The Ultimate 10's C Programming Exercise Collection

corso di Informatica Generale
Ivano Salvo – Sapienza Università di Roma

1 Esercizi su Liste

1.1 Divisione di Liste (21 settembre 2016)

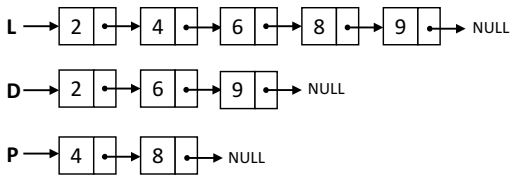
Considerare il problema di dividere una lista di interi in due liste, una contenente tutti gli elementi che occorrono in posizione pari e una contenente tutti gli elementi in posizione dispari. Scrivere due funzioni in linguaggio C*:*

1. *La prima,* `void dividiFun(lista L, lista* D, lista *P)`, alloca *nuova memoria e copia gli elementi di posto dispari nella lista* `D` *e gli elementi di posto pari nella lista* `P`*.*

2. *La seconda,* `void dividi(lista L, lista* D, lista *P)` *modifica la lista originaria* `L` *modificando adeguatamente i puntatori restituendo nella variabile* `P` *un puntatore alla testa della lista degli elementi di posto pari e nella variabile* `D` *un puntatore alla testa della lista degli elementi di posto dispari.*

Esempio: *Entrambe le funzioni, se la lista di ingresso L fosse $\langle 2, 4, 6, 8, 9\rangle$, dovrebbero costruire le liste $D = \langle 2, 6, 9\rangle$ e $P = \langle 4, 8\rangle$. Tuttavia, i loro effetti in memoria saranno molto diversi, come esemplificato in Fig. 1.*

a) risultato in memoria dopo l'esecuzione di `divideFun(L, &D, &P)`



b) risultato in memoria dopo l'esecuzione di `divide(L, &D, &P)`

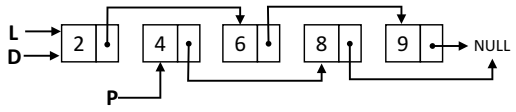


Figura 1: Effetti in memoria delle due funzioni.

Punto 1. Esistono innumerevoli soluzioni. Ad esempio scrivere una funzione ausiliaria con un parametro ausiliario per ricordarci se il prossimo elemento va messo nella lista dei dispari o in quella dei pari. Qui, ve ne propongo due di carine: la prima usa due funzioni *mutuamente ricorsive*. La prima funzione aggiunge alla lista dei dispari e chiama quella che aggiunge alla lista dei pari, che si comporta in modo simmetrico.

```
void dividi2Fun(list L, list *D, list *P);

void dividiFun(list L, list *D, list*P){
  if (L){
     dividi2Fun(L->next, D, P);
     *D = cons(L->val, *D);
     }
}

void dividi2Fun(list L, list *D, list*P){
  if (L){
     dividiFun(L->next, D, P);
     *P = cons(L->val, *P);
     }
}
```

A ben vedere, la mutua ricorsione, benchè elegante, risulta una finezza inutile. Le due funzioni fanno la stessa cosa, ma invertendo il ruolo dei paramentri. Beh, allora tanto vale scrivere una funzione e invertire i parametri ad ogni chiamata ricorsiva!

```
void dividiSwapFun(list L, list *D, list *P){
  if (L){
     dividiSwapFun(L->next, P, D);
     *D = cons(L->val, *D);
     }
 }
```

Punto 2. Un po' come nel caso della `reverse`, la funzione *in place* è decisamente più critica, perchè occorre assicurarsi di non distruggere la struttura della lista prima di separarla correttamente. Comunque, in analogia con la soluzione al punto 1, basta dividerla e sistemare i puntatori `next` al ritorno delle chiamate ricorsive.

Si può partire dalle 2 soluzioni mostrate prima, ma per far vedere che la fantasia non ha limiti, usiamo un ulteriore approccio, che scansiona la lista a 2 a 2. In questo caso, devo tuttavia prestare attenzione che ci sono due casi base (lista vuota e lista con un solo elemento, che forzatamente andrà a finire tra i dispari!).

```
void dividiInPlace(list L, list *D, list *P){
  if (!L || !L->next){
     *D = L;
     *P = NULL;
     } else {
          dividiInPlace(L->next->next, D, P);
          L->next->next = *P;
          *P=L->next;
          L->next = *D;
          *D=L;
    }
}
```

1.2 Somma Precedenti (2 luglio 2012)

Si consideri il problema di prendere in input una lista di interi e calcolare la lista in cui tutti gli elementi sono sostituiti con la somma dei precedenti (elemento incluso). Ad esempio, presa in input la lista $\langle 1, 7, 11\rangle$ *si vuole ottenere in output la lista* $\langle 1, 8, 19\rangle$*. Procedere come segue:*

1. Specificare la funzione mediante equazioni ricorsive su sequenze;

Trattasi del tipico caso in cui occorre trasmettere al chiamato dei valori calcolati dal chiamante. In questo modo, posso ottenere il risultato in un'unica scansione della lista. Come è noto, ciò si può fare utilizzando un parametro ausiliario. In questo caso, il nostro parametro manterrà l'informazione relativa alla somma degli elementi incontrati fino ad ora.

$$\begin{aligned} \mathsf{sommaPrec}(s) &= \mathsf{sommaPrecAux}(s, 0) \\ \mathsf{sommaPrecAux}(\langle\,\rangle, n) &= \langle\,\rangle \\ \mathsf{sommaPrecAux}(h \cdot t, n) &= h + n \cdot \mathsf{sommaPrecAux}(t, h + n) \end{aligned}$$

2. Scrivere una funzione `sommaPrecFun` *che alloca una* nuova *lista;*

Al solito, è sufficiente tradurre le equazioni ricorsive interpretando $\cdot$ con `cons`, $\langle\,\rangle$ con `NULL`, etc.

```
lista sommaPrecAux(lista L, int n){
   if (!L) return NULL;
   return cons(L->val + n,
               sommaPrecAux(L->next, L->val + n)
               );
}

lista sommaPrecFun(lista L){
    return sommaPrecAux(L, 0);
}
```

3. Scrivere una funzione `sommaPrecRec` ricorsiva *che* modifica *la lista di ingresso (*Nota Bene: *non è ammesso l'uso di nessun costrutto iterativo nella soluzione);*

Proponiamo due soluzioni. La prima scimmiotta opportunamente la funzione `sommaPrecFun` scritta sopra. Osservare che la funzione può essere definita di tipo `void` perché il pointer di inizio lista non cambia e noi ci limitiamo a modificare le informazioni contenute nei nodi della lista.

```
void sommaPrecRecAux(lista L, int n){
   if (L) {
        L->val = L->val + n;
        sommaPrecRecAux(L->next, L->val);
        }

void sommaPrecRec(lista L){
    sommaPrecRecAux(L, 0);
}
```

Possiamo fare a meno del parametro ausiliario, accumulando le somme parziali nella lista stessa, come segue. Osservate che andando a modificare il *prossimo* elemento della lista, dobbiamo considerare due casi base (lista vuota e lista con un elemento).

```
void sommaPrecRecAux(lista L){
   if (L && L->next) {
        L->next->val = L->next->val + L->val;
        sommaPrecRecAux(L->next);
        }
   }
```

4.Scrivere una funzione C `sommaPrecIter` iterativa *che* modifica *la lista di ingresso.*

Scriviamo due soluzioni, ciascuna delle quali è il corrispondente iterativo delle due funzioni ricorsive scritte sopra:

```
void sommaPrecIter1(lista L){
  int s=0;
  while (L){
     L->val = L->val + s;
     s = L->val;
     L = L->next;
  }
}
```

```
void sommaPrecIter2(lista L){

   while (L && L->next){
      L->next->val += L->val;
      L = L->next;
   }
}
```

1.3 Differenza Simmetrica tra Liste (12 novembre 2012)

Definiamo differenza simmetrica *tra due liste di interi L_1 ed L_2, una qualsiasi lista L che contiene tutti gli elementi che compaiono in L_1 ma non in L_2 e tutti gli elementi che compaiono in L_2 ma non in L_1.*

Ad esempio, se $L_1 = \langle 1, 15, 8, 7\rangle$ e $L_2 = \langle 7, 1, 5, 11\rangle$, avremo che la differenza simmetrica di L_1 ed L_2 sarà una qualsiasi L lista che contiene gli interi $15, 8, 5, 11$. Supponendo per semplicità che non ci siano ripetizioni in ciascuna lista:

1. scrivere una funzione C `lista diffSim(lista L1, lista L2)` *che calcoli la differenza simmetrica tra due liste.*

2. scrivere una funzione C `lista diffSimOrd(lista L1, lista L2)` *che risolve lo stesso problema, traendo vantaggio dal fatto che le liste di ingresso* `L1` *ed* `L2` *siano ordinate in modo crescente. Discutere brevemente la complessità delle funzioni scritte nei due casi.*

Punto 1: La differenza simmetrica tra liste può essere facilmente definita dalla seguente equazione:

$$\mathsf{diffSimm}(L_1, L_2) \quad = \quad \mathsf{append}(\mathsf{diff}(L_1, L_2), \mathsf{diff}(L_2, L_1))$$

Scrivendo un programma che calcola la differenza tra liste (vedi dispensa **D8**, sezione **3.6**), che genera *una nuova lista* senza modificare i parametri (funzione `diffFun`), la differenza simmetrica come:

```
diffSimm(lista L1, lista L2){
   return appendRec(diffFun(L1,L2), diffFun(L2,L1));
}
```

Ovviamente il giochino non funziona se le liste L_1 ed L_2 vengono modificate durante il calcolo. Riflettere sul perché sia opportuno chiamare `appendRec` invece di `appendFun`.

Punto 2: Nel caso in cui le due liste di ingresso siano ordinate, è possibile scrivere una funzione che segue la stessa logica della funzione merge, scorrendo parallelamente le due liste ed evitando di inserire gli elementi comuni. Anche qui, conviene specificare con equazioni ricorsive la funzione diffSimmOrd come segue:

$$\begin{aligned} \mathsf{diffSimmOrd}(s, \langle\,\rangle) &= \mathsf{diffSimmOrd}(\langle\,\rangle, s) = s \\ \mathsf{diffSimmOrd}(h_1 \cdot t_1, h_2 \cdot t_2) &= \begin{cases} \mathsf{diffSimmOrd}(t_1, t_2) & \text{se } h_1 = h_2 \\ h_1 \cdot \mathsf{diffSimmOrd}(t_1, h_2 \cdot t_2) & \text{se } h_1 < h_2 \\ h_2 \cdot \mathsf{diffSimmOrd}(h_1 \cdot t_1, t_2) & \text{se } h_1 > h_2 \end{cases} \end{aligned}$$

Queste equazioni possono facilmente essere tradotte in un programma C come segue:

```
lista diffSimmOrd(lista L1, lista L2){
   if (!L1) return copia(L2);
   if (!L2) return copia(L1);
   if (L1->val==L2->val)
          return diffSimOrd(L1->next, L2->next);
   if (L1->val<L2->val)
          return cons(L1->val, diffSimOrd(L1->next, L2));
   return cons(L2->val, diffSimOrd(L1, L2->next));
}
```

Complessità: La prima funzione ha chiaramente complessità $|L_1| \cdot |L_2|$, dove $|L|$ è la lunghezza della lista L. Infatti, per verificare se un certo elemento di L_1 sta nella differenza tra L_1 ed L_2 occorre, alla peggio, scorrere tutta la lista L_2.

Nel caso ordinato, è chiaro che ciascun elemento di una lista viene ispezionato una sola volta e quindi la complessità di `diffSimmOrd` è nell'ordine di $|L_1| + |L_2|$. Alternativamente, è sufficiente osservare che $|L_1| + |L_2|$ decresce a ogni attivazione ricorsiva di *almeno* 1.

Esercizi: Perché `diffSimmOrd` potrebbe non funzionare correttamente in caso di ripetizioni di elementi? Fornire un controesempio e modificare opportunamente il codice per renderla corretta nel caso in cui le liste di ingresso abbiano ripetizioni.

Nel caso ordinato, è possibile scrivere una funzione che usa i nodi allocati per le liste originarie e sposta opportunamente i puntatori? Scrivere il programma e deallocare i nodi rimossi dalle due 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;

La funzione immersa può essere facilmente specificata con equazioni ricorsive come segue:

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

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;*

La specifica ricorsiva al punto precedente dà una facile implementazione della funzione `immersa`, da usare poi per implementare `immersaSimm`. Vediamo:

```
int immersa(lista L1, lista L2){
   if (!L1) return 1;
   if (!L2) return 0;
   if (L1->val==L2->val)
          return immersa(L1->next, L2->next);
   return immersa(L1, L2->next);
}

int immersaSimmRec(lista L, lista M){
    if (immersa(L, M)) return 1;
    if (immersa(M, L)) return -1;
    return 0;
}
```

3. scrivere una funzione iterativa `int immersaIt(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.*

Procediamo come nel punto 2, e limitiamoci a scrivere la procedura `immersaIt`. Sarà sufficiente scorrere le liste secondo la stessa logica del programma ricorsivo: la lista $L2$ viene sempre fatta avanzare, mentre la lista $L1$ avanza solo quando la sua testa è uguale a quella di $L2$. Il controllo su quale lista sia finita, ci dice se $L1$ sia immersa in $L2$ o meno.

```
int immersaIt(lista L1, lista L2){
   while (L1 && L2) {
      if (L1->val == L2->val) L1=L1->next;
      L2=L2->next;
      }
   if (!L1) return 1;
   return 0;
}
```

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.*

Come sapete, l'idea migliore è dimenticare per un attimo la sintassi del C e concentrarsi sul problema con le amiche equazioni ricorsive.

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

Da cui deriva, facilmente il seguente programma C ricorsivo:

```
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 (M->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).

1.6 Massimi Locali di una Lista (16 settembre 2013)

Un elemento di una sequenza di interi è massimo locale*, se è maggiore o uguale del suo predecessore e del suo successore (per il primo è sufficiente che sia maggiore del successore e per l'ultimo è sufficiente che sia maggiore del predecessore). Nella sequenza* $\langle \mathbf{4}, 3, 1, 4, \mathbf{6}, 2\rangle$ *abbiamo evidenziato in grassetto i massimi locali.*

1. Scrivere una funzione `C lista massimiLocali(lista L, int* n)` *che ricevendo come parametro di ingresso un puntatore* `L` *alla testa di una lista di interi, restituisca come risultato un puntatore alla testa di una nuova lista contenente tutti i massimi locali e carichi nel parametro* `n` *passato per indirizzo il numero di tali massimi locali. Nel nostro esempio, la lista risultato sarebbe* $\langle 4, 6\rangle$ *e il parametro* `n` *dovrebbe assumere il valore 2 al termine dell'esecuzione della funzione.*

Cominciamo con lo scrivere una funzione ricorsiva. Osservate che ci serve conoscere non solo il successivo dell'elemento corrente (accessibile via il puntatore `next`), ma anche il precedente (se esiste). Abbiamo bisogno di un parametro ausiliario. Abbiamo molte scelte, ma potremo passare un pointer. Se questo fosse `NULL`, significa che sono sul primo elemento della lista.

Onde evitare di scrivere costrutti condizionali troppo complicati, *conto* quanti elementi adiacenti sono minori dell'elemento corrente usando la variabile `m` (che potrà valere 0, 1, o 2).

```
lista maxLocAux(lista L, int* n, lista Prec){
   int m=0;
   if (!L) return NULL;
   if (!Prec || L->val > Prec->val) m++;
   if (!L->next || L->val > L->next->val) m++;
   lista M = maxLocAux(L->next, n, L);
   if (m==2){
         *n++;
         M = cons(L->val, M);
      }
   return M;
}

lista maxLoc(Lista L, int *n){
   return maxLocAux(L, n, NULL);
   }
```

2. Se al punto precedente si è data una soluzione ricorsiva, scrivere una versione iterativa della stessa funzione. Analogamente, se al punto precedente si è data una soluzione iterativa, scrivere una versione ricorsiva della stessa funzione.

Come usuale nei problemi su liste, possiamo al solito trarre vantaggio dalla versione ricorsiva per programmarci `maxLocIt` (purtroppo in questo caso la lista dei massimi locali esce rovesciata!):

```
lista maxLocIt(lista L, int* n){
   int m;
   *n=0;
   lista prec=NULL;
   lista res=NULL;
   while (!L){
      m=0;
      if (!prec || L->val>prec->val) m++;
      if (!L->next || L->val > L->next->val) m++;
      if (m==2) {
         *n++;
         res = cons(L->val, res);
      }
      prec=L;
      L=L->next;
      }
   return res;
}
```

1.7 Tipo Naturale con Liste (23 settembre 2014)

Rappresentare il tipo di dato numero naturale*, in modo che un numero naturale n sia rappresentato da una lista con n nodi. Scrivere il codice* C *corrispondente a:*

1. definizione del tipo di dato `naturale`*; 2. funzione successore; 3. funzione predecessore; 4. funzione somma; 5. funzione moltiplicazione; 6. funzione esponenziale.*

In questo esercizio è vietato usare costrutti iterativi (`while`*,* `for`*,* `do ... while`*).*

Dovendo usare liste la cui unica informazione interessante è la loro lunghezza, possiamo dare la seguente definizione (osservate che non c'è nessun campo informazione):

```
struct N {
   struct N *  pred;
 };

 typedef struct N * naturale;
```

A questo punto, è chiaro che il successore consiste nell'aggiungere un nodo (una specie di cons), il predecessore a togliere un nodo (restituendo per esempio la coda), e poi le altre funzioni semplicemente si possono ottenere come visto a inizio corso in TinyC, la somma iterando il successore (in realtà la somma può essere vista come l'append tra liste), il prodotto iterando la somma e, infine, l'esponenziale iterando la moltiplicazione. Fino a qui tutto semplice e sufficiente per portare a casa il massimo dei punti.

Tuttavia, volendo fare i sofisticati, saltano agli occhi due tristi alternative: 1) far generare a ogni funzione una nuova lista; 2) modificare la lista di ingresso. Il dramma di 1) è l'eccessiva occupazione di memoria, mentre il dramma di 2) è strani side-effects. Ad esempio, definendo `succ` semplicemente aggiungendo un nodo in testa e ritornando il pointer a questo, si avrà che dopo `naturale m = succ(p)`, modifiche a `p` potrebbero influenzare in modo "silente" e inaspettato il valore memorizzato dalla variabile `m`.

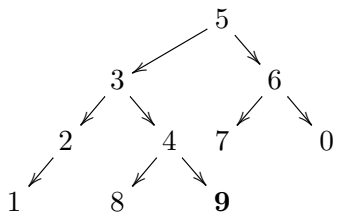
2 Esercizi su Alberi

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.*

Scriviamo direttamente il semplice codice `C` che ritengo autoesplicativo:

```
binTree seleziona(binTree B, list L){
   if (!L) return B;
   if (!B) return NULL;
   if (L->val == 0) return seleziona(B->left, L->next);
      else return seleziona(B->right, L->next);
}
```

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.*

Purtroppo, molti studenti, leggendo la parola *livello*, si sono lanciati a capofitto in una visita per livelli dell'albero, cosa anche corretta, ma non necessaria. La difficoltà di questa soluzione, oltre alla corretta implementazione della visita a livelli, consiste nel fatto che occorre marcare in qualche modo a quale livello appartiene il nodo dell'albero che stiamo visitando. Alcuni hanno dato soluzioni molto ingegnose,

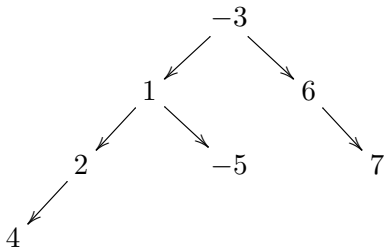


Figura 2: Albero binario nell'esempio

essenzialmente basate sull'idea che (durante l'esecuzione di una visita) nella coda necessaria alla visita per livelli compaiono solo nodi di due livelli successivi.

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

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

Lascio ai lettori, la facile traduzione in C. Tuttavia, gli studenti sofisticati potrebbero essere insoddisfatti di questa soluzione, perché append sulle liste "basic" non è costante, ma *lineare* nella lunghezza del primo argomento.

In effetti, sacrificando un po' di chiarezza, e con una funzione leggermente più complicata, è possibile costruire la lista risultato a colpi di `cons`, ogni volta che si trova un nodo al livello giusto. In questo caso, però occorre in qualche modo che tutte le attivazioni ricorsive condividano informazione sulla lista in costruzione. Come sempre, abbiamo due modi per fare questo: 1) usare un parametro per passare "avanti" la lista costruita (e usare il valore di ritorno per passarla "indietro" al chiamante); 2) oppure usare un parametro supplementare per indirizzo. Prima di vedere entrambe le soluzioni, una finezza finale: se vogliamo i nodi nella lista ordinati come si vedono nell'albero da sinistra a destra (lo stesso ordine in cui comparirebbero in una qualsiasi visita), occorre prima visitare i sottoalberi destri e poi i sinistri, perchè le inserzioni in testa "rovescerebbero" il risultato.

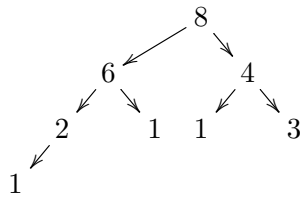
```
lista livelloKAux(binTree B,
                  int l, lista L){

  if (!B) return L;
  if (!l) return cons(B->val, L);
  L = livelloKAux(B->right, l-1, L);
  L = livelloKAux(B->left, l-1, L);
  return L;
  }

lista livelloK(binTree B, int l){
  return livelloKAux(B, l, NULL);
  }
```

```
lista livelloKAux(binTree B,
                  int l, lista* L){
  if (B) {
     if (!l) *L = cons(B->val, *L);
     livelloKAux(B->right, l-1, L);
     livelloKAux(B->left, l-1, L);
     }
  }
lista livelloK(binTree B, int l){
  lista L = NULL;
  livelloKAux(B, l, &L);
  return L;
  }
```

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 $\langle 8, 4, 1\rangle$.*



Per risolvere questo esercizio è sufficiente visitare l'albero. Qualora, si verifichi che in un sotto-albero x non ci sia, viene restituito il valore `NULL`, che comunica al chiamate anche il fallimento della ricerca nel sotto-albero. Viceversa, una volta trovato x, si comincia a costruire una lista mentre si *rientra* dalle chiamate ricorsive, evitando l'ispezione di ulteriori sotto-alberi ancora da visitare. Ecco il codice:

```
lista pathToX(binTree B, int x){
   if (!B) return NULL;
   if (B->val==x) return cons(x, NULL);
   lista L = pathToX(B->left, x);
   if (!L) L = pathToX(B->right, x);
   if (L) return cons(B->val, L);
   return NULL;
}
```

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.*

Cominciamo assiomatizziamo la relazione di prefixTree con equazioni ricorsive:

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

Conviene, a questo punto, scrivere la funzione C `prefixTree` che implementa questa relazione tra alberi.

```
int prefixTree(binTree B, binTree C){
   if (!B) return 1;
   if (!C) return 0;
   return (B->val == C->val) &&
       (prefixTree(B->left, C->left)) &&
       (prefixTree(B->right, C->right));
}
```

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;
```

Data una qualsiasi relazione d'ordine $\preceq$, la proprietà di una lista ordinata può essere facilmente assiomatizzata come segue (anche qui, i casi base sono la lista vuota e la lista con un solo elemento che sono evidentemente entrambe ordinate).

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

A questo punto, una normale iterazione (o ricorsione) su liste in cui viene invocata la funzione `prefixTree` conclude l'esercizio. Vediamo, entrambe le soluzioni.

```
int orderedRec(binTreeList L){

  if (!L || !L->next) return 1;
  if (!prefixTree(L->val,
              L->next->val))
         return 0;
  return orderedRec(L->next);
  }
}
```

```
int orderedIt(binTreeList L){
   while (L && L->next){
      if (!prefixTree(L->val,
            L->next->val))
         return 0;
      L=L->next;
   }
   return 1;
}
```