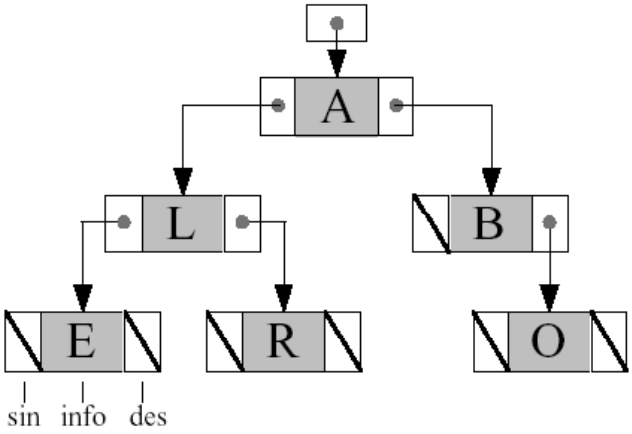
ADT ALBERO BINARIO (tree)

- Rappresentazione collegata (puntatori a strutture)



ADT ALBERO BINARIO

OPERAZIONI PRIMITIVE DA REALIZZARE

Operazione	Descrizione
cons_tree: D 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: -> list	Restituisce (costruisce) l'albero vuoto
empty: tree-> boolean	Restituisce vero se l'albero dato è vuoto, falso altrimenti

ADT ALBERO BINARIO

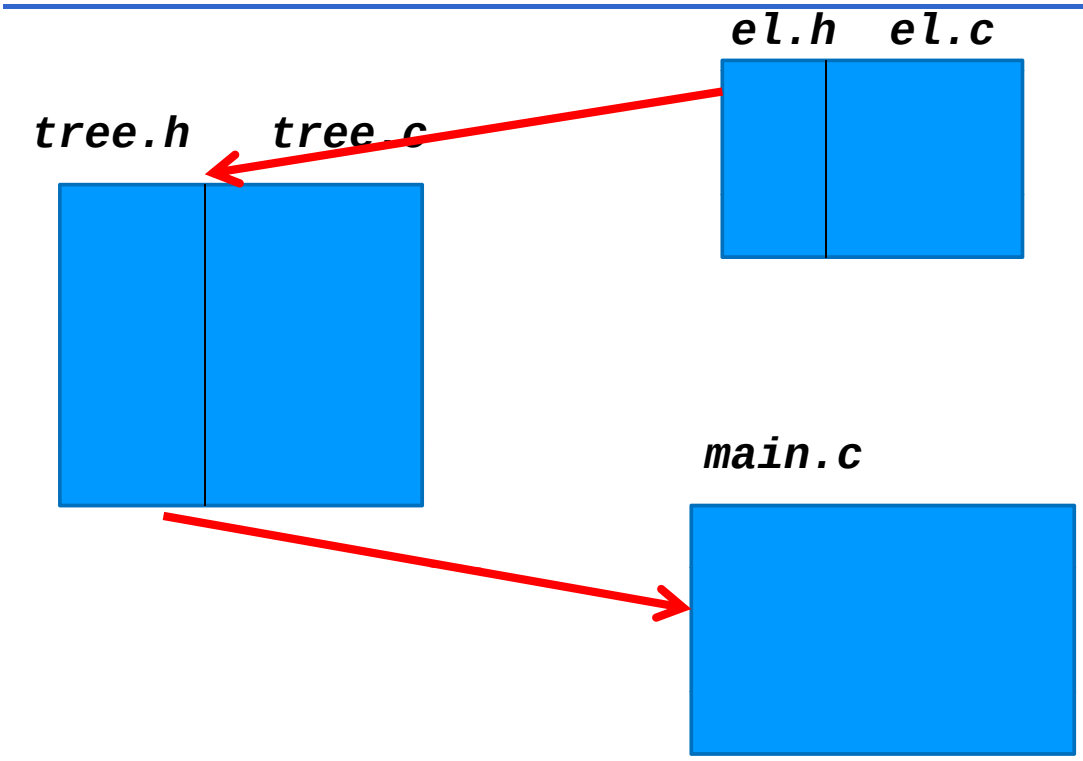
OPERAZIONI DERIVATE (etc etc)

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
height: tree -> int	Calcola l’altezza di un albero
nodi: tree -> int	Conta il numero dei nodi

Esercizio 10.3.0

- Un file binario ad accesso diretto (PERSONE.DAT) contiene un elenco di informazioni su persone (cognome, nome, data di nascita, città di residenza). Il cognome di ciascuna persona costituisce la ***chiave unica*** di tale elenco. Per effettuare ricerche su questo elenco, si costruisca un ***albero binario di ricerca*** mantenendo, per ciascun cognome il numero di record corrispondente. Si definisca un programma C che:
  a) costruisca la strutture dati in memoria principale a partire dal file PERSONE.DAT;
  b) calcoli l'altezza dell’albero così ottenuto e il numero dei suoi nodi;
  c) letto un cognome a terminale, verifichi se esiste un elemento nell’indice un elemento con quella chiave e trovatolo nell’albero, acceda al file e legga il record corrispondente, visualizzando i campi nome, datan e città;
  d) letto un cognome a terminale, stampi ordinatamente a video i cognomi minori o uguali di quello letto.

COMPONENTI



ADT ELEMENT: el.h

```
/* ELEMENT TYPE - file el.h*/
typedef struct
      {     char nome[15];
            char cognome[15];
            char datan[7];
            char citta[15]; } record_type;

typedef struct {  char cognome[15];
                  int pos;} el_type;

typedef enum {false,true} boolean;

/* operatori esportabili */
boolean isequal(el_type, el_type);
boolean isless(el_type, el_type);
void showel (el_type);
```

ADT ELEMENT: el.c

```
/* ELEMENT TYPE - file el.c*/
#include <stdio.h>
#include <string.h>
#include "el.h"

boolean isequal(el_type e1, el_type e2)
/* due elementi uguali se stesso cognome */
{  if (strcmp(e1.cognome,e2.cognome)==0)
       return true;
    else return false; }

boolean isless(el_type e1, el_type e2)
{   if (strcmp(e1.cognome,e2.cognome) <0)
        return true;
    else  return false; }

void showel (el_type e)
{   printf("%s\t%d\n",e.cognome,e.pos); }
```

ADT TREE: tree.h

```
/* TREE INTERFACE  - file trees.h*/
#include "el.h“
typedef struct nodo { el_type value;
                      struct nodo *left, *right; } NODO;
typedef NODO * tree;
/* operatori esportabili */
boolean empty(tree t);
tree  emptytree(void);
el_type root(tree t);
tree  left(tree t);
tree  right(tree t);
tree  cons_tree(el_type e, tree l, tree r);
tree ord_ins(el_type e, tree t);
void inorder(tree t);
int height (tree t);
int nodi (tree t);
int member_ord(el_type e, tree t); /*NOTA TIPO RESTITUITO */
void stampa_minori(tree t, el_type);
int max(int, int);
```

TO DO

Si modifichino i file *main.c e tree.c già disponibili*, implementando le operazioni richieste in funzione di quelle esportate dall'ADT degli elementi

Il *main* da realizzare deve: a) leggere il contenuto del file ricordando il numero del blocco letto e inserendo chiave, posizione la sequenza e inserire ogni elemento letto in un *albero binario di ricerca,* b) calcolarne numero di nodi e altezza; c) letto un cognome e accedendo all'albero recuperare le informazioni complete di quel cognome dal file; d) stampare i cognomi minori di quello letto.

main.c *(1 - stub)*

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "trees.h"
main() {
    tree indice=emptytree();
    int pos;
    FILE *f;
    record_type REC;
    el_type EL;
    char COGNOME[15];

    f = fopen("PERSONE.DAT", "rb");
    if(f==NULL){ printf("errore apertura file\n");
                 exit(-1);      }
    pos=0; /* CICLO DI SCANSIONE DEL FILE */
    while(fread(&REC,sizeof(record_type),1,f)>0){
        // da implementare
    }
```

main.c *(2 - stub)*

```
    fclose(f);
    printf("Chiusura del file\n");

    printf("Stampa dell'albero\n");
    inorder(indice);


    /* CALCOLO ALTEZZA E NUM.NODI DELL'ALBERO */

    printf("Altezza dell'albero: %d\n", height(indice));
    printf("Num. Nodi dell'albero: %d\n", nodi(indice));
```

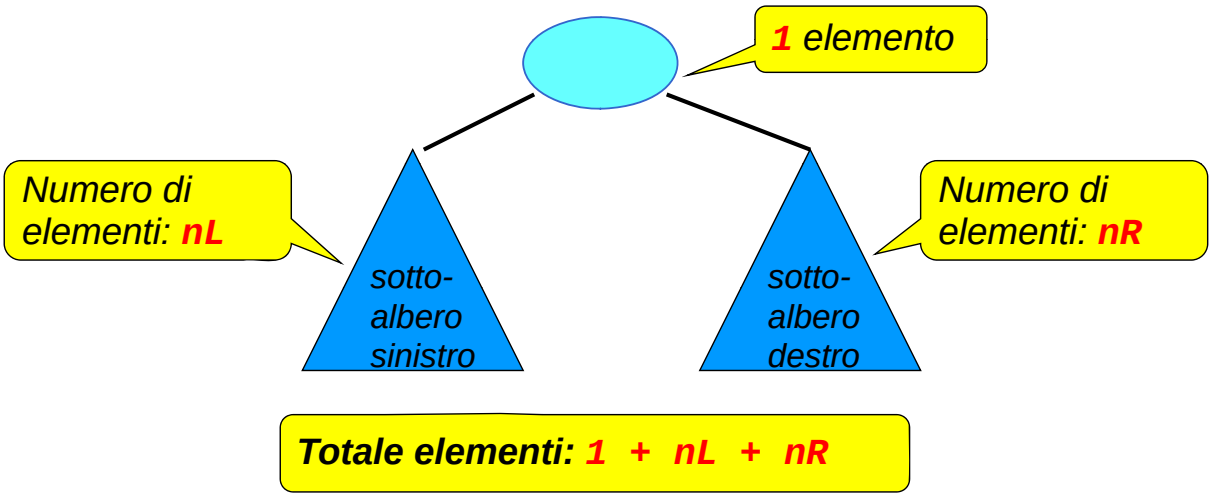
main.c *(3 - stub)*

```
   /* VISUALIZZAZIONE RECORD */
    printf("Inserisci un cognome: ");
    scanf("%s", COGNOME);
    f = fopen("PERSONE.DAT", "rb");

    if(f==NULL){
        printf("errore apertura file\n");
        exit(-1);
    }

    // invocare opportunamente member_ord,
    // trovare il record sul file con fseek
    // e stampare risultato

    // invocare opportunamente stampa_minori
}
```

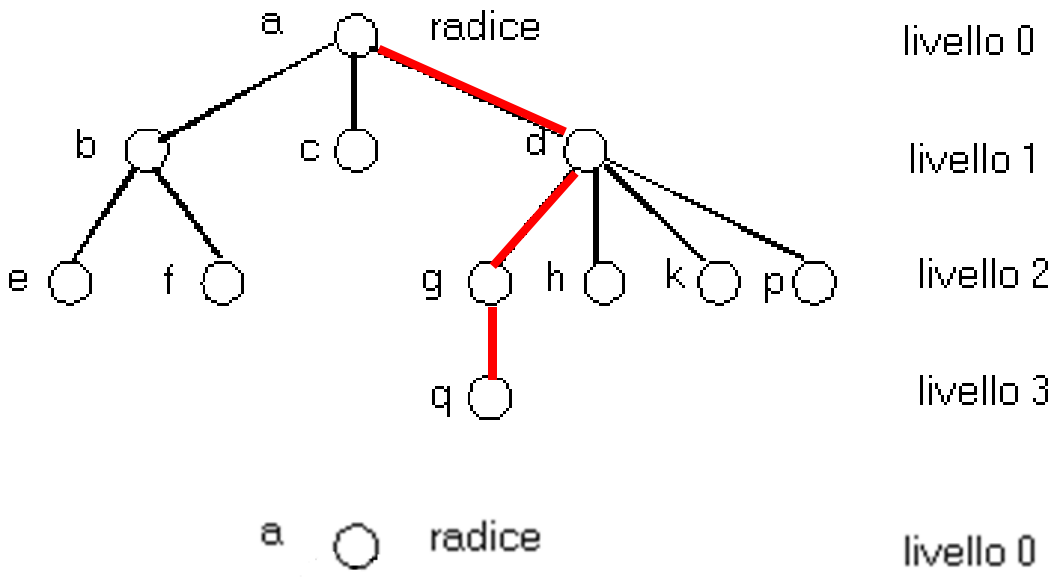
CONTARE GLI ELEMENTI

Algoritmo (per un albero binario)

- se l'albero è vuoto, gli elementi sono 0
- altrimenti, gli elementi sono 1 (la radice) + quelli del figlio sinistro + quelli del figlio destro



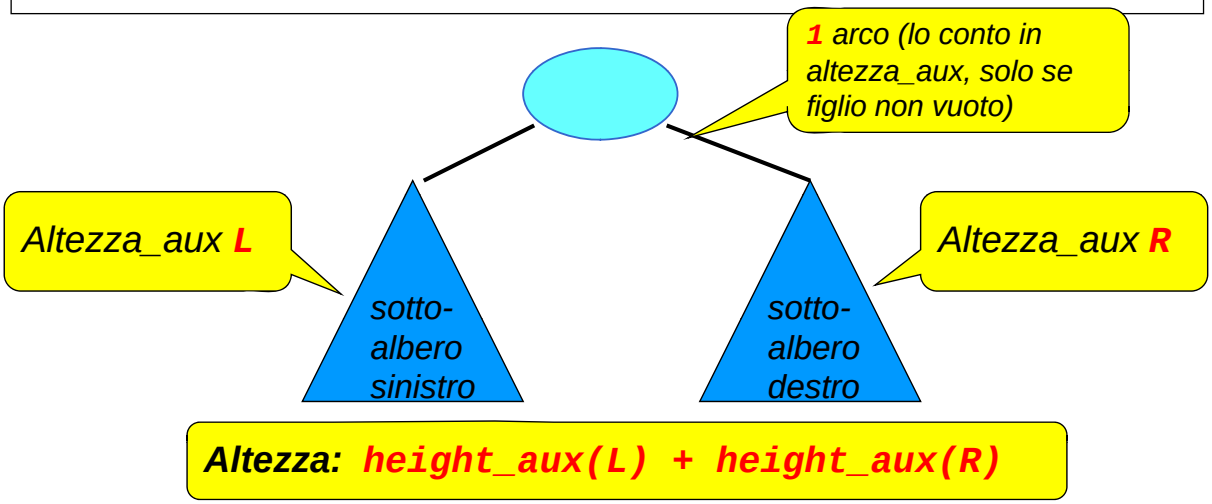
ALTEZZA DI UN ALBERO



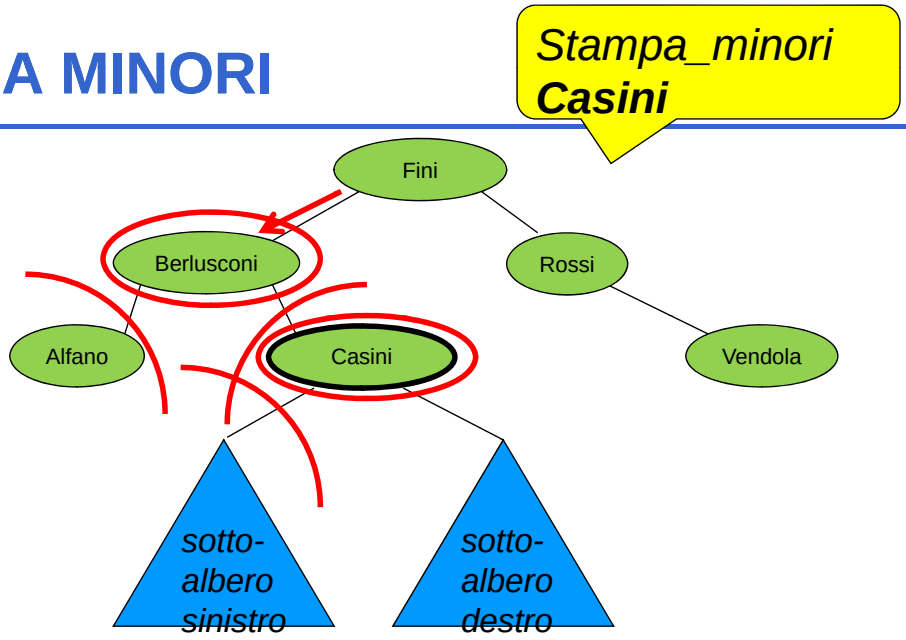
ALTEZZA DI UN ALBERO

Algoritmo (per un albero binario)

- se l'albero è vuoto, l'altezza è nulla (0)
- altrimenti, è il massimo tra l'altezza_aux del figlio sinistro e l'altezza_aux del figlio destro



STAMPA MINORI



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 (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); }
```

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 (4)

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

tree.c (5 – ord_ins)

```
tree ord_ins(el_type e, tree t)
/* albero binario di ricerca senza duplicazioni */
{ if (empty(t))    /* inserimento */
      return cons_tree(e,emptytree(),emptytree());
  else
    { if (isless(e,root(t)))
            t->left = ord_ins(e,left(t));
      else  t->right = ord_ins(e,right(t));
      return t;
    }
}
```


tree.c (6 – member_ord)

```
int  member_ord(el_type e, tree t)
{ while (t!=NULL){
        if (isless(e,root(t)))
            t=t->left;
        else
            if (isequal(e,root(t)))
                return (t->value).pos;
            else
                t=t->right;
    }
    return -1;
}
```

tree.c (7 - nodi)

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

- E' tail ricorsiva?

tree.c (8 - height)

```
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)) ));
}
```

- E' tail ricorsiva?

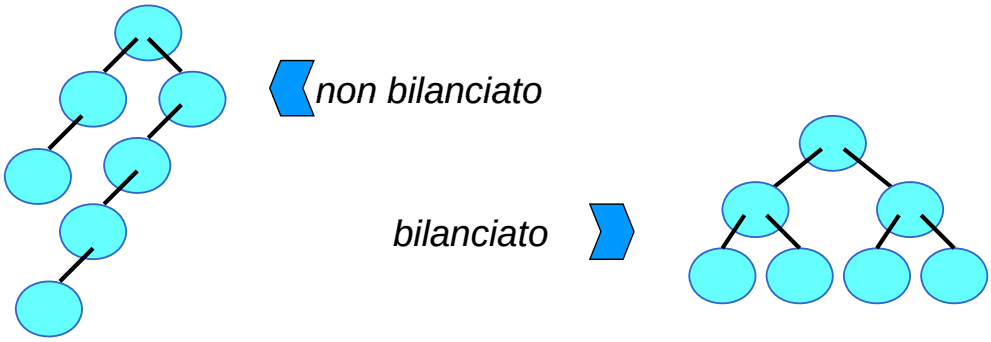
tree.c (9 – stampa_minori)

```
void stampa_minori(tree t, el_type EL2)
{   el_type EL;
    if (!empty(t)){ EL=root(t);
       if(isequal(EL, EL2))  inorder(left(t)) ;
        else
            if (isless(EL,EL2)){
                inorder(left(t));
                showel(EL);
                stampa_minori(right(t),EL2);
            }
            else{
                stampa_minori(left(t),EL2);
            }
    }
}
```


main.c

UN PO' DI CONSIDERAZIONI …

- **Quanto è bilanciato l'albero binario di ricerca ottenuto?**
- **Quanti nodi? Quale altezza?**
- ***Ci sono molti algoritmi per bilanciare alberi non bilanciati***



Alberi bilanciati

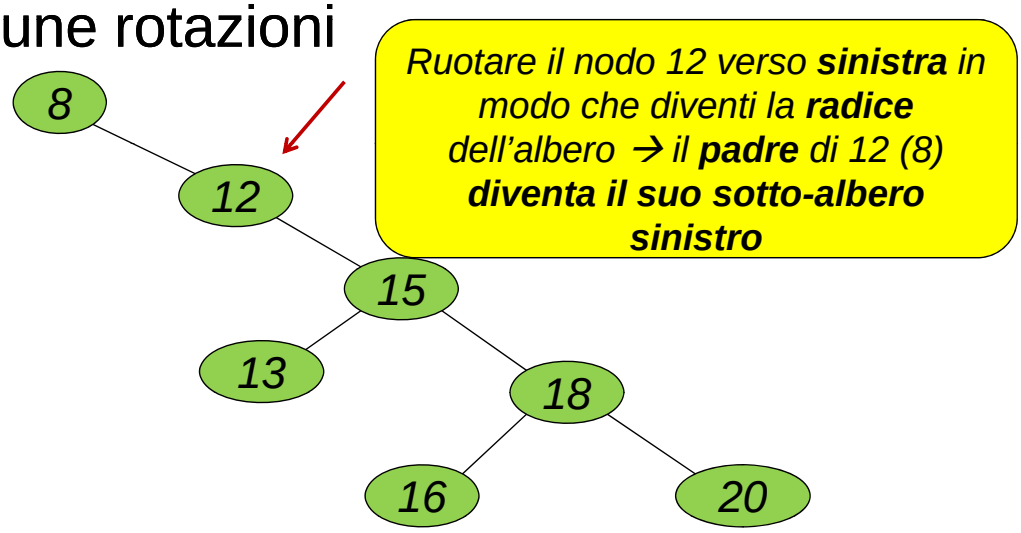
- Il problema di BST è che normalmente NON sono bilanciati:
  - Inserimenti e cancellazioni sbilanciano l'albero
  - Se l'albero non è correttamente bilanciato le operazioni (tutte) costano "parecchio"
- Soluzione: alberi che si autobilanciano
  - AVL (Adel'son-Vel'skii-Landis)
  - Red-Black
  - …

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 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

Rotazioni

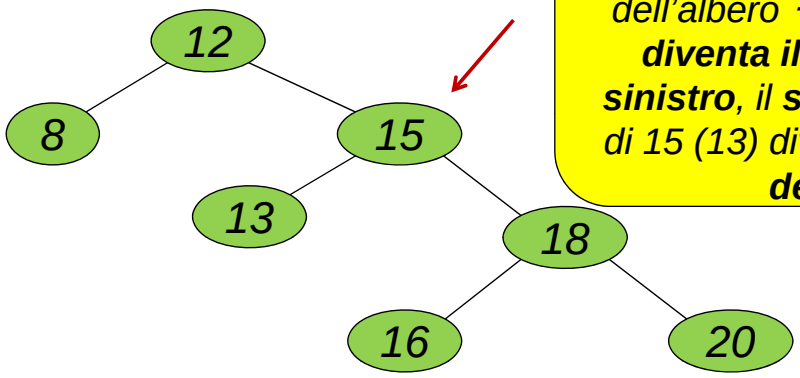
- Si supponga di disporre di un albero sbilanciato – è possibile bilanciarlo tramite opportune rotazioni



Rotazioni

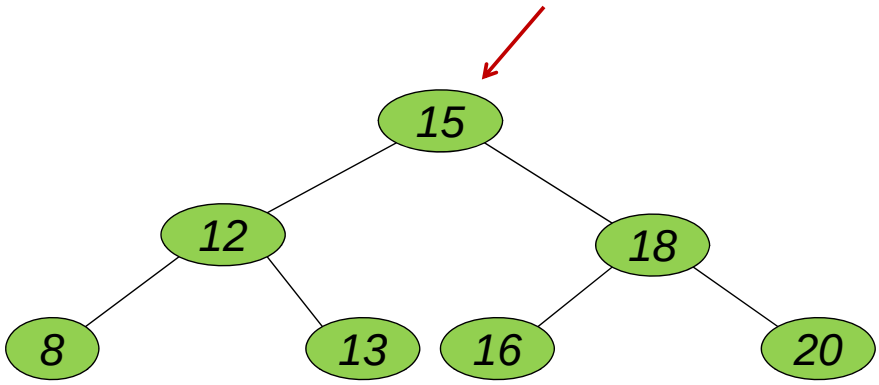
- Rotazione 1: il nodo 12 diventa la radice

*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



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

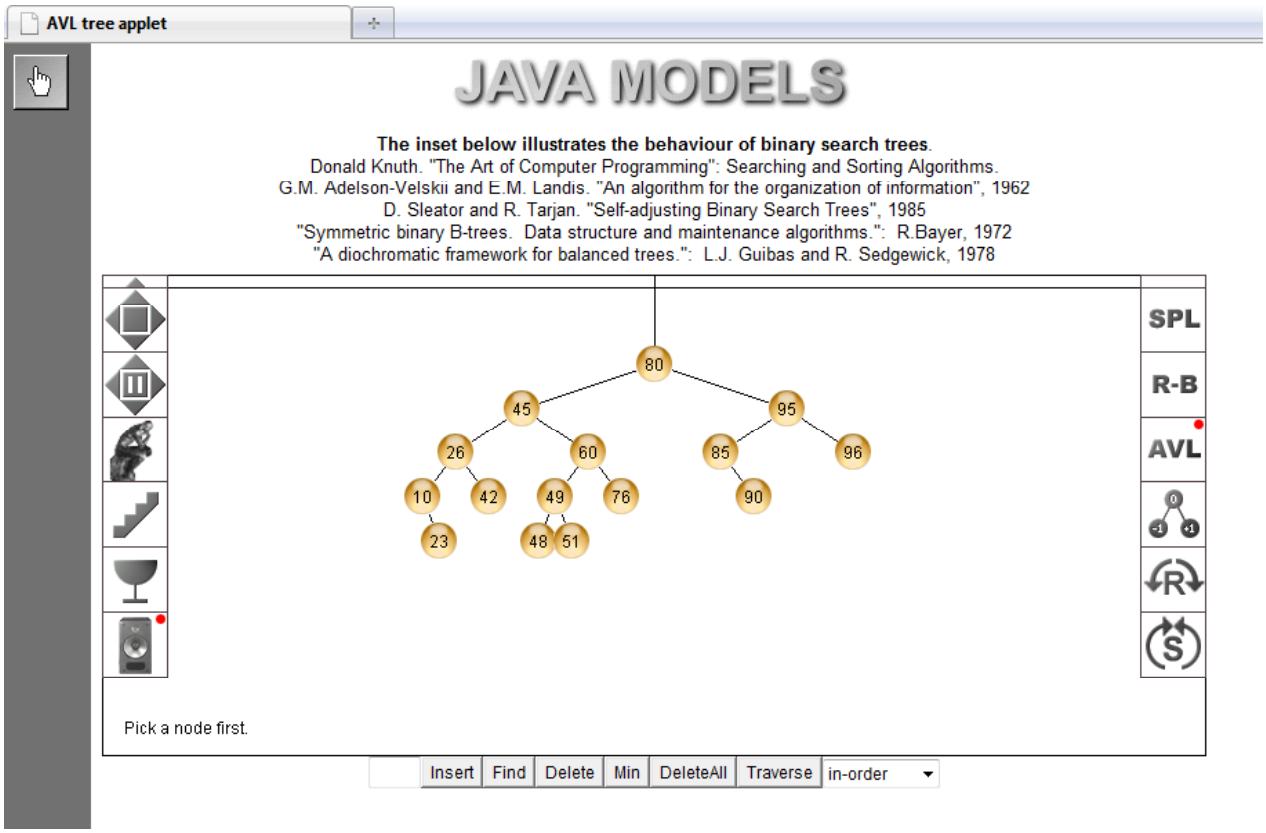
- Naturalmente i costi di inserimento/cancellazione crescono di molt,o ma la ricerca rimane sempre molto efficiente! (O($\log_2$ N) se N nodi)

Per giocare un po'…

http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm

- Applet java (codice disponibile) 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://cprogramminglanguage.net/avl-tree.aspx**
- Realizzazione modulare in C delle operazioni su AVL tree (insert, delete, etc)


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