Tipi di Dato V: Alberi Binari

Ivano Salvo
Sapienza Università di Roma

Anno Accademico 2011-12

1 Alberi Binari

In questa sezione introdurremo un'altra struttura dati induttiva estrememamente importate in informatica: gli *alberi binari*. Gli alberi sono strutture dati pervasive: la struttura sintattica di un'espressione o di un programma C sono alberi (in generale ogni linguaggio con parentesi definisce parole che sono convenientemente rappresentabili con alberi). Ma la loro importanza va oltre l'informatica. Per esempio, sono alberi nel nostro senso, anche gli alberi genealogici.

Cominciamo con la definizione induttiva di albero binario.

Definizione 1.1 Una *albero binario* con nodi etichettati con elementi di tipo T è:

1. l'albero vuoto (cioè l'albero senza alcun nodo);
2. oppure una *radice* etichettata con un elemento di tipo T, con due sottoalberi chiamati rispettivamente *sottoalbero sinistro* e *destro*.

In Fig. 1 è mostrato un albero binario etichettato con interi nella usuale rappresentazione grafica. Il nodo etichettato con 4 è la radice, che ha due sottoalberi,

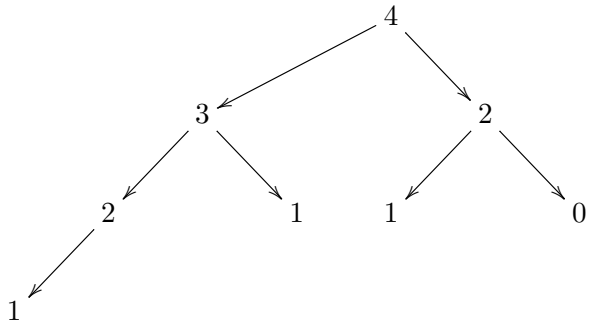

Figura 1: Esempio di albero binario

il *sottoalbero sinistro*, radicato nel nodo etichettato con 3, e il *sottoalbero destro* radicato nel nodo etichettato con 2.

Come nel caso delle sequenze, è opportuno avere una sintassi per rappresentare gli alberi in modo succinto non grafico. Come abbiamo già accennato gli aberi rappresentano sostanzialmente *linguaggi parentesizzati* e possono quindi essere convenientemente rappresentati con la seguente sintassi:

$$\_ \in \text{Tree}[A]$$
$$l, r \in \text{Tree}[A], a \in A \Rightarrow a(l, r) \in \text{Tree}[A]$$

Con questa sintassi, l'albero in Fig. 1 può essere rappresentato dall'espressione $4(3(2(1(\_,\_),\_),1(\_,\_)),2(1(\_,\_),0(\_,\_)))$. I nodi i cui figli sono entrambi alberi vuoti sono detti *foglie*. Evitando di scrivere i sottoalberi vuoti delle foglie, possiamo ottenere una sintassi più concisa, scrivendo lo stesso albero come $4(3(2(1,\_),1),2(1,0))$.

1.1 Rappresentazione in C di Alberi Binari

Come nel caso delle liste, una rappresentazione adeguata per gli alberi binari, fa uso di puntatori laddove la definizione del tipo induttivo riferisce a sè stessa. Useremo, come per le liste, il pointer `NULL` come rappresentazione dell'albero vuoto. Per semplicità, ancora una volta ci occupiamo di *alberi binari di interi*:

```
typedef struct tNode{
     struct tNode *left;
     int info;
     struct tNode *right;
   } treeNode;

typedef treeNode * tree;
```

Volendo dimenticare i puntatori nei nostri programmi, dobbiamo definire dapprima le funzioni base che permettono di costruire e decomporre un albero binario, i cosidetti *costruttori* e *distruttori*. Chiaramente dovremmo avere:

1. funzioni per accedere alle componenti di un albero: radice, sottoalbero destro e sinistro;
2. una funzione per verificare se un albero sia vuoto o meno;
3. funzioni per costruire un albero ricevuti come parametri di ingresso due alberi e l'etichetta della radice.

Ovviamente esistono molti modi per implementare le operazioni descritte sopra. Scegliamo la strada (vista anche con le liste) di definire un'unica funzione `isEmptyTree` con 4 parametri, che restituisce 1 se l'albero dato in ingresso (sul primo parametro) è l'albero vuoto, e 0 altrimenti: nel caso la risposta sia 0 (quindi l'albero *non* è vuoto) userà i rimanenti tre paramentri (passati per indirizzo!) per comunicare al chiamante radice, albero sinistro e destro:

```
int isEmptyTree(tree T, int* r, tree* L, tree* R){
    if (!T) return 0;
    *L = T->left;
    *R = T->right;
    *r = T->info;
    return 1;
}
```

Per costruire tutti gli alberi binari etichettati con interi, devo saper costruire l'albero vuoto e dati un intero r e due alberi L ed S, devo saper costruire l'albero con radice r ed alberi sinistro e destro L ed R rispettivamente:

```
tree makeTree(int r, tree L, tree R){
    tree tmp = malloc(sizeof(treeNode));

    tmp->info = r;
    tmp->left = L;
    tmp->right = R;
    return tmp;
}

tree makeEmptyTree(){
    return NULL;
}
```

Come al solito non saremo mai così integralisti da accedere alla struttura dati solo attraverso queste funzioni. Vediamo tuttavia qualche esempio di semplice funzione che usa i costruttori e i distruttori definiti sopra: il test se un nodo *è una foglia* (funzione `isLeaf`), la *profondità* di un albero (cioè la lunghezza del cammino più lungo radice-foglia, funzione `depth`)), e il conteggio del *numero dei nodi* di un albero (funzione `howManyNodes`). Osservate che fare ricorsione sugli alberi implica attivare in generale *due chiamate ricorsive* e che le corrispondenti versioni iterative di questi semplici programmi non sono affatto banali.

```
int isLeaf(tree T){
   int r;
   tree L, R;

   if (isEmptyTree(T, &r, &L, &R)) return 0;
      else return (!L && !R);
}

int depth (tree T){
   int r;
   tree L, R;

   if (isEmptyTree(T, &r, &L, &R)) return -1;
      else return 1 + max(depth(L), depth(R));
}
```

```
int howManyNodes (tree T){
   int r;
   tree L, R;

   if (isEmptyTree(T,&r, &L, &R)) return 0;
      else return 1 + howManyNodes(L) + howManyNodes(R));
}
```

1.2 Visite in Profondità

Un tipico problema sugli alberi, consiste nel visitare tutti i nodi dell'albero. Vedremo brevemente, come stampare in diversi ordini i nodi dell'albero. Le principali visite in profondità che vengono considerate in letteratura sono:

1. visita *in-order* o *simmetrica*: si visita prima il sottoalbero sinistro, quindi la radice ed infine il sottoalbero destro;
2. visita *pre-order* o *anticipata*: si visita prima la radice, poi il sottoalbero sinistro ed infine il sottoalbero destro;
3. visita *post-order* o *differita*: si visitano prima i sottoalberi ed infine la radice.

Come già osservato sulle liste, si possono ottenere diverse stampe di una lista (diritta o rovescia) a seconda se la stampa del contenuto di un nodo venga fatta all'"andata" o al "ritorno" delle chiamate ricorsive che percorrono la lista. Analogamente, negli alberi, sarà sufficiente spostare il punto in cui si stampa un nodo per ottenere i diversi tipi di visita in profondità. Osserviamo che le funzioni `depth` e `howManyNodes` sono funzioni che adottano una strategia analoga alla visita differita: prima computano il loro valore sui sottoalberi e poi combinano questi valori (ed eventualmente quello contenuto nella radice) per calcolare il loro valore. Vedremo nel seguito funzioni che traggono invece vantaggio dal trasmettere valori "in avanti" alle attivazioni della funzione sui sottoalberi.

```
void inOrder(tree T){
   if (T) {
      inOrder(T->left);
      printf("%3d", T->info);
      inOrder(T->right);
   }
}
```

```
void preOrder(tree T){
   if (T) {
      printf("%3d", T->info);
      preOrder(T->left);
      preOrder(T->right);
   }
 }
```

```
void postOrder(tree T){
   if (T) {
      postOrder(T->left);
      postOrder(T->right);
      printf("%3d", T->info);
   }
}
```

Sempre considerando il nostro albero in Fig. 1 la visita simmetrica produrrà la sequenza `1 2 3 1 4 1 2 0`, quella anticipata la sequenza `4 3 2 1 1 2 1 0`, mentre quella differita la sequenza `1 2 1 3 1 0 2 4`.

È interessante anche vedere come le visite, piuttosto che semplicemente stampare il contenuto dei nodi possano fare qualcos'altro, ad esempio restituire al chiamante una lista con i contenuti dei nodi (nell'ordine corrispondente alla visita scelta). Per l'occasione rispolveriamo la funzione `isEmptyTree`, ma ovviamente potete scrivere il codice anche nello stile delle funzioni viste sopra.

```
lista inOrder(tree T){
   int r;
   tree L, R;

   if (isEmptyTree(T, &r, &L, &R)) return NULL;
      else return append(inOrder(L), addHead(inOrder(R), r));
}

lista preOrder(tree T){
   int r;
   tree L, R;

   if (isEmptyTree(T, &r, &L, &R)) return NULL;
      else return addHead(append(preOrder(L), preOrder(R)), r);
}

lista postOrder(tree T){
   int r;
   tree L, R;

   if (isEmptyTree(T, &r, &L, &R)) return NULL;
      else return addTail(append(postOrder(L), postOrder(R)), r);
}
```

Concludiamo la sezione con un ultimo problema, sempre legato alle visite: dato un albero, restituire la lista che contiene tutte le sue foglie, da destra a sinistra.

```
lista frontier(tree T){
   int r;
   tree L, R;

   if (isEmptyTree(T, &r, &L, &R)) return NULL;
      else if(isLeaf(T)) return addHead(NULL, r);
      else return append(frontier(L), frontier(R));
}
```

1.3 Visita per Livelli

Visitare un albero per livelli significa visitare i nodi dell'albero, cominciando dalla radice, e poi visitare tutto il primo livello, poi il secondo e così via. Nel nostro esempio, la visita per livelli dovrebbe produrre la sequenza `4 3 2 2 1 1 0 1`.

La difficoltà dipende dal fatto che l'ordine in cui vanno visitati i nodi non è relato con la rappresentazione degli alberi: infatti ogni nodo è collegato col padre e i figli, ma non con i "fratelli" (e "cugini" di vario grado), ossia gli altri nodi che stanno sullo stesso livello.

La tecnica consiste nel mantenere una coda coi sottoalberi ancora da visitare. Ad ogni iterazione, prendere il primo sottoalbero della coda, visitare la radice e mettere i sottoalberi in fondo alla coda. Così facendo, sottoalberi che hanno la radice allo stesso livello, saranno sempre memorizzati in nodi adiacenti nella coda.

Osservate che la lunghezza della coda di norma, cresce durante l'algoritmo (finchè non si arriva ai sottoalberi vuoti) e quindi non è una buona funzione di terminazione. La misura che decresce è il numero complessivo dei nodi dei sottoalberi memorizzati nella coda. Tutti i nodi degli alberi radicati nei nodi messi sulla coda, infatti, sono esattamente tutti i nodi ancora da visitare. Questa è anche la proprietà invariante del ciclo.

```
lista visitaLivelli(tree T) {
   queue Q = createQueue();
   lista M = NULL;
   tree L, R, U;
   int r;

      /* metto la radice nella coda */
   enqueue(Q, T);
   while (!isEmptyQueue(Q)){
       U = first(Q);
       dequeue(Q);
       if (!isEmptyTree(U, &r, &L, &R)) {
          M = addTail(M, r);
          enqueue(Q, L);
          enqueue(Q, R);
       }
   }
   return M;
}
```

Facendo inserimenti in coda a una lista, questo programma ha complessità quadratica nel numero dei nodi dell'albero, mentre è legittimo attendersi che una visita abbia complessità lineare. Una possibile soluzione a questo problema è semplicemente aggiungere in testa a `M` e concludere la funzione con `return reverseRecEff(M);`.

1.4 Bilanciamento

Molte applicazioni degli alberi generano algoritmi la cui complessità è proporzionale alla *profondità* dell'albero. È facile rendersi conto che, presi due alberi con lo stesso numero di nodi, sarà meno profondo l'albero più *bilanciato*, o se preferite di struttura più simmetrica. Il caso estremo è l'albero completamente sbilanciato, che ha un unico cammino, la cui profondità è lineare nel numero dei nodi.

Cominciamo con il dare alcune definizioni, poi scriveremo delle funzioni che verificano il bilanciamento di un albero. La costruzione di alberi bilanciati esula dagli obiettivi della presente dispensa.

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.

Occupiamoci ora di scrivere due funzioni che verificano il bilanciamento di un albero, `depthBalanced` per il bilanciamento in altezza e `NNbalanced` per il bilanciamento nel numero dei nodi. Forti delle funzioni `depth` e `howManyNodes` possiamo scrivere due funzioni molto eleganti e semplici che seguono sostanzialmente lo schema di programmazione già abbondantemente visto in precedenza, e cioè, stabilire il valore della funzione sul caso base e calcolare il valore della funzione su alberi non vuoti, componendo in modo opportuno i valori calcolati dalle chiamate ricorsive sui sottoalberi:

```
int depthBalanced(tree T) {
   int r;
   tree L, R;

   if (isEmpty(T, &r, &L, &R)) return 1;
   if (depthBalanced(L) && depthBalanced(R)
       && abs(depth(L)-depth(R))<=1) return 1;
   return 0;
}

int NNBalanced(tree T) {
   int r;
   tree L, R;

   if (isEmpty(T, &r, &L, &R)) return 1;
   if (NNBalanced(L) && NNBalanced(R)
       && abs(howManyNodes(L)-howManyNodes(R))<=1) return 1;
   return 0;
}
```

Le due funzioni precedenti sono molto eleganti, ma hanno il difetto di eseguire numerose visite inutili di sottoalberi. È infatti possibile scrivere una funzione ricorsiva che verifica il bilanciamento attraversando *una sola volta* tutti i nodi dell'albero. In entrambi i casi, l'informazione sintetica restituita dalle funzioni (bilanciato/non bilanciato) costringe a riesaminare i sottoalberi con le funzioni che calcolano profondità e numero dei nodi.

Quindi, riscriveremo le funzioni in modo che contemporaneamente, in un unica scansione, la funzione verifichi il bilanciamento e calcoli rispettivamente la profondità e il numero dei nodi. Vedremo due tecniche per "restituire" più informazioni:

In `depthBalancedEff` useremo un parametro ausiliario passato per *indirizzo*: nel caso in cui un albero sia bilanciato, questo parametro viene opportunamente aggiornato con l'altezza dell'albero.

In `NNBalancedEff` useremo il valore ritornato per *codificare* entrambe le informazioni: se la funzione ritorna un numero maggiore o uguale a 0, significa che l'albero è bilanciato e il valore ritornato è il numero dei nodi dell'albero. Altrimenti viene ritornato il valore -1 (osservare che questo valore è diverso da qualsiasi possibile numero di nodi dell'albero).

```
int depthBalancedEff(tree T, int *h){
   int hr, hl;
   int r;
   tree L, R;

   if (isEmptyTree(T, &r, &L, &R)) {
         *h = -1;
         return 1;
   }
   if (depthBalancedEff(L, &hl) && depthBalancedEff(R, &hr)
       && abs(hr - hl)<=1){
          *h = 1+max(hl,hr);
          return 1;
          }
   return 0;
}

int NNbalancedEffAux(tree T){
   int nnr, nnl;
   int r;
   tree L, R;

   if (isEmptyTree(T, &r, &L, &R)) return 0;
   if ((nnl=NNbalancedAux(L))>=0 && (nnr=NNbalancedAux(R))>=0
           && abs(nnr-nnl)<=1) return 1+nnl+nnr;
   return -1;
}
```

Affinché il chiamante abbia l'interfaccia che si aspetta (1 se l'albero e' bilanciato e zero altrimenti) possiamo definire una funzione che si limita a trasformare i risultati della funzione `NNbalancedEffAux` in modo opportuno:

```
int NNbalancedEff(tree T){
   if (NNbalancedEffAux(T)>=0) return 1;
      else return 0;
}
```

1.5 Relazioni di Uguaglianza e SottoAlbero

Concludiamo questa breve sezione introduttiva agli alberi, con tre problemini: verificare se due alberi binari sono uguali, se uno e' sottoalbero dell'altro, e se uno è lo *specchio* dell'altro. Per l'occasione vediamo di riprendere l'ottima abitudine di specificare con equazioni ricorsive queste proprietà.

$$\mathsf{equals}(a_1(l_1, r_1), a_2(l_2, r_2)) = a_1 = a_2 \ \wedge\ \mathsf{equals}(l_1, l_2) \ \wedge\ \mathsf{equals}(r_1, r_2)$$
$$\mathsf{equals}(\_, \_) = \mathsf{true} \qquad \mathsf{equals}(t, \_) = \mathsf{equals}(\_, t) = \mathsf{false}$$

Come visto per le liste, le equazioni ricorsive si traducono immediatamente in un elegante programma ricorsivo per casi.

```
int eqTree(tree T, tree U){
   if (!T && !U) return 1;
   if (!T || !U) return 0;
      /* T e U entrambi non vuoti */
   if (T->info==U->info && eqTree(T->left, U->left))
      return eqTree(T->right, U->right);
   return 0;
 }
```

Facciamo lo stesso per la relazione di sottoalbero:

$$\mathsf{subtree}(t, a(l, r)) = \mathsf{equals}(t, a(l, r)) \ \vee\ \mathsf{subtree}(t, l) \ \vee\ \mathsf{subtree}(t, r)$$
$$\mathsf{subtree}(\_, t) = \mathsf{true} \qquad \mathsf{subtree}(t, \_) = \mathsf{false}\ (\text{se } t \neq \_)$$

da cui consegue immediatamente il seguente programma ricorsivo:

```
int subTree(tree T, tree U) {
   if (!T) return 1;
   if (!U) return 0;
   if (eqTree(T, U)) return 1;
   return subTree(T, U->left) || subTree(T, U->right);
}
```

Concludiamo questa nota introduttiva con un unico esempio di funzione che genera come risultato un albero. La funzione che restituisce un albero specchiato.

$$\mathsf{mirror}(a(l, r)) = a(\mathsf{mirror}(r), \mathsf{mirror}(l)) \qquad \mathsf{mirror}(\_) = \_$$

In questo caso, dobbiamo porci il problema se vogliamo che la funzione che produce l'albero specchiato, generi un *nuovo* albero, allocando nuova memoria, oppure si limiti a spostare adeguatamente i puntatori. Per amore di esercizio, vediamo entrambe le versioni.

```
tree mirrorFun(tree T){
     int r;
     tree L, R;

     if (isEmptyTree(T, &r, &L, &R))
        return NULL;
     return makeTree(r, mirrorFun(R), mirrorFun(L));
}

tree mirrorRec(tree T){
     if (!T) return NULL;
     T->left = mirror(T->right);
     T->right = mirror(T->left);
     return T;
}
```

Esercizi e Spunti di Riflessione

1. Scrivere una funzione che prende in input un albero binario e restituisce in output una *stampa indentata* preorder dell'albero. Ad esempio, ricevendo in input l'albero in Fig. 1 restituisce come output:

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

   Fare lo stesso per le visite preOrder, postOrder e ★ a livelli.

2. Scrivere una funzione che prende in input un albero binario e restituisce in output una stampa dell'albero in notazione a parentesi, come visto nella prima sezione.

3. ★ Scrivere una funzione che prende in input una stringa che rappresenta un albero in notazione a parentesi, e ricostruisce l'albero associato.

4. ♣ Considerare la seguente definizione di albero binario:

   Definizione 1.4 Un *albero binario* con nodi etichettati con elementi di tipo T è:

(a) una foglia etichettata con un elemento di tipo T;
(b) oppure una radice etichettata con un elemento di tipo T, con due sottoalberi chiamati rispettivamente sottoalbero sinistro e destro.

Riflettere su quali alberi vengono definiti e confrontare con l'insieme degli alberi definiti dalla definizione 1.1. Scrivere le opportune funzioni costruttori e distruttori che garantiscano che gli alberi ottenuti appartengano a questa famiglia di alberi.

5. Considerate le funzioni che visitano un albero restituendo la lista dei nodi nell'albero, nell'ordine prescritto dalla visita. Sono ottime da un punto di vista di complessità computazionale? Quale dovrebbe essere la loro complessità?

   Nel caso la vostra risposta sia NO, scrivere delle funzioni computazionalmente ottime.

6. ♣ Dimostrare per induzione che se un albero ha profondità n ha al più $2^n - 1$ nodi.

7. ♣ Dimostrare per induzione che se un albero ha profondità n ed è completo (cioè è bilanciato perfettamente e ogni nodo o è una foglia o un nodo interno) ha esattamente $2^n - 1$ nodi.

8. ♣ Dimostrare per induzione che se un albero è bilanciato nel numero dei nodi, allora è bilanciato in profondità.

9. Scrivere una funzione che prendendo un albero binario in input restituisce un puntatore al sottoalbero di somma massima.

10. Scrivere una funzione che che prendendo un albero binario in input restituisce un puntatore a un sottoalbero *massimale* che contiene solo valori positivi.

11. ★ ♣ Chiaramente ci sono molti alberi che hanno la stessa visita inOrder, postOrder o preOrder. Supponendo di avere due visite (ad esempio inOrder e preOrder) sotto quali condizioni è possibile ricostruire in modo univoco l'albero binario associato?

12. Scrivere una funzione che prende in input due array di ugual lunghezza che rappresentano rispettivamente la visita preOrder e inOrder di un albero e costruiscono un albero che ha produce le visite in input.

13. Scrivere una funzione che legge una stringa che rappresenta una espressione aritmetica e la carica in un albero, dimodochè i nodi interni contengano gli operatori e le foglie contengano i numeri.

    Rispettare le regole di precedenza, per cui `5 * 2 + 4` dovrà essere caricata nell'albero `+(*(5,2),4)`.

14. Arricchire la sintassi delle espressioni aritmetiche con le parentesi.

15. Arricchire la sintassi delle espressioni aritmetiche con le variabili.