Strutture dato ad albero

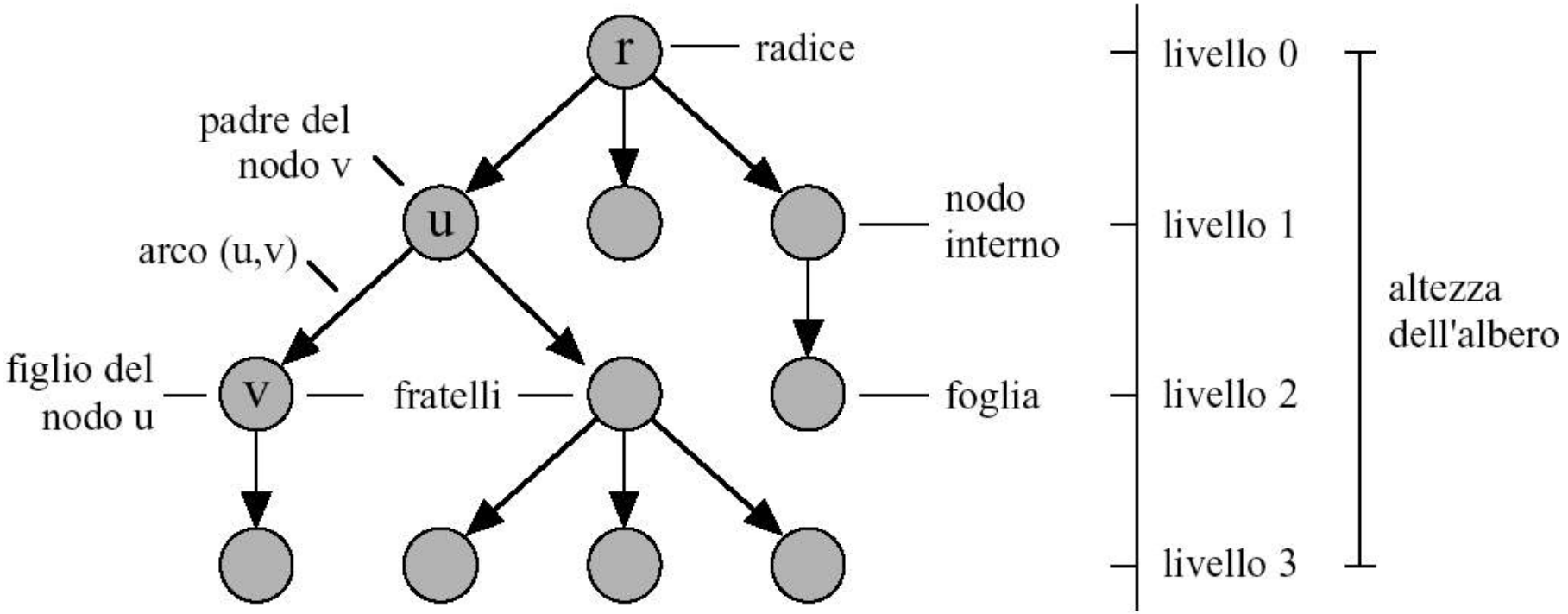
- Obiettivi:
  - Introdurre gli alberi binari (e la loro rappresentazione collegata)
  - Mostrare le procedure inserimento in testa, visita e ricerca in essi
  - Introdurre la nozione di ADT albero binario

Oltre le liste

- Le liste risolvono le carenze degli array (dimensione statica), ma sono strutture dati *sequenziali* e quindi
  - le operazioni su liste (ricerca) implicano sempre un *accesso sequenziale*
- Esempio: in una lista di N elementi, la *ricerca di un elemento* si effettua con l'algoritmno di ricerca sequanziale, che ha complessità pari a O(N) (nel caso peggiore)
  - Costo impensabile per *grandi moli di dati*
- La gestione di *grandi quantità* di dati può avvantaggiarsi dall'adozione di altre strutture dati *collegate*
  - possono rendere *nettamente più efficienti* operazioni come la *ricerca* (si può applicare – in certi casi – la ricerca binaria)

Alberi n-ari

- Gli alberi sono il caso più usato e rilevante di struttura dati non sequenziale



- Dati contenuti nei nodi, relazioni gerarchiche definite dagli archi che li collegano

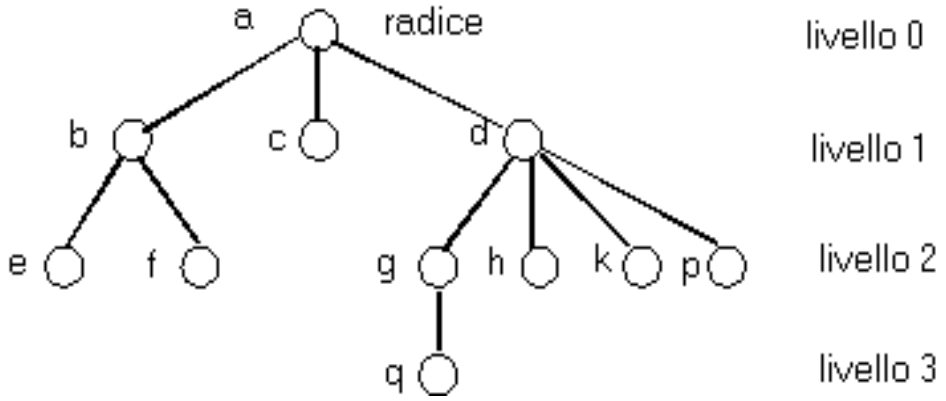
Alberi n-ari

Un ALBERO è un *grafo orientato aciclico* tale che

- esiste un nodo (*radice*) con grado d'ingresso 0
- ogni altro nodo ha grado d'ingresso 1.

DEFINIZIONI

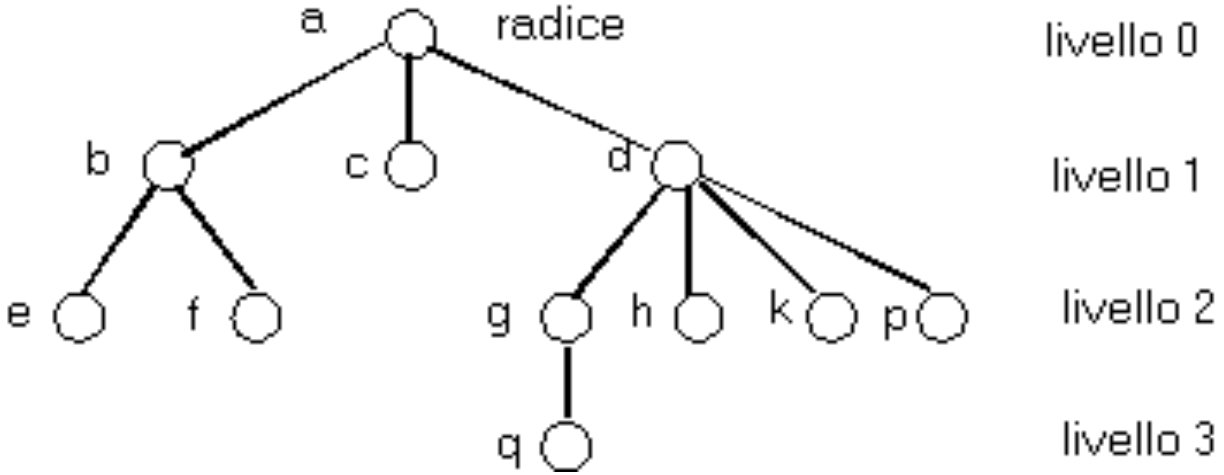
- *I nodi con grado di uscita 0 si dicono foglie.*
- *La lunghezza del cammino dalla radice a un dato nodo si dice livello di quel nodo.*
- *La lunghezza del cammino più lungo dalla radice a una foglia si dice altezza dell'albero.*
- *Se un arco collega il nodo α al nodo β, il nodo α si dice nodo padre di β, il quale è detto nodo figlio (o discendente diretto) di α.*



*Poiché in un albero il grado d'ingresso di ogni nodo è noto a priori, in luogo di **"grado di uscita"** si dice spesso semplicemente **"grado"**.*

Conseguenze:

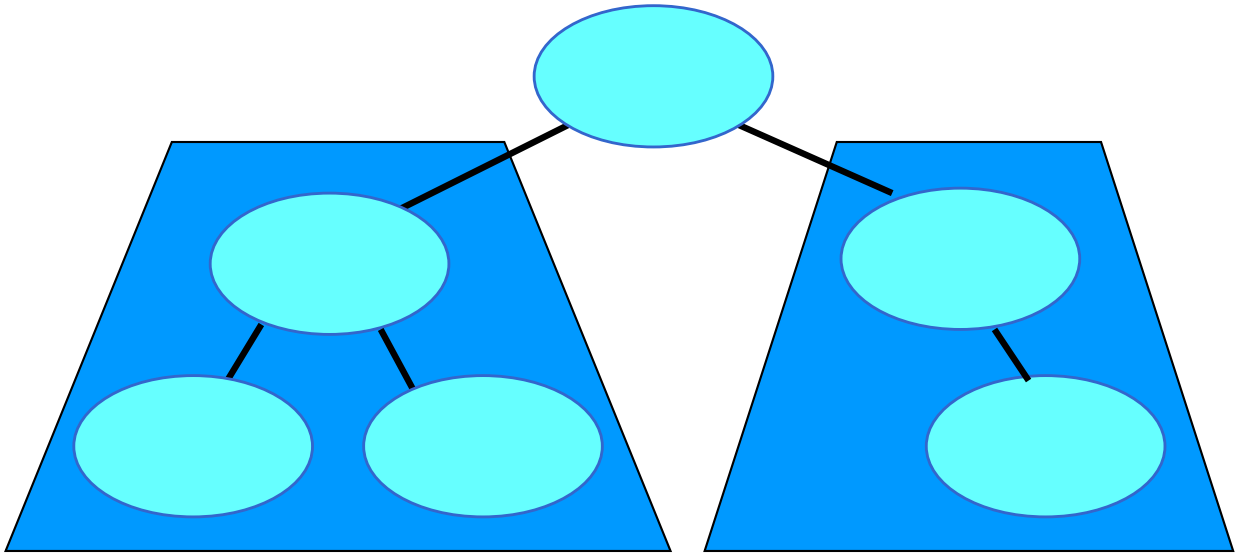
- esiste esattamente un cammino (semplice) dalla radice a qualsiasi altro nodo.
- tranne la radice, tutti i nodi hanno esattamente un padre
- un padre può avere 0 o più figli
- tra i figli di un nodo esiste un ordine che distingue il 1° nodo, il 2° nodo, etc (disegnati solitamente da sinistra a destra).



Alberi binari

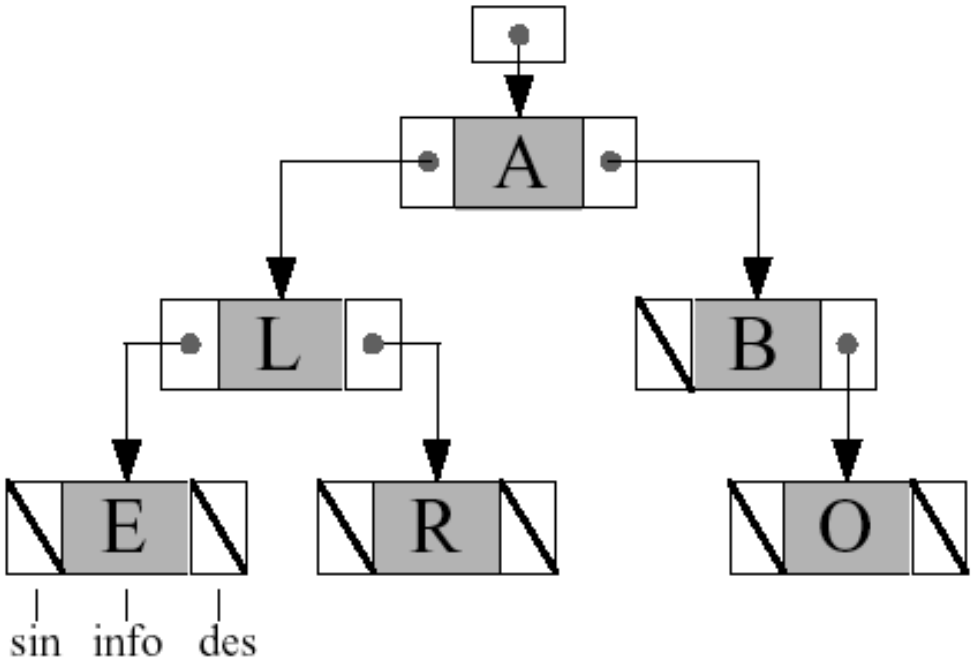
Il caso più semplice di albero è l'albero *binario*

- E' vuoto
- Oppure ha:
  - un elemento (in testa, nel nodo "radice")
  - al più due sottoalberi figli, *sinistro* e *destro*



ADT albero binario (`tree`)

- Rappresentazione collegata (puntatori a strutture)



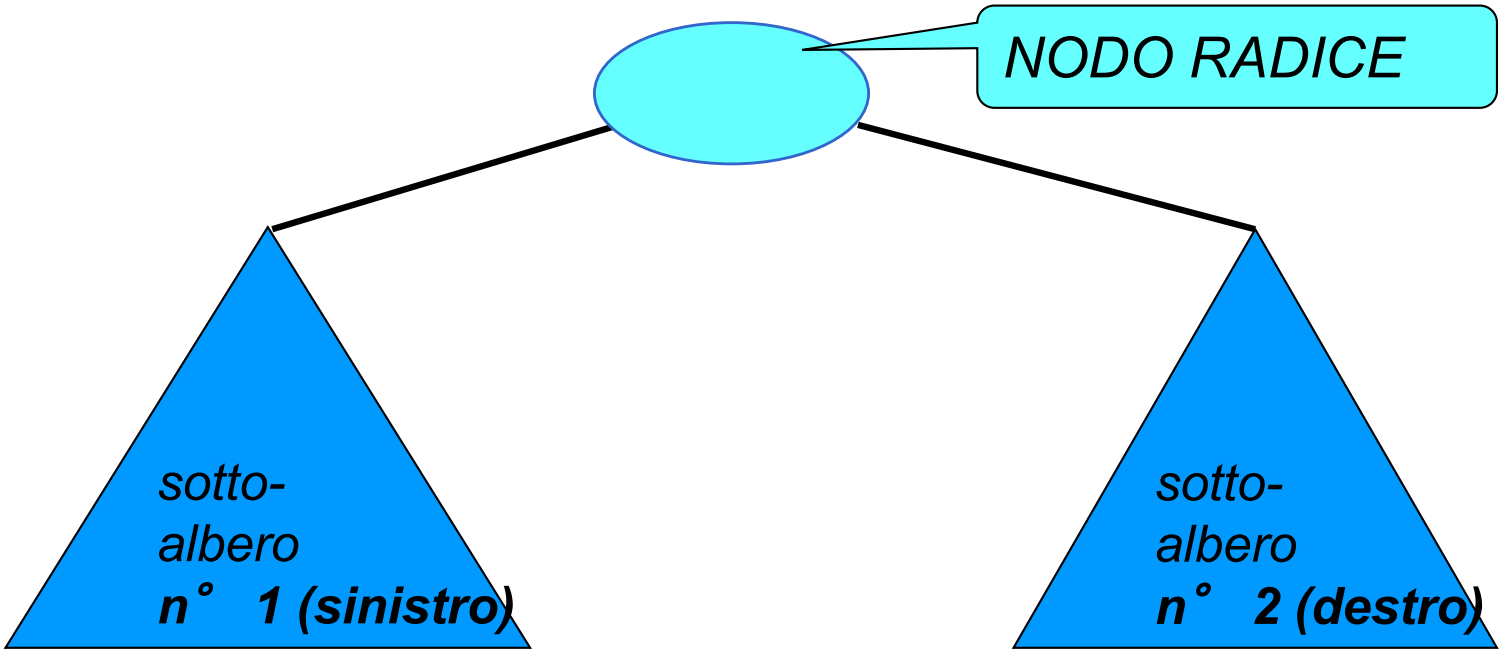
- E' possibile anche la rappresentazione mediante array (ma non la trattiamo)

Dichiarazioni e puntatore radice:

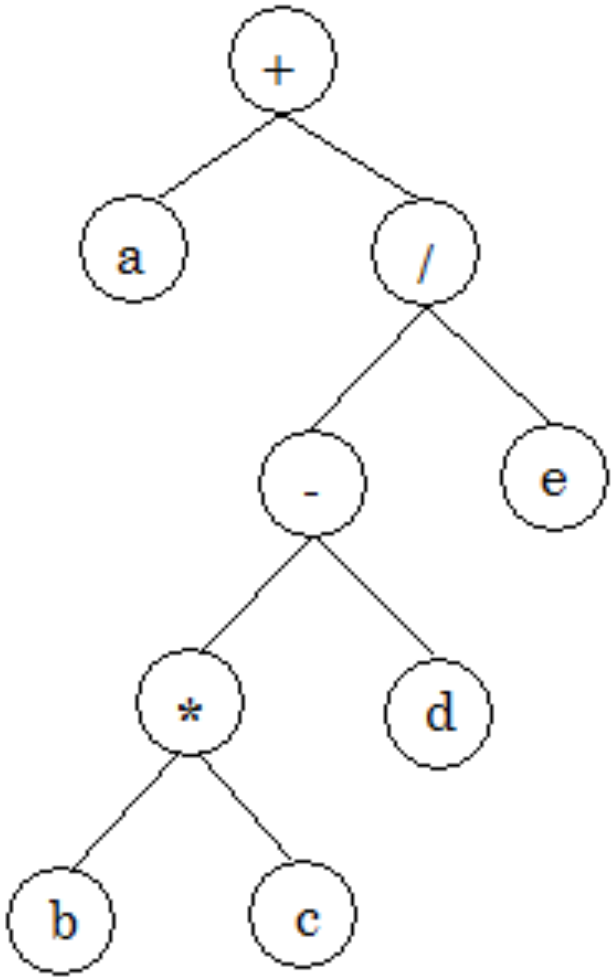
```
typedef char element;              //qui o ADT ...
typedef enum {false, true}  boolean;

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

tree root=NULL;
```

Alberi come strutture ricorsive

- Escluso il nodo radice, in un albero binario i nodi possono essere ripartiti in *due insiemi disgiunti*
- Ciascuno di tali sottoinsiemi comprende *un figlio del nodo radice più tutti (e soli) i suoi discendenti*
- Ognuno di questi sottoinsiemi individua un *(sotto)albero*



Esempio: a+(b*c – d)/e



Laboratorio:

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

Ci servono tre funzioni:

- `main`
- `cons_tree` *inserimento "in testa"*
- `showTree` *stampa albero*

tree.h

```
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 t1=NULL;
```

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 procedure di visita per l'albero)

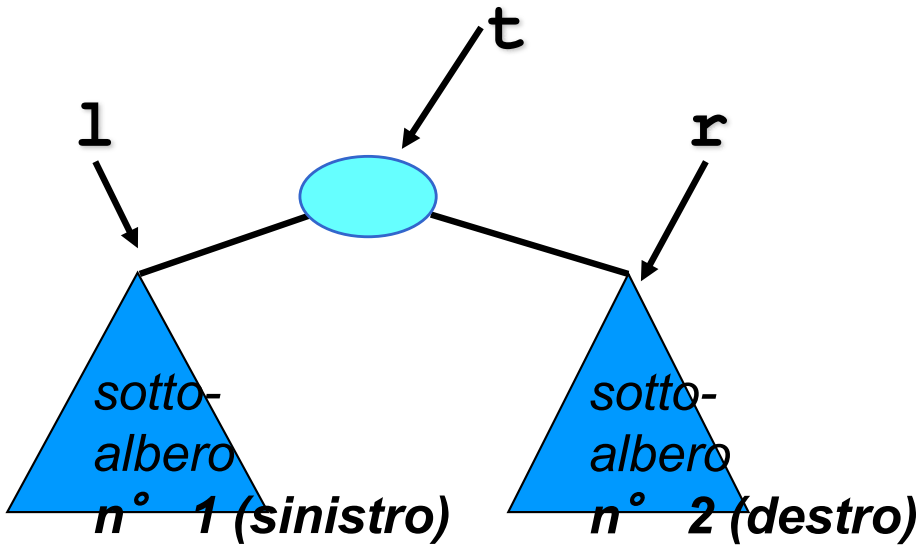
Inserimento in lista: **cons**

```
list cons(element e, list l)
{ list t;
  t = (item *) malloc(sizeof(item));
   t-> value = e;
   t-> next = l;
  return (t);  }
```

- E per un albero binario cosa dobbiamo cambiare?

cons_tree

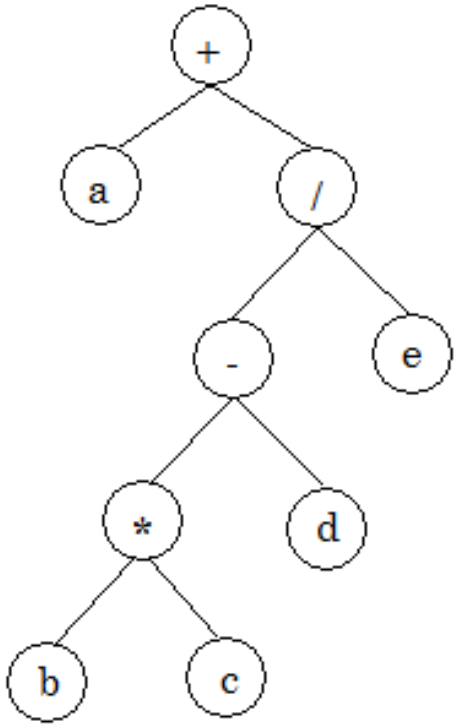
```
tree  cons_tree(element e, tree l, tree r)
/* costruisce un albero che ha nella radice e; per sotto-
   alberi sinistro e destro l ed r rispettivamente    */
{ tree t;
  t = (NODO *) malloc(sizeof(NODO));
  t-> value = e;
  t-> left = l;
  t-> right = r;
  return (t);  }
```



main.c

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

```
. . .
void main (void)
{ tree    t1=NULL,t2=NUll;
  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");
  showTree(t1); }
```



Alberi binari & Algoritmi su alberi binari

- Poiché ogni albero binario (non vuoto) è caratterizzato da
  - un valore nel nodo radice
  - due figli, *che sono anch'essi degli alberi binari*

  anche l'albero (come la lista) è una struttura dati intrinsecamente *ricorsiva.*
- Conseguentemente, gli algoritmi sono naturalmente esprimibili in modo *ricorsivo.*
  - non è rilevante come gli alberi siano realizzati
  - non importa il linguaggio (C, Java…) di implementazione
  - gli algoritmi si esprimono in modo generale basandosi solo sulla *struttura concettuale* dell'albero in termini di nodi.

Stampa di una lista: `showList` ricorsiva

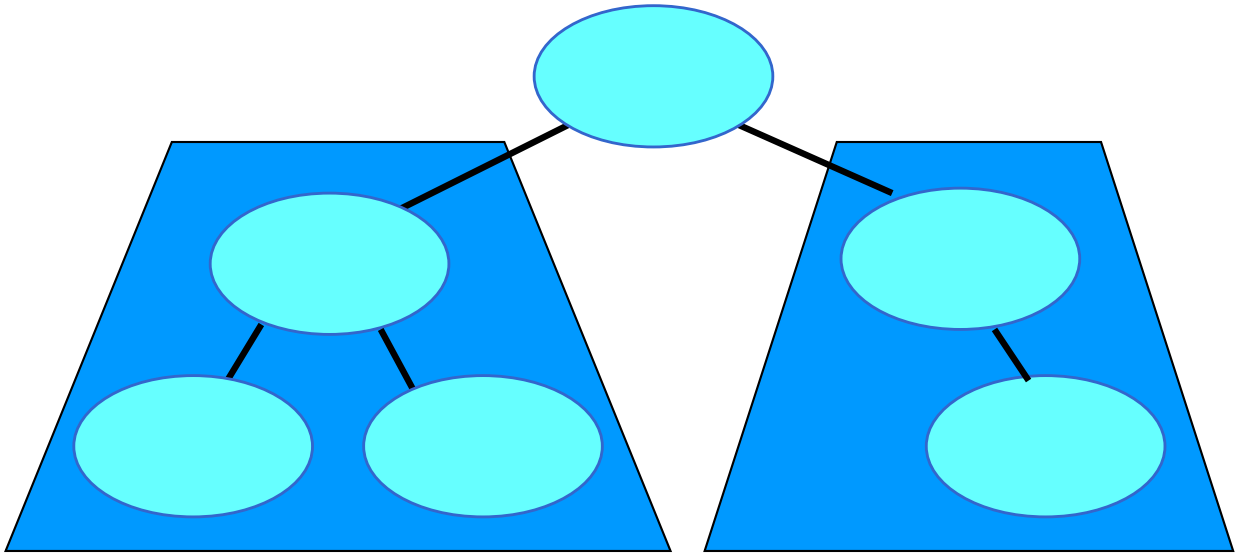
```
void showList(list l) {
     if( l!=NULL )
        { printf("%d", l->value);
          showListr( l->next );
        }
}
```

- E per un albero binario cosa dobbiamo cambiare?

Alberi binari

Un albero *binario*

- E' vuoto
- Oppure ha:
  - un elemento (in testa, nel nodo "radice")
  - al più due sottoalberi figli, *sinistro* e *destro*

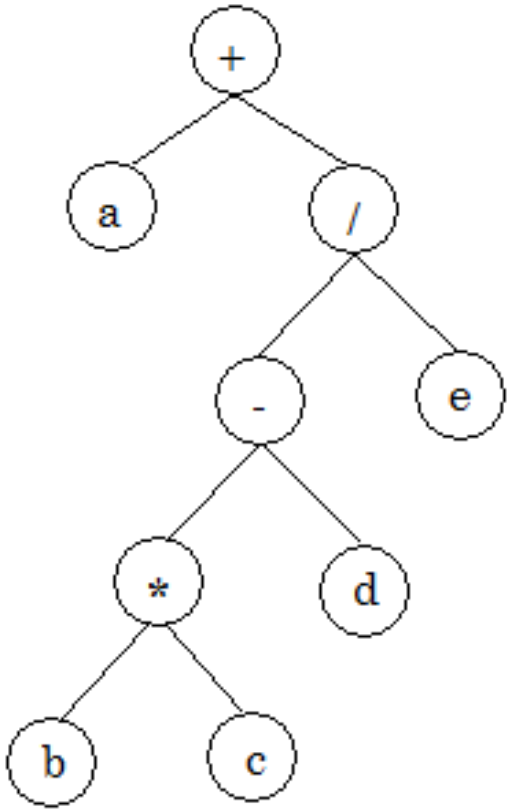


Stampa di un albero binario: **showTree**

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

main.c

```
#include <stdio.h>
#include "tree.h"
. . .
void main (void)
{ tree    t1=NULL,t2=NUll;
  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");
  showTree(t1); }
```



+a/-*bcde

Visita di un albero

- Con il termine *visita* si intende *percorrere l'albero* secondo un qualche criterio, in modo da *transitare una e una sola volta in ogni nodo.*
- Data la natura *intrinsecamente non sequenziale* dell'albero, non esiste un'unica "sequenza" degli elementi (come esisteva invece nelle liste).
- Occorre definire *uno o più criteri di visita* che assicurino di visitare tutti gli elementi dell'albero senza passare mai due volte dallo stesso nodo.

Visita di un albero

Dato che un albero (non vuoto) è definito come una struttura
caratterizzata da

- un elemento (*nodo radice*)
- N (N=2 per alberi binari) sotto-alberi

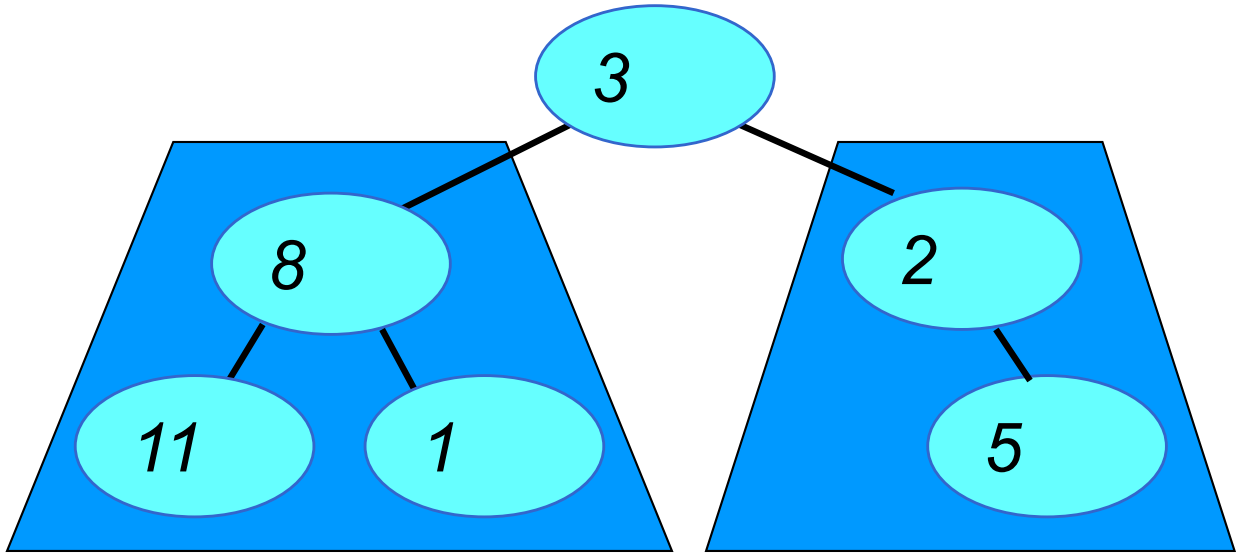
vi sono almeno due criteri "naturali" di visita:

- *visita in ordine anticipato (preorder)*
  - prima la radice
  - poi tutti i sottoalberi, in ordine da sinistra a destra
- *visita in ordine posticipato (postorder)*
  - prima tutti i sottoalberi, in ordine da sinistra a destra
  - poi la radice

Esempio:

Nel caso dell'albero sotto illustrato:

- *visita in preorder* (prima la radice, poi i sottoalberi):
  {3, *sinistro*, *destro* } = {3, {8,{11,1}}, {2,{5}}}
- *visita in postorder* (prima i sottoalberi, poi la radice):
  {*sinistro*, *destro*, 3} = {{{11,1},8}, {{5},2}, 3}



Visita di un albero binario

Nel caso di un albero binario, che ha *al più due* alberi figli, è naturale definire un terzo criterio di visita:

la *visita in ordine (inorder)*

ossia:

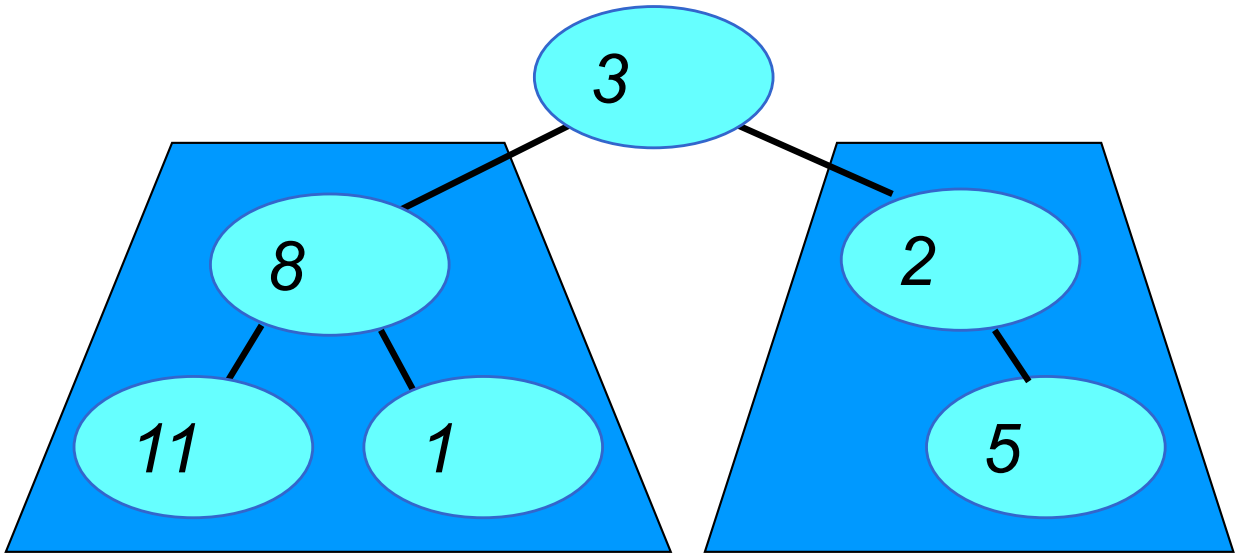
- prima il sottoalbero di sinistra,
- poi la radice,
- poi il sottoalbero di destra.

Questo criterio di visita *ha senso solo per un albero binario*

Esempio:

Nel caso dell'albero sotto illustrato:

- ***visita in ordine*** (figlio sinistro, radice, figlio destro):
  {*sinistro*, 3, *destro* } = {{{11},8,{1}}, 3, {2,{5}}}



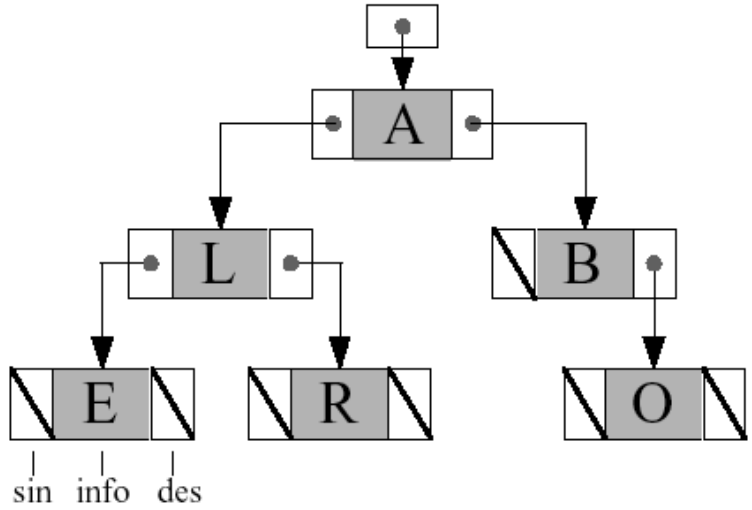
Algoritmi su alberi binari

- Gli algoritmi di elaborazione del contenuto di un albero (binario), quali:
  - visualizzazione dei valori memorizzati,
  - ricerca di un elemento,
  - calcolo del numero di elementi, etc.
- sono tutti realizzati in termini di procedure di visita

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 procedure 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)

Procedure di visita (1)

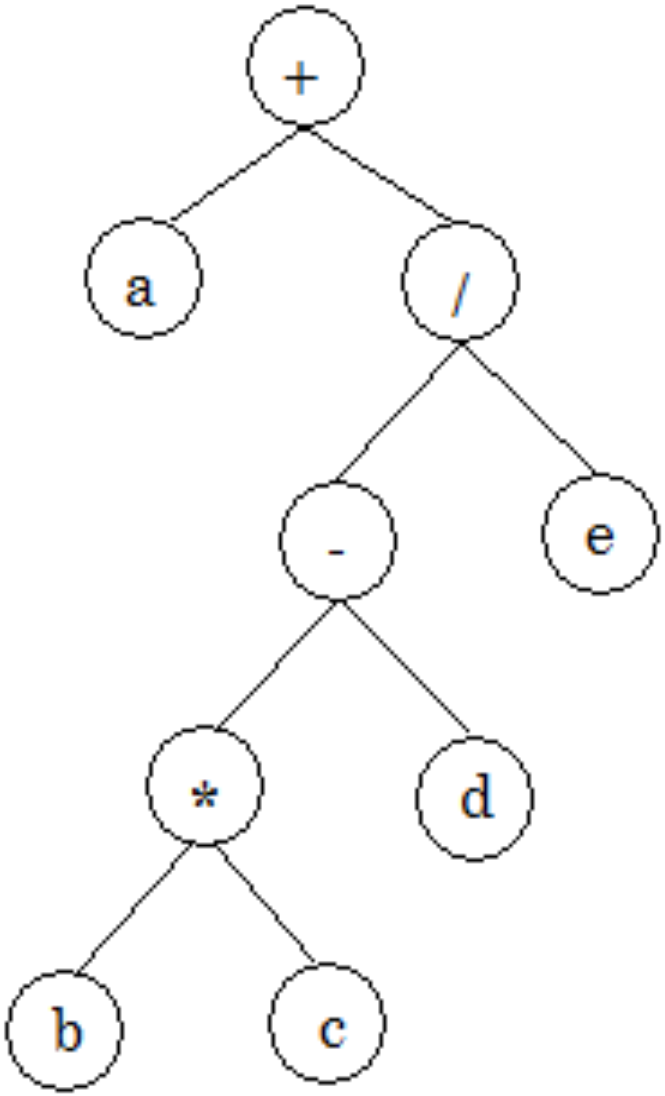
```
void preorder(tree t)
{ if (t!=NULL)
      { printf("%c",t->value);
        preorder(t->left);
        preorder (t->right);    }   }

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

Procedure di visita (2)

```
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, I^o operando, II^o operando*)

- **Postorder:**

  abc*d-e/+

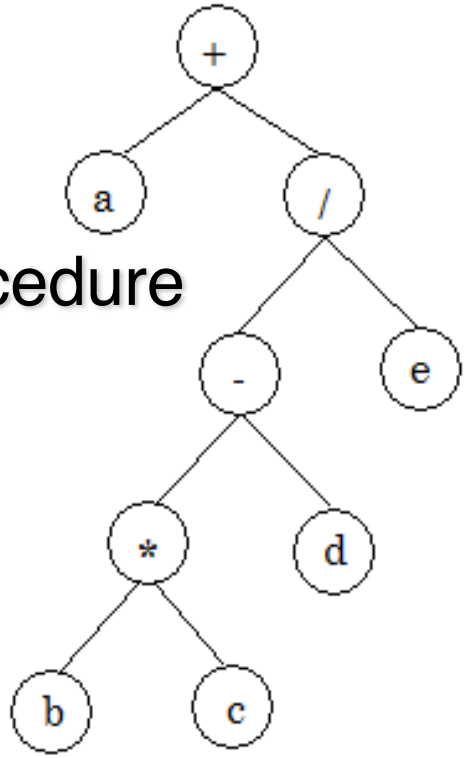
  (***polacca postfissa***, *I^o operando, II^o operando, operatore*)

- **Inorder:**

  a+b*c-d/e

  (si perde priorità operazioni e parentesi)

Laboratorio:



- Creare l'albero di caratteri in figura
- Stamparne il contenuto a video con le tre procedure di visita:
  - pre-order
  - post-order
  - in-order

- Leggere un carattere e cercarlo nell'albero
  `member(char, tree)`
- Contare (e stampare) il numero di nodi dell'albero
  `int nnodi(tree)`
- Contare (e stampare) quanti elementi uguali a quello letto ci sono `int contael(char, tree)`

Ricerca: `member`

- Poiché la visita ricorsiva *è il modo più semplice per scorrere uno ad uno tutti gli elementi di un albero, tutti gli algoritmi che operano su alberi sono una "variazione sul tema" di uno degli algoritmi di visita.*
- Cambia solo l'operazione da fare sulla radice.

✎ TO DO: ricerca di un elemento

- Se l'albero è vuoto, l'elemento non c'è, altrimenti
- Se tale elemento è quello nella radice, lo si è trovato, altrimenti
- va cercato nei sottoalberi figli.

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?

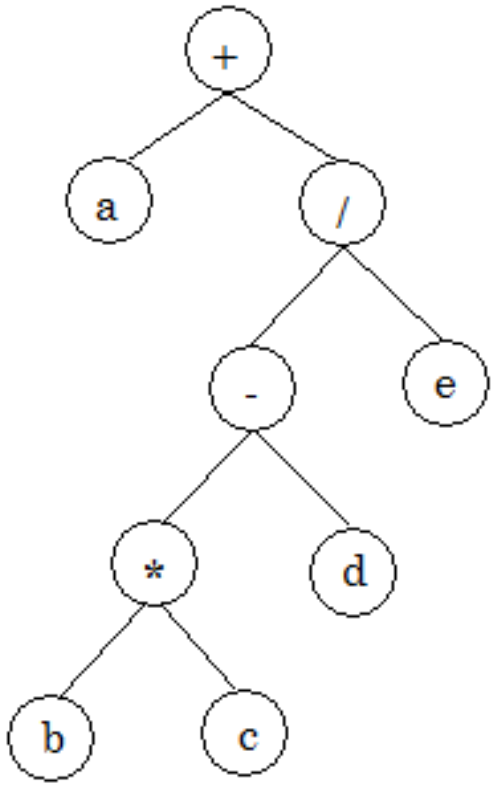
Ricerca: member

```
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) ;
}
```

- Che complessità?

Esempio: a+(b*c – d)/e

```
if (member(‘q’,t1)
     printf(“Trovato");
else printf(“No");
```



- Caso peggiore: O(N) dove N numero dei nodi dell’albero

- Come ottimizzare la ricerca?

  Alberi binari di ricerca

Conteggio del numero nodi: `nnodi`

- Poiché la visita ricorsiva *è il solo modo per scorrere uno ad uno tutti gli elementi di un albero, tutti gli algoritmi che operano su alberi sono una "variazione sul tema" di uno degli algoritmi di visita.*
- Cambia solo l'operazione da fare sulla radice.

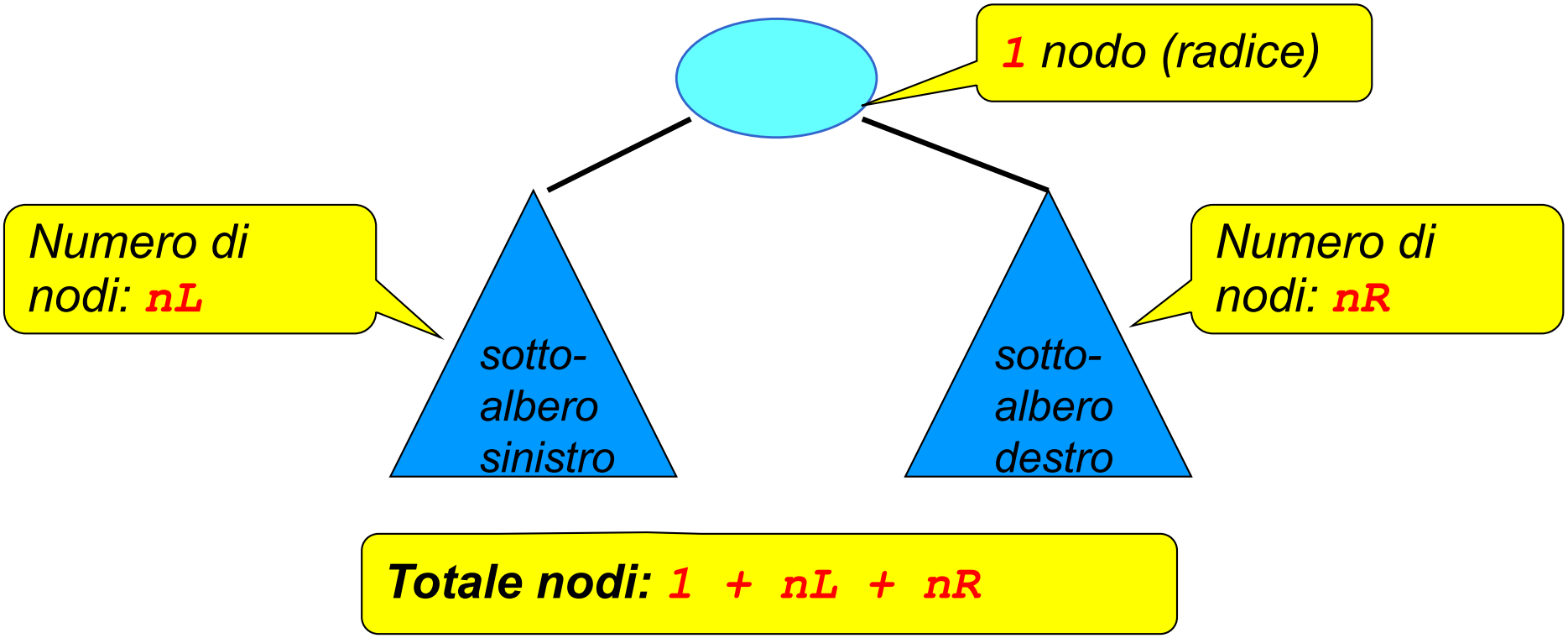
TO DO: contare il numero di nodi

- Se l'albero è vuoto, il numero è 0, altrimenti
- (c'è un nodo radice), conta 1 +numero di nodi dei due sottoal***beri***

Contare i nodi di un albero binario:

Algoritmo (per un albero binario)

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



Numero nodi: **nnodi**

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

- E' tail ricorsiva?

Conta numero elementi uguali a uno dato

Contare il numero di nodi

- Se l'albero è vuoto, il numero è 0, altrimenti
- (c'è un nodo radice), conta 1 +numero di nodi dei due sottoalberi

Inseriamo un test (==) come elaborazione sulla radice.

Quindi, modifichiamo l'algoritmo di conteggio precedente contando solo i nodi per i quali il test di uguaglianza è positivo.

Contare gli elementi dell'albero uguali a uno dato (el)

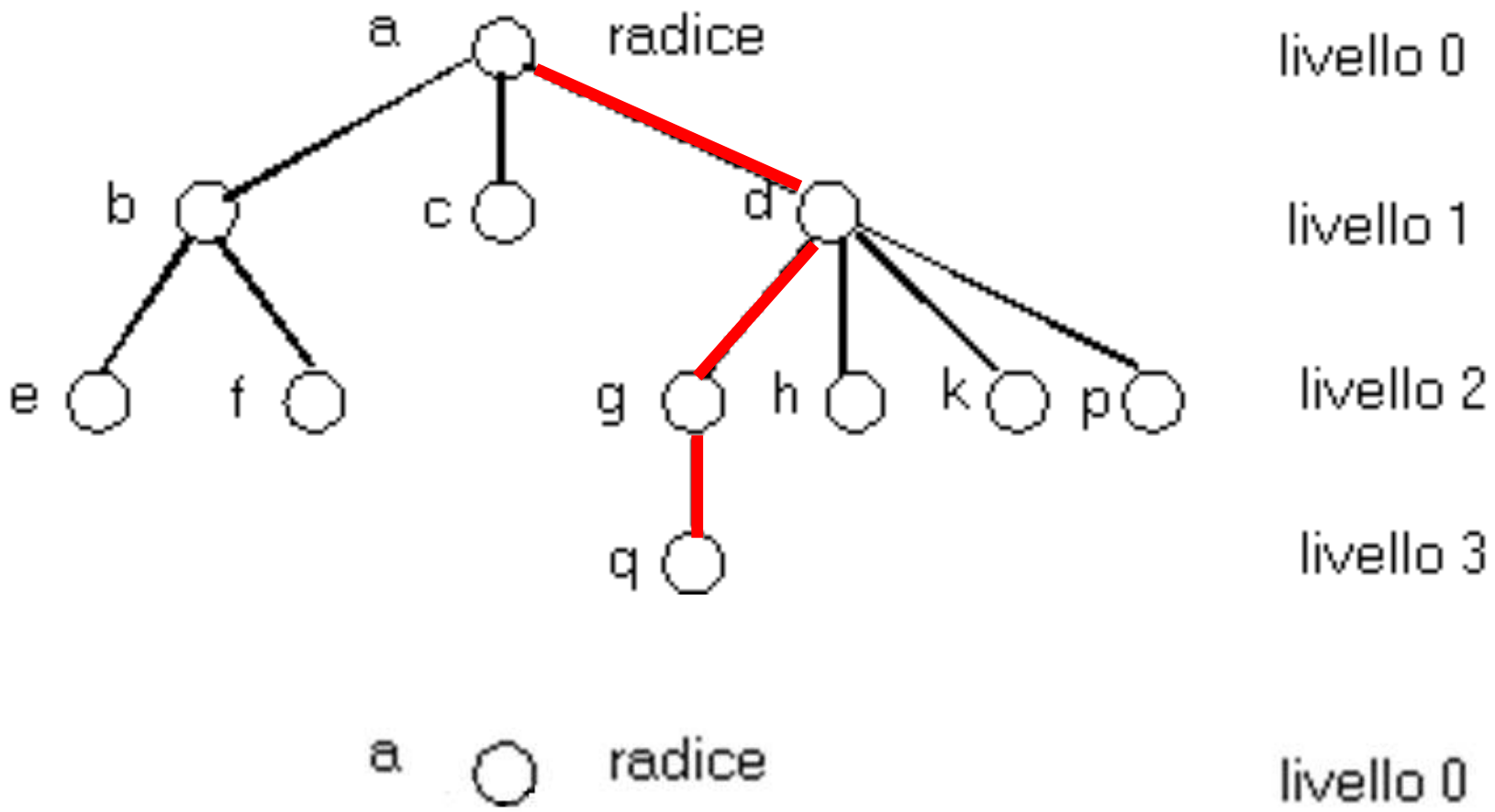
- Se l'albero è vuoto, il numero è 0, altrimenti
- (c'è un nodo radice),
  - se (el == contenuto nodo radice) conta 1 +numero di nodi dei due sottoalberi
  - altrimenti conta 0 +numero di nodi dei due sottoalberi

Conta elementi:

```
int conta_el(element e,tree t)
{ if (t==NULL) return 0;
  else
    if (e==t->value)
          return  1 + conta_el(e,t->left)
                    + conta_el(e,t->right);
    else
          return    conta_el(e,t->left)
                    + conta_el(e,t->right);
}
```

- E' tail ricorsiva?

Altezza di un albero:

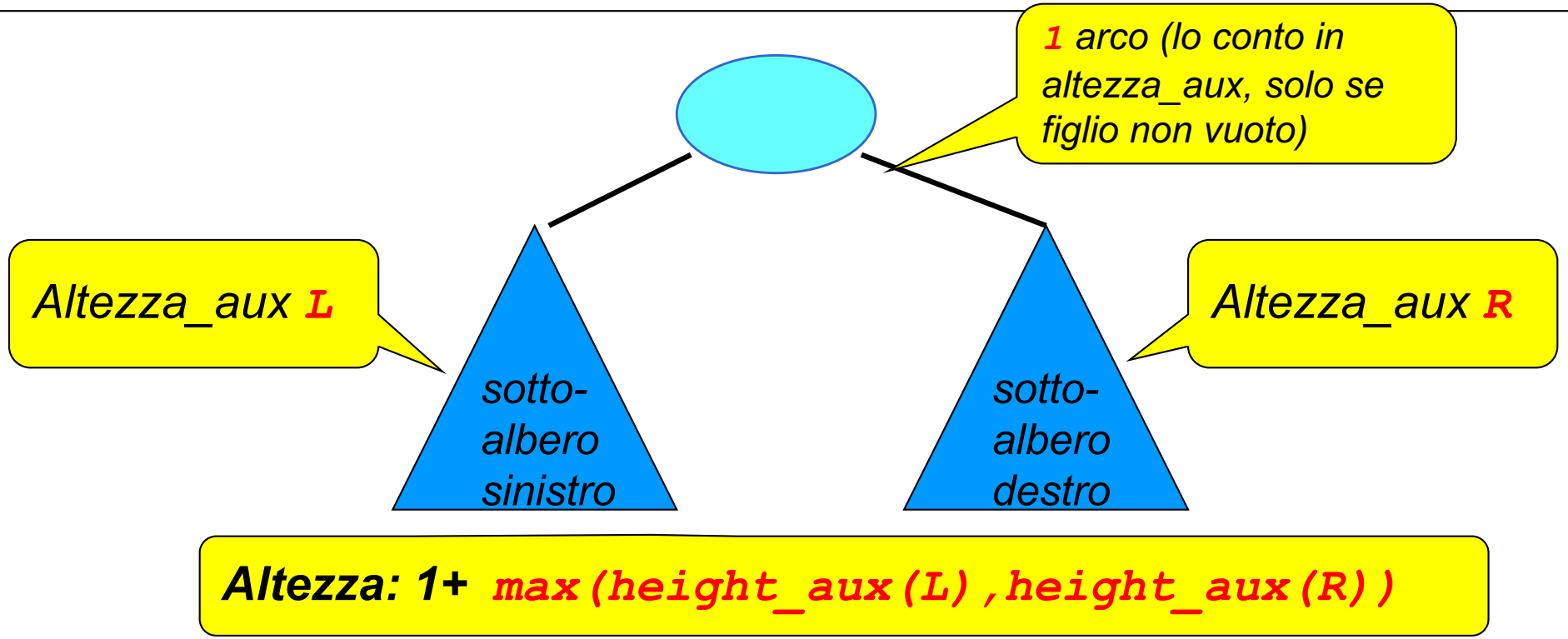


- Altezza di un albero, lunghezza del cammino più lungo dalla radice ad una delle foglie

Altezza di un albero binario:

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 (+1 se almeno una non è nulla)

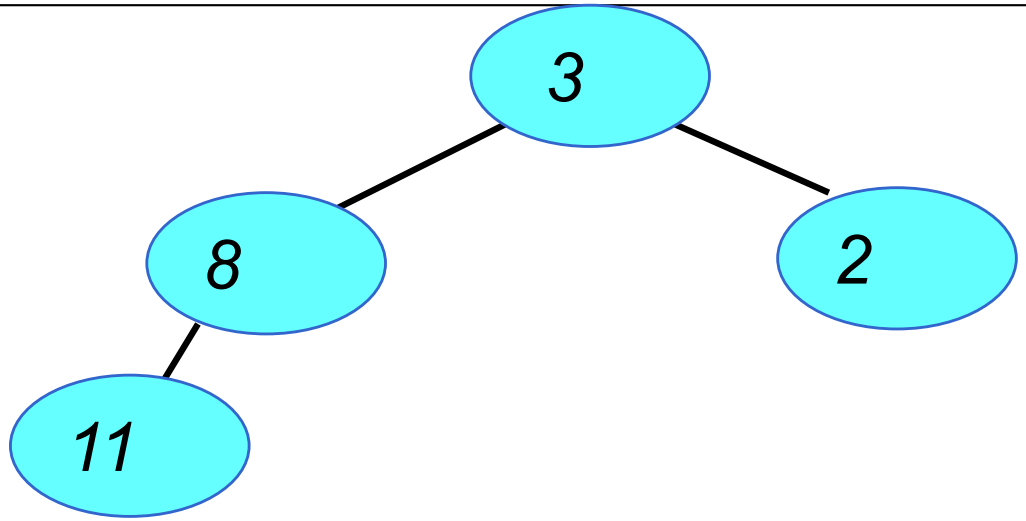


To Do: altezza di un albero

- Altezza di un albero, lunghezza del cammino più lungo dalla radice a una delle foglie

Algoritmo:

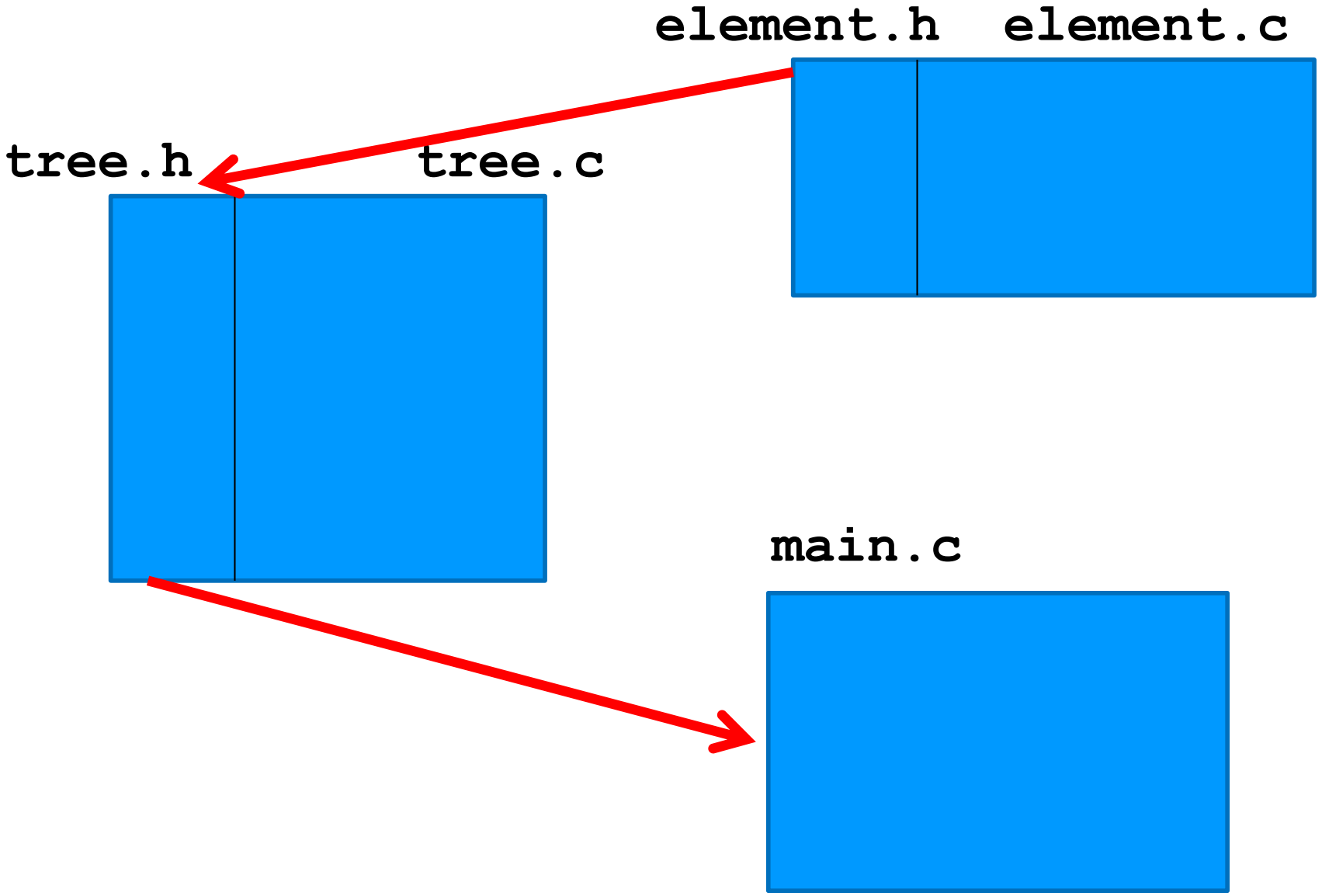
- se l'albero è vuoto, o ha solo un nodo (radice) la sua altezza è 0
- altrimenti, è 1 +
  max(altezza del figlio sinistro , altezza del figlio destro)



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))) );
}
```

Componenti e ADT



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);  . . .    // etc. etc.
```

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); }
```

Ricerca: **member**

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

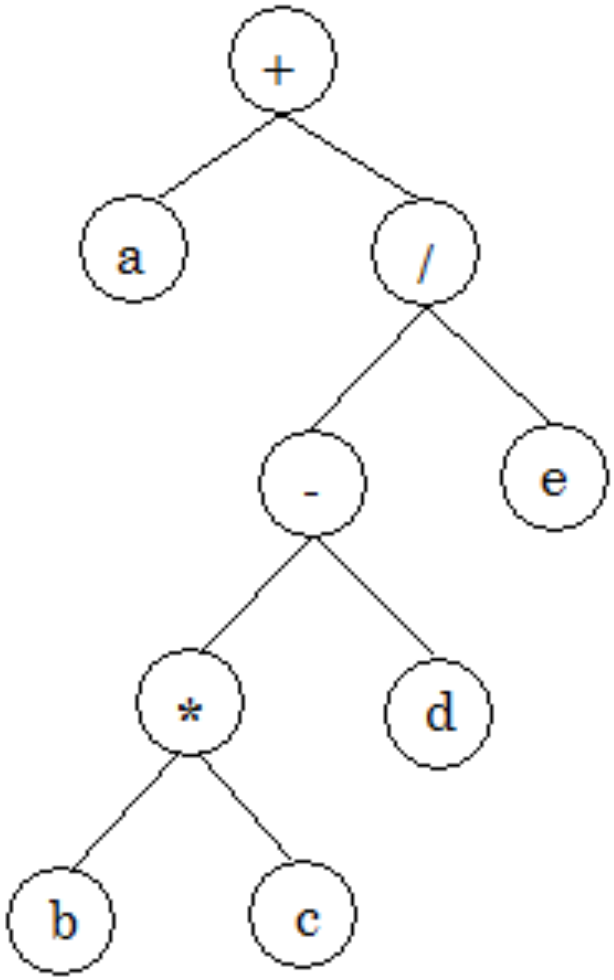
- E' tail ricorsiva?

Ricerca: **member**

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

- E' tail ricorsiva?

Esempio: a+(b*c – d)/e



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); }
```