Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Alberi e Strutture Dati Generiche in C

Corso di Laurea in **Informatica**, III anno

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **24**, 31 maggio 2021

Lezione 24a:

Alberi binari

Alberi (binari e non)

Abbiamo visto in Haskell vari tipi di albero (binario e non) polimorfo. Ricordiamo brevemente alcune definizioni:

```
-- alberi con alberi vuoti:
data BinTree a = Empty |
               Node a (BinTree a)(BinTree a)

-- a partire dalle foglie, con etichette
-- solo sulle foglie:
data BinTree' a = Leaf a |
               Node (BinTree a) (BinTree a)

-- alberi non necessariamente binari:
data Tree a = Fork [Tree a]
```

Ma in **C**?

Alberi: rappresentazione in C

Come per le liste vedremo prevalentemente una **rappresentazione a puntatori** (saprete che ci sono altre rappresentazioni)

Un albero non vuoto *r*(*L, R*) contiene sempre l'elemento *r* (**radice** dell'albero) attaccato ai sotto-alberi *L* ed *R* (**sottoalbero destro** e **sinistro**).

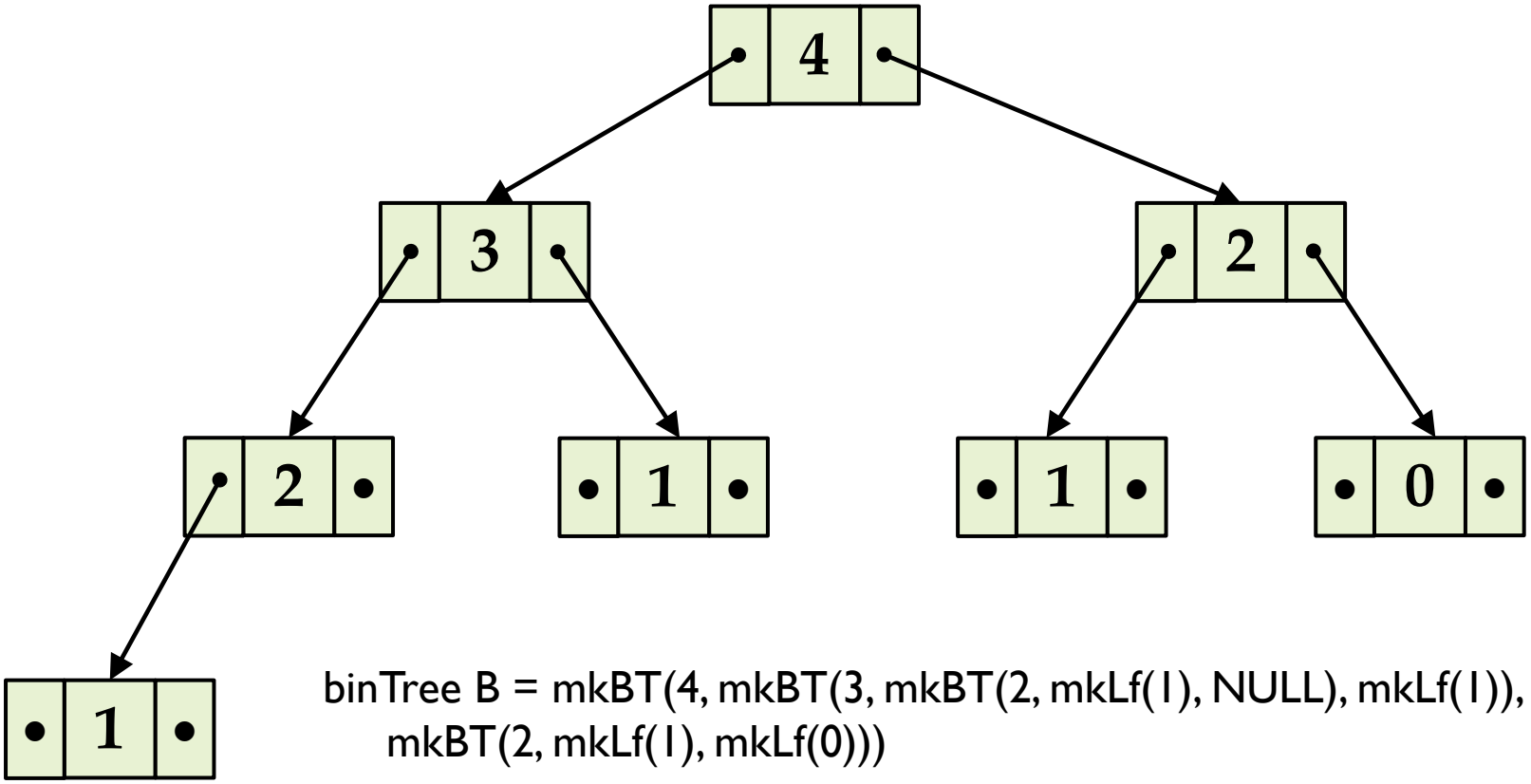
Come nel caso delle liste, useremo una struct per memorizzare un nodo dell'albero che consiste 1) di un campo **informazione** e 2) due campi puntatore ai **sottoalberi** (e anche qui i puntatori vengono usati per le definizioni ricorsive di tipo).

```
typedef
struct B {
   struct B * left;
   int info;
   struct B * right;
} binTreeNode;

typedef binTreeNode* binTree;
```

Alberi: rappresentazione in memoria

Con le definizioni date, gli alberi in memoria sono rappresentati come tante `perline' tenute insieme dai puntatori. Ecco l'albero descritto prima in memoria (dove • rappresenta il pointer `NULL`)




```
binTree B = mkBT(4, mkBT(3, mkBT(2, mkLf(1), NULL), mkLf(1)),
    mkBT(2, mkLf(1), mkLf(0)))
```

Detour: equivalenza di tipi

Forse qualcuno si è accorto che **da un punto di vista strutturale**, gli **alberi** sono del tutto **uguali alle liste doppiamente concatenate**: ogni nodo contiene un campo informazione e due campi puntatore.

Tuttavia è molto diverso come i campi puntatore vengono usati, a riprova che anche le **operazioni determinano un tipo**!

I linguaggi di programmazione hanno due forme di equivalenza tra tipi:

- **Equivalenza strutturale**: due tipi sono equivalenti se hanno la stessa struttura (ad esempio alberi e liste doppiamente concatenate).
- **Equivalenza per nome**: due tipi sono equivalenti se hanno lo stesso nome.

L'equivalenza per nome sottintende che se il programmatore definisce due tipi con la stessa struttura ma nomi diversi si protegge da sé stesso a usare in modo incoerente i tipi.

Alberi: funzioni costruttori in C

Anche nel caso degli alberi, useremo il puntatore NULL per rappresentare l'**albero vuoto #**.

Il costruttore canonico degli alberi di interi sarà una funzione binTree makeBT(int, binTree, binTree). Lo costruiamo in due passi, definendo anche un costruttore binTree makeLeaf(int).

```
binTree makeLeaf(int r){
   binTree B;
   B = (binTree)malloc(sizeof(binTreeNode));
   B->info = r;
   B->left = NULL;
   B->right= NULL;
   return B;
}
```

```
binTree makeTree(int r, binTree L,
                                binTree R){
   binTree B = makeLeaf(r);
   B->left = L;
   B->right= R;
   return B;
}
```

Alberi: distruttori in C

È sufficiente un **unico distruttore**:

```
int isEmptyBT(binTree B, int* r,
              binTree *L, binTree *R){
   if (!B) return 1;
   *r = B->info;
   *L = B->left;
   *R = B->right;
   return 0;
}
```

… oppure una suite di 4 distruttori:

```
int isEmptyBT2(binTree B)
   if (!B) return 1;
   return 0;
}
```

```
binTree right(binTree B)
/* PREC: B!= # */
   return B->right;
}
```

```
binTree left(binTree B)
/* PREC: B!= # */
   return B->left;
}
```

```
int root(binTree B)
/* PREC: B!= # */
   return B->info;
}
```

Prime funzioni su alberi in C

Vediamo l'implementazione in **C** di alcune semplici funzioni:

```
int nodes(binTree B){
   int r;
   binTree L, R;
   if (isEmptyBT(B, &r, &L, &R))
         return 0;
   return 1+nodes(L)+nodes(R);
}
```

```
int weight(binTree B){
   int r;
   binTree L, R;
   if (isEmptyBT(B, &r, &L, &R))
         return 0;
   return r+weight(L)+weight(R);
}
```

```
int depth(binTree B){
   int r;
   binTree L, R;
   if (isEmptyBT(B, &r, &L, &R))
         return -1;
   return 1+max(depth(L), depth(R));
}
```

Prime funzioni su alberi in C

Leggermente più laboriosa la funzione
`int equalsBT(binTree, binTree)`:

```
int equalsBT(binTree B1, binTree B2){
   int r1, r2, e1, e2;
   binTree L1, L2, R1, R2;
   e1 = isEmptyBT(B1, &r1, &L1, &R1));
   e2 = isEmptyBT(B2, &r2, &L2, &R2));
   if (e1 && e2) return 1;
      /* almeno uno è non vuoto */
   if (e1 || e2) return 0;
      /* entrambi sono non vuoti */
   return r1==r2 && equalsBT(L1, L2)
         && equalsBT(R1, R2);
}
```

Verificare `r1==r2` *è più a buon mercato quindi conviene metterlo prima, sfruttando l'ordine di valutazione di* `&&`

Lezione 24b:

Strutture Dati Generiche in C

Come gestire il polimorfismo in C?

Ovviamente un gran numero di funzioni sugli alberi **non dipendono** dal fatto che gli alberi contengano **valori interi** o di altro tipo (profondità, bilanciamento etc.).

Come è possibile avere **alberi generici** o **polimorfi**?

In **C** c'è un'opportunità data dal tipo `void *` che è compatibile per assegnazione con ogni altro tipo puntatore.

Quindi, è possibile definire strutture dati generici, mettendo nel campo informazione dei `void *`.

Si tratta di una tecnica del tutto analoga a quella usata ad esempio in **Java prima dell'introduzione dei Generics**: strutture dati generiche venivano implementate mantenendo nelle strutture dati, oggetti di tipo `Object`.

Alberi Generici in C

La rappresentazione è del tutto analoga a quella degli alberi appena visti.

Cambia solo il tipo del campo informazione, che stavolta sarà di tipo `void *`.

Attenzione: occorre modificare opportunamente le funzioni che ad esempio aggiungono elementi in un albero: in particolare, occorre sempre allocare memoria dinamica per le informazioni inserite in un albero.

```
typedef
struct BG {
   struct BG * left;
   void* info;
   struct BG * right;
} binTreeGNode;

typedef binTreeGNode* binTreeGen;
```

Tipi Generici in C: Limiti e Vantaggi

Ovviamente questo trucco ha alcuni punti deboli:

1. Non c'è **nessun controllo** sulle informazioni inserite in una struttura dati: il sistema dei tipi non controlla il tipo delle informazioni inserite;
2. Quando vado a estrarre informazioni, il programmatore deve assegnare il valore di tipo `void *` a una variabile di tipo T* per poterlo usare (con `T` tipo noto).

Al solito, come contraltare, guadagno in flessibilità, in particolare, in alcuni casi potrebbe essere "vantaggioso" o quantomeno "comodo" poter inserire valori di tipi non omogenei dentro una struttura dati.

Esempio: una pila generica potrebbe rappresentare la pila di sistema, dove ogni funzione ha un record di attivazione di tipo diverso rispetto alle altre funzioni.

Esempio: Pile Generiche (1)

Vediamo rapidamente l'esempio delle pile generiche.

Implementiamo le Pile, usando una lista semplice.

```
typedef struct S {
    void * val;
    struct S * next;
} Snode;

typedef struct {
    Snode* top;
    int numElem;
} stackDescriptor;

typedef stackDescriptor *stack;
```

Esempio: Pile Generiche (2)

Vediamo le operazioni della pila.

```
stack createEmptyStack(){
    stack S;
    S = malloc(sizeof(stackDescriptor));
        /* la cima della pila viene inizializzata a NULL  */
    S->top = NULL;
        /* numero di elementi a 0  */
    S->numElem = 0;
    return S;
}
```

```
int isEmpty(stack S){
        /* equivalente a return S->top==NULL */
    return S->numElem==0;
    }
```

```
void pop(stack S){
    Snode *tmp;
        /* si mantiene l'invariante del tipo di dato */
    S->numElem--;
        /* salva il putatore al primo elemento per la free */
    tmp = S->top;
        /* modifica il puntatore al top della pila */
     S->top = (S->top)->next;
     free(tmp);
}
```

```
void* top(stack S){
    return (S->top)->elem;
}
```

Esempio: Pile Generiche (3)

Vediamo le operazioni della pila.

```
void push(stack S, void* el){
   Snode *tmp;
      /* alloca memoria per inserire un nuovo nodo */
   tmp = malloc(sizeof(Qnode));
       /* il link del nuovo nodo punta al vecchio top */
   tmp->next = S->top;
   tmp->elem = el;
   S->top = tmp;
      /* si mantiene l'invariante del tipo di dato */
   S->numElem++;
}
```

Uso delle Pile Generiche (2)

Consideriamo il problema della Torre di Hanoi, mantenendo lo stato del gioco in tre pile di dischi.

Inoltre, useremo le stesse pile per produrre un programma iterativo che elimina la ricorsione simulando esplicitamente la pila di sistema.

Cominciamo con le torri del gioco.

```
void inizializzaTorri(stack torri[], int n){
    int *x;
    int i;

    for (i=0; i<3; i++)
        torri[i]=createEmptyStack();
    for (i=n; i>0; i--){
          x = malloc(sizeof(int));
          *x=i;
          push(torri[0], x);
        }
}
```

Sarebbe un errore da principianti fare **`push(torri[0], &i)`**

Uso delle Pile Generiche (2)

Mostriamo la versione ricorsiva e la mossa di un disco.

```
void hanoi(int sorg, int aux, int dest, int n, stack torri[]){
/* inizio */
   if (n==1) move(sorg, dest, torri);
      else {
         hanoi(sorg, dest, aux, n-1, torri);
/* mezzo */
         move(sorg, dest, torri);
         hanoi(aux, sorg, dest, n-1, torri);
/* fine */
      }
 }
```

```
void move(int sorg, int dest, stack torri[]){
    push(torri[dest], top(torri[sorg]));
    pop(torri[sorg]);
    printTorri(torri);
}
```

Lo stato delle torri può essere necessario ad es. per una versione grafica

Uso delle Pile Generiche (3)

```
void hanoiIterOpt(stack torri[]){
    hanoiAR2 *newAR, *AR;
    stack controlStack;
    int aux, sorg, dest, nd, w;
    controlStack = createStack(2048);
    sorg=0; aux=1; dest=2; nd=howManyStack(torri[0]);
```

```
while (1) {
      if (nd == 1) {
         move(sorg, dest, torri);
         if (!isEmpty(controlStack)) {
             AR = top(controlStack);
             pop(controlStack);
             sorg = AR->sorg; dest = AR->dest;
             aux = AR->aux; nd = AR->nd;
         } else break;
       } else {
         newAR = createAR2(aux, sorg, dest, nd - 1);
         push(controlStack, newAR);
         newAR = createAR2(sorg, aux, dest, 1);
         push(controlStack, newAR);
         nd--; w=aux; aux=dest; dest=w;
       } /* end else */
 }/* end while */
```

Lezione 24c:

Problemi un po' più seri sugli alberi binari

"Ricostruzione" di un albero binario

Avendo una sola visita, non è possibile conoscere la struttura di un albero binario. Ma avendone due?

Supponiamo di avere due vettori caricati uno con una visita preorder e uno con una visita inorder. La visita **preorder ci permette di identificare subito la radice**, che è il primo elemento del vettore.

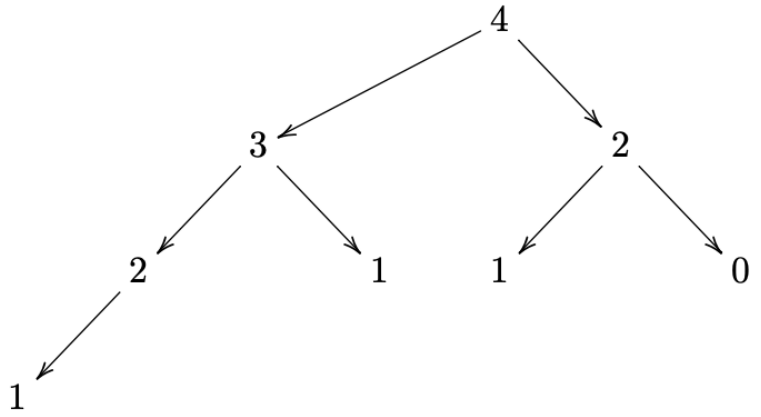
Conoscendo la radice, e **supponendo che essa sia unica**, la visita **inorder ci permette di separare la visita del sottoalbero sinistro dalla visita del sottoalbero destro**.

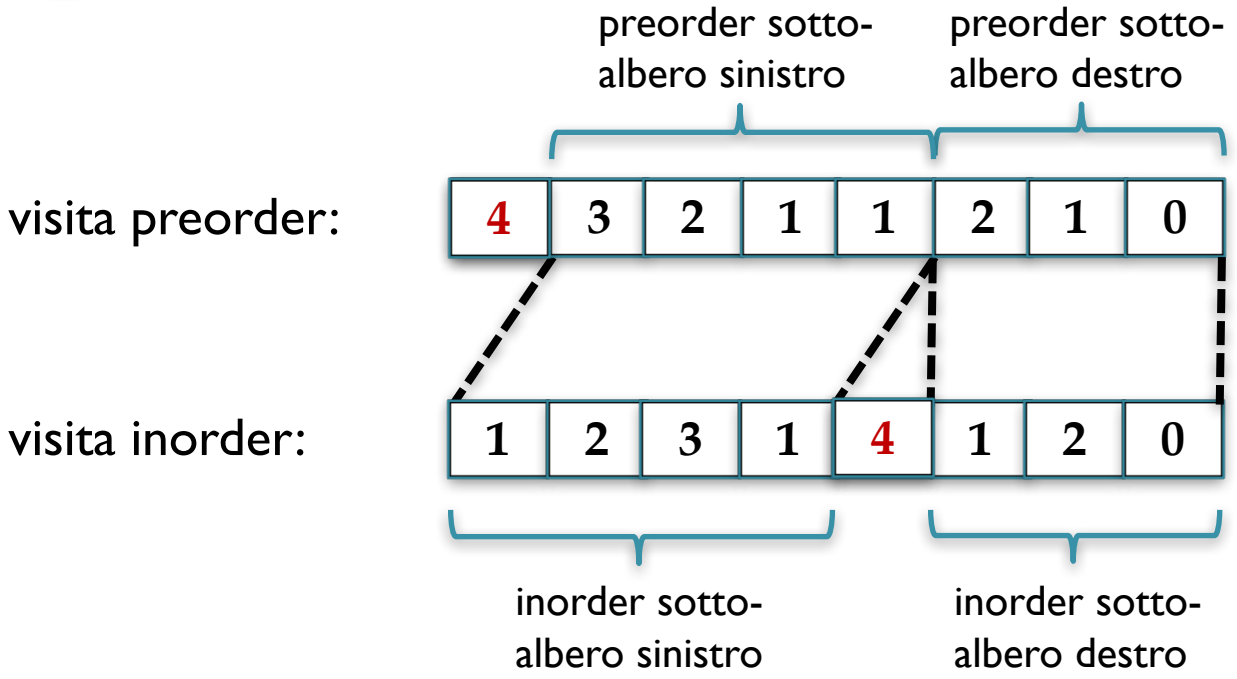
E poi di ricostruirle nella visita preorder (è sufficiente la cardinalità dei sottoalberi) e quindi chiamarsi ricorsivamente.

Quindi, se la radici di tutti i sottoalberi sono uniche (detto in altro modo: **in ogni cammino radice-foglia non ci sono ripetizioni di etichette**) posso ricostruire in modo univoco un albero binario partendo da una visita in-order e una preorder (o postorder).

Esempio: idea ricorsiva

Ci sono etichette ripetute, ma non su cammini radice-foglia





"Ricostruzione" di un albero binario

```
binTree visitsToTree(int* preO, int*inO, int n){
   binTree B;
   int p;
   if (n<=0) return NULL;
   B = makeLeaf(preO[0]);
      /* cerco la radice nella visita inOrder: */
   find(preO[0], inO, n, &p);
   B->left = visitsToTree(preO+1, inO, p);
   B->right= visitsToTree(preO+p+1, inO+p+1, n-p-1);
   return B;
}
```

```
int find(int x, int* v, int n,
                        int* res){
   for(int i=0; i<n; i++)
      if (v[i]==x){
            *res = i;
            return 1;
         }
   return 0;
}
```

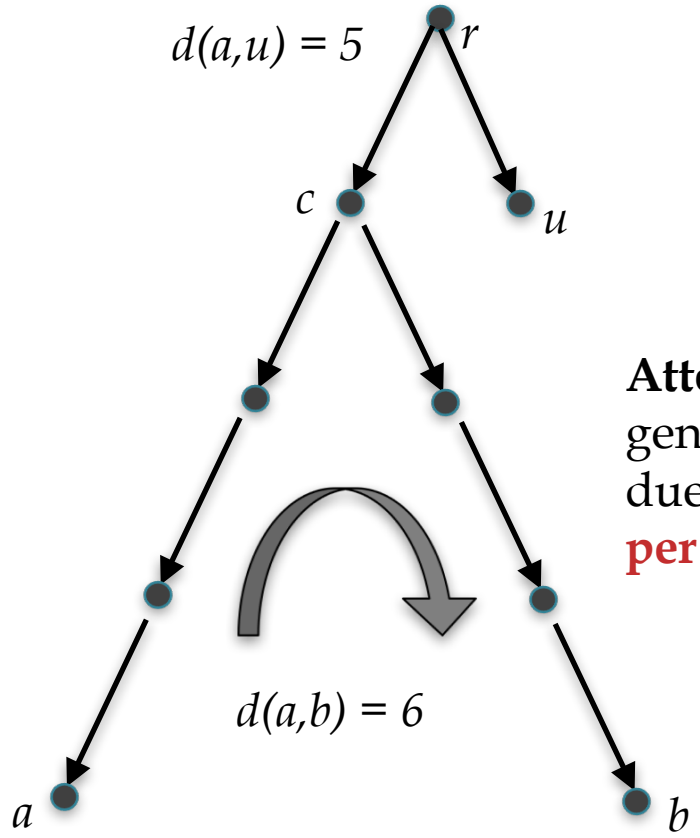
"Ricostruzione" di un albero binario

Con uno stile più "**funzionale**" e **compatto**:

```
binTree visitsToTree(int* preO, int*inO, int n){
   int p;
   if (!n) return NULL;
   find(preO[0], inO, n, &p);
   return makeTree(*preO,
           visitToTree(preO+1, inO, p),
           visitToTree(preO+p+1, inO+p+1, n-p-1)
           );
}
```

Diametro di un albero

Definizione: Il **diametro** di un albero **è la massima distanza** (in numero di archi) **tra due nodi** dell'albero.



Attenzione! Osservate che, **non è** in generale **detto** che il cammino più lungo tra due nodi nell'albero **passi necessariamente per la radice dell'albero**.

$$\begin{aligned} \text{diameter}(r(L,R)) &= \max\{\text{diameter}(L), \text{diameter}(R), \text{depth}(L) + \text{depth}(R) + 2\} \\ \text{diameter}(\sharp) &= 0 \end{aligned}$$

Diametro di un albero

Come nel caso del bilanciamento, la sfida è calcolare il diametro con **un'unica scansione** ricorsiva dell'albero.

```
int diameterAux(binTree B, int* p){
   int r, p1, p2; /* profond. Sottoalb.*/
   binTree L, R;
   if (isEmptyBT(B, &r, &L, &R)){
            *p=-1;
            return -1;
      }
   int d = max(diameterAux(L, &p1),
               diameterAux(R, &p2));
   *p = max(p1, p2) + 1;
   return max(d, p1+p2+2));
}

int diameter(binTree B){
   int p;
   return diameterAux(B, &p);
}
```

Ritornare i nodi al livello k

2.2 Lista dei Nodi al Livello k-esimo (secondo esonero 2016)

Scrivere una funzione `C Lista livelloK(binTree B, int k)` *che, presi in input un albero binario di interi B e un numero intero k, ritorna in output una lista di interi contenente le etichette di B al livello k.*

ESEMPIO: *Ricevendo in input l'albero in Fig. 2 e 0, la funzione torna la lista $\langle -3 \rangle$. Con input 1 torna la lista $\langle 1, 6 \rangle$. Con input 2, la lista $\langle 2, -5, 7 \rangle$. Con input 3, la lista $\langle 4 \rangle$. Infine, per ogni valore $k > 3$, torna la lista vuota.*

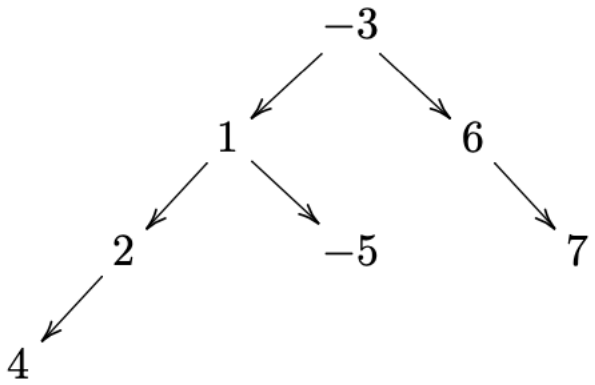


Figura 2: Albero binario nell'esempio

Difficoltà

Purtroppo, molti studenti, alla parola livello si sono lanciati a capofitto a visitare l'albero per livelli.

Alcuni sono riusciti a fare osservazioni interessanti (ad esempio in una visita a livelli, nella cosa ci sono nodi di al più 2 livelli consecutivi) e a uscirne vivi.

Tuttavia il programma era molto laborioso: in particolare **occorre trovare un modo di capire quando comincia (e finisce) il livello di nostro interesse** (forse il modo più semplice era usare una coda in cui inserire coppie nella forma (l, b), dove l rappresenta il livello del sottoalbero b.

Tuttavia, si poteva fare ricorsione `naturale' su alberi binari:

Vediamo la semplice specifica di livelloK per induzione sulla struttura dell'albero (e su k):

$$\begin{aligned} \mathsf{livelloK}(\_, n) &= \langle\,\rangle \\ \mathsf{livelloK}(r(L,R), 0) &= \langle r \rangle \\ \mathsf{livelloK}(r(L,R), k+1) &= \mathsf{append}(\mathsf{livelloK}(L,k), \mathsf{livelloK}(R,k)) \end{aligned}$$

Soluzione 1

Tuttavia, sappiamo che è sempre preferibile costruire liste a colpi di `cons`, in quanto `append` è lineare e non costante.

I **risultati vengono via via accumulati in un parametro ausiliario** mentre **l'albero viene percorso `a rovescio'** (prima R e poi L).

```
list livelloKAux(binTree B, int k, lista M){
   int r, h;
   binTree L, R;
   if (isEmptyBT(B, &r, &L, &R)
      return M;
   if (!k) return cons(r, M);
   M = livelloKAux(R, k-1, M);
   return livelloKAux(L, k-1, M);
}/* anche:
  * return livelloKAux(L, k-1,
  *                  livelloKAux(R, k-1, M) */

list livelloK(binTreeList B, int k){
   return livelloKAux(B, k, NULL);
}
```

Soluzione 2

In alternativa si poteva **accumulare i risultati in un parametro passato per indirizzo**. Ma il concetto era uguale.

```
void livelloKAux(binTree B, int k, lista* M){
   int r;
   binTree L, R;
   if (!isEmptyBT(B, &r, &L, &R){
      if (!k) *M = cons(r, *M);
         else {
            livelloKAux(R, k-1, M);
            livelloKAux(L, k-1, M);
         }
}

list livelloK(binTree B, int k){
   list L = NULL
   livelloKAux(B, k, &L);
   return L;
}
```

Lezione 24c:

Piccole note pratiche

Suddivisione programmi C (1)

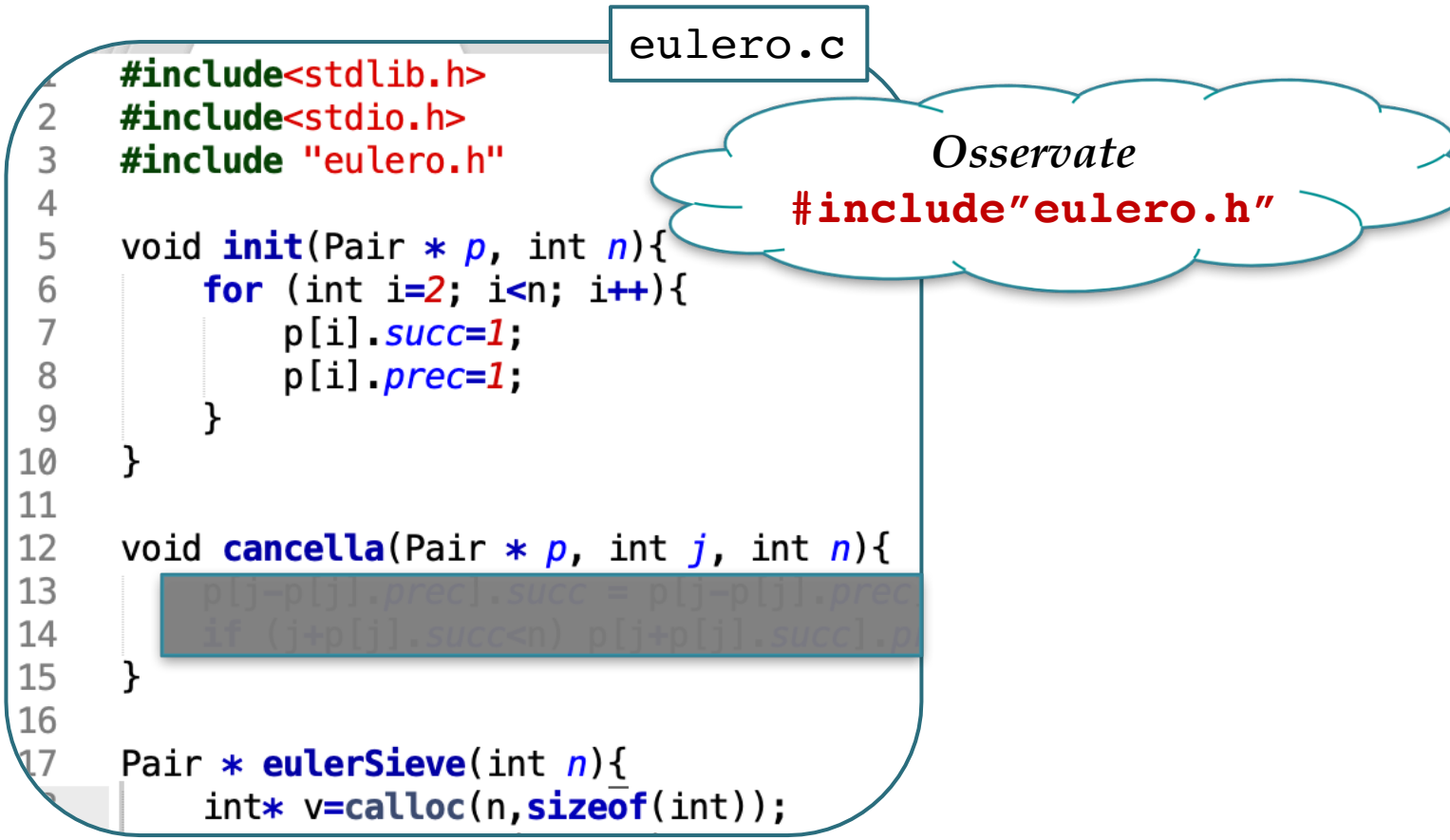
Usualmente le definizioni dei tipi e i prototipi di alcune funzioni relative a una struttura dati vengono messi in un file .h

eulero.h

```
1   #include<stdlib.h>
2   #include<stdio.h>
3
4   typedef
5      struct{
6         short unsigned int prec;
7         short unsigned int succ;
8       } Pair;
9
```

Suddivisione programmi C (2)

Usualmente il codice C di tali funzioni viene scritto in un file .c



Suddivisione programmi C (3)

Alla fine si compila tutto insieme:

```
gcc –o eulero eulero.c main.c
```

(gli `#include "eulero.h"` vanno messi ovunque si chiamino funzioni il cui prototipo è definite in `eulero.h`)

Ogni file dovrebbe **compilare separatamente** con il comando:

```
gcc –c eulero.c
```

```
gcc –c main.c
```

che producono file **non eseguibili** `eulero.o` e `main.o`.

Lezione 24

That's all Folks!

Grazie per l'attenzione…

…Domande?