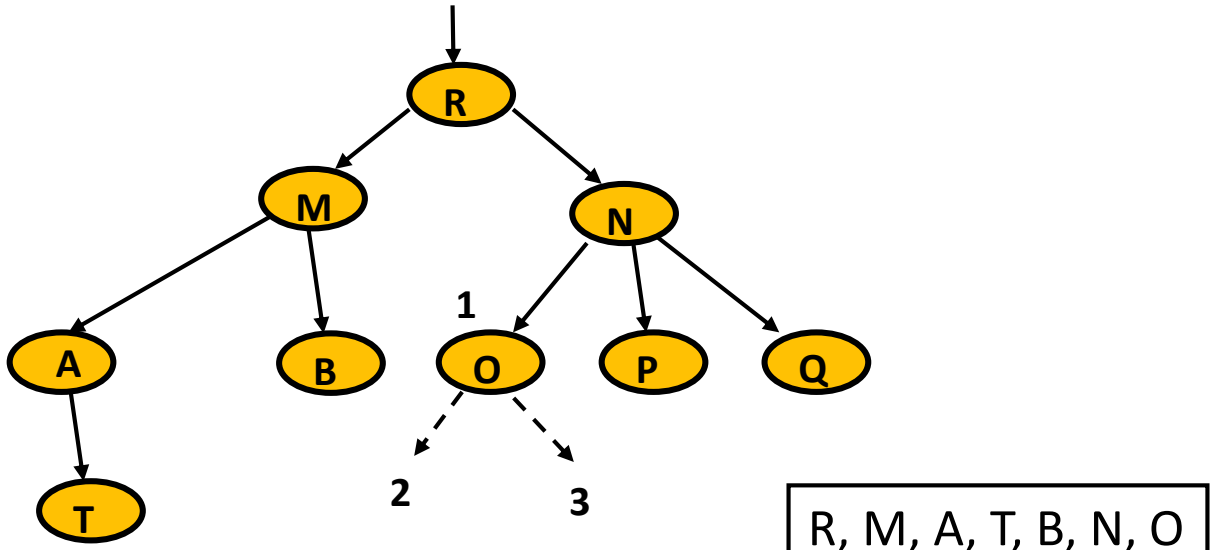
- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - ➢ Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



R, M, A, T, B, N

Itérateurs d'arbre

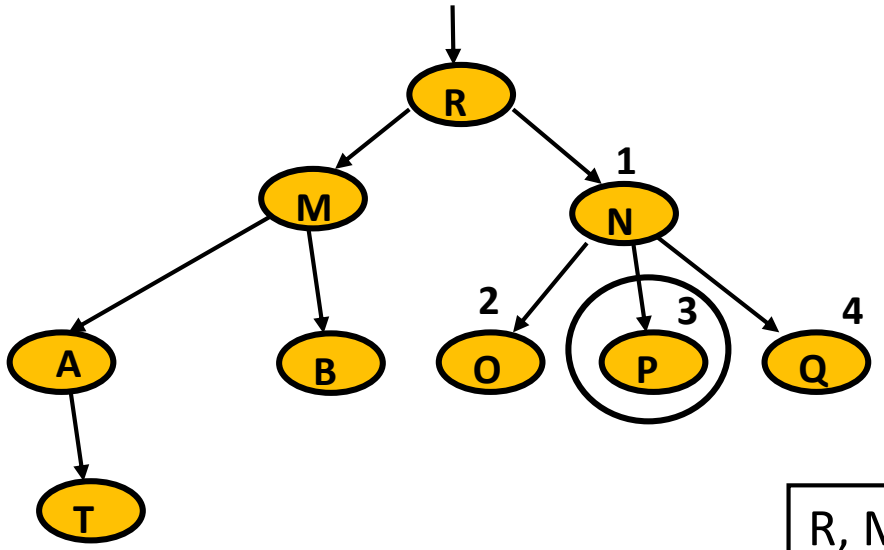
- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - ➢ Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



R, M, A, T, B, N, O

Itérateurs d'arbre

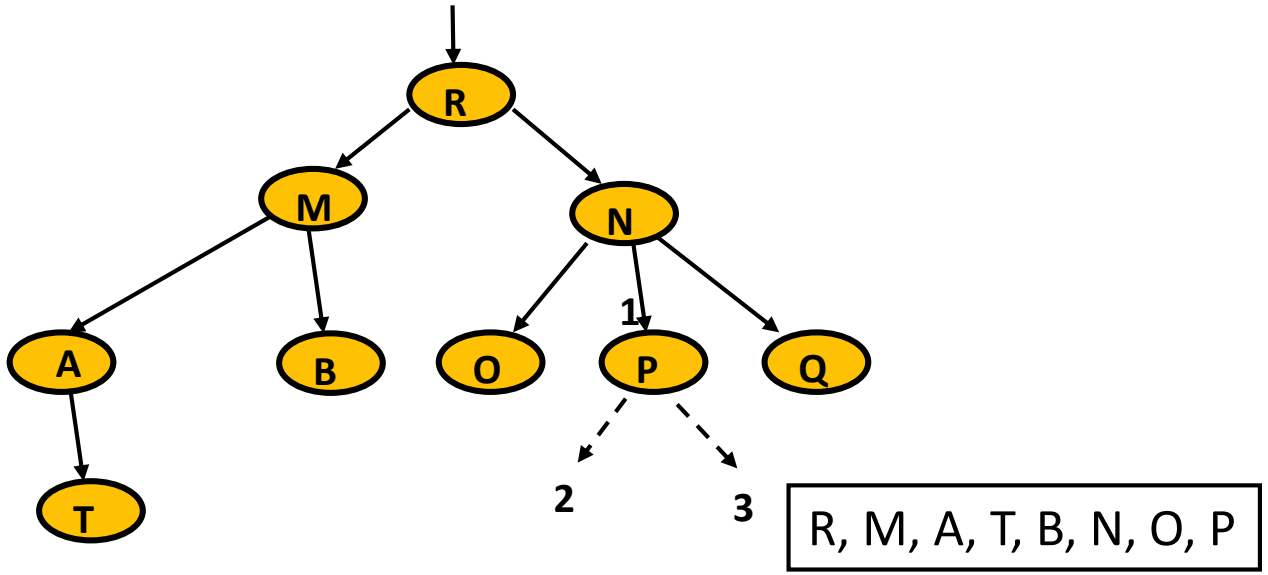
- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - ➢ Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



R, M, A, T, B, N, O

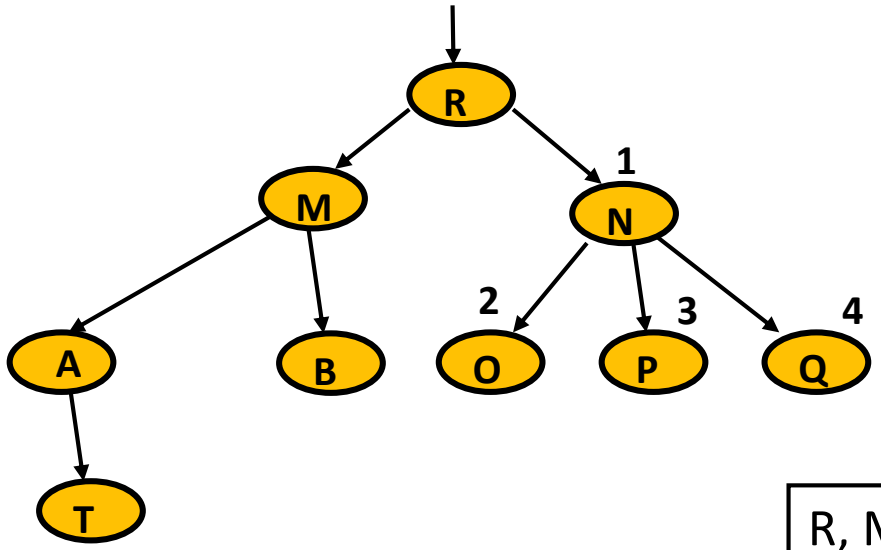
Itérateurs d'arbre

- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



Itérateurs d'arbre

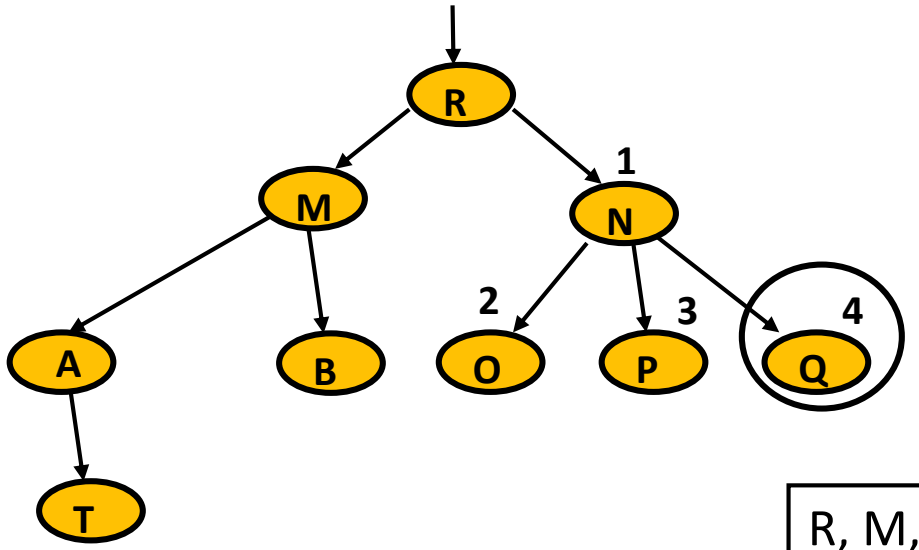
- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - ➢ Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



R, M, A, T, B, N, O, P

Itérateurs d'arbre

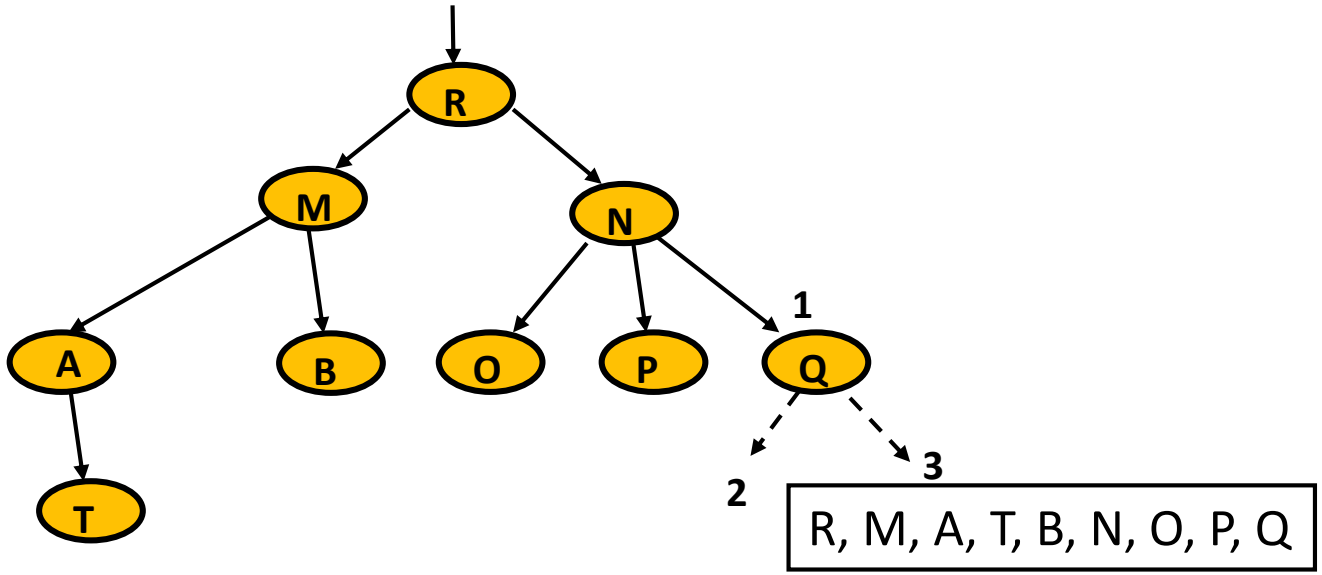
- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - ➢ Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



R, M, A, T, B, N, O, P

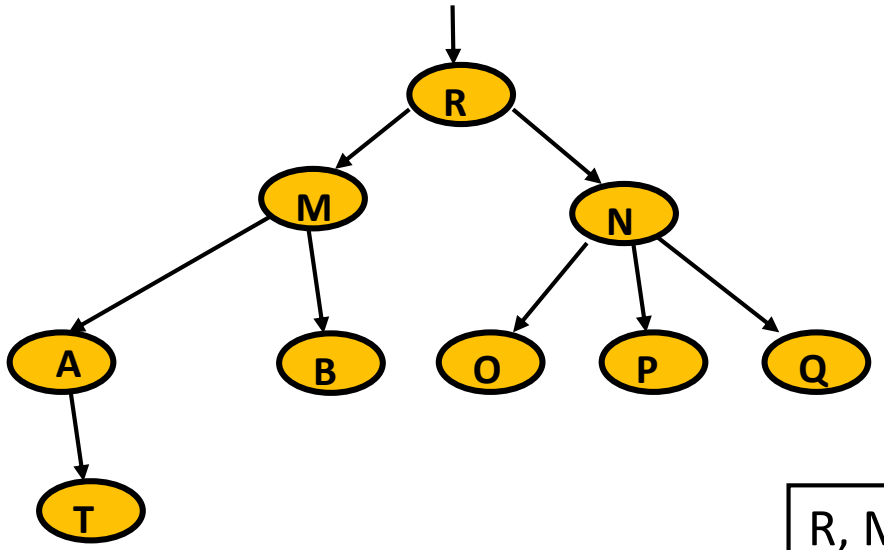
Itérateurs d'arbre

- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



Itérateurs d'arbre

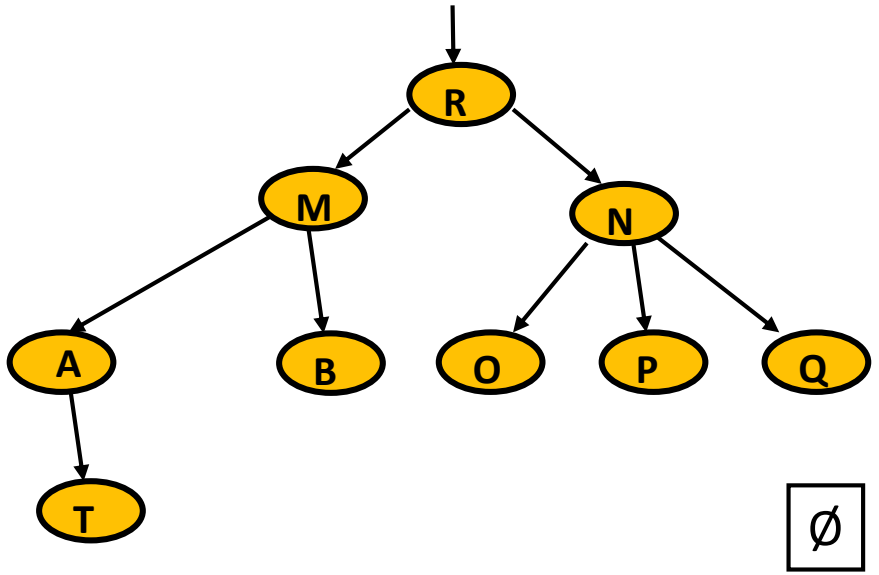
- Parcours **pré-ordre** : les descendants d'un nœud sont traités **après** lui
  - ➢ Priorité au père (pré-ordre)
    - ✓ 1. Visiter la racine r ;
    - ✓ 2. Visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$



R, M, A, T, B, N, O, P, Q

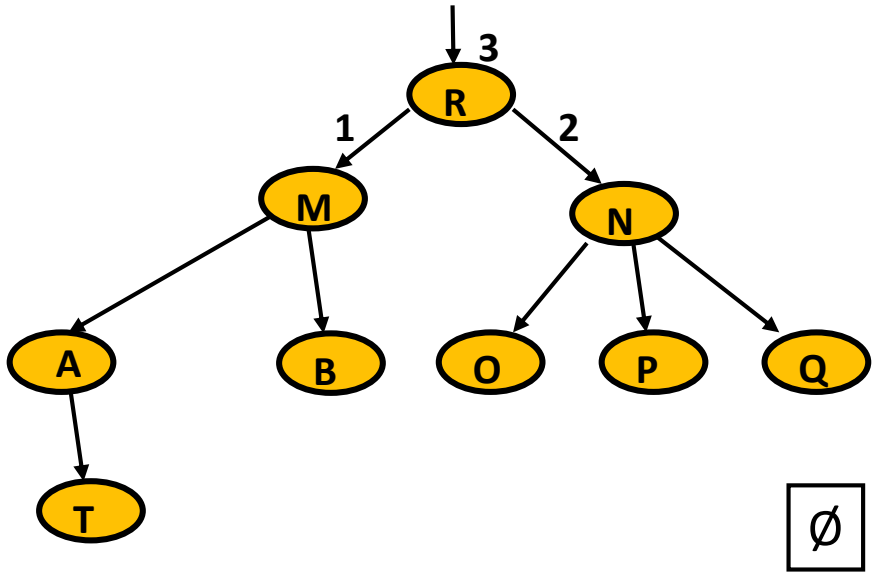
Itérateurs d'arbre

- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



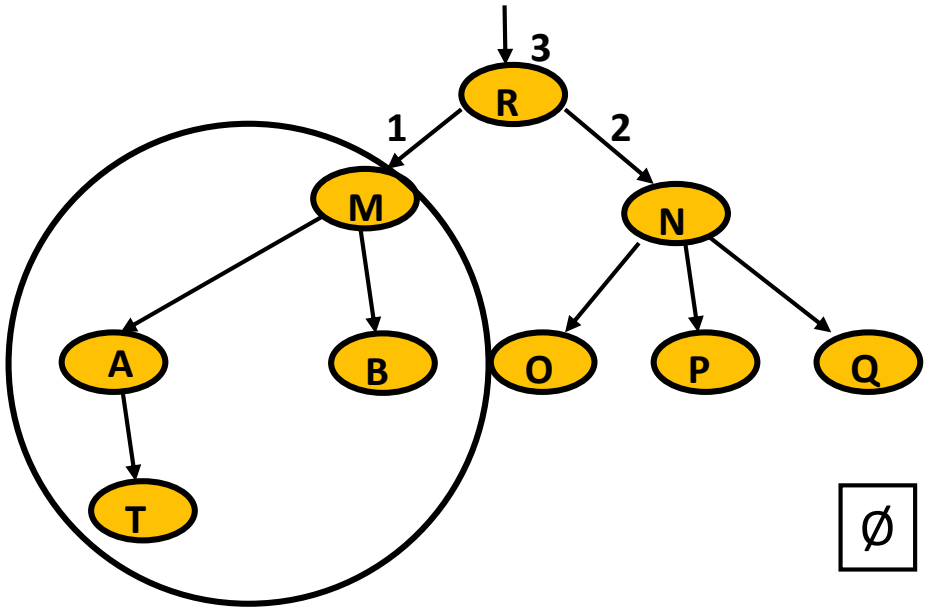
Itérateurs d'arbre

- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



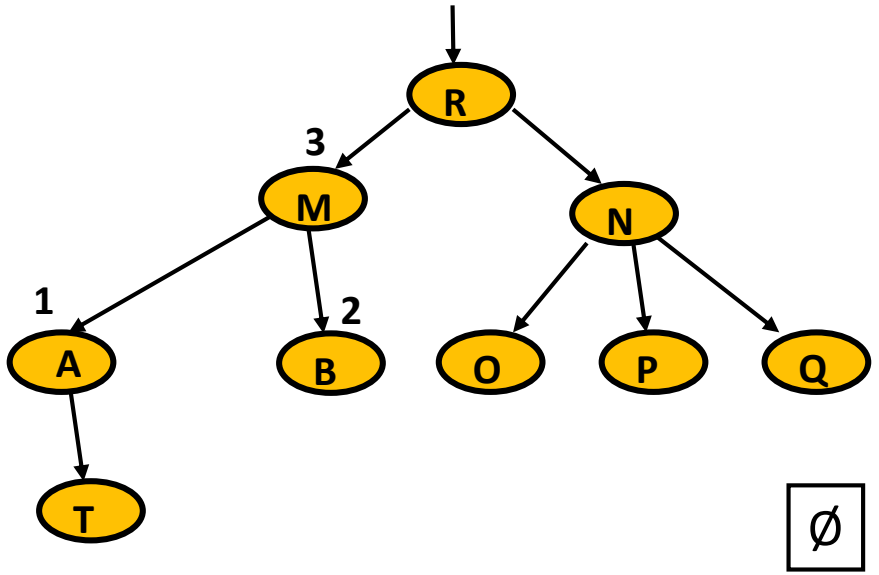
Itérateurs d'arbre

- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



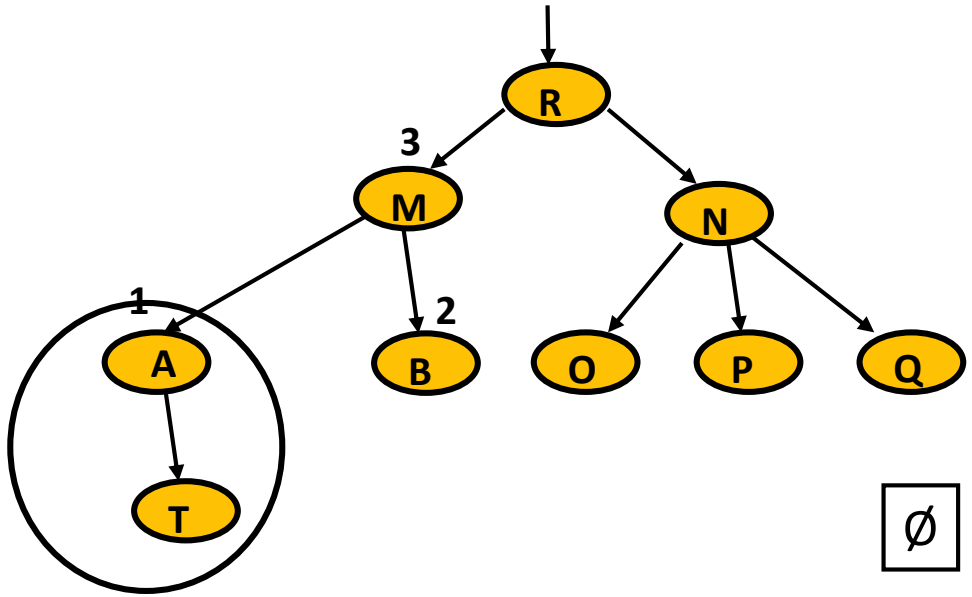
Itérateurs d'arbre

- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



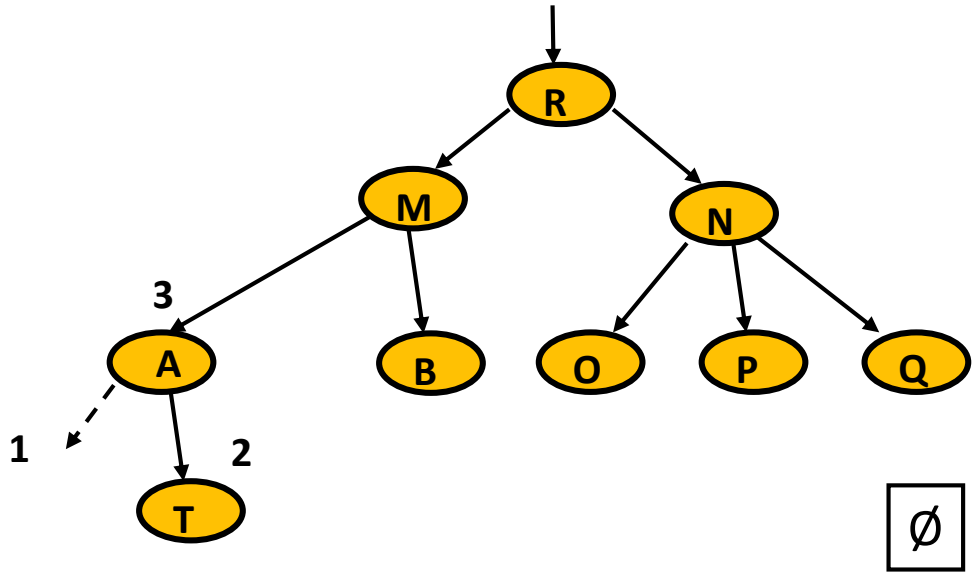
Itérateurs d'arbre

- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



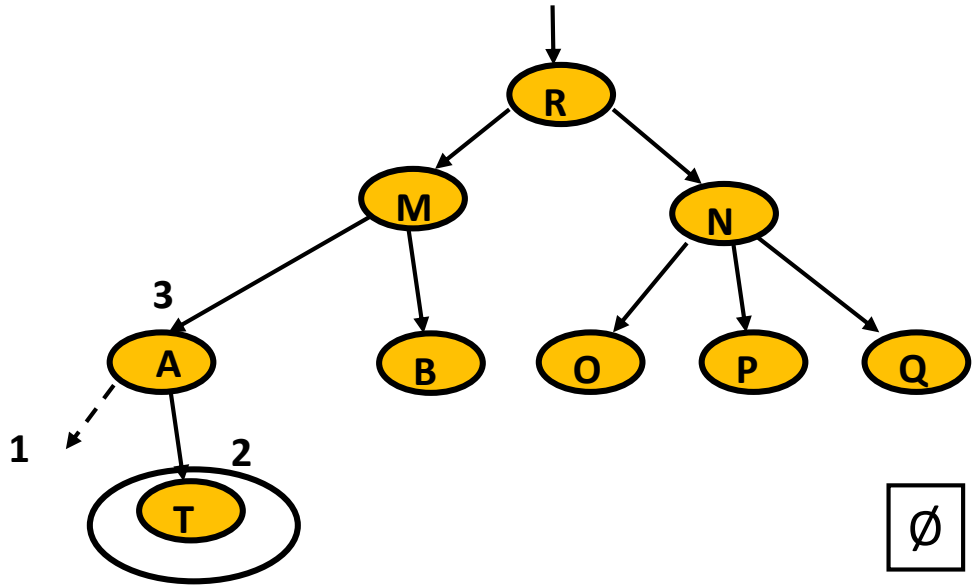
Itérateurs d'arbre

- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



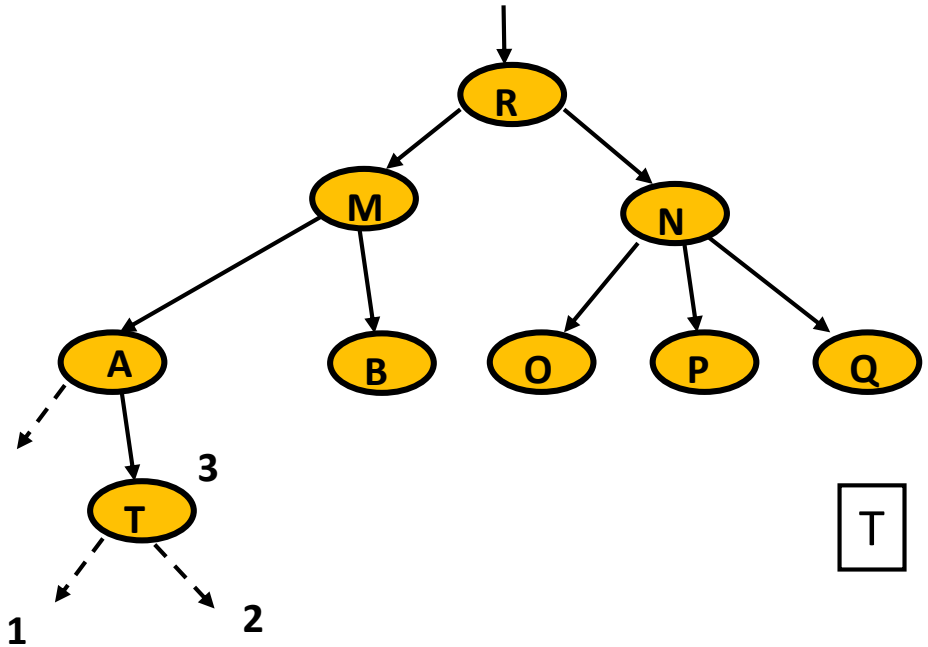
Itérateurs d'arbre

- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



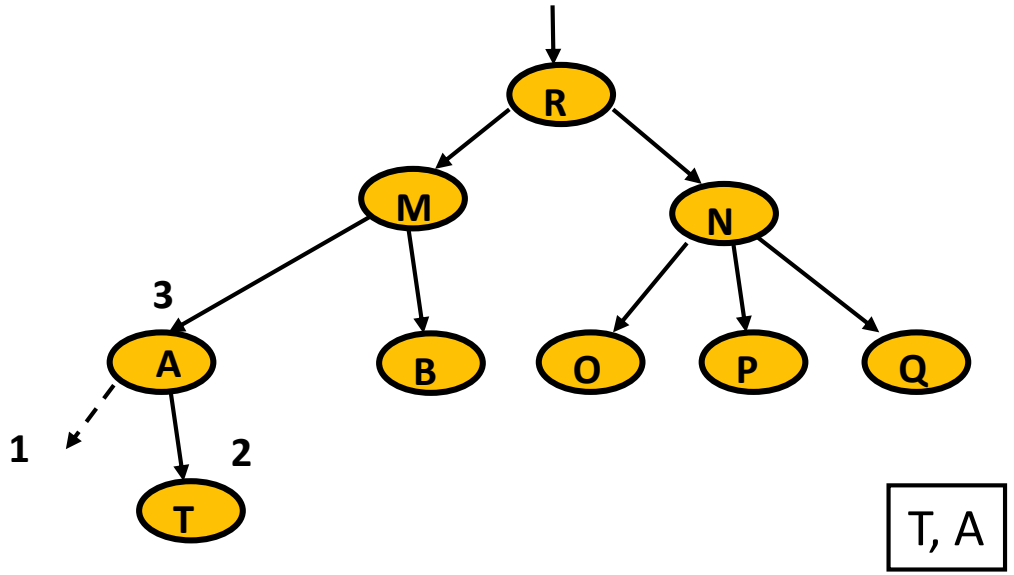
Itérateurs d'arbre

- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



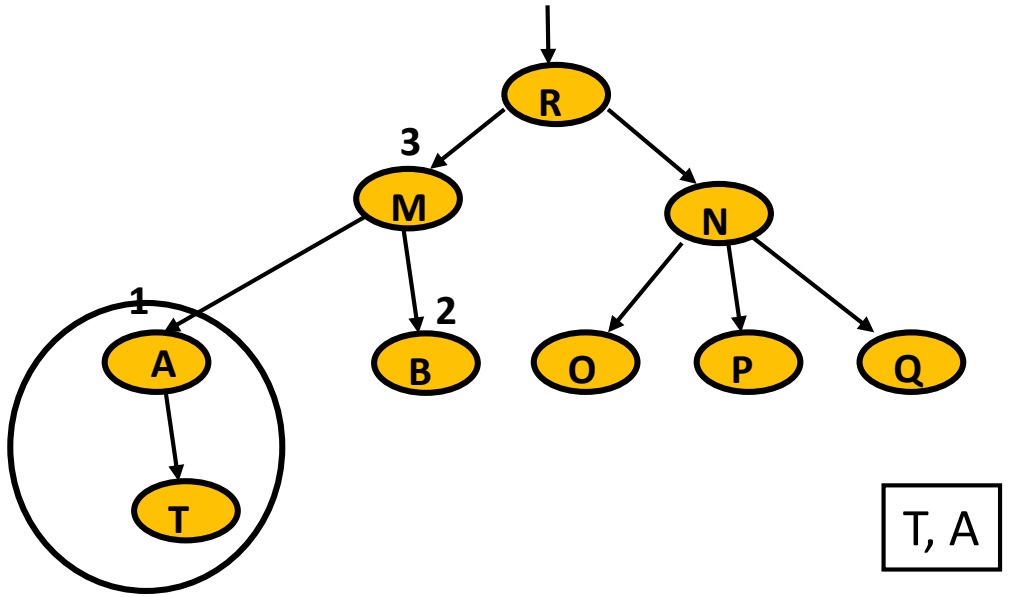
Itérateurs d'arbre

- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



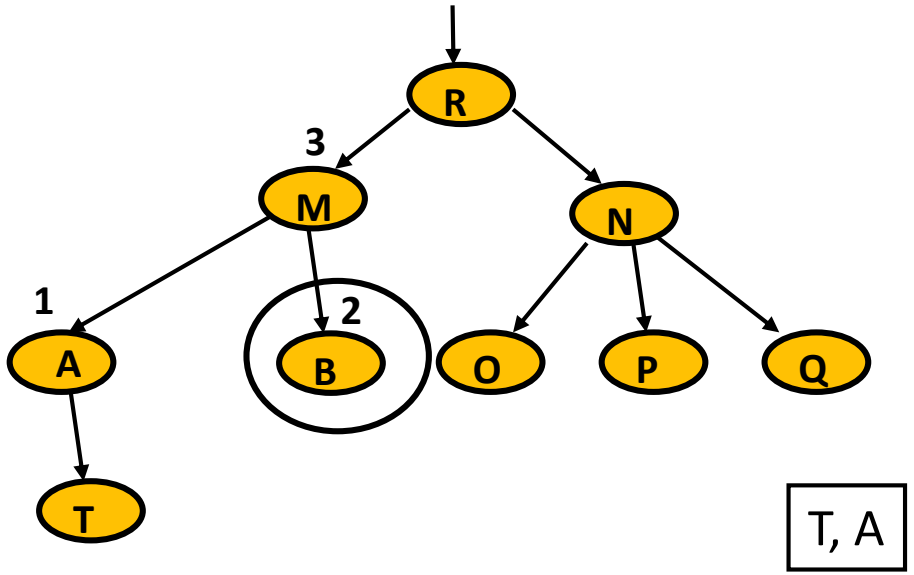
Itérateurs d'arbre

- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



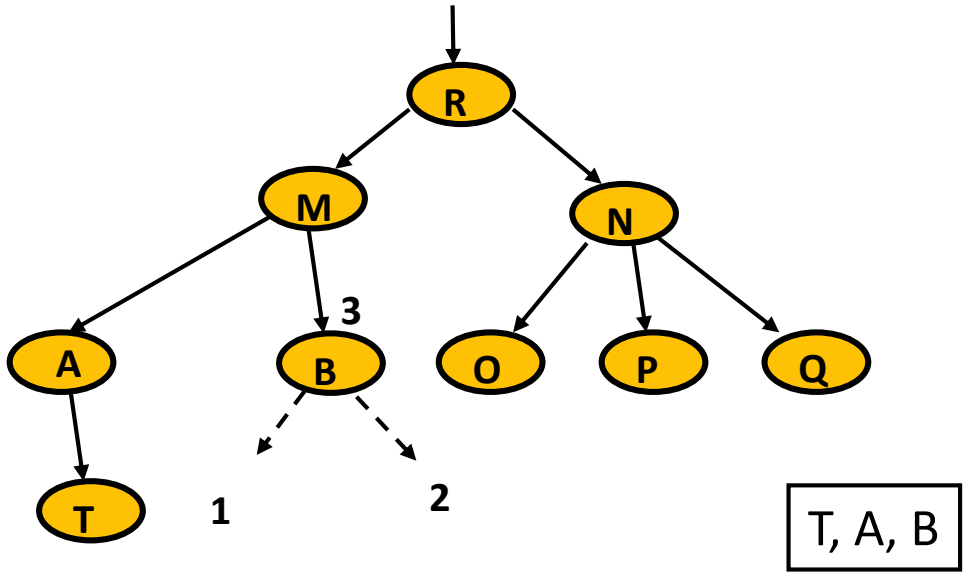
Itérateurs d'arbre

- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



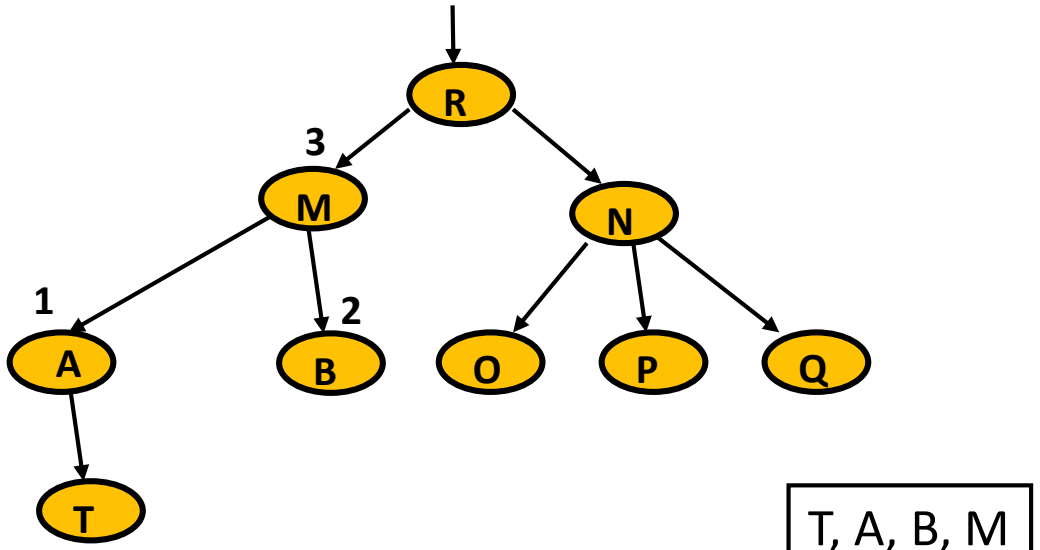
Itérateurs d'arbre

- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



Itérateurs d'arbre

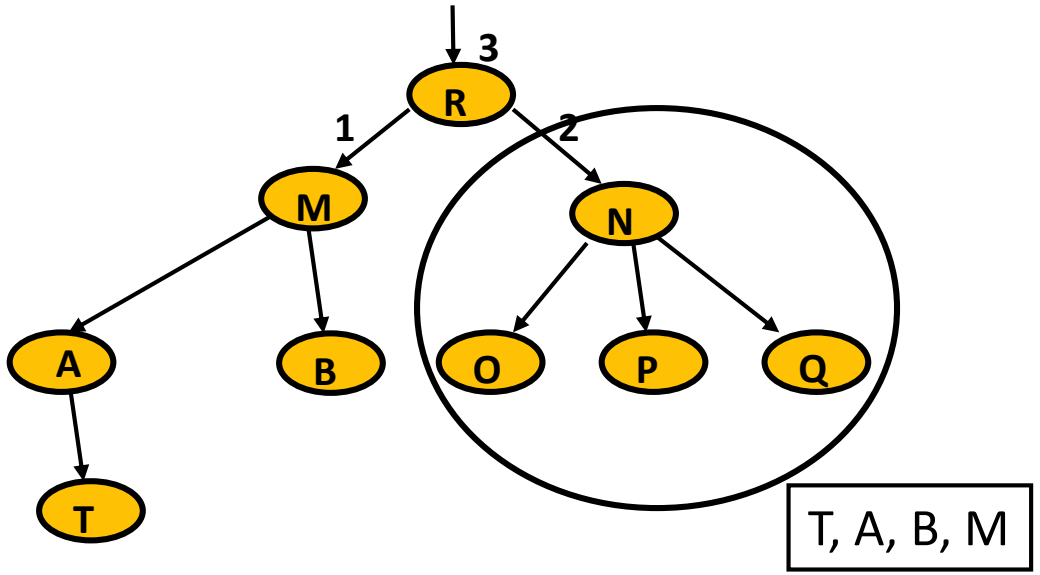
- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



T, A, B, M

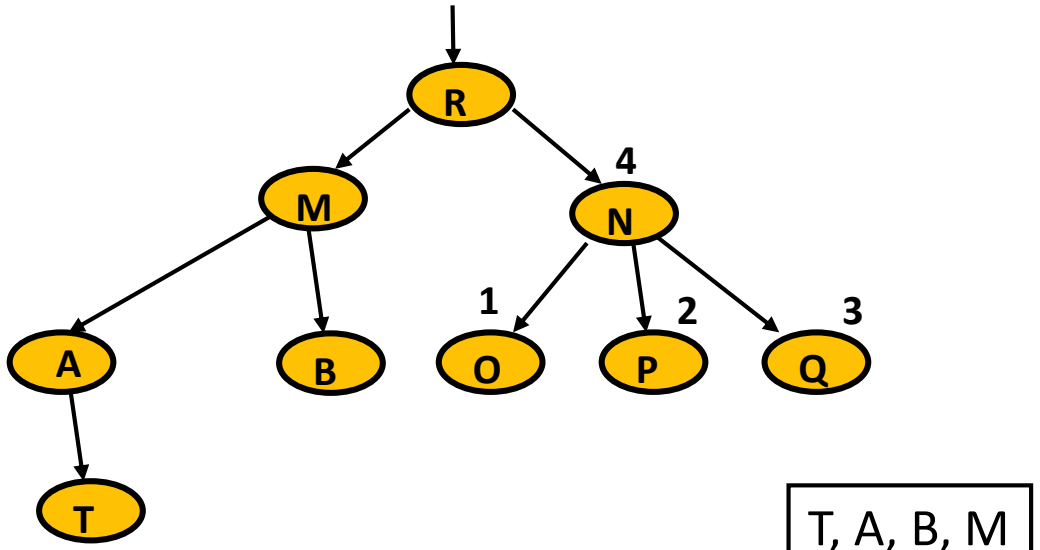
Itérateurs d'arbre

- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



Itérateurs d'arbre

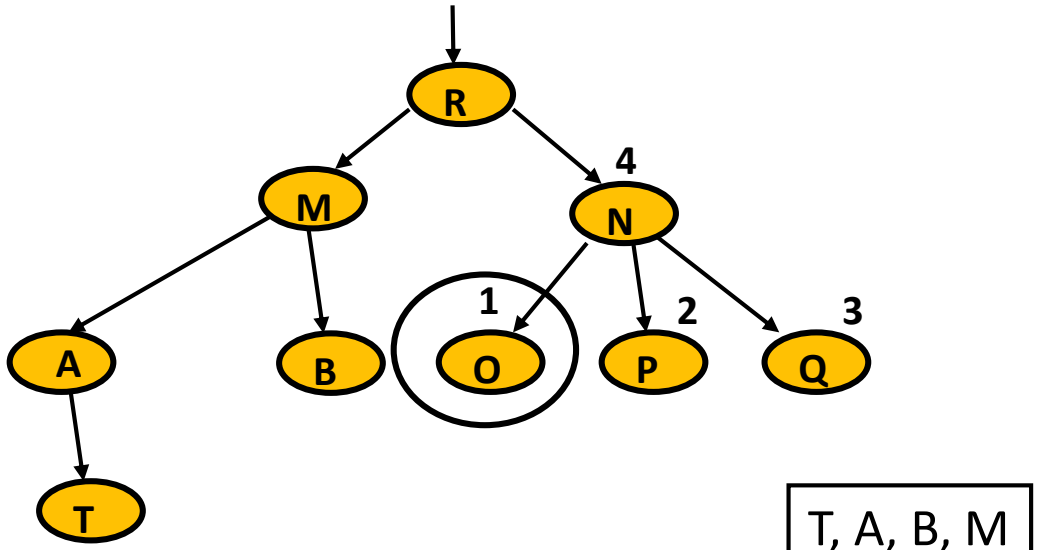
- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



T, A, B, M

Itérateurs d'arbre

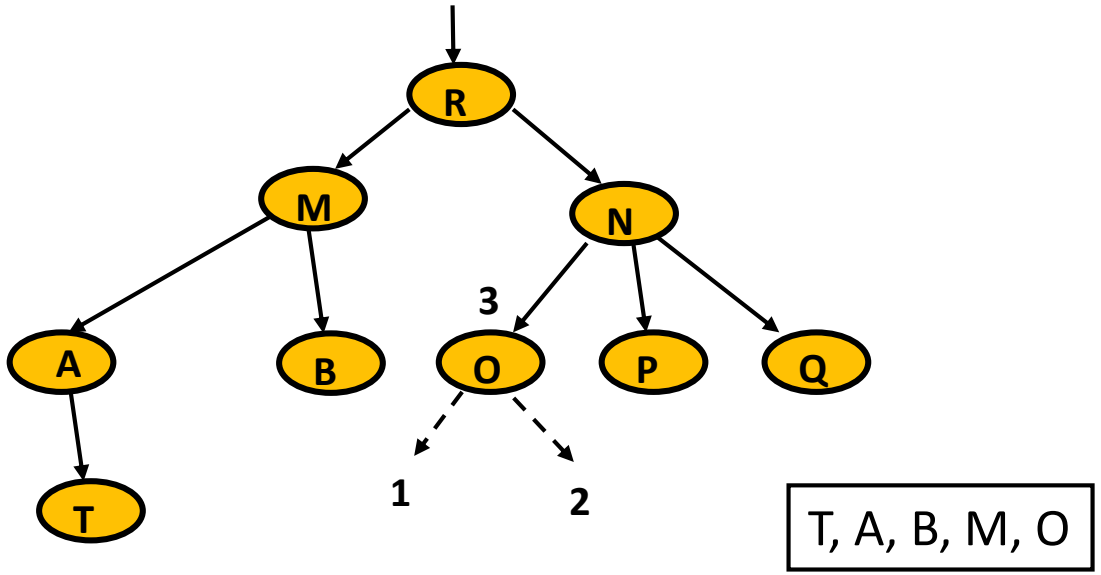
- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



T, A, B, M

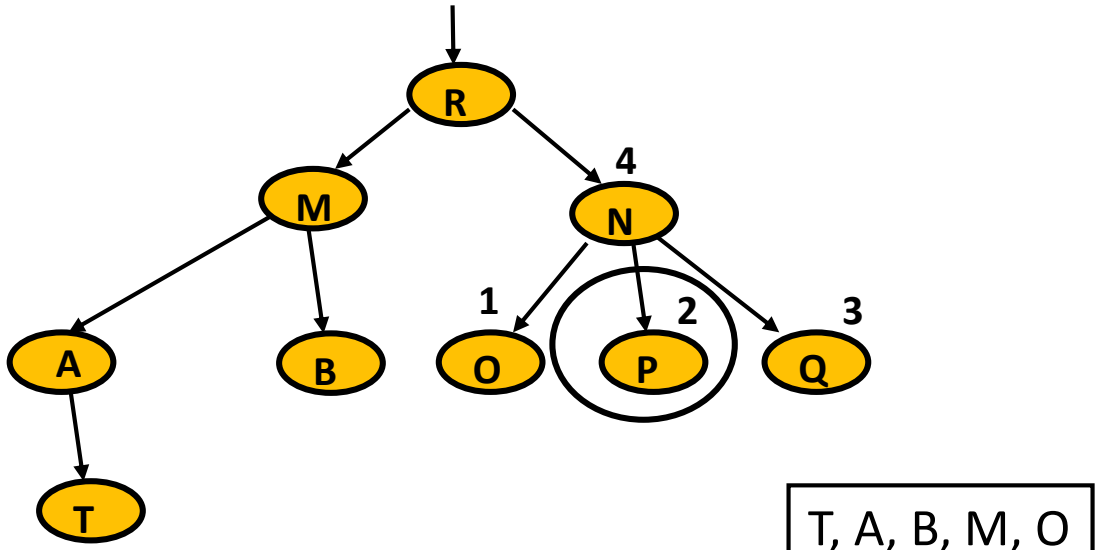
Itérateurs d'arbre

- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



Itérateurs d'arbre

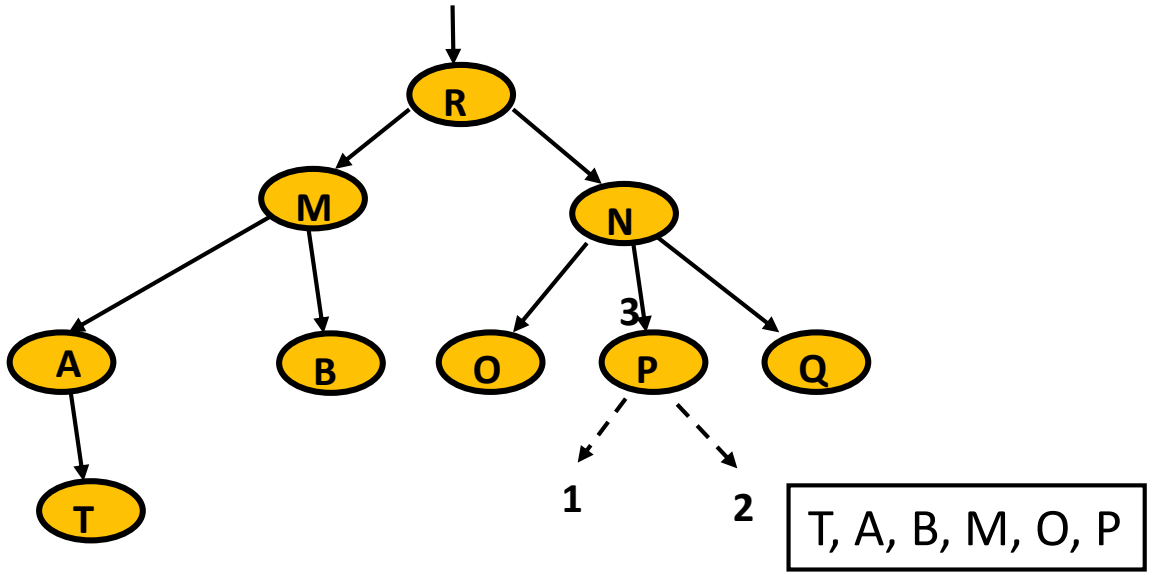
- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



T, A, B, M, O

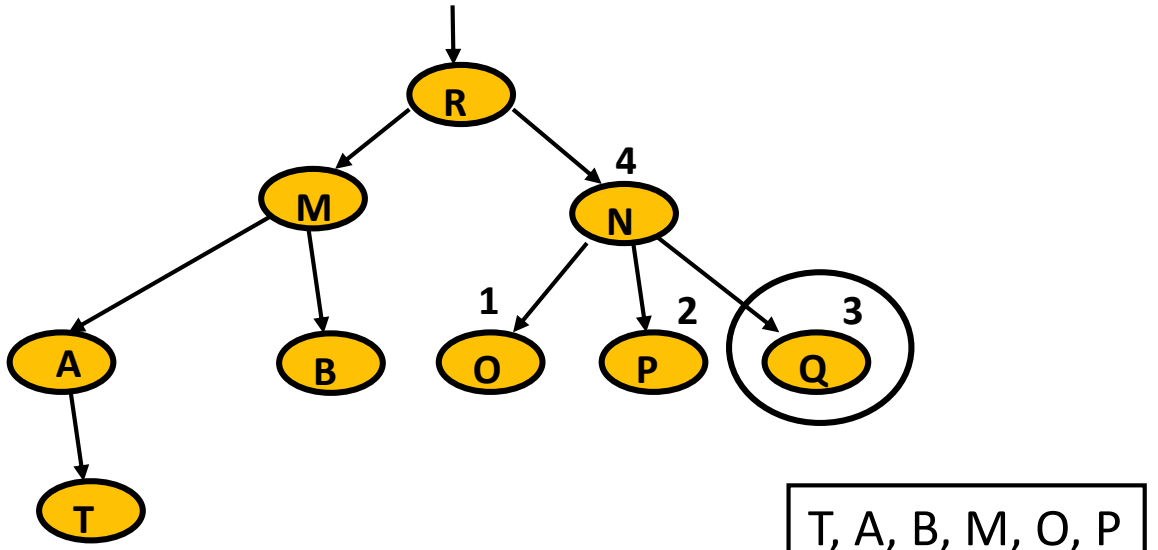
Itérateurs d'arbre

- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



Itérateurs d'arbre

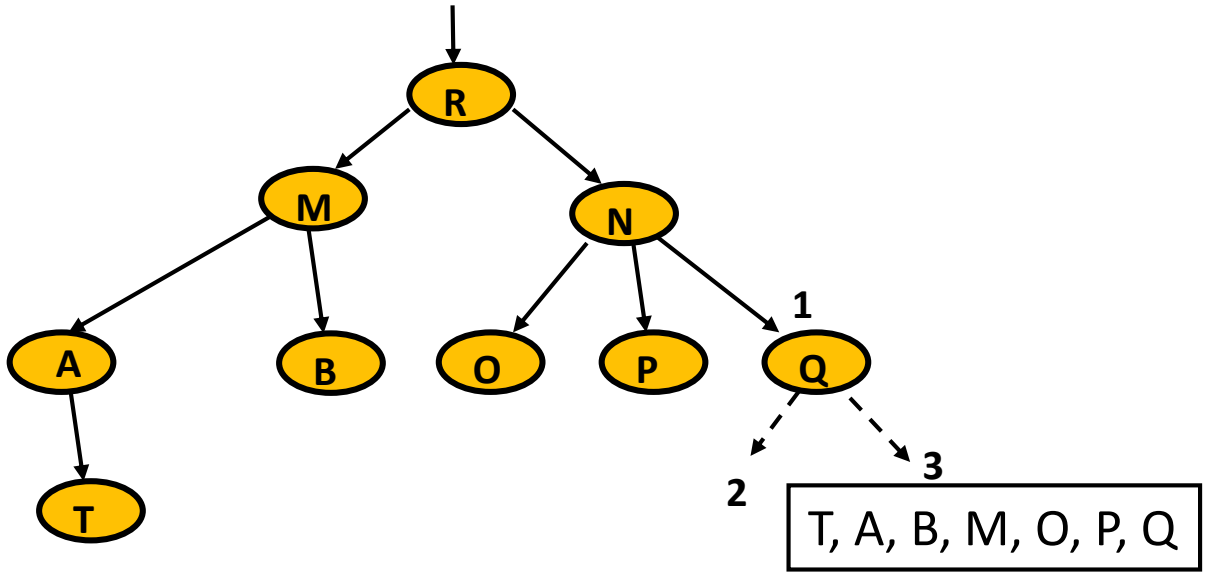
- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



T, A, B, M, O, P

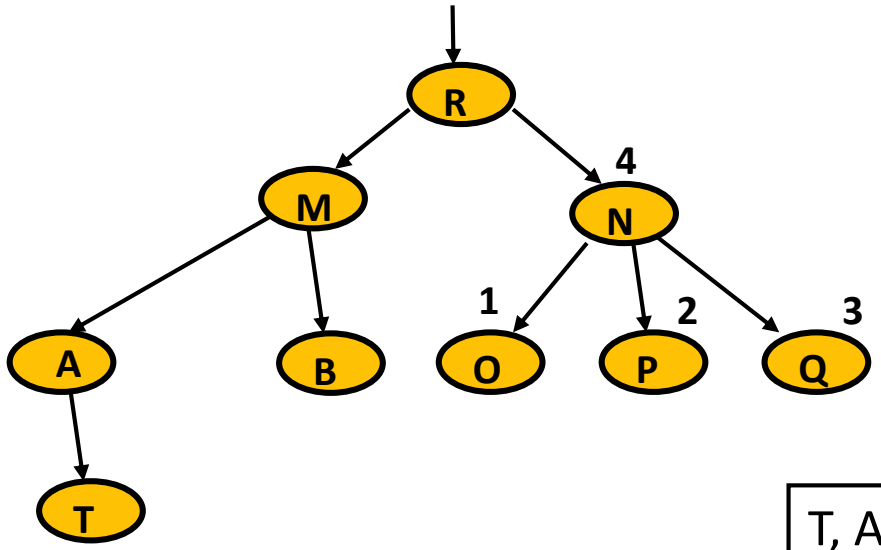
Itérateurs d'arbre

- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



Itérateurs d'arbre

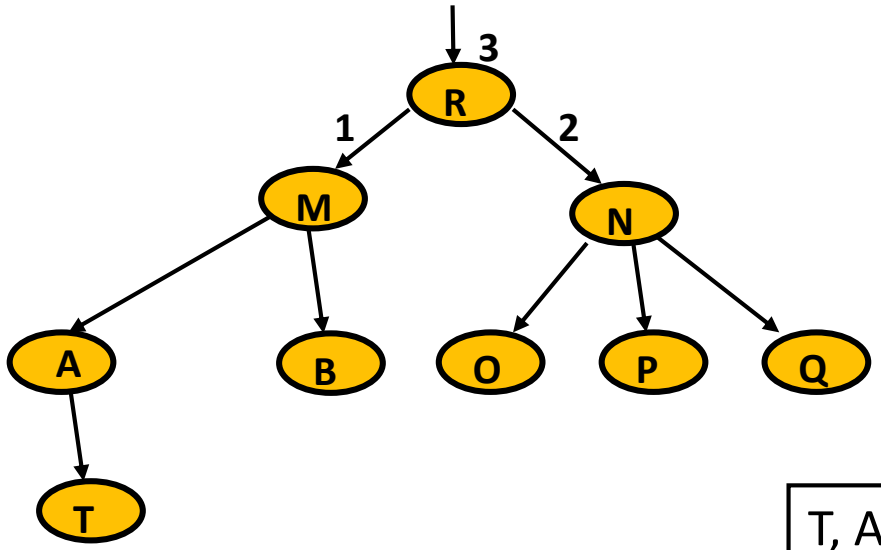
- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



T, A, B, M, O, P, Q, N

Itérateurs d'arbre

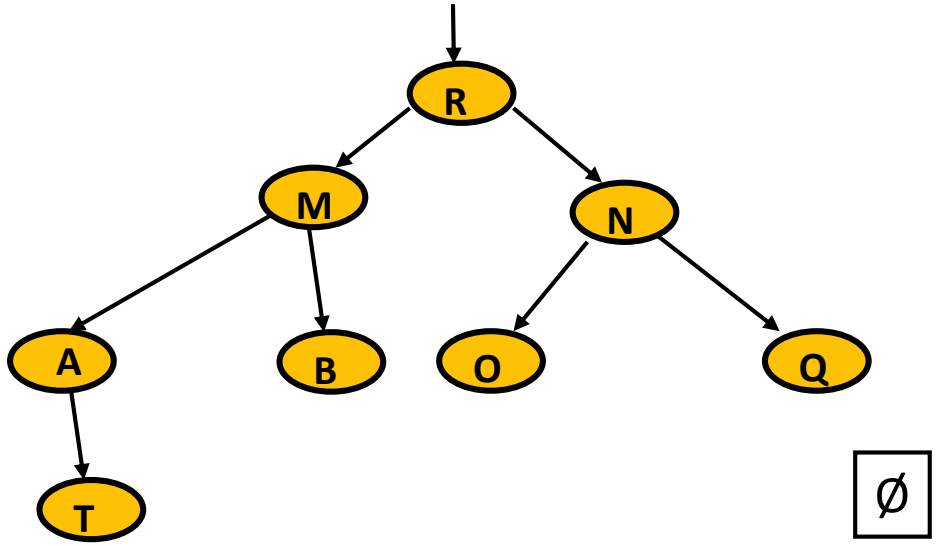
- Parcours **post-ordre** : les descendants d'un nœud sont traités **avant** lui
  - ➢ priorité aux fils (post-ordre)
    - ✓ 1. visiter récursivement les enfants : $v_1, v_2, \ldots, v_k$
    - ✓ 2. visiter la racine r ;



T, A, B, M, O, P, Q, N, R

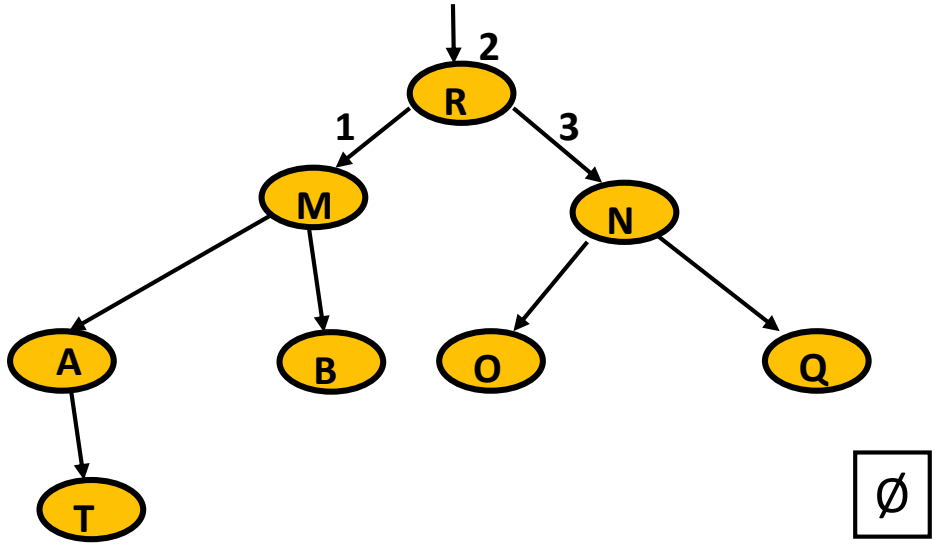
Itérateurs d'arbre

- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



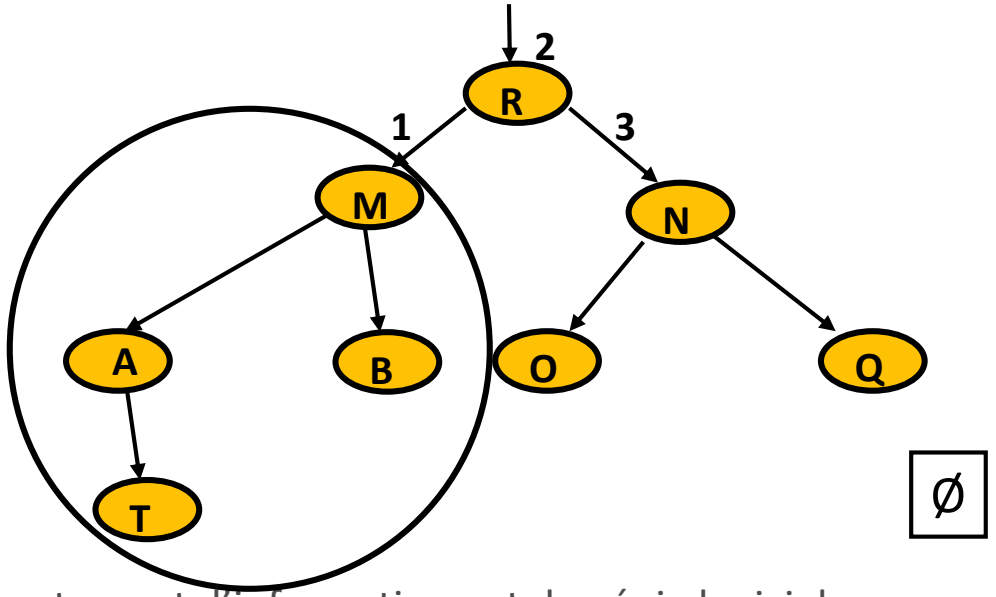
Itérateurs d'arbre

- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



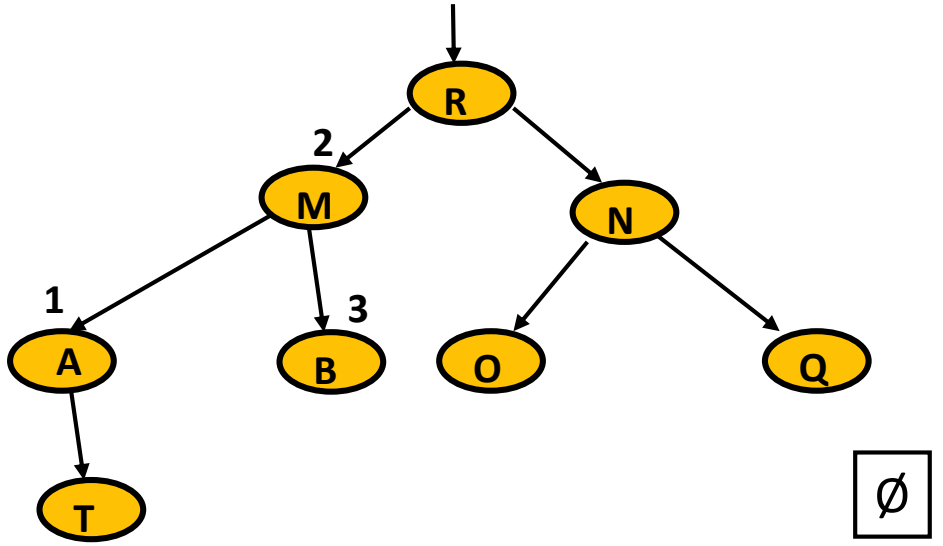
Itérateurs d'arbre

- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



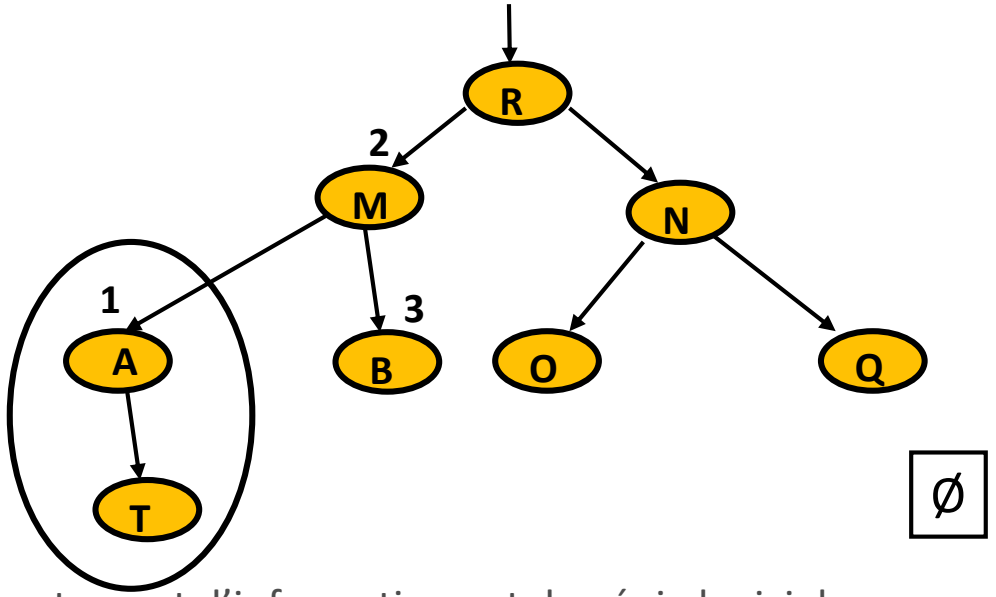
Itérateurs d'arbre

- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



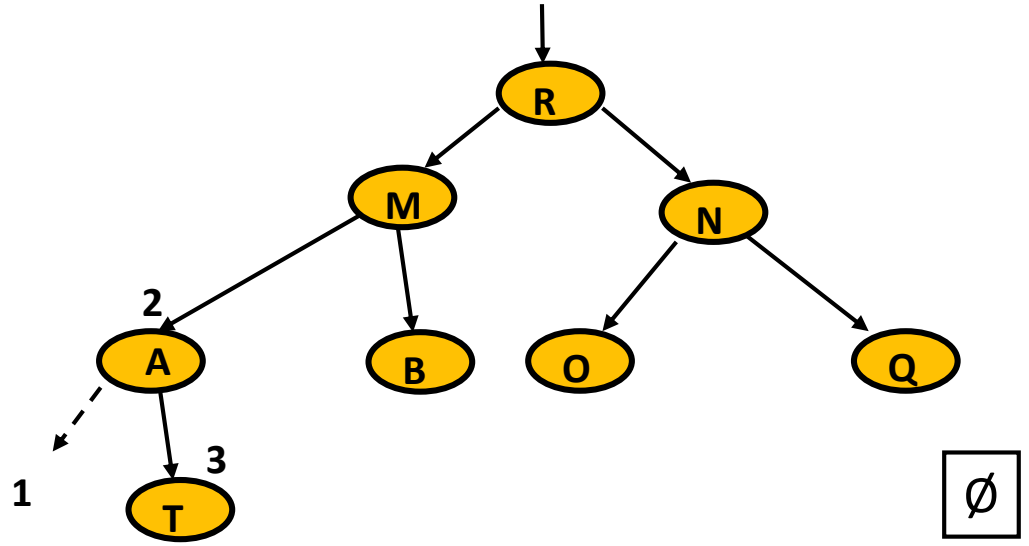
Itérateurs d'arbre

- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



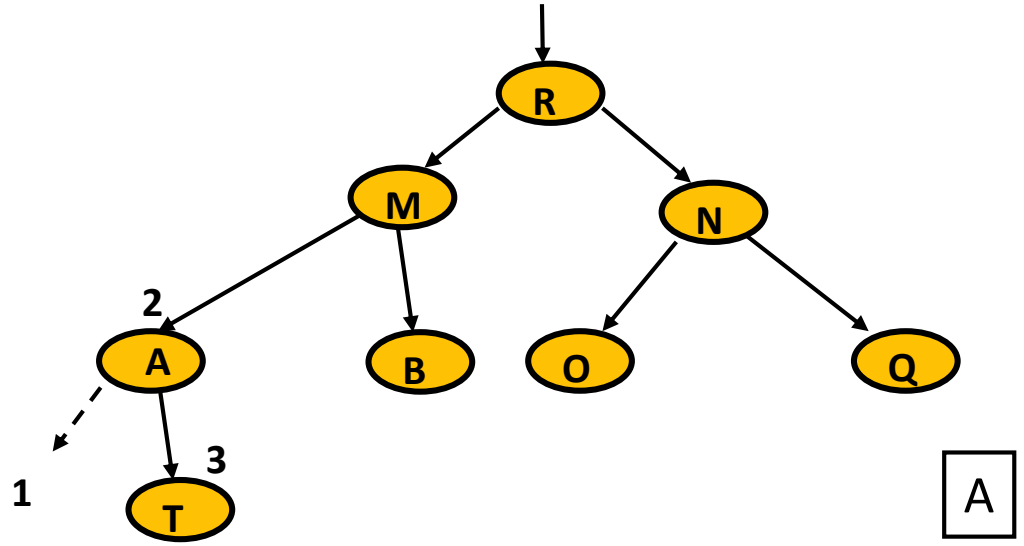
Itérateurs d'arbre

- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



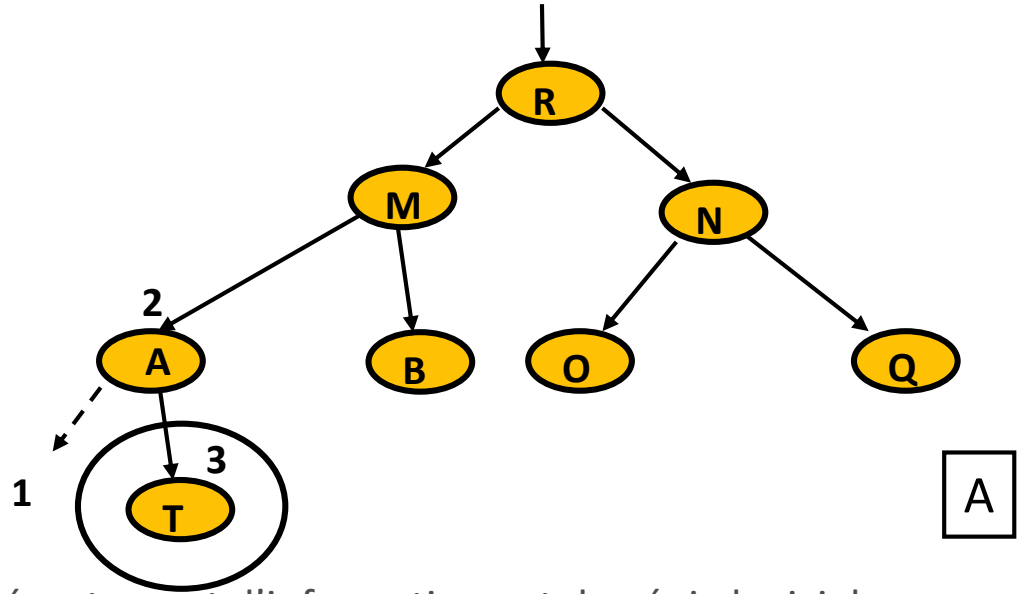
Itérateurs d'arbre

- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



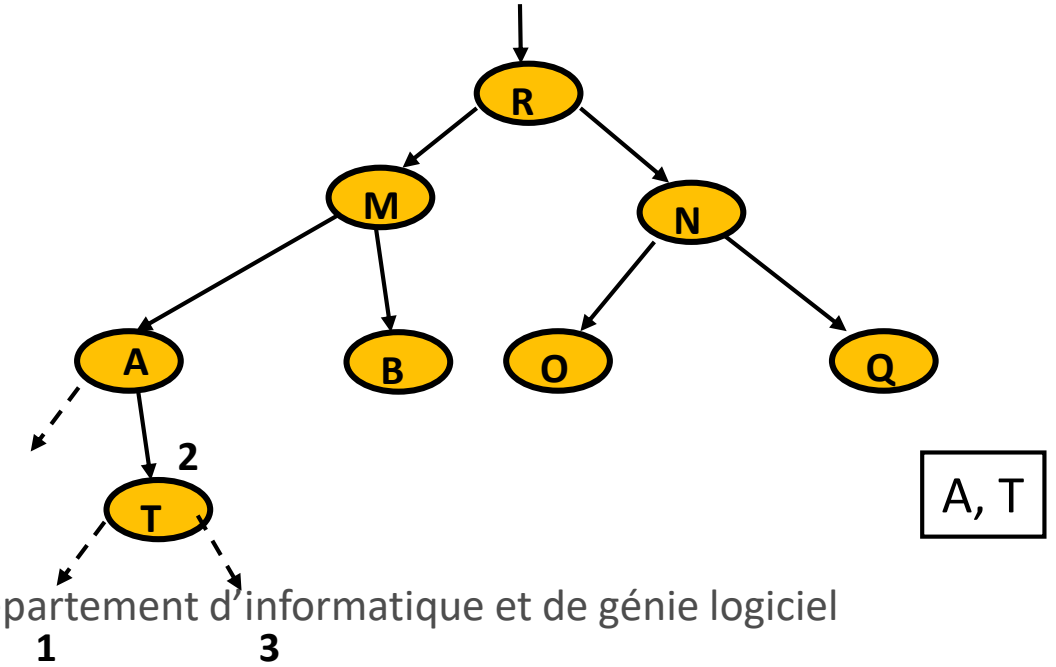
Itérateurs d'arbre

- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



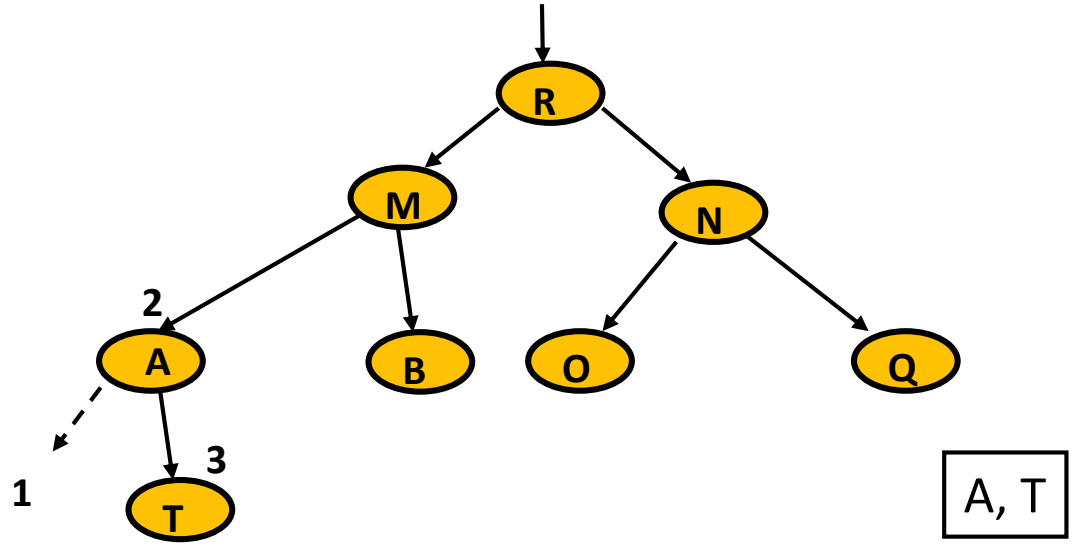
Itérateurs d'arbre

- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



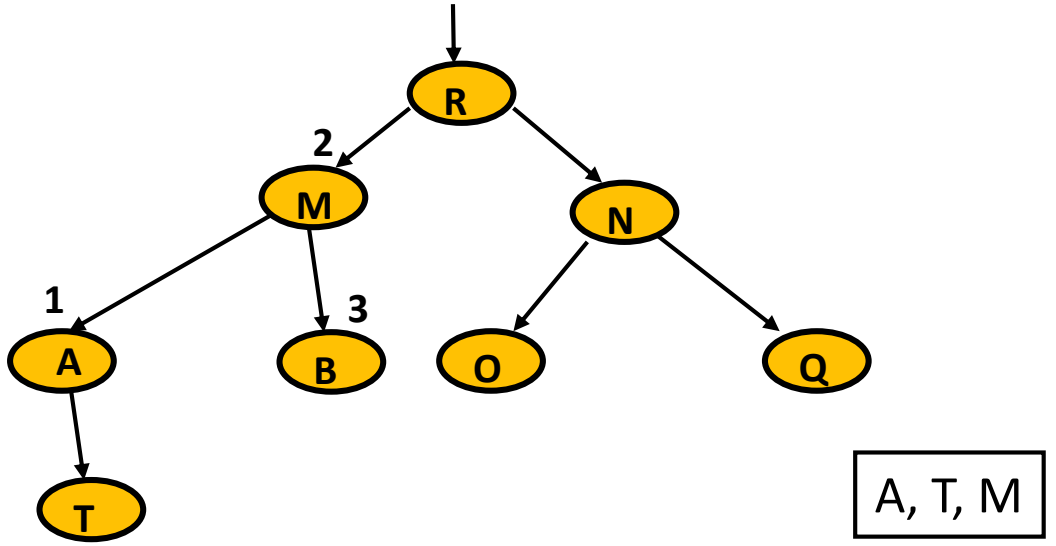
Itérateurs d'arbre

- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



Itérateurs d'arbre

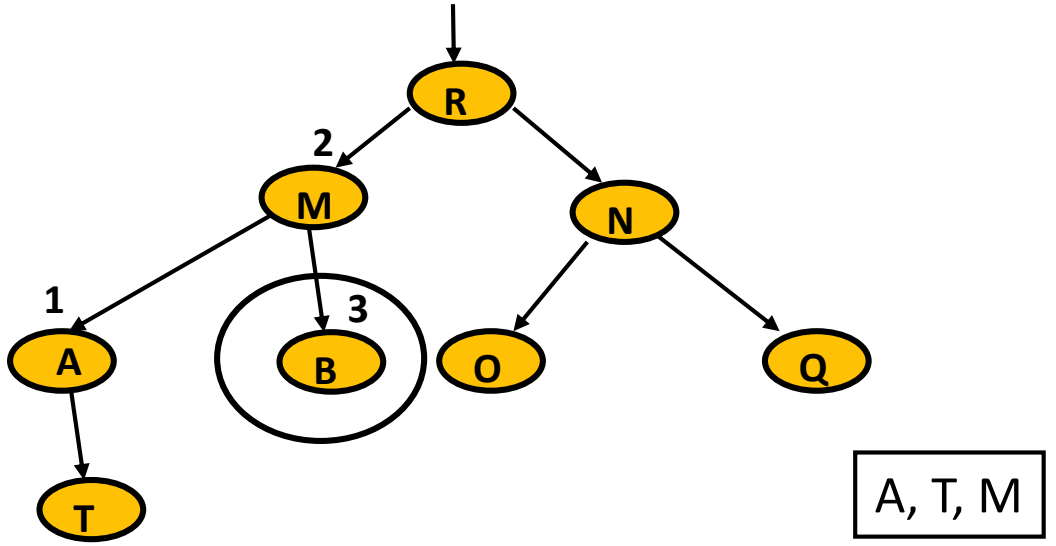
- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



A, T, M

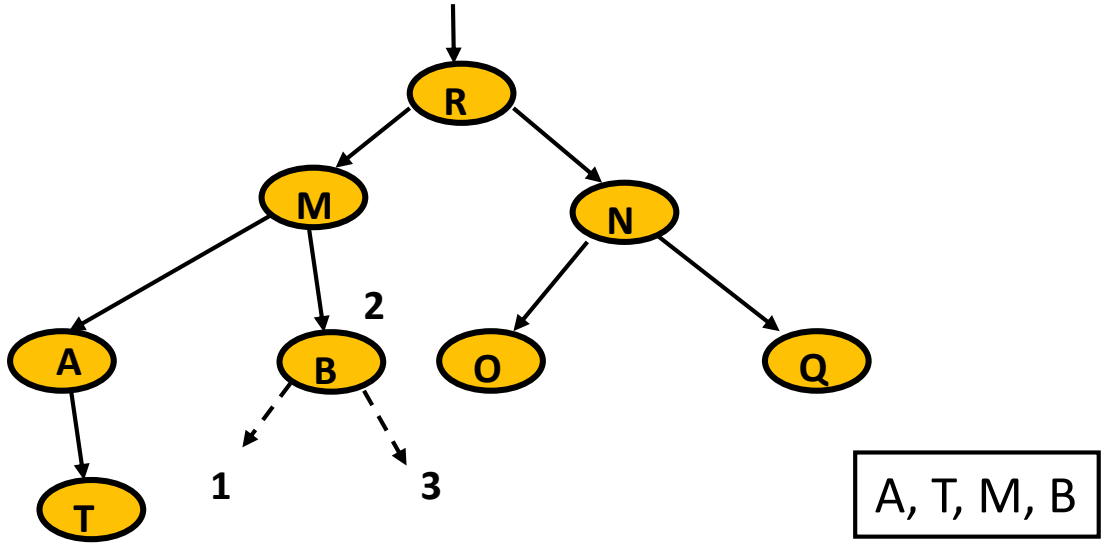
Itérateurs d'arbre

- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



Itérateurs d'arbre

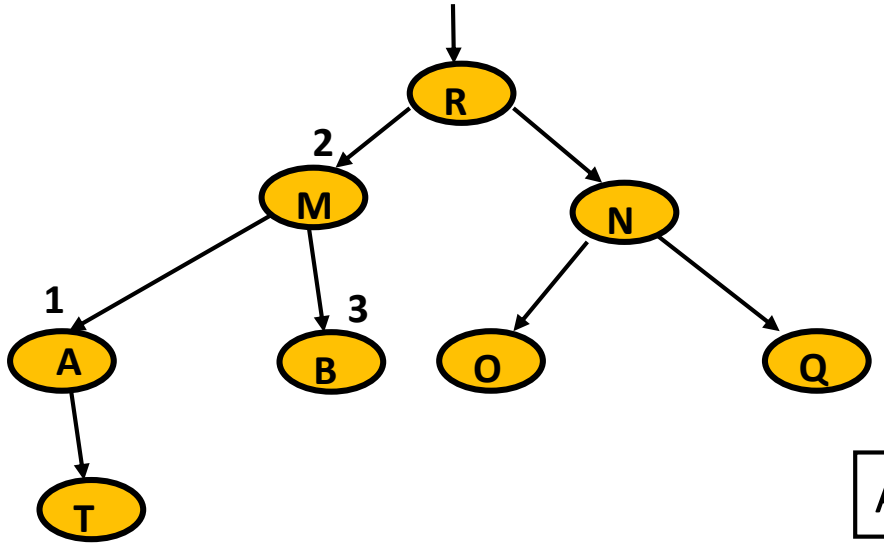
- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



A, T, M, B

Itérateurs d'arbre

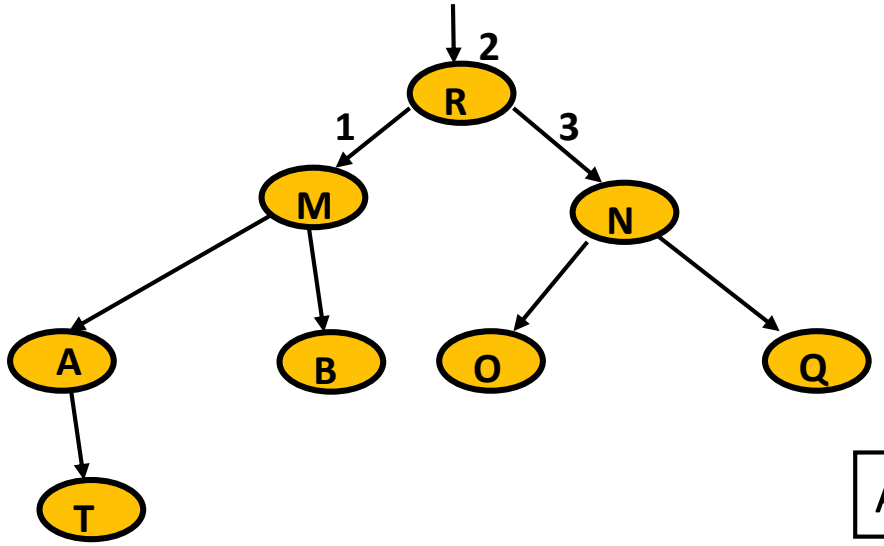
- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



A, T, M, B

Itérateurs d'arbre

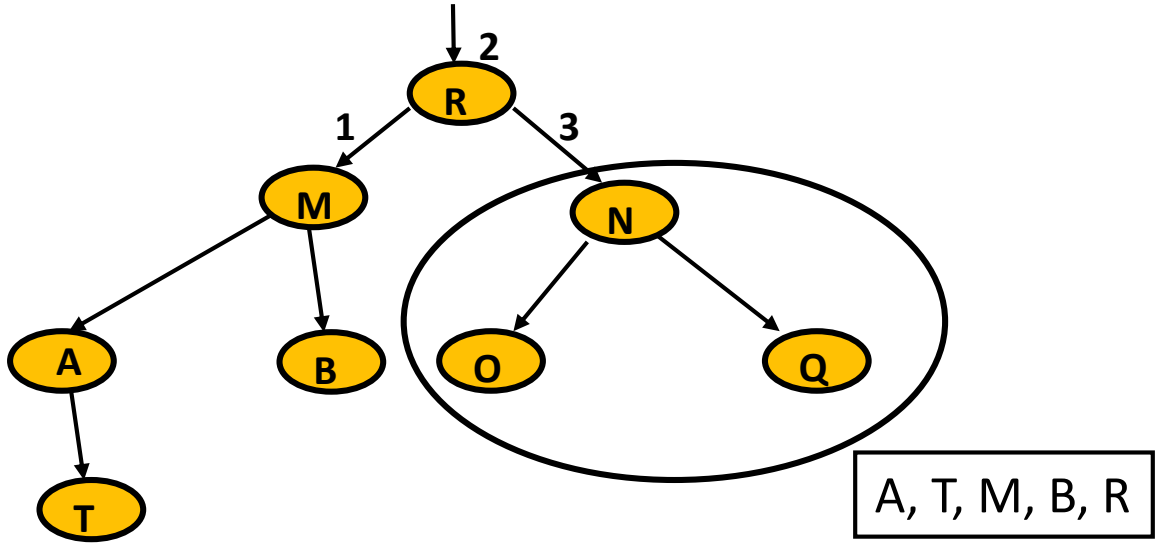
- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



A, T, M, B, R

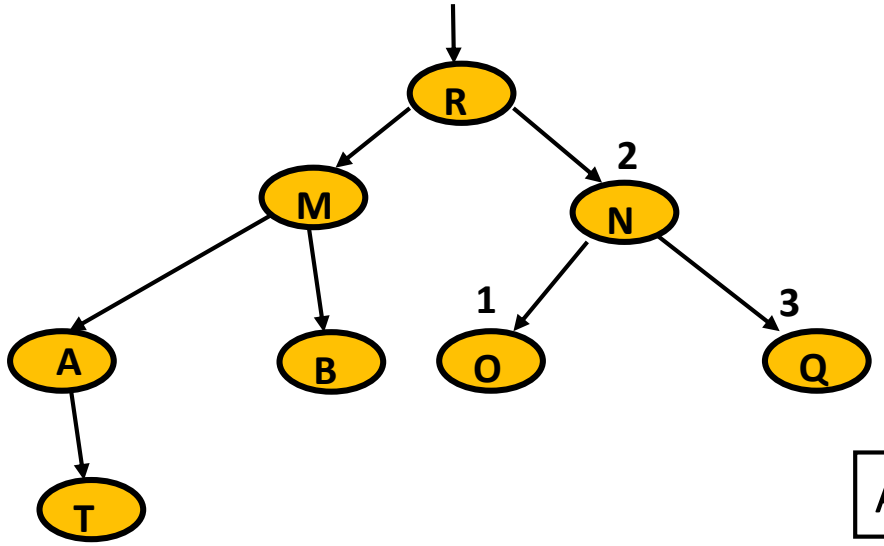
Itérateurs d'arbre

- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



Itérateurs d'arbre

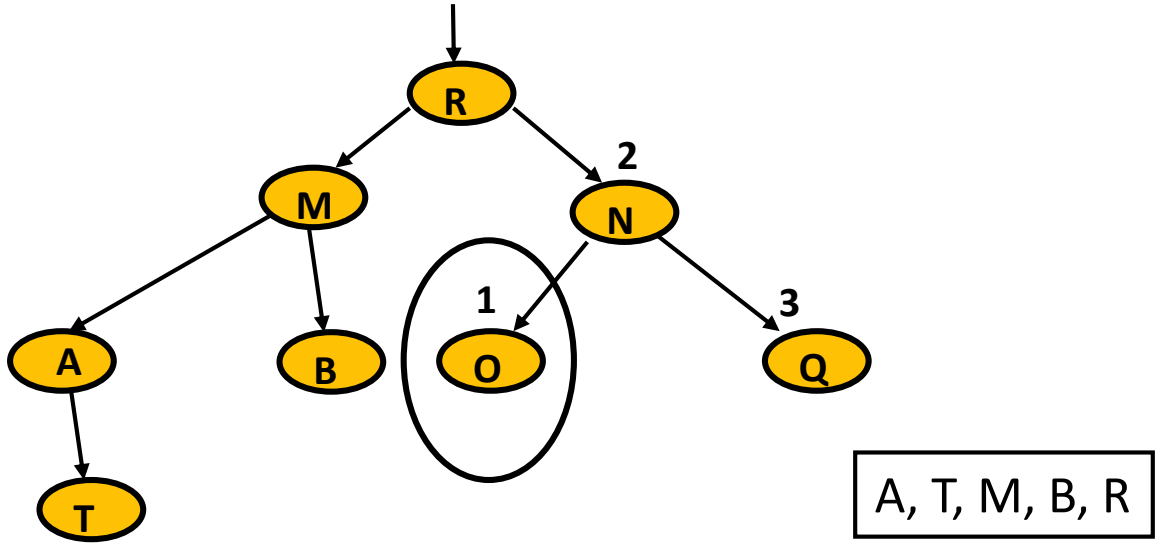
- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



A, T, M, B, R

Itérateurs d'arbre

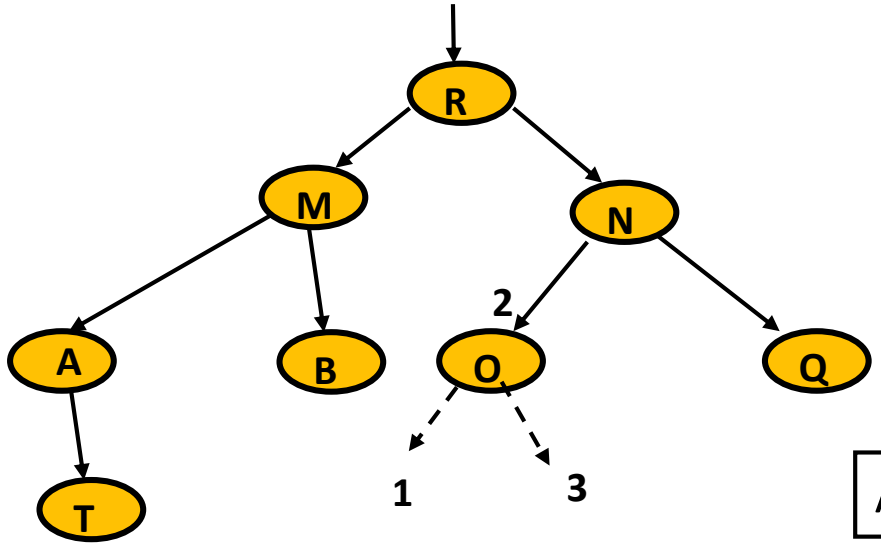
- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



A, T, M, B, R

Itérateurs d'arbre

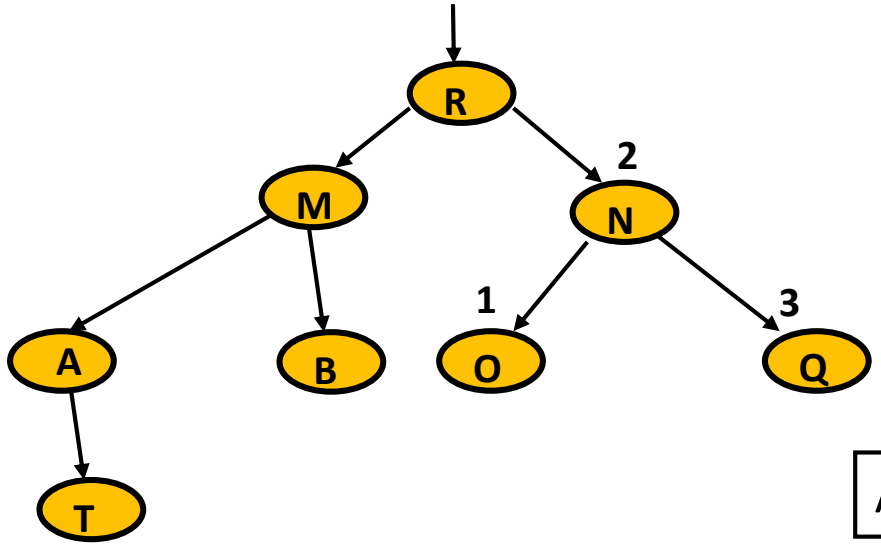
- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



A, T, M, B, R, O

Itérateurs d'arbre

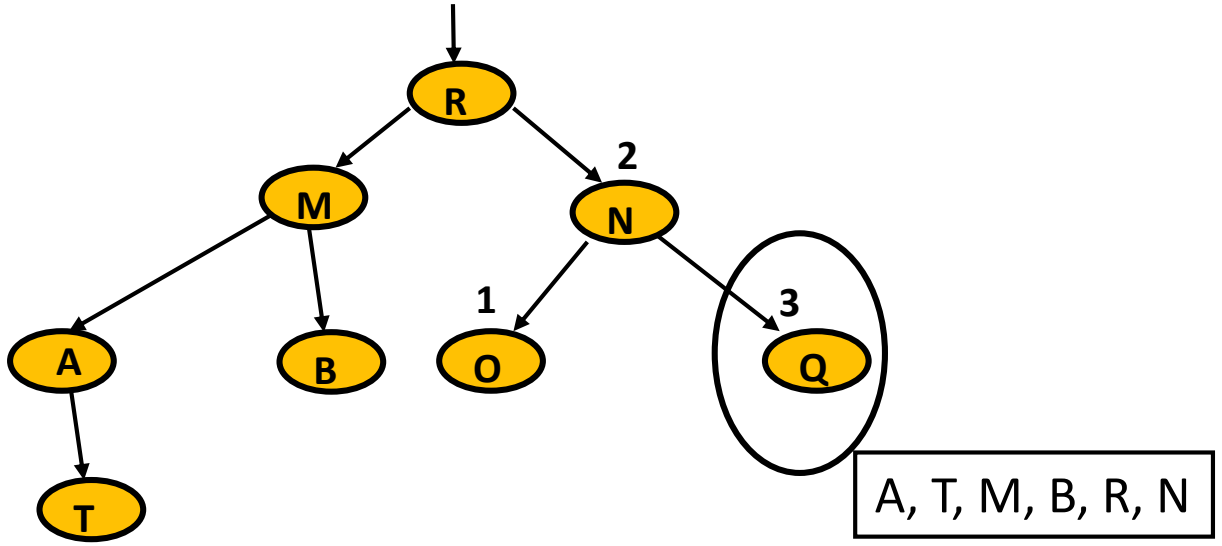
- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



A, T, M, B, R, N

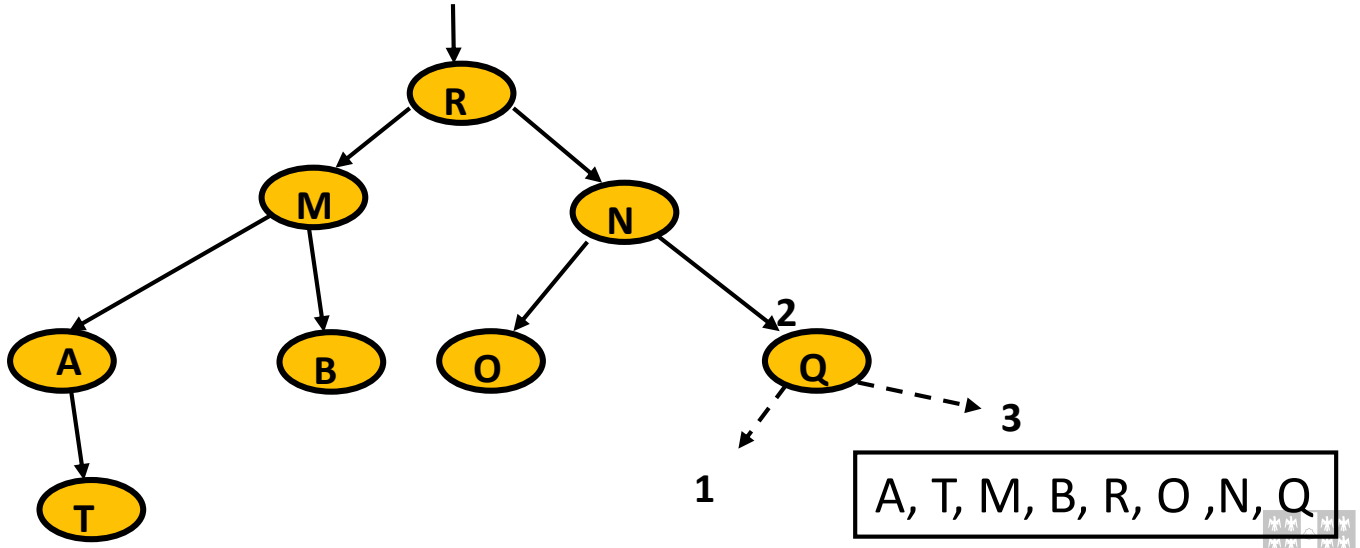
Itérateurs d'arbre

- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



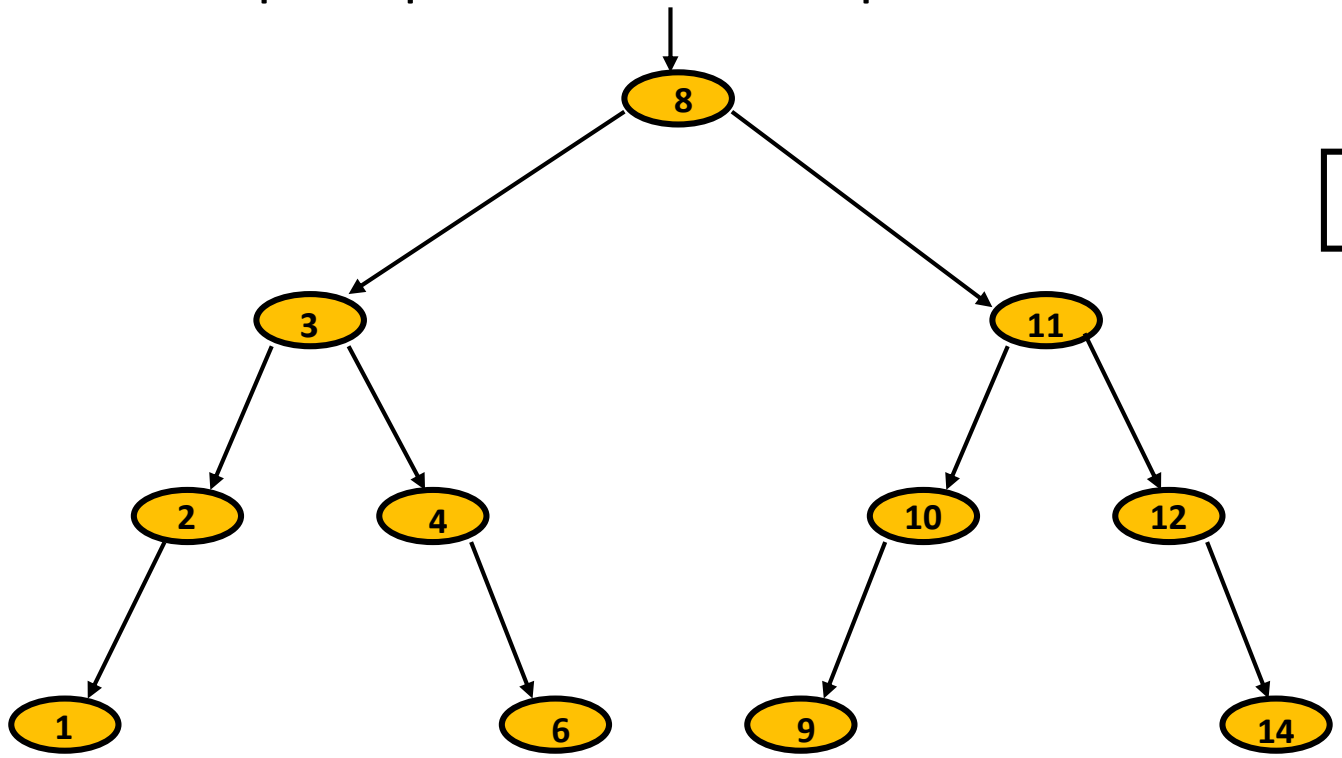
Itérateurs d'arbre

- Parcours **en ordre** : un descendant est traité avant le nœud, l'autre est traité après lui
  - ordre symétrique (en-ordre)
    - ✓ 1. visiter l'enfant de gauche (v_1)
    - ✓ 2. visiter la racine r ;
    - ✓ 3. visiter l'enfant de droite (v_2)



Parcours en-ordre d'un arbre binaire de recherche

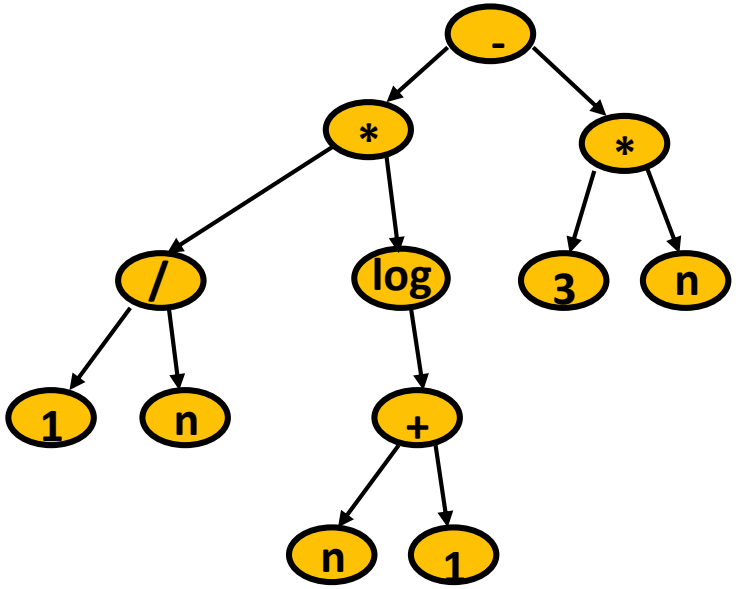
- Donne les éléments triés par ordre croissant des clés!
  - Une des principales caractéristiques des arbres binaires de recherche



1,2,3,4,6,8,9,10,11,12,14

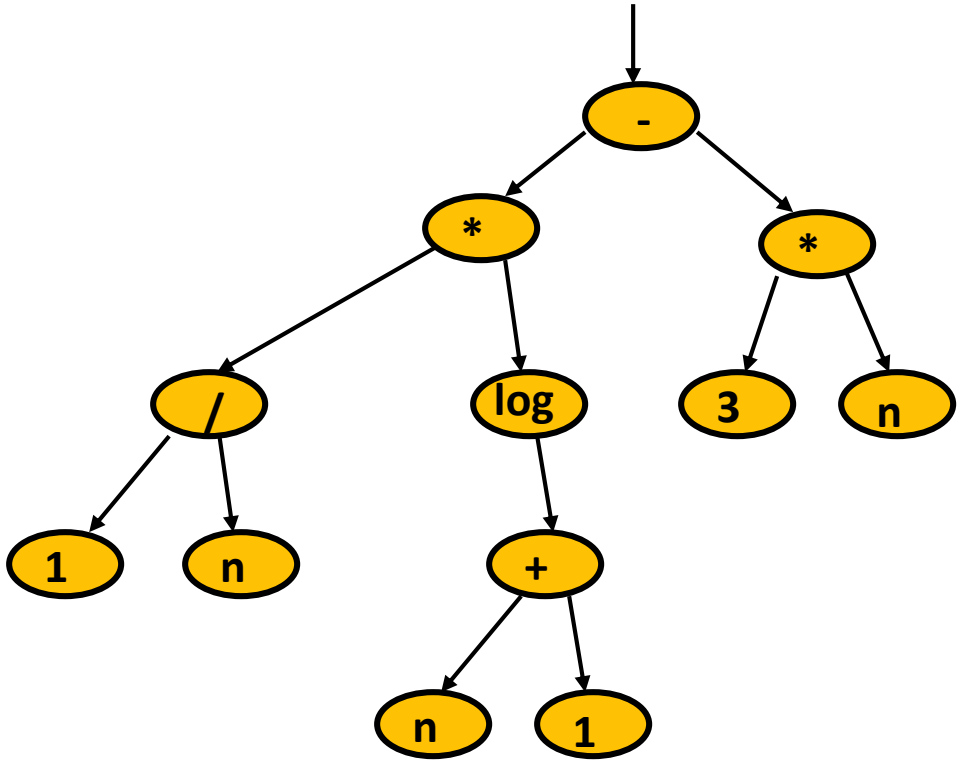
Application: arbres d'expression

- Arbre dont les feuilles sont des opérandes (variables ou constantes) et dont les nœuds internes sont des opérateurs
  - Un opérateur binaire possède deux enfants
  - Un opérateur unaire possède un enfant (de droite)
  - Arbre d'expression pour $((1/n) * (\log(n+1))) - (3 * n)$



Parcours en-ordre de l'arbre d'expression

- Le parcours **en-ordre** donne la représentation **infix** de l'expression

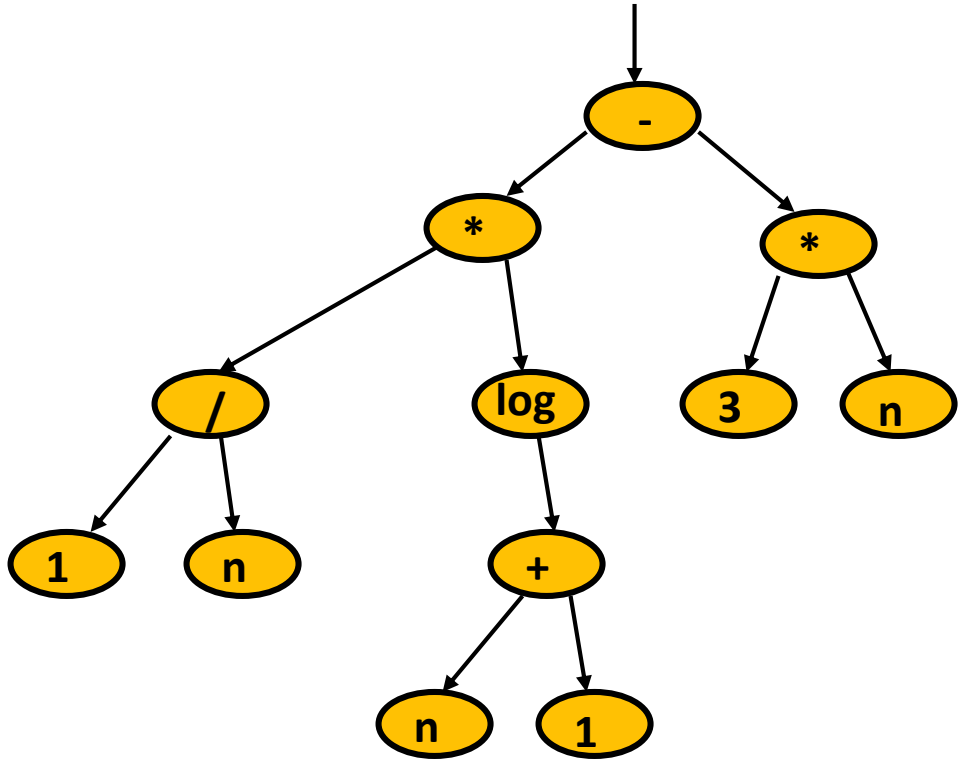


1 / n * log n + 1 - 3 * n

(((1/n) * log(n+1)) - (3*n))

Parcours post-ordre de l'arbre d'expression

- Le parcours **post-ordre** donne la représentation **postfix** de l'expression
  - Aussi appelé la notation Polonaise inversée

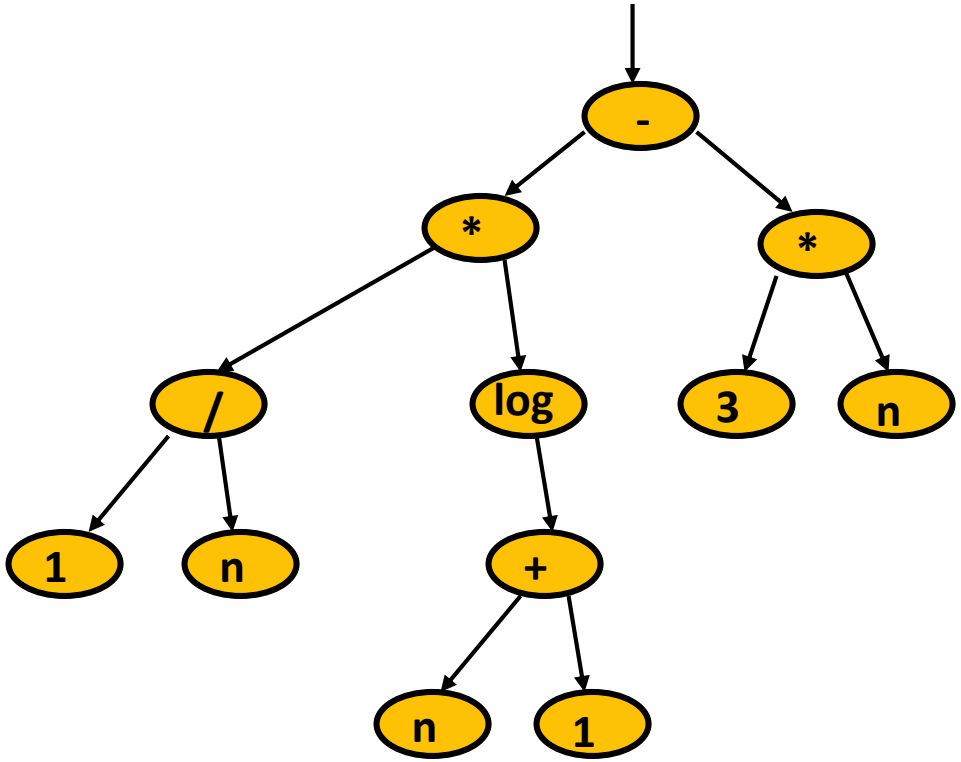


1 n / n 1 + log * 3 n * -

Utilisée pour la première fois par le mathématicien Polonais Jan Lukasiewicz

Parcours pré-ordre de l'arbre d'expression

- Le parcours **pré-ordre** donne la représentation **préfix** de l'expression
  - Pas vraiment utilisé …



- * / 1 n log + n 1 * 3 n

Synthèse

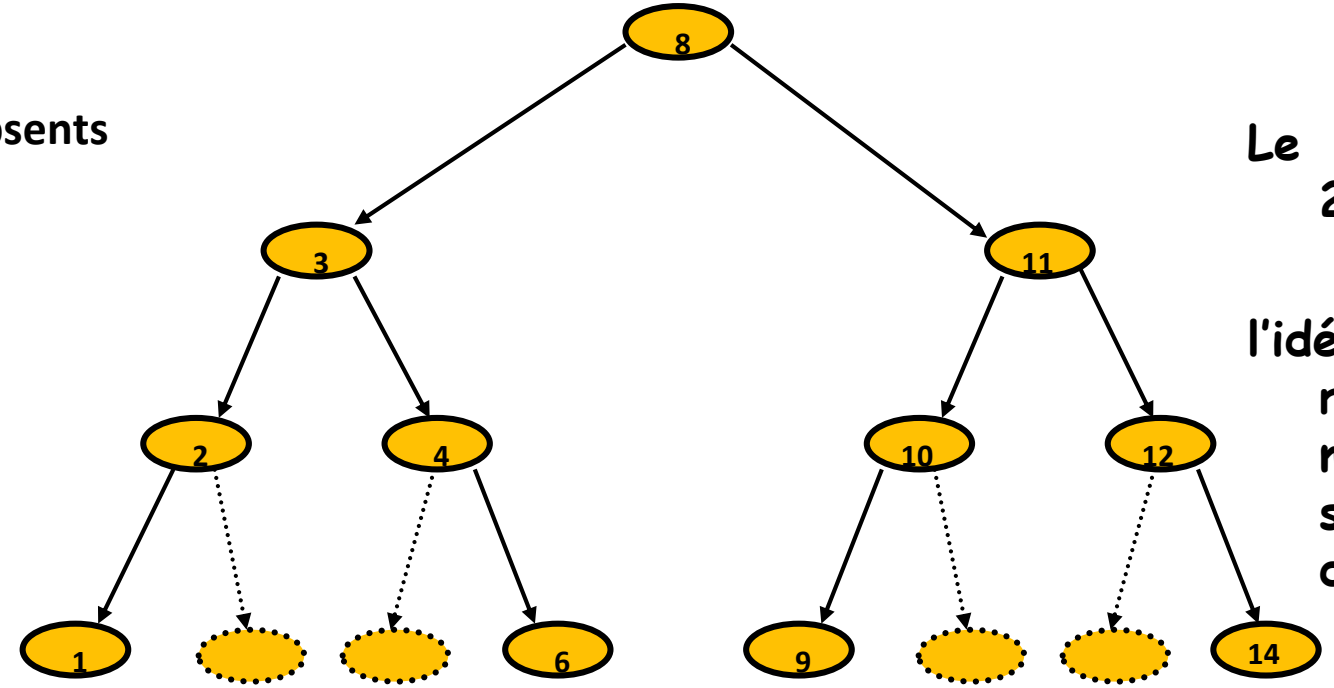
- Parcours d'arbre : visite
- itérateur : définit l'ordre dans lequel seront visités les nœuds
  - en pré-ordre (donne la même chose qu'un PP ou DFS)
  - en post-ordre
  - en-ordre (symétrique)
- Applications
  - Parcours en-ordre d'un arbre binaire de recherche : ordre croissant des clés
  - Parcours post-ordre de l'arbre d'expression : représentation postfix de l'expression (notation Polonaise inversée)
  - Parcours pré-ordre : représentation préfix de l'expression
    - ✓ (Pas vraiment utilisé …)

Implémentations des arbres

- Deux grandes méthodes d'implémentation.
  - dans un tableau
    - ✓ Les nœuds sont insérés dans un tableau
    - ✓ Aucune utilisation de pointeurs
    - ✓ La position des enfants est obtenu par une opération arithmétique très simple
    - ✓ Utilisé pour les monceaux/tas (objet de ce chapitre)
  - par chaînage
    - ✓ Chaque nœud pointe sur ses enfants (un pointeur par enfant)
    - ✓ Utilisé pour les arbres binaires de recherche (prochain chapitre)

Implémentation en tableau des arbres binaires

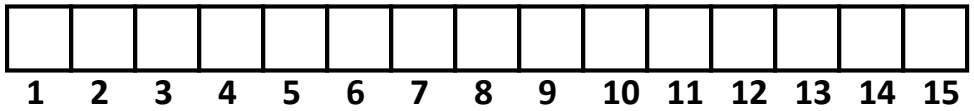
: nœuds absents



Le niveau h peut contenir jusqu'à 2^h nœuds (pour h = 0, 1, 2 …)

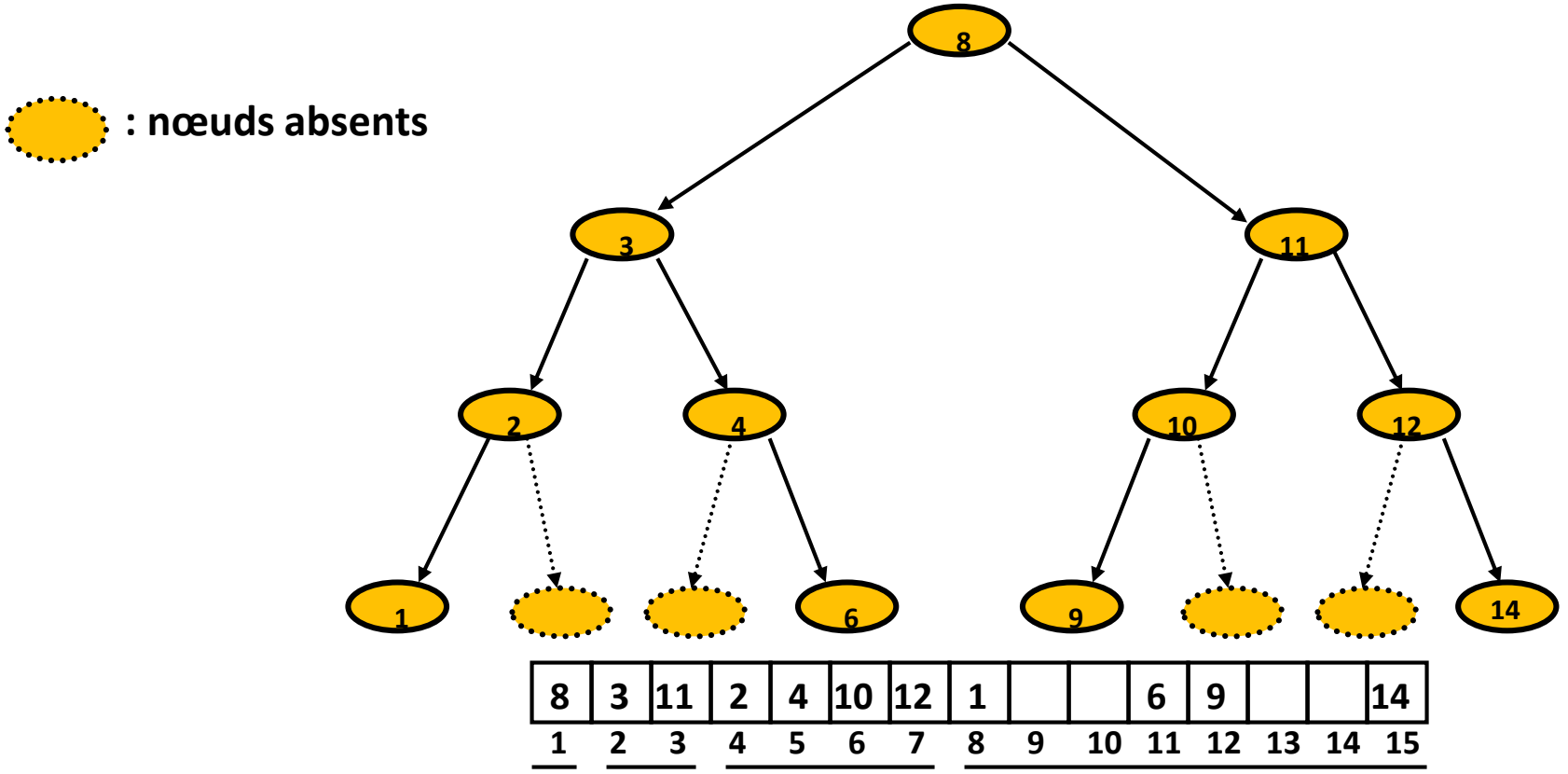
l'idée: en débutant par h=0 (la racine), on réserve 2^h cases mémoires consécutives pour stocker les noeuds du niveau h de gauche à droite

Comment faire?



Implémentation en tableau des arbres binaires

- Emplacement des éléments dans le tableau:
  - (les nœuds absents ne sont donc pas stockés dans l'espace mémoire réservé)

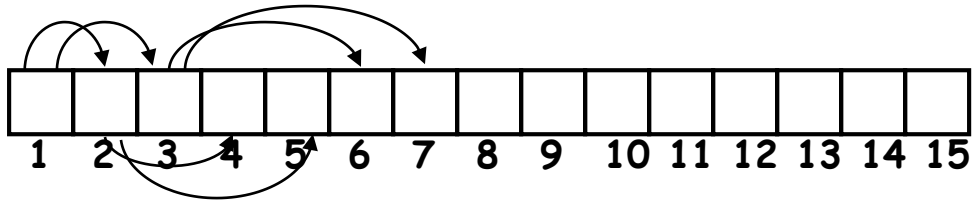


Implémentation en tableau des arbres binaires

- Pour faciliter la description des algorithmes
  - convention : le premier indice du tableau est 1
    - ✓ (corriger pour l'implémentation des algorithmes en C++)
- L'indice du 1er nœud du niveau h est donné par (voir page suivante) :
  - $2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} + 1 = 2^h$
- Le ième nœud du niveau h a donc pour indice $2^h + (i-1) \equiv j$.

Implémentation en tableau des arbres binaires

- Les enfants de ce nœud sont à la position $2i - 1$ et $2i$ du niveau $h + 1$.
- L'indice de l'enfant gauche est alors $2^{h+1} + (2i - 1) - 1 = 2(2^h + (i - 1)) = 2j$
- L'indice de l'enfant droit est alors = $2j + 1$.
- Conclusion: Les enfants du nœud en position j se trouvent respectivement aux positions $2j$ et $2j + 1$ (s'ils existent)



Pause Math: séries géométriques

- Pourquoi avons-nous $2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} = 2^h - 1$
  - Cette série est un cas particulier de la série géométrique.

$$S = \sum_{i=0}^{n} r^i$$

- Nous avons alors :

$$rS = \sum_{i=1}^{n+1} r^i = S + r^{n+1} - 1$$

Pause Math: séries géométriques

$$rS = \sum_{i=1}^{n+1} r^i = S + r^{n+1} - 1$$

- Alors

$$S(r-1) = r^{n+1} - 1$$

- Donc

$$S = \frac{r^{n+1} - 1}{r - 1}$$

- Ce qui donne le résultat avec $r = 2$ et $n = h - 1$

Pause math: fonctions floor() et ceiling() (Révision)

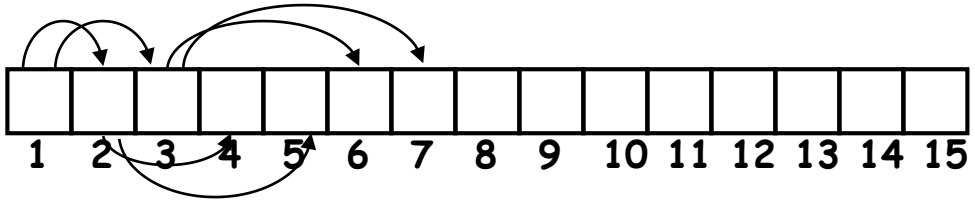
- Pour la suite de ce chapitre, nous utiliserons ces fonctions.
  - floor(x) = $\lfloor x \rfloor$ = le plus grand entier ≤ x (partie entière inférieure ou partie entière)
    - ✓ Exemples:
      - $\lfloor 3.72 \rfloor$ = 3
      - $\lfloor 3 \rfloor$ = 3
  - ceiling(x) = $\lceil x \rceil$ = le plus petit entier ≥ x (partie entière supérieure ou par excès)
    - ✓ Exemples:
      - $\lceil 3.44 \rceil$ = 4
      - $\lceil 3 \rceil$ = 3

Pause math: fonctions floor() et ceiling()

- Propriétés:
  - Pour tout réel x: $\lfloor x \rfloor \leq x \leq \lceil x \rceil < x+1$
  - Pour tout entier positif n: $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$
  - Preuve pour n impair: $\lfloor n/2 \rfloor = (n-1)/2$ et $\lceil n/2 \rceil = (n+1)/2$
  - En C++: pour tout entiers $n \geq 0$ et $k \geq 1$, on a:
  - $\lfloor n/k \rfloor$ s'écrit n/k (quotient de la division entière)
  - $\lceil n/k \rceil$ s'écrit n%k==0 ? n/k : n/k + 1

Positionnement des parents

- Les enfants du nœud en position j se trouvent aux positions $2j$ et $2j + 1$ s'ils existent.
  - Alors le parent du nœud en position i se trouve en position $\lfloor \boldsymbol{i/2} \rfloor$



Parcours en-ordre pour un arbre binaire implémenté dans un vecteur

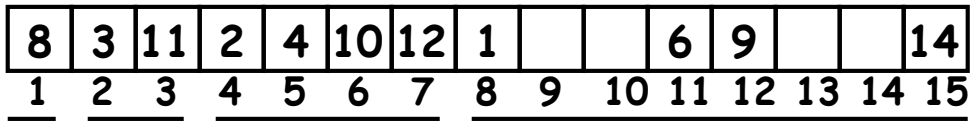
- **Attention: en C++ les indices débutent à la position 0 .**
  - Si les enfants du parent i étaient en positions $2i$ et $2i + 1$.
  - Position en C++ de ce parent : en $j = i - 1$
  - Ceux de ses enfants : en $j' = (2i) - 1$ et $j'' = 2i$. Puisque $i = j + 1$, ces enfants sont positionnés en $j' = 2j + 1$ et $j'' = 2j + 2$.

Implémentation C++

```
template <typename T>
void Arbre<T>:: _affiche(size_t j) const
{
	if (j < v.size()) //v est le vecteur contenant les clés
	{ // afficher l'enfant de gauche
		_affiche(2*j + 1);
		// afficher le nœud j s'il est présent
		if(v[j]!=-1)
		{
			std::cout << v[j] << " ";
		}
		// afficher l'enfant de droite
		_affiche(2*j + 2);
	}
}
template <typename T>
void Arbre<T>::affiche() const { _affiche(0);}
```

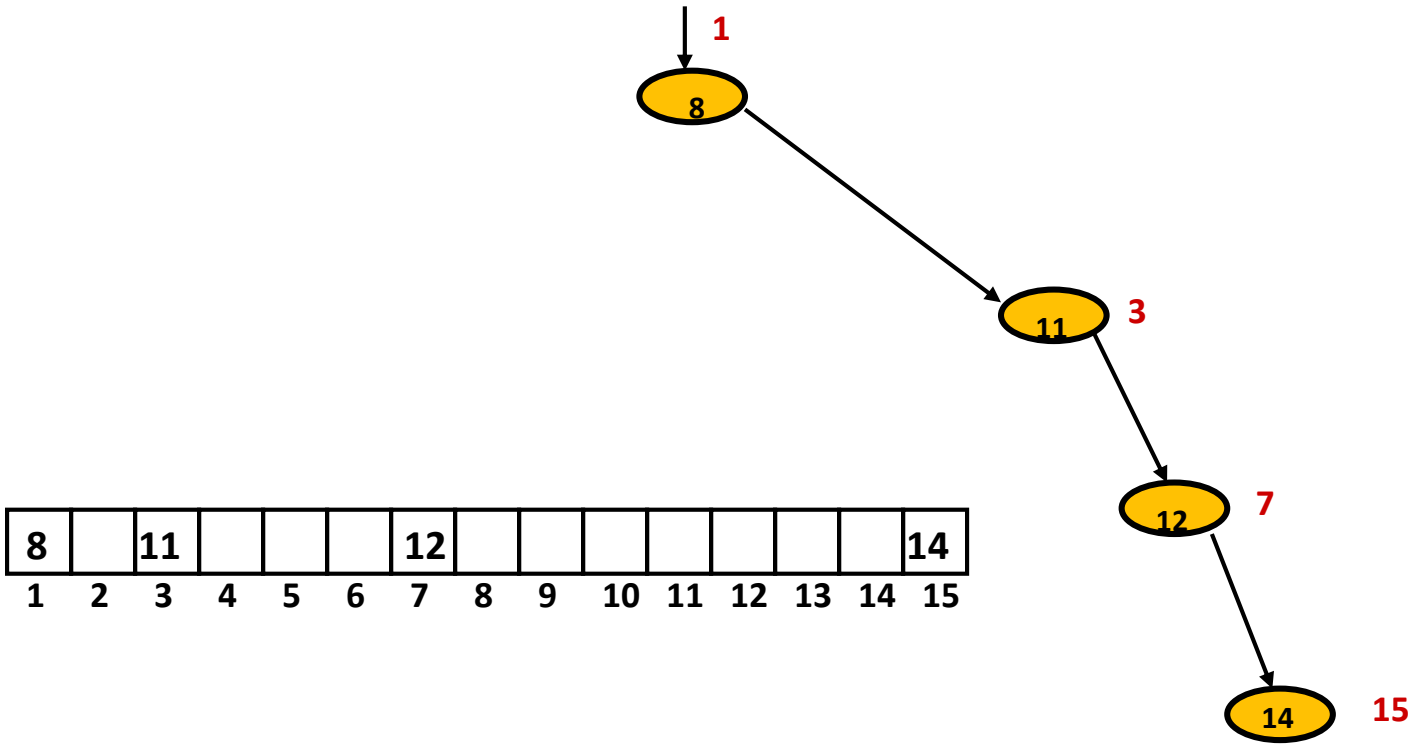
Implantation en tableau

- Avantages :
  - Simplicité pour visiter les enfants
  - Aucun espace utilisé pour stocker des pointeurs
  - L'espace pour insérer un nœud est déjà disponible
- Désavantages :
  - espace perdu pour les trous
  - Ré-allocation d'un tableau plus grand si la position du nœud que l'on désire ajouter déborde du tableau



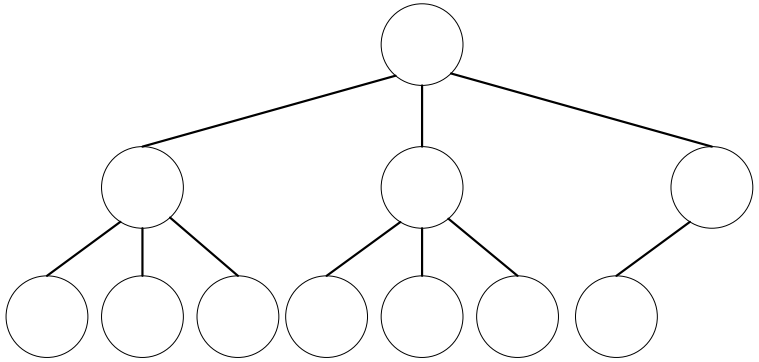
Implantation en tableau

- Pire cas : arbre dégénéré vers la droite

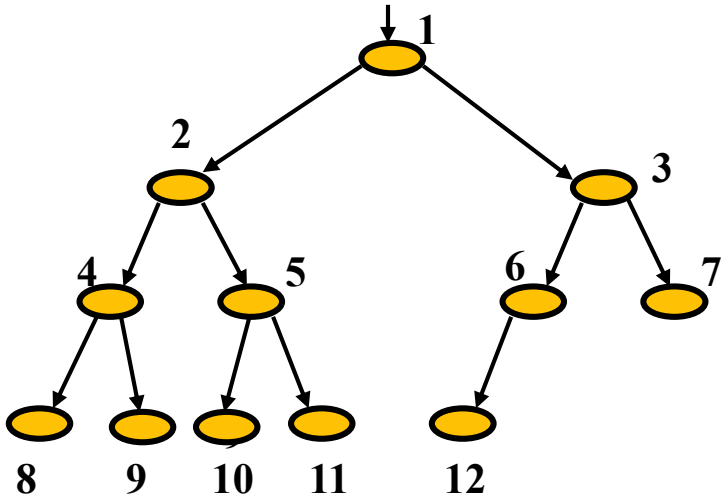


Arbre feuillu ou complet

- **Arbre complet**: Un arbre de degré n est dit complet lorsque tous ses niveaux possèdent un nombre maximal de nœuds, sauf possiblement le dernier, auquel cas ce dernier niveau est rempli de gauche à droite, sans trou.



Arbre de degré 3 complet



Arbre de degré 2 complet

Synthèse

- Implémentations des arbres
  - par chaînage
  - dans un tableau
    - ✓ Le niveau h peut contenir jusqu'à 2h nœuds (pour h = 0, 1, 2 …)
    - ✓ description des algorithmes : convention le premier indice du tableau est 1
    - ✓ L'indice du 1er nœud du niveau h est donné par :
      - $2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} + 1 = 2^h$
    - ✓ Les enfants du nœud en position j se trouvent respectivement aux positions $2j$ et $2j + 1$ (s'ils existent)
    - ✓ le parent du nœud en position i se trouve en position $\lfloor \boldsymbol{i/2} \rfloor$
  - Avantages/inconvenients
  - Arbre dégénéré
  - Arbre feuillu ou complet

Définition du tas/monceau (« heap »)

- Arbre binaire complet dont la valeur de la clé d'un nœud est toujours supérieure ou égale à celle de ses enfants (propriété du tas_max)
  - ➢ Valeur de la racine : valeur maximale du tas

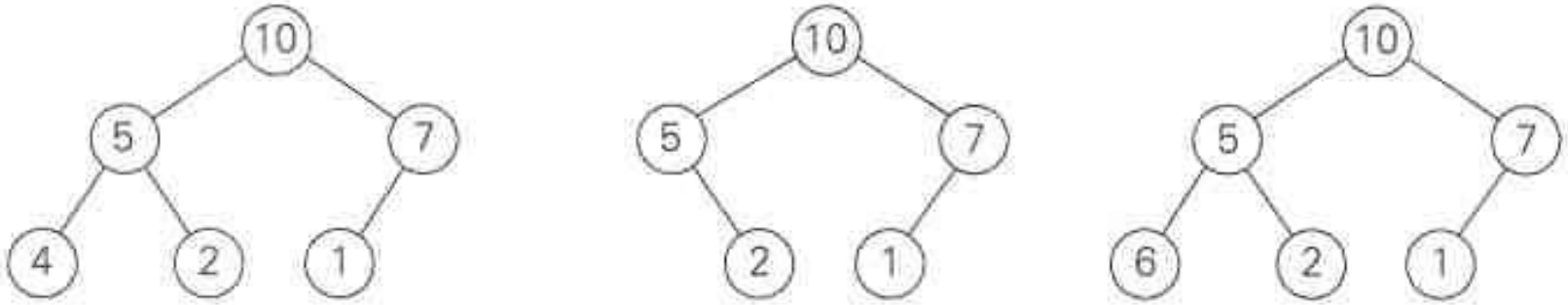


FIGURE 6.9 Illustration of the definition of "heap": only the leftmost tree is a heap.

[source: Introduction to the design & analysis of Algorithms, Anany Levitin, Pearson 2003, ISBN 0-201-74395-7.]

Utilité des tas

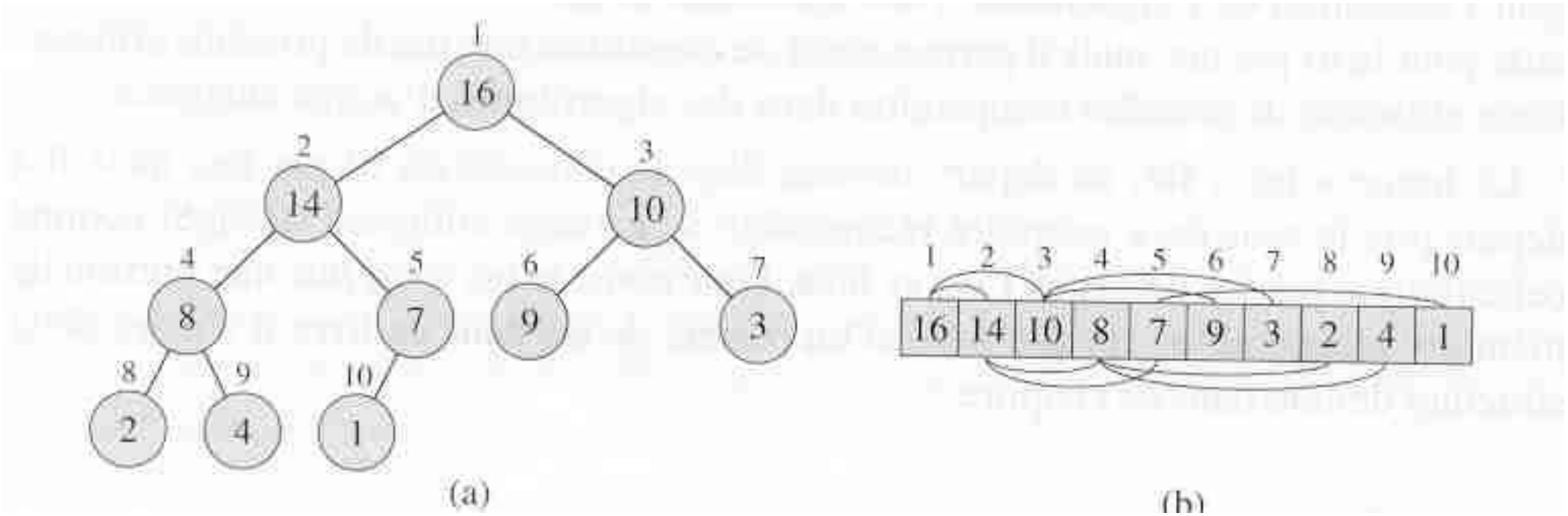
- Structure de données utilisée pour la mise en œuvre des files de priorité.
  - Exemples:
    - ✓ File d'impression de fichiers (fichiers plus courts d'abord)
    - ✓ Utilisation pour algorithmes glouton (ex: Dijkstra)
  - File de priorité : doit supporter efficacement les opérations suivantes:
    - ✓ Trouver un item avec la priorité la plus élevée
    - ✓ Enlever un item ayant la priorité la plus élevée
    - ✓ Ajouter un item à la file de priorité
- Structure intermédiaire utilisée par l'algorithme du tri par tas («heapsort»)
  - Tableau d'abord transformé en un tas
  - Éléments ensuite repositionnés (rapidement) en ordre croissant.
  - Tableau trié.

Le tas et son tableau associé

- Les éléments (valeurs des clés) du tas sont positionnés dans un tableau $H[1..n]$ comme suit:
  - La valeur de la racine est placé en $H[1]$
  - Le premier élément du niveau suivant est placé en $H[2]$
  - Le second élément de ce niveau est placé en $H[3]$ …
  - L'assignation se fait donc du niveau supérieur au niveau inférieur en balayant chaque niveau de gauche à droite
  - Pas de trou dans le tableau: implémentation efficace.

Le tas et son tableau associé

- Exemple



La hauteur d'un tas de n noeuds

- Considérez un tas de n nœuds possédant l nœuds au dernier niveau (le niveau h). Nous avons alors:
  - n = $2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} + l$ avec: $1 \leq l \leq 2^h$
    - ✓ Alors: $n = 2^h - 1 + l$
  - Alors:
    - ✓ $l \geq 1 \Rightarrow n \geq 2^h \Rightarrow h \leq log_2(n)$
- De plus:
  - $l \leq 2^h \Rightarrow n \leq 2^h - 1 + 2^h = 2^{h+1} - 1 \Rightarrow n < 2^{h+1} \Rightarrow log_2(n) < h + 1$
- Alors: $h \leq log_2(n) < h + 1 \Rightarrow \lfloor log_2(n) \rfloor = h$
- La hauteur h d'un tas de n nœuds est alors donné par:
  - h = $\lfloor log_2(n) \rfloor$

Le nombre de feuilles d'un tas de n noeuds

- **Théorème**: dans un tas de n nœuds, le nombre f de feuilles et le nombre p de parents (nœuds internes) sont donnés resprectivement par f = $\lceil \boldsymbol{n/2} \rceil$ et p = $\lfloor \boldsymbol{n/2} \rfloor$.
- Preuve:
  - Si le premier noeud du tas est en position 1, le dernier noeud du tas est en position n.
  - Le dernier parent du tas est le parent du noeud en position n.
  - Le dernier parent du tas est donc en position $\lfloor n/2 \rfloor$
  - Puisque tous les nœuds qui précèdent le dernier parent sont également des parents, alors p = $\lfloor n/2 \rfloor$
  - Puisque n = p + f, alors f = n - $\lfloor n/2 \rfloor$ = $\lceil n/2 \rceil$ CQFD.

Construction d'un tas du bas vers le haut

- Pour que le tableau $H[1..n]$ initial soit un tas il faut satisfaire
  - $H[i] \geq H[2i]$ et $H[i] \geq H[2i+1]$ pour $i = 1 \ldots \lfloor n/2 \rfloor$ (les nœuds parents).
  - Permuter certains éléments pour obtenir cette propriété.
- L'algorithme de construction débute avec le (dernier) parent i = $\lfloor n/2 \rfloor$.
- Si $H[i] < \max\{H[2i], H[2i+1]\}$, on «swap» $H[i]$ avec $\max\{H[2i], H[2i+1]\}$
- On recommence avec le parent $i-1$ jusqu'à la racine $(i = 1)$.
- Lorsque l'on interchange $H[i]$ avec l'un de ses enfants $H[j]$ il faut recommencer cette procédure avec cet enfant $H[j]$ (et non avec l'autre).
- Il y a percolation vers le bas d'une valeur dans le tas.

Construction d'un tas du bas vers le haut

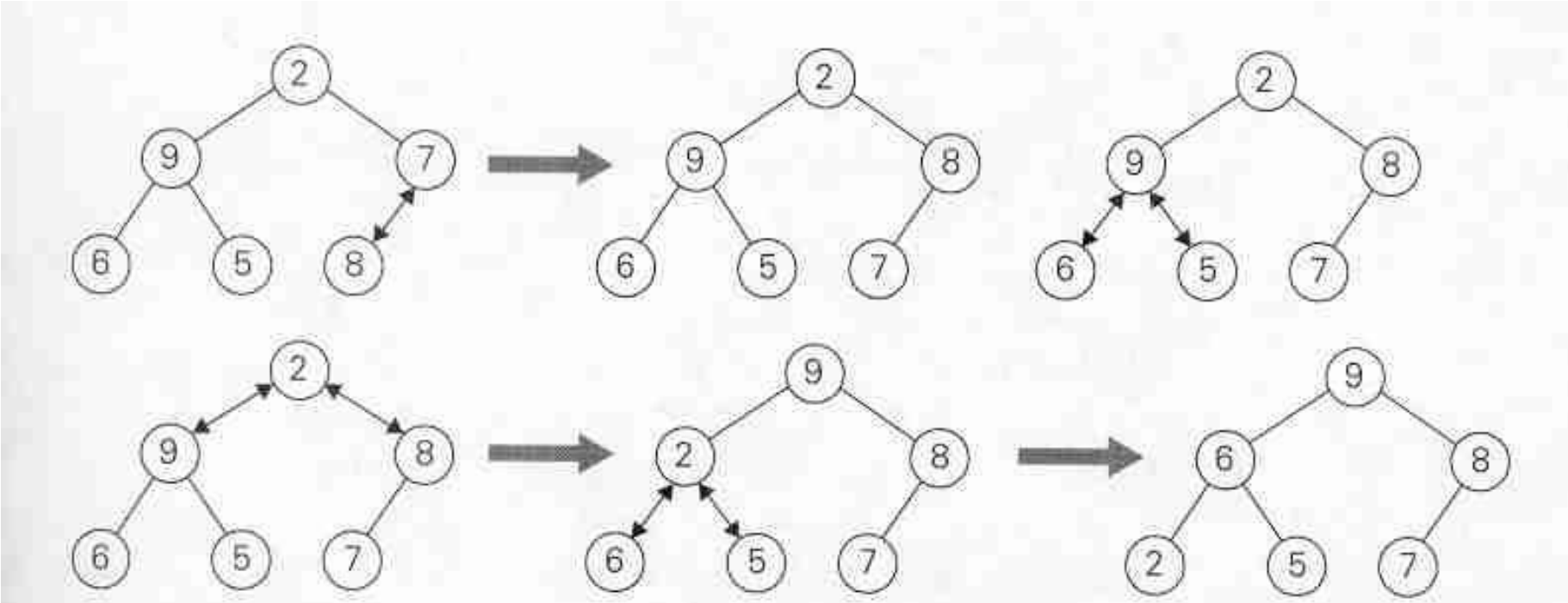
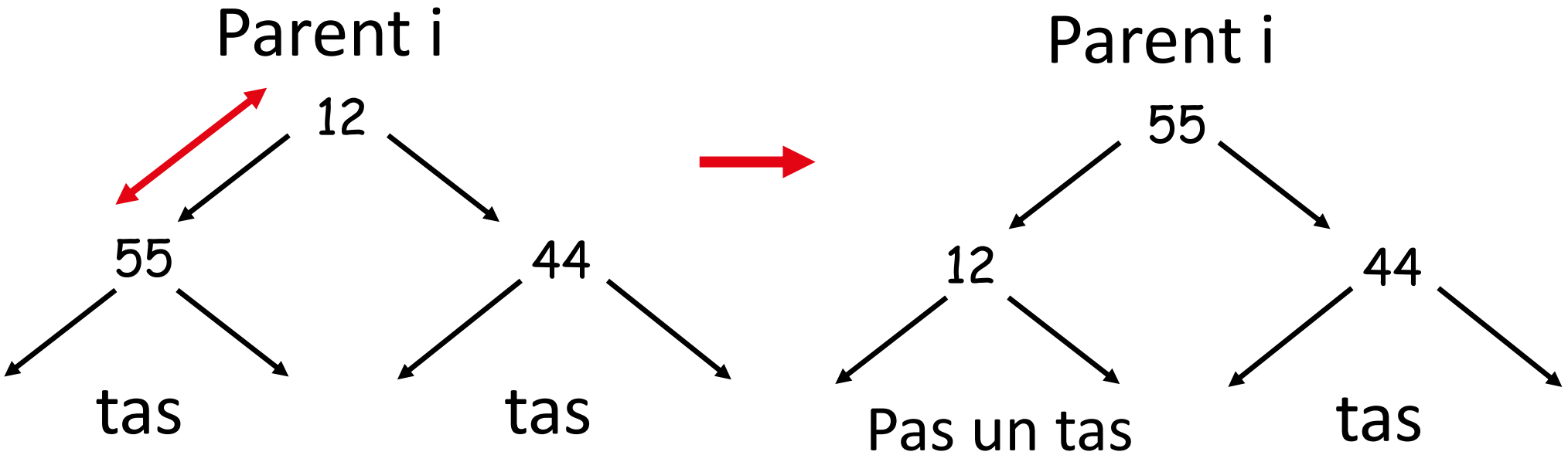


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8

Exactitude de la Construction d'un tas du bas vers le haut

- L'exactitude de cet algorithme vient du fait que pour chaque parent i que l'on visite, du bas vers le haut, on a que les sous arbres de leurs enfants sont des tas avant la percolation vers le bas.



Exactitude de la Construction d'un tas du bas vers le haut

- Si le sous arbre du parent i n'est pas un tas
  - ➢ dû uniquement à la valeur de sa clé
- En swappant la valeur de sa clé avec celle de son enfant maximal, on impose à cet enfant une valeur de clé plus faible que celle qu'il avait précédemment, mais la clé du parent i devient la plus élevé parmi les clés du sous arbre i.
  - ➢ Nous nous retrouvons alors dans la même situation que celle du parent i, mais pour le sous arbre de l'enfant maximal.
- La hauteur du sous arbre du parent i étant fini,
  - ➢ implique qu'il sera un tas après avoir percolé vers le bas la valeur de sa clé.

Construction du tas, bas vers le haut

```
template <typename Comparable>
void heapBottomUp( vector<Comparable>& heap)
{
	if (heap.size() <= 1) return; //car on a terminé
	size_t lastParent = heap.size()/2 - 1; //car les indices débutent à 0
	for(size_t i = lastParent; ; i-- ) //pas de critère d'arrêt ici
	{
		percDown(heap, i, heap.size());
		if (i==0) break;
	}
}
```

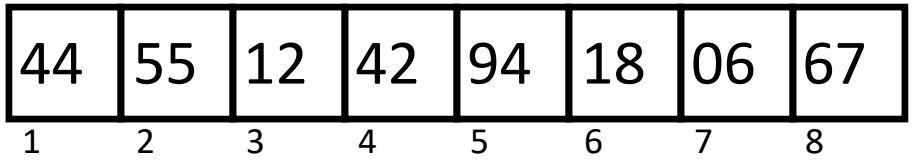
Construction du tas, bas vers le haut

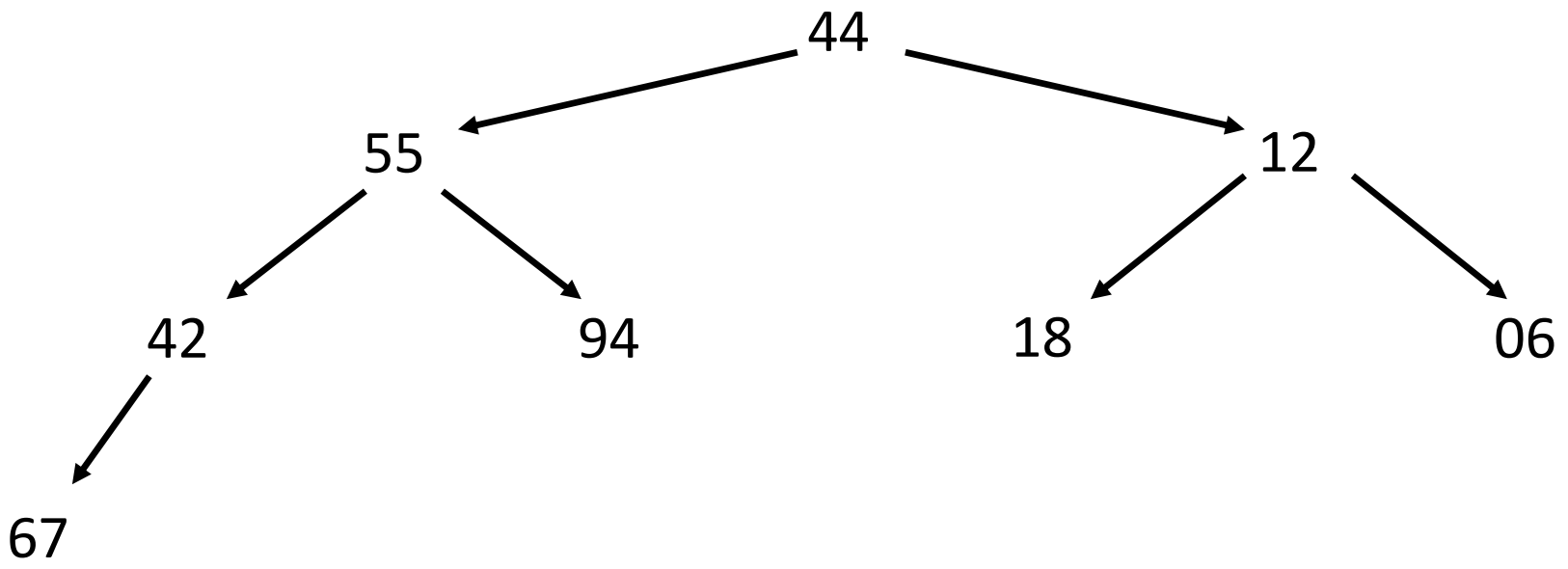
```
size_t leftChild(size_t i) { return 2 * i + 1; }

template <typename Comparable>
void percDown( vector<Comparable> & heap, size_t i, size_t n )
{
        size_t child;
        while(leftChild(i) < n) //tant que heap[i] a un enfant
        {
                child = leftChild(i);
                if(child < n - 1 && heap[child] < heap[child + 1])
                        child++; //child est l'indice du plus grand enfant
                if(heap[i] < heap[child]) //alors interchanger et recommencer avec l'enfant
                {
                        std::swap( heap[i], heap[child] ;
                        i = child;
                }
                else break; //sinon retourner, il n'y a plus rien à faire
        }
}
```

Construction d’un tas bas vers le haut

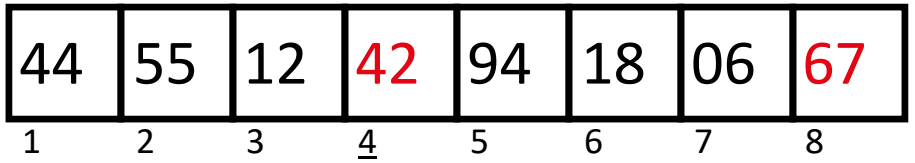
Création du monceau tas-max

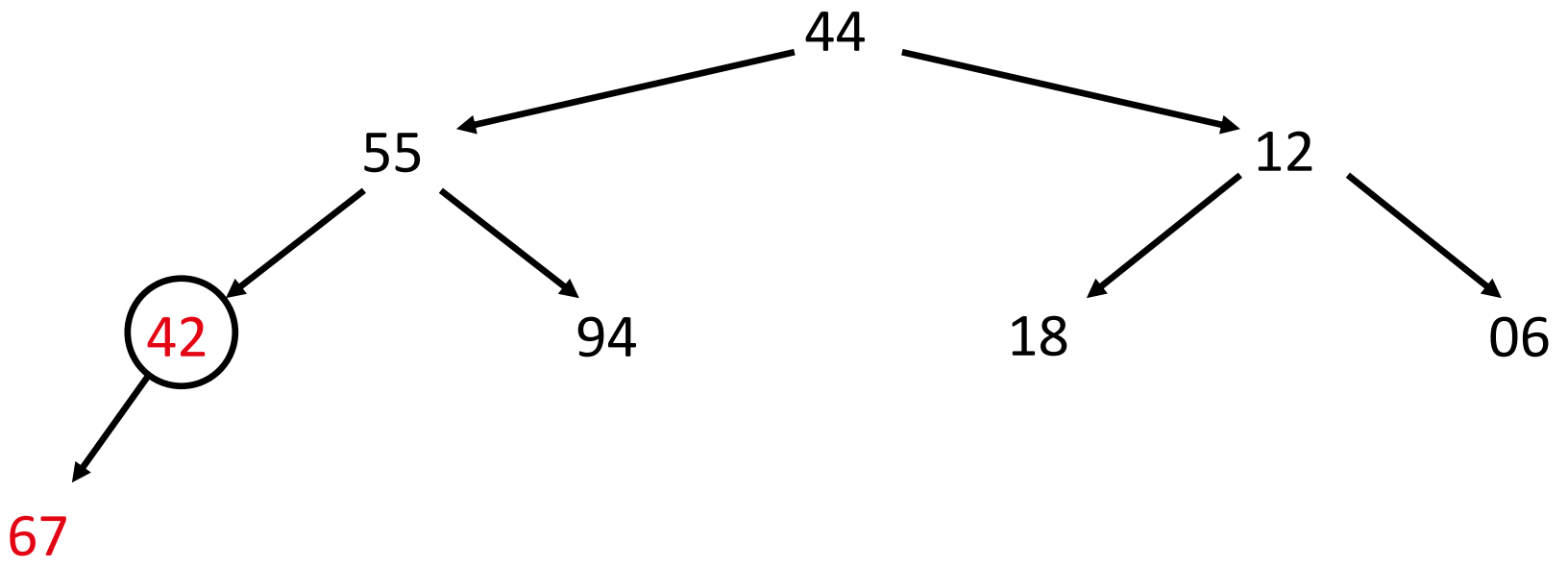




Construction d'un tas bas vers le haut

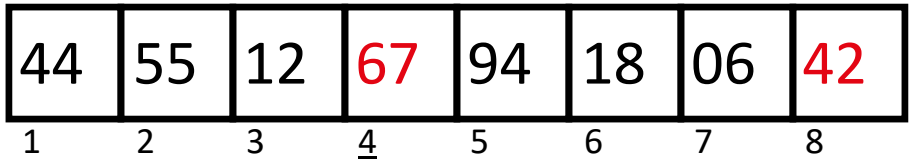
Création du monceau tas-max

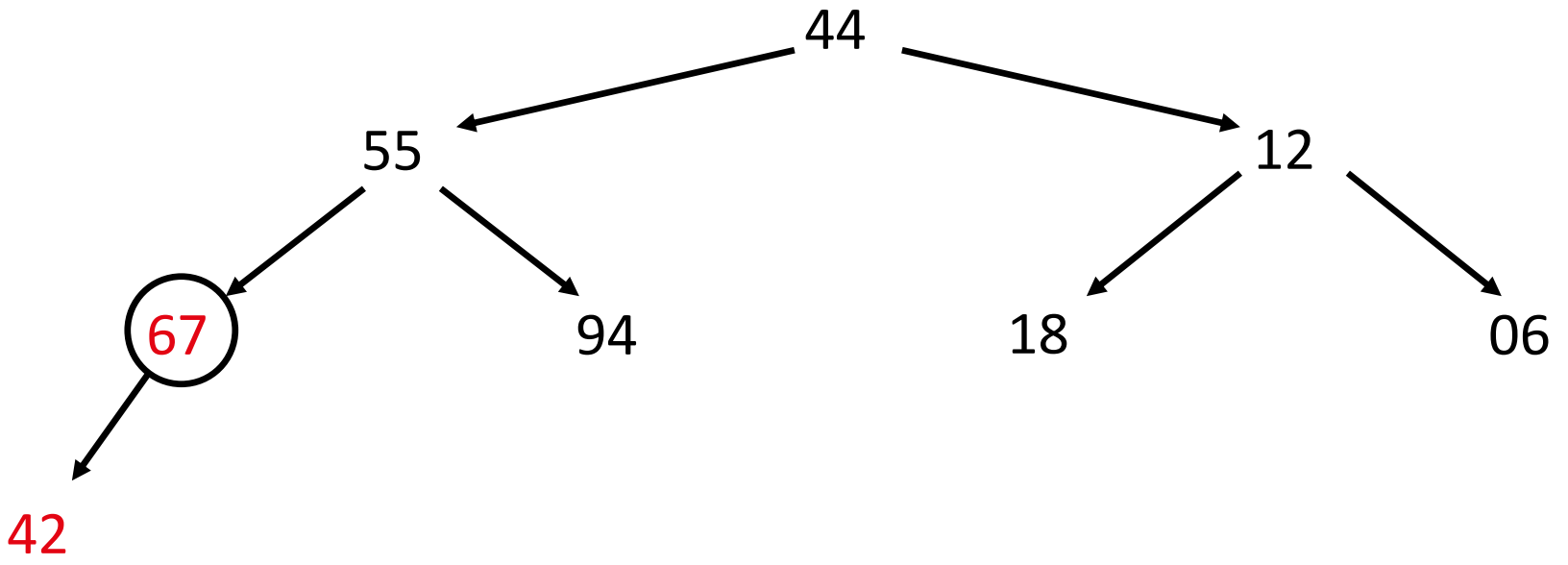




Construction d'un tas bas vers le haut

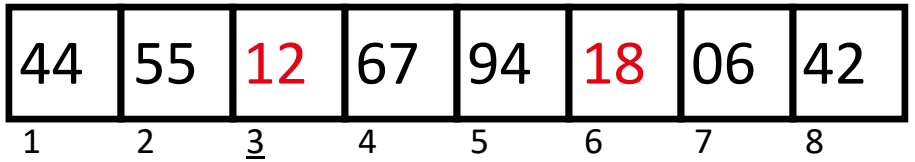
Création du monceau tas-max

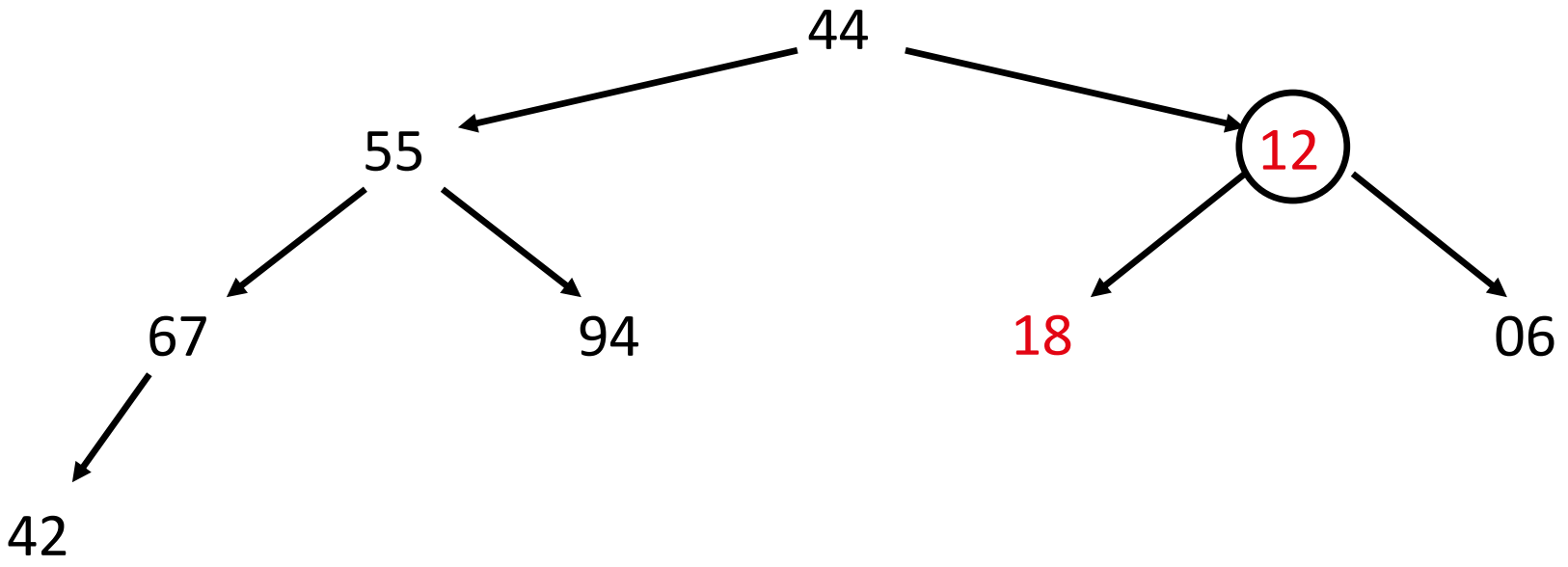




Construction d'un tas bas vers le haut

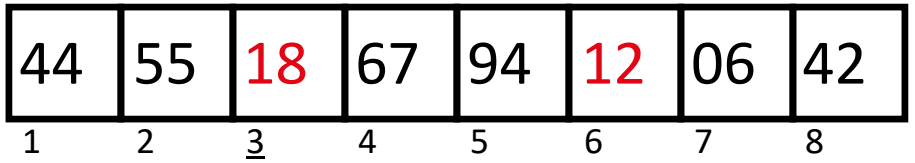
Création du monceau tas-max

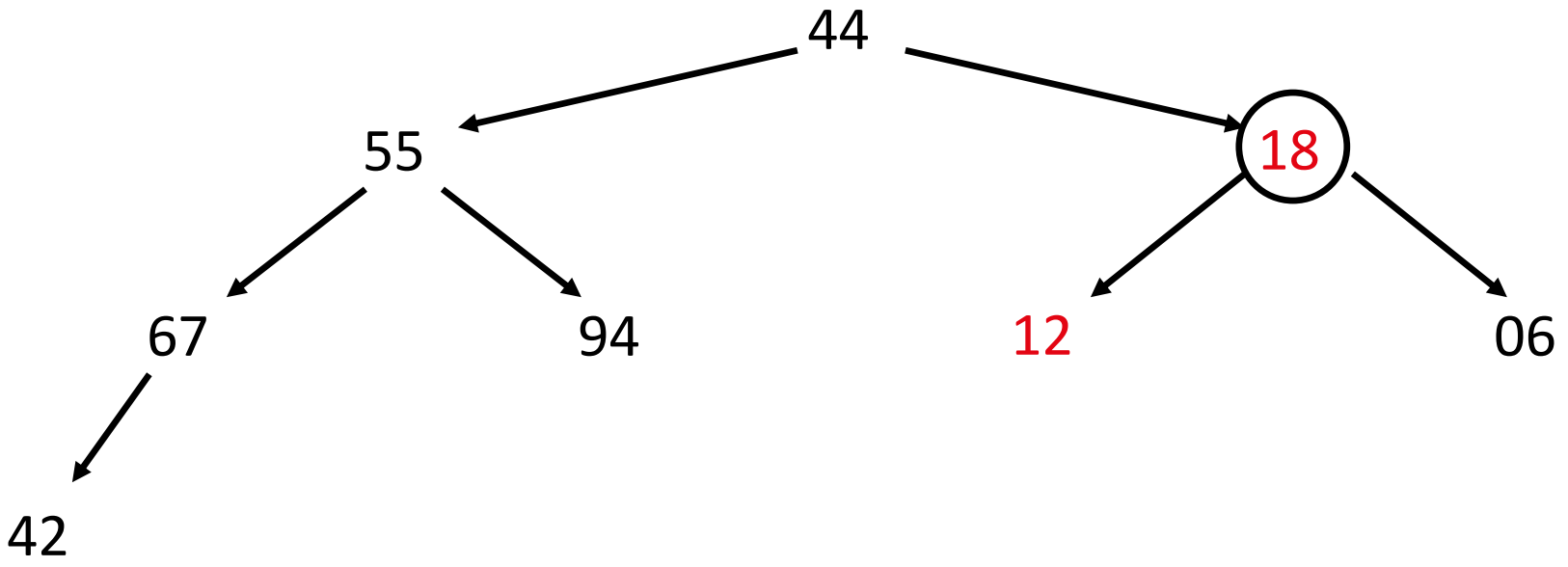




Construction d'un tas bas vers le haut

Création du monceau tas-max

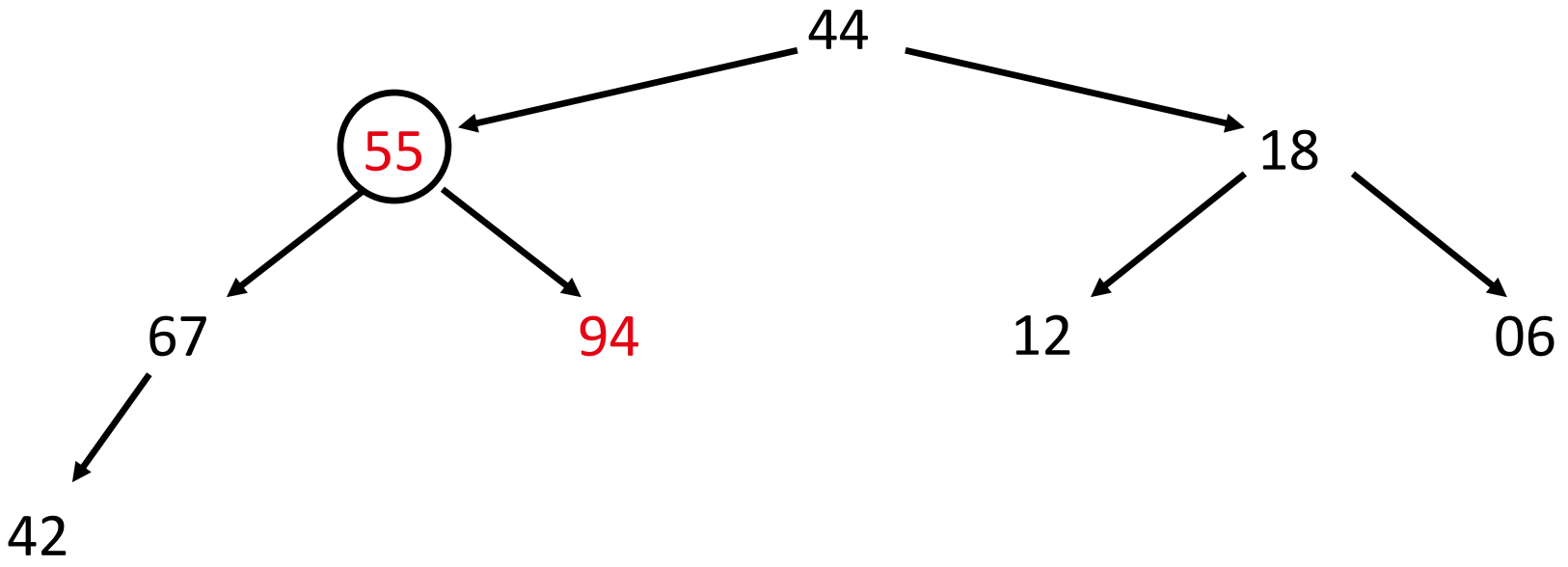




Construction d'un tas bas vers le haut

Création du monceau tas-max

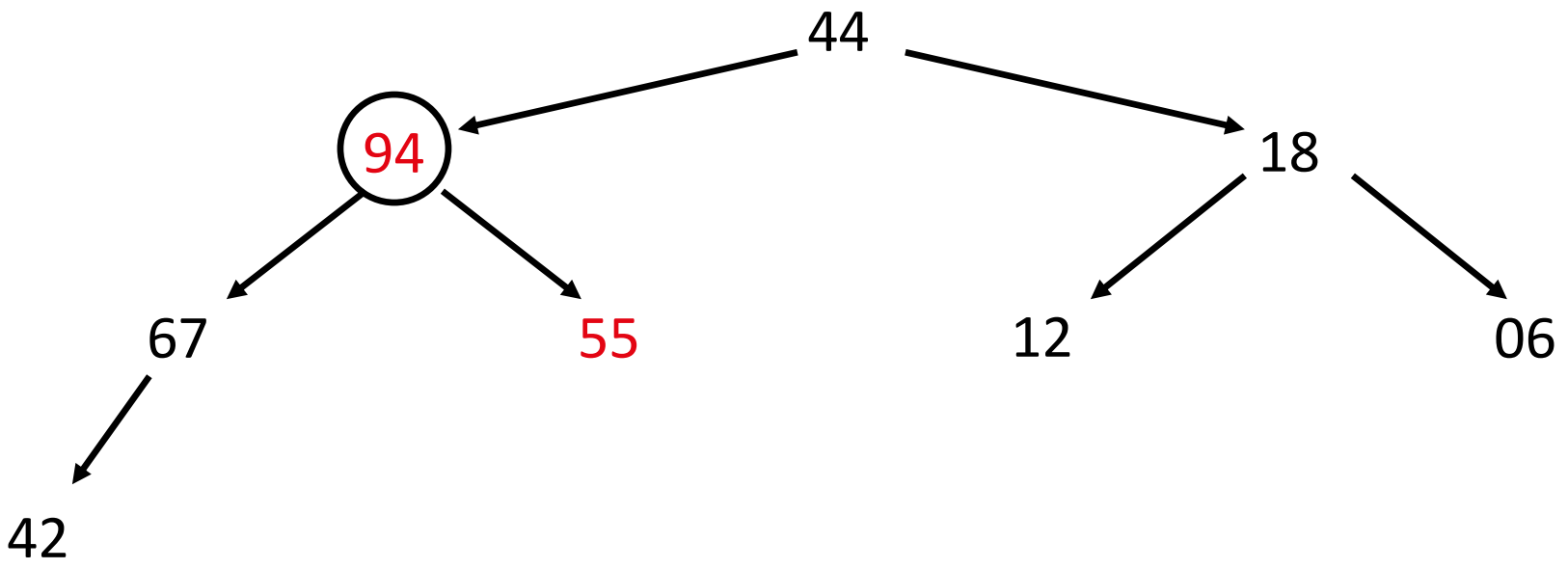
44	55	18	67	94	12	06	42
1	2	3	4	5	6	7	8



Construction d'un tas bas vers le haut

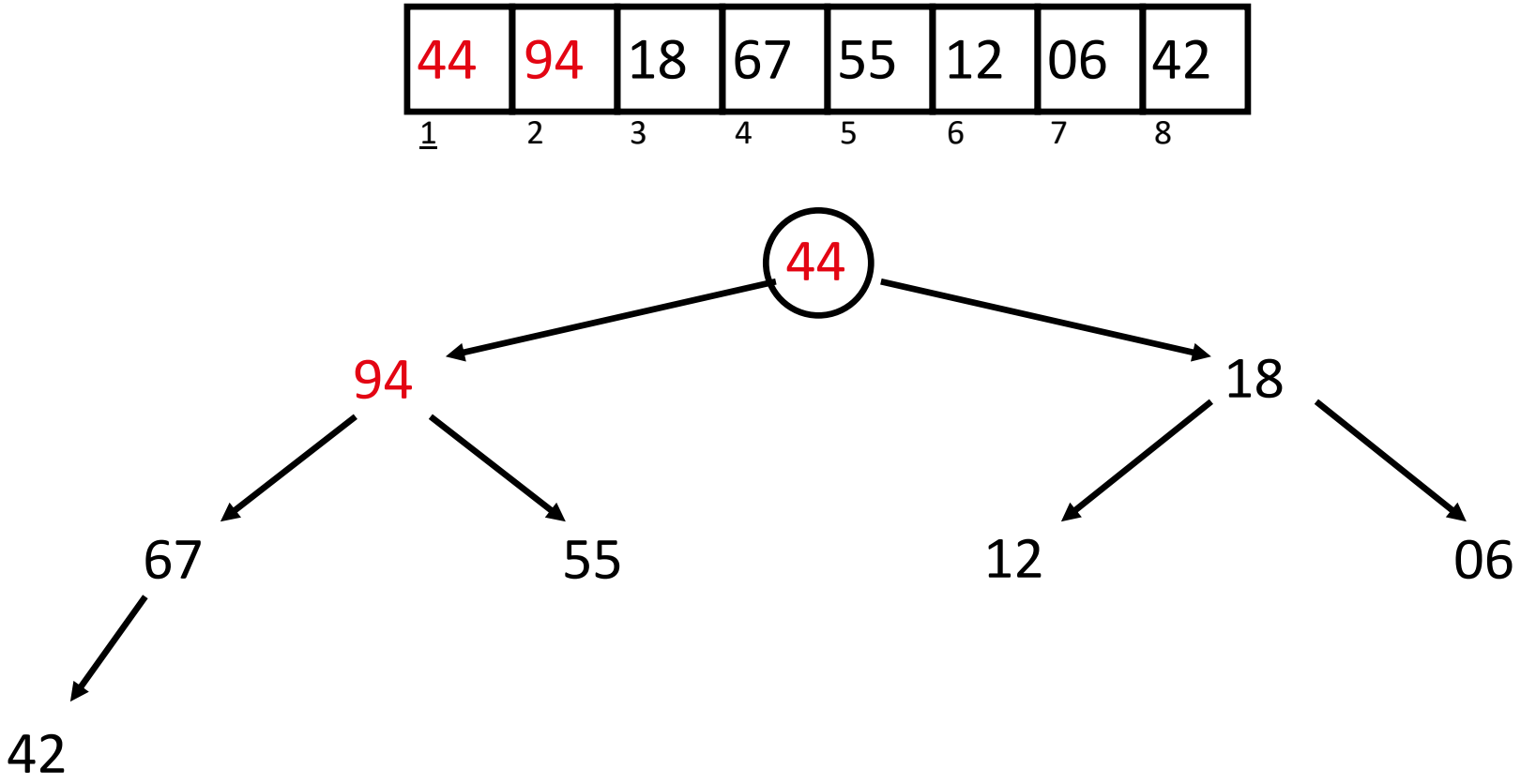
Création du monceau tas-max

44	94	18	67	55	12	06	42
1	2	3	4	5	6	7	8



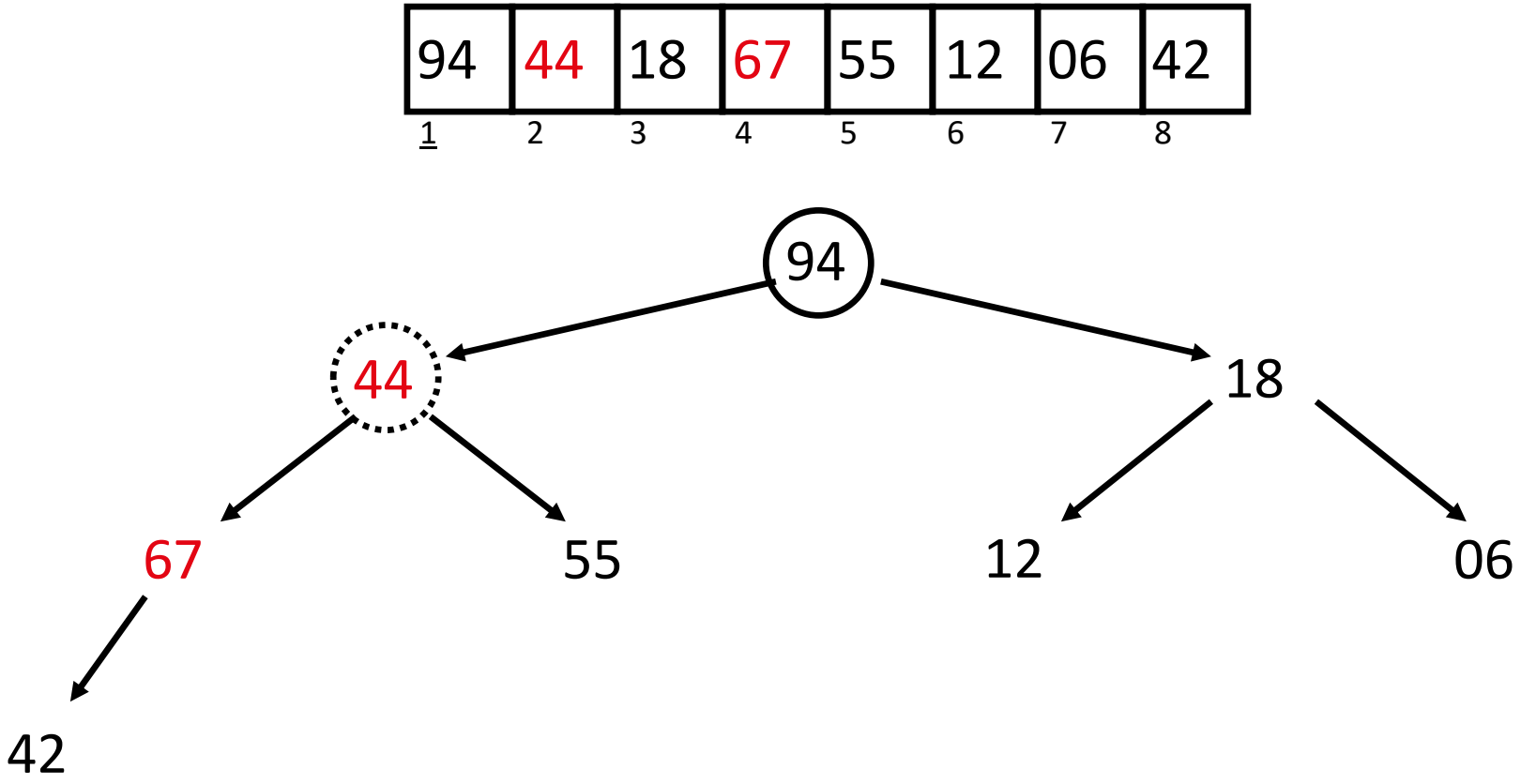
Construction d'un tas bas vers le haut

Création du monceau tas-max



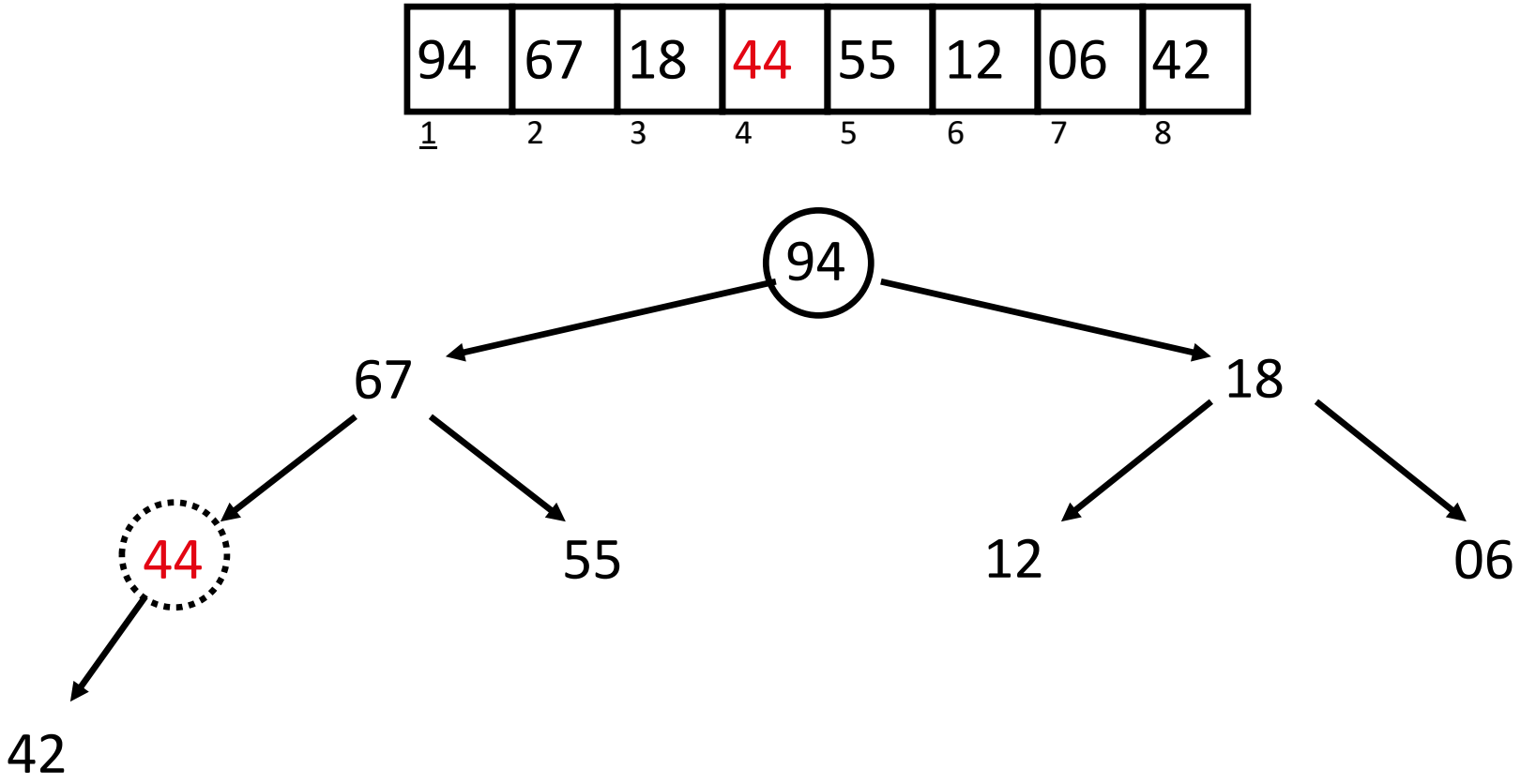
Construction d'un tas bas vers le haut

Création du monceau tas-max



Construction d'un tas bas vers le haut

Création du monceau tas-max



Analyse de HeapButtomUp

- Pour estimer le temps d'exécution de heapBottomUp() :
  - Comptons le nombre de fois qu'un parent est comparé à son enfant maximal (opération baromètre)
  - Pour chaque parent i de $\lfloor n/2 \rfloor$ à 1, $H[i]$ est comparé à $\max\{H[2i], H[2i+1]\}$
  - Dénotons par $C(n)$ le nombre de fois que cette comparaison est effectuée
- En meilleur cas, $H[1..n]$ est déjà (initialement) un tas
  - Satisfait $H[i] \geq H[2i]$ et $H[i] \geq H[2i+1]$ pour $i = 1$ à $\lfloor n/2 \rfloor$
  - Jamais nécessaire de vérifier la propriété du tas pour les enfants
  - $C_{best}(n) = \lfloor n/2 \rfloor$. Donc $C_{best}(n)$ est en $O(n)$.

Analyse de HeapButtomUp

- En pire cas, on peut avoir $H[i] < \max\{H[2i], H[2i+1]\}$ pour $i = 1$ à $\lfloor n/2 \rfloor$
  - Dans ce cas, vérifier la propriété du tas pour les enfants
  - Si le nœud i se trouve au niveau k, le nombre de niveaux qu'il y a sous le noeud i est donné par $h - k$ (où h est la hauteur du tas)
  - Le nombre de comparaisons de H[i] avec max{H[2i], H[2i+1]} qu'il faut faire en pire cas pour le parent i est alors de $(h - k)$
  - Soit $p(k)$ le nombre de parents présents au niveau k
  - Nous avons $p(k) = 2^k$ pour $k = 0, 1, \ldots, h - 2$. ($h - 1$ = niveau max des parents)
  - Et $p(k) \leq 2^k$ pour $k = h - 1$.

Analyse de HeapButtomUp

- Le nombre de comparaisons $C_{worst}(n)$ effectuées au total en pire cas donné par:

$$C_{worst}(n) \leq \sum_{k=0}^{h-1} 2^k(h-k) = h\sum_{k=0}^{h-1} 2^k - \sum_{k=0}^{h-1} k2^k = h\left(2^h - 1\right) - \sum_{k=0}^{h-1} k2^k$$

$$\boxed{\sum_{i=0}^{n} r^i = \frac{r^{n+1}-1}{r-1}}$$

- Sachant que (voir dictionnaire de série)

$$\sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$$

Analyse de HeapButtomUp

- On a donc

$$\sum_{k=0}^{h-1} k2^k = \sum_{k=1}^{h-1} k2^k = (h-2)2^h + 2$$

- Alors

$$C_{worst}(n) \leq h\left(2^h - 1\right) - \left[(h-2)2^h + 2\right] = -h + 2\left(2^h - 1\right)$$

Analyse de HeapButtomUp

- Or, nous avons $n = 2^h - 1 + l$ pour un tas de n nœuds avec l feuilles au niveau h
- Alors $C_{worst}(n) \leq -h + 2(n - l) < 2n$
- Alors $C_{worst}(n)$ est en $O(n)$.
- Or nous avions que $C_{best}(n)$ est en $O(n)$.
- Donc le temps d'exécution de heapBottomUp est en $O(n)$ dans tous les cas.

Insertion d'un élément dans un tas

- Pour la mise en œuvre d'une file de priorité
  - Pouvoir insérer rapidement un nouvel item dans un tas
- Insérer d'abord le nouvel élément K dans une nouvelle feuille positionnée juste après la dernière feuille du tas (ou, plus simplement, nous faisons $H[n+1] = K$)
- Comparer K avec son parent $H[\lfloor (n+1)/2 \rfloor]$:
  - Si $K \leq H[\left\lfloor \frac{n+1}{2} \right\rfloor]$ ne rien faire car $H[1..n+1]$ est un tas
  - Sinon on interchange K avec$H[\left\lfloor \frac{n+1}{2} \right\rfloor]$
  - Recommencer jusqu'à ce que K ≤ à la valeur de son parent (ou jusqu'à ce que K devienne la racine)
- Le nombre maximal de comparaisons requises est donc de $\lfloor log_2(\boldsymbol{n+1}) \rfloor$.

Insertion d'un élément dans un tas

Rappel : lorsque les indices débutent à 1: parent(i) = $\lfloor i/2 \rfloor$

- La position en C++ du nœud $i = i - 1 \equiv j$
- La position en C++ du parent(i) = $\lfloor i/2 \rfloor - 1$
- Donc, parent(j) = $\left\lfloor \frac{j+1}{2} \right\rfloor - 1 = \lfloor (j+1)/2 - 1 \rfloor = \lfloor (j-1)/2 \rfloor$

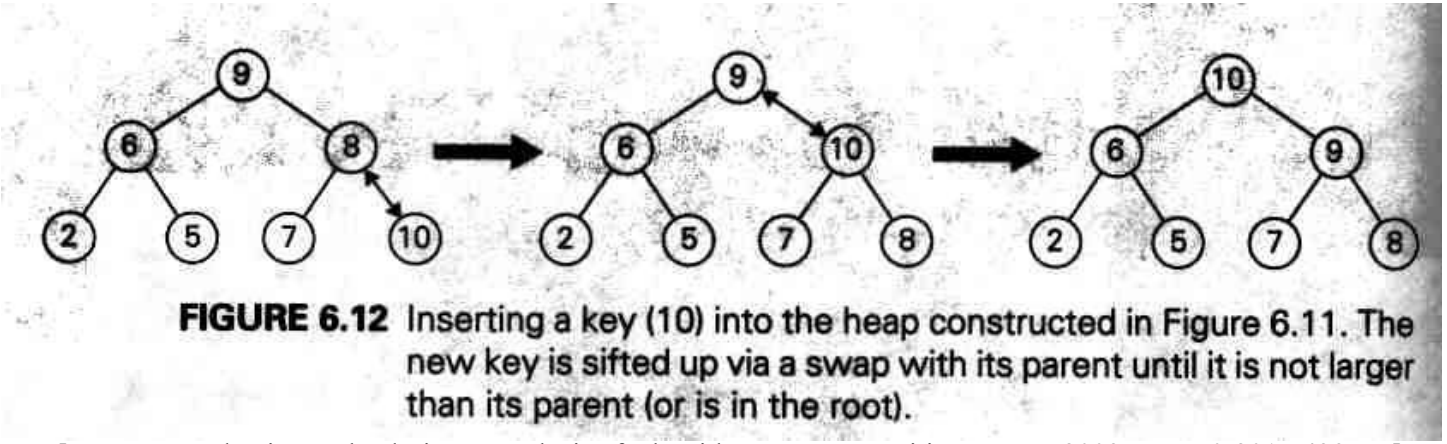


FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

[source: Introduction to the design & analysis of Algorithms, Anany Levitin, Pearson 2003, ISBN 0-201-74395-7.]

Insertion d'un élément dans un tas

```
template<typename Comparable>
void insertInHeap(vector<Comparable>& heap, const Comparable& element)
{
	heap.push_back(element);
	size_t child = heap.size()-1; //indices débutent en 0
	if (child !=0) //déjà inséré: on a terminé
	{
		size_t parent = (child-1)/2; //== heap.size()/2 - 1
		while(child > 0 && heap[parent] < heap[child])
		{
			std::swap(heap[parent], heap[child]);
			child = parent;
			parent = (child-1)/2;
		}
	}
}
```

Enlever la racine d'un tas

- Pour la mise en œuvre d'une file de priorité, nous devons fréquemment enlever la racine d'un tas : un élément dont la priorité est la plus élevée
  - Interchanger l'élément $H[n]$ avec $H[1]$
  - Reconstruire le tas $H[1..n-1]$ en percolant $H[1]$ vers le bas avec percDown(1):
    - ✓ Comparer la valeur de H[1] avec celle de ses enfants et l'interchanger avec le max de ses enfants si c'est nécessaire
    - ✓ Continuer jusqu'au niveau inférieur (si c'est nécessaire) tel que prescrit par percDown() pour l'élément i = 1.
- Cela nécessite au plus $O(\log n)$ comparaisons (pour reconstruire $H[1..n-1]$)

Enlever la racine d'un tas

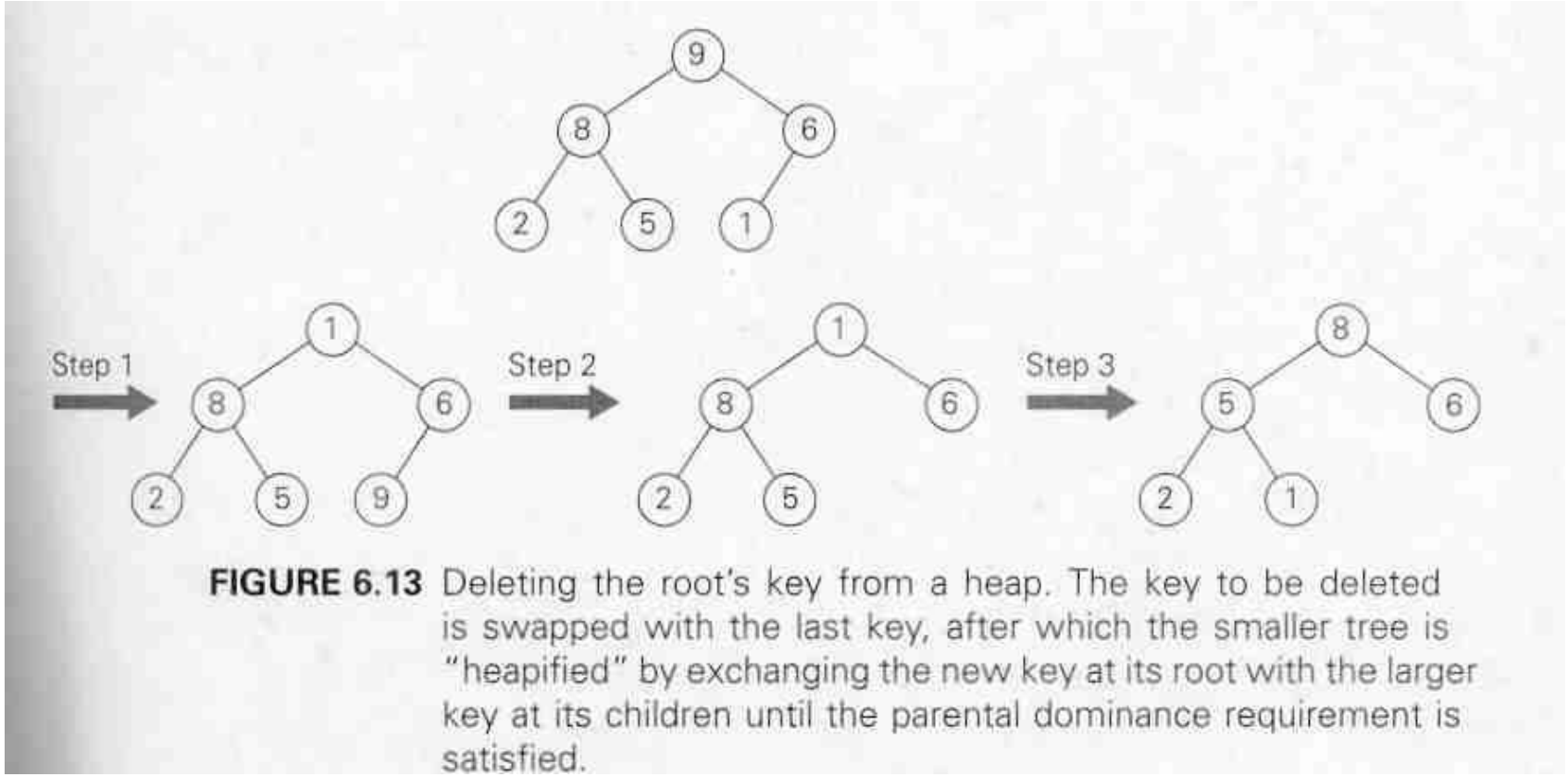


FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key, after which the smaller tree is "heapified" by exchanging the new key at its root with the larger key at its children until the parental dominance requirement is satisfied.

[source: Introduction to the design & analysis of Algorithms, Anany Levitin, Pearson 2003, ISBN 0-201-74395-7.]

Enlever la racine d'un tas

```
template<typename Comparable>
void removeFromHeap(vector<Comparable>& heap)
{
	if (heap.empty()) throw logic_error("heap must be nonempty");
	heap[0] = heap[heap.size()-1]; //écrasement de la racine
	heap.pop_back(); //enlever le dernier élément de heap
	percDown(heap, 0, heap.size()); //reconstruire le reste du tas
}
```

Le tri par tas (« Heapsort »)

- À partir d'un tableau non trié, construire d'abord un tas $H[1..n]$ à l'aide de l'algorithme heapBottumUp en un temps en $O(n)$ (dans tous les cas).
- La racine est donc un élément de valeur la plus élevée.
- Interchanger $H[1]$ avec $H[n]$ et reconstruire le tas $H[1..n-1]$ à l'aide de percDown(1) en $O(\log n)$ comparaisons en pire cas et $O(1)$ comparaisons en meilleur cas (réalisé lorsque tous les éléments ont la même valeur)
- Recommencer ces 2 opérations avec $H[1..n-1]$, ensuite $H[1..n-2]$, et puis $H[1..n-3]$ … finalement l'on s'arrête en $H[1]$.
- $H[1..n]$ est trié en ordre croissant.

Le tri par tas (« Heapsort »)

- En utilisant la formule de Stirling pour $n!$, on trouve que le nombre de comparaison requises en pire cas et en meilleur cas :

$$C_{worst}(n) \in O(\log(n-1) + \log(n-2) + \cdots + \log(1)) = O(\log((n-1)\,!) = O(nlog(n))$$

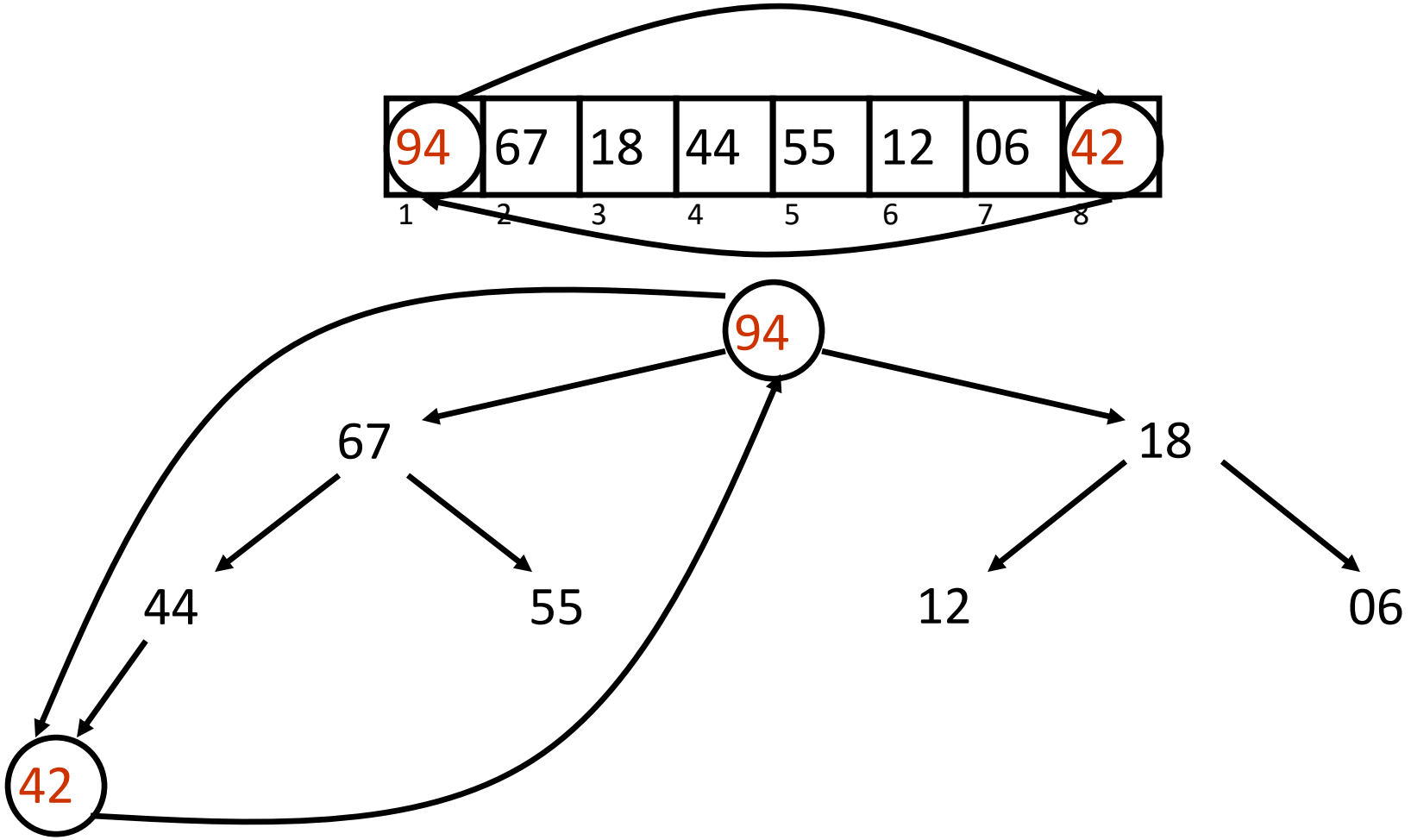
$$C_{best}(n) \in O(1 + 1 + \cdots + 1) = O(n)$$

- Le tri par tas s'exécute donc en $O(n\log(n))$ en pire cas et en $O(n)$ en meilleur cas.
  - Algorithme de tri performant
  - Empiriquement, nous observons, qu'en moyenne, le tri par tas est légèrement plus rapide que le tri fusion (que vous verrez en Ift-3001).

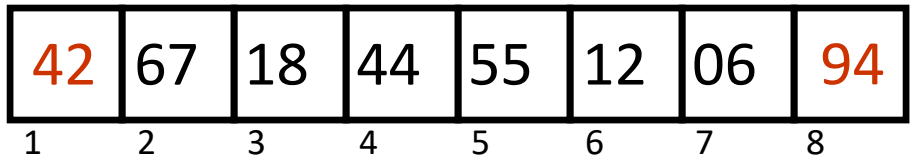
Tri par tas (« Heapsort »)

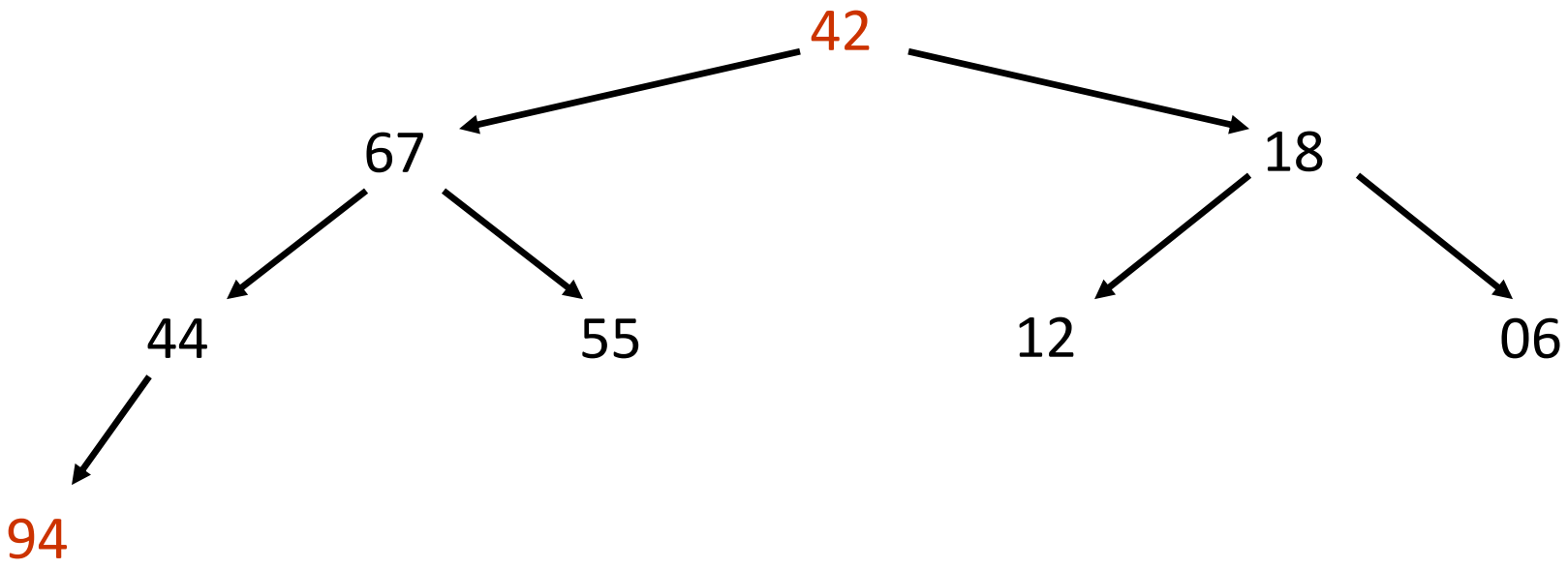
```
template <typename Comparable>
void heapsort(vector<Comparable>& heap )
{
	if (heap.size() > 1)
	{
		heapBottumUp(heap); // construction du monceau
		for(size_t i = heap.size() - 1; i > 0; i--)
		{
			swap(heap[0], heap[i]); // positionner la racine dans le tableau trié
			percDown(heap, 0, i); //reconstruction du tas sans l'élément i
		}
	}
}
```

Tri par tas (« Heapsort »)

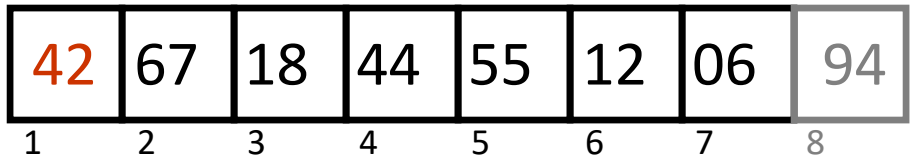


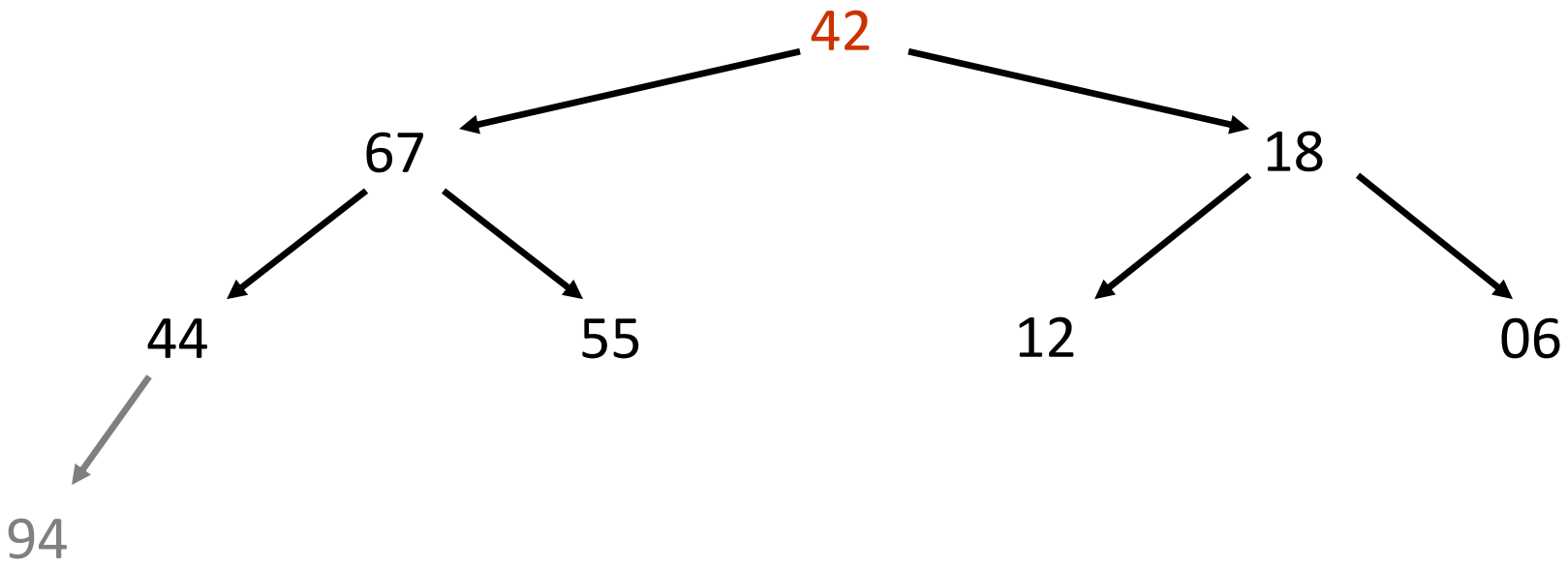
Tri par tas (« Heapsort »)



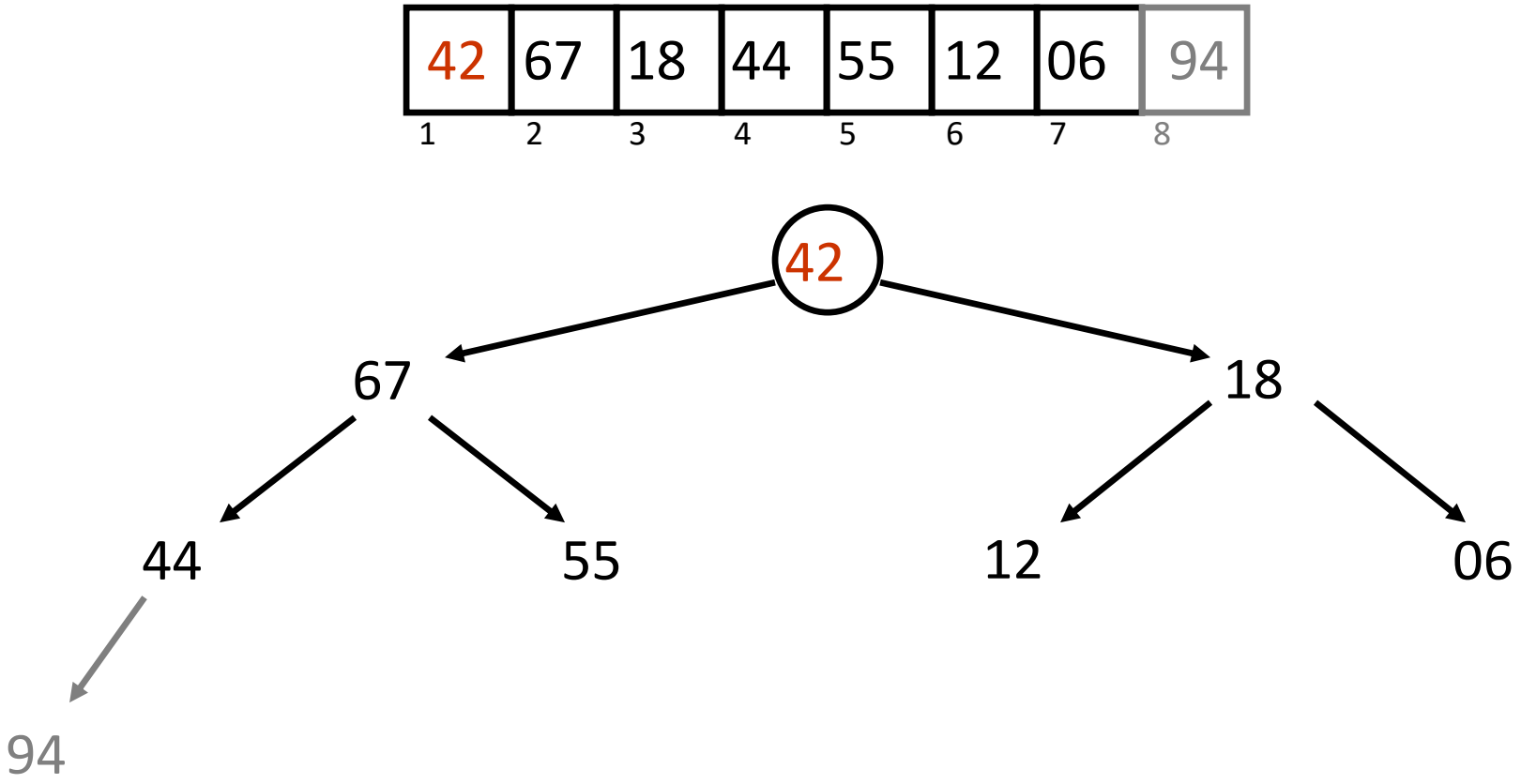


Tri par tas (« Heapsort »)

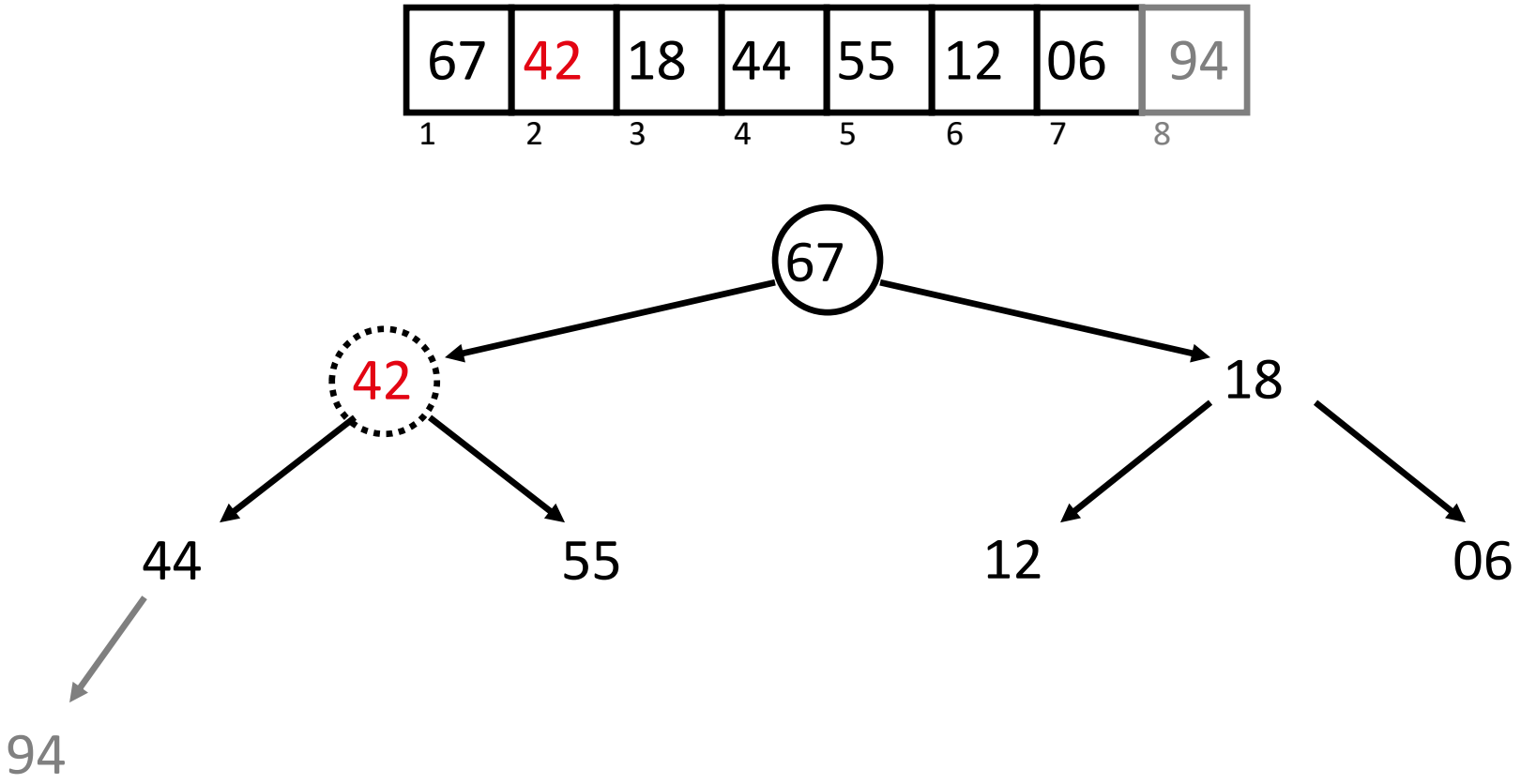




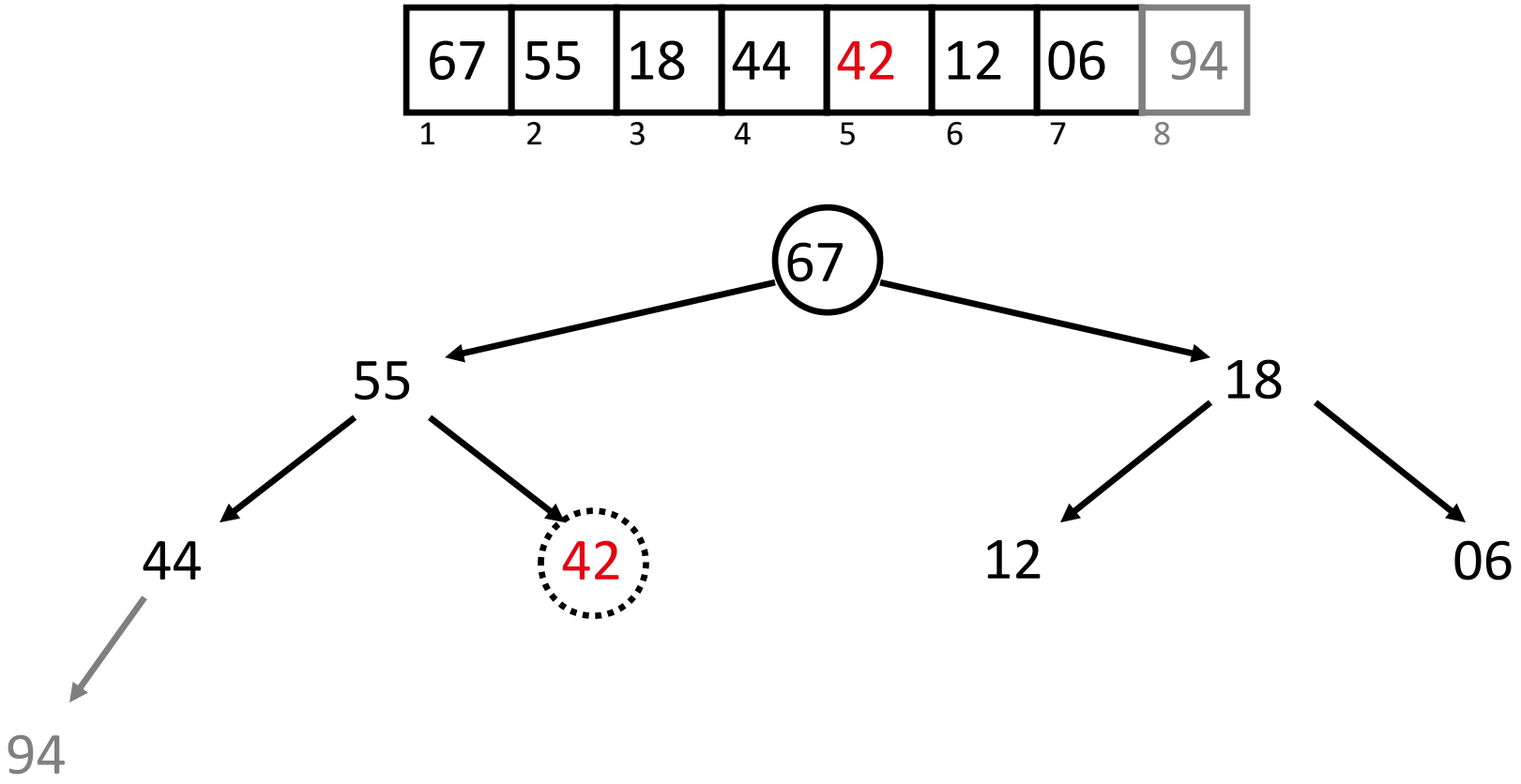
Tri par tas (« Heapsort »)



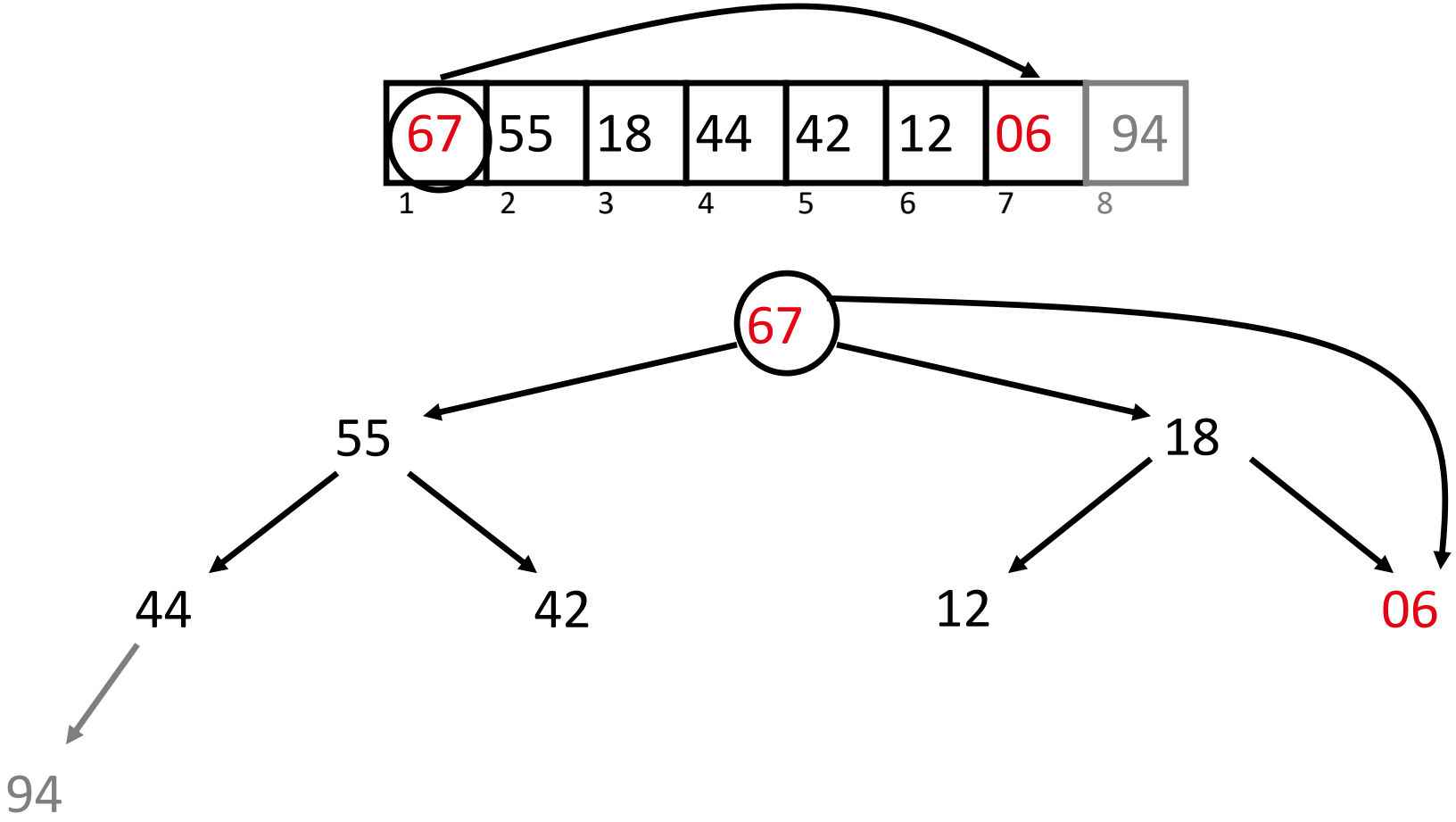
Tri par tas (« Heapsort »)



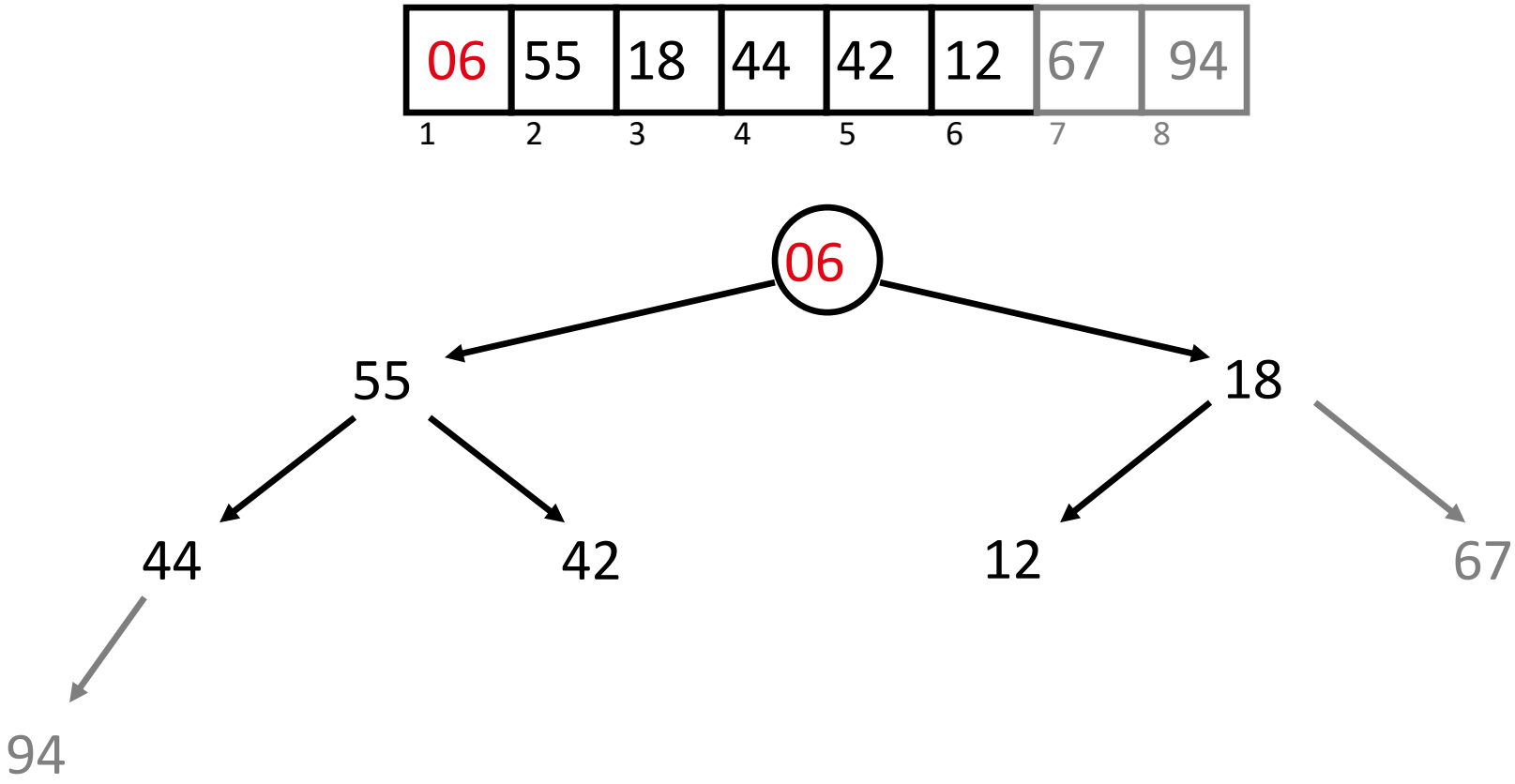
Tri par tas (« Heapsort »)



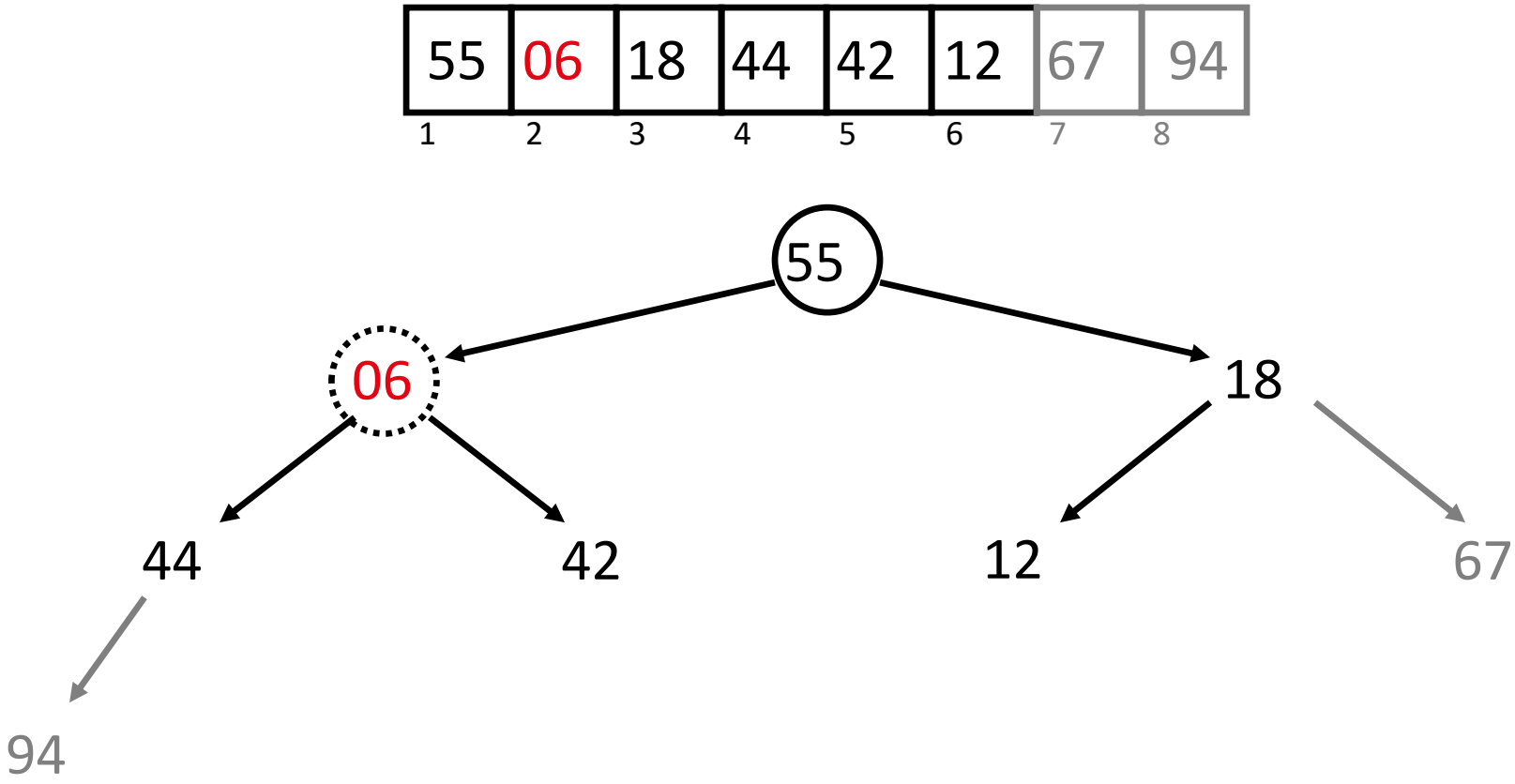
Tri par tas (« Heapsort »)



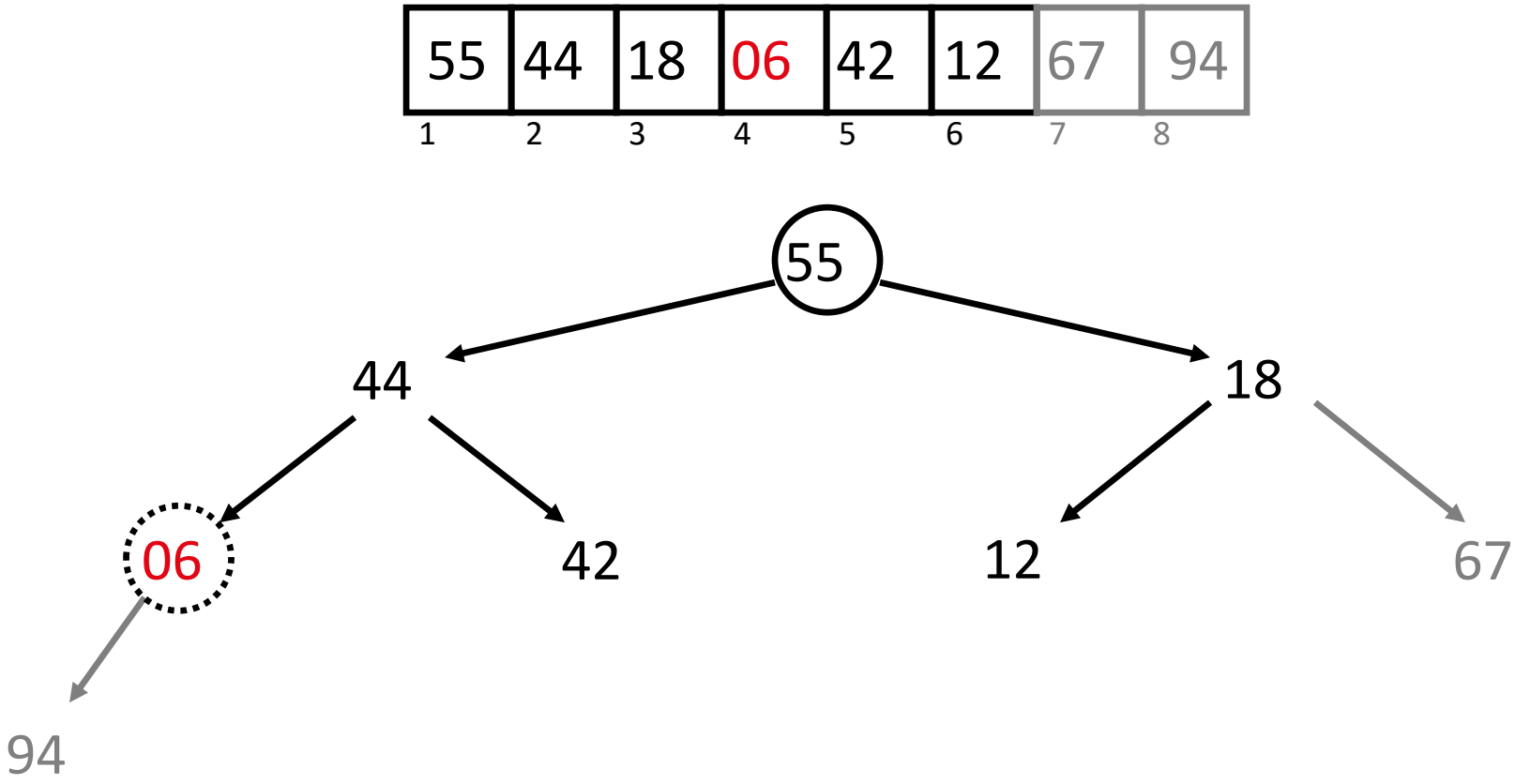
Tri par tas (« Heapsort »)



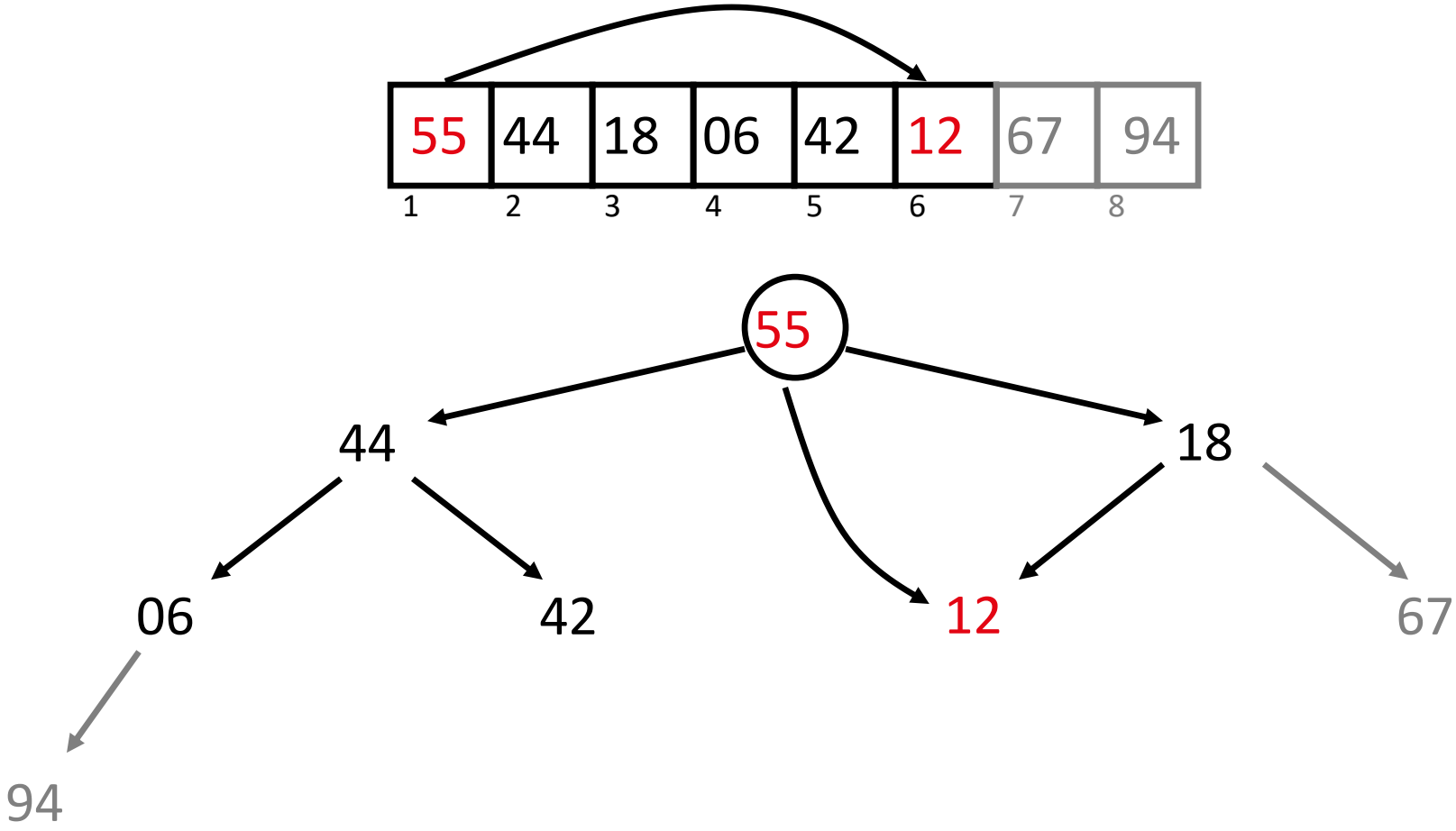
Tri par tas (« Heapsort »)



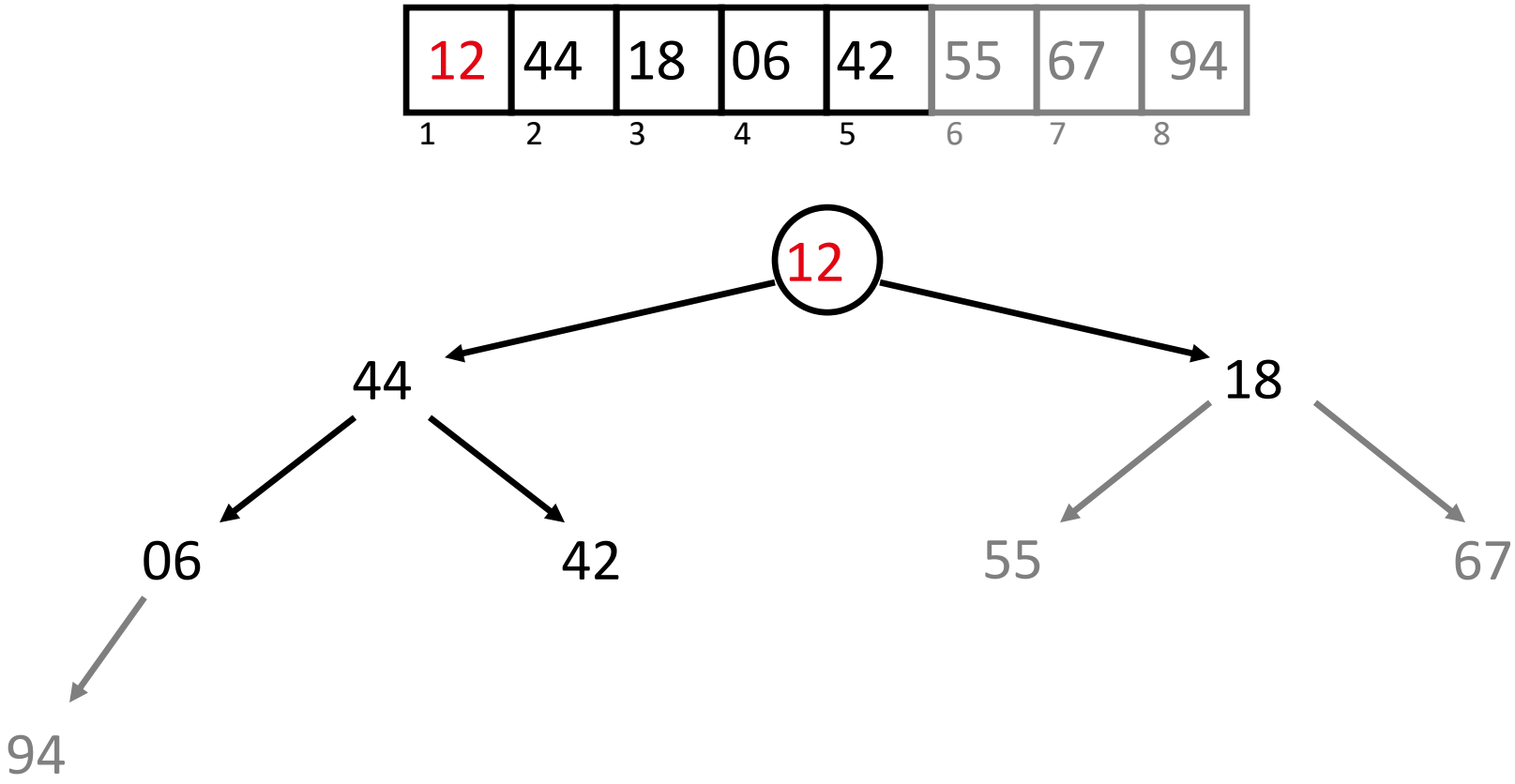
Tri par tas (« Heapsort »)



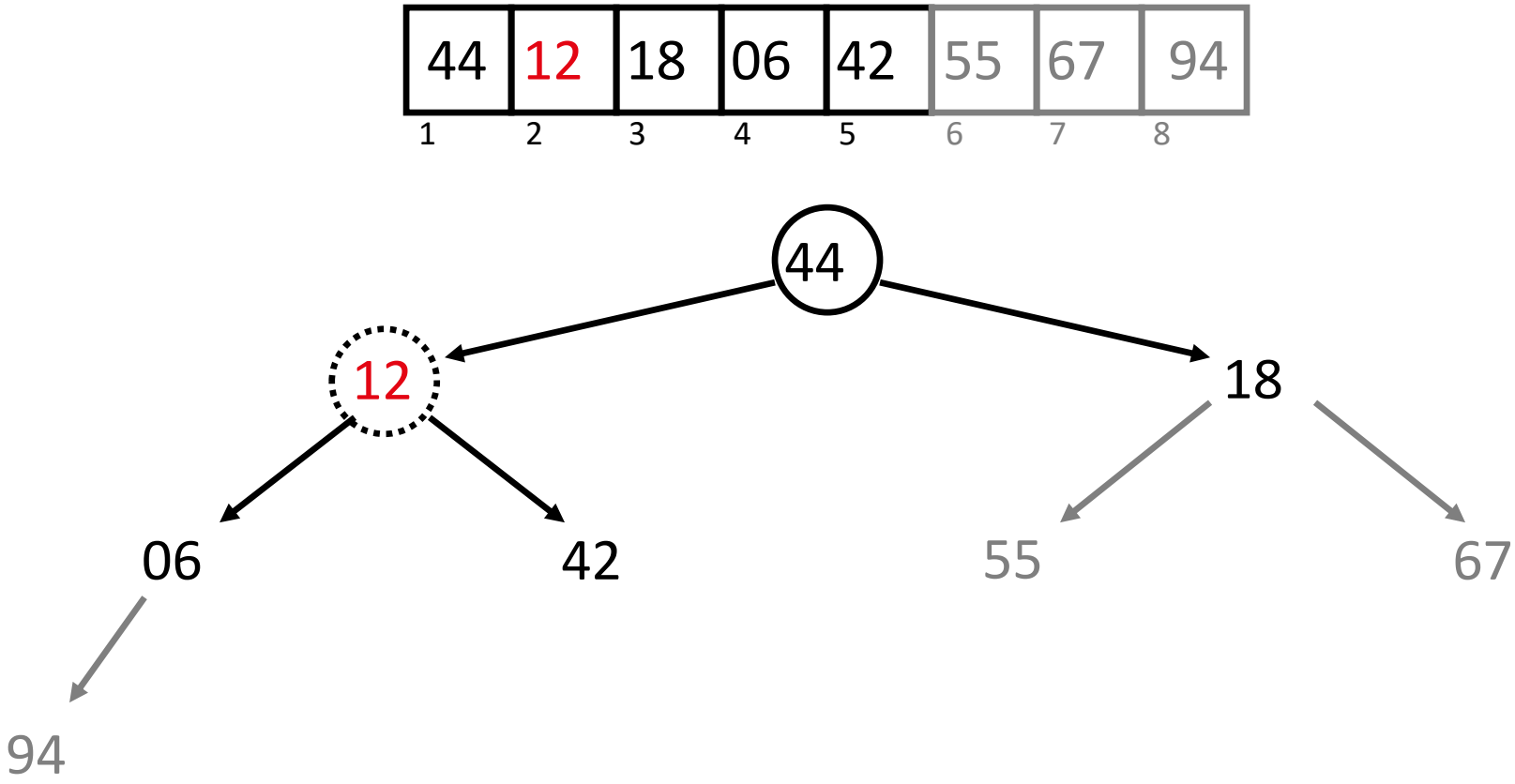
Tri par tas (« Heapsort »)



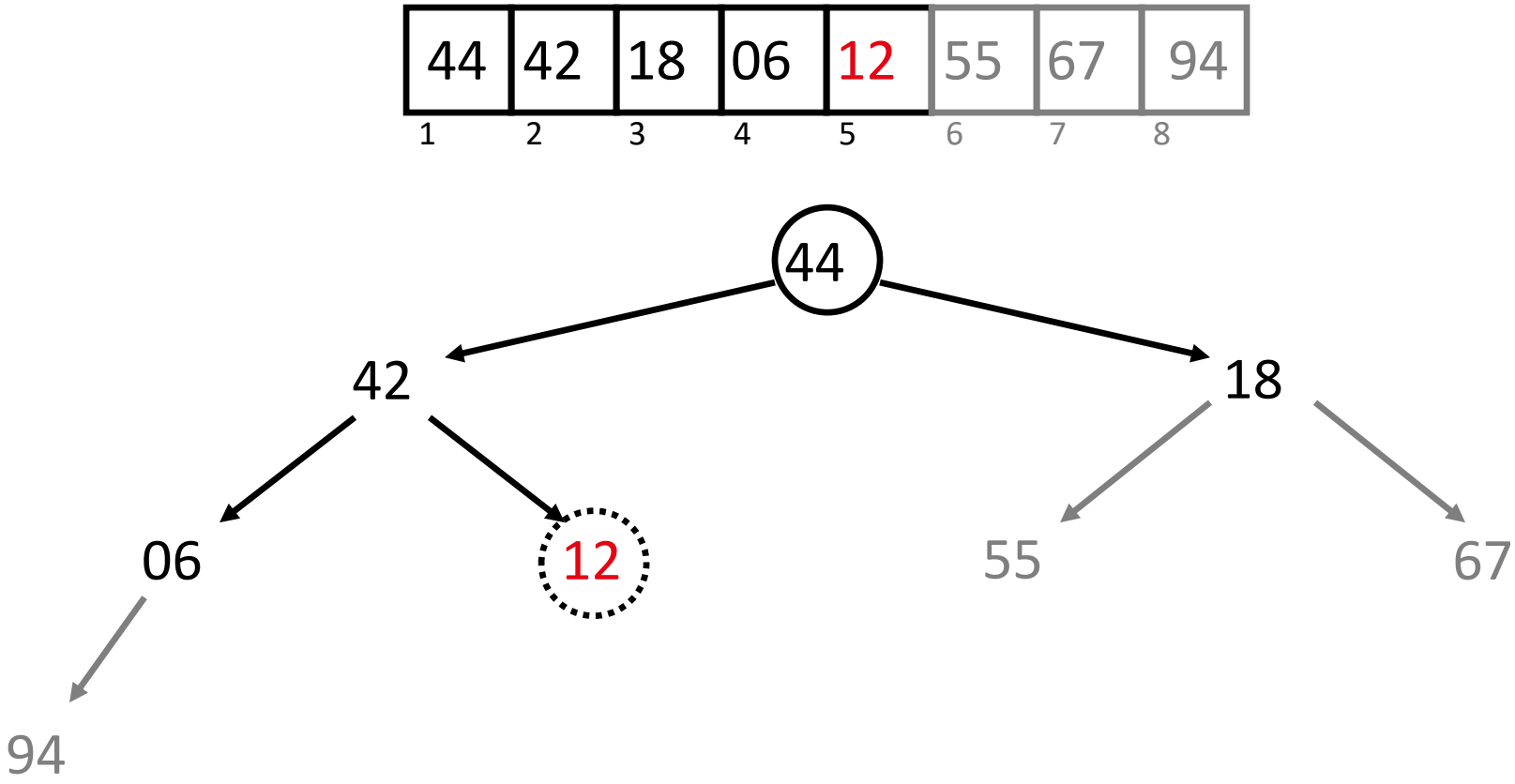
Tri par tas (« Heapsort »)



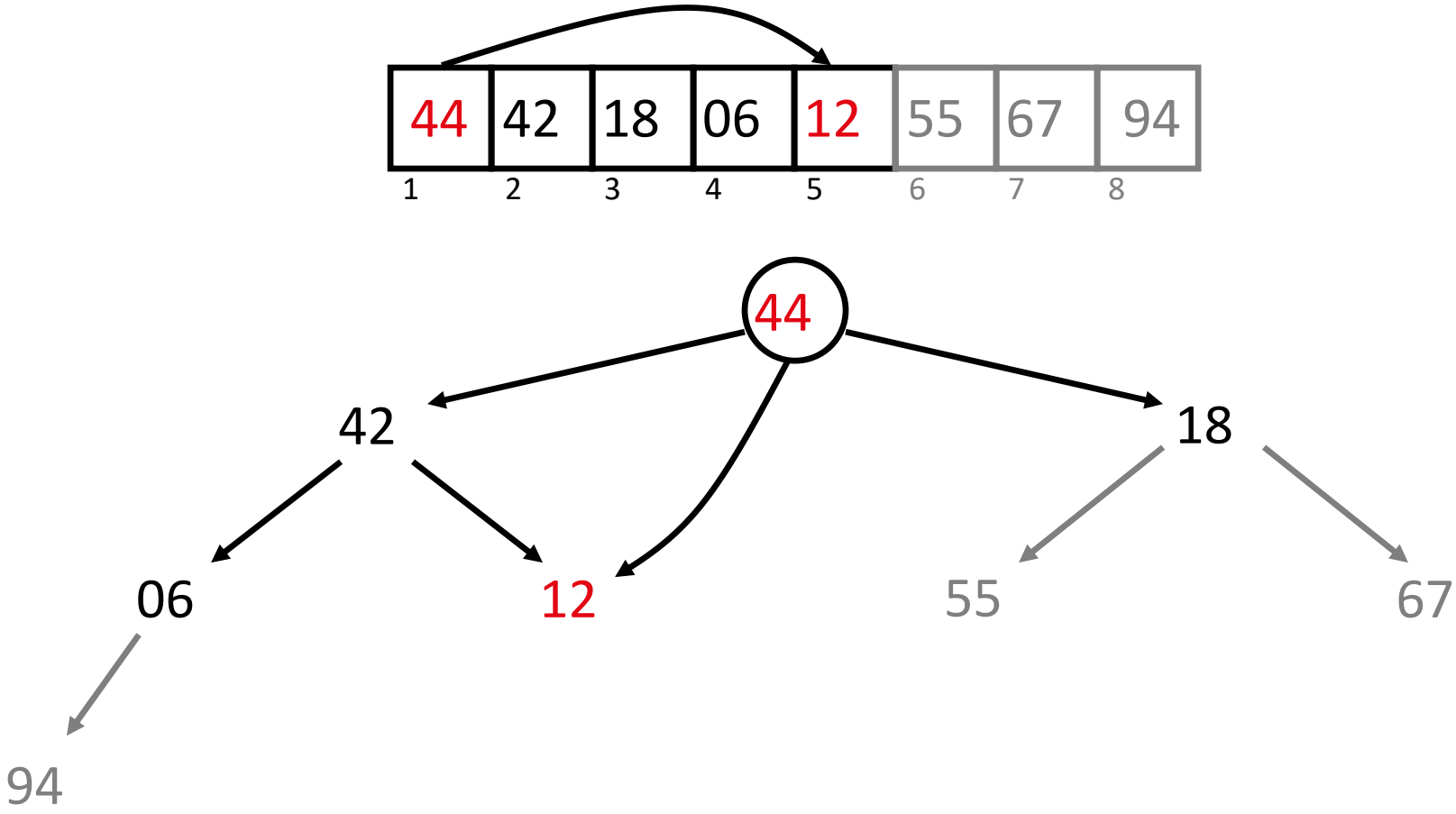
Tri par tas (« Heapsort »)



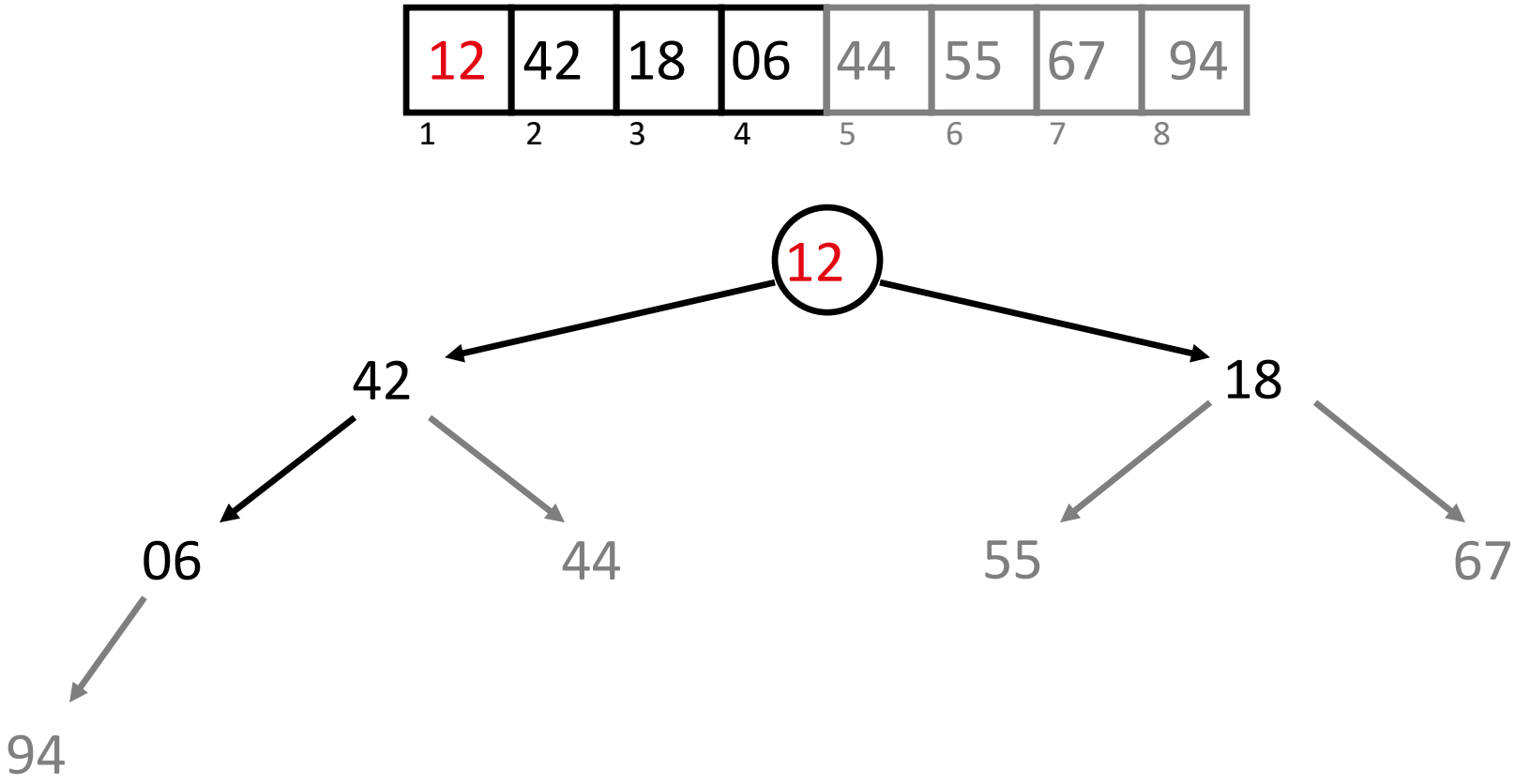
Tri par tas (« Heapsort »)



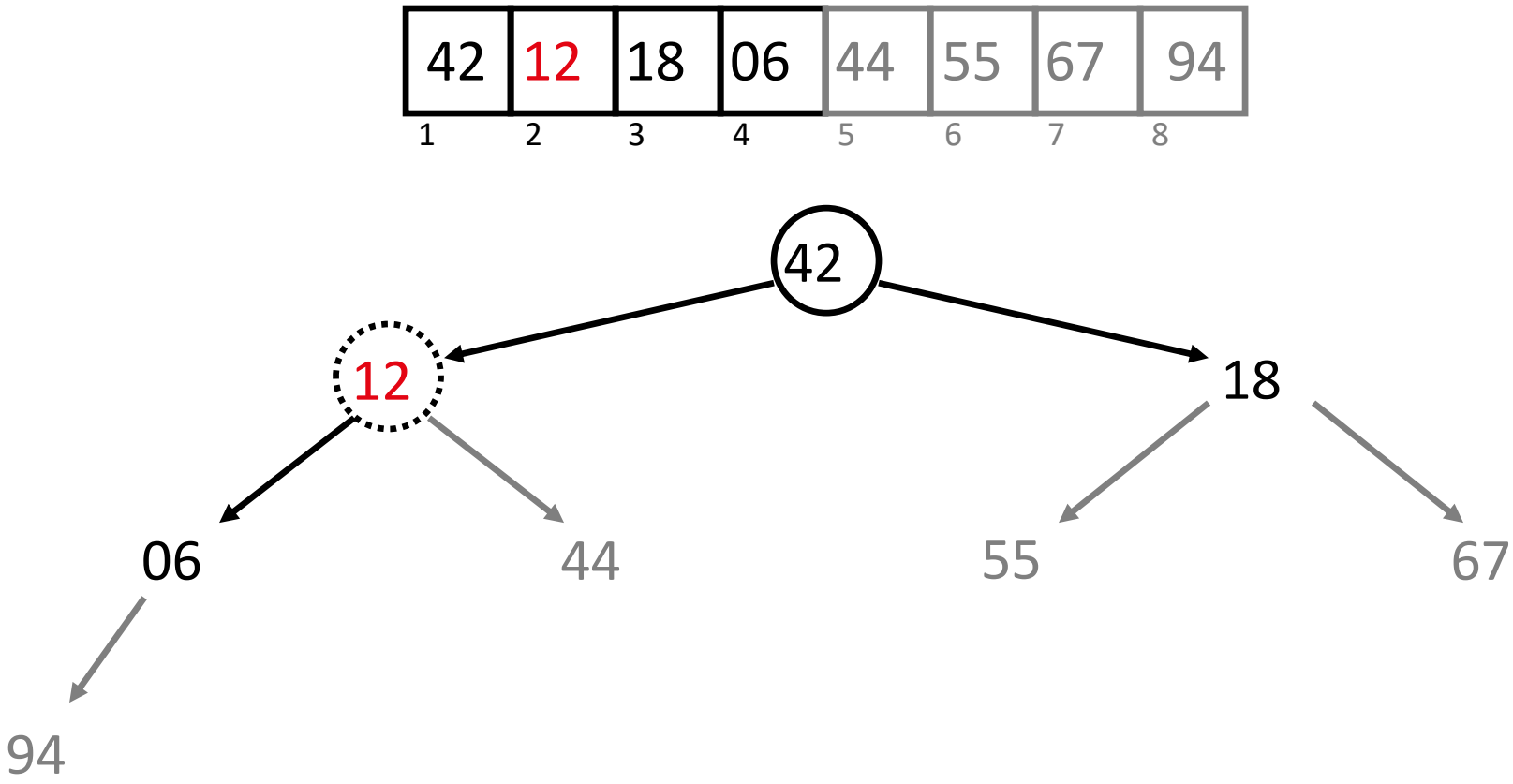
Tri par tas (« Heapsort »)



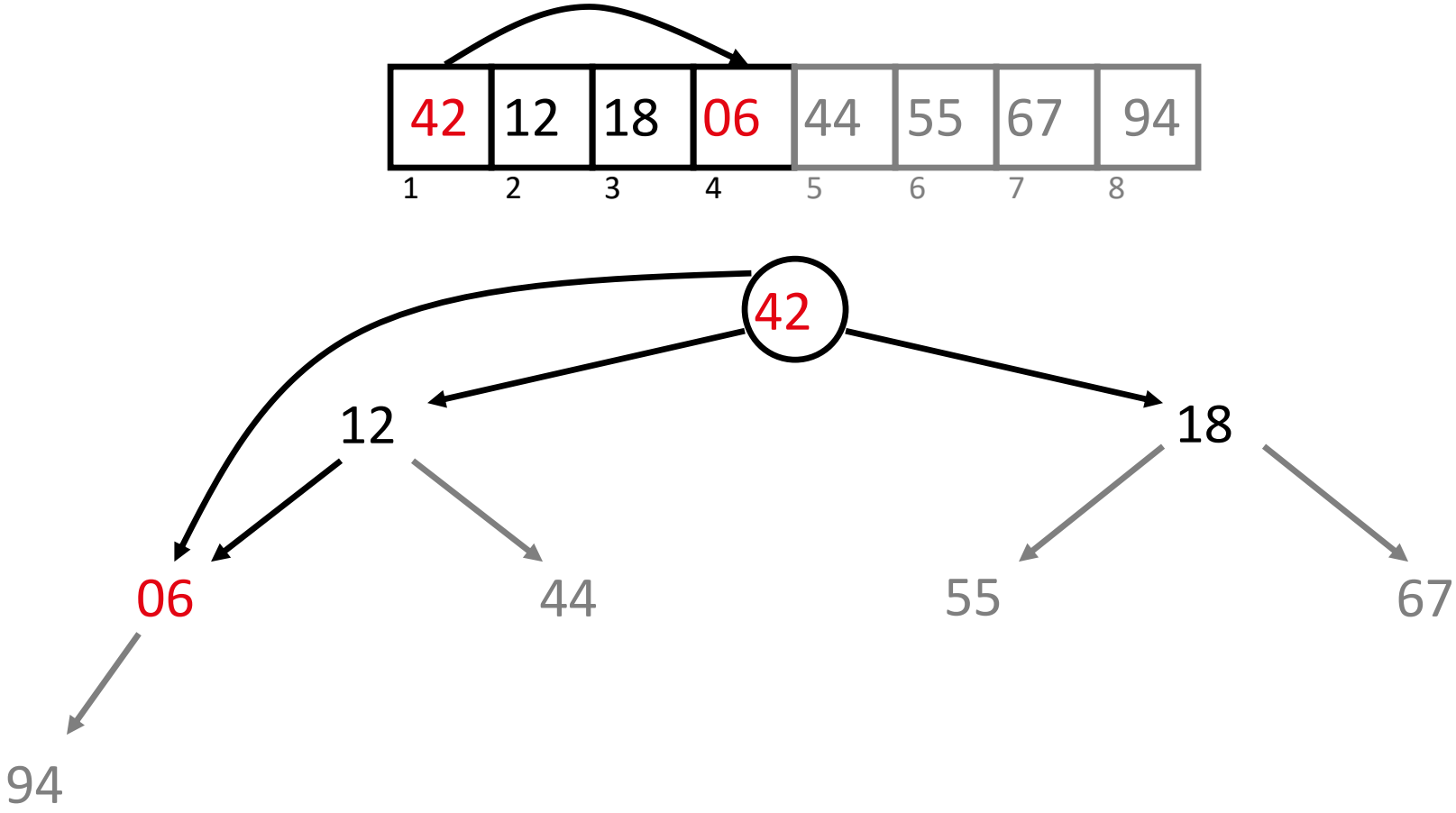
Tri par tas (« Heapsort »)



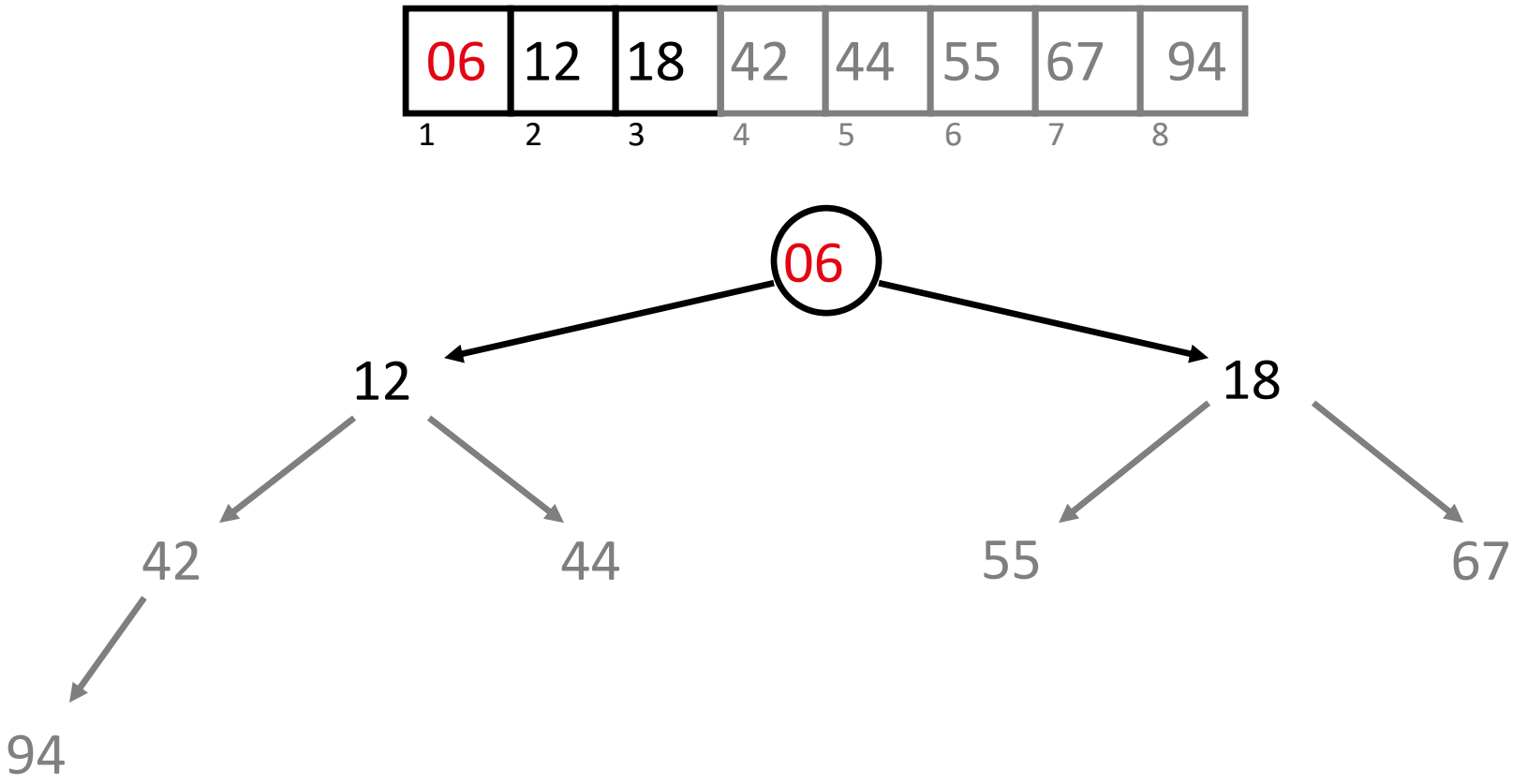
Tri par tas (« Heapsort »)



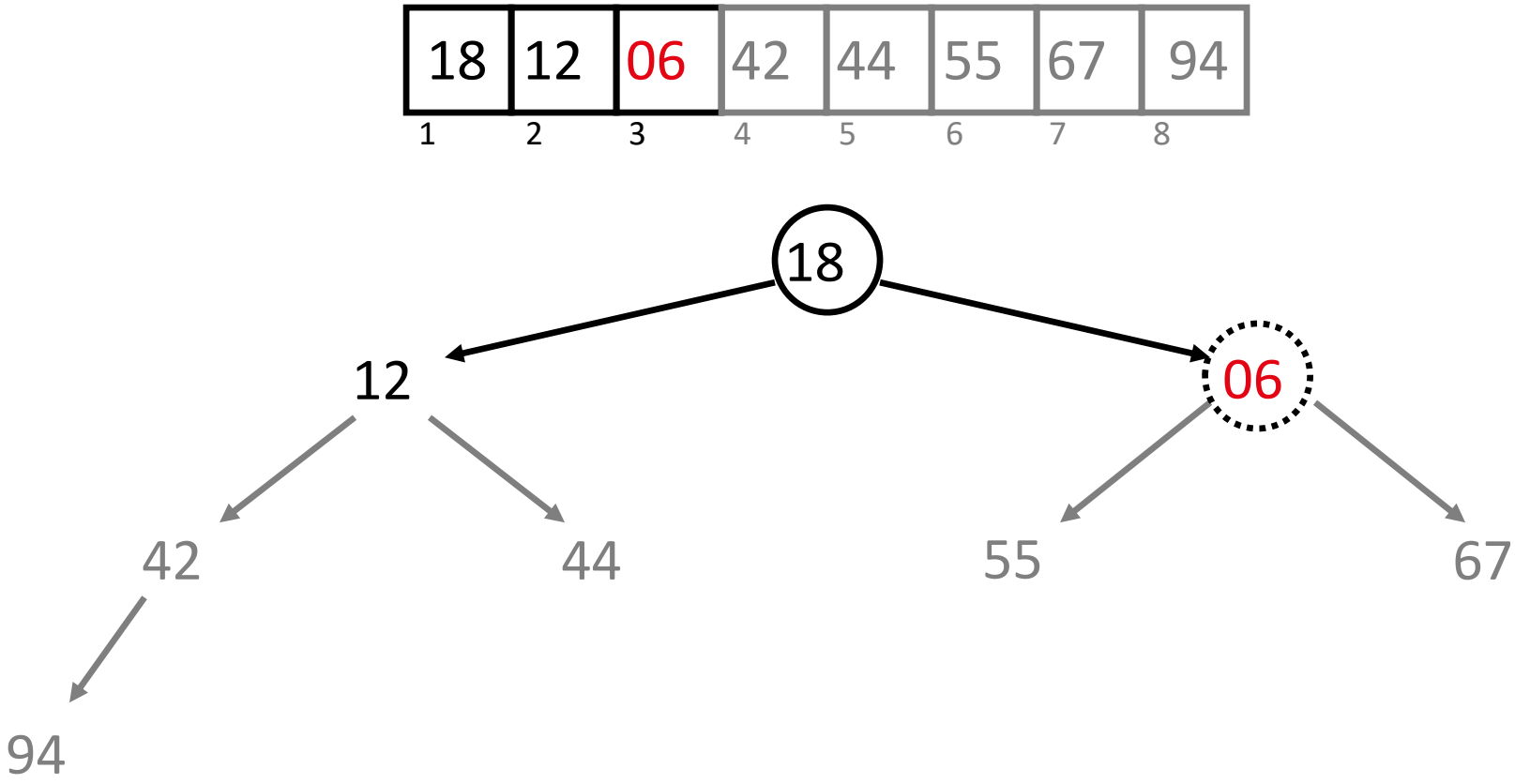
Tri par tas (« Heapsort »)



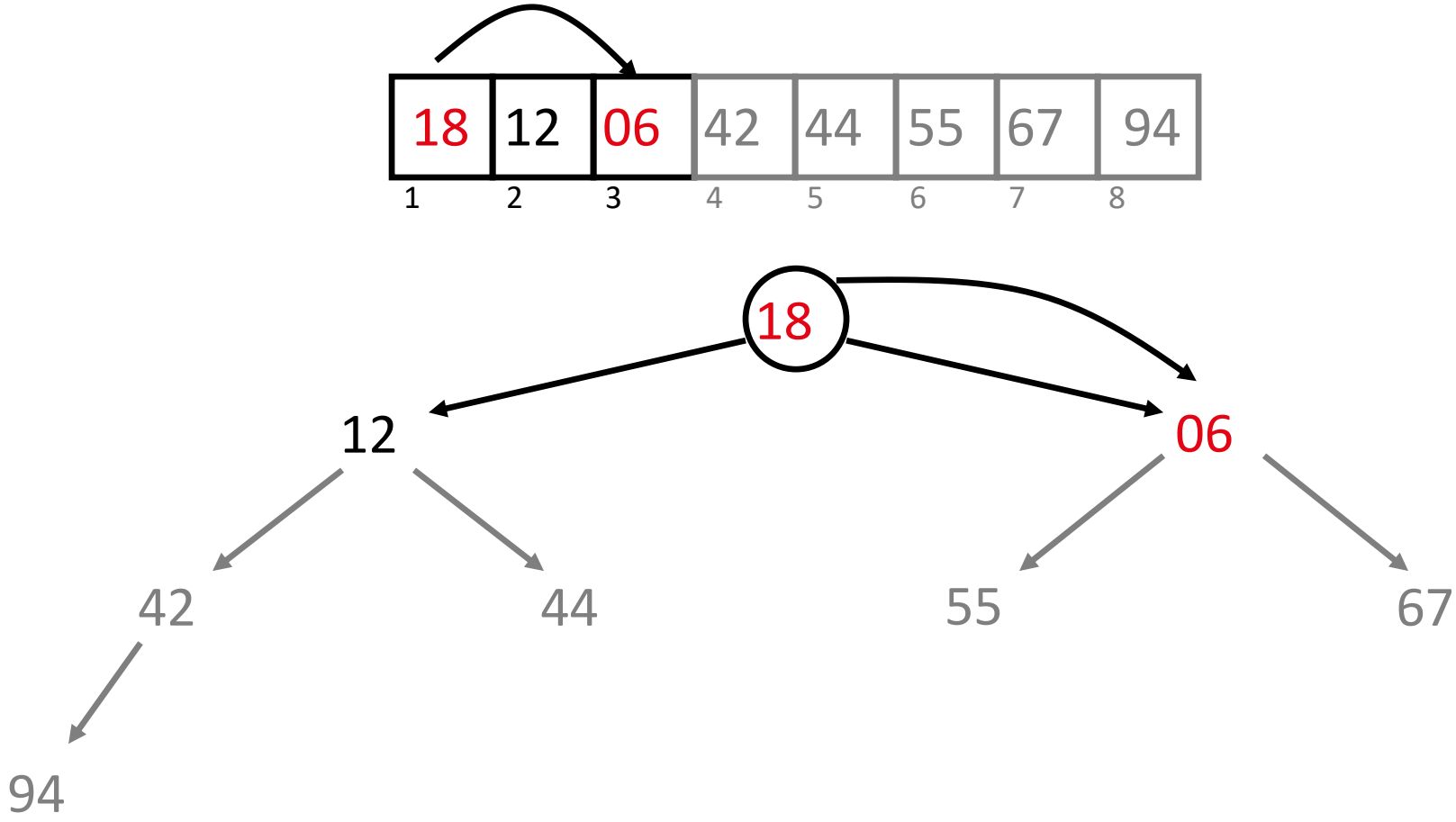
Tri par tas (« Heapsort »)



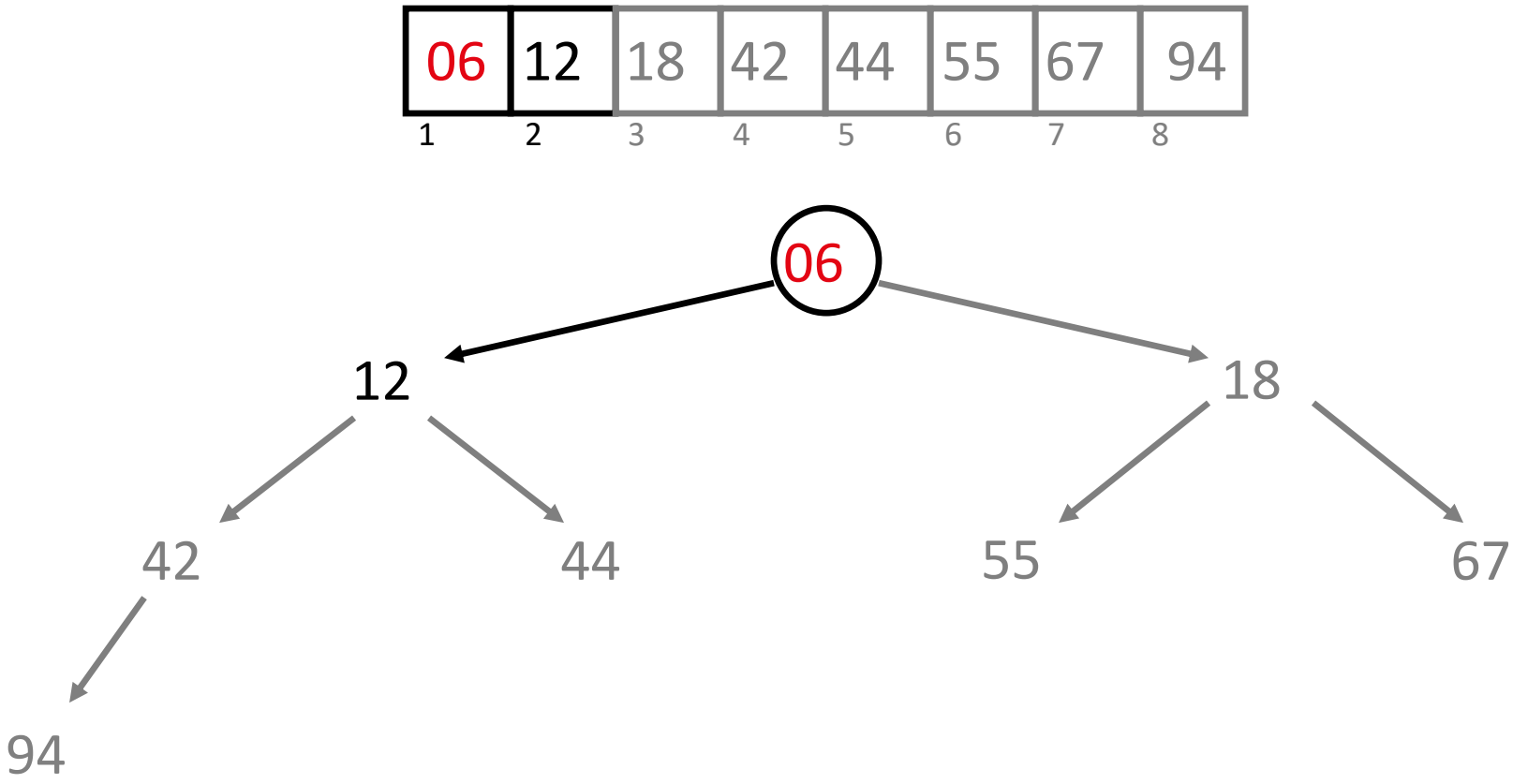
Tri par tas (« Heapsort »)



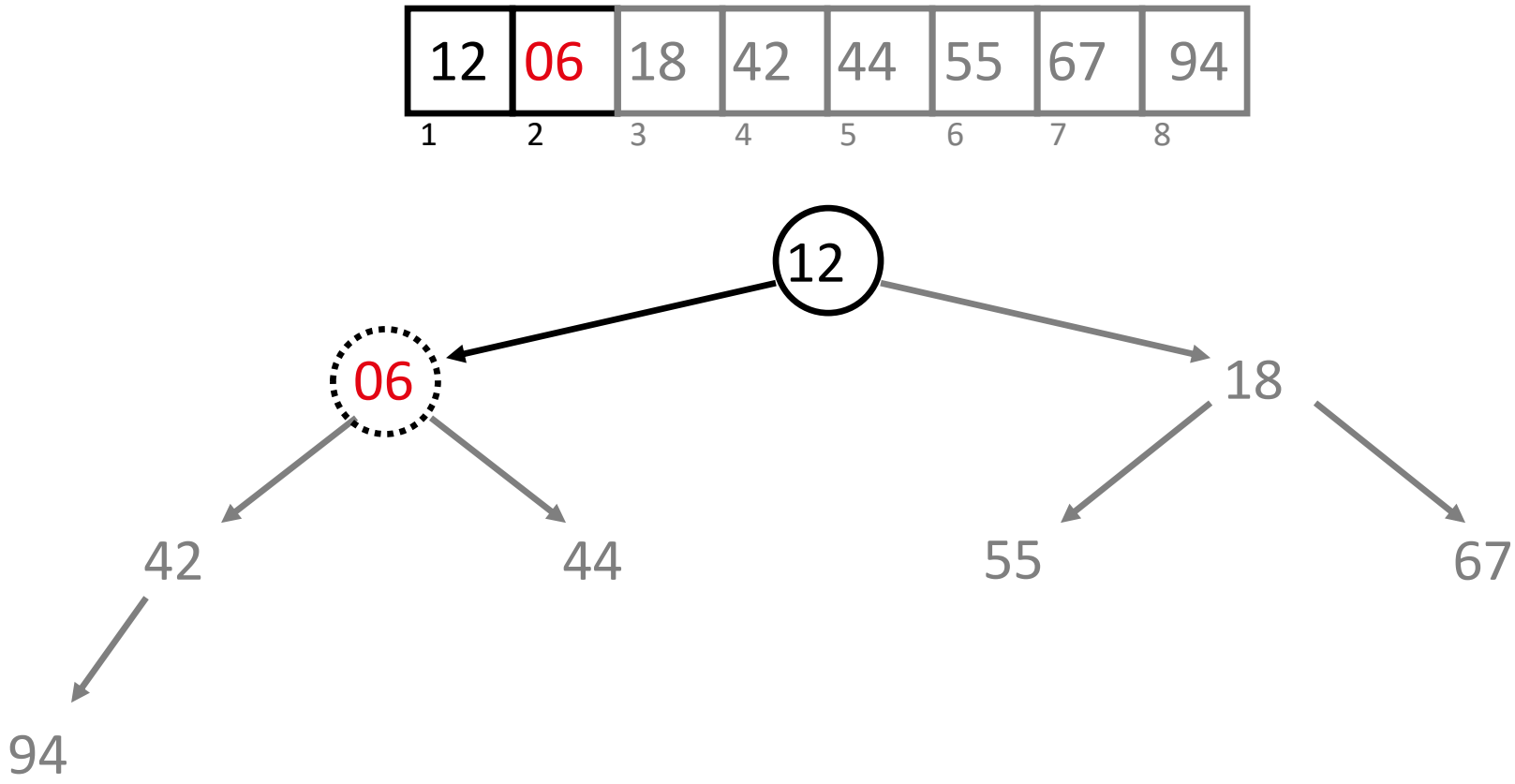
Tri par tas (« Heapsort »)



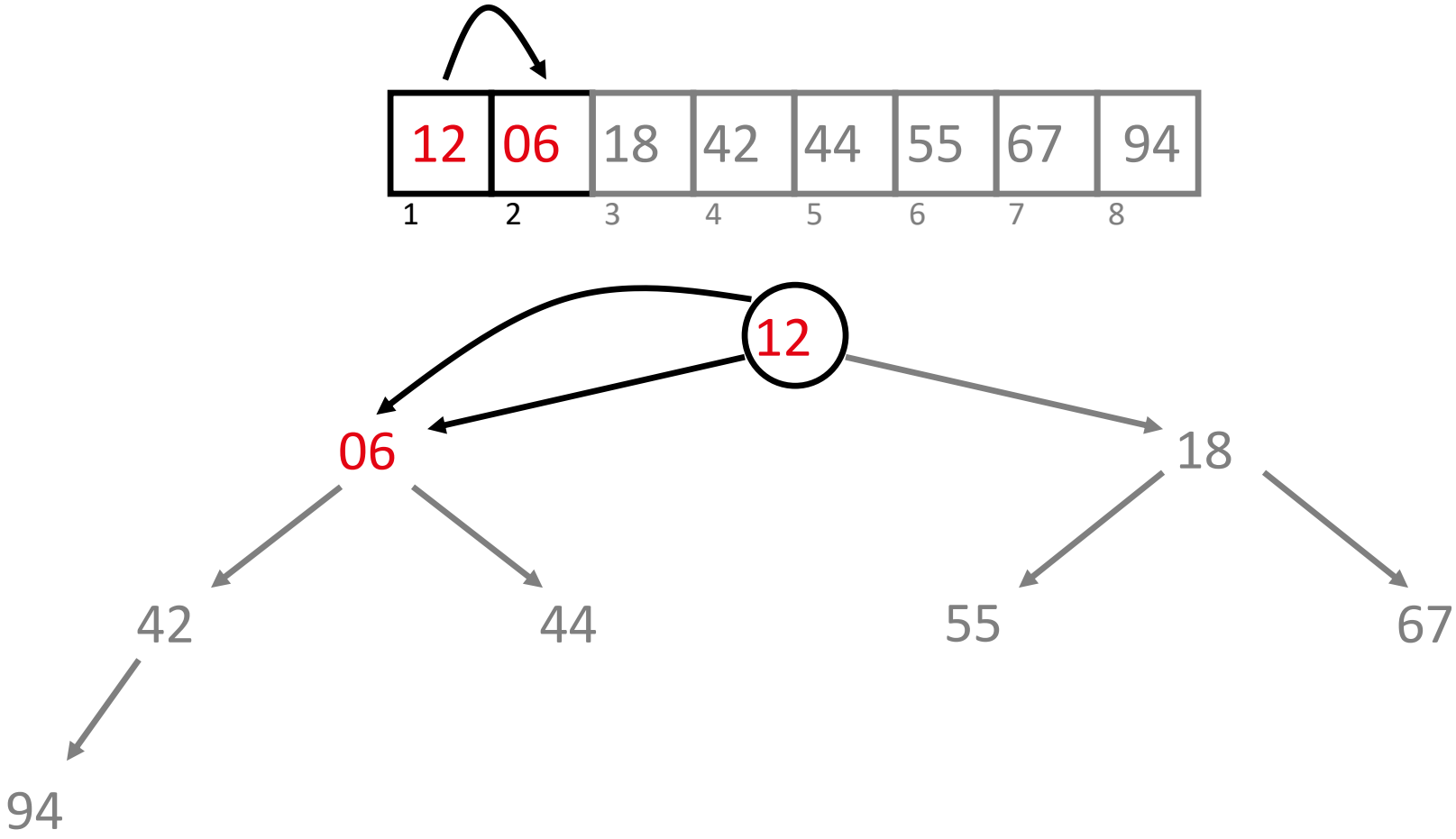
Tri par tas (« Heapsort »)



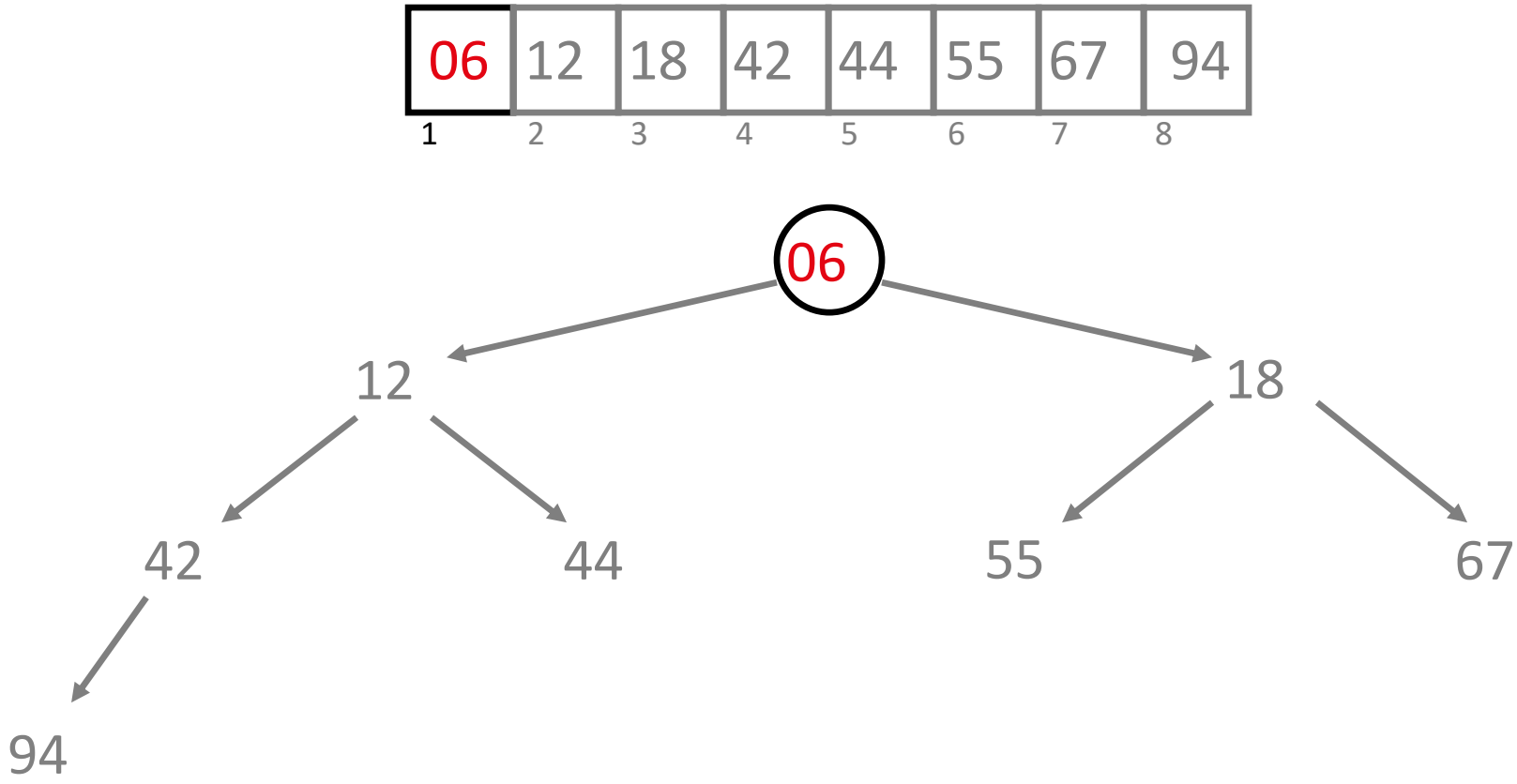
Tri par tas (« Heapsort »)



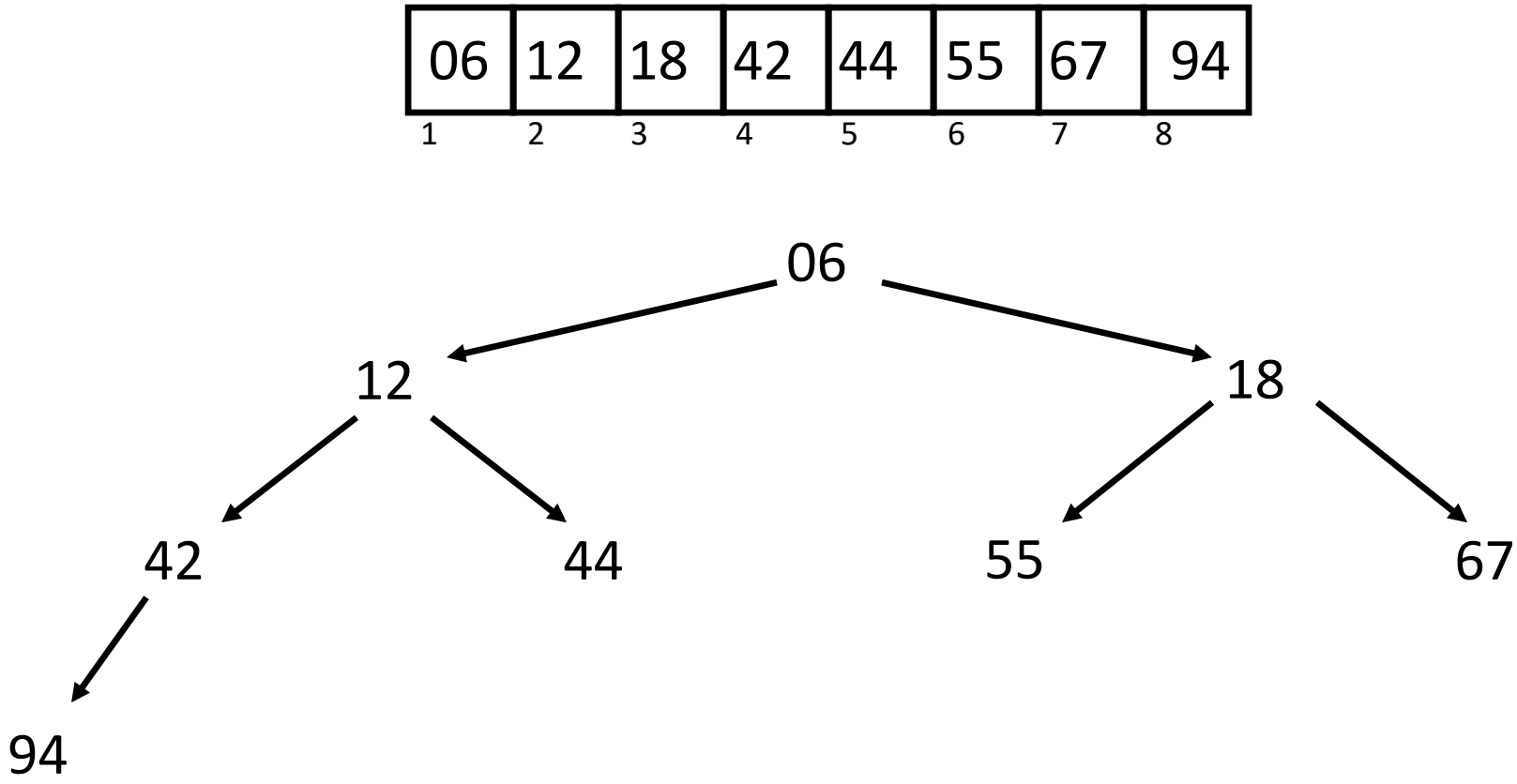
Tri par tas (« Heapsort »)



Tri par tas (« Heapsort »)



Tri par tas (« Heapsort »)



Synthèse

- Monceau/Tas
  - Arbre binaire complet dont la valeur de la clé d'un nœud est toujours supérieure où égale à celle de ses enfants (propriété du tas_max)
- Utilités
  - File à priorité
  - Tri par tas
- Hauteur d'un tas de n noeuds
- Nombre de feuilles d'un tas de n nœuds

Synthèse

- Construction bas vers le haut
  - en $O(n)$ dans tous les cas
- Insertion
  - nombre maximal de comparaisons requises $\lfloor log_2(n+1) \rfloor$
- Enlever la racine
  - nécessite au plus $O(\log n)$ comparaisons
- Tri par tas (« Heapsort »)
  - s'exécute donc en $O(n \log(n))$ en pire cas et en $O(n)$ en meilleur cas

Monceaux dans la STL

- Fonctions pour créer et manipuler des monceaux définies dans
  - <algorithm>
- Pour réarranger des éléments d'un conteneur se situant entre debut et fin (incluant debut et excluant fin) :
  - make_heap(Iterator debut, Iterator fin)
- Fonctions pour insérer et retirer un élément dans l'intervalle [debut, fin[:
  - push_heap(Iterator debut, Iterator fin)
  - pop_heap(Iterator debut, Iterator fin)

Monceaux dans la STL

- Ces fonctions ont une seconde version avec un troisième argument qui est un foncteur booléen qui permet de redéfinir l'opérateur de comparaison utilisé.
- Par défaut, c'est operator< du type des éléments qui est utilisé pour les comparaisons.
- Note: l'itérateur doit être un itérateur à accès direct (pouvant sauter de k éléments), limite l'utilisation de ces fonctions aux conteneurs fournissant de tels itérateurs (comme <vector> et <deque>; impossible avec <list>).

Exemples

```
#include <iostream>
#include <algorithm>
#include <vector>
int main() //exemple extrait de www.cplusplus.com
{
using namespace std;

        vector<int> v { 10, 20, 30, 5, 15 }; //vector de 5 entiers
        make_heap(v.begin(), v.end()); //repositionne les éléments de v en un tas_max
        cout << "initial max heap : " << v.front() << endl; //affiche 1er élément de v
        pop_heap(v.begin(), v.end()); //swap 1er et dernier de v et
                                                        //reconstruit tas sans le dernier élem
        v.pop_back(); //enlève ce dernier élément
        cout << "max heap after pop : " << v.front() << endl;

         //suite prochaine diapo ::
```

Exemples

```
// suite de int main() //exemple extrait de www.cplusplus.com


      v.push_back(99); //insère 99 à la fin de v qui contient un élément de plus
      push_heap(v.begin(), v.end()); //reconstruit le tas avec ce dernier élément ajouté
      cout << "max heap after push: " << v.front() << endl;
      sort_heap(v.begin(), v.end()); //tri le tas (il faut que v soit d'abord un tas)

      cout << "final sorted range :";
      for (unsigned int i = 0; i < v.size(); i++)
      {
            cout << ' ' << v[i];
            cout << endl;
      }
      return 0
}
```