Algoritmi e Strutture Dati
24 gennaio 2022

Note

1. La leggibilità è un prerequisito: parti difficili da leggere potranno essere ignorate.
2. Quando si presenta un algoritmo è fondamentale spiegare l'idea e motivarne la correttezza.
3. L'efficienza e l'aderenza alla traccia sono criteri di valutazione delle soluzioni proposte.
4. Si consegnano tutti i fogli, con nome, cognome, matricola e l'indicazione *bella copia* o *brutta copia*.

Domande

Domanda A (8 punti) Si consideri la seguente funzione ricorsiva con argomento un array A di interi e due indici $1 \leq p \leq r \leq A.length$:

```
Minimum(A,p,r)
   if p=r
      return A[p]
   else
      q = (p+r)/2
      return min(Minimum(A,p,q), Minimum(A,q+1,r))
```

Dimostrare induttivamente che la funzione calcola il minimo del sottoarray $A[p..r]$. Inoltre, determinare la ricorrenza che esprime la complessità della funzione e mostrare che la soluzione è $\Theta(n)$, dove n indica la lunghezza del sottoarray. Motivare le risposte.

Soluzione: Per quanto riguarda la funzione calcolata, procediamo per induzione sulla lunghezza $n = r - p + 1$ del sottoarray $A[p..q]$. Se $n = 1$, il sottoarray di interesse contiene un solo elemento $A[p]$ che è correttamente l'elemento ritornato. Se invece $n > 1$, le due chiamate ricorsive $Minimum(A, p, q-1)$ e $Minimum(A, q, r)$, su sottoarray di dimensione $< n$ ritornano il minimi in $A[p..q]$ e $A[q+1, r]$. Viene ritornato il valore mimimimo tra i due, che srà correttamente il minimo valore in $A[p..r]$.

Per quanto riguarda la ricorrenza, nel caso ricorsivo, se n è la lunghezza del sottoarray di interesse, si effettuano due chiamate ricorsive su sottorray di dimensione $n/2$ e un'operazione di minimo (operazione di tempo costante) si ottiene quindi

$$T(n) = 2T(n/2) + c$$

Per risolvere la ricorrenza possiamo applicare il Master Theorem (siamo in una istanza dello schema generale con $a = b = 2$ e $f(n) = c$). Si osserva che $n^{\log_b a} = n^{\log_2 2} = n^1 = n$. Quindi $f(n) = c = O(n^{\log_b a - \epsilon}) = O(n^{1-\epsilon})$ qualunque sia $0 < \epsilon < 1$. Quindi siamo nel primo caso del Master Theprem e si conclude che $T(n) = \Theta(n^{\log_b a}) = \Theta(n)$.

Domanda B (6 punti) Realizzare una funzione `SearchUnique(T,k)` che dato un albero binario di ricerca `T` (che si assume non vuoto), verifica se la chiave `k` è presente in un unico nodo, e, in caso affermativo restituisce il nodo, altrimenti (ovvero se la chiave non è presente oppure è presente in più nodi) restituisce `nil`. Si possono usare le funzioni standard degli alberi binari di ricerca. Valutarne la complessità.

Soluzione: L'idea è quella di cercare la chiave k con l'usuale ricerca dei BST. Se la chiave è presente, detto x il nodo trovato, per verificare se sia unica, si controlla che sottoalbero destro e sinistro non contengano ulteriori occorrenze della chiave. Questo si può fare con due ulteriori ricerche. La usuale funzione dei BST che cerca una chiave k nel sottoalbero radicato in x si indica con `Search(x,k)`.

```
SearchUnique(T,k)
   x = Search(T.root,k)

   if (x == nil) or (Search(x.left, k) <> nil) or (Search(x.right, k) <> nil)
      return x
   else
      return nil
```

L'algoritmo è corretto. L'osservazione fondamentale è che dato un nodo x di un BST, se la chiave k compare nel sottoalbero destro e in quello sinistro allora è anche la chiave della radice (infatti, per le proprietà dei BST, si avrebbe $k \leq x.key \leq k$). Ne segue che tutti i nodi con chiave k in T saranno nel sottoalbero del nodo x, ritornato dalla prima chiamata della funzione `Search`. Questo immediatamente implica la correttezza dell'algoritmo.

La complessità è quella di tre ricerche ciascuna con costo $O(h)$, dove h è l'altezza dell'albero, e quindi $O(h)$.

In alternativa, si può osservare che, se la chiave k dovesse comparire nel sottoalbero sinistro, sarebbe il massimo di questo sottoalbero e, dualmente, se comparisse nel secondo sottoalbero, sarebbe il minimo. Questo porta alla soluzione che segue

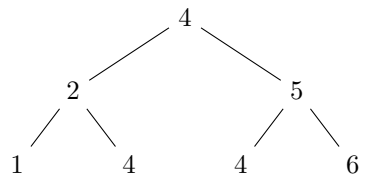
```
SearchUnique(T,k)
   x = Search(T.root,k)

   if (x <> nil)
      y = Max(x.left)
      z = Min(x.right)

      if ((y <> nil) and (y.key == k)) or (z <> nil) and (z.key == k))
         return nil
      else
         return x
   else
      return nil
```

La complessità asintotica non cambia, ma questa funzione è più efficiente della precedente dato che `Min` e `Max`, diversamente a `Search` non eseguono confronti sulle chiavi.

Per concludere, è da osservare che l'assunzione che le chiavi duplicate siano solo nel sottoalbero destro (o in quello sinistro) o che siano adiacenti non è legittima. Ad esempio, il seguente è un BST valido:



Esercizi

Esercizio 1 (9 punti) Si consideri una variante degli alberi binari di ricerca nella quale i nodi `x` hanno un campo addizionale `x.min`, che rappresenta il minimo delle chiavi nel sottoalbero radicato in `x`. Realizzare la procedura `Insert(T,z)` che inserisce un nodo `z` nell'albero e la rotazione a sinistra `Left(T,x)` (assumendo che `x` e `x.right` non siano `nil`). Valutarne la complessità.

Soluzione: La procedura di inserimento scende dalla radice fino ad una foglia dove il nodo viene inserito z. Dato che z sarà nel sottoalbero di ogni nodo attraversato durante la discesa, per ciascuno di questi nodi e soltanto per essi il campo *min* deve essere possibilmente aggiornato. Una piccola ottimizzazione: la verifica che `z.key < x.min` ha senso farla solo se `z.key < x.key` dato che `x.min <= x.key`.

```
Insert(T,z)
   y = nil
   x = T.root

   while (x <> nil)
      y = x

      if (z.key < x.key)
         if z.key < x.min     /* il sottoalbero radicato in z contiene z  */
            x.min = x.key     /* quindi si aggiorna x.min                 */

         x = x.left
      else
         x = x.right

   z.p = y
   if (y <> nil)
      if (z.key < y.key)
         y.left =  z
      else
         y.right = z

    z.min = z.key    /* il sottoalbero radicato in z contiene solo z */
```

La complessità rimane la stessa di *Insert* negli alberi binari di ricerca "standard", ovvero $O(h)$, dove h è l'altezza dell'albero.

Anche la rotazione a sinistra si ottiene inserendo, in coda al codice standard, l'aggiornamento del campo min. Si osservi che gli unici nodi per i quali si modificano i nodi contenuti nel corrispondente sottoalbero sono `x` e `y = x.right`. I nodi che prima erano nel sottolbero radicato in `x` poi sono in quello radicato in `y`, quindi i valori dei campi `min` e `max` per `y` dopo la rotazione sono quelli che prima erano in `x`. Per calcolare il nuovo valore del campo `min` per `x` si usa la conoscenza di tali valori per i figli destro e sinistro.

```
Left(T,z)
   y = x.right
   x.right = y.left
   x.right.p = x
   Transplant(T,x,y)
   y.left = x
   x.p = y

   y.min = x.min

   x.min = min(x.key, x.left.min, x.right.min)
```

La complessità resta costante $O(1)$.

Esercizio 2 (10 punti) Abbiamo n programmi da eseguire sul nostro computer. Ogni programma j, dove $j \in \{1, 2, \ldots, n\}$, ha lunghezza ℓ_j, che rappresenta la quantità di tempo richiesta per la sua esecuzione. Dato un ordine di esecuzione $\sigma = j_1, j_2, \ldots, j_n$ dei programmi (cioè, una permutazione di $\{1, 2, \ldots, n\}$), il tempo di completamento $C_{j_i}(\sigma)$ del j_i-esimo programma è dato quindi dalle somma delle lunghezze dei programmi $j_1, j_2, \ldots, j_i$. L'obiettivo è trovare un ordine di esecuzione σ che minimizza la somma dei tempi di completamento di tutti i programmi, cioè $\sum_{j=1}^{n} C_j(\sigma)$.

(a) Dare un semplice algoritmo greedy per questo problema, e valutarne la complessità.

(b) Dimostrare la proprietà di scelta greedy dell'algoritmo del punto (a), cioè che esiste un ordine di esecuzione ottimo $\sigma^\star$ che contiene la scelta greedy.

Soluzione:

(a) Ordina i programmi per lunghezza crescente. Complessità: $O(n \log n)$.

(b) La scelta greedy consiste nello scegliere, come prossimo programma da eseguire, quello di lunghezza minima. Sia $\sigma^\star$ una soluzione ottima. Se il programma di lunghezza minima è il primo in $\sigma^\star$, abbiamo finito. Consideriamo quindi il caso in cui il programma di lunghezza minima sia in posizione $k > 1$ in $\sigma^\star$. Costruiamo una nuova soluzione σ' scambiando, in $\sigma^\star$, il k-esimo programma con il primo. Possiamo osservare che:

- l'insieme dei primi k programmi $j_1, j_2, \ldots, j_k$ è lo stesso in $\sigma^\star$ e σ', quindi il k-esimo programma ha lo stesso tempo di completamento in $\sigma^\star$ e σ'; lo stesso vale per tutti i programmi successivi al k-esimo, visto che lo scambio non influisce su di loro;
- per quanto riguarda tutti gli altri programmi, cioè quelli fino alla posizione $k-1$, questi hanno un tempo di completamento inferiore o uguale in σ', perché lo scambio può solo avere ridotto la lunghezza del primo programma.

Quindi

$$\sum_{j=1}^{n} C_j(\sigma') \leq \sum_{j=1}^{n} C_j(\sigma^\star);$$

siccome $\sigma^\star$ è una soluzione ottima, allora deve valere che

$$\sum_{j=1}^{n} C_j(\sigma') = \sum_{j=1}^{n} C_j(\sigma^\star),$$

cioè anche σ' è una soluzione ottima.