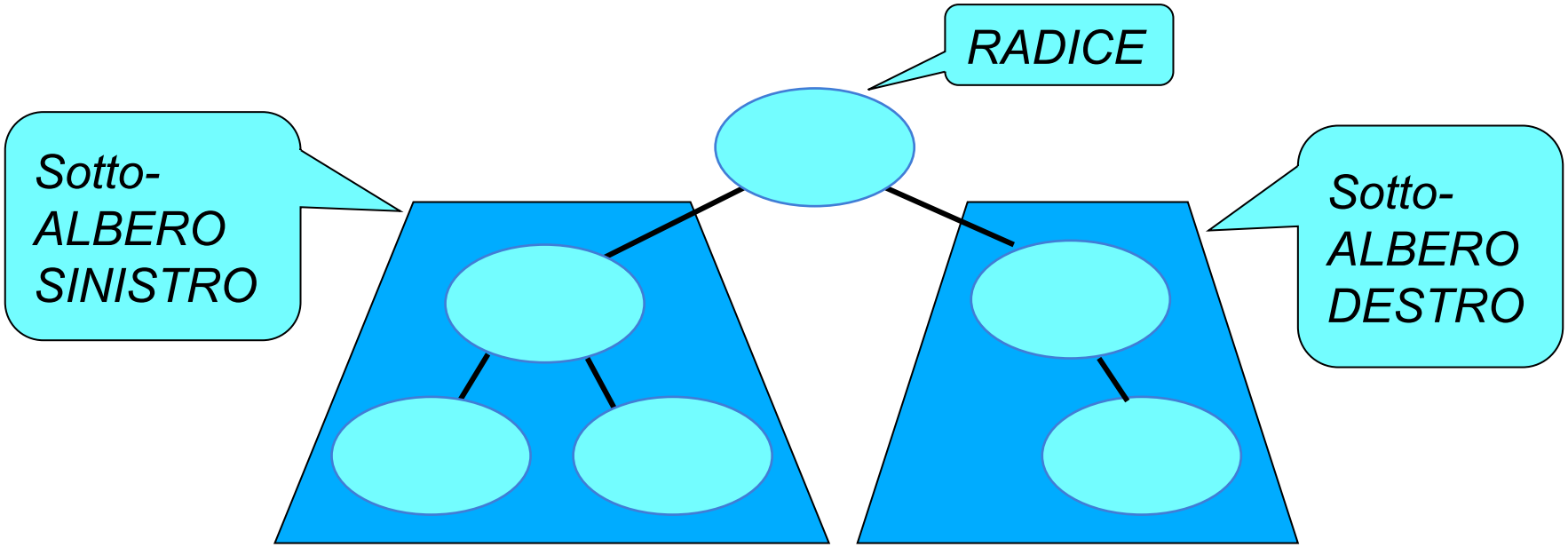
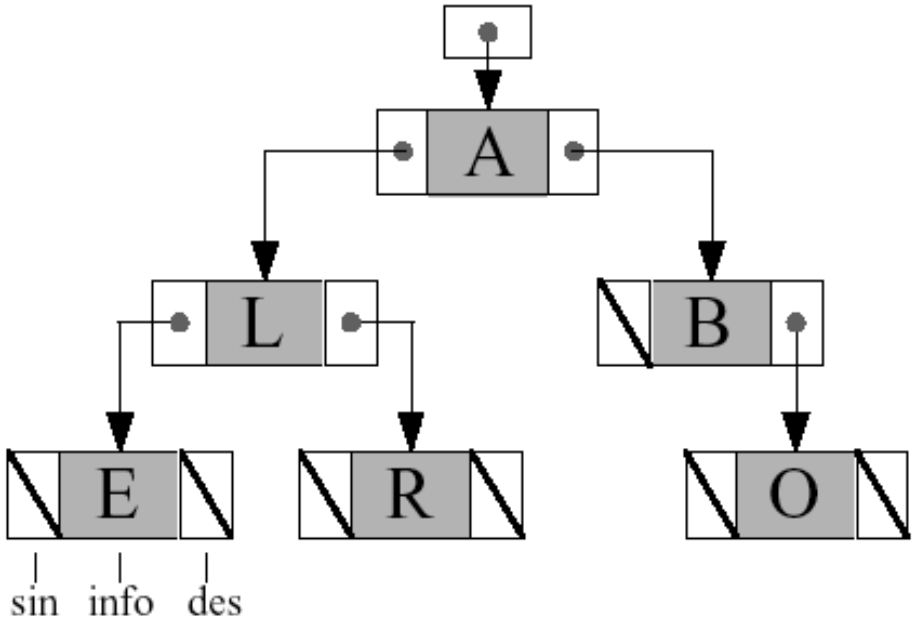
ALBERI BINARI

In un albero *binario*, abbiamo

- **un elemento**
- **al più due sottoalberi figli, *sinistro* e *destro***



Rappresentazione collegata



- Vantaggi:
  - si ha un'occupazione di memoria efficiente;
  - operare modifiche è agevole;
  - le procedure di visita sono realizzate in modo diretto (ricorsione).

Cosa cambia rispetto alle liste?

- Sempre un puntatore radice (eventualmente NULL)
- Due successori per ciascun nodo (campi left e right di ogni nodo)
- L'elaborazione del contenuto della struttura dati (per stampa, ricerca, conteggio, etc etc) si complica (elaborazione sequenziale per la lista, sostituita da metodiche di visita per l'albero)

ADT ALBERO BINARIO

OPERAZIONI PRIMITIVE DA REALIZZARE

Operazione	Descrizione
cons_tree: D x tree x tree -> tree	Costruisce un nuovo albero, che ha l'elemento fornito come radice con figli i due sottoalberi dati
root: tree -> D	Restituisce l'elemento radice dell'albero
left: tree -> tree	Restituisce il sottoalbero sinistro
right: tree -> tree	Restituisce il sottoalbero destro
emptytree: -> tree	Restituisce (costruisce) l'albero vuoto
empty: tree-> boolean	Restituisce vero se l'albero dato è vuoto, falso altrimenti

ADT ALBERO BINARIO

OPERAZIONI DERIVATE DA REALIZZARE

Operazione	Descrizione
preorder: tree	Visita in preordine (ordine anticipato)
inorder: tree	Visita in ordine (“” simmetrico)
postorder: tree	Visita in postordine (“” ritardato)
member: D x tree -> boolean	Ricerca di un elemento nell’albero
nnodi: tree -> int	Conta il numero dei nodi

tree.h (1)

```
typedef char element;          //qui o ADT ...
typedef enum {false, true}  boolean;

typedef struct nodo
          {element  value;
       struct nodo *left, *right; } NODO;
typedef NODO * tree;
```

tree.h (2)

```
boolean empty(tree);    // OP. PRIMITIVE
tree  emptytree(void);
element root(tree);
tree  left(tree);
tree  right(tree);
tree  cons_tree(element,tree ,tree);

void preorder(tree);      // OP. DERIVATE
void inorder(tree);
void postorder(tree);
boolean member(element, tree);
int nnodi(tree);
```

tree.c (1)

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

boolean empty(tree t)
/* test di albero vuoto */
{ return (t==NULL); }

tree  emptytree(void)
/* inizializza un albero vuoto */
{ return NULL; }
```

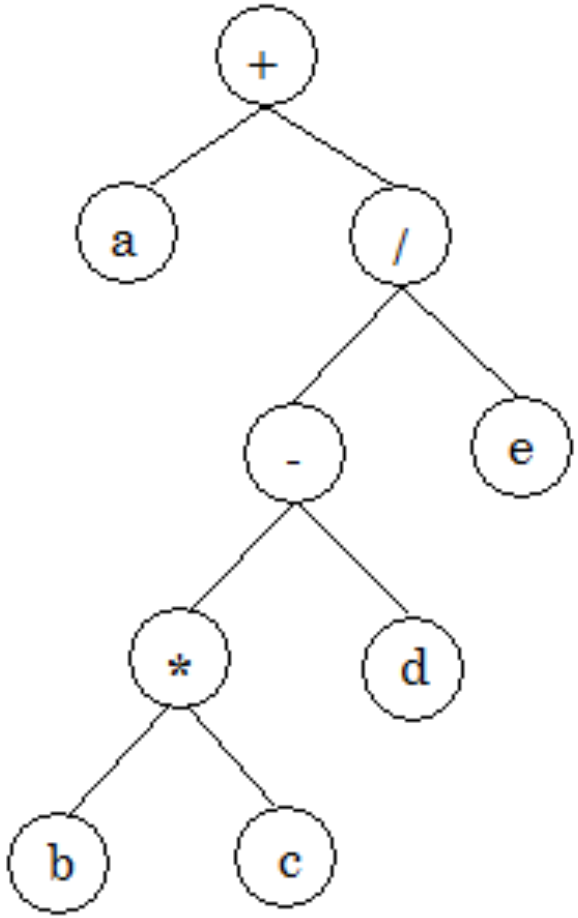
tree.c (3)

```
tree  cons_tree(element e, tree l, tree r)
/* costruisce un albero che ha nella
 radice e; per sottoalberi sinistro e
 destro l ed r rispettivamente    */
{ tree t;
  t = (NODO *) malloc(sizeof(NODO));
  t-> value = e;
  t-> left = l;
  t-> right = r;
  return (t);  }
```

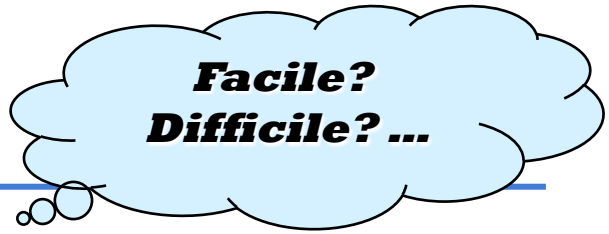
tree.c (2)

```
element root (tree t)
/* restituisce la radice dell'albero t */
{ if (empty(t)) abort();
  else return(t->value); }

tree  left (tree t)
/* restituisce il sottoalbero sinistro */
{ if (empty(t)) return(NULL);
  else return(t->left); }

tree  right (tree t)
/* restituisce il sottoalbero destro */
{ if (empty(t)) return(NULL);
  else return(t->right); }
```

Esempio: a+(b*c – d)/e



To do:



Costruiamo l'albero in figura (albero di caratteri) e stampiamolo (in ordine)

Ci servono tre funzioni:
main
cons_tree
inorder

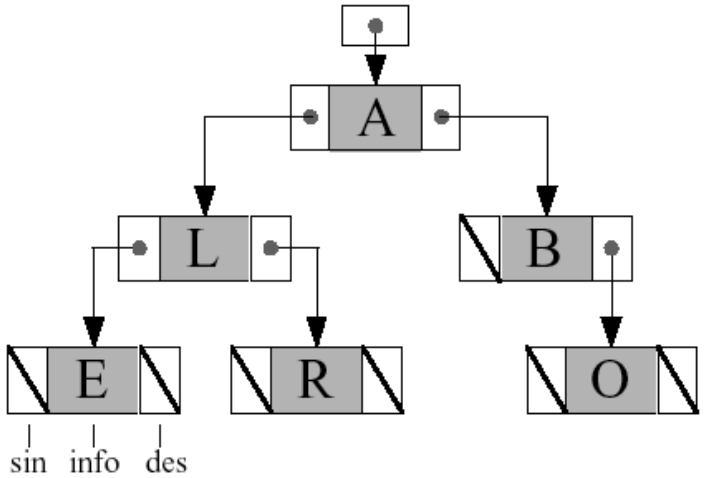
main.c

```
#include <stdio.h>
#include "tree.h"
void main (void)
{ tree    t1,t2;
  t1=cons_tree('b',NULL,NULL);
  t2=cons_tree('c',NULL,NULL);
  t1=cons_tree('*',t1,t2);
  t2=cons_tree('d',NULL,NULL);
  t1=cons_tree('-',t1,t2);
  t2=cons_tree('e', NULL,NULL);
  t2=cons_tree('/',t1,t2);
  t1=cons_tree('a', NULL,NULL);
  t1=cons_tree('+',t1,t2);
  printf("\nStampa in ordine\n");
  inorder(t1); }
```

Cosa cambia rispetto alle liste?

- Sempre un puntatore radice (eventualmente NULL)
- Due successori per ciascun nodo (campi left e right per ogni nodo)
- L'elaborazione del contenuto della struttura dati (per stampa, ricerca, conteggio, etc etc) si complica (elaborazione sequenziale per la lista, sostituita da metodiche di visita per l'albero)

Visite di alberi binari



- Definite in modo ricorsivo:
  - Preorder (radice, sotto-albero sinistro, destro)
  - Postorder (sotto-albero sinistro, destro, radice)
  - Inorder (sotto-albero sinistro , radice, sotto-albero destro)

tree.c (4)

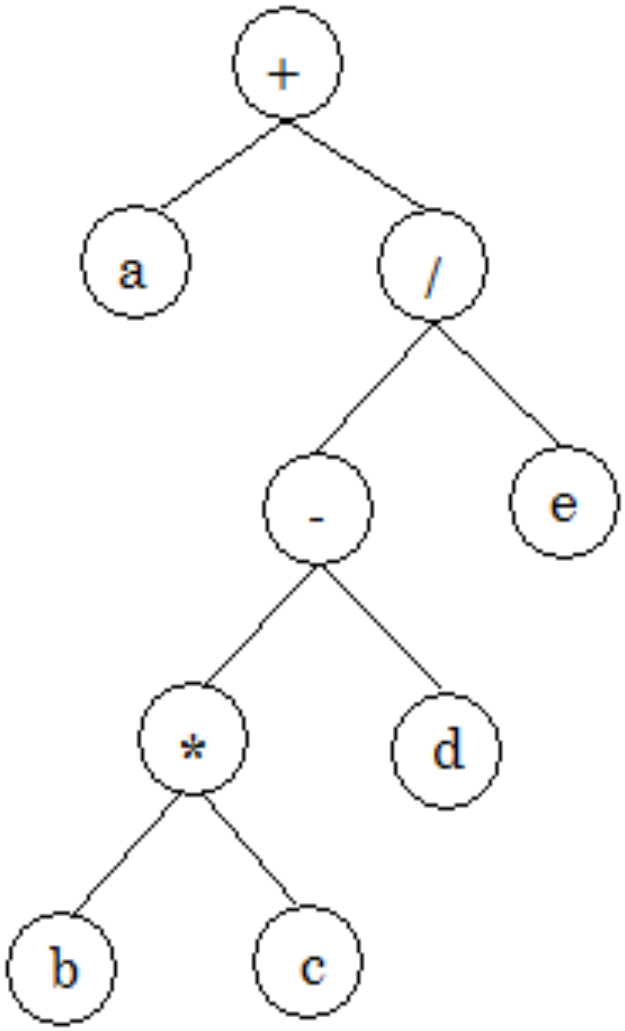
```
void preorder(tree t)
{ if (t!=NULL)
      { printf("%c",t->value);
        preorder(t->left);
        preorder (t->right);    }   }

void postorder(tree t)
{ if (t!=NULL)
      {  postorder(t->left);
         postorder (t->right);
         printf("%c",t->value);}   }
```

tree.c (5)

```
void inorder(tree t)
{ if (t!=NULL)
      { inorder(t->left);
        printf("%c",t->value);
        inorder (t->right);     }   }
```

Qualcuna delle tre è tail ricorsiva?

Esempio: a+(b*c – d)/e



- **Preorder:**

  +a/-*bcde

  (***polacca prefissa***, *operatore, Io operando, IIo operando)*

- **Postorder:**

  abc*d-e/+

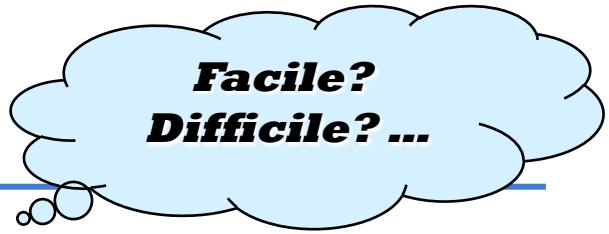
  (***polacca postfissa***, *Io operando, IIo operando, operatore)*

- **Inorder:**

  a+b*c-d/e

  (si perde priorità operazioni e parentesi)

To do:



Leggere un carattere e cercarlo nell'albero costruito.

Stampare il numero di nodi dell'albero costruito.

Ricerca: **member**

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

- E' tail ricorsiva?

Numero nodi: **nnodi**

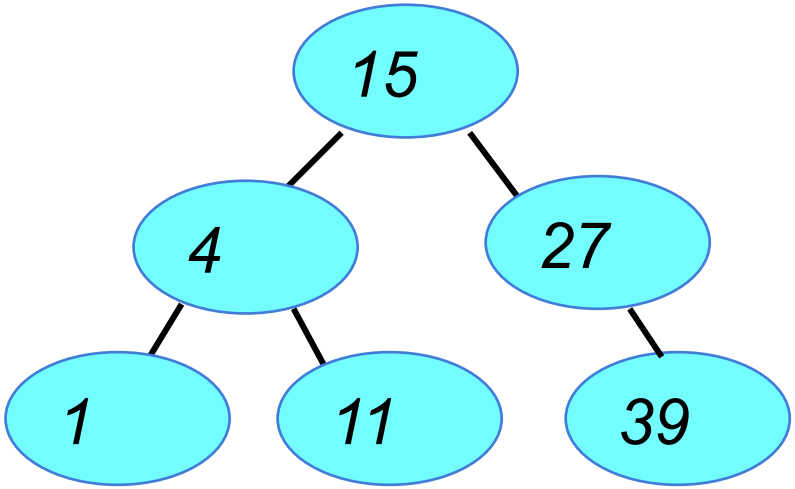
```
int nnodi(tree t)
{ if (t==NULL) return 0;
  else
  return(1+nnodi(t->left) + nnodi(t->right));
}
```

- E' tail ricorsiva?

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: ESEMPIO

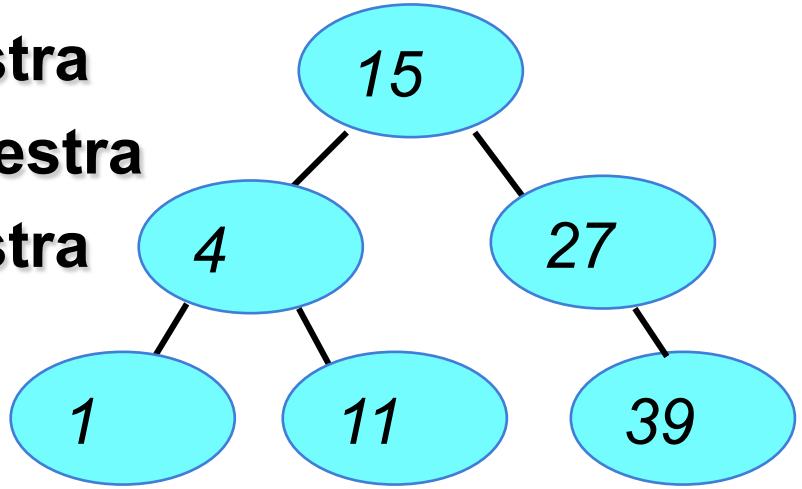


La visita inorder ci dà l'elenco ordinato: 1 4 11 15 27 39

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

ALBERI BINARI DI RICERCA: ESEMPIO

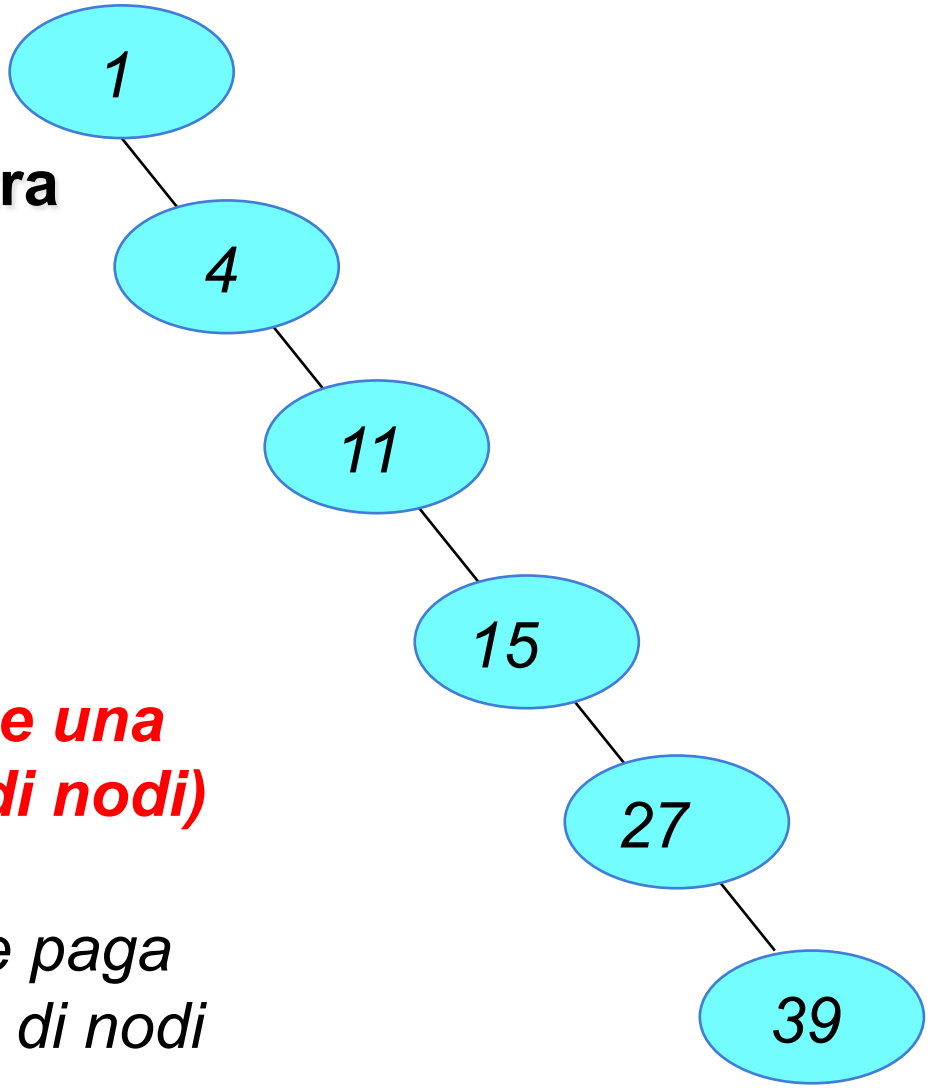
1) **Inserisci il 15 → radice**
2) **Inserisci 27 → va a destra**
3) **Inserisci 4 → va a sinistra**
4) **Inserisci 39 → in fondo a destra**
5) **Inserisci 11 → sinistra, poi destra**
6) **Inserisci 1 → in fondo a sinistra**



L'ordine con cui si inseriscono gli elementi è rilevante:
se essi appaiono 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

La ricerca nel caso peggiore paga un costo lineare nel numero di nodi

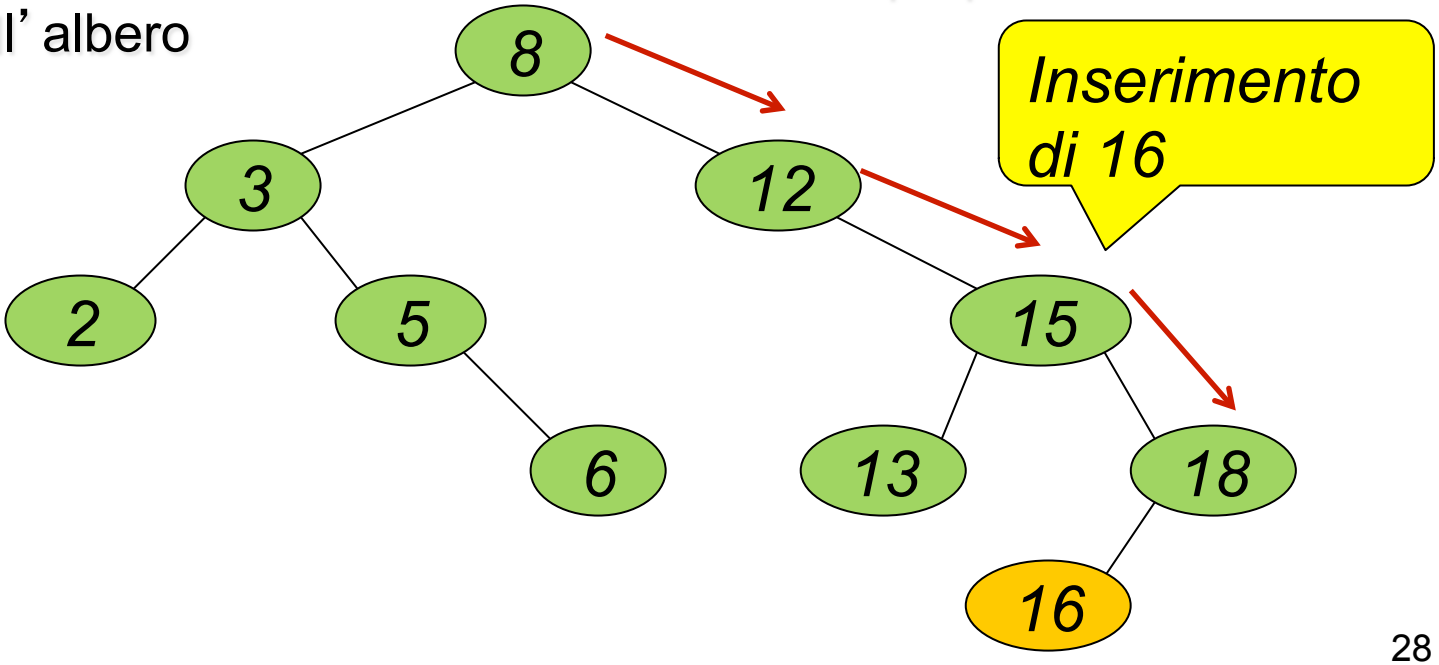
Realizzazione

- Fra gli elementi contenuti nei nodi dell'albero deve essere definita **una relazione d'ordine totale**

- In C, ADT degli elementi, con esportati i prototipi delle funzioni predicative **`isLess`** e **`isEqual`**

BST – Inserimento iterativo

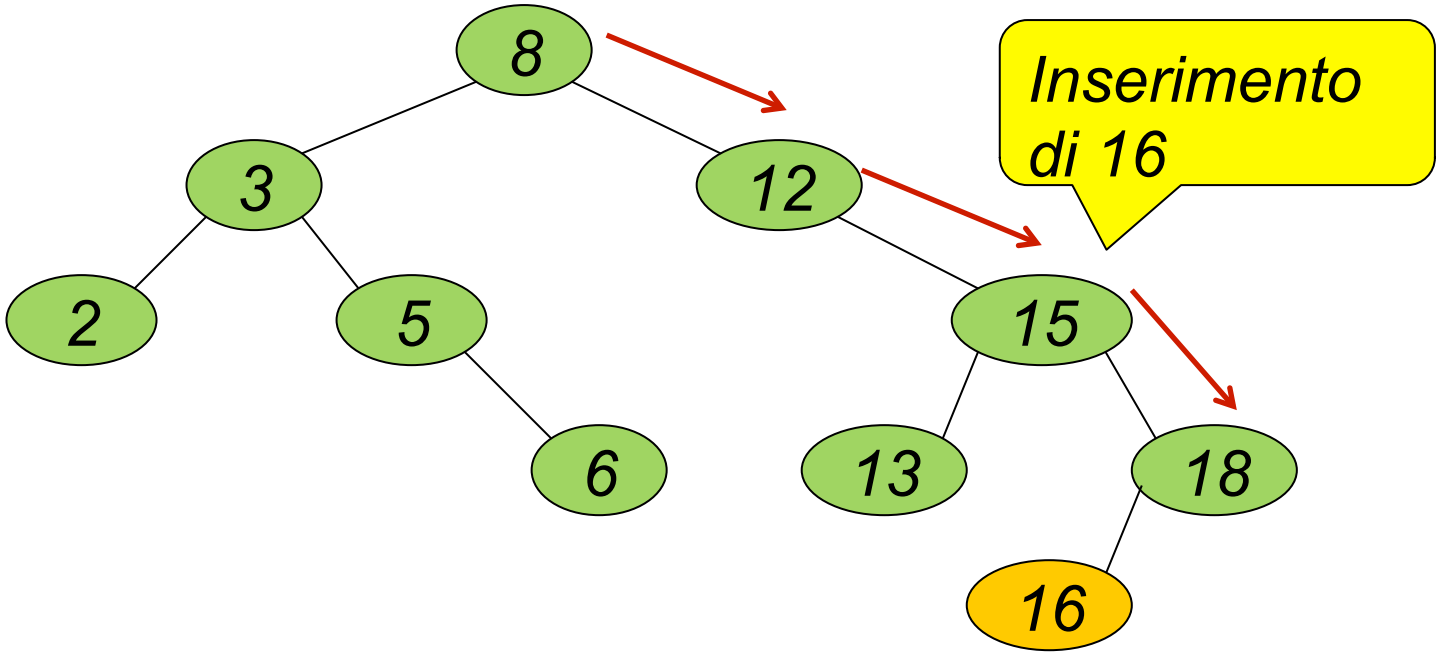
- L'algoritmo di inserimento iterativo:
  - Aggiorna iterativamente un puntatore, cercando, nell'albero, la posizione corretta di inserimento, ovvero **il nodo che diventerà il padre di quello da inserire**
  - Una volta trovato il nodo, appende il nuovo nodo come figlio sinistro/destro in modo da mantenere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

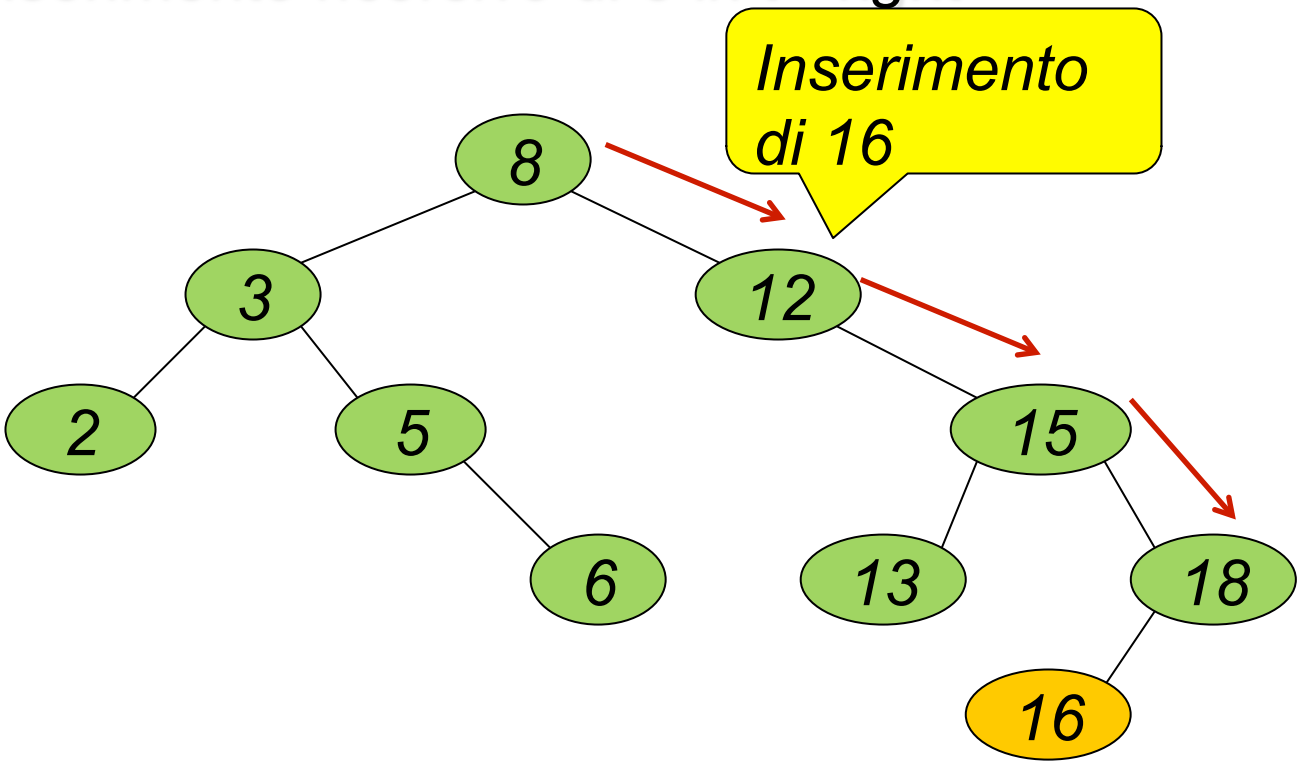


Inserimento iterativo

```
tree ordins_it(element e,tree root)
{tree p, 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;  }
```

Inserimento – ricorsivo

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



Inserimento - ricorsivo

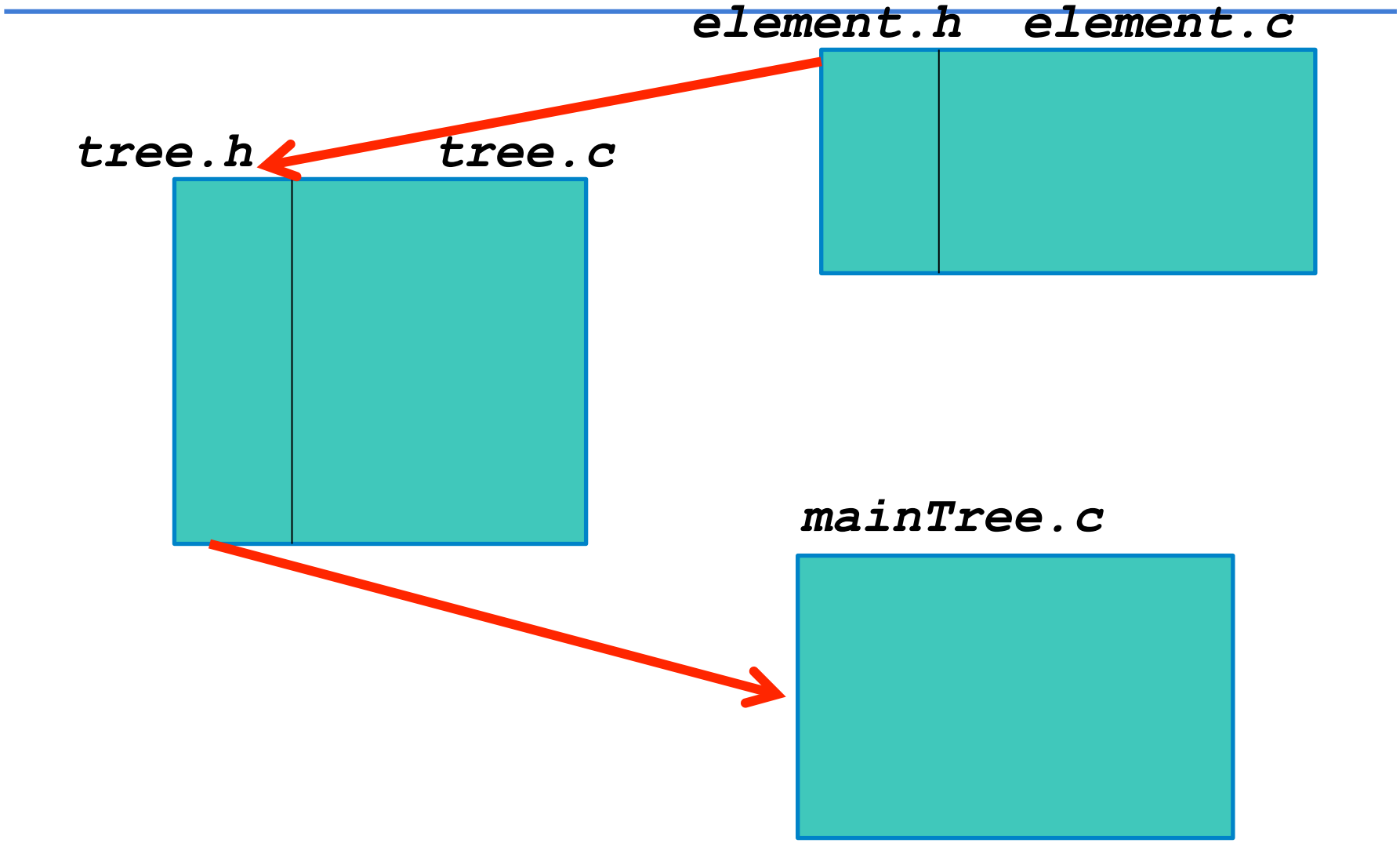
1. Sia *t* il puntatore (**`tree`**) che identifica il sotto-albero in cui inserire l'elemento *e*
2. Se *t* è nullo (albero vuoto), restituire un nuovo nodo contenente l'elemento da inserire (e sottoalberi nulli) e **terminare**
3. Se *e* è minore dell'elemento contenuto nella radice di *t*
    1. Assegnare 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)
{                          //BST con duplicazioni
  if (t==NULL)
      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;
   }
}
```

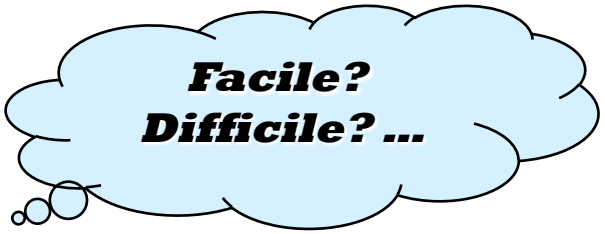
- E' tail ricorsiva?

To do:



To do:

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.**

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); }
```

ADT Tree: 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);
}
```

tree.c (5)

```
void inorder(tree t)
{ if (! empty(t))
      {  inorder(left(t));
         printElement( root(t) );
         inorder(right(t));     }
}
```

Qualcuna delle tre è tail ricorsiva?

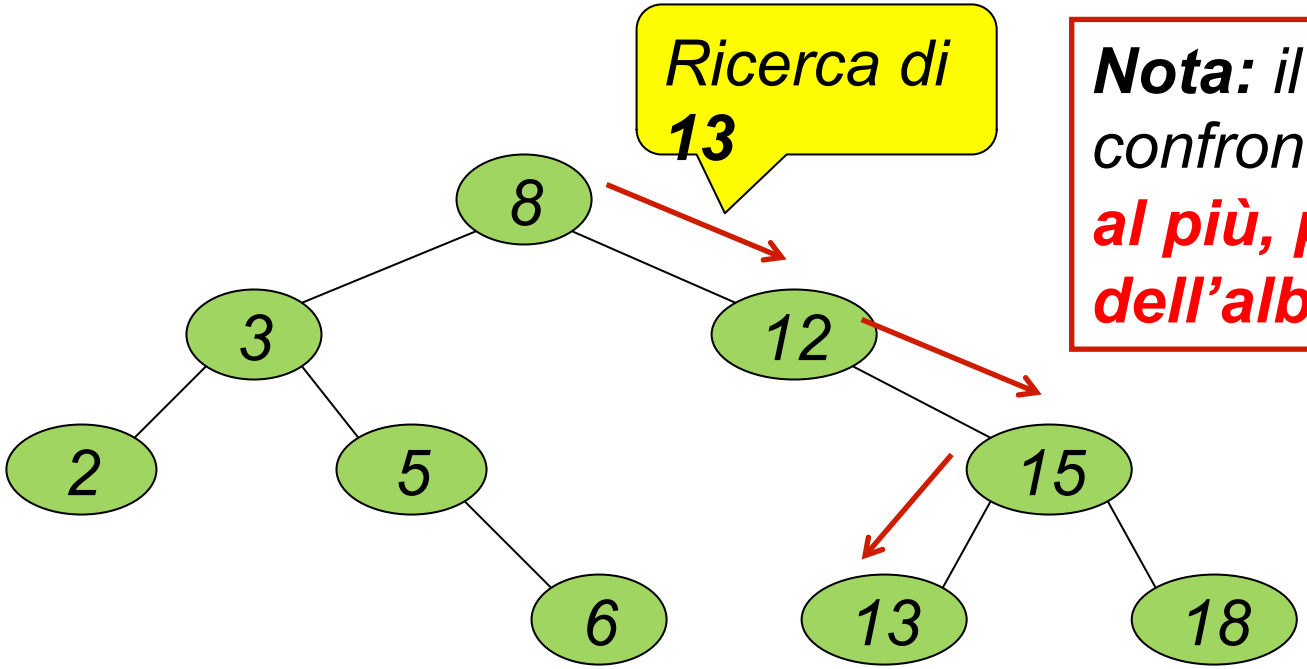
To do:

Dopo che il *main* ha costruito l'**albero binario di ricerca,** legga da input un valore intero e lo cerchi nell'albero binario di ricerca

Quale algoritmo di ricerca?

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***)



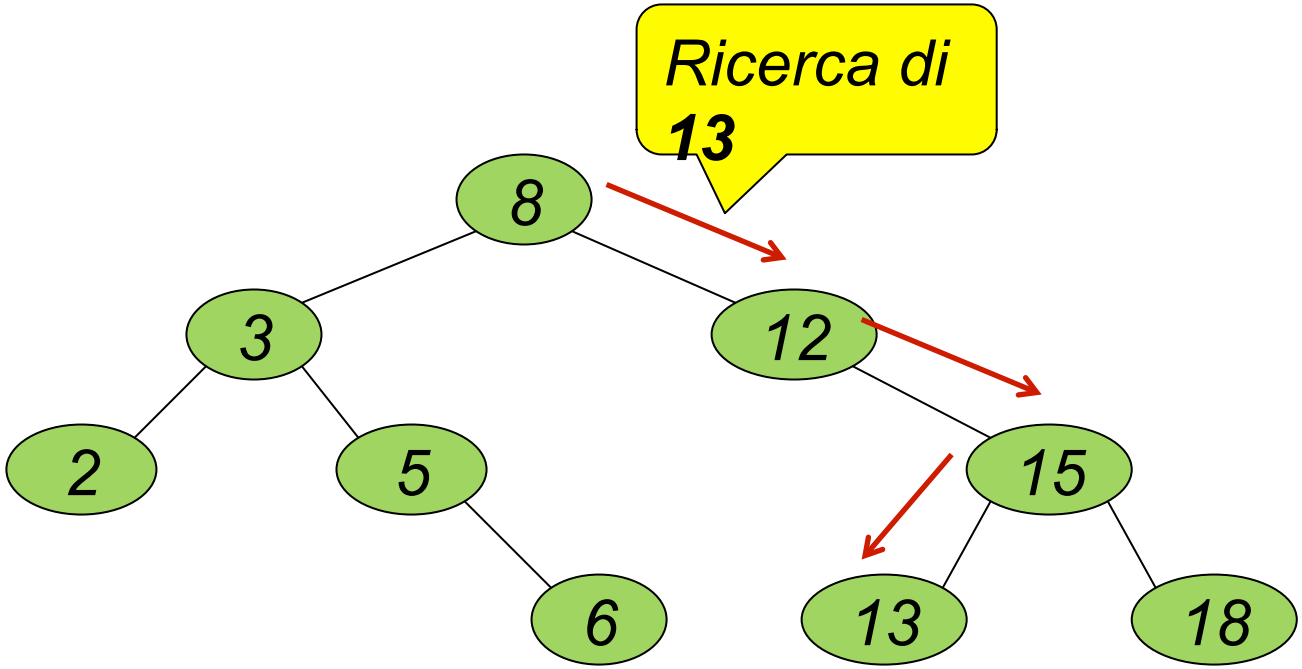
Nota: *il numero di passi/ confronti da compiere è,* ***al più, pari all'altezza dell'albero****!*

Ricerca binaria - 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



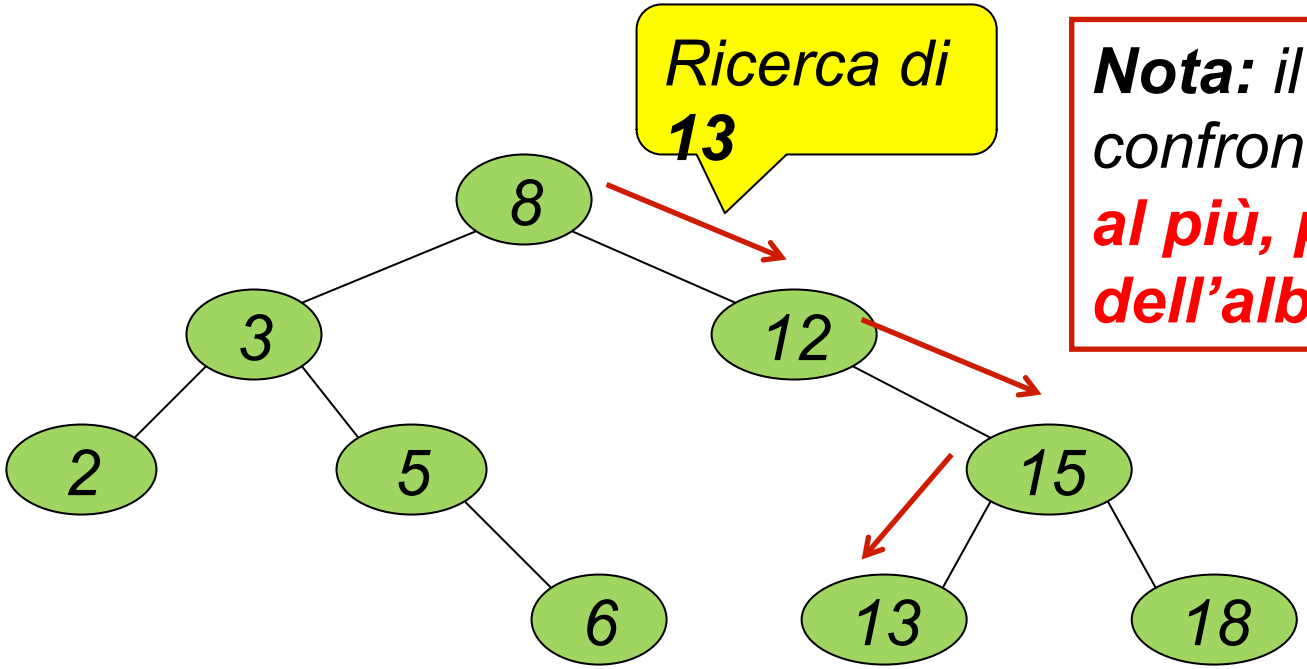
Ricerca (iterativa) in BST:

- Versione iterativa della ricerca binaria su BST

```
boolean member_ord_it(element e,tree t)
{ while(!empty(t))
    { if (isEqual(e,root(t))) return true;
      else
          if (isLess(e,root(t)))  t=left(t);
          else t=right(t);
    }
  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

tree.c (7)

```
boolean member_ord(element e,tree t)
{ if (empty(t)) return false;
  else
    if (isEqual(e,root(t))) return true;
    else
      if (isLess(e,root(t)))
            return member_ord(e,left(t));
      else return member_ord(e,right(t)) ;
}
```

- E' tail ricorsiva?

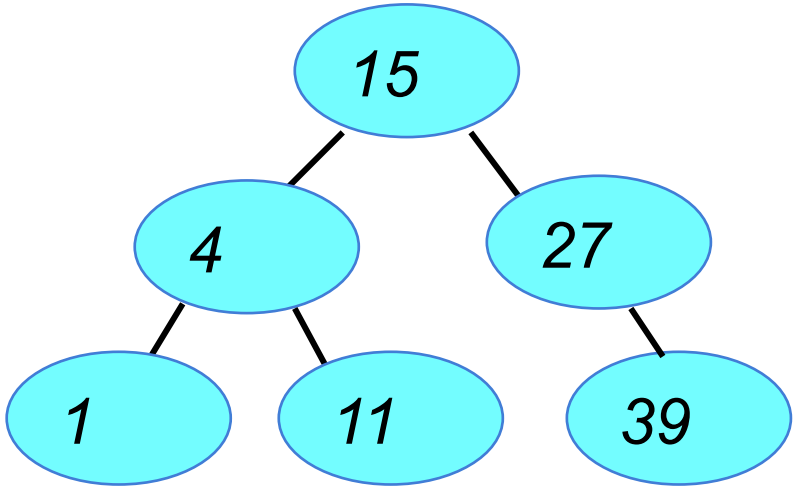
Complessità ricerca in BST

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 *maggiore* 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).

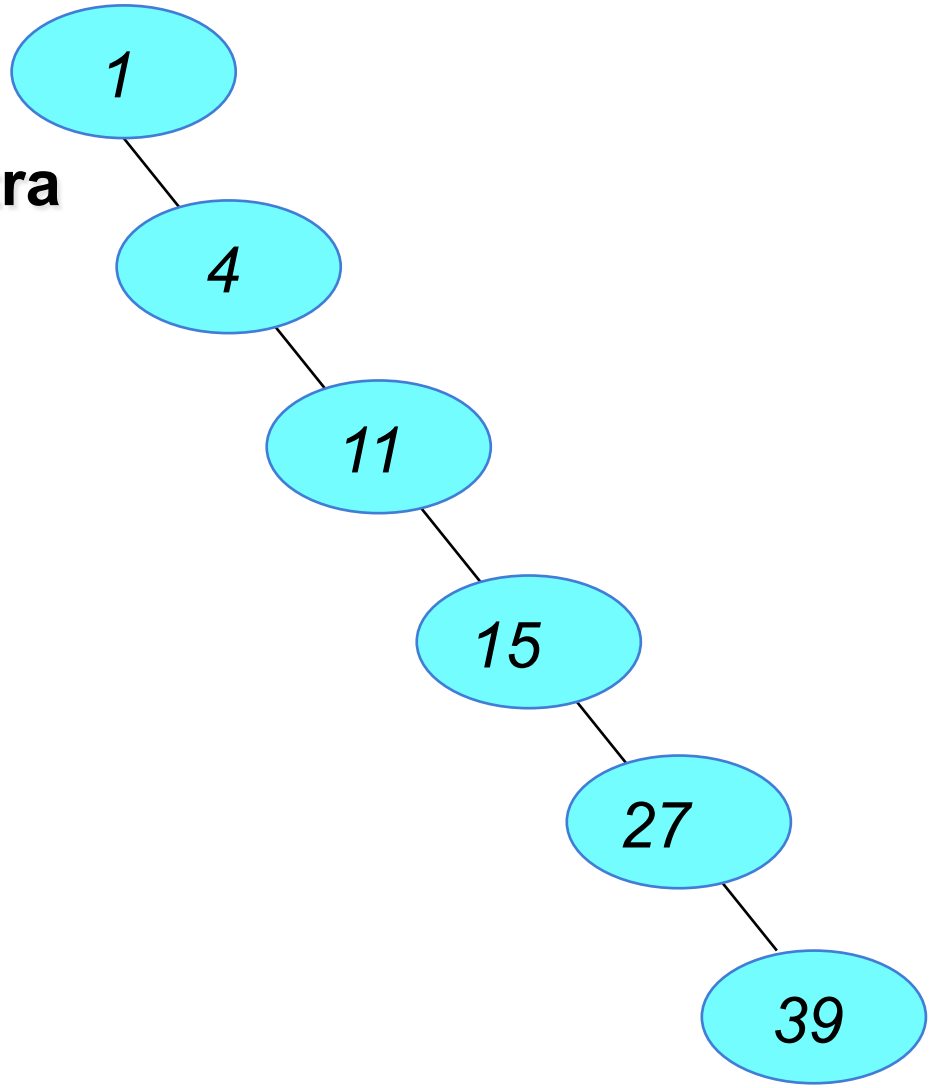
BST BILANCIATO

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



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 “”

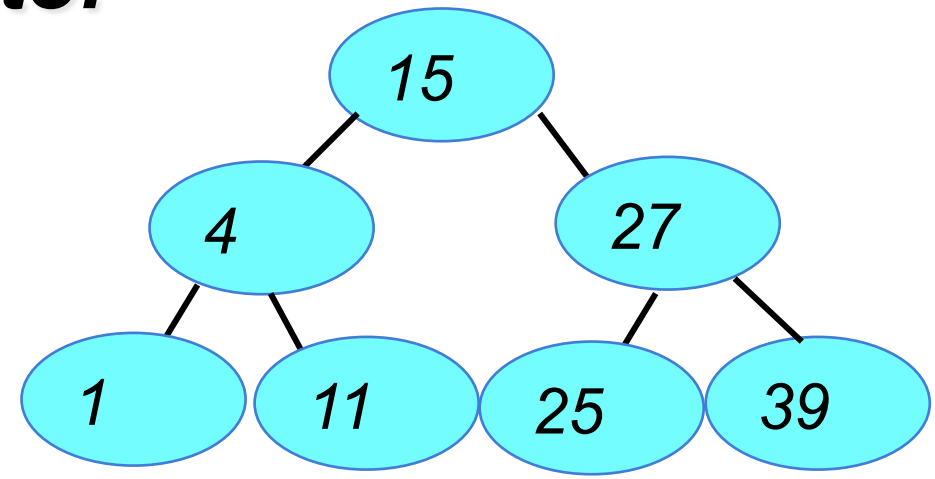


Correlazione tra altezza e numero nodi

- Albero completo, sia k l'altezza (o profondità), il numero di nodi è:

$$N=2^{k+1}-1$$

- Albero completo e ***perfettamente bilanciato:***



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, secondo 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

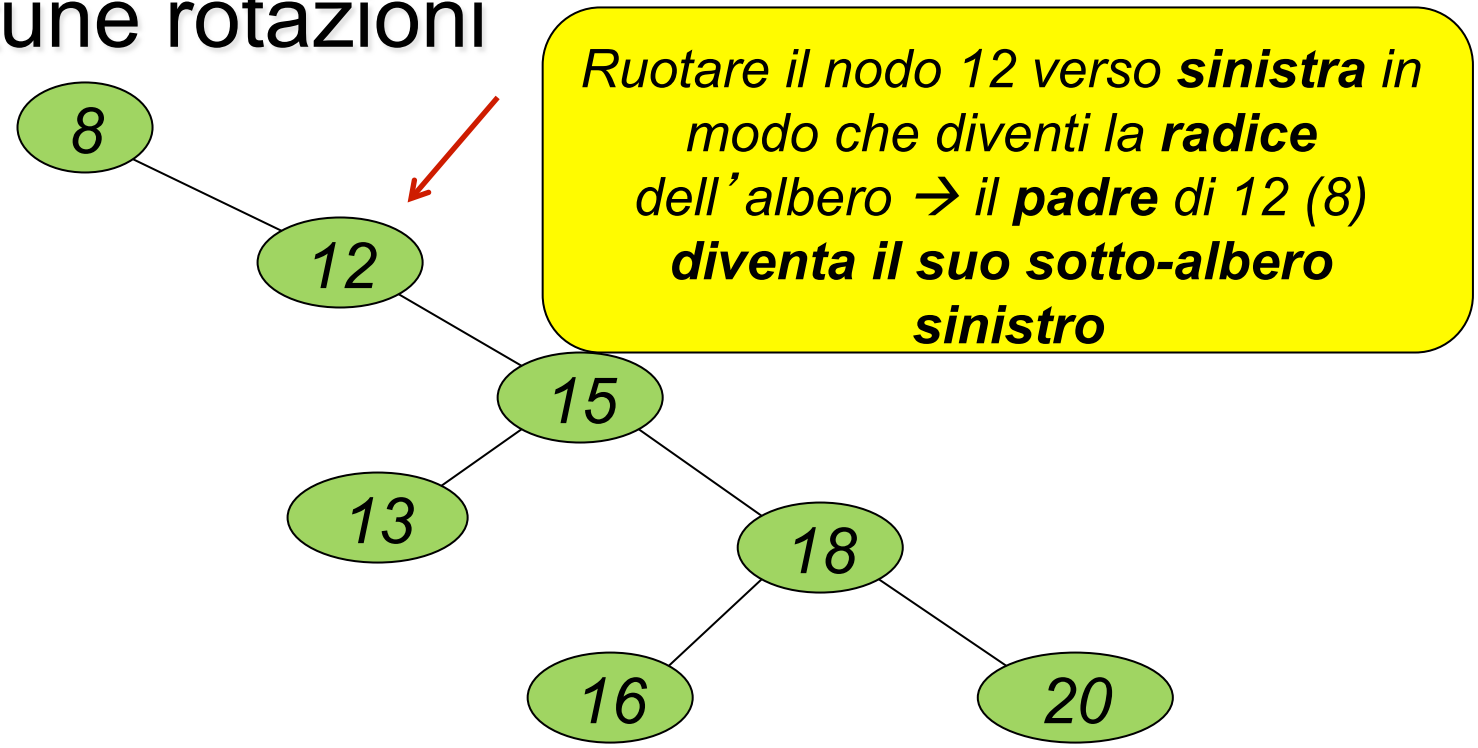
Alberi AVL

- 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)

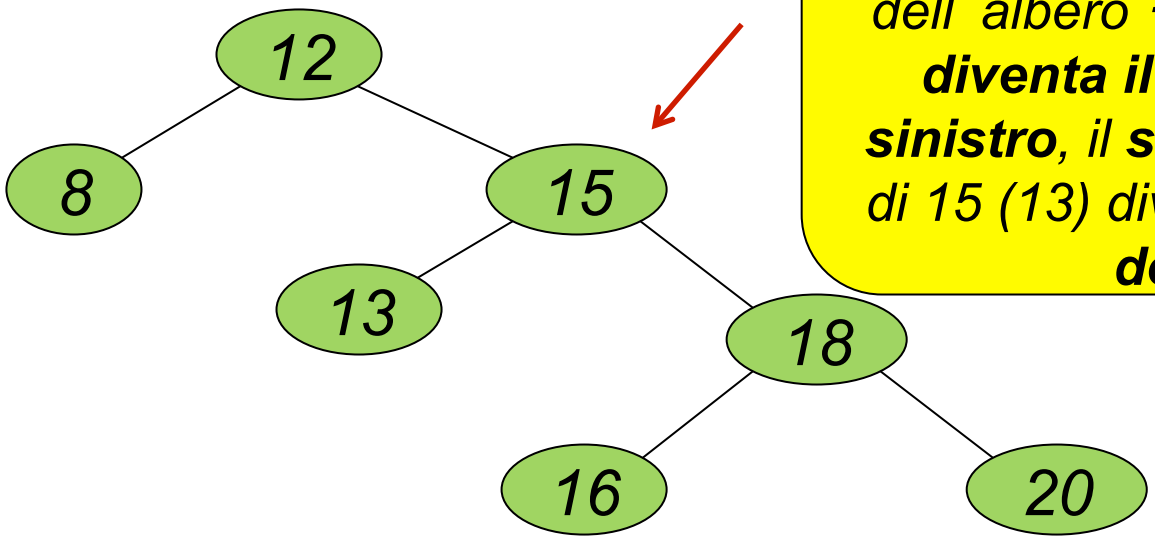
Rotazioni

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



Rotazioni

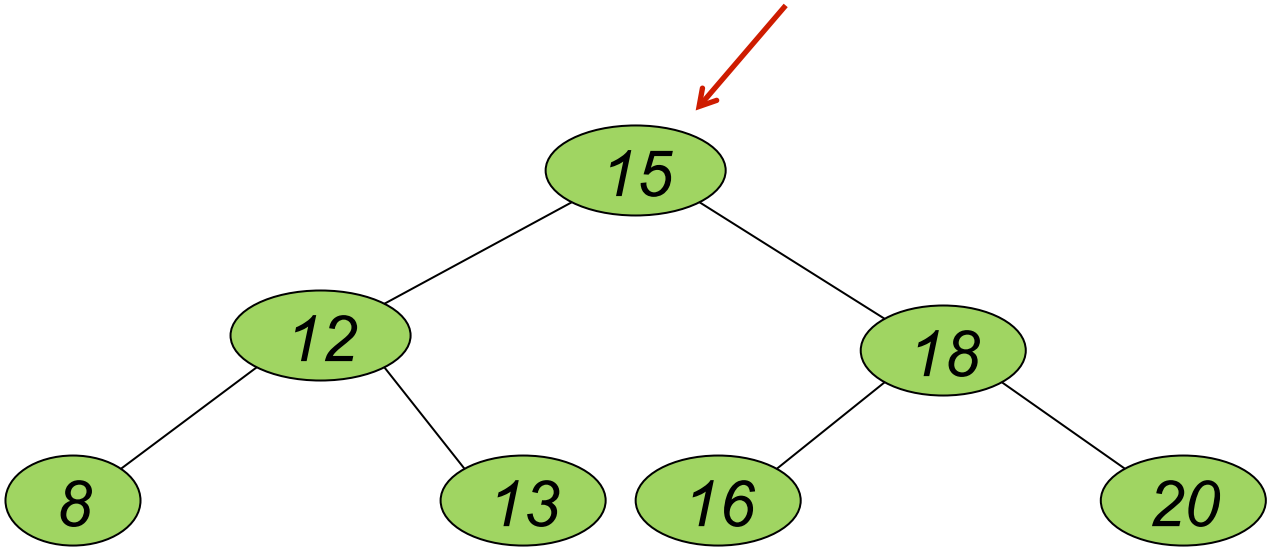
- Rotazione 1: il nodo 12 diventa la radice



*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



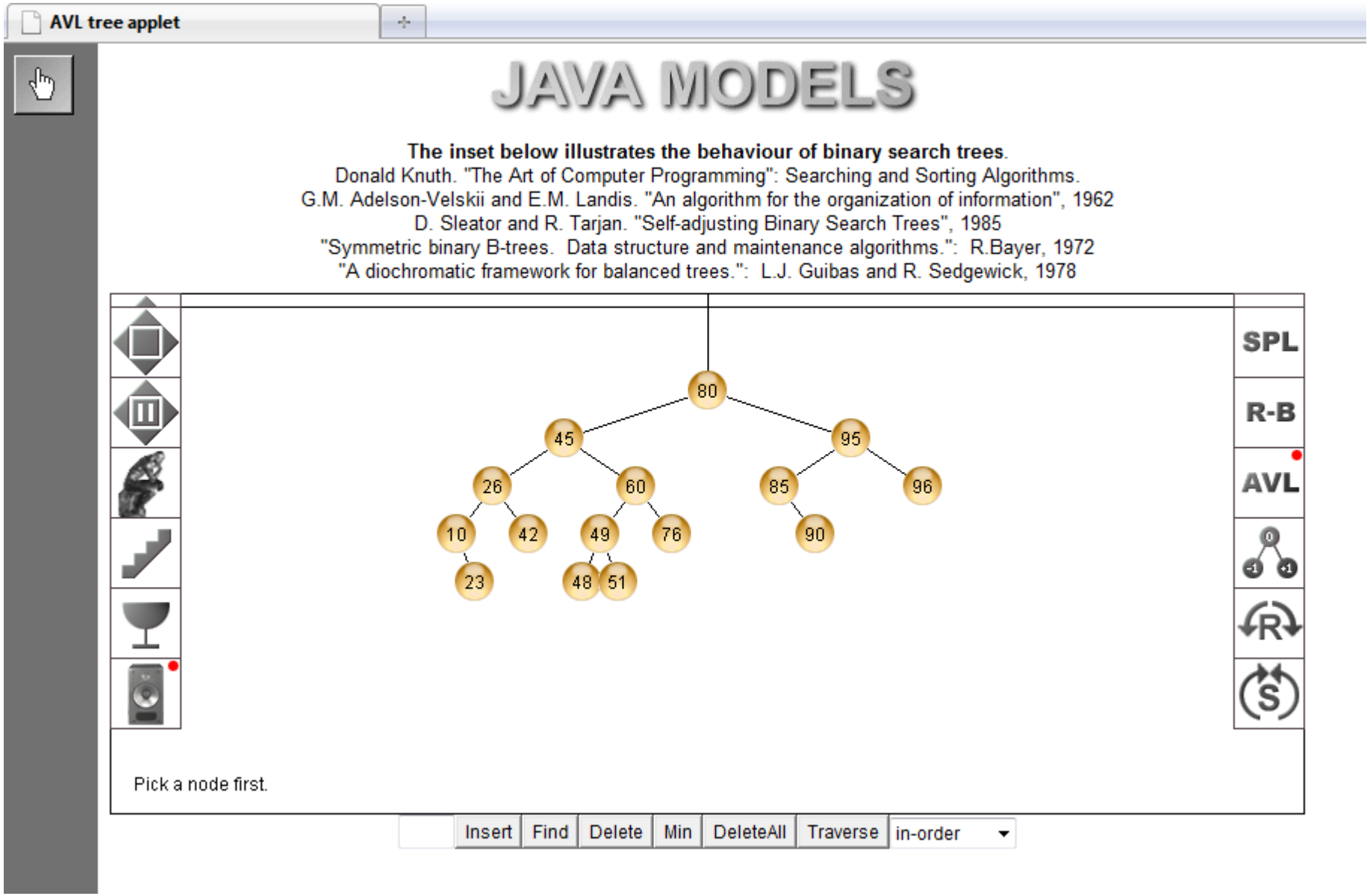
Per giocare un po' …

http://www.qmatica.com/DataStructures/Trees/AVL/AVLTree.html

- Applet java per giocare con alberi di ricerca di vario tipo, con possibilità di inserire, cancellare, ruotare e vedere come si comportano i vari tipi di alberi supportati

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

- Realizzazione modulare in C (e Java) delle operazioni su AVL tree (insert)


AVL tree applet
JAVA MODELS
The inset below illustrates the behaviour of binary search trees.
Donald Knuth. "The Art of Computer Programming": Searching and Sorting Algorithms.
G.M. Adelson-Velskii and E.M. Landis. "An algorithm for the organization of information", 1962
D. Sleator and R. Tarjan. "Self-adjusting Binary Search Trees", 1985
"Symmetric binary B-trees. Data structure and maintenance algorithms.": R.Bayer, 1972
"A diochromatic framework for balanced trees.": L.J. Guibas and R. Sedgewick, 1978
80
45
95
26
60
85
96
10
42
49
76
90
23
48
51
SPL
R-B
AVL
Pick a node first.
Insert
Find
Delete
Min
DeleteAll
Traverse
in-order

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)));
}
```