

Alberi binari di ricerca

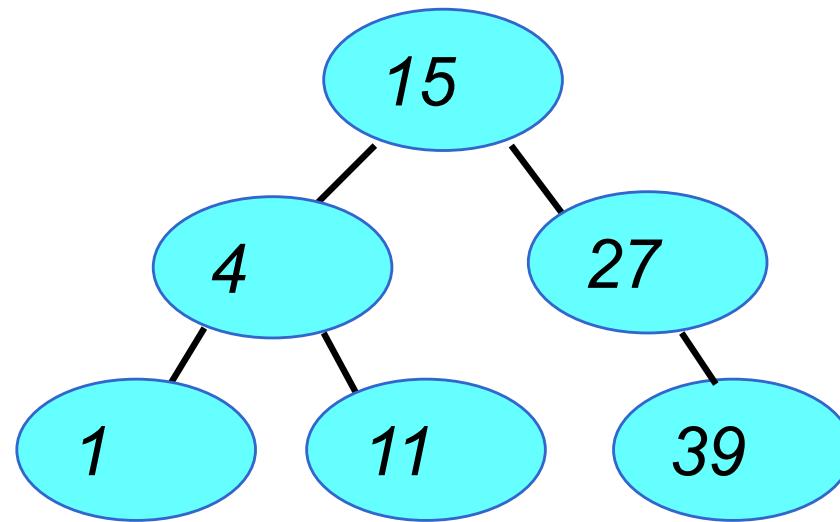
■ Obiettivi:

- Introdurre gli alberi binari di ricerca
- Mostrare la procedura di inserimento in alberi binari di ricerca
- Introdurre la nozione di albero bilanciato

Alberi binari di ricerca

- Un *albero binario di ricerca* è un albero binario i cui elementi rispettano la seguente proprietà:
- detto R il valore della radice:
 - il sottoalbero di sinistra contiene solo elementi *minori o uguali* a R
 - il sottoalbero di destra contiene solo elementi *maggiori* di R
- La proprietà vale anche per i sotto-alberi (che sono tutti alberi binari di ricerca)
- Sul tipo degli elementi deve esistere una relazione d'ordine totale (“minore”, <, isLess ...)

Alberi binari di ricerca

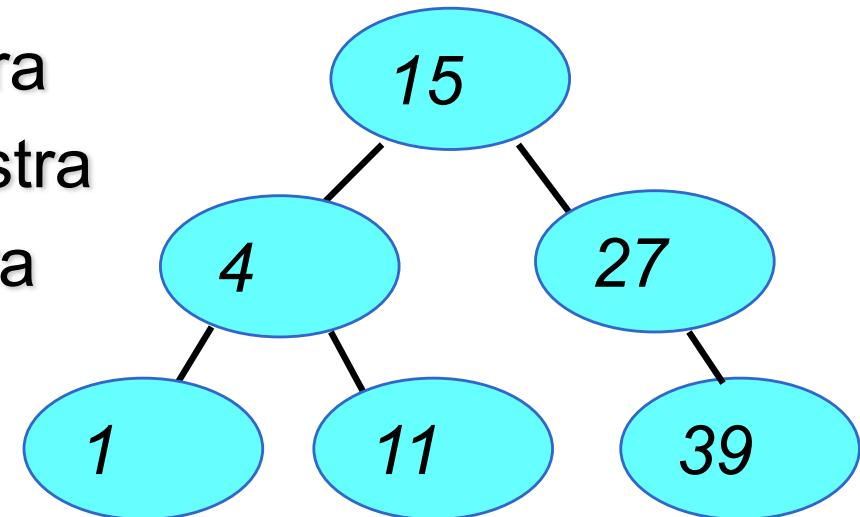


La visita inorder ci dà l'elenco ordinato: 1 4 11 15 27 39

Per la costruzione dell'albero, non ci si basa sulla cons_tree vista

Alberi binari di ricerca

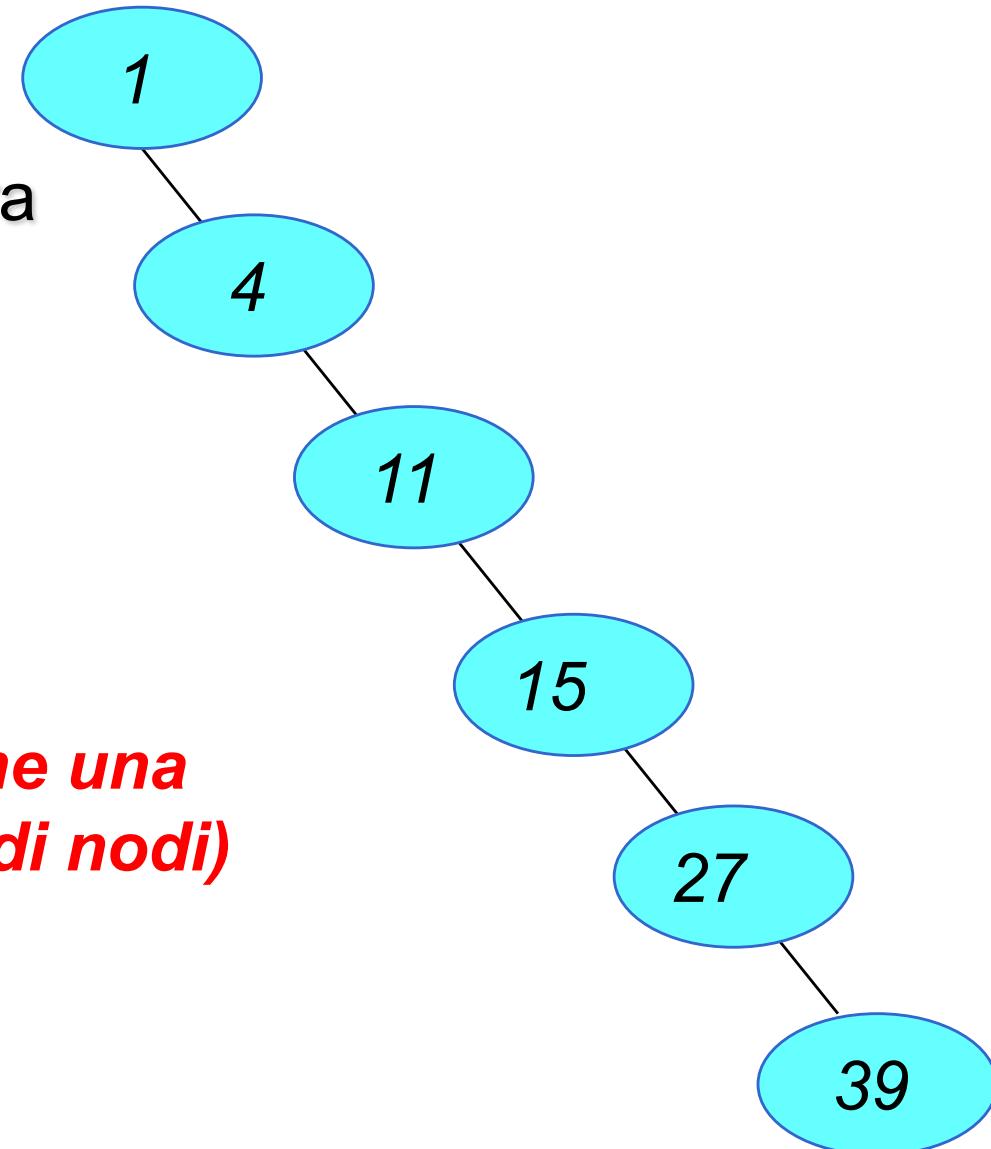
- 1) Inserisci il 15 → radice
- 2) Inserisci 27 → va a destra
- 3) Inserisci 4 → va a sinistra
- 4) Inserisci 39 → in fondo a destra
- 5) Inserisci 11 → sinistra, poi destra
- 6) Inserisci 1 → in fondo a sinistra



L'ordine con cui si inseriscono gli elementi è rilevante:
se essi sono inseriti in un ordine diverso, si ottiene, in generale, un albero diverso.

Esempio non bilanciato

- 1) Inserisci 1 → radice
- 2) Inserisci 4 → va a destra
- 3) Inserisci 11 → va a “”
- 4) Inserisci 15 → va a “”
- 5) Inserisci 27 → va a “”
- 6) Inserisci 39 → va a “”



L' albero degenera (è come una lista, un'unica sequenza di nodi)

Completamente sbilanciato

Binary Search Trees (BST)

- I BST servono per “gestire” insiemi ordinati in modo ottimizzato consentendo di avere minore complessità in:
 - Ricerche (se alberi ben bilanciati)
- maggiore velocità in:
 - Inserimenti e cancellazioni
- Può valere la pena di “**bilanciare**” un albero sbilanciato dopo vari inserimenti/cancellazioni per ottenere maggiore velocità nella ricerca

Algoritmi sui BST

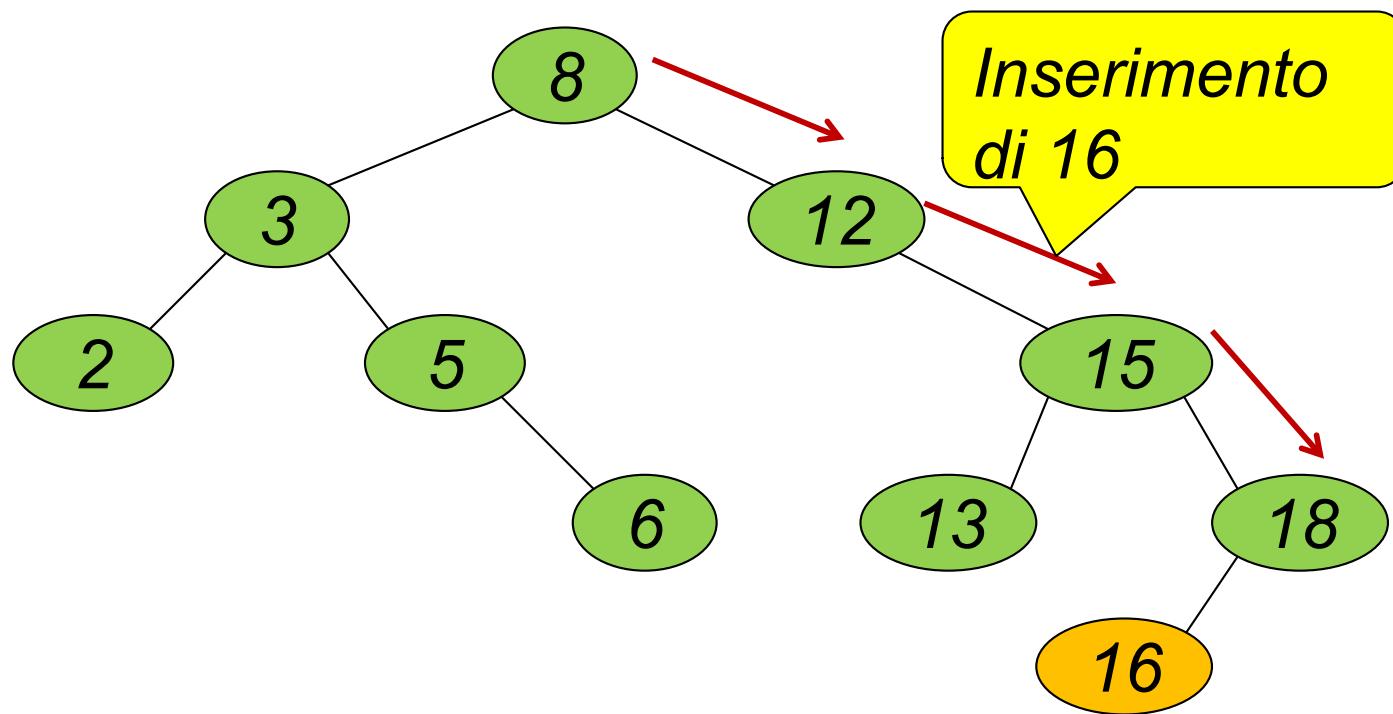
- Alcuni algoritmi interessanti
 - Inserimento
 - Ricerca
 - Cancellazione
- Le operazioni che modificano l' albero devono lasciare inalterate le proprietà di ordinamento

Realizzazione

- Fra gli elementi contenuti nei nodi dell' albero deve essere definita **una relazione d' ordine totale**
- In C, ADT degli elementi, esportando i prototipi delle funzioni predicative **isLess** e **isEqual**

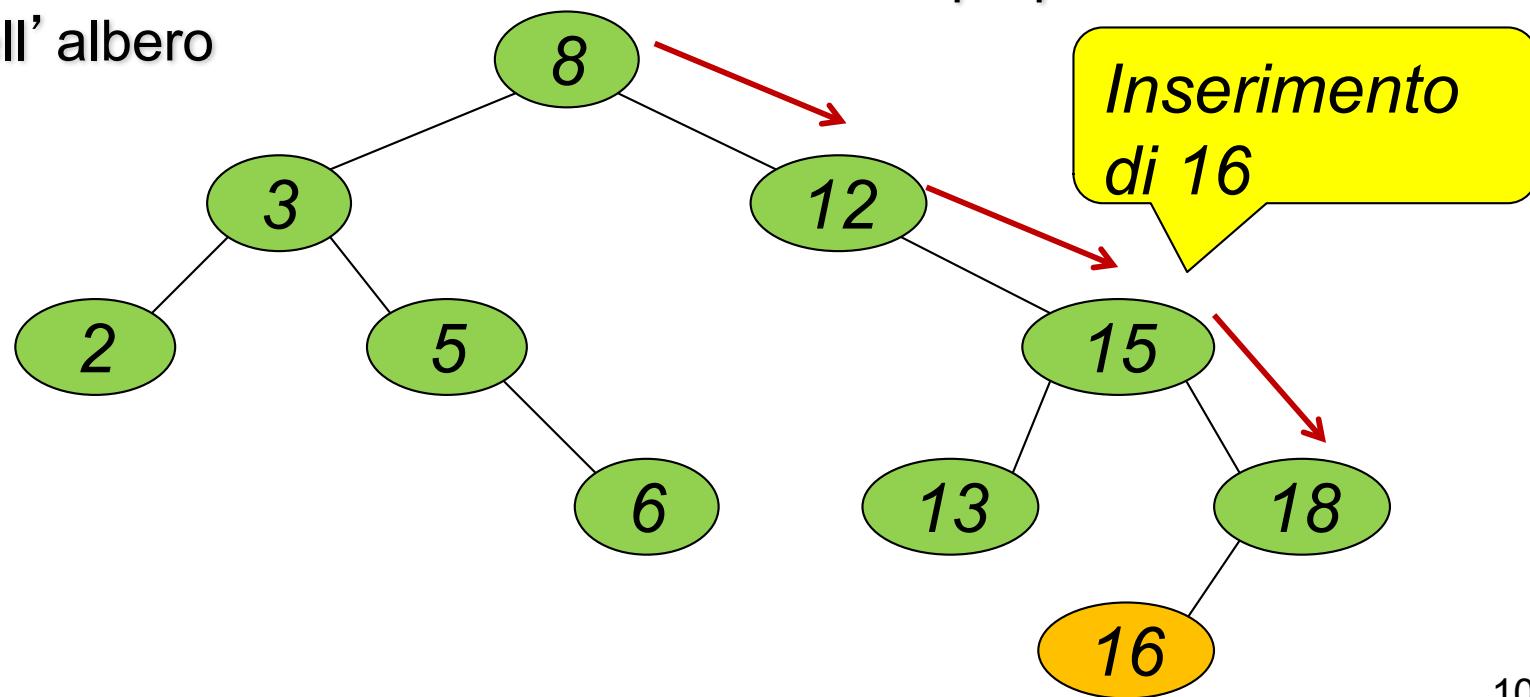
Inserimento – versioni

- **Iterativa** che sposta un puntatore a sinistra o a destra fino a raggiungere una foglia, e sotto essa collega il nuovo nodo
- **Ricorsiva**, che aggiorna il sottoalbero destro (o sinistro) chiedendo di inserire lì' il nuovo elemento



BST – Inserimento iterativo

- L' algoritmo di inserimento iterativo:
 - Aggiorna iterativamente un puntatore, cercando, nell' albero, la posizione corretta di inserimento, ovvero **il nodo che diventerà il padre di quello da inserire**
 - Una volta trovato il nodo, appende il nuovo nodo come figlio sinistro/destro in modo da mantenere la proprietà di ordinamento dell' albero

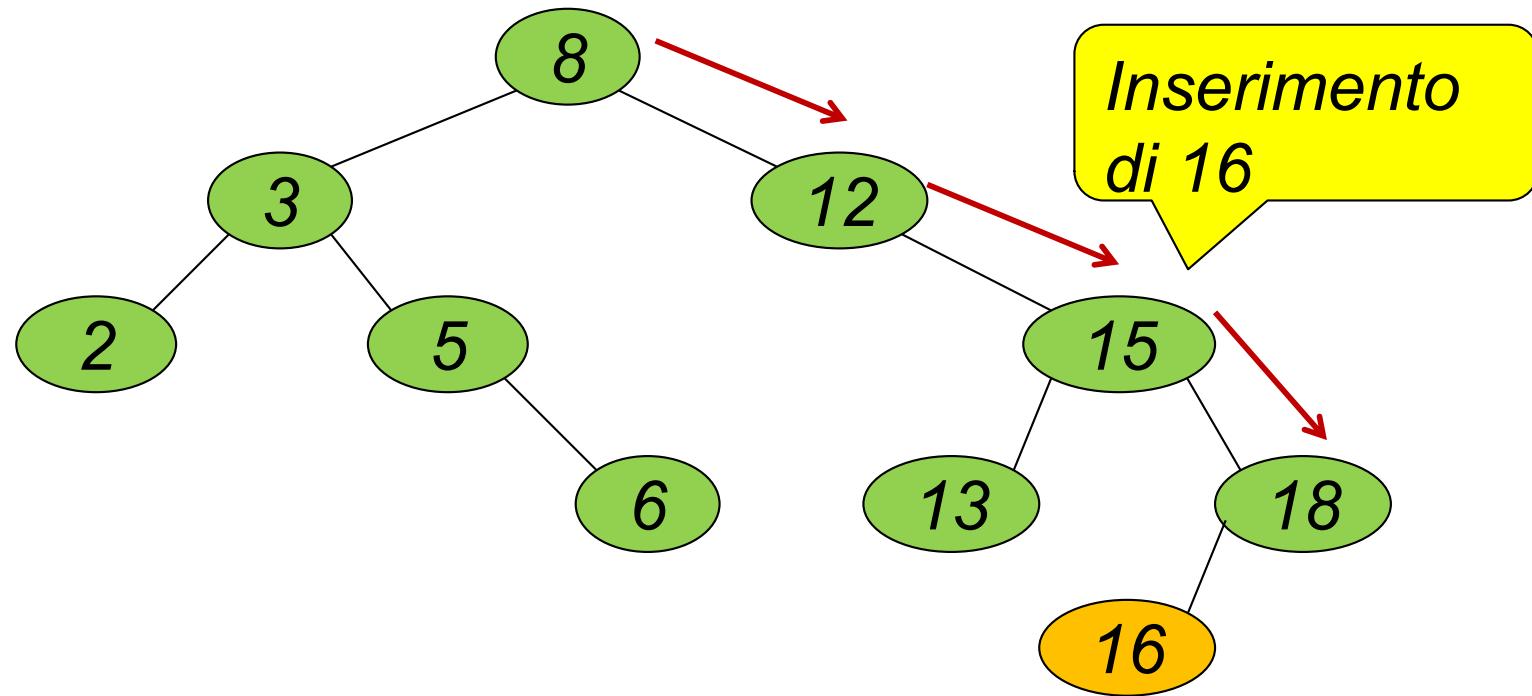




Inserimento - iterativo

- La versione iterativa dell' algoritmo di inserimento di un elemento e in un BST:
 - Utilizza un puntatore t (`tree`) aggiornato man mano, ma salvando in un secondo puntatore p (padre) il suo valore
 - Inserisce sotto il nodo padre trovato (quando t è diventato nullo), mantenendo le proprietà dell' albero di ricerca (quindi a sinistra o a destra)

BST – Inserimento iterativo

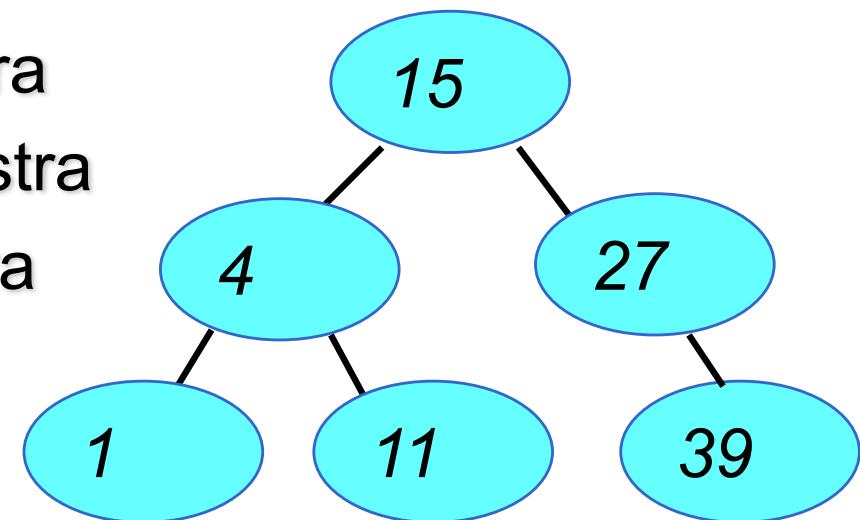


BST - Inserimento iterativo

```
tree ordins_it(element e,tree root)
{tree p=NULL, t=root;
 if (root==NULL) return cons_tree(e,NULL,NULL);
else
{ while (t!=NULL)
    if (e<=t->value)
        {p=t; t=t->left; }
    else
        {p=t; t=t->right; }
}
                                //p punta al padre
if (e<=p->value)
    p->left = cons_tree(e,NULL,NULL);
else
    p->right = cons_tree(e,NULL,NULL);
return root; }
```

BST - Esempio

- 1) Inserisci il 15 → radice
- 2) Inserisci 27 → va a destra
- 3) Inserisci 4 → va a sinistra
- 4) Inserisci 39 → in fondo a destra
- 5) Inserisci 11 → sinistra, poi destra
- 6) Inserisci 1 → in fondo a sinistra

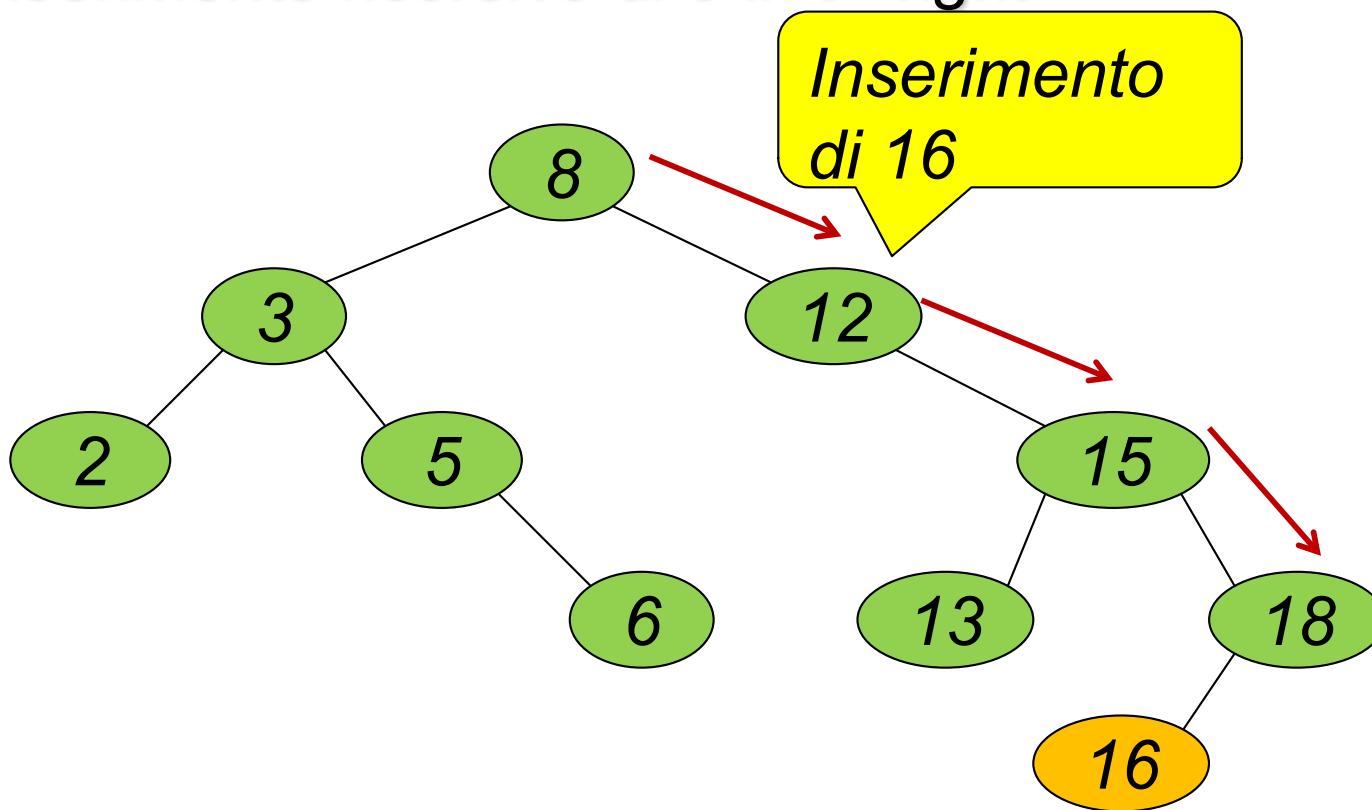


Stampa in ordine:

1 4 11 15 27 39

Inserimento – ricorsivo (v.1)

- Ricorsiva, che aggiorna il sottoalbero destro (o sinistro) inserirendo lì' il nuovo elemento
- Nell'esempio, t->right va aggiornato con l'esito dell'inserimento ricorsivo di e in t->right



Inserimento – ricorsivo (v.1)

1. Sia t il puntatore (`tree`) che identifica il sotto-albero in cui inserire l' elemento e
2. Se t è nullo (albero vuoto), restituire un nuovo nodo contenente l' elemento da inserire (e sottoalberi nulli) e **terminare**
3. Se e è minore dell' elemento contenuto nella radice di t
 1. Assegnare al sotto-albero sinistro di t il risultato dell' inserimento di e nel sotto-albero sinistro corrente
 2. Altrimenti, assegnare al sotto-albero destro di t il risultato dell' inserimento di e nel sotto-albero destro corrente
4. Restituire t

tree.c (8)

```
tree ord_ins(element e, tree t)
{
    if (t==NULL) //BST con duplicazioni
        return(cons_tree(e,NULL,NULL));
    else
        { if (e<=t->value)
            t->left = ord_ins(e,t->left);
        else
            t->right = ord_ins(e,t->right);
        return t;
    }
}
```

- E' tail ricorsiva?

Inserimento – ricorsivo (v.2)

- Abbiamo visto epr l'inserimento in lista anche una versione ricorsiva che richiamandosi sul resto della lista, ricostruisce a posteriori la lista dopo avere effettuato l'inserimento ordinato (versione 2)
- Posso fare lo stesso in un albero bianrio di ricerca, man mano che scorro la struttura dati spostandomi a sinistra o destra, devo ricostruire il tratto di albero percorso ...

Inserimento – ricorsivo (v.2)

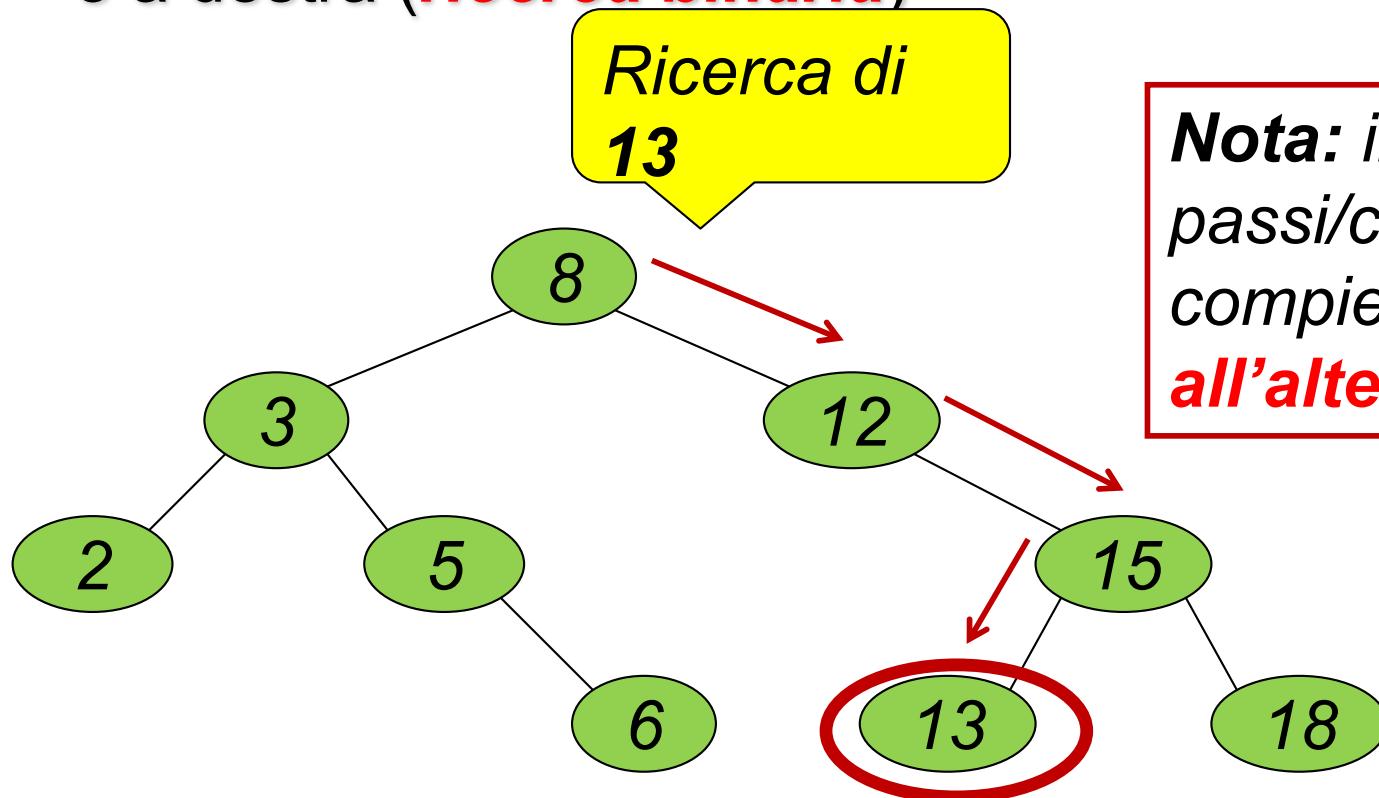
tree.c (8)

```
tree ordins(element e, tree t)
{
    if (t==NULL) //BST con duplicazioni
        return cons_tree(e,NULL,NULL);
    else
        if (e <= t->value)
            return cons_tree(t->value,ordins(e,t->left),t->right);
        else
            return cons_tree(t->value,t->left,ordins(e,t->right));
}
```

- Molto dispendiosa, duplica la porzione di struttura dati percorsa

Ricerca in un BST

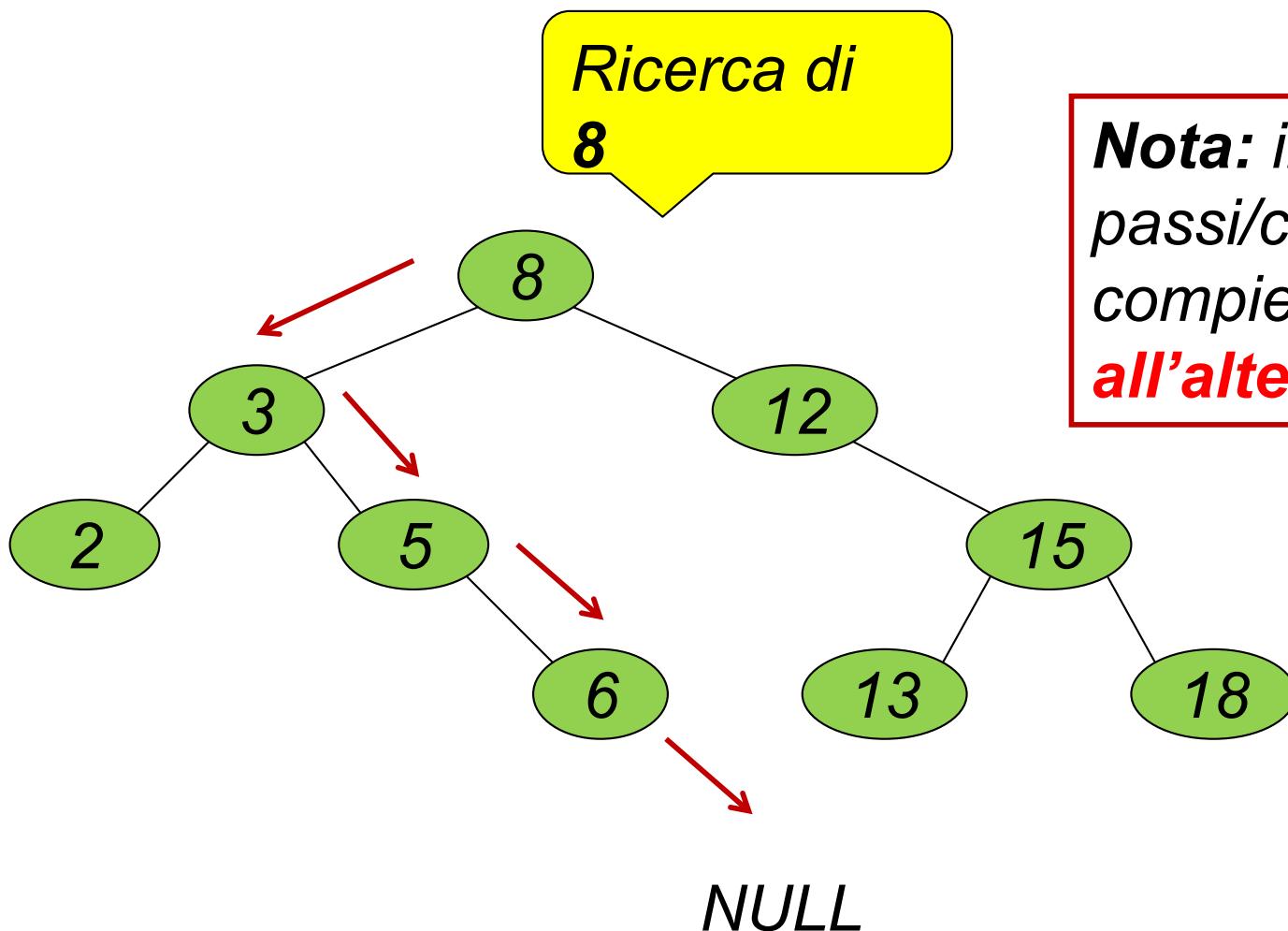
- Per le proprietà dei Binary Search Trees, è possibile decidere, per ogni nodo, se proseguire la ricerca a sinistra o a destra (**ricerca binaria**)



Nota: il numero di passi/confronti da compiere è, **al più, pari all'altezza dell'albero!**

Ricerca in un BST

- Anche in caso di fallimento della ricerca:



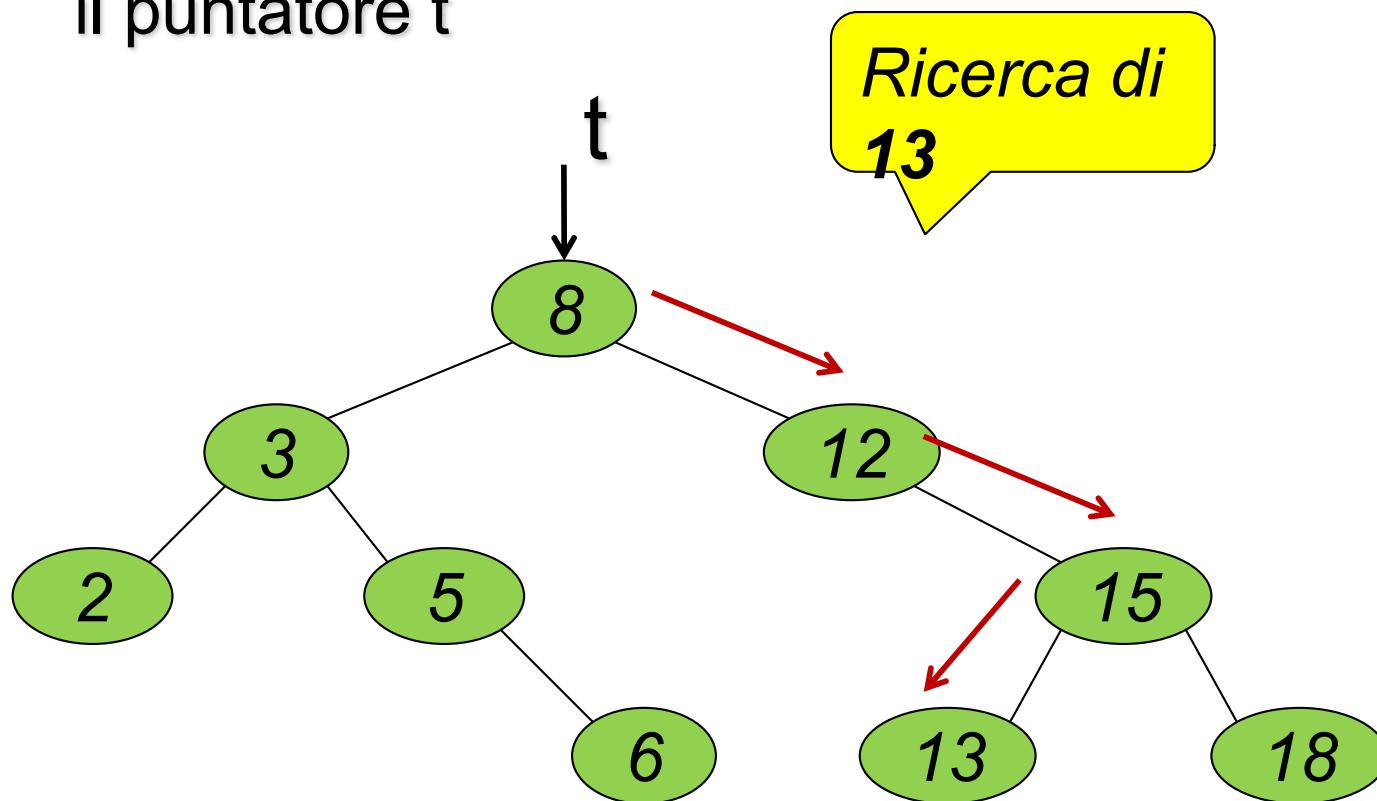
Nota: il numero di passi/confronti da compiere è, **al più, pari all'altezza dell'albero!**

Ricerca binaria - iterativa

1. Sia t un puntatore ad albero binario (`tree`) e gli sia inizialmente assegnata la radice dell' albero
2. Finché t non è nullo e la sua radice non contiene il valore cercato, confrontare il valore contenuto nella radice di t con il valore cercato
 - a. Se è uguale, restituire il valore trovato
 - b. Se è minore, assegnare a t il figlio destro e procedere con 2
 - c. Se è maggiore, assegnare a t il figlio sinistro e procedere con 2

Ricerca (iterativa) in BST

- E' possibile decidere, per ogni nodo, se proseguire la ricerca a sinistra o a destra (**ricerca binaria**), aggiornando il puntatore t

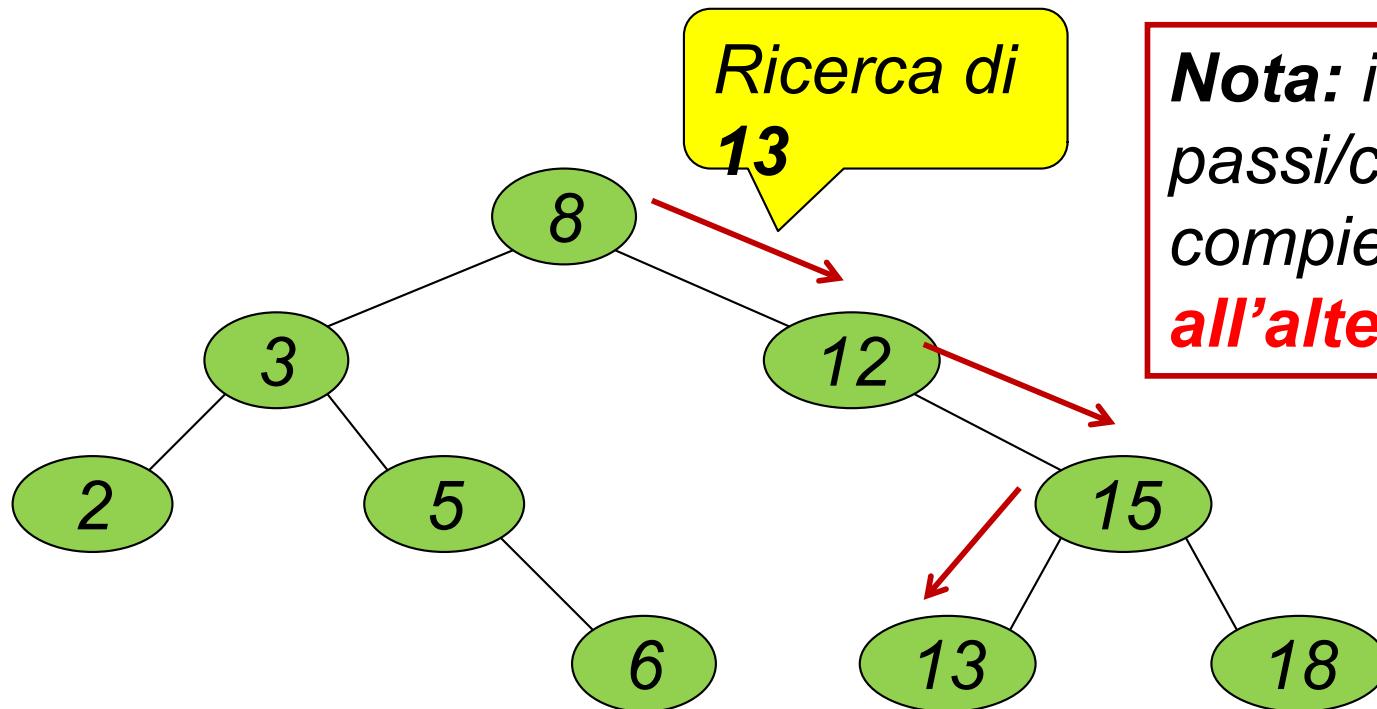


Ricerca (iterativa) in BST:

```
boolean member_ord_it(element e, tree t)
{ while(t!=NULL)
  { if (e==t->value) return true;
    else
      if (e<t->value)
        t=t->left;
      else
        t=t->right;
    }
  return false;
}
```

Algoritmi - Ricerca

- Per le proprietà dei Binary Search Trees, è possibile decidere, per ogni nodo, se proseguire la ricerca a sinistra o a destra (**ricerca binaria**)



Nota: il numero di passi/confronti da compiere è, **al più, pari all'altezza dell'albero!**

Ricerca binaria - ricorsiva

1. Se albero vuoto, falso
2. Se l' elemento cercato è uguale al contenuto della radice, vero
3. Se l' elemento da cercare è minore dell' elemento contenuto nella radice del sotto-albero corrente
 - a. Cercare nel sottoalbero di sinistra
 - b. Altrimenti, cercare nel sottoalbero di destra

Ricerca ricorsiva in BST:

```
boolean member_ord(element e, tree t)
{ if (t==NULL) return false;
else
    if (e==t->value) return true;
else
    if (e < t->value)
        return member_ord(e, t->left);
    else return member_ord(e, t->right) ;
}
```

- E' tail ricorsiva?

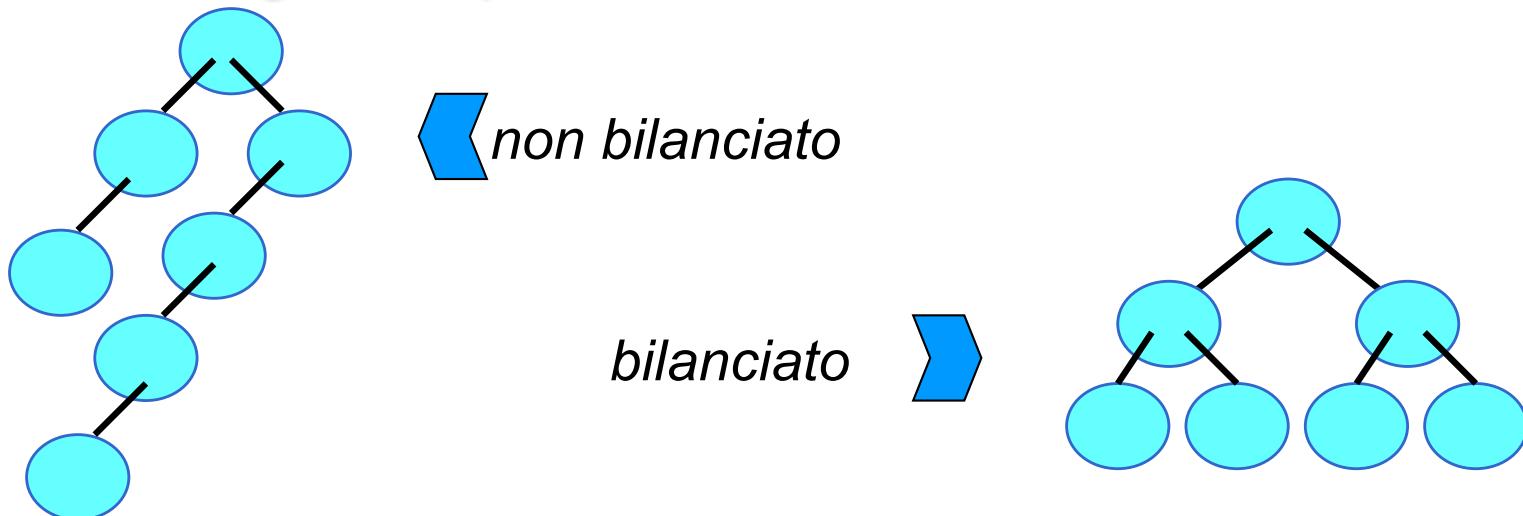
Complessità ricerca in BST

Un albero binario di ricerca *riduce notevolmente la complessità della ricerca di un elemento*, perché esclude metà albero a ogni confronto

- l'esito del confronto dice da che parte sta l'elemento:
 - nel sottoalbero di sinistra, se l'elemento cercato è *minore* della radice
 - nel sottoalbero di destra, se l'elemento cercato è *maggior*e della radice.
- Il numero di confronti è (**nel caso peggiore**) **proporzionale alla profondità (altezza) dell'albero.**
- È perciò importante mantenere l'albero *bilanciato* (tutti i cammini dalla radice alle foglie hanno più o meno la stessa altezza).

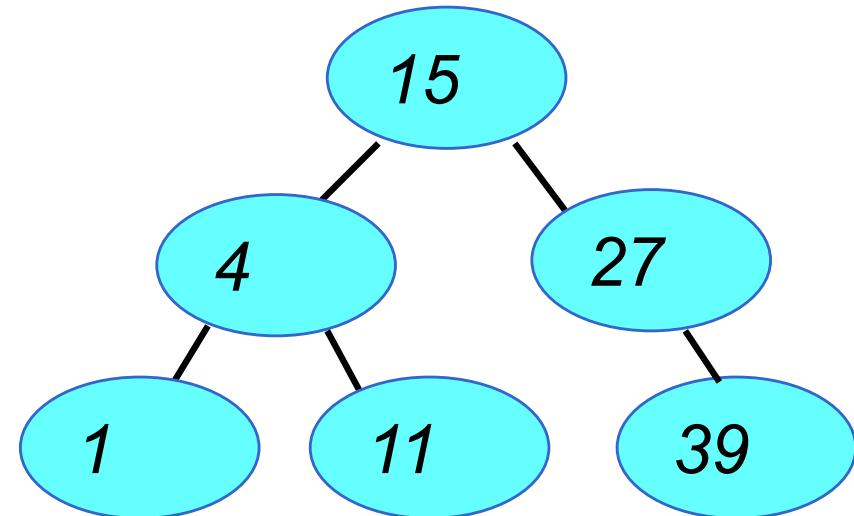
Alberi bilanciati

- Un albero i cui elementi sono distribuiti uniformemente fra i sottoalberi si dice *bilanciato*
- La ricerca binaria ha costo minore se gli alberi sono bilanciati
- Esistono algoritmi per bilanciare alberi non bilanciati



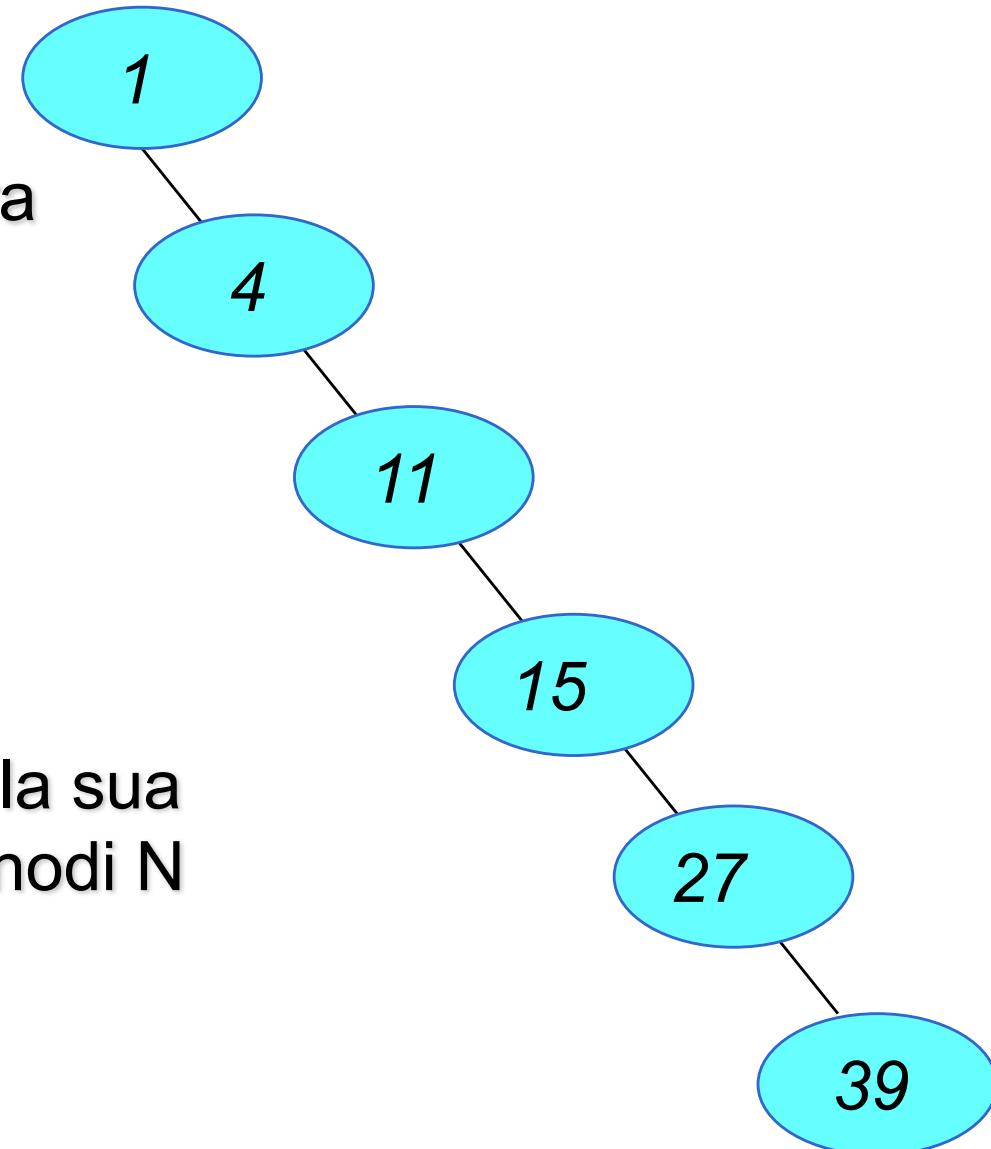
Esempio BST bilanciato

- Nei **BST bilanciati**, per ciascun nodo, l' altezza del sottoalbero sinistro e destro differiscono al più di una unità



Esempio BST non bilanciato

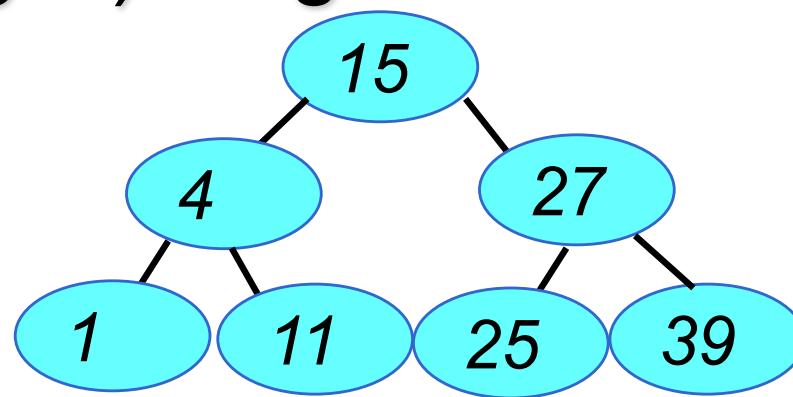
- 1) Inserisci 1 → radice
- 2) Inserisci 4 → va a destra
- 3) Inserisci 11 → va a “”
- 4) Inserisci 15 → va a “”
- 5) Inserisci 27 → va a “”
- 6) Inserisci 39 → va a “”



L'albero degenera (lista), e la sua altezza è pari al numero di nodi N (meno 1)

Correlazione tra altezza e numero nodi

- In un albero binario completo ciascun nodo (tranne le foglie) ha grado di uscita uguale a 2



- Albero completo, sia p l'altezza (o profondità), il numero di nodi è: $N=2^{p+1}-1$
- Se albero (non completo) di altezza p , al massimo $2^{p+1}-1$ nodi

Complessità della ricerca in BST

- Nel caso di **BST bilanciati**, nel caso peggiore, la ricerca opera $K+1$ confronti (dove K è altezza dell' albero) per raggiungere, lungo un cammino, una foglia
- Ad ogni passo, si dimezza lo spazio di nodi, per cui dopo K passi, ha operato K dimezzamenti del numero di nodi considerati
- **RISULTATO: Ricerca binaria** = per esplorare uno spazio di N elementi occorrono **al più $k+1$ confronti, pari a $O(\log_2 N)$ confronti per alberi bilanciati**

Alberi bilanciati

- Il problema di BST è che **normalmente NON sono bilanciati**:
 - Inserimenti e cancellazioni sbilanciano l' albero
 - Se l' albero non è correttamente bilanciato le operazioni (tutte) costano “parecchio”
- Soluzione: alberi che si autobilanciano
 - AVL (Adel'son-Vel'skii-Landis) trees
 - Red-Black trees
 - ...

Alberi AVL

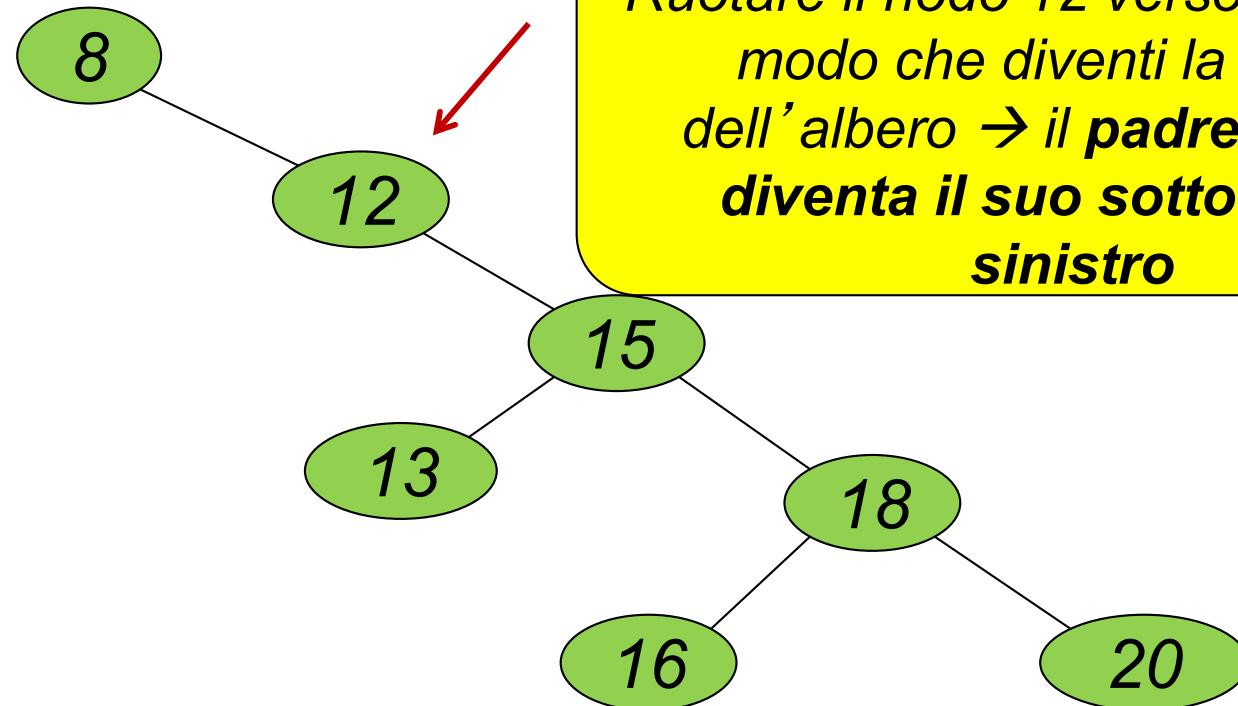
- Un albero AVL è un Albero Binario di Ricerca **bilanciato**
- Un nodo si dice bilanciato quando l' altezza del sotto-albero sinistro **differisce** dall' altezza del sotto-albero destro **di al più una unità**
- Un albero si dice **bilanciato** quando **tutti i nodi sono bilanciati**
- Le operazioni sono le stesse che si possono eseguire su un albero binario di ricerca

Alberi AVL

- Si supponga di partire con un albero bilanciato, secondo la definizione data in precedenza
 - Una serie di inserimenti/cancellazioni può sbilanciare l' albero
 - Opportune *rotazioni* sono in grado di ribilanciare l' albero
-
- Naturalmente i costi di inserimento/cancellazione crescono di molto, ma la ricerca rimane sempre molto efficiente! ($O(\log_2 N)$ se N nodi)

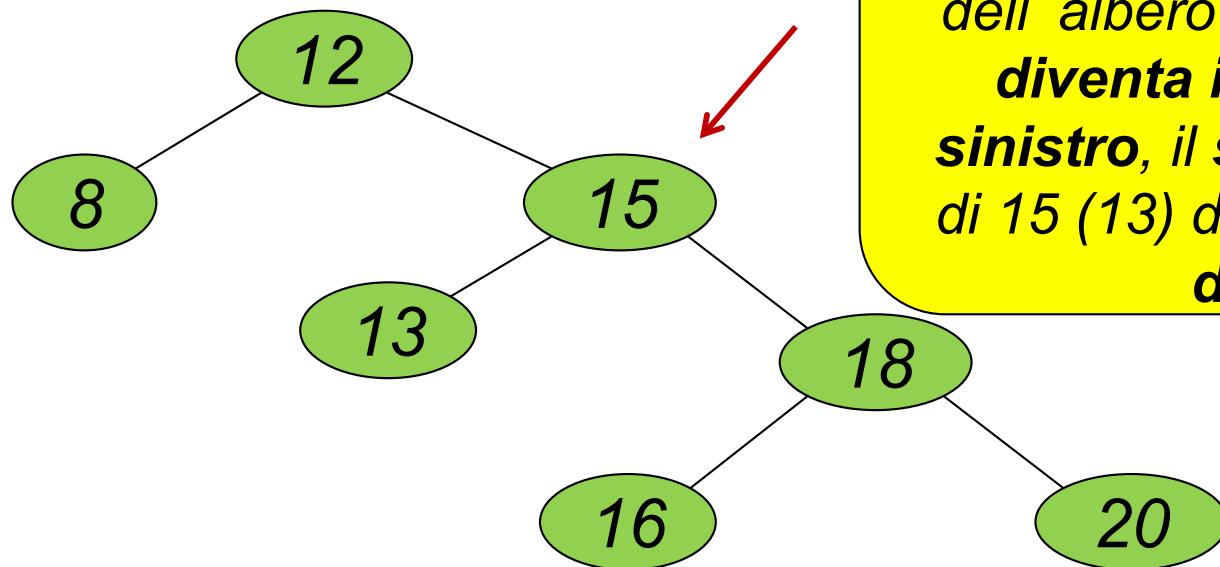
Rotazioni

- Si supponga di disporre di un albero sbilanciato – è possibile bilanciarlo tramite opportune rotazioni



Rotazioni

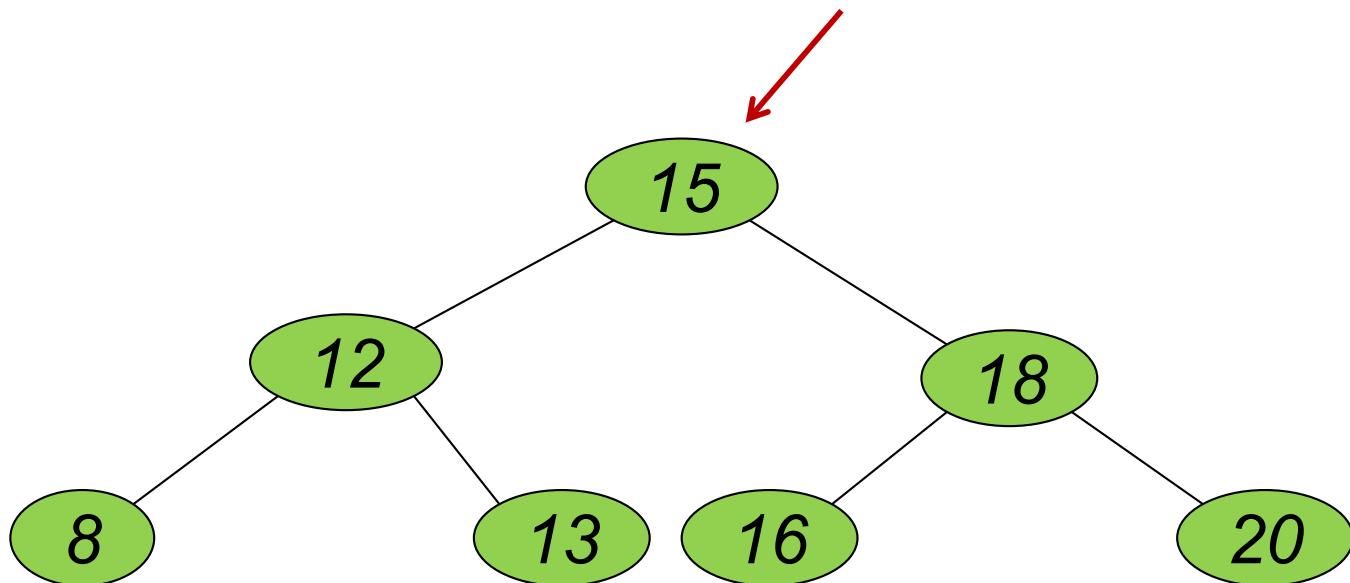
- Rotazione 1: il nodo 12 diventa la radice



*Ruotare il nodo 15 verso **sinistra** in modo che diventi la **radice** dell' albero → il **padre** di 15 (12) diventa **il suo sotto-albero sinistro**, il **sotto-albero sinistro** di 15 (13) diventa **il sotto-albero destro** di 12*

Rotazioni

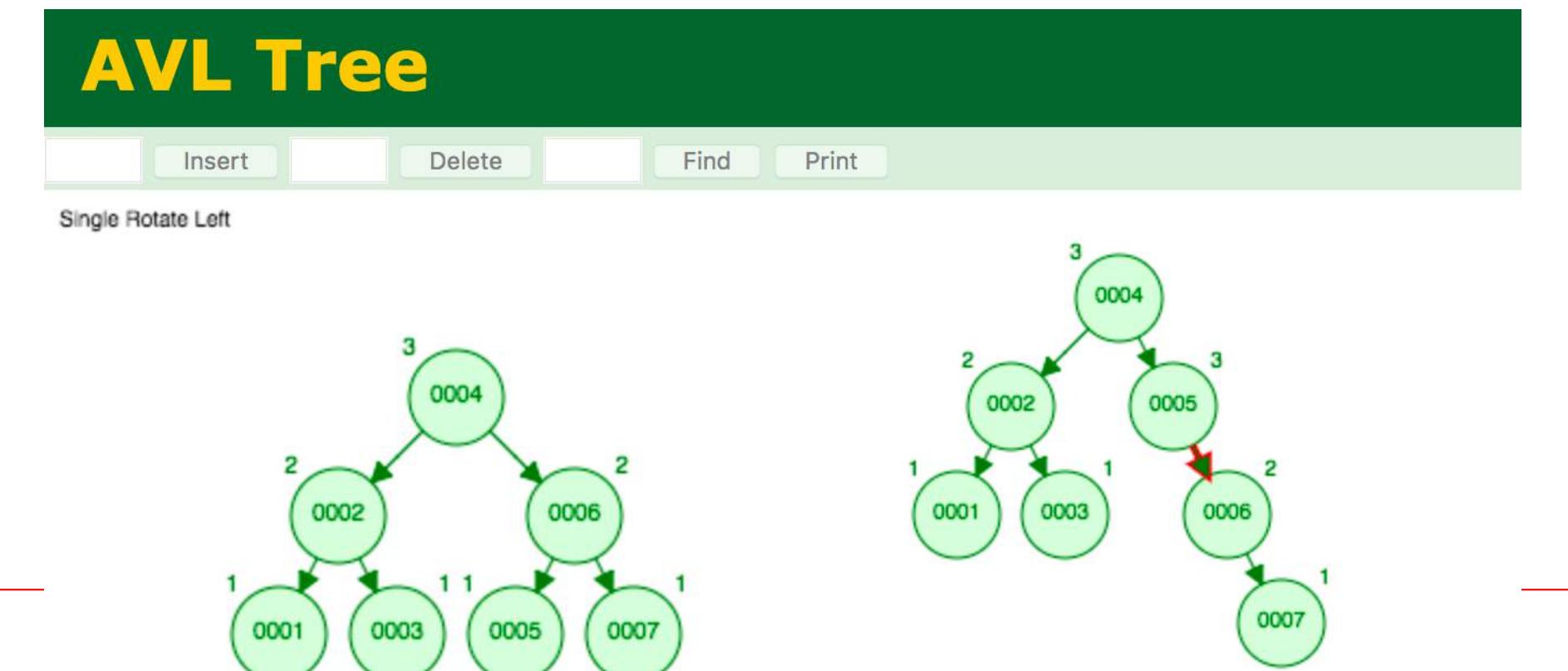
- Rotazione 2: il nodo 15 diventa la radice e l' albero risulta bilanciato



Per giocare un po' ...

<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

- Applet java per giocare con alberi AVL, con possibilità di inserire, cancellare, ruotare e vedere come si comportano i vari alberi



Per giocare un po' ...

<http://www.geeksforgeeks.org/avl-tree-set-1-insertion/>

- Realizzazione in C (e Java) delle operazioni su AVL tree (insert, search, etc)

GeeksforGeeks
A computer science portal for geeks

Google Custom Search

OG Algo DS Languages Interview Students GATE CS Subjects Quizzes

DS

Quick Links for Binary Search Tree

Quizzes on BST

Quizzes on Balanced BST

Binary Search Tree

Search and Insert in BST

Deletion from BST

Data Structure for a single resource reservations

Advantages of BST over Hash Table

Check if a binary tree is BST or not

Sorted Array to Balanced BST

Print BST keys in the given range

Binary Search Tree I Set 1 (Search and Insertion)

The following is definition of Binary Search Tree(BST) according to [Wikipedia](#)

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

There must be no duplicate nodes.

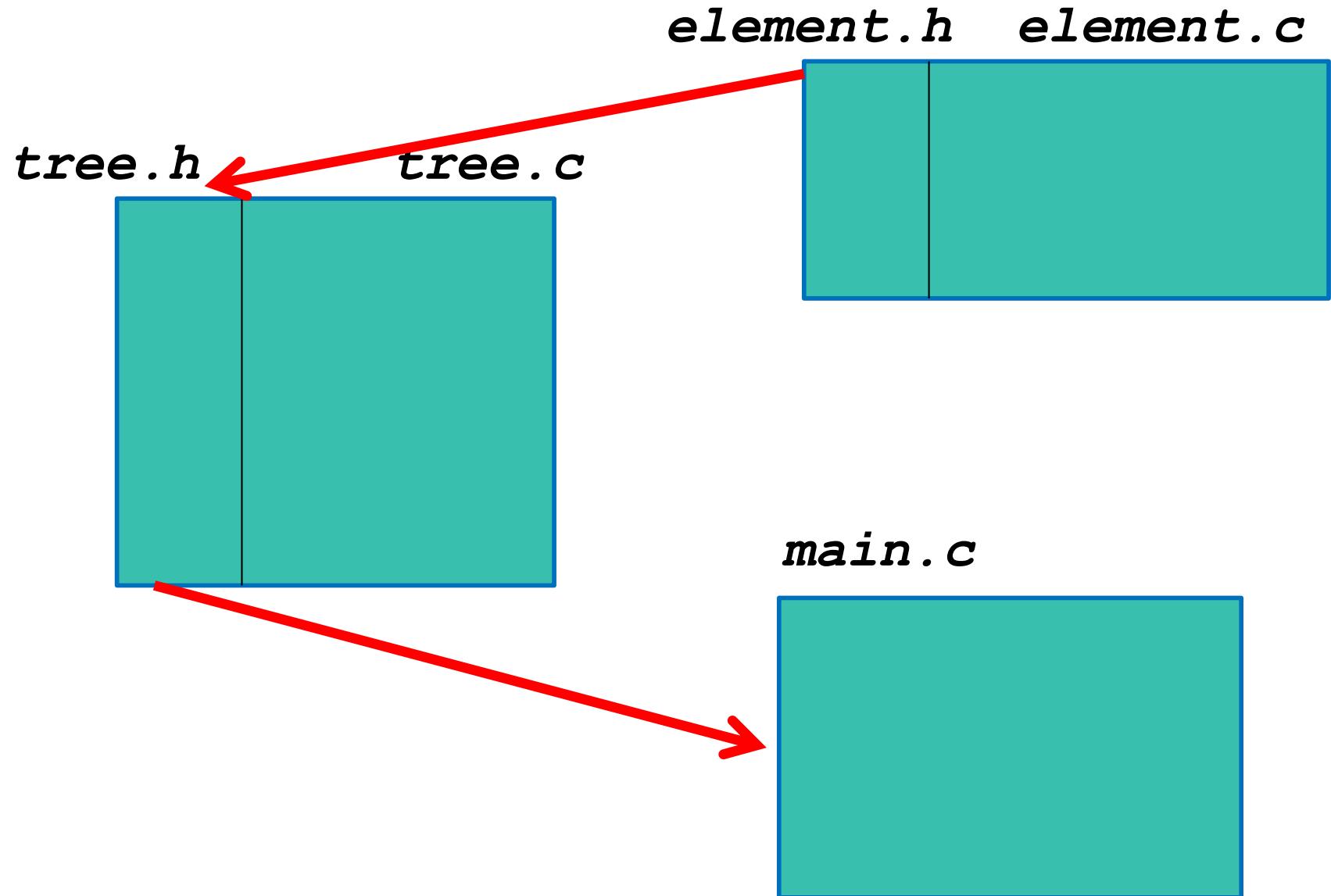
```
graph TD; 8((8)) --> 3((3)); 8 --> 10((10))
```

ADT Albero binario di ricerca

■ Obiettivi:

- Introdurre l'ADT per gli alberi binari di ricerca
- Riorganizzare il codice delle funzioni viste per componenti

Componenti



Laboratorio:

Dati i file *element.c* e *element.h* che realizzano l' ADT *element* (come intero)

Si implementino i file *tree.h* e *tree.c* , con le operazioni primitive, visite, e **inserimento in albero binario di ricerca** (**manca cons_tree e ord_ins**).

Il *main* da realizzare deve leggere una sequenza di interi e inserire ogni elemento letto in un **albero binario di ricerca** e infine stampare il contenuto dell'albero **con la visita in ordine**

Il *main* deve poi leggere da input un valore intero e cercarlo nell'albero binario di ricerca. **Quale algoritmo di ricerca?**

ADT ELEMENT: element.h

Header element.h contiene:

- *definizione del tipo element*
- *dichiarazioni delle varie funzioni fornite*

```
#ifndef ELEMENT_H
#define ELEMENT_H

typedef int element;
typedef enum { false, true } boolean;

boolean isLess(element, element);
boolean isEqual(element, element);
element getElement(void);
void printElement(element);

#endif
```

ADT ELEMENT: element.c

```
#include "element.h"
#include <stdio.h>

boolean isEqual(element e1, element e2) {
    return (e1==e2); }

boolean isLess(element e1, element e2) {
    return (e1<e2); }

element getElement(void) {
    element el;
    scanf("%d", &el);
    return el; }

void printElement(element el) {
    printf("%d", el); }
```

Il cliente (main.c)

```
#include <stdio.h>
#include "tree.h"

main() {
    tree t = emptytree();
    element el;
    do { printf("\n Introdurre valore:\t");
        el=getElement();
        t = ordins_it(el, t);
    } while (!isEqual(el, 0));
    inorder(t);
}
```

Esercizi molto consigliati...

- Calcolare l' altezza di un albero
- Calcolare il bilanciamento di un nodo
(differenza fra le altezze dei sotto-alberi sinistro e destro)
- Albero di interi, calcolo della somma dei valori dei nodi; trovare il valore maggiore; etc



To Do: altezza di un albero

```
int height (tree t)
{ if (empty(t)) return 0;
  else return (max(height_aux(left(t)),
                     height_aux(right(t))) );
}
```

```
int height_aux (tree t)
{ if (empty(t)) return 0;
  else return (1+ max(height_aux(left(t)),
                       height_aux(right(t)))) ;
}
```



To Do:

- Scrivere una funzione che calcola il bilanciamento di un nodo (di radice di t)

```
int balance (tree t)
{ if (empty(t)) return 0;
else
    return (height(left(t))-height(right(t)));
}
```