Complementi di Algoritmi e Strutture Dati

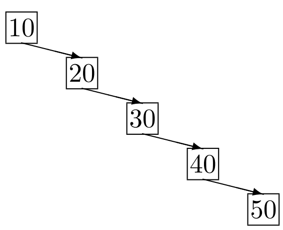
(III anno Laurea Triennale - a.a. 2017/18)

Soluzione Prova scritta 6 settembre 2018

Esercizio 1 – Alberi AVL (Punti 6)

1. Mostrare l'albero binario di ricerca (BST) che risulta dall'inserimento di cinque elementi con chiavi 10, 20, 30, 40, 50 in questo ordine (cioè chiavi crescenti).

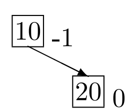
   Soluzione



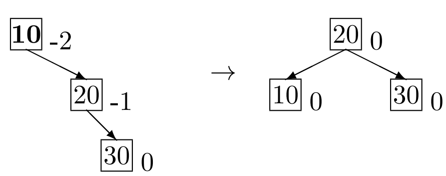
2. Inserire adesso la stessa sequenza di elementi in un albero AVL. Per ogni elemento inserito, mostrare e spiegare che cosa succede, indicando anche i fattori di bilanciamento dei nodi.

   Soluzione

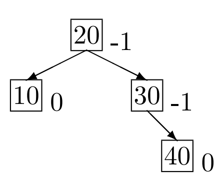
Inserisco 10. Inserisco 20 come in BST e poi aggiorno i fattori di bilanciamento:



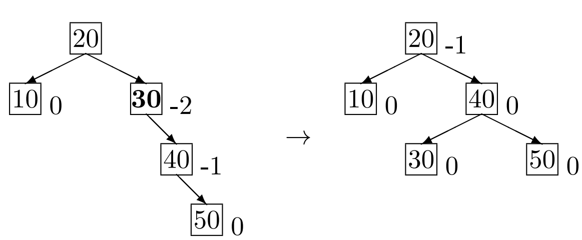
Inserisco 30. Risalendo verso la radice aggiorno i fattori di bilanciamento e vedo che il nodo radice, contenente 10, è sbilanciato a destra. Faccio rotazione semplice verso sinistra che coinvolge i nodi con 10,20,30:



Inserisco 40 e aggiorno i fattori di bilanciamento. Nessun nodo risulta sbilanciato e perciò non sono necessarie rotazioni:

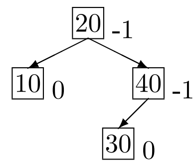


Inserendo 50, il nodo contenente 30 diventa sbilanciato a destra. Faccio rotazione semplice verso sinistra che coinvolge i nodi 30,40:

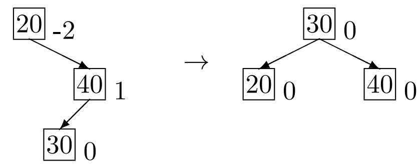


3. Dall'albero risultante cancellare uno dopo l'altro gli elementi di chiave 50 e di chiave 10.

L'elemento di chiave 50 è in una foglia. Elimino la foglia e, risalendo verso la radice, aggiorno i fattori di bilanciamento. L'albero resta bilanciato e non richiede rotazioni.



Anche l'elemento di chiave 10 è in una foglia. Elimino la foglia. Diversamente da prima, l'albero diventa sbilanciato in radice (nodo contenente 20). Per ribilanciare è necessaria una rotazione doppia:



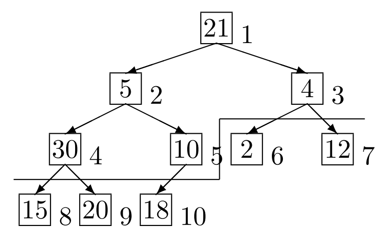
Esercizio 2 – Sorting (punti 6) Dato il seguente array:

21	5	4	30	10	2	12	15	20	18

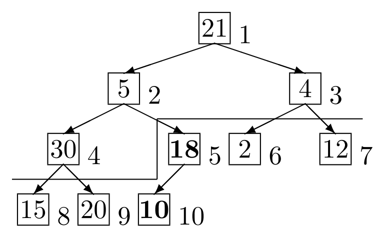
Eseguire la prima fase dell'algoritmo heapsort, cioè quella che trasforma l'array in uno heap a massimo. Si chiede di eseguirla mediante la procedura *heapify*.
Ad ogni passo disegnare tutto l'array come albero ed indicare quali sotto-parti sono già heap. Ricordare che deve essere uno heap *a massimo*.

Soluzione

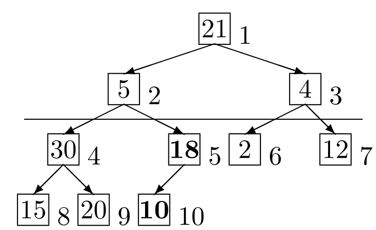
Consideriamo che il primo elemento dell'array abbia indice 1. Disegnando l'array come heap, le foglie si trovano negli indici da 6 (compreso) in poi. Ciascuna foglia è già uno heap.
La procedura prevede di applicare la funzione *moveDown* ad ogni nodo non foglia, a partire da quello di indice più grande (qui indice 5) verso quello di indice 1 (radice).



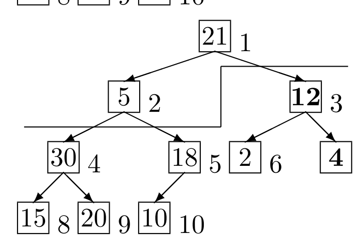
moveDown sul nodo di indice 5 (contenente chiave 10): siccome $10 < 18$ e lo heap è a massimo, scambio 10 e 18.



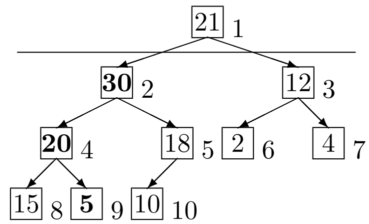
moveDown sul nodo di indice 4 (contenente chiave 30): siccome $30 > 20$ (dove 20 è la chiave maggiore tra i due figli), non occorrono scambi.



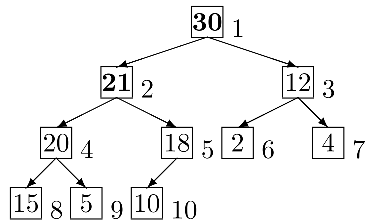
moveDown sul nodo di indice 3 (chiave 4): siccome $4 < 12$ (dove 12 è la chiave maggiore tra i figli), scambio 4 e 12.



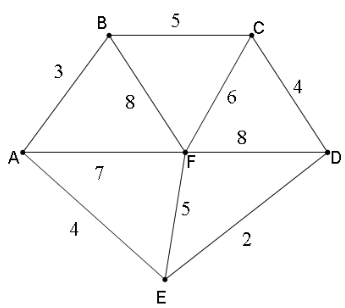
moveDown sul nodo di indice 2 (chiave 5): siccome $5 < 30$ scambio 5 e 30; poi siccome $5 < 20$ scambio ancora 5 e 20.



moveDown sul nodo di indice 1 (chiave 21): siccome $21 < 30$ scambio 21 e 30; non occorrono altri scambi poiché $21 > 20$. Tutto l'array è heap.



Esercizio 3 - Grafi (Punti 7) Si esegua l'algoritmo di Kruskal sul seguente grafo pesato:

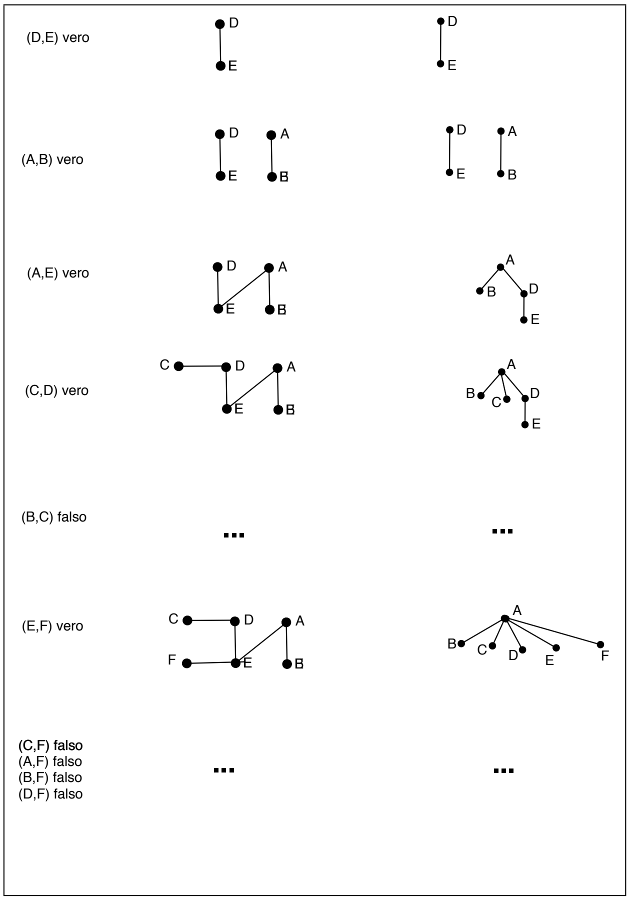


In particolare, per ogni iterazione si diano:

- l'arco estratto e il risultato (vero o falso) dell'operazione `union_by_need`
- la foresta ricoprente corrente
- la foresta union-find corrente (per brevità date solo gli alberi non singleton).

Si assuma che la `find` effettui la compressione dei cammini, e nell'unione di due alberi union-find si utilizzi la union-by-rank. Nel caso di più scelte possibili si consideri l'ordine alfabetico.

Soluzione



Esercizio 4 - Tecniche algoritmiche (Punti 7) Chiamiamo *picco* (*peak*) di una sequenza, rappresentata con un array $A[\texttt{inf}..\texttt{sup}]$ ($\texttt{inf} \leq \texttt{sup}$), un (qualunque) indice $\texttt{i}$ tale che l'elemento a sinistra, se esiste ($\texttt{i} \neq \texttt{inf}$), è minore o uguale ($A[\texttt{i}-1] \leq A[\texttt{i}]$), e analogamente l'elemento a destra, se esiste ($\texttt{i} \neq \texttt{sup}$), è minore o uguale ($A[\texttt{i}+1] \leq A[\texttt{i}]$).

1. Esiste sempre almeno un picco?
2. Si descriva (anche solo a parole) un algoritmo brute-force che risolve il problema (ossia restituisce l'indice di un qualunque picco) e se ne indichi la complessità.
3. Si descriva in pseudocodice un algoritmo divide-et-impera che risolve il problema in modo più efficiente, e se ne calcoli la complessità.
4. Se ne giustifichi la correttezza.

Soluzione

1. Ovviamente sì, dato che ogni sequenza non vuota ha un massimo.
2. Un algoritmo brute-force scorre tutto l'array controllando per ogni indice la condizione. La complessità è $O(n)$. Alternativamente basta cercare il massimo, sempre con complessità lineare.
3. Un algoritmo divide-et-impera procede in modo analogo alla ricerca binaria: se l'elemento centrale è un picco ne restituisce l'indice, altrimenti se la condizione è violata a sinistra cerca ricorsivamente nella sequenza a sinistra, e analogamente se la condizione è violata a destra. Più precisamente in pseudocodice:

```
peak(A)// A[0..n-1], n >= 1
        return peak(A,0,n-1)

peak(A,inf,sup)// 0<=inf<=sup<=n-1,
  if (inf == sup) return inf;
  mid =(inf + sup)/2;
  if (mid!=inf && A[mid-1] > A[mid]) return peak(A,inf,mid-1);
  if (mid!=sup && A[mid+1] > A[mid]) return peak(A,mid+1,sup);
  return mid;
```

Poiché segue lo stesso schema della ricerca binaria, l'algoritmo ha la stessa relazione di ricorrenza e complessità logaritmica.

4. Proviamo che `peak(A,inf,sup)` restituisce l'indice di un picco della sequenza $A[\texttt{inf}..\texttt{sup}]$, assumendo $0 \leq \texttt{inf} \leq \texttt{sup} \leq n-1$. Ragioniamo per induzione aritmetica forte sulla lunghezza di $A[\texttt{inf}..\texttt{sup}]$. Se la sequenza consiste di un solo elemento l'algoritmo restituisce correttamente il suo indice. Altrimenti, l'algoritmo considera l'elemento centrale. Se nella sequenza considerata l'elemento a sinistra di `mid` esiste ed è maggiore, l'algoritmo restituisce per ipotesi induttiva il picco relativo alla sequenza $A[\texttt{inf}..\texttt{mid}-1]$. Possiamo applicare l'ipotesi induttiva in quanto $\texttt{mid} \neq \texttt{inf}$ implica la precondizione per la chiamata $\texttt{inf} \leq \texttt{mid}-1$, e la chiamata è effettuata su una sequenza di lunghezza strettamente minore. Se il picco trovato per $A[\texttt{inf}..\texttt{mid}-1]$ è un indice diverso da $\texttt{mid}-1$, è un picco anche per la sequenza $A[\texttt{inf}..\texttt{sup}]$. Se è proprio $\texttt{mid}-1$, sappiamo che è un picco anche per la sequenza $A[\texttt{inf}..\texttt{sup}]$ poiché $A[\texttt{mid}-1] > A[\texttt{mid}]$. Analogamente se l'elemento a destra esiste ed è maggiore. Se non vale nessuna di queste due condizioni `mid` è un picco e l'algoritmo restituisce correttamente il suo indice.

Esercizio 5 - Notazioni asintotiche (Punti 6) Usando la definizione dimostrare che:

1. se $f(n) = \Theta(g(n))$ allora anche $g(n) = \Theta(f(n))$
2. se $f(n) = O(g(n))$ e $g(n) = O(h(n))$, allora $f(n) = O(h(n))$.

Soluzione

1. $f(n) = \Theta(g(n))$ significa che $\exists c_1, c_2 > 0, n_0 \geq 0$ tali che

   $c_1 g(n) \leq f(n) \leq c_2 g(n)$ per ogni $n \geq n_0$

   Ma allora si ha anche

   $\frac{1}{c_2} f(n) \leq g(n) \leq \frac{1}{c_1} f(n)$ per ogni $n \geq n_0$

   quindi $g(n) = \Theta(f(n))$.

2. $f(n) = O(g(n))$ significa che $\exists c, n_0 \geq 0$ tali che

   $f(n) \leq c g(n)$ per ogni $n \geq n_0$

   Inoltre, $g(n) = O(h(n))$ significa che $\exists c', n'_0 \geq 0$ tali che

   $g(n) \leq c' g(n)$ per ogni $n \geq n'_0$

   Ma allora si ha

   $f(n) \leq c g(n) \leq (c \cdot c') h(n)$ per ogni $n \geq \max(n_0, n'_0)$

   quindi $f(n) = O(h(n))$.