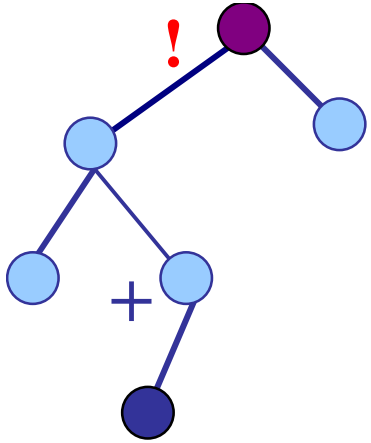
Algorithmes et structures de données
IFT-2008/GLO-2100

Mondher Bouden

Les structures arborescentes et les monceaux



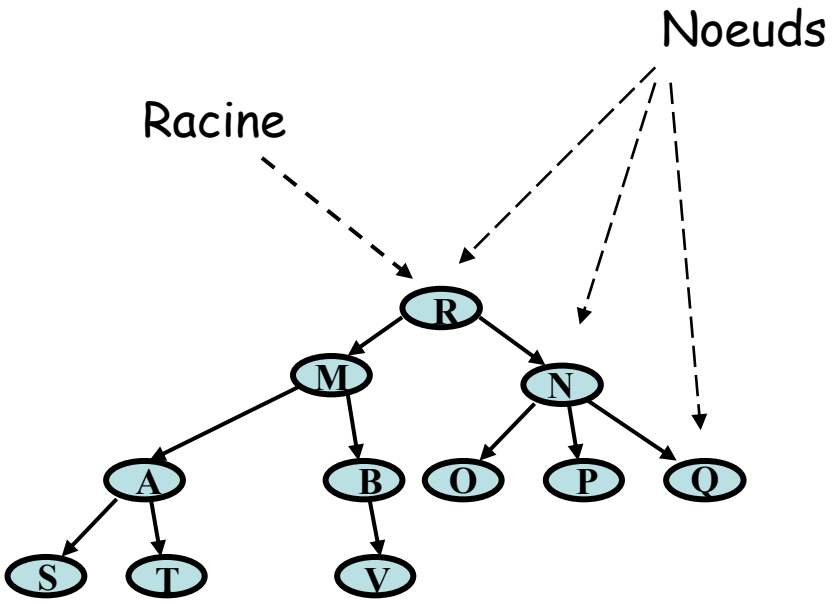
Édition HIVER 2022

Département d'informatique
et de génie logiciel

Notions importantes de ce chapitre

- 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).

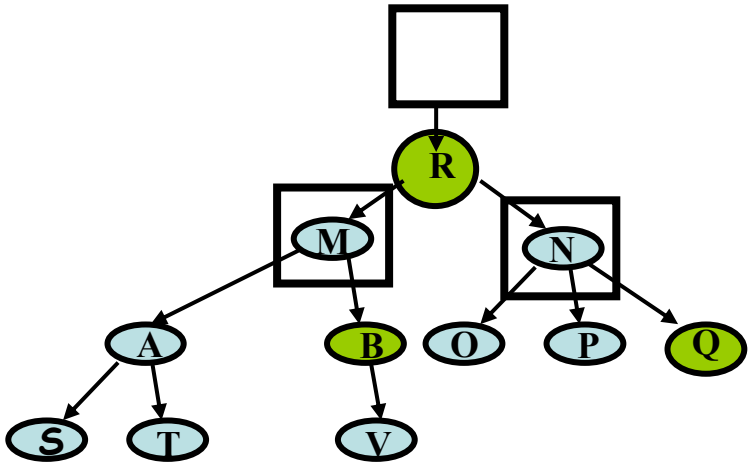
Terminologie des arbres

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

Parent(B) = M

Parent(R) = Nil

Parent(Q) = N



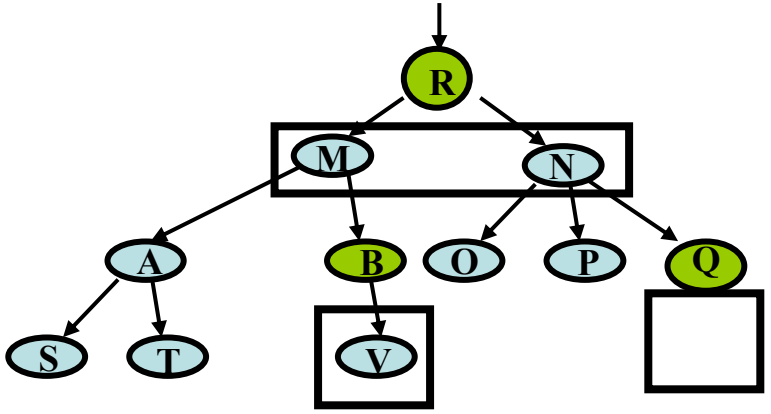
Terminologie des arbres

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

Enfant(B) = {V}

Enfants(R) = {M,N}

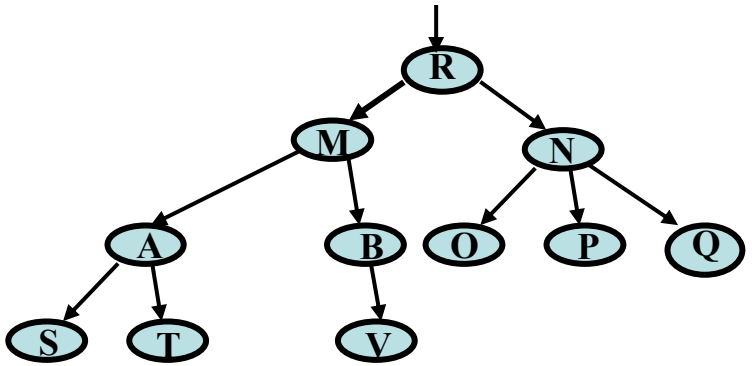
Enfant(Q) = { }



Terminologie des arbres

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

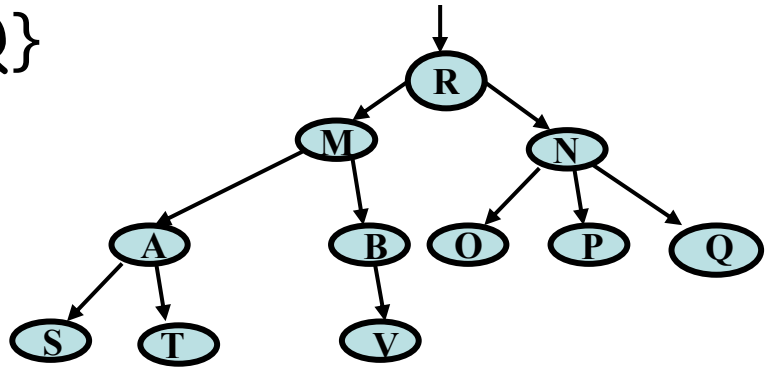
Racine(Arbre) = R



Terminologie des arbres

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



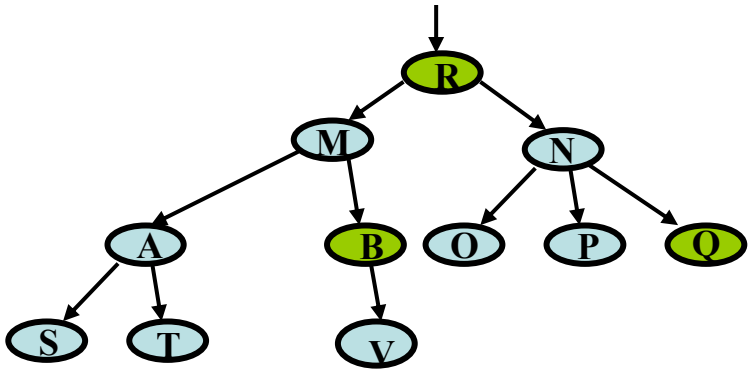
Terminologie des arbres

- **Parent d'un nœud (Père)** : Le nœud immédiatement prédécesseur.
- **Enfants d'un nœud (fils)** : Les nœuds immédiatement successeurs du nœud.
- **Racine** : Un 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}

Ancêtre(R) = { }

Ancêtres(Q) = {N,R}



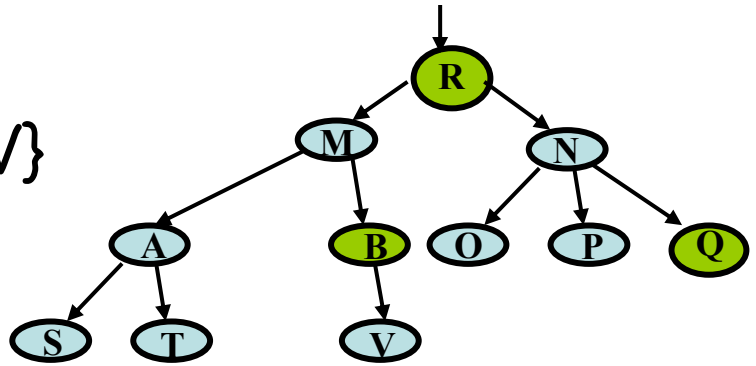
Terminologie des arbres

- **Parent d'un nœud** : Le nœud immédiatement prédécesseur.
- **Enfants d'un nœud** : 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.
- **Descendants d'un nœud** : Tous les nœuds successeurs jusqu'aux feuilles accessibles par ce nœud.

Descendant(B) = {V}

Descendants(R) = {M,N,A,…,V}

Descendant(Q) = { }



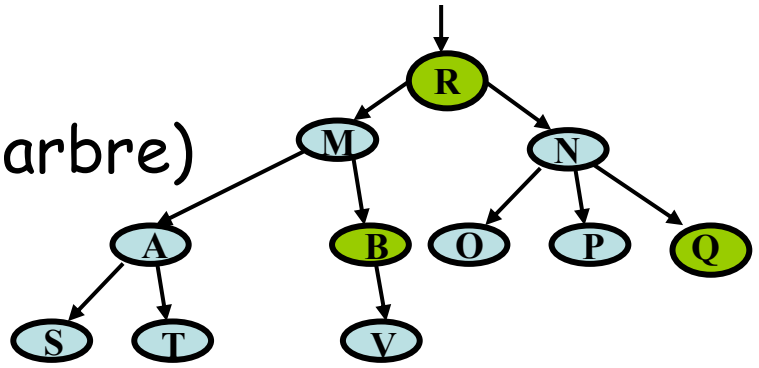
Terminologie des arbres

- **Parent d'un nœud** : Le nœud immédiatement prédécesseur.
- **Enfant(s) d'un nœud** : 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.
- **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



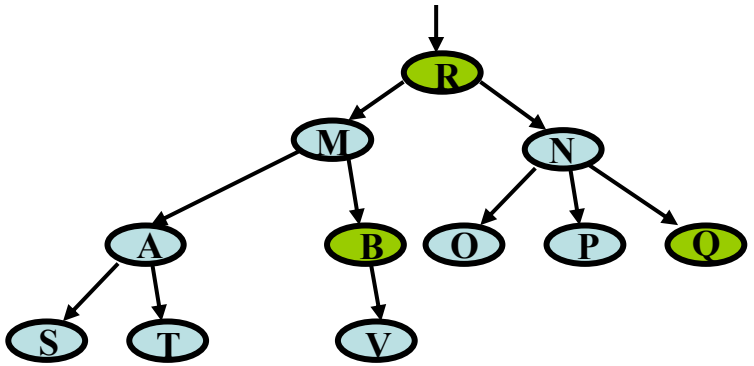
Terminologie des arbres

- **Parent d'un nœud** : Le nœud immédiatement prédécesseur.
- **Enfant(s) d'un nœud** : 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.
- **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

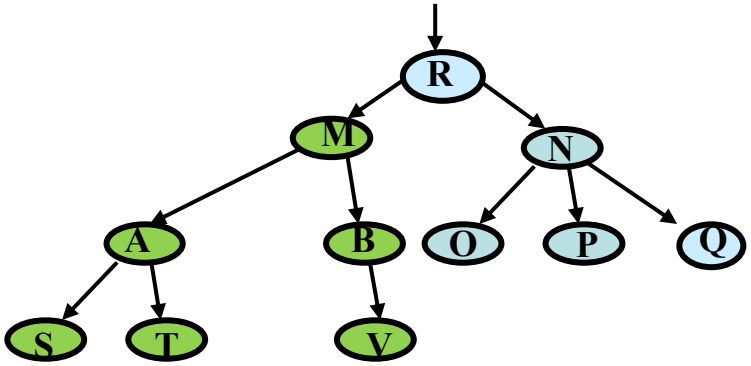
Niveau(R) = 0

Niveau(Q) = 2



Terminologie des arbres

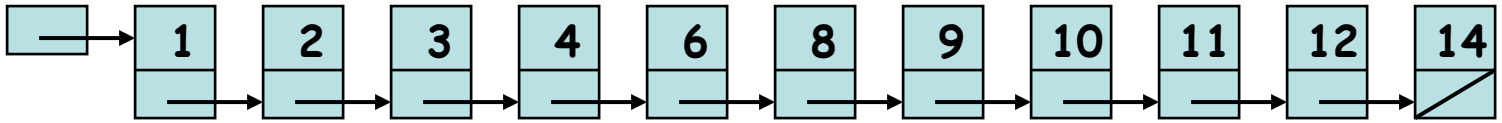
- Le **sous-arbre** de racine M est l'arbre constitué de M et de ses descendants. Ce sous-arbre est aussi appelé le **sous-arbre gauche de R**



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

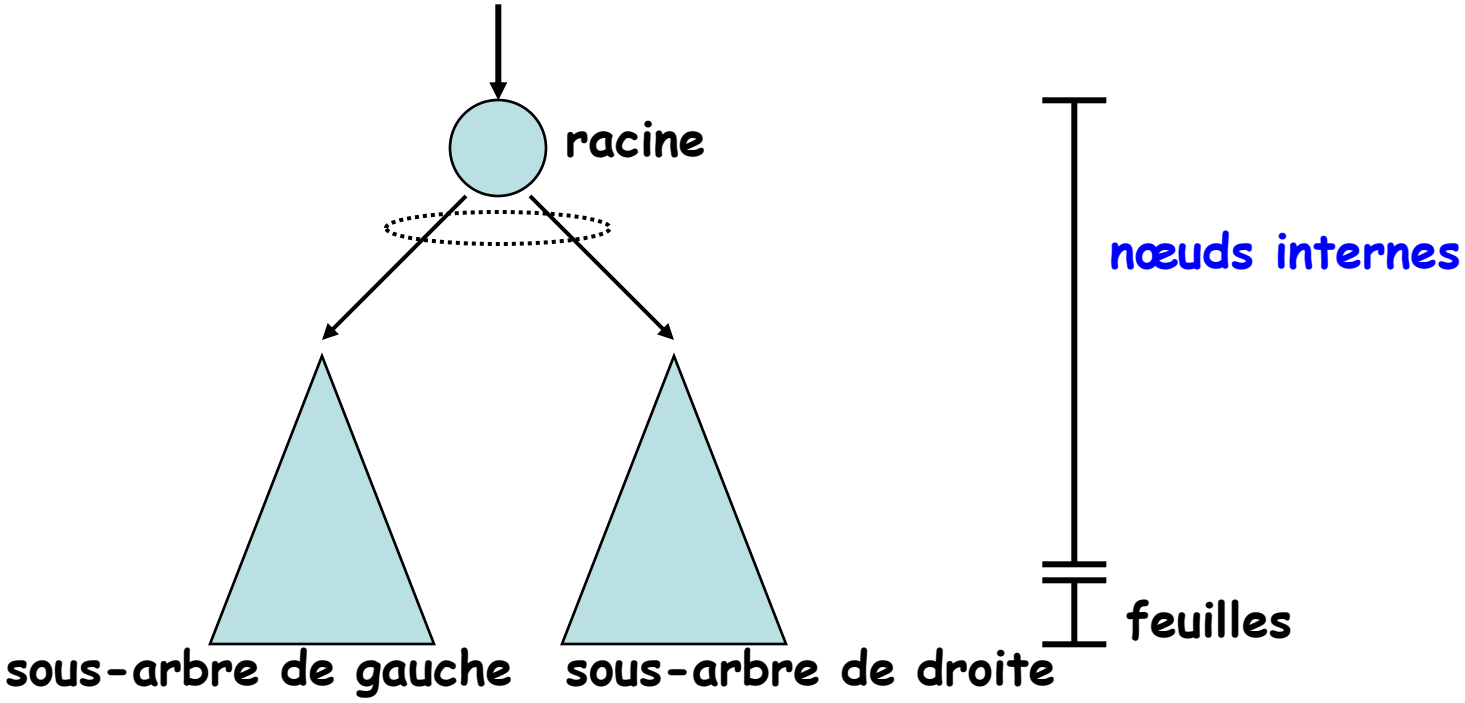
Une liste simplement chaînée est un arbre de degré 1

- **liste simplement chaînée:**
  - On dit que l'arbre est **dégénéré**



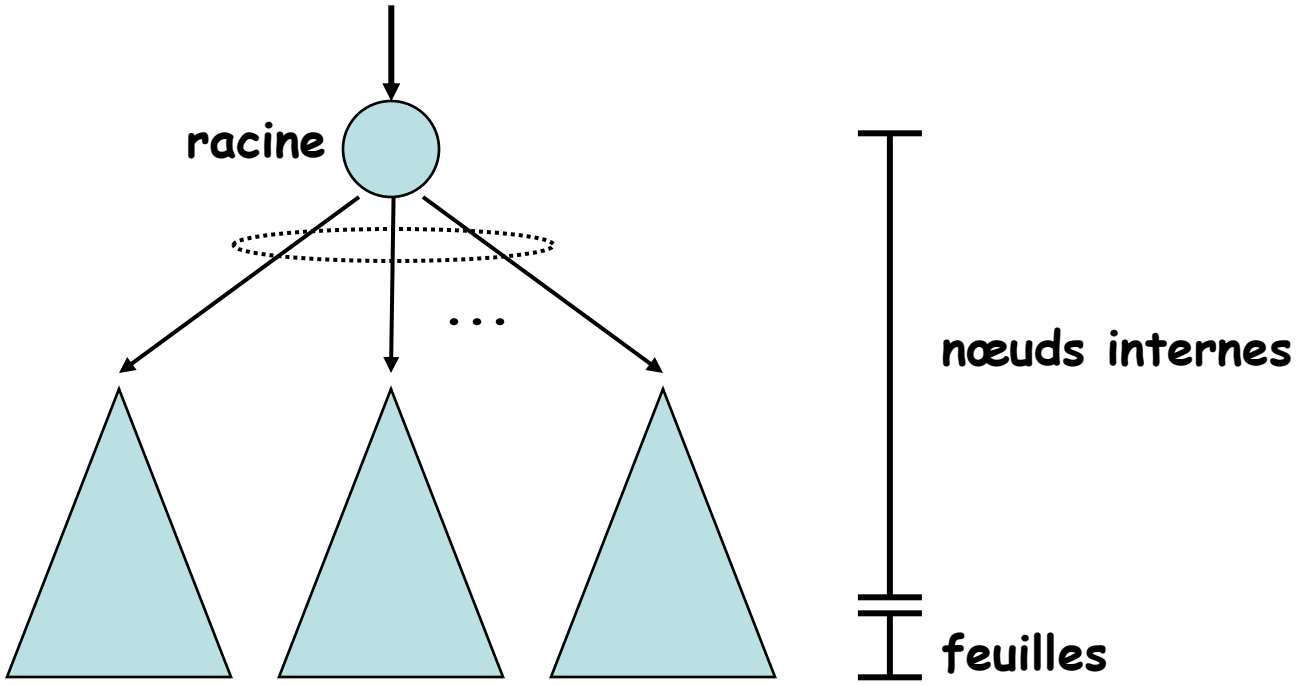
Arbre binaire

Un arbre de **degré 2** est appelé ***arbre binaire***.

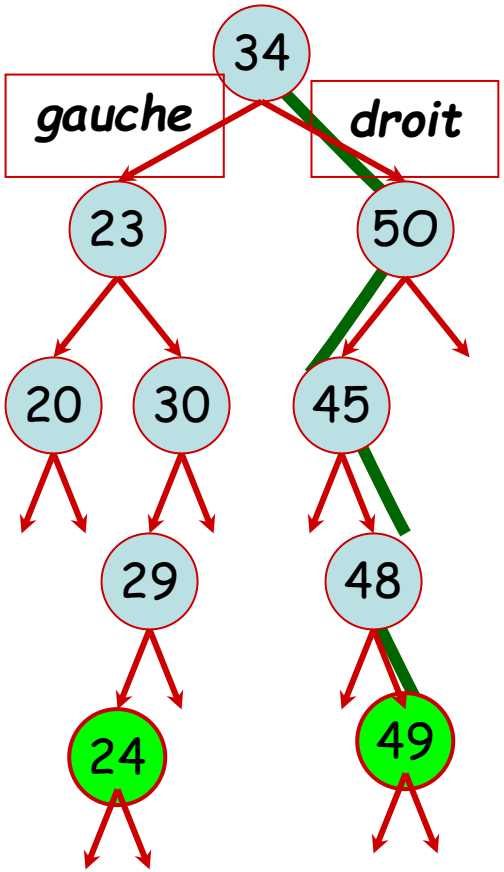


Arbres n-aire

Un arbre de **degré n** est appelé ***arbre n-aire***.



Les arbres binaires de recherche



Un arbre binaire est dit de recherche si chaque nœud possède une clé dont les valeurs satisfont la propriété suivante:

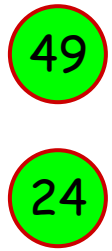
- Pour tout noeud de l'arbre la valeur de sa clé est:
  - > aux clés de tous les noeuds de son sous-arbre gauche.
  - < aux clés de tous les nœuds de son sous-arbre droit.

- Toutes les clés doivent donc être distinctes.
- Tout ajout ou suppression de nœud doit donc maintenir cette propriété vraie.

Ajout de la valeur **49** :

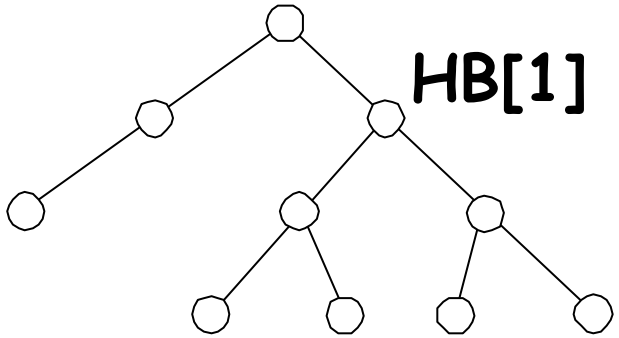
Ajoute de la valeur **24**

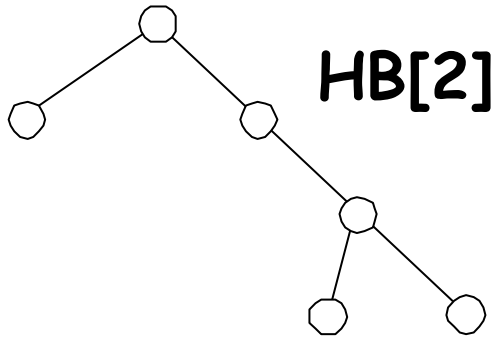
Remarque :
Tout ajout se fait par une feuille.



Le facteur d'équilibre pour arbres binaires de recherche

- 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.
- Le **facteur d'équilibre** **(Height-Balanced, HB[k])** d'un arbre est donné par la valeur maximale de cette différence de hauteur parmi tous les nœuds de l'arbre.

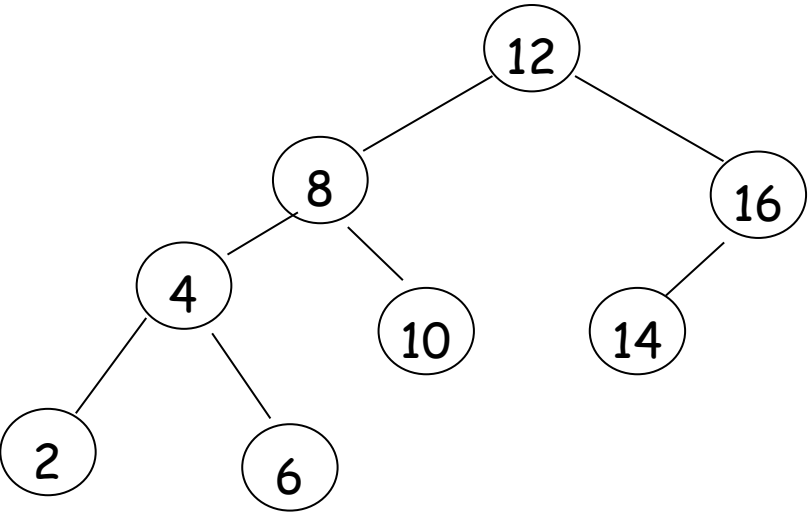




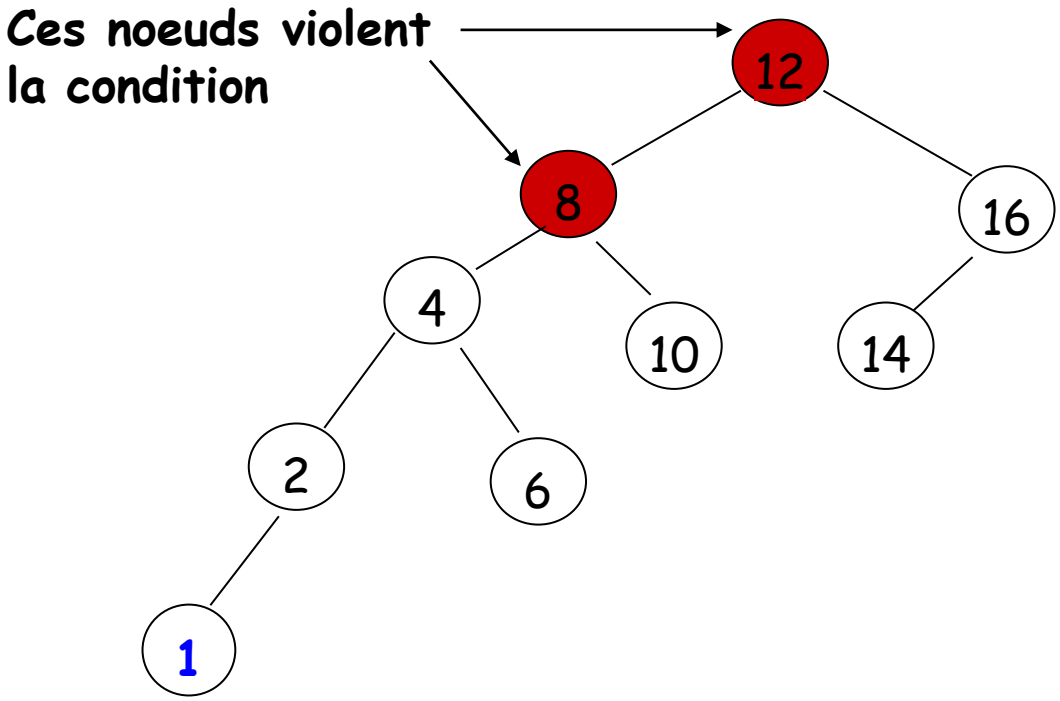
- **Un arbre AVL** (du nom de leurs inventeurs Adelson-Velsky et Landis en 1962) est un arbre HB[1] maintenu à l'équilibre grâce à des opérations de **rotations** effectuées lors des insertions et suppressions (chapitre suivant).

- **Nous verrons qu'un arbre équilibré HB[1] de n noeuds possède une hauteur en O(Log n).**
  - La recherche d'un élément se fera donc en temps O(Log n).
  - c'est la 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

Parcours d'arbre

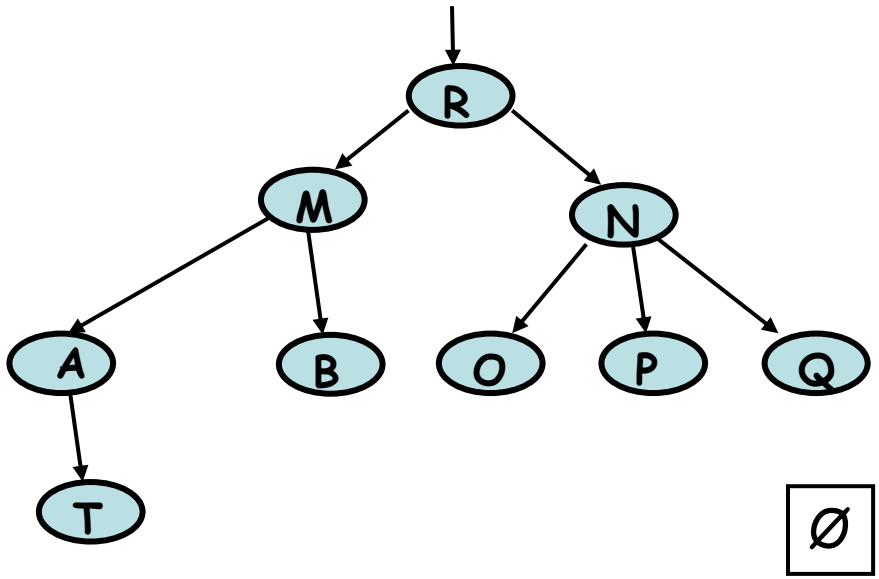
- Fréquemment, nous devons parcourir (ou visiter) tous les nœuds d'un arbre. Typiquement, lorsqu'un nœud est visité:
  - on affiche une clé, une valeur, ou un mot
  - on mets à jour une valeur, un objet, etc…
- La méthode de parcours utilisée nous défini un **itérateur** pour l'arbre
  - cela défini l'ordre dans lequel seront visités les nœuds.
- On peut utiliser les parcours en profondeur ou en largeur définis pour les graphes (et donc aussi définis pour les arbres).
- Mais on utilise habituellement l'un des trois ordres de visite suivants qui sont définis uniquement pour les arbres:
  - Le parcours en pré-ordre (donne la même chose qu'un DFS)
  - Le parcours en post-ordre
  - Le parcours en-ordre (symétrique)

Itérateurs d'arbre

Dans le parcours **pré-ordre**, les descendants d'un nœud sont traités **<u>après</u>** lui:

priorité au père (pré-ordre)

1. *visiter la racine r ;*
2. *visiter* ***récursivement*** *les enfants :* $v_1, v_2, \ldots, v_k$

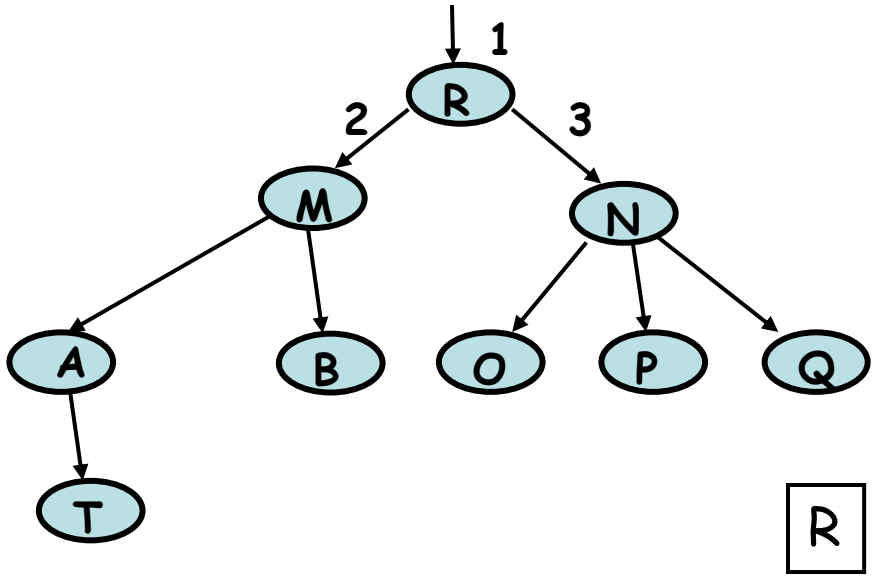


Itérateurs d'arbre

priorité au père (pré-ordre)

1. visiter la racine r ;

2. visiter récursivement les enfants : v_1, v_2, …, v_k

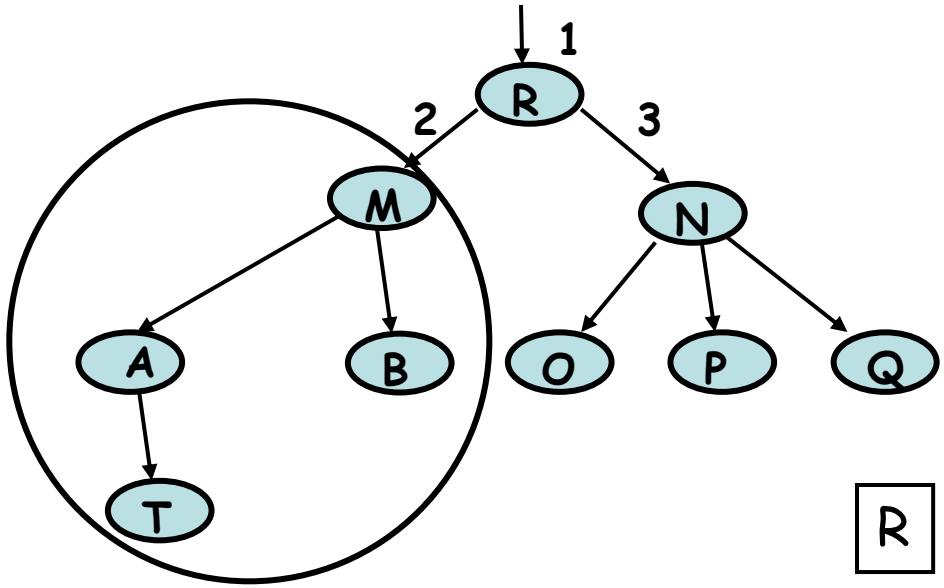


Itérateurs d'arbre

priorité au père (pré-ordre)

1. visiter la racine r ;

2. visiter récursivement les enfants : v_1, v_2, …, v_k

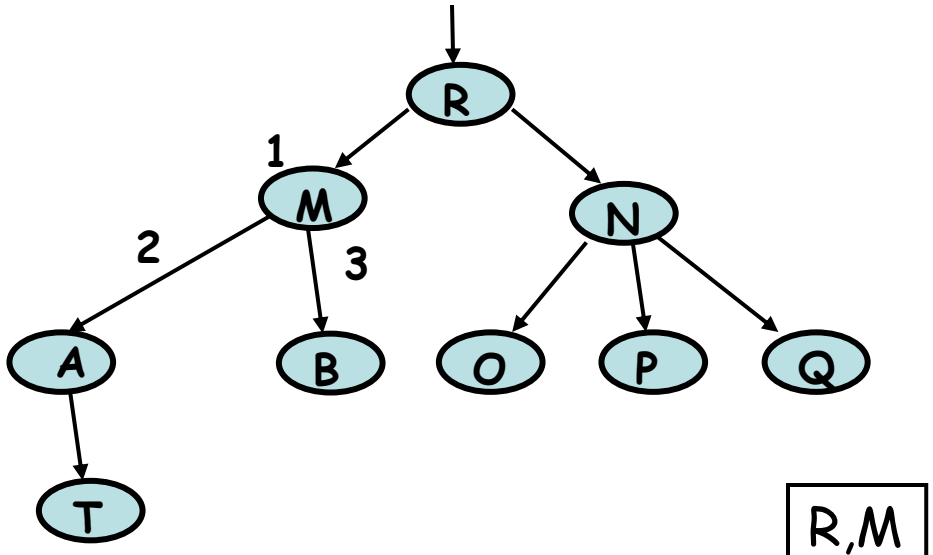


Itérateurs d'arbre

priorité au père (pré-ordre)

1. visiter la racine r ;

2. visiter récursivement les enfants : v_1, v_2, …, v_k

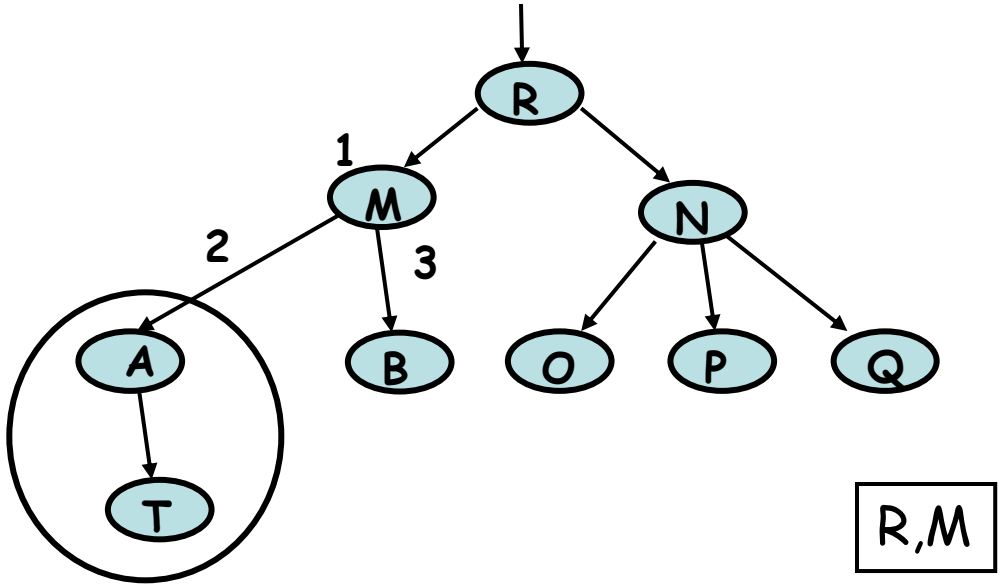


Itérateurs d'arbre

priorité au père (pré-ordre)

1. visiter la racine r ;

2. visiter récursivement les enfants : v_1, v_2, …, v_k

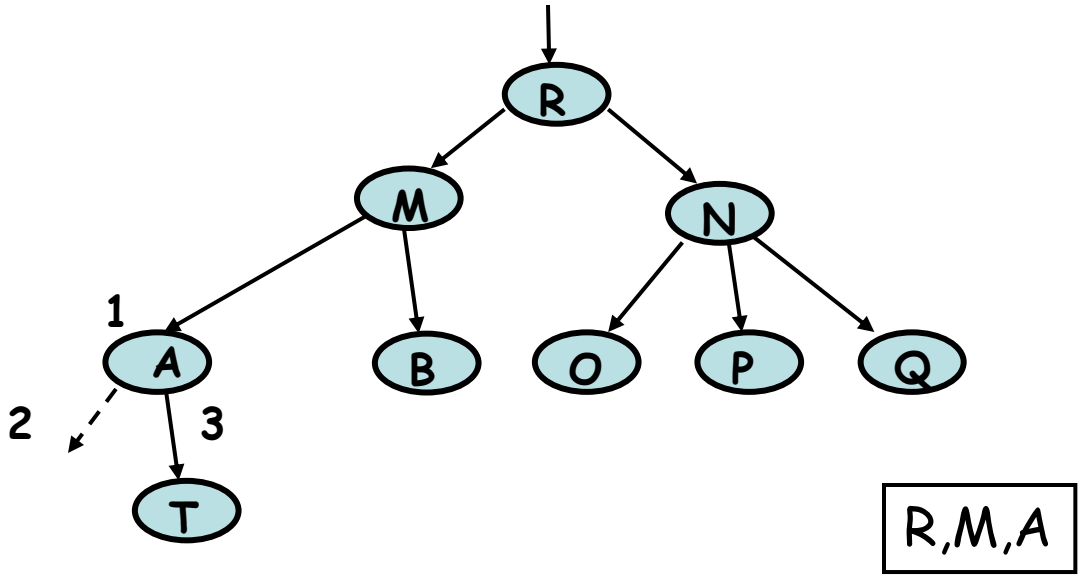


Itérateurs d'arbre

priorité au père (pré-ordre)

1. visiter la racine r ;

2. visiter récursivement les enfants : v_1, v_2, …, v_k

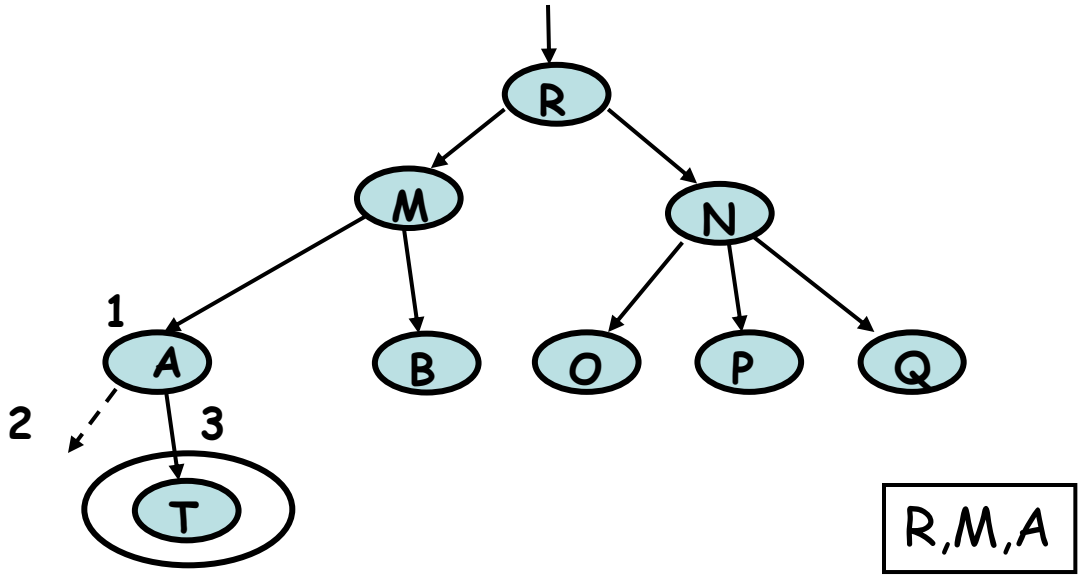


Itérateurs d'arbre

priorité au père (pré-ordre)

1. visiter la racine r ;

2. visiter récursivement les enfants : v_1, v_2, …, v_k

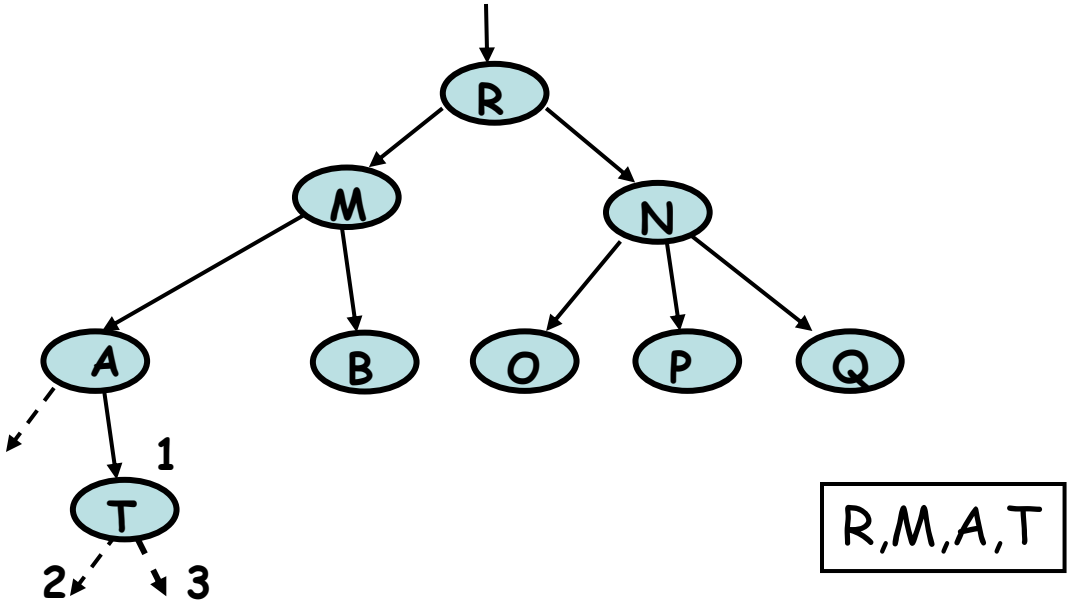


Itérateurs d'arbre

priorité au père (pré-ordre)

1. visiter la racine r ;

2. visiter récursivement les enfants : v_1, v_2, …, v_k



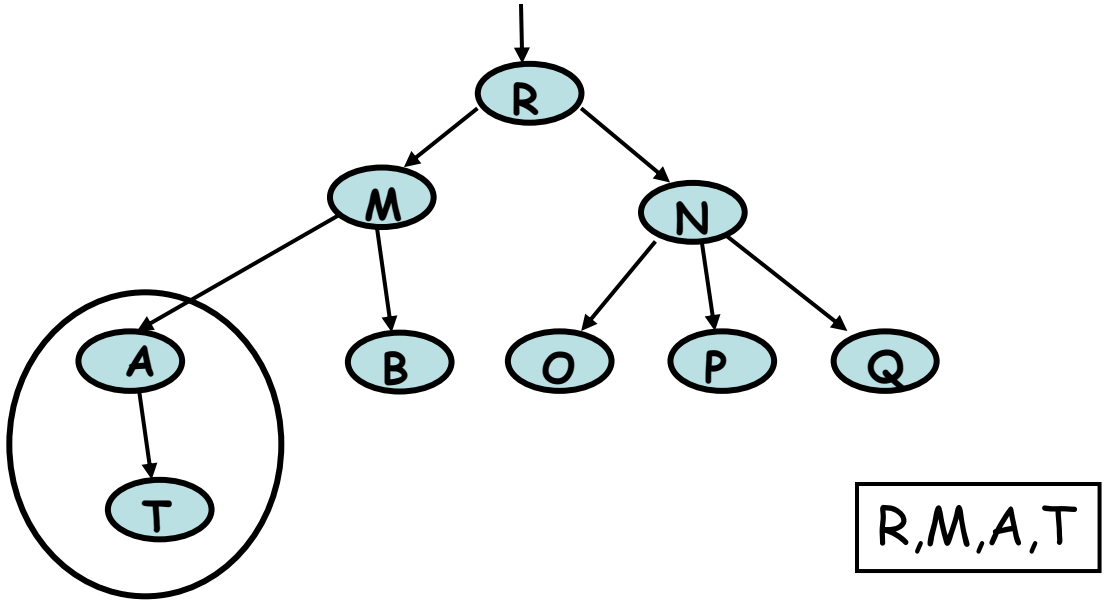
R,M,A,T

Itérateurs d'arbre

priorité au père (pré-ordre)

1. visiter la racine r ;

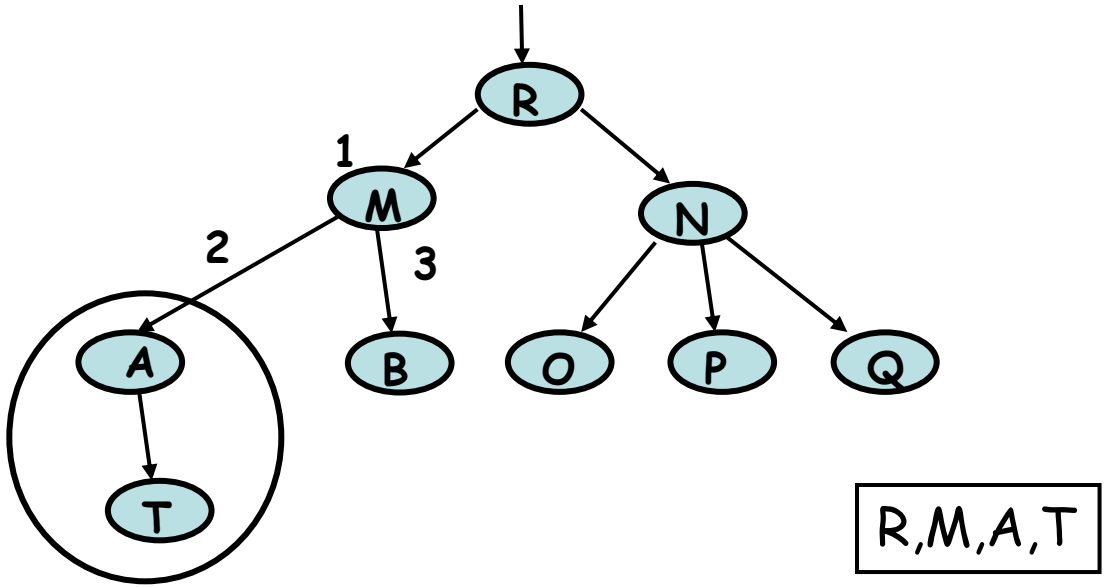
2. visiter récursivement les enfants : v_1, v_2, …, v_k



Itérateurs d'arbre

priorité au père (pré-ordre)

1. *visiter la racine r ;*
2. *visiter récursivement les enfants : v_1, v_2, …, v_k*

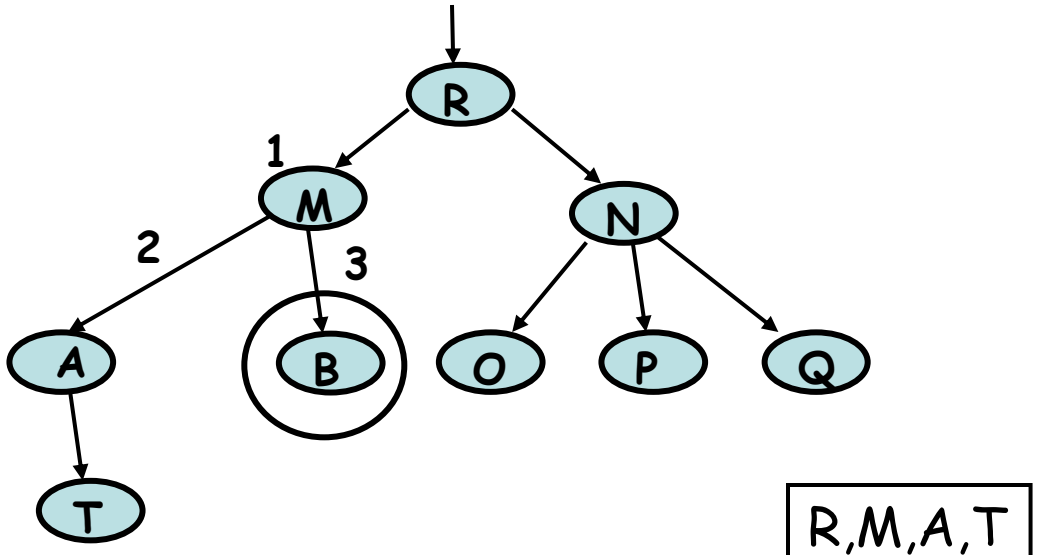


Itérateurs d'arbre

priorité au père (pré-ordre)

1. visiter la racine r ;

2. visiter récursivement les enfants : v_1, v_2, …, v_k



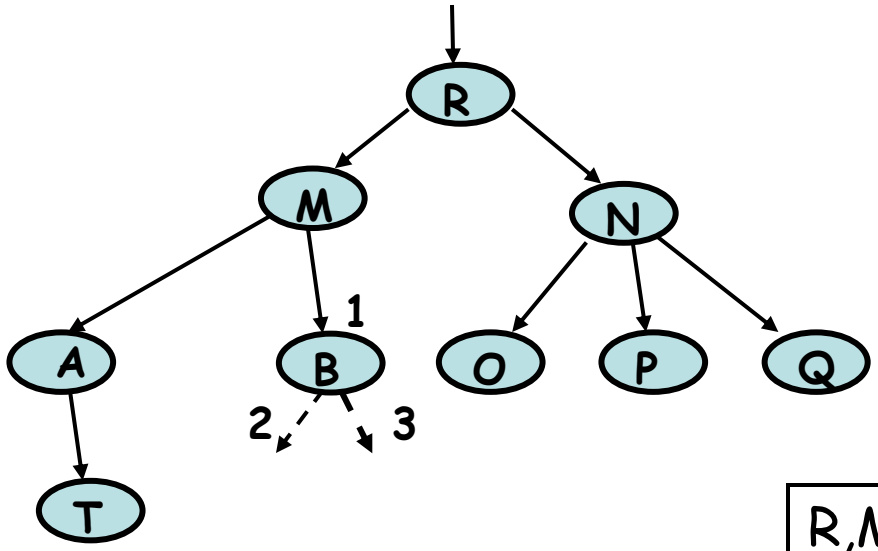
R,M,A,T

Itérateurs d'arbre

priorité au père (pré-ordre)

1. visiter la racine r ;

2. visiter récursivement les enfants : v_1, v_2, …, v_k



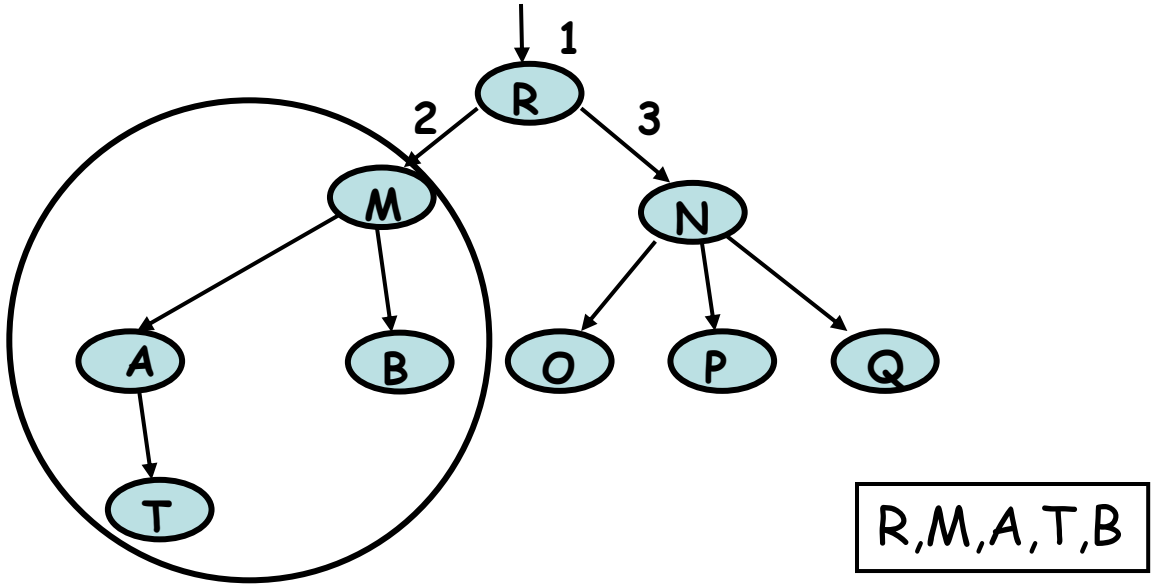
R,M,A,T,B

Itérateurs d'arbre

priorité au père (pré-ordre)

1. visiter la racine r ;

2. visiter récursivement les enfants : v_1, v_2, …, v_k

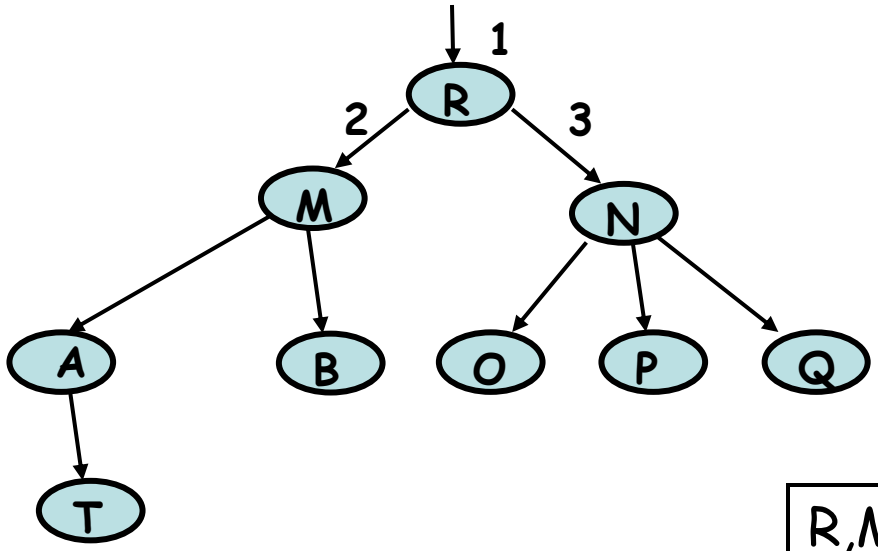


Itérateurs d'arbre

priorité au père (pré-ordre)

1. visiter la racine r ;

2. visiter récursivement les enfants : v_1, v_2, …, v_k



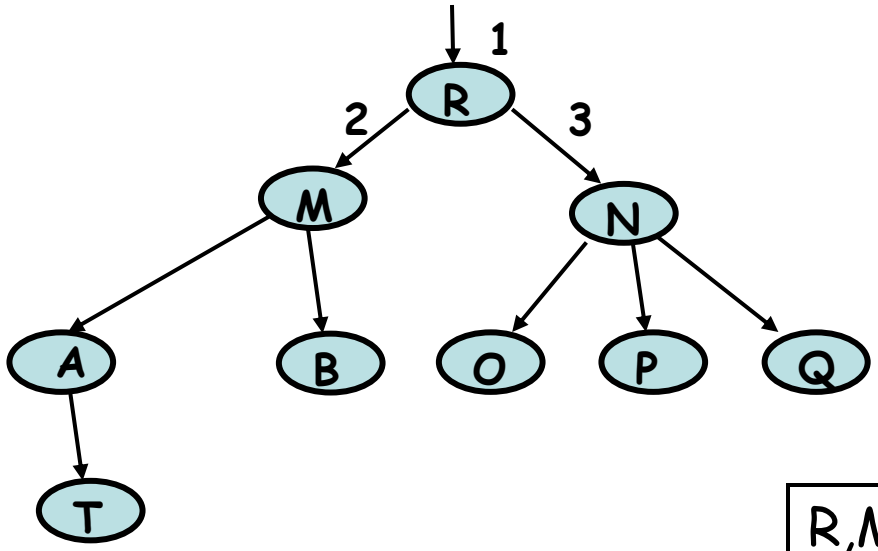
R,M,A,T,B,...

Itérateurs d'arbre

priorité au père (pré-ordre)

1. visiter la racine r ;

2. visiter récursivement les enfants : v_1, v_2, …, v_k



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

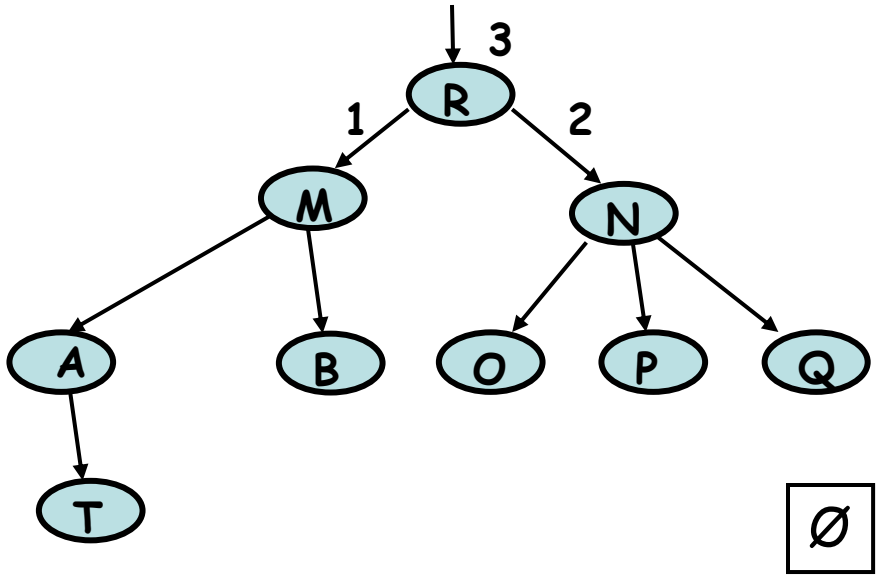
Itérateurs d'arbre

Dans le 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, …, v_k

2. visiter la racine r ;

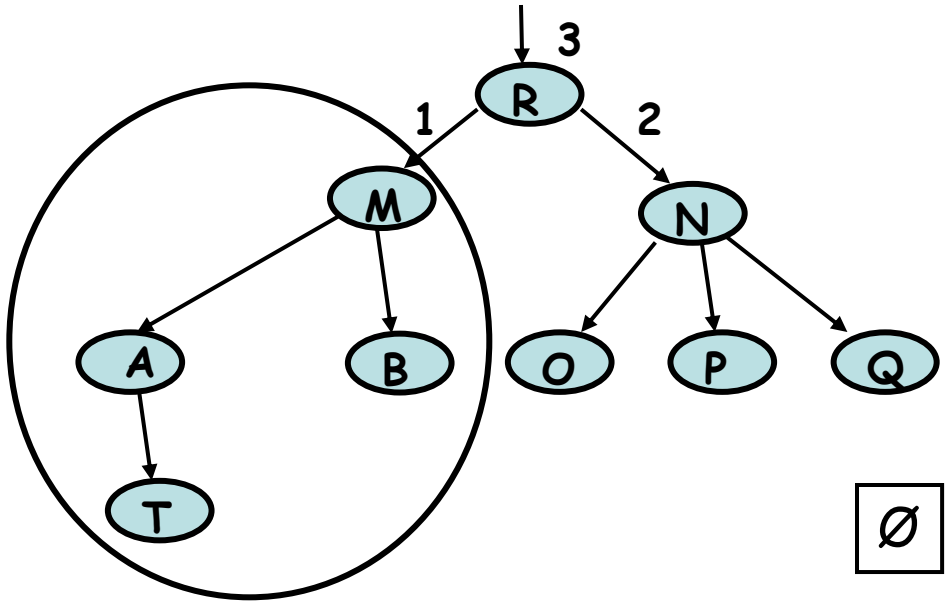


Itérateurs d'arbre

priorité aux fils (post-ordre)

1. visiter récursivement les enfants : v_1, v_2, …, v_k

2. visiter la racine r ;

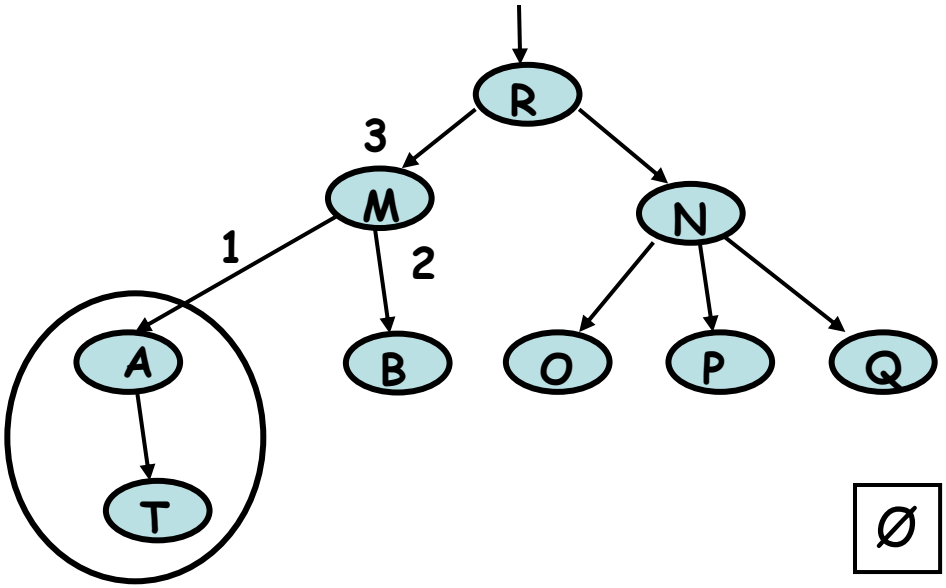


Itérateurs d'arbre

priorité aux fils (post-ordre)

1. visiter récursivement les enfants : v_1, v_2, …, v_k

2. visiter la racine r ;

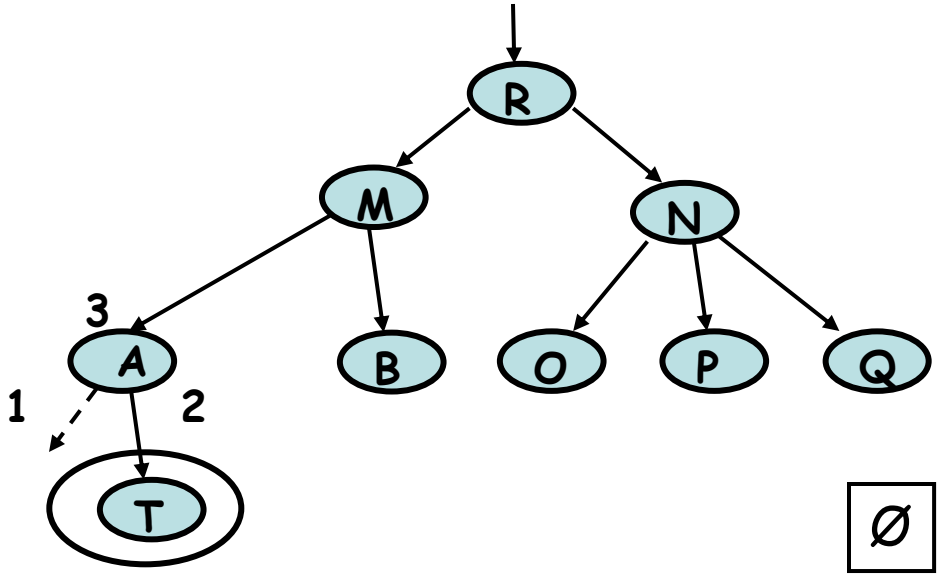


Itérateurs d'arbre

priorité aux fils (post-ordre)

1. visiter récursivement les enfants : v_1, v_2, …, v_k

2. visiter la racine r ;

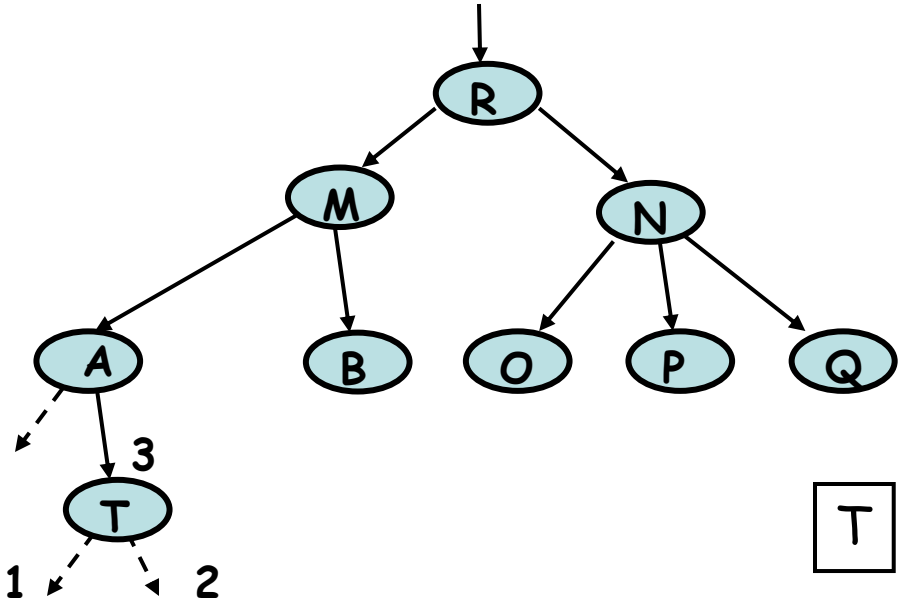


Itérateurs d'arbre

priorité aux fils (post-ordre)

1. visiter récursivement les enfants : v_1, v_2, …, v_k

2. visiter la racine r ;

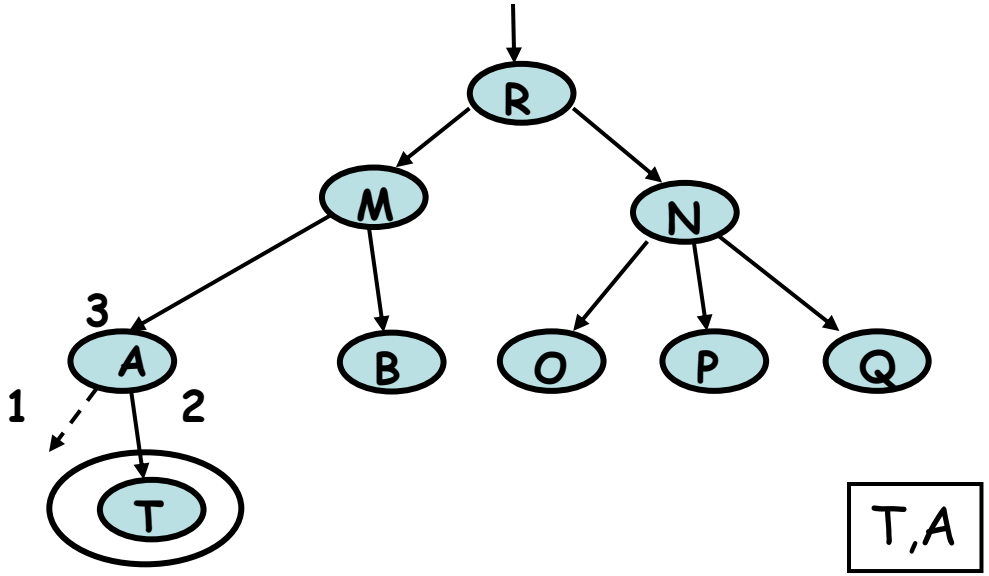


Itérateurs d'arbre

priorité aux fils (post-ordre)

1. visiter récursivement les enfants : v_1, v_2, …, v_k

2. visiter la racine r ;

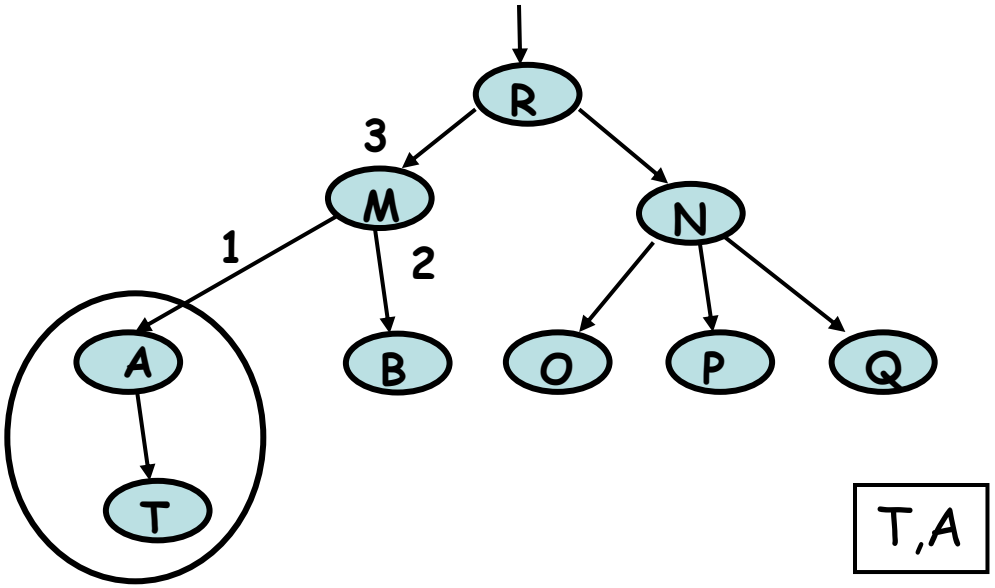


Itérateurs d'arbre

priorité aux fils (post-ordre)

1. visiter récursivement les enfants : v_1, v_2, …, v_k

2. visiter la racine r ;

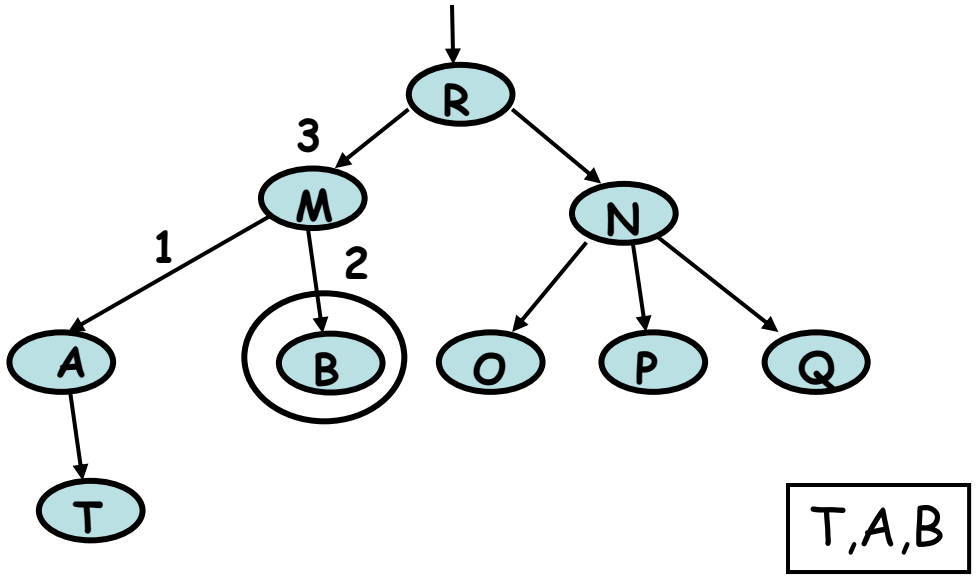


Itérateurs d'arbre

priorité aux fils (post-ordre)

1. visiter récursivement les enfants : v_1, v_2, …, v_k

2. visiter la racine r ;

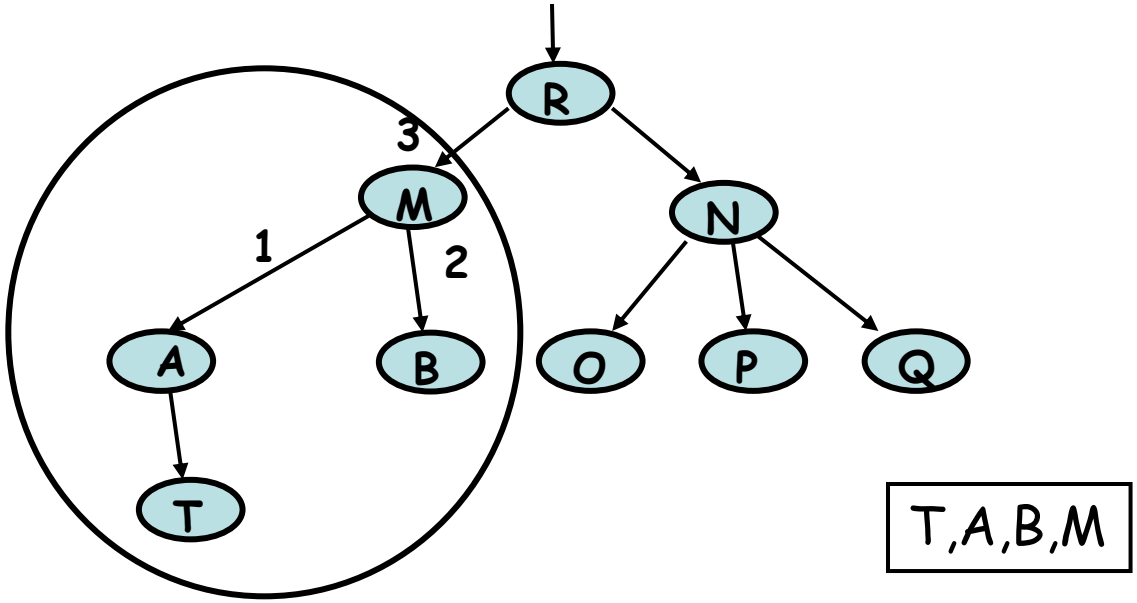


Itérateurs d'arbre

priorité aux fils (post-ordre)

1. visiter récursivement les enfants : v_1, v_2, …, v_k

2. visiter la racine r ;



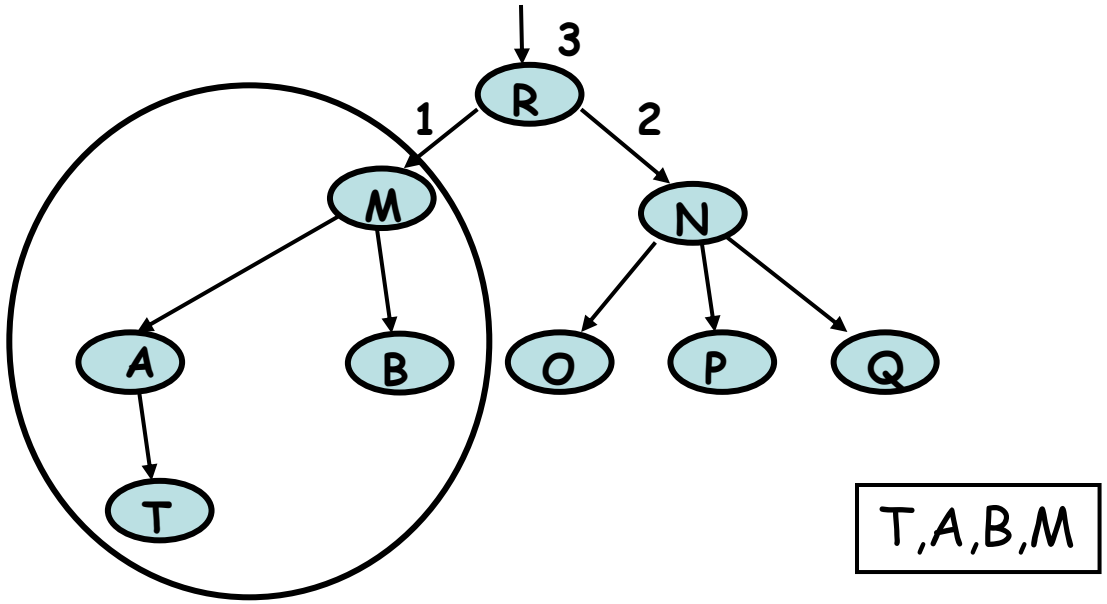
T,A,B,M

Itérateurs d'arbre

priorité au fils (post-ordre)

1. visiter récursivement les enfants : v_1, v_2, …, v_k

2. visiter la racine r ;

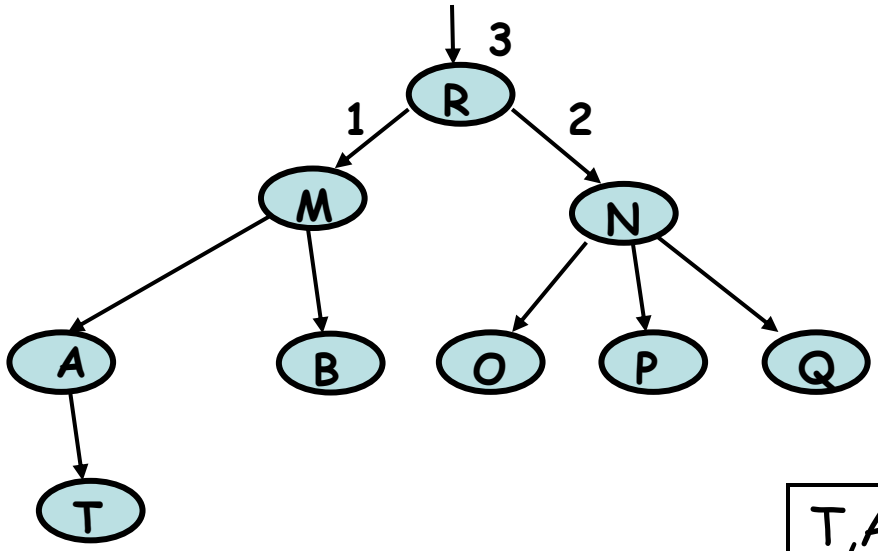


Itérateurs d'arbre

priorité aux fils (post-ordre)

1. visiter récursivement les enfants : v_1, v_2, …, v_k

2. visiter la racine r ;



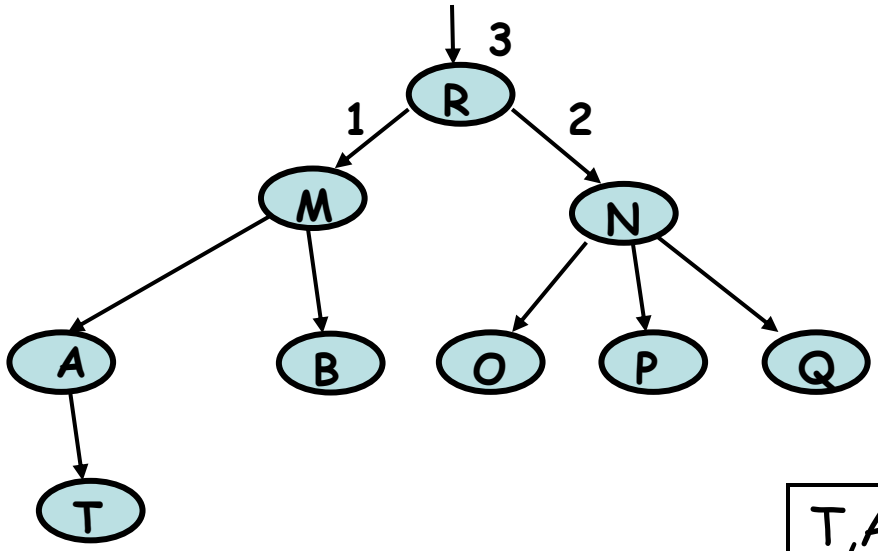
T,A,B,M,...

Itérateurs d'arbre

priorité aux fils (post-ordre)

1. visiter récursivement les enfants : v_1, v_2, …, v_k

2. visiter la racine r ;



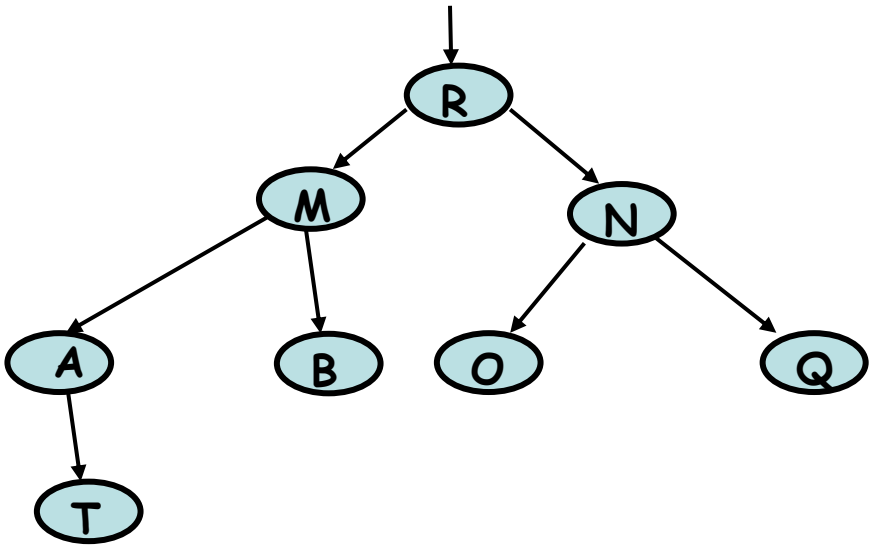
T,A,B,M,O,P,Q,N,R

Itérateurs d'arbre

Dans le parcours en ordre d'un arbre binaire, un descendant est traité **avant** le nœud, l'autre est traité **après** lui:

ordre symétrique (en-ordre)

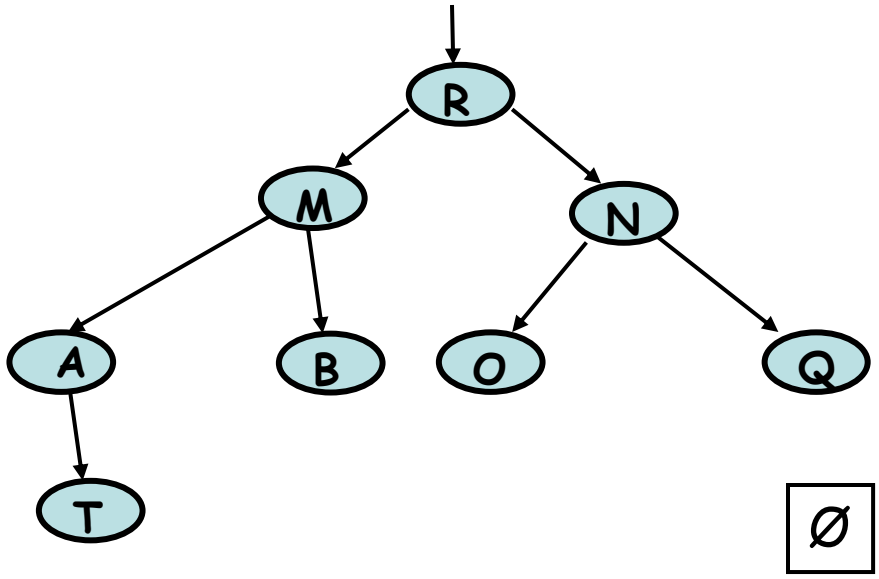
1. *visiter l'enfant de gauche (v_1)*
2. *visiter la racine r ;*
3. *visiter l'enfant de droite (v_2)*



Itérateurs d'arbre

ordre symétrique (en-ordre)

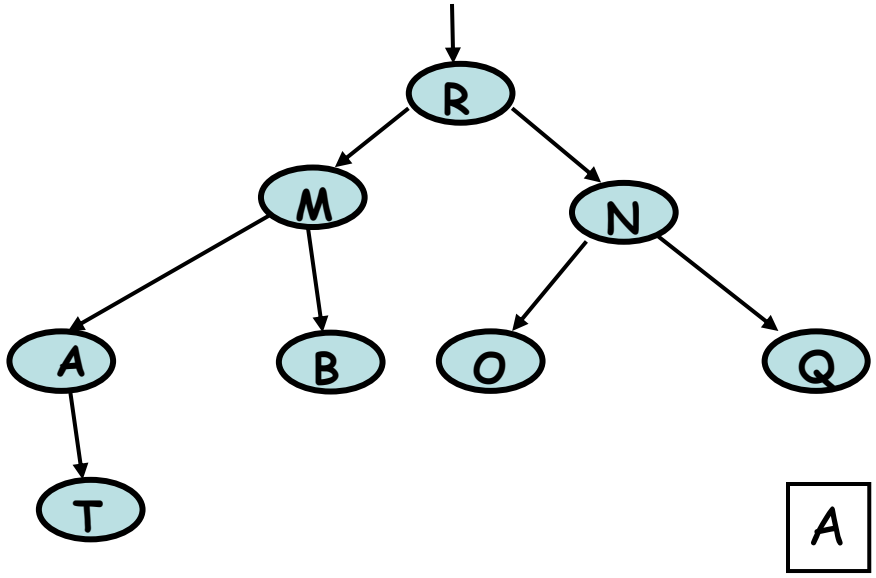
1. *visiter l'enfant de gauche (v_1)*
2. *visiter la racine r ;*
3. *visiter l'enfant de droite (v_2)*



Itérateurs d'arbre

ordre symétrique (en-ordre)

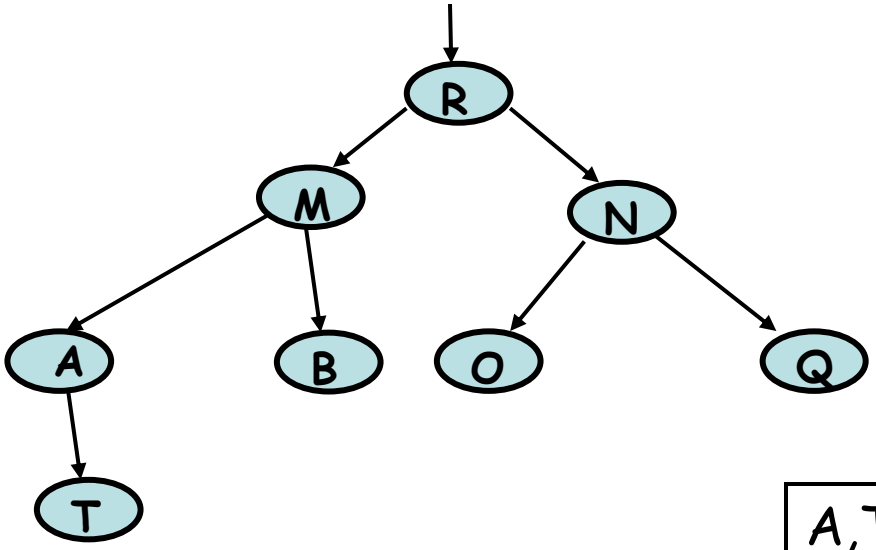
1. *visiter l'enfant de gauche (v_1)*
2. *visiter la racine r ;*
3. *visiter l'enfant de droite (v_2)*



Itérateurs d'arbre

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)*

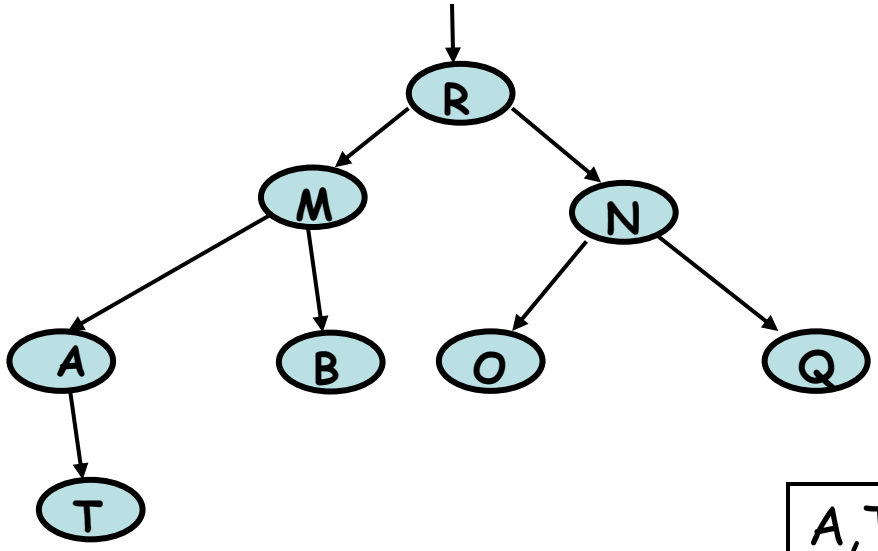


A,T

Itérateurs d'arbre

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)*

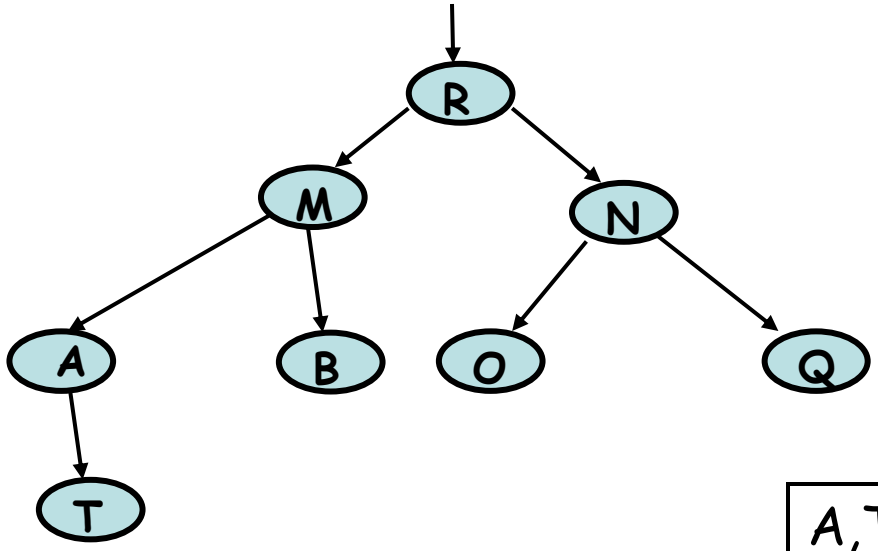


A,T,M

Itérateurs d'arbre

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)*

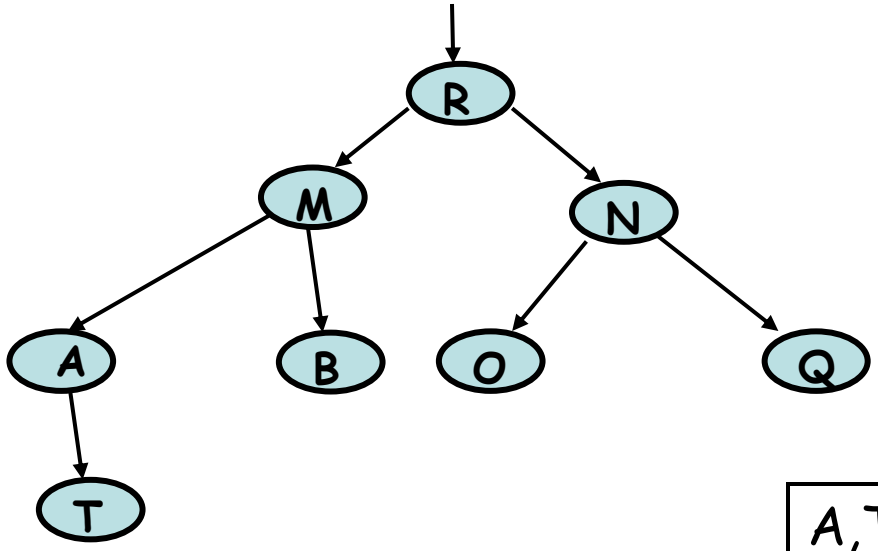


A,T,M,B

Itérateurs d'arbre

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)*

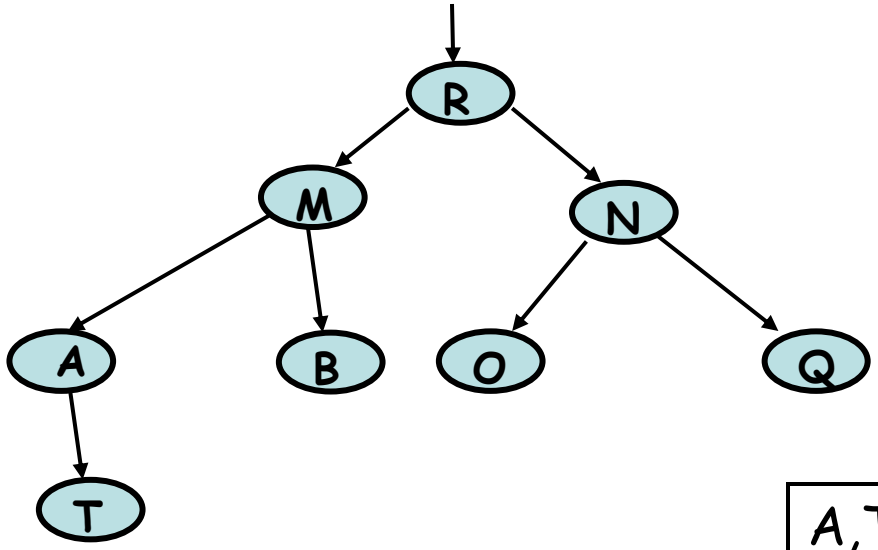


A,T,M,B,R

Itérateurs d'arbre

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)*

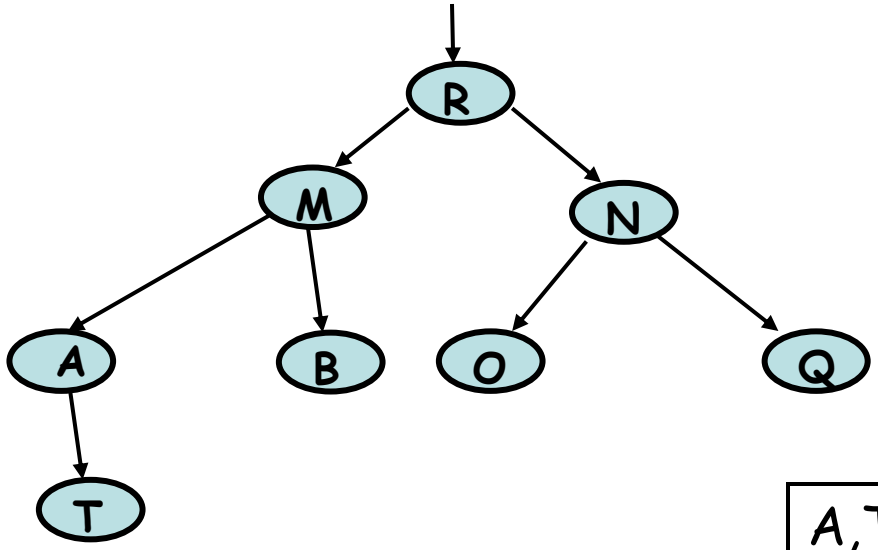


A,T,M,B,R,O

Itérateurs d'arbre

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)*

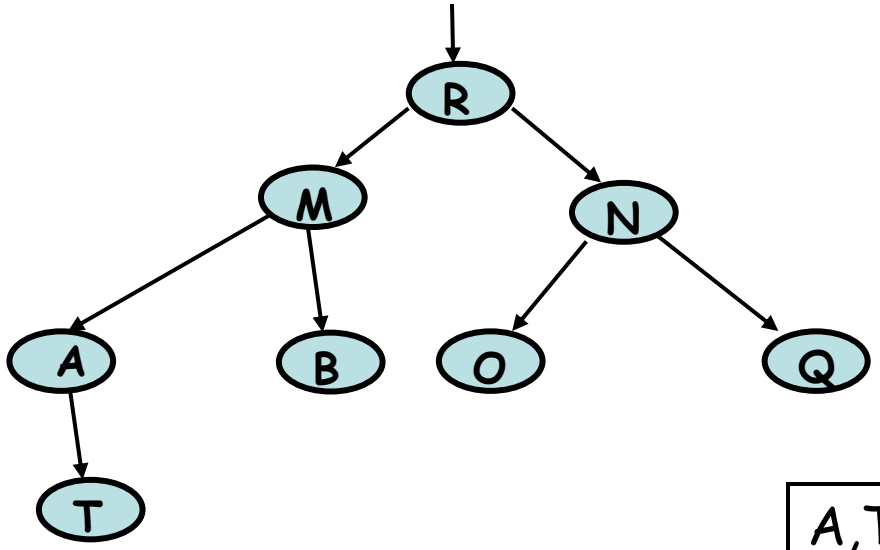


A,T,M,B,R,O,N

Itérateurs d'arbre

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)*

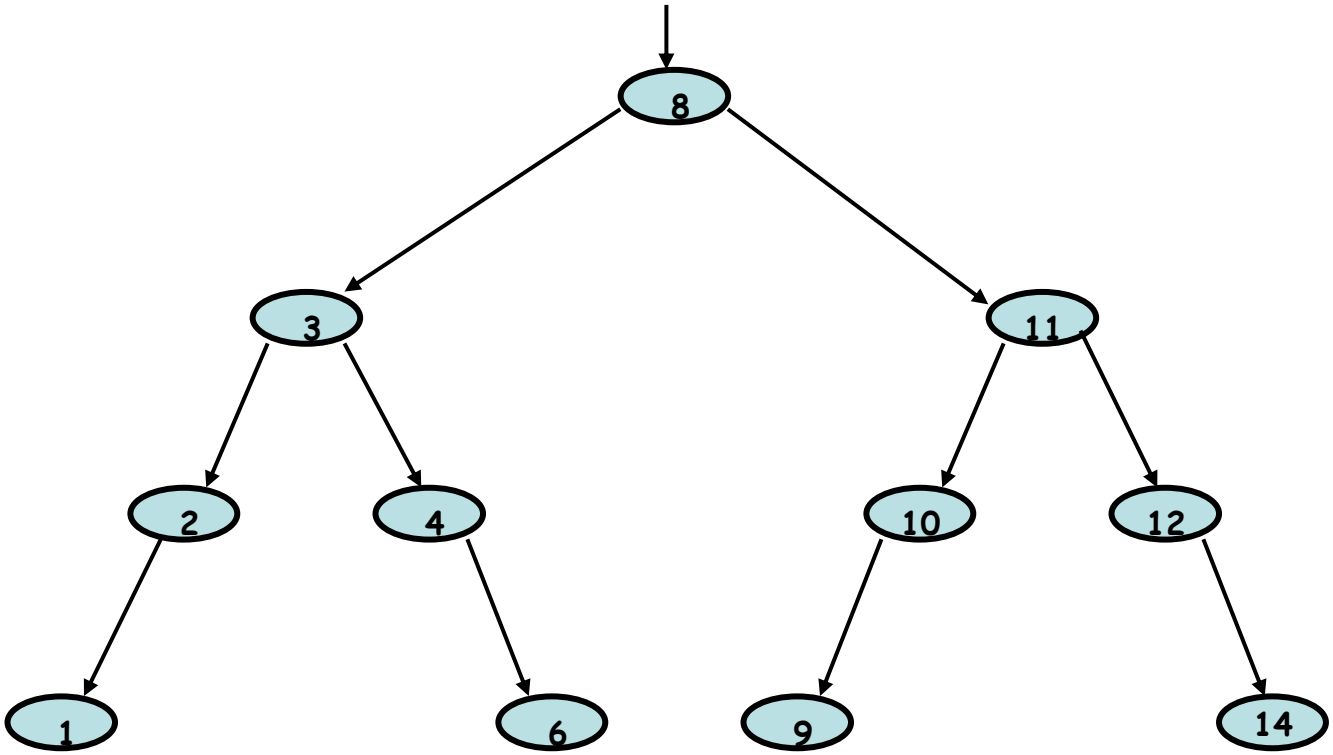


A,T,M,B,R,O,N,Q

Parcours en-ordre d'un arbre binaire de recherche

- Le parcours en-ordre d'un **arbre binaire de recherche** nous donne les éléments triés par **ordre croissant** des clés!

  - Cela constitue une des principales caractéristiques des arbres binaires de recherche

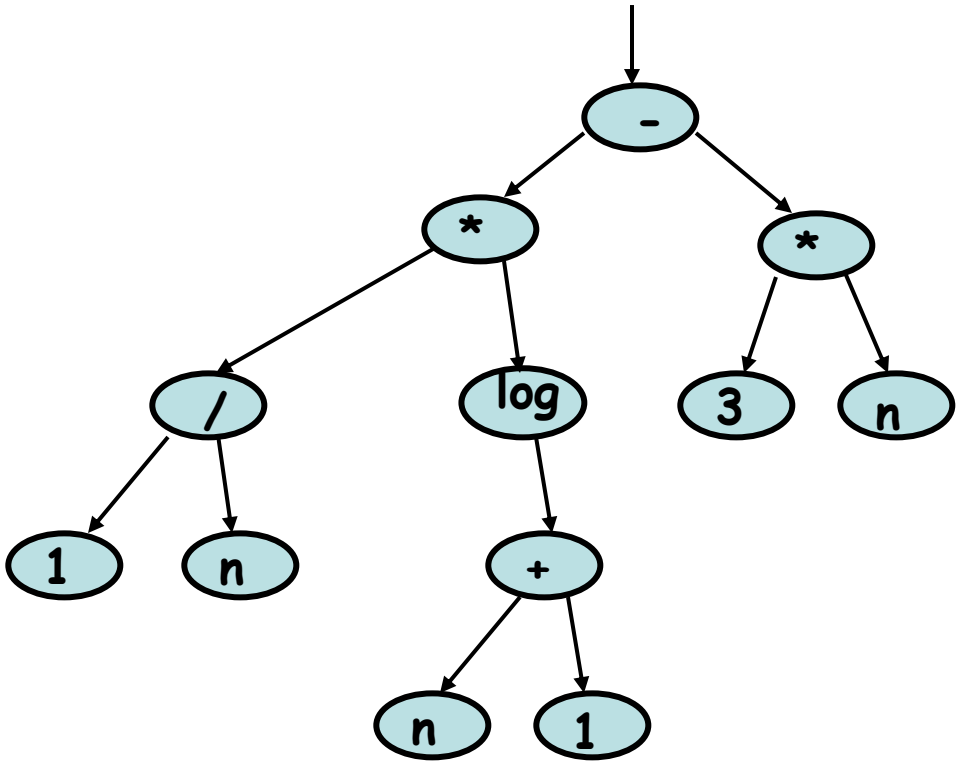


1,2,3,4,6,8,9,10,11,12,14

Application: arbres d'expression

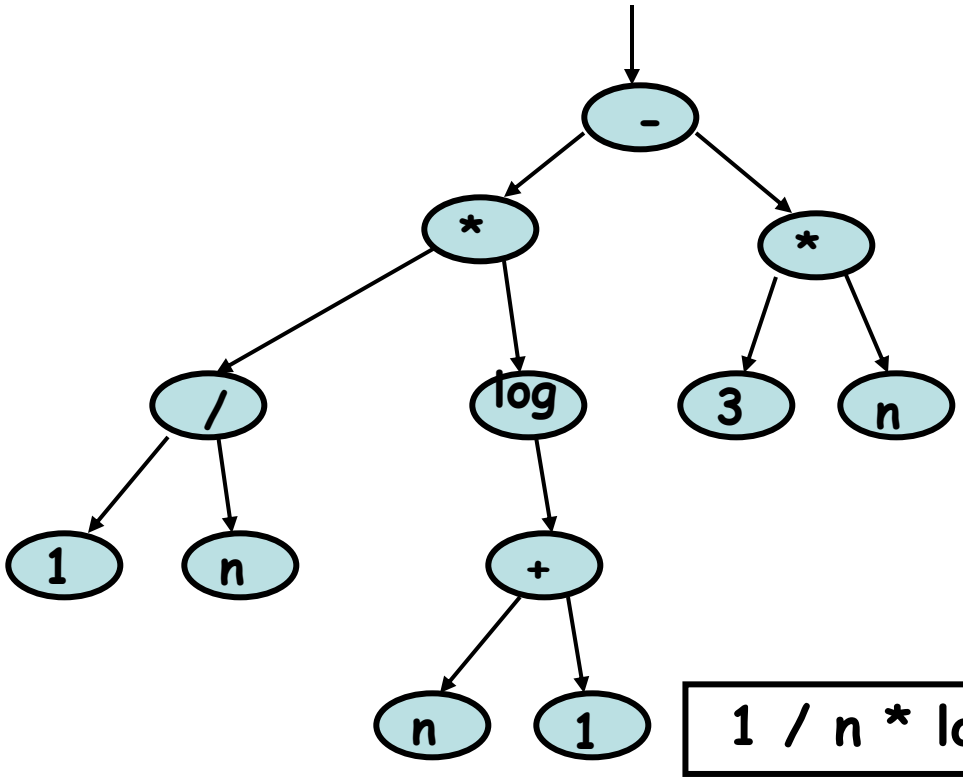
- Un **arbre d'expression** est un arbre dont les feuilles sont des **opérandes** (variables ou constantes) et dont les nœuds internes sont des **opérateurs**
  - Un opérateur binaire possède deux enfants
  - Un opérateur unaire possède un enfant (de droite)

- Voici l'arbre d'expression pour **((1/n) * (log(n+1))) - (3*n)**



Parcours en-ordre de l'arbre d'expression

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



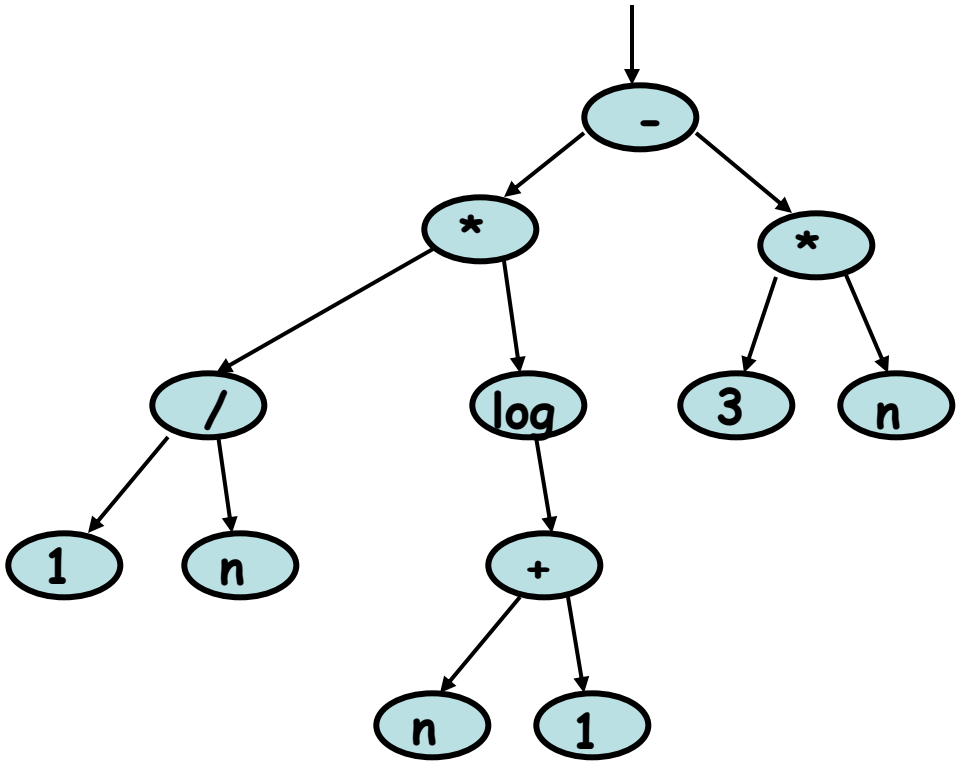
1 / n * log n + 1 - 3 * n

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

Parcours post-ordre de l'arbre d'expression

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

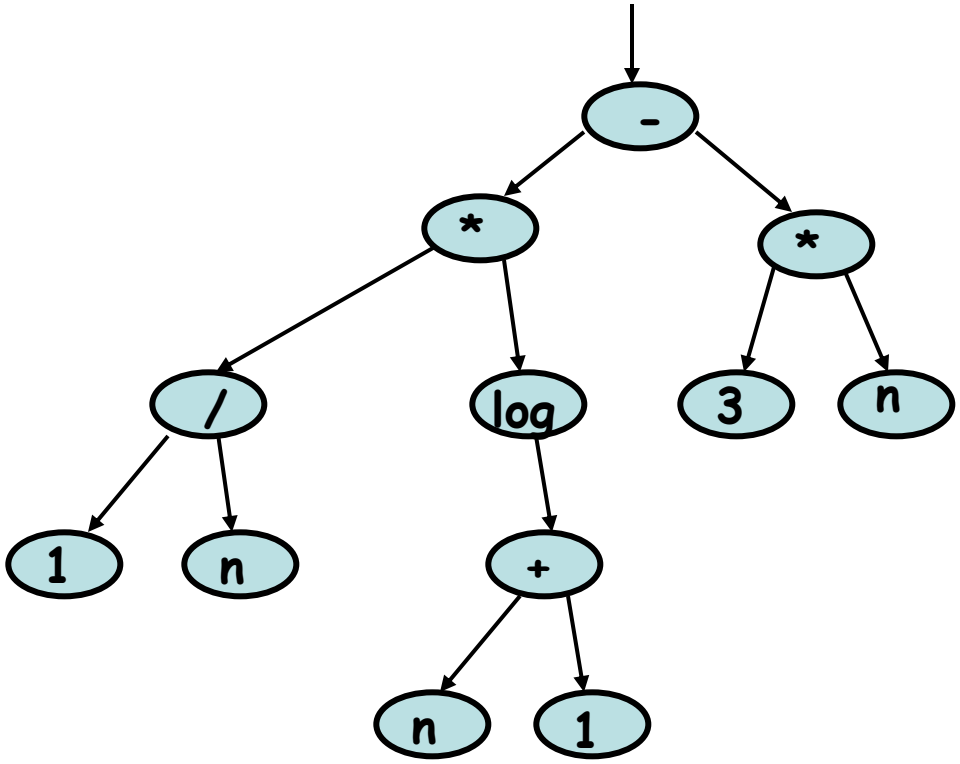
Utilisée pour la première fois par le mathématicien Polonais Jan Lukasiewicz



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

Parcours pré-ordre de l'arbre d'expression

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



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

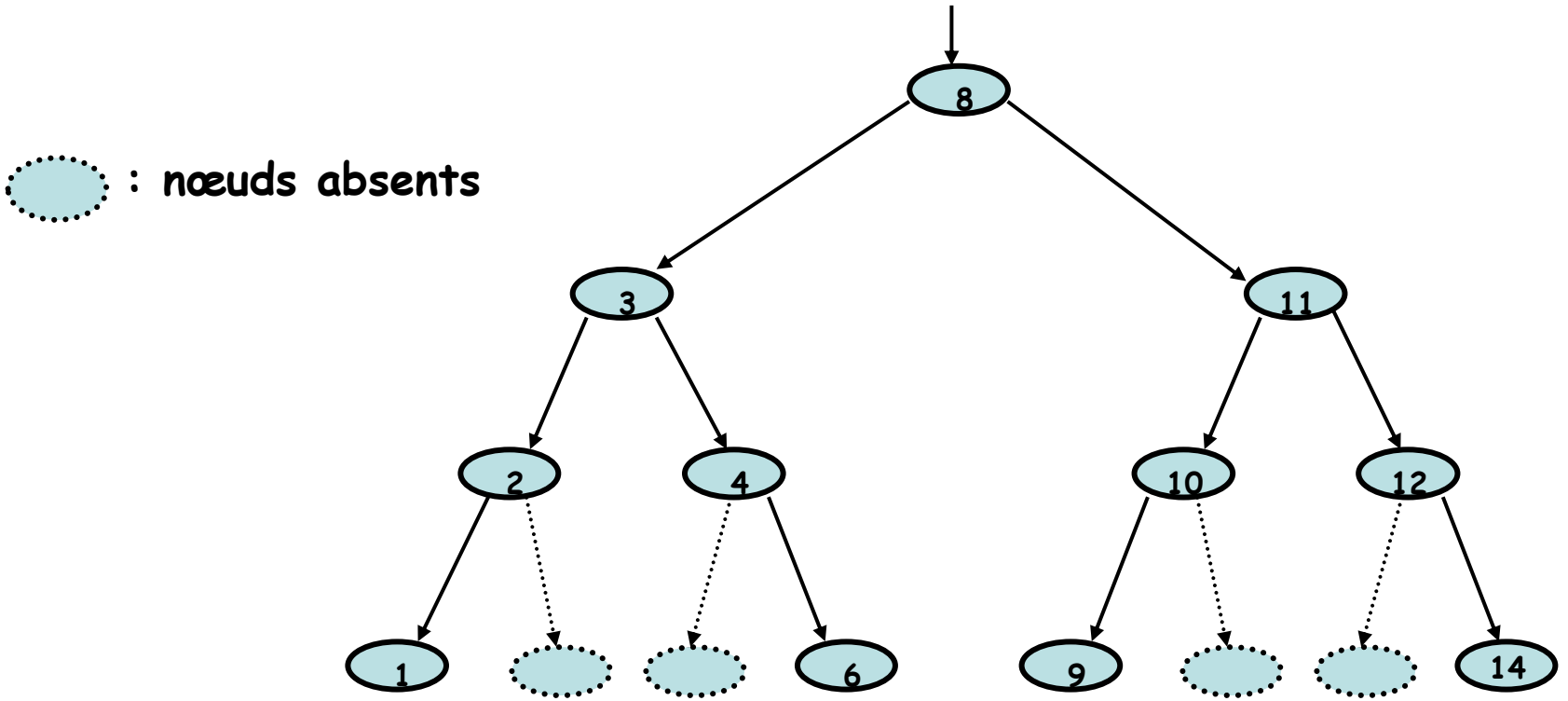
Implémentations des arbres

- Il existe deux grandes méthodes d'implémentation.

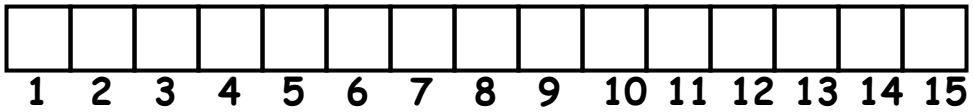
- Implémentation dans un tableau
  - Les nœuds sont insérés dans un tableau
  - Aucune utilisation de pointeurs
  - la position des enfants est obtenu par une opération arithmétique très simple
  - Utilisé pour les monceaux/tas (objet de ce chapitre)

- Implémentation 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



Comment faire?



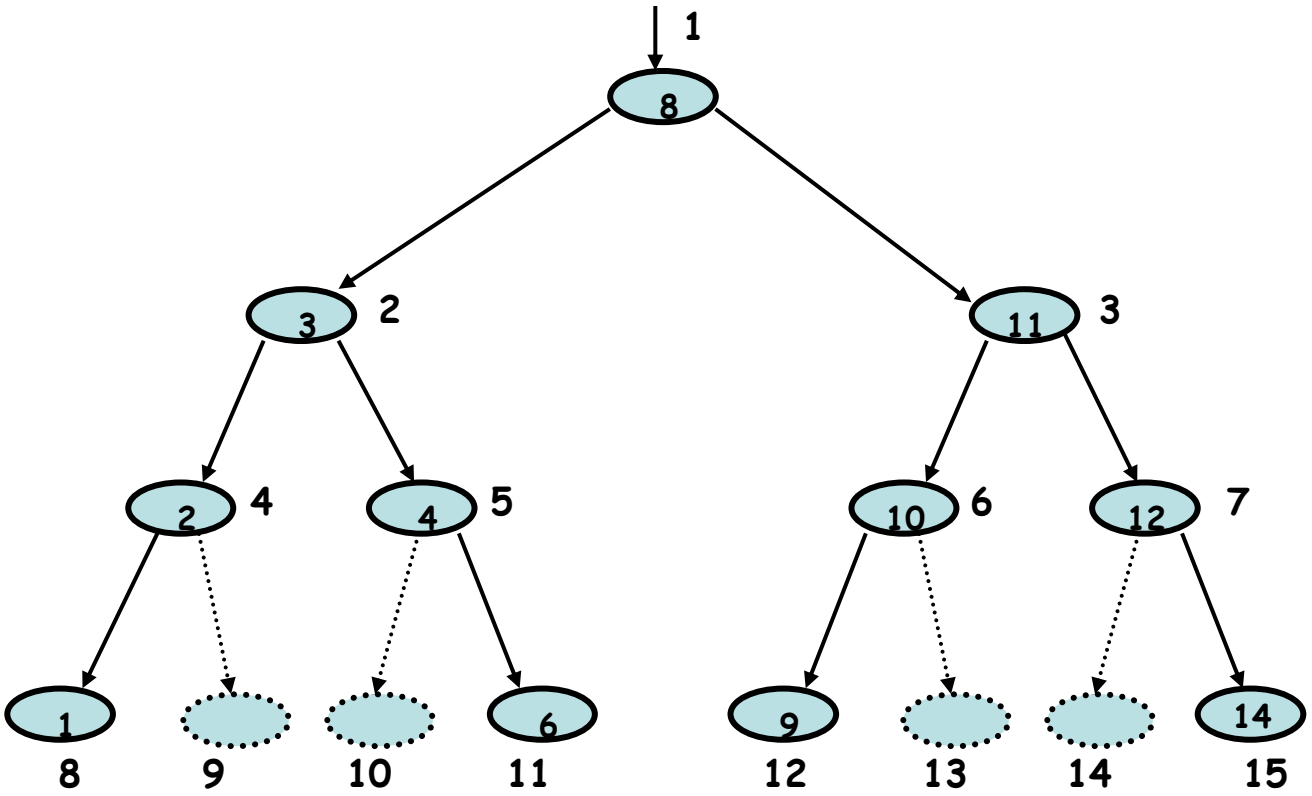
Le niveau h peut contenir jusqu'à 2^h nœuds (pour h = 0, 1, 2 …)

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

Implémentation en tableau des arbres binaires

Voici donc l'emplacement des éléments dans le tableau:

(les nœuds absents ne sont donc pas stockés dans l'espace mémoire réservé)



8	3	11	2	4	10	12	1			6	9			14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Implémentation en tableau des arbres binaires

Pour faciliter la description des algorithmes nous utilisons la convention que le premier indice du tableau est 1

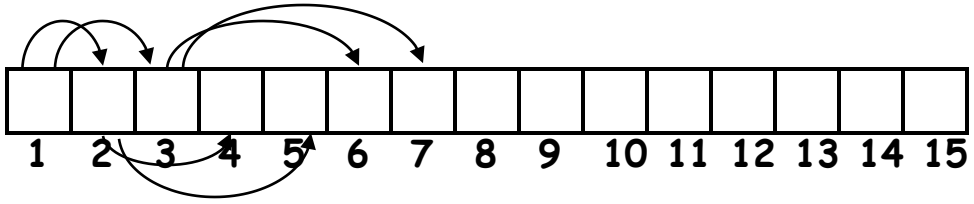
(il faudra alors corriger cela pour l'implémentation des algorithmes en C++)

L'indice du 1er nœud du niveau h est donné par (voir page suivante) :

$$2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} + 1 = 2^h$$

- Le ième nœud du niveau h a donc pour indice $2^h + (i-1) \equiv j$
- 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 $2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} = 2^h - 1$?

Cette série est un cas particulier de la série géométrique. Soit:

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

Nous avons alors

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

Alors

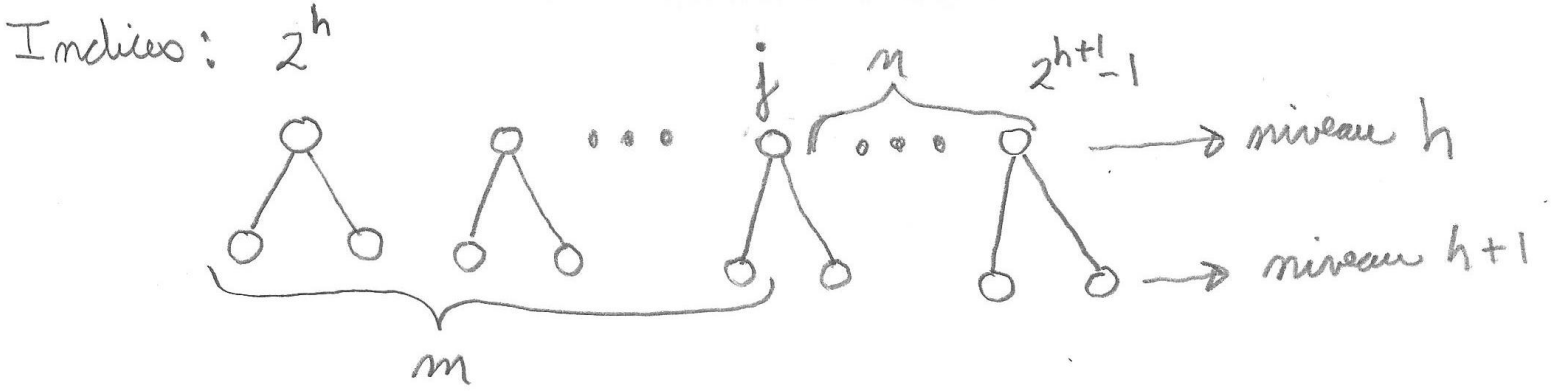
$$S(r-1) = r^{n+1} - 1 \,.$$

Donc

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

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

Autre preuve pour EnfantGauche(j) = 2j

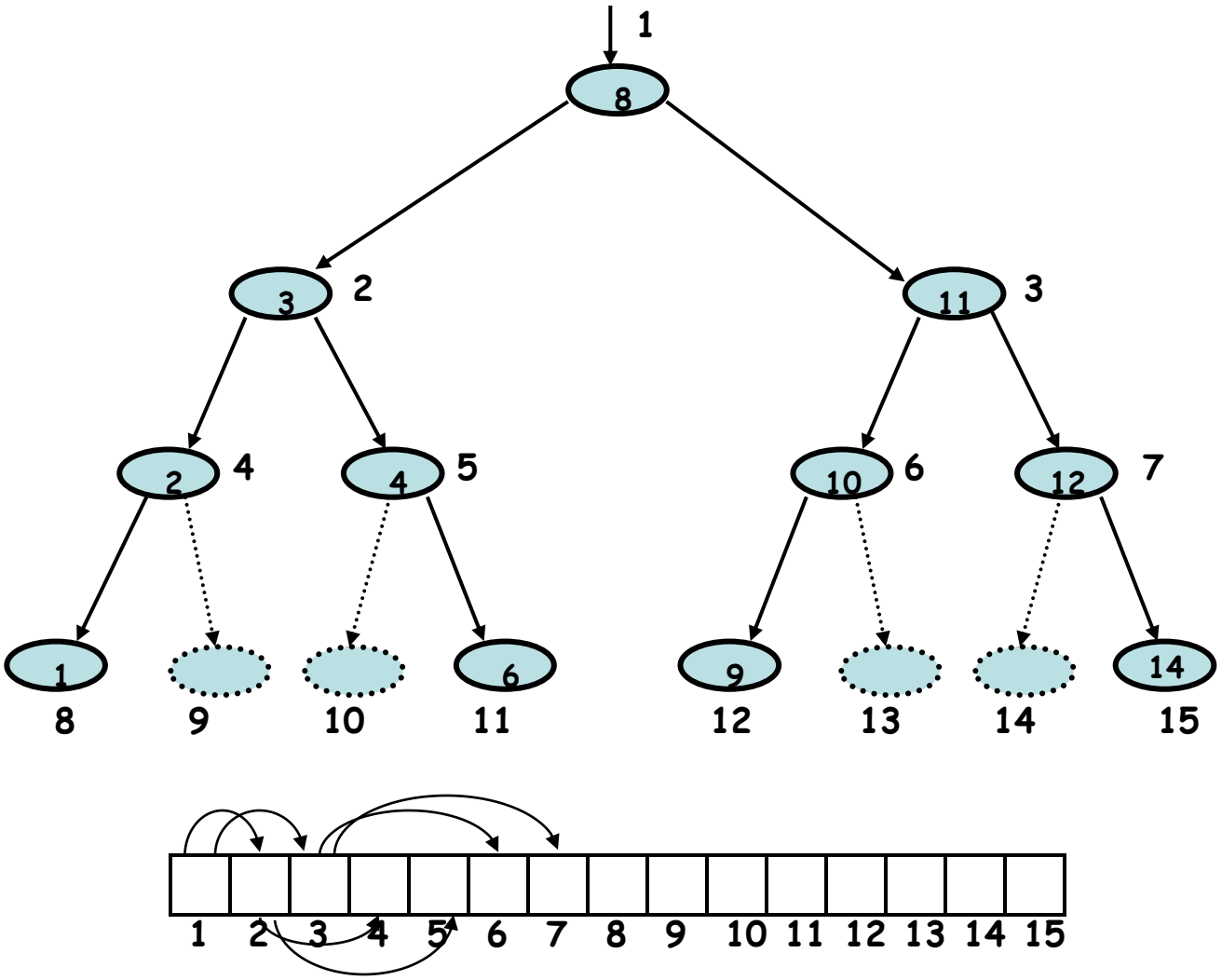


- L'indice du premier nœud au niveau h est: 2^h
- L'indice du dernier nœud au niveau h est: $2^{h+1} - 1$

- EnfantGauche(j) = j + n + m
- $n = (2^{h+1} - 1) - j$
- $m = 2(j - 2^h + 1) - 1$

- Donc, EnfantGauche(j) = $(2^{h+1} - 1) + 2(j - 2^h + 1) - 1 = 2j$

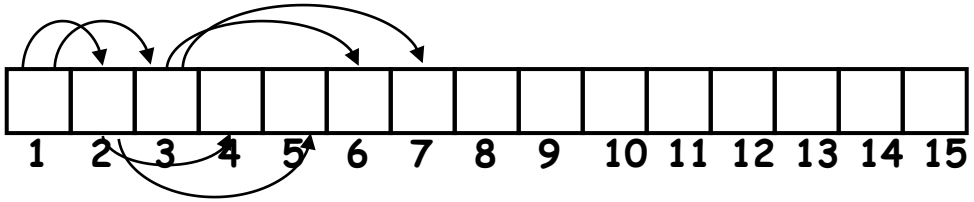
Positionnement des enfants dans le tableau



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 $\lfloor i/2 \rfloor$**



Pause math: fonctions floor() et ceiling()

- Pour la suite de ce chapitre, nous utiliserons ces fonctions.

- **floor(x) = $\lfloor x \rfloor$ = le plus grand entier ≤ x**
  - Exemples:
    - $\lfloor 3.72 \rfloor$ = 3
    - $\lfloor 3 \rfloor$ = 3
- **ceiling(x) = $\lceil x \rceil$ = le plus petit entier ≥ x**
  - Exemples:
    - $\lceil 3.44 \rceil$ = 4
    - $\lceil 3 \rceil$ = 3

- 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 ≥ 0 et k ≥ 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

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

Attention: en C++ les indices débutent à la position 0 . Donc, si les enfants du parent i étaient en positions 2i et 2i+1. La position en C++ de ce parent sera en j = i-1 et ceux de ses enfants seront j' = (2i)-1 et j'' = 2i. Puisque i = j+1, ces enfants sont positionnés en j' = 2j+1 et j'' = 2j+2.

```
template <typename T>
void Arbre<T>::affiche() const { _affiche(0);}

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);
        }
}
```

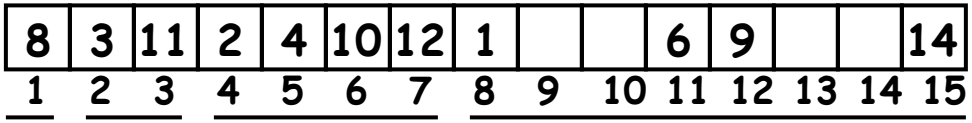
Implantation en tableau

Avantages :

- simplicité pour visiter les enfants
- aucun espace utilisé pour stocker des pointeurs
- l'espace pour insérer un nœud est déjà disponible

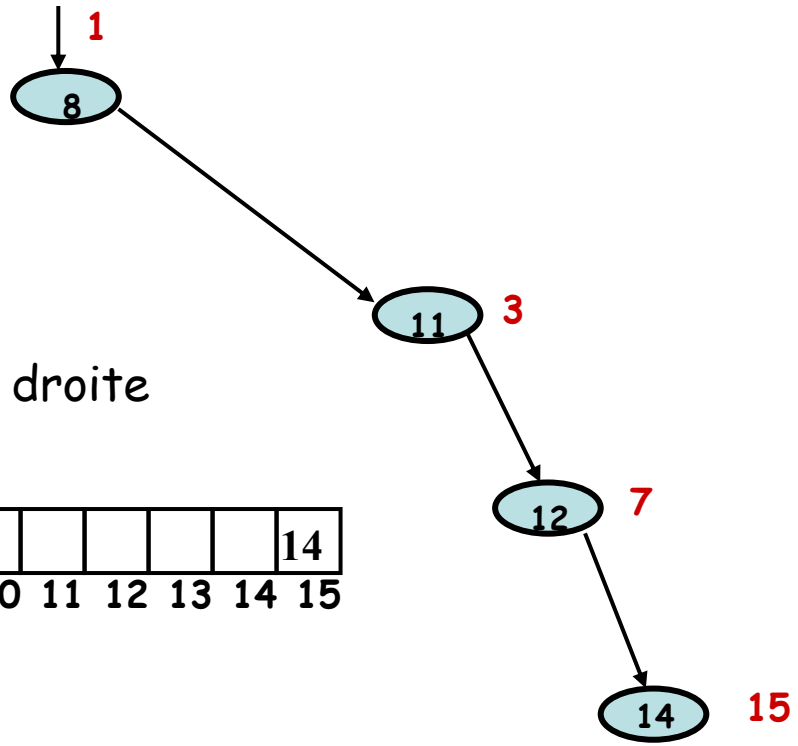
Désavantages :

- 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:

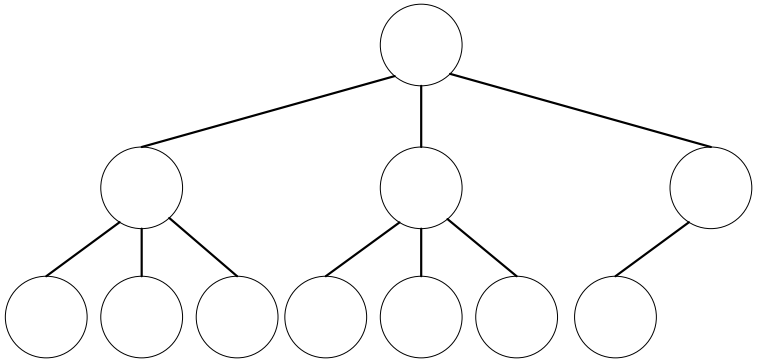


Arbre **dégénéré** vers la droite

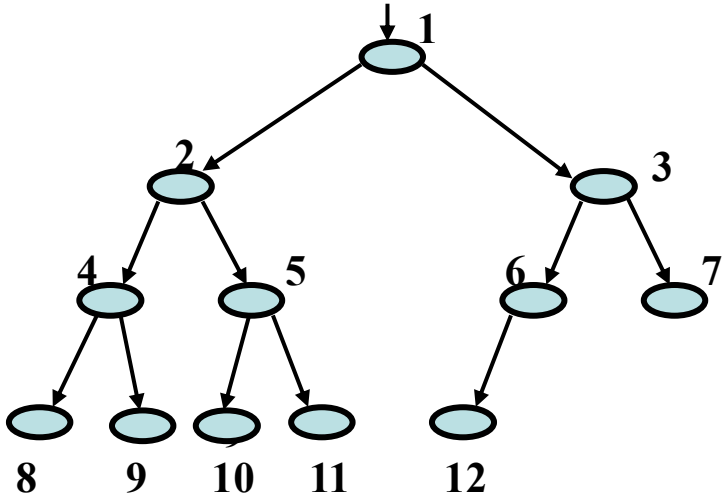
8		11				12								14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

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

Définition du tas/monceau (« heap »)

- Un tas est 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)
    - la valeur de la racine est donc la valeur maximale du tas

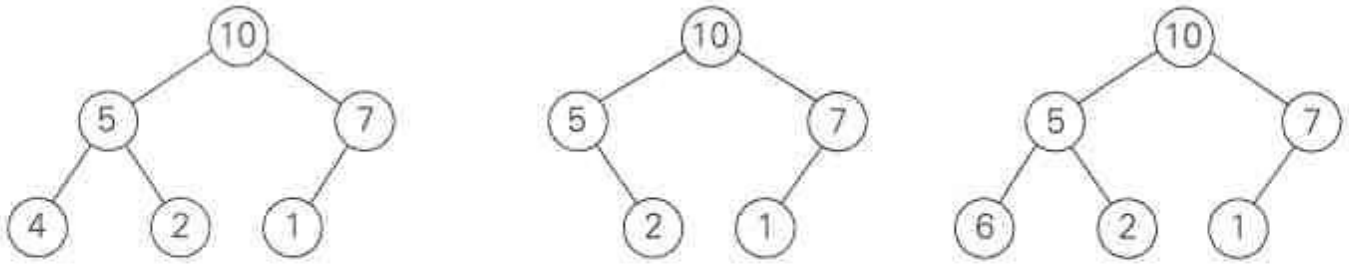


FIGURE 6.9 Illustration of the definition of "heap": only the leftmost tree is a heap.

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

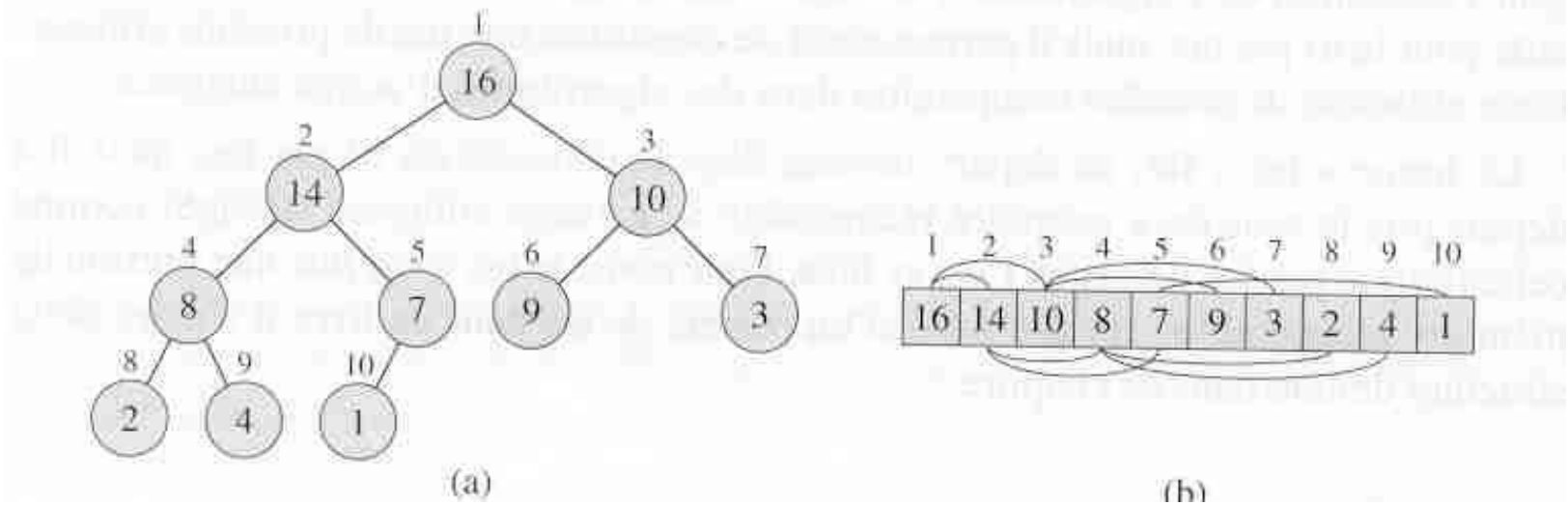
Utilité des tas

- Le **tas** est la 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)
- Une 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

- Le tas est une structure intermédiaire utilisée par l'algorithme du **tri par tas** (« heapsort »)
  - Un tableau est d'abord transformé en un tas
  - Les éléments sont ensuite repositionnés (rapidement) en ordre croissant. Le tableau est alors 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
  - Il n'y a pas de trous dans le tableau: implémentation efficace.



[source: Introduction à l'algorithmique, Cormen, Leiserson, Rivest et Stein, Dunod 2002, ISBN13 978-2-10-003922-7]

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:
  - $n = 2^0 + 2^1 + \ldots 2^{h-1} + l$ avec: $1 \leq l \leq 2^h$
  - Alors: $\mathbf{n = 2^h - 1 + l}$
- Alors:
  - $l \geq 1 \Rightarrow n \geq 2^h \Rightarrow \mathbf{h \leq \log_2(n)}$

- De plus:
  - $l \leq 2^h \Rightarrow n \leq 2^h - 1 + 2^h = 2^{h+1} - 1 \Rightarrow n < 2^{h+1} \Rightarrow \mathbf{\log_2(n) < h + 1}$
- 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:**

$$\mathbf{h = \lfloor \log_2(n) \rfloor}$$

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 = $\lceil n/2 \rceil$ et p = $\lfloor n/2 \rfloor$.**
- 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] ≥ H[2i] et H[i] ≥ H[2i+1] pour i = 1…$\lfloor n/2 \rfloor$ (les nœuds parents).
- Nous devons donc 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

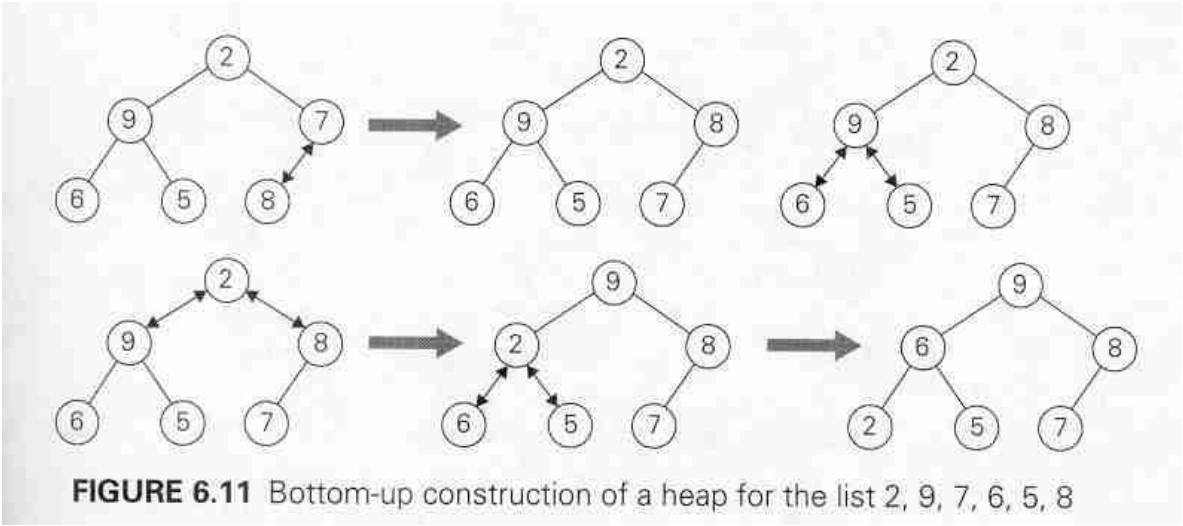
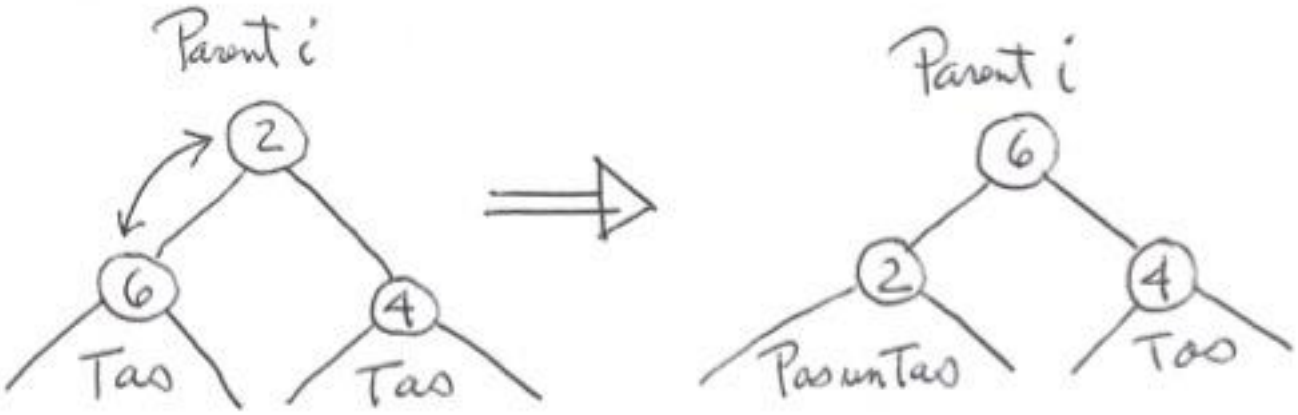


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8

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

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.
  - Ce ne serait pas le cas si on débutait avec le 1er parent (la racine)



- Lors de la percolation vers le bas, on impose à un enfant une valeur de clé plus faible que celle qu'il avait précédemment.
- Mais si on impose à la racine d'un tas une valeur plus petite que celle d'avant, la percolation vers le bas de cette valeur fera en sorte, qu'à la fin, nous obtiendrons un tas.
- Cela implique que le sous arbre de racine i sera un tas après avoir percolé vers le bas la valeur à son enfant maximal.

Construction du tas, bas vers le haut

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

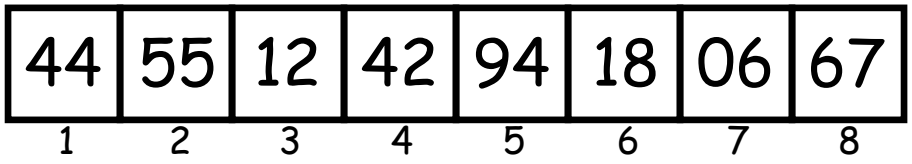
Percolation vers le bas dans un tas

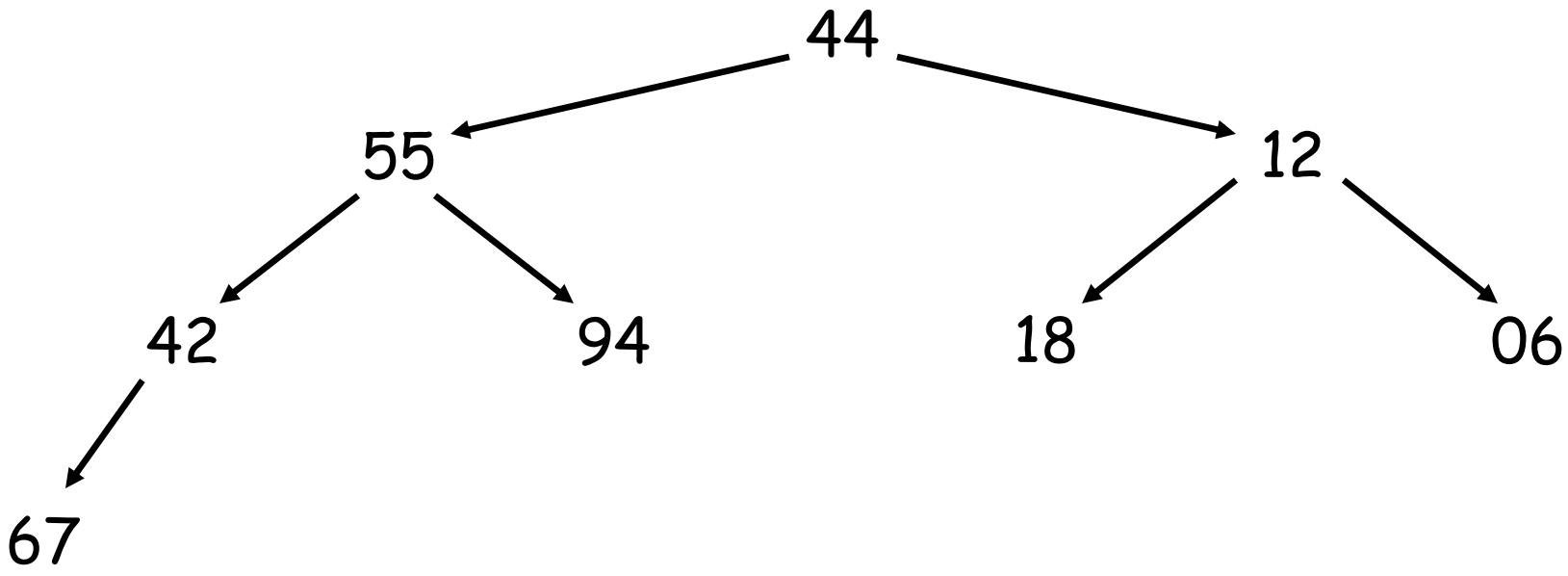
```
template <typename Comparable>
void percDown( vector<Comparable> & a, size_t i, size_t n )
{
    size_t child;

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

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

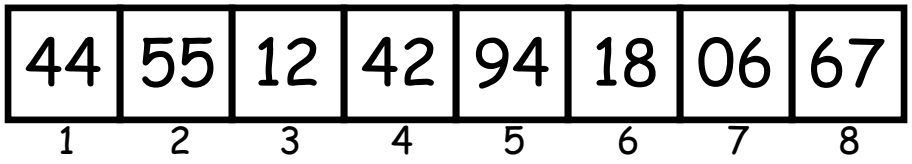
Construction d'un tas bas vers le haut

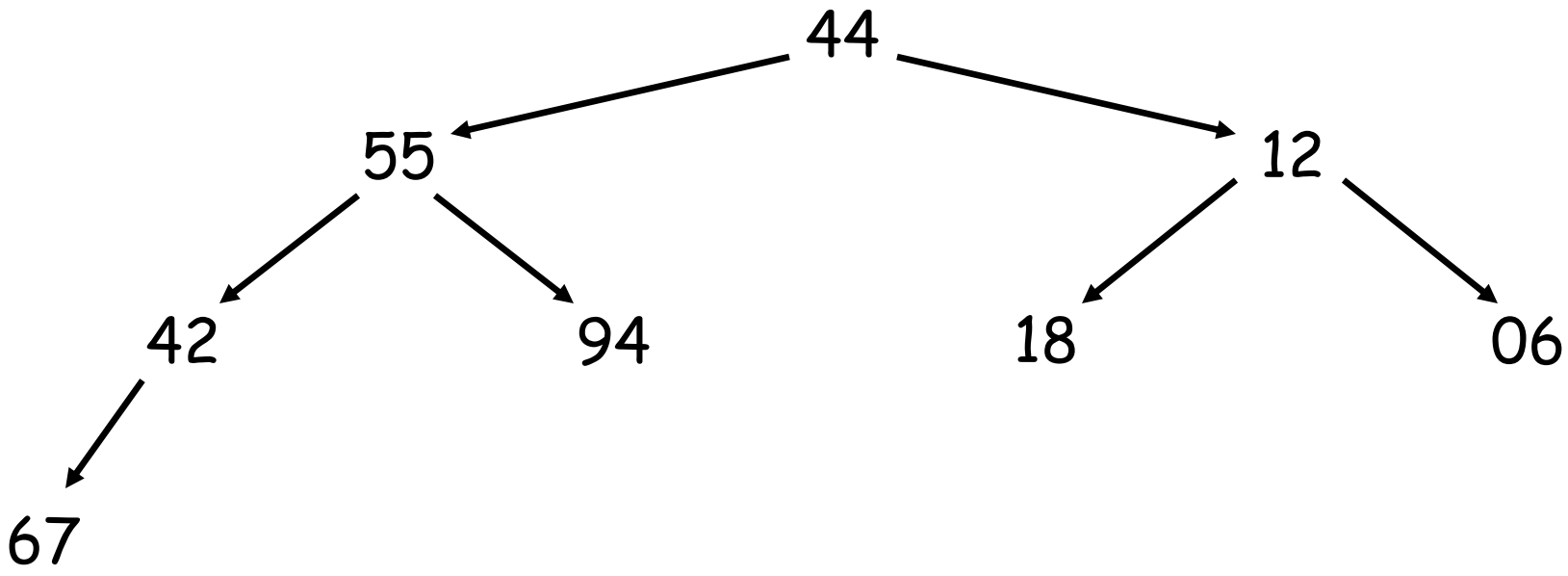




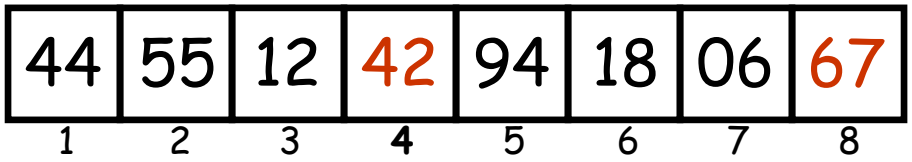
Construction d'un tas bas vers le haut

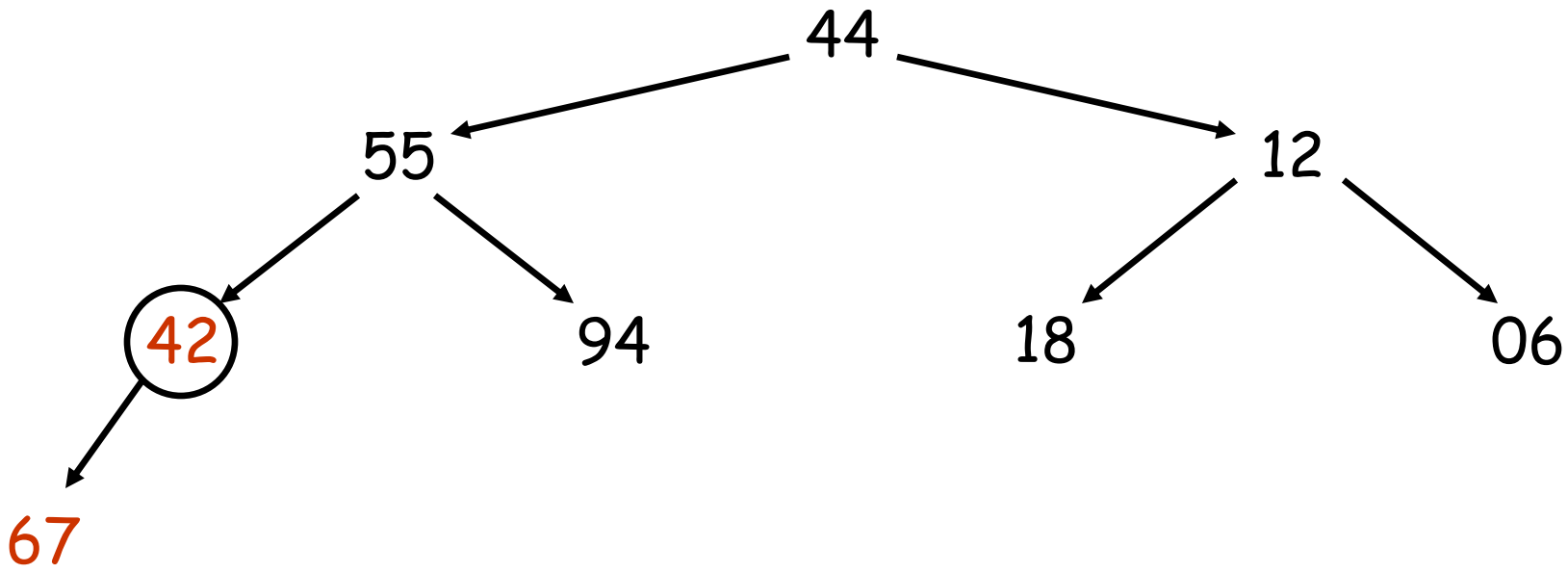
Création du monceau tas-max



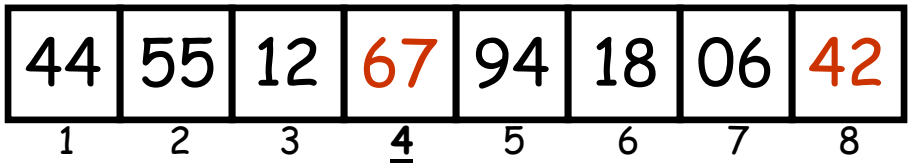


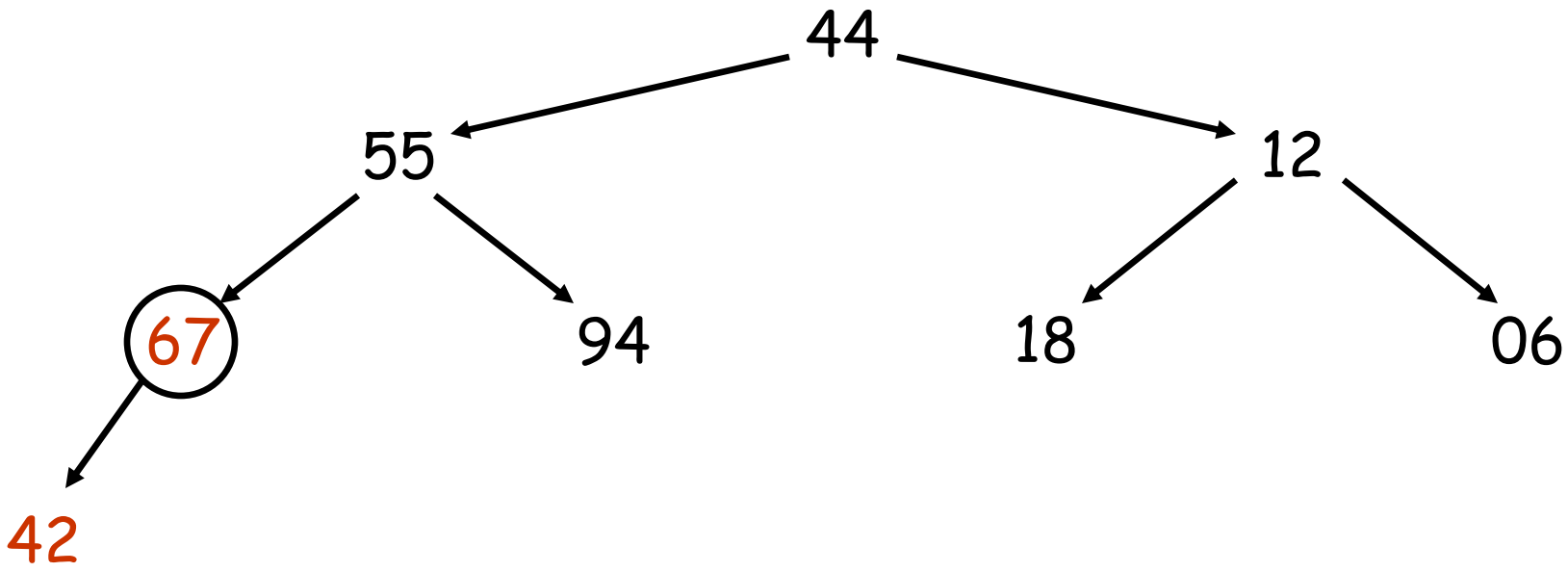
Construction d'un tas bas vers le haut



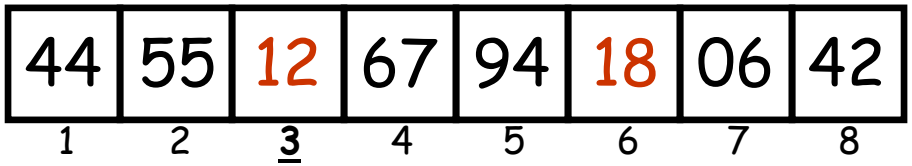


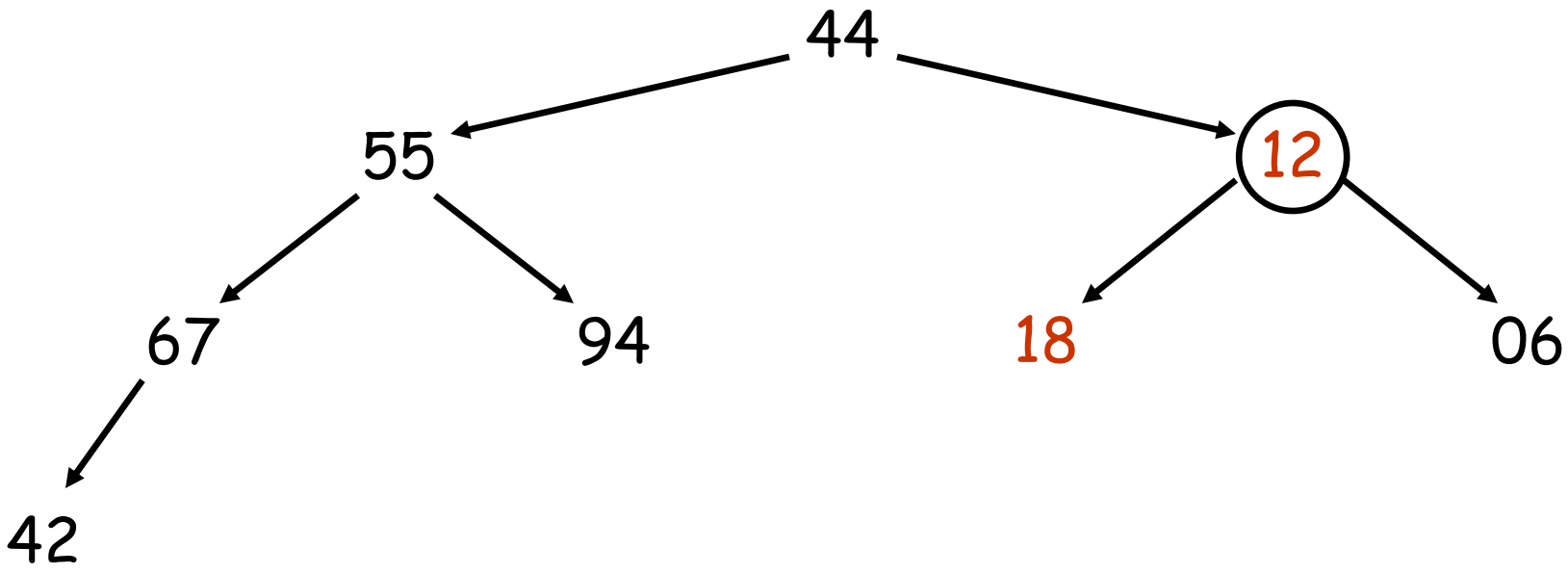
Construction d'un tas bas vers le haut



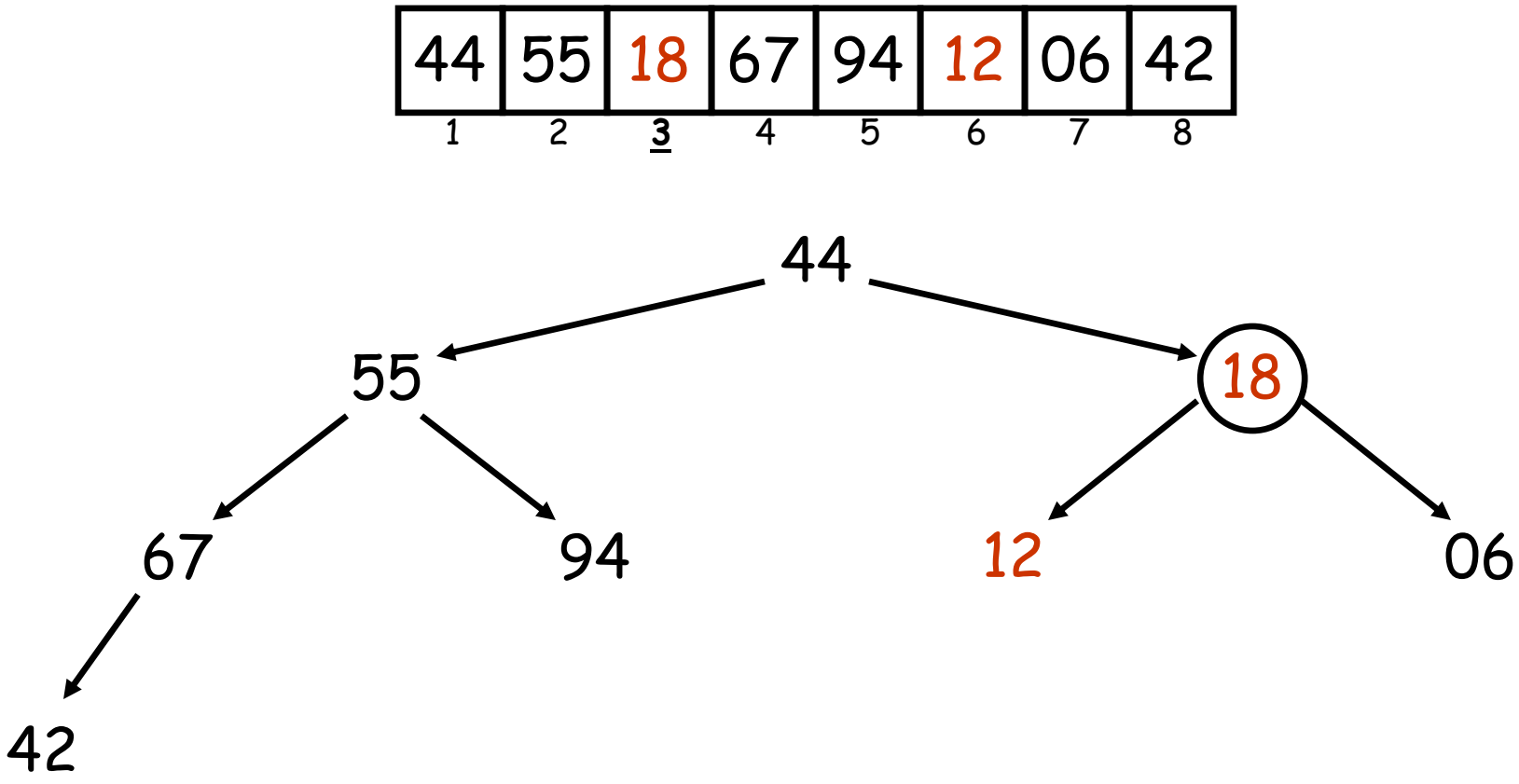


Construction d'un tas bas vers le haut

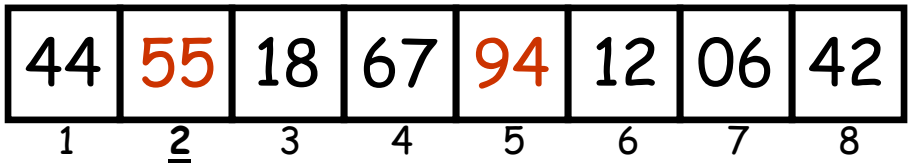


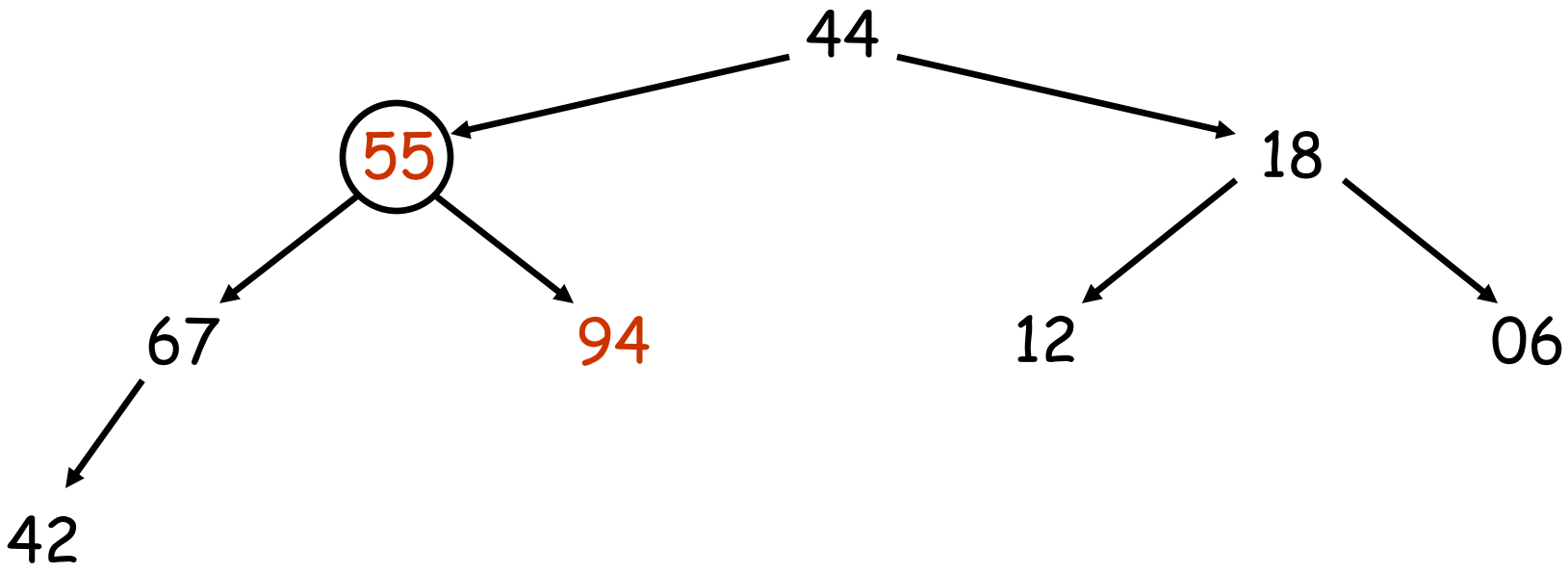


Construction d'un tas bas vers le haut

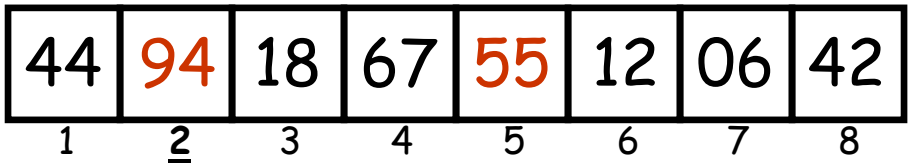


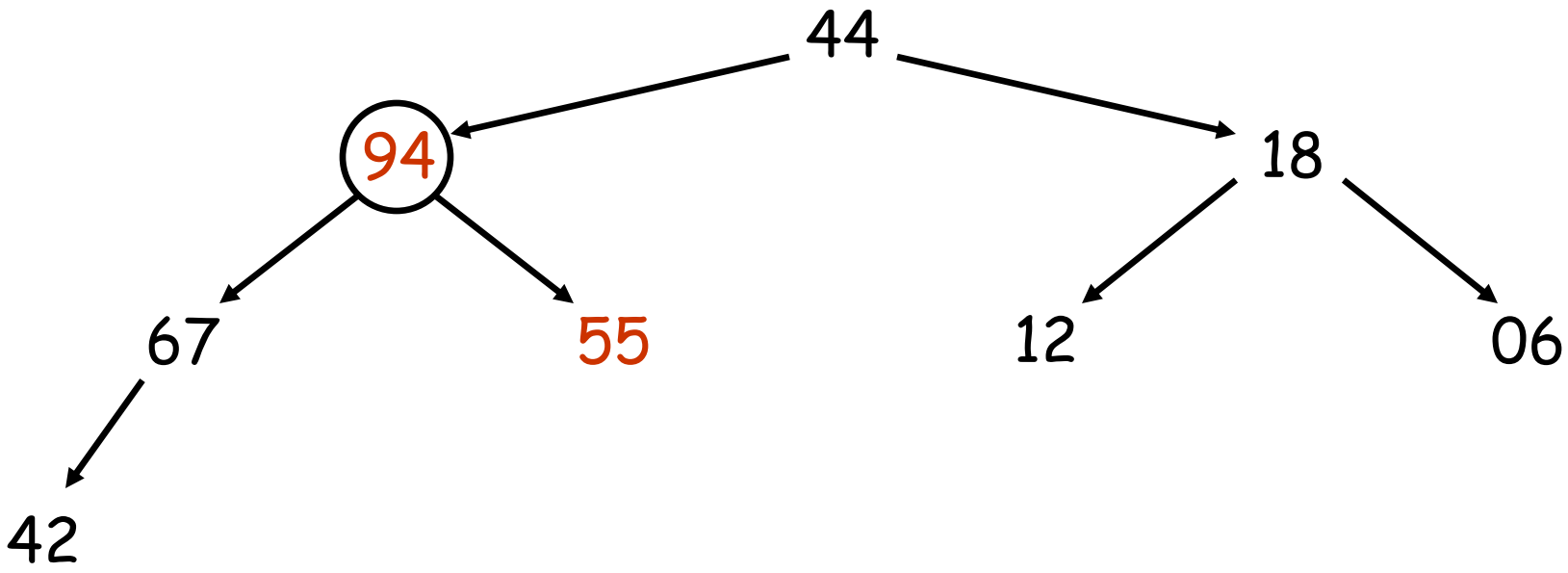
Construction d'un tas bas vers le haut



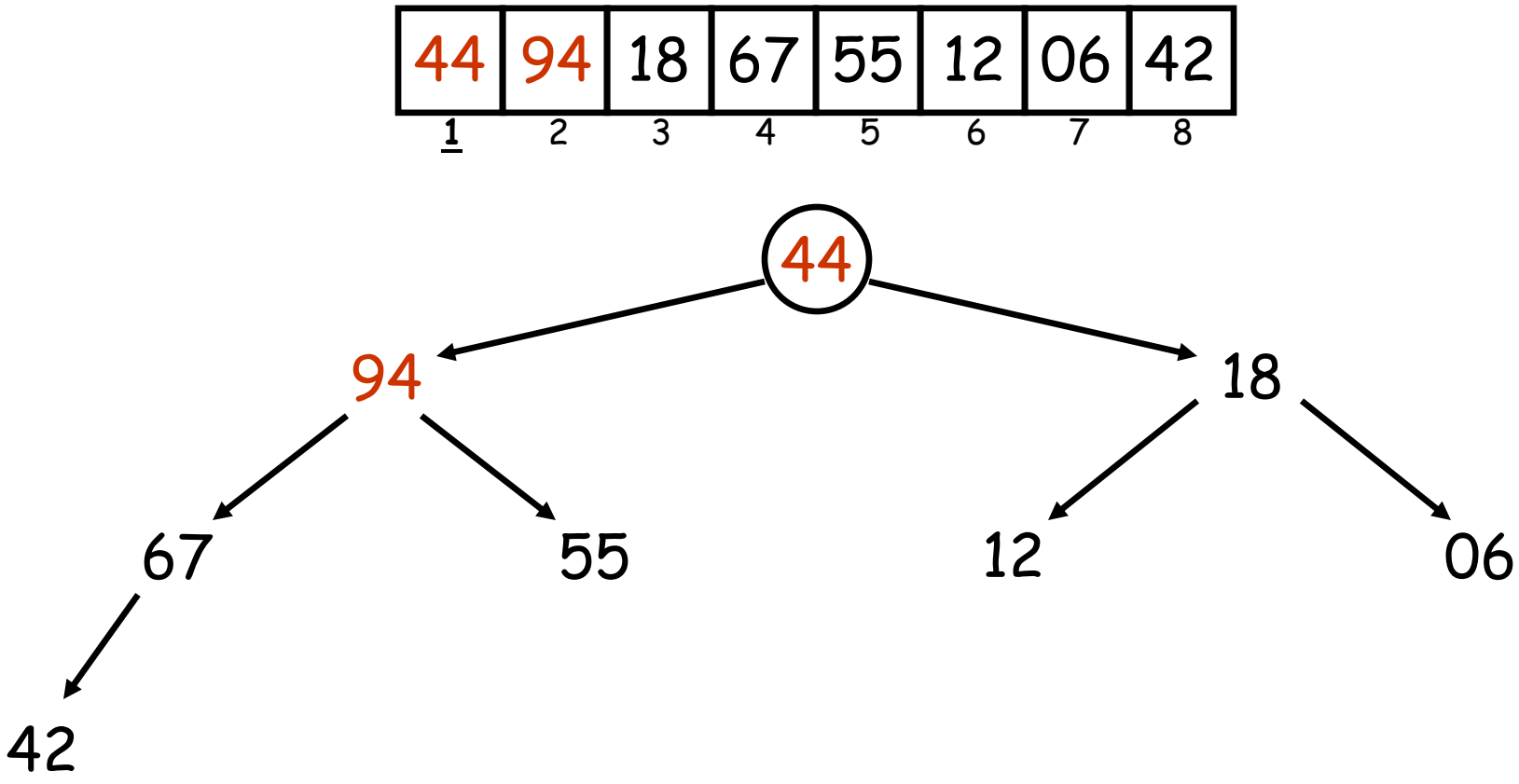


Construction d'un tas bas vers le haut

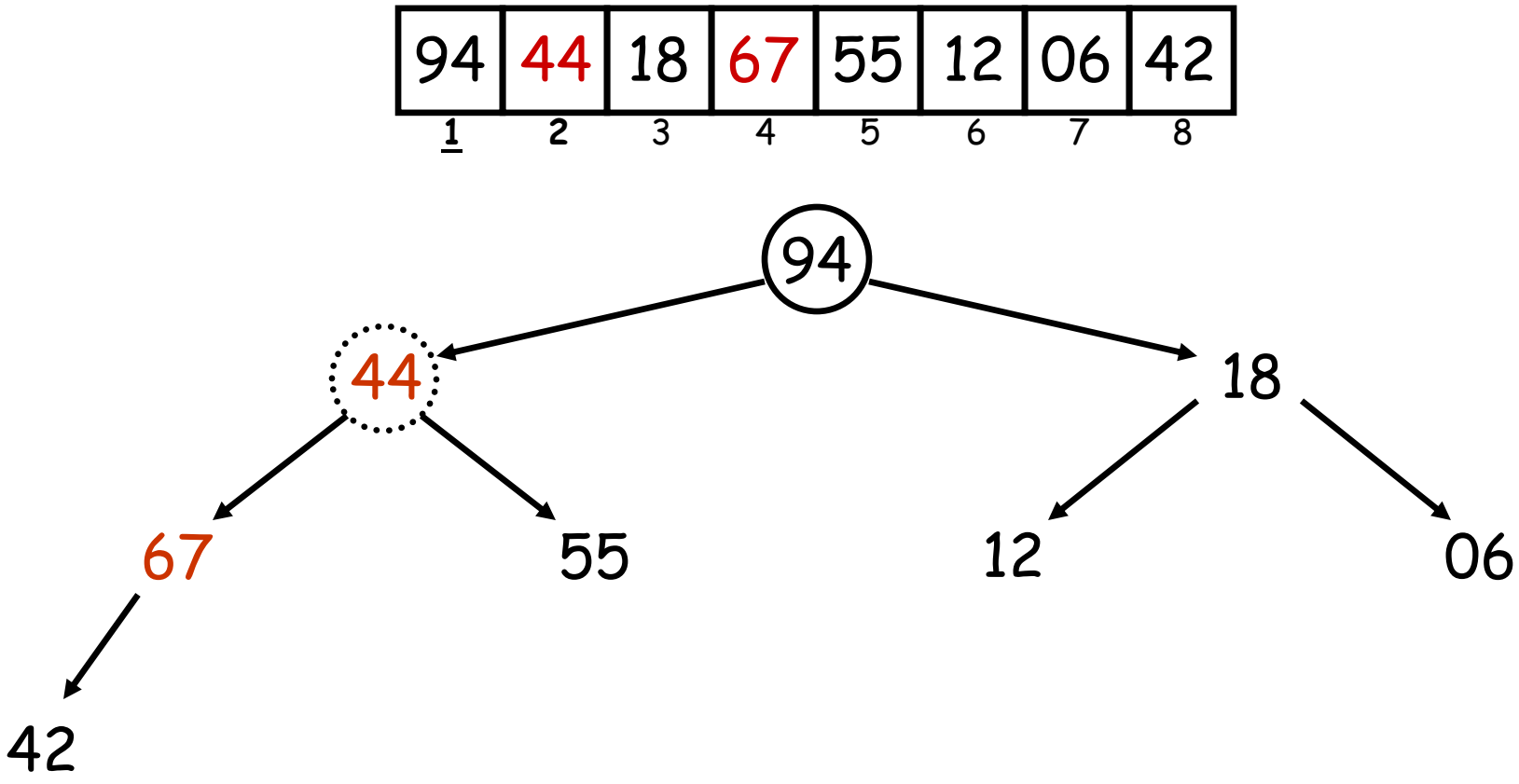




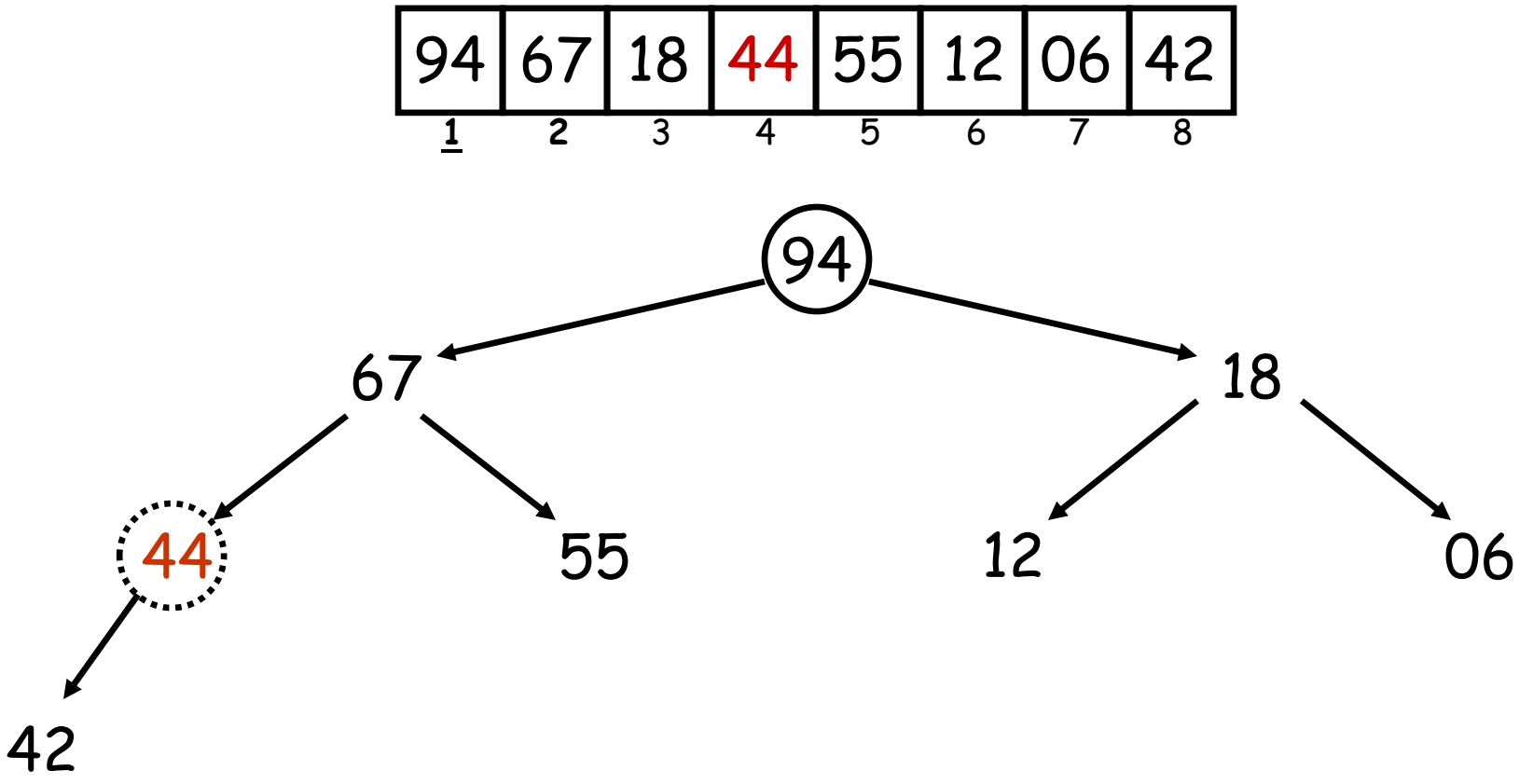
Construction d'un tas bas vers le haut



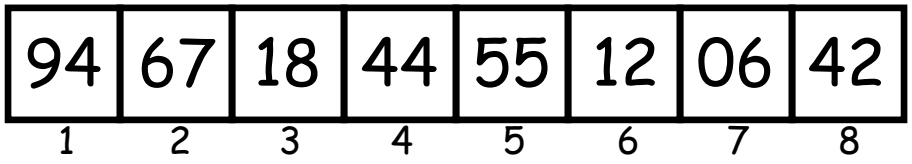
Construction d'un tas bas vers le haut

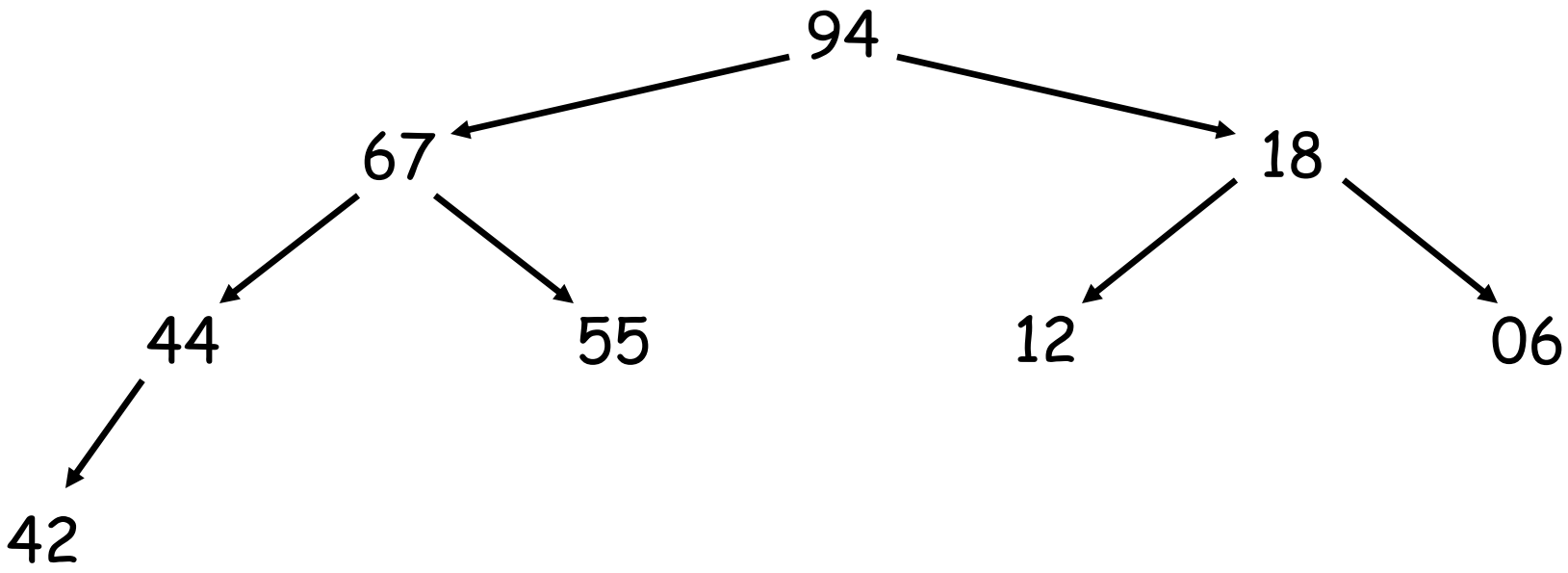


Construction d'un tas bas vers le haut



Construction d'un tas bas vers le haut





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 et satisfait H[i] ≥ H[2i] et H[i] ≥ H[2i+1] pour i = 1 à $\lfloor n/2 \rfloor$
  - il ne sera jamais nécessaire de vérifier la propriété du tas pour les enfants
  - On a alors $C_{best}(n) = \lfloor n/2 \rfloor$. Donc **$C_{best}$(n) est en Θ(n)**.

- **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, il faut 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, ..., h – 2. (h-1 = niveau max des parents)
  - Et $p(k) \leq 2^k$ pour k = h-1.

Analyse de HeapButtomUp (suite)

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

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

- Or il est bien connu que (voir un «dictionnaire de séries»)

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

- Nous avons donc

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

- Alors

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

- 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 $\Theta(n)$.
- **Donc le temps d'exécution de heapBottomUp est en $\Theta(n)$ dans tous les cas.**

Insertion d'un élément dans un tas

- Pour la mise en œuvre d'une file de priorité, nous devons pouvoir insérer rapidement un nouvel item dans un tas
- Pour cela, nous insérons d'abord le nouvel élément K dans une nouvelle feuille que nous positionnons juste après la dernière feuille du tas (ou, plus simplement, nous faisons H[n+1] = K)
- Nous comparons K avec son parent H[$\lfloor (n+1)/2 \rfloor$] :
  - Si K ≤ H[$\lfloor (n+1)/2 \rfloor$] ne rien faire car H[1..n+1] est un tas
  - Sinon on interchange K avec H[$\lfloor (n+1)/2 \rfloor$]
  - Nous recommençons jusqu'à ce que K est ≤ à la valeur de son parent (ou jusqu'à ce que K devienne la racine)
- **Le nombre maximal de comparaisons requises est donc de $\lfloor \log_2(n+1) \rfloor$**

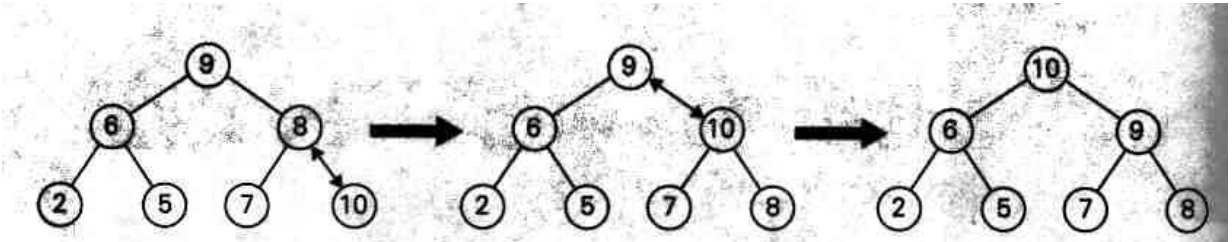


FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

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

Insertion dans un tas

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

- Rappel : lorsque les indices débutent à 1: parent(i) = $\lfloor i/2 \rfloor$
- La position en C++ du nœud i = i-1 ≡ j
- La position en C++ du parent(i) = $\lfloor i/2 \rfloor$ - 1
- Donc, parent(j) = $\lfloor (j+1)/2 \rfloor$ - 1 = $\left\lfloor \frac{j+1}{2} - 1 \right\rfloor$ = $\lfloor (j-1)/2 \rfloor$

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 car c'est un élément dont la priorité est la plus élevée
- Pour cela, **nous interchangeons l'élément H[n] avec H[1] et nous reconstruisons le tas H[1..n-1] en percolant H[1] vers le bas avec percDown(1)** :
  - nous comparons la valeur de H[1] avec celle de ses enfants et l'interchangeons avec le max de ses enfants si c'est nécessaire
  - Nous continuons 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])

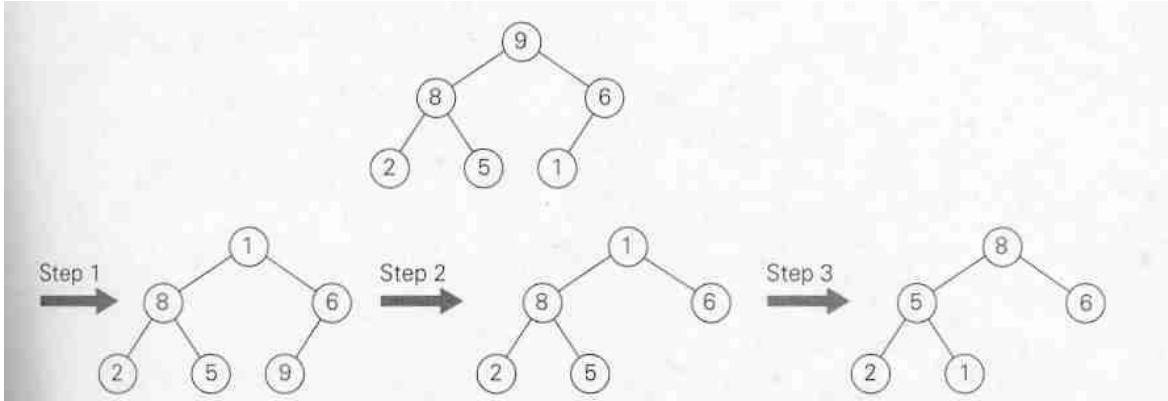


FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key, after which the smaller tree is "heapified" by exchanging the new key at its root with the larger key at its children until the parental dominance requirement is satisfied.

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

Le tri par tas

- À partir d'un tableau non trié, nous construisons d'abord un tas H[1..n] à l'aide de l'algorithme heapBottumUp en un temps en Θ(n) (dans tous les cas)
  - La racine est donc un élément de valeur la plus élevée
- Nous interchangeons H[1] avec H[n] et reconstruisons le tas H[1..n-1] à l'aide de percDown(1) en Θ(log n) comparaisons en pire cas et O(1) comparaisons en meilleur cas (réalisé lorsque tous les éléments on même valeur)
- Nous recommençons 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 alors trié en ordre croissant
- En utilisant la formule de Stirling pour n!, on trouve que le nombre de comparaison requises en pire cas et en meilleur cas sont alors donnés par:

$$C_{worst}(n) \in \Theta\big(\log(n-1) + \log(n-2) + \ldots + \log(1)\big) = \Theta(\log((n-1)!)) = \Theta(n\log(n))$$

$$C_{best}(n) \in \Theta\big(1 + 1 + \ldots + 1\big) = \Theta(n)$$

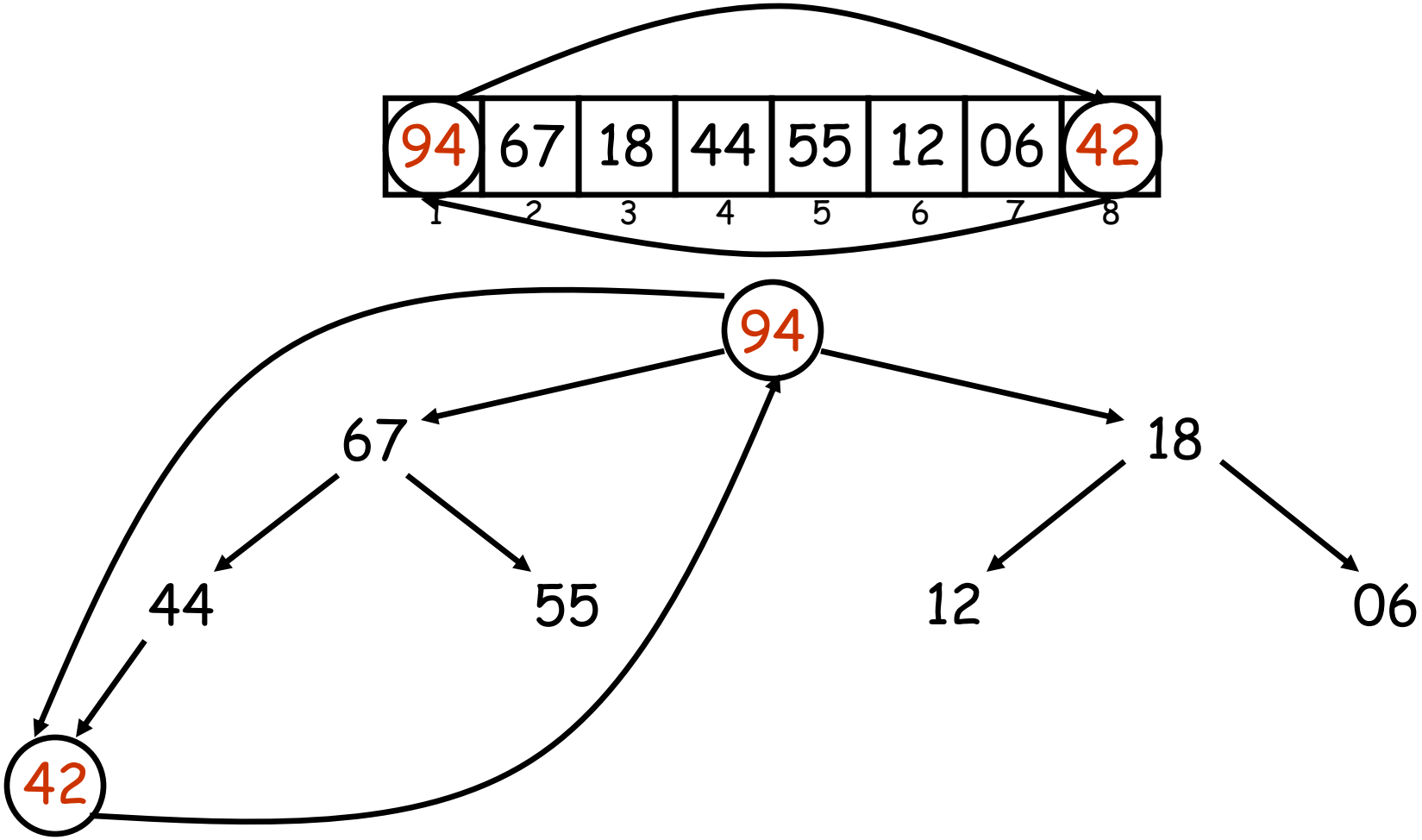
- **Le tri par tas s'exécute donc en Θ(n log n) en pire cas et en Θ(n) en meilleur cas**
  - C'est donc un algorithme de tri performant
  - Empiriquement, nous observons, qu'**en moyenne**, le **tri par tas** est légèrement plus rapide que le **tri fusion** (que nous verrons plus tard)

Tri par tas (« Heapsort »)

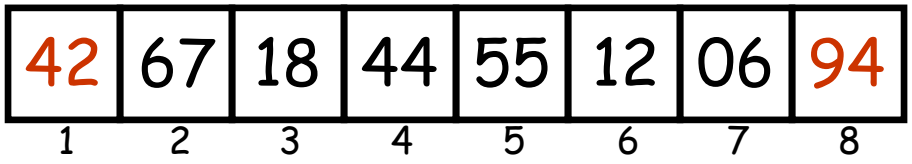
```
template <typename Comparable>
void heapsort( vector<Comparable> & a )
{
    if (a.size() <= 1) return;
    heapBottumUp(a); // construction du monceau
    for( size_t j = a.size( ) - 1; j > 0; j-- )
    {
        swap( a[ 0 ], a[ j ] ); // positionner la racine dans le tableau trié
        percDown(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

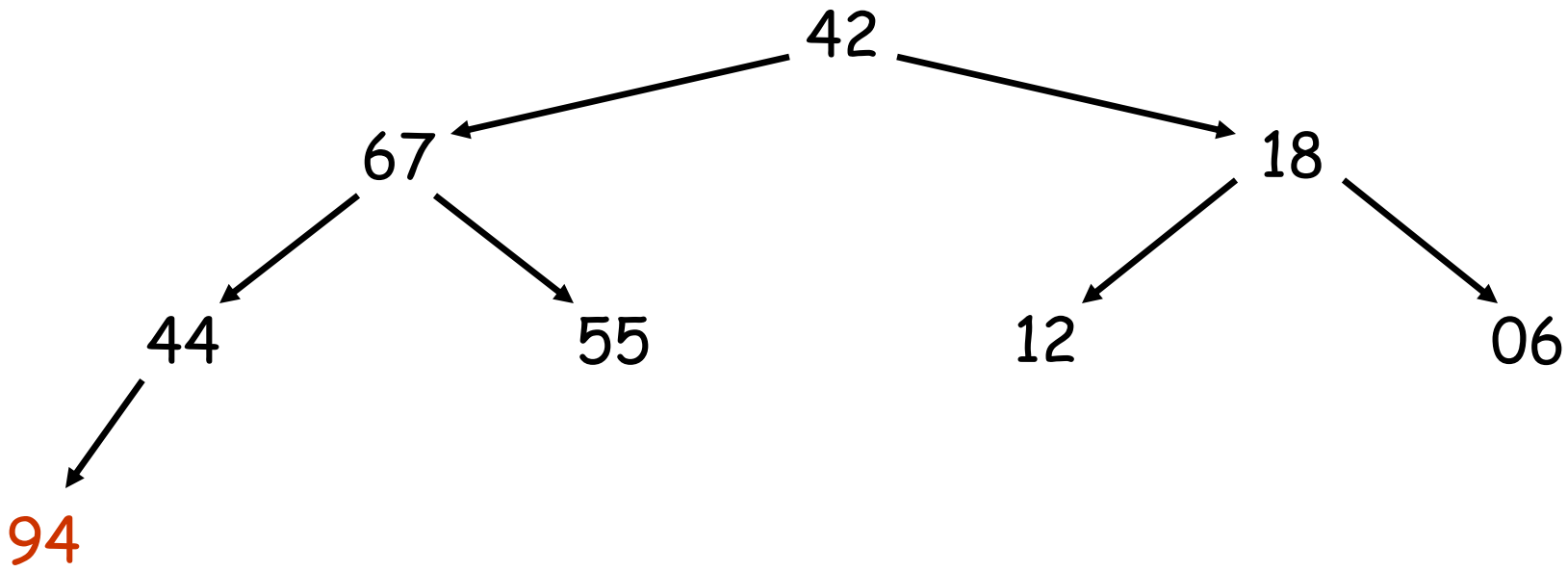
Tri par monceau (« Heapsort »)

Le tri (après avoir construit le tas)

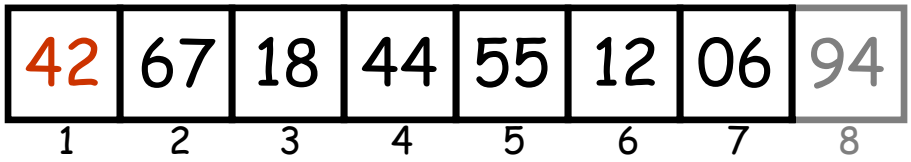


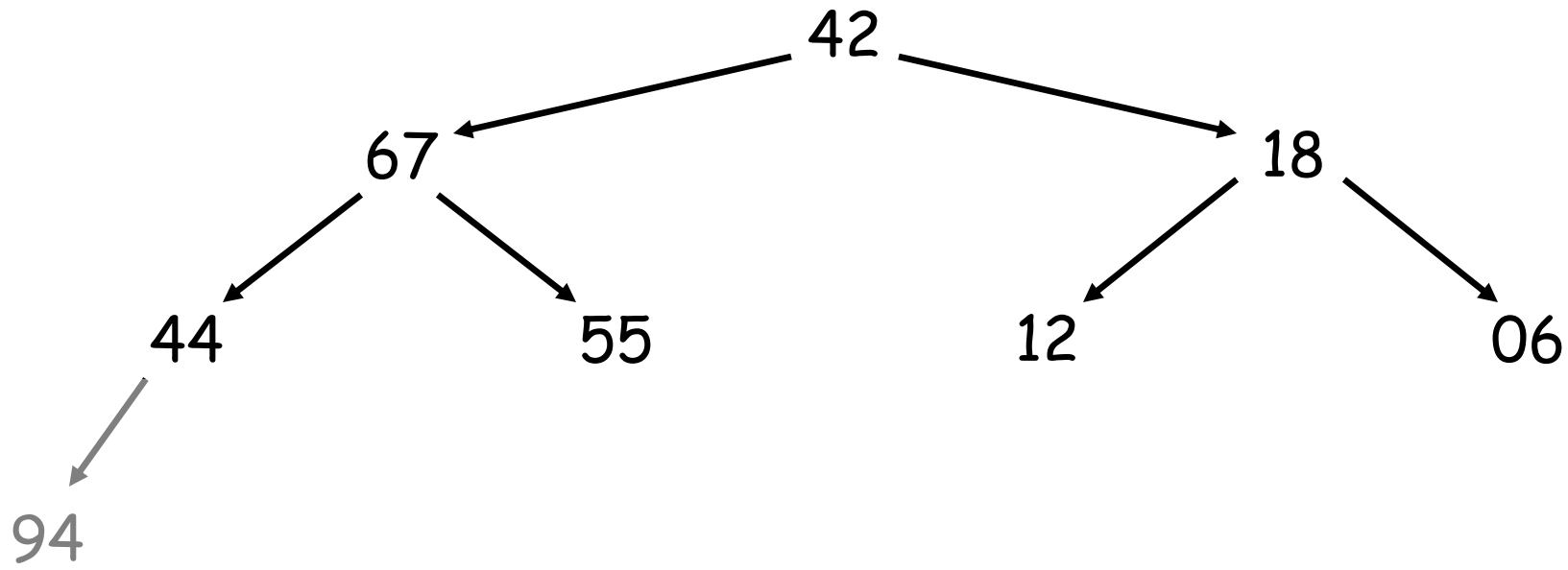
Tri par monceau (« Heapsort »)



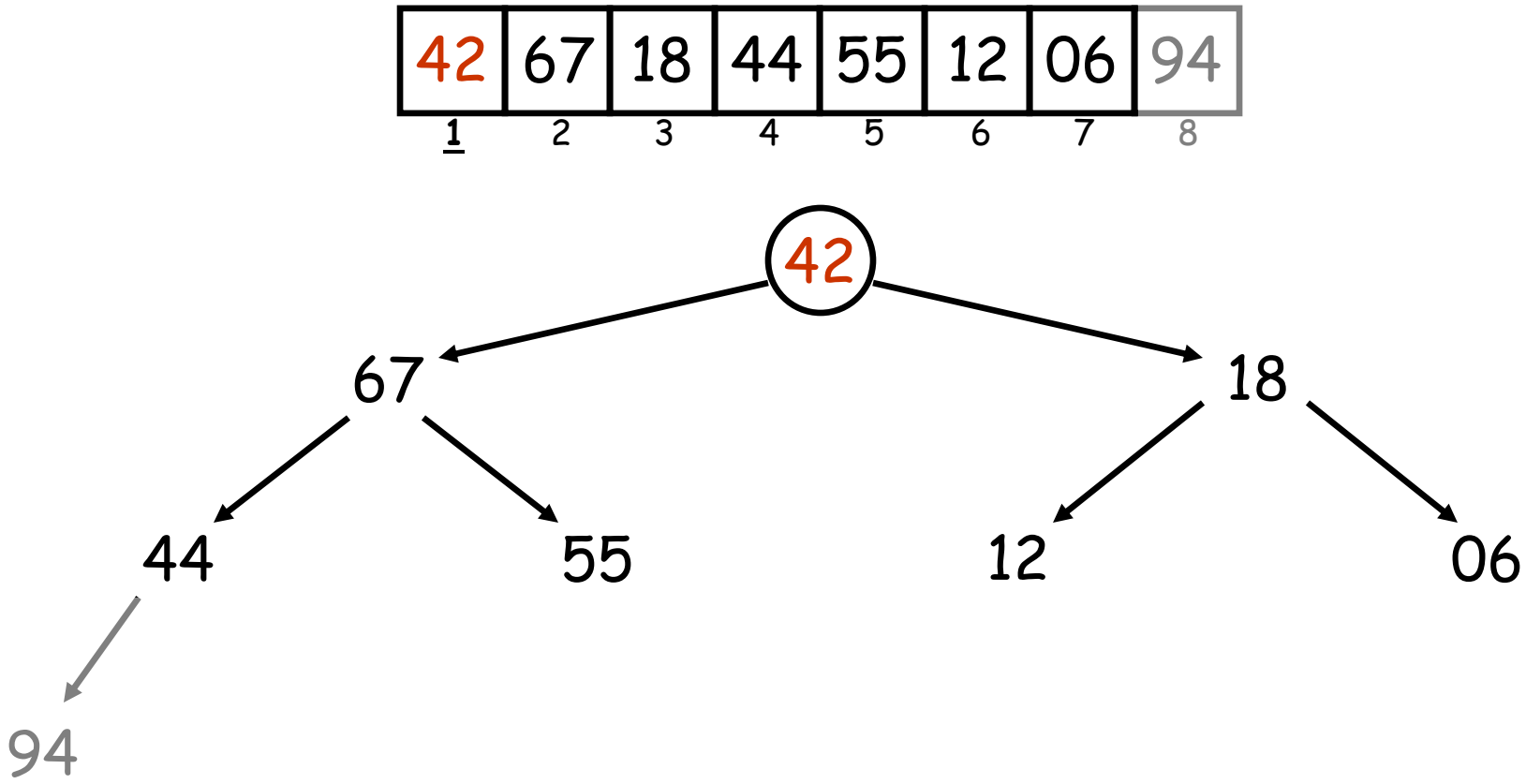


Tri par monceau (« Heapsort »)

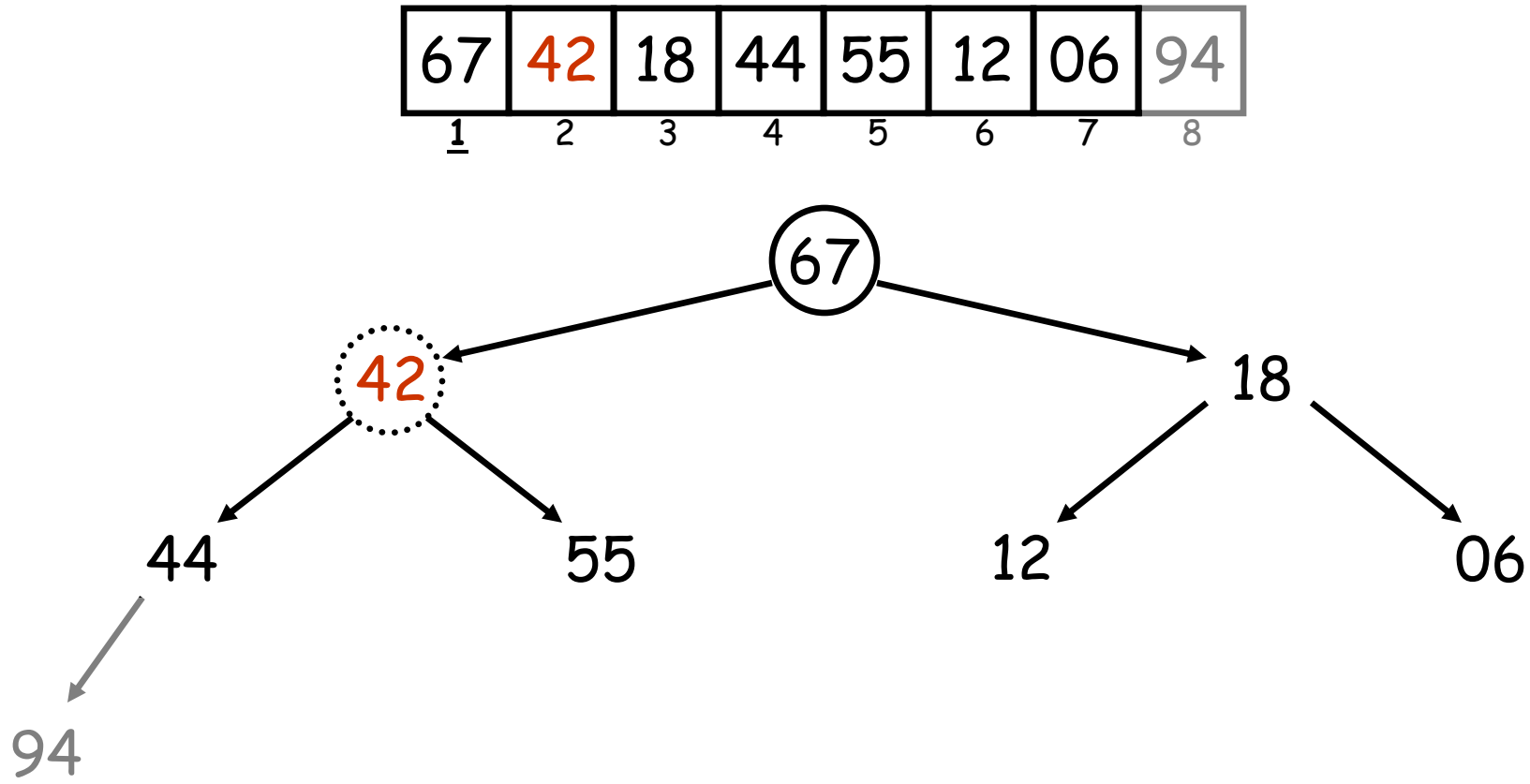




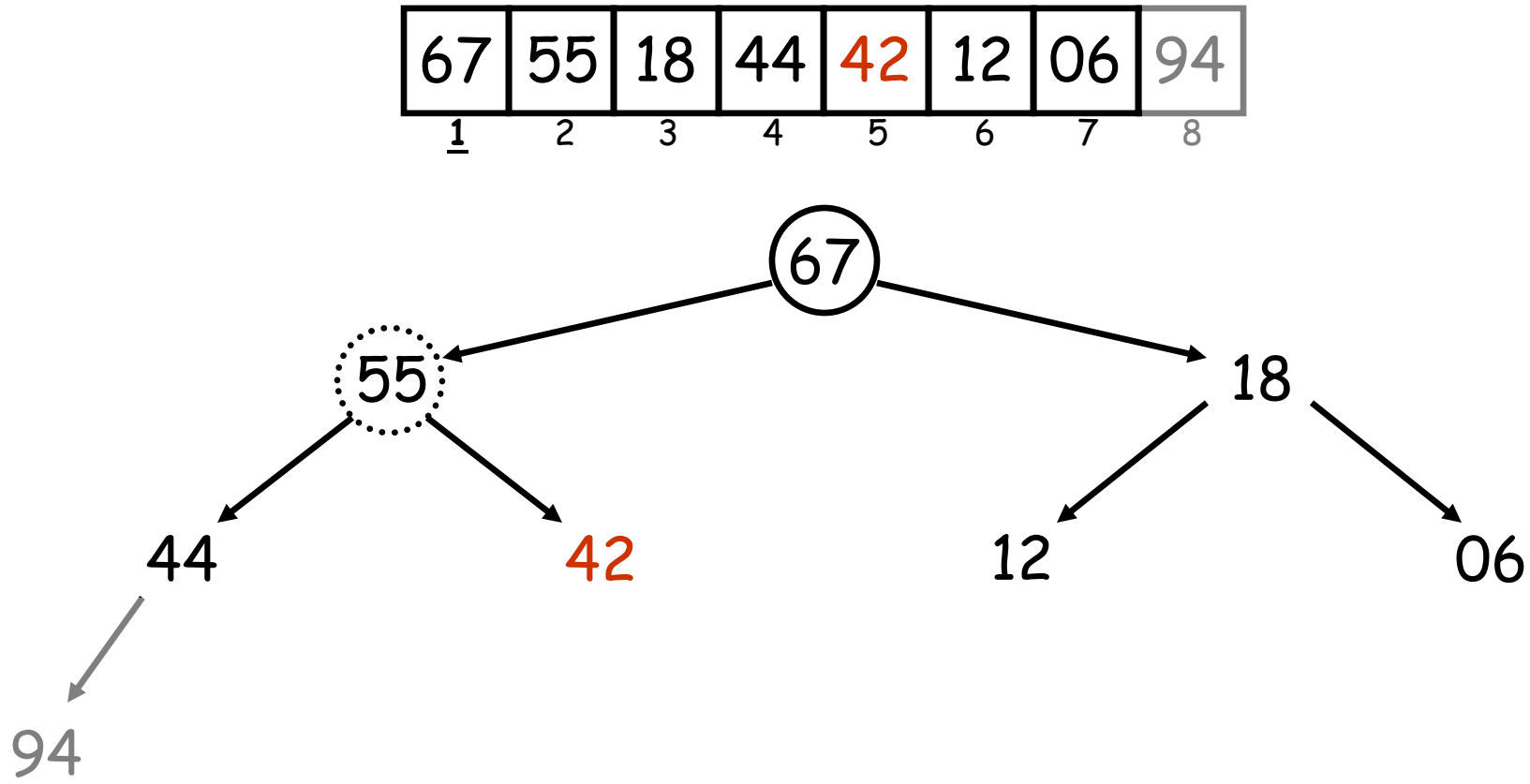
Tri par monceau (« Heapsort »)



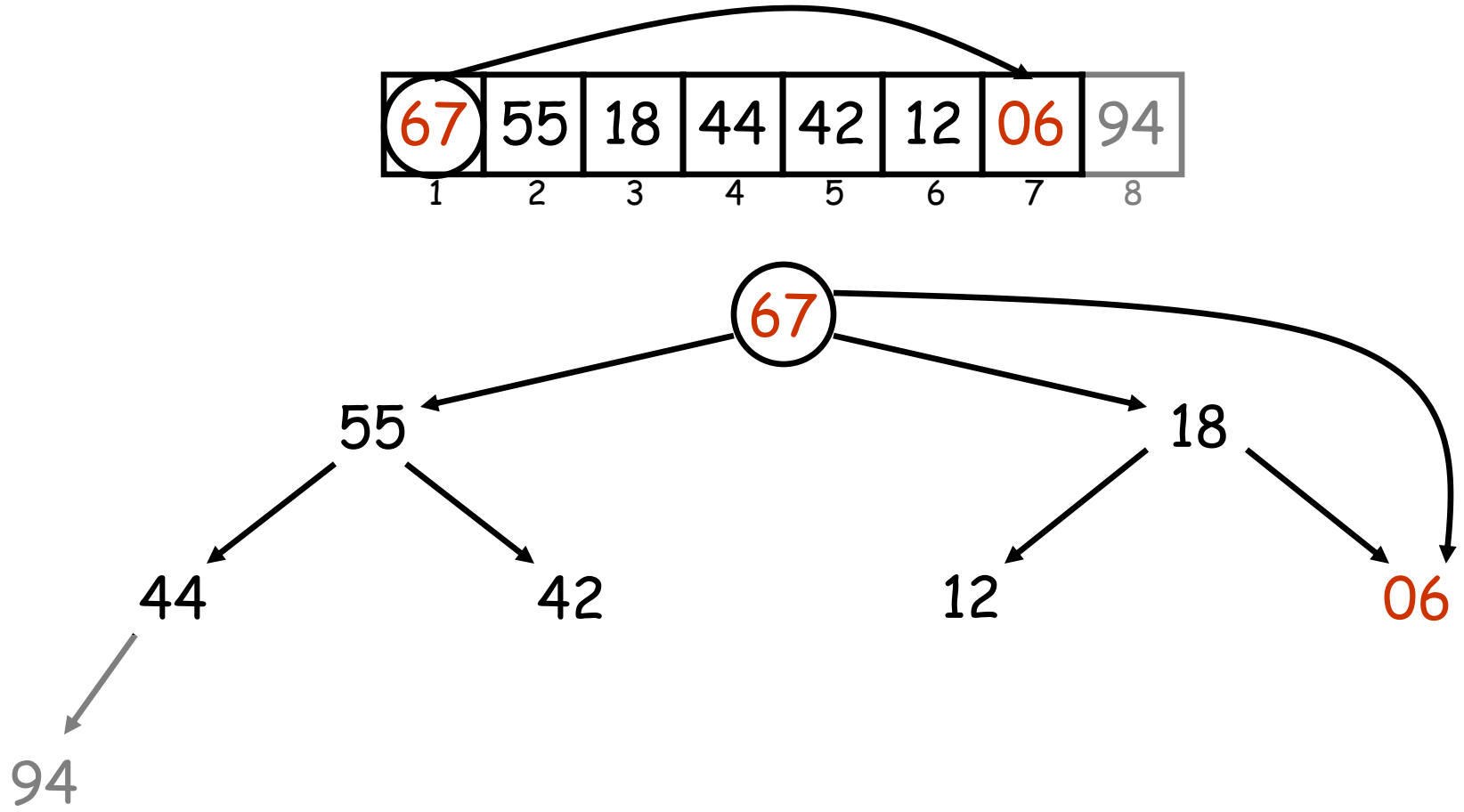
Tri par monceau (« Heapsort »)



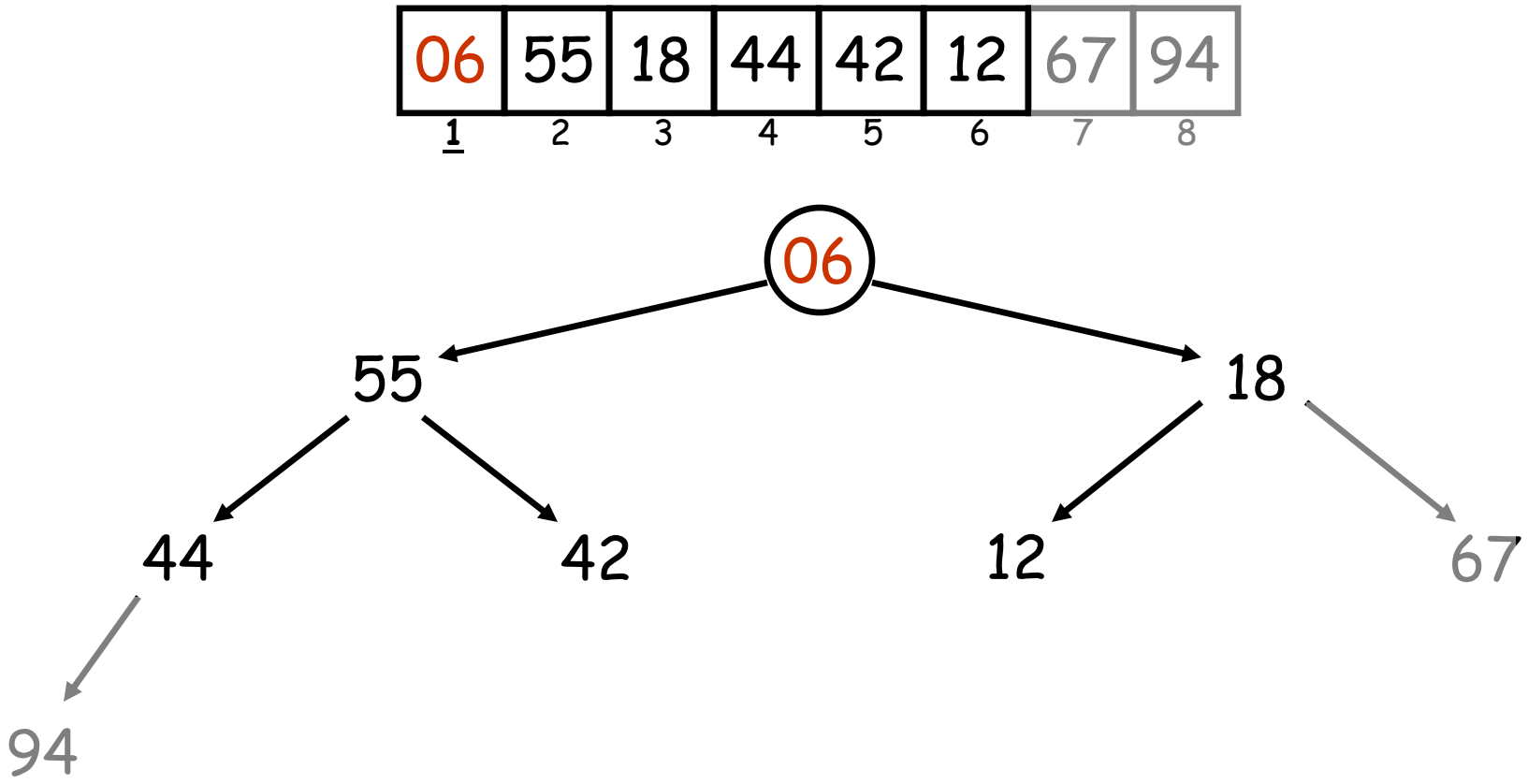
Tri par monceau (« Heapsort »)



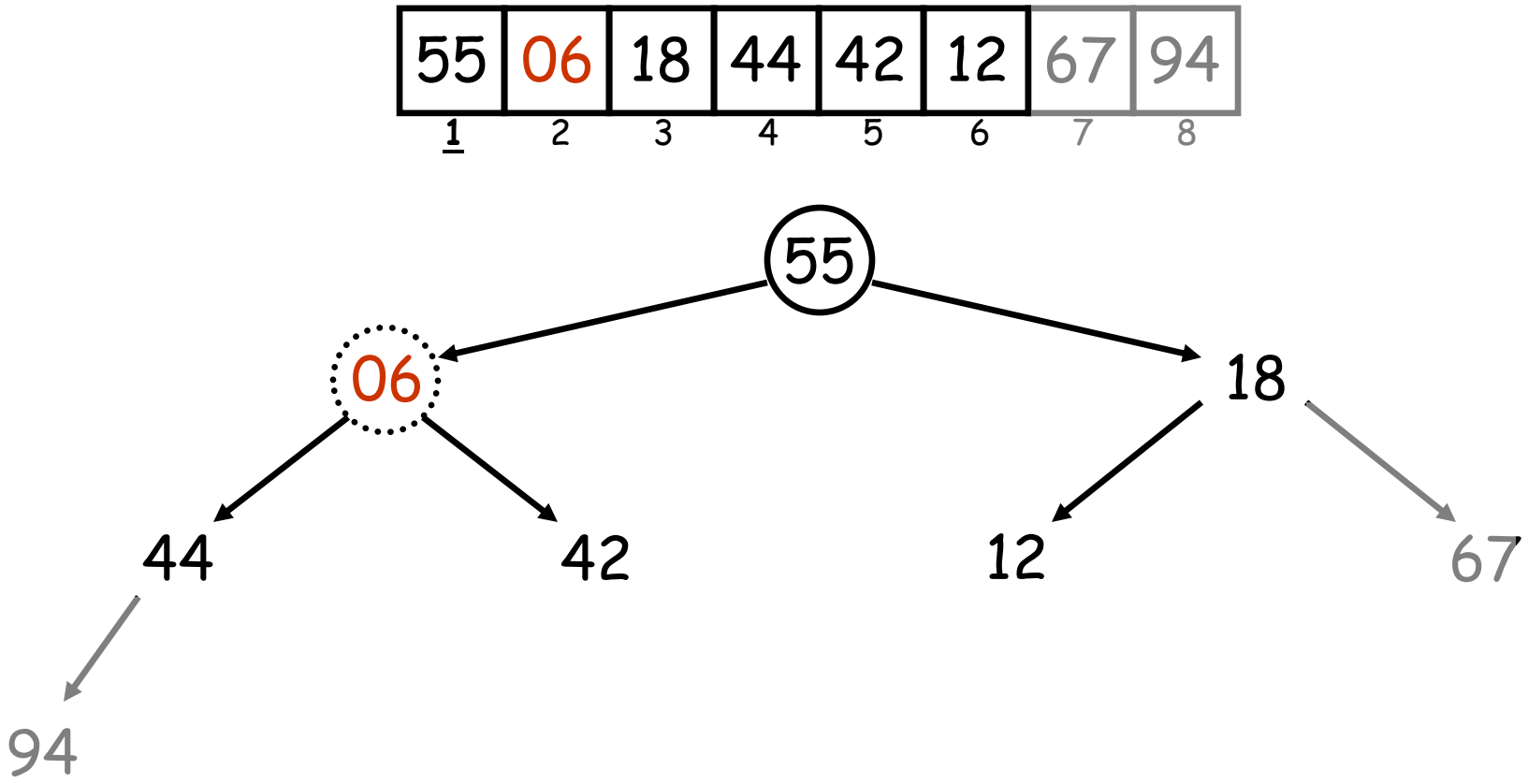
Tri par monceau (« Heapsort »)



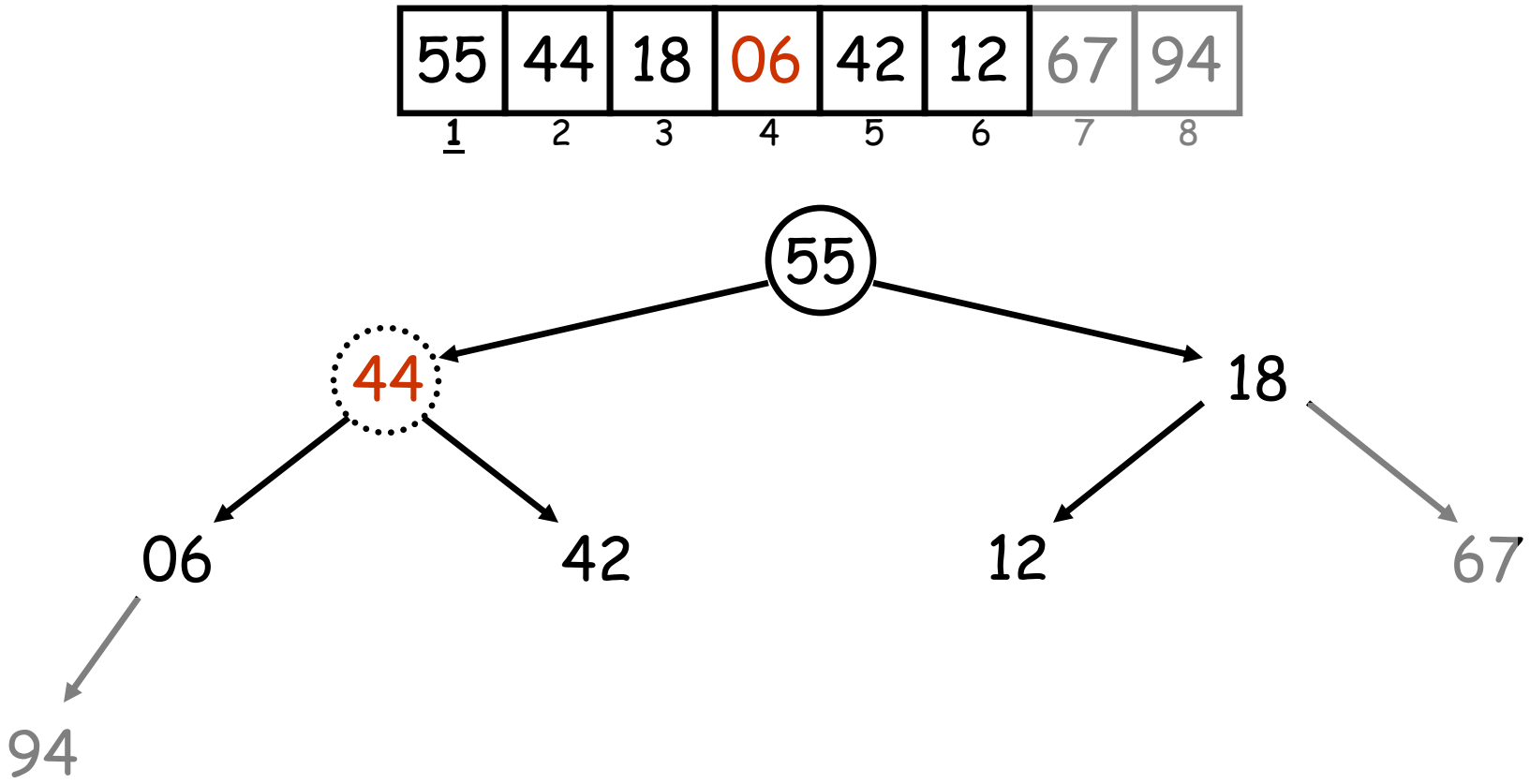
Tri par monceau (« Heapsort »)



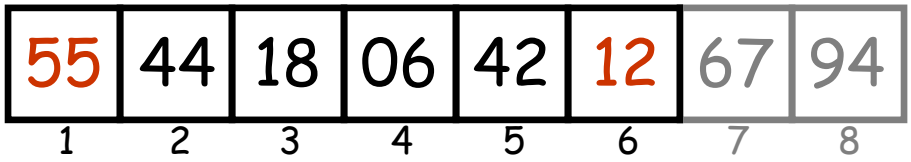
Tri par monceau (« Heapsort »)

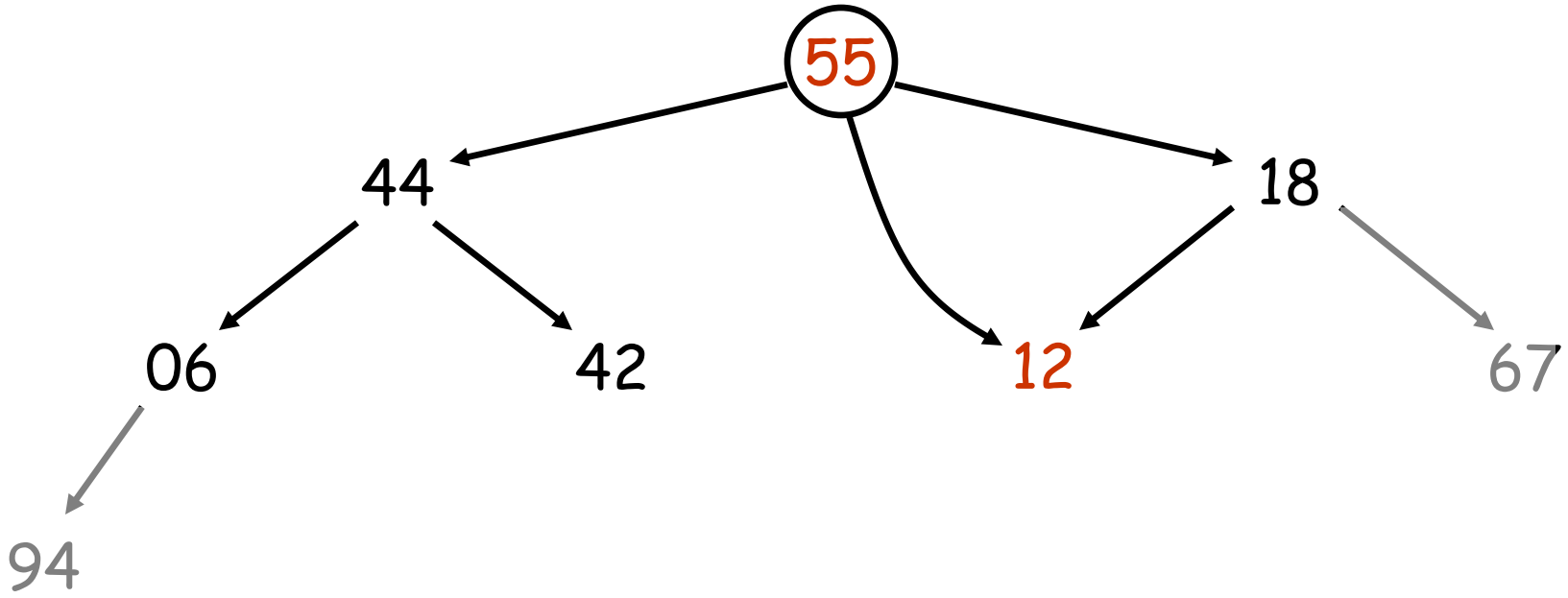


Tri par monceau (« Heapsort »)

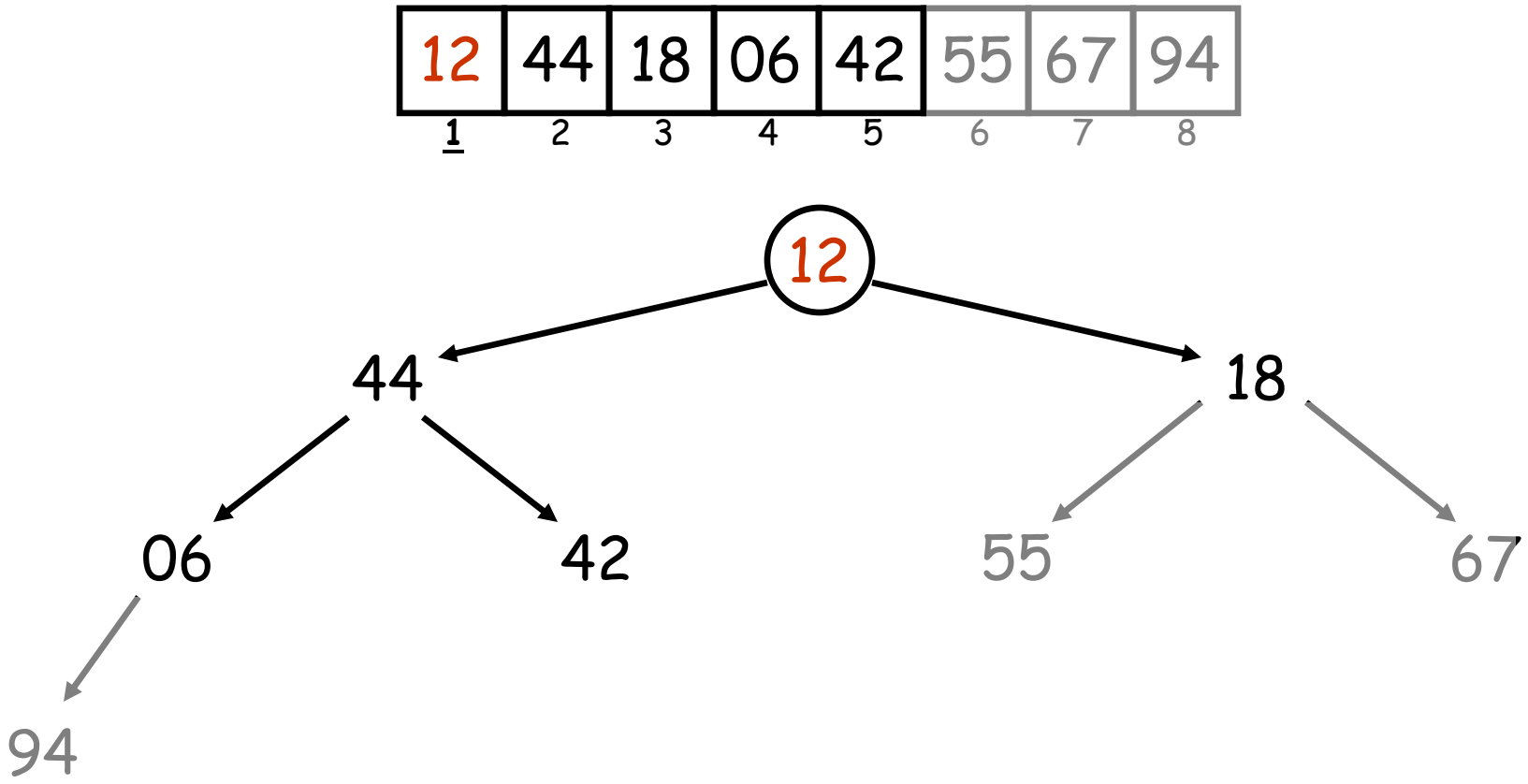


Tri par monceau (« Heapsort »)

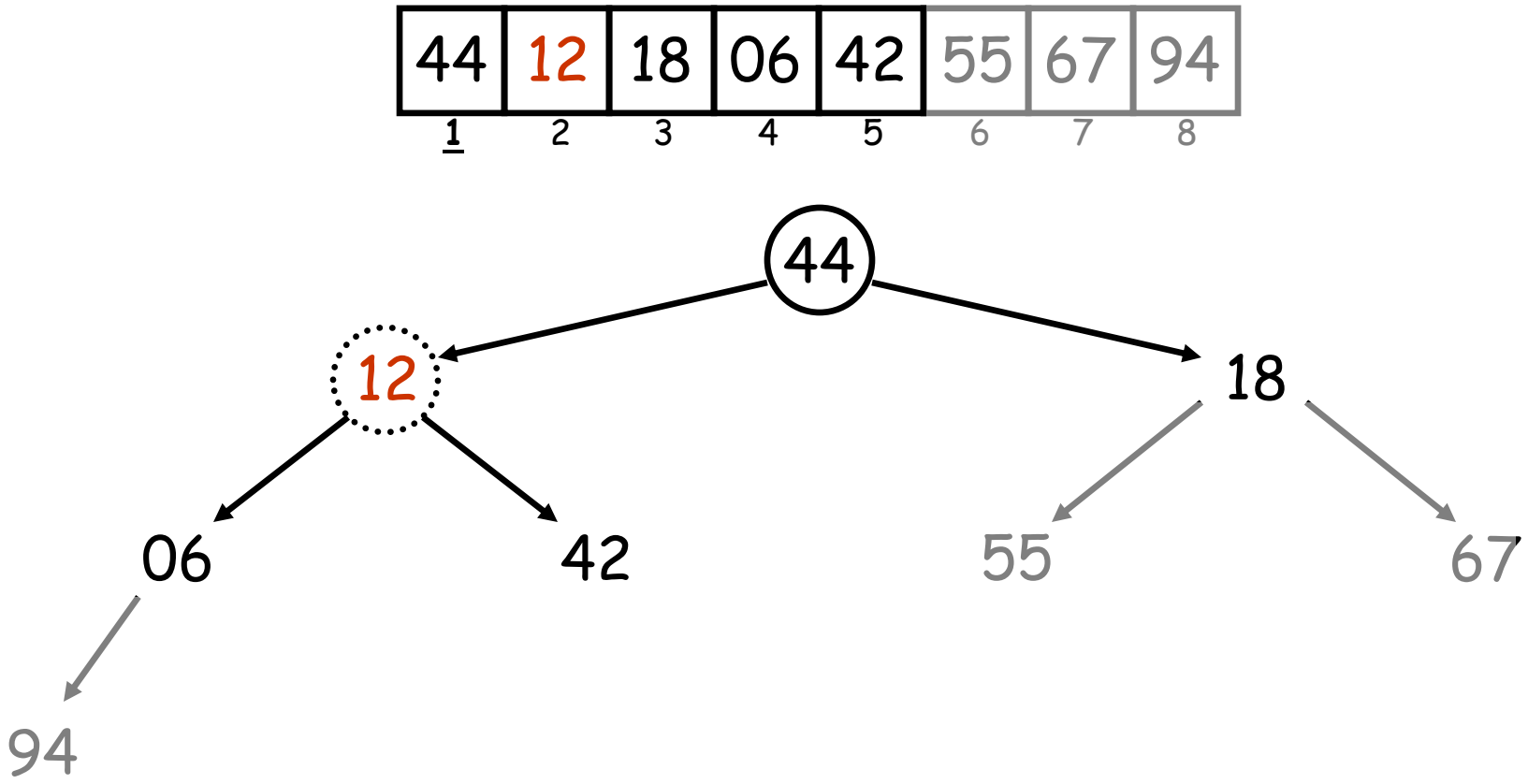




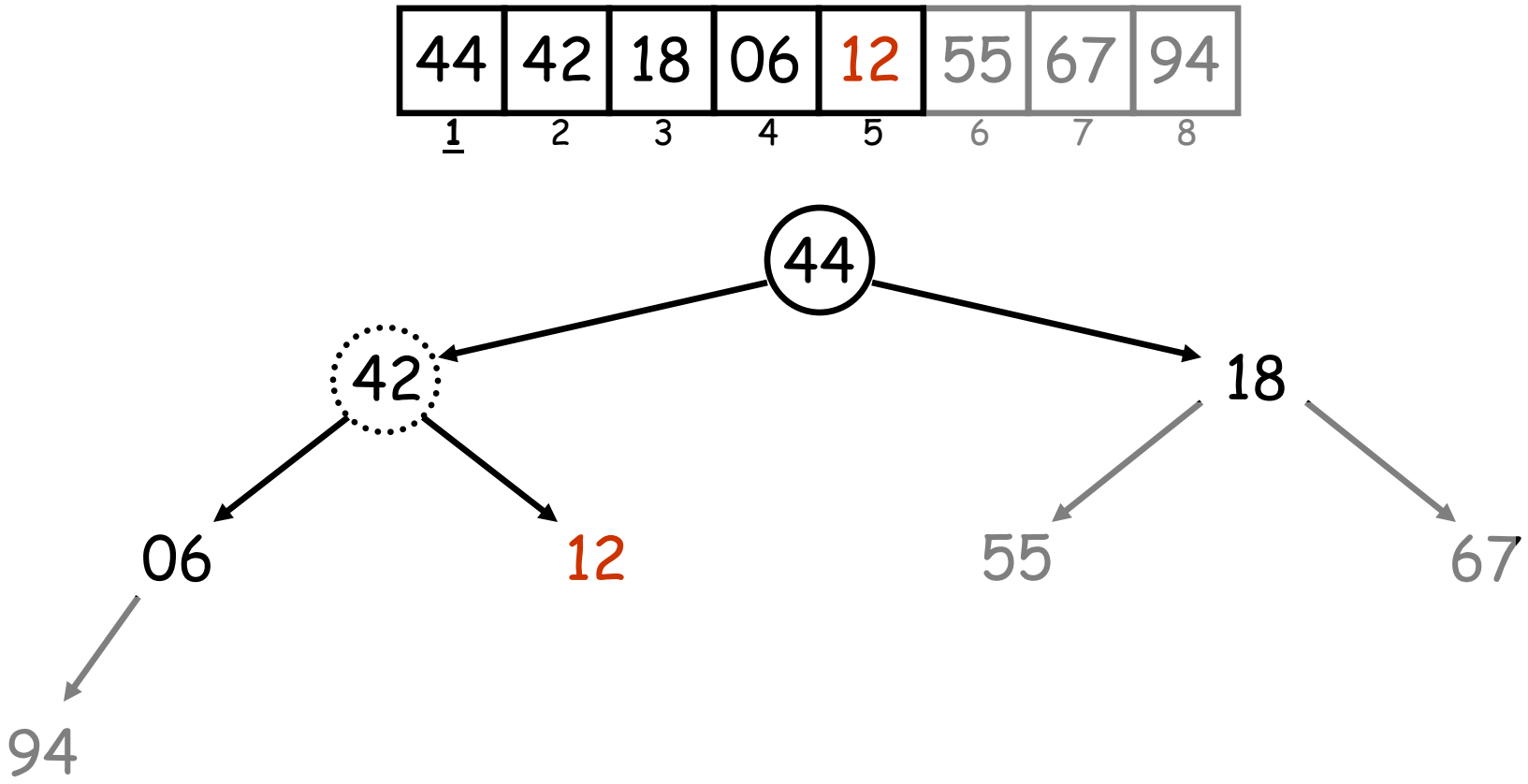
Tri par monceau (« Heapsort »)



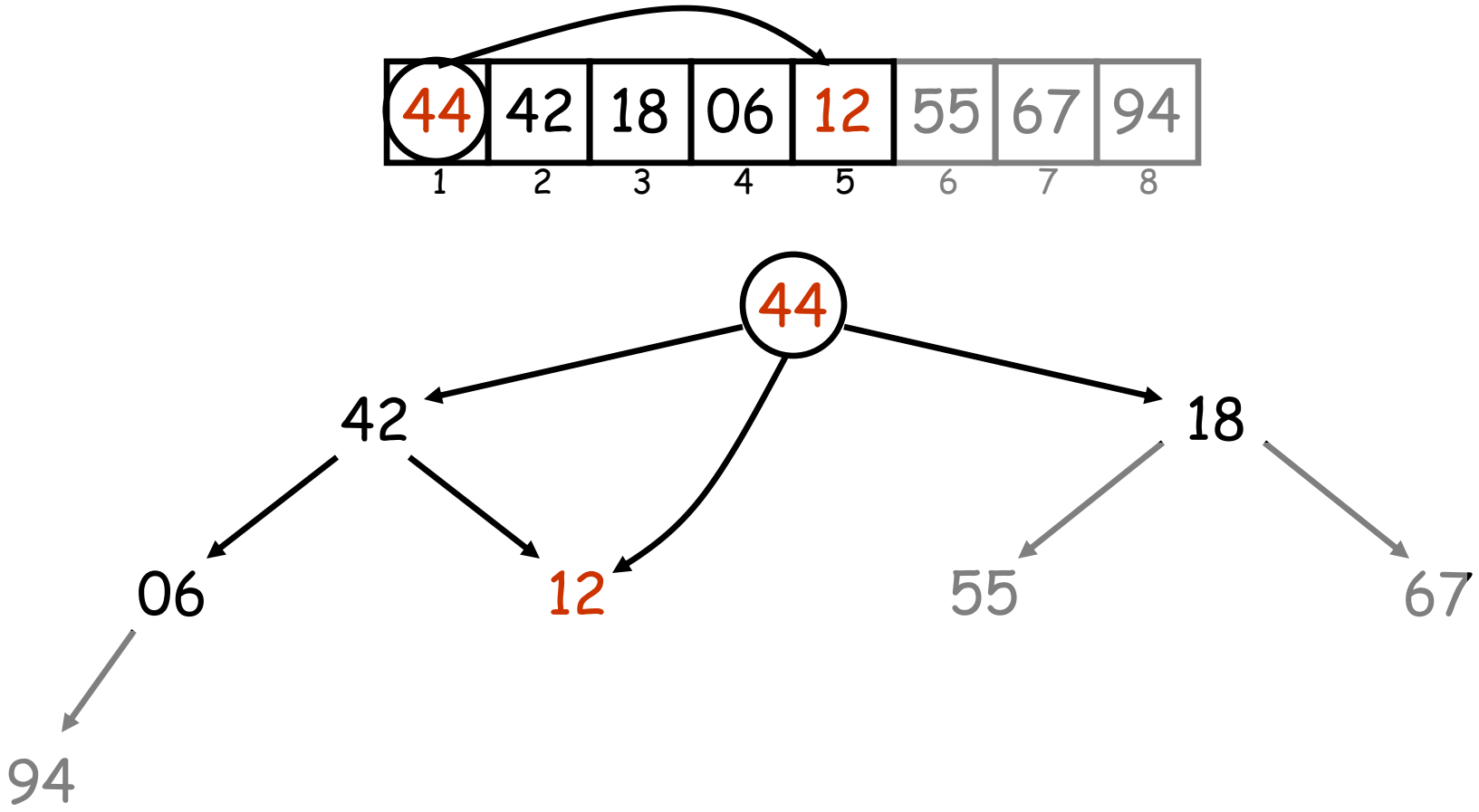
Tri par monceau (« Heapsort »)



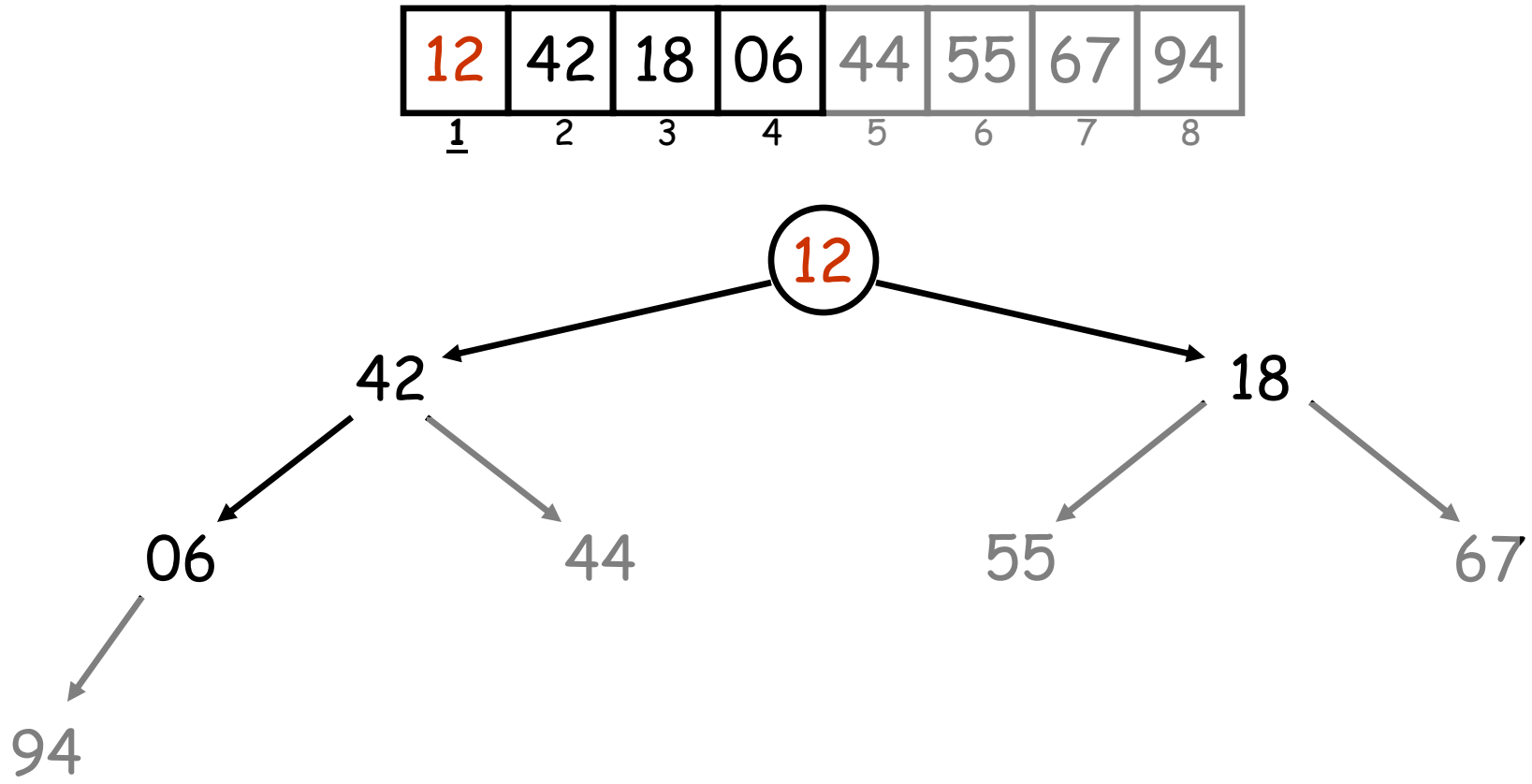
Tri par monceau (« Heapsort »)



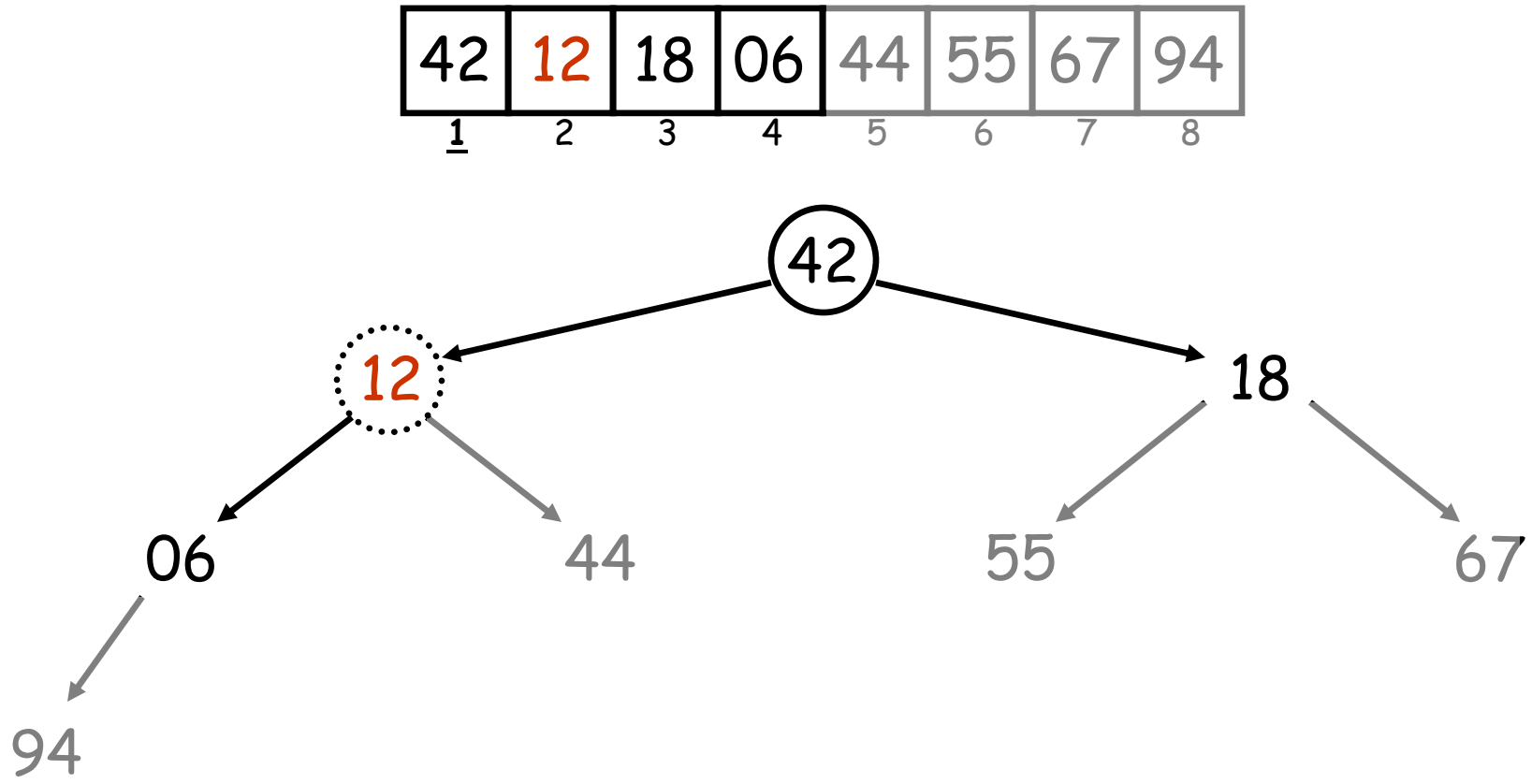
Tri par monceau (« Heapsort »)



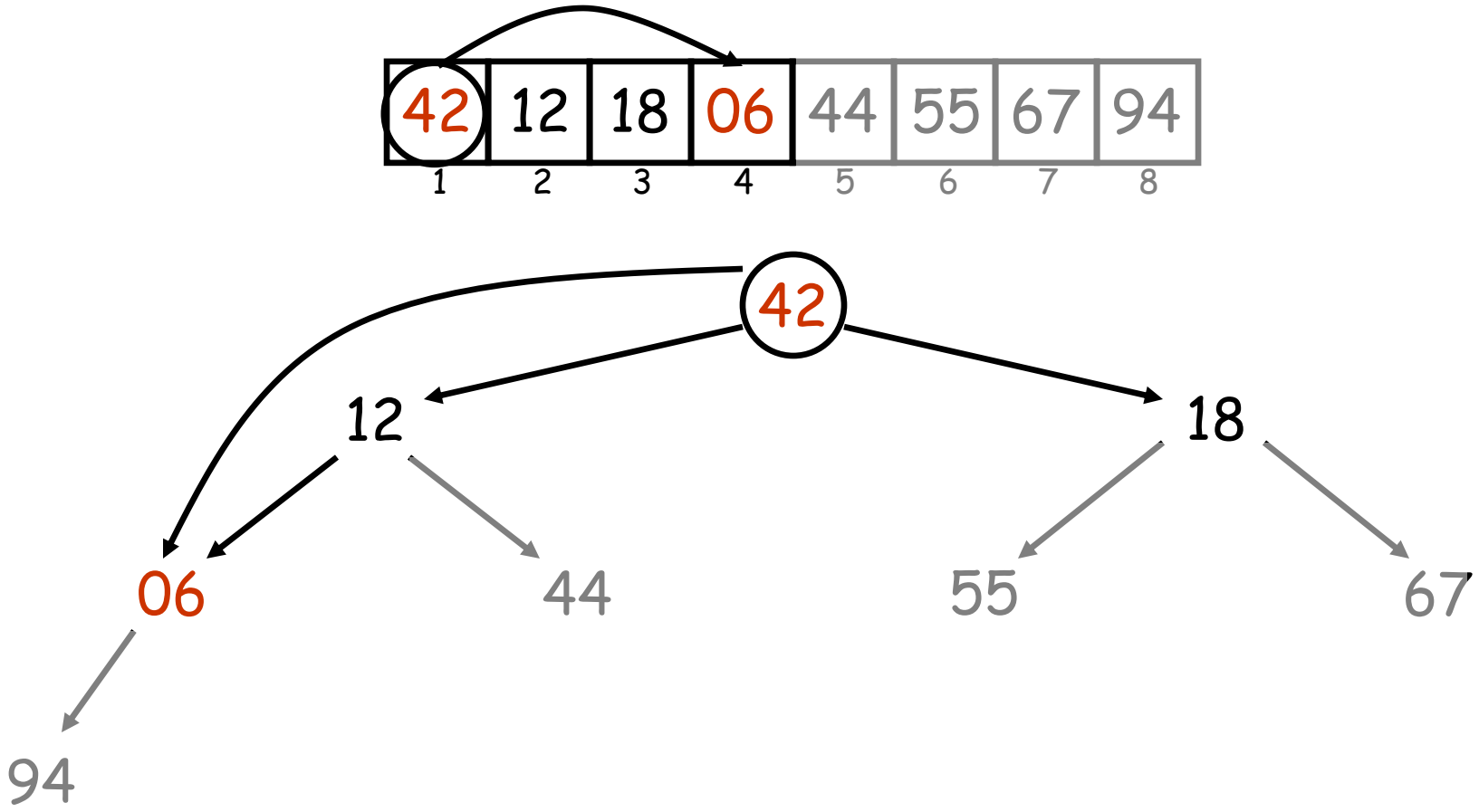
Tri par monceau (« Heapsort »)



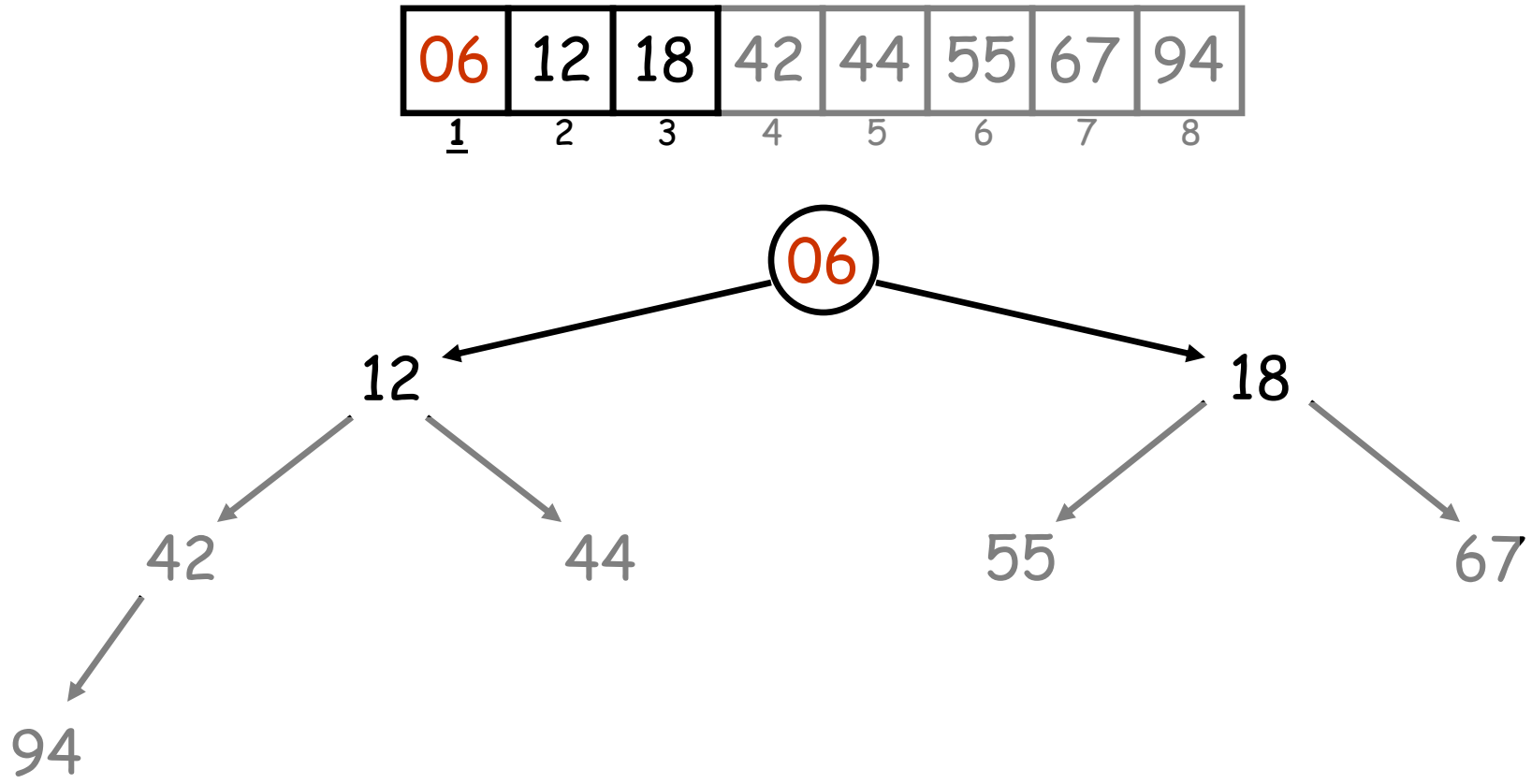
Tri par monceau (« Heapsort »)



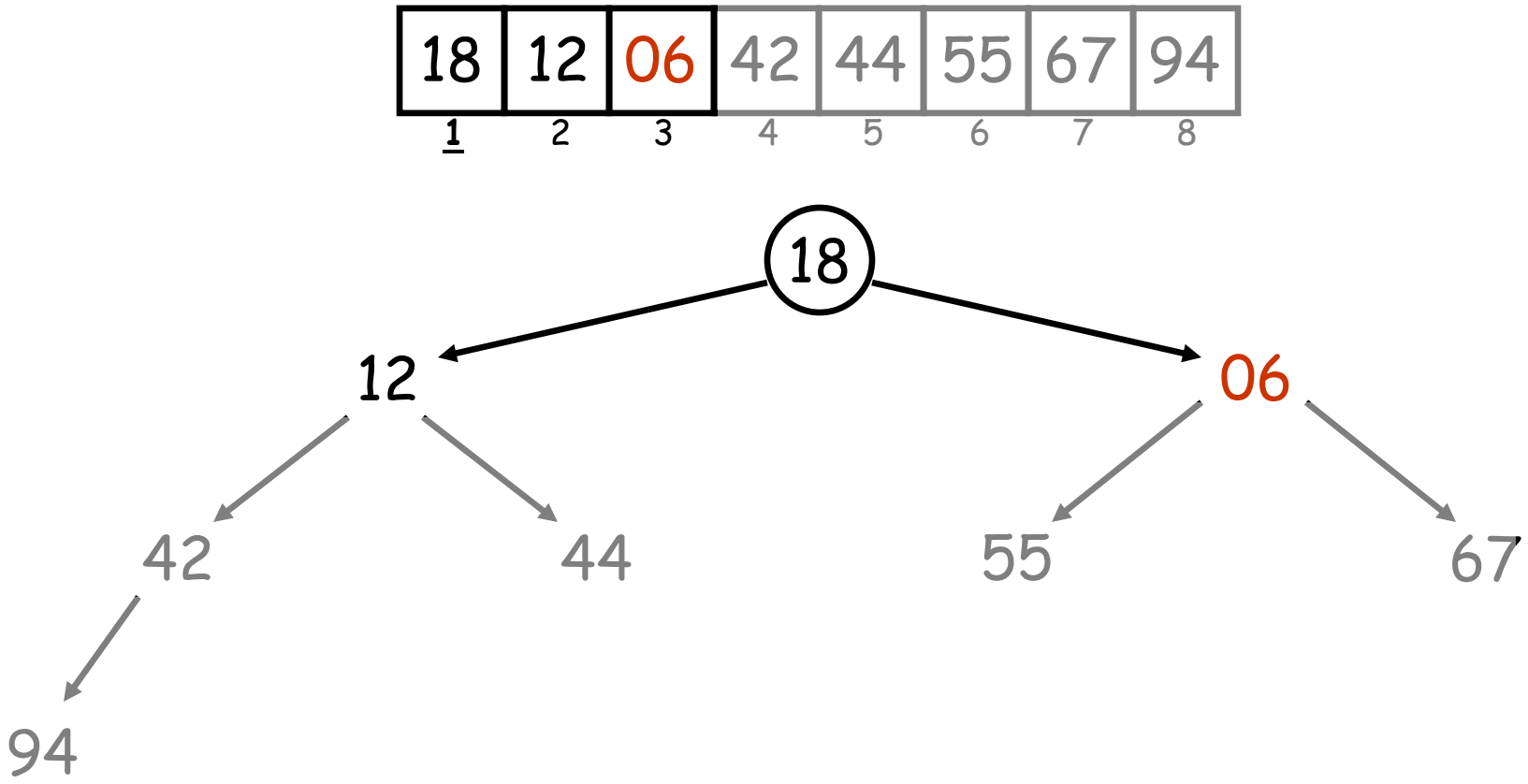
Tri par monceau (« Heapsort »)



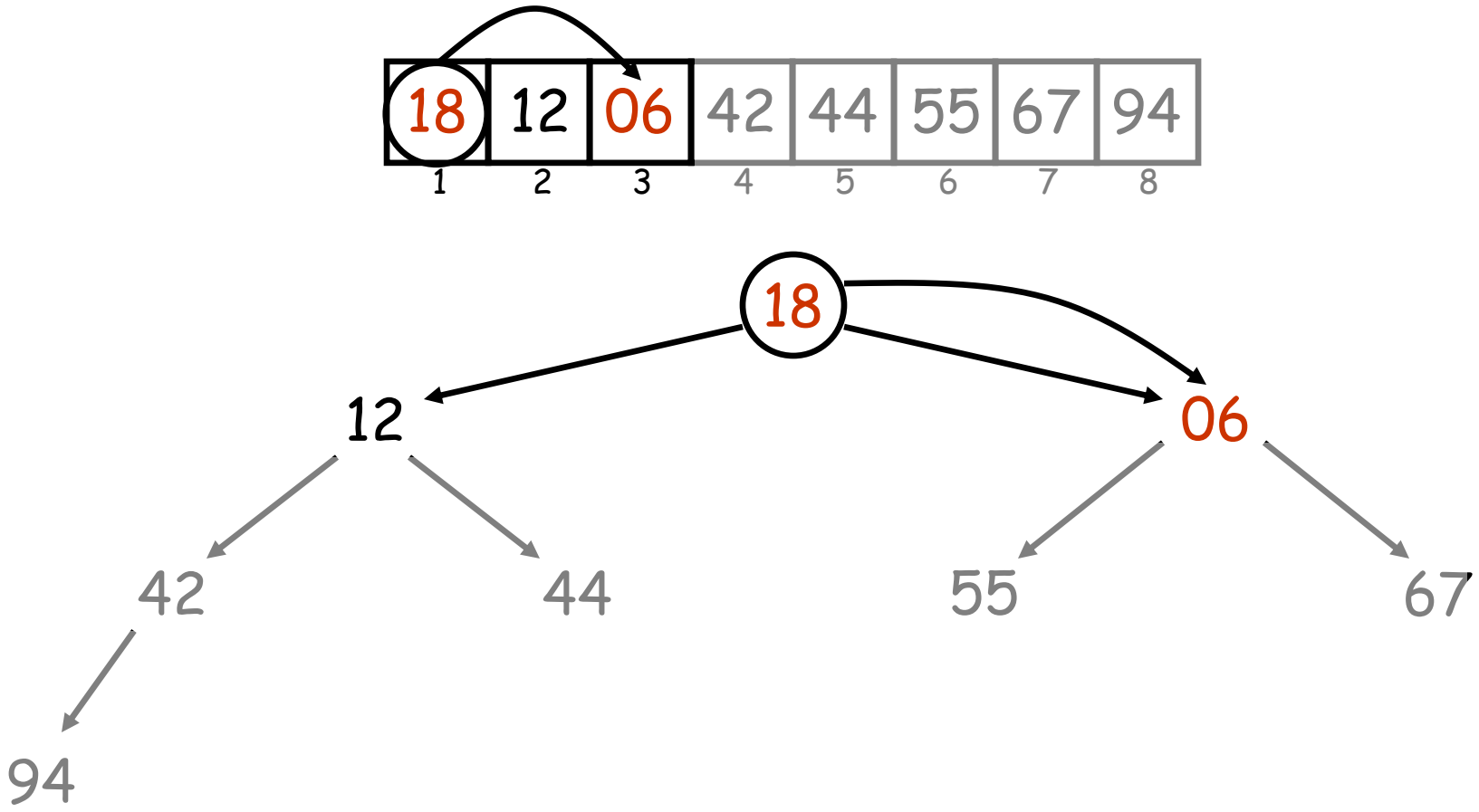
Tri par monceau (« Heapsort »)



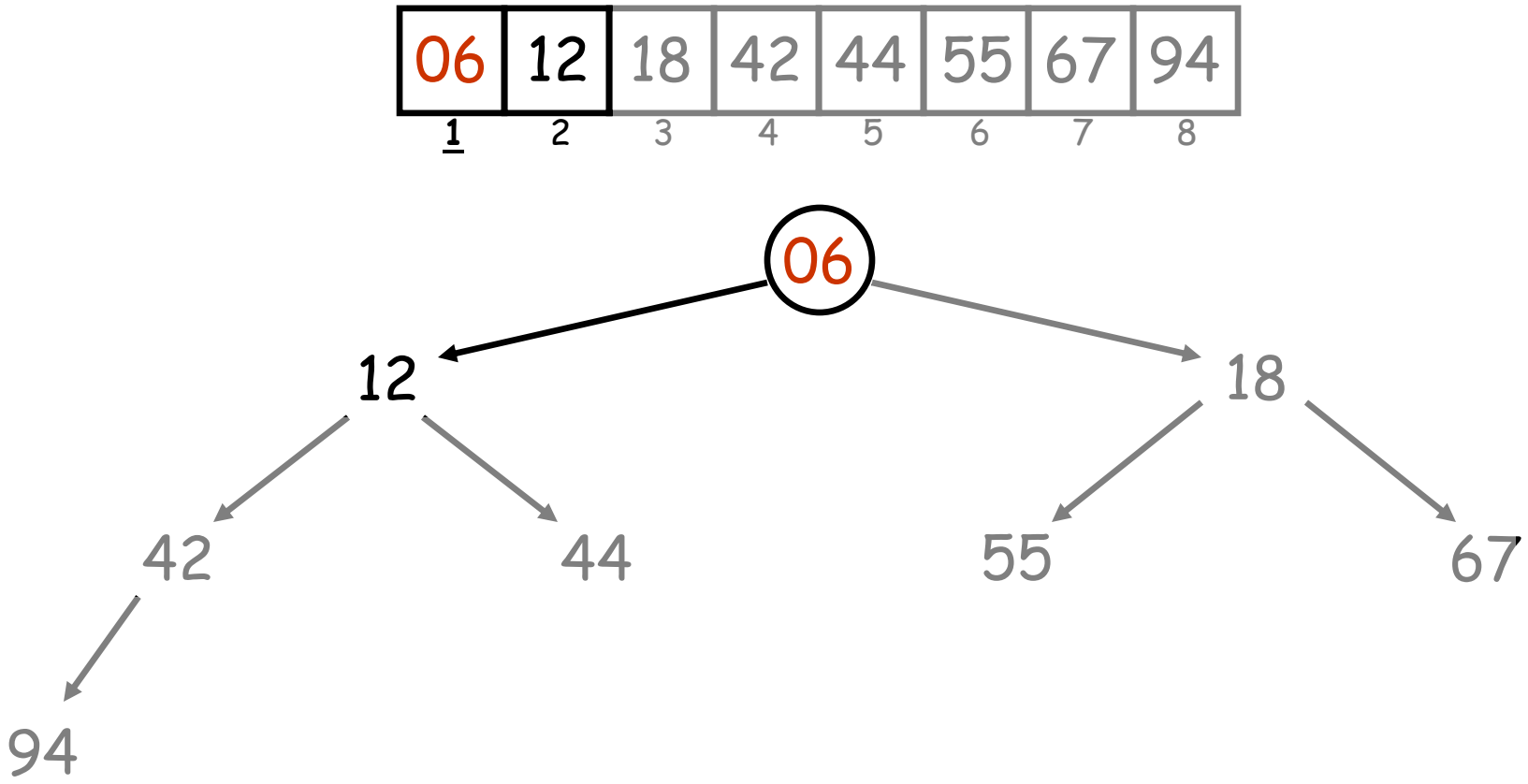
Tri par monceau (« Heapsort »)



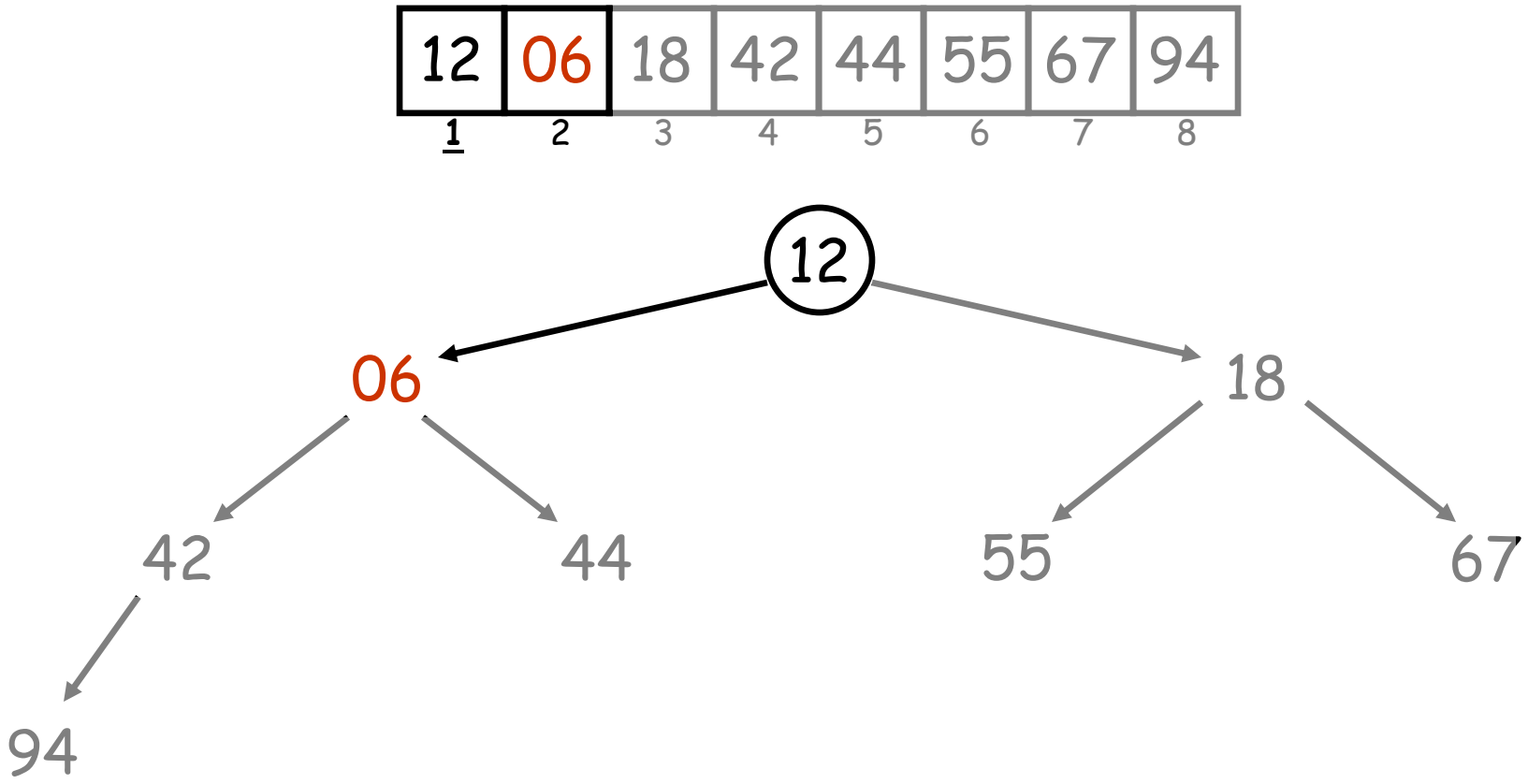
Tri par monceau (« Heapsort »)



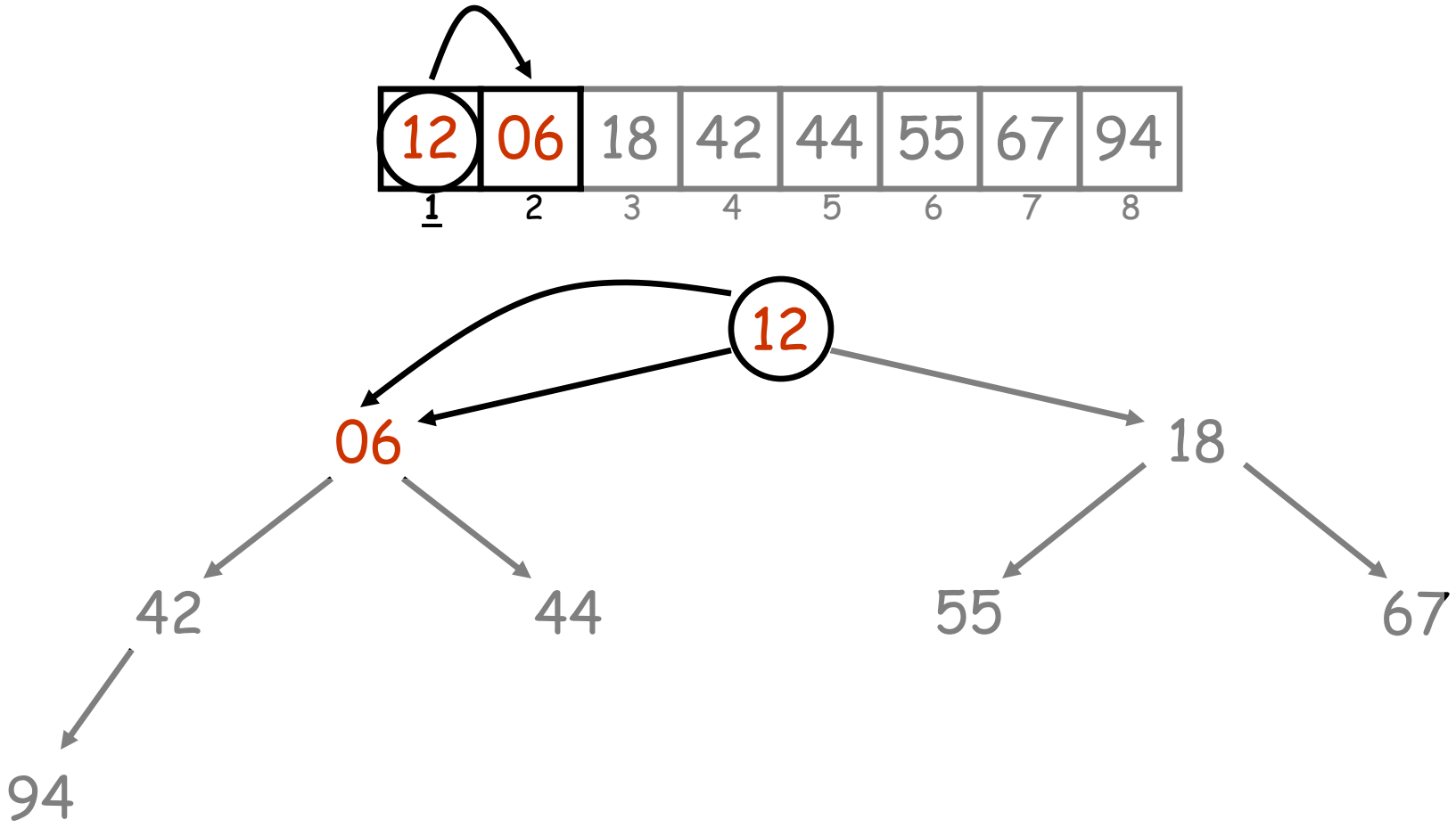
Tri par monceau (« Heapsort »)



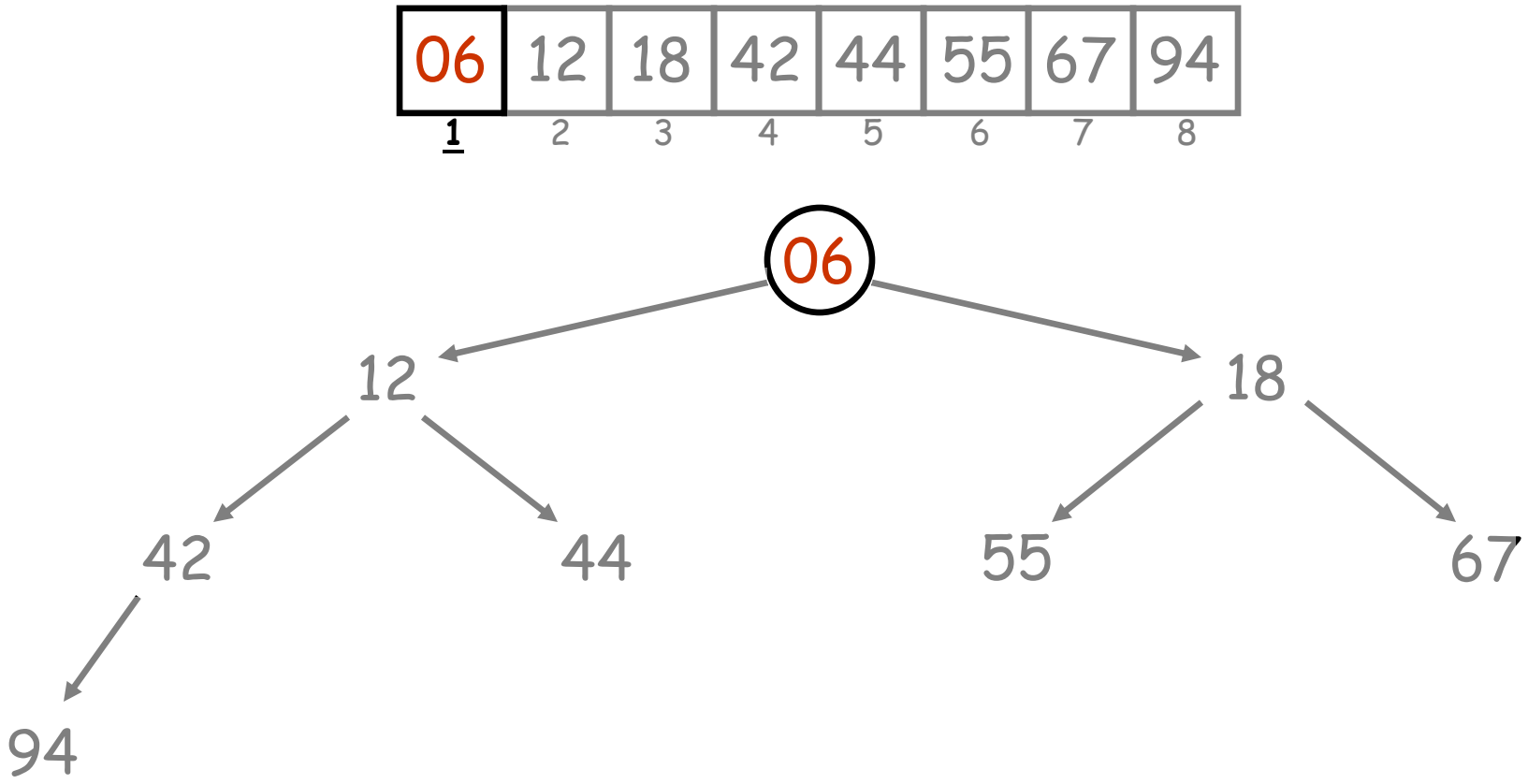
Tri par monceau (« Heapsort »)



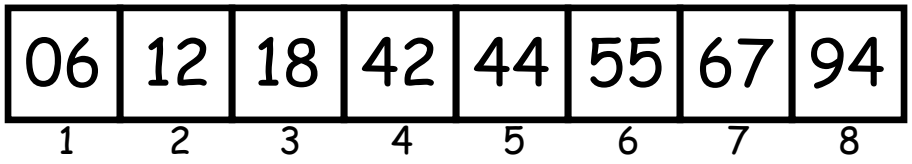
Tri par monceau (« Heapsort »)

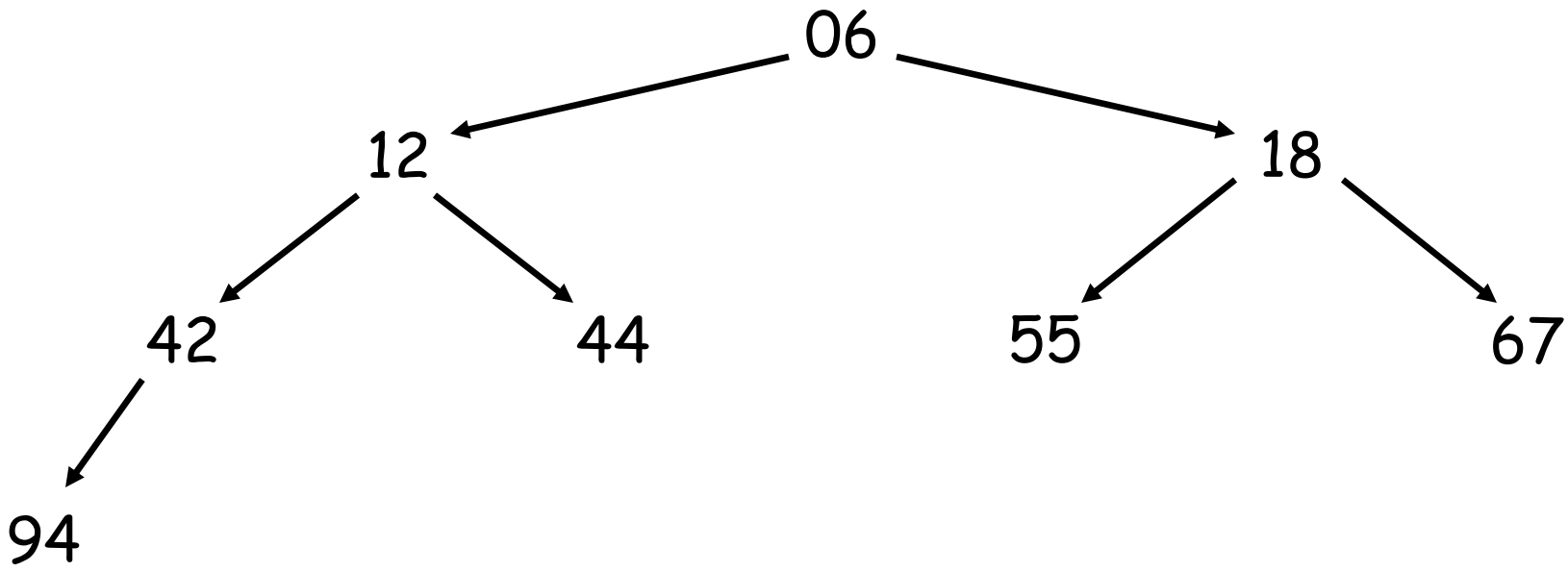


Tri par monceau (« Heapsort »)



Tri par monceau (« Heapsort »)





Monceaux dans la STL

- Les fonctions pour créer et manipuler des monceaux sont définis dans la bibliothèque **<algorithm>** de la STL.
- Pour réarranger des éléments d'un conteneur se situant entre **debut** et **fin** (incluant debut et excluant fin) on utilise :

  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)

- Voir exemple d'utilisation en page suivante

- 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), ce qui limite l'utilisation de ces fonctions aux conteneurs fournissant de tels itérateurs (comme <vector> et <deque>; impossible avec <list>).

Monceaux dans la STL (exemple à essayer)

```
#include <iostream>
#include <algorithm>
#include <vector>

int main()  //exemple extrait de www.cplusplus.com
{
     using namespace std;

     int myints[] = { 10, 20, 30, 5, 15 };
     vector<int> v(myints, myints + 5); //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;

     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;
}
```