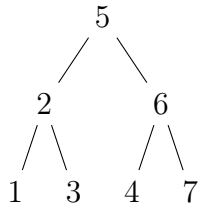
Appunti Algoritmi e Strutture Dati

Luca Seggiani

27 Marzo 2024

1 Alberi Binari di Ricerca

Un'albero binario di ricerca è un albero binario tale che per ogni nodo p, i nodi del sottoalbero sinistro di p hanno etichetta minore dell'etichetta di p. Di contro, i nodi del sottoalbero destro avranno etichettta maggiore dell'etichetta di p. Si stabilisce un ordine tra i nodi, più forte del semplice meccanismo antecedente-precedente, in base all'ordinamento tra le etichette. Ad esempio:



Notiamo che non possono esistere nodi doppioni, e che l'ordinamento risulta infine da sinistra verso destra, ergo una stampa simmetrica dell'albero binario di ricerca fornisce sostanzialmente una lista dei nodi in ordine crescente.
Possiamo definire sugli alberi di ricerca alcune operazioni, fra cui la ricerca di un nodo (ottimizzato!), l'inserimento ordinato di un nodo, e la cancellazione di un nodo.

Ricerca su alberi di ricerca

Si spera che, se l'hanno chiamati alberi di ricerca, almeno quella ci si possa fà per benino. Effettivamente la ricerca su alberi di ricerca si può eseguire sempre come ricerca binaria, essendo gli stessi sempre ordinati per definizione. Possiamo quindi implementare, in pseudocodice, la funzione ricorsiva:

```
Node* findNode(int n, Node* tree) {
  if(tree == NULL) return 0;
  if(n == tree->label) return tree;
  if(n < tree->label) return findNode(n, tree->left);
  return findNode(n, tree->right);
}
```

Questo ci permetterà di dividere per 2 la dimensione di istanza ad ogni passaggio, ottenendo una complessità di $O(\log n)$. Chiaramente, questo vale solo nel caso migliore: nel caso medio, il valore verrà fortemente influenzato dal bilanciamento (o mancanza di tale) del nostro albero. E' fondamentale avere alberi bilanciati per ottimizzare le operazioni di ricerca e inserzione. Nel caso l'albero sia completamente sbilanciato, si dice che è degenere, e le operazioni di ricerca costeranno su di esso $O(n)$.

Inserzione in alberi di ricerca

Inserire nodi in alberi di ricerca significa fare l'inserimento ordinato, per mantenere appunto l'ordine dell'albero:

```
void insertNode(int n, Node *& tree) {
  if(tree == NULL) {
    tree = new Node;
    tree->label = n;
    tree->left = tree->right = NULL;
  }
  if(n < tree->label) insertNode(n, tree->left);
  if(n > tree->label) insertNode(n, tree->right);
}
```

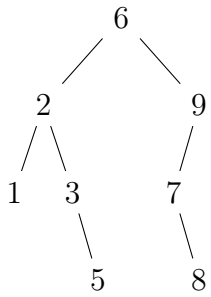
L'algoritmo aggiunge, se l'albero considerato è vuoto, l'elemento considerato. Sennò confronta l'elemento con i sottoalberi sinistri e destro dell'albero, e chiamandosi ricorsivamente sul ramo che rispetta l'ordinamento. La complessità è sempre $O(\log n)$, che può arrivare a $O(n)$ se l'albero è degenere.

Cancellazione di alberi di ricerca

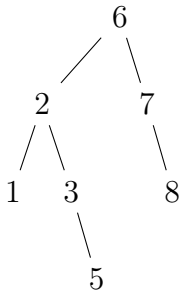
Poniamoci adesso il problema di cercare e successivamente cancellare un determinato nodo. Occorrerà innanzitutto trovare il nodo desiderato attraverso i meccanismi sopra definiti, e valutare poi le due possibili situazioni che potremmo incontrare:

- Se p (l'albero trovato) ha un sottoalbero vuoto, il padre di p viene connesso all'unico sottoalbero non vuoto di p.
- Se p ha entrambi i sottoalberi non vuoti si cerca il nodo con etichetta minore nel sottoalbero destro di p, si cancella e si mette la sua etichetta come etichetta di p.

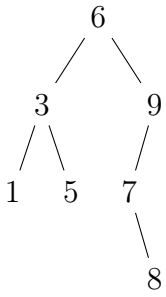
Visualizziamo i diversi casi:



rimuovere il 9 significa:



e rimuovere il 2 significa:



Ciò si può implementare con:

```
//helper (trova minimo)
void deleteMin(Node* &tree, int &m) {
  if(tree->left) deleteMin(tree->left, m);
  else {
    m = tree->label;
    Node* a = tree;
    tree = tree->right;
    delete a;
  }
}
```

```
11
12 //funzione
13 void deleteNode(int n, Node* &tree) {
14   if(tree) {
15     if(n < tree->label) {
16       deleteNode(n, tree->left);
17       return;
18     }
19     else if(n > tree->label) {
20       deleteNode(n, tree->right);
21       return;
22     }
23     else if(tree->left == NULL) {
24       Node* a = tree;
25       tree = tree->right;
26       delete a;
27       return;
28     }
29     else if(tree->right == NULL) {
30       Node* a = tree;
31       tree = tree->left;
32       delete a;
33       return;
34     }
35     else deleteMin(tree->right, tree->label);
36   }
37 }
```

L'helper deleteMin() trova il minimo, lo elimina e ne inserisce l'etichetta nell'integer m. Questo torna utile nel secondo caso, quando abbiamo intenzione di eliminare il nodo minimo e sostituirne l'etichetta al nodo eliminato. La funzione a questo punto si occupa soltanto di scorrere l'albero in maniera ricorsiva, applicando l'algoritmo giusto a seconda del caso incontrato. Nello specifico, finchè non trova il nodo desiderato prosegue di ricerca binaria. Quando raggiunge a una foglia che corrisponde, sceglie in base al numero di sottoalberi il percorso da prendere: nel caso esista solamente un sottoalbero, si elimina dopo aver sostituito il suo sottoalbero esistente a se stesso. Nel caso entrambi i sottoalberi esistano, chiama deleteMin() sul suo sottoalbero destro, passando la sua stessa etichetta per riferimento. Notiamo che il caso dove l'etichetta cercata non esiste è gestito dal controllo su tree in capo a entrambe le funzioni (albero vuoto $\rightarrow$ caso base).

Costo di riempimento di un'albero binario di ricerca

Vediamo quanto costa in funzione di n elementi da aggiungere, riempire un albero binario di ricerca. Potremmo bovinamente dire $O(n \log n)$, visto che svolgo effettivamente n operazioni di costo $O(\log n)$. Il problema sta nel fatto che il costo di $O(\log n)$ vale nel caso ottimale, non nel caso generale: fosse

considerato il caso peggiore, avremmo n volte qualcosa che ha complessita $O(n)$, che però verrebbe ogni volta chiamato su dimensioni di istanza $n-1$, da cui $O(n\frac{n-1}{2})$, ovvero $O(n^2)$. Dobbiamo allora ricordare l'albero di ricerca binaria è una struttura di dati estremamente efficiente, ma che la sua creazione ha il costo non indifferente di $O(n^2)$.