Appunti Senza Pretese di Programmazione II: **Alberi di Ricerca**

Alessandro Panconesi
DSI, *La Sapienza*, Roma

Vediamo adesso un altro modo di risolvere *Il Problema del Dizionario*, cioé quello di realizzare una struttura dati che consenta di inserire, cancellare e cercare delle **voci** tramite una chiave di ricerca in modo semplice e rapido. La struttura dati in questione é detta **albero di ricerca**. Essa consiste di un albero binario in cui ogni nodo contiene un valore detto **chiave** che per semplicitá assumeremo essere un intero positivo. L'albero soddisfa la seguente proprietá:

> *Dato un qualsiasi nodo x dell'albero, il sottoalbero di sinistra contiene valori minori del valore di x, mentre il sottoalbero di destra contiene valori maggiori di quello di x.*

Nel proseguio denoteremo con x sia il nodo che il suo valore. A rigore dovremmo distinguere tra i due usando, ad esempio, x per il nodo e v_x o qualcosa di simile per il valore ivi contenuto. Ma dato che la cosa non dovrebbe creare ambiguitá di sorta, per semplicitá tipografica adotteremo la suddetta convenzione. Dato x, denoteremo con L_x il suo sottoalbero di sinistra (L=left) e con R_x quello di destra (R=right). Dato un albero T, la proprietá che T é un albero di ricerca puó quindi esprimersi come segue ("$\wedge$" é un simbolo della logica proposizionale che denota la congiunzione "e"):

$$\forall x \in T, (\forall y \in L_x \; y < x) \wedge (\forall z \in R_x \; x > z).$$

Diremo inoltre per semplicitá che *"y é minore (maggiore) di x"* come abbreviazione della frase *"il valore contenuto nel nodo y é minore (maggiore) del valore contenuto nel nodo x."*

1 Operazioni sugli alberi di ricerca

Per fissare le idee nella discussione che segue definiamo le strutture dati PASCAL per implementare un albero di ricerca. Ció é fatto in Figura 1. Sebbene `Tree` e `↑Node` siano la stessa cosa é opportuno distinguerli perché ció rende i programmi piú leggibili. La struttura degli alberi di ricerca consente di ricercare ed inserire elementi con grande facilitá. La ricerca avviene tramite il seguente algoritmo ricorsivo, che ricorda la ricerca binaria (o dicotomica) vista in precedenza.

PROCEDURA FIND: dati due parametri, un albero `t` ed una chiave x, la procedura restituisce un puntatore ad un nodo se l'elemento esiste e `nil` altrimenti.

- Se l'albero `t` é vuoto (nil) restituisci `nil`;
- se la radice di `t` contiene x restituisci `t`;
- se la chiave della radice di `t` é maggiore di x, invoca ricorsivamente `find` con x ed il sottoalbero sinistro;
- altrimenti, invoca ricorsivamente `find` con x ed il sottoalbero destro.

L'algoritmo si traduce immediatamente nel codice di Figura 2.

```
type
  keyType: integer;
  Tree: ↑Node;
  record Node =
      key: keyType;
      parent: ↑Node ;
      left: Tree;
      right: Tree;
      end;
```

Figure 1: Tipi di dato per rappresentare alberi di ricerca.

```
function find(x: keyType; t: Tree): ↑Node;
begin
  if (t=nil) then find := nil
  else if (t↑.key=x) then find:= t;
  else if (t↑.key > x) then find:= find(x, t↑.left);
  else { t↑.key < x } find:= find(x, t↑.right);
end;
```

Figure 2: Implementazione di `find`

L'inserimento si ottiene tramite un algoritmo molto simile. La procedura assume che l'elemento da inserire non sia presente nell'albero. É possibile controllare questa condizione ad esempio tramite la procedura `find` vista in precedenza. La ricerca attraversa l'albero alla maniera della ricerca binaria sino a quando non viene individuato il "posto giusto" per l'inserimento: se l'albero é vuoto l'elemento da inserire diventa la nuova radice, altrimenti la ricerca procede nel sottoalbero sinistro, se la chiave da inserire é piú piccola di quella del nodo corrente, oppure nel sottoalbero destro, se la chiave da inserire é piú grande del nodo corrente.

Esercizio 1 *In quanti posti diversi puó essere inserito un nuovo elemento?*

PROCEDURA INSERT: dati due parametri, un albero `t` ed un nodo esterno puntato da `p`, la cui chiave non é presente in `t`, la procedura inserisce il nuovo elemento al posto giusto.

- Se l'albero `t` é vuoto (il nodo puntato da) `p` ne diventa la nuova radice;
- se la chiave della radice di `t` é maggiore di quella di `p`, invoca ricorsivamente `insert` con `p` ed il sottoalbero sinistro;
- altrimenti, invoca ricorsivamente `insert` con `p` ed il sottoalbero destro.

L'implementazione di questo algoritmo presenta una sottigliezza. La ricerca ricorsiva del "posto giusto" inizia col parametro `t` che punta alla radice dell'albero il quale poi via via assume i valori dei campi `t↑.left` e `t↑.right` fino a quando non assume il valore `nil`. Tipicamente a quel punto `t` é il campo puntatore `left` o `right` di una foglia dell'albero. Per poterne modificare il contenuto

```
procedure insert(p: ↑Node, var t: Tree);
begin
  if (t=nil) then t := p
  else if (t↑.key> p↑.key>) then insert(p, t↑.left);
  else if (t↑.key<p↑.key>) then insert(p, t↑.right);
end;
```

Figure 3: Implementazione di `insert`

bisognerebbe avere un puntatore alla foglia. Per farció é possibile trascinarsi dietro un puntatore che punti "un passo indietro" oppure, piú elegantemente, passare `t` come `var`. Questa é la soluzione adottata in Figura 3.

Esercizio 2 *Scrivere una versione iterativa della procedura* `find`*. (Suggerimento:* **non** *utilizzare pile).*

Esercizio 3 *Scrivere la* `insert` *come* **funzione***. Come prima versione fatelo senza aggiornare il campo* `parent`*.*

Esercizio 4 *La procedura* `insert` *di Figura 3 non aggiorna il campo* `parent`*. Riscrivetela in modo che esso venga aggiornato. Per la radice dell'albero il campo deve essere posto a* `nil`*.*

Esercizio 5 *Scrivere una versione iterativa delle procedure* `find` *e* `insert`*.*

Esercizio 6 *Quale parametro di un albero di ricerca dá il costo delle operazioni* `find` *e* `insert`*?*

L'implementazione di `delete` é invece piú complessa. Prima di vedere come fare esploriamo altre proprietá degli alberi di ricerca.

2 Proprietá degli Alberi di Ricerca

La struttura di un albero di ricerca consente di estrarre notevoli informazioni. Come al solito il miglior modo per rendersene conto é di risolvere degli esercizi.

Esercizio 7 *Scrivere una procedura (ricorsiva!) che, dato un albero di ricerca, ne stampi i valori in ordine crescente.*

Esercizio 8 *Scrivere una procedura che, dato un albero di ricerca, ne stampi il valore minimo.*

Esercizio 9 *Sia dato un albero binario T con la seguente proprietá ricorsiva: per ogni nodo x, il figlio sinistro é minore o uguale di x, mentre quello destro é maggiore o uguale ad x. T é un albero di ricerca?*

Esercizio 10 *Un albero binario é di ricerca se e solo se una visita in-order ne stampa i valori in senso crescente. Vero o falso?*

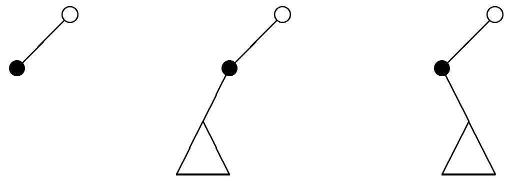

Figure 4: I tre casi per `spliceLeft`. Il nodo da cancellare é in neretto.

3 Cancellazione

Veniamo ora al problema della cancellazione. Costruiremo la soluzione analizzando i vari casi che possono presentarsi. Il problema con la cancellazione é che, contrariamente all'inserimento, la eliminazione di un nodo puó distruggere la struttura dell'albero. Il nodo da cancellare puó presentare un numero abbastanza alto di tipologie, ma come vedremo ad una analisi piú attenta, i casi veramente diversi, quelli che necessitano trattamenti diversi tra loro, sono pochi. Di primo acchitto sembrerebbe che le variabili in gioco siano tre. La prima é il "ruolo" del nodo, che puó essere quello di foglia, nodo interno o radice. Il secondo é il numero di figli, che possono essere 0, 1 oppure 2. Infine, potrebbe essere rilevante se il nodo da eliminare fosse il figlio sinistro o destro di suo padre (la radice é esclusa da questa tipologia). Andando per gradi riusciremo a condensare tutti questi casi in un numero minore di casi diversi.

Denotiamo d'ora in poi con p il nodo da cancellare e con f_p suo padre. Siano inoltre L_p ed R_p, rispettivamente, i sottoalberi sinistro e destro. Iniziamo con il considerare la seguente situazione: almeno uno dei due sottoalberi di p è vuoto. Possono quindi darsi tre casi, illustrati in Figura 4, vale a dire:

1. p é il figlio sinistro di f_p e non ha figli;
2. p é il figlio sinistro di f_p ed ha L_p ma R_p é vuoto;
3. p é il figlio sinistro di f_p ed ha R_p ma L_p é vuoto.

Questi tre casi possono essere gestiti molto semplicemente in questo modo: *Se L_p é vuoto allora R_p diventa il sottoalbero sinistro di f_p, altrimenti lo diventerá L_p.* Chiameremo questa procedura `spliceLeft`.

Esercizio 11 *Implementare* `spliceLeft` *la quale, dato in ingresso un puntatore* `p` *ad un nodo che soddisfa alle tre condizioni di cui sopra, lo cancelli.*

La situazione simmetrica é la seguente:

1. p è il figlio destro di f_p e non ha figli;
2. p è il figlio destro di f_p ed ha L_p ma R_p é vuoto;
3. p è il figlio destro di f_p ed ha R_p ma L_p é vuoto.

... ed ovviamente puó essere trattata in modo identico.

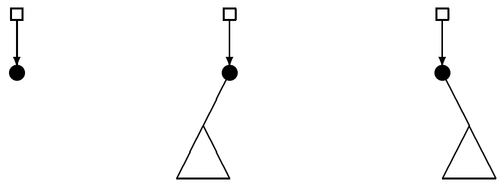

Figure 5: p, in nero, é radice dell'albero. Il puntatore `root` é raffigurato con un quadrato.

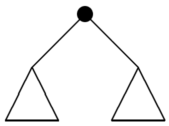

Figure 6: p é un nodo interno con entrambi i sottoalberi non vuoti

Esercizio 12 *Implementare* `spliceRight` *la quale, dato in ingresso un puntatore* `p` *ad un nodo che soddisfa alle tre condizioni di cui sopra, lo cancella.*

Il prossimo caso che consideriamo è quello in cui p è la radice dell'albero ed ha almeno uno dei suoi due sottoalberi vuoti. I tre casi relativi sono illustarti in Figura 5. D'ora in poi assumeremo l'esistenza di una variabile globale `root` di tipo `Tree` che punta alla radice.

Esercizio 13 *Scrivere una procedura* `spliceRoot` *che cancelli il nodo radice nei tre casi illustrati in Figura 5.*

D'ora in poi assumeremo di poter disporre di una procedura `splice` in grado di cancellare un nodo che non abbia due figli. Tale procedura userá opportunamente le tre procedure `spliceLeft,` `spliceRight` e `spliceRoot`.

Veniamo quindi all'ultimo caso da considerare, illustrato in Figura 6: il nodo p ha entrambi i sottoalberi pieni (non vuoti) che è il caso piú interessante. Definiamo innazitutto il *successore* di un nodo p. Questi altri non é che, tra tutti gli elementi dell'albero T, il piú piccolo dei maggioranti di p (ricordate che stiamo assumendo che nessuna chiave sia ripetuta):

$$s(p) := \min\{x : x \in T, x > p\}.$$

Esercizio 14 *Dimostrare che se* $R_p \neq \emptyset$ *allora*

$$s(p) = \min\{x : x \in R_p\}.$$

Se quindi p ha il sottoalbero destro non vuoto il calcolo del successore é piuttosto agevole. Il caso piú generale é invece piú complesso e viene lasciato come utile esercizio.

```
procedure delete(p: ↑Node);
var
  hasBothChildren: boolean;
  s, f: ↑Node;
begin
  f := p↑.parent;
  hasBothChildren := ((p↑.left <> nil) and (p↑.right <> nil));
  if (hasBothChildren) then
  begin
    s := successor(p);
    copy(s,p);
    splice(s);
  end
  else splice(p)
end;
```

Figure 7: Implementazione di `delete`

Esercizio 15 *Scrivere una funzione* `successor` *la quale, dato in ingresso un puntatore* `p` *ad un nodo dell'albero, restituisca un puntatore al successore, se esiste, oppure* `nil`*, se non esiste.*

Ma nel caso che ci interessa il compito é, come risulta dall'esercizio 3, abbastanza semplice: il successore di p é l'elemento piú piccolo di R_p, il sottoalbero destro di p.

Esercizio 16 *Scrivere una funzione* `easySuccessor` *la quale, dato in ingresso un puntatore* `p` *ad un nodo dell'albero, restituisca un puntatore al successore nel caso in cui il sottoalbero destro sia non vuoto.*

Ora, l'idea per cancellare p é la seguente. Il successore di p, chiamiamolo s, non ha sottoalbero sinistro (dimostratelo!), per cui cancellare s puó essere fatto agevolmente tramite la procedura `splice` vista in precedenza. A questo punto, basta rimpiazzare p con s ed il gioco é fatto. Ricapitolando:

- sia s il successore di p e sia f_s il padre di s;
- si rimpiazza p con s; all'atto pratico si copierá il contenuto di s in p;
- si esegue `splice(s)` il cui effetto è di far diventare R_s, il sottoalbero destro di s, il nuovo sottoalbero sinistro di f_s.

Esercizio 17 *Dimostrare che la procedura di cui sopra é corretta, cioé che l'albero risultante é un albero di ricerca.*

Per cui il codice PASCAL per la procedura `delete` è dato in Figura 7. Si ricordi che assumiamo l'esistenza di una variabile globale `root` che punta al nodo radice. Si assume anche di disporre di due procedure `copy` e `successor` anche se la procedura `easySuccessor` dell'Esercizio 16 puó essere utilizzata al posto di quest'ultima.