Complementi di Algoritmi e Strutture Dati

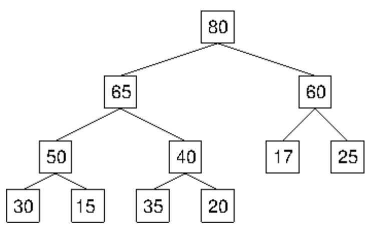
(III anno Laurea Triennale - a.a. 2016/17)

Soluzione della prova scritta 19 luglio 2017

Esercizio 1 – Ordinamenti Si consideri il seguente array, sul quale è stata già eseguita la prima fase dell'algoritmo heapsort, cioè l'array è già stato trasformato in uno heap a massimo:

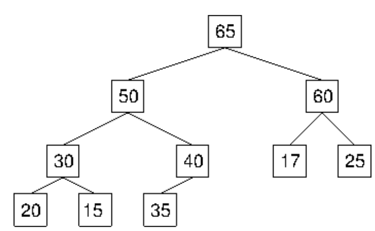
80	65	60	50	40	17	25	30	15	35	20

1. Disegnare la rappresentazione ad albero dello heap.

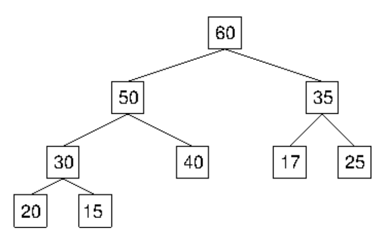


2. Eseguire la seconda fase dell'algoritmo heapsort, cioè quella che produce l'array ordinato. Ad ogni passo mostrare il nuovo stato di tutto l'array e la rappresentazione ad albero della parte di array che è heap.

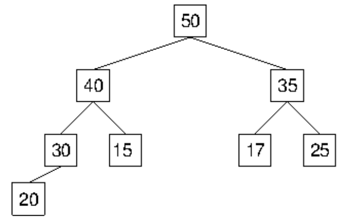
   La doppia riga separa la parte di array che è ancora heap (a sinistra) dalla parte di array che è già ordinato (a destra).



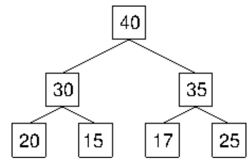
65	50	60	30	40	17	25	20	15	35	80



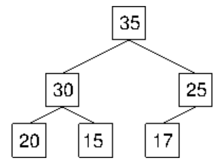
60	50	35	30	40	17	25	20	15	65	80



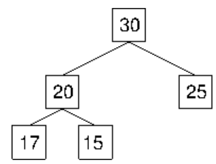
50	40	35	30	15	17	25	20	60	65	80



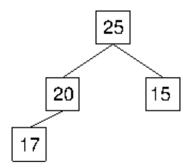
40	30	35	20	15	17	25	50	60	65	80



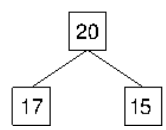
35	30	25	20	15	17	40	50	60	65	80



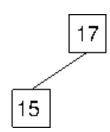
30	20	25	17	15	35	40	50	60	65	80



25	20	15	17	30	35	40	50	60	65	80



20	17	15	25	30	35	40	50	60	65	80

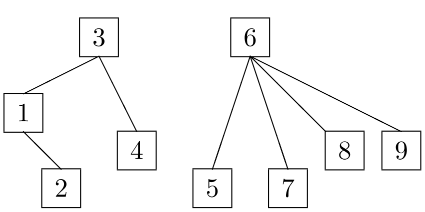


17	15	20	25	30	35	40	50	60	65	80

15	17	20	25	30	35	40	50	60	65	80

L'array è ordinato.

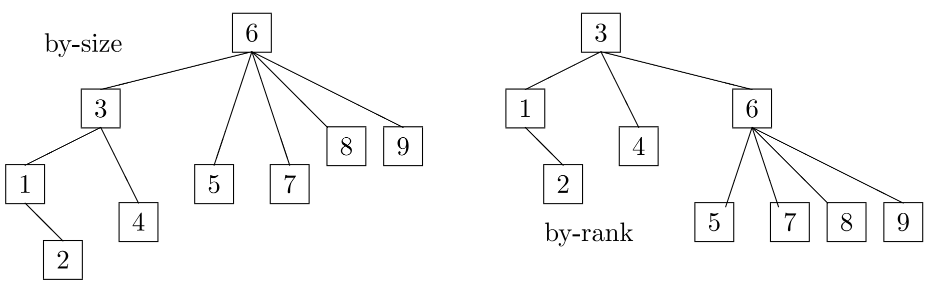
Esercizio 2 – Union-find Considerare la foresta union-find sottostante e un'implementazione di tipo *quick-union*.



1. Indicare, per ogni radice, il "size" e il "rank" del corrispondente albero.
Radice 3: size-4, rank=2. Radice 6: size=5, rank=1.
Nota: rank = altezza = lunghezza del massimo cammino radice-foglia in termini di numero di archi, qualcuno l'ha intesa in termini di numero di nodi, per cui viene uno in più (3 e 2) – va bene lo stesso, basta poi essere coerenti...

2. Eseguire l'operazione *union*(4,7) nelle due versioni:
a) supponendo *union-by-size*
b) supponendo *union-by-rank*
SENZA compressione dei cammini. Spiegare che cosa viene fatto e perché, disegnare il nuovo albero e indicare il "size" e il "rank" della sua radice.

   In entrambi i casi eseguo find(4)=3 e find(7)=6. Poi con union-by-size il nodo 3 (radice dell'albero di size minore) diventa figlio di 6, e il nuovo albero ha size=9 (somma dei due size). Invece con union-by-rank il nodo 6 (radice dell'albero di rank minore) diventa figlio di 3, e il nuovo albero ha rank=2 (essendo i due rank diversi, prendo il massimo dei due).



   Nota: se facciamo union-by-size, il rank dell'albero non conta, e viceversa. Nel caso union-by-rank, nessuno ha detto come mai il rank del risultato viene quello – non è stato contato.

3. Domanda identica alla precedente ma CON compressione dei cammini.

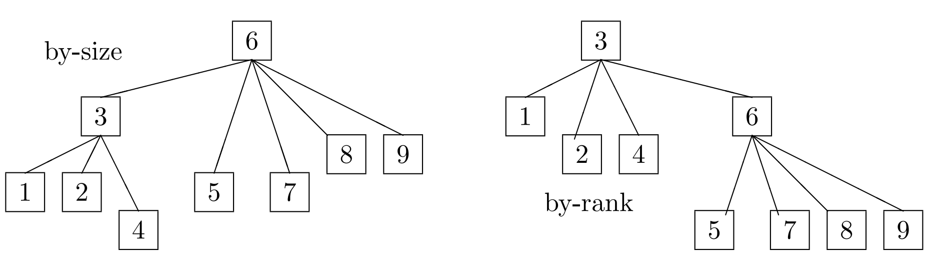
   Con la compressione dei cammini eseguendo find(4) devo attaccare alla radice 3 tutti i nodi attraversati, ma è solo 4 che è già figlio di 3. Allo stesso modo eseguendo find(7) non cambia nulla perchè l'unico nodo attraversato, 7, è già figlio di 6.

4. Domanda identica alla domanda 2 ma per l'operazione *union*(2,7).
Nota bene: eseguire l'operazione sulla foresta iniziale (NON sul risultato dell'operazione di cui alla domanda 2.

   In entrambi i casi eseguo find(2)=3 e find(7)=6. Poi l'esecuzione dell'operazione prosegue in entrambi i casi come nella risposta alla domanda 2.

5. Domanda identica alla precedente ma CON compressione dei cammini.

   Eseguendo find(2) devo attaccare alla radice 3 tutti i nodi attraversati, che sono 2 e 1: attacco quindi 2 come figlio di 3, mentre 1 lo è già. Eseguendo find(7) invece non cambia nulla.



   Nota: nel caso di union-by-rank con compressione dei cammini, se per effetto della compressione l'altezza dell'albero diminuisce (come avveniva qui), tuttavia il rank non viene aggiornato e rimane un *maggiorante* dell'altezza!

Esercizio 3 - Analisi di algoritmi Si consideri il seguente algoritmo ricorsivo che prende in input un numero naturale.

```
fun(n)
if (n>1)
  a = 0
  for (i = 1; i < n; i++)
    for (j =i+1; j <=n; j++) a = a + 2 * (i + j)
  for (i = 1; i <= 16; i++) a = a + fun(n/4);
  return a
else return n-1
```

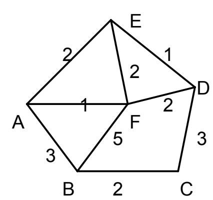
Scrivere e risolvere la relazione di ricorrenza che descrive il costo computazionale di `fun` in funzione di `n`.
La relazione di ricorrenza è la seguente:

$T(n) = \Theta(1)$ se $n \leq 1$
$T(n) = 16T(\frac{n}{4}) + \Theta(n^2)$, per $n > 1$.

Si noti infatti che l'assegnazione all'interno dei due cicli for innestati viene eseguita la prima volta $n-1$ volte, la seconda $n-2$ volte, fino a 1, quindi in tutto $\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2)$. In base al teorema master questa relazione di ricorrenza ha soluzione $T(n) = \Theta(n^2 \log(n))$, infatti si ha:

$a = 16, b^s = 16, a = b^s$
$T(n) = \Theta(n^2 \log_4 n)$

Esercizio 4 - Grafi Si consideri il seguente grafo pesato.



Si costruisca il minimo albero ricoprente utilizzando:

1. l'algoritmo di Kruskal (per ogni iterazione, si diano l'arco esaminato e la foresta corrente)
2. l'algoritmo di Prim a partire dal nodo A (per ogni iterazione, si diano le distanza provvisorie `dist` di tutti i nodi e l'albero corrente, evidenziandone la parte definitiva)

Nel caso di più scelte possibili si usi come convenzione l'ordine alfabetico.

1. Per semplicità di scrittura rappresentiamo la foresta corrente come insieme di archi.

arco esaminato	**foresta corrente**
(A,F)	(A,F)
(D,E)	$(A,F)(D,E)$
(A,E)	$(A,E)(A,F)(D,E)$
(B,C)	$(A,E)(A,F)(B,C)(D,E)$
(D,F)	$(A,E)(A,F)(B,C)(D,E)$
(E,F)	$(A,E)(A,F)(B,C)(D,E)$
(A,B)	$(A,B)(A,E)(A,F)(B,C)(D,E)$
(C,D)	$(A,B)(A,E)(A,F)(B,C)(D,E)$
(B,F)	$(A,B)(A,E)(A,F)(B,C)(D,E)$

2. Diamo solo le distanze modificate. Anche in questo caso rappresentiamo l'albero corrente come insieme di archi (quelli in neretto sono definitivi).

getMin	`d(A)`	`d(B)`	`d(C)`	`d(D)`	`d(E)`	`d(F)`	
	0	∞	∞	∞	∞	∞	
A		3			2	1	$(A,B)(A,E)(A,F)$
F				2			$(A,B)(A,E)$**(A,F)**(F,D)
D			3		1		(A,B)**(A,F)(F,D)**$(D,C)(D,E)$
E							(A,B)**(A,F)(F,D)**(D,C)**(D,E)**
B			2				**(A,B)(A,F)(F,D)(D,E)**(B,C)
C							**(A,B)(A,F)(F,D)(D,E)(B,C)**

Esercizio 5 - Analisi di algoritmi Si consideri il seguente *coffee can problem* (*problema del barattolo di caffè*), dovuto a David Gries. Sia C un barattolo che contiene chicchi bianchi o neri, e indichiamo con $N(C)$ il numero di chicchi contenuti in C. Supponiamo inoltre di avere un numero illimitato di chicchi neri a disposizione fuori dal barattolo, che possono essere inseriti dentro.

```
//Pre: N(C) ≥ 2
while (N(C)>1)
  estrai due chicchi da C
  se hanno lo stesso colore eliminali e inserisci un chicco nero in C
  altrimenti ri-inserisci il chicco bianco in C ed elimina il nero
```

1. Si provi che l'algoritmo termina.
   L'algoritmo termina perché a ogni passo il numero di chicchi nel barattolo diminuisce di uno, e questo numero è limitato inferiormente da uno.

2. Si provi, utilizzando un'opportuna invariante, che il colore dell'ultimo chicco rimasto è bianco se e solo se il numero di chicchi bianchi presenti inizialmente è dispari.
   Se il numero di chicchi bianchi presenti inizialmente è dispari si mantiene dispari a ogni iterazione (infatti o non ne tolgo o ne tolgo due), quindi alla fine l'unico chicco rimasto è necessariamente bianco. Analogamente, se il numero di chicchi bianchi presenti inizialmente è pari si mantiene pari a ogni iterazione, quindi alla fine l'unico chicco rimasto è necessariamente nero.

Versione un po' più formale: scriviamo *odd-white*(C) per indicare che il numero di chicchi bianchi in C è dispari. Assumendo questa precondizione si ha:

```
//Pre: N(C) ≥ 2 ∧ odd-white(C)
while (N(C)>1)
//Inv: odd-white(C) ∧ N(C) ≥ 1
  estrai due chicchi da C
  se hanno lo stesso colore eliminali e inserisci un chicco nero in C
  altrimenti ri-inserisci il chicco bianco in C ed elimina il nero
//Post: odd-white(C) ∧ N(C) = 1
```

L'invariante si preserva a ogni iterazione e insieme alla condizione di uscita ($N(C) \leq 1$) implica la postcondizione. La funzione di terminazione è $N(C)$, decresce strettamente a ogni iterazione ed è limitata inferiormente da 1 se vale *Inv*.
Si ha una prova del tutto analoga considerando come precondizione $\neg$*odd-white*(C).