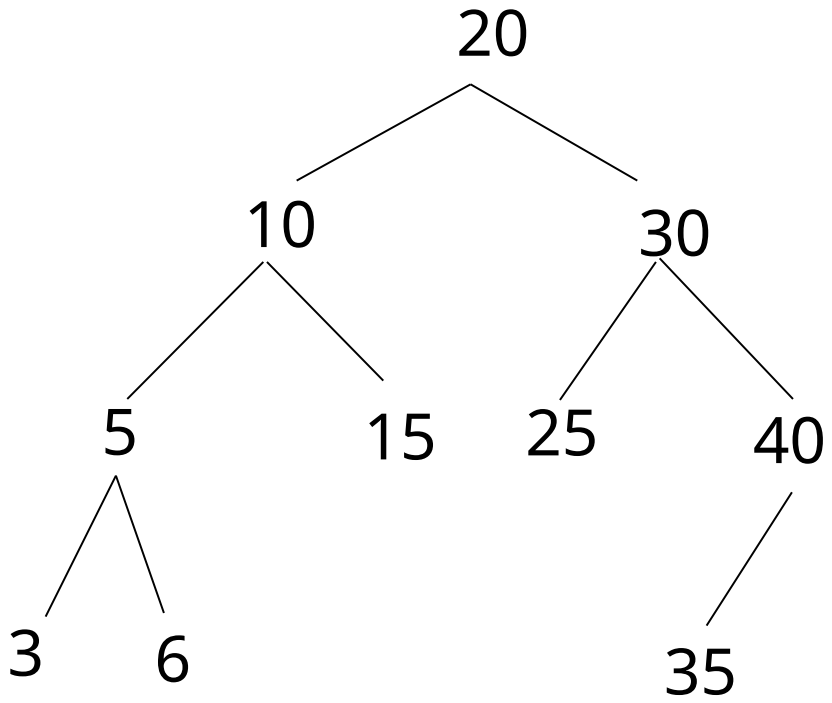
Alberi di ricerca binaria

- Utilizzato per la realizzazione di insiemi ordinati
- Operazioni efficienti di
  - ricerca
  - inserimento
  - cancellazione

Alberi di ricerca binaria: definizione

- Se l'albero non è vuoto
  - ogni elemento del sottoalbero di sinistra precede (<) la radice
  - ogni elemento del sottoalbero di destra segue (>) la radice
  - i sottoalberi sinistro e destro sono alberi di ricerca binaria

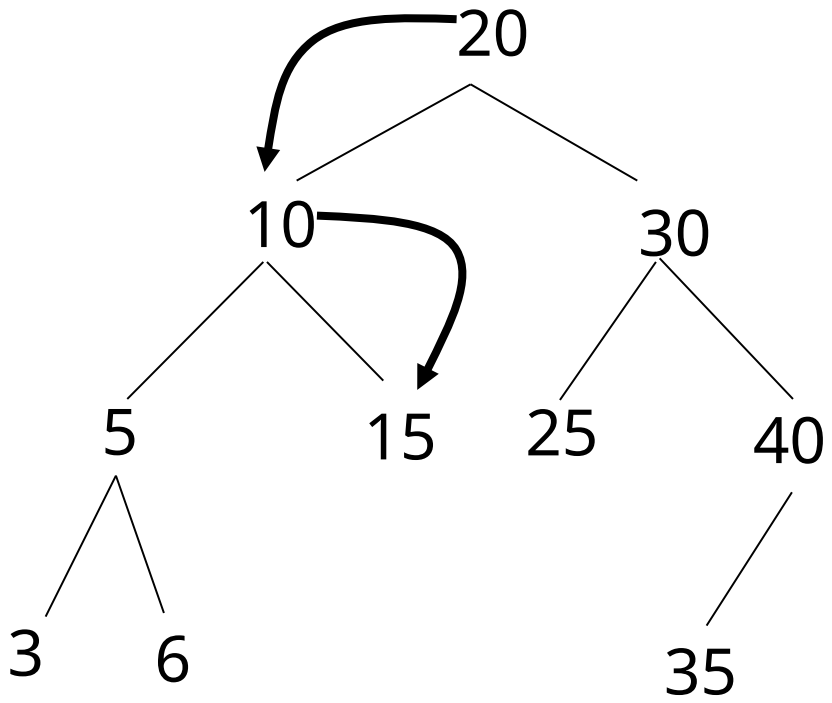
Alberi di ricerca binaria: esempio



Operazioni: contains

- Ricerca di un elemento
  - Se l'albero è vuoto allora restituisce false
  - Se l'elemento cercato coincide con la radice dell'albero restituisce true
  - Se l'elemento cercato è minore della radice restituisce il risultato della ricerca dell'elemento nel sottoalbero sinistro
  - Se l'elemento cercato è maggiore della radice restituisce il risultato della ricerca dell'elemento nel sottoalbero destro

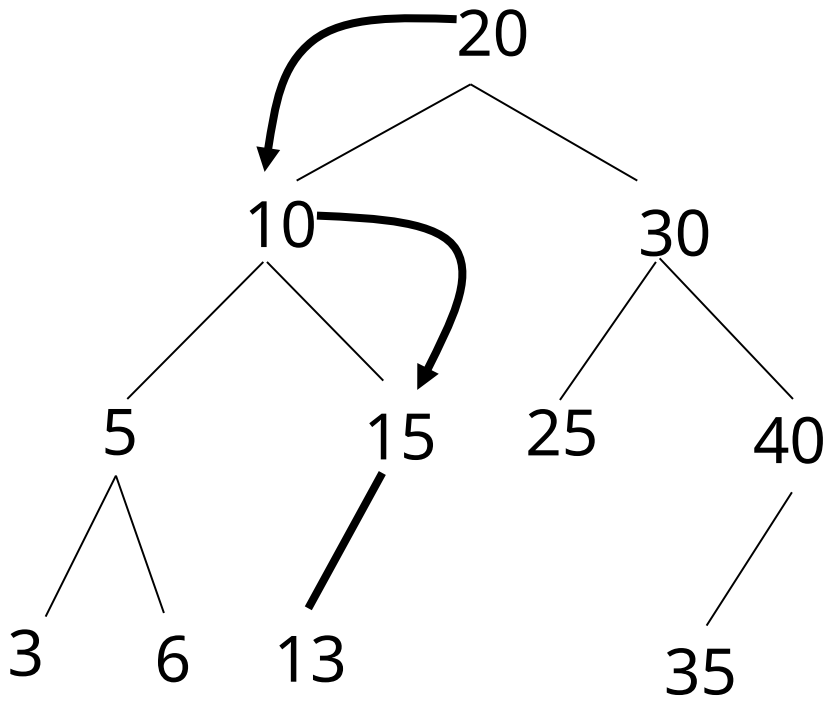
Esempio: ricerca di 15



Operazioni: insert

- Inserimento di un elemento
  - Se l'albero è vuoto allora crea un nuovo albero con un solo elemento
  - Se l'albero non è vuoto
    - se l'elemento coincide con la radice non fa niente
    - se l'elemento è minore della radice allora lo inserisce nel sottoalbero sinistro
    - se l'elemento è maggiore della radice allora lo inserisce nel sottoalbero destro

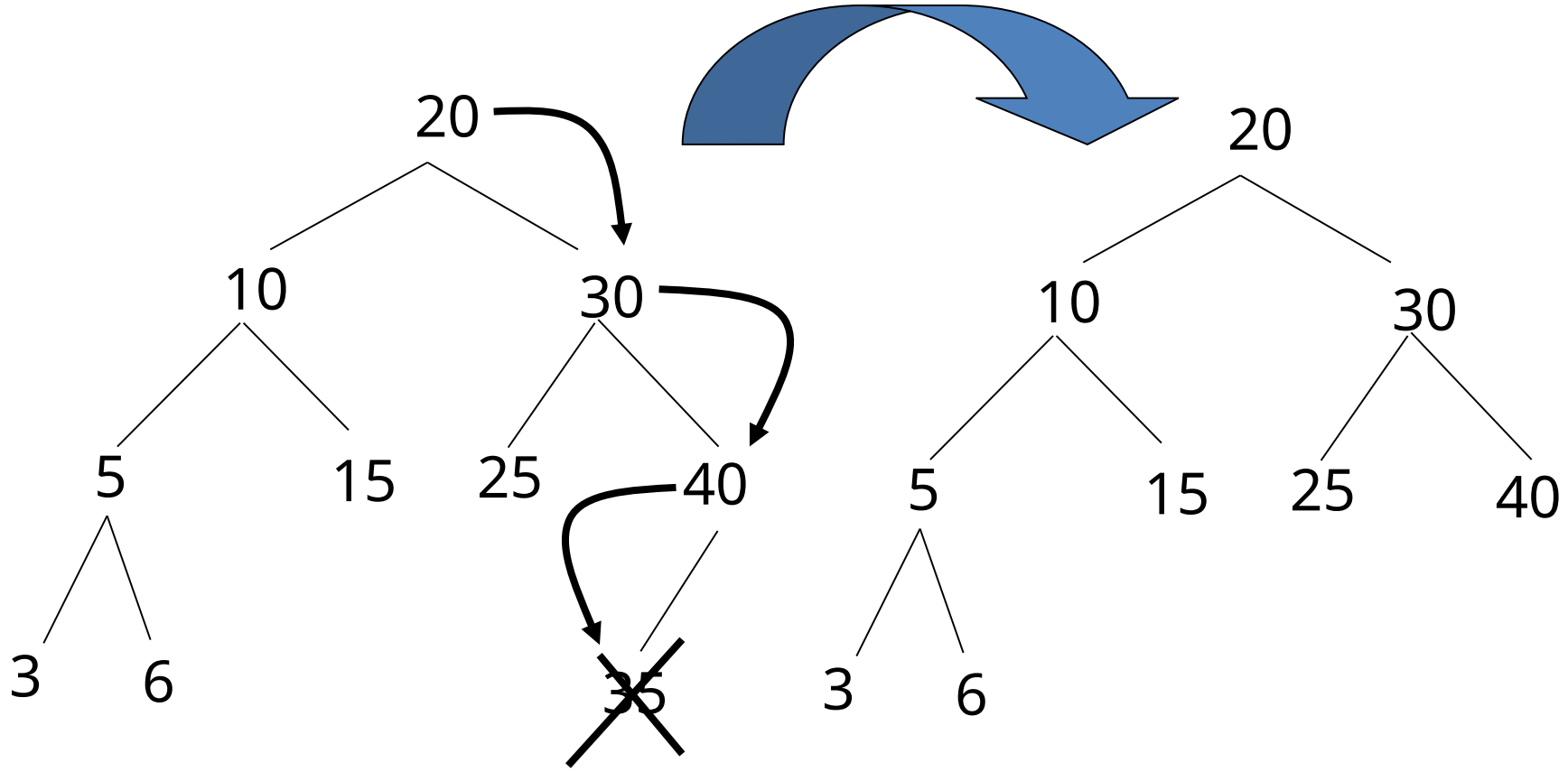
Esempio: inserimento di 13



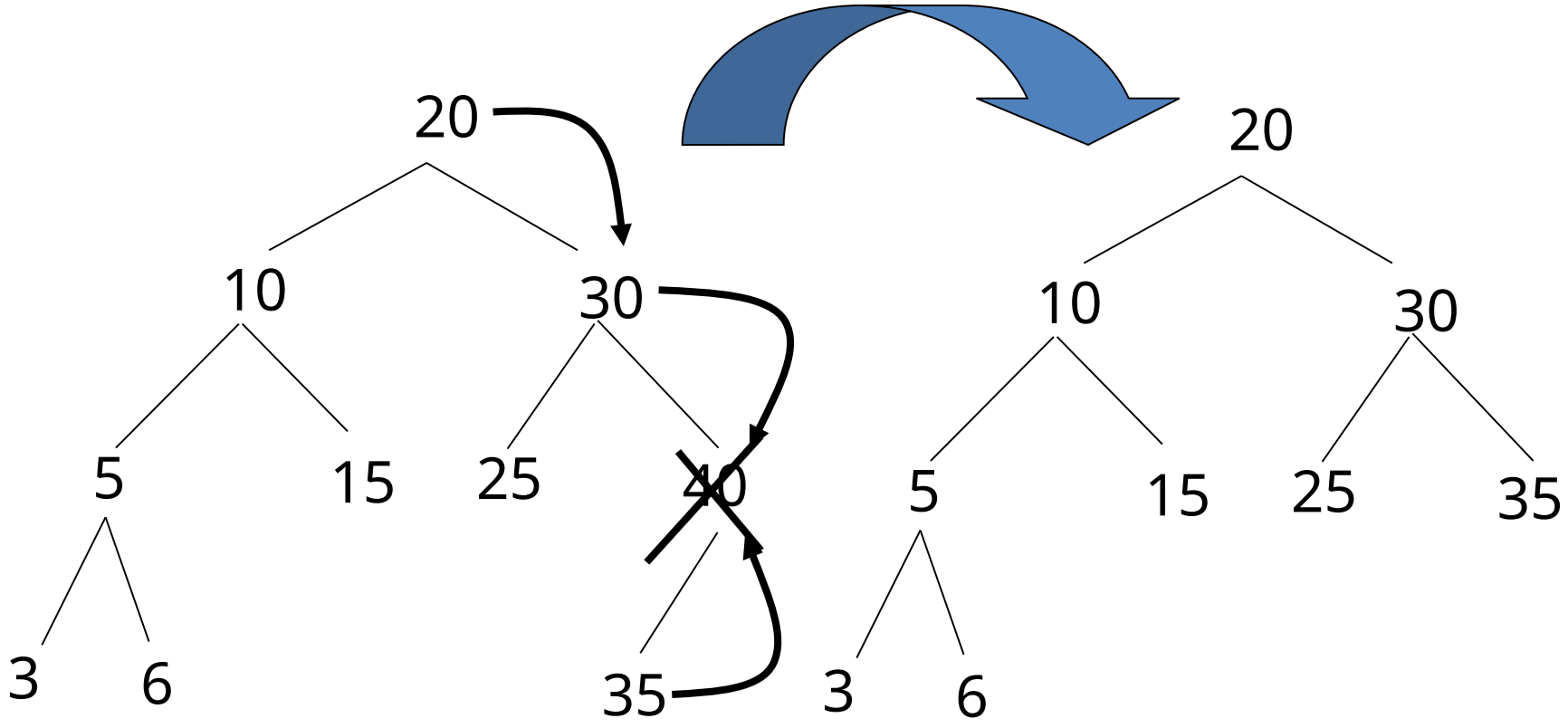
Operazioni: delete

- Cancellazione di un elemento e
  - Nessun problema se il nodo è una foglia
  - Se il nodo ha un solo sottoalbero di radice r
    - se il nodo da rimuovere è la radice allora r prende il suo posto (diventa radice dell'albero)
    - se il nodo ha un padre p, allora viene rimosso e r prende il suo posto (diventa figlio di p)

Esempio: Eliminazione di 35



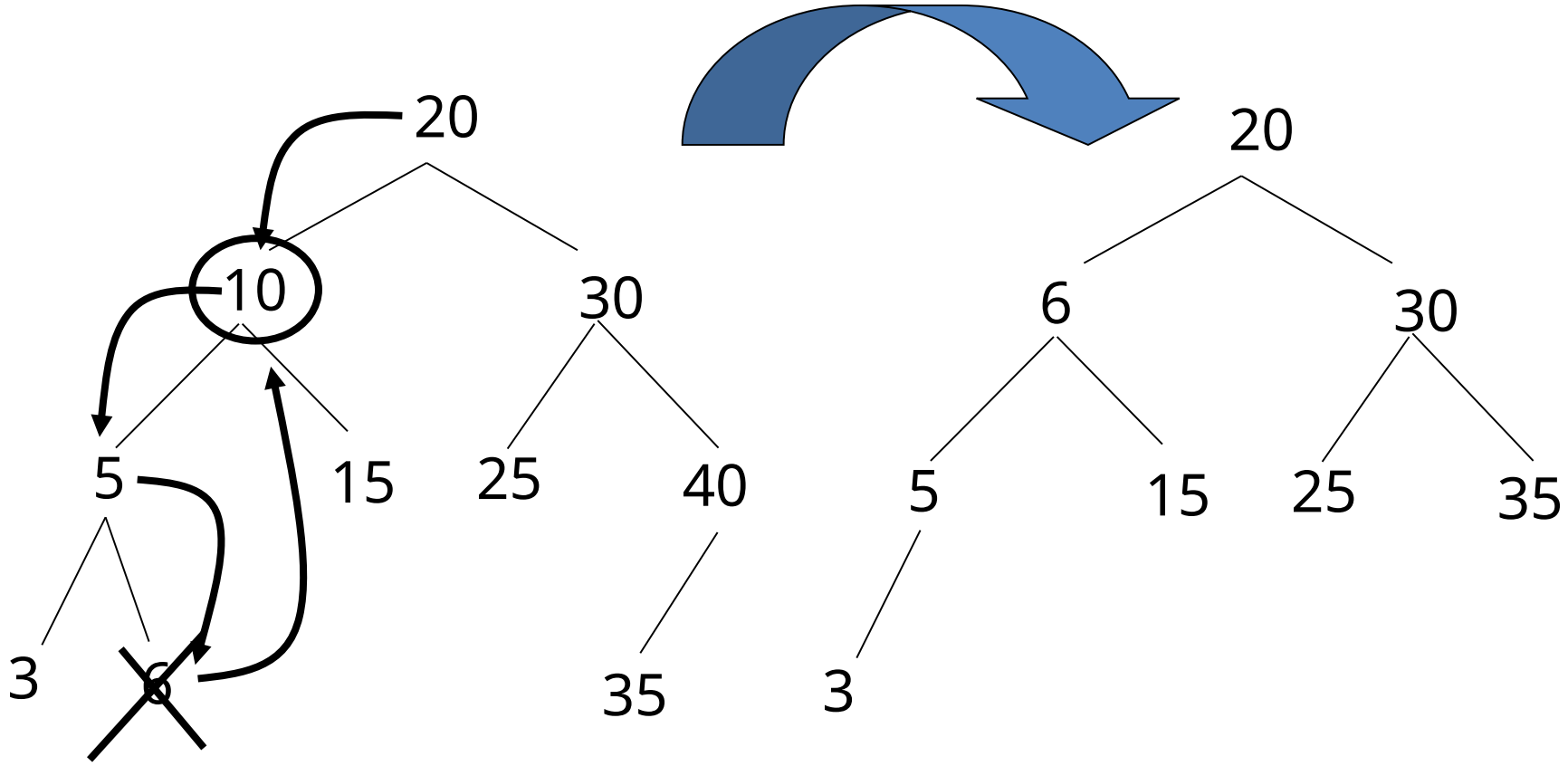
Esempio: Eliminazione di 40



Operazioni: delete

- Cancellazione di un elemento e
  - Se il nodo ha entrambi i sottoalberi
    - Si cerca l'elemento max nel sottoalbero sinistro (da notare che tale elemento non ha sottoalbero destro, la cui radice altrimenti sarebbe maggiore)
      - Alternativamente si cerca l'elemento minimo nel sottoalbero destro
    - Il nodo contenente l'elemento max viene eliminato, mentre a quello contenente l'elemento da eliminare si assegna max
  - L'albero risultante è un albero di ricerca binaria

Esempio: Eliminazione di 10



Realizzare il modulo *BST*: header file BST.h

```
// file BST.h
typedef struct node *BST;
// prototipi
BST newBST(void);
int emptyBST(BST T);
BST figlioSX(BST T);
BST figlioDX(BST T);
BST insert(BST T, item elem);
int contains(BST T, item elem);
BST deleteNode(BST T, item elem);
```

Realizzazione di BST: file BST.c

```
#include <stdio.h>
#include <stdlib.h>
#include “item.h”
#include “BST.h”


struct node {
    item value;
    struct node *left;
    struct node *right;
};

item getItem(struct node *N);
void setItem(struct node *N, item el);

item getItem(struct node *N)
{
     if (N == NULL)  return NULLITEM;
     return N->value;
}

void setItem(struct node *N, item el)
{
    if (N==NULL) return;
    N->value = el;  // correttezza di =
}                   // dipende dal tipo item
```

```
BST newBST (void)
{
    return NULL;
}

int emptyBST (BST T)
{
    return T == NULL;
}

int contains(BST T, item val)
{
     if (T == NULL)  return 0;
     if (eq(val, getItem(T)))  return 1;
     if (minore(val, getItem(T)))
          return (contains(figlioSX(T), val));
     else
          return (contains(figlioDX(T), val));
}

BST insert(BST T, item elem)
{      /* REALIZZATA IN SEGUITO */   }

BST deleteNode(BST T, item elem)
{      /* REALIZZATA IN SEGUITO */   }
```

Realizzazione di BST: file BST.c

```
BST insert(BST T, item elem)
{
   if (T==NULL)    return creaFoglia(elem);
   else   if (minore(elem, getItem(T)))
              T->left = insert(T->left, elem);
          else  if (minore(getItem(T), elem))
              T->right = insert(T->right, elem);
   return T;
}

//  deve essere usata sempre nel modo
//   bst = insert(bst, elem);
```

```
BST creaFoglia(item elem)
{
      struct node *N;
      N = malloc (sizeof(struct node));
      if (N == NULL)   return NULL;
      setItem (N, elem);
      N -> left = NULL;
      N -> right = NULL;
      return N;
}
```

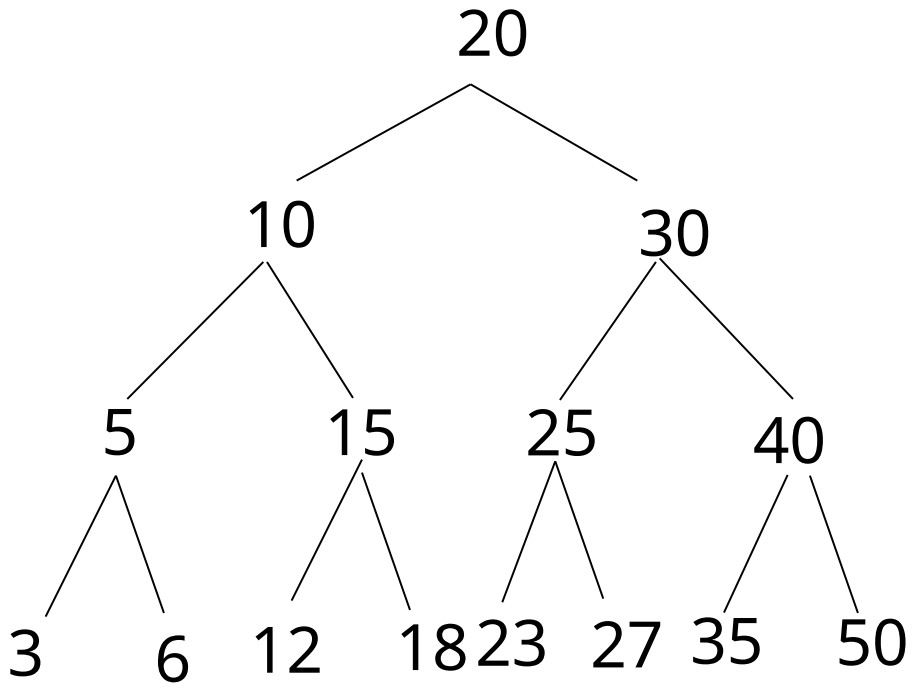
Realizzazione di deleteNode(): file BST.c

```
struct node* deleteNode(struct node* root, item key)
{
    if (root == NULL) return root;

    if (minore(key, root->value))
        root->left = deleteNode(root->left, key);
    else if (minore(root->value, key))
        root->right = deleteNode(root->right, key);

    else
    {
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }
        struct node* temp = minValue(root->right);
        root->value = temp->value;
        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->value);
    }
    return root;
}


struct node * minValue(struct node* node)
{
    struct node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}
```

Alberi perfettamente bilanciati e alberi D bilanciati

- Le operazioni sull'albero di ricerca binaria hanno complessità logaritmica se l'albero è (perfettamente) bilanciato
  - In un albero bilanciato tutti i nodi interni hanno entrambi i sottoalberi e le foglie sono a livello massimo
  - Se l'albero ha n nodi l'altezza dell'albero è $\log_2 n$
- Un albero di ricerca binaria si dice D bilanciato se per ogni nodo accade che la differenza (in valore assoluto) tra le altezze dei suoi due sottoalberi è minore o uguale a D
  - Si può dimostrare che l'altezza dell'albero è D + $\log_2 n$

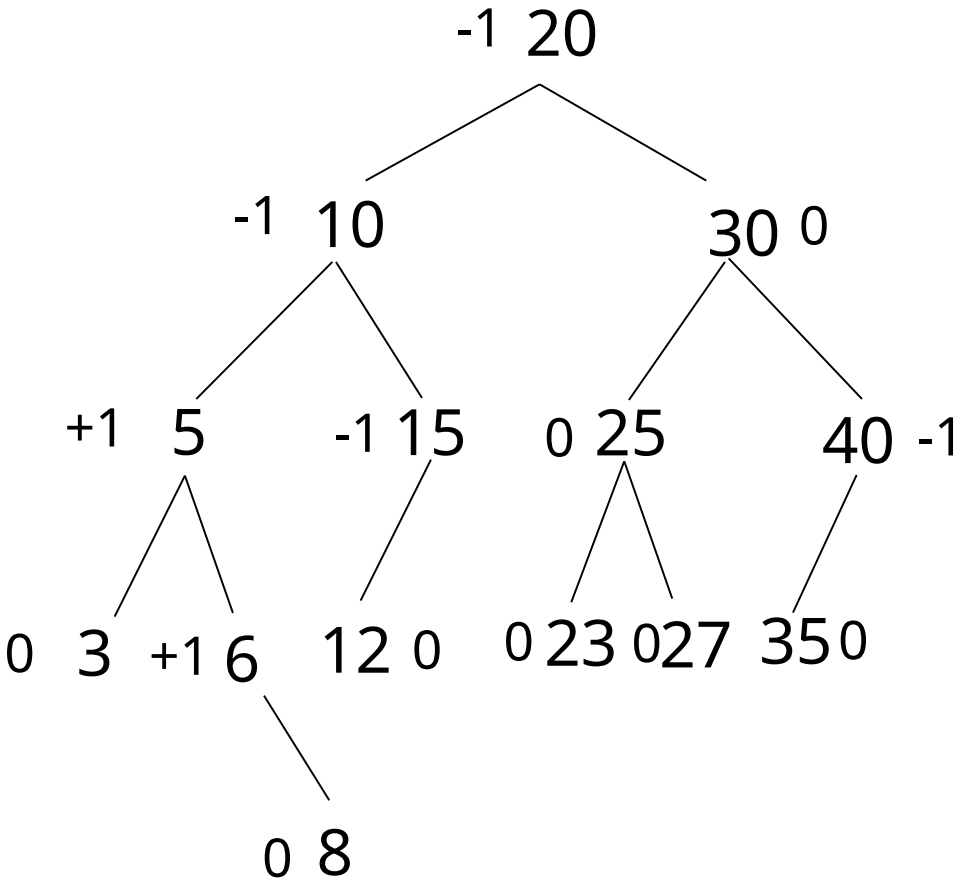
Esempio di albero perfettamente bilanciato



Alberi AVL

- Per D = 1 si parla di alberi AVL
  - Dal nome dei suoi ideatori (Adel'son, Vel'skii e Landis)
- Per prevenire il non bilanciamento ad ogni nodo bisogna aggiungere un indicatore che può assumere i seguenti valori
  - –1, se l'altezza del sottoalbero sinistro è maggiore (di 1) dell'altezza del sottoalbero destro
  - 0, se l'altezza del sottoalbero sinistro è uguale all'altezza del sottoalbero destro
  - +1, se l'altezza del sottoalbero sinistro è minore (di 1) dell'altezza del sottoalbero destro

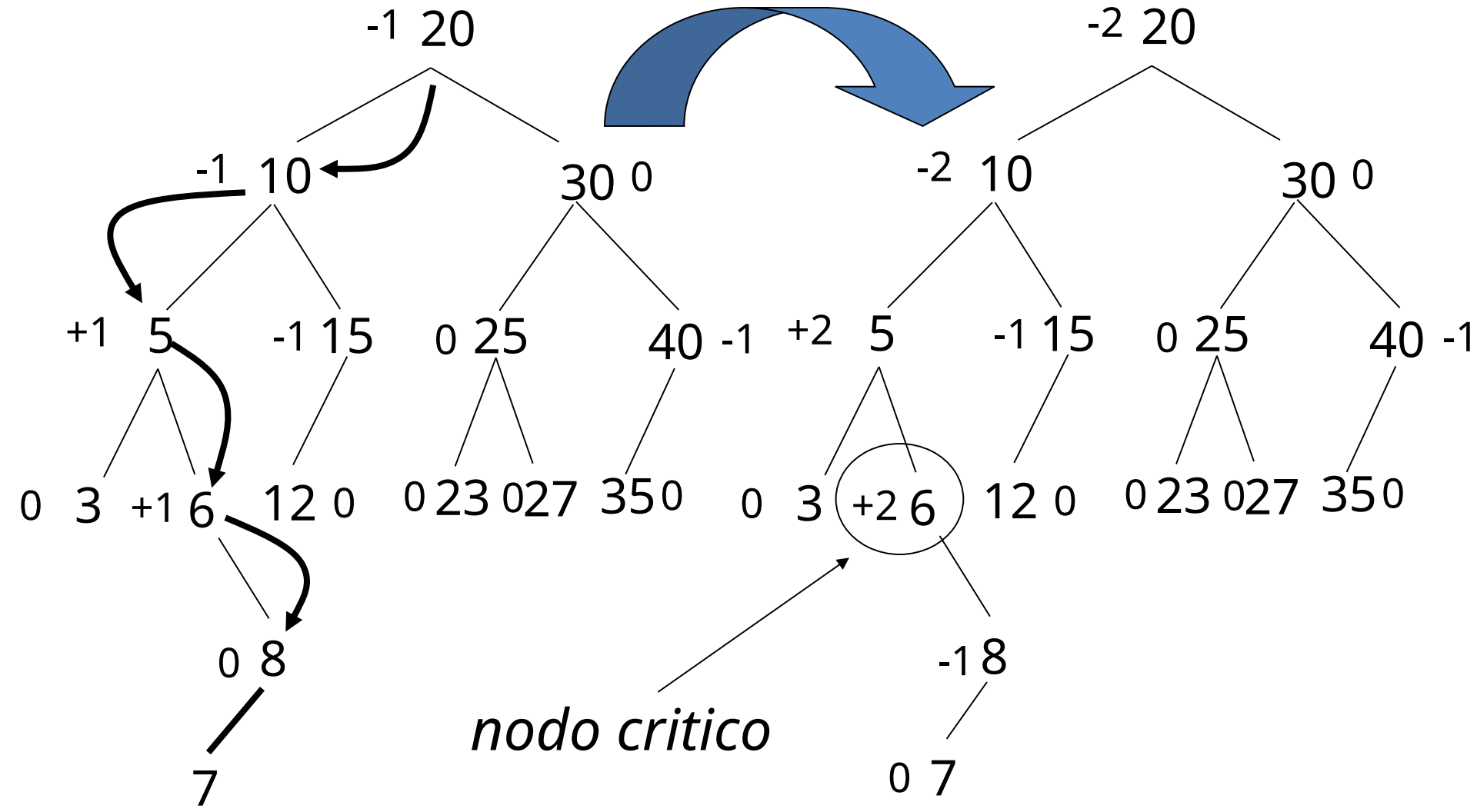
Esempio di albero AVL



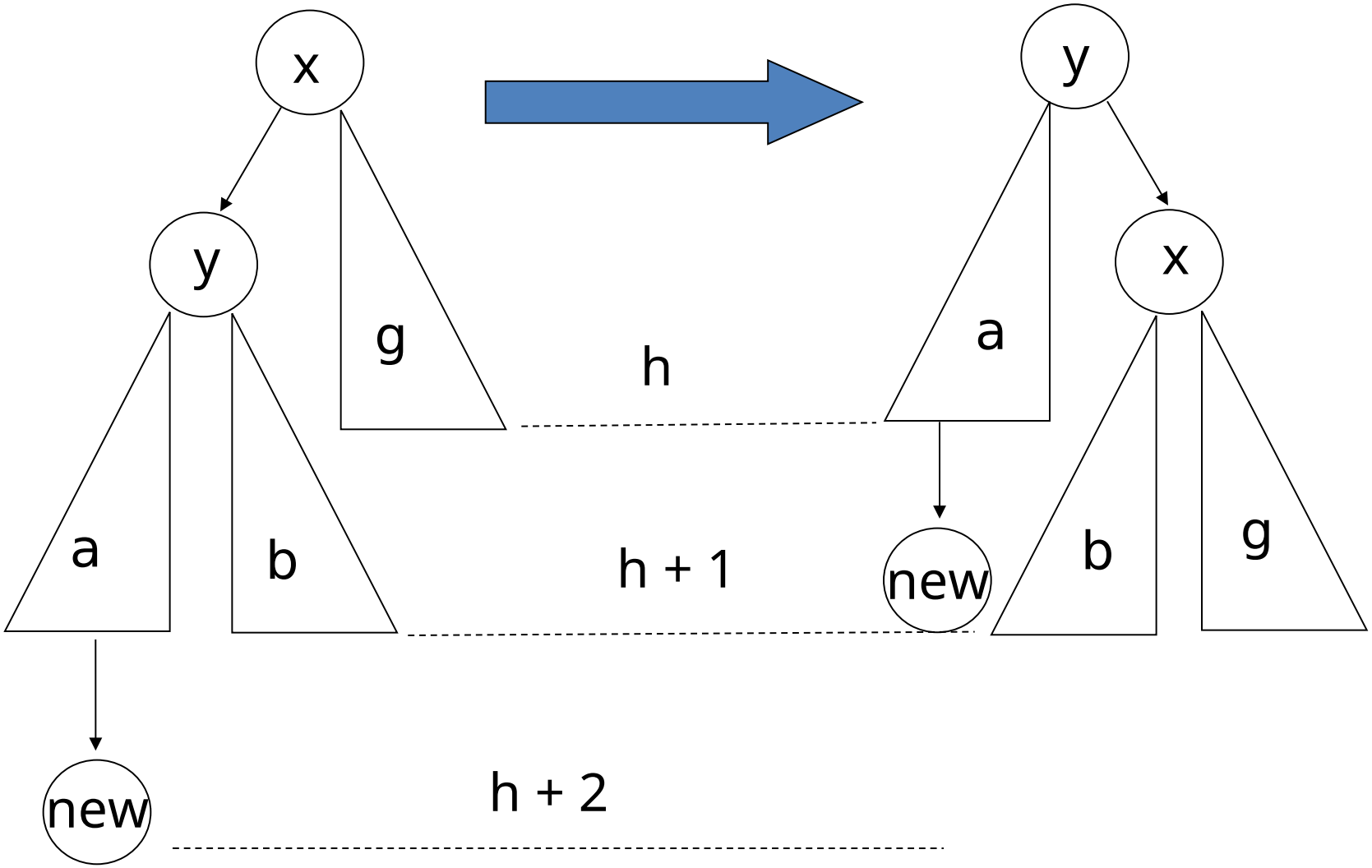
Ribilanciamento di alberi AVL

- Un inserimento di una foglia può provocare uno sbilanciamento dell'albero
  - Per almeno uno dei nodi l'indicatore non rispetta più uno dei tre stati precedenti
- In tal caso bisogna ribilanciare l'albero con operazioni di rotazione (semplice o doppia) agendo sul nodo x a profondità massima che presenta un non bilanciamento
  - Tale nodo viene detto nodo critico e si trova sul percorso che va dalla radice al nodo foglia inserito
- Considerazioni simili si possono fare anche per la rimozione di un nodo …

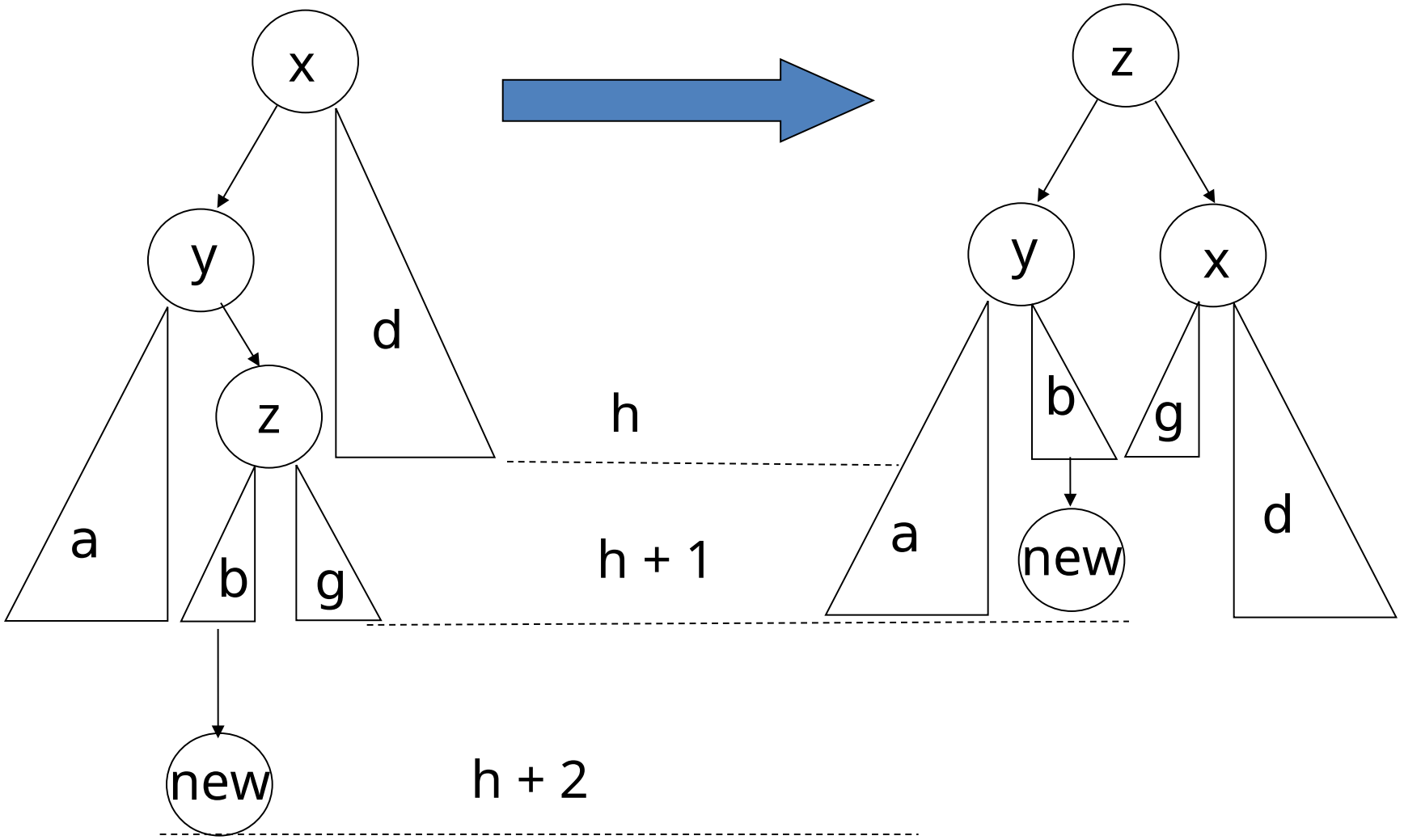
Esempio di sbilanciamento: inserimento di 7



Ribilanciamento di alberi AVL: Rotazione semplice



Ribilanciamento di alberi AVL: Rotazione doppia



Heap

- Un albero quasi perfettamente bilanciato di altezza h è un albero perfettamente bilanciato fino a livello h-1
- Un heap è un albero binario con le seguenti proprietà
  - **Proprietà strutturale**: quasi perfettamente bilanciato e le foglie a livello h sono tutte addossate a sinistra
  - **Proprietà di ordinamento**: ogni nodo v ha la caratteristica che l'informazione ad esso associata è la più grande tra tutte le informazioni presenti nel sottoalbero che ha v come radice
- Usato per realizzare code a priorità
  - Le operazioni sono inserimento di un elemento e rimozione del max

ADT Code a priorità

- Struttura dati i cui elementi sono coppie (key, value) dette *entry*, dove key e value appartengono a due insiemi qualsiasi K e V
- Le entry vengono inserite in ordine qualsiasi, ma estratte in ordine di priorità secondo il valore della key
  - **Ordinamento**: sull'insieme delle chiavi è definita una relazione d'ordine **≤**
  - **Priorità**: per convenzione, una entry E1=(k1, v1) ha priorità su E2=(k2, v2) se e solo se k2 ≤ k1
- Le operazioni fondamentali sono l'inserimento di una entry e la rimozione della entry con max priorità

Code a priorità

SPECIFICA SINTATTICA

TIPI: *PRIORITYQUEUE, BOOLEAN, ITEM*

OPERATORI:

newPQ	:	( ) → PRIORITYQUEUE
emptyPQ	:	(PRIORITYQUEUE) → BOOLEAN
getMax	:	(PRIORITYQUEUE) → ITEM
deleteMax	:	(PRIORITYQUEUE) → PRIORITYQUEUE
insertPQ	:	(PRIORITYQUEUE, ITEM) → PRIORITYQUEUE

Code a priorità

SPECIFICA SEMANTICA

TIPI:

PRIORITYQUEUE= insieme delle code a priorità, dove:
$\Lambda \in$ *PRIORITYQUEUE* (coda vuota)

BOOLEAN = {vero, falso}

ITEM = (K x V) è l'insieme delle coppie (k, v) con k∈K e v∈V

K è un insieme qualsiasi non vuoto sul quale è definita una relazione d'ordine **≤**

V è un insieme qualsiasi non vuoto

Code a priorità

SPECIFICA SEMANTICA

OPERATORI:

newPQ ( ) = PQ
pre:
post: PQ = ∧

emptyPQ (PQ) = v
pre:
post: se PQ è vuota, allora v = vero, altrimenti v = falso

getMax (PQ) = elem
pre: PQ non è vuota
post: elem è la entry con la massima priorità fra quelle contenute in PQ

Code a priorità

SPECIFICA SEMANTICA

OPERATORI:

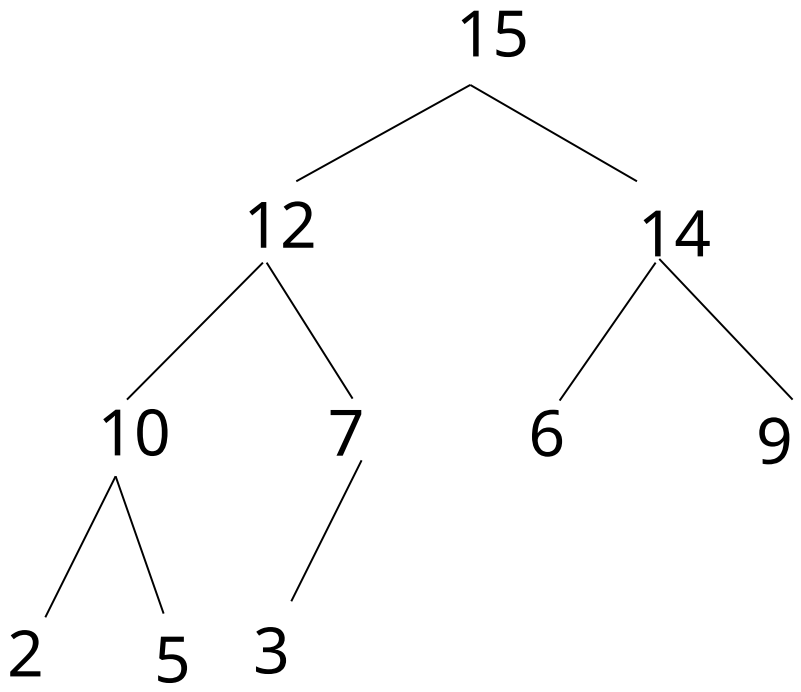
deleteMax (PQ) = PQ'
pre: PQ non è vuota
post: PQ' contiene tutte le entry di PQ tranne quella con massima priorità

insertPQ (PQ, elem) = Q'
pre:
post: PQ' contiene *elem* e tutte le entry contenute in PQ

Realizzazione di code a priorità tramite heap

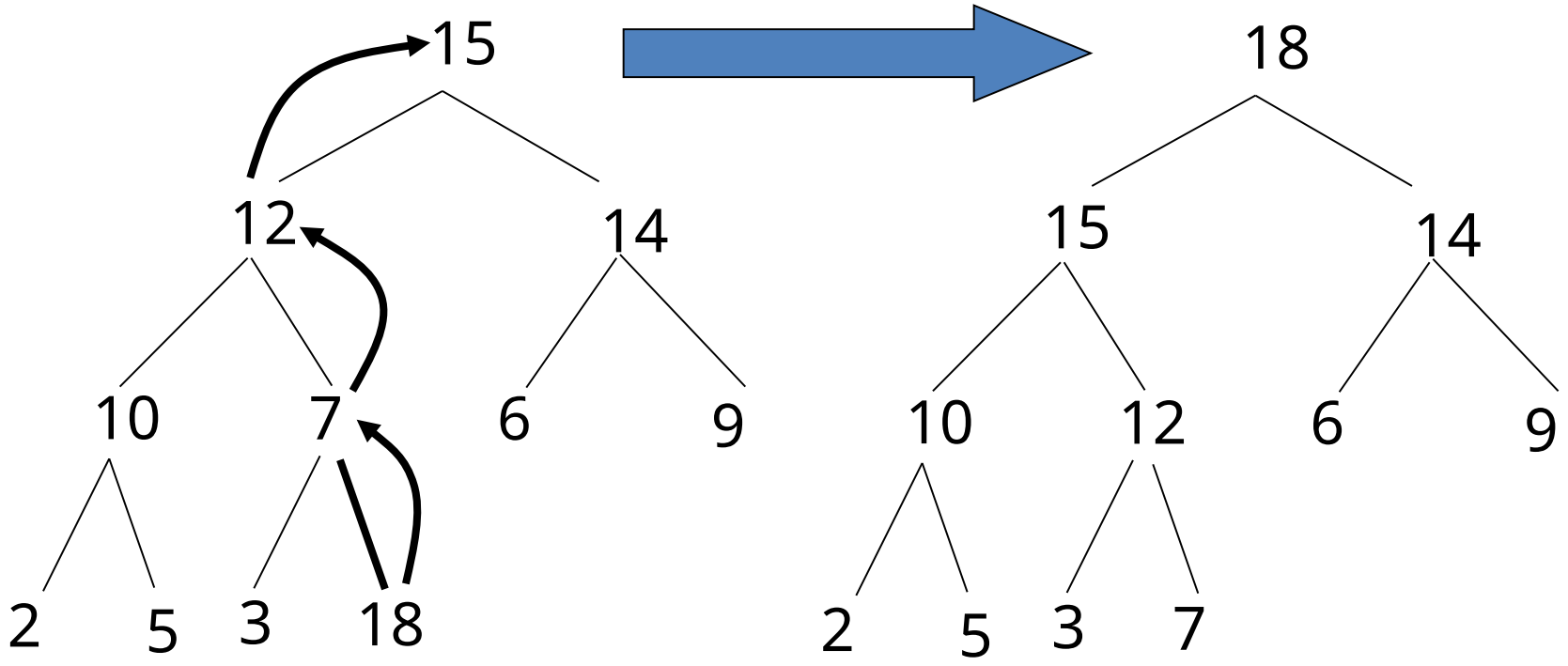
- Per semplicità, supponiamo che le chiavi siano valori interi
- Negli esempi e nel codice seguente vengono mostrate e considerate solo le chiavi, senza il valore ad esse associato

Esempio di Heap



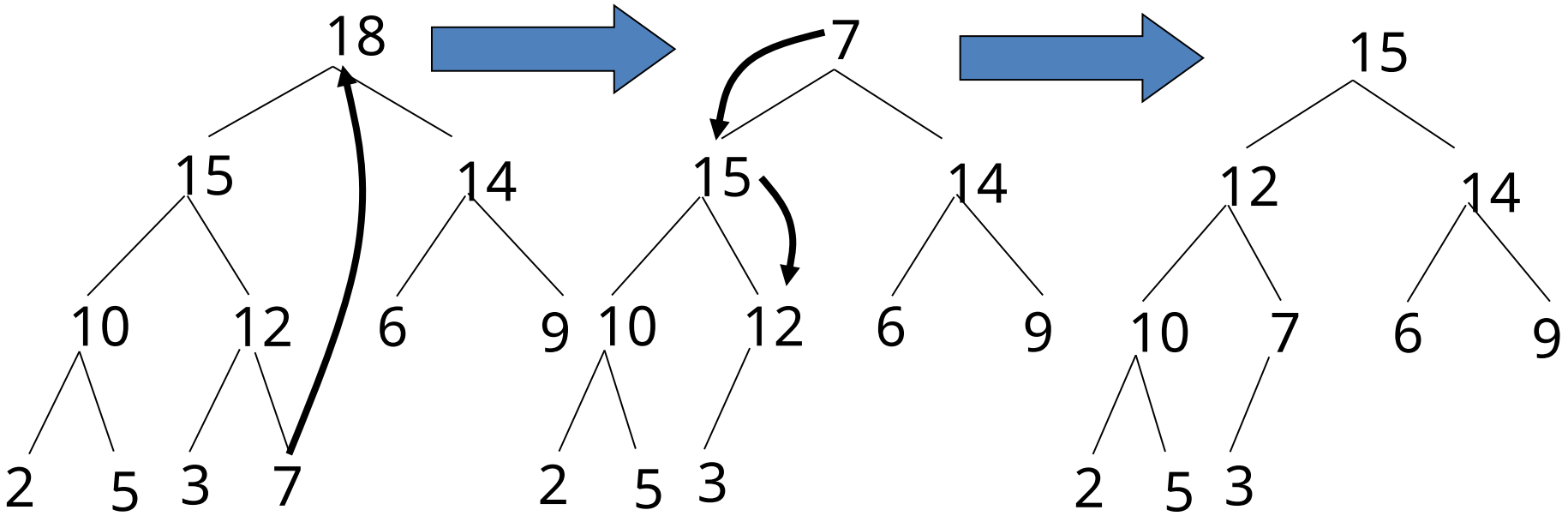
Inserimento

- Si inserisce il nuovo nodo come ultima foglia
- Il nodo inserito risale lungo il percorso che porta alla radice per individuare la posizione giusta (eventualmente fino alla radice)
- Esempio: inserimento di 18



Rimozione

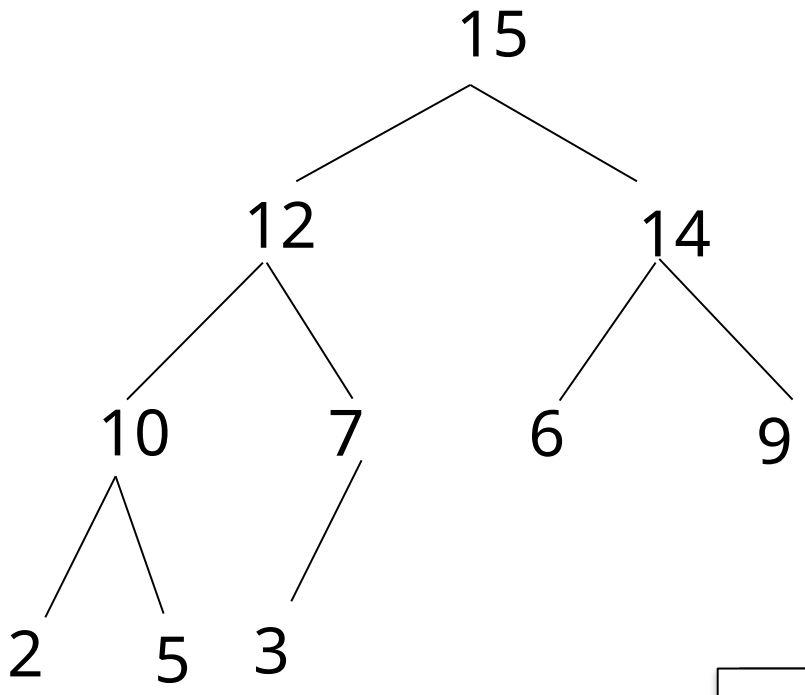
- Si rimuove sempre la radice (la chiave maggiore, in quanto un heap viene usato per realizzare code a priorità) e si pone l'ultima foglia al posto della radice
- Se la chiave della radice è più piccola di quella dei suoi figli, si scambia con il maggiore dei due, e si ripete il procedimento finché non si arriva alla condizione in cui il nodo corrente o non ha figli oppure i suoi figli hanno chiavi più piccole



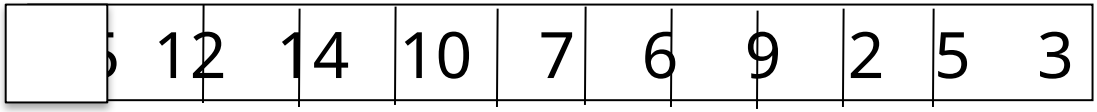
Realizzazione di un heap

- Per le sue caratteristiche un heap può essere realizzato con un array
  - I nodi sono disposti nell'array per livelli
    - La radice occupa la posizione 1
    - Se un nodo occupa la posizione i, il suo figlio sinistro occupa la posizione 2*i e il suo figlio destro occupa la posizione 2*i+1

Esempio di Heap



rappresentazione



Implementazione di *Code a priorità* con heap rappresentato con un array

```
// file PQueue.h

typedef struct c_PQ *PQueue;

// prototipi

PQueue newPQ(void);

int emptyPQ(PQueue q);

int getMax(PQueue q);

int deleteMax(PQueue q);

int insert (PQueue q, int key);
```

L'ADT coda a priorità è realizzato in maniera semplificata.

Vengono inserite solo chiavi di tipo intero, senza valori associati

Gli operatori *insert()* e *deleteMax()* restituiscono un valore 0 se l'operazione fallisce, 1 se termina con successo

file PQueue.c

```
#include <stdio.h>
#include <stdlib.h>
#include “PQueue.h”
#define MAXPQ 50

struct c_PQ {
    int vet[MAXPQ];
    int numel;
};

static void scendi (PQueue q);
static void sali (PQueue q);

PQueue newPQ(void)
{
    PQueue q;
    q = malloc (sizeof(struct c_PQ));
    if (q == NULL) return NULL;
    q->numel = 0;
    return q;
}
```

```
int emptyPQ(PQueue q)
{
    if (!q)  return 1;
    return q->numel == 0;
}


int getMax(PQueue q)
{
    return q->vet[1];
    // NON VERIFICA LA
    // PRECONDIZIONE
    // LA CODA NON PUO’
    // ESSERE VUOTA
}
```

file PQueue.c

```
int deleteMax(PQueue q)
{
    if (!q || q->numel==0) return 0;   // CODA VUOTA

    q->vet[1] = q->vet[q->numel];  //SPOSTO L'ULTIMO ELEMENTO
    q->numel --;                   // IN POSIZIONE 1

    scendi(q);      // RIAGGIUSTO LO HEAP SCENDENDO

    return 1;
}
```

```
static void scendi (PQueue q)
{
   int temp, n=q->numel, i=1, pos;

   while (1)
   {
     if (2*i+1 <= n)              // IL NODO CORRENTE HA 2 FIGLI
            pos =  (q->vet[i*2] > q->vet[i*2+1])  ?   i*2  :   i*2+1;

     else  if (2*i <= n)          // IL NODO CORRENTE HA 1 FIGLIO
            pos = 2*i;
     else  break;                 // IL NODO CORRENTE NON HA FIGLI

     if (q->vet[pos] > q->vet[i])     // SCAMBIO LE CHIAVI E PROSEGUO
     {
          temp = q->vet[i];
          q->vet[i] = q->vet[pos];
          q->vet[pos] = temp;
          i = pos;
     }
     else
         break;   // NON CI SONO PIU' SCAMBI DA FARE, MI FERMO
    }
}
```

file PQueue.c

```
int insert (PQueue q, int key)
{

    if (!q || q->numel==MAXPQ) return 0; // CODA PIENA

    q->numel++;
    q->vet[q->numel] = key;   // INSERISCI IN ULTIMA POSIZIONE

    sali(q);     // AGGIUSTA LO HEAP RISALENDO

    return 1;
}
```

```
static void sali (PQueue q)
{
   int temp, pos=q->numel, i=pos/2;

   while (pos>1)
   {

      if (q->vet[pos] > q->vet[i])
      {
            temp = q->vet[i];
            q->vet[i] = q->vet[pos];
            q->vet[pos] = temp;
            pos = i;
            i = pos/2;
      }

      else
            break;
    }
}
```