Informatica Generale
Programmazione C

Ivano Salvo

Corso di Laurea in Matematica

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **12**, 19-20 maggio 2020

Privilegiati dello Smart Working

"I can't remember—do I work at home or do I live at work?"

Lezione 12a:

Problemi di riscaldamento sugli alberi binari

Relazione di sottoalbero

Abbiamo visto la relazione di prefisso tra alberi. Ora vediamo di calcolare la relazione **di sotto-albero**, definendola prima con equazioni ricorsive:

$$\mathsf{subTree}(t, r(L,R)) = \mathsf{equalsBT}(t, r(L,R)) \vee \mathsf{subTree}(t, L) \vee \mathsf{subTree}(t, R)$$

$$\mathsf{subTree}(\_, t) = \mathsf{true} \qquad \mathsf{subTree}(t, \_) = \mathsf{false}\ (\text{se } t \neq \_)$$

```
binTree subTree(binTree B1, binTree B2){
/* restituisce un pointer al sottoalbero
 * in B2, oppure NULL */
   int r2;
   binTree L2, R2;
   if (!B1) return B2;
   if (isEmptyBT(B2, &r2, &L2, &R2))
         return NULL;
   if (equalsBT(B1, B2)) return B2;
   return subTree(B1, L2) ||
               subTree(B1, R2);
}
```

Restituire la lista di nodi…

A volte può essere utile avere la lista di tutti (od alcuni) nodi dell'albero in un certo ordine. Vediamo alcuni esempi:

```
list preOrdToList(binTree B){
   if (B)
      return cons(B->info,
              append(preOrdtoList(B->left),
                 preOrdtoList(B->right)));
}
```

```
list inOrdToList(binTree B){
   if (B)
      return append(inOrdtoList(B->left),
         cons(B->info, inOrdtoList(B->right)));
}
```

```
list postOrdToList(binTree B){
   if (B)
      return append(inOrdtoList(B->left),
         addTail(B->info, inOrdtoList(B->right)));
}
```

E se volessimo **evitare l'append** e **addTail** fare solo inserzioni in testa? In tutti e 3 i casi.

Frontiera

La **frontiera** è la **lista delle foglie** di un albero.

```
list frontier(binTree B){
   if (!B) return NULL;
   if (isLeaf(B)) return cons(B->info, NULL);
   return append(frontier(B->left),
                 frontier(B->right));
}
```

```
int isLeaf(binTree B){
   if (!B) return 0;
   if (!B->left && !B->right)
      return 1;
   return 0;
}
```

E se volessimo evitare l'append e fare solo inserzioni in testa? (**Esercizio istruttivo**). Il tutto da ripetere con tutti gli ordini di scansioni dei nodi.

Lezione 12b:

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.

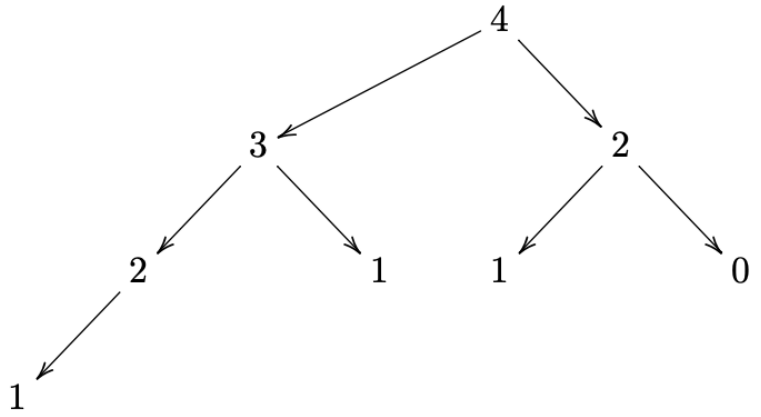
Conoscendo la radice, e **supponendo che essa sia unica**, la visita **inorder ci permette di separare la visita del sottoalbero sinistro dalla visita del sottoalbero destro**.

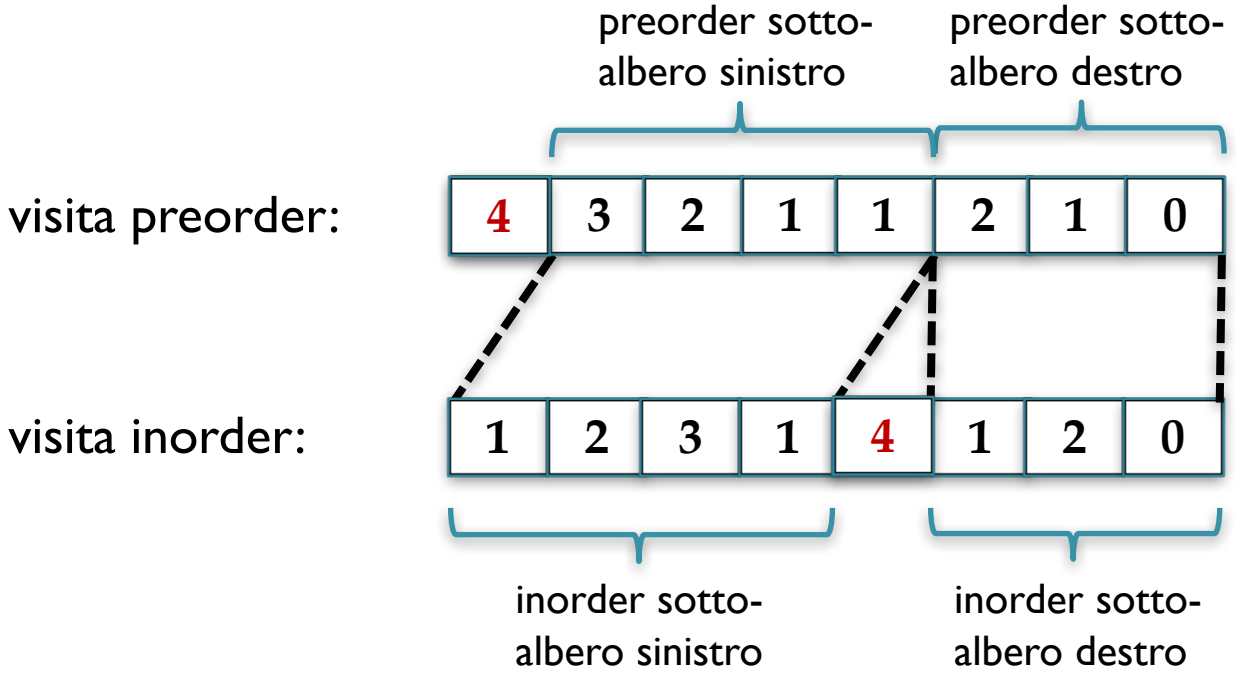
E poi di ricostruirle nella visita preorder (è sufficiente la cardinalità dei sottoalberi) e quindi chiamarsi ricorsivamente.

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 = visitToTree(preO+1, inO, p);
   B->right= visitToTree(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;
}
```

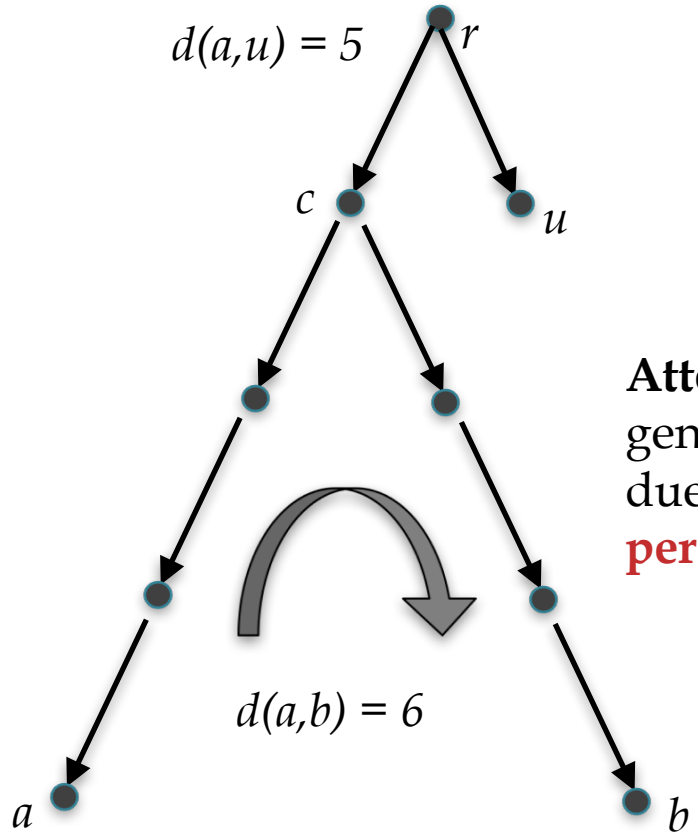
"Ricostruzione" di un albero binario

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 B1, int* p){
   int r, p1, p2; /* profond. Sottoalb.*/
   binTree L, R;
   if (isEmptyBT(B, &r, &L, &R)){
            *p=-1;
            return -1;
      }
   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);
}
```

Lezione 12c:

Funzioni che generano o modificano alberi binari

Funzioni che modificano/creano alberi

Non abbiamo visto finora funzioni che modificano/creano alberi. Esattamente come nel caso delle liste, **possiamo sempre scrivere queste funzioni in versione Fun** (creando un nuovo albero come risultato) oppure **modificando l'albero in input**.

Esempio: **albero specchiato**.

$$\text{mirror}(r(L,R)) = r(\text{mirror}(R), \text{mirror}(L)) \qquad \text{mirror}(\_) = \_$$

```
binTree mirrorFun(binTree B){
   int r;
   binTree L, R;
   if (isEmptyBT(B, &r, &L, &R))
      return NULL;
   return makeTree(r, mirrorTun(R),
                           mirrorFun(L));
}
```

Mirror che modifica l'albero

La versione che modifica l'albero è **leggermente più complessa** perché **devo evitare di perdere i puntatori**.

```
void mirror(binTree B){
   if (B)){
      mirror(B->left);
      mirror (B->right);
      binTree tmp = B->left;
      B->left = B->right;
      B->right = tmp;
   }
}
```

Somma sottoalberi

Finiamo con un classico esempio (analogo alla **somma successivi** per le liste): mettere in ogni nodo **la somma delle etichette del sottoalbero radicato** in quel nodo.

```
void sommaSottoAlberi(binTree B){
   int sl=0, sr=0;
   if (B)){
      if (B->left){
         sommaSottoAlberi(B->left);
         sl = B->left->info;
      }
      if (B->right){
         sommaSottoAlberi(B->right);
         sl = B->right->info;
      }
   B->info = B->info + rl + sl;
   }
}
```

Somma sottoalberi: versione Fun

Ci sono varie soluzioni, forse questa è la più intuitiva:

```
binTree sommaSottoAlberi(binTree B){
   binTree L, R;
   int r=0;

   if (B)){
      r += B->info;
      if (B->left){
            L = sommaSottoAlberi(B->left);
            r += L->info;
      } else L = NULL;
      if (B->right){
            R = sommaSottoAlberi(B->right);
            r+= R->info;
      } else R = NULL;
   return makeBinTree(r, L, R);
}
```

Lezione 12d:

Alcune soluzioni di Compiti passati

Immersione di Liste

1.4 Relazione di *Immersione* tra Liste (21 gennaio 2013)

Una sequenza di interi s_1 *è* immersa *in* s_2 *se gli elementi di* s_1 *occorrono ordinatamente in* s_2. *Ad esempio, la sequenza* $\langle 1, 7, 0 \rangle$ *è immersa nella sequenza* $\langle 4, 1, 8, 7, 9, 0 \rangle$.

1. *Dare una definizione induttiva della relazione di immersione tra sequenze, mediante equazioni ricorsive;*

$$
\begin{aligned}
\mathsf{immersa}(\langle\,\rangle, s) &= \mathsf{true} \\
\mathsf{immersa}(s, \langle\,\rangle) &= \mathsf{false} \quad (s \neq \langle\,\rangle) \\
\mathsf{immersa}(h_1 \cdot t_1, h_2 \cdot t_2) &= \begin{cases} \mathsf{immersa}(t_1, t_2) & \text{se } h_1 = h_2 \\ \mathsf{immersa}(h_1 \cdot t_1, t_2) & \text{se } h_1 \neq h_2 \end{cases}
\end{aligned}
$$

Soluzione

2. *Scrivere una funzione* ricorsiva `int immersaSimmRec(lista L, lista M)` *che restituisca* 1 *se la lista* `L` *è immersa nella lista* `M`, −1 *se la lista* `M` *è immersa nella lista* `L`, *e* 0 *altrimenti;*

```
int immersa(list L, list M){
   if (!L) return 1;
   if (!M) return 0;

   if (!L->val == M->val)
      return immersa(L->next, M->next);
   return immersa(L, M->next);
}

int immersaSimmRec(list L, list M){
   int LinM = immersa(L, M);
   int MinL = immersa(M, L);
   if ((LinM && MinL) ||
       (! LinM && ! MinL)) return 0;
   if (LinM) return 1;
   return -1;
}
```

Forse la traccia doveva essere più precisa perché può accadere che L ed M siano reciprocamente immerse una nell'altra.

Il problema del mescolamento

1.5 Relazione di *Shuffle* tra Liste (24 settembre 2015)

Scrivere una funzione C *di prototipo:* `int shuffle(lista L, lista M, lista N);` *che restituisca 1 se la lista puntata dal parametro* `N` *contiene l'unione degli elementi delle liste puntate da* `L` *ed* `M`*, collocati in modo tale che, nella lista puntata da* `N`*, la sequenza di elementi di ciascuna delle altre due liste mantenga lo stesso ordine.*

Ad esempio, se L fosse la lista $\langle 1,2,3\rangle$ ed M la lista $\langle 4,5\rangle$, la funzione shuffle *deve restituire 1 se N fosse la lista $\langle 1,4,5,2,3\rangle$, mentre deve restituire 0 se N fosse la lista $\langle 1,4,5,3,2\rangle$.*

Nota*: Si assuma che le liste L ed M non abbiano elementi in comune.*

$$
\begin{aligned}
\mathsf{shuffle}(\langle\,\rangle, s, s) &= \mathsf{true} \\
\mathsf{shuffle}(s, \langle\,\rangle, s) &= \mathsf{true} \\
\mathsf{shuffle}(s, t, \langle\,\rangle) &= \mathsf{false} \\
\mathsf{shuffle}(h \cdot t, m, h \cdot u) &= \mathsf{shuffle}(t, m, u) \\
\mathsf{shuffle}(l, h \cdot t, h \cdot u) &= \mathsf{shuffle}(l, t, u) \\
\mathsf{shuffle}(h_1 \cdot t_1, h_2 \cdot t_2, h \cdot u) &= \mathsf{false} \quad \text{se } h_1, h_2 \neq h
\end{aligned}
$$

Il problema del mescolamento

```
int shuffle(list L, list M, list N){
   if (!L) return uguali(M, N);
   if (!M) return uguali(L, N);
   if (!N) return 0;

   if (!L->val == N->val)
      return shuffle(L->next, M, N->next);
   if (!L->val == N->val)
      return shuffle(L, M->next, N->next)
   return 0;
}
```

Esercizio: Se le liste di ingresso hanno elementi in comuni la cosa si fa più complicata, perchè non è chiaro quale lista "mandare avanti". Ad esempio, considerate questo esempio: $\langle 3, 1 \rangle$, $\langle 3, 2 \rangle$ e $\langle 3, 2, 3, 1 \rangle$. Il programma scritto sopra, manda avanti la prima lista, ma poi non trova nessuna lista che comincia per 2. Sarebbe stato corretto in questo esempio mandare avanti la seconda!

Scrivete un programma che funziona correttamente nel caso di elementi comuni (occorre sdoppiare le computazioni come si stesse visitando un albero binario, e rispondere 1 non appena si trova un cammino che risponde 1, mentre si risponde 0 solo dopo aver verificato che *tutti* i cammini rispondono 0).

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.*

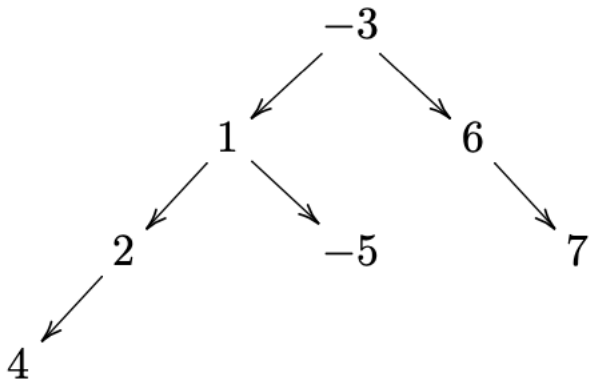


Figura 2: Albero binario nell'esempio

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.

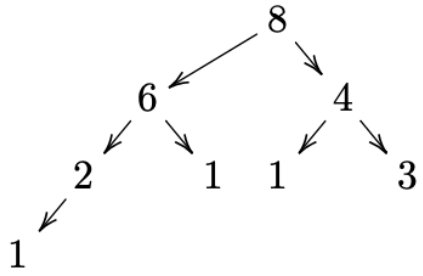
```
void livelloKAux(binTree B, int k, lista* M){
   int r;
   binTree L, R;
   if (!isEmptyBT(B, &r, &L, &R){
      if (!k) *M = cons(r, *M);
         else {
            livelloKAux(R, k-1, M);
            livelloKAux(L, k-1, M);
         }
}

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

Cammino fino a x

2.3 Lista dei Nodi di un Cammino Fino a x (17 giugno 2016)

Scrivere una funzione `C lista pathToX(binTree T, int x)` *che, ricevendo come parametri di input un albero binario di interi T ed un intero x, restituisce la lista vuota se x non appare come etichetta in T, altrimenti restituisce una lista con tutte le etichette che appaiono nel cammino dalla radice a x. Se x occorre più volte in T, restituire un cammino ad una qualsiasi delle occorrenze di x in T.*

ESEMPIO: *Dato l'albero in figura e l'intero 5, la funzione dovrà restituire lista vuota. Se viene passato alla funzione l'intero 3, la funzione dovrà restituire la lista $\langle 8, 4, 3 \rangle$. Se viene passato l'intero 1, la funzione può restituire una qualsiasi delle seguenti liste: $\langle 8, 6, 2, 1 \rangle$, $\langle 8, 6, 1 \rangle$, oppure $\langle 8, 4, 1 \rangle$.*



Cammino fino a x: soluzione

L'idea base è quella di visitare l'albero in profondità (**in qualsiasi ordine** visto che la specifica richiede di **ritornare un qualsiasi cammino**)

La funzione ricorsivamente può ritornare `NULL` se *x* non viene trovato oppure la lista del sotto-cammino (a cui rientrando attaccheremo in testa i nodi tornando alla radice).

Quando capiamo di aver trovato x, sospendiamo le altre eventuali chiamate ricorsive su altri sotto-alberi.

```
list pathToX(binTreeList B){
   int r;
   binTree L, R;
   list P;

   if (!isEmptyBT(B, &r, &L, &R) return NULL;
   if (r==x) return cons(x, NULL);
   P = pathToX(L); if (P) return cons(r, P);
   P = pathToX(R); if (P) return cons(r, P);
   return NULL;
}
```

Lezione 12
That's all Folks

...Domande?