Informatica Generale
Programmazione C

Ivano Salvo

Corso di Laurea in Matematica

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **11**, 12-13 maggio 2020

Banksy teneramente aperto al futuro

Lezione 11a:

Alberi binari (radicati)

Alberi: visione algebrica

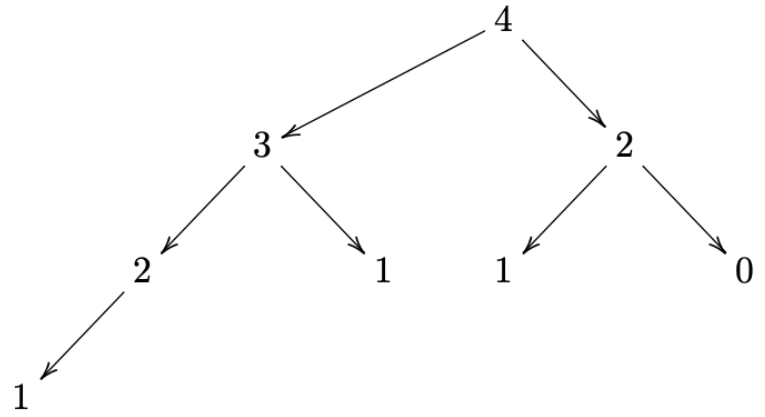
Dato un insieme (o un tipo) A, possiamo considerare il tipo degli **alberi binari *finiti*** (ma arbitrariamente grandi) con **etichette** in A, $BinTree\langle A\rangle$.

Tale insieme può essere definito **induttivamente** come segue:

- $\# \in BinTree\langle A\rangle$
- $r \in A$ e $L, R \in BinTree\langle A\rangle$ allora $r(L, R) \in BinTree\langle A\rangle$

e **(,)** sono detti **costruttori**, in quanto permettono di costruire ogni albero su A. Per indicare un albero possiamo sempre usare la notazione a parentesi.

L'albero in figura può essere scritto come: 4(3(2(1,#),1),2(1,0) dove abbiamo convenuto di abbreviare le **foglie** scrivendo 1 e 0 per 1(#, #) e 0(#, #).

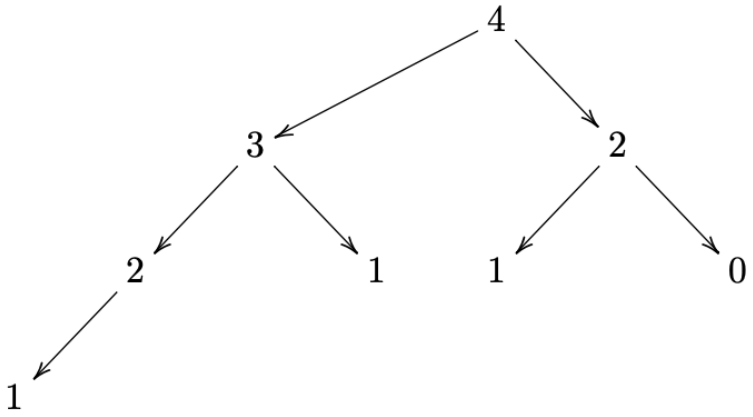


Alberi: definizione alternativa

Ci sono molte definizioni alternative di albero su un insieme A di etichette. Ad esempio, potremo definire $BinTree'\langle A\rangle$ come segue:

- $a \in A$ allora $leaf(a) \in BinTree'\langle A\rangle$ (**foglia**)
- $r \in A$ e $L, R \in BinTree'\langle A\rangle$ allora $r(L, R) \in BinTree'\langle A\rangle$

Cosa cambierebbe? L'albero sotto in figura è un $BinTree'\langle int\rangle$?
E nella rappresentazione in memoria (slides successive)?



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**).

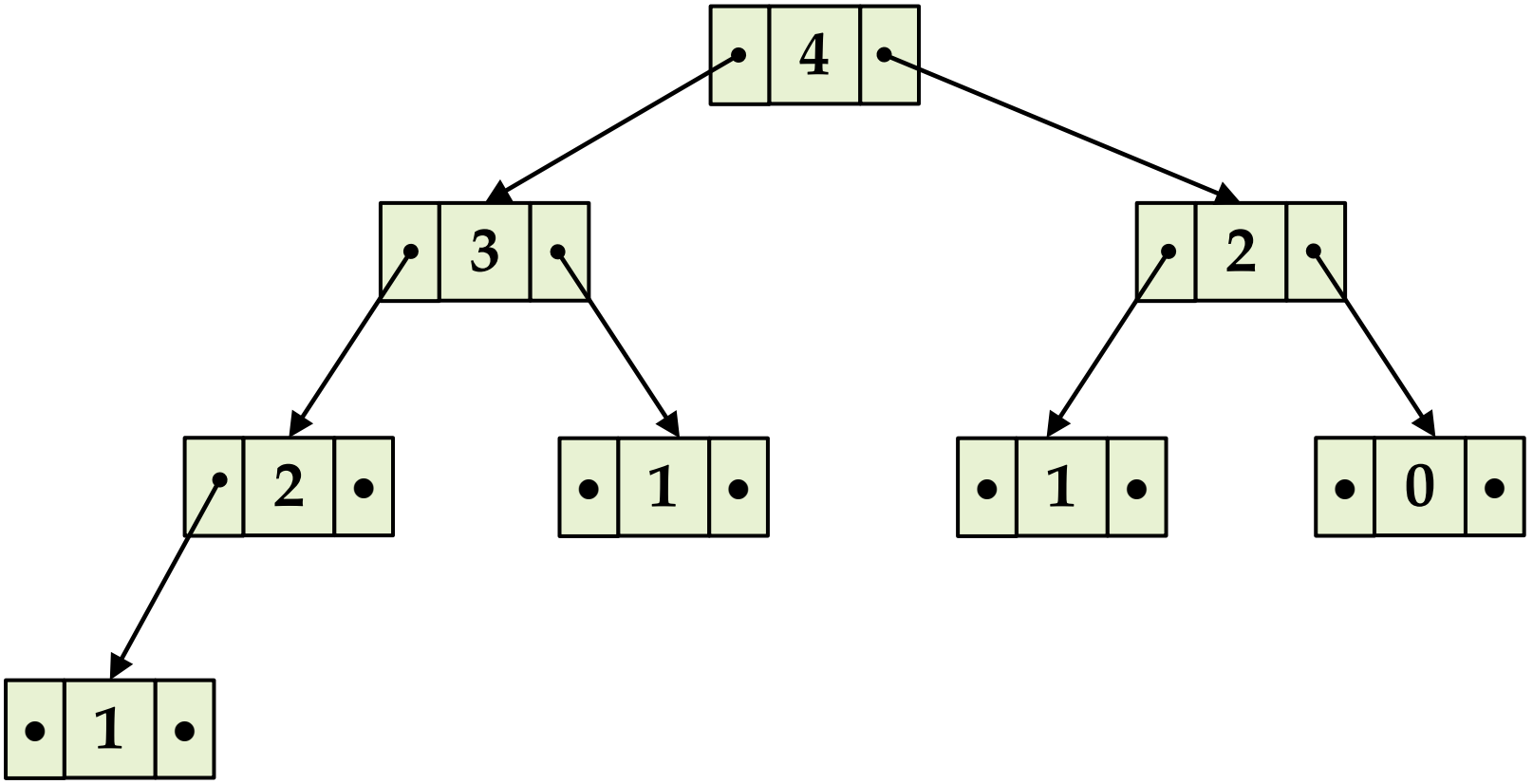
Come nel caso delle liste, useremo una struct per memorizzare un nodo dell'albero che consiste 1) di un campo **informazione** e 2) due campi puntatore ai **sottoalberi** (e anche qui i puntatori vengono usati per le definizioni ricorsive di tipo).

```
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`)



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

Come nel caso delle liste, per **definire funzioni per ricorsione** sugli alberi è necessario accedere alle componenti: radice, sottoalbero destro e sinistro.

Vediamo alcune funzioni base sugli alberi definite con equazioni ricorsive (**numero nodi**, **peso**, **profondità**, **uguaglianza**) :

$$\begin{aligned} \mathsf{nodes}(\sharp) &= 0 \\ \mathsf{nodes}(r(L,R)) &= 1 + \mathsf{nodes}(L) + \mathsf{nodes}(R) \end{aligned}$$

$$\begin{aligned} \mathsf{weight}(\sharp) &= 0 \\ \mathsf{weight}(r(L,R)) &= r + \mathsf{weight}(L) + \mathsf{weight}(R) \end{aligned}$$

$$\begin{aligned} \mathsf{depth}(\sharp) &= -1 \\ \mathsf{depth}(r(L,R)) &= 1 + \max(\mathsf{depth}(L), \mathsf{depth}(R)) \end{aligned}$$

$$\begin{aligned} \mathsf{equals}(\sharp,\sharp) &= \mathsf{true} \\ \mathsf{equals}(\sharp, r(L,R)) &= \mathsf{false} = \mathsf{equals}(r(L,R),\sharp) \\ \mathsf{equals}(r_1(L_1,R_1), r_2(L_2,R_2)) &= r_1 = r_2 \wedge \mathsf{equals}(L_1,L_2) \wedge \mathsf{equals}(R_1,R_2) \end{aligned}$$

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 delle funzioni viste sopra.

```
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 11b:

Alberi binari: visite e stampe

Visite in profondità

La maggior parte delle funzioni su alberi si calcolano, di fatto, con una visita. Sono visite post-order tutte le funzioni viste finora: raccolgono e combinano il risultato al rientro delle visite.

```
void postOrder(binTree B){
   if (B){
      postOrder(B->left);
      postOrder(B->rigth);
      printf(" %d", B->info);
   }
}
```

```
void preOrder(binTree B){
   if (B){
      printf(" %d", B->info);
      preOrder(B->left);
      preOrder(B->rigth);
   }
}
```

```
void inOrder(binTree B){
   if (B){
      inOrder(B->left);
      printf(" %d", B->info);
      inOrder(B->rigth);
   }
}
```

Stampa a parentesi

Possiamo usare le visite per fare stampe più significative che possano mettere in evidenza la struttura dell'albero. Vediamone due: la stampa a parentesi $r(L, R)$ e la stampa indentata.

```
void printTreeP(binTree B){
   if (B){
      if (isLeaf(B)) printf("%d", B->info);
         else {
            printf("%d(", B->info);
            preTreeP(B->left);
            printf(", ");
            preTreeP(B->rigth);
            printf(")");
         } /* end else */
   } else printf("#");
}
```

Stampa indentata

La **stampa indentata** (qui in versione pre-order) stampa i nodi spostati a destra **a seconda del livello**. Il livello (0 la radice, 1 i suoi figli, etc.) è un'informazione che è conveniente calcolare "in avanti". Quindi il modo migliore è **usare un parametro ausiliario**.

```
void printTreeIaux(binTree B, int l){
   if (B){
         /* stampiamo l*k spazi */
      printSpaces(l);
      printf("%d\n", B->info);
      printTreeIaux(B->left, l+1);
      printTreeIaux(B->ritht, l+1);
   } /* end if */
}

void printTreeI(binTree B){
   printTreeIAux(B, 0);
}
```

```
4
    3
        2
            1
        1
    2
        1
        0
```

Visita per Livelli

Fa eccezione la visita per livelli: siccome occorre esplorare i nodi dell'albero **in un ordine che `non rispetta' la struttura ricorsiva degli alberi**, è necessario aiutarsi con una struttura dati.

Inoltre, è unico dei pochi programmi sugli alberi che risulta essere **risolto più efficacemente da una procedura iterativa**.

```
void levels(binTree B){
   binTree C;
   if (!B) return;
   queue Q = emptyQueue();
   enqueue(Q, B);
   while (!isEmptyQ(Q)){
      /* INV: nodes(Q)= nodes(B)\nodi già espl.
       * TERM: |nodes(Q)| */
      C = dequeue(Q);
      printf(" %d", C->info);
      if (C->left) enqueue(Q, C->left);
      if (C->right) enqueue(Q, C->right);
   }
}
```

nodes(Q) è l'insieme dei nodi raggiungibili dalle radici dei sottoalberi in Q

Lezione 11a:

Il problema del bilanciamento

Alberi bilanciati

Definizione 1.2 Un albero è *bilanciato in altezza* se:

1. è l'albero vuoto;
2. i sottoalberi destro e sinistro sono bilanciati in altezza e la differenza delle loro altezze differisce al più di 1.

Definizione 1.3 Un albero è *bilanciato nel numero dei nodi* se:

1. è l'albero vuoto;
2. i sottoalberi destro e sinistro sono bilanciati nel numero dei nodi e la differenza dei numeri dei nodi differisce al più di 1.

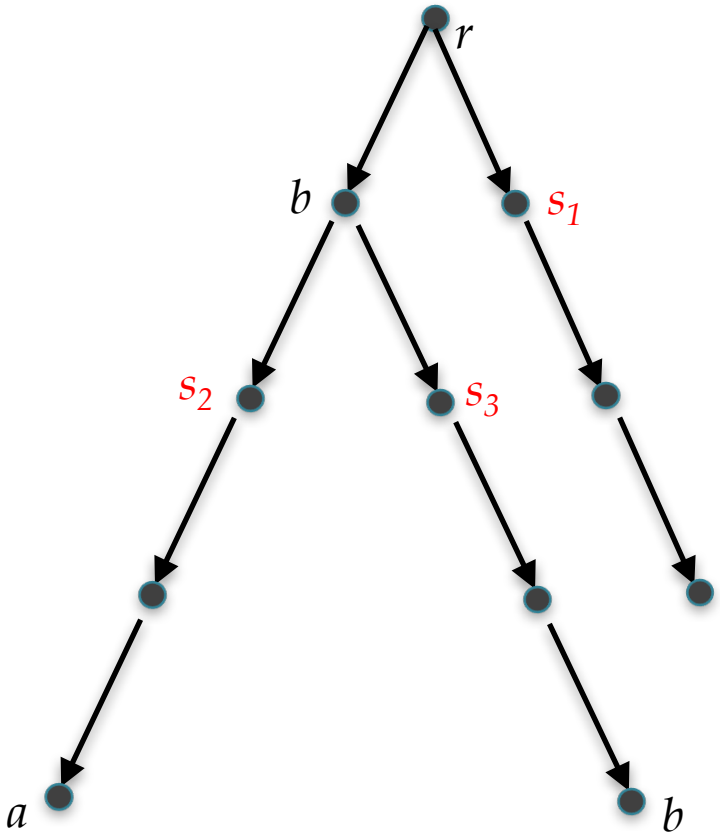
Esercizio: Dimostrare che un albero bilanciato nel numero dei nodi è bilanciato anche in altezza.

Trovare un controesempio all'implicazione opposta, cioè trovare un albero bilanciato in altezza, ma non nel numero dei nodi.

Alberi bilanciati: esempio

Attenzione: il bilanciamento deve valere **su tutti i sottoalberi**.

Ad esempio il seguente albero **non è bilanciato**:



I nodi rossi, s_1, s_2 e s_3 sono radici di sotto-alberi non bilanciati.

Mentre alla radice *r* le profondità destre e sinistre differiscono di 1.

Verifica bilanciamento "ingenuo"

Usando le funzioni `int depth(binTree)` e `int nodes(binTree)` possiamo facilmente verificare il bilanciamento.

```
int depthBalanced(binTree B){
   int r;
   binTree L, R;
   if (isEmpty(B, &r, &L, &R)) return 1;
   return depthBalanced(L) && depthBalanced(R)
          && abs(depth(L) - depth(R)) <=1;
}
```

```
int nodeBalanced(binTree B){
   int r;
   binTree L, R;
   if (isEmpty(B, &r, &L, &R)) return 1;
   return nodeBalanced(L) && nodeBalanced(R)
          && abs(nodes(L) - nodes(R)) <=1;
}
```

Queste funzioni ricalcolano molte volte depth e nodes sugli stessi sotto-alberi e sono quadratiche

Bilanciamento in un'unica visita

Al solito, la cura consiste nel **memorizzare più informazione** durante la computazione. Questa è una **tipica computazione all'indietro** (post-order) e quindi possiamo usare:

1. Il valore di ritorno;
2. Un parametro passato per indirizzo.

Nel seguito vedremo due soluzioni. Nel bilanciamento nel numero dei nodi sfrutteremo il fatto che il numero dei nodi è non negativo e **useremo un valore negativo per codificare l'informazione che il sotto-albero non è bilanciato** (si può fare anche per la profondità, ricordandosi però che la profondità può essere -1).

Nel bilanciamento in altezza, useremo **un parametro passato per indirizzo** per condividere l'altezza calcolata nei sottoalberi tra diverse chiamate ricorsive.

Bilanciamento nel numero dei nodi

```
int nodeBalancedAux(binTree B){
   int r, nL, nR;
   binTree L, R;
   if (isEmpty(B, &r, &L, &R))
      return 0;
   nL = nodeBalancedAux(L);
   if (nL < 0) return -1;
   nR = depthBalancedAux(R);
   if (nR < 0) return -1;
   if (abs(nL – nR) <= 1)
      return 1 + max(dL, dR);
   return -1;
}

int nodeBalanced(binTree B){
   int n = nodeBalancedAux(B);
   if (n>0) return 1; else return 0;
}
```

Si poteva fare un unico if, ma così evito di scendere sull'albero destro quando il sinistro è già sbilanciato

Anche se il prototipo è lo stesso, uso una funzione ausiliaria perché il valore di ritorno è interpretato in modo diverso!

Bilanciamento in altezza

```
int depthBalancedAux(binTree B, int *d){
   int r, dL, dR;
   binTree L, R;
   if (isEmpty(B, &r, &L, &R))
      return 1;
   if (depthBalancedAux(L, &dL)
      && depthBalancedAux(R, &dR)
      && abs(dL – dR) < 1){
         *d = 1 + max(dL, dR);
         return 1;
         }
   return 0;
}

int depthBalanced(binTree B){
   int d;
   return depthBalancedAux(B, &d);
}
```

Qui sfrutto la regola di computazione di `&&`

`d` *non serve in questa funzione. Molti linguaggi permettono parametri anonimi.*

Lezione 11c:

Alcune soluzioni di Compiti passati

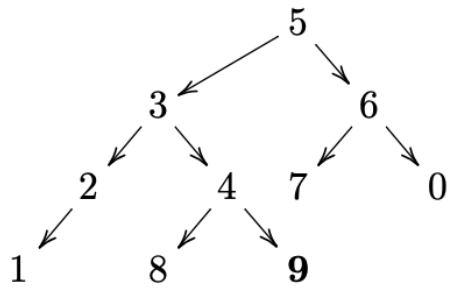
Selezionare un sottoalbero

2.1 Selezione di un Cammino in un Albero (1 luglio 2013)

Scrivere una funzione `C tree seleziona(binTree B, list L)` *che riceve come parametri di ingresso un albero binario* `B` *di interi e una lista di interi contenente solo i valori 0 ed 1. La funzione deve interpretare la lista* `L` *come la rappresentazione di un cammino (0 significa vai al sottoalbero sinistro, 1 significa vai al sottoalbero destro) e ritornare un puntatore al sottoalbero che si trova alla fine del cammino rappresentato da* `L`.

Se il cammino rappresentato da `L` *non esiste, la funzione deve ritornare* `NULL`.

ESEMPIO:



Ricevuto in input l'albero in figura, e la lista $\langle 0, 1, 1 \rangle$ *la funzione deve tornare un puntatore al sottoalbero radicato in 9. Dovrebbe tornare* `NULL`, *ricevendo in ingresso la lista* $\langle 0, 0, 1 \rangle$, *perchè il sottoalbero radicato in 2 non ha figlio destro.*

Soluzione

```
binTree seleziona(binTree B, list L){
   if (!B) return NULL;
   if (!L) return B;

   if (!L->val)
      return seleziona(B->left, L->next);
   return seleziona(B->right, L->next);
}
```

Relazione di Prefisso tra Alberi

2.4 Relazione di Prefisso tra Alberi (15 luglio 2014)

Un albero binario di interi T_1 *è* prefisso *di* T_2 *se:*

1. T_1 *è l'albero vuoto;*
2. *oppure* T_1 *e* T_2 *hanno la stessa radice e:*
   - *(a)* $\mathsf{left}(T_1)$ *è prefisso di* $\mathsf{left}(T_2)$*;*
   - *(b)* **e** $\mathsf{right}(T_1)$ *è prefisso di* $\mathsf{right}(T_2)$

   dove $\mathsf{left}(T)$ *e* $\mathsf{right}(T)$ *sono rispettivamente il sottoalbero sinistro e destro dell'albero* T*.*

Scrivere una funzione C *che prende in input una lista di alberi e ritorna 1 se è ordinata rispetto alla nozione di prefisso tra alberi e 0 altrimenti.*

Dare la definizione del tipo di dato `binTreeList` *e organizzare opportunamente il codice in funzioni.*

Soluzione

A questo punto, occorre definire il tipo `binTreeList` che è del tutto analogo al tipo `list` delle liste di interi, con la differenza che il campo `val` sarà di tipo `binTree` invece che `int`.

```
typedef struct BTL{
     binTree val;
     struct BTL * next;
} binTreeListNode;

typedef binTreeListNode* binTreeList;
```

Cominciamo assiomatizziamo la relazione di prefixTree con equazioni ricorsive:

$$
\begin{aligned}
\mathsf{prefixTree}(\_, b) &= \mathsf{true} \\
\mathsf{prefixTree}(b, \_) &= \mathsf{false} \quad (b \neq \_) \\
\mathsf{prefixTree}(r_1(L_1, R_1), r_2(L_2, R_2)) &= r_1 = r_2 \wedge \mathsf{prefixTree}(\mathsf{L}_1, \mathsf{L}_2) \wedge \mathsf{prefixTree}(\mathsf{R}_1, \mathsf{R}_2)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{ordered}(\langle\,\rangle) &= \mathsf{true} \\
\mathsf{ordered}(\langle x \rangle) &= \mathsf{true} \\
\mathsf{ordered}(h_1 \cdot h_2 \cdot t) &= h_1 \preceq h_2 \wedge \mathsf{ordered}(h_2 \cdot t)
\end{aligned}
$$

Soluzione

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

int ordered(binTreeList L){
   if (!L || !L->next) return 1;
   return prefixBT(L->val, L->next->val)
      && ordered(L->next);
}
```

Lezione 11 That's all Folks

...Domande?