BDav-M1
Bases de données avancées

Cours de Cristina Sirangelo
IRIF, Université Paris Diderot
Assuré en 2021-2022 par Amélie Gheerbrant
amelie@irif.fr

Implémentation des index

Sources (quelques slides empruntés et réadaptés) :

- cours *Database systems principles* - H.G. Molina, Stanford Univ.
- slides du livre *Database systems concepts* -
  A. Silberschatz, Yale U. & H. Korth, Lehigh U. & S.Sudarshan, IIT Bombay

Index comme type abstrait

Indépendamment du type d'index (dense/ non-dense, primaire/secondaire), la manipulation d'un fichier de données indexé nécessite trois opérations de base sur les index :

- **recherche** du couple avec clef c
- **insertion** d'un couple <c, p>
- **suppression** d'un couple <c, p>

Remarque : la mise à jour d'un élément de l'index peut être vue comme une suppression suivie d'une insertion

Index comme type abstrait

Type abstrait Index

- ***Données*** : une collection de couples <clef, pointeur>
- ***Opérations*** :
  - **recherche** du couple avec clef c
  - **insertion** d'un couple <c, p>
  - **suppression** d'un couple <c, p>

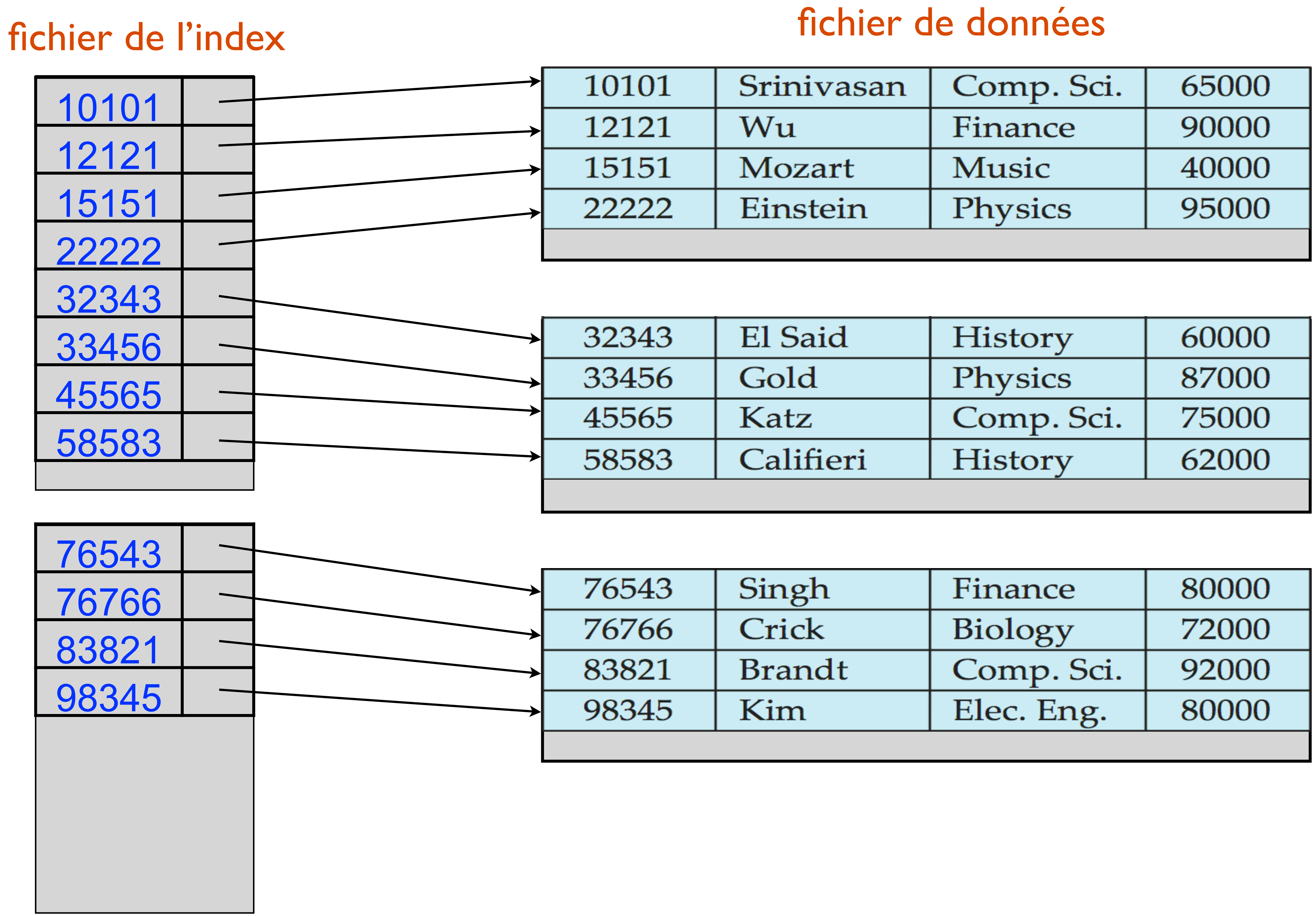
On reconnaît un type abstrait familier. Lequel?

Index et dictionnaires

- En tant que type abstrait un index n'est rien d'autre qu'un **dictionnaire**!
- Implémentation des index :
  - implémentation du type dictionnaire adaptée et optimisée pour une représentation en mémoire secondaire
- Plusieurs implémentations possibles :
  - Séquentielle (intérêt historique, pas utilisée en pratique)
  - Arbres B^+
  - Hachage
  - Bitmap (pas abordé)
  - Filtre de Bloom
  - …

Index séquentiels

- L'ensemble des couples <c, p> est stocké dans un fichier séquentiel trié par clef



Index séquentiels

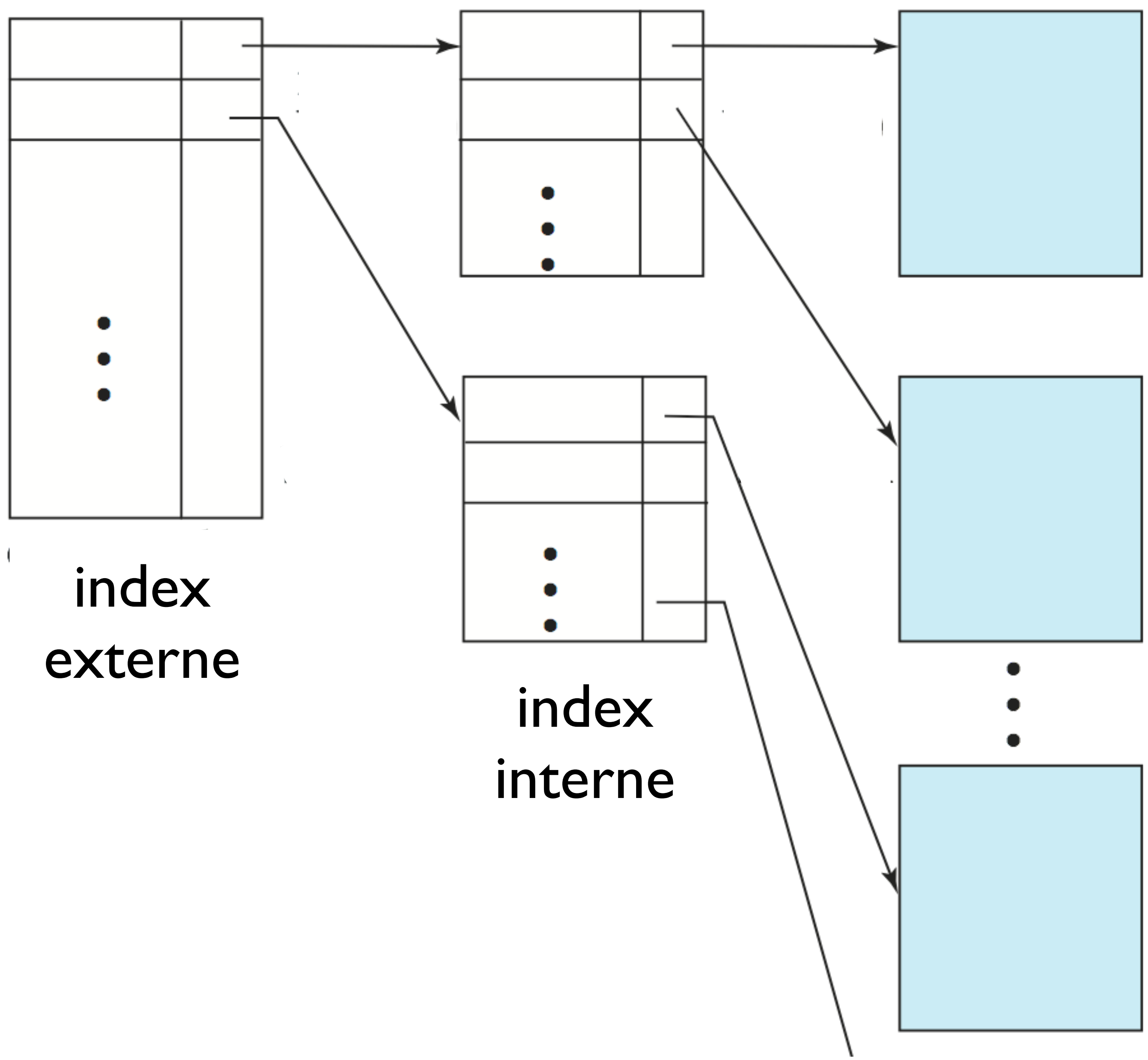
Insertion / suppression : comme pour les fichiers séquentiels de données

Recherche : dichotomique

- Coût : $\lceil \log b \rceil$ b : nombre de blocs occupés par l'index
- Remarque: ce coût peut être élevé pour un gros index
  - Exemple: index dense sur un fichier de 1 000 000 d'enregistrements
    - supposer par exemple 100 entrées de l'index par bloc
    - $\Rightarrow$ l'index occupe 10 000 blocs
    - $\Rightarrow$ un recherche demande $\lceil \log 10000 \rceil = 14$ accès aux blocs du disque
    - temps typique d'accès à un bloc : 10 ms $\Rightarrow$ 140 ms par recherche
    - (seulement) 7 recherches par seconde!
- En cas de blocs d'overflow la recherche peut devenir linéaire
- Réorganisation périodique du fichier nécessaire

Index séquentiels multi-niveau

- Pour améliorer les performances de recherche sur des gros index :
  - traiter le fichier de l'index comme un fichier séquentiel de données : construire un index non-dense sur l'index (même clef de recherche)

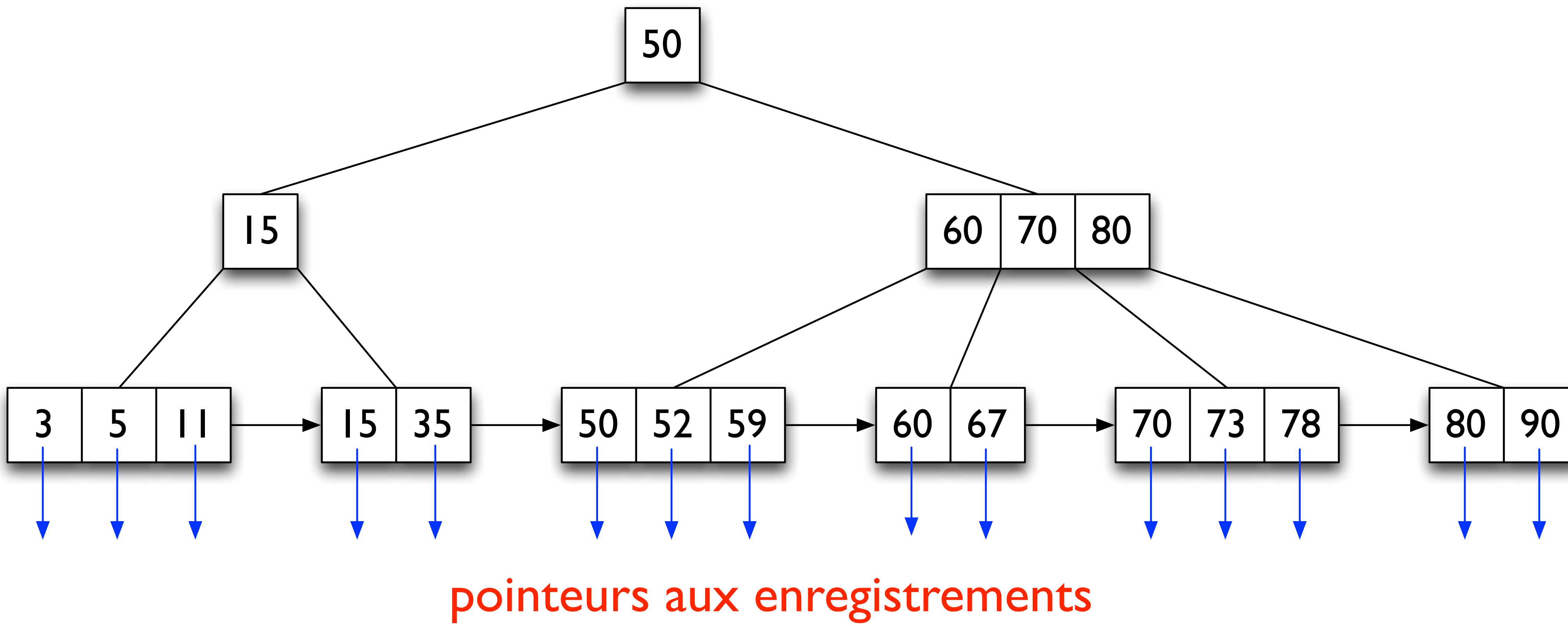


- beaucoup moins de clefs dans l'index externe
  (une par bloc de l'index interne)
- ⇒ beaucoup moins de blocs
- s'il est encore trop gros : un autre niveau d'indexation etc.
- Recherche :
  - dichot. sur l'index externe
  - ensuite un accès par niveau

Index séquentiels

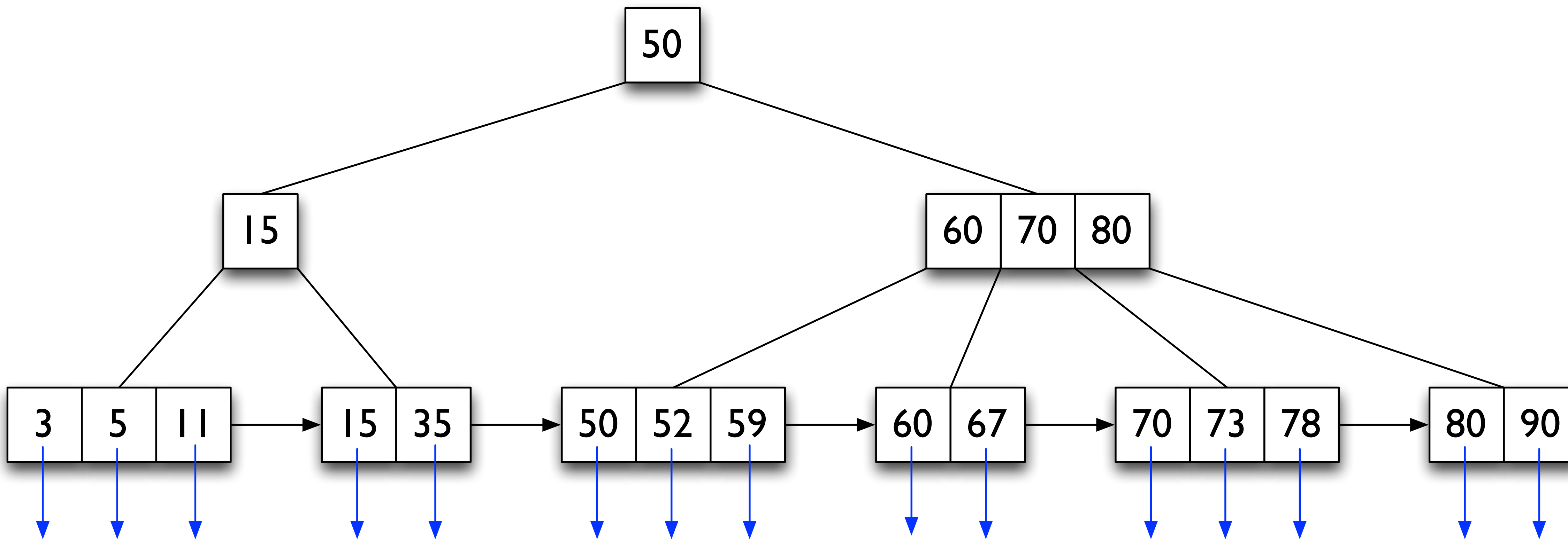
- Avantages :
  - simple
  - ordonné : facilite les parcours linéaires du fichiers de données
- Inconvénients :
  - Index séquentiels à un niveau : le coût de la recherche peut être élevé
  - Index à un ou plusieurs niveaux :
    - les prestations se détériorent avec les blocs en *overflow*
    - réorganisations périodiques nécessaires
- Pour surmonter ces limites :
  - structures de dictionnaires plus efficaces : arbres B^+, hachage
  - adaptation à la mémoire secondaire : structures qui se réorganisaient automatiquement à chaque mise à jour

Arbres B$^+$: introduction



- Structure arborescente (développe l'idée de l'index multi-niveau)
  - un noeud de l'arbre = un bloc du disque
  - les noeuds feuille contiennent les couples <*clef, pointeur*> de l'index
  - les noeuds internes contiennent des doublons des clefs (appelé *balises*) pour orienter la recherche
  - tous les noeuds feuille sont liés en liste chainé, de gauche à droite (raison : permettre un parcours linéaire de la table par clef de recherche)

Arbres B⁺ : introduction



- Si un noeud (interne ou feuille) contient k clefs, il contient k+1 pointeurs
  - dans un noeud interne : pointeurs aux sous-arbres
  - dans un noeud feuille : pointeurs aux enregistrements et à la prochaine feuille

Soit n l'entier maximal tel qu'un bloc du disque contienne n clefs et n+1 pointeurs

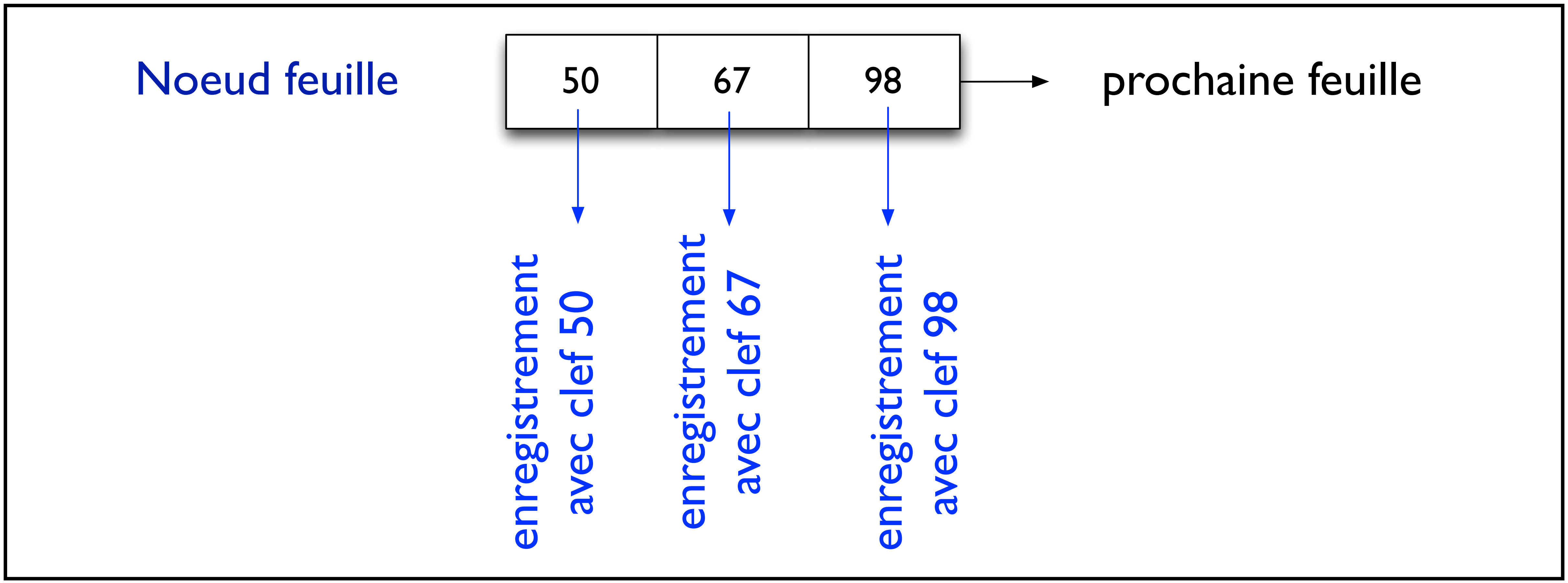
⇒ n est le nombre maximal de clefs dans un noeud de l'arbre

Arbre B+ : définition

Un arbre B+ est un arbre enraciné avec les propriétés suivantes :

1) les feuilles

- ont toutes la même profondeur,
- contiennent des couples <*clef, pointeur*>, triées par clef, et un pointeur à la prochaine feuille



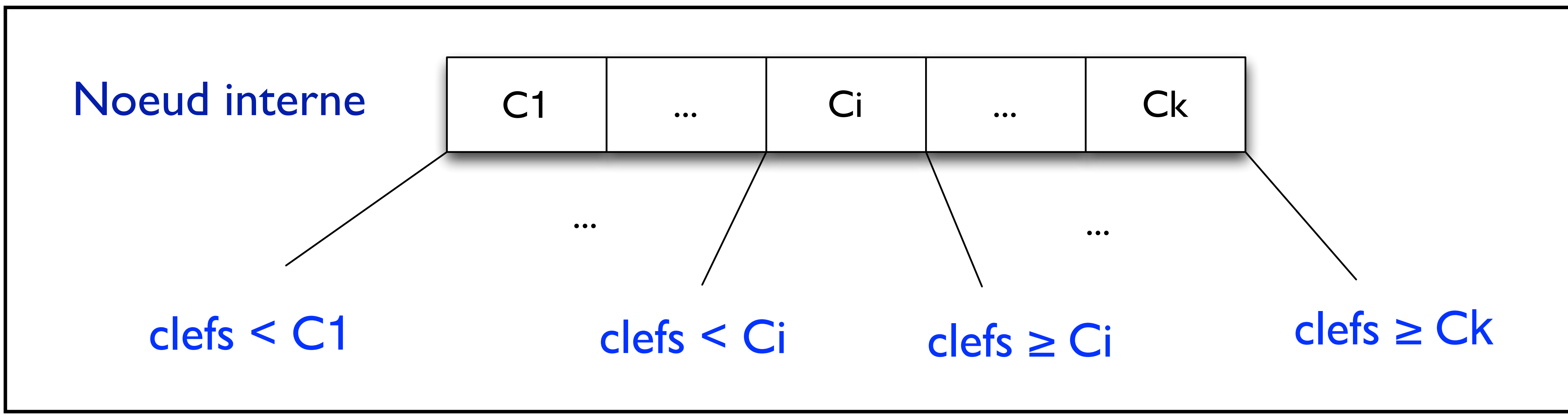
- contiennent entre $\lceil n/2 \rceil$ et n clefs (n défini comme au slide précédent)

Arbre B+ : définition

Un arbre B+ est un arbre enraciné avec les propriétés suivantes :

2) les noeuds internes contiennent

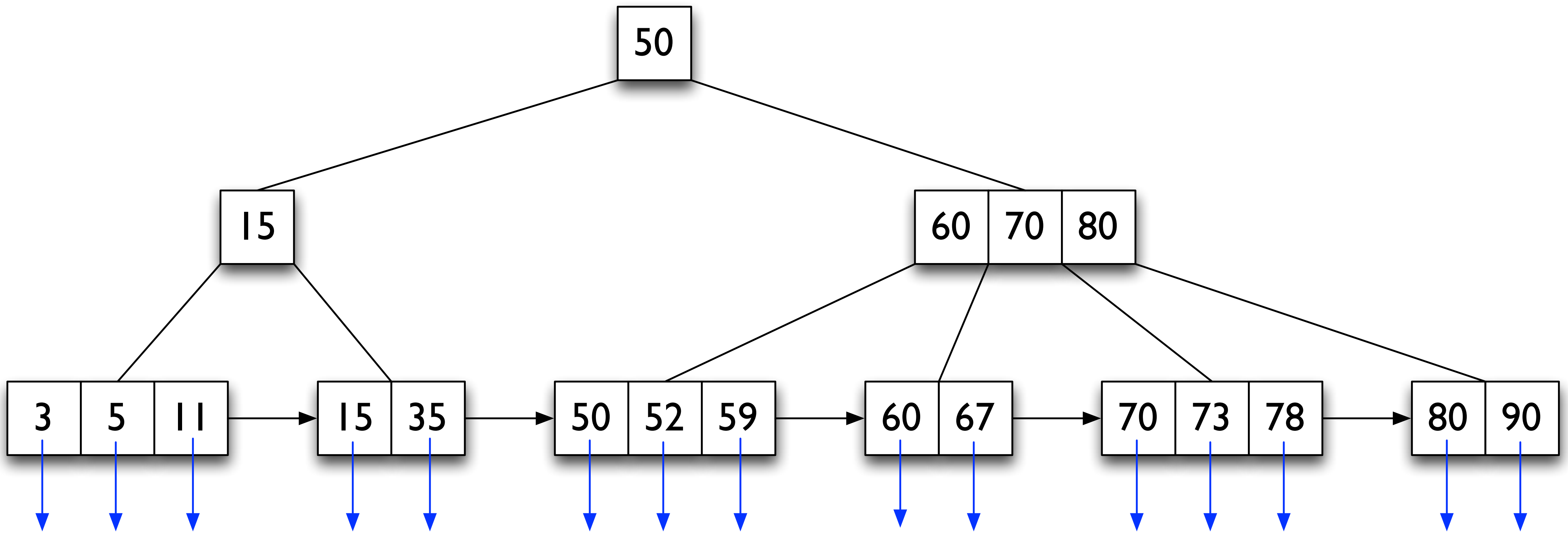
- une suite de clefs triées par ordre croissant
- un nombre de pointeurs à sous-arbres = nombre de clefs + 1,
  avec la propriété suivante pour chaque clef Ci:



- tous noeuds internes sauf la racine : entre $\lfloor n/2 \rfloor$ et n clefs
- la racine : entre 1 et n clefs

Arbres B$^+$: exemple

- Exemple avec n= 3



- noeuds feuilles : entre 2 et 3 clefs
- noeuds internes sauf la racine : entre 1 et 3 clefs
- la racine : entre 1 et 3 clefs

Arbres B^+ : bornes

- But des bornes sur le nombre de clefs:
  - borne supérieure : pour pouvoir stocker un noeud dans un bloc
  - borne inférieure :
    - pour la procédure de rééquilibrage de l'arbre après insertion / suppression (voir plus loin)
    - pour éviter des blocs trop vides (les blocs sont au moins à moitié pleins)
- On dénote min le nombre minimal de clefs dans un noeud,
- min a une valeur différente sur les feuilles, les noeuds internes et la racine
- Rappel : n dénote le nombre maximal de clefs dans tous les noeuds

Arbres B^+ : opérations

- **On suppose absence de clefs doublons dans l'index**
  - Exemples :
    - index sur une clef primaire de la table
    - index primaire dense

- En présence de doublons : une version légèrement plus complexe des arbres B+ (pas abordé)

Arbres B^+ : recherche

- Recherche d'une clef c dans l'arbre: descente dans la profondeur de l'arbre

B := le bloc racine

Tant que le noeud courant B n'est pas une feuille

trouver l'enfant i de B tel que

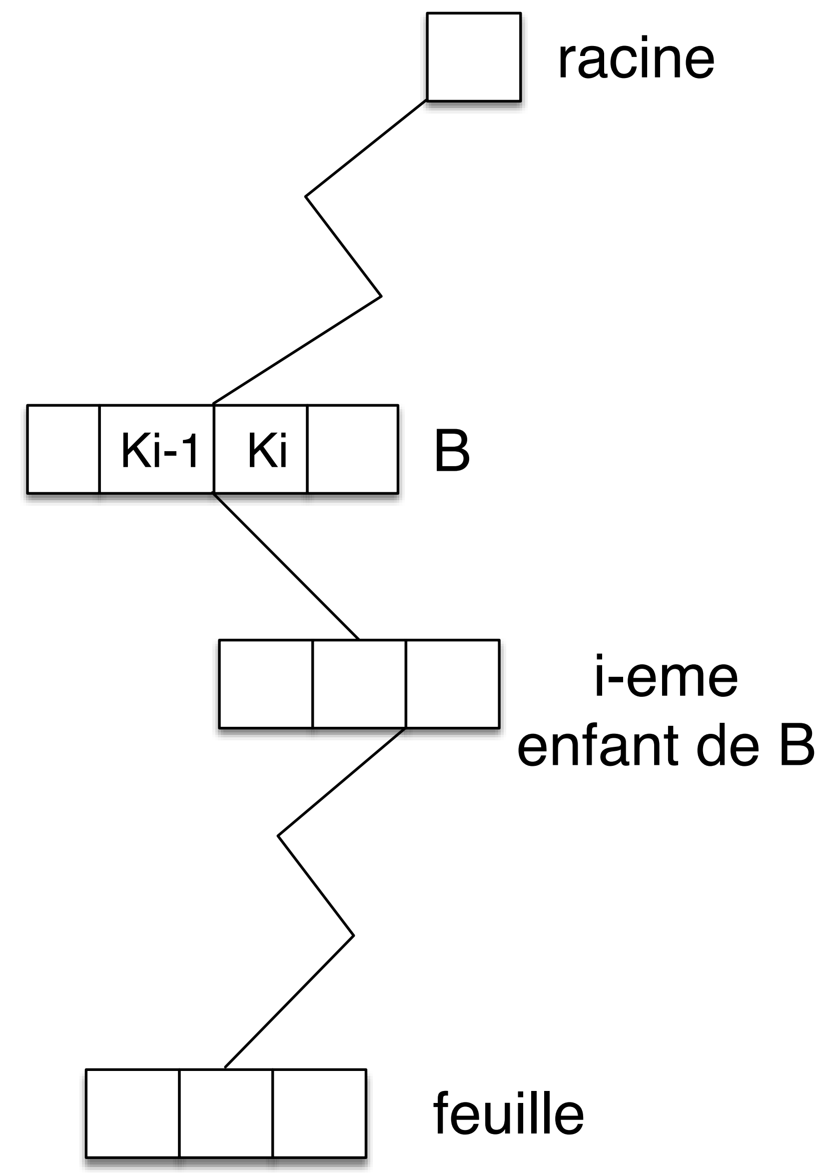
$K_{i-1} \leq c < K_i$ ou

(i = premier enfant, si c < première clef *de B*
et i = dernier enfant si c ≥ dernière clef *de B)*

B := i-eme enfant de B

Fin tan que

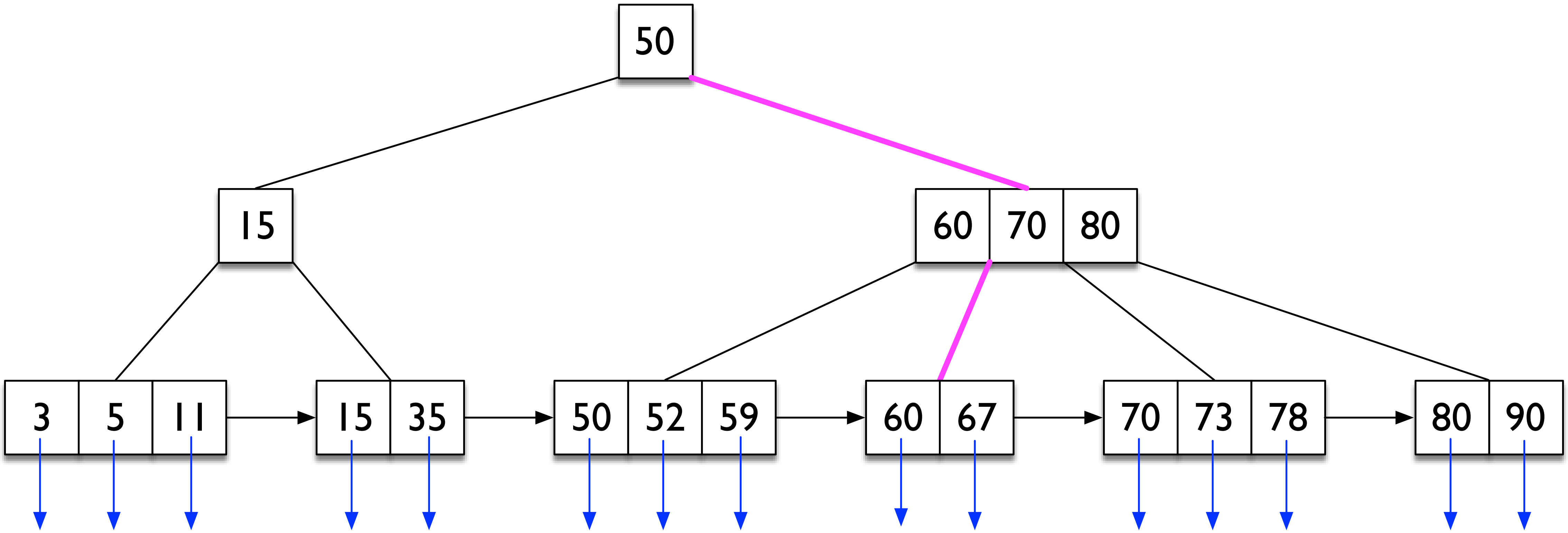
chercher c dans la feuille B



- **Coût:** O(hauteur)

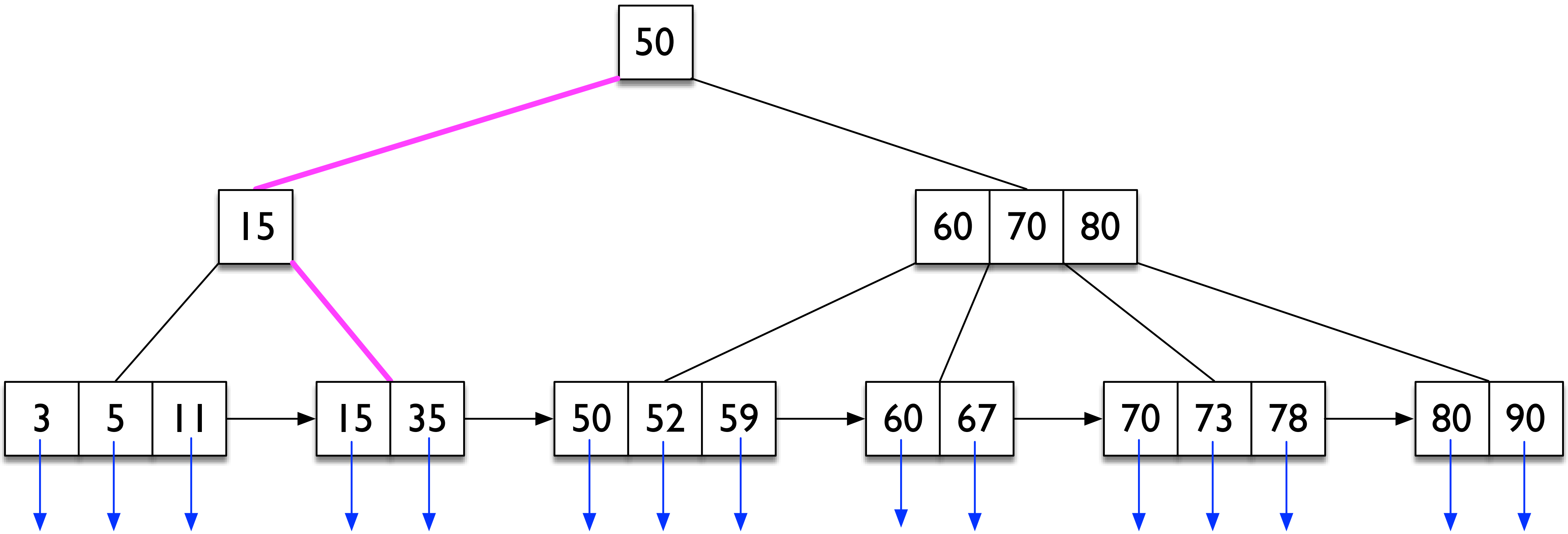
Arbres B$^+$: recherche

- Exemple c = 67



Arbres B$^+$: recherche

- Exemple c = 49



Arbres B$^+$: insertion

Insertion de <c, p> (c absente de l'index)

1. Recherche de c dans l'arbre

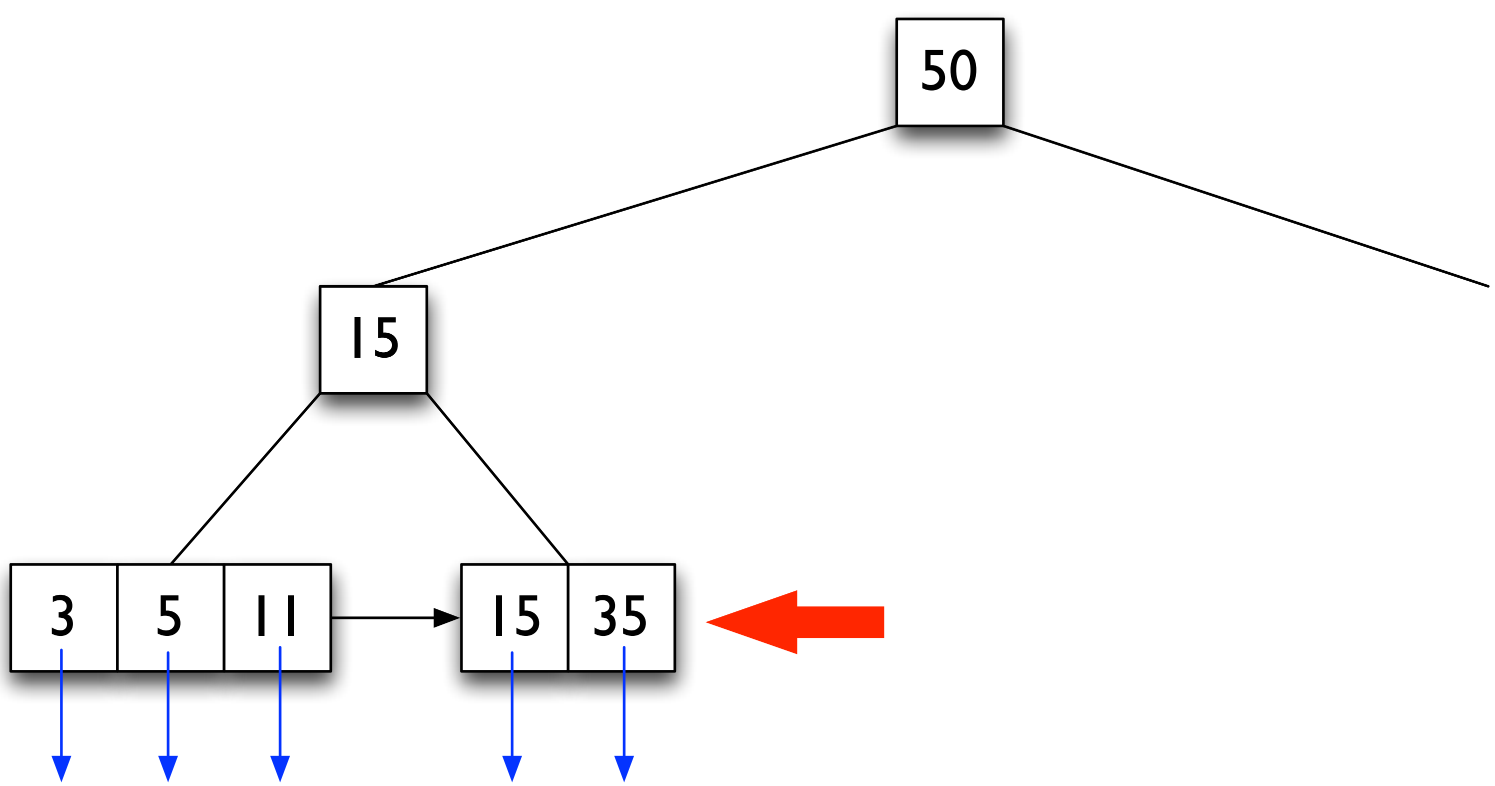
   ⇒ individuation de la feuille B où c devrait être insérée

2. S'il y a de la place dans B (moins de n clefs)

   insertion de <c, p> dans B (en préservant l'ordre)

Exemple. Insérer c = 37

n = 3



Arbres B^+ : insertion

Insertion de <c, p> (c absente de l'index)

1. Recherche de c dans l'arbre

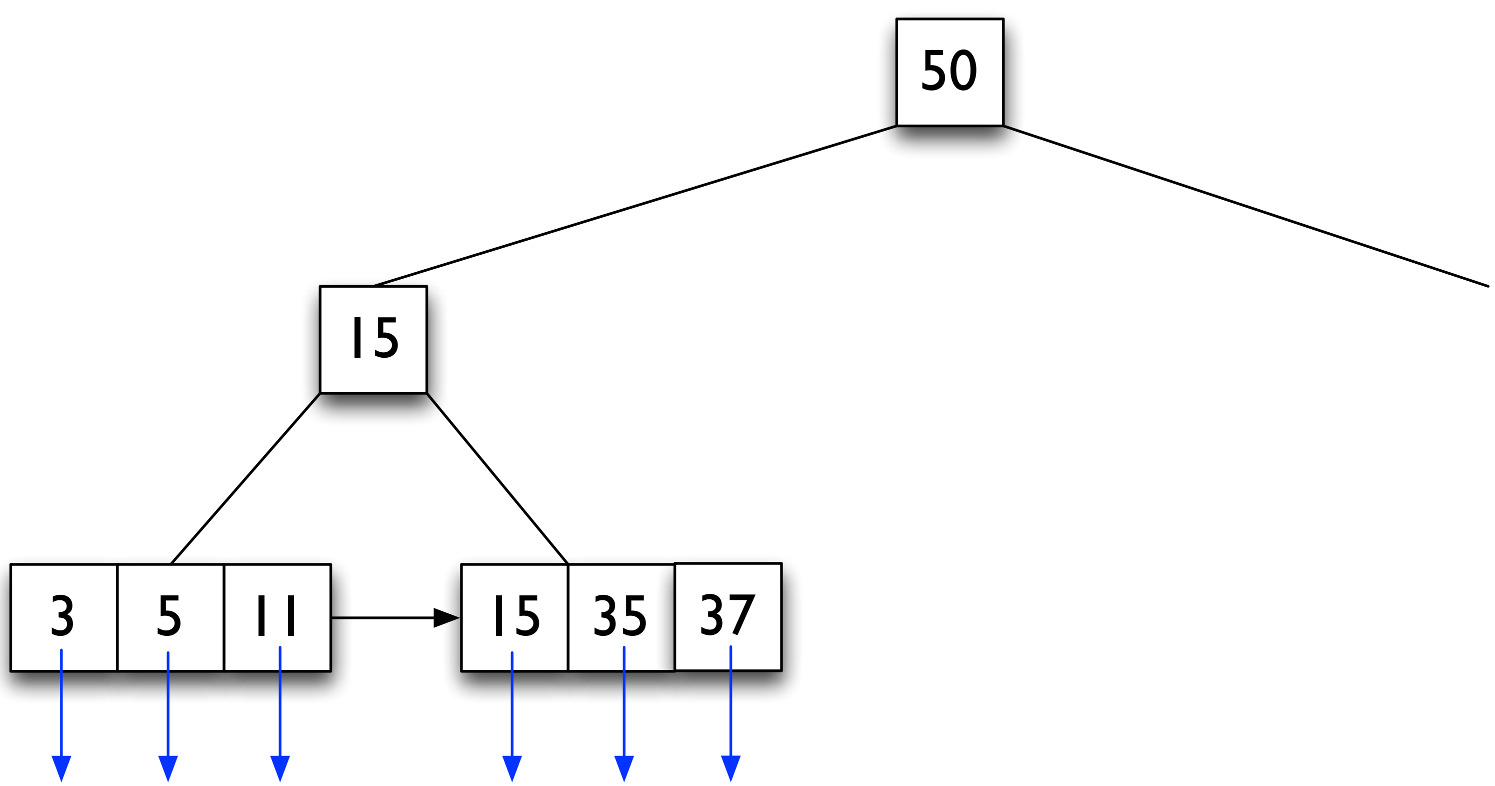
   ⇒ individuation de la feuille B où c devrait être insérée

2. S'il y a de la place dans B (moins de n clefs)

   insertion de <c, p> dans B (en préservant l'ordre)

Exemple. Insérer c = 37

n = 3



Arbres B+ : insertion

Insertion de <c, p> (c absente de l'index)

1. Recherche de c dans l'arbre

   ⇒ individuation de la feuille B où c devrait être insérée

2. S'il y a de la place dans B (moins de n clefs)

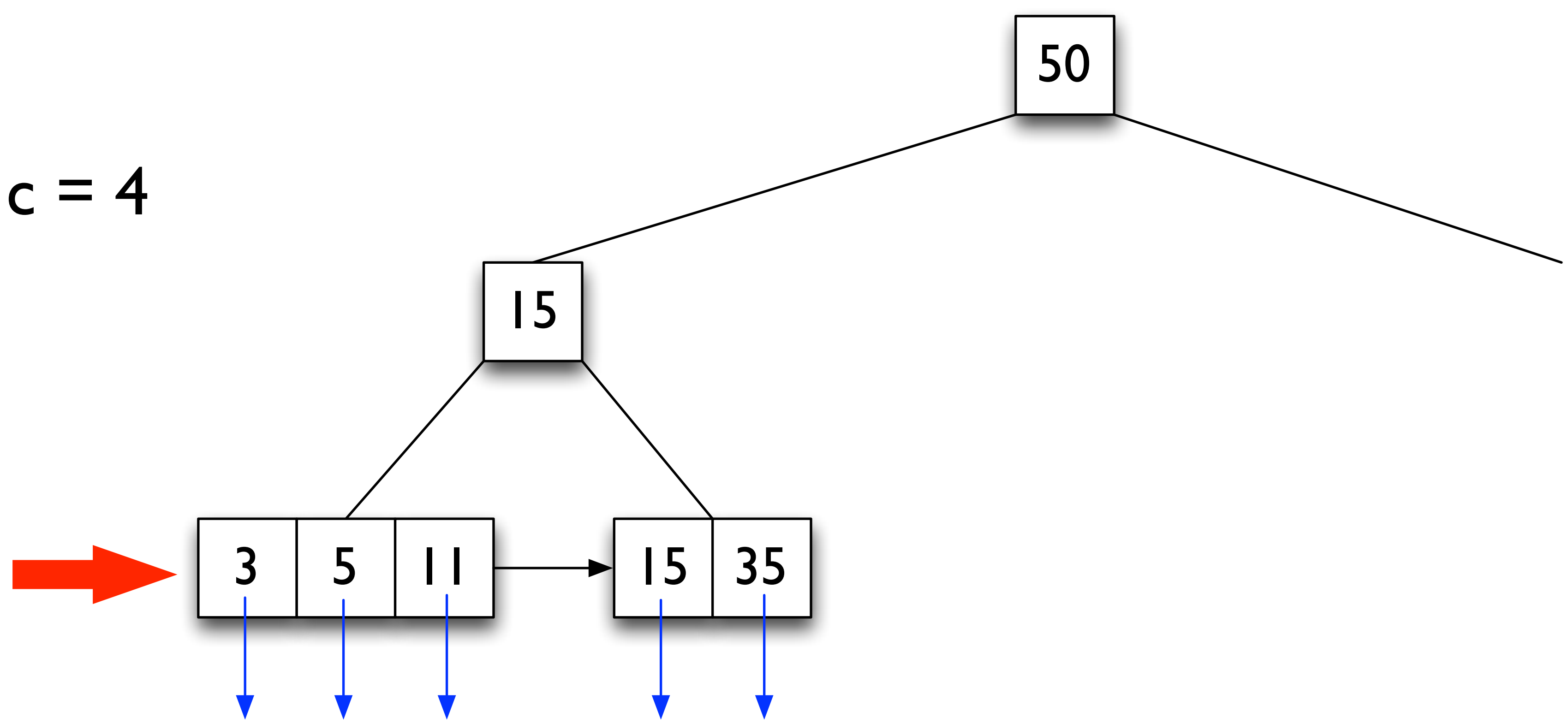
   insertion de <c, p> dans B (en préservant l'ordre)

3. S'il n'y a pas de place dans B (B contient n clefs)

   rééquilibrage par une suite d'**éclatements**

Exemple. Insérer c = 4

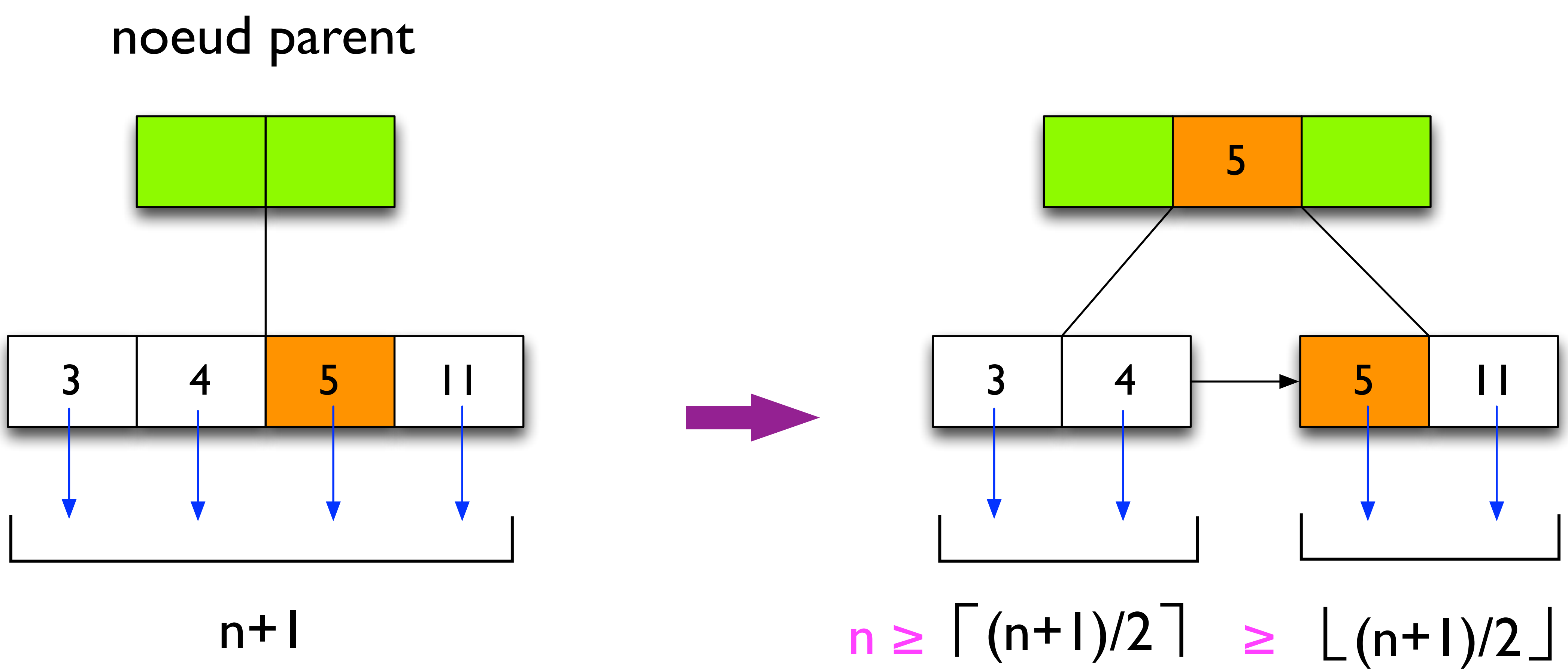
n = 3



Arbres B+ : insertion

Eclatement d'un noeud qui devrait contenir n+1 clefs

a) Noeud feuille (non racine)



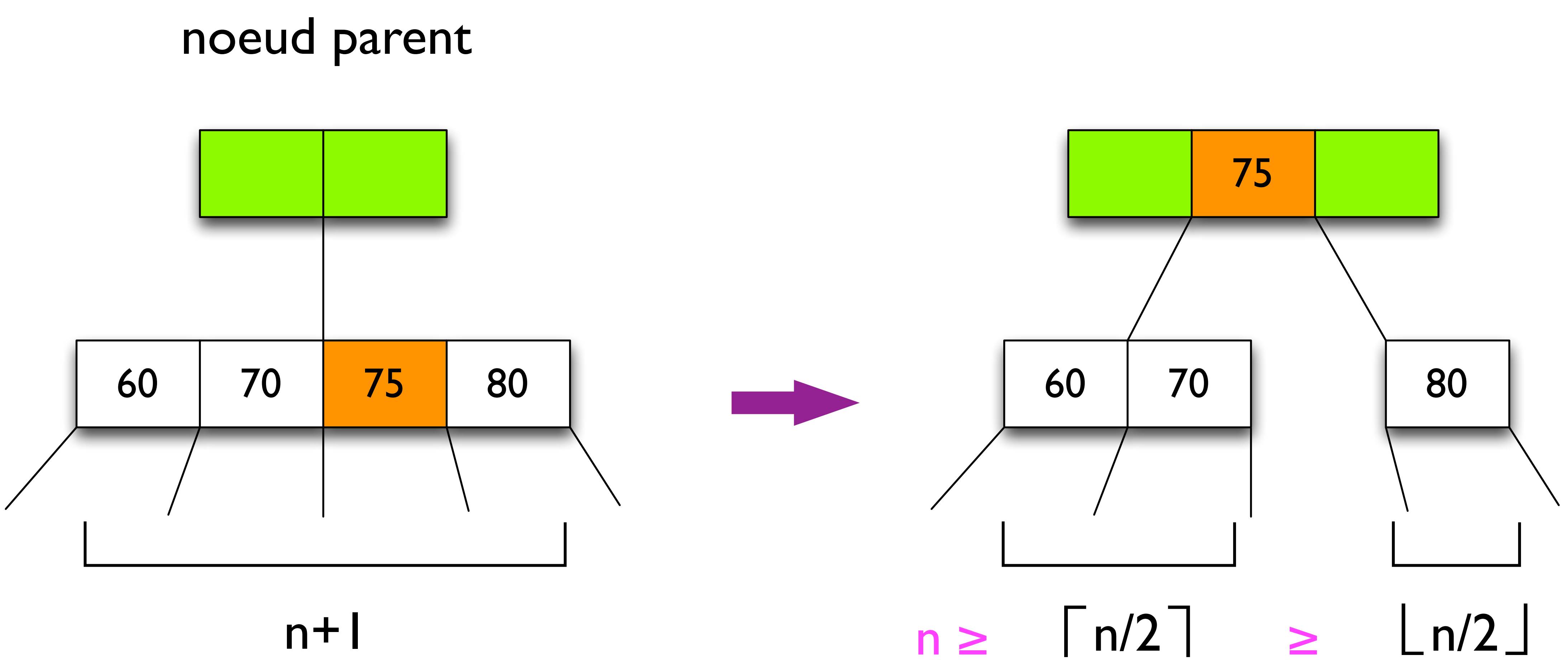
$n \geq \lceil (n+1)/2 \rceil \geq \lfloor (n+1)/2 \rfloor = \lceil n/2 \rceil$

- **Remarque :** les deux nouveaux noeuds respectent les bornes sur le nombre de clefs pour un noeud feuille

Arbres B+ : insertion

Eclatement d'un noeud qui devrait contenir n+1 clefs

b) Noeud interne (non racine)

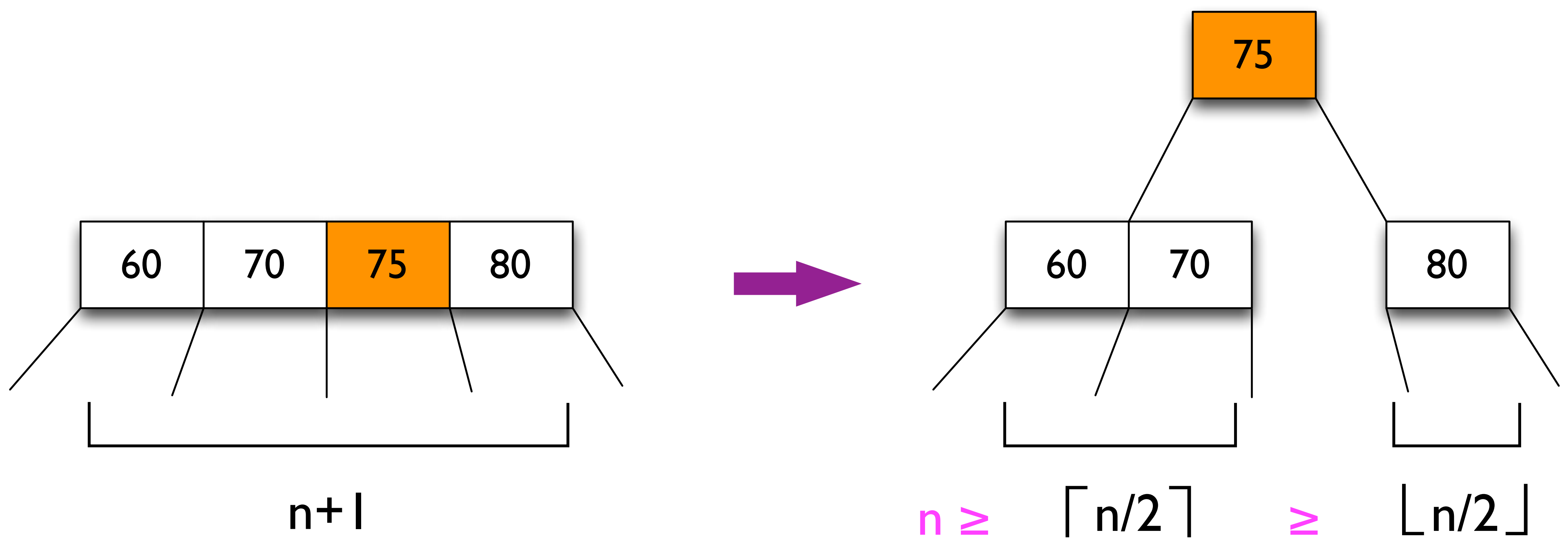


- Remarque : les deux nouveaux noeuds respectent les bornes sur le nombre de clefs pour un noeud interne

Arbres B$^+$: insertion

Eclatement d'un noeud qui devrait contenir n+1 clefs

c) Noeud racine : comme les deux cas précédents (selon que la racine soit noeud interne ou feuille), mais une nouvelle racine avec une seule clef est créée



- Remarque : les **trois** nouveaux noeuds respectent les bornes respectives sur le nombre de clefs

Arbres B^+ : insertion

Rééquilibrage de l'arbre après insertion dans une feuille pleine :

- Eclatement de la feuille pleine
  - $\Rightarrow$ insertion d'une clef dans le noeud parent
- si cela entraine n+1 clefs sur le noeud parent $\Rightarrow$ éclatement du parent
- et ainsi de suite jusqu'à :
  - insertion dans un noeud non-plein ou
  - éclatement de la racine

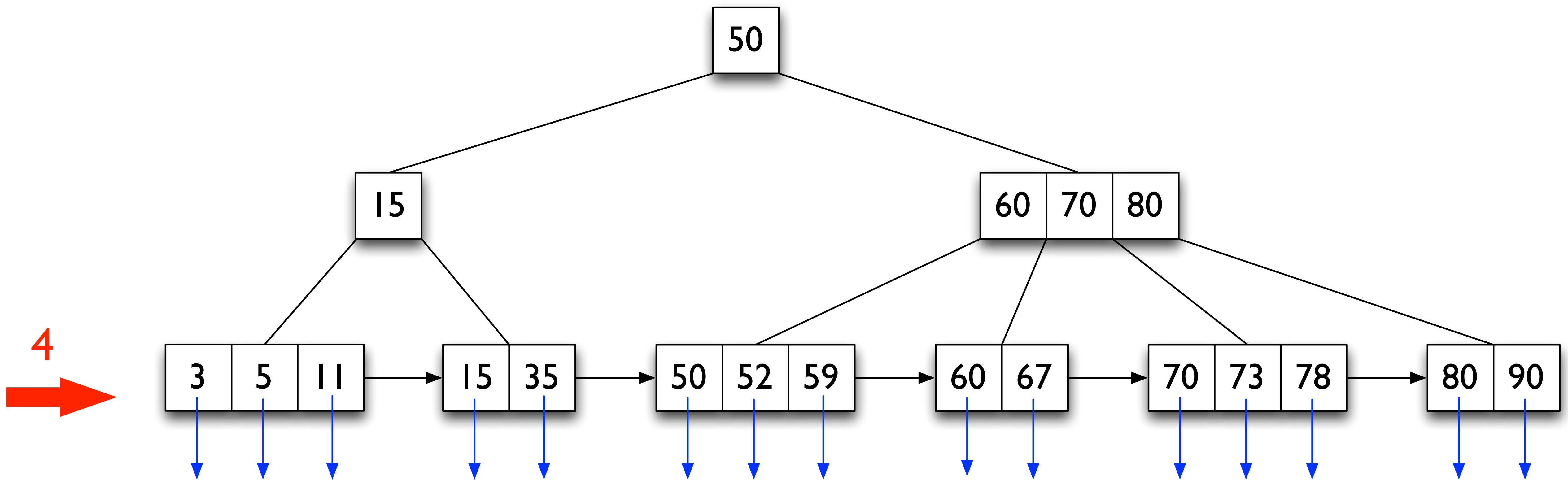
  les deux re-établissent la structure d'arbre B^+

- Coût de l'insertion (recherche + rééquilibrage) : O(hauteur)

Arbres B+ : insertion

Exemple. Insérer c = 4

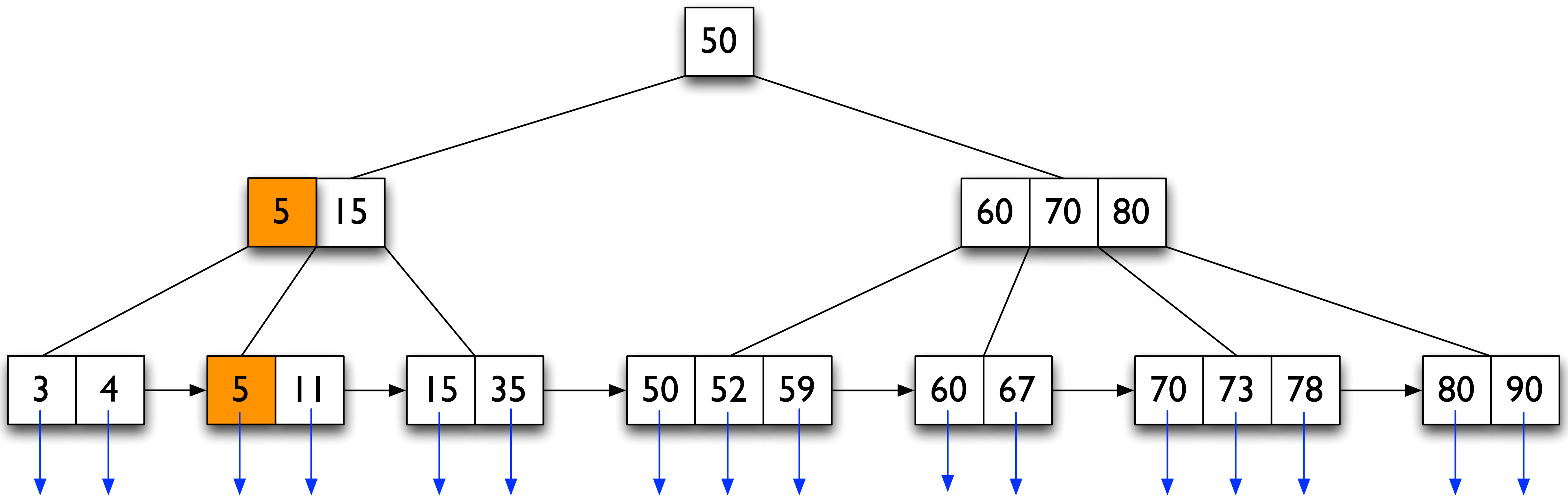
n = 3



Arbres B^+ : insertion

Exemple. Insérer c = 4

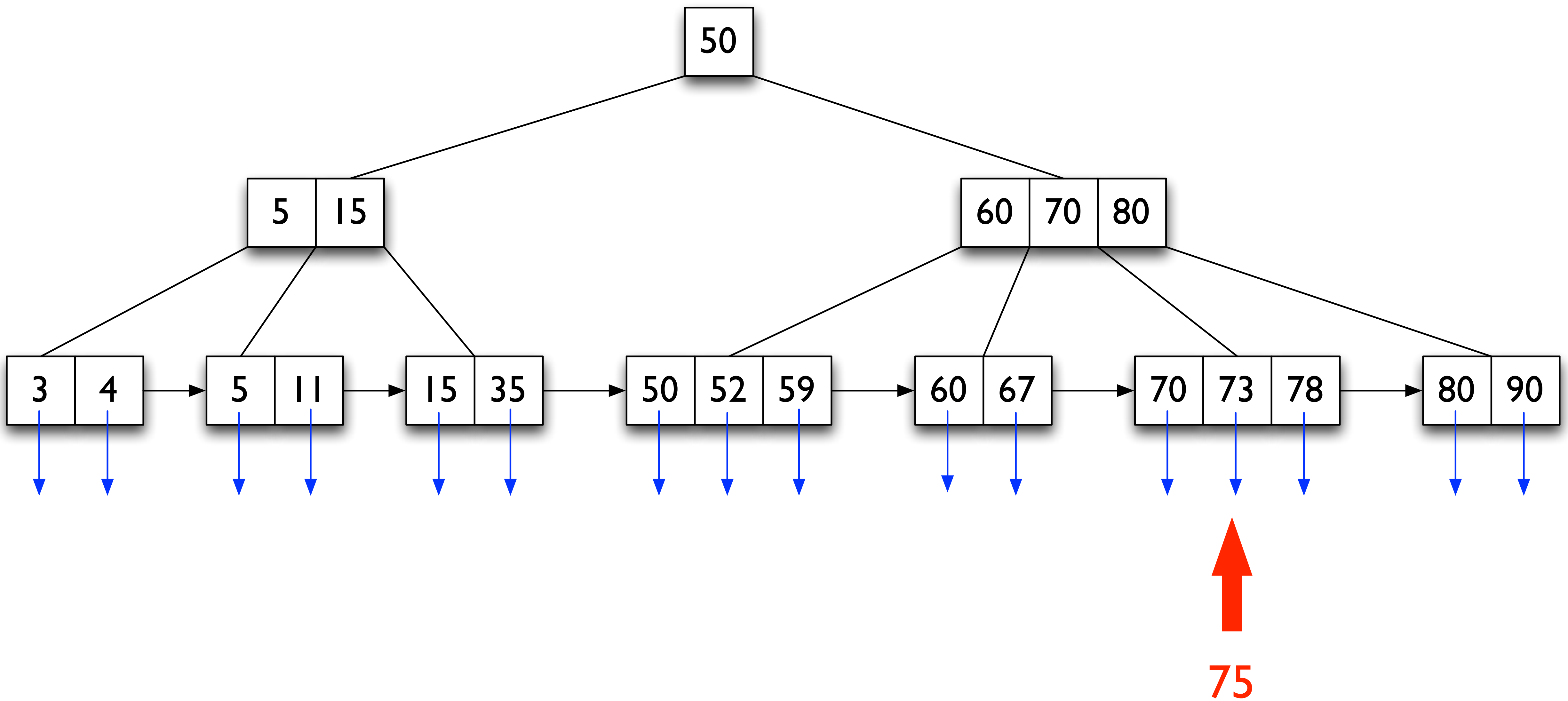
n = 3



Arbres B^+ : insertion

Exemple. Insérer c = 75

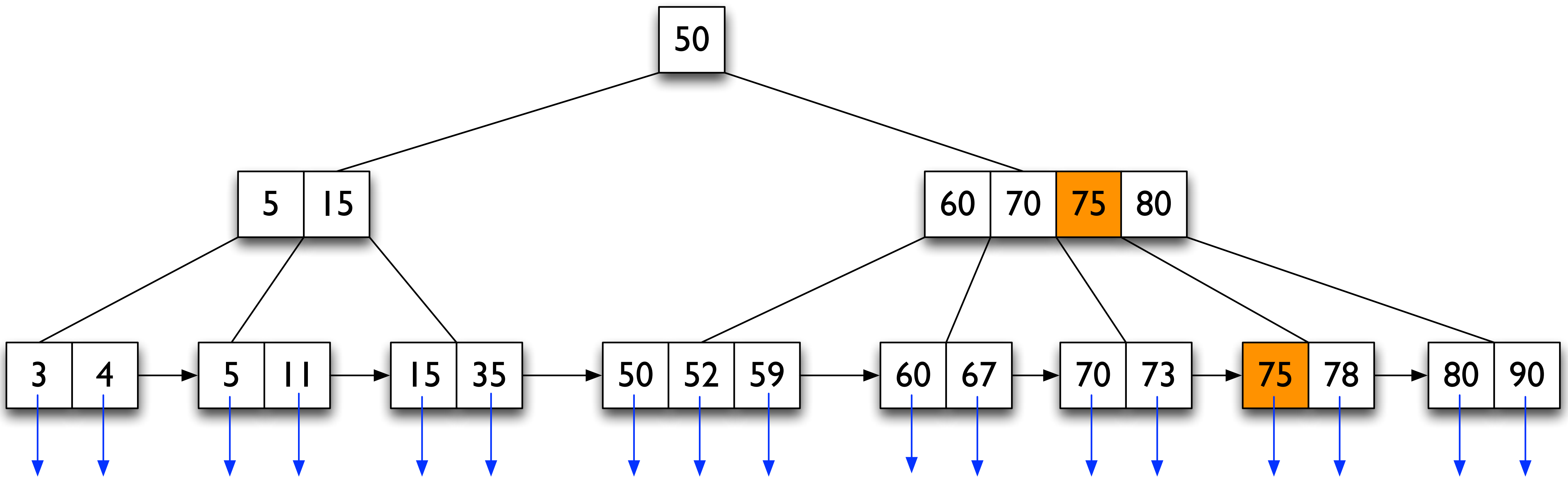
n = 3



Arbres B$^+$: insertion

Exemple. Insérer c = 75

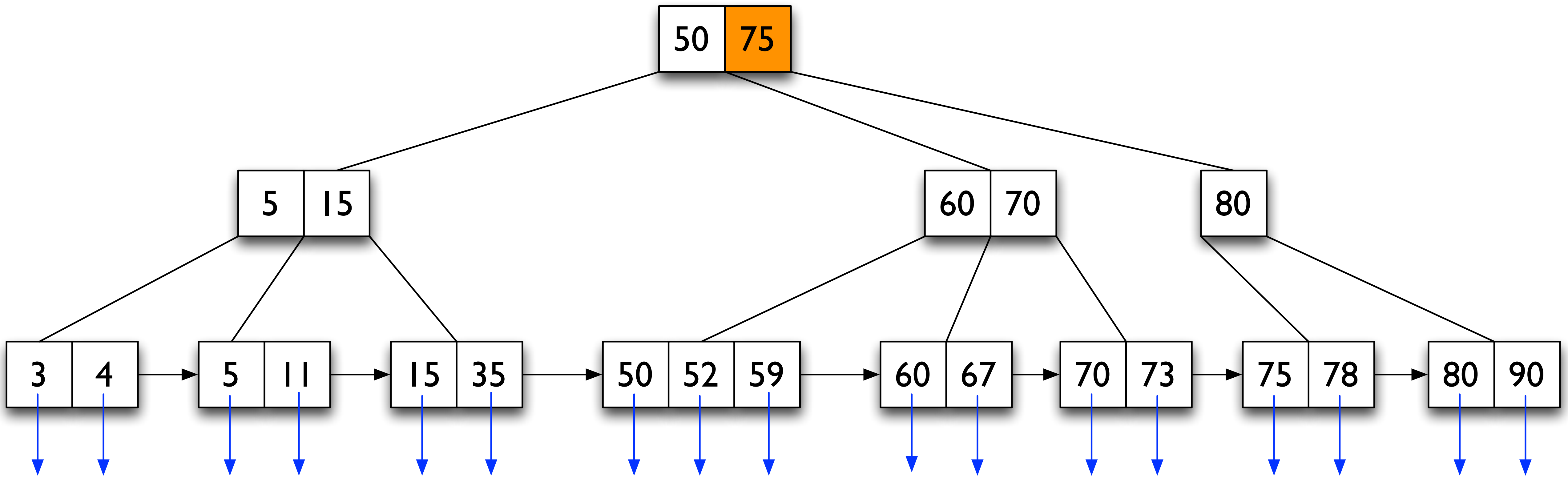
n = 3



Arbres B$^+$: insertion

Exemple. Insérer c = 75

n = 3



Arbres B^+ : suppression

Suppression de <c, p>

1. Recherche de c dans l'arbre
2. suppression de <c, p> de la feuille B la contenant
3. si après suppression, B contient < $\lceil n/2 \rceil$:

   rééquilibrage par une suite de **partages / fusions**

Arbres B^+ : suppression

Rééquilibrage d'un noeud B contenant le nombre minimale de clef - 1

a) B est la racine ⇒ B a zéro clef et un seul fils

- supprimer la racine, la remplacer par son unique fils

b) B n'est pas la racine

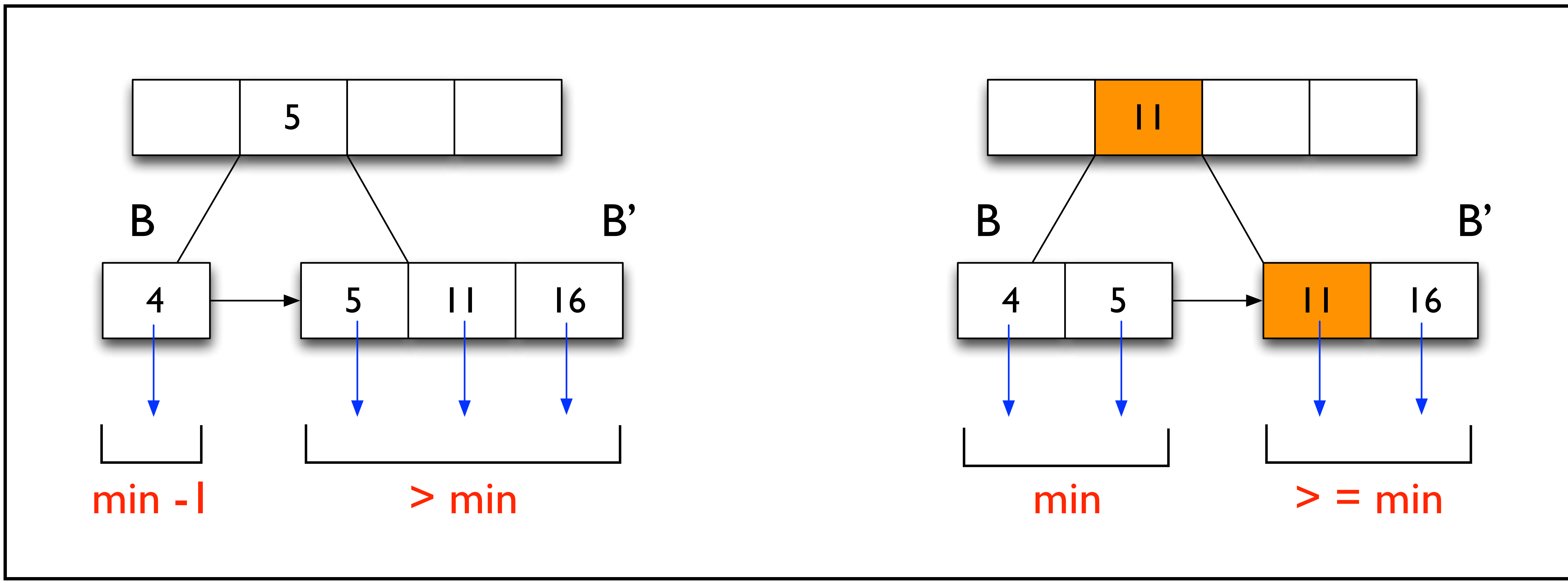
- si B a un frère B' contigu contenant plus que le nombre minimal de clefs ⇒ **partage** entre B et B'
- sinon **fusion** entre B et un frère contigu

Arbres B$^+$: suppression

Partage entre un noeud B à min-1 clef et un frère B' avec > min clefs

1) B est une feuille

- B' donne une clef à B
- la balise entre B et B' dans le noeud parent est mise à jour



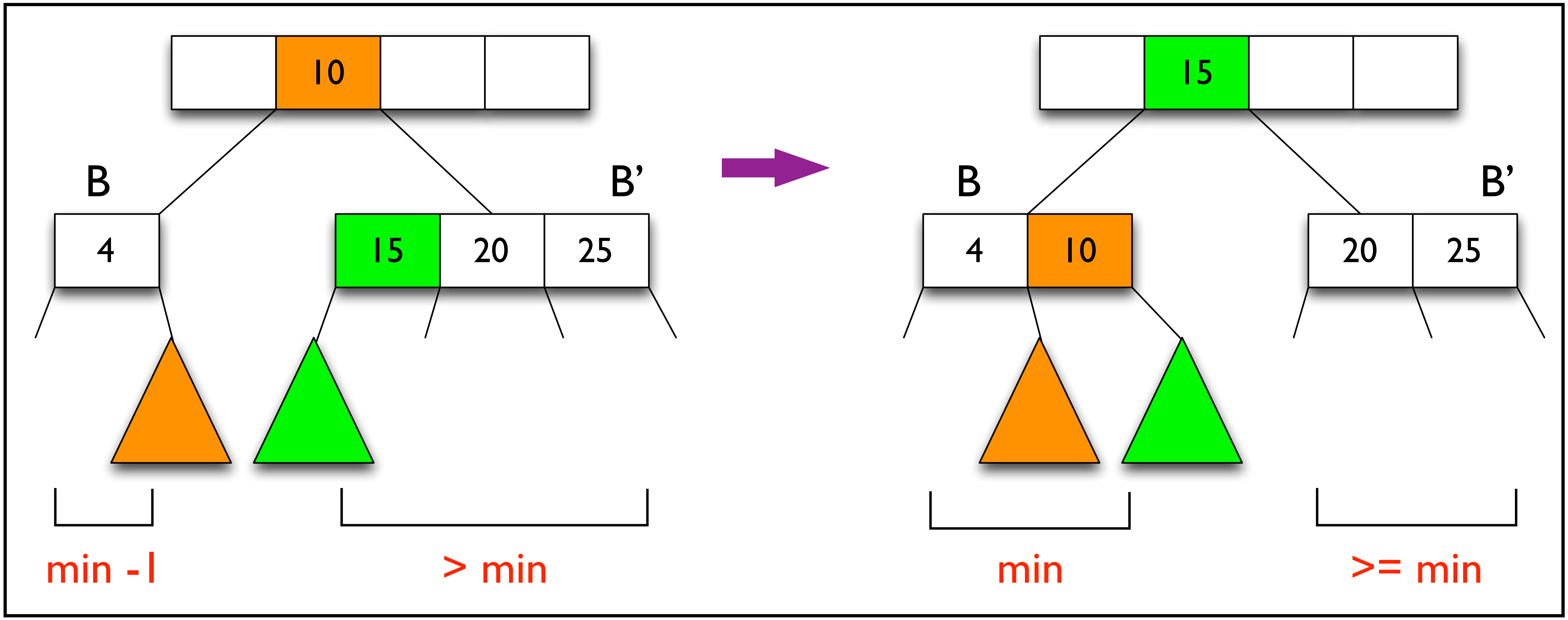
Ex. n = 4 min = $\lceil n/2 \rceil = 2$

Arbres B$^+$: suppression

Partage entre un noeud B à min-1 clef et un frère B' avec > min clefs

2) B est noeud interne

- rotations de clefs avec le noeud parent

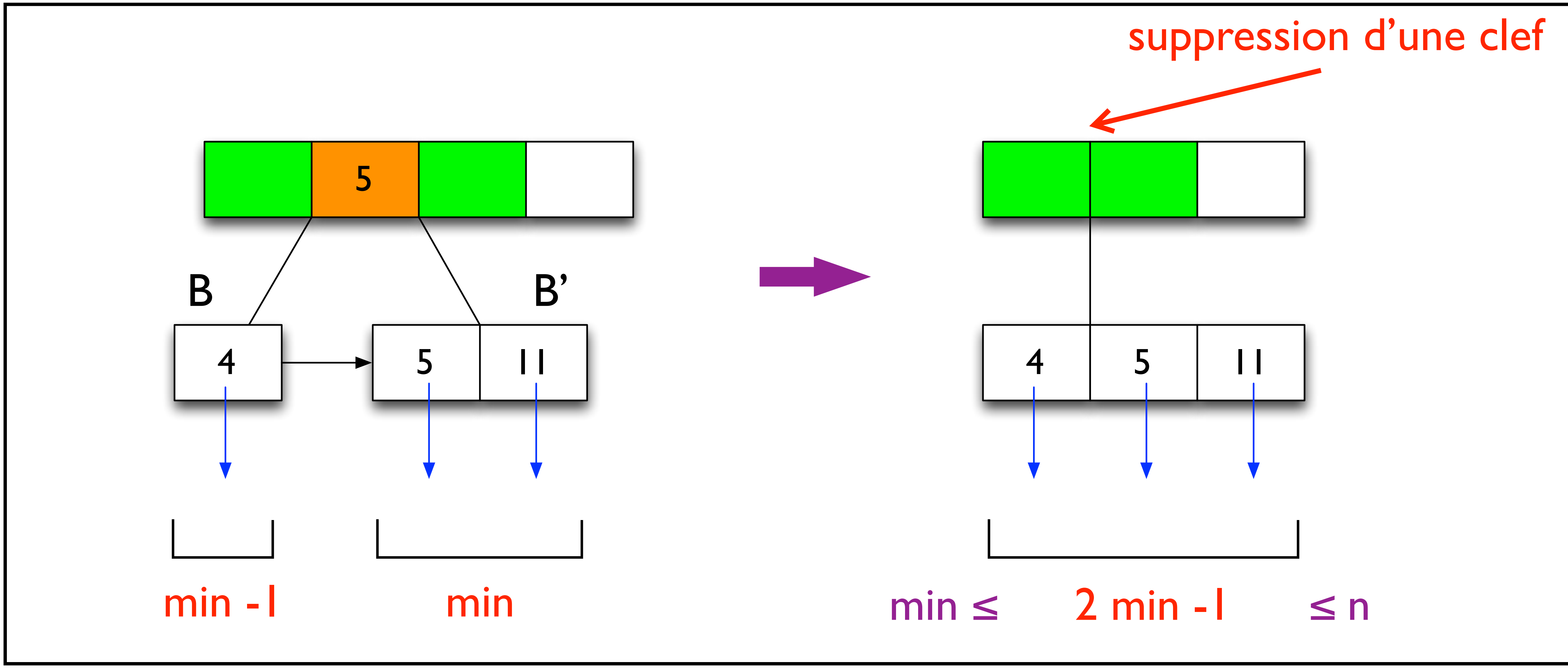


Ex. $n = 4$ $\quad$ $min = \lfloor n/2 \rfloor = 2$

Arbres B$^+$: suppression

Fusion entre un noeud B à min-1 clef et un frère B' avec min clefs

1) B est une feuille



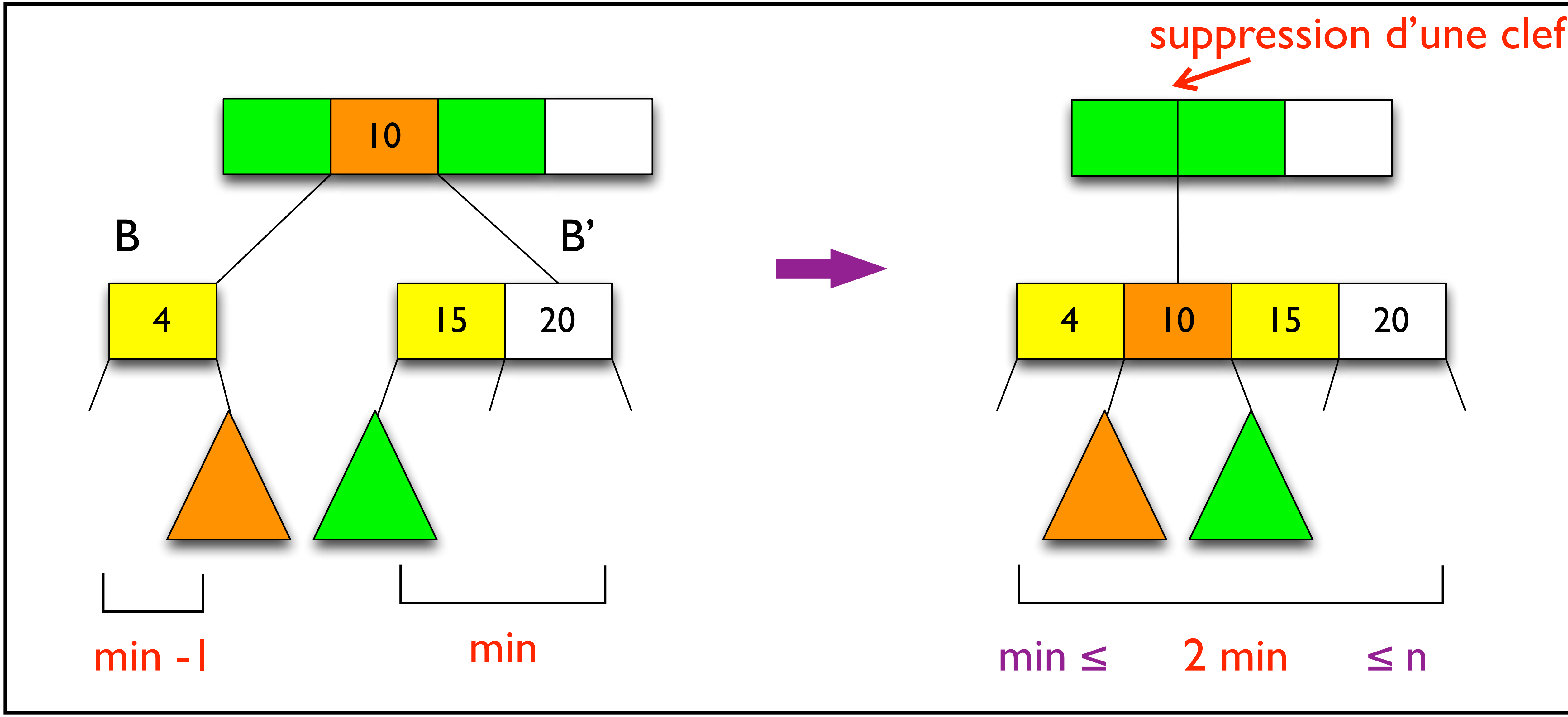
Remarque. $\min = \lceil n/2 \rceil = \lfloor (n+1)/2 \rfloor \leq (n+1)/2 \quad \Rightarrow \quad 2\min - 1 \leq n$

Ex. $n = 4 \quad \min = \lceil n/2 \rceil = 2 \quad 2\min-1 = 3$

Arbres B⁺ : suppression

Fusion entre un noeud B à min-1 clef et un frère B' avec min clefs

2) B est noeud interne



Remarque. $\text{min} = \lfloor n/2 \rfloor \quad \Rightarrow \quad 2\ \text{min} \leq n$

Ex. $n = 4 \qquad \text{min} = \lfloor n/2 \rfloor = 2 \qquad 2\ \text{min} = 4$

Arbres B^+ : suppression

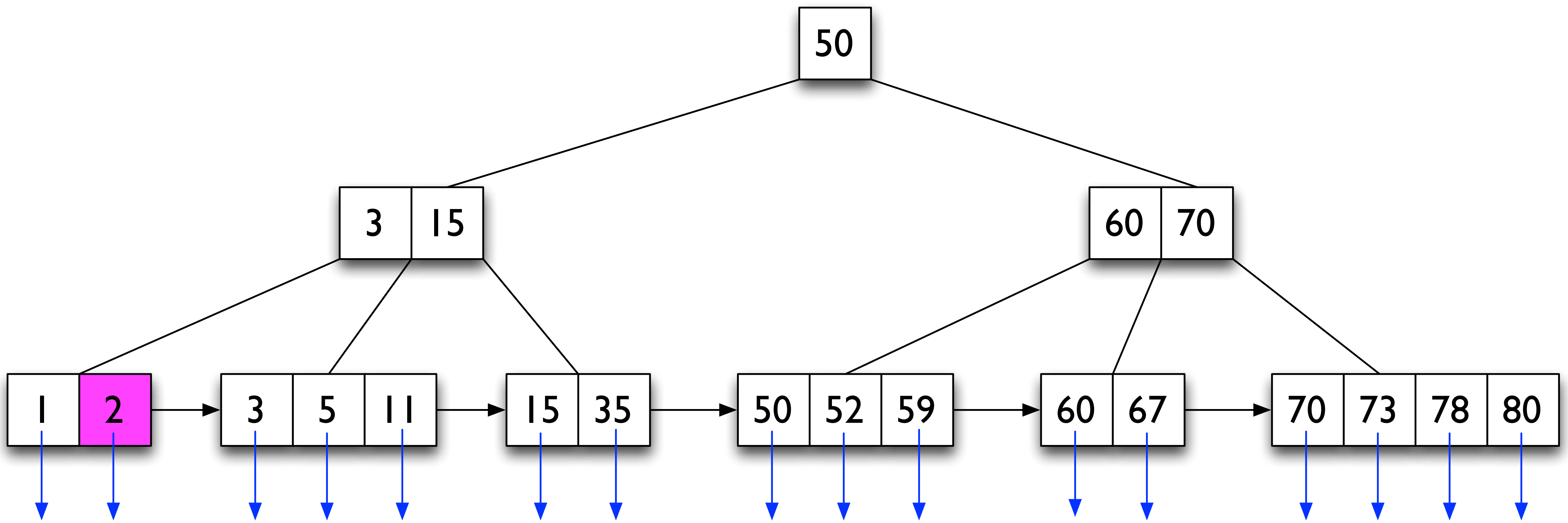
Rééquilibrage de l'arbre après suppression d'une feuille avec min clefs :

- rééquilibrage (partage/ fusion) de la feuille concernée
  - ⇒ **si fusion,** suppression d'une clef dans le noeud parent
- si cela entraine moins que min clefs sur le noeud parent
  - ⇒ rééquilibrage (partage/ fusion) du parent
  - et ainsi de suite jusqu'à :
    - suppression d'une clef d'un noeud contenant plus que min clefs **ou**
    - partage **ou**
    - suppression de la racine
  - les trois re-établissent la structure d'arbre B^+
- Coût de la suppression (recherche + rééquilibrage) : O(hauteur)

Arbres B+ : suppression

- **Exemple.** n=4, min= 1 sur la racine, min =2 sur tous les autres noeuds

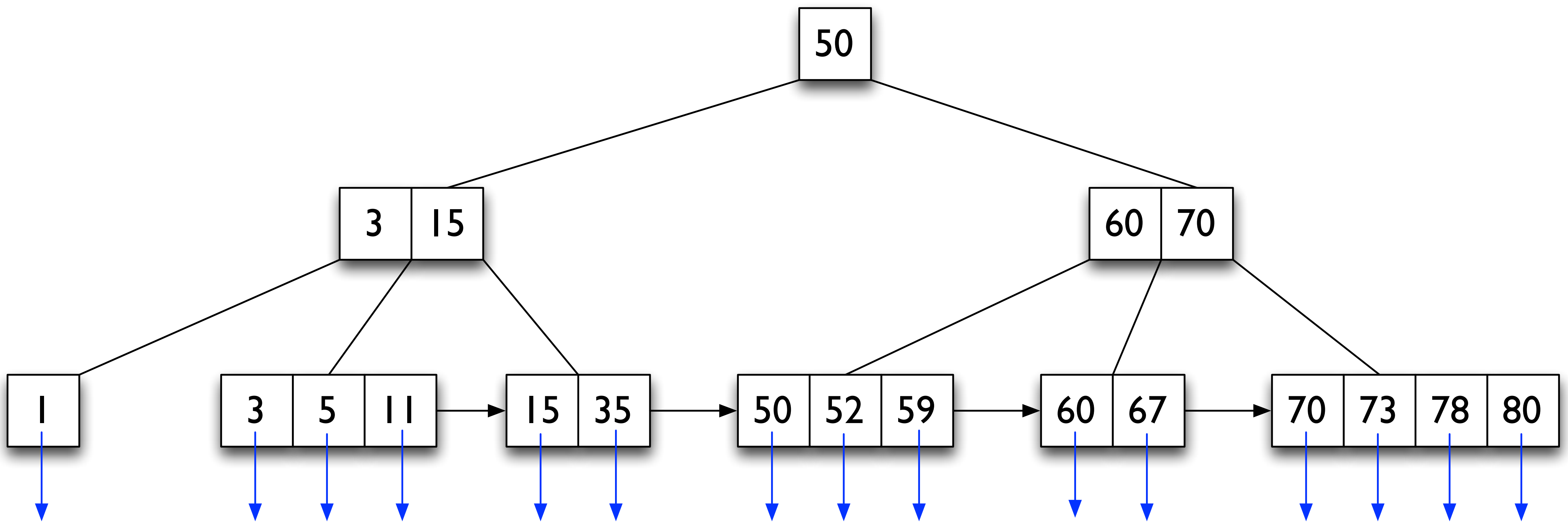
Suppression de 2



Arbres B+ : suppression

- **Exemple.** n=4, min= 1 sur la racine, min =2 sur tous les autres noeuds

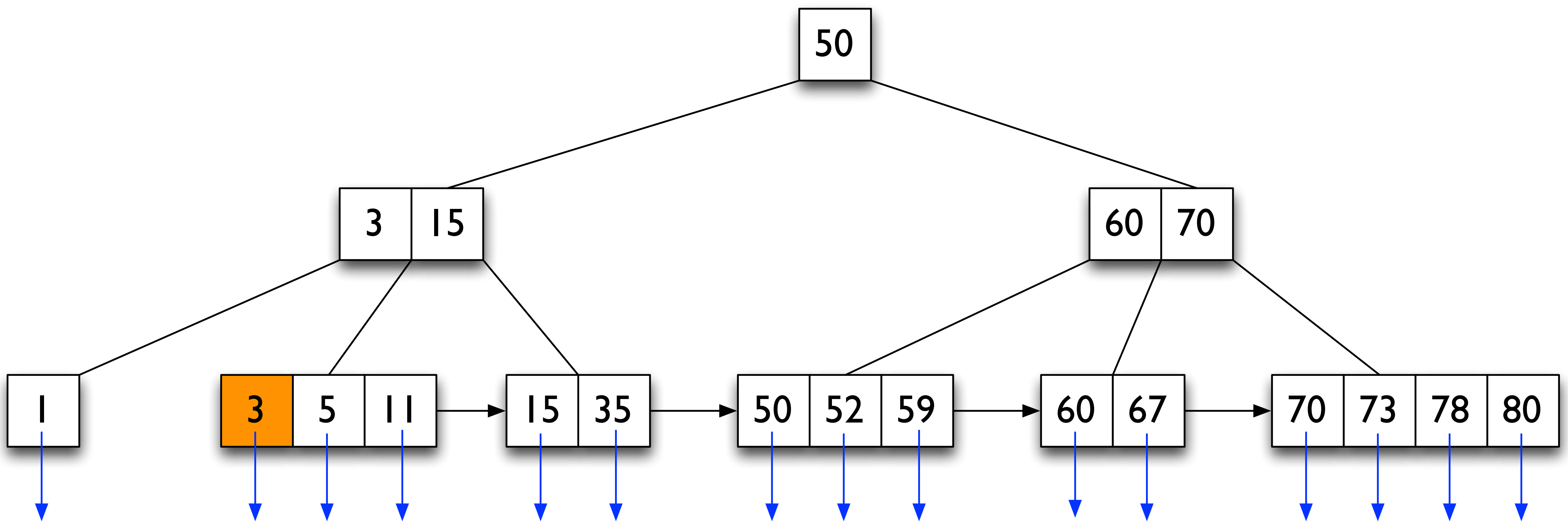
Suppression de 2



Arbres B^{+} : suppression

- **Exemple.** n=4, min= 1 sur la racine, min =2 sur tous les autres noeuds

Suppression de 2 - rééquilibrage

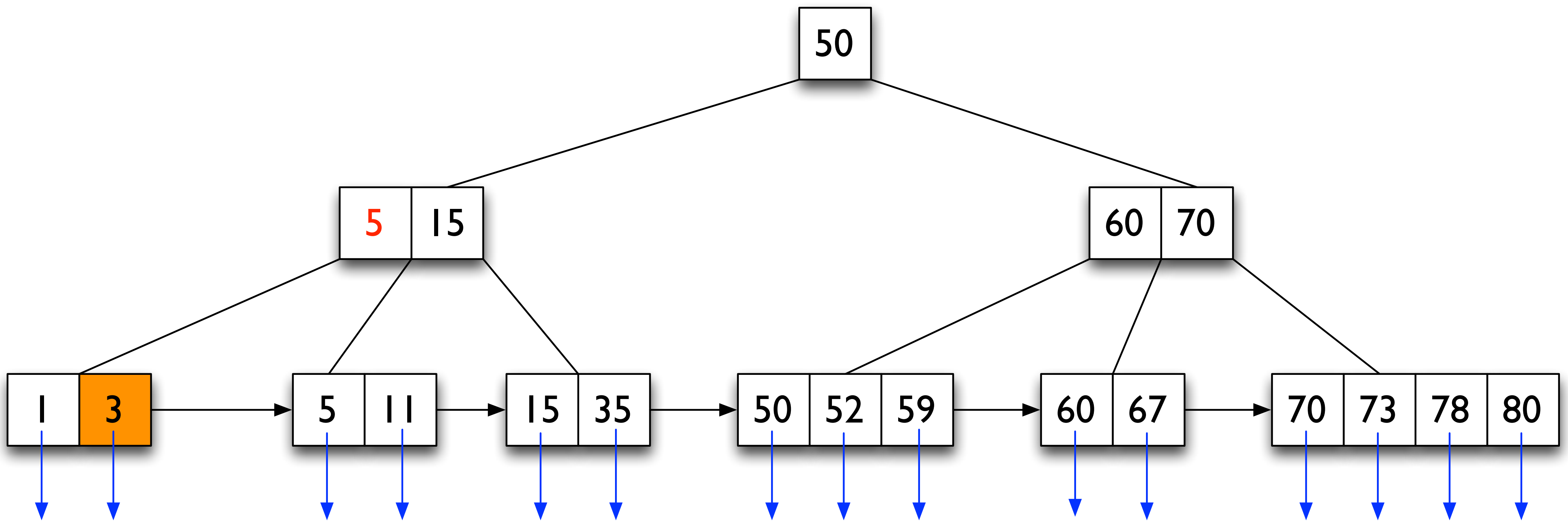


partage

Arbres B+ : suppression

- **Exemple.** n=4, min= 1 sur la racine, min =2 sur tous les autres noeuds

Suppression de 2 - rééquilibrage

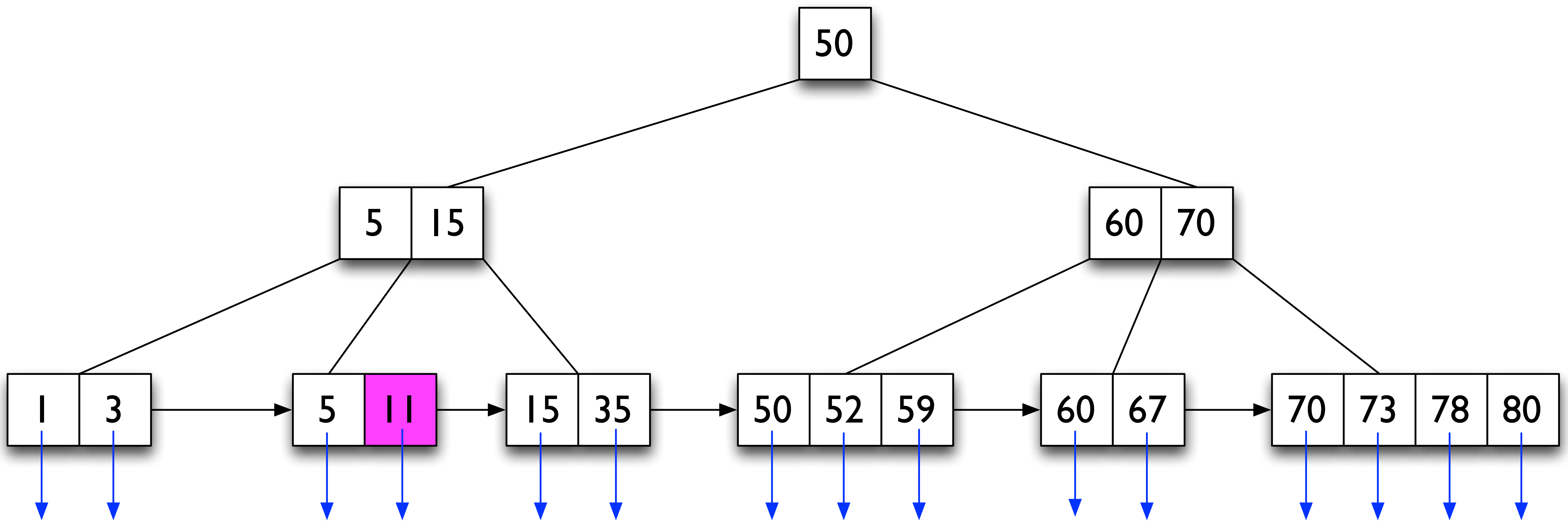


Arbre rééquilibré

Arbres B^+ : suppression

- **Exemple.** n=4, min= 1 sur la racine, min =2 sur tous les autres noeuds

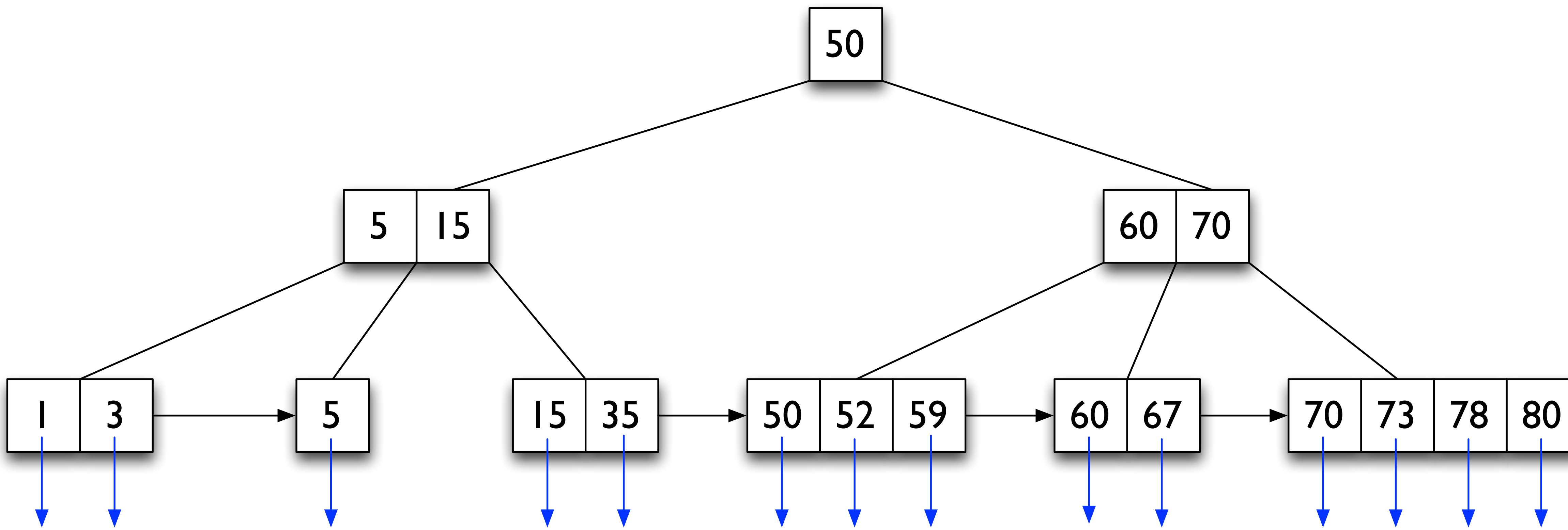
Suppression de 11



Arbres B+ : suppression

- **Exemple.** n=4, min= 1 sur la racine, min =2 sur tous les autres noeuds

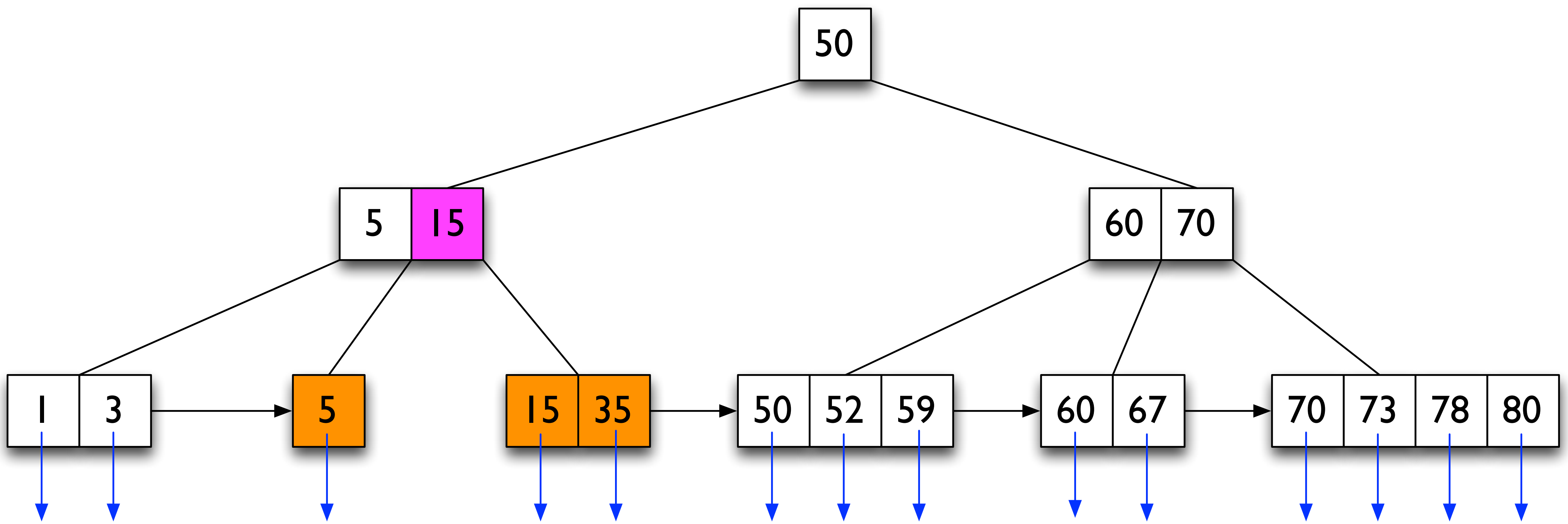
Suppression de 11



Arbres B+ : suppression

- **Exemple.** n=4, min= 1 sur la racine, min =2 sur tous les autres noeuds

Suppression de 11 - rééquilibrage

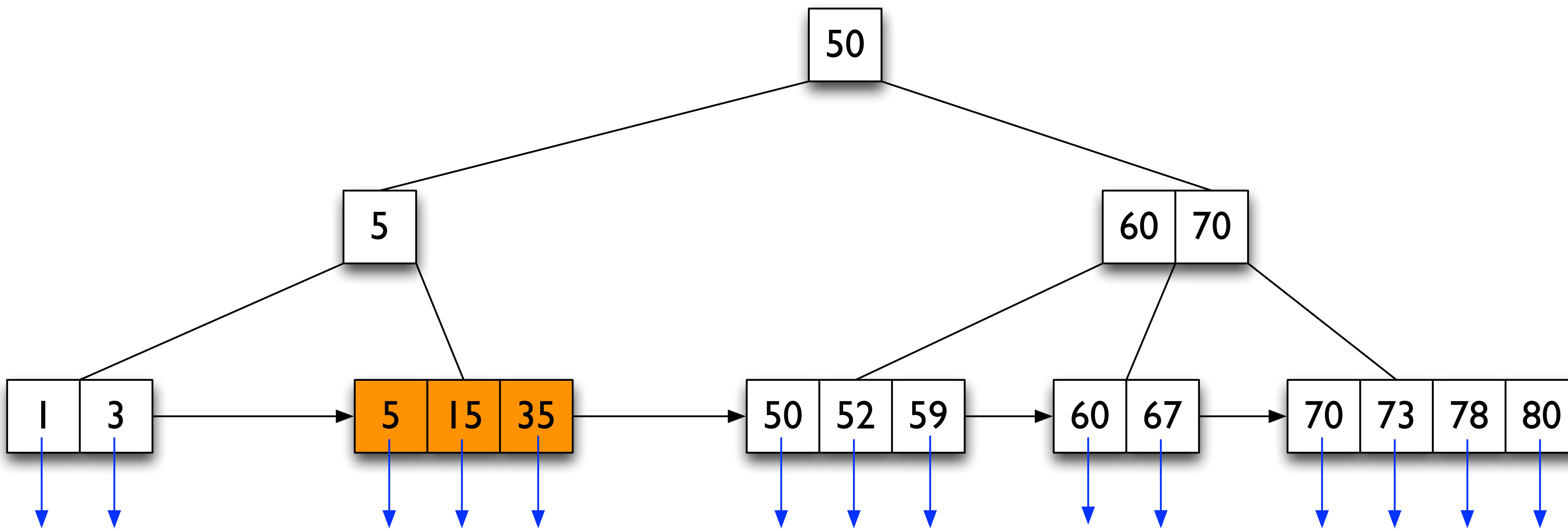


fusion

Arbres B$^+$: suppression

- **Exemple.** n=4, min= 1 sur la racine, min =2 sur tous les autres noeuds

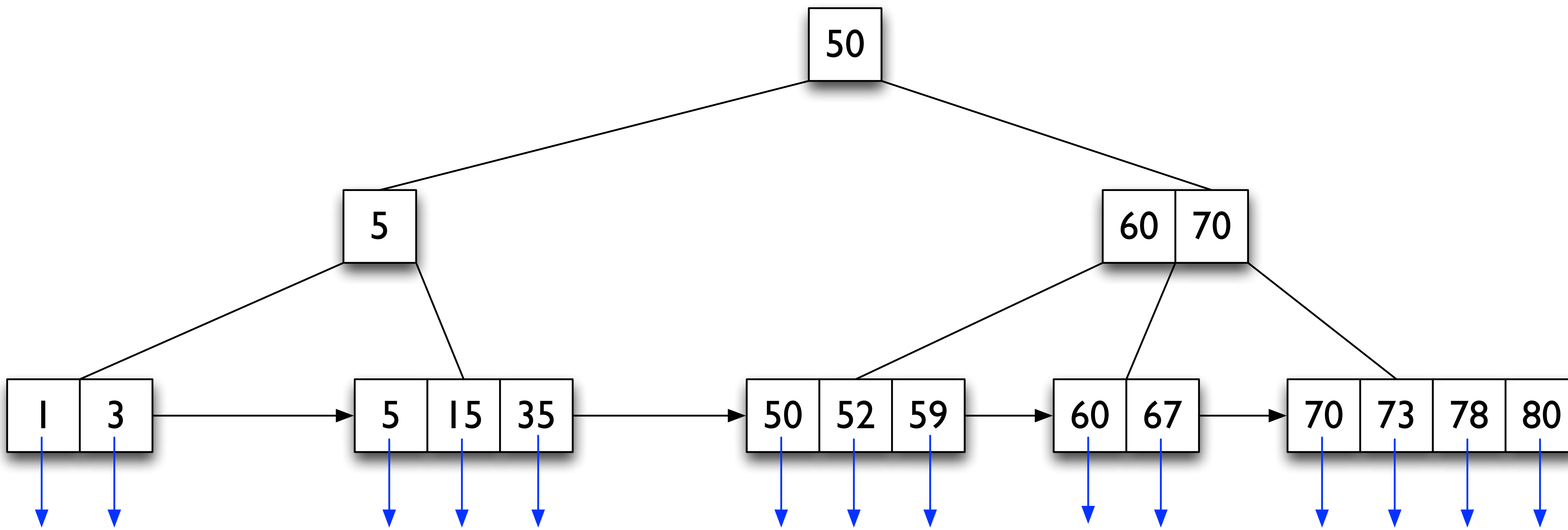
Suppression de 11 - rééquilibrage



Arbres B$^+$: suppression

- **Exemple.** n=4, min= 1 sur la racine, min =2 sur tous les autres noeuds

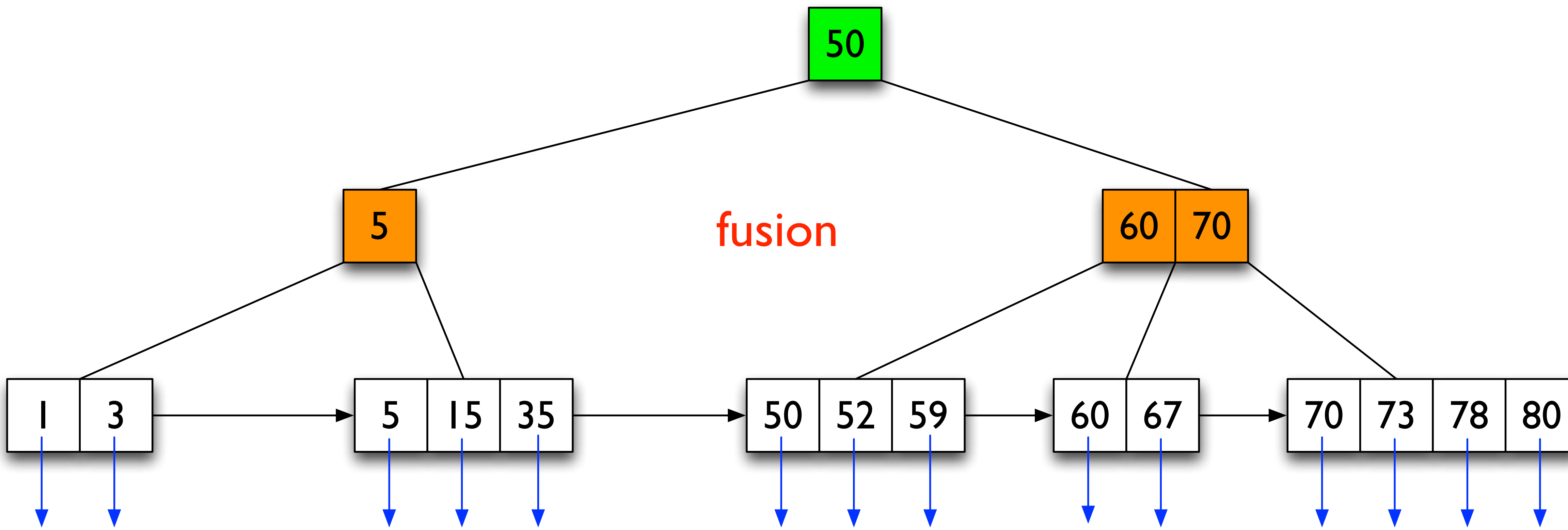
Suppression de 11 - rééquilibrage



Arbres B$^+$: suppression

- **Exemple.** n=4, min= 1 sur la racine, min =2 sur tous les autres noeuds

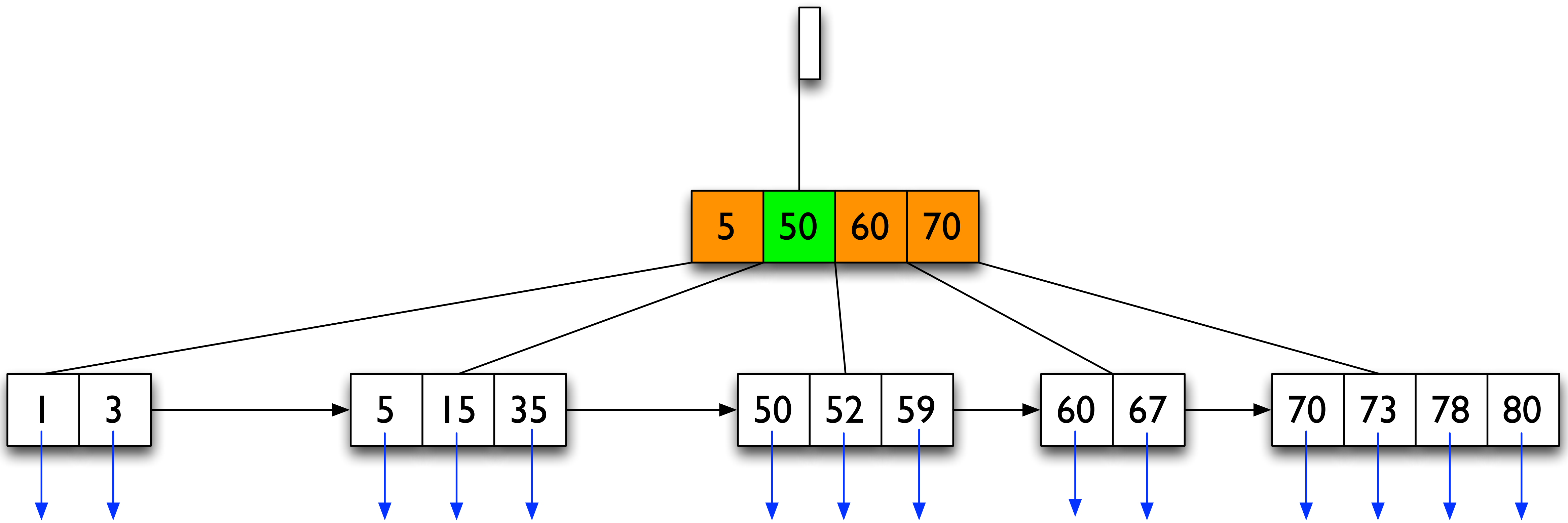
Suppression de 11 - rééquilibrage



Arbres B+ : suppression

- **Exemple.** n=4, min= 1 sur la racine, min =2 sur tous les autres noeuds

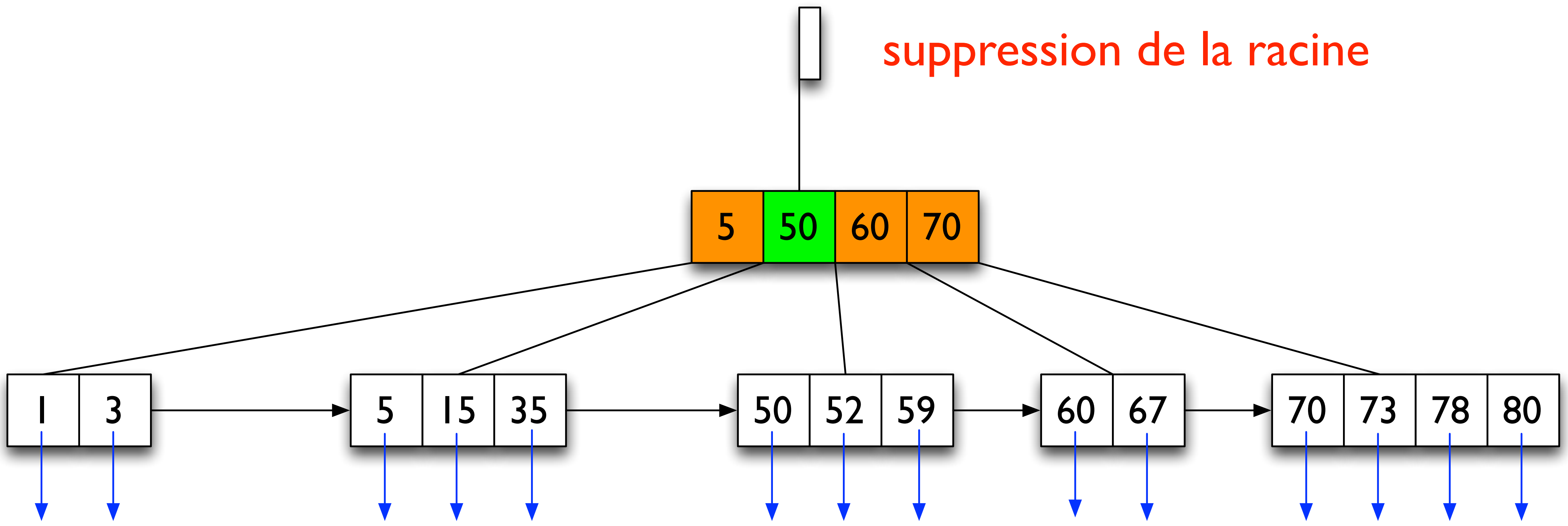
Suppression de 11 - rééquilibrage



Arbres B+ : suppression

- **Exemple.** n=4, min= 1 sur la racine, min =2 sur tous les autres noeuds

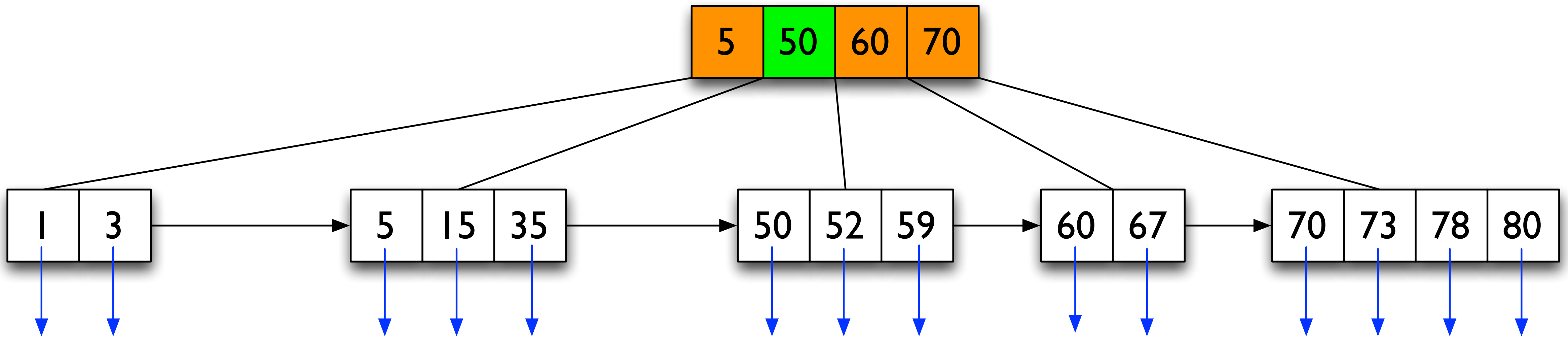
Suppression de 11 - rééquilibrage



Arbres B$^+$: suppression

- **Exemple.** n=4, min= 1 sur la racine, min =2 sur tous les autres noeuds

Suppression de 11 - rééquilibrage



Arbre rééquilibré

Hauteur d'un arbre B^+

- Recherche, insertion et suppression (y compris le rééquilibrage) ont un coût proportionnel à la hauteur de l'arbre
- Un arbre B+ contient un nombre relativement "petit" de niveaux en pratique :
  - racine : au moins 2 fils,
  - tout noeud interne non-racine : au moins $d = \lfloor n/2 \rfloor + 1$ fils
  - nombre de feuilles d'un arbre B+ de hauteur h : au moins $2\, d^{\,h-1}$
  - chaque feuille contient au moins $\lceil n/2 \rceil \geq d-1 \geq d/2$ clefs (d "grand")
  - ⇒ le nombre de clefs dans un arbre B+ de hauteur h est

$$\geq 2\, d^{\,h-1} \lceil n/2 \rceil \geq d^{\,h}$$

- ⇒ un arbre B+ avec K clefs a hauteur $O(\log_d(K))$
- ⇒ nombre logarithmique d'accès au disque pour une recherche / mise à jour de l'index
- en pratique : opérations en temps "presque" constant (d élevé)

Hauteur d'un arbre B^+ et prestations en pratique

- Taille typique d'un noeud (bloc du disque) : 4 Kilobytes
- Taille typique d'une entrée de l'index : 40 bytes
- $\Rightarrow$ typiquement $n = 4096/40 \sim 100$ $\Rightarrow d=50$
- Avec 1 000 000 de clefs de recherche :
  - hauteur : $\log_{50} (1\ 000\ 000) < 5$
- $\Rightarrow$ au plus 5 accès au disques pour une recherche

  (environ le double pour une insertion/ suppression)
- comparer avec l'hauteur d'un ABR équilibré : $\log_2 (1\ 000\ 000) \sim 20$
- la différence est importante puisque chaque accès au disque peut demander environ 20 millisecondes

Arbres B

- Différence par rapport aux arbres B^+: des couples

  <clef, pointeur à enregistrement>

  sont stockées dans tous les noeuds de l'arbre, pas uniquement les feuilles (pas de balises)

- Avantages des arbres B :
  - gagne de mémoire : l'espace occupé par les balises
  - la recherche peut s'arrêter avant d'arriver aux feuilles
- Inconvénients des arbres B :
  - moins de clefs dans les noeud internes (qui stockent : clefs, pointeurs aux enregistrements et pointeurs aux sous-arbres)
    $\Rightarrow$ dégrée plus petit $\Rightarrow$ arbre plus profond
  - rééquilibrage plus complexe
  - parcours linéaire de la table par clef de recherche plus complexe
- Inconvénients plus importants que les avantages

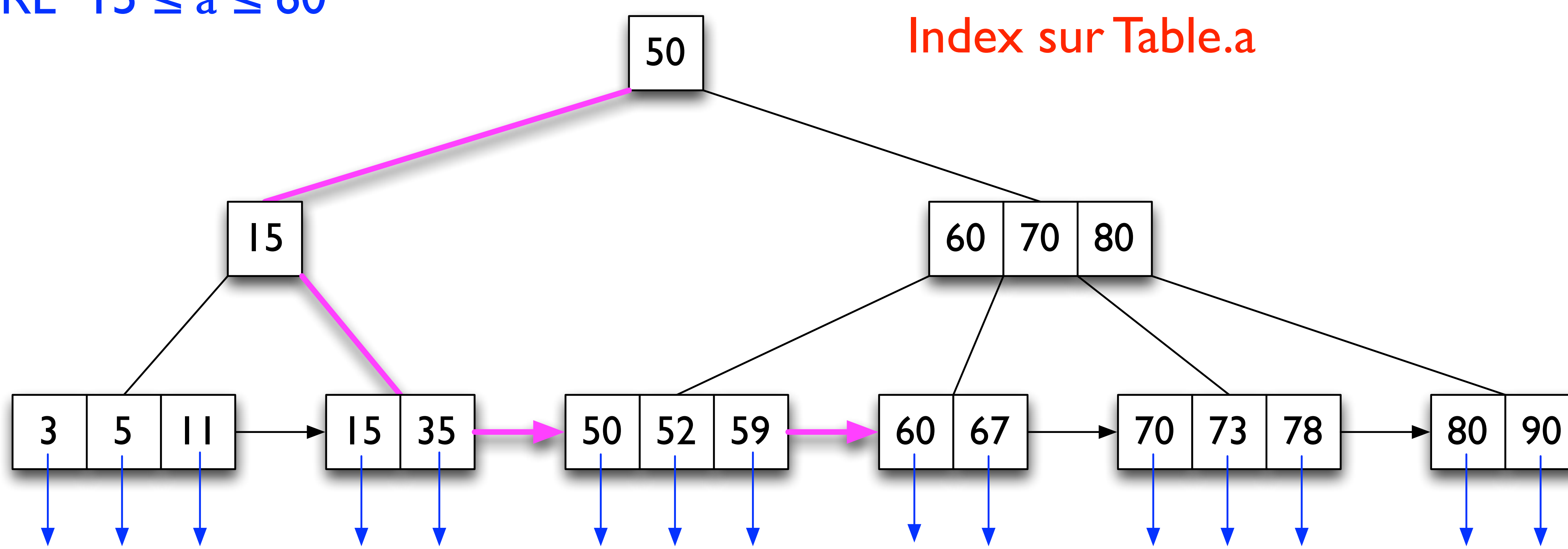
Arbres B⁺ : conclusion

- Très utilisés en pratique
  - coût de la recherche et du rééquilibrage pratiquement constant pour des index pas trop gros
- Très utilisés surtout pour les recherches d'intervalle, i.e.

SELECT *

FROM Table

WHERE 15 ≤ a ≤ 60



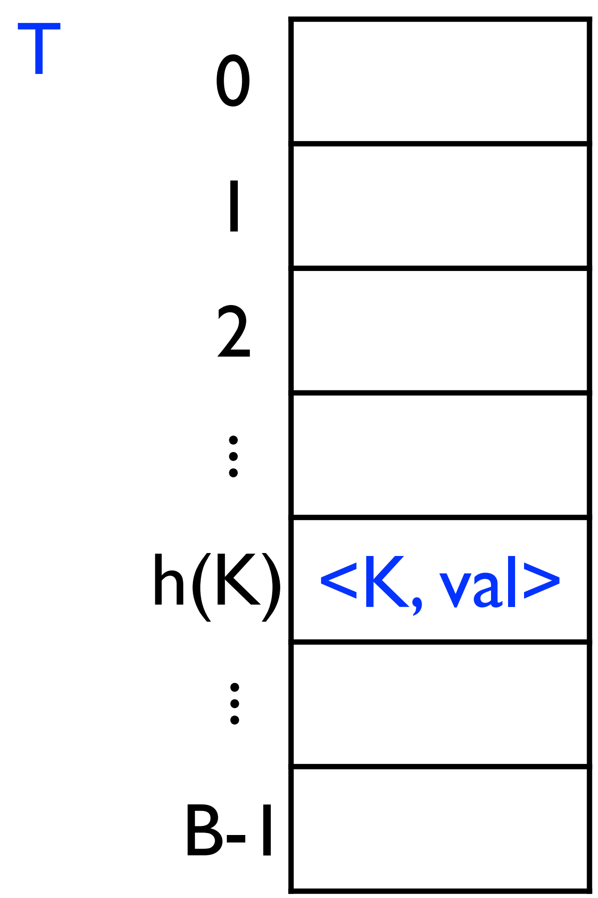
Rappel : Implémentation des index

- En tant que type abstrait un index n'est rien d'autre qu'un **dictionnaire**!
- Implémentation des index :
  - implémentation du type dictionnaire adaptée et optimisée pour une représentation en mémoire secondaire
- Plusieurs implémentations possibles :
  - Séquentielle (intérêt historique, pas très utilisé)
  - Arbres B^+
  - Hachage
  - Bitmap (pas abordé)

Rappel : hachage en mémoire principale

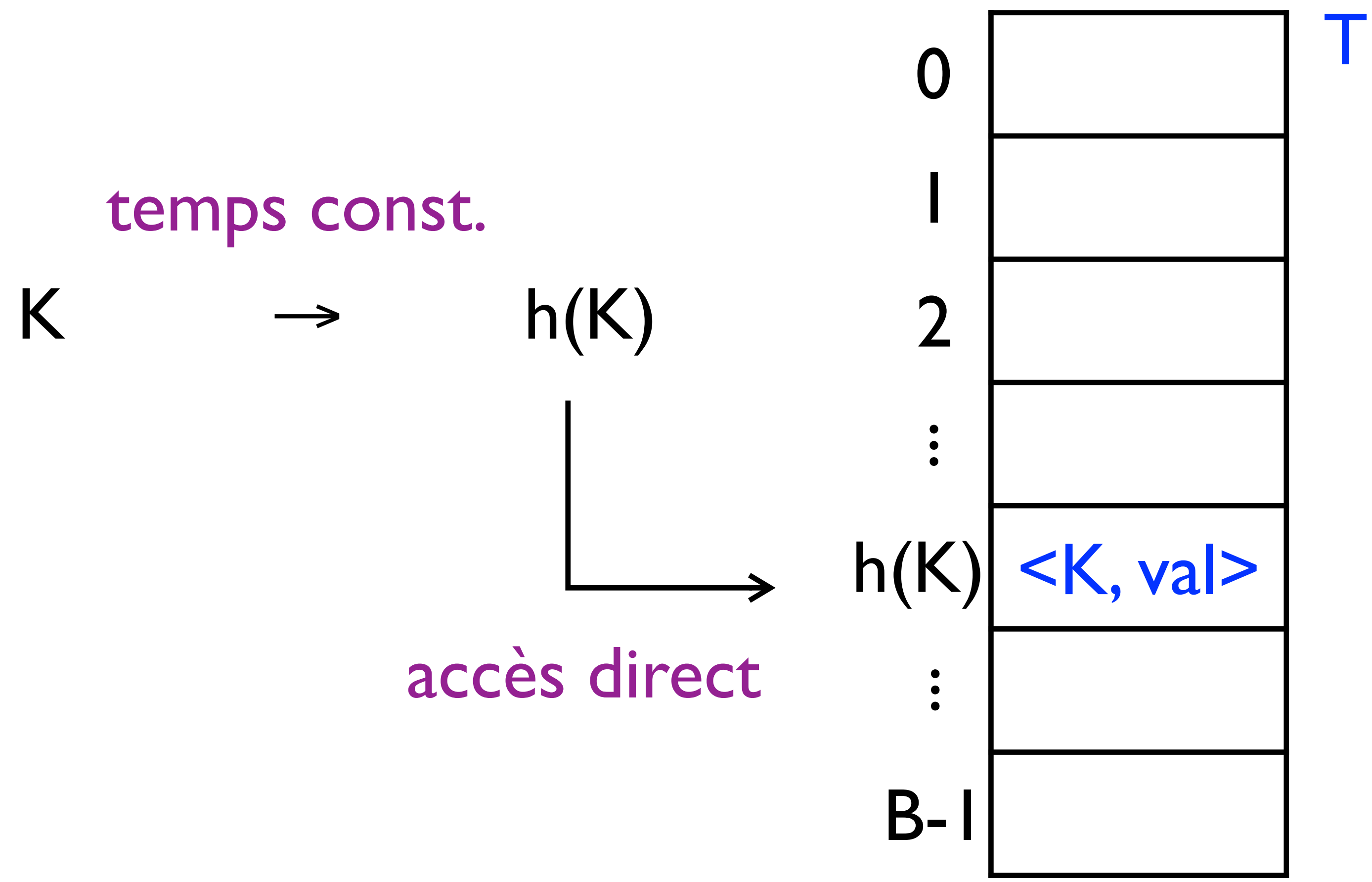
- Hachage : une implémentation de dictionnaire courante en mémoire principale
  - Rappel : table de hachage
    - Couples <*clef, valeur*> stockées dans un tableau T[0..B-1]
    - L'indice de T où l'élément de clef K est stocké est *calculé* à partir de K, par une fonction:

h: *Domaine des clefs* → {0,..,B-1}   fonction de hachage



Hachage en mémoire principale

- Idéalement : recherche d'une clef en un seul accès au tableau (temps constant)



- Idem pour l'insertion : insérer <K, val> en position h(K)

- Mais le domaine des clefs est en général "très grand"
  ⇒ B << |*Domaine des clefs*|

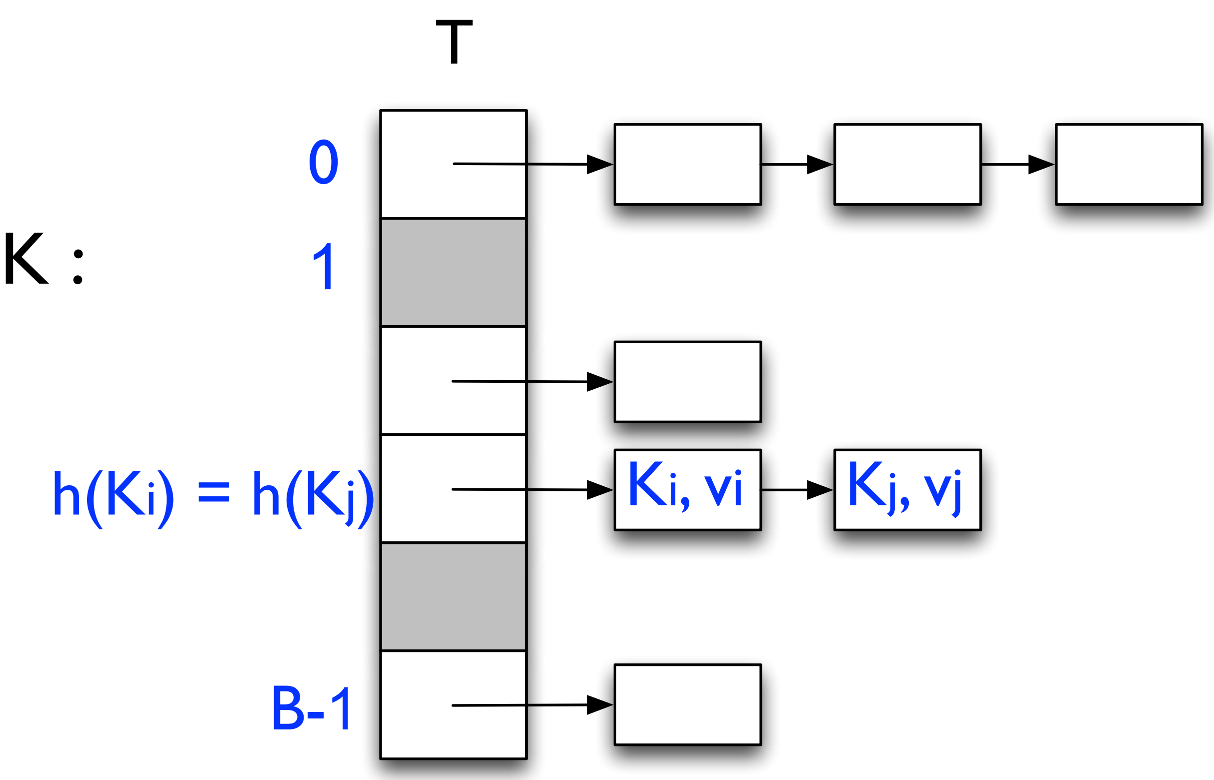
  ⇒ h en générale non-injective ⇒ collisions possibles

  ($h(K_i) = h(K_j)$ pour $K_i \neq K_j$)

Hachage en mémoire principale

Résolution des collisions. Approche courante : *chaînage*

- Chaque élément de T contient un pointeur vers une liste de couples <clef, valeur>
- Les éléments dont la clef a valeur de hachage j sont placés dans la liste pointée par T[j] (liste d'*overflow*)



- **Recherche/Suppression** d'une clef K : parcours de la liste T[h(K)]
- **Insertion** de (K,v): insertion en tête de la liste T[h(K)]

D'autres techniques : *adressage ouvert*

- pas d'*overflow* : la fonction de hachage renvoie une séquence de plusieurs positions à sonder jusqu'à en trouver une libre (séquence de sondage)

Hachage en mémoire principale

Coût de la recherche :

- Dans le pire des cas, linéaire, indépendamment de la technique de résolution des collisions
- Mais avec :

- une table bien dimensionnée

- des propriétés d'uniformité de la fonction de hachage par rapport à la distribution des clefs

Coût de la recherche constant en moyenne

Hachage en mémoire principale

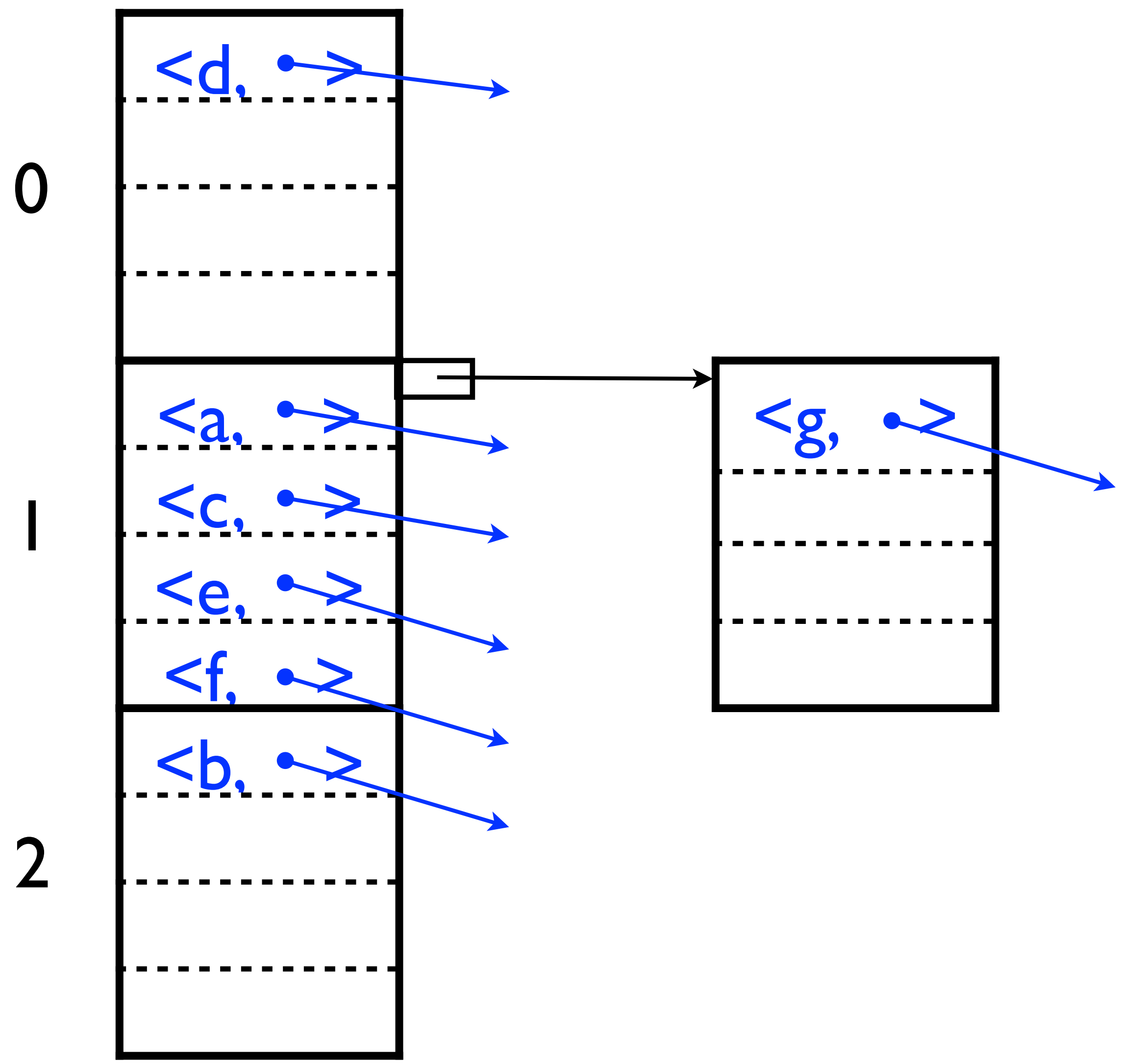
Fonctions de hachage typiques :

- Pour clefs entière $h(c) = c \bmod B$
  - Heuristique : B premier et loin d'une puissance de 2
  - Des variantes existent pour les techniques de hachage qui demandent B = puissance de 2
- Pour clefs de type chaîne de caractères :

$h(c)$ = (somme des octets de c) mod B

Index hash : hachage en mémoire secondaire

- Différences :
  - tableau de blocs du disque
  - liste de blocs d'*overflow*
  - couples <clef, pointeur à enregistrement>



h(a) = 1
h(b) = 2
h(c) = 1
h(d) = 0
h(e)=1
h(f) =1
h(g) =1

Index hash : hachage en mémoire secondaire

- Recherche de la clef K :
  - charger en mémoire le bloc d'indice h(K)
  - parcourir les clefs de ces bloc
  - si K n'est pas trouvée, charger et parcourir les blocs d'*overflow* h(K)
- Insertion de K :
  - charger le bloc h(K), y insérer K, s'il y a de la place
  - sinon chercher un bloc qui a de la place dans la liste d'*overflow* h(K)
  - s'il n'y a pas de place, créer un nouveau bloc d'*overflow* dans la liste
- Suppression de K :
  - effectuer une recherche de K comme plus haut et supprimer K de son bloc
  - re-compacter la liste d'*overflow* h(K) :
    - si un bloc d'*overflow* devient vide le supprimer
    - si de la place se libère dans le bloc principal, y transférer des clefs depuis l'*overflow*

Index hash : hachage en mémoire secondaire

Coût en terme de nombre d'accès aux blocs du disque :

- On suppose accès direct au bloc d'indice h(K) (i.e. coût 1)
  - différentes façons de garantir cela :
    1. un tableau de pointeurs au blocs principaux, indexé par les valeurs de hachage, stocké en mémoire principale
    2. blocs de taille fixée et physiquement consécutifs sur disque : l'adresse du bloc i calculé depuis l'adresse du premier bloc et de i

- Toutes les opérations ont coût proportionnel à la taille des listes d'*overflow*
- Constant en moyenne avec une "bonne" fonction de hachage et un tableau suffisamment grand

Hachage en mémoire secondaire et performances

- Pour de bonnes performances :
  B choisi sur la base du nombre n de clefs à stocker
  - objectif : être le plus possible proche de B = nombre de blocs nécessaires pour stocker n clefs

- Possible dans d'autres applications classiques de tables de hachage (table de symboles d'un compilateur)
- Pas envisageable pour un index de BD : le nombre de clefs est fortement dynamique
- ⇒ dégradation des performances avec l'augmentation du fichier de données

Re-hachage

Une possibilité pour gérer la croissance de la structure :

- Tables de hachage combinées avec une technique de gestion de tables dynamiques
  - Exemple :
    - quand le tableau T de taille B est trop plein, doubler sa taille
    - adopter une autre fonction de hachage avec 2B valeurs
    - re-appliquer h sur toutes les clefs pour les re-mémoriser dans le nouveau tableau (re-hachage)
- Re-hachage pas souhaitable dans un SGBD
  - opération très couteuse
  - bloque l'utilisation de la table pendant son exécution

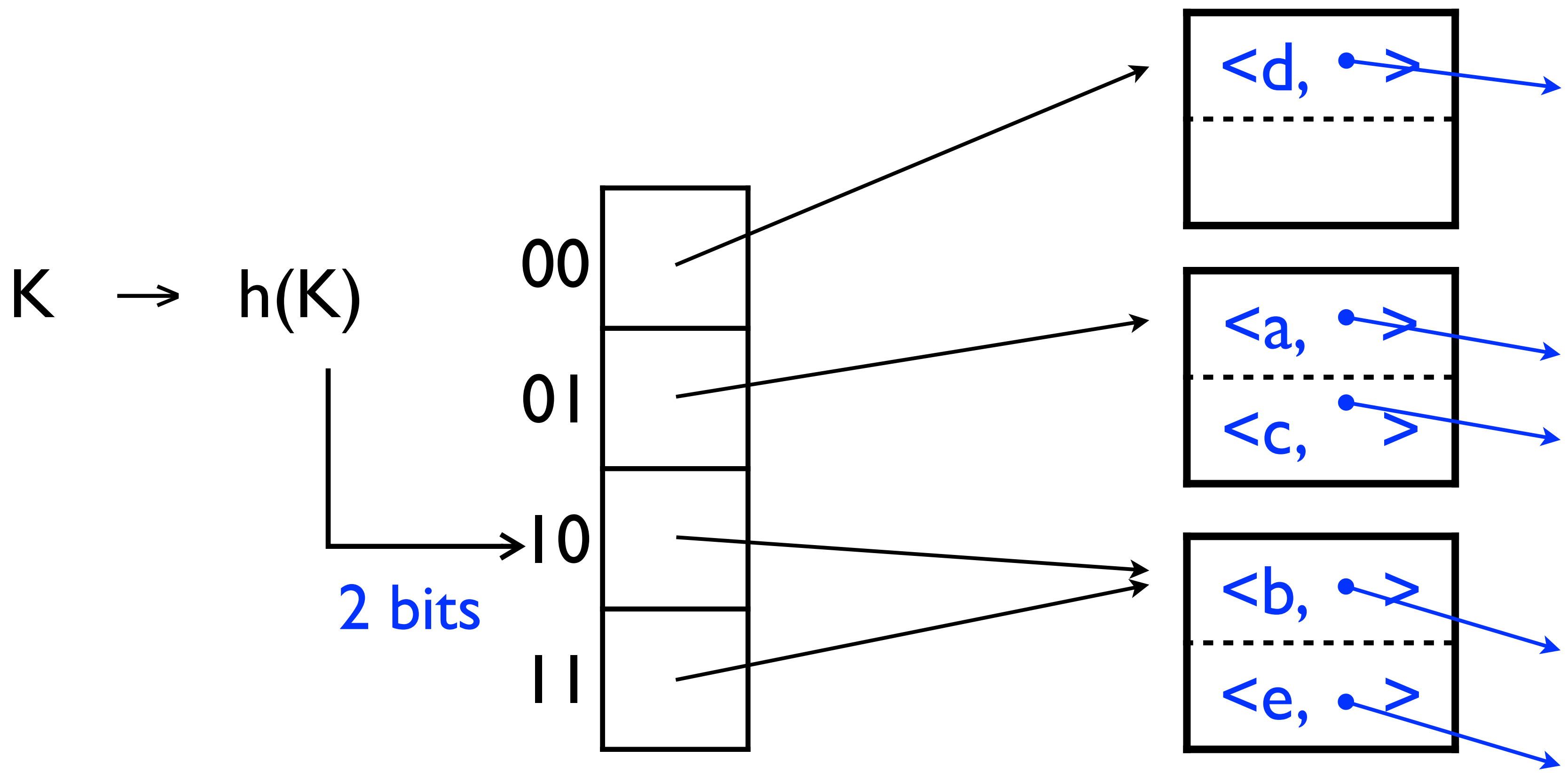
Hachage dynamique

- Une classe de techniques de hachage adaptées à gérer efficacement la croissance/réduction de l'ensemble de clefs à stocker
- Le hachage dynamique minimise le re-hachage nécessaire

- Plusieurs techniques dans cette classe. Les plus populaires :
  - hachage extensible
  - hachage linéaire (pas abordé)

Hachage extensible

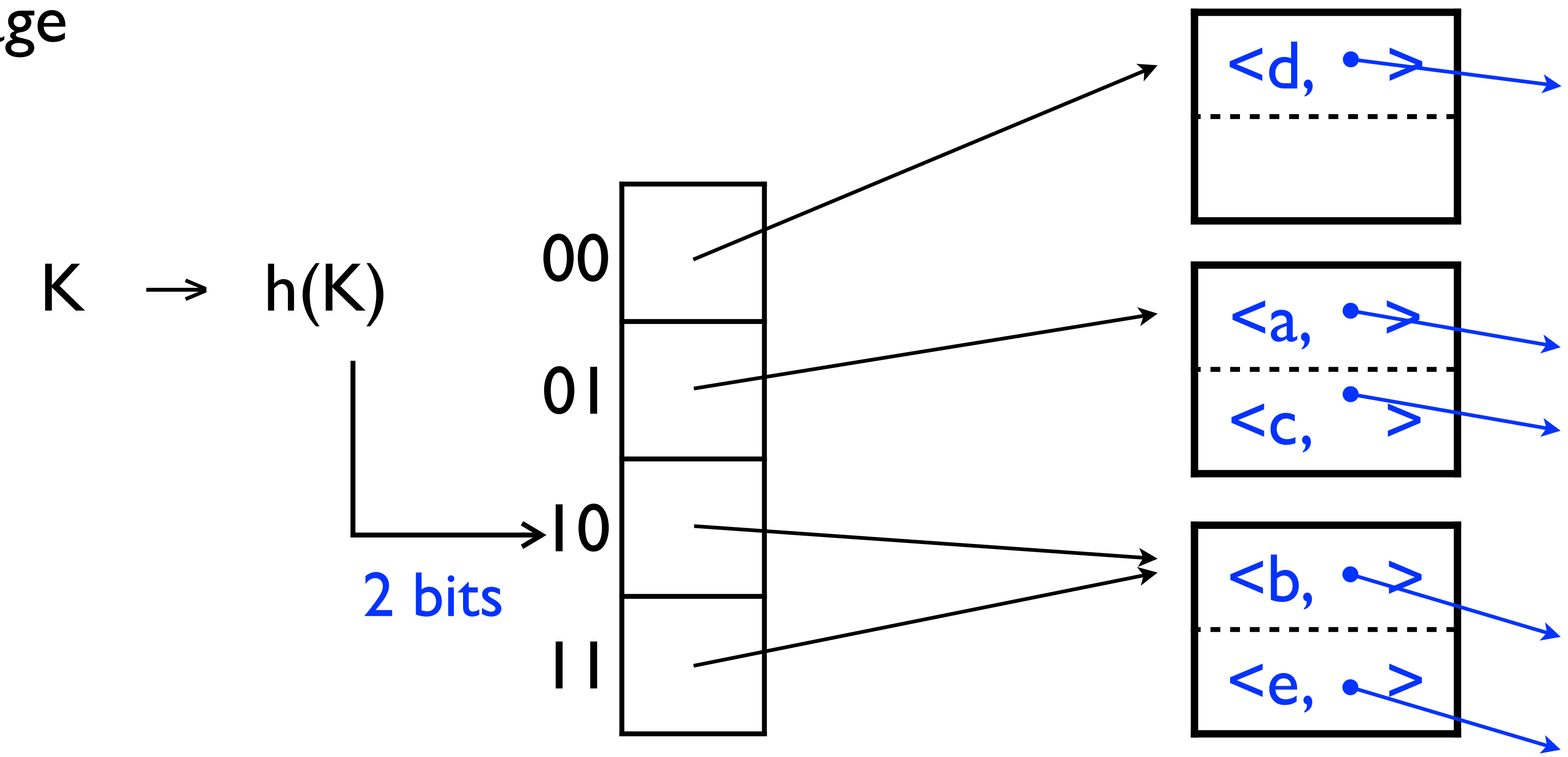
- L’index est constitué d’un tableau de pointeurs à blocs (à la place d’un tableau de blocs comme dans l’hachage statique)



- Un niveau d’indirection en plus ⇒ plusieurs valeurs de hachage peuvent partager le même bloc

Hachage extensible

- Valeurs de hachage h(K) : b bits (par exemple 32 bits)
- Indices du tableau : un préfixe de i des b bits produits par la fonction de hachage



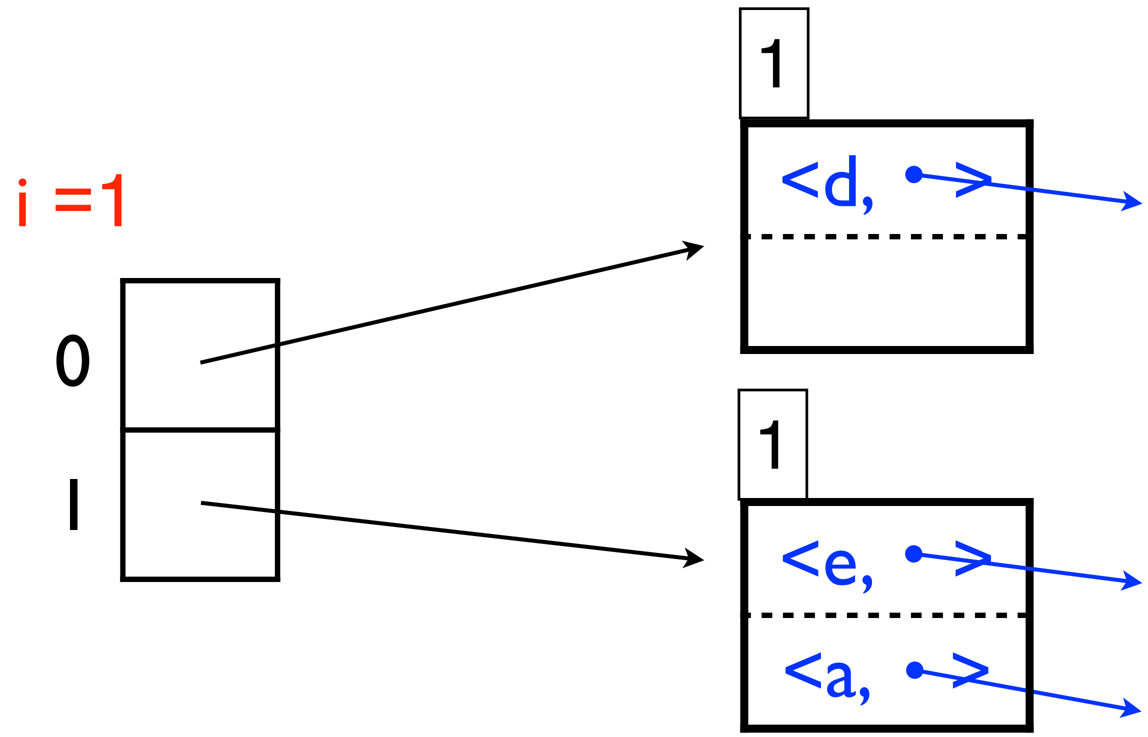
$\longleftarrow \; b \; \longrightarrow$

h(K) → $\boxed{00110101}$

$h(K)_i$ on en utilise i → i augmente dans le temps….

Hachage extensible

- Chaque bloc de clefs contient un compteur dans son *header*



Dans les exemples
- b = 4
- un bloc peut contenir 2 clefs

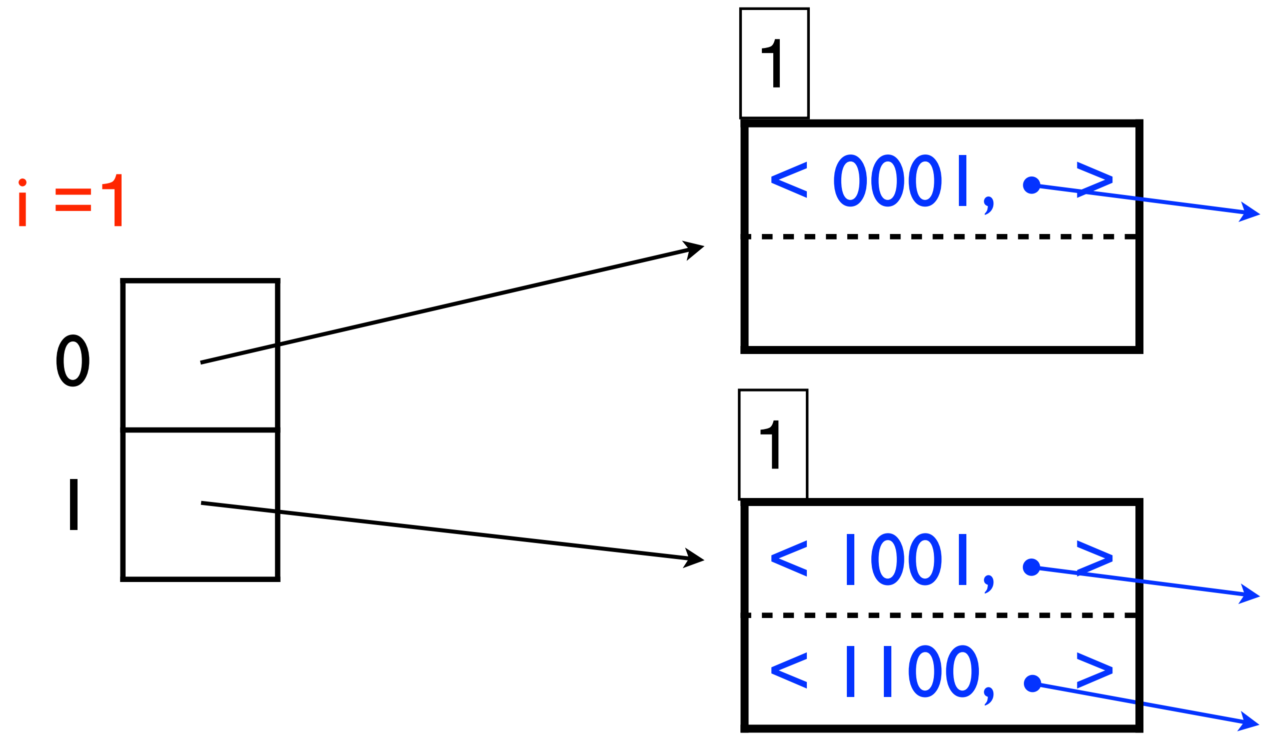
h(d) = 0001

h(e) = 1001
h(a) = 1100

- Compteur du bloc = p ⇔ le bloc est associé à une certaine valeur $b_1..b_p$ des p premiers bits de la valeur de hachage
  - I.e. le bloc contient exactement toutes les clefs K de la structure telles que $h(K)_p = b_1...b_p$
- Initialement le compteur de chaque bloc = i, mais peut devenir < i (cf. plus loin)

Hachage extensible

- Chaque bloc de clefs contient un compteur dans son *header*



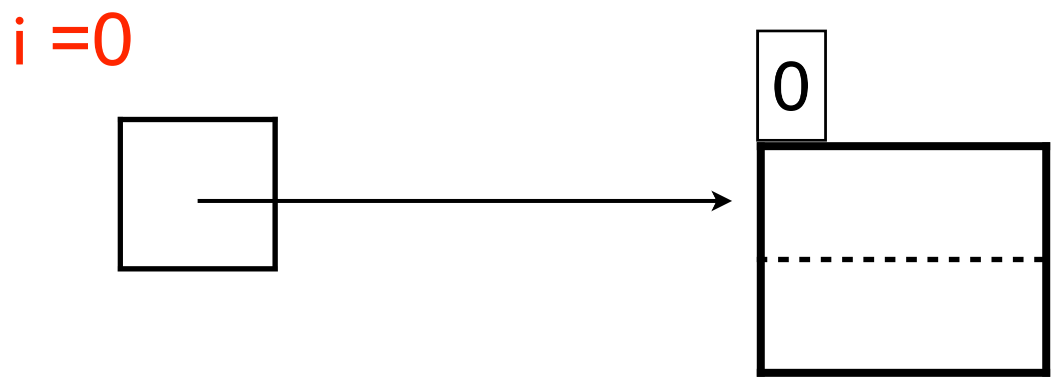
Dans les exemples
- b = 4
- un bloc peut contenir 2 clefs

On montre directement les valeurs de hachage à la place des clefs, par simplicité

- Compteur du bloc = p $\Leftrightarrow$ le bloc est associé à une certaine valeur $b_1..b_p$ des p premiers bits de la valeur de hachage
  - I.e. le bloc contient exactement toutes les clefs K de la structure telles que $h(K)_p = b_1...b_p$
- Initialement le compteur de chaque bloc = i, mais peut devenir < i (cf. plus loin)

Hachage extensible : index vide

- Remarque : état initial de l'index hash vide



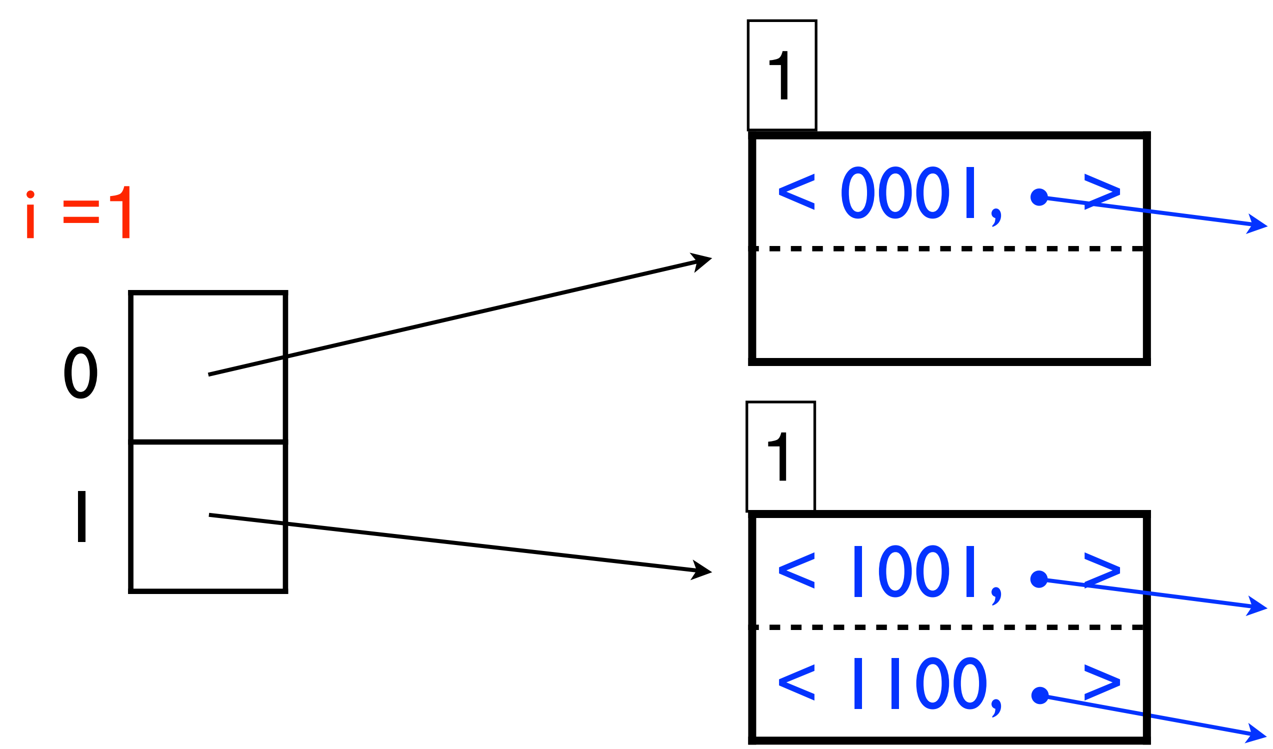
Hachage extensible : insertion

Insertion de K

- Calculer h(K)
- En extraire les *i* bits plus significatifs : h(K)***i***

S'il y a de la place dans le bloc pointé par h(K)***i*** , insérer K dans ce bloc

Exemple. h(K) = 0111



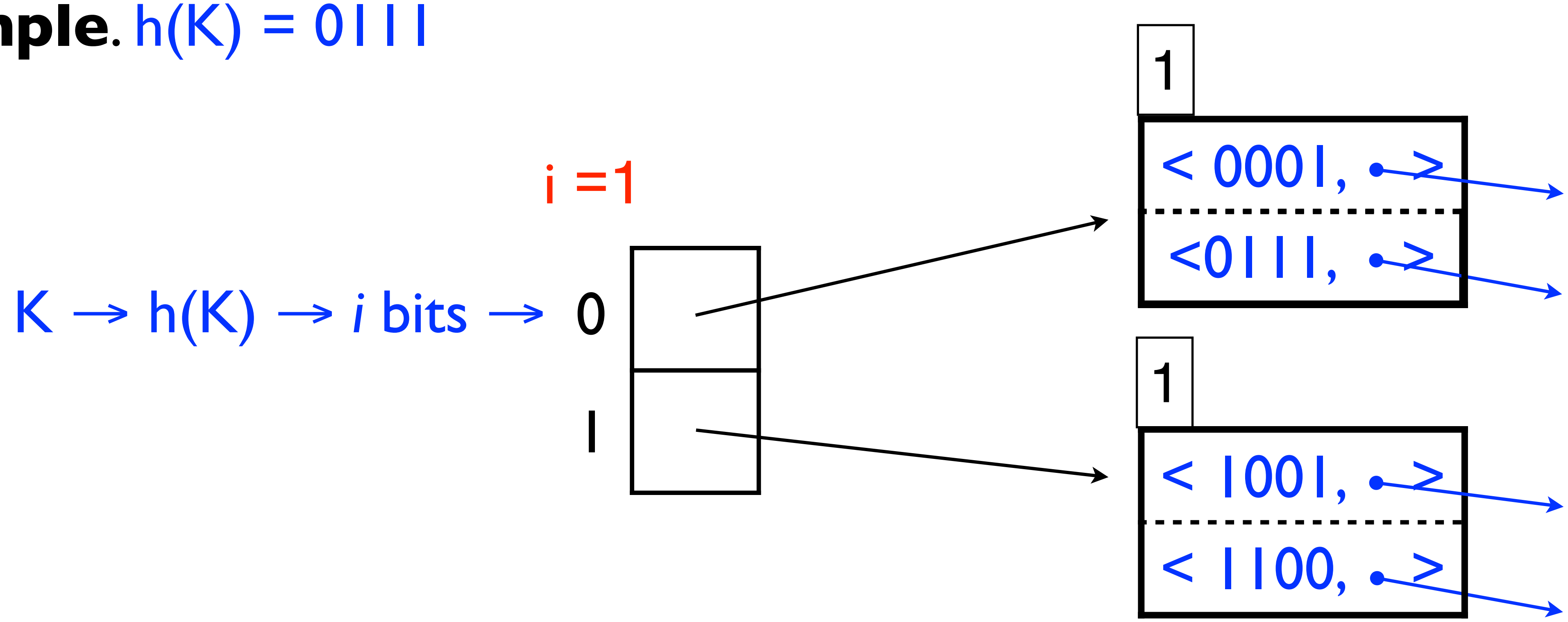
Hachage extensible : insertion

Insertion de K

- Calculer h(K)
- En extraire les *i* bits plus significatifs : h(K)***i***

S'il y a de la place dans le bloc pointé par h(K)***i*** , insérer K dans ce bloc

Exemple. h(K) = 0111



Hachage extensible : insertion

S'il n'y a pas de place dans le bloc pointé par h(K)$\boldsymbol{i}$. Deux cas.

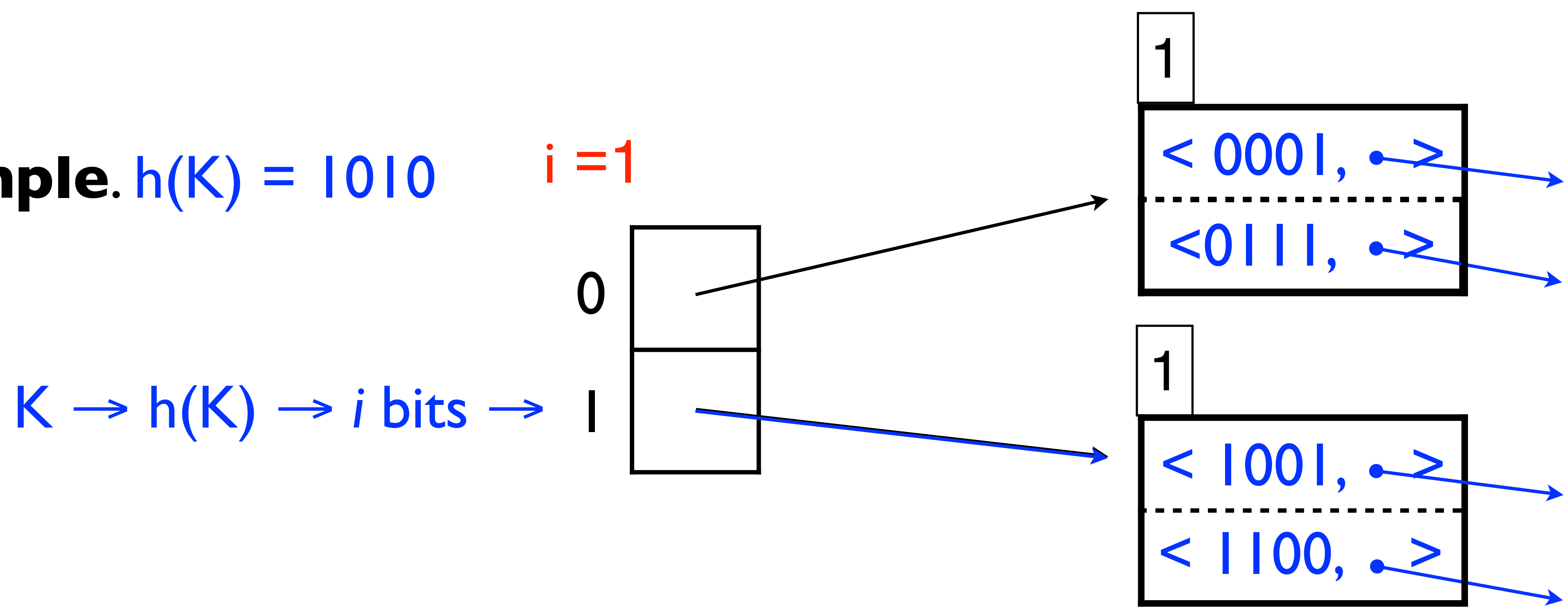
1) Le compteur du bloc est = i $\Rightarrow$

Idée : on veut créer un nouveau bloc et y re-distribuer les clefs du bloc plein.
Cela peut être obtenu en incrémentant le compteur de bloc

> (regarder plus de bits pour distinguer les clefs)

Mais le compteur doit rester ≤ i ⇒ nécessaire d'incrémenter i avant

Exemple. h(K) = 1010



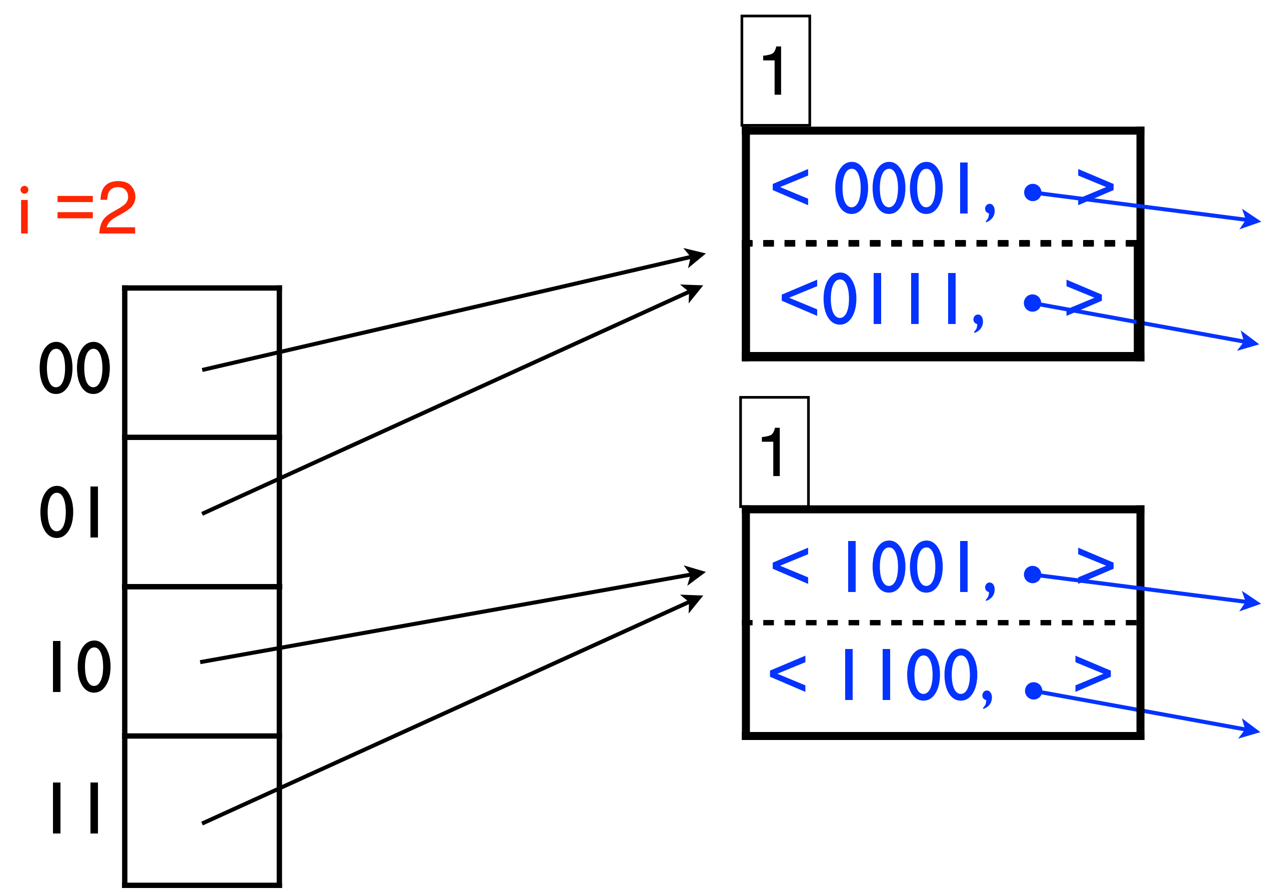
Hachage extensible : insertion

S'il n'y a pas de place dans le bloc pointé par h(K)$\boldsymbol{i}$. Deux cas.

1) Le compteur du bloc est = i $\quad\Rightarrow$

- augmenter i de 1,

Exemple. h(K) = 1010



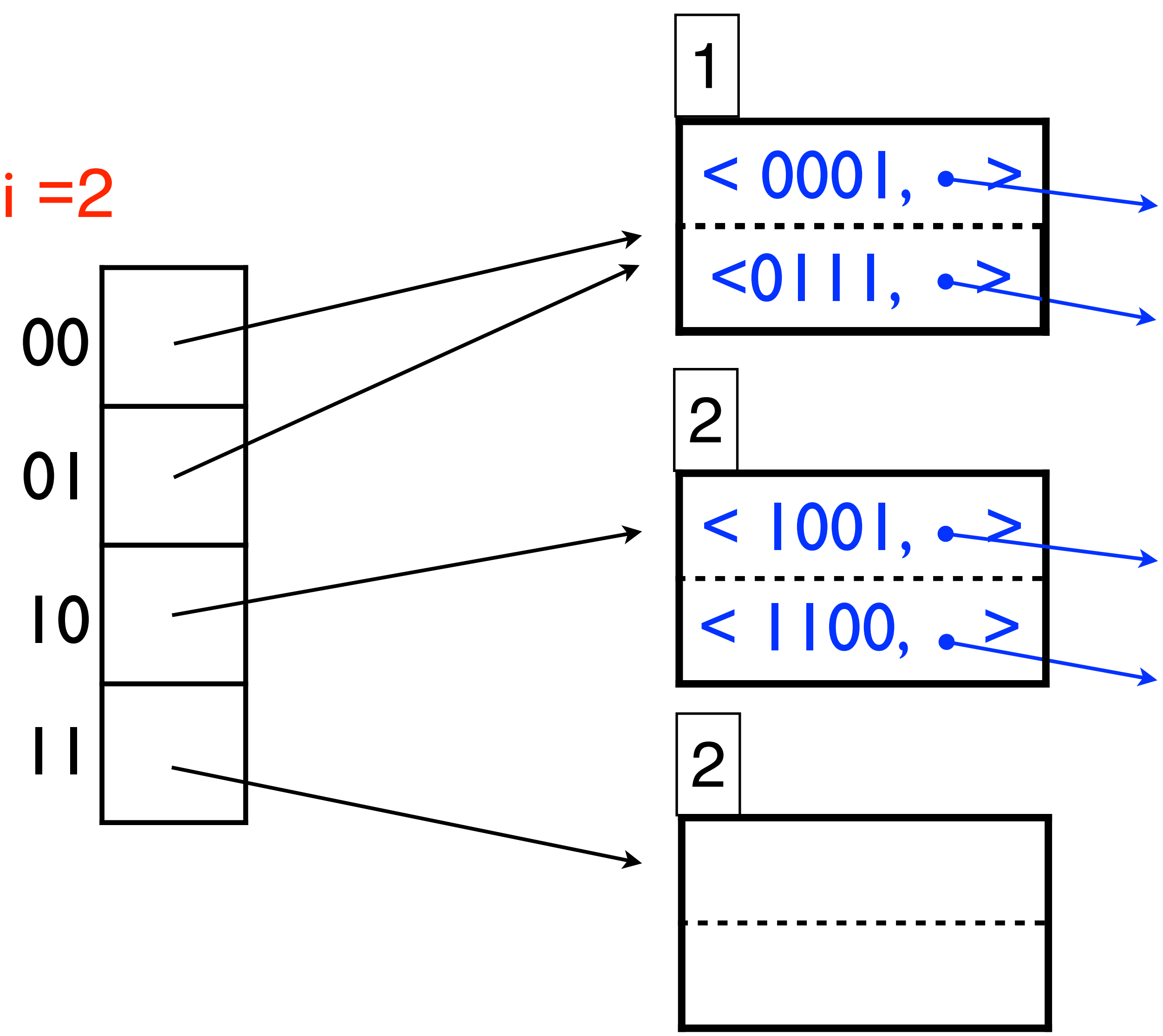
Hachage extensible : insertion

S'il n'y a pas de place dans le bloc pointé par h(K)$\boldsymbol{i}$. Deux cas.

1) Le compteur du bloc est = i $\Rightarrow$

- augmenter i de 1,
- créer un nouveau bloc, les compteurs du nouveau bloc et du bloc plein deviennent i

Exemple. h(K) = 1010 i =2



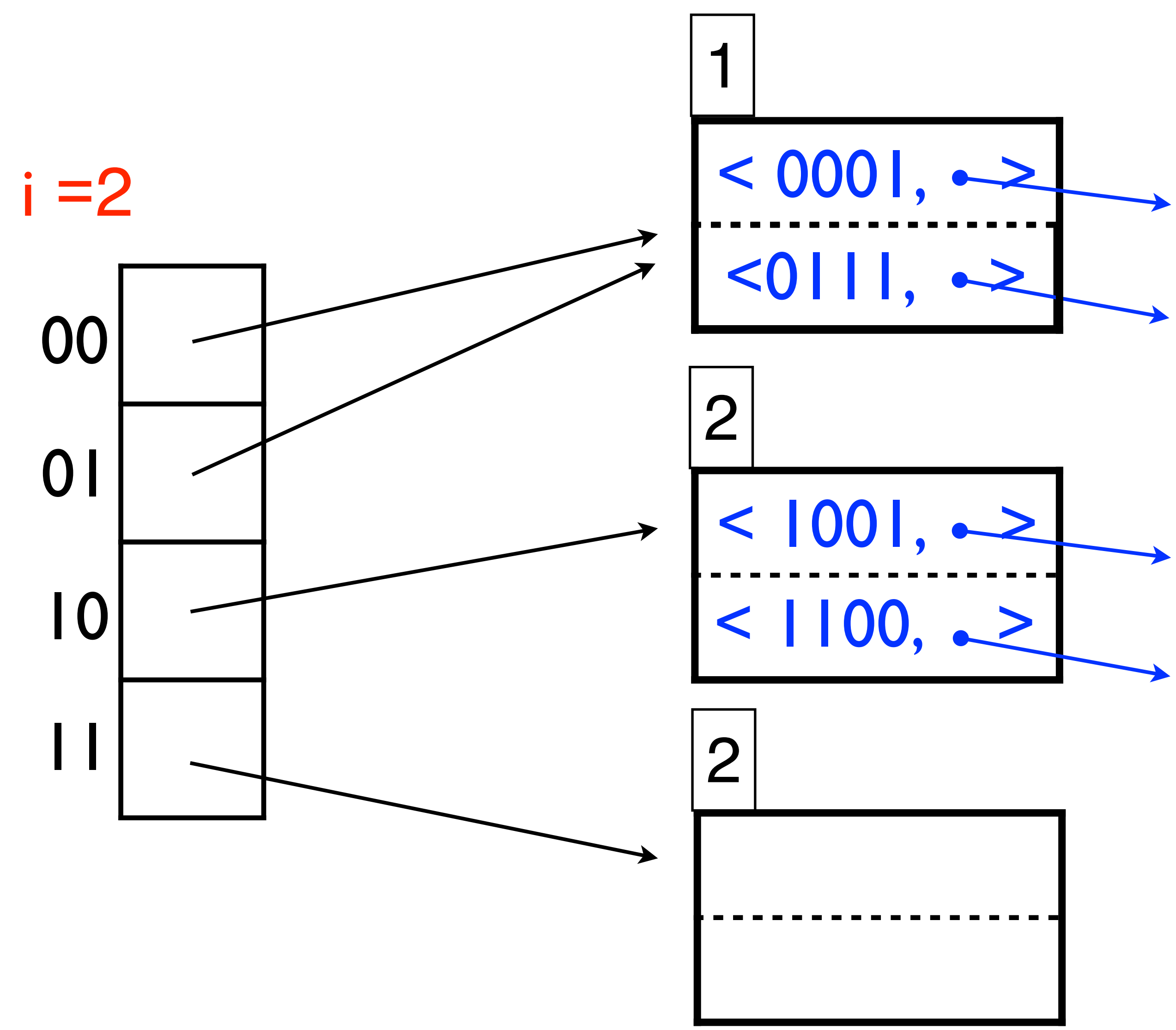
Hachage extensible : insertion

S'il n'y a pas de place dans le bloc pointé par h(K)$\boldsymbol{i}$. Deux cas.

1) Le compteur du bloc est = i $\Rightarrow$

- augmenter i de 1,
- créer un nouveau bloc, les compteurs du nouveau bloc et du bloc plein deviennent i
- re-hacher les clefs du bloc plein sur la base de i bits

Exemple. h(K) = 1010



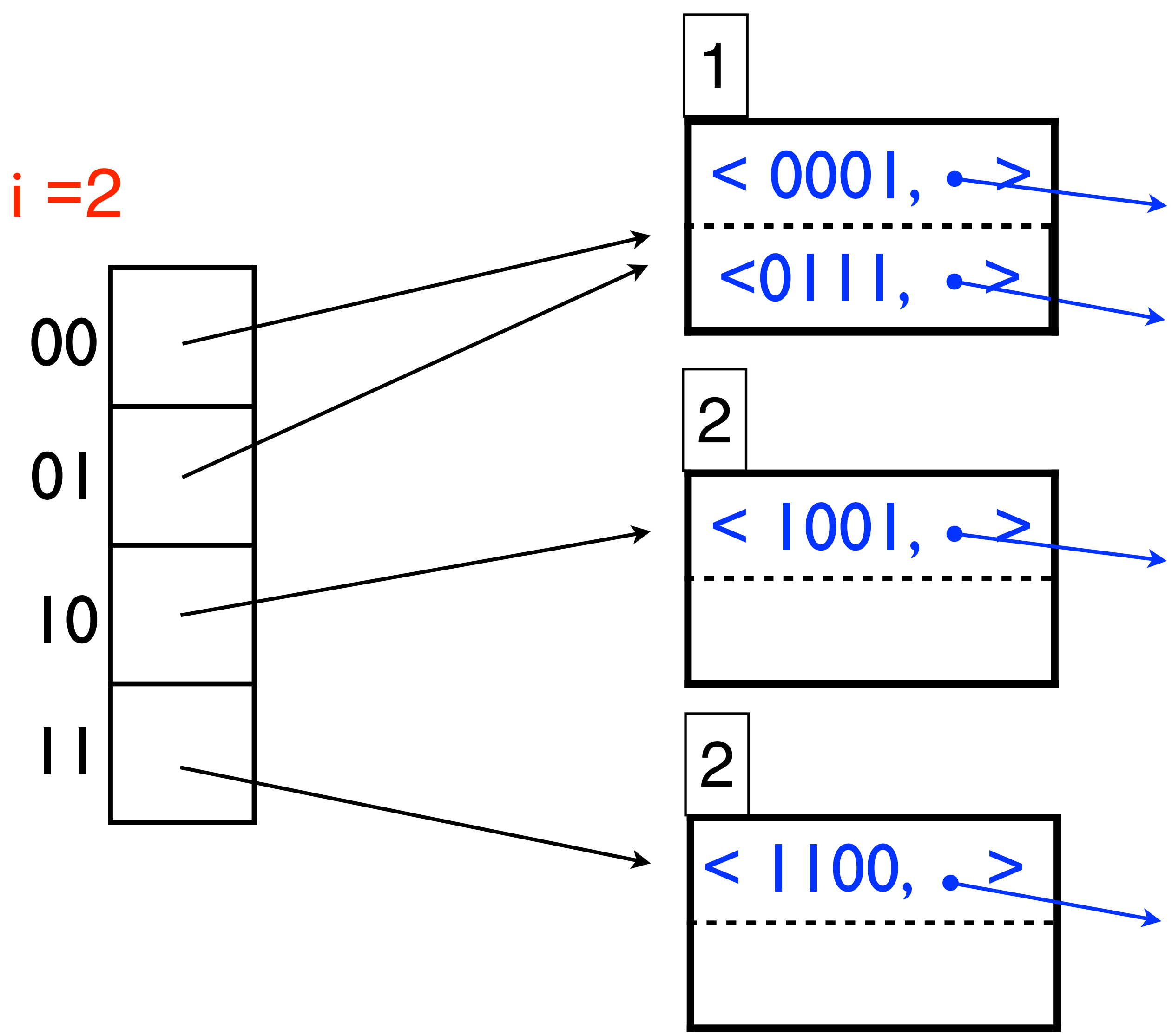
Hachage extensible : insertion

S'il n'y a pas de place dans le bloc pointé par h(K)$\boldsymbol{i}$. Deux cas.

1) Le compteur du bloc est = i $\Rightarrow$

- augmenter i de 1,
- créer un nouveau bloc, les compteurs du nouveau bloc et du bloc plein deviennent i
- re-hacher les clefs du bloc plein sur la base de i bits

Exemple. h(K) = 1010



Hachage extensible : insertion

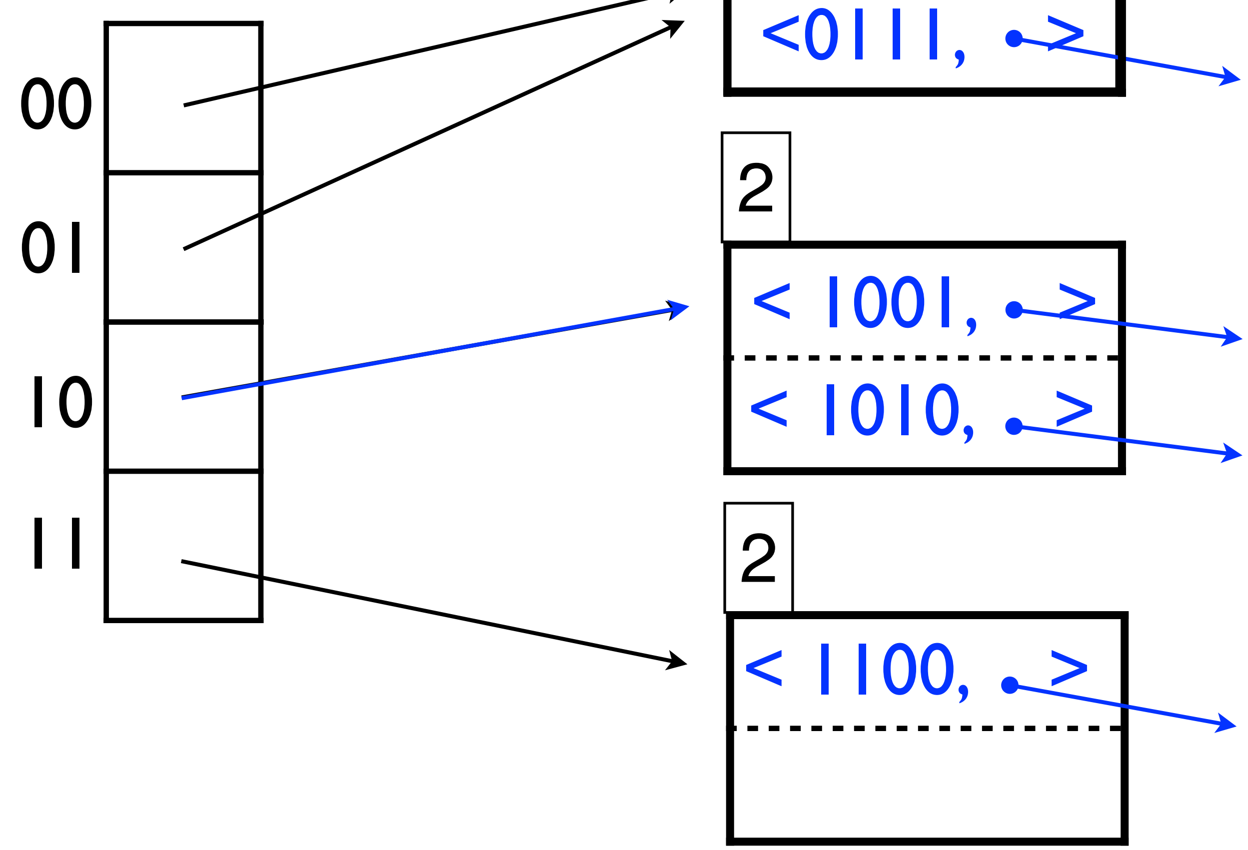
S'il n'y a pas de place dans le bloc pointé par h(K)$\boldsymbol{i}$. Deux cas.

1) Le compteur du bloc est = i $\Rightarrow$

- augmenter i de 1,
- créer un nouveau bloc, les compteurs du nouveau bloc et du bloc plein deviennent i
- re-hacher les clefs du bloc plein sur la base de i bits
- re-essayer l'insertion de K

Exemple. h(K) = 1010 i =2

K $\rightarrow$ h(K) $\rightarrow$ i bits $\rightarrow$



Hachage extensible : insertion

S’il n’y a pas de place dans le bloc pointé par h(K)$\boldsymbol{i}$. Deux cas.

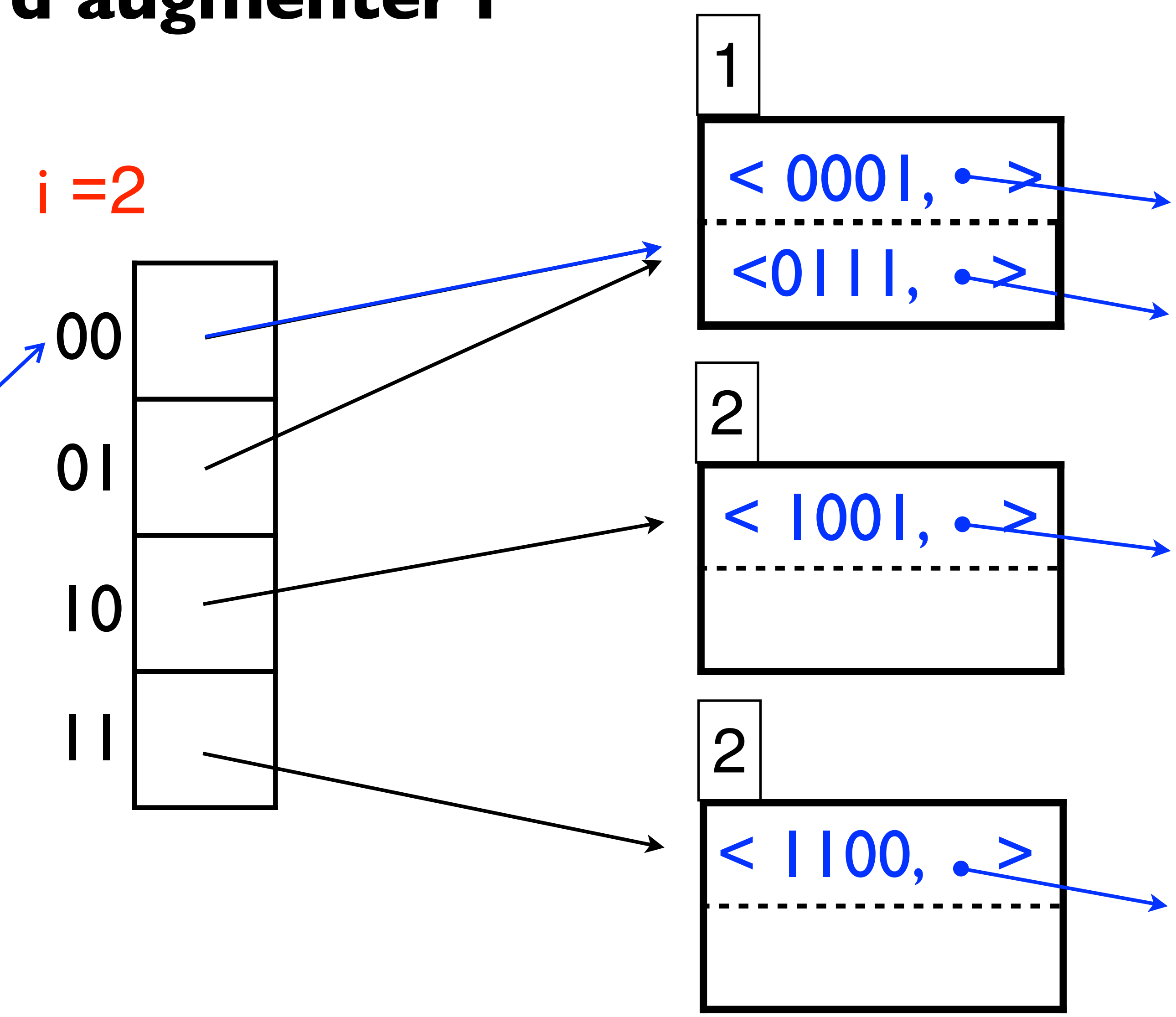
2) Le compteur du bloc est < i $\Rightarrow$

Idée : on veut créer un nouveau bloc et y re-distribuer les clefs du bloc plein.
Cela peut être obtenu en incrémentant le compteur de bloc
> (regarder plus de bits pour distinguer les clefs)

Puisque compteur < i, **pas nécessaire d’augmenter i**

Exemple. h(K) = 0000

K → h(K) → *i* bits



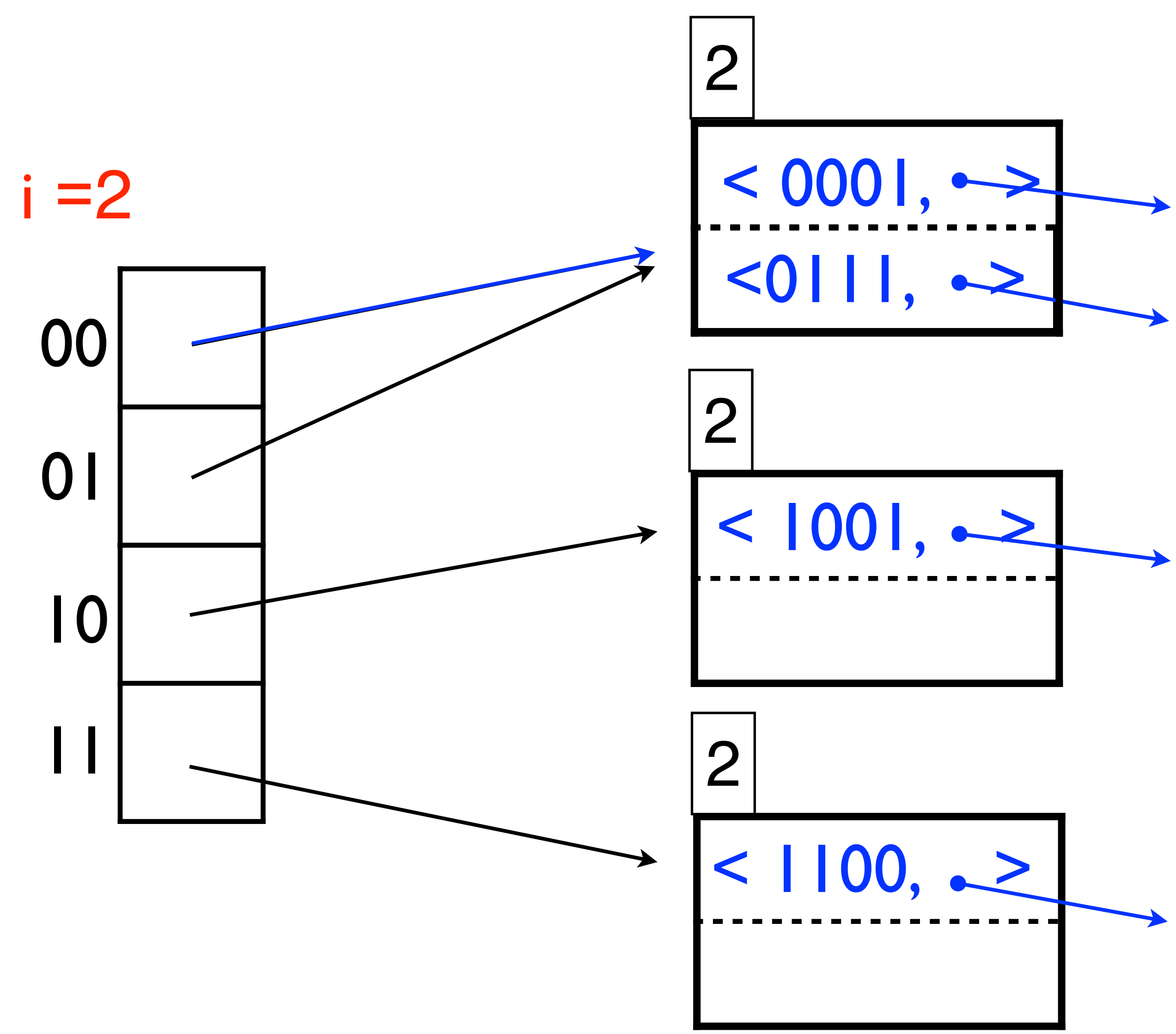
Hachage extensible : insertion

S'il n'y a pas de place dans le bloc pointé par h(K)$\boldsymbol{i}$. Deux cas.

2) Le compteur du bloc est < i $\Rightarrow$

- augmenter le compteur du bloc plein de 1,

Exemple. h(K) = 0000



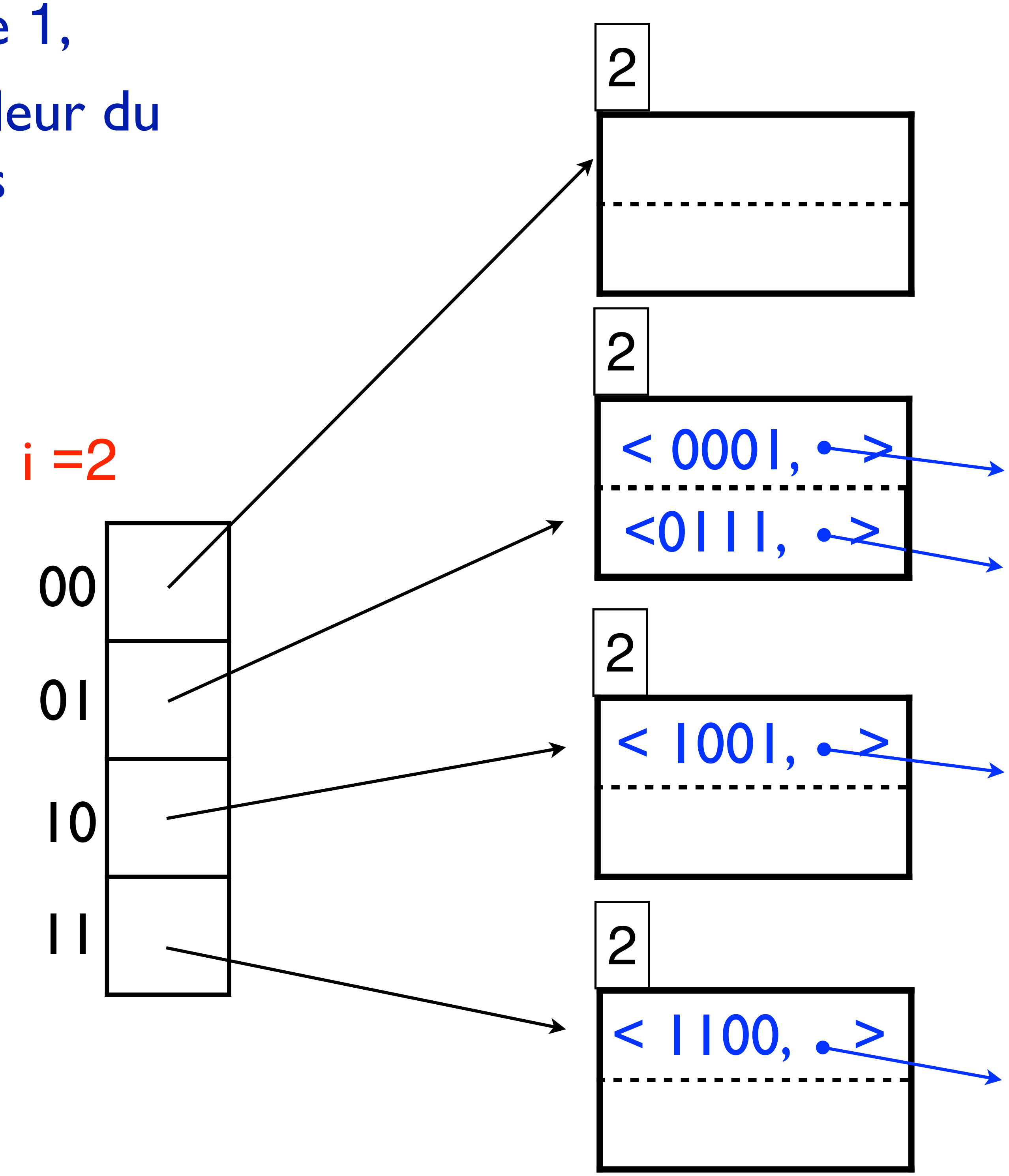
Hachage extensible : insertion

S'il n'y a pas de place dans le bloc pointé par h(K)$\boldsymbol{i}$. Deux cas.

2) Le compteur du bloc est < i $\Rightarrow$

- augmenter le compteur du bloc plein de 1,
- créer un nouveau bloc avec la même valeur du compteur et mettre à jour les pointeurs

Exemple. h(K) = 0000



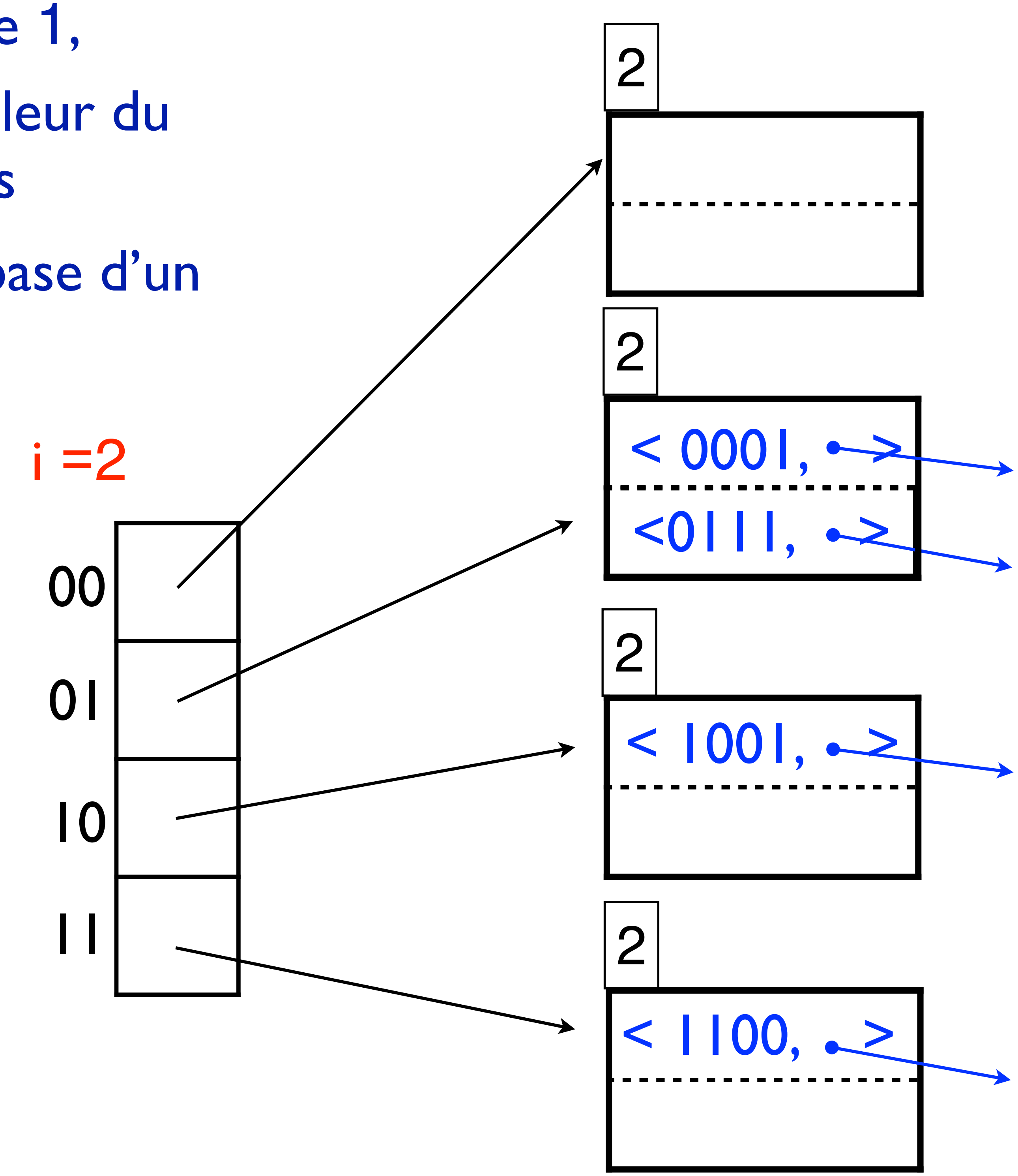
Hachage extensible : insertion

S'il n'y a pas de place dans le bloc pointé par h(K)i . Deux cas.

2) Le compteur du bloc est < i $\Rightarrow$

- augmenter le compteur du bloc plein de 1,
- créer un nouveau bloc avec la même valeur du compteur et mettre à jour les pointeurs
- re-hacher les clefs du bloc plein sur la base d'un nombre de bits = compteur de bloc

Exemple. h(K) = 0000

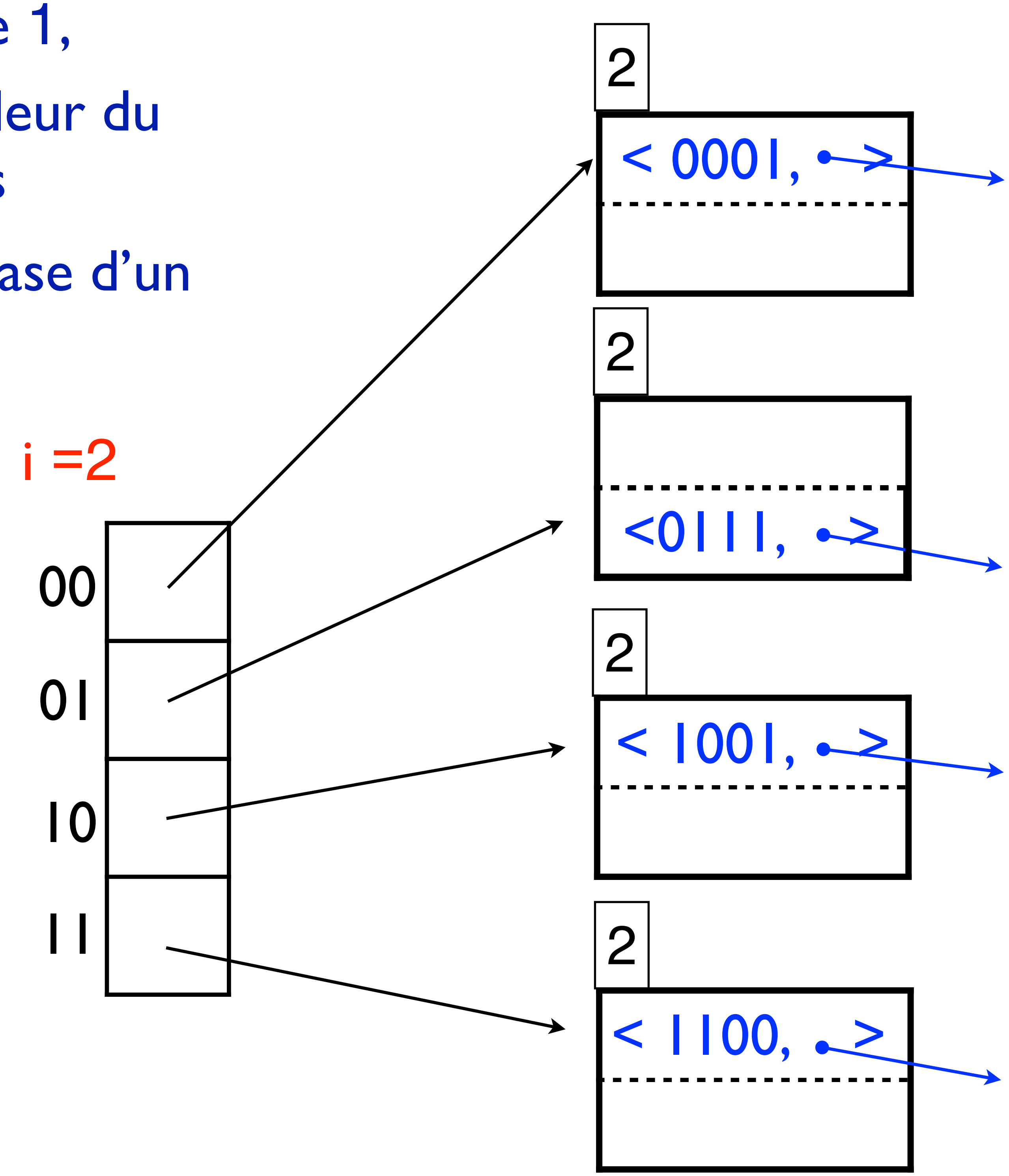


Hachage extensible : insertion

S'il n'y a pas de place dans le bloc pointé par h(K)$\boldsymbol{i}$. Deux cas.

2) Le compteur du bloc est $< i$ $\Rightarrow$

- augmenter le compteur du bloc plein de 1,
- créer un nouveau bloc avec la même valeur du compteur et mettre à jour les pointeurs
- re-hacher les clefs du bloc plein sur la base d'un nombre de bits = compteur de bloc



Exemple. h(K) = 0000

Hachage extensible : insertion

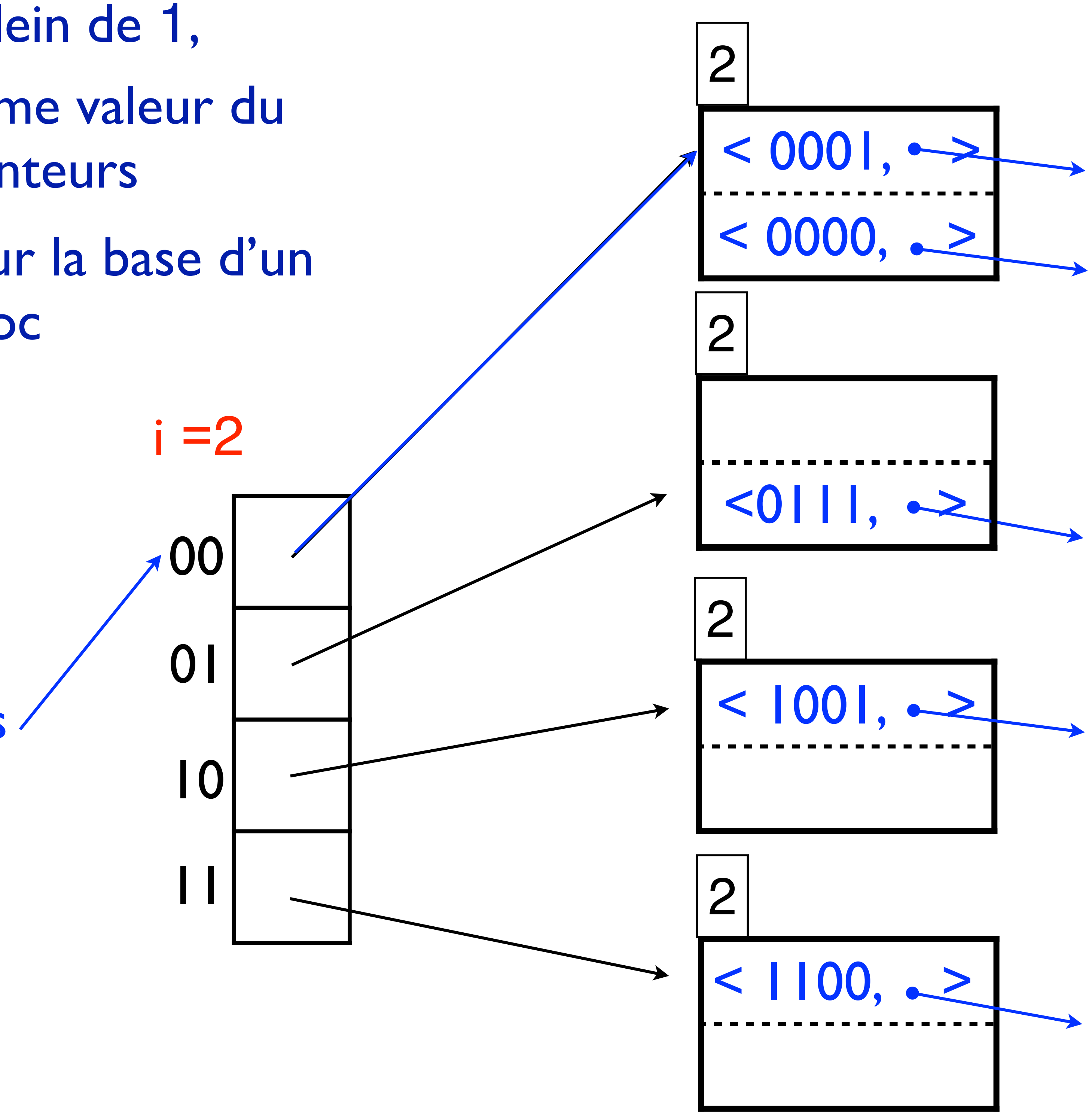
S'il n'y a pas de place dans le bloc pointé par h(K)$\boldsymbol{i}$. Deux cas.

2) Le compteur du bloc est < i $\Rightarrow$

- augmenter le compteur du bloc plein de 1,
- créer un nouveau bloc avec la même valeur du compteur et mettre à jour les pointeurs
- re-hacher les clefs du bloc plein sur la base d'un nombre de bits = compteur de bloc
- re-essayer l'insertion de K

Exemple. h(K) = 0000

K $\rightarrow$ h(K) $\rightarrow$ i bits



Hachage extensible : insertion

Dans tous les cas : on réessaie l'insertion jusqu'à ce qu'on trouve de la place

- Dans le cas de bloc plein, le prochain essai tente un des deux blocs concernés par le re-hachage à l'étape précédente
- Mais le re-hachage peut n'avoir aucun effet :
  - exemple : les valeurs de hachage des clefs dans les blocs diffèrent à partir du k-ieme bit, avec k grand, ou ne diffèrent pas du tout!
- En principe le tableau peut être doublé plusieurs fois sans libérer de la place
  - exemple : si les premiers 20 bits des clefs dans un blocs sont identiques, on double 20 fois ⇒ taille du tableau >= 2^{20}
- En pratique
  - avec un bonne h, c'est rare que une insertion demande plus que deux essais
  - avant de doubler la taille on examine les clefs du bloc plein
    - si le premier bit de hachage où elles diffèrent est "loin" ou inexistant ⇒ insérer dans un bloc d'*overflow*

Hachage extensible

- Recherche d'une clef
  - suivre le pointeur $h(K)_i$
  - examiner le clefs dans le bloc et potentiellement les blocs d'*overflow*
- Suppression
  - duale de l'insertion

Hachage extensible

- Avantages
  - une “bonne” fonction de hachage garantit en pratique absence d’overflow (la taille s’adapte au nombre de clefs)
  - re-hachage limité à un seul bloc à la fois
- Inconvénients
  - doubler la taille du tableau des pointeurs est coûteux et prend de la place
  - inefficace : on double la taille pour insérer un pointeur de plus
  - quand les clefs dans un même bloc plein ont des grands préfixes “proches” pas de bonne solution :
    - soit doubler la taille plusieurs fois
    - soit introduire de l’*overflow*

- L’*hachage linéaire* résout ces problèmes en garantissant une croissance plus lente de la taille du tableau des pointeurs

Arbres B$^+$ et hachage

- Index *hash* :
  - À préférer pour les requêtes de recherche de valeurs ponctuels

    SELECT A1, ...,An

    FROM T

    WHERE Ai = c

  - Nombre d'accès au disque constant en moyenne
    - pour les arbres B$^+$: logarithmique (avec une base élevée)

  - Dans le cas pire les arbre B$^+$ ont un meilleur cout asymptotique (logarithmique)
    - linéaire pour les index *hash*
  - Mais le cas pire est très rare en pratique avec une bonne fonction de hachage

Arbres B^+ et hachage

- Arbres B^+ :
  - À préférer pour les requêtes d'intervalle :

```
SELECT A1, ...,An
FROM  T
WHERE   Ai >= c   AND   Ai <= c'
```

    - coût d'une recherche ponctuelle (logarithmique)
    - plus parcours des clefs dans l'intervalle

  - Requêtes d'intervalle avec les index *hash* : en principe une nouvelle recherche pour chaque clef dans l'intervalle
    - il n'y a pas de raison que les clefs avec valeurs proches soient physiquement proches dans un index *hash*

Arbres B^+ et hachage pour l'organisation des fichiers de données

- Le hachage et les arbres B^+ ne sont pas utilisés uniquement pour les index, mais également pour organiser les fichiers de données
  - Idée : l'arbre B^+ ou table de hachage stocke des couples

    <clef, enregistrement>
  - Minimisent les problèmes de réorganisation périodique du fichier de données
  - En résumant : organisations possibles d'un fichier de données
    - en tas
    - séquentielle
    - en "grappe" multi-table (*clustered multi-relation*)
    - fichier à arbre B^+
    - fichier *hash*