

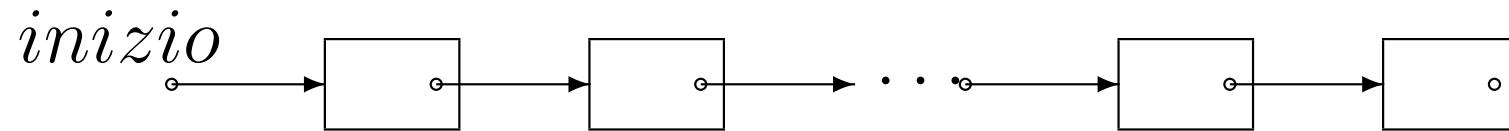
Algoritmi con liste e alberi

Integrazioni

(Roberto Montemanni)

Liste

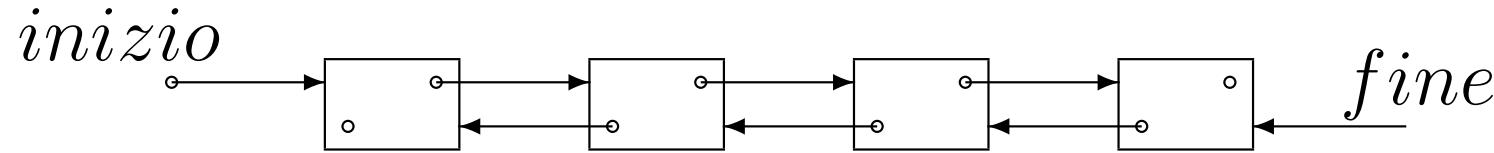
Lista a concatenamento semplice



Caso particolare: Pila

inserimento e prelevamento in testa

Lista a concatenamento doppio



Caso particolare: Coda

inserimento dietro e prelevamento davanti

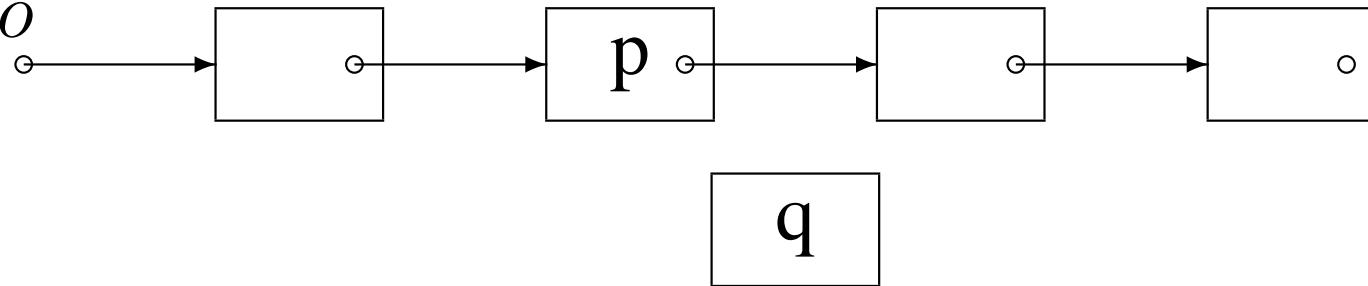
Struttura dati per lista

```
type elemento ;
type tipo_contenuto is ... ;
type lista      is access elemento ;
type elemento is record
    Contenuto  : tipo_contenuto ;
    Prossimo   : lista ;
end record ;
```

Liste semplici: inserimento

Inserimento *dopo* un elemento dato

inizio



```
q.prossimo := p.prossimo ;  
p.prossimo := q ;
```

in una pila

```
q.prossimo := inizio ;  
inizio := q ;
```

Liste semplici: inserimento

Inserimento *in testa*

```
q.prossimo := inizio ;  
inizio := q ;
```

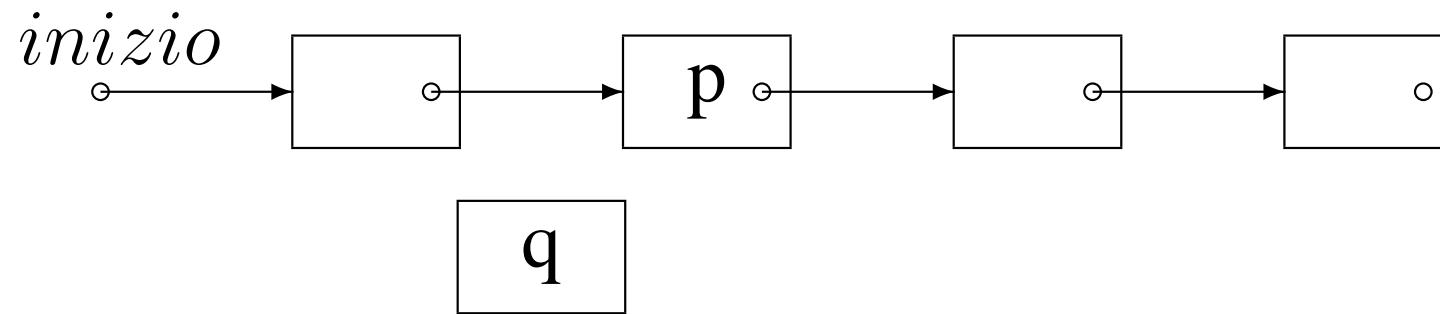
Costruzione di una lista con n elementi

```
inizio := null ;  
while n > 0 loop  
    q := new elemento'(..., inizio);  
    inizio := q ;  
    n := n - 1 ;  
end loop ;
```

Liste semplici: inserimento

Inserimento *prima*

Trucco: inserire q dopo p e scambiare contenuti



```
z.all := q.all ;
q.all := p.all ;
p.Contenuto := z.Contenuto ;
p.prossimo := q ;
```

Liste semplici: eliminazione

Eliminazione *del successore*

Salvare indirizzo di p in z , agganciare p al prossimo

```
if p.prossimo /= null then
    z := p.prossimo ;
    p.prossimo := z.prossimo ;
end if ;
```

Funziona se non esistono ulteriori puntatori a p !

Liste semplici: eliminazione

Eliminazione *dell'elemento attuale*

Ricopiare il contenuto del successore in p

```
z := p.prossimo ;
p.all := z.all ;
```

Non funziona se p è l'ultimo della lista.

Liste semplici: ricerca

```
p := inizio ;  
while p /= null and then p.Contenuto /= cercato loop  
  p := p.prossimo ;  
end loop ;
```

Alla fine p contiene l'indirizzo cercato o null.

Alternativa

```
flag := true ;  
while p /= null and flag loop  
  if p.Contenuto = cercato then  
    flag := false ;  
  else  
    p := p.prossimo ;  
  end loop ;
```

Trees

- Linear access time of linked lists is prohibitive
 - Does there exist any simple data structure for which the running time of most operations (search, insert, delete) is $O(\log N)$?

Trees

- A tree is a collection of nodes
 - The collection can be empty
 - (recursive definition) If not empty, a tree consists of a distinguished node r (the *root*), and zero or more nonempty *subtrees* T_1, T_2, \dots, T_k , each of whose roots are connected by a directed *edge* from r

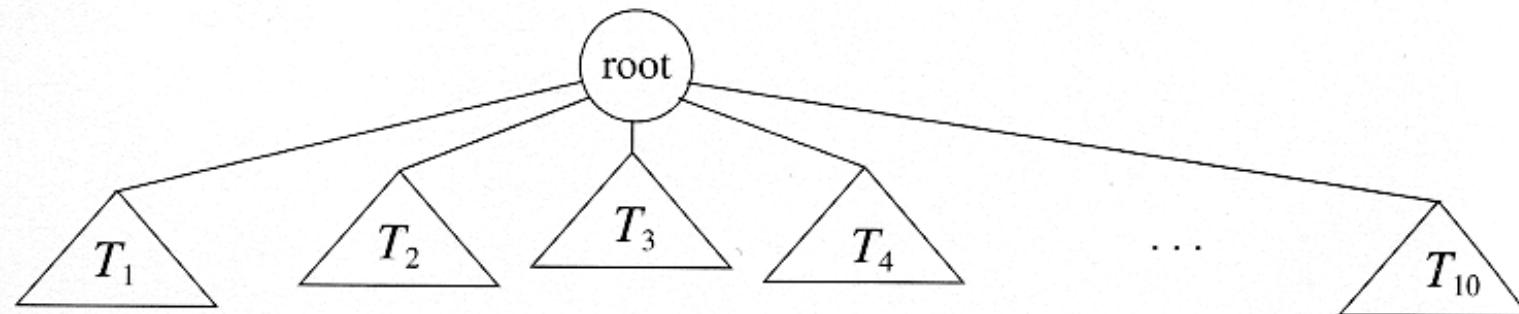


Figure 4.1 Generic tree

Some Terminologies

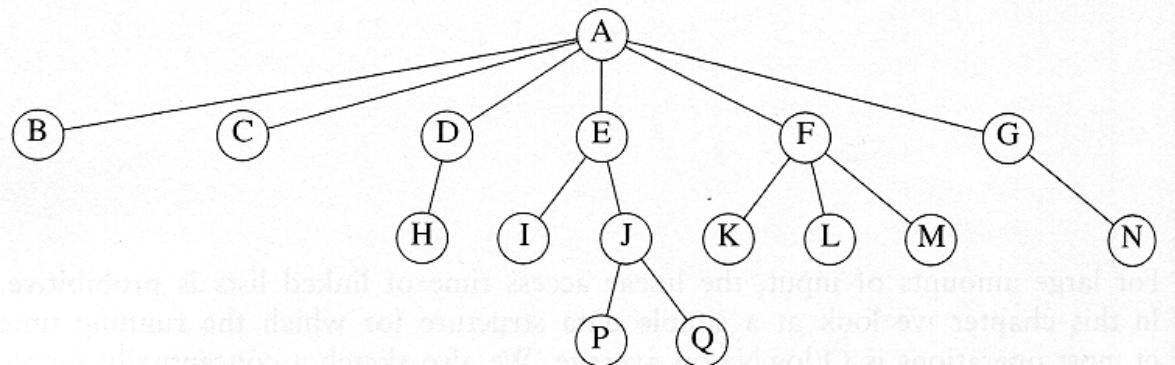


Figure 4.2 A tree of 15 nodes illustrating various terminologies.

- *Child and parent*
 - Every node except the root has one parent
 - A node can have an arbitrary number of children
- *Leaves*
 - Nodes with no children
- *Depth of a node*
 - length of the unique path from the root to that node
 - The depth of a tree is equal to the depth of the deepest leaf
- *Ancestor and descendant*
 - Proper ancestor and proper descendant

Example: UNIX Directory

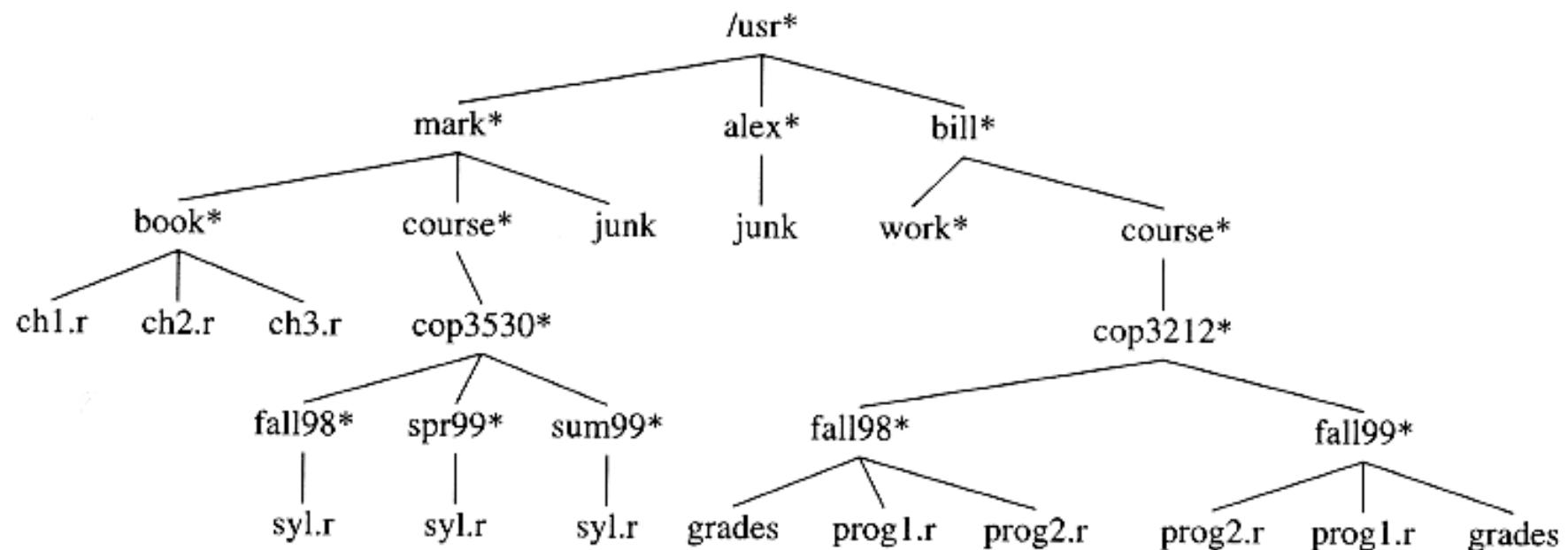
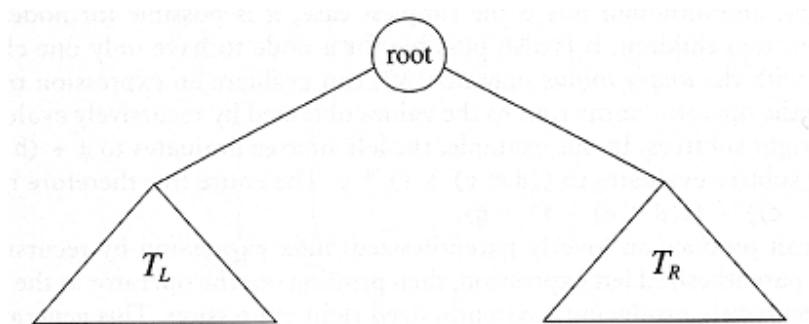


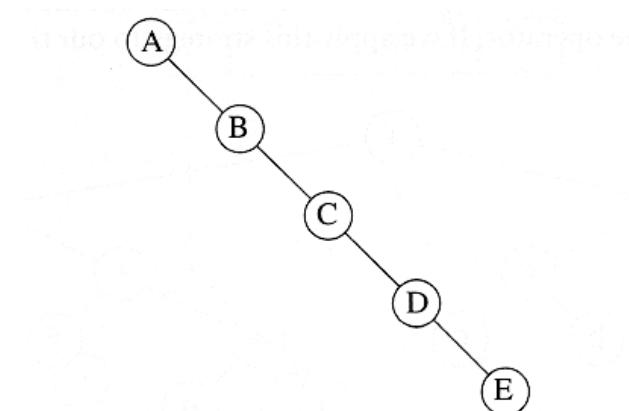
Figure 4.5 UNIX directory

Binary Trees

- A tree in which no node can have more than two children



- The depth of an “average” binary tree is considerably smaller than N , even though in the worst case, the depth can be as large as $N - 1$.



Example: Expression Trees

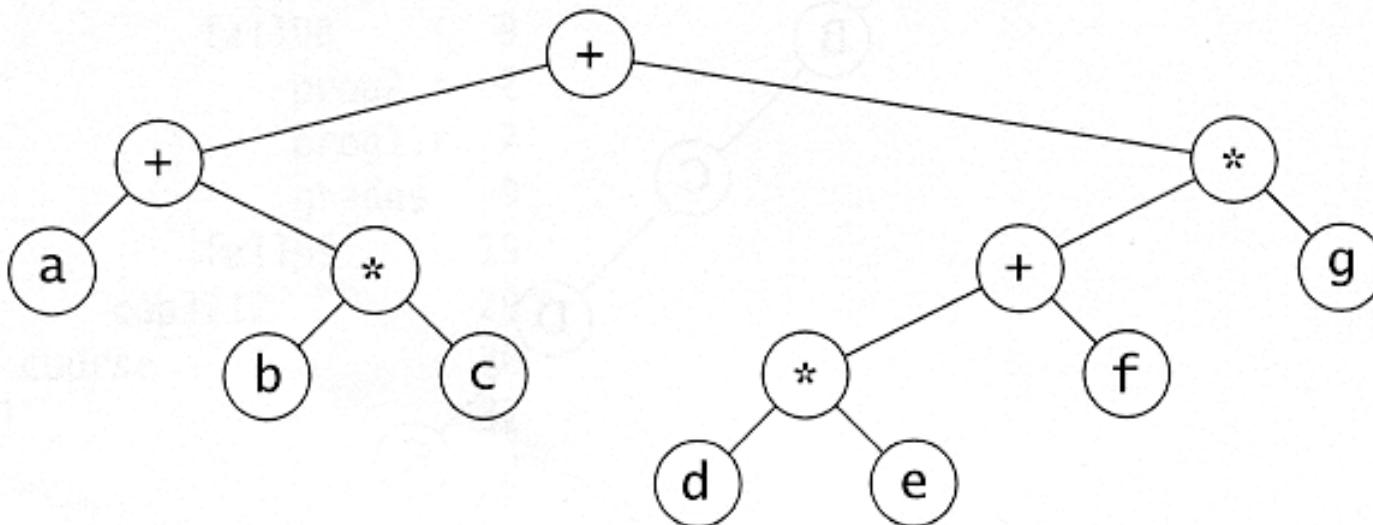


Figure 4.14 Expression tree for $(a + b * c) + ((d * e) * f) * g$

- Leaves are operands (constants or variables)
- The other nodes (internal nodes) contain operators
- Will not be a binary tree if some operators are not binary

Tree traversal

- Used to print out the data in a tree in a certain order
- Pre-order traversal
 - Print the data at the root
 - Recursively print out all data in the left subtree
 - Recursively print out all data in the right subtree

Preorder, Postorder and Inorder

- Preorder traversal
 - node, left, right
 - prefix expression
 - $\text{++a}^*\text{bc}^*+^*\text{defg}$

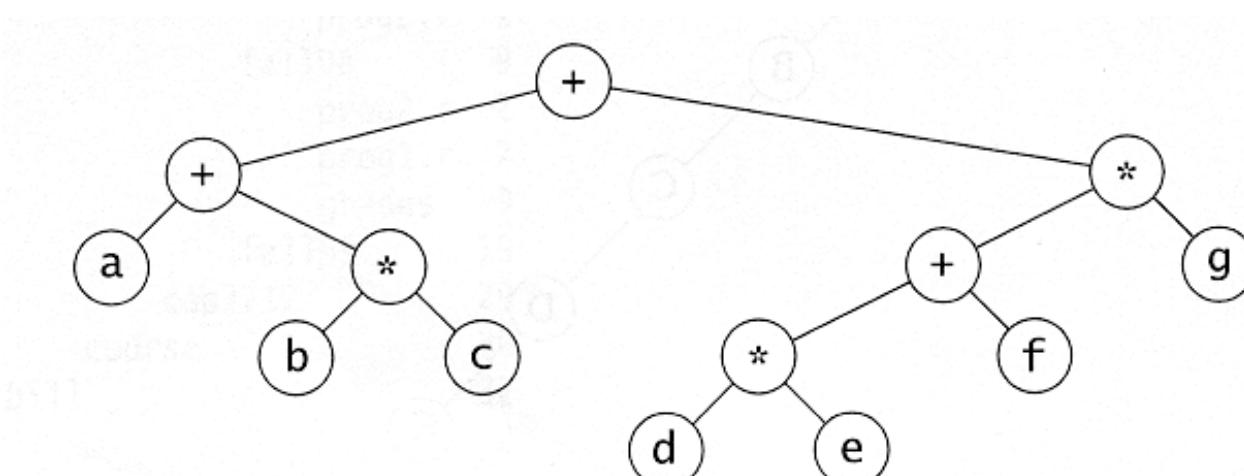


Figure 4.14 Expression tree for $(a + b * c) + ((d * e + f) * g)$

Preorder, Postorder and Inorder

- Postorder traversal
 - left, right, node
 - postfix expression
 - $abc^*+de^*f+g^*+$
- Inorder traversal
 - left, node, right.
 - infix expression
 - $a+b*c+d*e+f*g$

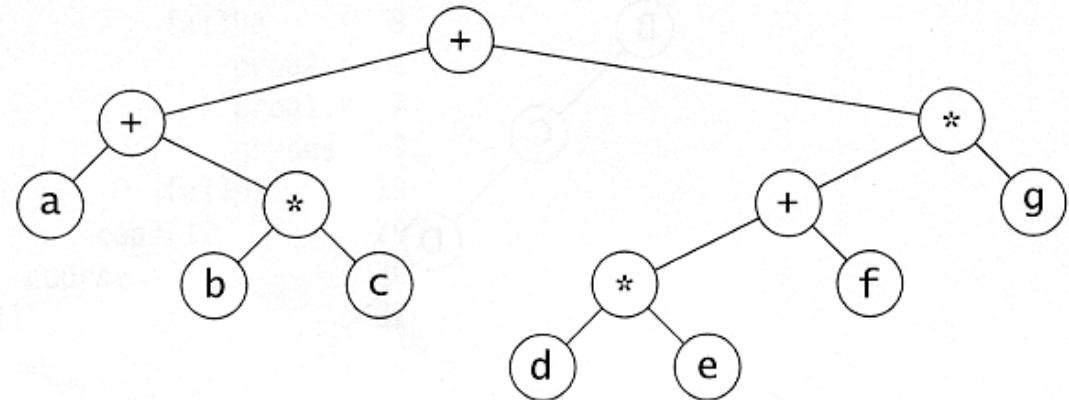


Figure 4.14 Expression tree for $(a + b * c) + ((d * e) + f) * g$

Preorder, Postorder and Inorder

Algorithm *Preorder*(x)

Input: x is the root of a subtree.

1. **if** $x \neq \text{NULL}$
2. **then** output key(x);
3. *Preorder*(left(x));
4. *Preorder*(right(x));

Algorithm *Postorder*(x)

Input: x is the root of a subtree.

1. **if** $x \neq \text{NULL}$
2. **then** *Postorder*(left(x));
3. *Postorder*(right(x));
4. output key(x);

Algorithm *Inorder*(x)

Input: x is the root of a subtree.

1. **if** $x \neq \text{NULL}$
2. **then** *Inorder*(left(x));
3. output key(x);
4. *Inorder*(right(x));

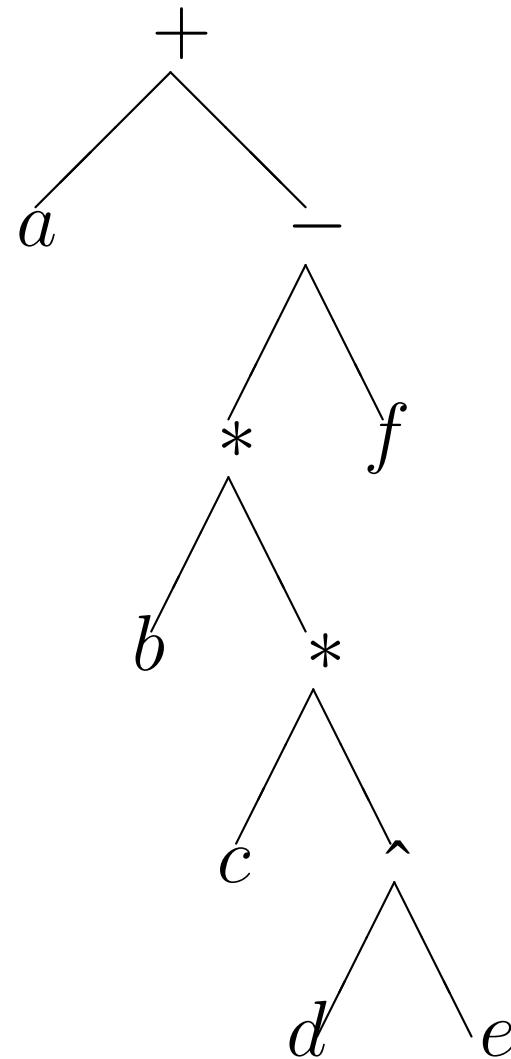
Applicazioni degli alberi

Esempi:

- **Rappresentazione di formule algebriche**
 - Formule costituite da:
 - operatori binari (+,-,*,/, $\hat{}$)
 - operandi che sono ancora formule
 - Struttura dinamica, facilmente ampliabile
- **Alberi di ricerca**
 - Struttura per ricerca con numero grande di elementi
 - Struttura dinamica, facilmente ampliabile

Alberi e formule algebriche

Esempio 1



$$a + bcd^e - f$$

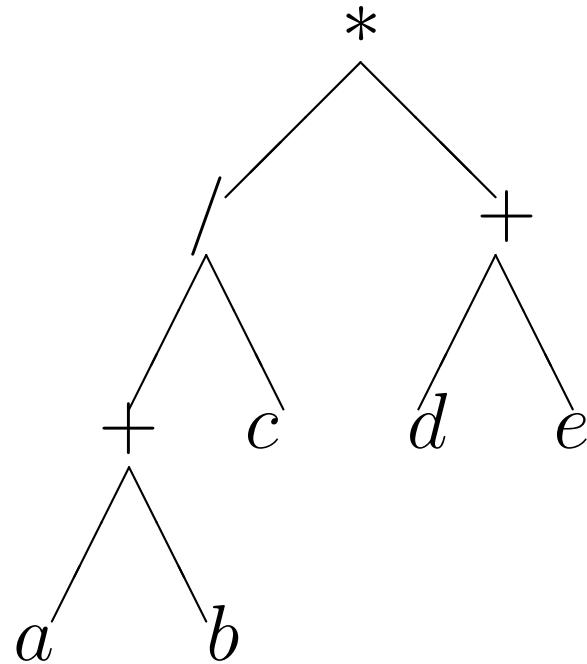
Problemi:

Albero \leftarrow Formula

Albero \rightarrow Formula

Alberi e formule algebriche

Esempio 2



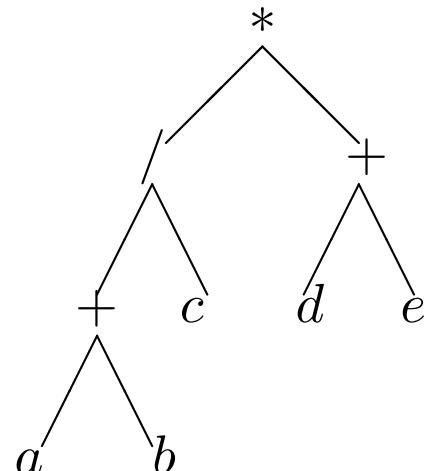
$$(a + b)/c * (d + e)$$

Problema
Gestione parentesi

Albero → Formula

Attraversamento sistematico. Per ogni nodo:

1. aprire parentesi (eccezione: inizio)
2. visita sottoalbero sinistro
3. visita radice
4. visita sottoalbero destro
5. chiudere parentesi (eccezione: fine)



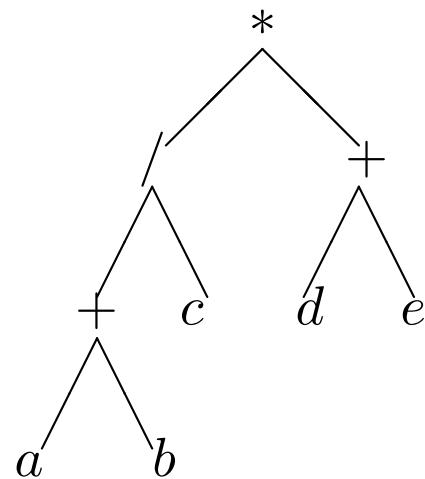
$$((a+b)/c)*(d+e)$$

Attraversamento in order

Albero → Formula

Alternativa Per ogni nodo:

1. visita sottoalbero sinistro
2. visita sottoalbero destro
3. visita radice



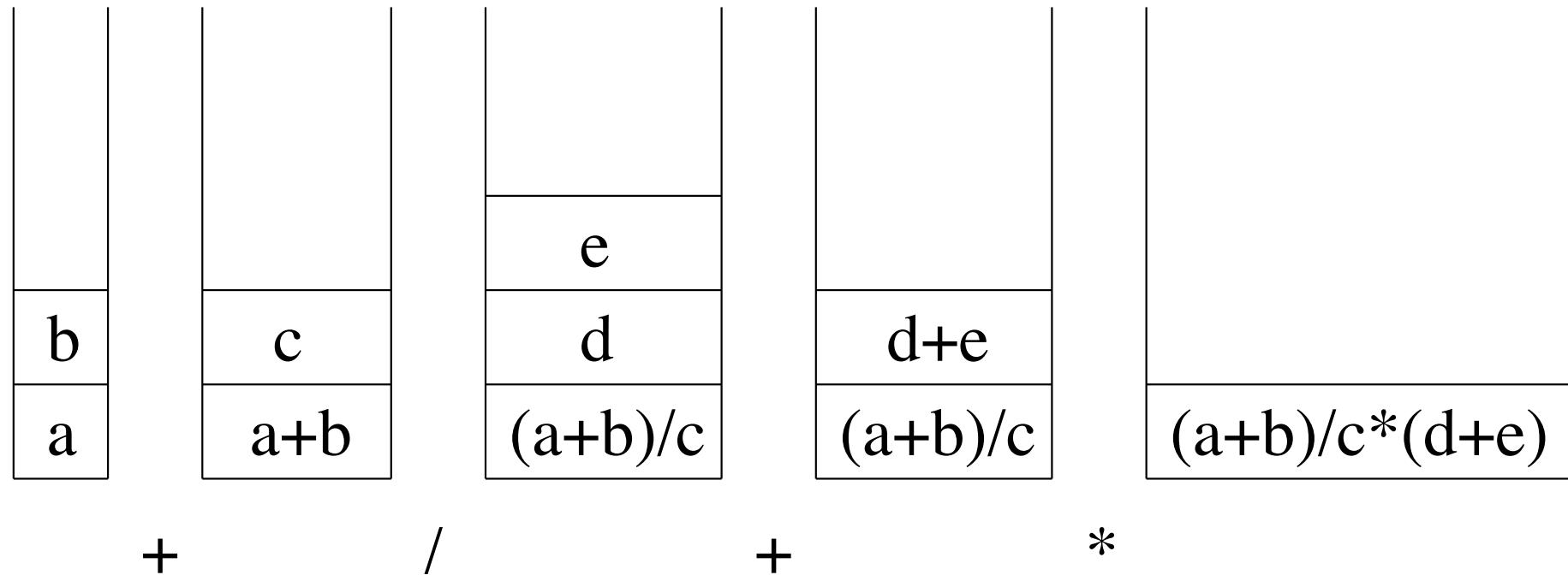
ab+c/de+*

Attraversamento post order
Notazione polacca inversa
(Postfix notation)

Notazione polacca inversa

Calcolatrice a stack:

$ab+c/de+*$

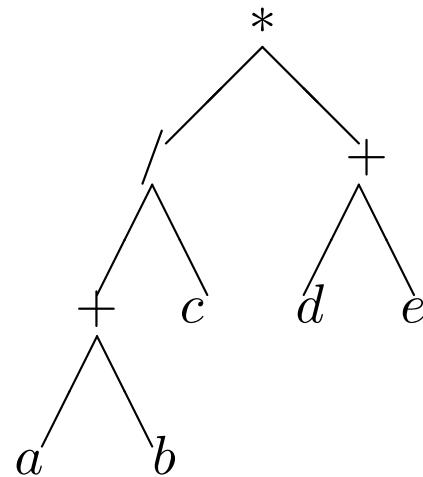


www.erikoeest.dk/rpn.htm

Albero → Formula

Terza possibilità:

1. visita radice
2. visita sottoalbero sinistro
3. visita sottoalbero destro



$*/+abc+de$

Attraversamento pre order
Notazione polacca
(Prefix notation)

Albero ← Formula

```
function albero_funzione (e : string) return albero is
    OK  : boolean ;
    ind : albero ;

begin
    OK := true ;
    genera_albero(ind , e) ;
    if OK then
        return ind ;
    else
        return null ;
    end if ;
end albero_funzione ;
```

Albero ← Formula

```
procedure genera_albero (i: in out albero ; Stringa: string) is

k          : integer ;

begin
  if OK and Stringa'length > 0 then
    if not (elabora_operatore('+','-',Stringa) or else
              elabora_operatore('*', '/',Stringa) or else
              elabora_operatore('^',' ',Stringa) or else
              elabora_funzione(Stringa) or else
              elabora_numero(Stringa) or else
              elabora_pi(Stringa) or else
              elabora_x(Stringa)) then
      OK := false ;
    end if ;
  else
    OK := false ;
  end if ;
end genera_albero ;
```

Albero ← Formula

```
function elabora_operatore
  (Stringa_1,Stringa_2:character; Stringa:string) return boolean is

{eliminare eventuale coppie di parentesi pi\u00f9 esterne}
  p :=0 ;
  k := Stringa'length +1
  loop
    k := k-1 ;
    case Stringa(k)
      when '(' => p := p+1;
      when ')' => p := p-1;
      when others
        if p = 0 and {Stringa(k) = Stringa_1 or Stringa_2}
          operatore_esiste := True ;
    end case
    exit when operatore_esiste or k=1 ;
  end loop ;

.......
```

Albero ← Formula

... continuazione

```
if {operatore_esiste}

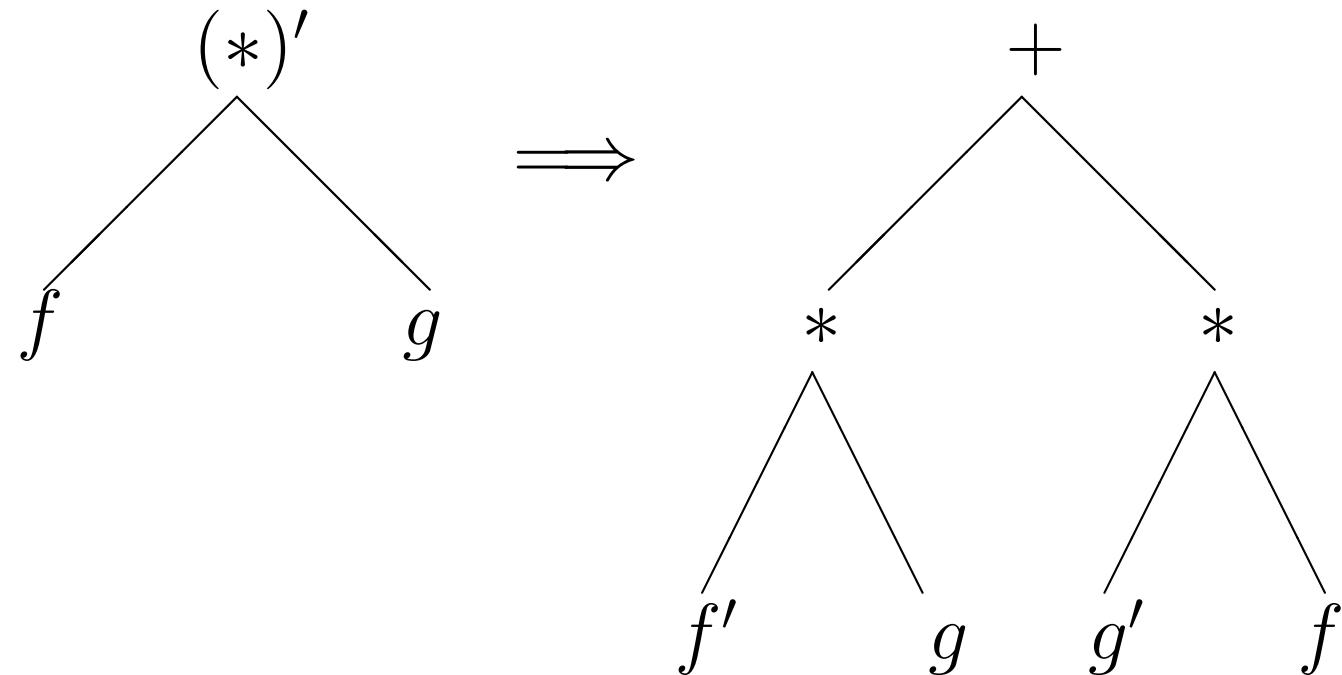
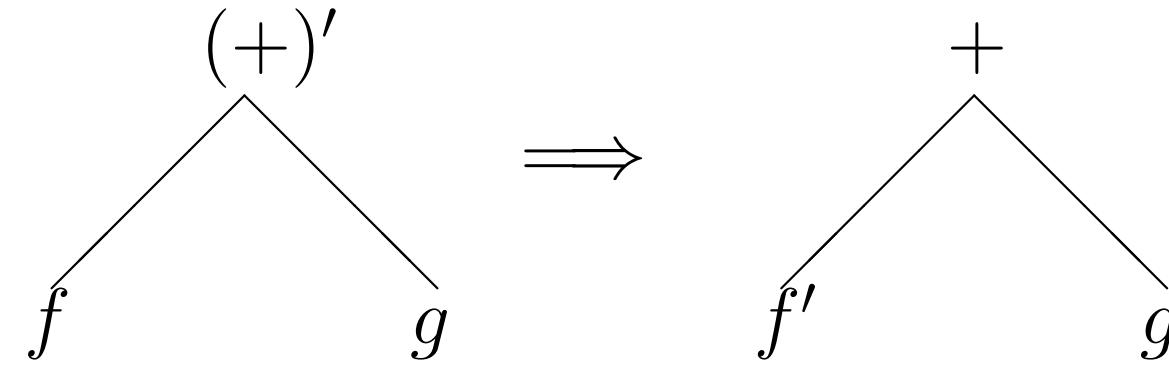
{si crea un nodo Operatore}

{genera_albero(i.s,sottestringa_sinistra)}
{genera_albero(i.d,sottestringa_destra)}

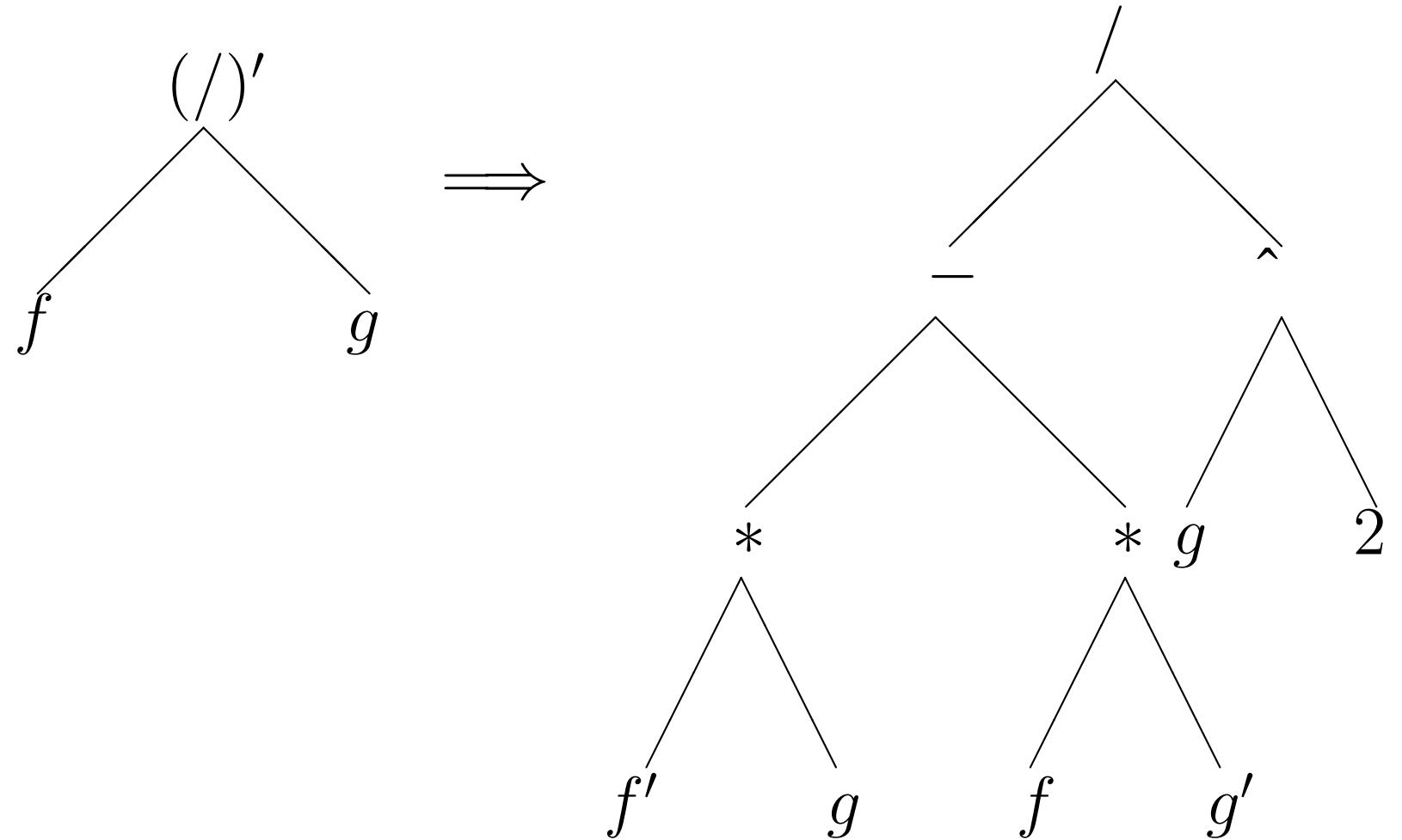
else if {operando_esiste}

{si crea un nodo Operando}
}
```

Albero della funzione derivata



Albero della funzione derivata



Binary Trees

- Implementation
 - Because a binary tree has at most two children, we can keep direct pointers to them

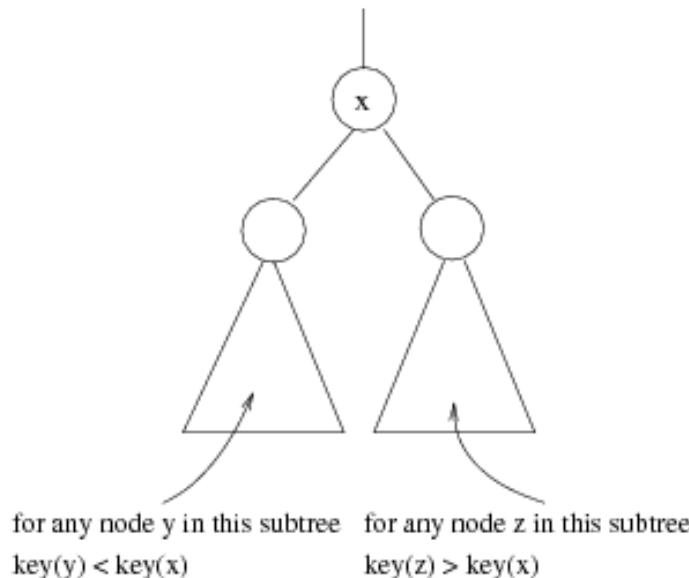
```
struct BinaryNode
{
    Object      element;      // The data in the node
    BinaryNode *left;         // Left child
    BinaryNode *right;        // Right child
};
```

Binary Search Trees

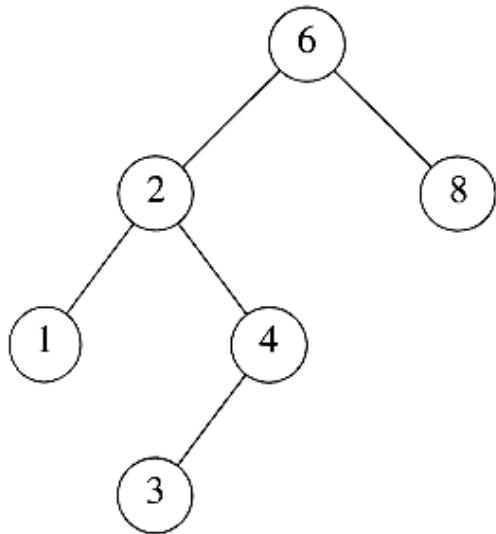
- Stores keys in the nodes in a way so that searching, insertion and deletion can be done efficiently.

Binary **search tree** property

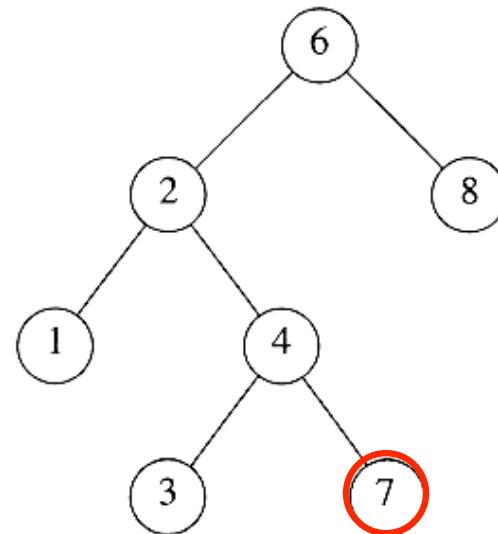
- For every node X, all the keys in its left subtree are smaller than the key value in X, and all the keys in its right subtree are larger than the key value in X



Binary Search Trees



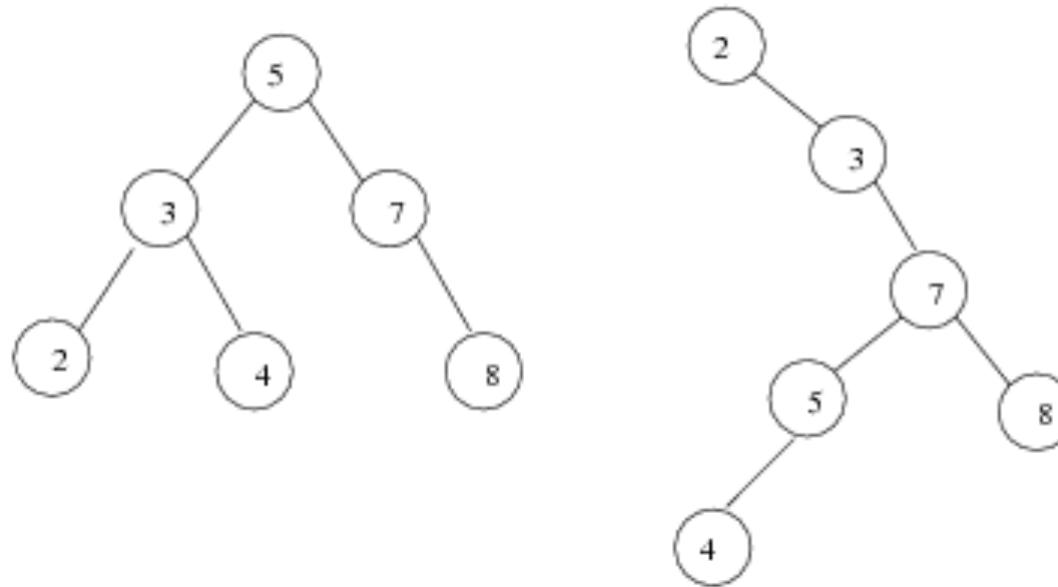
A binary search tree



Not a binary search tree

Binary search trees

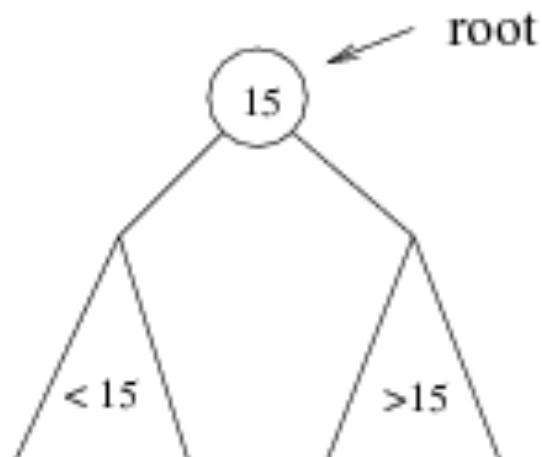
Two binary search trees representing
the same set:



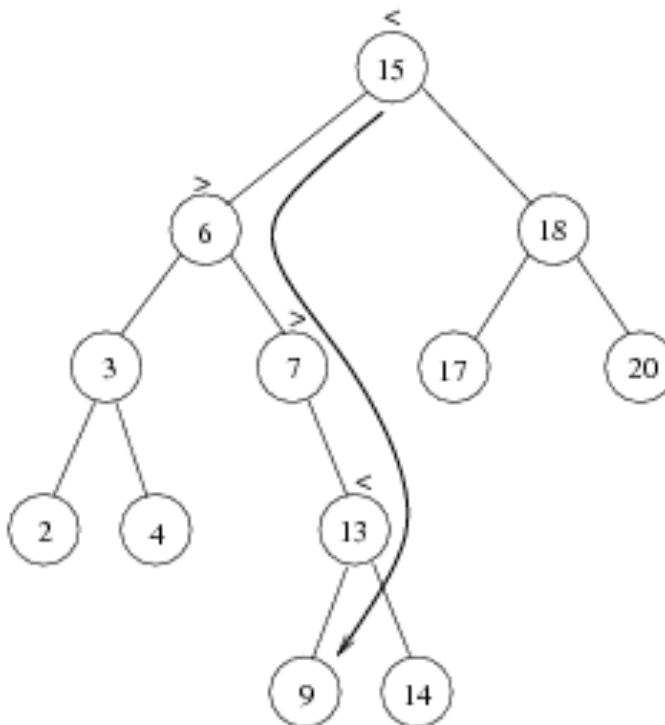
- Average depth of a node is $O(\log N)$; maximum depth of a node is $O(N)$

Searching BST

- If we are searching for 15, then we are done.
- If we are searching for a key < 15 , then we should search in the left subtree.
- If we are searching for a key > 15 , then we should search in the right subtree.



Example: Search for 9 ...



Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Searching (Find)

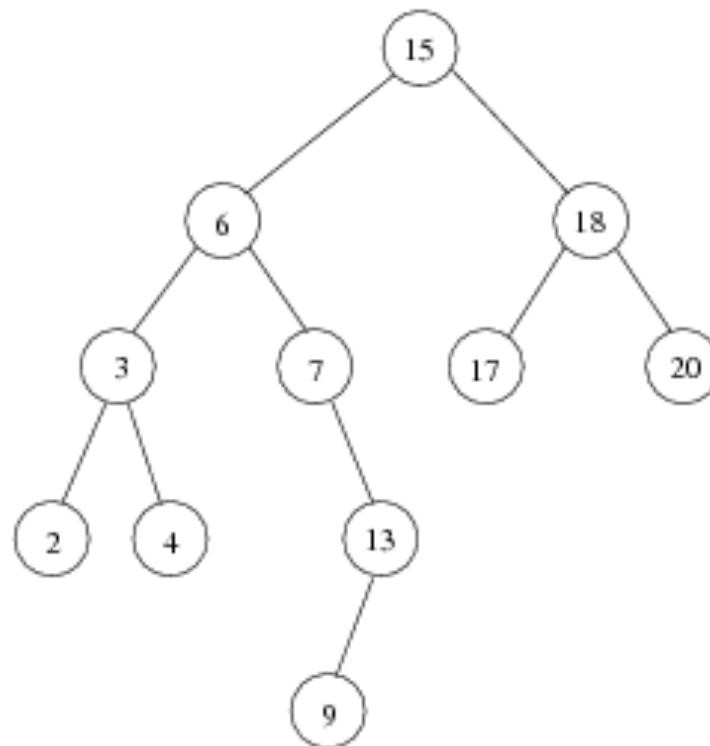
- Find X: return a pointer to the node that has key X, or NULL if there is no such node

```
template <class Comparable>
BinaryNode<Comparable> *
BinarySearchTree<Comparable>::
find( const Comparable & x, BinaryNode<Comparable> *t ) const
{
    if( t == NULL )
        return NULL;
    else if( x < t->element )
        return find( x, t->left );
    else if( t->element < x )
        return find( x, t->right );
    else
        return t;    // Match
}
```

- Time complexity
 - $O(\text{height of the tree})$

Inorder traversal of BST

- Print out all the keys in sorted order



Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

findMin/ findMax

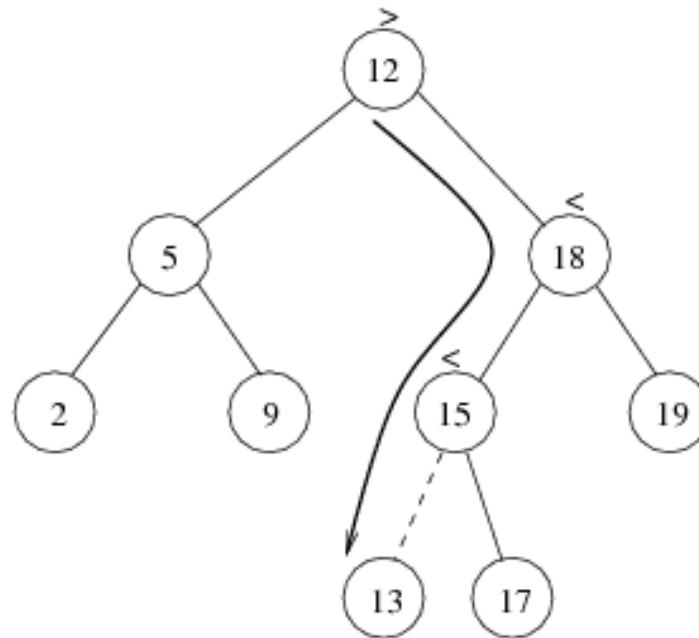
- Return the node containing the smallest element in the tree
- Start at the root and go left as long as there is a left child. The stopping point is the smallest element

```
template <class Comparable>
BinaryNode<Comparable> *
BinarySearchTree<Comparable>::findMin( BinaryNode<Comparable> *t ) const
{
    if( t == NULL )
        return NULL;
    if( t->left == NULL )
        return t;
    return findMin( t->left );
}
```

- Similarly for findMax
- Time complexity = $O(\text{height of the tree})$

insert

- Proceed down the tree as you would with a find
- If X is found, do nothing (or update something)
- Otherwise, insert X at the last spot on the path traversed



- Time complexity = $O(\text{height of the tree})$

delete

- When we delete a node, we need to consider how we take care of the children of the deleted node.
 - This has to be done such that the property of the search tree is maintained.

delete

Three cases:

(1) the node is a leaf

- Delete it immediately

(2) the node has one child

- Adjust a pointer from the parent to bypass that node

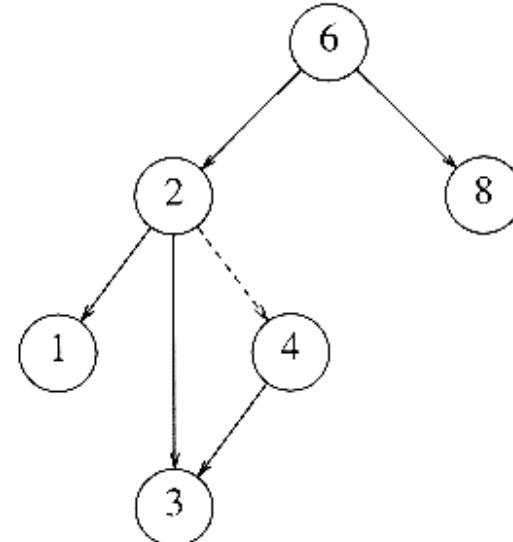
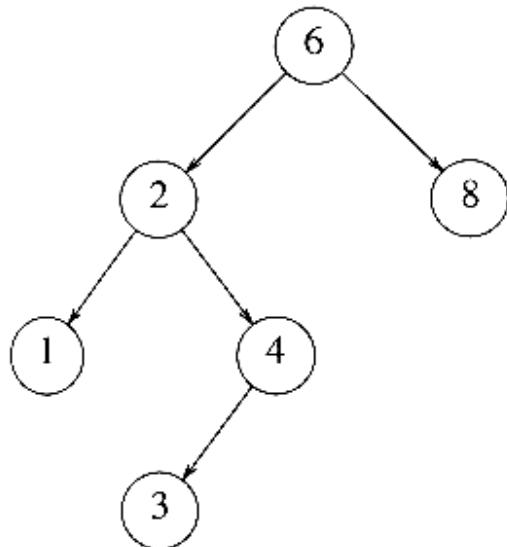


Figure 4.24 Deletion of a node (4) with one child, before and after

delete

(3) the node has 2 children

- replace the key of that node with the minimum element at the right subtree
- delete the minimum element
 - Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.

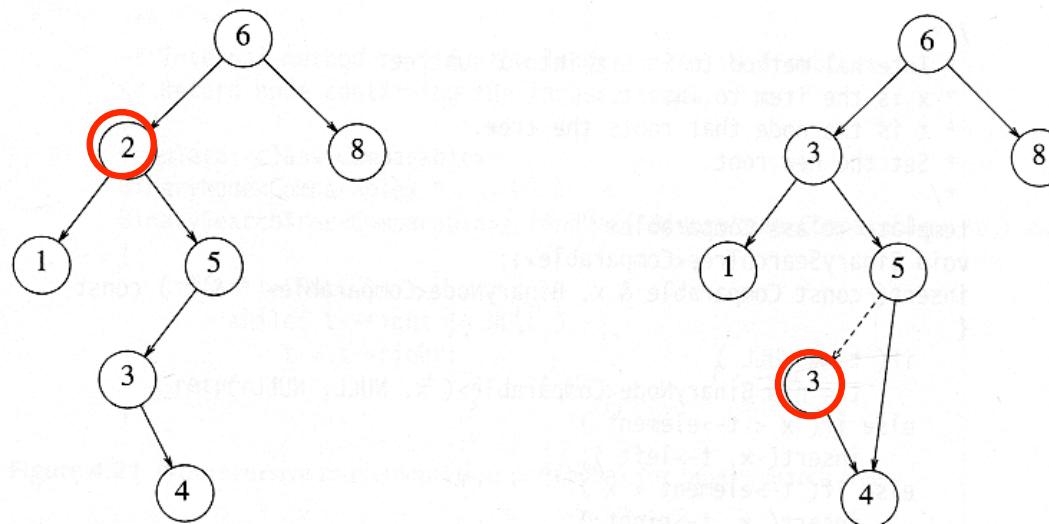
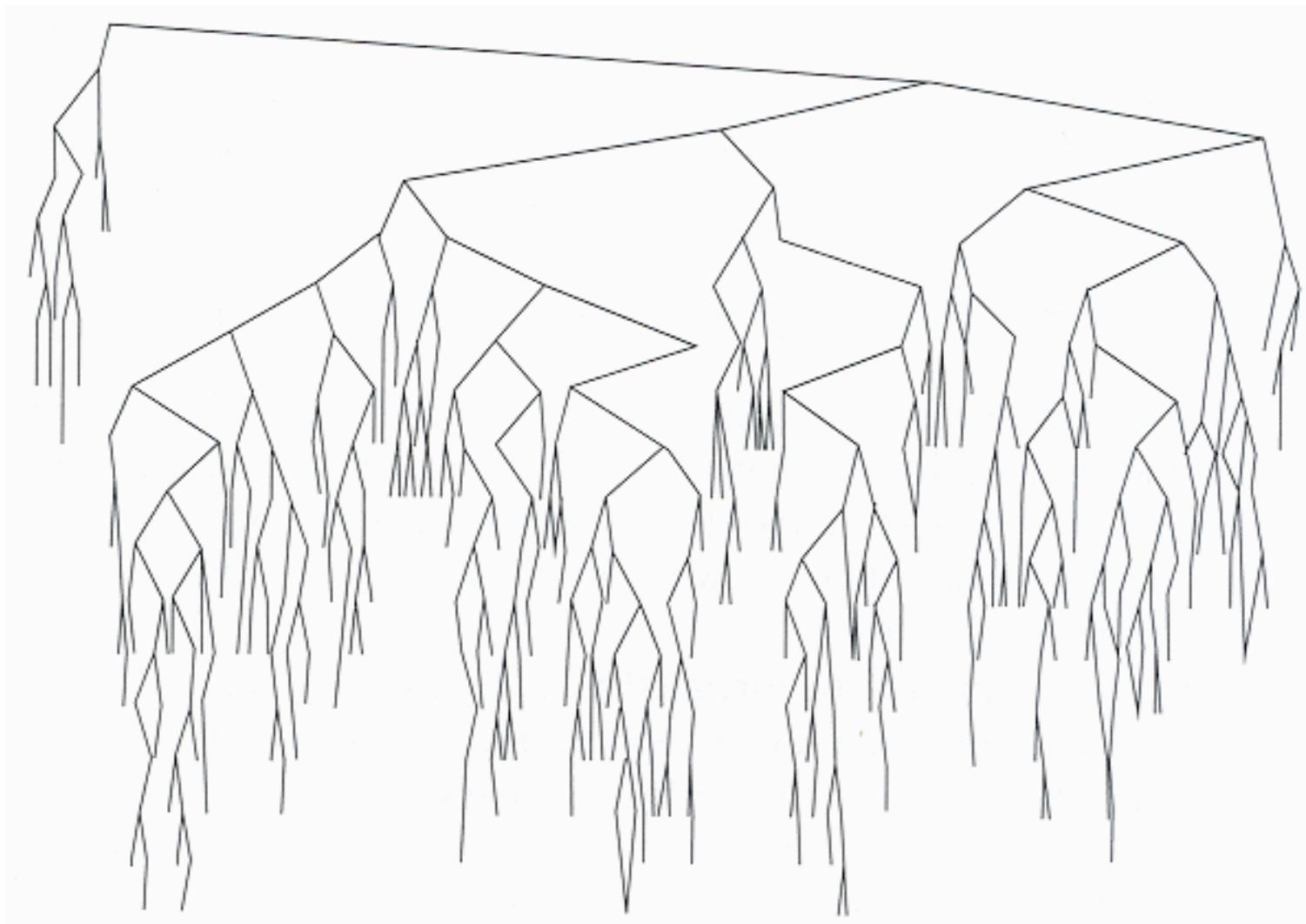
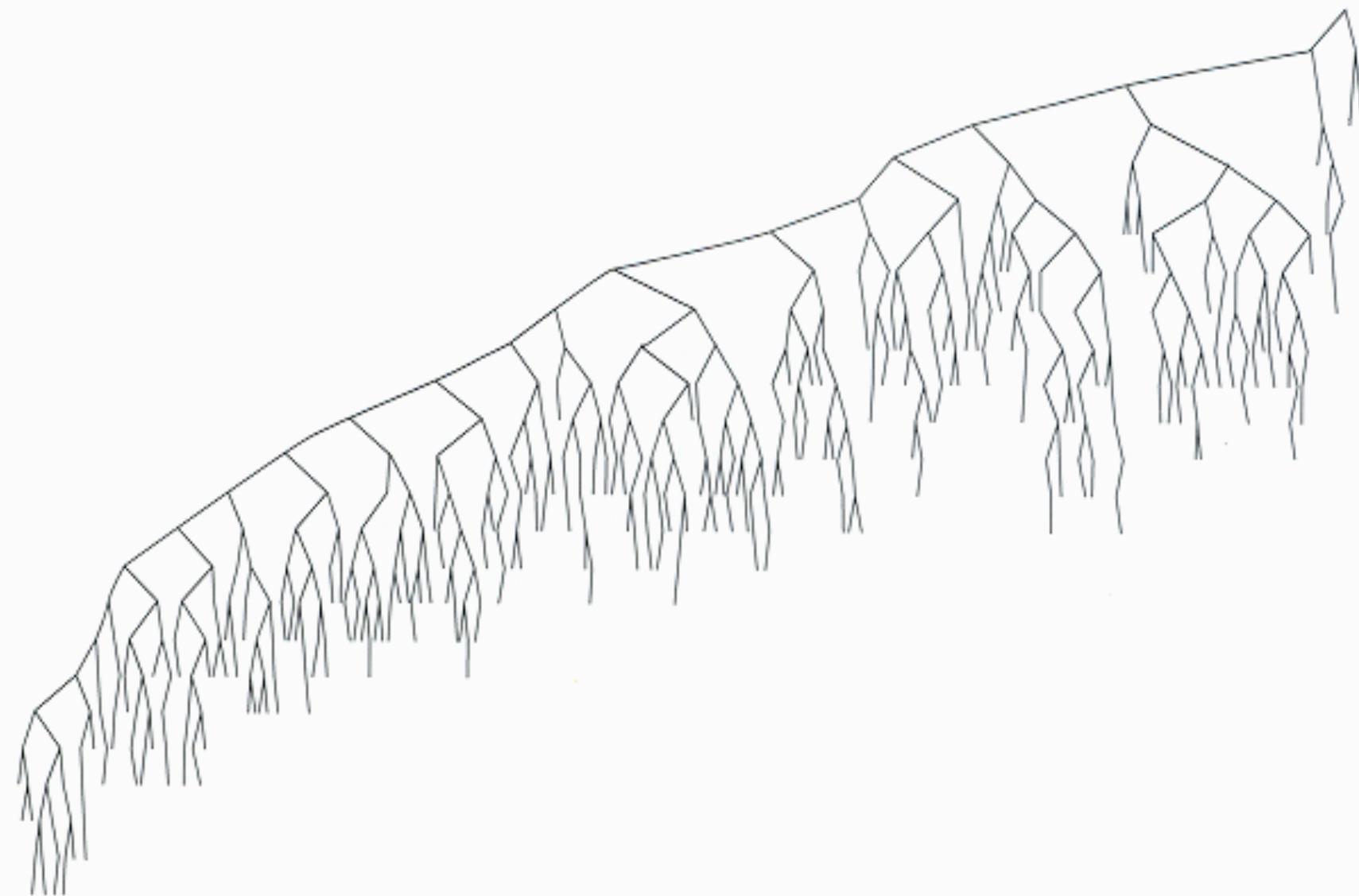


Figure 4.25 Deletion of a node (2) with two children, before and after

- Time complexity = $O(\text{height of the tree})$

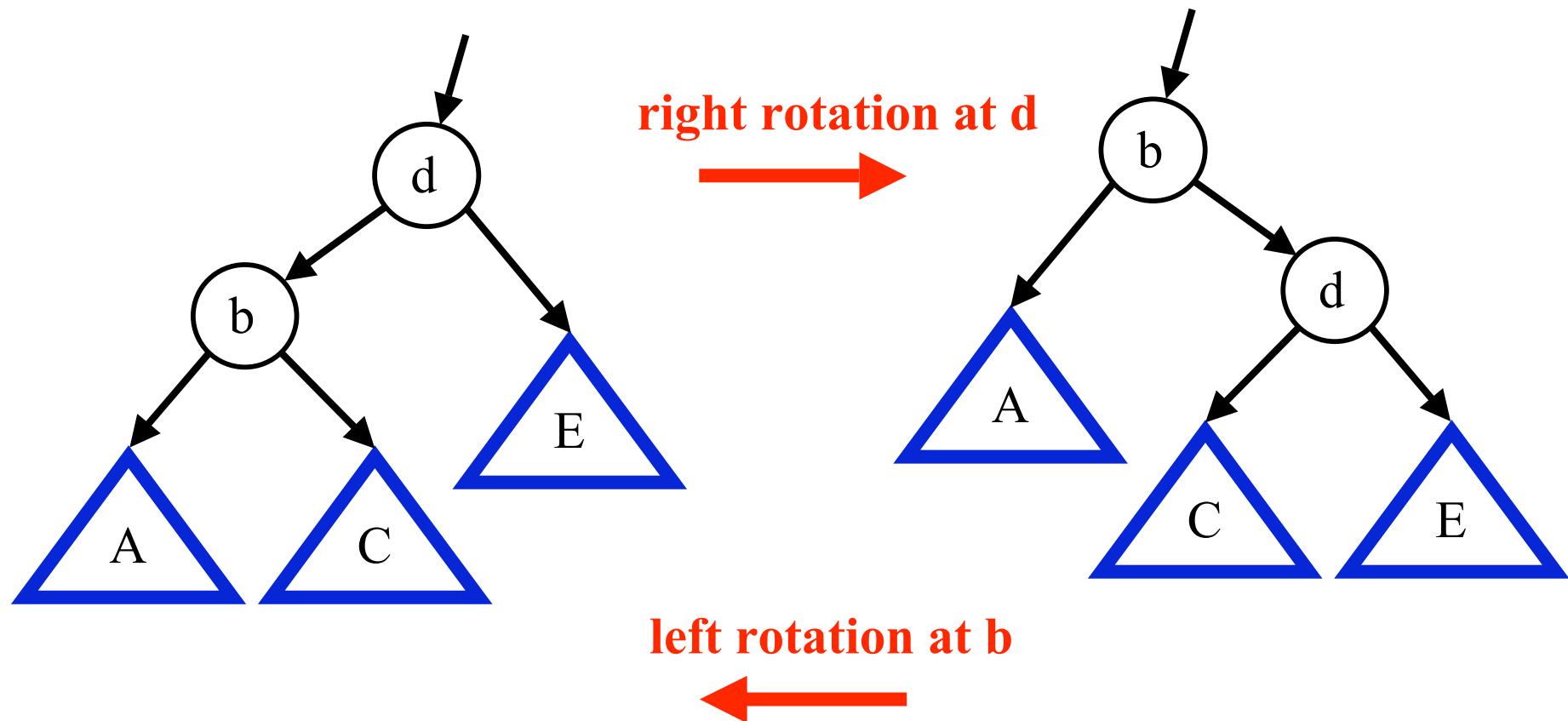


Tree generated by random insertions



Alternating random insertions and random deletions

Tree Rotations



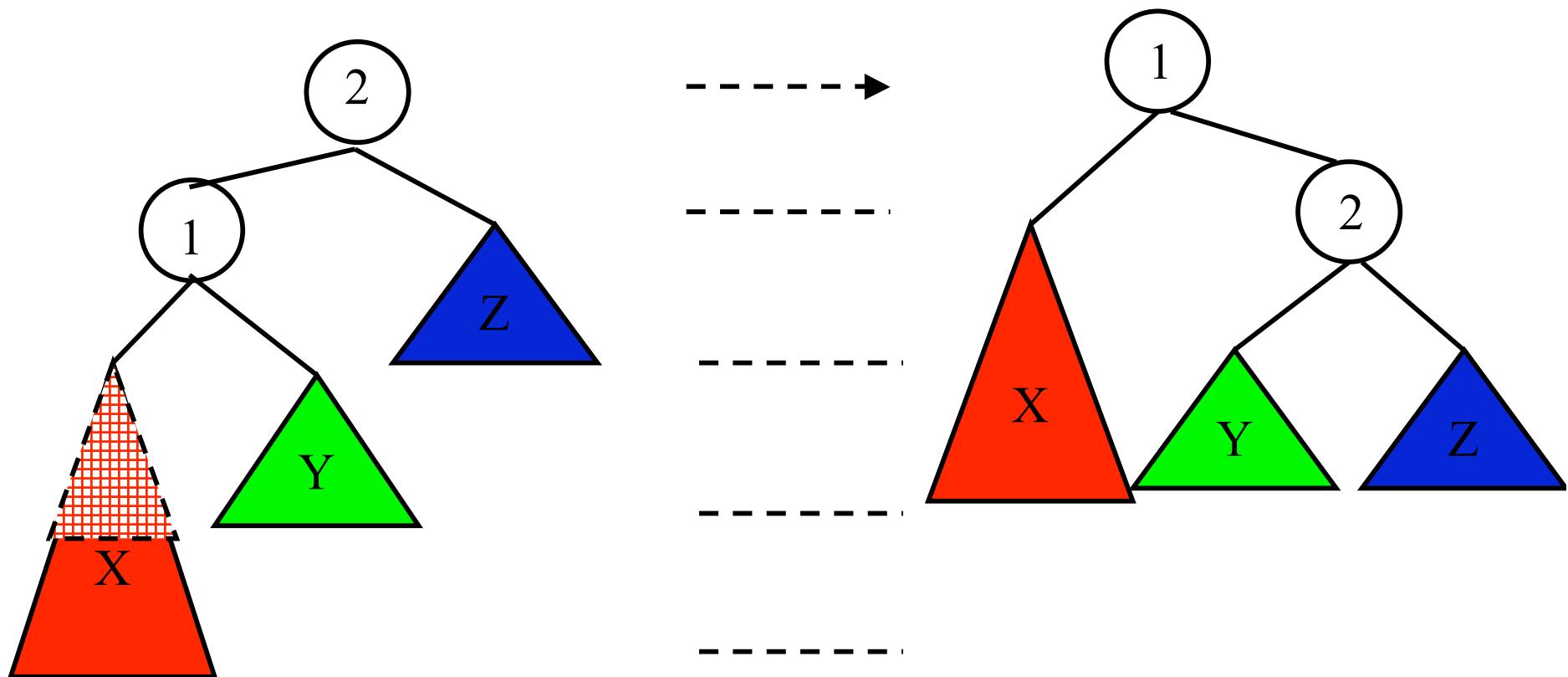
AVL trees

- *AVL* is from the inventors: Adelson-Velsky and Landis (1962)
- *height* of a tree: max depth of a node in the tree
- **AVL tree**: Binary search tree in which at every node X, the height of the left and right subtrees of X differ by at most 1.
- **Theorem**: An AVL tree with n nodes has height (at most) $O(\log n)$
- Proof: What's the minimum number of nodes an AVL tree of height h can have?

AVL Trees - Insertion

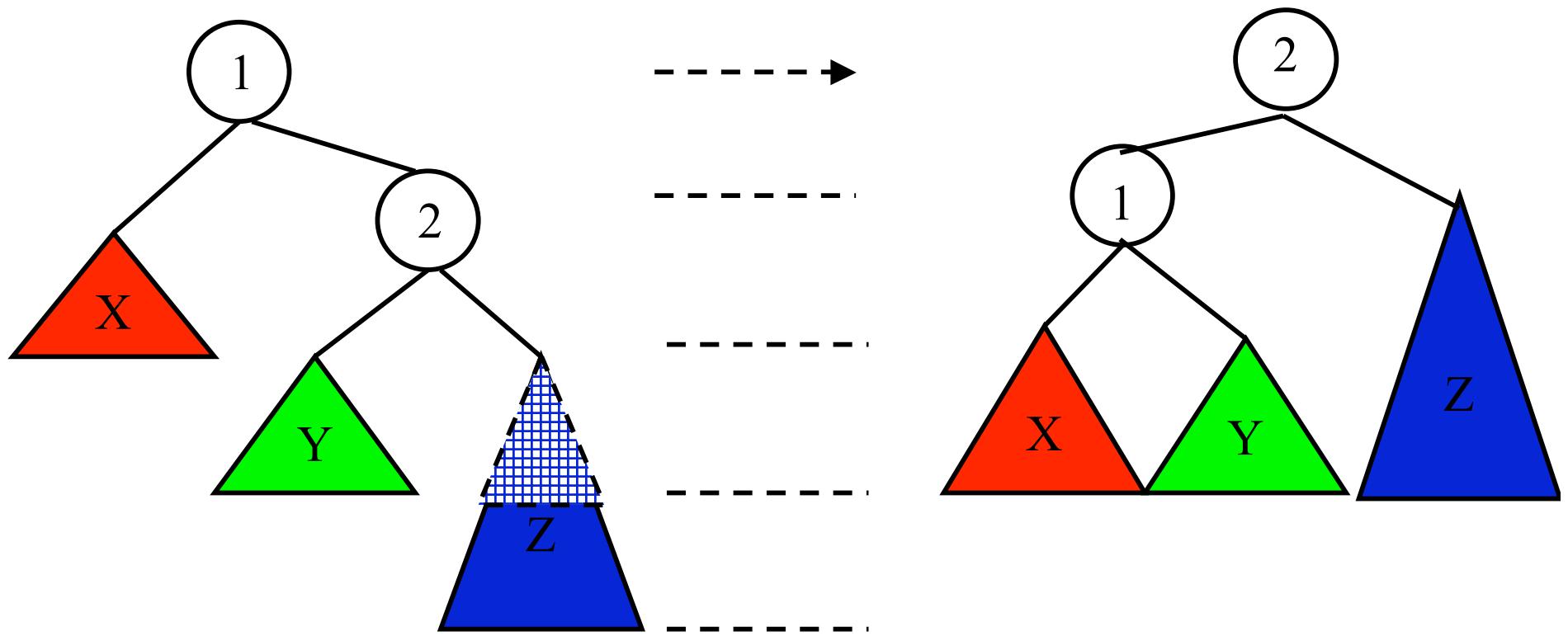
- Member (Find) operation (as in BST)
- Insert at leaf
- Update balance information upward from insertion
- At the first node that $|balance| \geq 2$, four cases:
 1. insertion was in left subtree of left child
 2. insertion was in left subtree of right child
 3. insertion was in right subtree of left child
 4. insertion was in right subtree of right child
- 1 and 4 are fixed by single rotation
- 2 and 3 are fixed by double rotation

Single Rotation (left-left)



- Y cannot be at X's level
- Y cannot be at Z's level either

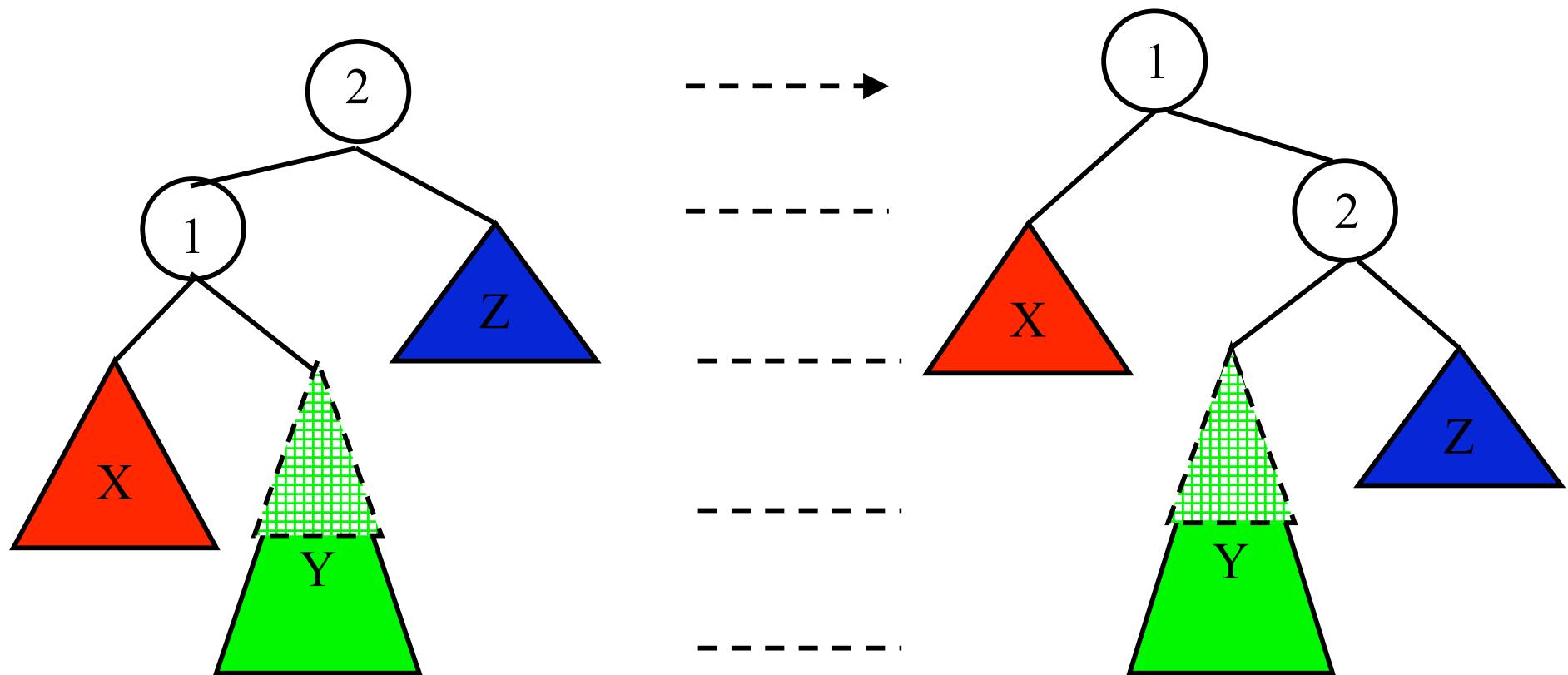
Single Rotation (right-right)



- Y cannot be at Z's level
- Y cannot be at X's level either

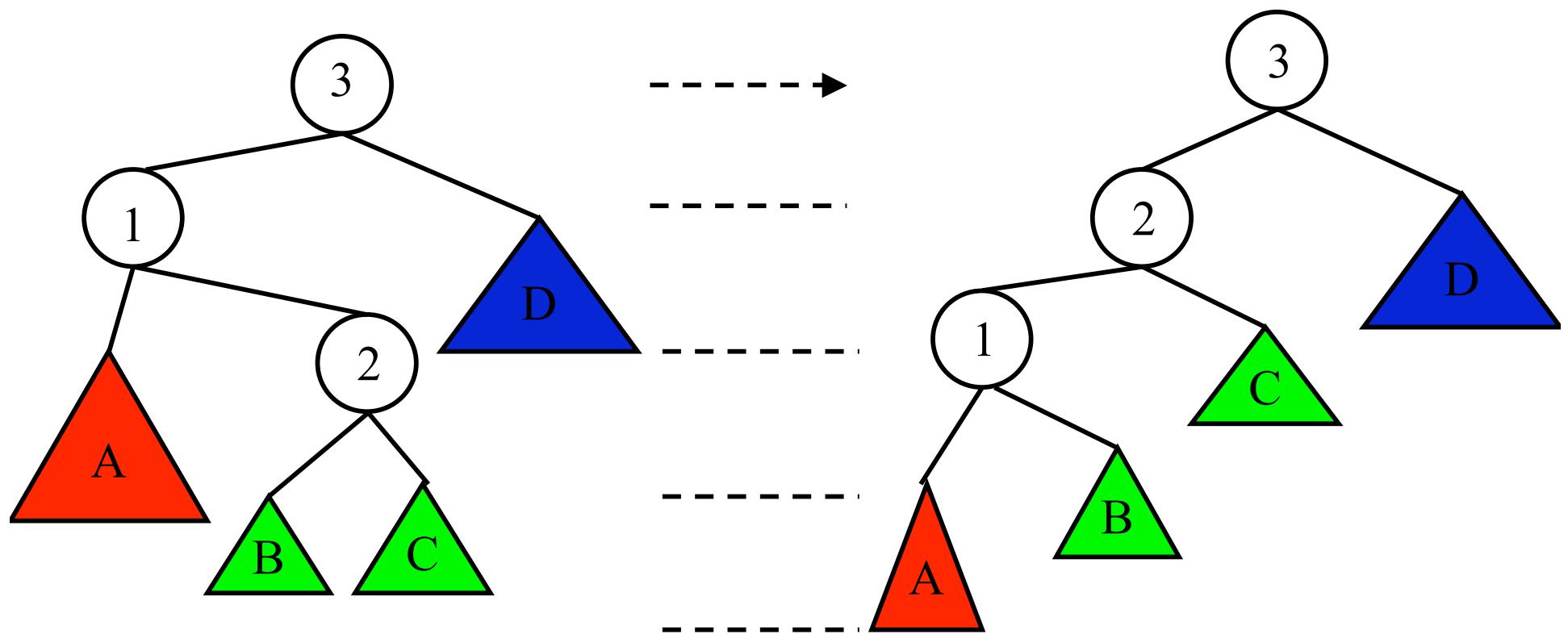
Double Rotation (left-right): Single Won't Work

- Single rotation does not work because it does not make Y any shorter



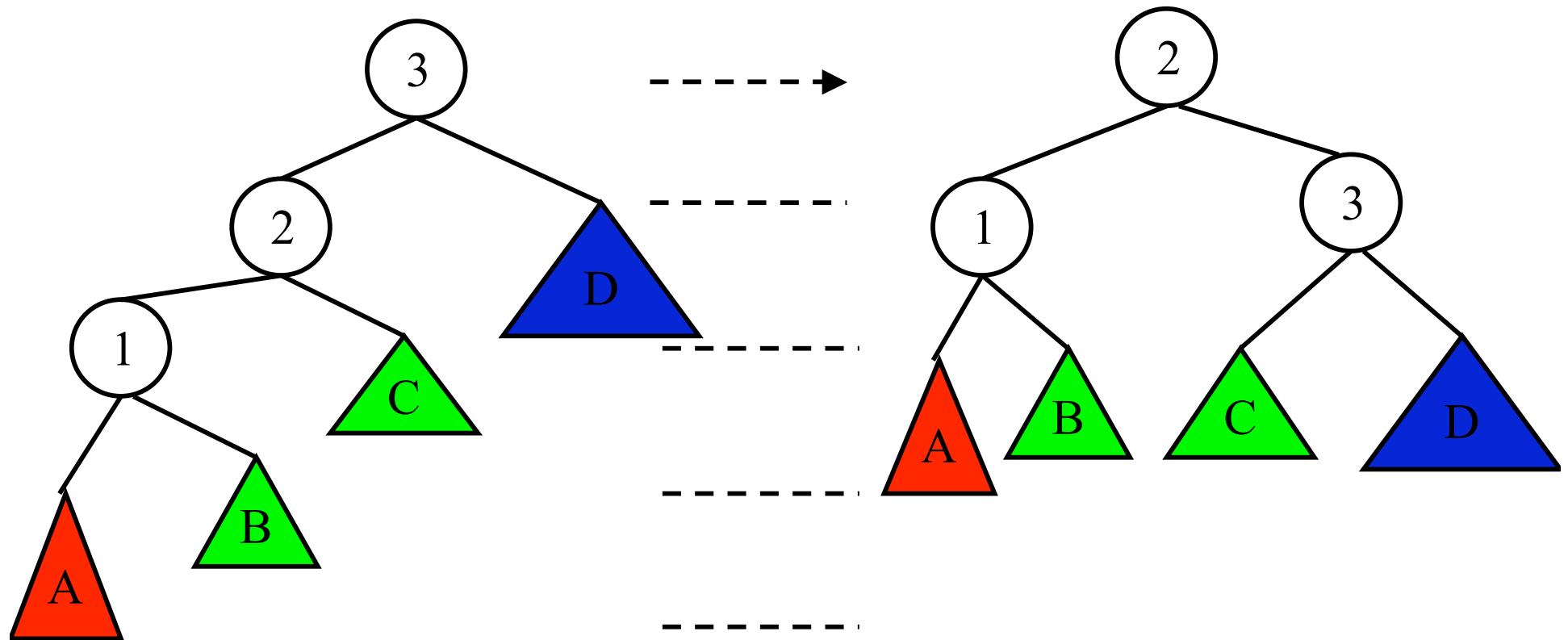
Double Rotation (left-right): First Step

- First rotation between 2 and 1

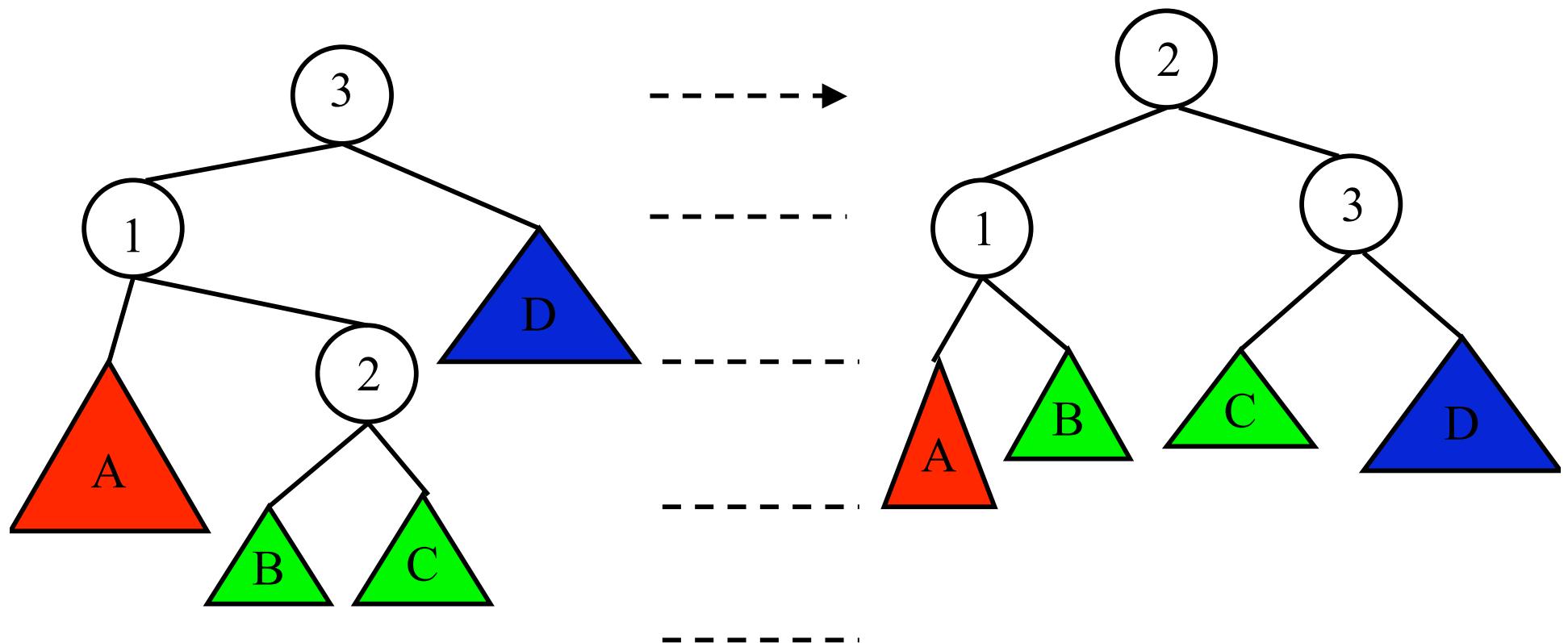


Double Rotation (left-right): Second Step

- Second rotation between 2 and 3

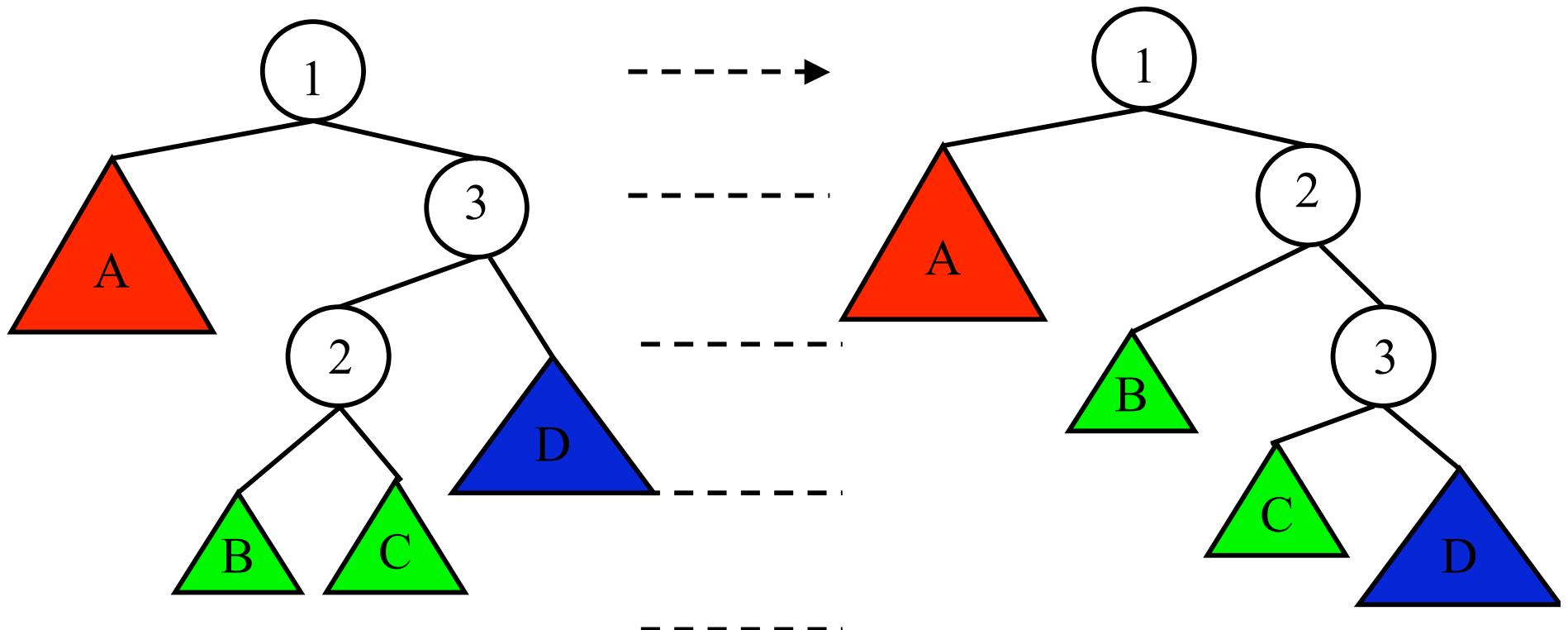


Double Rotation (left-right): Summary



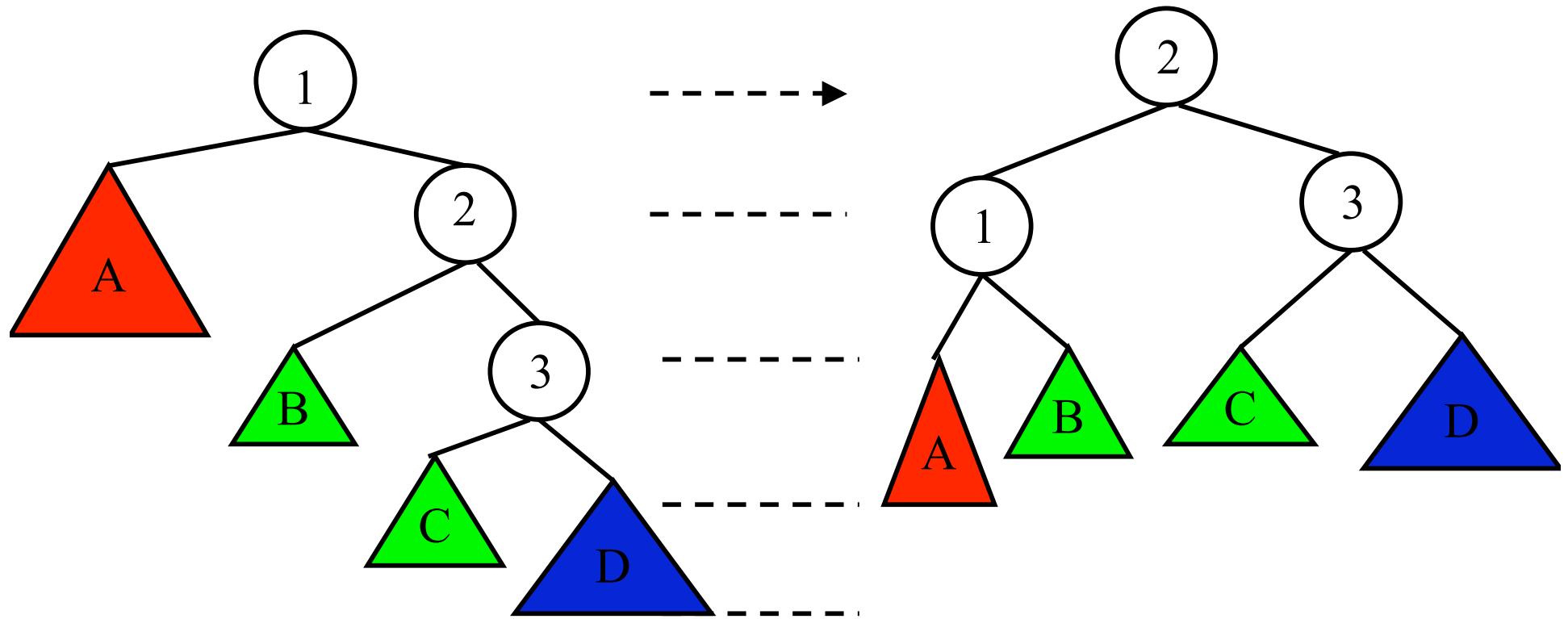
Double Rotation (right-left): First Step

- First Rotation between 2 and 3

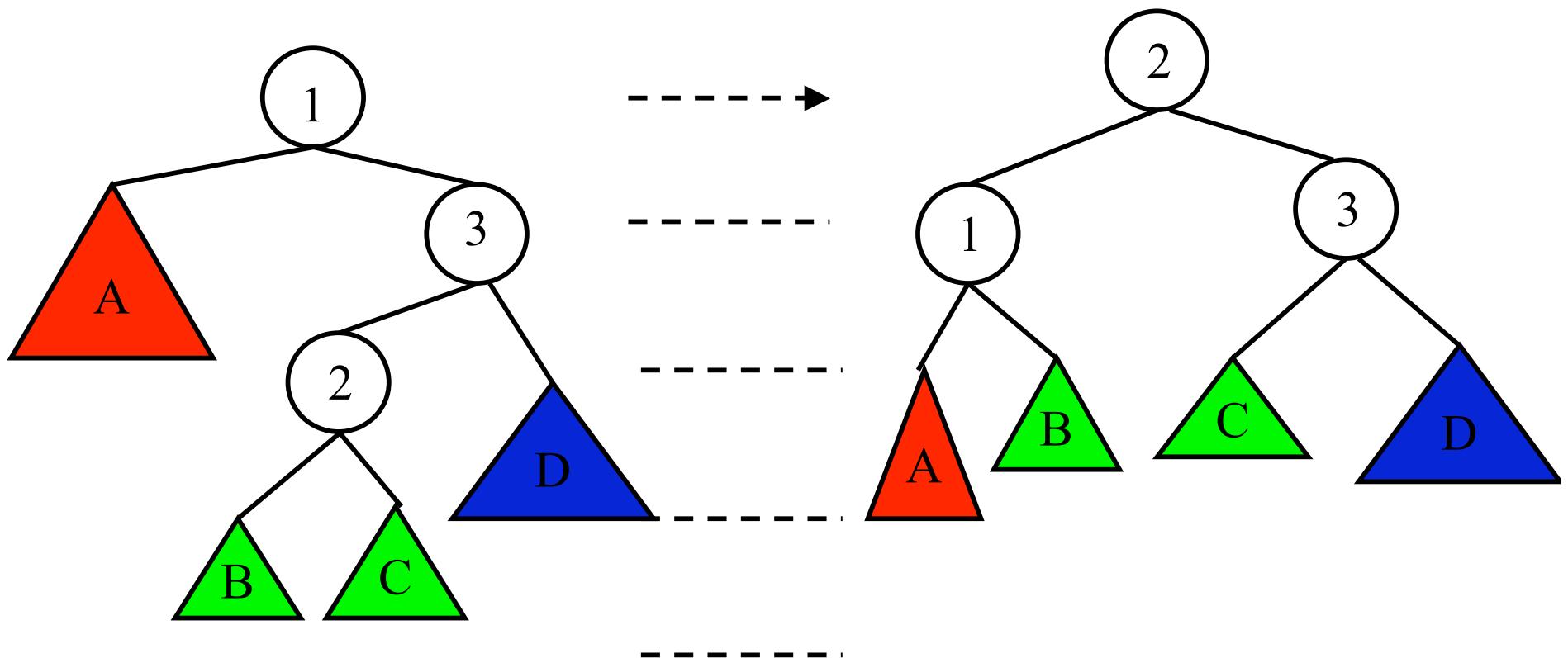


Double Rotation (right-left): Second Step

- Second Rotation between 1 and 2



Double Rotation (right-left): Summary



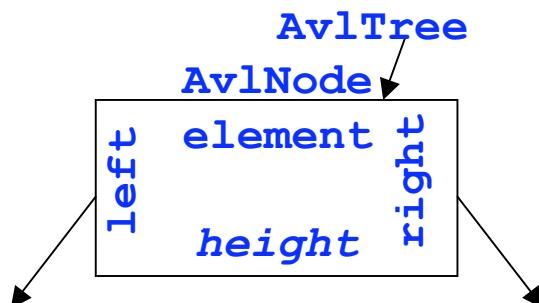
Insertion into an AVL tree

- Tree height

- Search

- Insertion:

- search to find insertion point
- adjust the tree height
- rotation if necessary

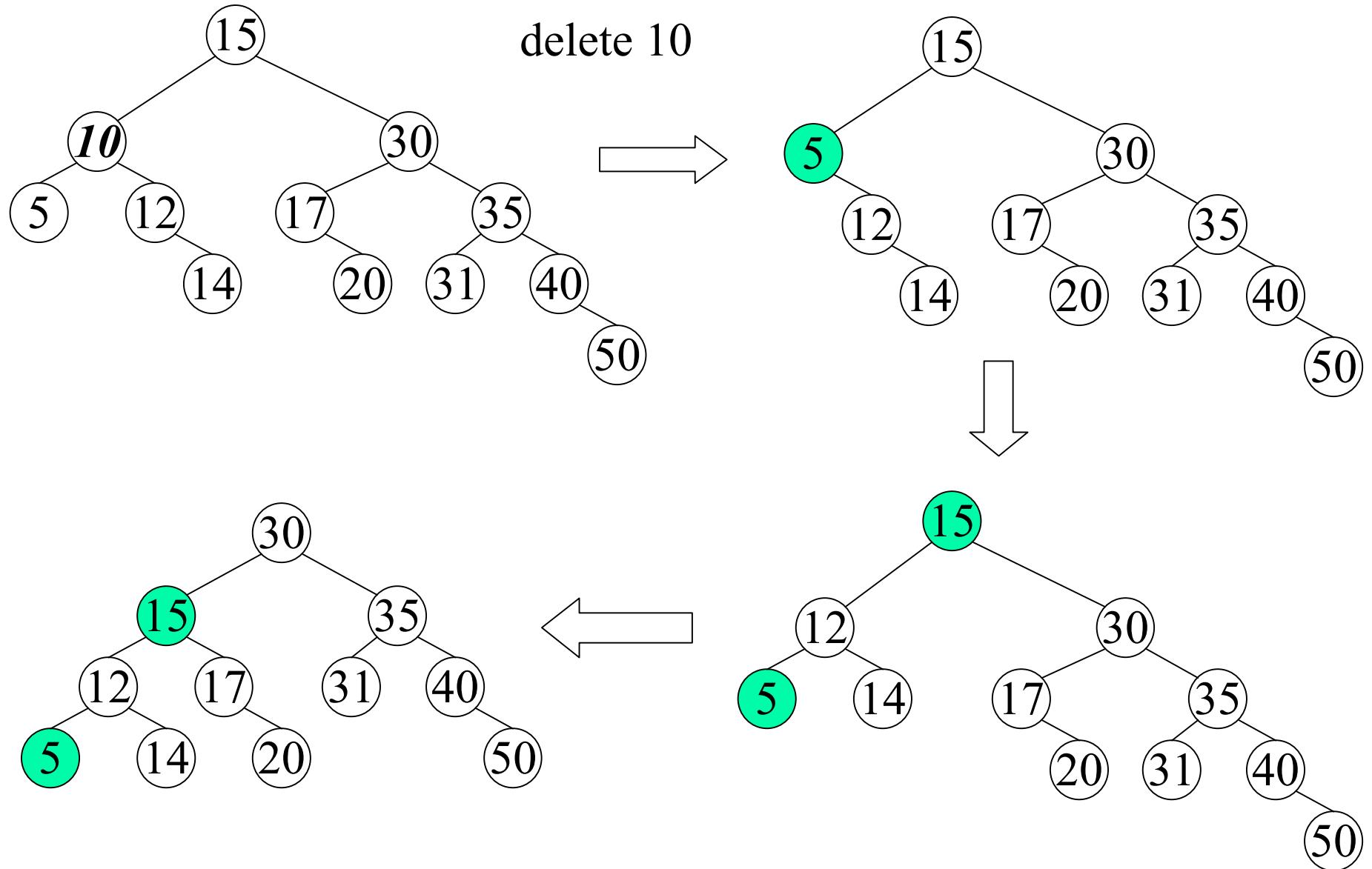


```
AvlTree::insert(x, t)
  if (t = NULL)
    then t = new AvlNode(x, ...);
  else if (x < t→element)
    then
      insert (x, t→left);
      if (height(t→left) - height(t→right) = 2)
        then if (x < t→left→element)
          then rotateWithLeftChild (t);
        else doubleWithLeftChild (t);
      else if (t→element < x )
        then
          insert (x, t→right);
          if (height(t→right) - height(t→left) = 2)
            then if (t→right→element < x)
              then rotateWithRightChild (t);
            else doubleWithRightChild (t);
    t→height =
      max{height(t→left), height(t→right)}+1;
```

Removing an element from an AVL tree

- Similar process:
 - locate & delete the element
 - adjust tree height
 - perform necessary rotations (up to $\log(n+1)$)
- Complexity of operations:
 - $O(h) \leq O(\log n)$
 n : number of nodes, h : tree height

Deletion Example



Alberi a pagine

- Ogni nodo è una *pagina*
- Una pagina è un insieme di *elementi*
- L'accesso alle pagine avviene in *modo random*
- L'accesso agli elementi all'interno di una pagina avviene in *modo sequenziale*
- Struttura dati vantaggiosa su disco poiché
 - L'accesso random è *lento*
 - L'accesso sequenziale nella pagina (se coincide con un *cluster* è *veloce*)

Alberi a pagine

Confronto fra albero binario e albero a pagine pieni

Esempio con 10^6 elementi e 100 elementi per pagina:

Numero accessi *random* nel caso peggiore:

Albero binario pieno

$$\log 10^6 \approx 20$$

Albero a pagine piano

$$\approx \log_{100} 10^6 \approx 3$$

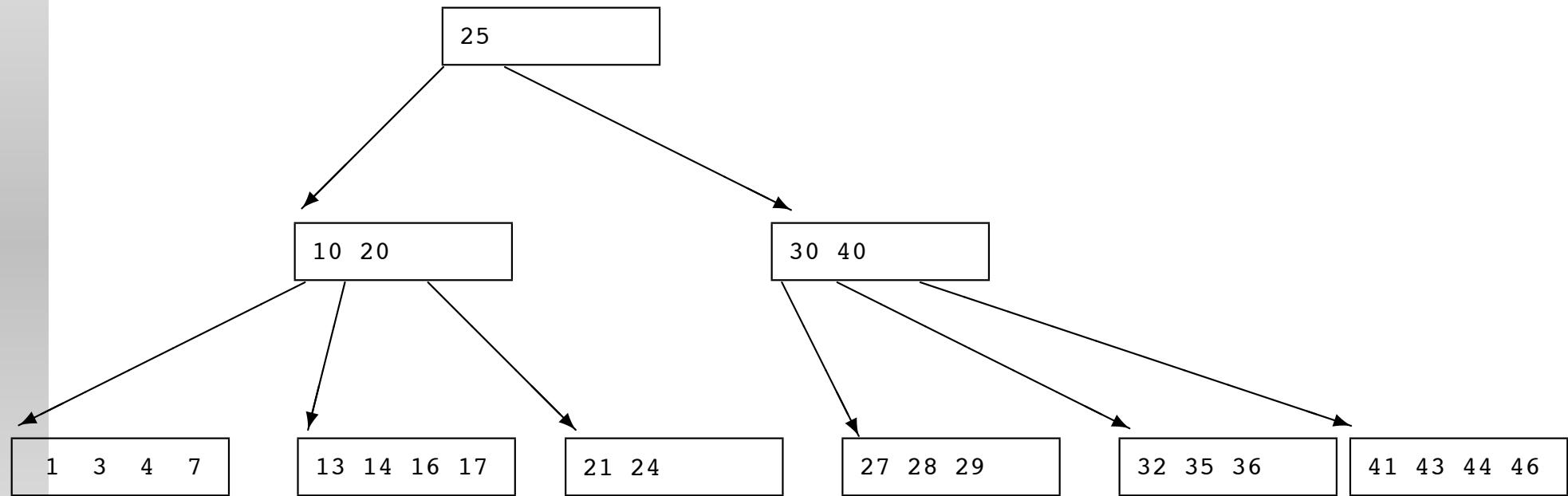
B-Tree (1970 R.Bayer , E. McCreight)

Sono alberi di ricerca a pagine dove:

- ogni pagina contiene elementi e accessi
- ogni pagina contiene al massimo $2n$ elementi
- n è detto *ordine* dell'albero
- ogni pagina, ad eccezione della radice, contiene almeno n elementi
- la radice ha almeno due accessi a discendenti, altrimenti è l'unico nodo
- ogni pagina con m elementi, salvo la foglia, ha $m + 1$ accessi a discendenti
- tutte le foglie sono situate allo stesso livello.

B-Tree

Esempio: Albero di ordine 2



Algoritmo di ricerca per B-Tree

Nodo: sequenza di *accessi* a_i e di elementi identificabili mediante **chiavi** c_i

$a_0 \ c_1 \ a_1 \ c_2 \ a_2 \ c_3 \ \cdots \ a_{m-1} \ c_m \ a_m$

```
Nodo := radice ;
loop i = 1 .. m-1
    if Nodo /= null then
        if c = c[i] then
            exit
        elsif c < c[1] then
            if a[0] /= null Nodo := a[0] else Nodo := null end if
        elsif c > c[m] then
            if a[m] /= null Nodo := a[m] else Nodo := null end if
        else
            if a[i] /= null
                Nodo := a[i] else Nodo := null end if
            end if
        end if
    end loop
```

B-tree: inserimento

Algoritmo per inserire un elemento

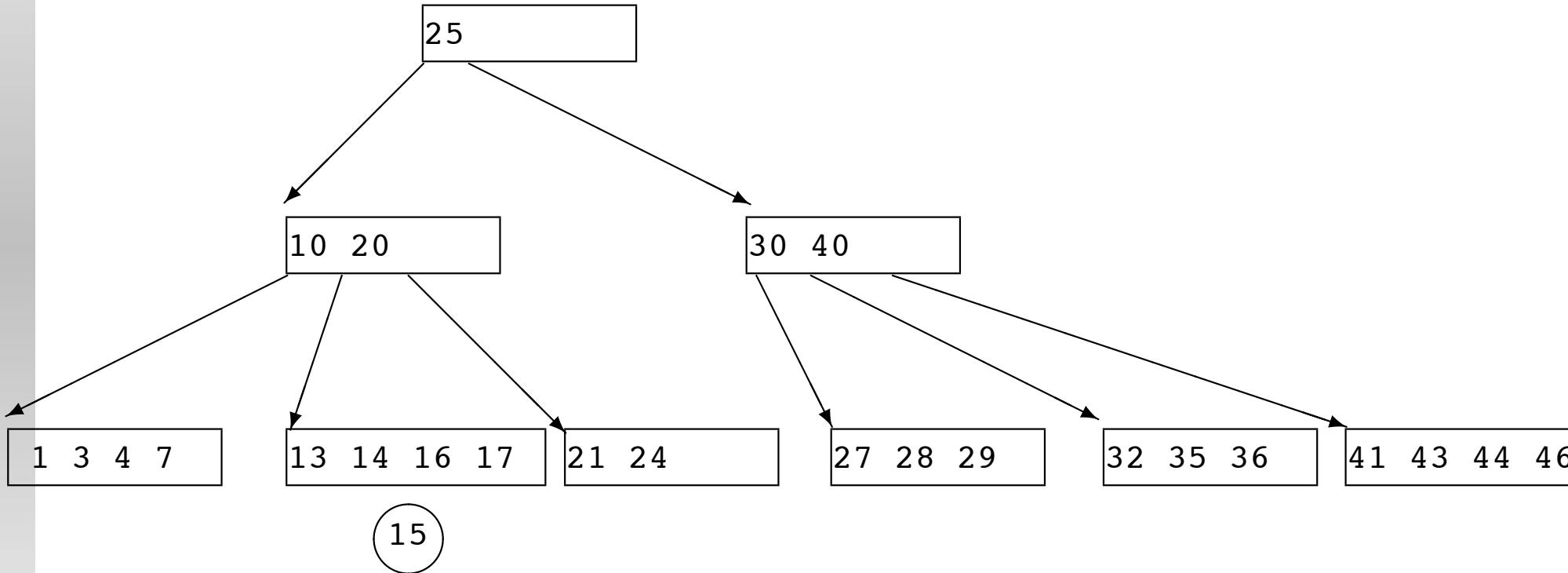
1. Determinare la pagina dove inserire l'elemento,
2. Se nella pagina $m < 2n$, inserire qui l'elemento, altrimenti:
 - (a) generare una nuova pagina,
 - (b) travasare metà del contenuto di quella piena nella nuova,
 - (c) far risalire di un livello l'elemento intermedio.
3. Se necessario iterare il secondo passo.

L'albero si modifica dalle foglie verso la radice.

<http://slady.net/java/bt/view.php>

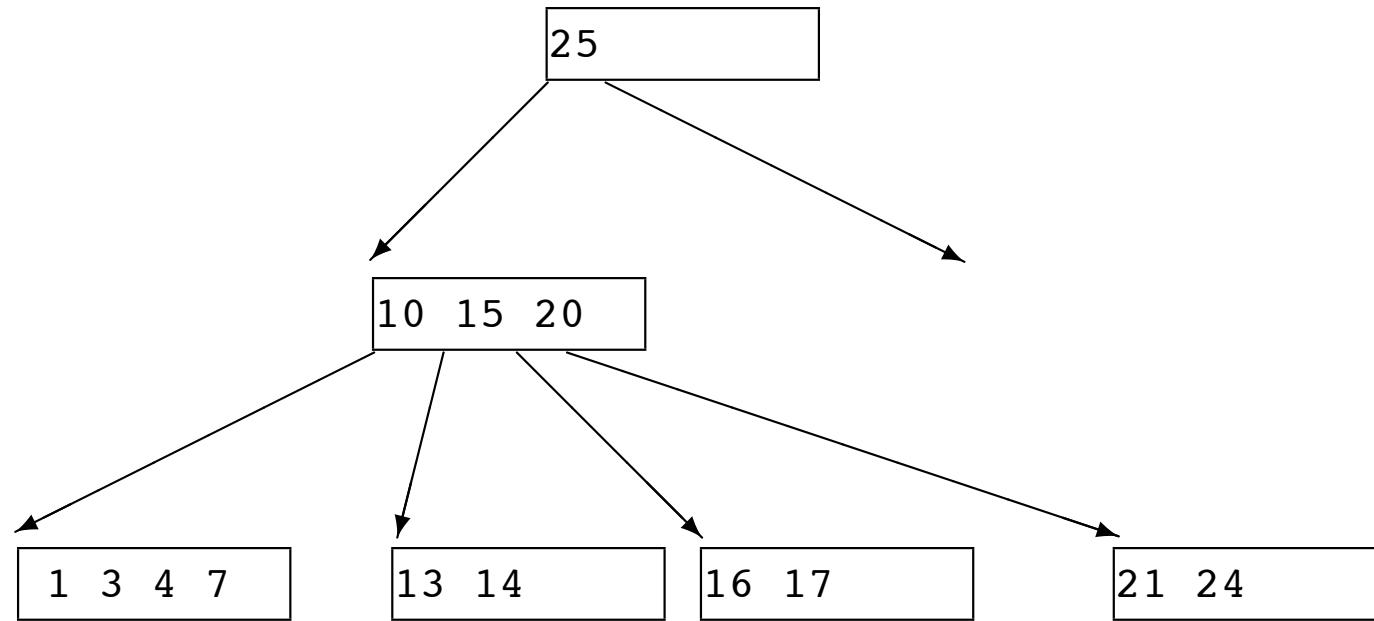
B-tree: inserimento

Esempio 1:



B-tree: inserimento

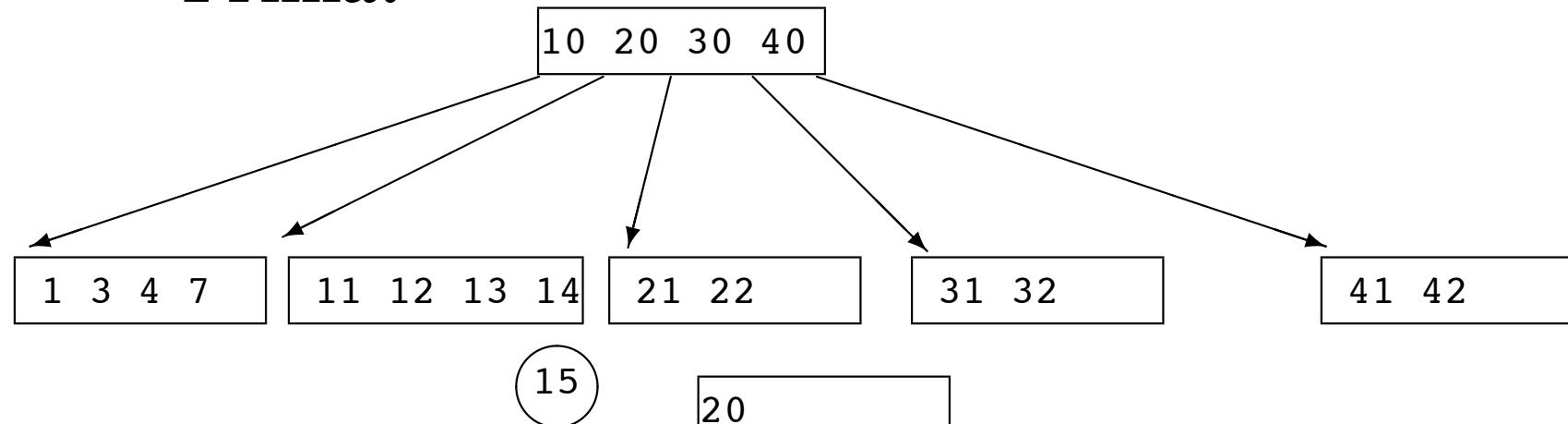
Esempio 1:



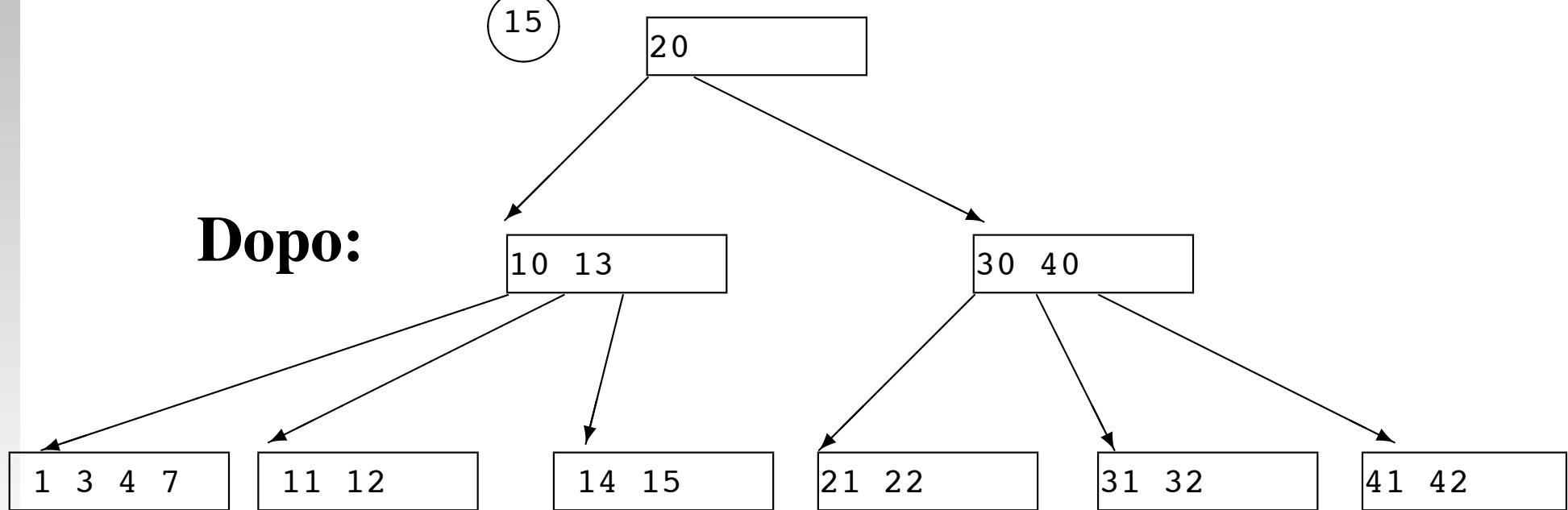
B-tree: inserimento

Esempio 2:

Prima:



Dopo:



B-tree: eliminazione

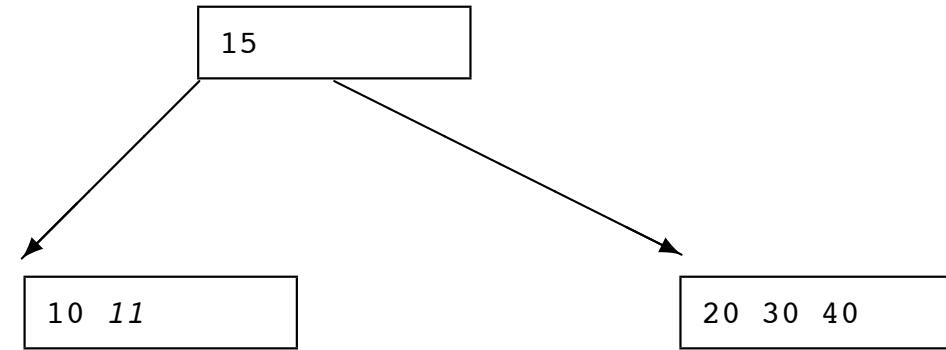
Algoritmo

1. L'elemento da eliminare si trova in una foglia
 - (a) Eliminare l'elemento
 - (b) Eventualmente equilibrare o concatenare
2. L'elemento da eliminare non si trova in una foglia:
 - (a) Meccanismo di eliminazione analogo a quello per gli alberi binari

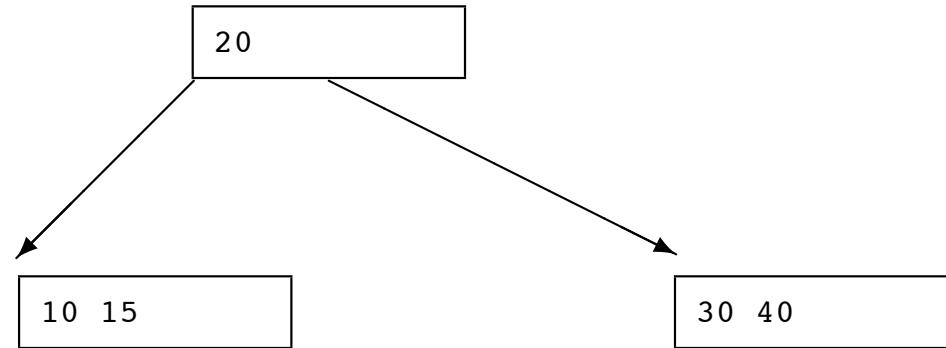
B-tree: eliminazione

Eliminazione di una foglia con equilibratura

Prima:



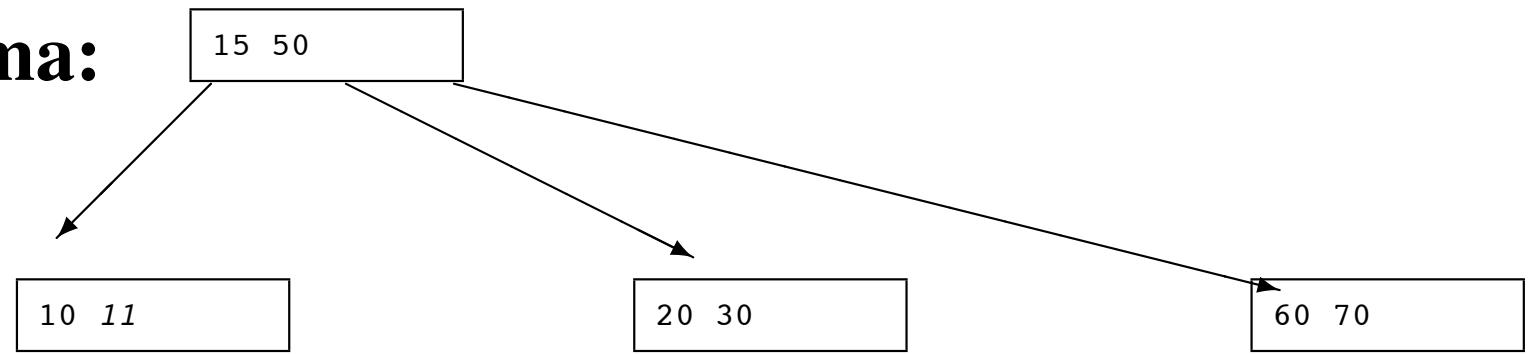
Dopo:



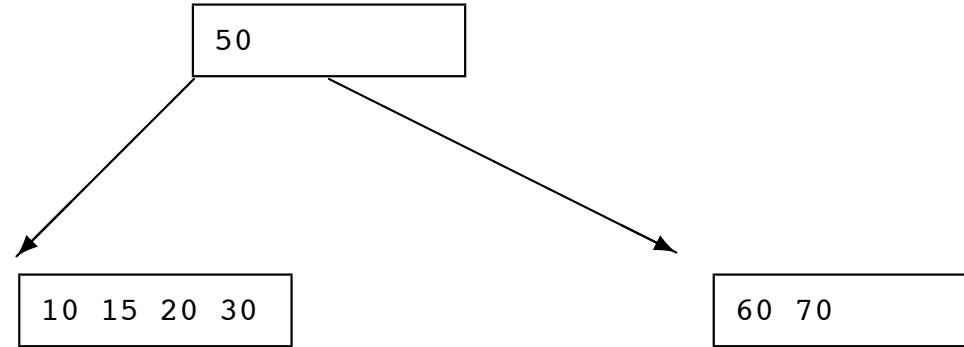
B-tree: eliminazione

Eliminazione di una foglia con concatenamento

Prima:



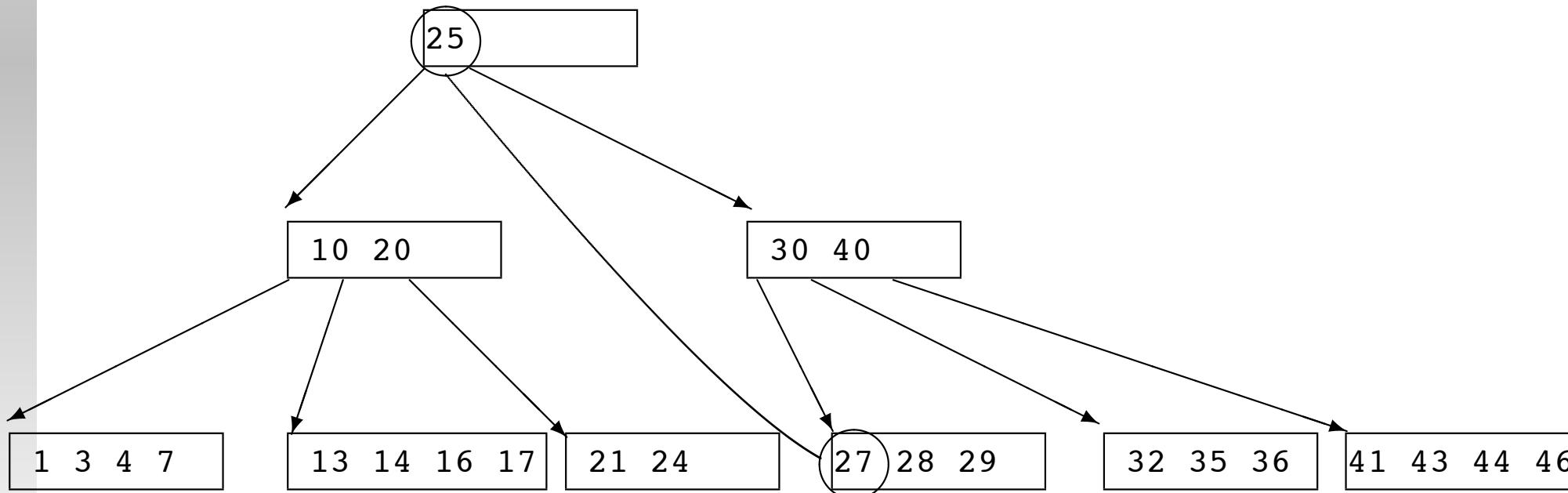
Dopo:



B-Tree: eliminazione

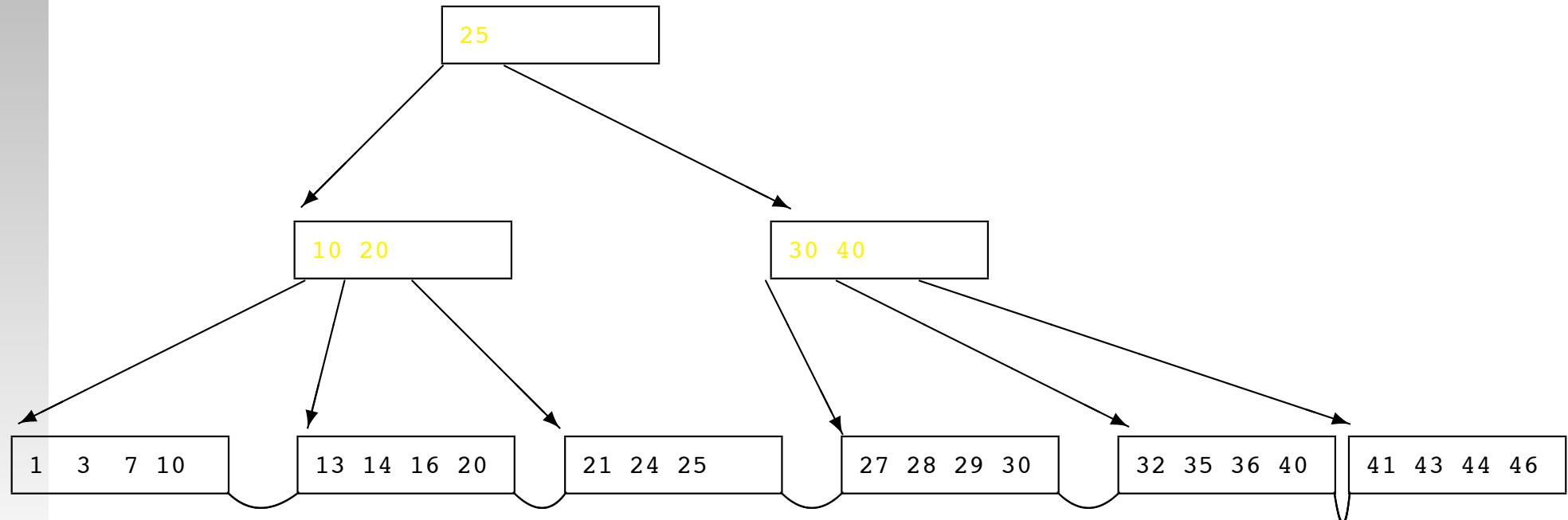
Eliminazione di un elemento interno

- Scambiare il contenuto dell'elemento da eliminare con quello più interno di uno dei due sottoalberi,
- Eliminare questo secondo elemento.



B⁺-Tree

- A differenza dei B-tree, tutti gli elementi sono memorizzati nelle foglie
- Le pagine interne contengono solo accessi e chiavi
- Permettono un attraversamento sequenziale efficiente
- Molto usati nei *file system* e nei *data base*



B⁺-Tree: applicazioni

- File system:
 - NTFS,
 - Reiser.
- Data base:
 - IBM DB2,
 - Informix,
 - Microsoft SQL Server,
 - Oracle,
 - Sybase,
 - MySQL.

Strutture dati

- **statiche:**

Lo spazio viene allocato tutto prima dell'utilizzo

- Vantaggi ?
- Svantaggi ?

- **dinamiche:**

Lo spazio necessario viene allocato durante l'utilizzo

- Vantaggi ?
- Svantaggi ?

Strutture dati statiche

Vettori

- **Vantaggi:**
 - Algoritmi semplici quando gli indici sono le chiavi
 - Accessi e modifiche veloci anche perchè lo spazio è già allocato
- **Svantaggi:**
 - Conoscenza iniziale numero massimo di elementi necessaria elementi
 - Spazio allocato tutto prima dell'utilizzo

Strutture dati dinamiche

Liste, alberi, ...

- **Vantaggi:**
 - Conoscenza iniziale numero massimo di elementi non necessaria
 - Nessuna requisito sul tipo di chiave
- **Svantaggi:**
 - Tempi di accesso variabili da elemento a elemento
 - Tempi necessari per le modifiche più lunghi che nel caso di strutture statiche
 - Algoritmi per la gestione talvolta complessi

Tabella *hash*

- Struttura statica o solo parzialmente dinamica
- Utilizzabile con qualsiasi tipo di chiave e anche quando l'insieme possibile delle chiavi è molto grande (rispetto a quelle effettivamente utilizzate)
- Si distingue fra
 - Tabella *hash* chiusa:
struttura statica
 - Tabella *hash* aperta:
struttura in parte statica e in parte dinamica

Insiemi grandi di possibili chiavi

Esempio: parole di un dizionario

- 26 lettere, in italiano parole fino a 30 lettere
(Psiconeuroendocrinoimmunologia)
- Combinazioni di 26 lettere con parola di lunghezza da 1 a 30 lettere:

$$26 + 26^2 + 26^3 + \cdots + 26^{30} =$$

$$= 26 \frac{26^{30} - 1}{25} \approx 10^{42}$$

- 200 parole per pagina, volume 2000 pagine: $2.5 \cdot 10^{36}$ volumi
- 4 decimetri cubi per volume: $6.25 \cdot 10^{32} m^3$
- il volume della terra è di circa $10^{12} m^3$
- I caratteri occuperebbero circa $3 \cdot 10^{31}$ Terabyte

Insiemi grandi di possibili chiavi

Quante parole ci sono in un dizionario ?

lingua	numero parole
francese	100.000
inglese	600.000 ... 800.000
italiano	250.000
spagnolo	200.000
tedesco	300.000 ... 500.000

Il numero di parola è quindi enormemente piccolo rispetto a quelle possibili combinando le lettere fra di loro.

Mappatura delle chiavi

- Dati l'insieme delle chiavi possibili e il numero massimo di chiavi esistenti
- Cercata una funzione in grado di mappare le chiavi esistenti in un insieme di indici di dimensione pari al numero di chiavi
- Missione impossibile se si cerca una funzione biunivoca
- Anche se la si trovasse occorrerebbe cambiarla per ogni insieme reale di dati
- Missione possibile con una funzione non biunivoca

Funzione di *hash*

- È una funzione non biunivoca che mappa le chiavi all'interno di un insieme di indici di dimensione data.
- Non essendo biunivoca esiste il rischio che la funzione mappi due chiavi diverse nello stesso indice. In questo caso si parla di *collisione*
- La scelta della funzione deve essere fatta in modo da minimizzare il numero di collisioni
- Le collisioni devono essere gestite

hash per chiavi intere

- $h(n) = n \bmod t$,
 n : chiave,
 t : dimensione dell'insieme degli indici
- $h(n) = (n^2/c) \bmod t$,
 $t \cdot c^2 = n_{max}^2$,
 $0 < n < n_{max}$
Esempio:
 $t = 100, n_{max} = 10^6, c = 10^5$
 $h(n) = (n^2/10^5) \bmod 100$
- *Mid-square Method*: si calcola n^2 e si prendono gli r bit centrali, quindi gli interi $0 \dots 2^r - 1$

<http://research.cs.vt.edu/AVresearch/hashing/midsquare.php>

hash per stringhe

- Dividere la stringa in blocchi di k caratteri
- Per ogni blocco $c_0c_1 \cdots c_{k-1}$ calcolare:
$$256^{k-1} \cdot \text{POS}(c_{k-1}) + 256^{k-2} \cdot \text{POS}(c_{k-2}) \\ \cdots 256 \cdot \text{POS}(c_1) + \text{POS}(c_0)$$
- Sommare i valori ottenuti per ogni blocco
- Applicare al risultato una funzione di *hash* per chiavi intere
- k va scelto in modo da evitare *overflow*

<http://research.cs.vt.edu/AVresearch/hashing/strings.php>

Gestione delle collisioni

- Tabella di *hash* chiusa:

In caso di collisione si applicano ulteriori funzioni di *hash* fino a trovare un indice libero.

- Tabella di *hash* aperta:

Se i è il valore restituito dalla funzione di *hash*, dall' elemento con corrispondente si diparte una lista che contiene gli altri elementi con lo stesso indice

Tabelle *hash* chiuse: collisioni

Linear probing:

Se una funzione di *hash* $h(x)$ produce una collisione, si calcola un nuovo indice con la funzione

$$h_1(x, p) = (h(x) + p) \bmod t \quad p = 1, 2, \dots$$

Il calcolo può essere ripetuto variando p (numero di tentativi) fino a trovare un indice libero.

Questo metodo ha il difetto di ammucchiare gli indici in vicinanza di $h(x)$ (*clustering phenomenon*).

Tabelle *hash* chiuse: collisioni

Quadratic probing:

-

$$h_1(x, p) = (h(x) + c_1 p + c_2 p^2) \bmod t$$

$$c_2 \neq 0$$

$$p = 1, 2, \dots$$

- Se $t = 2^n$, una buona scelta per le costanti è $c_1 = c_2 = 1/2$.

Perché ?

Tabelle *hash* chiuse: collisioni

Random probing:

$$h_1(x, p) = (h(x) + p) \bmod t$$

p : numero random $1 \leq p \leq t - 1$

Tabelle *hash*: collisioni e fattore di carico

- La probabilità di collisione è dipendente dal *fattore di carico (load factor)* definito come rapporto fra il numero di celle occupate e t .
- Se il fattore di carico supera un valore critico (p.e. 0.7) è necessario aumentare t , quindi allocare più memoria.

Tabelle *hash* chiusa: ricerca di un elemento

- Il primo passo consiste nel calcolare $h(x)$ e verificare se l'indice calcolato indica un elemento nella tabella (la tabella deve quindi indicare le posizioni vuote)
- Il secondo passo consiste nel verificare che l'elemento cercato è proprio quello con l'indice $h(x)$.
- Se non è il caso, si procede al calcolo di una nuova $h(x, p)$
- Fino a quando si continua ?

Tabelle *hash* chiusa: ricerca di un elemento

Soluzioni possibili:

- Inserire un contatore di registrazioni con $h(x)$
- Segnalare che su quell' elemento c'è già stata una collisione e quindi occorre continuare a cercare