Appunti Senza Pretese di Programmazione II: **Costruzione di un Albero Bilanciato**

Alessandro Panconesi
DSI, *La Sapienza*
via Salaria 113, 00198, Roma

Consideriamo un problema che ci consentirá di vedere un altro esempio della potenza del metodo *divide-et-impera* nonché di vedere all'opera in un altro contesto alcune importanti idee viste in precedenza: ricorsione ed induzione.

Il problema di cui ci vogliamo occupare è la costruzione di un **albero binario bilanciato**. Intuitivamente, un albero binario T è bilanciato se e solo se, per ogni nodo, il numero di nodi del sottoalbero sinistro é lo stesso di quelli del sottoalbero destro. Notare come le foglie, non avendo figli, ricadono nella definizione la quale peró è troppo restrittiva, come mostra l'esercizio seguente.

Esercizio 1 *Dimostrare che la definizione appena data implica che il numero di nodi dell'albero deve essere della forma* $2^k - 1$*, per un qualche* k *intero. In altre parole, con la definizione data possono esistere solo alberi bilanciati con numero di nodi pari a* $0, 1, 3, 7, 15, 31, 63, \ldots$

Pertanto rilassiamo leggermente la definizione nel modo seguente. Dato un nodo x denotiamo con ℓ_x il numero di nodi del sottoalbero sinistro di x (ℓ=left) e con r_x quello del sottoalbero destro (r=right). Per le foglie $\ell_x = r_x = 0$.

Definizione 1 *Un albero binario* T *si dice bilanciato se, per ogni nodo* x *di* T*,*

$$|\ell_x - r_x| \leq 1.$$

Esercizio 2 *Dimostrare (ad esempio per induzione) che con la nuova definizione, per ogni* n*, esiste un albero bilanciato di* n *nodi.*

Esercizio 3 *L'altezza di un albero bilanciato di* n *nodi è* $\lfloor \log n \rfloor$*. Vero o falso?*

Veniamo quindi alla definizione del nostro problema:

Dati n *ed una lista di* n *elementi, costruire un albero bilanciato.*

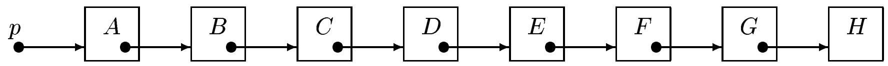
In altre parole, l'input del problema consiste di n e di una lista di n elementi, mentre l'output deve consistere di un albero (binario) bilanciato contenente gli elementi della lista.

Per trovare una soluzione ricorsiva del problema possiamo procedere in questo modo:

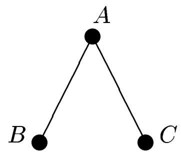
- Sinceriamoci del fatto che nel caso base il problema ha soluzione. Nel nostro caso per $n = 0$ ed una lista ℓ vuota la soluzione é banale ed è costituita dall'albero vuoto `nil`.
- Supponiamo di avere giá a disposizione un metodo risolutivo per ogni $i < n$ ed ogni lista ℓ di i elementi; possiamo estenderlo per ottenere un metodo che funziona anche per n ed una lista di almeno n elementi?

Notare come questo schema ricalchi una tipica dimostrazione per induzione. Ció non é casuale in quanto un algoritmo ricorsivo altri non é che una dimostrazione per induzione dell'esistenza di un algoritmo per calcolare la funzione *input* $\rightarrow$ *output* del problema che si sta cercando di risolvere!

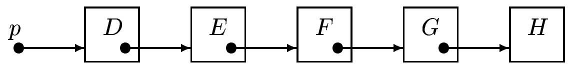
Nel mettere appunto la soluzione la difficoltá ovviamente consiste nel risolvere il passo induttivo. Nel nostro caso conviene adottare la seguente ipotesi: Supponiamo di disporre di una procedura `makeTree` con due parametri di ingresso: (a) un numero n e (b) un puntatore p ad una lista di almeno n elementi; l'ipotesi induttiva é che tale procedura produce un albero binario bilanciato di n elementi ottenuto "mangiando" i primi n elementi della lista. Ad esempio, dato $n = 3$ e la lista



...`makeTree(3,p)` produrrebbe un albero bilanciato con i tre elementi A, B e C:



... e la lista risultante sarebbe la seguente:



Implementando la lista e l'albero bilanciato con i seguenti tipi di dato

```
typedef struct Element {
  int value;
  struct Element *next;
  struct Element *left;
  struct Element *right;
  } Element;

typedef Element *List;
typedef Element *Tree;
```

... la procedura `makeTree` restituirebbe un puntatore di tipo `Tree`. Per semplicitá assumiamo che la variabile `list`, che punta all'inizio della nostra lista, sia globale.

```
List list = NULL;
```

L'algoritmo risultante per `makeTree(n)` sarebbe dunque il seguente:

- Se $n = 0$ restituisci `nil`, l'albero vuoto;
- Se $n > 0$ allora:
    1. estrai ("mangia") il primo elemento della lista, chiamiamolo A, facendolo diventare radice dell'albero;

```
Tree makeTree(int n) {        // make tree out of first n elements of list
  Tree root;
  int l=(n-1)/2, r=n-1-l; // number of elements of left and right subtree

  if (n == 0) return(NULL);
  else {
    root = list;
    list = list->next;
    root->left = makeTree(l);
    root->right = makeTree(r);
    return(root);
  }
}
```

Figure 1: Implementazione di `makeTree`

2. forma il sottoalbero sinistro di A estraendo i successivi $(n-1)/2$ elementi;
3. forma il sottoalbero destro di A con i seguenti $(n-1)/2$ elementi.

Alla fine restituisci il puntatore alla radice A.

Ad esempio, con la lista A, B, C, D, E, F, G, `makeTree` creerebbe A come radice e poi costruirebbe l'albero sinistro con B, C, D e quello destro con E, F, G. Ricorsivamente, l'albero sinistro avrebbe B come sottoradice con figlio sinistro C e figlio destro D. Il programma risultante, una traduzione quasi letterale dell'algoritmo ricorsivo di cui sopra, compare in Figura 1. L'unica differenza é che, potendo essere $(n-1)/2$ un numero frazionario, la lista viene suddivisa in due pezzi di dimensioni pari a $r := \lfloor (n-1)/2 \rfloor$ ed $\ell := n-1-r$. La Figura 2 mostra invece un programma completo per la generazione di una lista e per la sua successiva trasformazione tramite `makeTree`. Il programma contiene anche una routine `printTree` per la stampa indentata dell'albero binario.

Esercizio 4 *Riscrivere* `makeTree` *passando come parametro la variabile* `list`*, anziché usarla come variabile globale.*

Veniamo adesso al consueto problema: Qual'é la complessitá (tempo di calcolo) dell'algoritmo implementato tramite `makeTree` ? Data la natura ricorsiva dell'algoritmo é quasi automatico usare un' equazione di ricorrenza. Un ispezione del codice rivela che il lavoro svolto da `makeTree` viene descritto da questa equazione

$$T(n) = a + T(r) + T(\ell)$$

con condizione iniziale $T(1) = b$, dove a e b sono costanti ed r ed ℓ sono il numero di nodi dei sottoalberi destro e sinistro. Ora, questa equazione é un po' delicata per via del fatto che

$$r = \lfloor \frac{n-1}{2} \rfloor$$

ed

$$\ell = n - 1 - r.$$

```c
#include <stdio.h>

typedef struct Element {
  int value;
  struct Element *next, *left, *right;
  } Element;
typedef Element *List;
typedef Element *Tree;

List list = NULL;                                    // pointer to list

struct Element *myAlloc(void) {
  struct Element *p;

  if ((p = (struct Element *) malloc(sizeof(Element))) != NULL) {
    p->next = NULL; p->left = NULL; p->right = NULL;
    return (p);
  }
  else {
    printf("Fatal memory allocation error!");
    exit(1);
  }
}

List createList(int n) { // create list with n elements pointed to by p
  List new;

  if (n>0) {
    new = myAlloc();
    new->value = n;
    new->next = createList(n-1);
    return (new);
  }
  else return(NULL);
}

Tree makeTree(int n) {
  Tree root;
  int l=(n-1)/2, r=n-1-l;

  if (n == 0) return(NULL);
  else {
    root = list;
    list = list->next;
    root->left = makeTree(l);
    root->right = makeTree(r);
    return(root);
  }
}

void printTree(int displace, Tree t) {
  int i;

  if (t != NULL) {
    if (t->left == NULL && t->right == NULL) {
      for (i=0; i<displace; i++) printf(" ");
      printf("%d\n", t->value);
    }
    else {      printTree(displace+1, t->right);
      for (i=0; i<displace; i++) printf(" ");
      printf("%d \n", t->value);
      printTree(displace+1, t->left);
    }
  }
}

main() {
  int n;

  printf("Number of elements:");
  scanf("%d", &n);
  list = createList(n);
  printTree(0, makeTree(n));
}
```

Figure 2: Creazione di una lista di n elementi, sua trasformazione in albero bilanciato tramite `makeTree` e stampa dell'albero

Per valori di n grandi peró si ha che $r \approx \ell \approx n/2$ per cui sembrerebbe sufficiente considerare l'equazione

$$T(n) = a + 2\,T(n/2) \tag{1}$$

con $T(0) = T(1) = b$. In effetti, é possibile giustificare in modo rigoroso tale approssimazione, ma ció esula dai nostri scopi. Per cui prendiamo senz'altro per buona l'Equazione (1). Tale equazione é equivalente a:

$$T(n) = 1 + 2\,T(n/2) \tag{2}$$

con $T(0) = T(1) = 1$. La verifica che tale equazione ha soluzione $T(n) = 2n - 1 \approx n$ é lasciato come esercizio per il lettore.

É possibile arrivare alla stessa conclusione in altro modo. Si supponga di dover pagare un Euro per ogni operazione elementare svolta da `makeTree`. Si noti che l'algoritmo si limita ad estrarre ogni elemento dalla lista *una ed una sola volta* e che per ogni estrazione si effettuano un numero *costante* di operazioni elementari; cioé si paga un numero costante di Euro ogni volta che un elemento é estratto dalla lista e piazzato nell'albero. Per cui il costo totale sará al massimo cn Euro, per una qualche costante c.